Computing with spikes, architecture, properties and implementation of emerging paradigms

Horacio Rostro-Gonzalez

PhD THESIS

prepared at
INRIA Sophia Antipolis - Méditerranée

and presented at the
University of Nice-Sophia Antipolis
Graduate School of Information and Communication Sciences

A dissertation submitted in partial fulfillment
of the requirements for the degree of

DOCTOR OF SCIENCE
Specialized in Control, Signal and Image Processing
In the field of Computational Neuroscience

Computing with spikes, architecture, properties and implementation of emerging paradigms

Horacio ROSTRO-GONZÁLEZ

Advisors	Bruno CESSAC	INRIA Sophia Antipolis, France
	Thierry VIÉVILLE	INRIA Sophia Antipolis, France
President	Rachid DERICHE	INRIA Sophia Antipolis, France
Reviewers	Hélène PAUGAM-MOISY	INRIA Saclay - Île-de-France, France
	Sylvie RENAUD	CNRS - ENSEIRB, UMR5818, France
Examiners	Bernard GIRAU	LORIA/INRIA Nancy Grand Est, France
	César TORRES-HUITZIL	CINVESTAV LTI - Tamaulipas, Mexico

UNIVERSITÉ NICE-SOPHIA ANTIPOLIS - UFR Sciences

École Doctorale STIC
(Sciences et Technologies de l'Information et de la Communication)

THÈSE

pour obtenir le titre de
DOCTEUR EN SCIENCES
de l'UNIVERSITÉ de Nice-Sophia Antipolis
Discipline: Automatique, Traitement du Signal et des Images
Application: Neurosciences Computationnelles

présentée et soutenue par

Horacio ROSTRO-GONZÁLEZ

Computing with spikes, architecture, properties and implementation of emerging paradigms

Thèse dirigée par Bruno CESSAC et Thierry VIÉVILLE

24 janvier 2011

Composition du jury:

Président	Rachid DERICHE	INRIA Sophia Antipolis, France
Rapporteurs	Hélène PAUGAM-MOISY	INRIA Saclay - Île-de-France, France
	Sylvie RENAUD	CNRS - ENSEIRB, UMR5818, France
Examinateurs	Bernard GIRAU	LORIA/INRIA Nancy Grand Est, France
	César TORRES-HUITZIL	CINVESTAV LTI - Tamaulipas, Mexico

Abstract

In this thesis we study at a concrete practical level how computation with action potentials (spikes) can be performed. We address the problem of programming a dynamical system modeled as a neural network and considering both, hardware and software implementations. For this, we use a discrete-time spiking neuron model, which has been introduced in Soula et al. (2006), and called BMS in the sequel, whose dynamics is rather rich (see section 1.2.4). On one hand, we propose an efficient method to properly estimate the parameters (delayed synaptic weights) of a neural network from the observation of its spiking dynamics. The idea is to avoid the underlying NP-complete problem (when both weights and inter-neural transmission delays are considered in the parameters estimation). So far, our method defines a Linear Programming (LP) system to perform the parameters estimation. Another aspect considered in this part of the work is the fact that we include a reservoir computing mechanism (hidden network), which permits us, as we show, to increase the computational power and to add robustness in the system. Furthermore these ideas are applied to implement input-output transformations, with a method learning the implicit parameters of the corresponding transfer function.

On the other hand we have worked on the development of numerical implementations permitting us to validate our algorithms. We also made contributions to code methods for spike trains statistics analysis and simulations of spiking neural networks. Thus, we co-develop a C++ library, called EnaS[1], which is distributed under the CeCILL-C free license. This library is also compatible with other simulators and could be used as a plugin.

Finally we consider the emergent field of bio-inspired hardware implementations, where FPGA (Field Programmable Gate Array) and GPU (Graphic Processing Unit) technologies are studied. In this sense, we evaluate the hardware implementations of the proposed neuron models (gIF-type neuron models) under periodic and chaotic activity regimes. The FPGA-based implementation has been achieved using a precision analysis and its performance compared with that based on GPU.

[1] http://enas.gforge.inria.fr

Résumé

Dans cette thèse, nous étudions à un niveau pratique comment nous pouvons réaliser des processus computationnels avec des potentiels d'action (*spikes*). Nous étudions le problème de la programmation d'un système dynamique modélisé comme un réseau de neurones, et nous considérons des implémentations en *software* et en *hardware*. Tout d'abord, nous révisons le modèle de réseau de neurones à temps discret introduit par Soula et al. (2006) et nommé ici BMS. L'intérêt d'utiliser ce modèle est dû à son habilité à reproduire des dynamiques assez riches (voir la section 1.2.4) et aussi permettre d'établir un lien direct entre le potentiel de la membrane et les impulsions de la neurone (*spiking activity*). En se basant sur une généralisation de ce modèle, nous proposons une méthode afin d'estimer d'une manière efficace les paramètres (les poids synaptiques à différents délais) d'un réseau de neurones à partir de l'observation de sa dynamique (train d'impulsions). L'idée est d'éviter le problème NP-complet qui se pose dès que nous considérons les poids synaptiques et les délais de transmission. Notre méthode permet de définir un système de programmation linéale à partir du modèle BMS et d'effectuer l'estimation des paramètres de manière polynomiale. Ensuite nous introduisons un mécanisme de *reservoir computing* (réseau de neurones cachés) afin de faire une estimation plus robuste. Finalement nous appliquons cette idée à l'implémentation de transformations entrée-sortie, où la méthode est capable d'apprendre les paramètres implicites correspondants à la fonction de transfert.

Dans un second temps, nous travaillons au développement d'implémentations numériques permettant de valider nos algorithmes. De plus nous faisons des contributions au niveau de la programmation de méthodes pour l'analyse de trains d'impulsions et la simulation de réseaux de neurones à impulsion. Nous co-développons une librairie numérique en C++, nommée EnaS et distribuée sous une licence gratuite CeCILL-C. Cette librairie est également compatible avec d'autres simulateurs et peut être utilisée comme un plugin.

La dernière partie de la thèse se focalisee sur les implémentations en *hardware* de modèles bio-inspires. Nous faisons le choix de regarder

des technologies à bas coût basées sur les FPGA (réseau de portes programmables *in situ*) et les GPU (processeur graphique). Nous évaluons la réponse des implémentations en *hardware* des modèles de neurones du type intègre-et-tire quand ils sont soumis aux différents régimes d'activité neuronale. L'implémentation sur le FPGA à été accomplie en faisant une analyse sur la précision et sa performance a été comparée avec celle du GPU.

Acknowledgments

This PhD thesis has been developed in the NeuroMathComp[2] INRIA project-team, (formerly Odyssée), in link with the CORTEX[3] INRIA project-team.

First, I would like to express my deeply grateful and sincere acknowledgments to my advisors, Bruno Cessac and Thierry Viéville for all the guidance that they provide me during my PhD thesis.

Next, I would also like to express my deep thanks to Hélène Paugam-Moisy and Sylvie Renaud for having accepted to review my manuscript despite the important amount of work this required. I am also very grateful to Rachid Deriche, Bernard Girau and César Torres-Huitzil for having kindly accepted to examine this thesis.

Finally, I would like to express my gratitude and love to all my family and friends who supported me during the last few years.

This research work has been partially supported by the grant **185246** of the CONACYT[4] and the SEP[5] of Mexico, the ANR MAPS[6] project and the INRIA MACCAC ARC[7].

[2] http://www-sop.inria.fr/neuromathcomp/
[3] http://cortex.loria.fr/
[4] http://www.conacyt.gob.mx/
[5] http://www.sep.gob.mx/
[6] http://maps.loria.fr/
[7] http://www-sop.inria.fr/odyssee/contracts/MACACC/macacc.html

Contents

Abstract **i**

Résumé **iii**

Acknowledgments **v**

Contents **x**

List of Figures **xvi**

List of Tables **1**

I gIF-type Spiking Neuron Models **13**

1 gIF-type spiking neuron models **15**

1.1 Neuron Anatomy 16

1.1.1 The spiking neuron 17

1.2 Discretized integrate and fire neuron models. 17

1.2.1 From a LIF model to a discrete-time spiking neuron model. 18

1.2.2 Time constrained continuous formulation. 20

1.2.3 Network dynamics discrete approximation. 21

1.2.4 Properties of the discrete-time spiking neuron model . 23

1.2.5 Asymptotic dynamics in the discrete-time spiking neuron model 26

1.2.6 The analog-spiking neuron model 27

1.2.7 Biological plausibility of the gIF-type models 28

II Learning spiking neural networks parameters 29

2 Reverse-engineering in spiking neural networks parameters: exact deterministic estimation 31

2.1 Methods: Weights and delayed weights estimation 32
2.1.1 Retrieving weights and delayed weights from the observation of spikes and membrane potential 33
2.1.2 Retrieving weights and delayed weights from the observation of spikes 35
2.1.3 Retrieving signed and delayed weights from the observation of spikes 37
2.1.4 Retrieving delayed weights and external currents from the observation of spikes 38
2.1.5 Considering non-constant leak when retrieving parametrized delayed weights 38
2.1.6 Retrieving parametrized delayed weights from the observation of spikes 39
2.1.7 About retrieving delays from the observation of spikes 39
2.2 Methods: Exact spike train simulation 40
2.2.1 Introducing hidden units to reproduce any finite raster 40
2.2.2 Sparse trivial reservoir 41
2.2.3 The linear structure of a network raster 43
2.2.4 On optimal hidden neuron layer design 44
2.2.5 A maximal entropy heuristic 45
2.3 Application: Input/Output transfer identification 46
2.4 Numerical Results 49
2.4.1 Retrieving weights from the observation of spikes and membrane potential 49
2.4.2 Retrieving weights from the observation of spikes . . . 52
2.4.3 Retrieving delayed weights from the observation of spikes 53
2.4.4 Retrieving delayed weights from the observation of spikes, using hidden units 58
2.4.5 Input/Output estimation 63

III Programming resetting non-linear networks 65

3 Programming resetting non-linear networks 67

3.1 Problem Position 67
3.2 Mollification in the spiking metrics 70
3.2.1 Threshold mollification 73
3.2.2 Mollified coincidence metric 74

3.2.3 Indexing the alignment distance 75
3.3 Learning paradigms. 76
3.3.1 The learning scenario. 77
3.3.2 Supervised learning scheme. 77
3.3.3 Robust learning scheme. 78
3.3.4 Reinforcement learning scheme. 81
3.3.5 Other learning paradigms. 82
3.4 Recurrent weights adjustment 83
3.5 Application to resetting non-linear (RNL) networks. 89
3.5.1 Computational properties. 90
3.6 Preliminary conclusion 93

IV Hardware Implementations of gIF-type Spiking Neural Networks 95

4 Hardware implementations of gIF-type neural networks 97
4.1 Hardware Description 98
4.1.1 FPGA (Field-Programmable Gate Array) 98
4.1.2 GPU (Graphics Processing Unit) 99
4.2 Programming Languages 100
4.2.1 Handel-C 100
4.2.2 VHDL (VHSIC hardware description language; VHSIC: very-high-speed integrated circuit) 101
4.2.3 CUDA (Compute Unified Device Architecture) 102
4.3 Fixed-point arithmetic for data representation 104
4.3.1 Resolution for the fixed-point data representation . . . 105
4.3.2 Range for the fixed-point data representation 105
4.3.3 A precision analysis of the gIF-type neuron models . . 105
4.4 FPGA-based architectures of gIF-type neuron models 110
4.4.1 A FPGA-based architecture of gIF-type neural networks using Handel-C 111
4.4.2 A FPGA-based architecture of gIF-type neural networks using VHDL 113
4.5 GPU-based implementation of gIF-type neuron models 119
4.6 Numerical Results 122
4.6.1 Synthesis and Performance 132

V Conclusion 135

5 Conclusion 137

A Publications of the Author Arising from this Work **141**

Publications of the Author Arising from this Work **141**

Bibliography **143**

List of Figures

1.1 Anatomy of a neuron. (Illustration from Mariana Ruiz Villarreal) 16

1.2 Action potentials arriving at the synapses of the upper right neuron stimulate currents in its dendrites; these currents depolarize the membrane at its axon, provoking an action potential that propagates down the axon to its synaptic knobs, releasing neurotransmitter and stimulating the post-synaptic neuron (lower left). Illustration from [8] 18

1.3 Schematic diagram of the integrate-and-fire model. The basic circuit is the module inside the dashed circle on the right-hand side. A current $I(t)$ charges the RC circuit. The voltage $u(t)$ across the capacitance (points) is compared to a threshold ϑ. If $u(t) = \vartheta$ at time $t_i^{(f)}$ and output pulse $\delta(t - t_i^{(t)})$ is generated. Left part: a presynaptic spike $\delta(t - t_j^{(t)})$ is low-pass filtered at the synapse and generates an input current pulse $\alpha(t - t_j^{(t)})$ (Illustration from Gerstner and Kistler (2002a)). 19

1.4 A profile $\rho(t) = H(t)\frac{t}{\tau}e^{-\frac{t}{\tau}}$, which represents the synaptic weights mapped at different delays. 21

1.5 Two examples of the partition space for non-perturbed balls in a system with two neurons. The space is partitioned from the firing state of the neurons and is labeled as $\boldsymbol{\omega} = \binom{\omega_1}{\omega_2}$ 24

1.6 The effects on the dynamics when the trajectories of $\mathbf{V}$ has been perturbed. The non-critical case. 25

1.7 The effects on the dynamics when a perturbed ball intersects the singularity set. The critical case. 26

1.8 Average value of the distance $d(\mathcal{A}, \mathcal{S})$ versus γ, C, for $N = 50$. Illustration from Cessac (2008) 27

1.9 A sigmoid function 28

2.1 Schematic representation of a raster plot with N neurons observed during a time T after a period D (in red), which represents an initial state. 32

2.2 Schematic representation of a raster of N output neuron observed during a time interval T after an initial conditions interval D, with an add-on of S hidden neurons, in order increase the number of degree of freedom of the estimation problem. See text for further details. 41

2.3 Schematic representation of a sparse trivial set of hidden neurons, allowing to generate any raster of length T. 42

2.4 A spike train representing input-output dynamics. Red lines define an initial state. Purple lines define a spiking activity in input neurons. Black lines define the output spiking activity, which is defined by an OR function applied in the input activity. 49

2.5 A "chaotic" dynamics with 30 neurons fully connected within network, initial conditions $D = 3$ and observation time $T = 100$, using both excitatory (70%) and inhibitory (30%) connections, with $\sigma = 5$ (weight standard-deviation). After estimation, we have checked that master and servant generate **exactly the same raster plot, thus only show the servant raster, here and in the next figures.** 51

2.6 A "periodic" (58.2 periods of period 5) dynamics with 20 neurons fully connected within network and observation time $T = 300$, using both excitatory (70%) and inhibitory (30%) connections, with $\sigma = 1$. Again the master and servant rasters are the same. 51

2.7 Example of rather complex "chaotic" dynamics retrieved by a the LP estimation defined in (2.7) using the master / servant paradigm with 50 neurons fully connected, initial conditions $D = 3$ and observation time $T = 200$, used here to validate the method. 52

2.8 Example of periodic dynamics retrieved by a the LP estimation defined in (2.7) using the master / servant paradigm, here a periodic raster of period 30 is observed during 8.3 periods. ($N = 30$, $T = 300$ and $D = 3$) As expected from by the theory, the estimated dynamics remains periodic after the estimation time, thus corresponding to a parsimonious estimation. 53

2.9 Weights distribution (positive and negative) used to generate delayed weights, with $D = 10$. 53

2.10 An example of periodic dynamics obtained with excitatory weights profiles shown in the top-left view (master weight's profile), with $N = 30$, $\gamma = 0.98$, $D = 10$ $T = 100$. The estimated weights profile (servant weight's profile) is shown in the top-right view. To generate such trivial periodic raster, shown in the bottom view, only weights with a delay equal to the period have not zero values. This corresponds to a minimal edge of the estimation simplex, this parsimonious estimation being a consequence of the chosen algorithm. 54

2.11 An example of non-trivial dynamics,with $N = 30$, $\gamma = 0.98$, $D = 10$ $T = 100$. Profiles corresponding to the master's excitatory profiles are superimposed in the top-left figure, those corresponding to the master's inhibitory profiles are superimposed in the top-left figure. The estimated raster is shown in the bottom view. This clearly shows that, in the absence of additional constraint, the optimal solution corresponds to a wide distribution of weight's profiles. 55

2.12 In this figure we show that whatever be the weights and delays in the master (left), with $N = 20$, $\gamma = 0.98$, $D = 10$ $T = 100$, the estimator uses all the weights and delays to calculate the raster, in order to obtain a solution. 56

2.13 Two examples of observation of the raster period, on the slave network, observing the raster after the time T where it matches the master raster (shown by an arrow in the figure). (a) With $N = 20$, $\gamma = 0.98$, $\sigma = 5$, $D = 5$, $T = 50$, a periodic regime of periode $P = 1$ is installed after a change in the dynamics. (b) With $N = 20$, $\gamma = 0.98$, $\sigma = 5$, $D = 5$, $T = 50$, a periodic regime of periode $P = 1$ corresponds to the master periodic regime. 57

2.14 Relation between the number of hidden neurons S and the observation time T, here $N = 10$, $T = 470$, $D = 5$, $\gamma = 0.95$ for this simulation. The right-view is a zoom of the left view. This curves shows the required number of hidden neurons, using the proposed algorithm, in order to obtain an exact simulation. We observe that $S = \frac{T}{D} - N$, thus that an almost maximal number of hidden neuron is required. This curve has been drawn from 45000 independent randomly selected inputs. 58

2.15 Finding the expected dynamics from a raster with uniform distribution. (a), (c), (e) and (g) correspond to different raster with Bernoulli distribution. In addition (b), (d), (f) and (h) show the raster calculated by the methodology proposed. The red lines correspond to initial conditions (initial raster), the black ones are the spikes calculated by the method and the blue ones are the spikes in the hidden layer obtained with a Bernoulli Distribution. We can also observe that the number of neurons in the hidden layer increases, 1 by 1, between (b), (d), (f) and (h), this is because the observation time T is augmented by 4, as predicted. Here $N = 5$, $\gamma = 0.95$, $D = 3$; in (a)(b) $T = 15$ with $S = 0$, in (c)(d) $T = 19$ with $S = 1$, in (e)(f) $T = 23$ with $S = 2$, in (g)(h) $T = 27$ with $S = 3$, while S correspond to the number of neurons in the hidden layer, detailed in the text. 59

2.16 Raster calculated, by the proposed method, from a highly correlated Gibbs distribution. Here $r = -1$, $C_t = -0.5$ and $C_0 = -1$. The other parameters are $N = 4$, $\gamma = 0.95$, $D = 3$, $T = 330$ with $S = 106$. The red lines correspond to initial conditions (initial raster), the black ones are the input/output spikes and the blue ones are the spikes in the hidden layer. . . 60

2.17 Raster calculated, by the proposed method, from a very sparse raster, with $N = 30$, $\gamma = 0.95$, $D = 3$, $T = 100$ and $S = 23$. The hidden neurons derived by the present algorithm simply allow to maintain the network activity in order to fire the sparse spikes at the right time. Color codes are the same as previously. 61

2.18 Raster calculated, by the proposed method, from a biological data set, with $N = 50$, $\gamma = 0.95$, $D = 3$ $T = 391$ and $S = 80$. From Riehle et al. (2000) by the courtesy of the authors. . . . 62

2.19 Raster calculated, by the proposed method, from a biological data set, with $N = 50$, $\gamma = 0.95$, $D = 5$ $T = 291$ and $S = 8$. From Riehle et al. (2000) by the courtesy of the authors. . . . 63

2.20 Input-Output dynamics matching, the purple lines represent the input dynamics, the black ones are the OR function of the inputs, it means that if at least one of the input neurons fire a spike in t the output fire a spike in $t + 1$, finally the red ones represent the initial conditions. $Ni = 5$, $No = 1$, $D = 0$, $T = 100$ and $S = 6$. Exact matching (diff $= 0$). 63

2.21 Input-Output dynamics matching, the purple lines represent the input dynamics, the black ones are the OR function of the inputs, it means that if at least one of the input neurons fire a spike in t the output fire a spike in $t+1$, finally the red ones represent the initial conditions. $Ni = 10$, $No = 1$, $D = 0$, $T = 100$ and $S = 10$. Approximate matching (diff = 2). 64

2.22 Input-Output dynamics matching, the purple lines represent the input dynamics, the black ones are the OR function of the inputs, it means that if at least one of the input neurons fire a spike in t the output fire a spike in $t+1$, finally the red ones represent the initial conditions. $Ni = 15$, $No = 1$, $D = 0$, $T = 300$ and $S = 15$. Approximate matching (diff = 21). 64

3.1 An input/ouput transformation based on spikes. 69

3.2 Defining the mollification of the Heaviside function $H()$. It is drawn here for $\nu = 0$ and in $black$, $brown$, red, $orange$, $yellow$, $green$, $blue$, for $\upsilon = [1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]$, respectively. The curves are convex below the magenta horizontal line. . . . 73

3.3 **A** Left view: a typical non-convex M-estimator profile $\varrho()$, for robust estimation: for small errors the profile corresponds to a convex, e.g. quadratic shape, while for higher errors the contribution "saturates" and the sample is no more taken into account in the variational minimization. **B** Middle view: showing a few trials of a L-estimator, for a set $\mathcal{S}$ with expected results r, an outlier o and inliers i corresponding to another spurious trial set; the trial (1) contains expected result and outlier, thus generates large marginal errors; the trial (2) contains expected results only, thus generates several small marginal errors; the trial (3) contains expected results only, but is too imprecise to generate several small marginal errors; the trial (4) contains inliers, thus generates several small marginal errors but for a smaller number of results than the trial (2). If the trial with a maximal number of small marginal errors is kept, i.e. trial (2) in this example, the method is successful. **C** Right-view: qualitative view of the marginal errors histogram for a "correct" (in plain line) versus "spurious" trials (in dashed lines): a large amount of small marginal errors is expected in the former case, while a dispersion of errors is expected in the second case. A typical threshold value, as used in a RANSAC method, is represented by the vertical dot line. 80

3.4 Defining the mollification of the reset function $\rho_\varepsilon = 1 - \sigma_\varepsilon$, and the activation function σ_ε (left view), for $\theta = 1$. It is drawn here in $black$, $brown$, red, $orange$, $yellow$, $green$, for $\varepsilon = [1, 0.5, 0.2, 0.1, 0.05, 0.01]$, respectively. Comparing the bounded profile (drawn in $black$ in the right view) for $\theta = \varepsilon = 1$ with a sigmoid profile of the same maximal slope (in $green$) or the same surface (in $blue$), showing that profiles are very similar though the former has a bounded support. 91

4.1 A FPGA xilinx board (Illustration from Xilinx throughout wikipedia) 99
4.2 Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU (Illustration from Kirk and Hwu (2003)) 100
4.3 A FPGA-based architecture. 114
4.4 Block diagram of individual discrete-time spiking neuron . . . 116
4.5 Analog-spiking neuron model 118
4.6 A GPU-based architecture 120
4.7 Numerical results on periodic dynamics ($N = 4$, $T = 50$). . . . 126
4.8 Numerical results on periodic dynamics ($N = 4$, $T = 50$). . . . 127
4.9 Numerical results on quasi-periodic dynamics ($N = 10$, $T = 50$). 127
4.10 Numerical results on quasi-periodic dynamics ($N = 10$, $T = 50$). 128
4.11 Numerical results on quasi-periodic dynamics ($N = 4$, $T = 50$). 128
4.12 Numerical results on semi-chaotic dynamics ($N = 4$, $T = 50$). . 129
4.13 Numerical results on semi- chaotic dynamics ($N = 4$, $T = 50$). 129
4.14 Numerical results on semi-chaotic dynamics ($N = 4$, $T = 50$). . 130
4.15 Numerical results on chaotic dynamics ($N = 10$, $T = 50$). . . . 130
4.16 Numerical results on chaotic dynamics ($N = 10$, $T = 50$). . . . 131
4.17 Numerical results on chaotic dynamics ($N = 10$, $T = 50$). . . . 131

List of Tables

4.1 Precision for different values of b, where b is the fractional part in the fixed-point representation. 123
4.2 Precision for different values of b, where b is the fractional part in the fixed-point representation. (Continuation) 123
4.3 Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$. 123
4.4 Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$. 124
4.5 Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$. 124
4.6 Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$. 125

Introduction

Neuroscience is an amazing field of knowledge described by multidisciplinary bodies of science. It is aimed to study the nervous system and to understand the biological basis of its behavior. The fast-growing of this field carried out by biologists gave inspiration to physicists and mathematicians to develop models permitting the analysis of the nervous system behavior in three directions. First, at the cellular level, where fundamental questions are addressed to understand the mechanisms of how neurons process signals physiologically and electrochemically. Second, at the systems level, where the questions addressed include how the circuits are formed, using anatomically and physiologically to produce the physiological functions, such as reflexes, sensory integration, motor coordination, emotional responses, learning and memory. Finally at the cognitive level, where the goal is to know how psychological/cognitive functions are produced by the neural circuitry. In recent years these questions have been studied from a rather pragmatical aspect thanks to a new branch, called *Computational Neuroscience*, which is the study of brain function in terms of the information processing properties of the structures that make up the nervous system.

The present manuscript aims to better understand the dynamics of a spiking neural networks and to develop computational methods who permit us to implement new calculations paradigms, in both, software and hardware. The thesis is divided in four chapters. First we briefly review the theoretical background used in the sequel, while generalized Integrate-and-Fire (gIF) type neuron models and their mathematical properties are presented. Then, in chapter 2 we present a method that permits us to estimate properly the parameters (delayed synaptic weights) of a spiking neural network in order to reproduce neural dynamics. This idea is also applied to perform input-ouput transformations, where the goal is to learn exactly the parameters of such transformation. In chapter 3 we discuss the previous idea to develop a mechanism that enables the approximation of spiking dynamics, by using a mollification in the spiking metrics combined with a learning method. Finally in chapter 4 we present hardware implementations for the proposed spiking neuron models, where efficient architectures based on FPGA and

GPU have been developed.

PART I: SPIKING NEURONS

Neurons in the brain communicate by short electrical pulses, the so-called actions potentials or spikes. Theoretically, spiking neurons can perform very powerful computations with precise timed spikes. They are at least as computationally powerful as the sigmoidal neurons traditionally used in artificial neural networks Maass (1997); Maass and Natschlager (1997). This results has been shown using a spike-response model (see Maass and Bishop (2003) for a review) and considering piece-wise linear approximations of the potential profiles. In this context, analog inputs and outputs are encoded by temporal delays of spikes. The authors show that any feed-forward or recurrent (multi-layer) analog neuronal network (e.g., McCulloch and Pitts (1943)) can be simulated arbitrarily closely by an insignificantly larger network of spiking neurons. This holds even in the presence of noise Maass (1997); Maass and Natschlager (1997). These results highly motivate the use of spiking neural networks, as studied here.

Among the wide variety of neuron models, the generalized Integrate and Fire (gIF) model stands out thanks to its relative simplicity and its ability in reproducing many behaviors observed in real neurons. In this direction a new model derived from the LIF model has been introduced in Soula et al. (2006) and analyzed in Cessac (2008, 2010). The model is a discrete-time version of the LIF model or in other words it is a discrete-time spiking neuron model. Thus, the discretization makes possible to have a one-to-one correspondence between the dynamics of the membrane potential and the sequences of spike patterns ("raster plots", in the asymptotic dynamics, Cessac (2008)). Further this correspondence permit us to focus on information processing Gerstner and Kistler (2002a) and to switch easily from one representation to the other. The model is also able to produce simple (periodic) and complex (chaotic) dynamics.

In the present work the original form of the discrete-time spiking neuron model is modified in two ways. On one hand we add transmission interneural delays in the model and synaptic time-responses. In this way, we get closer to gIF models introduced by Rudolph and Destexhe (2006) and analyzed in Cessac and Viéville (2008). On the other hand considering an analog-spiking neuron model, which has the same reset mechanism than the discrete case and where the activation function is given by a non-linear function (e.g. sigmoid) instead of a function handled by a binary signal as in the discrete model.

PART II: LEARNING THE PARAMETERS OF A NEURAL NETWORK MODEL

Neuronal networks have tremendous computational capacity, but their biological complexity makes the exact reproduction of all the mechanisms involved in these networks dynamics essentially impossible, even at the numerical simulation level, as soon as the number of neurons becomes too large. One crucial issue is thus to be able to reproduce the "output" of a neuronal network using approximated models easy to implement numerically. The issue addressed here is "Can we program an Integrate-and-Fire network, i.e. tune the parameters, in order to exactly reproduce another network output, on a bounded time horizon, given the input".

The main aspect we are interesting here is the *calculability* of neural network models. It is known that recurrent neural networks with frequency rates are universal approximators Schäfer and Zimmermann (2006), as multilayer feed-forward networks are Hornik et al. (1989). This means that neural networks are able to simulate dynamical systems, not only to approximate measurable functions on a compact domain, as originally stated (see, e.g., Schäfer and Zimmermann (2006) for a detailed introduction on these notions). Spiking neuron networks can be also universal approximators Maass (2001).

In a computational context, spiking neural networks are mainly implemented through specific network architectures, such as Echo State Networks Jaeger (2003) and Liquid Sate Machines Maass et al. (2002), that are called "reservoir computing" (see Verstraeten et al. (2007) for unification of reservoir computing methods at the experimental level). In this framework, the *reservoir* is a network of neurons (it can be linear or sigmoidal neurons, but more usually spiking neurons), with a random topology and a sparse connectivity. Usually, this is a recurrent network, with weights than can be either fixed or driven by an unsupervised learning mechanism. In the case of spiking neurons (e.g. in the model of Paugam-Moisy et al. (2008)), the learning mechanism is a form of synaptic plasticity, usually STDP (Spike-Time-Dependent Plasticity), or a temporal Hebbian unsupervised learning rule, biologically inspired. The output layer of the network (the so-called "readout neurons") is driven by a supervised learning rule, generated from any type of classifier or regressor, ranging from a least mean squares rule to sophisticated discriminant or regression algorithms. The ease of training and a guaranteed optimality guides the choice of the method. It appears that simple methods yield good results Verstraeten et al. (2007). This distinction between a readout layer and an internal reservoir is indeed induced by the fact that only the output of the neuron network activity is constrained,

whereas the internal state is not controlled.

In biological context, learning is mainly related to synaptic plasticity Gerstner and Kistler (2002b); Cooper et al. (2004) and STDP (see e.g., Toyoizumi et al. (2007) for a recent formalization), as far as spiking neuron networks are concerned. This unsupervised learning mechanism is known to reduce the variability of neuron responses Bohte and Mozer (2007) and is related to the maximization of information transmission Toyoizumi et al. (2005) and mutual information Chechik (2003). It has also other interesting computational properties such as tuning neurons to react as soon as possible to the earliest spikes, or segregate the network response in two classes depending on the input to be discriminated, and more general structuring such as emergence of orientation selectivity Guyonneau et al. (2005).

In the present study, the viewpoint is quite different: we consider supervised learning, where "each spike matter", as e.g. in the special case of a feed-forward sweep of visual activity in response to a brief visual presentation Guyonneau et al. (2005); Delorme et al. (2001); thus, we want, not only to statistically reproduce the spiking output, but also to reproduce it exactly.

The motivation to explore this track is twofold. On one hand, we want to better understand what can be learned, at a theoretical level, by spiking neural networks, tuning weights and delays. The key point is the non-learnability of spiking neurons Šíma and Sgall (2005), since it is proved that this problem is NP-complete, when considering the estimation of both weights and delays. Here we show that we can "elude" this caveat and propose an alternate efficient estimation, inspired by biological models.

We also have to notice that the same restriction apply not only to simulation but, as far as this model is biologically plausible, also holds at the biological level. It is thus an issue to wonder if, in biological neuron networks, delays are really estimated during learning processes, or if a weaker form of weights adaptation, as developed now, is considered.

On the other hand, the computational use of spiking neuron networks in the framework of reservoir computing or beyond Schrauwen (2007), at application levels, requires efficient tuning methods not only in "average", but in the deterministic case. This is the reason why we must consider how to *exactly* generate a given spike train.

PART III: PROGRAMMING RESETTING NON-LINEAR NETWORKS

A deterministic framework, i.e. in a context where not the "average" input/output response, but an exact or approximate *specific* input/output response is targeted (i.e. considering that "each spike may matter" Guyonneau

et al. (2005); Delorme et al. (2001), which seems to be the case, e.g., in the biological visual system working in natural scenario Baudot (2007); Masland and Martin (2007); Koch et al. (2004)). More precisely, from Cessac et al. (2010), we propose the following pragmatic view of the network result coding scheme (i.e. the "neural code" in a biological context VanVreeswijk (2004); Rieke et al. (1996)): two results correspond approximately to the same code if their *distance* with respect to a given metric or pseudo-metric is small. For instance, considering a rank coding scheme (i.e. in a context where the *relative temporal sorting* of the spikes matter, but not their exact temporal values Delorme et al. (2001)) the related pseudo-metric is discrete and easy to state: network output are equivalent if the ranks of spike trains match, and not-equivalent otherwise. Contrary to this binary choice, our proposal is to introduce a richer structure: The proposed modeling view is not only to consider a weak notion of network coding where two codes can only be either equal or different, but a more general notion where two codes are similar up to some quantified distance. This seems to correspond to a more realistic view of, for instance, the still mysterious "neural code" (see Cessac et al. (2010) for a discussion), and at the estimation level allows variational optimization mechanisms to be used.

This permits us to not only address the *exact* estimation problem, i.e. to find an exact input/output mapping if and only if there is one, but also the *approximate* estimation problem, i.e. to find an approximate input/output mapping that minimizes a given distance.

Thus, we propose a mollification in the spiking metrics, which allows to replace the sudden spike generation by a progressive effect, in order to be able to evaluate its variation. This means that the metric is no more a function of the spike explicitly, but of the state value itself around the spike. Though we are able to evaluate its variation, it is not clear to which extents this trick allows to always solve the following key problem: As soon as an spike is modified, this may induce a dramatic change on the network dynamics, thus completely change spikes in the future, including re-creating false spikes with respect to what is expected. We thus may expect our method to be unusable when the system is chaotic since the effect of an spike modification will be significant after a time of order the inverse of the largest positive Lyapunov exponent.

In the present variational framework, weights are adjusted in order to drive the state values away from the threshold barriers that induce spike changes. As a consequence, it tends to produce a dynamic that is robust and stable, but not necessarily optimal, in the sense that it drives the parameters towards a reasonable, but local optimum of the chosen criterion. In other words, it tends to qualitatively generate the input/output response

with a large margin, which is known to provide statistically robust transformations, with good generalization performances Vapnik (1998); Bartlett and Shawe-Taylor (1999). This is what is targeted here at heuristic level.

It also tends to drive the system away from the "edge of chaos" (see Cessac (2008); Cessac and Viéville (2008) for a precise definition in spike system, different from the usual notion of chaos in differentiable systems : the terminology "stable chaos" has been proposed for this type of behaviour, by Politi and Torcini (2009)).

Though it is believed that maintaining a system close to the edge of chaos increases the versatility of its computational power, while this point of view remains in discussion (see, e.g., Langton (1990); Packard (1988)), the present framework tends to provide something different. The network is not targeted to remain at the edge of chaos, with the goal to allow a large set of computation to be performed, but instead to perform a restrained class of computations, in a robust way. It is an interesting perspective of this work, to study to which extents such variational approach can be turned out toward the alternate objective.

PART IV: HARDWARE IMPLEMENTATIONS OF SPIKING NEURAL NETWORKS

In recent years the hardware implementation of artificial neural networks has evolved to neuromorphic, FPGA-based and GPU-based approaches in order to emulate the biological neural design. In this sense spiking neuron models attempt to reproduce the neural behavior at different realism levels (biologic plausibility). However, realism implies complexity, which from a computational viewpoint it corresponds to high computational costs. In this context, dedicated hardware take advantage of the inherent parallelism in the neural processing to accelerate it. Furthermore the design process of a hardware architecture depends mainly on two aspects: the characteristics of the hardware platform and the complexity of the neuron model Johnston et al. (2005); Belatreche et al. (2006). Thus, in this relation neuron model-hardware platform we can find a wide range of possible combinations (see Misra and Saha (2010) for a recent review of the progress in hardware implementations of neural networks). However, as we know all neuron model is described by mathematical operations, in this sense certain hardware devices are better adapted than others according the complexity of such operations.

Through this huge universe of hardware implementations of neural networks, we focus on those related to spiking neuron models. In this direction, several research groups have concurred in important projects such as

FACETS[9] and SYNAPSE[10], where have as common goal to develop neuromorphic hardware in order to simulate large scale of realistic neuron models. Currently, there exist a large number of environments for spiking neural networks that enable the simulation of realistic neuron models. For example the NEURON[11] simulator is well-suited option when we require to simulate many ion specific channels and ion accumulation involved in complex models of membrane cell Hines and Carnevale (1997). GENESIS[12] is another simulator developed by Wilson et al. (1989) and widely used in the neuroscience community to simulate neural systems ranging from subcellular components and biochemical reactions to complex models of single neurons, simulations of large networks, and systems-level models. In the last years some simulators, such as SpikeNNS[13], SpikeNET[14], SpikeLAB[15] and BRIAN[16] attempt to simulate large scale neural networks based on spiking neuron models at different realism levels. However, only the SpikeLAB simulator enable the integration of digital or analog neuromorphic circuits in the simulation process Grassmann and Anlauf (1999).

From a more specific level we can find hardware implementations of spiking neuron models in analog and digital domains and also a combination of both. Thus, analog-based implementations focus mainly on more realistic neuron models, this fact is due that in the analog domain the non-linearity can be captured directly Mahowald and Douglas (1991); Tomas et al. (2006); Lewis and Renaud (2007); Renaud et al. (2007); Schemmel et al. (2008). In contrast to analog implementations, digital devices such as FPGA provide an efficient and low-cost programmable platform for implementing spiking neuron models Maya et al. (2000); Graas et al. (2004); Glackin et al. (2005); Cassidy et al. (2007); Girau and Torres-Huitzil (2007); Maguire et al. (2007); Pearson et al. (2007); Thomas and Luk (2009). However, the circuit density in FPGAs is still limited to implement large scale models with a big number of interconnected neurons.

Further, neuromorphic systems have been efficiently implemented in Vogelstein et al. (2007); Hashimoto and Torikai (2009); Saighi et al. (2010). Here, authors combine analog and digital hardware to simulate neural networks based on complex neuron models. On one hand the realism involved in complex models is captured by the analog hardware. On the other hand the connectivity among neurons is handle by digital hardware i.e. FPGA or

[9] http://facets.kip.uni-heidelberg.de
[10] http://celest.bu.edu/outreach-and-impacts/the-synapse-project
[11] http://www.neuron.yale.edu/neuron/
[12] http://www.genesis-sim.org/GENESIS/
[13] http://cortex.cs.nuim.ie/tools/spikeNNS/index.html
[14] http://sccn.ucsd.edu/ arno/spikenet/
[15] http://spikelab.jbpierce.org/
[16] http://www.briansimulator.org/

PC.

Moreover GPU's have been frequently used to solve complex mathematical problems Kirk and Hwu (2003). GPU uses an heterogeneous programming scheme, which allows us to use different programming language in order to simulate the related mathematical model. This means that a GPU programming language can easily interact with other programming languages permitting us to have a programming scheme more flexible and optimal. The programmable GPU has evolved into a highly parallel, multithreaded, multi-core processor with tremendous computational power and very high memory bandwidth far beyond the graphical applications they have been designed initially for. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations, the same program being executed on many data elements in parallel Che et al. (2008). In this direction, Spiking Neural Networks could take profit of this powerful technology, since a Spiking Neuron Model could be coded as a GPU kernel and reproduce several times to build a neural network, therefore executed in parallel Fidjeland et al. (2009); Nageswaran et al. (2009).

The aim of this section is to show the feasibility of the gIF-type neuron models to be implemented on dedicated hardware, such as FPGA. On one hand we present the FPGA-based implementation of a discrete-time spiking neuron model, where we show that such model avoid the use of multipliers in the weighting sum, thus the resource usage in the device is highly reduced. Besides, this model allows the one-to-one correspondence between membrane potential and spiking activity as was shown in Cessac (2008). Furthermore this correspondence allows that the spiking activity can be directly and easily handle by the FPGA due that the spiking activity is represented as a sampled signal (spike train). On the other hand we present the implementation of an analog-spiking neuron model, which permits us to combine the spiking and the analog approaches to reproduce spiking activities. More specific this model uses a non-linear function to evaluate the synaptic interactions among neurons but at the same time the spiking activity continue being described by a sampled signal (spike train). In both cases we perform a precision analysis by considering a fixed-point arithmetic.

The mathematical study presented in Cessac (2008) has evidenced three different activity regimes on the Integrate and Fire neuron models. In this work the author has also identified numerically the limits among these regimes. Following this research we have extended such study to the field of the hardware implementation. Hence, we show several results that demonstrate such phenomena and its consequences in the design of FPGA-based architectures of gIF-type neuron models.

Finally we present a performance analysis where we compare the FPGA-based implementations with a GPU-based implementation.

Part I

gIF-type Spiking Neuron Models

CHAPTER 1

gIF-TYPE SPIKING NEURON MODELS

"You must keep an open mind, but not so open that your brains fall out"
–James Oberg ()

OVERVIEW

The aim of this chapter is to briefly review the principles of neuroscience and the modeling basis of the neural activity. This short description addresses two scales: first at cellular level, studying the basic neuron physiology, and second exploring its behavior at a system level (neural network). Then, we review a mathematical model, which defines the dynamics of spiking neurons. In this sense the model is based on the fact that a spike is described as an action potential emitted by a neuron, when it has been excited enough to reach a given positive threshold. Besides, from the spike state of the neuron, and the membrane potential, we can deduce that there exist an one-to-one correspondence between spike train and membrane potential trajectories. This correspondence permits us to switch from the membrane potential to spikes when focus on information processing, while neural code (information) is encoded as a spike activity.

We focus on discrete-time spiking neuron model described by a simplified form of the gIF-type model. The main interest to consider this model as the basis of this thesis is due to its simplicity to describe the neural behavior, and the one-to-one correspondence between the membrane potential and its respective spiking activity in the asymptotic stage: no information is lost when switching between representations, even if the spiking sequences have a complex structure. Also, this model has a reasonable biological plausibility since it can approximate complex dynamics.

1.1 Neuron Anatomy

A neuron is an electrically excitable cell that processes and transmits information by electrical and chemical signaling. Chemical signaling occurs via synapses, specialized connections with other cells. Neurons connect to each other to form **networks**. Given the diversity of functions performed by neurons in different parts of the nervous system, there is, as expected, a wide variety in the shape, size, and electrochemical properties of neurons. A typical neuron can be divided into three anatomical and functional parts, called dendrites, soma and axon; see figure 1.1.

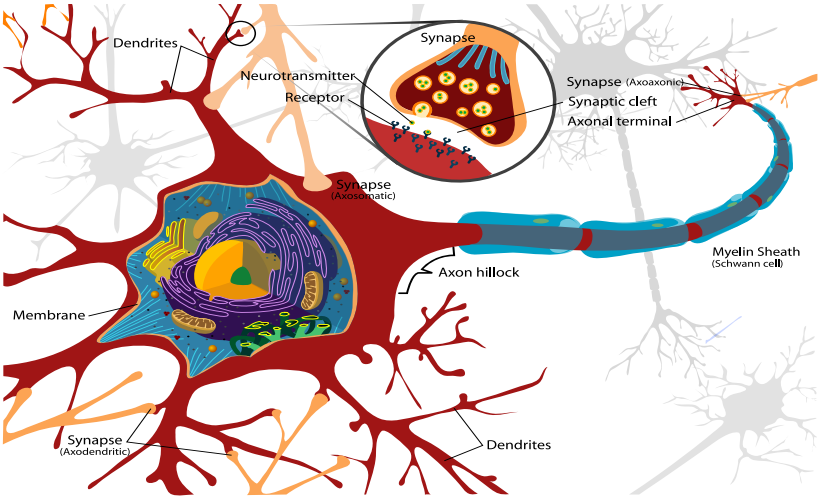


Figure 1.1: Anatomy of a neuron. (Illustration from Mariana Ruiz Villarreal)

- The **soma** is the central part of the neuron. It contains the nucleus of the cell, where most protein synthesis occurs. The soma is considered as a central processing unit that performs an important nonlinear processing.
- The **dendrites** of a neuron are cellular extensions with many branches. This overall shape and structure is referred to as a dendritic tree. This is where the majority of inputs to the neuron occurs. It is considered as a linear combiners of these inputs.
- The **axon** is a fine cable-like projection which can extend tens, hundreds, or even tens of thousands of times the diameter of the soma in length. The axon carries nerve signals away from the soma (and also carries some types of information back to it). Neurons have only one axon, but this axon may and usually will undergo extensive branching, enabling communication with many target cells. The part of the

axon where it emerges from the soma is called the axon hillock. Besides being an anatomical structure, the axon hillock is also the part of the neuron that has the greatest density of voltage-dependent sodium channels. This makes it the most easily-excited part of the neuron and the **spike** initiation zone for the axon: in electrophysiological terms it has the most negative action potential threshold. While the axon and axon hillock are generally involved in information outflow, this region can also receive input from other neurons. The **axon terminal** contains synapses, specialized structures where neurotransmitter chemicals are released in order to communicate with target neurons.

1.1.1 The spiking neuron

The neuron is a dynamic element that emits output pulses whenever the excitation exceeds some threshold. The resulting sequence of pulses or “spikes” contains all the information that is transmitted from one neuron to the next.

Action potentials

In physiology, an action potential (Figure 1.2) is a short-lasting event in which the electrical membrane potential of a cell rapidly rises and falls, following a stereotyped trajectory. Action potentials play a central role in cell-to-cell communication. In other types of cells, their main function is to activate intracellular processes. Action potentials in neurons are also known as “nerve impulses” or “spikes”, and the temporal sequence of action potentials generated by a neuron is called its "spike train". A neuron that emits an action potential is said to“fire”.

1.2 Discretized integrate and fire neuron models.

Let us now consider a normalized and reduced “punctual conductance based generalized integrate and fire” (gIF) neural unit model Destexhe (1997) as reviewed in Rudolph and Destexhe (2006). The model is reduced in the sense that both adaptive currents and non-linear ionic currents are no more explicitly depending on the potential membrane, but on time and previous spikes only (see Cessac and Viéville (2008) for a development).

Here we follow Cessac (2008); Cessac and Viéville (2008); Cessac and Viéville (2008) after Soula and Chow (2007) and review how to properly discretize a gIF model.

[2]NIA - Alzheimer’s Disease: Unraveling the Mystery. National Institutes of Health. http://www.nia.nih.gov/Alzheimers/Publications/Unraveling/

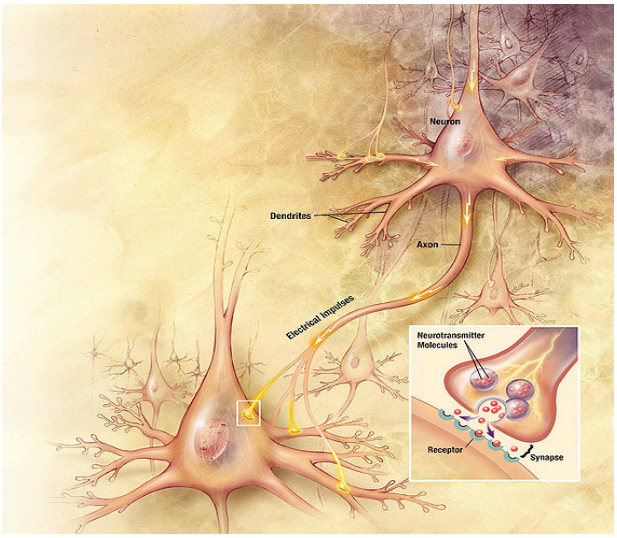


Figure 1.2: Action potentials arriving at the synapses of the upper right neuron stimulate currents in its dendrites; these currents depolarize the membrane at its axon, provoking an action potential that propagates down the axon to its synaptic knobs, releasing neurotransmitter and stimulating the post-synaptic neuron (lower left). Illustration from [2]

1.2.1 From a LIF model to a discrete-time spiking neuron model.

In this section we present the correct discretization of a LIF model, which has been introduced in Soula et al. (2006) and called BMS. The discretization process is carried out thanks to the well-done analogy presented in Gerstner and Kistler (2002a) between a RC circuit and a biological neuron (figure (1.3)).

The analogy represented in figure 1.3 permits us to study the neuron behavior using the well-defined techniques of the electrical circuits analysis. In this direction applying the Kirchhoff's current law to the RC circuit shown in figure (1.3) we have:

$$I_i(t) = I_R + I_C \tag{1.1}$$

then applying the Ohm's law and deriving the current-voltage relation on the capacitor, yields in a differential equation:

$$I_i(t) = \frac{V_i(t)}{R} + C\frac{dV_i(t)}{dt}$$

$$\frac{dV_i(t)}{dt} = -\frac{V_i(t)}{\tau} + \frac{I_i(t)}{C}$$

where $\tau = RC$ is the time required for the voltage to fall to V_0 and is called the time constant.

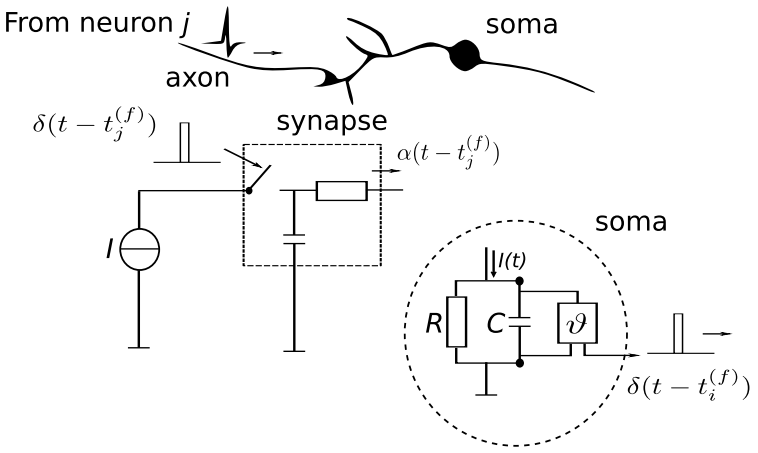


Figure 1.3: Schematic diagram of the integrate-and-fire model. The basic circuit is the module inside the dashed circle on the right-hand side. A current $I(t)$ charges the RC circuit. The voltage $u(t)$ across the capacitance (points) is compared to a threshold ϑ. If $u(t) = \vartheta$ at time $t_i^{(f)}$ and output pulse $\delta(t - t_i^{(t)})$ is generated. Left part: a presynaptic spike $\delta(t - t_j^{(t)})$ is low-pass filtered at the synapse and generates an input current pulse $\alpha(t - t_j^{(t)})$ (Illustration from Gerstner and Kistler (2002a)).

Furthermore, we consider that the spiking neuron model will be used in numerical simulations, thus the Euler method is applied in order to solve the differential equation. Then we fix some constant values, such as the sampling time scale $dt = 1$ and the capacitance $C = 1$. Finally, we consider that $\gamma = 1 - \frac{1}{\tau}$, where $\tau \geq 1$, thus $\gamma \in [0, 1[$, the equation reads:

$$\frac{V_i(t+dt) - V_i(t)}{dt} = -\frac{V_i(t)}{\tau} + \frac{I_i(t)}{C}$$

$$V_i[k+1] = V_i[k] - \frac{V_i[k]}{\tau} + \frac{I_i(t)}{C}$$

$$V_i[k+1] = V_i[k]\Big(1 - \frac{1}{\tau}\Big) + \frac{I_i[k]}{C}$$

$$V_i[k+1] = \gamma V_i[k] + \frac{I_i[k]}{C}$$

$$V_i[k+1] = \gamma V_i[k] + I_i[k]$$

The next is to consider that $I_i[k]$ is the sum of all signals flowing into the neuron, it is $I_i[k] = I_i^S[k] + I_i^{ext}[k]$. Hence, on one hand the synaptic current $I_i^S[k]$ describes the strength of the connections among neurons and is defined by $\sum_{j=1}^{N} W_{ij}\chi(V_j[k])$, where χ is the activation function described by the equation 1.3 as the firing state of the pre-synaptic neurons. On the

other hand $I_i^{\text{ext}}[k]$ corresponds to a current from an external stimulus. These assumptions allow us to define a preliminary approximation towards the discrete-time spiking neuron model as follow:

$$V_i[k+1] = \gamma V_i[k] + \sum_{j=1}^{N} W_{ij}\chi(V_j[k]) + I_i^{\text{ext}}[k]$$

Furthermore the term $Z[k]$ is used to specify the indicator function $\chi_{[1,+\infty[}(V[k])$, $\chi_A(x) = 1$ if $x \in A$ and 0 otherwise. More precisely Z defines the firing state of the neuron. Finally, we introduce a reset mechanism $(1 - Z[k])$ in order to return the neuron to its initial state, once that it has reached the fixed firing threshold θ. Thus, the discrete-time neuron model reads:

$$V_i[k] = \gamma V_i[k-1](1 - Z_i[k-1]) + \sum_{j=1}^{N} W_{ij} Z_j[k-1] + I_i^{\text{ext}}[k] \tag{1.2}$$

$$Z_i[k] = \chi(V[k]) = \begin{cases} 1 & \text{if } V_i[k] \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{1.3}$$

where $V_i[k]$ defines the membrane potential of the neuron with index i at time t. W are the synaptic weights. γ corresponds to the leak factor and I^{ext} is an external stimulus.

1.2.2 Time constrained continuous formulation.

Let V be the normalized membrane potential, which triggers a spike for $V = 1$ and is reset at a given value. This is an approximation characteristic of IF models: the reset value is a constant. This constraint can be released by adding noise in the system. The fire regime (spike emission) reads $V(t) = 1 \Rightarrow V(t^+) = 0$. Let us write $\tilde{\omega}_t = \chi(V(t)) = \{\cdots t_i^n \cdots\}$, the list of spike times $t_i^n < t$. Here t_i^n is the n-th spike-time of the neuron of index i. The dynamic of the integrate regime reads:

$$\frac{dV}{dt} + \frac{1}{\tau_L}\left[V - E_L\right] + \sum_j \sum_n \rho_j\left(t - t_j^n\right)\left[V - E_j\right] = I_m(\tilde{\omega}_t),$$

Here, τ_L and E_l are the membrane leak time-constant and reverse potential, while $\rho_j(t)$ and E_j are the spike responses and reversal potentials for excitatory/inhibitory synapses and gap-junctions. Furthermore, $\rho_j(t)$ is the synaptic or gap-junction response, accounting for the connection delay and time constant; shape showed in figure 1.4.

Finally, $I_m()$ is the reduced membrane current, including simplified adaptive and non-linear ionic current (see Cessac and Viéville (2008) for details).

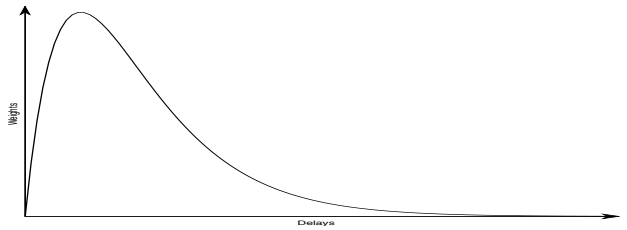


Figure 1.4: A profile $\rho(t) = H(t)\frac{t}{\tau}e^{-\frac{t}{\tau}}$, which represents the synaptic weights mapped at different delays.

Consequently the dynamic of the integrate regime reads:

$$\frac{dV}{dt} + g(t, \tilde{\omega}_t)\, V = I(t, \tilde{\omega}_t),$$

where g and I are the membrane conductance and the current respectively and their value only depend on the firing states $\tilde{\omega}$. Once that we know the membrane potential at time t, we can deduce it at time $t + \delta$, which reads:

$$V(t+\delta) = \nu(t, t+\delta, \tilde{\omega}_t)\, V(t) + \int_t^{t+\delta} \nu(s, t+\delta, \tilde{\omega}_s)\, I(s, \tilde{\omega}_s)\, ds$$

with:

$$\nu(t_0, t_1, \tilde{\omega}_{t_0}) = e^{-\int_{t_0}^{t_1} g(s, \tilde{\omega}_s)\, ds}.$$

The key point is that temporal constraints must be taken into account Cessac et al. (2008b). Spike-times are bounded by a refractory period $r, r < d_i^{n+1}$, defined up to some absolute precision δt, while there is always a minimal delay dt for one spike to influence another spike, and there might be (depending on the model assumptions) a maximal inter-spike interval D such that either the neuron fires within a time delay $< D$ or remains quiescent forever). For biological neurons, orders of magnitude are typically, in milliseconds:

r	δt	dt	D
1	0.1	$10^{-[1,2]}$	$10^{[3,4]}$

1.2.3 Network dynamics discrete approximation.

Combining these assumptions with equations 1.2 and 1.3 we can to write the following discrete equation for a sampling period δ:

$$V_i[k] = \gamma_i\, V_i[k-1]\,(1 - Z_i[k-1]) + \sum_{j=1}^{N}\sum_{d=1}^{D} W_{ijd}\, Z_j[k-d] + I_i[k], \qquad (1.4)$$

Thus, the cumulative effects of conductances reads:

$$\gamma_i(t) \equiv \nu(t, t+\delta, \tilde{\omega}_t)|_{t=k\,\delta} \tag{1.5}$$

considering random independent and identically distributed Gaussian weights.

The numerical analysis performed in Cessac and Viéville (2008) demonstrates that, for numerical values taken from bio-physical models, considering here $\delta \simeq 0.1ms$, this quantity is remarkably constant, with small variations within the range:

$$\gamma_i \in [0.965, 0.995] \simeq 0.98,$$

It has been numerically verified that taking this quantity constant over time and neurons does not significantly influence the dynamics. This the reason why we write γ_i as a constant here. This corresponds to a "current based" (instead of "conductance based") approximation of the connections dynamics.

The additive current

$$I_i(t) \equiv \int_t^{t+\delta} \nu(s, t+\delta, \tilde{\omega}_s) \left(i_m(\tilde{\omega}_s) + \frac{E_L}{\tau_L} \right) ds \Big|_{t=k\,\delta} \simeq \delta\,\gamma_i \left(i_m(\tilde{\omega}_t) + \frac{E_L}{\tau_L} \right) \Big|_{t=k\,\delta} \tag{1.6}$$

accounts for membrane currents, including leak. The right-hand size approximation assume γ_i is a constant. Furthemore, we have to assume that the additive currents are independent from the spikes. This means that we neglect the membrane current non-linearity and adaptation.

On the contrary, the term related to the connection weights W_{ijd} is not straightforward to write and requires to use the previous numerical approximation. Let us write:

$$\begin{aligned} W_{ij}[k - k_j^n] &\equiv E_j \int_t^{t+\delta} \nu(s, t+\delta, \tilde{\omega}_t)\, \rho_j \left(t - t_j^n \right) ds \Big|_{t=k\,\delta, t_j^n = k_j^n\,\delta} \\ &\simeq E_j\,\delta\,\gamma_i\,\rho_j \left(t - t_j^n \right) \Big|_{t=k\,\delta, t_j^n = k_j^n\,\delta}, \end{aligned} \tag{1.7}$$

assuming $\nu(s, t+\delta, \tilde{\omega}_t) \simeq \gamma_i$ as discussed previously. This allows us to consider the spike response effect at time $t_j^n = k_j^n\,\delta$ as a function only of $k - k_j^n$. The response $W_{ij}[d]$ vanishes after a delay $D, \tau_r = D\,\delta$, as stated previously. We assume here that $\delta < \delta t$ i.e. that the spike-time precision allows one to define the spike time as $k_j^n, t_j^n = k_j^n\,\delta$ (see Cessac and Viéville (2008); Cessac and Viéville (2008) for an extensive discussion). We further assume that only zero or one spike is fired by the neuron of index j, during a period δ, which is obvious as soon as $\delta < r$.

This allows us to write $W_{ijd} = W_{ij}[d]$ so that:

$$\begin{aligned}\sum_{j=1}^{n} W_{ij}[k-k_j^n] &= \sum_{d=1}^{D}\sum_{j=1}^{n} W_{ij}[d]\chi_{\{k_j^n\}}(k-d)\\ &= \sum_{d=1}^{D} W_{ij}[d]\chi_{\{k_j^1\cdots,k_j^n,\cdots\}}(k-d)\\ &= \sum_{d=1}^{D} W_{ijd}\, Z_j[k-d]\end{aligned}$$

where $Z_j[l] = \chi_{\{k_j^1\cdots,k_j^n,\cdots\}}(l)$ is precisely equal to 1 on spike time and 0 otherwise, thus completes the derivation of (1.4).

1.2.4 Properties of the discrete-time spiking neuron model

Now, we make a review on the main properties of the discrete-time spiking neuron model described by equation 1.4. These properties have been defined in Cessac (2008).

Since $\gamma < 1$ we can define the phase space for V as a set $\mathcal{M} = [V_{\min}, V_{\max}]^N$.

$$V_{\min} \leq V_i[k] \leq V_{\max}, \tag{1.8}$$

where:

$$V_{\min} = \min\Big(0,\ \frac{1}{1-\gamma}\Big[\min_{i=1\ldots N}\sum_{j|W_{ijd}<0} W_{ijd} + I_i^{\text{ext}}\Big]\Big) \tag{1.9}$$

and:

$$V_{\max} = \max\Big(0,\ \frac{1}{1-\gamma}\Big[\max_{i=1\ldots N}\sum_{j|W_{ijd}>0} W_{ijd} + I_i^{\text{ext}}\Big]\Big) \tag{1.10}$$

where the notation $j|W_{ijd} < 0$ ($j|W_{ijd} > 0$) specifies that all the weights are negative (positive).

Equations 1.9 and 1.10 hold even if I_i^{ext} depends on time.

For each neuron we can decompose the interval $\mathcal{I} = [V_{\min}, V_{\max}]$ into $\mathcal{I}_0 \cup \mathcal{I}_1$ with $\mathcal{I}_0 = [V_{\min}, \theta[$ and $\mathcal{I}_1 = [\theta, V_{\max}]$. If $V \in \mathcal{I}_0$ neuron is quiescent and otherwise it fires. Further, this partition permits us to define the *spiking state*, which can be expressed as a N dimensional binary vector, $\boldsymbol{\omega} = \{\omega_1, ..., \omega_N\} \in \Lambda$, where $\Lambda = \{0,1\}^N$. Then

$$\mathcal{M} = \bigcup_{\boldsymbol{\omega}\in\Lambda} \mathcal{M}_{\boldsymbol{\omega}}$$

where:

$$\mathcal{M}_{\boldsymbol{\omega}} = \{\mathbf{V} \in \mathcal{M} | V_i \in \mathcal{I}_{\omega_i}\} \tag{1.11}$$

More precisely, this allows us to classify the membrane potential vectors according to their spiking state.

The coefficient $\gamma(1 - Z(V_i))$ corresponds to contraction in direction i. This can be explained as follow: let us to suppose that $V_i \in \mathcal{I}_1$, which means that $Z(V_i) = 1$ (neuron fires) by consequence the contraction coefficient will be zero and the membrane potential reseted, otherwise the membrane potential will be contracted by a factor γ. These effects are shown in figures 1.5(b) and 1.5(a) respectively.

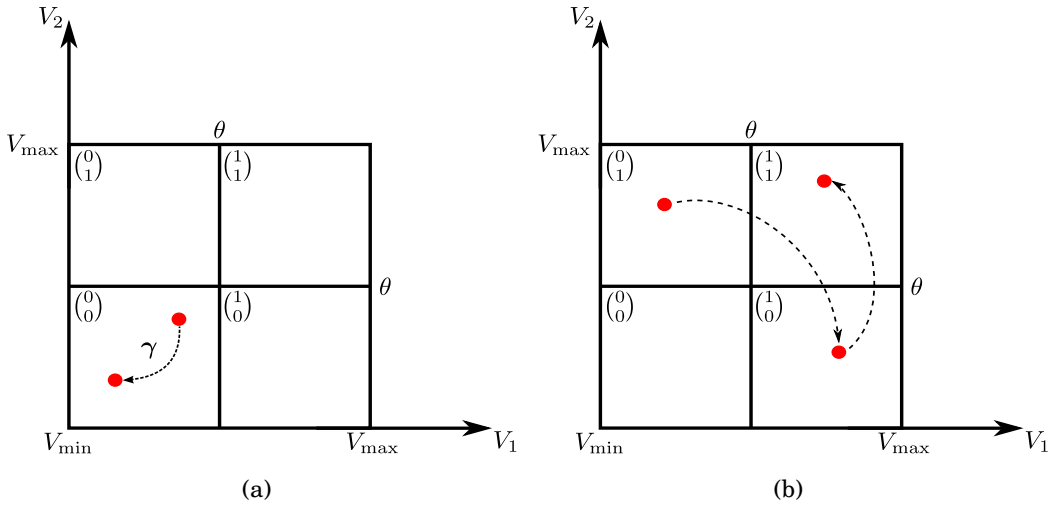


Figure 1.5: Two examples of the partition space for non-perturbed balls in a system with two neurons. The space is partitioned from the firing state of the neurons and is labeled as $\boldsymbol{\omega} = \binom{\omega_1}{\omega_2}$

Furthermore this property can be extended to a $\mathbf{R}^N$ space, since 1.4 is originally defined on this space. If we consider the δ-ball, this ball is contracted by dynamics.

The equation 1.4 is discontinuous on the set $\mathcal{S}$ defined as follow:

$$\mathcal{S} = \{\mathbf{V} \in \mathcal{M}, |\exists i, V_i = \theta\} \tag{1.12}$$

This is due to singularities in dynamics. Let us to consider the trajectory of a point $\mathbf{V} \in \mathcal{M}$, which has perturbations with an amplitude $< \epsilon$. Equivalently to consider the evolution of the ϵ-ball $\mathcal{B}(\mathbf{V}, \epsilon)$ under 1.4.

There are two cases:

1. ***The non-critical case:*** *when $\mathcal{B}$ does not intersects $\mathcal{S}$, $\mathcal{B}(\mathbf{V}, \epsilon) \cap \mathcal{S} = 0$*

   In this scenario the perturbations have not a considerable effect on the trajectory of $\mathbf{V}$, in fact they become asymptotically indistinguishable. At the most, there exist a contraction of the initial ball, since $\gamma < 1$.

   In order to clarify the non-critical case let us to consider the examples shown in figure 1.6. In both cases the real trajectories (Figure 1.5) have been perturbed in $\mathcal{B}(\mathbf{V}, \epsilon)$ ( ).

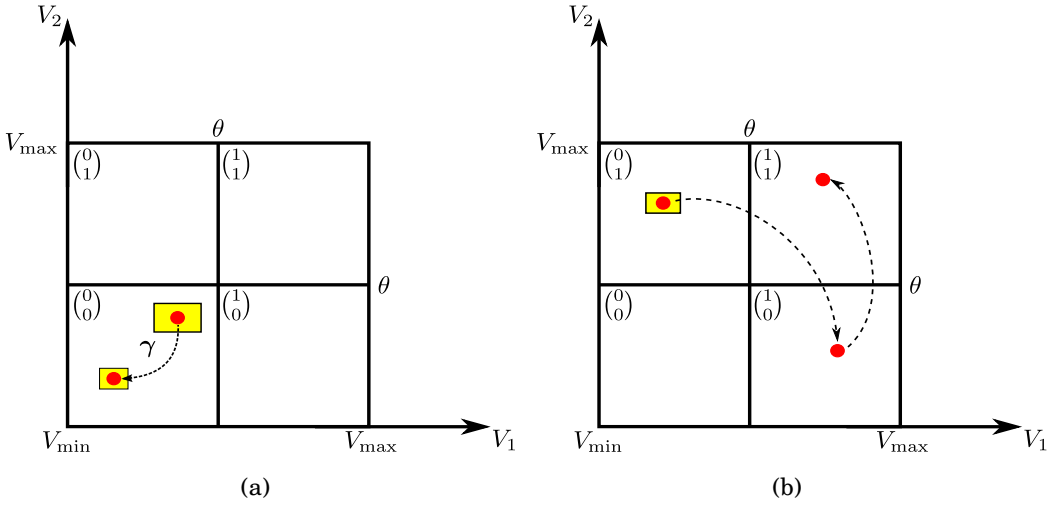


Figure 1.6: The effects on the dynamics when the trajectories of $\mathbf{V}$ has been perturbed. The non-critical case.

First, we concentrate in the partition $\binom{0}{0}$ of the figure 1.6(a). Note that even if the real trajectory shown in figure 1.5(a) has been perturbed, the contraction effect is similar in both cases.

Now, in figure 1.6(b) we can observe that the perturbed ball has evoked a spike in partition $\binom{0}{1}$, which corresponds to neuron 2. However the ϵ-perturbation is not propagated throughout the system due that neuron 2 is reseted in $\binom{1}{0}$

2. ***The critical case:*** *when $\mathcal{B}$ intersects $\mathcal{S}$, $\mathcal{B}(\mathbf{V}, \epsilon) \cap \mathcal{S} \neq 0$*

   On the contrary, the crossing of $\mathcal{S}$ by the ϵ-ball induces a strong effect reminiscent on the sensitivity of initial conditions for chaotic dynamics. In other words, when $\mathcal{B}$ and $\mathcal{S}$ are intersected the trajectory of $\mathbf{V}$ change drastically, since the perturbed trajectory induces a new spiking state.

   Two important remarks of this:

   - The main difference with chaos is that the present effect occurs only when the ball crosses the singularity. (Otherwise the ball is contracted).
   - The singularity $\mathcal{S}$ is the only source of complexity of the discrete-time spiking neuron model, and its existence is due to the strict threshold in the definition of neuron firing.

The critical case is schematized in figure 1.7 where we can observe a drastic effect on the dynamics when the perturbed ball $\mathcal{B}(\mathbf{V}, \epsilon)$ ( ) intersects the singularity set. More specific, a perturbed ball localized very close to the threshold can to induce a firing in the neuron.

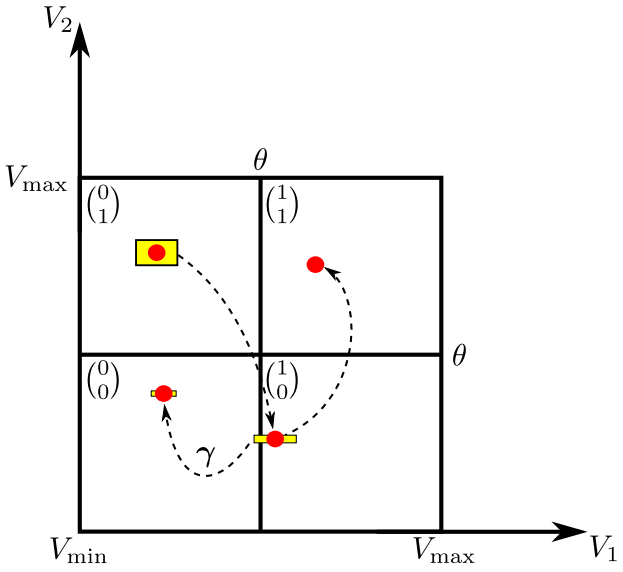


Figure 1.7: The effects on the dynamics when a perturbed ball intersects the singularity set. The critical case.

Further these cases have inspired the chapter 4 of this manuscript, where we show how limited precision on hardware implementations induce perturbations on the dynamics. Thus, we demonstrate numerically this phenomena.

1.2.5 Asymptotic dynamics in the discrete-time spiking neuron model

The mathematical study presented in Cessac (2008) has demonstrated that gIF-type models evidence three main regimes in its asymptotic dynamics for a finite size. These regimes are: neural death, periodic and chaotic. (see figure 1.8)

- **Neural Death.** correspond to a regime where neurons stop to fire due mainly to a weak activity in their synapses or a weak external stimulus. This definition assumes that $I_i^{\text{ext}} < (1-\gamma)\theta$ and consider the set $\mathcal{M}_0 = \{\mathbf{V} | V_i < \theta, \forall i\}$.
- **Periodic.** correspond to trivial activities, where neurons present repetitive patterns and occurs in the domain $\mathcal{M}_1 = \{\mathbf{V} | V_i \geq \theta, \forall i\}$ when σ is large enough.
- **"Chaotic".** this regime also occurs in the domain of $\mathcal{M}_1 = \{\mathbf{V} | V_i \geq \theta, \forall i\}$ and evidences complex dynamic, where the period tends to be very large.

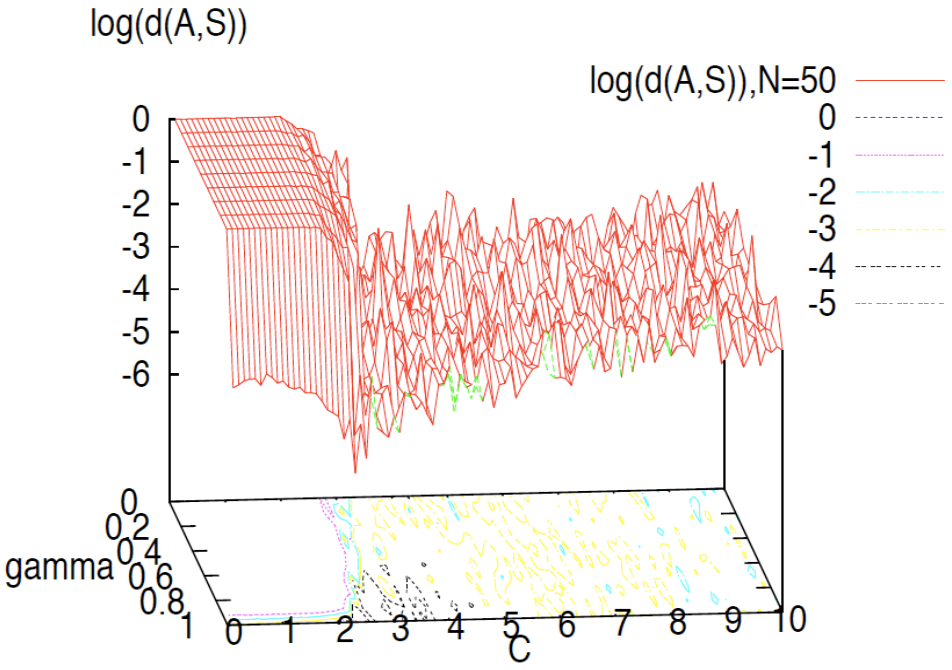


Figure 1.8: Average value of the distance $d(\mathcal{A}, \mathcal{S})$ versus γ, C, for $N = 50$. Illustration from Cessac (2008)

1.2.6 The analog-spiking neuron model

In biologically-inspired neural networks, such as the gIF neuron model, the activation function is usually an abstraction representing the rate of action potentials and stereotyped trajectory called spikes. In this sense the equation 1.4 shows the simplest form of this definition, where the state of the synaptic connections is given by their firing states. In this case a large number of neurons must be computed in order to reach the activation state.

A spike has a discrete-form, which is modeled by a Heaviside step function. This description yields discontinuity, and there is not a mechanism that permits us to smooth the transition between the quiescent and the firing state of a neuron.

In order to elude this discontinuity we can make an extension of 1.4 to an analog-spiking form. The term analog refers to a non-linear activation function (Figure 1.9), however the neural activity continues being a spiking activity. This new gIF-type model is given by the next equation:

$$V_i[k] = \gamma\, V_i[k-1]\, \rho(V_i[k-1]) + \sum_{j=1}^{N} \sum_{d=1}^{D} W_{ijd}\, \sigma(V_j[k-d]) + I_i[k], \qquad (1.13)$$

where:

$$\rho(V_i[k-1]) = Z_i[k-1] \in \{0,1\}$$

and,

$$\sigma(V_j[k-d]) = \frac{1}{1+e^{-(V_j[k-d])}} \text{ (Sigmoid function)} \tag{1.14}$$

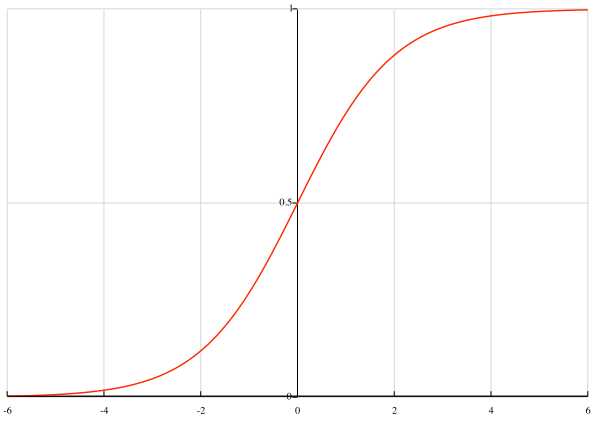


Figure 1.9: A sigmoid function

1.2.7 Biological plausibility of the gIF-type models

In computational neuroscience, the biological plausibility is based on the ability of the neuron models to reproduce real neural dynamics. In this sense the gIF-type models have been studied in Destexhe (1997); Rudolph and Destexhe (2006); Jolivet et al. (2004). The authors have proved that gIF-type models can approximate spike trains of a detailed model with a high degree of accuracy. In addition these models can be extended to the study of learning mechanism such as spike-timing-dependent plasticity Markram et al. (1997); Pfister and Gerstner (2006). Also, the description of the neural activity carried out by these models allow one to reproduce important neural regimes (tonic spiking, phasic spiking, tonic bursting, etc.)

In this chapter we have defined the basis of the gIF-type neuron models, which will be used throughout this manuscript. In the next chapters we show the capacity of the spiking neuron models to perform powerful computations.

Part II

Learning spiking neural networks parameters

CHAPTER 2

REVERSE-ENGINEERING IN SPIKING NEURAL NETWORKS PARAMETERS: EXACT DETERMINISTIC ESTIMATION

Scientists investigate that which already is; Engineers create that which has never been.
–Albert Einstein

OVERVIEW

We consider the deterministic evolution of a time-discretized network with spiking neurons, where synaptic transmission has delays, modeled as a neural network of the generalized integrate-and-fire (gIF) type. The purpose is to study a class of algorithmic methods permitting us to calculate the proper parameters in order to reproduce exactly a given spike train, generated by an hidden (unknown) neural network. This standard problem is known as NP-hard when delays are to be calculated. So far we propose a reformulation, now expressed as a Linear-Programming (LP) problem, which provides an efficient resolution, thus avoiding the NP complexity. Such reformulation makes possible the "reverse-engineering" of a neural network, i.e. to find out, given a set of initial conditions, which parameters (i.e., synaptic weights in this case), allow one to simulate the network spike dynamics.

More precisely we make explicit the fact that the reverse-engineering of a spike train, is a Linear (L) problem if the membrane potentials are observed and a LP problem if only spike times are observed. Numerical robustness is discussed. We also explain how is the use of a generalized IF neuron model instead of a leaky IF model that enable us to derive this algorithm.

Furthermore, we point out how the L or LP adjustment mechanism is

local to each unit and has the same structure as an "Hebbian" rule. A step further, this paradigm is easily generalizable to the design of input-output spike train transformations. Since a numerical viewpoint we are able to "program" a spiking network, i.e. find a set of parameters in order to exactly reproduce the network output, given an input. Numerical verifications and illustrations are provided.

The proposed methods are detailed in two steps: first, detailing the family of estimation problems corresponding to what is called reverse-engineering and discussing the related computational properties; then, making explicit how a general input/output mapping can be "compiled" for a spiking neural network thanks to the previous developments. We also present some numerical experiments.

2.1 Methods: Weights and delayed weights estimation

Let us consider a network of N spiking neurons, whose dynamics is defined by the equation (1.4). Such dynamics is schematized in figure 2.1 as a raster plot (spike train).

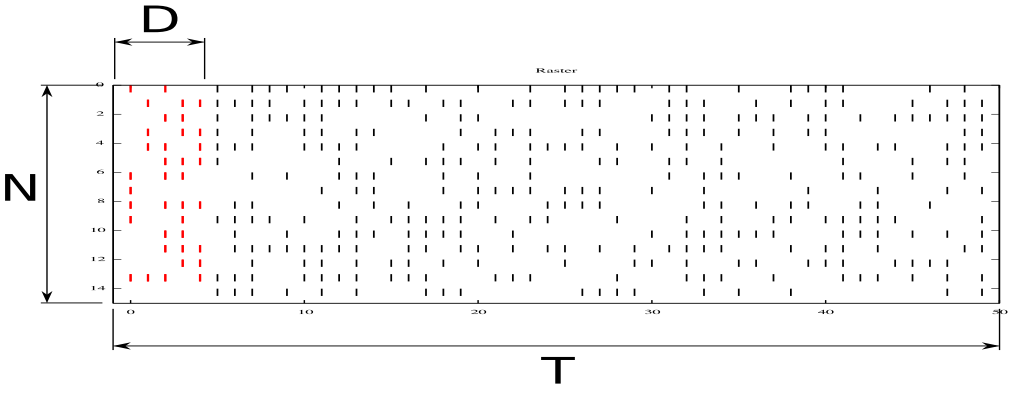


Figure 2.1: Schematic representation of a raster plot with N neurons observed during a time T after a period D (in red), which represents an initial state.

In order to estimate the dynamics of the neural network, we require the knowledge of its initial state. Here, due to the particular structure of equation (1.4) with a delay D, the initial state is defined by a trajectory $V_i[k], k \in \{0, D\{$. The notation $k \in \{0, D\{$ stands for $0 \leq k < D$.

In equation (1.4) if neuron i has fired at least once, the dependence in the initial condition is removed thanks to the reset mechanism. This means that

its state does not depend on $V_i[0]$ anymore. We further assume $V_i[0] = 0$, for the sake of simplicity.

The dynamics is parametrized by the weights W_{ijd} thus $N \times N \times D$ values. Here it is assumed that the γ_i are known and constant, while I_{ik} are also known, as discussed below.

When the potential and/or spikes are observed during a period T, $N \times T$ numerical/binary values are measured.

With the assumption that $V_i[0] = 0$, (1.4) reads:

$$V_i[k] = \sum_{j=1}^{N} \sum_{d=1}^{D} W_{ijd} \sum_{\tau=0}^{\tau_{ik}} \gamma^{\tau} \, Z_j[k - \tau - d] + I_{ik\tau} \tag{2.1}$$

writing $I_{ik\tau} = \sum_{\tau=0}^{\tau_{ik}} \gamma^{\tau} \, I_i[k - \tau]$, where $I_i[k - \tau]$ is given by (1.6), (last time before k when neuron has fired with: $\tau_{ik} = k - \arg\min_{l>0}\{Z_i[l - 1] = 1\}$) the derivation of this last form being easily obtained by induction from (1.4). Here τ_{ik} is the delay from the last spiking time, i.e., the last membrane potential reset. If no spike, we simply set $\tau_{ik} = k$.

Let us now discuss how to retrieve the model parameters from the observation of the network activity. We propose different solutions depending on the paradigm assumptions.

2.1.1 Retrieving weights and delayed weights from the observation of spikes and membrane potential

Let us assume that we can observe both the spiking activity $Z_i[k]$ and the membrane potential $V_i[k]$. Here, (2.1) reads in matrix form:

$$\mathbf{C}_i \, \mathbf{w}_i = \mathbf{d}_i \tag{2.2}$$

with:

$$\begin{array}{rcll} \mathbf{C}_i & = & \begin{pmatrix} \dots & \dots & \dots \\ \dots & \sum_{\tau=0}^{\tau_{ik}} \gamma^{\tau} Z_j[k - \tau - d] & \dots \\ \dots & \dots & \dots \end{pmatrix} & \in \; R^{T-D \times N\,D}, \\ \mathbf{d}_i & = & (\dots \quad V_i[k] - I_{ik\tau} \quad \dots)^t & \in \; R^{T-D}, \\ \mathbf{w}_i & = & (\dots \quad W_{ijd} \quad \dots)^t & \in \; R^{N\,D}. \end{array}$$

writing $\mathbf{u}^t$ the transpose of $\mathbf{u}$.

Here, $\mathbf{C}_i$ is defined by the neuron spike inputs, $\mathbf{d}_i$ is defined by the neuron membrane potential outputs and membrane currents, and the network parameters by the weights vector $\mathbf{w}_i$.

More precisely, $\mathbf{C}_i$ is a rectangular matrix with:

- $N\,D$ columns, corresponding to product with the $N\,D$ unknowns W_{ijd}, for $j \in \{1, N\}$ and $d \in \{1, D\}$,

- $T - D$ rows, corresponding to the $T - D$ measures $V_i[k]$, for $k \in \{D, T\}$, and,

- $T - D \times N\,D$ coefficients corresponding to the raster, i.e. the spikes $Z_j[k]$.

The weights are thus directly *defined by a set of linear equalities for each neuron*. Let us call this a Linear (L) problem.

The equation defined in (2.2), concerns only the weights of one neuron of index i. It is thus a weight estimation local to a neuron, and not global to the network. Furthermore, the weight estimation is given by the observation of the input $Z_i[k]$ and output $V_i[k]$. These two characteristics correspond to usual Hebbian-like learning rules architecture. See Gerstner and Kistler (2002b) for a discussion.

Given a general raster (i.e., assuming $\mathbf{C}_i$ is of full rank $\min(T - D, N\,D)$):

- This linear system of equations has always solutions, in the general case, if:

$$N > \frac{T-D}{D} = O\left(\frac{T}{D}\right) \Leftrightarrow D > \frac{T}{N+1} = O\left(\frac{T}{N}\right) \Leftrightarrow D\,(N{+}1) > T. \quad (2.3)$$

  This requires enough non-redundant neurons N or weight profile delays D, with respect to the observation time T. In this case, given any membrane potential and spikes values, *there are always weights able to map the spikes input onto the desired potential output*.

- On the other hand, if $N\,D \leq T - D$, then the system has no solution in the general case. This is due to the fact that we have a system with more equations than unknowns, thus with no solution in the general case. However, there is obviously a solution if the potentials and spikes have been generated by a neural network model of the form of (1.4).

If $\mathbf{C}_i$ is not of full rank, this may correspond to several cases, e.g.:

- Redundant spike pattern: some neurons do not provide linearly independent spike trains.

- Redundant or trivial spike train: for instance with a lot of bursts (with many $Z_j[k] = 1$) or a very sparse train (with many $Z_j[k] = 0$). Or periodic spike trains.

Regarding the observation duration T, it has been demonstrated in Cessac (2008); Cessac and Viéville (2008) that the dynamic of an integrate and

fire neural network is generically[1] periodic. This however depends on parameters such as external current or synaptic weights, while periods can be larger than any accessible computational time.

In any case, several choices of weights $\mathbf{w}_i$ (in the general case a $D\,(N+1)-T$ dimensional affine space) may lead to the same membrane potential and spikes. The problem of retrieving weights from the observation of spikes and membrane potential may thus have many solutions.

The particular case where $D=1$ i.e. where there is no delayed weights but a simple weight scalar value to define a connection strengths is included in this framework.

2.1.2 Retrieving weights and delayed weights from the observation of spikes

Let us now assume that we can observe the spiking activity $Z_i[k]$ only (and not the membrane potentials) which corresponds to the usual assumption, when observing a spiking neural network.

In this case, the value of $V_i[k]$ is not known, whereas only its position with respect to the firing threshold is provided:

$$Z_i[k]=0 \Leftrightarrow V_i[k]<1 \text{ and } Z_i[k]=1 \Leftrightarrow V_i[k]\geq 1,$$

which is equivalent to write the condition:

$$e_{ik}=(2\,Z_i[k]-1)\,(V_i[k]-1)\geq 0.$$

If the previous condition is verified for all time index k and all neuron index i, then the spiking activity of the network exactly corresponds to the desired firing pattern.

Expanding (2.1), with the previous condition allows us to write, in matrix form:

$$\mathbf{e}_i=\mathbf{A}_i\,\mathbf{w}_i+\mathbf{b}_i\geq 0 \tag{2.4}$$

writing:

$$\begin{array}{rcll}
\mathbf{A}_i & = & \begin{pmatrix} \dots & \dots & \dots \\ \dots & (2\,Z_i[k]-1)\sum_{\tau=0}^{\tau_{jk}}\gamma^\tau Z_j[k-\tau-d] & \dots \\ \dots & \dots & \dots \end{pmatrix} & \in\ R^{T-D\times N\,D},\\
\mathbf{b}_i & = & (\dots \quad (2\,Z_i[k]-1)\,(I_{ik\tau}-1) \quad \dots)^t & \in\ R^{T-D},\\
\mathbf{w}_i & = & (\dots \quad W_{ijd} \quad \dots)^t & \in\ R^{N\,D},\\
\mathbf{e}_i & = & (\dots \quad (2\,Z_i[k]-1)\,(V_i[k]-1) \quad \dots)^t & \in\ R^{T-D},
\end{array}$$

[1] Considering a basic leaky integrate and fire neuron network the result is true except for a negligible set of parameters. Considering an integrate and fire neuron model with conductance synapses the result is true, providing synaptic responses have a finite memory.

thus $\mathbf{A}_i = \mathbf{D}_i\,\mathbf{C}_i$ where $\mathbf{D}_i$ is the non-singular $R^{T-D\times T-D}$ diagonal matrix with $\mathbf{D}_i^{kk} = 2\,Z_i[k] - 1 \in \{-1, 1\}$.

The weights are now thus directly *defined by a set of linear inequalities for each neuron*. This is therefore a Linear Programming (LP) problem. See Darst (1990) for an introduction and Bixby (1992) for the detailed method used here to implement the LP problem.

Furthermore, the same discussion about the dimension of the set of solutions applies to this new paradigm except that we now have to consider a *simplex* of solution, instead of a simple *affine sub-space*.

A step further, $0 \leq e_{ik}$ is the "membrane potential distance to the threshold". Constraining the e_{ik} is equivalent to constraining the membrane potential value $V_i[k]$.

It has been shown in Cessac (2008) how:

$$|\mathbf{e}|_\infty = \min_i \inf_{k\geq 0} e_{ik} \tag{2.5}$$

can be interpreted as a "edge of chaos" distance, the smallest $|\mathbf{e}|$ the higher the dynamics complexity, and the orbits periods.

On the other hand, the higher e_{ik}, the more robust the estimation. If e_{ik} is high, sub-threshold and sup-threshold values are clearly distinct. This means that numerical errors are not going to generate spurious spikes or cancel expected spikes.

Furthermore, the higher $|\mathbf{e}|_\infty$ the smaller the orbits period Cessac (2008). As a consequence, the generated network is expected to have rather minimal orbit periods.

In the sequel in order to be able to use an efficient numerical implementation, we are going to consider a weaker but more robust norm, than $|\mathbf{e}|_\infty$:

$$|\mathbf{e}_i|_1 = \sum_k e_{ik} \tag{2.6}$$

We are thus going to maximize, for each neuron, the sum, thus, up to a scale factor, the average value of e_{ik}.

Let us now derive a bound for e_{ik}. Since $0 \leq V_i[k] < 1$ for sub-threshold values and reset as soon as $V_i[k] > 1$, it is easily bounded by:

$$V_i^{min} = \sum_{jd, W_{ijd}<0} W_{ijd} \leq V_i[k] \leq V_i^{max} = \sum_{jd, W_{ijd}>0} W_{ijd}$$

and we must have at least $V_i^{max} > 1$ in order for a spike to be fired while $V_i^{min} \leq 0$ by construction. These bounds are attained in the high-activity mode when either all excitatory or all inhibitory neurons fire. From this derivation, $e^{max} > 0$ and we easily obtain:

$$e^{max} = \max_i (1 - V_i^{min}, V_i^{max} - 1)$$

$$0 < e_{ik} \leq e^{max}$$

thus an explicit bound for e_{ik}.

Collecting all elements of the previous discussion, the present estimation problem reads:

$$\max_{\mathbf{e}_i, \mathbf{w}_i} \sum_k e_k, \text{ with, } 0 < e_{ik} \leq e^{max}, \text{ and, } \mathbf{e}_i = \mathbf{A}_i \, \mathbf{w}_i + \mathbf{b}_i \tag{2.7}$$

which is a standard bounded linear-programming problem. Note: this method amounts to approximating an arbitrary spike train with a minimal period.

The key point is that a LP problem can be solved in polynomial time, thus is not a NP-complete problem, subject to the curse of combinatorial complexity. In practice, this LP problem can be solved using one of the several LP solution methods proposed in the literature (i.e., Simplex Method, which is, in principle, NP-complete in the worst case, but in practice, as fast as, when not faster, than polynomial methods).

2.1.3 Retrieving signed and delayed weights from the observation of spikes

In order to illustrate how the present method is easy to adapt to miscellaneous paradigms, let us now consider the fact that the weights emitted by each neuron have a fixed sign, either positive for excitatory neurons, or negative for inhibitory neurons. This additional constraint, known as the "Dale principle" Strata and R.Harvey (1999), is usually introduced to take into account the fact that synaptic weights signs are fixed by the excitatory or inhibitory property of the presynaptic neuron.

Although we do not focus on the biology here, it is interesting to notice that this additional constraint is obvious to introduce in the present framework, writing:

$$W_{ijd} = S_{ijd} \, W^{\bullet}_{ijd}, \text{ with } S_{ijd} \in \{-1, 1\}, \text{ and } W^{\bullet}_{ijd} \geq 0$$

thus separating the weight sign S_{ijd} which is a-priory given and the weight value $W^{\bullet}_{ijd}$ which now always positive.

Then, writing:

$$\mathbf{A}^{\bullet} ijkd = \mathbf{A}_{ijkd} \, S_{ijd}$$

the previous estimation problem becomes:

$$\max_{\mathbf{e}_i, \mathbf{w}^{\bullet}_i} \sum_k e_k, \text{ with, } 0 < e_{ik} \leq e^{max}, 0 \leq W^{\bullet}_{ijd} \leq 1, \text{ and, } \mathbf{e}_i = \mathbf{A}^{\bullet}_i \, \mathbf{w}^{\bullet}_i + \mathbf{b}_i \tag{2.8}$$

which is still a similar standard linear-programming problem.

2.1.4 Retrieving delayed weights and external currents from the observation of spikes

In the previous derivations, we have considered the membrane currents I_{ik} as inputs, i.e. they are known in the estimation. Let us briefly discuss the case where they are to be estimated too.

For adjustable non-stationary current I_{ik}, the estimation problem becomes trivial. An obvious solution is $W_{ijd} = 0, I_{ik} = 1 + a\,(Z_i[k] - 1/2)$ for any $a > 0$, since each current value can directly drive the occurrence or inhibition of a spike, without any need of the network dynamics.

Too many degrees of freedom make the problem uninteresting: adjusting the non-stationary currents leads to a trivial problem.

To a smaller extends, considering adjustable stationary currents I_i also "eases" the estimation problem, providing more adjustment variables. It is obvious to estimate not only weights, but also the external currents, since the reader can easily notice that yet another linear-programming problem can be derived.

This is the reason why we do not further address the problem here, and prefer to explore in details a more constrained estimation problem.

2.1.5 Considering non-constant leak when retrieving parametrized delayed weights

For the sake of simplicity and because this corresponds to numerical observations, we have assumed here that the neural leak γ is constant. The proposed method still works if the leak varies with the neuron and with time i.e. is of the form γ_{it} (Equation 1.5), since this is simply yet another input to the problem. The only difference is that, in (2.1) and the related equations, the term γ^τ is to be replaced by products of γ_{it}.

However, if γ is a function of the neural dynamics, $\gamma \equiv \gamma(\omega^t_{-\infty})$, thus of W_{ijd}, where $\{\omega^t_{-\infty}\}$ is the list of firing times of the neuron up to time t, the previous method must be embedded in a non linear estimation loop. Since we know from Cessac and Viéville (2008) that this dependency is numerically negligible in this context, we can propose the following loop:

1. Fix at step $t = 0$, $\gamma^0_{it} \equiv \gamma(\omega^0_{-D})$, to initial values.
2. k- Estimate the weights W_{ijd}, given leaks γ^k_{it} at $k = 0, 1, ...$
3. k- Re-simulate the dynamics, given the weights and to obtain corrected values $\tilde{\gamma}^k_{it}$.
4. k- Smoothly modify $\gamma^{k+1}_{it} = (1 - \upsilon)\,\gamma^{k+1}_{it} + \upsilon\,\tilde{\gamma}^k_{it}$

5. $k+1$ Repeat step 2, k- for $k+1$, the convergence of this non-linear relaxation method being guarantied for sufficiently small υ. See Viéville et al. (2001) for an extended discussion about this class of methods.

This shows that considering models with leaks depending on the dynamics itself is no more a LP-problem, but an iterative solving of LP-problems.

2.1.6 Retrieving parametrized delayed weights from the observation of spikes

In order to further show the interest of the proposed method, let us now consider that the *profile* of the weights is fixed, i.e. that

$$W_{ijd} = W^{\circ}_{ij}\,\alpha_\tau(d) \text{ with, e.g., } \alpha(d) = \tfrac{d}{\tau}\,e^{-\frac{d}{\tau}}$$

thus the weights is now only parametrized by a magnitude W°_{ij}, while the temporal profile is known.

Here $\alpha_\tau(d)$ is a predefined synaptic profile, while τ is fixed by biology (e.g., $\tau = 2\,ms$ for excitatory connections and $\tau = 10ms$ for inhibitory ones). Let us note that the adjustment of τ would have been a much more complex problem, as discussed previously in the non-parametric case.

This new estimation is defined by:

$$\mathbf{e}_i = \mathbf{A}^{\circ}_i\,\mathbf{w}^{\circ}_i + \mathbf{b}_i > 0 \tag{2.9}$$

writing:

$$\begin{array}{rcll} \mathbf{A}^{\circ}_i & = & \begin{pmatrix} \dots & \dots & \dots \\ \dots & (2\,Z_i[k]-1)\,\sum_d \sum_{\tau=0}^{\tau_{jk}} \gamma^\tau Z_j[k-\tau-d]\,\alpha(d) & \dots \\ \dots & \dots & \dots \end{pmatrix} & \in \; R^{T-D\times N} \\ \mathbf{w}^{\circ}_i & = & (\dots \quad W_{ij} \quad \dots)^t & \in \; R^N \end{array}$$

thus a variant of the previously discussed mechanisms.

This illustrates the nice versatility of the method. Several other variants or combinations could be discussed (e.g. parametrized delayed weights from the observation of spikes and potential, ..), but they finally leads to the same estimations.

2.1.7 About retrieving delays from the observation of spikes

In the previous derivations, we have considered delayed weights, i.e. a quantitative weight value W_{ijd} at each delay $d \in \{1, D\}$.

Another point of view is to consider a network with adjustable synaptic delays. Such estimation problem may, e.g., correspond to the "simpler" model:

$$V_i[k] = \gamma_i\, V_i[k-1]\,(1 - Z_i[k-1]) + \sum_{j=1}^{N} W_{ij}\, Z_j[k - d_{ij}] + I_{ik},$$

where now the weights W_{ij} and delays d_{ij} are to estimated.

As pointed out previously, the non-learnability of spiking neurons is known Šíma and Sgall (2005), i.e. the previous estimation is proved to be NP-complete. We have carefully checked in Šíma and Sgall (2005) that the result still apply to the present setup. This means that in order to "learn" the proper parameters we have to "try all possible combinations of delays". This is intuitively due to the fact that each delay has no "smooth" effect on the dynamics but may change the whole dynamics in a unpredictable way.

We see here that the estimation problem of delays d_{ij} seems not compatible with usable algorithms, as reviewed in the introduction.

We propose to elude this NP-complete problem by considering *another* estimation problem. Here we do not estimate *one* delay (for each synapse) but consider connection weights at several delay and then estimate a weighted pondering of their relative contribution. This means that we consider a *weak* delay estimation problem.

Obviously, the case where there is a weight W_{ij} with a corresponding delay $d_{ij} \in \{0, D\}$ is a particular case of considering several delayed weights W_{ijd} (corresponding to have all equal weights to zero except at d_{ij}, i.e., $W_{ijd} =$ if $d = d_{ij}$ then W_{ij} else 0).

We thus do not restrain the neural network model by changing the position of the problem, but enlarge it. In fact, the present estimation provides a smooth approximation of the previous NP-complete problem.

We can easily conjecture that the same restriction also apply of the case where the observation of spikes and membrane potential is considered.

We also have to notice, that the same restriction apply not only to simulation but, as far as this model is biologically plausible, also true at the biological level. It is thus an issue to wonder if, in biological neural network, delays are really estimated during learning processes, or if a weaker form of weight adaptation, as discussed in this work, is considered.

2.2 Methods: Exact spike train simulation

2.2.1 Introducing hidden units to reproduce any finite raster

Up to now, we have assumed that a raster $\bar{Z}_i[k], i \in \{1, N\}, k \in \{1, T\}$ is to be generated by a network whose dynamics is defined by (1.4), with initial

conditions $\bar{Z}_j[k], j \in \{1, N\}, k \in \{1, D\}$ and $V_j[0] = 0$. In the case where a solution exists, we have discussed how to compute it.

We have seen that a solution always exists, *in the general case*, if the observation period is small enough, i.e., $T < O(N\,D)$. Let us now consider the case where $T \gg O(N\,D)$.

In this case, there is, in general, no solution. This is especially the case when the raster has not been generated by a network given by (1.4), for example in the case when the raster is random.

The key idea, borrowed from the reservoir computing paradigm, is to add a reservoir of "hidden neurons", i.e., to consider not N but $N+S$ neurons. The set of N "output" neurons is going to reproduce the expected raster $\bar{Z}_i[k]$ and the set of S "hidden" neurons to increase the number of degree of freedom in order to obtain $T < O((N + S)\,D)$, thus being able to apply the previous algorithms to estimate the optimal delayed weights. Clearly, in the worst case, we have to add $S = O(T/D)$ hidden neurons. This is illustrated in Fig. 2.2.

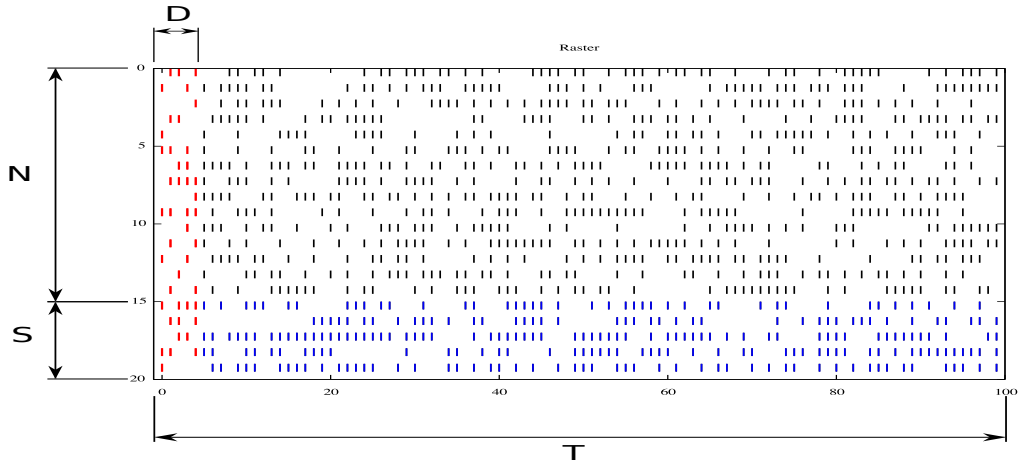


Figure 2.2: Schematic representation of a raster of N output neuron observed during a time interval T after an initial conditions interval D, with an add-on of S hidden neurons, in order increase the number of degree of freedom of the estimation problem. See text for further details.

In order to make this idea clear, let us consider a trivial example.

2.2.2 Sparse trivial reservoir

Let us consider, as illustrated in Fig. 2.3, $S = T/D + 1$ hidden neurons of index $i' \in \{0, S\{$ each neuron firing once at $t_{i'} = i'\,D$, except the last once always firing (in order to maintain a spiking activity), thus:

$$Z_{i'}[k] = \delta(i'\,D - k), 0 \leq i' < S, Z_S[k] = 1$$

Let us choose:

$$\begin{array}{rcl} W_{SS1} & > & 1 \\ W_{i'S1} & = & \frac{1-\gamma}{1-\gamma^{t_{i'}-1/2}} \\ W_{i'i'1} & < & -\frac{\gamma^{2\,t_{i'}}-\gamma^T}{\gamma^T\,(1-\gamma^{t_{i'}})} < 0 \\ W_{i'j'd} & = & 0 \text{ otherwise} \end{array}$$

with initial conditions $Z_{i'}[k] = 0, i' \in \{0, S\{$ and $Z_S[k] = 1, k \in \{1, D\}$, while $I_{i'k} = 0$.

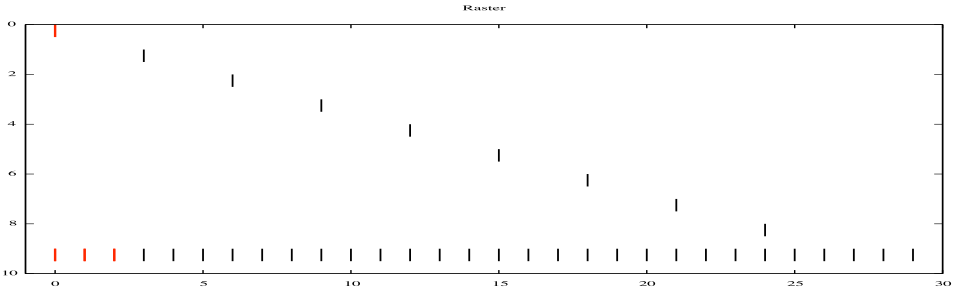


Figure 2.3: Schematic representation of a sparse trivial set of hidden neurons, allowing to generate any raster of length T.

A straight-forward derivation over equation (1.4) allows us to verify that this choice generates the specified $Z_{i'}[k]$. In words, as the reader can easily verify, it appears that:

- the neuron of index S is always firing since (through W_{SS1}) a positive internal loop maintains its activity;
- the neurons of index $i' \in \{0, S\{$, whose equation writes:

  $$V_{i'}[k] = \gamma\,V_{i'}[k-1]\,(1 - Z_{i'}[k-1]) + W_{i'S1} + W_{i'i'1}\,Z_{i'}[k-1]$$

  is firing at $t_{i'}$ integrating the constant input $W_{i'S1}$;
- the neurons of index $i' \in \{0, S\{$, after firing is inhibited (through $W_{i'i'1}$) by a negative internal loop, thus reset at value negative enough not to fire anymore before T. We thus generate $Z_{i'}[k]$ as expected.

Alternatively, the use of the firing neuron of index S can be avoided by introducing a constant current $I_{i'k} = W_{i'S1}$.

However, without the firing neuron of index S or some input current, the sparse trivial raster *can not* be generated, although $T < O(N\,D)$. This

comes from the fact that the activity is too sparse to be self-maintained. This illustrates that when stating that "a solution exists, *in the general case*, if the observation period is small enough, i.e., $T < O(N\,D)$", a set of singular cases, such as this one, was to be excluded.

The hidden neurons reservoir raster being generated, it is straightforward to generate the output neuron raster, considering:

- no recurrent connection between the N output neurons, i.e., $W_{ijd} = 0, i \in \{1, N\}, j \in \{1, N\}, d \in \{1, D\}$,
- no backward connection from the N output neurons to the S hidden neurons i.e., $W_{i'jd} = 0, i' \in \{0, N\{, j \in \{1, N\}, d \in \{1, D\}$,
- but forward excitatory connections between hidden and output neurons:

$$W_{ij'd} = (1+\epsilon)\,\bar{Z}_i[j'\,D + d] \quad \text{for some small} \quad \epsilon > 0$$

yielding, from (1.4) :

$$\begin{aligned} V_i[k] &= \textstyle\sum_{j'=1}^{n} \sum_{d=1}^{D} W_{ij'd}\, Z_{j'}[k-d] \\ &= \textstyle\sum_{j'=1}^{n} \sum_{d=1}^{D} (1+\epsilon)\,\bar{Z}_i[j'\,D + d]\,\delta(j'\,D - (k-d)) \\ &= (1+\epsilon)\,\bar{Z}_i[k] \end{aligned}$$

setting $\gamma = 0$ for the output neuron and $I_{i'k} = 0$, so that $Z_i[k] = \bar{Z}_i[k]$, i.e., the generated spikes $Z_i[k]$ correspond to the desired $\bar{Z}_i[k]$, as expected. After a time T, the term ϵ induces changes in the dynamics that can not be controlled.

2.2.3 The linear structure of a network raster

The previous construction allows us to state: *given any raster of N neurons and observation time T, there is always a network of size $N + T/D + 1$ with weights delayed up to D, which exactly simulates this raster*. What do we learn from this fact ?

This helps to better understand one aspect of the reservoir computing paradigm: *Although it is not always possible to simulate any raster plot using a "simple" integrate and fire model such as the one defined in (1.4), adding hidden neurons allows to embed the problem in a higher-dimensional space where a solution can be found.*

This results is induced by the fact, made explicit, in the previous section, that learning the network weights is essentially a linear (L or LP) problem. With this interpretation, a neuron spiking sequence is a vector in this linear space, while a network raster is a vector set. Designing a "reservoir" simply means choosing a set of neurons whose spiking activity *spans the space of expected rasters*. We are going to see in the next section that this point of view still holds in our framework when considering network inputs.

This linear-algebra interpretation further explains our "trivial sparse" choice: We have simply chosen a somehow canonical orthonormal basis of the raster linear trajectory. One consequence of this view is that, *given a raster, any other raster which is a linear combination of this raster* can be generated by the same network, by a little variation in the weights. This is due to the fact that a set of neurons defining a given raster corresponds to the set of vectors spanning the linear space of all possible raster generated by this network. Generating another raster corresponds to a simple change of generating vectors in the spanning set. This also allows us to define, for a given raster linear space, a minimal set of generating neurons, i.e. a vector basis. The "redundant" neurons are those which spiking sequence is obtained by feed-forward connections from other neurons.

We must however take care in the fact that the numerical values of the vector are binary values, not real numbers. This is a linear space over a finite field, whereas its scalar product is over the real numbers.

2.2.4 On optimal hidden neuron layer design

In the previous paragraph, we have fixed the hidden neuron spiking activity, choosing a sparse ad-hoc activity. It is clearly not the only one solution, very likely not the best one.

Given N output neurons and S hidden neurons, we may consider the following question: which are the "best" weights $\mathbf{W}$ and the hidden neuron activity $Z_{j'}[k]$ allowing one to reproduce the output raster.

By "best", we mean optimal weights estimation with the smaller number of hidden neurons in order to estimate a given spike dynamics. In that case, instead of having to solve a LP-problem, as specified in (2.7), we have to consider a much more complicated problem now:

- not a linear problem anymore instead we need to define a bi-linear problem that permit us to consider both, the weights estimation and the desired spiking activity.
- not a standard linear programming problem with real values to estimate, but a mixed integer programming problem with both integer values to estimate.

This has a dramatic consequence, since such problem is known as being NP-hard, thus not solvable in practice, as discussed previously for the estimating of delays.

This means that we can not consider this very general question, but must propose heuristics in order to choose or constraint the hidden neuron activity, and then estimate the weights, given the output and hidden neuron's spikes, in order to still consider a LP-problem.

Let us consider one of such heuristic.

2.2.5 A maximal entropy heuristic

Since we now understand that hidden neuron activity must be chosen in order to span as much as possible the expected raster space, and since we have no a-priori information about the kind of raster we want to reproduce, the natural idea is to randomly choose the hidden neuron activity with a maximal randomness.

Although it is used here at a very simple level, this idea is profound and is related to random sampling and sparse approximation of complex signal in noise (see Tropp (2004b,a) for a didactic introduction), leading to greedy algorithms and convex relaxation Tropp (2006); Tropp et al. (2006). Since inspired by these elaborated ideas, the proposed method is simple enough to be described without any reference to such formalisms.

In this context, maximizing the chance to consider a hidden neuron with a spiking activity independent from the others, and which adds new independent potential information, simply corresponds to choose the activity "as random as possible". This corresponds to a so called Bernouilli process, i.e., simply to randomly choose each spike state independently with equiprobability.

Since we want to simulate the expected raster with a minimal number of hidden neuron, we may consider the following algorithmic scheme:

1. Starts with no hidden but only output neurons.
2. Attempts to solve (2.7) on hidden (if any) and output neurons, in order to obtain weights which allows the reproduction of the expected raster on the output neurons.
3. If the estimation fails, add a new hidden neuron and randomly draw its spiking activity
4. Repeat step 2 and 3 until an exact reproduction of the expected raster is obtained

Clearly, adding more and more random points to the family of generating elements must generate a spanning family after a finite time, since randomly choosing point in an affine space, there is no chance to always stay in a given affine sub-space. This means that we generate a spanning family of neuron after a finite time, with a probability of one. So that the algorithm converges.

What is to be numerically experimented is the fact we likely obtain a somehow minimal set of hidden neurons or not. This is going to be experimented in section 2.4.

2.3 APPLICATION: INPUT/OUTPUT TRANSFER IDENTIFICATION

Let us now describe the main practical application of the algorithm previously developed, which is to "program" a spiking network in order to generate a given spike train or realize a given input/output spike train function. In the present context, this means finding properly the spiking network parameters in order to map an input's set onto an output's set.

Let us to rewrite equation (2.1) as an input/output system:

$$V_i[k] = \underbrace{\sum_{j=1}^{No+S}\sum_{d=1}^{D} W_{ijd}\sum_{\tau=0}^{\tau_{ik}}\gamma^{\tau} Z_j[k-\tau-d]}_{\text{output + hidden}} + \\ + \underbrace{\sum_{l=1}^{Ni}\sum_{d=1}^{D} W'_{ild}\sum_{\tau=0}^{\tau_{ik}}\gamma^{\tau} Z'_l[k-\tau-d]}_{\text{input}} + \sum_{\tau=0}^{\tau_{ik}}\gamma^{\tau} I_i[k-\tau] \qquad (2.10)$$

All variables involved in equation (2.1) have been defined throughout this chapter. Observing the equation, we can analyze it in three parts. The first part corresponds to an output+hidden activity. Here the output describes an spiking activity, which has been estimated from a given function, the hidden layer corresponds to an ensemble of neurons, which has been set to reinforce the parameters estimation. The second part corresponds to input dynamics, which is basically a spike train containing the data set that will be processed by the given function. The third part defines an external stimulus.

The goal in this section is to define a LP problem from equation (2.10), similar to section 2.1.2, that permit us to estimate the parameters (W and W') involved in a input-output transformations. We consider the exact case, further in chapter 3 we study the problem for an approximate solution.

$$\mathbf{A}_i\mathbf{W}_i + \mathbf{B}_i\mathbf{W}'_i + \mathbf{c}_i > 0$$

thus:

$$
\begin{aligned}
\mathbf{A}_i &= \begin{pmatrix} \dots & \dots & \dots \\ \dots & (2Z_i[k]-1)\sum_{\tau=0}^{\tau_{ik}} \gamma^\tau Z_j[k-\tau-d] & \\ \dots & \dots & \dots \end{pmatrix} \in R^{N_S(T-D)\times(N+Nh)D} \\
\mathbf{B}_i &= \begin{pmatrix} \dots & \dots & \dots \\ \dots & (2Z_i[k]-1)\sum_{\tau=0}^{\tau_{ik}} \gamma^\tau Z'_l[k-\tau-d] & \\ \dots & \dots & \dots \end{pmatrix} \in R^{N_S(T-D)\times N_i D} \\
\mathbf{c}_i &= (\dots \quad (2Z_i[k]-1)(\textstyle\sum_{\tau=0}^{\tau_{ik}} \gamma_I^\tau i[k-\tau]-1) \quad \dots)^t \in R^{N_S(T-D)}
\end{aligned}
$$

On the number of hidden neurons

As we describe in 2.2.1 hidden neurons are necessaries in order to perform a robust parameters estimation. In this sense the system has a solution in the general case when $N = Ni + No + S$. The number of hidden neurons S for an input/output system is given by the next inequality:

$$S \geq \frac{T \times N_S}{D} + (N_S - 1) - Ni - No$$

where N_S defines the number of samples (input-output spiking dynamics) that will be used for the estimator to perform a solution in order to learn the transfer function.

What is pointed out here, is the fact that the previous formalism does not only apply to the simulation of a unique, input less, fully connected network, but is applicable to a much wider set of problems.

In order to make this explicit, let us consider the following specification of spiking neural networks with units defined by the recurrent equation (1.4).

- **Connectivity.** We assume a partially connected network with a connection graph $\mathcal{K}$, i.e., some connections weights can be zero.
- **Input current.** We consider that any neurons can be driven by an input current I_{ik}, defining an "analog" input.
- **Input spikes.** We have also to consider that the network can also be driven by external incoming spikes.
- **Output neurons.** We consider that a subset of neurons define an output layer with *readout* state that must be constrained, as was defined in (2.4). Other neurons are hidden neurons.

  As discussed previously, the best heuristics is to randomly generate the hidden neurons required to estimate the spiking activity.
- **Weighted estimation.** We further consider that depending on neuron and time the estimation requirement is not homogeneous, whereas

there are times and neurons for which the importance of potential to threshold distance estimation differs from others. This generalized estimation is obvious to introduce in our formalism, defining:

$$|\mathbf{e}_i|_{1,\Lambda} = \sum_k \Lambda_{ik}\, e_{ik}, \Lambda_{ik} \geq 0 \tag{2.11}$$

for some metric (described in chapter 3) Λ.

We further consider a "supervised learning paradigm" in the following sense. We now consider a family of L input current or spikes vectors:

$$\mathbf{I}^l = (\ldots \quad I^l_{ik} \quad \ldots)^t \in R^{N\times T-D},$$

to be mapped on family of output spike trains:

$$\mathbf{Z}^l = (\ldots \quad l \quad \ldots)^t \in R^{N\times T-D},$$

given initial states:

$$\mathbf{Z}^l_0 = (\ldots \quad l \quad \ldots)^t \in R^{N\times D}, k \in \{0, D\{,$$

for $l \in \{0, L\{$. We would like to find the correct weights $\mathbf{W}$ allowing one to perform this input/output mapping. The estimation problem is in fact strictly equivalent to (2.4), by concatenating the input information (except the initial states). This reads:

$$\begin{array}{rcl} \mathbf{A}_i & = & \begin{pmatrix} \ldots & \ldots & \ldots \\ \ldots & (2\,Z_i[k]^l - 1)\sum_{\tau=\tau_{jk}}^{0} \gamma^\tau Z_j[k-\tau-d]^l & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix} \in R^{L\,(T-D)\times N\,D}, \\ \mathbf{b}_i & = & (\ldots \quad (2\,Z_i[k]^l - 1)\,(I^l_{ik\tau} - 1) \quad \ldots)^t \in R^{L\,(T-D)}. \end{array}$$

This formalism, thus allows us to find an exact input/output mapping, adding hidden neurons in order to have enough degree of freedom to obtain a solution.

Example: IO transfer identification of the "OR" Function

In order to illustrate our method in a input-output system we consider as a starting example, a simple "OR" function. Therefore we have only one spike as output if at least one neuron fire a spike in the precedent time. We have chosen the "OR" function because it has a trivial solution and we can observe the evolution on the weights. The system is trained with N_S inputs and theirs respective outputs, where each output is the "OR" function calculated on each input raster; finally it is tested with a different input (not used in the learning phase), the output of this input is calculated with the weights

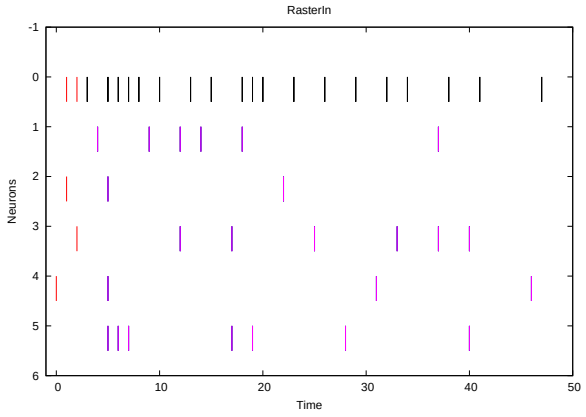


Figure 2.4: A spike train representing input-output dynamics. Red lines define an initial state. Purple lines define a spiking activity in input neurons. Black lines define the output spiking activity, which is defined by an OR function applied in the input activity.

estimated using the Eq. (2.10), here the distance between the estimated output with the weights obtained and the expected output is calculated. Figure 2.4.

2.4 Numerical Results

2.4.1 Retrieving weights from the observation of spikes and membrane potential

In a first experiment, we consider the linear problem defined in (2.2) and use the singular value decomposition (SVD) mechanism Gantmatcher (1977) in order to obtain a solution in the least-square sense. Here the well-established `GSL`[2] library SVD implementation is used.

This permits us to find:

- if more than one solution, the weights of minimal magnitude $|\mathbf{w}_i|^2 = \sum_{jd} W_{ijd}^2$;
- if no exact solution, the solution which minimizes $\sum_k (V_i[k] - \tilde{V}_i[k])^2$ where $\tilde{V}_i[k]$ is the membrane potential predicted by the estimation.

[2] `http://www.gnu.org/software/gsl`

The master and servant paradigm.

We have seen that, if $D\,(N+1) > T$, i.e., if the observation time is small enough for any raster, there exist a solution. Also, there is always a solution wherever the raster is generated by a model of the form of (1.4). We consider the second case here and consider a master/servant paradigm, as follows:

1. In a first step we randomly choose weights and generate a "master" raster.

2. The corresponding output raster is submitted to our estimation method (the "servant"), while the master weights are hidden. The weights are taken from a normal distribution $\mathcal{N}(0, \frac{\sigma^2}{N})$ with 70% excitatory connections and 30% for inhibitory one. The standard deviation $\sigma \in [1, 10]$ has been chosen in order to obtain simple (periodic) and complex dynamics, as discussed in Cessac (2008).

The algorithm defined in (2.3) or in (2.7) has a set of spikes as input for which we are sure that a solution exists. Therefore it can be used and leads to a solution with a raster which must exactly correspond to the master input raster.

Note that this does not mean that the servant is going to generate the raster with the same set of weights, since several solutions likely exist in the general case. Moreover, except for the paradigm (2.3), the master and servant potential $V_i[k]$ are expected to be different, since we attempt to find potentials whose distance to the threshold is maximal, in order to increase the numerical robustness of the method.

This is going to be the validation test of our method. As an illustration we show two results in Fig. 2.5 and Fig. 2.6 for two different dynamics. The former is "chaotic" in the sense that the period is higher than the observation time.

In the non trivial case in Fig. 2.5, it is expected that only one weight's set can generate such a non-trivial raster, since, as discussed before, we are in the "full rank" case, thus with a unique solution. We observe the same weights for both master and servant in this case, as expected. This would not be the case for simpler periodic raster, e.g. in Fig. 2.6, where the weight's estimation by the servant differs from the master's weights, since several solutions are possible.

Retrieving weights from the observation of spikes *and* membrane potential has been introduced here in order to explain and validate the general method in a easy to explain case. Let us now turn to the more interesting cases where only the observation of spikes are available.

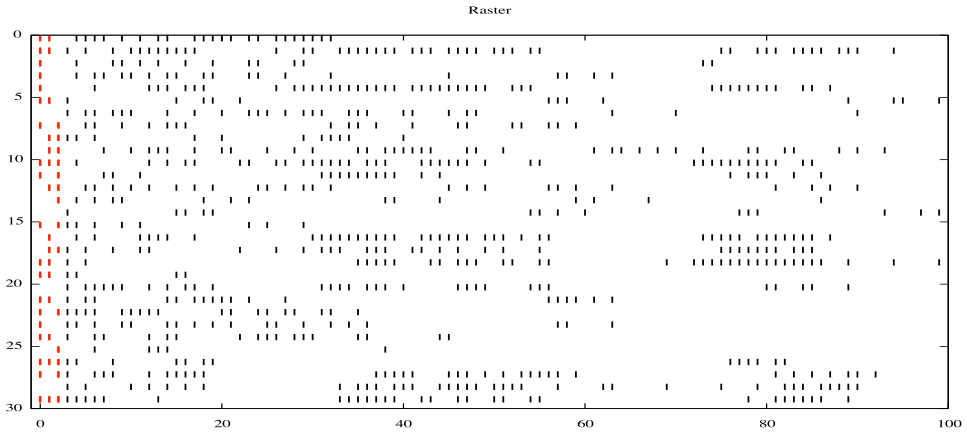


Figure 2.5: A "chaotic" dynamics with 30 neurons fully connected within network, initial conditions $D = 3$ and observation time $T = 100$, using both excitatory (70%) and inhibitory (30%) connections, with $\sigma = 5$ (weight standard-deviation). After estimation, we have checked that master and servant generate **exactly the same raster plot, thus only show the servant raster, here and in the next figures.**

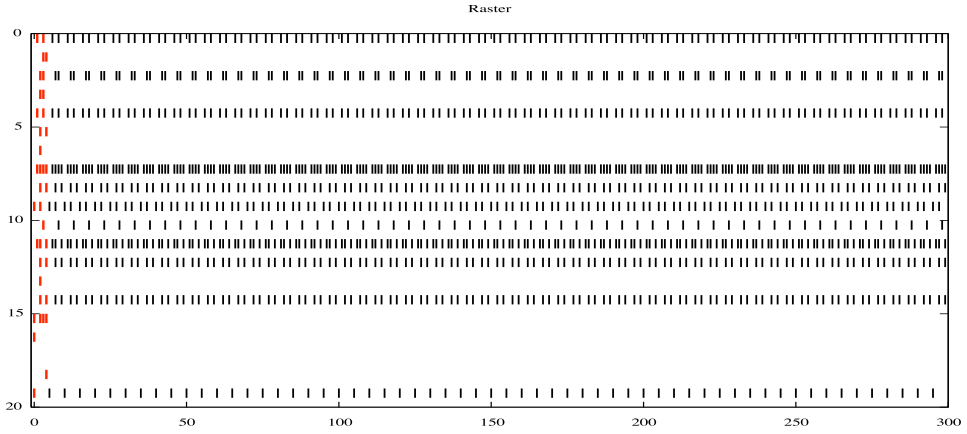


Figure 2.6: A "periodic" (58.2 periods of period 5) dynamics with 20 neurons fully connected within network and observation time $T = 300$, using both excitatory (70%) and inhibitory (30%) connections, with $\sigma = 1$. Again the master and servant rasters are the same.

2.4.2 Retrieving weights from the observation of spikes

In this setup we still use the master / servant paradigm, but now consider the LP problem defined previously. The numerical solutions are derived thanks to the well-established improved simplex method as implemented in `GLPK`[3].

As an illustration we show two results in Fig. 2.7 and Fig. 2.8 for two different dynamics.

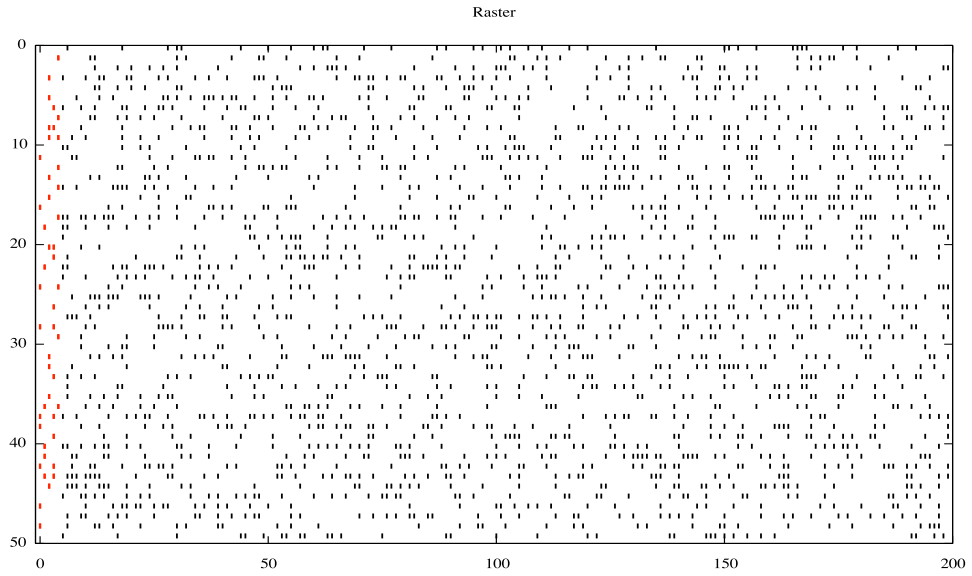


Figure 2.7: Example of rather complex "chaotic" dynamics retrieved by a the LP estimation defined in (2.7) using the master / servant paradigm with 50 neurons fully connected, initial conditions $D = 3$ and observation time $T = 200$, used here to validate the method.

Interesting is the fact that, numerically, the estimated weights correspond to a parsimonious dynamics in the sense that the servant raster period tends to be minimal:

- if the master raster appears periodic, the servant raster is also, with the same period;
- if the master raster appears aperiodic (i.e., "chaotic") during the observation interval, the servant raster is periodic with a period close to the observation time T.

[3] `http://www.gnu.org/software/glpk`

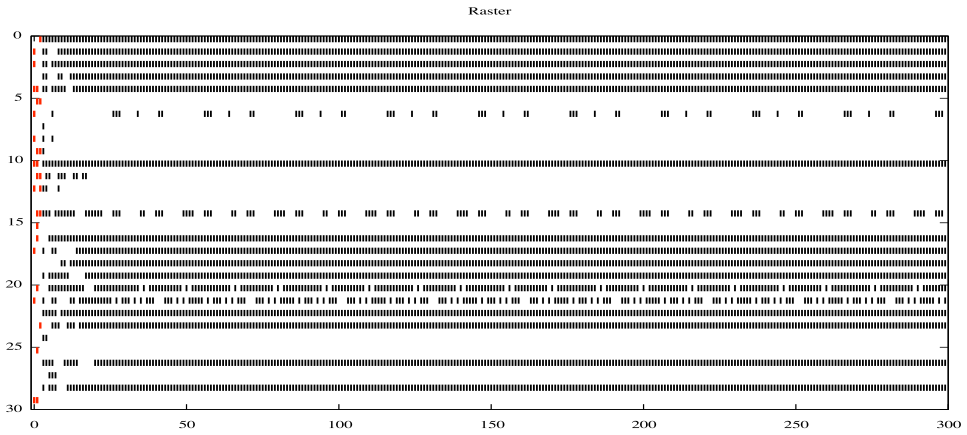


Figure 2.8: Example of periodic dynamics retrieved by a the LP estimation defined in (2.7) using the master / servant paradigm, here a periodic raster of period 30 is observed during 8.3 periods. ($N = 30$, $T = 300$ and $D = 3$) As expected from by the theory, the estimated dynamics remains periodic after the estimation time, thus corresponding to a parsimonious estimation.

2.4.3 Retrieving delayed weights from the observation of spikes

In this next setup we still consider the same master/servant paradigm, for $N = 50$ units, with a leak $\gamma = 0.95$ and an external current $I = 0.3$, but in this case where the master delayed weight profile has the standard form shown in Fig. 2.9.

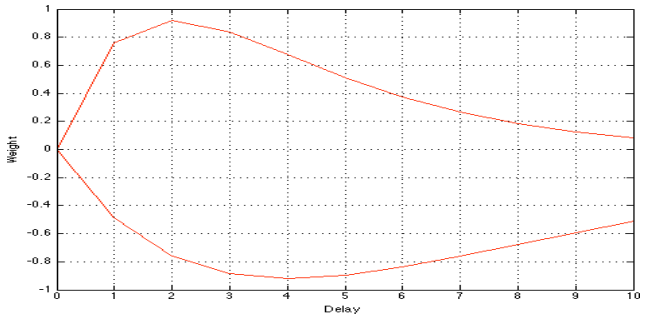


Figure 2.9: Weights distribution (positive and negative) used to generate delayed weights, with $D = 10$.

In the case of periodic dynamics, it is observed that the estimated servant weights distribution is periodic as illustrated in Fig. 2.10. However, as

soon as the dynamics is non trivial, the proposed algorithm uses all delayed weight parameters in order to find an optimal solution, without any correspondence between the master initial weight distribution and the servant estimated weight distribution. This is illustrated in Fig. 2.12, where instead of the standard profiles shown in Fig. 2.9, a "Dirac" profile has been used in the master, while the estimated weights are distributed at all possible delays. In order to complete this illustration a non trivial dynamics is shown in Fig. 2.11.

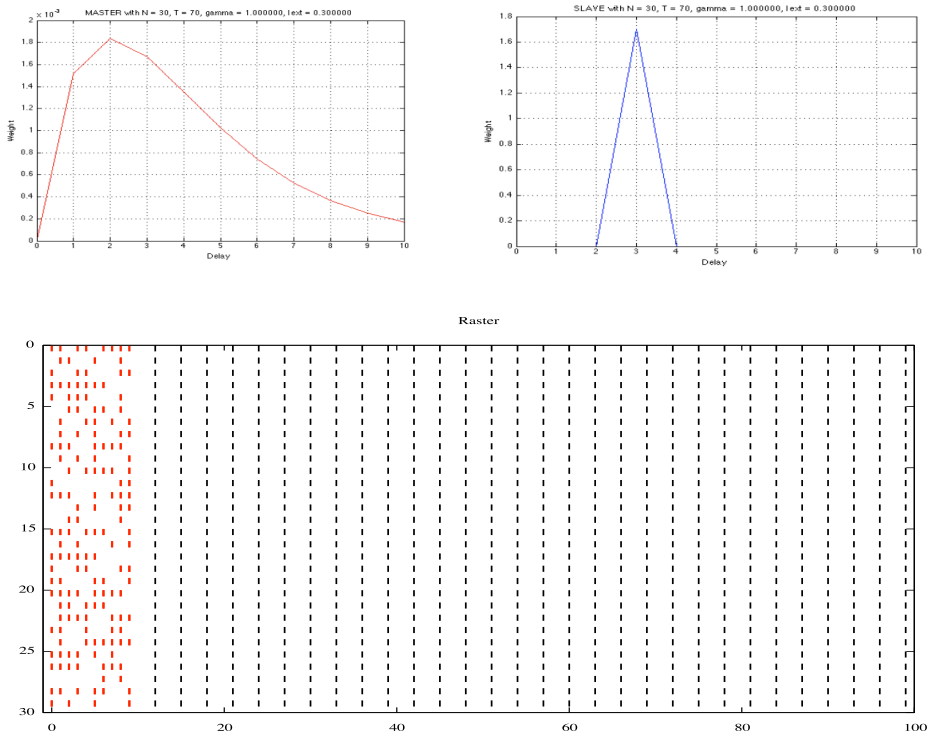


Figure 2.10: An example of periodic dynamics obtained with excitatory weights profiles shown in the top-left view (master weight's profile), with $N = 30$, $\gamma = 0.98$, $D = 10$ $T = 100$. The estimated weights profile (servant weight's profile) is shown in the top-right view. To generate such trivial periodic raster, shown in the bottom view, only weights with a delay equal to the period have not zero values. This corresponds to a minimal edge of the estimation simplex, this parsimonious estimation being a consequence of the chosen algorithm.

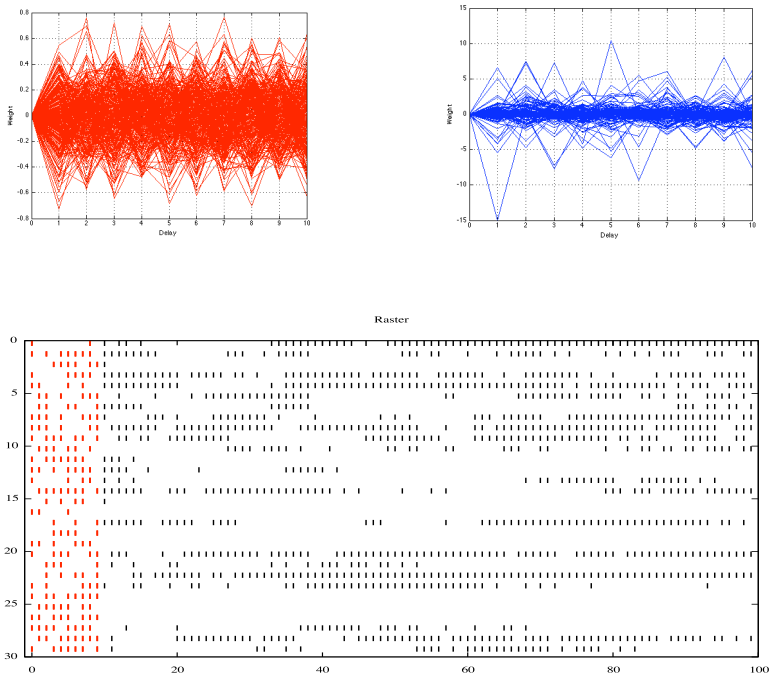


Figure 2.11: An example of non-trivial dynamics,with $N = 30$, $\gamma = 0.98$, $D = 10$ $T = 100$. Profiles corresponding to the master's excitatory profiles are superimposed in the top-left figure, those corresponding to the master's inhibitory profiles are superimposed in the top-left figure. The estimated raster is shown in the bottom view. This clearly shows that, in the absence of additional constraint, the optimal solution corresponds to a wide distribution of weight's profiles.

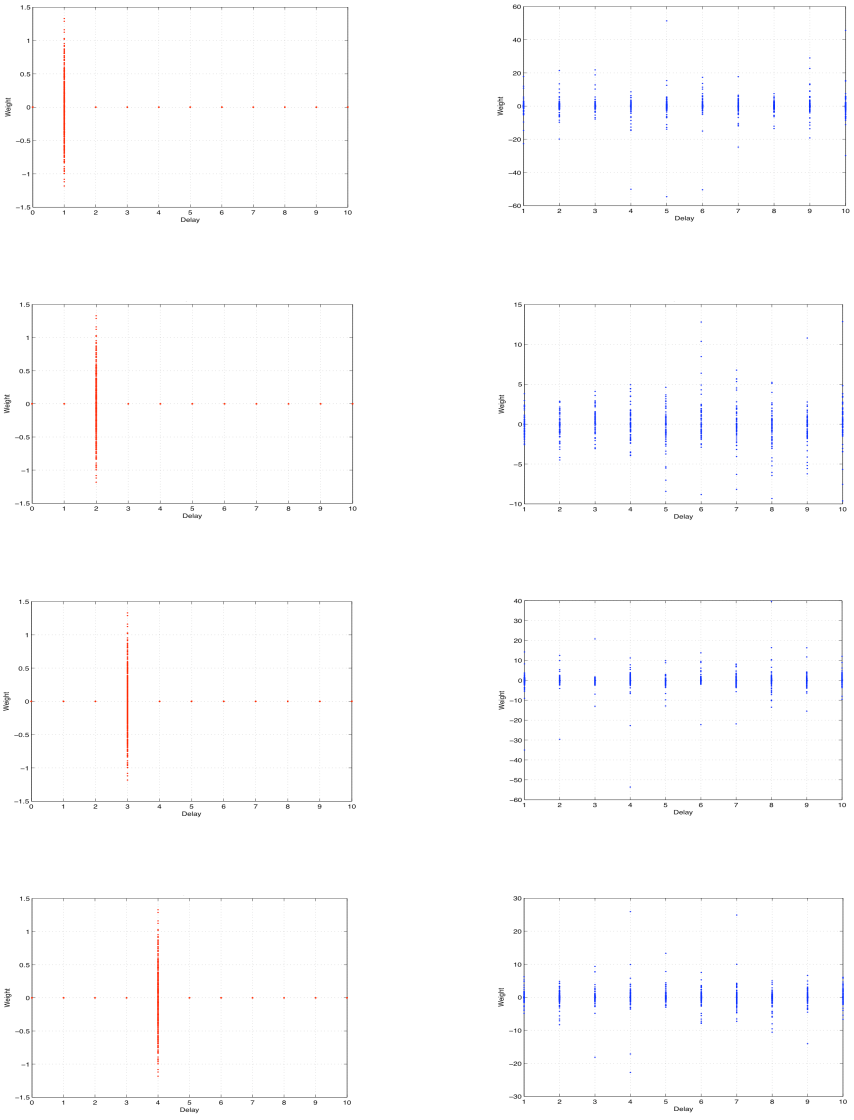


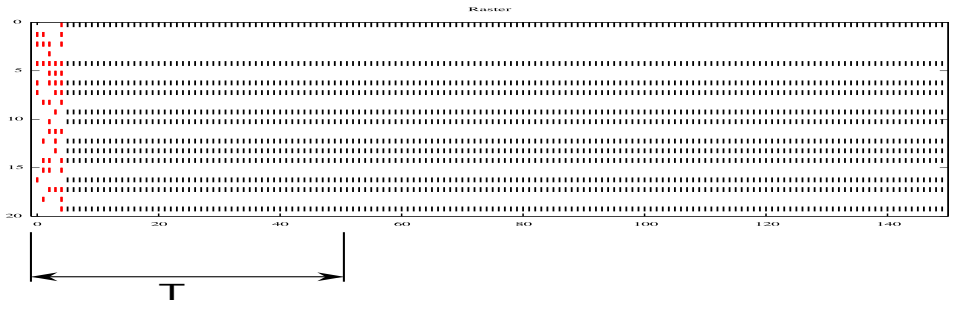
Figure 2.12: In this figure we show that whatever be the weights and delays in the master (left), with $N = 20$, $\gamma = 0.98$, $D = 10$ $T = 100$, the estimator uses all the weights and delays to calculate the raster, in order to obtain a solution.

On the complexity of the generated network

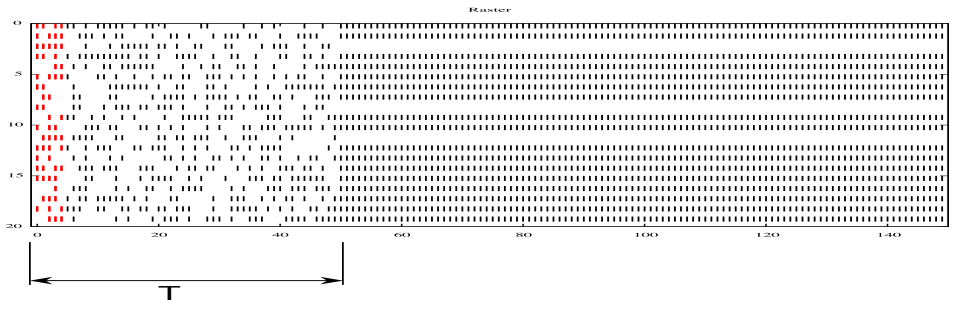
Maximizing (2.5) we obtain, among networks which generate exactly the required master, the "less complex" network, i.e. the one with the smallest period. A very simple way to figure out how complex is the servant network is to observe its generated raster after T, i.e., after the period of time where it matches exactly the required master's raster. They are indeed the same before T.

After T, in the case of delayed weights, we still observe that if the original raster is periodic, the generated raster is still periodic with the same period.

If the original raster is aperiodic, for small N and T, we have observed that the generated master is still periodic, as illustrated in Fig. 2.13. We however, have not observed any further regularity, for instance changes of regime can occur after the T delay, huge period can be observed for relatively small numbers of N and T, etc.



(a)



(b)

Figure 2.13: Two examples of observation of the raster period, on the slave network, observing the raster after the time T where it matches the master raster (shown by an arrow in the figure). (a) With $N = 20$, $\gamma = 0.98$, $\sigma = 5$, $D = 5$, $T = 50$, a periodic regime of periode $P = 1$ is installed after a change in the dynamics. (b) With $N = 20$, $\gamma = 0.98$, $\sigma = 5$, $D = 5$, $T = 50$, a periodic regime of periode $P = 1$ corresponds to the master periodic regime.

2.4.4 Retrieving delayed weights from the observation of spikes, using hidden units

In this last set of numerical experiments we want to verify that the proposed method in section 2.2 is "universal" and allows one to evaluate the number of hidden neurons to be recruited in order to exactly simulate the required raster. If it is true, this means that we have here available a new "reservoir computing" mechanism.

Considering Bernoulli distribution

We start with a completely random input, drawn from a uniform Bernoulli distribution. This corresponds to an input with maximal entropy. Here the master/servant trick is no more used. Thus, the raster to reproduce has no chance to verify the neural network dynamics constraints induced by (1.4), unless we add hidden neurons as proposed in section 2.2.

In Fig. 2.15, we show an exact reproduction of a given raster using hidden neurons. The number of these is increased when the relation between the number of neurons changes as shows Fig. 2.14. This is expected since we are in a situation of maximal randomness.

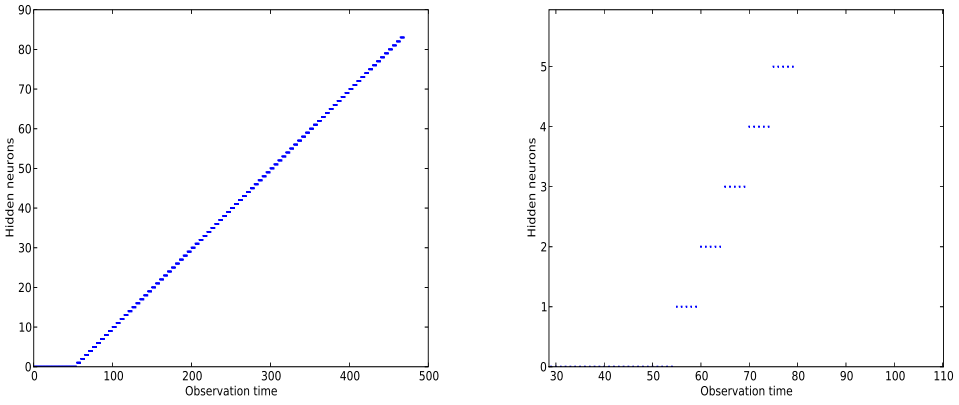


Figure 2.14: Relation between the number of hidden neurons S and the observation time T, here $N = 10$, $T = 470$, $D = 5$, $\gamma = 0.95$ for this simulation. The right-view is a zoom of the left view. This curves shows the required number of hidden neurons, using the proposed algorithm, in order to obtain an exact simulation. We observe that $S = \frac{T}{D} - N$, thus that an almost maximal number of hidden neuron is required. This curve has been drawn from 45000 independent randomly selected inputs.

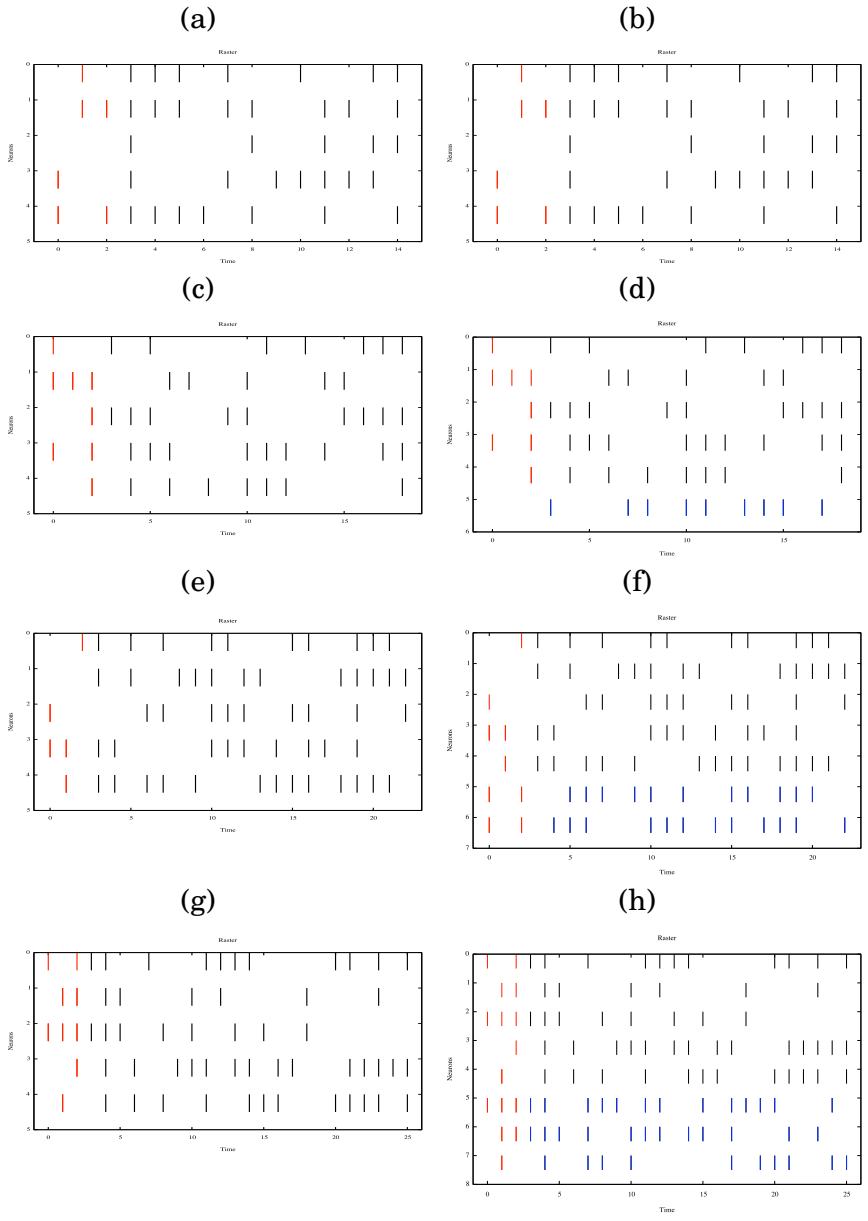


Figure 2.15: Finding the expected dynamics from a raster with uniform distribution. (a), (c), (e) and (g) correspond to different raster with Bernoulli distribution. In addition (b), (d), (f) and (h) show the raster calculated by the methodology proposed. The red lines correspond to initial conditions (initial raster), the black ones are the spikes calculated by the method and the blue ones are the spikes in the hidden layer obtained with a Bernoulli Distribution. We can also observe that the number of neurons in the hidden layer increases, 1 by 1, between (b), (d), (f) and (h), this is because the observation time T is augmented by 4, as predicted. Here $N = 5$, $\gamma = 0.95$, $D = 3$; in (a)(b) $T = 15$ with $S = 0$, in (c)(d) $T = 19$ with $S = 1$, in (e)(f) $T = 23$ with $S = 2$, in (g)(h) $T = 27$ with $S = 3$, while S correspond to the number of neurons in the hidden layer, detailed in the text.

Considering correlated distributions

We now consider a correlated random input, drawn from a Gibbs distribution Chazottes et al. (1998); Cessac et al. (2008a). To make it simple, the raster input is drawn from a Gibbs distribution, i.e. a parametrized range R Markov conditional probability of the form:

$$P(\{Z_i[k], 1 \leq i \leq N\} | \{Z_i[k-l], 1 \leq i \leq N, 1 \leq l < R\}) = exp\left(\Phi_\lambda(\{Z_i[k-l], 1 \leq i \leq N, 0 \geq l > -R\})\right)$$

where $\Phi_\lambda()$ is the related Gibbs potential parametrized by λ and Z a normalization constant.

This allows to test our method on highly-correlated rasters. We have chosen a potential of the form:

$$\Phi_\lambda(Z|_{k=0}) = r \sum_{i=1}^{N} Z_i[0] + C_t \sum_{i=1}^{N} \prod_{l=0}^{R} Z_i[l] + C_0 \prod_{i=1}^{N} Z_i[0]$$

thus with a term related to the firing rate r, a term related to multi-temporal autocorrelations C_t, and a term related to inter-unit synchronization C_0.

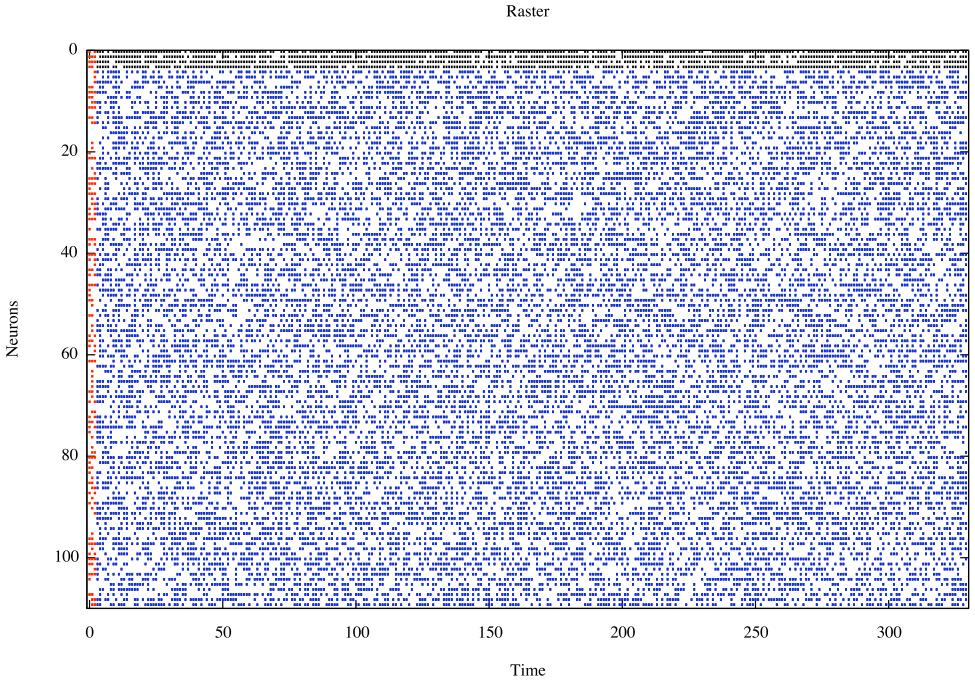


Figure 2.16: Raster calculated, by the proposed method, from a highly correlated Gibbs distribution. Here $r = -1$, $C_t = -0.5$ and $C_0 = -1$. The other parameters are $N = 4$, $\gamma = 0.95$, $D = 3$, $T = 330$ with $S = 106$. The red lines correspond to initial conditions (initial raster), the black ones are the input/output spikes and the blue ones are the spikes in the hidden layer.

We obtain a less intuitive result in this case, as illustrated in Fig. 2.16: even strongly correlated (but aperiodic) rasters are reproduced only if using

as many hidden neurons as in the non-correlated case.

This result is due to the fact that since the raster is aperiodic, non predictable changes occur in the general case, at any time. The raster must thus be generated by a maximal number of degrees of freedom, as discussed in the previous sections.

In order to further illustrate this aspect, we also show in Fig. 2.17 how a very sparse raster is simulated. We again obtain a solution with the same ratio of hidden neurons. This shows that the algorithm is very general, but not optimal in terms of number of hidden neurons.

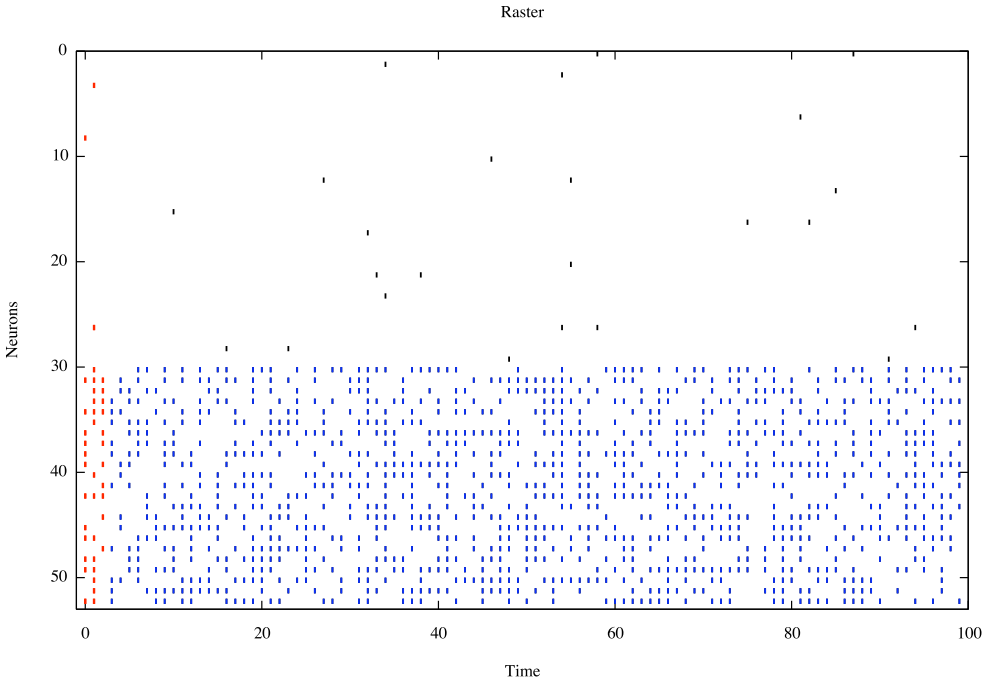


Figure 2.17: Raster calculated, by the proposed method, from a very sparse raster, with $N = 30$, $\gamma = 0.95$, $D = 3$, $T = 100$ and $S = 23$. The hidden neurons derived by the present algorithm simply allow to maintain the network activity in order to fire the sparse spikes at the right time. Color codes are the same as previously.

Considering biological data

As a final numerical experiment, we consider two examples of biological data set borrowed from Riehle et al. (2000) by the courtesy of the authors. Data are related to spike synchronization in a population of motor cortical neurons in the monkey, during preparation for movement, in a movement direction

and reaction time paradigm. Raw data are presented trial by trial (without synchronization on the input), for different motion directions and the input signal is not represented, since meaningless for the purpose. Original data resolution was $0.1ms$ while we have have considered a $1ms$ scale here.

What is interesting here is that we can apply the proposed method on non-stationary rasters, qualitatively very different, such as a very sparse raster, similar to the one shown in Fig. 2.17, a raster with two activity phases (presently movement preparation and execution) in Fig. 2.18 and a raster with a rich non-stationary activity in Fig. 2.19. In fact a dozen of such data sets have been tested, with the same result: exact raster reconstruction, with the same hidden unit ratio.

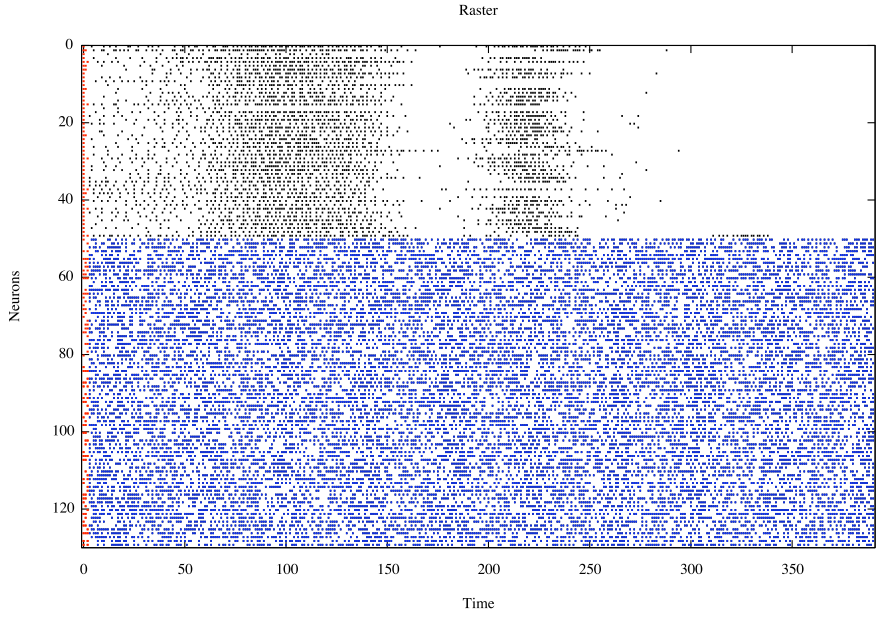


Figure 2.18: Raster calculated, by the proposed method, from a biological data set, with $N = 50$, $\gamma = 0.95$, $D = 3$ $T = 391$ and $S = 80$. From Riehle et al. (2000) by the courtesy of the authors.

On the computation time

Since the computation time is exclusively the LP problem resolution computation time we have simply verify that we roughly obtain what is generaly observed with this algorithm, i.e. that the computation time order of magnitude is:

$$O\left(S\,T\right)$$

when $N \ll T$, which is the case in our experimenation. On a standard laptop computer, this means a few seconds.

Figure 2.19: Raster calculated, by the proposed method, from a biological data set, with $N = 50$, $\gamma = 0.95$, $D = 5$ $T = 291$ and $S = 8$. From Riehle et al. (2000) by the courtesy of the authors.

2.4.5 Input/Output estimation

In this section we present results on the Input/Output matching, the objective is to find the parameters (delayed weights) for a transfer function and demonstrate that the methodology proposed in this work is also capable to learn certain functions in order to approximate input-output functions.

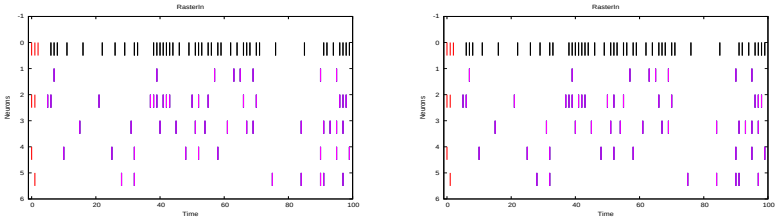


Figure 2.20: Input-Output dynamics matching, the purple lines represent the input dynamics, the black ones are the OR function of the inputs, it means that if at least one of the input neurons fire a spike in t the output fire a spike in $t + 1$, finally the red ones represent the initial conditions. $Ni = 5$, $No = 1$, $D = 0$, $T = 100$ and $S = 6$. Exact matching (diff $= 0$).

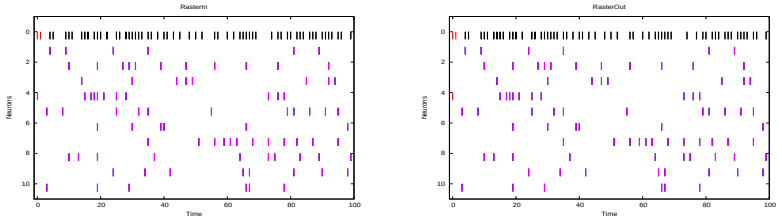


Figure 2.21: Input-Output dynamics matching, the purple lines represent the input dynamics, the black ones are the OR function of the inputs, it means that if at least one of the input neurons fire a spike in t the output fire a spike in $t+1$, finally the red ones represent the initial conditions. $Ni = 10$, $No = 1$, $D = 0$, $T = 100$ and $S = 10$. Approximate matching (diff $= 2$).

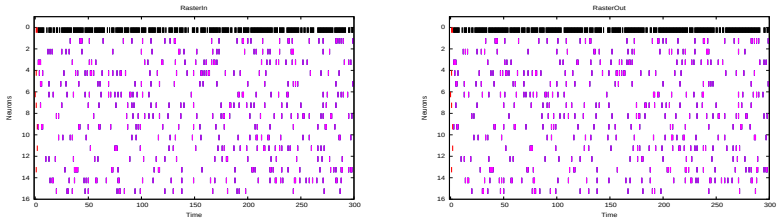


Figure 2.22: Input-Output dynamics matching, the purple lines represent the input dynamics, the black ones are the OR function of the inputs, it means that if at least one of the input neurons fire a spike in t the output fire a spike in $t+1$, finally the red ones represent the initial conditions. $Ni = 15$, $No = 1$, $D = 0$, $T = 300$ and $S = 15$. Approximate matching (diff $= 21$).

Part III

Programming resetting non-linear networks

CHAPTER 3

PROGRAMMING RESETTING NON-LINEAR NETWORKS

Science is more asking the right question than providing falsifiable answers
–Alex Williams

OVERVIEW

The aim of this more prospective chapter is to discuss how the ideas introduced in chapter 2 can be used in neural learning schemes and generalize to more complex models by considering an approximate estimation. More precisely, we sketch-out a novel computational approach that enables us to program non-linear neural networks by performing a mollification in the spiking metrics and deriving a temporal learning mechanism. On one hand, we propose an indexation of the spiking metrics permitting us to make more explicit each operation (*insertion, deletion and shift*) in the distance estimation, when we compare two spike trains. On the other hand the improvement in the learning mechanism consists in proposing a variational approximation of the synaptic weights using the mollified metric.

In any case, this chapter is more a prospective, and simply points out key facts in order to go beyond what has been done in chapter 2.

3.1 PROBLEM POSITION

Let us consider a network of $N_i + N_o + N_h$ spiking neurons, whose dynamics is defined by the equation (3.1) and schematized in figure 3.1 as a raster

plot (spike train).

$$V_o[k] = \gamma V_o[k-1] U'''_o[k-1] + \underbrace{\sum_{j=1}^{N_o+N_h} \sum_{d=1}^{D_o} W''_{ojd} U''_j[k-d]}_{Recurrent\,network} + \underbrace{\sum_{i=1}^{N_i} \sum_{d=1}^{D_i} W'_{oid} U'_i[k-d]}_{Feed-forward\,network} + I_o^{ext} \tag{3.1}$$

or in a compact way, writes:

$$\begin{aligned} V_o[k] &= \mathcal{F}_{\mathbf{W}'',\mathbf{W}'}(\cdots V_j[k''] \cdots, \cdots U_i[k'] \cdots), \qquad k - D^o \leq k'' < k,\ k - D^i \leq k' < k, \\ &= \mathcal{F}''_{\mathbf{W}''}(\cdots V_j[k''] \cdots) + \mathcal{F}'_{\mathbf{W}'}(\cdots U_i[k'] \cdots), \end{aligned} \tag{3.2}$$

where:

- $V_o[k]$ represents the membrane potential of the neuron with index o at time k.
- γ fixes a constant leak value between 0 and 1. Further we consider it variable $\gamma_i[k]$.
- $U'''[k-1]$ represents the reset mechanism. In both cases, analog (ρ) and discrete (Z) is considered as a binary function, $Z_o[k] = \rho(V_o[k]) \in \{0, 1\}$.
- U', U'' model the activation function. On one hand for a discrete system it is defined as a binary function, $Z \in \{0, 1\}$. On the other hand for the analog case, it is defined as a non-linear function, $\sigma(V) \in [0, 1]$.
- N_o, N_h and N_i determine the size of the network. Number of neurons in the output, hidden and input layers respectively.
- D_i, D_o define the maximum inter-neural delays for each layer (input and output).
- W', W'' represent the synaptic weights, which are randomly estimated from a random truncated normal distribution function.
- I_o^{ext} simulates an external stimulus.

In the case of a spiking unit at a microscopic scale, which resets its state when the value becomes higher than a threshold θ, we usually choose $U''_j[k] = \xi_{[\theta,+\infty[}(V_i[k]) \in \{0, 1\}$ defining the firing state ($U_i[k] = 1$ when spiking) of the unit, while $U'''_o[k] = 1 - U''_i[k]$ is the complementary state variable, defining the reset. See Cessac (2008) for a study of such dynamical system.

In the case of an analog unit at a mesoscopic scale, we typically choose $U''_j[k] = \sigma(g_{ok}\, V_i[k])$, i.e., a sigmoid profile with a slope of g_{ok} at zero, while $U'''_o[k] = 1$, thus without reset mechanism. See Rougier (2006) for a review on

discrete neural field with local connectivity. In this case, the state variable corresponds to firing rates, the sigmoid profile corresponding to the firing probability repartition function.

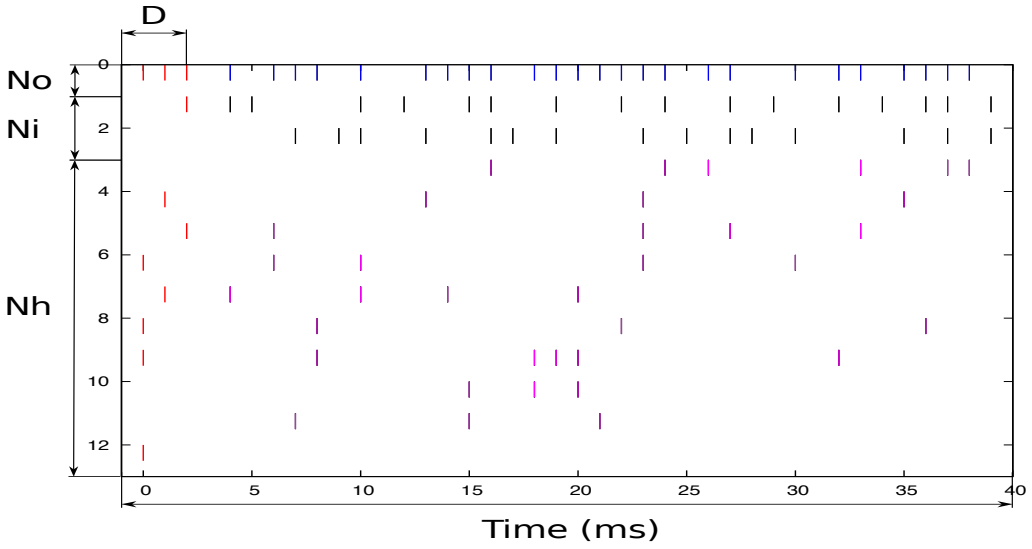


Figure 3.1: An input/ouput transformation based on spikes.

Thus, the equation (3.1) defines entirely an input/output system based on a discrete-time (or analog) spiking neural network. The system considers both, recurrent (W) and feed-forward (W') synaptic weights, where they can be written together:

$$\mathbf{W} = (\mathbf{W}', \mathbf{W}'')$$

Further, $\mathbf{W}$ is considered constant and the goal is to adjust it in order to make an adequate input-ouput mapping. Equation (3.1) also represents a deterministic system, where the estimation of the current membrane potential $V_o[k]$ depends on its initial state. In this sense for this work we have set it in zero ($V_o[k],\ 0 \leq k < D$), knowing that from a computational viewpoint it is equivalent to have transient inputs.

Figure 3.1 shows us the firing state $\mathbf{Z}$ of the network, where each color line represents an spike or spike. Spikes occur ever that the membrane potential V reaches a firing threshold θ. This relation writes:

$$\mathbf{R} = (\mathbf{Z}, \mathbf{V})$$

Since, the time has been discretized, we have a twofold issue. On one hand, it corresponds to the fact that not all continuous time sequences correspond to spike trains, since they are constrained by the network dynamics,

yielding global time constraints such as the fact that inter-spike intervals are bounded by a refractory period r and the fact that spike times are defined up to some absolute precision δt (see Cessac et al. (2010) for a detailed discussion), this being true in both biological and numerical implementations. The maximal amount of information, for one unit, is thus bounded during a finite period $[0, T]$ as stated in Cessac et al. (2010):

$$\frac{T}{r} \log_2 \left(\frac{T}{\delta t} \right) bits.$$

In a biological context, the order of magnitude is 1 Kbit/second for a neuron, which is coherent with biological observations Rieke et al. (1996). On the other hand, at a pragmatic level, time discretized network models are rather easy to study theoretically Cessac (2008); Cessac and Viéville (2008), trivial to simulate (contrary to continuous time models, see Brette et al. (2007) for discussion), and correspond without bias to what happens in a computer. We thus focus on discrete time and are going to briefly point out, from step to step, to which extent the present development can be applied to continuous time frameworks.

3.2 Mollification in the spiking metrics

From a mathematical definition, a metric is a function which defines a distance between elements of a set. In the particular case of a spiking metric in a **discrete** space, it gives us the distance between two finite spike trains $\bar{\mathbf{Z}}$ (desired output), $\mathbf{Z}$ (estimated output) and is defined as *the minimum cost of transforming one spike train into another*. In other words, two spike trains are similar if, with a few operations, we can edit one to equal the other. See Victor (2005) for an recent introduction. This distance is known as the Hamming distance and is defined as:

$$d\left(\bar{\mathbf{Z}}, \mathbf{Z}\right) = \sum_{ok} \Lambda_{ok} \left| \bar{Z}_o[k] - Z_o[k] \right| \tag{3.3}$$

counting the number of non-coincident spikes (e.g. an expected spike not obtained or an obtained spike not expected). Here since $|\bar{Z}_o[k] - Z_o[k]| \in \{0, 1\}$ all L^p-norms yield the same value, up to a convex increasing power function. For $\Lambda_{ok} = 1$ the value is exactly the number of non-coincident spikes, while other values of $\mathbf{\Lambda}$ allows us to enrich the semantic of such criterion.

Two kinds of operations are defined to this purpose:

- spike insertion or spike deletion, the cost of each operation being set to 1;

- spike shift, the cost to shift from $\bar{k}$ to k in order to obtain $\bar{Z}_o[\bar{k}] = 1 \Rightarrow Z_o[k] = 1$, increases with the delay $|\bar{k} - k|$, e.g. with $|\bar{k} - k|/\tau$ for a time constant τ.

In the case, where both spike trains are equal the distance will be zero (no editing operation). We also can deduce that the distance of two spike trains will be bounded by their number of spikes (i.e. the cost of delete/insert all spikes).

For small τ, the distance approaches the number of non-coincident spikes, since instead of shifting spikes it is cheaper to insert/delete non-coincident spikes. At the limit, when $\tau \to 0$ we re-obtain the coincidence distance.

Further, we study the spiking metric for the **analog** space, where spiking dynamics are at beginning described as analog values. Here, we could consider a well-defined metric i.e., using a quadratic form Λ

$$d\left(\bar{\mathbf{V}}, \mathbf{V}\right) = |\bar{\mathbf{V}} - \mathbf{V}|_{\mathbf{\Lambda}} = \sqrt{\sum_{oko'k'} \Lambda_{oko'k'} \left(\bar{V}_o[k] - V_o[k]\right) \left(\bar{V}_{o'}[k'] - V_{o'}[k']\right)} \quad (3.4)$$

or a weighted L^p-norm:

$$d\left(\bar{\mathbf{V}}, \mathbf{V}\right) = |\bar{\mathbf{V}} - \mathbf{V}|^p_{\mathbf{\Lambda}} = \left(\sum_{ok} \Lambda_{ok} \left|\bar{V}_o[k] - V_o[k]\right|^p\right)^{\frac{1}{p}} \quad (3.5)$$

where the cumulative difference between the desired and obtained output is weighted in order to take into account relative precisions between units and/or times, the fact that some components may not matter (the related weights Λ_{ok} being set to zero).

Using (3.4), for $\mathbf{Z}$, instead of (3.5), the coupling between units and time, can also be specified. So far, this metric will be mollified i.e. to *replace the metric by a convergent series of regular functions*. Here, since the non-differentiable ingredient is $\rho(u)$, this means replacing $\rho(u)$ by a parameterized family fo function $\rho_\upsilon(u)$, with $\lim_{\upsilon \to 0} \rho_\upsilon(u) = \rho(u)$.

We are also considering networks based on spikes with more than one unit, where our approach computes the distance for each alignment unit-to-unit. It means that we are considering that an spike can not "jump" from one unit to another. If this assumption is not true, the related estimation suffers from NP-completeness. See Aronov (2003) for a development and Victor et al. (2007) for a recent development.

This original metric can be generalized as follows Cessac et al. (2010):

Causality: Causal alignment metric. At a given time, the cost of the alignment of previous spikes decreases with the obsolescence of the spike, say, with an exponential profile parametrized by a time-constant τ'. When $\tau' \to \infty$, the original alignment metric is retrieved. This is a very interesting extension, allowing to work on on-line and not only off-line paradigms. It also takes into account adaptation to new spikes. With other weighting this can also implement mechanisms such as habituation (new spikes are not taken into account because, things are "already known").

Non-linearity: Non-linear shift cost. The cost of a shift is not necessarily a linear function of the delay, while any suitable non-linear function $\phi((\bar{k}-k)/\tau)$ can be used, as soon as $\phi()$ is symmetric and positive (i.e. with $\phi(u) = \phi(-u) \geq 0$), non decreasing, and with $\phi(0) = 0$. This allows us to enrich the notion of precision, for instance to state that below a threshold shifts has no importance, thus is cost-less. This also enable us to modulate the cost depending on the time scale, etc.. See also Dubbs et al. (2009) for the development of a true L^p alignment metric.

However, the theory is not complete, because the alignment distance calculation give us the minimal cost by editing one spike train to match the other, but does *not* make explicit what are the editing operations to obtain such a minimal cost. It is thus not possible to *match* spikes from one train to another, thus not possible to adjust the parameters in order to improve this match. In order to overcome this difficulty, we are going to introduce a so-called *alignment divergence*, in which an indexing function is going to break this barrier.

Integrating the two elements proposed here: differentiable *mollification* $\rho_\upsilon(u)$ of the spike binarization and design of an *alignment divergence* in the next section will allow us to solve the problem of choosing a suitable metric which can be properly minimized, i.e. solve the $\nabla_{\mathbf{R}} d\left(\bar{\mathbf{R}}, \mathbf{R}_{\mathbf{W}}\right)$ part of the formal optimization rule:

$$\nabla_{\mathbf{W}} d\left(\bar{\mathbf{R}}, \mathbf{R}_{\mathbf{W}}\right) = \nabla_{\mathbf{R}} d\left(\bar{\mathbf{R}}, \mathbf{R}_{\mathbf{W}}\right) \nabla_{\mathbf{W}} \mathbf{R}_{\mathbf{W}}.$$

Let us now to mollify the spike generation function $\rho()$ and see how it applies to the coincidence metric (Equation 3.3). In order to be able to enter into details, we focus on $\rho(u) = H(u-\theta)$, $H() = \xi_{]1,+\infty[}(x)$ being the Heaviside function defined with $H(0) = 0$. In other words, we focus on the fact an spike is defined by a state value above a given threshold θ. The generalization

to other semi-algebraic conditions is straightforward, since they always can be stated as a combination of Heaviside functions (see Benedetti and Risler (1990) for a treatise on the subject).

3.2.1 Threshold mollification

In words, we precisely need to replace $H(u)$ by a regular function that can influence the estimation,

- either if the condition is incorrect,
- or if the condition is correct, but close to be incorrect, i.e. at margin ν of the correctness boundary, while
- we better require the function to have no influence if the condition is correct and beyond this boundary.

A suitable function that fits with this requirement is the $H_{\upsilon,\nu}(u)$ non-linear regular profile represented in figure 3.2 and defined as:

$$H_{\upsilon,\nu}(u) \stackrel{\text{def}}{=} H(u + \upsilon\,\nu)\,\exp\left(-\frac{\upsilon}{u+\upsilon\,\nu}\right), \tag{3.6}$$

thus with $H_{\upsilon,\nu}(u) = 0, u \leq -\upsilon\,\nu$ and $H_{\upsilon,\nu}(+\infty) = 1$, while $\lim_{\upsilon\to 0} H_{\upsilon,\nu}(u) = H(u)$.

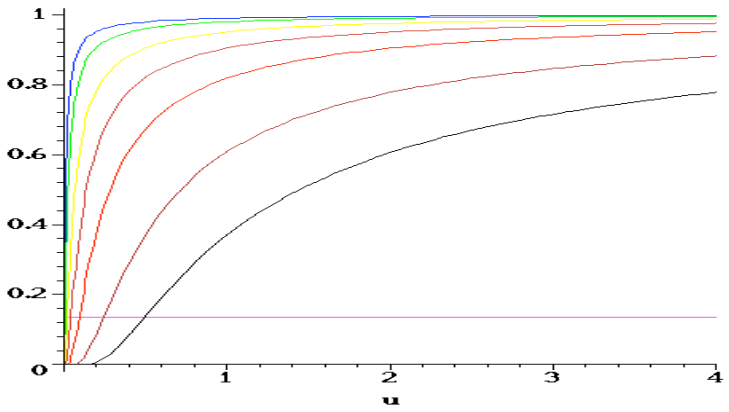


Figure 3.2: Defining the mollification of the Heaviside function $H()$. It is drawn here for $\nu = 0$ and in *black*, *brown*, *red*, *orange*, *yellow*, *green*, *blue*, for $\upsilon = [1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]$, respectively. The curves are convex below the magenta horizontal line.

3.2.2 Mollified coincidence metric

Now considering the coincidence metric (Equation 3.3) in the mollification, we can write the next:

$$d\left(\bar{\mathbf{Z}},\mathbf{Z}\right)=\lim_{\upsilon\to 0} d_{\upsilon,\varepsilon}(\bar{\mathbf{Z}},\mathbf{V})\ ,\ d_{\upsilon,\varepsilon}(\bar{\mathbf{Z}},\mathbf{V})=\sum_{ok}\Lambda_{ok}\,|\bar{Z}_o[k]-H_{\upsilon,\nu}\left(V_o[k]-\theta\right)|$$

for an spike threshold θ, a mollification parameter $\upsilon>0$, and a margin $\nu>0$.

The mollified metric $d_{\upsilon,\varepsilon}(\bar{\mathbf{Z}},\mathbf{V})$, is not a function of $\mathbf{Z}$ but of $\mathbf{V}$, since we need to observe the state value in order to drive towards the correct side of the threshold. Qualitatively[1], an increase of $V_i[k]$ tends to reduce shift delay, avoid deletion but induce insertion of spike, whereas a decrease of $V_i[k]$ tends to enlarge shift delay, induce deletion but avoid insertion of spike.

However, this is not the only possible mollification of (3.3), we can also write:

$$d\left(\bar{\mathbf{Z}},\mathbf{Z}\right)=\lim_{\upsilon\to 0} d_{\upsilon,\varepsilon}(\bar{\mathbf{Z}},\mathbf{V})\ ,\ d_{\upsilon,\varepsilon}(\bar{\mathbf{Z}},\mathbf{V})=\sum_{ok}\Lambda_{ok}\,H_{\upsilon,\nu}\left(\left(1-2\,\bar{Z}_o[k]\right)\left(V_o[k]-\theta\right)\right) \tag{3.7}$$

since $H\left(\left(1-2\,\bar{Z}_o[k]\right)\left(V_o[k]-\theta\right)\right)=|\bar{Z}_o[k]-H\left(V_o[k]-\theta\right)|$, while, now:

$$\nabla_{V_o[k]} d_{\upsilon,\varepsilon}\left(\bar{\mathbf{Z}},\mathbf{V}\right)=\upsilon\,\Lambda_{ok}\,\frac{H_{\upsilon,\nu}\left(\left(1-2\,\bar{Z}_o[k]\right)\left(V_o[k]-\theta\right)\right)}{(V_o[k]-\theta_{ok})^2} \tag{3.8}$$

with $\theta_{ok}=\theta-(1-2\,\bar{Z}_o[k])\,\upsilon\,\nu$, corresponding the threshold thickened, defining a margin.

The interest of using this variant is that it has a symmetric effect below and above the theshold: the state values are always driven away without any limited range, which is not the case with the former form as made explicit in footnote[1].

This criterion enjoys the following properties. *There is always a υ, such that the criterion is convex.* This is the case if we consider $\upsilon>2\,\max\left(V_{max}-\theta,\theta-V_{min}\right)$, thus with $H_{\upsilon,\nu}$ convex, for all $V\in]V_{min},V_{max}[$. Then the criterion is convex in this range, as the positive linear combination of convex functions.

[1] Quantitatively, the gradient reads:

$$\nabla_{V_o[k]} d_{\upsilon,\varepsilon}\left(\bar{\mathbf{Z}},\mathbf{V}\right)=\upsilon\,\Lambda_{ok}\,\xi\,\tfrac{H_{\upsilon,\nu}(V_o[k]-\theta)}{(V_i[k]-\theta_{\upsilon,\nu})^2}\quad \xi=\begin{cases} 1 & , \quad \bar{Z}_o[k]=1 \\ -1 & , \quad \bar{Z}_o[k]=0 \quad , \quad V_i[k]>\theta_{\upsilon,\nu} \\ 0 & , \quad \bar{Z}_o[k]=0 \quad , \quad V_i[k]\le\theta_{\upsilon,\nu} \end{cases}$$

with $\theta_{\upsilon,\nu}=\theta-\upsilon\,\nu$, while $\xi=sg(\bar{Z}_o[k]-H_{\upsilon,\nu}\left(V_o[k]-\theta\right))$. This corresponds to what is qualitatively described in the text.

3.2.3 Indexing the alignment distance

Let us introduce an alignment indexation between the "expected spike train" $\bar{\mathbf{Z}}$ and the "observed spike train" $\mathbf{Z}$, in the discrete case.

$$\delta : \bar{k} \to \delta(\bar{k}) \tag{3.9}$$

so that, given the alignment distance $d(\bar{\mathbf{Z}}, \mathbf{Z})$: $\delta(\bar{k}) = k - \bar{k}$ if $\bar{Z}[\bar{k}] = 1$ and $\bar{Z}[\bar{k}]$ and $Z[k]$ are aligned by a shift, while $\delta(\bar{k}) = 0$ otherwise.

In words, an indexing $\delta()$ allows to align $\bar{\mathbf{Z}}$ onto $\mathbf{Z}$, in accordance to $d(\bar{\mathbf{Z}}, \mathbf{Z})$. More precisely, we not only compute the distance but make explicit the alignment operations ($shift$, $deletion$, $insertion$) allowing to "edit" $\mathbf{Z}$ in order to obtain $\bar{\mathbf{Z}}$. Obviously, several alignment operation sequences may lead to the same minimal alignment cost. In order to make a choice, from the last time to the previous time, we consider that shift is preferable to edition (i.e., insertion/deletion), since it is a reasonable assumption to heuristic that it is going to have a less important influence on the dynamics than the apparition/cancelation of an unexpected spike. It is going to be made algorithmically explicit now that this defines a unique indexing function.

The distance $d_{\bar{n},n}$ between the first $\bar{n}$ spikes in $\bar{\mathbf{Z}}$ and the first n spikes $\mathbf{Z}$ and the indexing function δ defining this so-called divergence are iteratively defined as follows, after Victor (2005) but now including the indexing mechanism.

On one hand, $d_{0,n} = n$ due to the fact that the distance between any spike train and the empty spike train corresponds to the cost of deleting all spikes, while $\delta(\bar{k}_{\bar{n}}) = +0$ in this case. Similarly, $d_{\bar{n},0} = \bar{n}$ corresponds to inserting all spikes, while $\delta(\bar{k}_{\bar{n}}) = -0$ in this case.

On the other hand, by induction, writing k_n the n-th value such that $Z[k_n] = 1$ and $\bar{k}_n$ the n-th value such that $\bar{Z}[\bar{k}_n] = 1$, for $n \geq 0$ and $\bar{n} \geq 0$:

$$d_{\bar{n}+1,n+1} = \min \begin{cases} d_{\bar{n},n+1} & + & 1 & (deletion) & \Rightarrow & \delta(\bar{k}_{\bar{n}}) & = & +0 \\ d_{\bar{n}+1,n} & + & 1 & (insertion) & \Rightarrow & \delta(\bar{k}_{\bar{n}}) & = & -0 \\ d_{\bar{n},n} & + & c_{\bar{n},n} & (shift) & \Rightarrow & \delta(\bar{k}_{\bar{n}}) & = & k_n - \bar{k}_n \end{cases} \tag{3.10}$$

where:

$$c_{\bar{n},n} \stackrel{\text{def}}{=} \phi\left(\frac{\bar{k}_{\bar{n}} - k_n}{\tau}\right) \tag{3.11}$$

while choosing the $shift$ operation if the relative cost increase is not higher than $deletion$ and $insertion$, i.e. not higher than one, otherwise the required $deletion$ or $insertion$. By doing this, there is now a unique well-defined indexing function for a given distance, that solves some ambiguities, while, on the reverse, solving these two ambiguities allows us to define algorithmically a unique indexing function.

There is also a variational definition of the alignment divergence, and its related indexing, which writes:

$$d\left(\bar{\mathbf{Z}},\mathbf{Z}\right)=\min_{\delta}\quad\begin{array}{ll}\sum_{k,\forall k'\neq k,\delta[k']\neq k}\Lambda_o^k\left[|\bar{Z}[k+\delta[k]]-Z[k]|+\phi\left(\frac{\delta[k]}{\tau}\right)\right]\\ +\sum_{k,\,\bar{Z}[k]=1,\;k'=\mathbf{argmin}_{k'>k}\bar{Z}[k']=1}\mu_k\left[1+\frac{\delta[k']-\delta[k]}{k'-k}\right]\end{array}\tag{3.12}$$

using $\delta[k]=0$ as initial value. Here $\Lambda_o\leq 1$ for the definition to be well-defined. For the standard alignment distance $\phi(u)=|u|$ and $\Lambda_o=1$. The Kuhn-Tucker multipliers μ_k allows to preserves spike sorting and provides a one-to-one mapping, i.e. guaranty that the indexing is strictly increasing.

Finally we can write the final mollification distance taking into account the shift indexing:

$$\boxed{\begin{array}{ll}d\left(\bar{\mathbf{Z}},\mathbf{Z}\right)=\lim_{\upsilon\to 0}&\sum_{ok,\delta[k]=0}\Lambda_{ok}\,H_{\upsilon,\nu}\left(\left(1-2\,\bar{Z}_o[k]\right)\left(V_o[k]-\theta\right)\right)+\\&\sum_{ok,\delta[k]\neq 0}\Lambda_{ok}\,\phi\left(\frac{\delta[k]}{\tau+\nu}\right)\,H_{\upsilon,\nu}\left(\left(1-2\,\bar{Z}_o[k+\delta[k]]\right)\left(V_o[k]-\theta\right)\right)\end{array}}\tag{3.13}$$

with the interesting property that for $\Lambda_{ok}=1$ and $\phi(u)=|u|$ this corresponds exactly to standard alignment metric, as the reader can easily verified.

The criterion variation balances two contradictory effects: a required deletion/insertion or shift may have been satisfied, versus a correct reset/spike state may have been canceled, with the key aspect that changes are now differentiable, thanks to the mollification.

The term with $\delta[k]=0$ includes correct reset/spike states and deletion/insertion corresponds to the coincidence metric, the second term with $\delta[k]\neq 0$ correspond to shifts. The gradient is obvious to derive and corresponds to (3.8), if $\delta[k]=0$, with an additional factor otherwise. For a fixed indexing δ, we obviously have the same local convexity property as detailed for the coincidence metric.

This gives us a heuristic tool to optimize the weights with respect to a general class of alignment metric. This metric being not convex, it may have several minima, and the present methods, though well-defined and well-founded, is only expected to find a local minimum, not necessarily "the" optimal solution.

3.3 LEARNING PARADIGMS.

As pointed out, in section 3.1, our problem position is to solve the "network programming" problem which can be stated as follows: *find, whenever*

possible, network weights verifying input/output relations, under some constraints.

Such a class of paradigms differs from what is often addressed in biological context, where learning is mainly related to synaptic plasticity Gerstner and Kistler (2002b); Cooper et al. (2004) and STDP (Spike-Time-Dependent Plasticity), as far as spiking neural networks are concerned (see e.g., Toyoizumi et al. (2007) for a recent formalization). Such unsupervised learning mechanism is known to reduce the variability of neuron responses Bohte and Mozer (2007) and to be related to the maximization of information transmission Toyoizumi et al. (2005) and mutual information Chechik (2003). Our point of view is different, since we consider supervised and reinforcement learning as detailed now: Given some *learning samples* of input/output data, the goal is to adjust the weights in order the network output to be as "close as possible" to the desired output.

3.3.1 The learning scenario.

A learning sample is defined here by a finite set of T spikes and/or values, as schematized in figure 3.1. This corresponds to an *epoch*, i.e. an arbitrary predefined period of time T. We are going to consider the simple scenario in which the learning data is given as a set of such epochs. The present development is easily generalizable to more realistic temporal scenari, the present restriction being only chosen for the sake of simplicity.

For a given input, in the present paradigm, we consider both *desired* output, the network being required to produce *exactly* or *approximately* such output, and *hidden* output, which values do not matter. Hidden ouput are added in order to increase the number of degrees of freedom of the network. This defines a *reservoir* of spikes and/or values with the goal to increase the computational power.

In this context, let us make explicit three learning schemes.

3.3.2 Supervised learning scheme.

In a *supervised learning scheme*, a set $\mathcal{S} = \{{}_1\bar{\mathbf{R}}, {}_2\bar{\mathbf{R}}, \cdots {}_M\bar{\mathbf{R}}\}$ (using the $\bar{}$ symbol for desired output) of M learning samples is presented and the learning algorithm is formally defined as:

$$\min_{\mathbf{W}} \mathcal{C}_s, \text{ where } \mathcal{C}_s = \sum_{mk} \Lambda_{mk}\, \varrho\left(d\left({}_m\bar{\mathbf{R}}[k], \mathbf{R}_{\mathbf{W}}[k]\right)\right) \tag{3.14}$$

for some increasing function $\varrho()$ (e.g. $\varrho : u \to u^2$, other profiles being discussed below), and for some weighting Λ_{mk}. For instance, if all samples are equivalent and $M < +\infty$, we simply choose $\Lambda_{mk} = 1$. We may also have

to consider an unbounded (in practice very large) number of samples, the previous criterion being well-defined as soon as $\lim_{M\to+\infty}\sum_m |\Lambda_{mk}| < +\infty$. Note that we consider a weighting at two temporal scales: between times indexed by k inside an epoch of index m, and between epochs.

In other words, we specify that the weights $\mathbf{W}$ must be adjusted in order to minimize some cumulative distances $d()$ between the desired output ${}_m\bar{\mathbf{R}}, m \in \{1, M\}$ and the obtained output $\mathbf{R}_\mathbf{W}$. We write $\mathbf{R}_\mathbf{W}$ to make explicit the fact that the obtained output depends on the weights $\mathbf{W}$.

Such a problem position leads to two key questions:

- Which suitable "temporal" metric $d()$, regarding spikes, can be considered and how to adjust such metric ? And how does the result depends on the metric ?
- How to compute the weights variation influence on such a metric ? Considering input and recurrent weights, in our case.

These are the two key points addressed and developped in this chapter, in sections 3.2 and 3.4 respectively. Interesting enough, addressing these issues not only enable us to program the studied class of networks in a supervised learning scheme but also in more complex learning schemes, as discussed now.

3.3.3 Robust learning scheme.

In a *robust learning scheme*, a set $\mathcal{S} = \{{}_1\bar{\mathbf{R}}, {}_2\bar{\mathbf{R}}, \cdots {}_M\bar{\mathbf{R}}\}$ of M learning data is also presented, but not all the data are relevant. Some are *outliers*, i.e. "mistakes", and must not be taken into account in the estimation. In other words, we have to both estimate the weights $\mathbf{W}$ and to discriminate which data correspond to these weights. Formally, this corresponds to define a weighting $\Lambda_{mk} \in \{0, 1\}$ with $\Lambda_{mk} = 0$ if and only if the result is an outlier, so that it is not taken into account in the minimization of (3.14). The goal is to obtain the capability to derive unbiased estimations even in the presence of numerous outliers, and in the presence of outliers with huge errors with respect to the expected values, but also in the presence of *inliers*, i.e. spurious data corresponding to another configuration not to be taken into account here. There is a strong difference between outliers and inliers: outliers are random samples, while inliers correspond to other structures, thus may trap the estimation in a spurious estimate. See, e.g., Rousseeuw and Leroy (1987) for a general introduction, Davé and Krishnapuram (1997) for unified view of these classes of methods and Comaniciu and Meer (2001) for a detailed discussion on their properties and limits.

In a nutshell, two main classes of methods are used to this purpose. On one hand, M-estimators use non-convex profiles $\varrho()$ as sketched out in

Fig. 3.3-A, in order to decrease the influence of potential outliers. This corresponds to a criterion of the form of (3.14). Such methods have rather limited performances in the presence of inliers.

On the other hand, L-estimators, e.g., RANSAC (RANdom SAmple Consensus) methods, introduce a complete different paradigm. Such algorithms perform *trials*, select the best trial, and then threshold correct data, before refining the estimation, as detailed now.

- Performing a trial means randomly drawing a minimal number D of data from $\mathcal{S}$, in order to be able to perform an exact estimation of the parameter $\mathbf{W}$, under the assumption that these data are correct (neither an outlier, nor an inlier). Such estimation can be implemented minimizing (3.14) for this minimal set of data, yielding $\mathcal{C}_s = 0$, since this yields an exact estimation. Given M data and K independent uniformly distributed trials, the need to select at least D data to perform a relevant estimation, and a probability p to draw a correct sample, the chance to draw at least one trial with correct data writes:

$$p_{\text{success}} = 1 - \left(1 - p^{\frac{D}{M}}\right)^K ,$$

  which means an exponential dependence with respect to the dimension D. This is the reason why we always need to consider a *minimal* number D of sample for this step.

- For the next step, we have to select the "best" trial among the K, assuming this corresponds to a correct sampling. The way to do it differs with the method. In the RANSAC algorithm, the trial with the maximal number of small marginal errors is selected, as sketched out in Fig. 3.3-B. More precisely, the count of data with an estimation error lower than a given threshold is used. Other design choices consist in selecting the trial with the lower median (or other percentile) error, or more sophisticated error histogram indicators Viéville et al. (2001), as sketched out in Fig. 3.3-C. These methods take into account the fact that due to noise or numerical errors, trials with correct data may lead to poor selection score. Hopefully, a trial with incorrect data is not expected to yield a spurious good score. Many variants exist (e.g., drawing trials until a "sufficiently correct" result is obtained, etc..) but are still based on the same ingredients.

- The final step is to select the subset of $\mathcal{S}$ with data with an estimation error lower than a given threshold, and use them in order to refine the estimation, as in a standard supervised scheme.

In our context, the implementation of such scheme thus leads to three key steps: (i) estimation of the weights with a minimal number of data, in order to obtain, if any, an *exact* input/output relation of the parameters $\mathbf{W}$, (ii) compare the distance on results between the obtained input/output relation and the expected results on other data to estimate the marginal error, (iii) refine the estimation as in the supervised paradigm. All steps are going to be designed and developed in the sequel. The issue (i) has been recently entirely solved Rostro-Gonzalez et al. (2009a,b, 2010) for a large class of models proposing a formulation leading to a polynomial algorithm. Issues (ii) and (iii) correspond to the same issues to be addressed in the supervised learning scheme. As a consequence, solving the supervised learning scheme, is going to provide the suitable ingredients to solve the robust learning scheme described here.

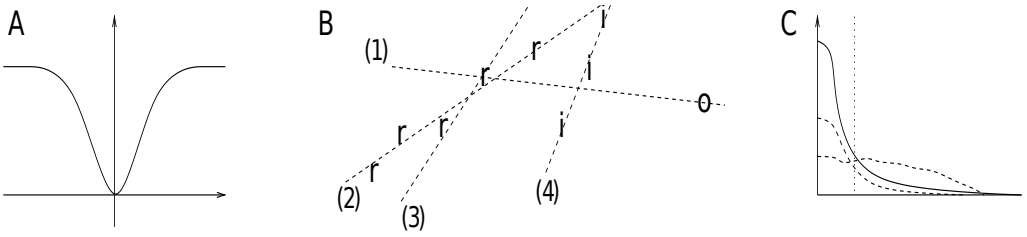


Figure 3.3: **A** Left view: a typical non-convex M-estimator profile $\varrho()$, for robust estimation: for small errors the profile corresponds to a convex, e.g. quadratic shape, while for higher errors the contribution "saturates" and the sample is no more taken into account in the variational minimization. **B** Middle view: showing a few trials of a L-estimator, for a set $\mathcal{S}$ with expected results r, an outlier o and inliers i corresponding to another spurious trial set; the trial (1) contains expected result and outlier, thus generates large marginal errors; the trial (2) contains expected results only, thus generates several small marginal errors; the trial (3) contains expected results only, but is too imprecise to generate several small marginal errors; the trial (4) contains inliers, thus generates several small marginal errors but for a smaller number of results than the trial (2). If the trial with a maximal number of small marginal errors is kept, i.e. trial (2) in this example, the method is successful. **C** Right-view: qualitative view of the marginal errors histogram for a "correct" (in plain line) versus "spurious" trials (in dashed lines): a large amount of small marginal errors is expected in the former case, while a dispersion of errors is expected in the second case. A typical threshold value, as used in a RANSAC method, is represented by the vertical dot line.

3.3.4 Reinforcement learning scheme.

In a *reinforcement learning scheme*, given a result ${}_m\mathbf{R}_{\mathbf{W}}, m \in \{1, M\}$ a feedback in terms of a bounded *reward* ${}_m\mathbf{r}_k \in \mathcal{R}$ is available for each sample, but without any explicit comparison between the (now unknown) desired output and the obtained output (see, e.g., Sutton and Barto (1998) for a general introduction and Wörgötter and Porr (2005) for a recent review in link with the present topic, more details being out of the scope of this work). It is thus necessary to estimate the functional link:

$$\nu : \mathbf{R}_{\mathbf{W}} \longrightarrow \mathbf{r}$$

between the produced output and the obtained reward, yielding an estimation / minimization (EM) problem Green (1990) (here, the *estimation* of ν and the *minimization* of $\mathbf{W}$) of the formal form:

$$\min_{\mathbf{W},\nu} \mathcal{C}_r, \text{ with } \mathcal{C}_r = \sum_{mk} {}_m r_k, \; {}_m r_k = \nu_k \left(\cdots {}_m\mathbf{R}_{k'\mathbf{W}} \cdots \right), k' \leq k$$

In the general case, the reward ${}_m r_k$ depends on previous samples, as made explicit here. Several particular cases are relevant, such as ${}_m r_k = \delta_{k(T-1)} \, {}_m r$ when a unique reward ${}_m r$ is given at the end of the epoch, or ${}_m r_k = \gamma^k \, {}_m r'_k$ when the weighting of the returned reward ${}_m r'_k$ is subject to an exponential decay.

In order to obtain a tractable framework in relation with the present context, we propose to assume that the local variation of $\nu()$ is a weighted regular function $\varrho()$ of the distance between the corresponding outputs, i.e.:

$$\nu_k \left({}_m\mathbf{R}_{k\mathbf{W}} \right) - \nu_{k-1} \left({}_m\mathbf{R}_{k-1\mathbf{W}} \right) = \frac{1}{T} L_{mk} \, \varrho \left(d \left({}_m\mathbf{R}_{k\mathbf{W}}, {}_m\mathbf{R}_{k-1\mathbf{W}} \right) \right),$$

for some weights L_{mk}. This is the way we propose to introduce a temporal learning (TD) scheme Sutton and Barto (1998) in this context. We further assume that the reward depends on the last sample only, while the generalization to a sum of distances with previous rewards is straightforward. Taking into account the distance only, simply means that we do not consider the influence of the orientation in the $(\mathbf{Z}, \mathbf{V})$ space, i.e. that we assume the reward's variation to be locally spherically symmetric. As a consequence, the likely intractable functional estimation of the vectorial function $\nu \in \mathcal{R}^{T\mathcal{R}^{(N^o+N^i)\,T}}$ reduces to the functional estimation of the scalar function $\varrho \in \mathcal{R}^{\mathcal{R}}$, which seems to be a reasonable choice. The estimation step of such a scalar function, for this EM problem, has been extensively addressed elsewhere (see, e.g., Green (1990) for a development directly reusable here) and is not going to be further reviewed here.

The minimization step is however not so trivial as made explicit now. Combining the two previous equations, we obtain:

$$\min_{\mathbf{W},\varrho} \mathcal{C}_r, \mathcal{C}_r = \sum_{mk} \Lambda_{mk}\, \varrho\left(d\left({}_m\mathbf{R}_k\mathbf{W}, {}_m\mathbf{R}_{k-1}\mathbf{W}\right)\right) + O\left(\frac{1}{T}\right) \tag{3.15}$$

for $\Lambda_{mk} = \frac{T-k}{T} L_{mk}$. With such a formulation, we have reduced the minimization step of this estimation/minimization problem to a variational problem [2] equivalent to the supervised scheme, if we neglect the last term in $O\left(\frac{1}{T}\right)$. As a consequence, solving the supervised learning scheme is going to provide the suitable ingredients to solve the specific form of reinforcement learning scheme described here. Generalzing the present issue to other reinforcement learning schemes is a perspective of the present work.

3.3.5 Other learning paradigms.

In addition to these three fundamental learning schemes, the estimation ingredients identified previously can be applied to other composite learning schemes, such as *robust multi-model* paradigms where different recurrent equations $\mathcal{F}$ can be compared and/or different inlier sets segmented (see, e.g. Viéville et al. (2001) in the case of visual processing, or Davé and Krishnapuram (1997) for segmentation tasks), in virtue of the computational richness of the variational formulation proposed here, including mixing reinforcement learning schemes with robust approaches.

In order to solve these variational problems, it appears that we "simply" have to choose a suitable temporal metric $d()$, and in order to adjust such a metric, make explicit how to compute the weight variation influence on such a metric. This is in fact not so trivial, as made explicit in the next sections.

The present development does not explicitly target unsupervised or semi-supervised learning as in deep-learning schemes[3]. A step further, we do not address "active learning" paradigms, i.e. paradigms in which we explicitly control the input or output. For instance, we know from the dynamic system theory that if we generate a dense trajectory (i.e. adding noise to avoid the orbit to be periodic), its observation allows to correctly approximate the system parameter, in the general case. Furthermore, active perceptual strategies allow the estimation to optimize the learning process, using an extra

[2] For instance, considering a new results ${}_m\mathbf{R}_k\mathbf{W}$ and assuming the information regarding the previous result ${}_m\bar{\mathbf{R}}_{k-1}$ to be worked out, we can write:

$$\nabla_W \mathcal{C}_r|_{{}_m\mathbf{R}_{k-1}\mathbf{W}={}_m\bar{\mathbf{R}}_{k-1}} = \Lambda'_{mk}\, \nabla_W d\left({}_m\mathbf{R}_k\mathbf{W}, {}_m\bar{\mathbf{R}}_{k-1}\right),$$

with $\Lambda'_{mk} = \varrho'_k\left(d\left({}_m\mathbf{R}_k\mathbf{W}, {}_m\mathbf{R}_{k-1}\mathbf{W}\right)\right)\, \Lambda_{mk}$, thus adjust the weights depending on the local increase/decrease of the reward. In words, the related local learning rule in this EM paradigm is proportional to the estimation of the rewards variation derivative $\nu'()$ and the metric gradient $\nabla_W d()$.

[3] See `http://deeplearning.net` for details.

mechanism of input/output control.

The add-on of the present work is elsewhere, i.e. to consider general recurrent networks without any restriction on the internal architecture, plus introduce the idea to optimize hidden states of the network, as discussed in section 3.4.

3.4 Recurrent weights adjustment

Once defined the mollification on spiking metrics, the question now is to know, how to compute the weights variation influence on such metric to yield in a robust learning mechanism?. In order to answer it, let us to consider the learning scenario, shown in figure 3.1. This scenario corresponds to a *sample* and considering the simple case where learning data is given as a set of such samples.

Let us analyze how we can compute the influence of weight variation, i.e. $\nabla_{\mathbf{W}}\mathbf{R}_{\mathbf{W}}$. We start discussing the influence on output values, i.e. $\nabla_{\mathbf{W}}\mathbf{V}_{\mathbf{W}}$, then we extend it to output spikes, i.e. $\nabla_{\mathbf{W}}\mathbf{Z}_{\mathbf{W}}$. From equation 3.1 we writes the variations of forward and recurrent weights as follows:

$$\begin{aligned}\nabla_{\mathbf{W}'}\mathbf{V}[k] &= \nabla_{\mathbf{V}}\mathcal{F}''_{\mathbf{W}''}(\mathbf{V}[k-1])\,\nabla_{\mathbf{W}'}\mathbf{V}[k-1] + \nabla_{\mathbf{W}'}\mathcal{F}'_{\mathbf{W}'}(\mathbf{U}[k-1]),\\ \nabla_{\mathbf{W}''}\mathbf{V}[k] &= \nabla_{\mathbf{W}''}\mathcal{F}''_{\mathbf{W}''}(\mathbf{V}[k-1])\,\nabla_{\mathbf{W}''}\mathbf{V}[k-1].\end{aligned} \tag{3.16}$$

The former equation is straightforward to use if there is no recurrent weights (i.e. if $\mathcal{F}''() = 0$), simplifying to $\nabla_{\mathbf{W}'}\mathbf{V}[k] = \nabla_{\mathbf{W}'}\mathcal{F}'_{\mathbf{W}'}(\mathbf{U}[k-1])$, which corresponds to a feed-forward network. Otherwise, both equations are more complex since the recurrent weights variation recursively depends on itself.

In computational context, networks dealing with spikes, e.g. spiking neural networks, are mainly implemented through specific network architectures, such as Echo State Networks Jaeger (2003) and Liquid Sate Machines Maass et al. (2002), that are called "reservoir computing" (see Verstraeten et al. (2007) for unification of reservoir computing methods at the experimental level). In this framework, the reservoir is a network model of neurons (can be linear or sigmoid neurons, but more usually spiking neurons), with a random topology and a sparse connectivity.

The key idea is that the forward layer of the network (the so-called "readout" layer, here the output layer) is driven by a supervised learning rule of the forward weights, generated from any type of classifier or regressor, ranging from a least mean squares rule to sophisticated discriminant or regression algorithms. However the recurrent layer of hidden units is not

explicitly learned, whereas recurrent weights are either randomly fixed or adjusted using unsupervised learning mechanism, without any direct connection with the learning samples (though the hidden unit statistics, for instance, is sometimes adjusted in relation with the desired output). In the case of temporal mechanisms, i.e. using spiking neurons (e.g. in the model of Paugam-Moisy et al. (2008)), the unsupervised learning mechanism of the recurrent weights is a form of synaptic plasticity, usually STDP (Spike-Time-Dependent Plasticity), a temporal Hebbian unsupervised learning rule, biologically inspired. It appears that simple methods yield good results Verstraeten et al. (2007), while the ease of training and a guaranteed optimality guides the choice of the method.

This distinction between a readout layer and an internal reservoir is induced by the fact that only the output of the neural network activity is constrained, whereas the internal state is not to be controlled.

It is obvious to notice that the reservoir architecture is a special case of the general architecture written in (3.1), where output units ${}_o\mathbf{V}$ and hidden units ${}_h\mathbf{V}$ have been explicitly separated, while the recurrent part $\mathcal{F}'$ of the update equation only depends on the hidden units ${}_h\mathbf{V}$ only, i.e.:

$$\begin{aligned} {}_oV_o[k] &= {}_o\mathcal{F}''_{{}_o\mathbf{W}''}(\cdots {}_hV_j[k'']\cdots) + {}_o\mathcal{F}'_{{}_o\mathbf{W}'}(\cdots U_i[k']\cdots), \\ {}_hV_h[k] &= {}_h\mathcal{F}''_{{}_h\mathbf{W}''}(\cdots {}_hV_j[k'']\cdots) + {}_h\mathcal{F}'_{{}_h\mathbf{W}'}(\cdots U_i[k']\cdots). \end{aligned}$$

Here, output forward weights ${}_o\mathbf{W}''$, ${}_h\mathbf{W}'$ are explicitly learned, whereas recurrent weights ${}_h\mathbf{W}''$ are not explicitly adjusted. In fact, the forward weights ${}_h\mathbf{W}'$ could be adjusted by back-propagation, though this is does not seem to be used, up to our best knowledge of the literature.

Contrary to specific architectures such as deep-learning Bengio and LeCun (2007) or reservoir computing Verstraeten et al. (2007), the variational framework proposed is not restrained to the design of a specific class of architecture. The solution is valid for recurrent network with or without connectivity restriction, including hidden values adjustment. This is computationally tractable because we are in a discretized time scheme.

With the notation of (3.2), for a desired output $\bar{\mathbf{V}}$, the present variational problem is to be written:

$$\min_{\mathbf{W},\mathbf{V}=({}_o\mathbf{V},{}_h\mathbf{V})} \max_{\mu_k} \mathcal{C}_v,\ \mathcal{C}_v = \sum_k d_k({}_o\bar{\mathbf{V}}[k], {}_o\mathbf{V}[k]) + \sum_k \mu_k^T \left(\mathbf{V}[k] - \mathcal{F}(\mathbf{V}[k-1])\right) \tag{3.17}$$

where:

- the dependency with respect to $\mathbf{U}$ has been omitted for the sake of clarity;
- the input and output weights $\mathbf{W} = (\mathbf{W}', \mathbf{W}'')$ are considered together, because of the interdepence made explicit in (3.16).

- we make explicit the *output* part, ${}_o\mathbf{V}$, of the state values, which variation modified the metric defining the criterion, from the *hidden* part, ${}_h\mathbf{V}$, which are not taken into account in this metric, but has nevertheless to be explicitly adjusted;

- we decompose the metric $d()^p = \sum_k d_k()$ with respect to its contribution $d_k()$ at each sampling time of index k, as the structure of metrics like (3.4) or (3.5), since minimizing $d()$ is equivalent to minimize $d()^p, p > 0$;

- we introduce Lagrange multipliers $\mu_k \in \mathcal{R}^{N\,D^o}$, used to formalize the fact that the output and hidden state values are interdependent with respect to the recurrent weights,thus must be estimated in the same process.

Stating the estimation this way, leads us to a simplified form of the Pontryagin's minimum principle, well-known in control theory Astrom (1983), and the effective related solution is derived from the normal equations of the proposed criterion, namely:

$$\left\{\begin{array}{rcll} \nabla_{\mu_k}\mathcal{C}_v = 0 \Rightarrow \mathbf{V}[k] & = & \mathcal{F}(\mathbf{V}[k-1]) & \text{forward simulation} \\ \nabla_{\mathbf{V}_k}\mathcal{C}_v = 0 \Rightarrow \quad \mu_k & = & \nabla_{\mathbf{V}}\mathcal{F}(\mathbf{V}[k])\ \left(\mu_{k+1} - \nabla_{\mathbf{V}} d_k({}_o\bar{\mathbf{V}}[k], {}_o\mathbf{V}[k])\right) & \text{backward tuning} \\ \dot{\mathbf{W}} \equiv -\nabla_{\mathbf{W}}\mathcal{C}_v^T & = & \sum_k \nabla_{\mathbf{W}}\mathcal{F}(\mathbf{V}[k-1])\ \left(\mu_k - \nabla_{\mathbf{V}} d_k({}_o\bar{\mathbf{V}}[k], {}_o\mathbf{V}[k])\right) & \text{weights adjustment} \end{array}\right. \tag{3.18}$$

The 1st *forward simulation* equation, simply states that as soon as we modified the weights, the whole output has to be recalculated from the beginning to the end, which is a natural requirement since weights influence state values at each time. Therefore, an algorithm that adjusts recurrent weights must naturally adjust both weights and state values in a coordinate way. In other words, it allows to implement the fact that adjusting the system parameters at time k is interdependent with the state a time *before* k. Here the induced paradigm is a relaxation scheme in which we alternatively adjust the weights and re-simulate the obtained network.

The 2nd *backward tuning* recurrent equation, allows the algorithm to also take into account the fact that adjusting the system parameters at time k is interdependent with the state a time *after* k. The very interesting point, offered by the variational theory, is that this backward adjustment has not to be implemented by solving explicitly the system update inverse function (likely intractable), but by introducing *dual* parameters μ. A closed-formed solution is available when computed from $k = T$ backward to $k = 0$, noticing that we have to write $\mu_{T+1} = 0$ for the backward tuning equation to be well-defined at $k = T$. The algorithmic complexity of these additional $N \times T$ steps has again the same order of magnitude as the forward simulation steps, the

overload[4] being the calculation of the system state gradient $\nabla_{\mathbf{V}}\mathcal{F}(\mathbf{V}[k])$.

The 3rd *weights adjustment* equation implements the usual minimization gradient scheme of such a non-linear criterion, with the expected corrective term μ_k, taking into account the interdependency, as discussed previously. Behind the $\dot{\mathbf{W}} \equiv -\nabla_{\mathbf{W}}\mathcal{C}_v^T$ equation, we quote usual non-linear standard minimization methods[5]. This corresponds to the usual "weights learning rule" applied on the learning set, in such kind of framework.

The key feature of the present specification is that we solve the "recurrent weight problem" without any assumption on the network connectivity. It could be fully connected, or constrained by some specific topology, or it could be structured in some dedicated architecture with a "reservoir" or some "hidden/deep" layers, as briefly reviewed previously. The rough message is that it is not "that costly" to solve the recurrent weight problem in its whole generality, providing we are in this deterministic time-discretized context.

Following this track, a very interesting perspective of the present work would be also to adjust the connectivity itself, i.e. to minimize the metric not only with respect to the weights values, but also with respect to the fact that some weights have either zero or non-zero values, i,e, with respect connection sets. Sparse estimation methods (see e.g. Tropp (2004b,a) for a didactic introduction) can be used to this end.

The second key feature of the present specification is that we not only solve the recurrent weight problem, but also the "hidden state values problem". Since hidden state values are generated by the hidden layers recurrent weights, we obtain for the same algorithmic cost the optimal hidden state values. The second rough message is that it is not "that costly" to add hidden units in a recurrent weight system when solving the recurrent weights problem in its whole generality. In fact, an output is hidden as soon as its weighting in the metric is zero.

Obviously, this method is neither a panacea, nor a revolutionary method. Not revolutionary but simply the suggestion to apply a well-known algorith-

[4] A careful look at the backward tuning and weights adjustment reveals that the former can also be calculated backward (since a simple summation), thus factorized with the former calculation, allowing $\nabla_{\mathbf{V}} d_k({}_o\bar{\mathbf{V}}[k], {}_o\mathbf{V}[k])$ to be calculated once, thus only temporary stored. The memory cost has the same order of magnitude, since μ has the dimension of $\mathbf{V}$ while it has to stored only temporarily.

[5] Considering a simple gradient scheme, there is always a ϵ small enough and to be adjusted numerically so that $\dot{\mathbf{W}} = -\epsilon\nabla_{\mathbf{W}}\mathcal{C}_v^T$ decreases the criterion. In the experimental section of this work we use the GSL http://www.gnu.org/software/gsl multi-dimensional minimization algorithms taking the criterion derivatives into account, mainly the Fletcher-Reeves conjugate gradient algorithm. Other methods such as the Polak-Ribiere conjugate gradient algorithm, and the Broyden-Fletcher-Goldfarb-Shannon quasi-Newton are alternatives to be considered, depending on the application. A simple wrapper to these method bundle is available at http://enas.gforge.inria.fr/classIterativeSolver.html.

mic scheme of the control theory to this context, things becoming tractable in the deterministic discrete case. Not a panacea since the induced scenario is not causal: the weights learning rule is defined as soon as the system has been both simulated and then backward tuned, thus use information coming from the future. It is thus not applicable as a model of on-line learning mechanism. It is however quite useful in both off-line learning paradigms, and might have also some representative power, when considering biological mechanisms where mental states simulate a future action in order to adapt a given behavior.

At the modeling level, the contrapositive consequence, may be also interesting as a "negative result". The variational specification (3.17) and its solution (3.18) seem canonical, since simply stating how to find the best weights (i.e. that minimize a metric) in this general case. Therefore, it shows that solving the recurrent weights problem at a general level is a simple but irreducible non-causal forward/backward simulation/tuning scheme. On the contrary, we can expect causal schemes or simple back-propagation to never solve the problem at a general level. This highlight why people usually attack the recurrent weight problem in suitable architectures only.

Obviously, this method being non-linear, is highly dependent on the initial parameter and variables values, as discussed in details now.

Initial weights and states values adjustment.

We have noticed, that the induced paradigm is a relaxation scheme in which we alternatively adjust the weights and re-simulate the obtained network. This means that we do not, strictly speaking, "estimate" the weight values whereas we only "optimize" the current weight values estimate. Furthermore, even if the mollified metric is convex $d()$, the criterion (3.17) is not convex unless the $\mathcal{F}()$ would have been convex, which is obviously not expected for usual dynamical system.

It is thus a crucial issue to be able to properly initialize these values. Since we target output values to be as close as possible to the expected values, it is obvious to take as initial output values the expected values, i.e. assume ${}_o\mathbf{V}[k] \simeq \mathcal{F}({}_o\bar{\mathbf{V}}[k-1])$, yielding an initial scheme of the form:

$$\begin{array}{rcl} \dot{\mathbf{W}} & \equiv & -\nabla_{\mathbf{W}}\mathcal{C}_v^T \\ & \simeq & -\sum_k \nabla_{\mathbf{W}}\mathcal{F}({}_h\mathbf{V}[k-1], {}_o\bar{\mathbf{V}}[k-1])\nabla_{\mathbf{V}} d_k({}_o\bar{\mathbf{V}}[k], \mathcal{F}(\bar{\mathbf{V}}[k-1])). \end{array} \tag{3.19}$$

With this modification of the adjustment scheme, we do not have to calculate the output values in parallel with the weights since we simply assume that they are "perfect", i.e., correspond to the expected values, i.e., syntactically replace ${}_o\mathbf{V}$ by ${}_o\bar{\mathbf{V}}$ when needed. Furthermore, in (refr-criterion), we can state $\mu_k = 0$, since we replace the recurrence equation constraint by the

exact value. This also means that we can minimize the weights of each unit independently, since there output $\mathbf{V}_o$ is no more coupled, whereas replaced by the expected value $\bar{\mathbf{V}}_o$, reducing the algorithmic complexity by a factor of N^o, during this initial phase.

This is a natural way, to 1st estimate the weights without bothering with the fact that the effective output are not the expected by the obtained output, and its likely going to provide a reasonable estimate of the recurrent weights if the final solution is close to the expected solution. It, in fact, provides the exact weights (i.e. with the expected values equal to the obtained values), if any.

However, this specification does not define anything regarding the hidden initial state values. The value ${}_h\mathbf{V}[k-1]$ is still there without any tool to provide a reasonable initial value. We thus must introduce some knowledge outside the present framework in order to initialize these values. The known heuristics are to start with a sparse random hidden activity, i.e. with ${}_h\mathbf{V}$ drawn from any suitable probability distribution, either as random as possible (i.e. with a maximal entropy, e.g., values uniformly drawn and spikes drawn from a balanced Bernoulli distribution) or sparse (minimizing some energy in addition to randomness), and then optionally adjust values using unsupervised learning mechanisms, as reviewed previously for reservoir computing methods.

Another track is to start *not* using hidden state values in the very first initial state, and consider the network connection weights related to the hidden units as zero. Then we can incrementally introduce hidden units and optimize the related weights to optimize their values, but now in a context where other weights have received relevant approximate values. This is also an interesting heuristic with respect to the fact that it allows to be parsimonious with respect to hidden units, thus search solutions with a small, when not minimal, number of hidden units. An exact solution is available in the case of exact input/output function estimation Rostro-Gonzalez et al. (2010). If the reservoir of hidden units has no special connectivity, then the order of introduction of the hidden units is irrelevant. Otherwise heuristics dedicated to such a choice have to be worked out, allowing the algorithm to add units related to the metric terms with a large errors seems relevant, which is an issue beyond the present development.

Finally, despite the fact that we simply consider $V_o[k] = 0$, for $0 \leq k < D^o$, i.e., consider that the trajectory is not defined by its temporal initial conditions, but by the inputs, we also could wonder if this formalism addresses the issue of initial stage adjustment. The answer seems negative. For the simple reason that, if the initial stage values are not known, in (3.18), the equation $\nabla_{\mu_k} \mathcal{C}_v = 0$ does not correspond anymore to a forward simulation (since we

known the initial stage and can then calculate the next stage, etc..) but to a strong, likely intractable, set of under-determined non-linear equations. The choice of choosing a known (e.g., 0) initial stage is thus crucial in this context.

3.5 Application to resetting non-linear (RNL) networks.

Let us to consider 3.1, where model equation corresponds to the main classes of artificial or biologically plausible models. Also, this equation allows to combine analog and spiking mechanisms. In this sense, models at the mesoscopic scale, consider both, unit's firing rates as analog signals and synchronizations effects represented by reset mechanisms. It has been shown elsewhere Cessac and Viéville (2008) that such models correctly represent time discretized versions of generalized Integrate and Fire (gIF) spiking neural models Rudolph and Destexhe (2006).

In both cases, relative refractoriness is defined by weights $W''_{ood} < 0, d \in \{1, D^o\}$ so that as soon as a unit has a high activity, its state is inhibited for a certain amount of time, as in spike response models, see Gerstner and Kistler (2002a) for a general introduction. In order to introduce a absolute refractoriness of one sampling period (3.1) has to be modified, writing $U'''_o[k] \stackrel{\text{def}}{=} \rho_{ok}(V_o[k]) \in \{0, 1\}$ in factor of all the right-hand side equation. With this trick a reset induces $V_o[k]$ for one sampling period.

Regarding the initial conditions, we set $U''_o[k] = U'_i[k] = 0, k < 0$ and, as in the general case, choose $V_o[0] = 0$ and $U''_o[k] = 0, k \in \{0, D^o\{$, thus a zero initial state. This entirely defines, for a given input, the deterministic system state, from (3.1). Given any other initial condition $V_o[k], k \in \{0, D^o\{$, from (3.1) we always can add N^o inputs $U'_i[k] = 0, i \in \{0, N^o\{, k \in \{0, D^o\{$, with unit weights $W'_{oid} = \delta_{oi}\,\delta_{d1}$ such that:

$U'_i[k] = V_o[k] - I_{ok} + \gamma_{ok}\,V_o[k-1]\,U'''_o[k-1] - \sum_{j=1}^{N^o}\sum_{d=1}^{D^o} W''_{ojd}\,U''_j[k-d]$

for $k \in \{0, D^o\{$ and $U'_i[k] = 0$ otherwise. We may also use existing inputs, if $N^i \geq N^o$, to introduce values that balance any initial conditions. This makes explicit the fact that there is no loss of generality to consider a zero initial state in this context.

As discussed in the general case, the variation with respect to the last reset time τ_{ok} is obviously not regular and must be understood in the distribution sense. The purpose of the previous framework is precisely to by-pass this problem, proposing a mollification of the non-regular input/output system update equation (3.1), when a discontinuous reset mechanism is used.

Here we proposed to use the function series, for $\varepsilon \in]0,1]$:

$$\rho_\varepsilon(v) \stackrel{\text{def}}{=} 1 - \sigma_\varepsilon(v) \tag{3.20}$$

where:

$$\sigma_\varepsilon(v) \stackrel{\text{def}}{=} \xi\left(\frac{v-\theta}{\varepsilon}\right)$$

and

$$\xi(u) = \begin{cases} 0 & \text{if } u \leq -1 \\ 1 & \text{if } u \geq 1 \\ \frac{1}{2} + \frac{15}{16}\left(x - \frac{2}{3}x^3 + \frac{1}{5}x^5\right) & \text{otherwise} \end{cases}$$

as shown in Fig. 3.4. The reset function ρ_ε is not a usual sigmoid profile but has the following key feature: if $V_o[k] \geq \theta - \varepsilon \Rightarrow \rho_\varepsilon(V_o[k]) = 0$, so that a true reset occurs. A sigmoid profile, e.g. $\chi(u) = (1 + exp(-x))^{-1}$, would have maintained $0 < \rho_\varepsilon(V_o[k])$, i.e. with the spurious effect to have $\tau_{ok} = 0$, thus without a true reset, thus likely with a dynamics having very different qualitative properties Cessac (2008). This is avoided here. Similarly the non-linear activation function σ_ε is designed to induce no effect if the state value is below $\theta - \varepsilon$. The profile has been chosen as the simplest polynomial profile[6] which is twice differentiable (as required by minimization algorithms) and with a sigmoid shape of bounded support. In both cases $\lim_{\varepsilon\to 0}\rho_\varepsilon = \xi_{[-\infty,\theta[}$, $\lim_{\varepsilon\to 0}\sigma_\varepsilon = \xi_{[\theta,+\infty[}$, as expected.

3.5.1 Computational properties.

A straightforward derivation, from (3.1), yields:

$$V_o[k] \;=\; I_{ok}^{\tau_{ok}} + \sum_{j=1}^{N^o}\sum_{d=1}^{D^o} W''_{ojd}\,U''_{ojkd} + \sum_{i=1}^{N^i}\sum_{d=1}^{D^i} W'_{oid}\,U'_{oikd} \tag{3.21}$$

using the following notations:

- $U''_{ojkd} = \sum_{\tau=0}^{\tau_{ok}} \gamma_{ok}^{\tau}\, U''_j[k-d-\tau]$, thus $U''_{ojkd} \geq 0$, since $\sigma() \geq 0$ and $\mathbf{U} \geq 0$.
- $U'_{oikd} = \sum_{\tau=0}^{\tau_{ok}} \gamma_{ok}^{\tau}\, U'_j[k-d-\tau]$, with the same remarks.
- The 1st reset time before k, reads[7]: $\tau_{ok} \stackrel{\text{def}}{=} k - \arg\min_{l>0}\{U'''_o[k-l] = 0\}$. and:
- $\gamma_{ok}^{\tau} \stackrel{\text{def}}{=} \prod_{k=0}^{\tau-1}\gamma_{ok}$ (with $\gamma_{ok}^{\tau} = (\bar{\gamma})^{\tau}$ if the $\gamma_{ok} = \bar{\gamma}$ are constant, while $\gamma_{ok}^{\tau} = \xi_{\{1\}}(\tau)$ if the $\gamma_{ok} = 0$ vanish)

[6] Up to our best knowledge, it is not possible to build an indefinitely differentiable left and right bounded support *sigmoid* profile, though it is possible to build such a indefinitely differentiable left and right bounded support *bell* profile (e.g. $b(u) = H(1-u^2)\,exp(-u^2/(1-u^2))$, with $b(0) = 1$) corresponding to the bounded form of the usual Gaussian profile. The reader is invited to take up the bet.

[7] Thanks to the chosen initial conditions, this notation includes the fact that $\tau_{ok} = k$ in the case where the unit never reset.

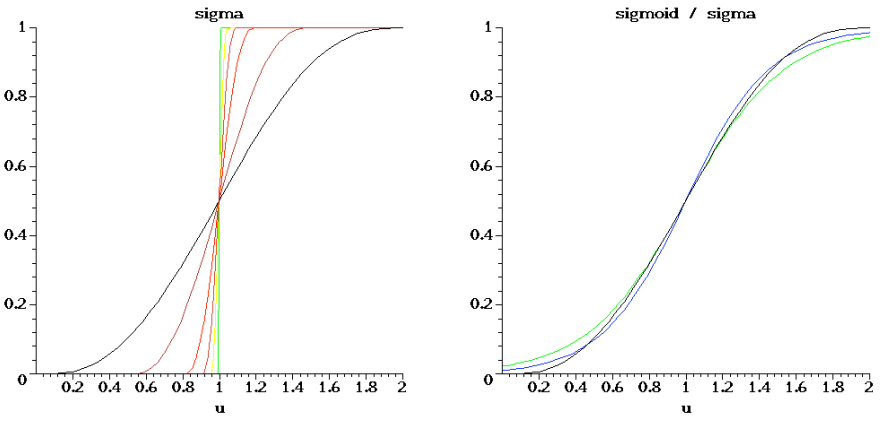


Figure 3.4: Defining the mollification of the reset function $\rho_\varepsilon = 1 - \sigma_\varepsilon$, and the activation function σ_ε (left view), for $\theta = 1$. It is drawn here in $black$, $brown$, red, $orange$, $yellow$, $green$, for $\varepsilon = [1, 0.5, 0.2, 0.1, 0.05, 0.01]$, respectively. Comparing the bounded profile (drawn in $black$ in the right view) for $\theta = \varepsilon = 1$ with a sigmoid profile of the same maximal slope (in $green$) or the same surface (in $blue$), showing that profiles are very similar though the former has a bounded support.

- $I_{ok}^{\tau} \stackrel{\text{def}}{=} \sum_{\tau=0}^{\tau_{ok}} \gamma_{ok}^{\tau} \, I_{o(k-\tau)}$ (defined by the systems constants γ_{ok} and I_{ok}).

This allows to make the following obvious but critical observation: *given some desired input U_i and related output V_o (which also define the recurrent variables), the programming problem involves linear constraints with respect to the weights parameters to adjust*.

Based on this, it has been made explicit in Rostro-Gonzalez et al. (2010) that the parameter estimation of a neural network in order to generate a given spike train, is a Linear (L) problem if the membrane potentials are observed and a Linear Progamming (LP) problem if only spike times are observed, with a gIF model. Such L or LP adjustment mechanisms are distributed and have the same structure as an "Hebbian" rule. A step further, this paradigm is easily generalizable to the design of input-output spike train transformations. This means that a practical method is available to "program" a spiking network, i.e. to find a set of parameters allowing us to exactly reproduce the network output, given an input.

The fact that there is an efficient LP algorithm that exactly matches input/output is crucial in robust learning schemes as pointed out in section 3.3.

In this linear framework, input/output relations can be specified, by constraining the output, with a rich semantic. Imposing inequalities means:

- stating the spiking state corresponding to inequalities of the form $\theta \leq$

$V_o[k] \leq V_{\max}$,

- stating the non-spiking state corresponding to inequalities of the form $V_{\min} \leq V_o[k] \leq \theta - \nu$,
  where $V_{\min}$ and $V_{\max}$ are the absolute minimal, maximal values, while ν is an infinitesimal value[8];

- stating the output analog values range, i.e. inequalities of the form $V_o^{equ}[k] - V_o^{\nu}[k] \leq V_o[k] \leq V_o^{equ}[k] + V_o^{\nu}[k]$, for desired values $V_o^{equ}[k]$ up to precisions $V_o^{\nu}[k]$.

Structural constraints on the weights $\mathbf{W}$ allows us to specify connectivity constraints:

- for *topographical connectivity*, when no connection between units of index j and o, we get $\forall d > 0, W_{ojd} = 0$,

- for *excitatory / inhibitory connections* $\forall d > 0, 0 \leq W_{ojd}$ / $\forall d > 0, W_{ojd} \leq 0$.

Furthermore, weight profile constraints take into account the fact that *weight profiles are "smooth"*, e.g.: that the weight's temporal variation is bounded by ΔW, $\forall d > 1, |W_{ojd} - W_{oj(d-1)}| \leq \Delta W$.

All together we can specify a rather large amount of semantic elements within this framework, including the major biologically plausible constraints taken into account in such a context.

A step further, we can rewrite (3.1) as:

$$V_o[k] = I_{ok}^{\tau} + \sum_{j=1}^{N^o} \sum_{s=1}^{D^o+\tau_{ok}} Y''_{okjs}\, \sigma_{ok}(V_j[k-s]) + \sum_{i=1}^{N^i} \sum_{s=1}^{D^i+\tau_{ok}} Y'_{okis}\, U'_i[k-s]$$
$$\left\{ \begin{array}{rcl} Y''_{okjs} & \stackrel{\text{def}}{=} & \sum_{d=\max(s-\tau_{ok},1)}^{\min(s,D^o)} \left[\prod_{h=k-d}^{k-1} \gamma_{oh}\right] W''_{ojd} \\ Y'_{okis} & \stackrel{\text{def}}{=} & \sum_{d=\max(s-\tau_{ok},1)}^{\min(s,D^i)} \left[\prod_{h=k-d}^{k-1} \gamma_{oh}\right] W'_{oid} \end{array} \right. \tag{3.22}$$

making explicit the auto-regressive / moving-average form of this recurrent equation.

In the case where the reset mechanism $U_o[k] = 1$ is not used, while the sigmoid profile $\sigma(g_{ok}\, u) \stackrel{\text{def}}{=} g_{ok}\, u + o_{ok}$ is a linear/affine equation, this corresponds to a standard so-called trivial linear ARMA mechanism. This case is a singular case with respect to the computational properties as made explicit now.

The serial combination of RNL networks (i.e. one network input being the network output of the other) or their parallel concatenation (i.e. the concatenation of their state) are RNL networks. Unless in the linear ARMA case

[8] In fact, at a theoretic level, the spiking / no-spiking state is simply specified by $\theta \leq V_o[k]$ / $V_o[k] < \theta$. Making explicit additional "technical" bounds $V_{\min}$ and $V_{\max}$ and transforming the strict equality in a weak equality up to some infinitesimal value ν is required by the numeric implementation discussed in the sequel.

these are the two possible combinations of RNL networks. Furthermore, RNL networks are irreducible up to a scale factor, and universal approximators, as it is the case for less restrictive artificial neural networks Hornik et al. (1989); Schäfer and Zimmermann (2006). More concretely, there is always a RNL network that exactly matches, in the general case, a finite input/output transformation, all Boolean input/output causal functions can be implemented with a delay of two samplings, and for a given finite spike pattern, there is always a RNL unit that separates this pattern among all others.

In any case, equation (3.22) corresponds to the instantiation of $\mathcal{F}_{\mathbf{W}'',\mathbf{W}'}$ in (3.2) and allows to instantiate the general framework developed here for a concrete class of models as numerically experimented now. It is straigtforward to integrate this model in the previous framework since, $\nabla_{W'_{oid}} V_o[k] = U'_{oikd}$ and $\nabla_{W''_{ojd}} V_o[k] = U''_{ojkd}$, while $\nabla_{V_j[k-s]} V_o[k] = Y''_{okjs}\, \sigma'_{ok}(V_j[k-s])$.

3.6 PRELIMINARY CONCLUSION

In this chapter we have defined a theoretical framework, which address the aspects of the computational capabilities of the spiking neuron models. On one hand we review on the spiking metrics, where we have evidenced the lack of information with respect to each performed operation. In this sense we propose a mollification in the spiking metrics permitting one to make explicit each operation. On the other hand we show how this mollification can be used in learning mechanism in order to implement input-output transformations in a general case.

A current on-going work based on this prospective analysis targets the implementation and numerical validation of these general principles.

Part IV

Hardware Implementations of gIF-type Spiking Neural Networks

CHAPTER 4

HARDWARE IMPLEMENTATIONS OF gIF-TYPE NEURAL NETWORKS

"Things should be made as simple as possible, but not any simpler."
–Albert Einstein

OVERVIEW

We know that spiking neuron models can perform very powerful computations. Moreover, the equations that describe the behavior of these models have inspired scientist and engineers, especially in the field of electronics, to use them in order to develop bio-inspired hardware implementations. The development of dedicated hardware permits us to accelerate the processing, since operations are carried out directly in the related device. However, even if the technology advances rapidly, this approach presents some difficulties: the area-greedy operators and the complex topologies are involved in a neural network.

In this sense the aim of this chapter is to study the feasibility of implementing the gIF-type neuron models on dedicated hardware, such as FPGA (*Field Programmable Gate Array*). The study is based on a precision analysis, where a fixed-point arithmetic is considered. More specifically, this analysis permits us to determine the best data representation to map our models onto a FPGA. Furthermore, we consider this representation as a perturbed signal (described in chapter 1) producing an approximation of the real trajectory and we evaluate its behavior on periodic and chaotic dynamics. The results evidence that the generalization in the data representation can induce drastic effects on the desired activity, especially for complex dynamics.

The FPGA-based architectures have been coded using the Handel-C and VHDL (VHSIC hardware description language; VHSIC: very-high-speed in-

tegrated circuit) hardware description languages. These architectures have a modular and reconfigurable scheme, which enables one to evolve in larger of more complex systems.

The VHDL designs have been synthesized for a Virtex-II Pro FPGA in order to evaluate the resources consuming. However, the designs are generic, which means that them can be mapped on updated FPGA devices.

Finally, we have developed GPU-based kernels in order to evaluate the performance of our FPGA-based architectures. The GPU (*Graphing Processing Unit*) technology uses parallel computing to perform complex tasks, since the hard processing is carried out directly on the GPU. The same gIF-type neuron models have been simulated on a GPU and further compared with the FPGA designs in terms of speed. At the present stage, however they can not be directly compared in terms of the maximal number of neurons that support each one, where the GPU is clearly superior. The GPU-kernel has been coded using a CUDA-C++ heterogeneous programming scheme.

The chapter is organized as follows. First we describe the hardware and programming languages in sections 4.1 and 4.2 respectively. Then in 4.3 we define a representation data based in the fixed-point arithmetic in order to make possible the implementations on hardware of these models. Finally in 4.4 and 4.5 the hardware implementations of spiking neural networks are described.

4.1 Hardware Description

In this work we are specially interested in low-cost hardware in order to perform spiking neural network implementations. For this reason we have chosen two low-cost technologies FPGA and GPU. They focus in the parallel processing, which solves much faster any task ever that it has been well-designed.

4.1.1 FPGA (Field-Programmable Gate Array)

A FPGA (Figure 4.1) is a reconfigurable integrated circuit. It has a dual nature combining the flexibility of software with the performance of hardware. The FPGA has been designed to be configured by the customer. They implement circuits, providing huge power, area, and performance benefits over software, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Sometimes reprogramming is merely a bug fix to correct faulty behavior, or it is used to add a new feature. Some times, it may be carried out to reconfigure a generic computation engine for a new task, or even to reconfigure a device during operation to allow a single piece

of silicon to simultaneously do the work of numerous special-purpose chips. FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. The FPGA configuration is generally specified using a hardware description language (HDL).

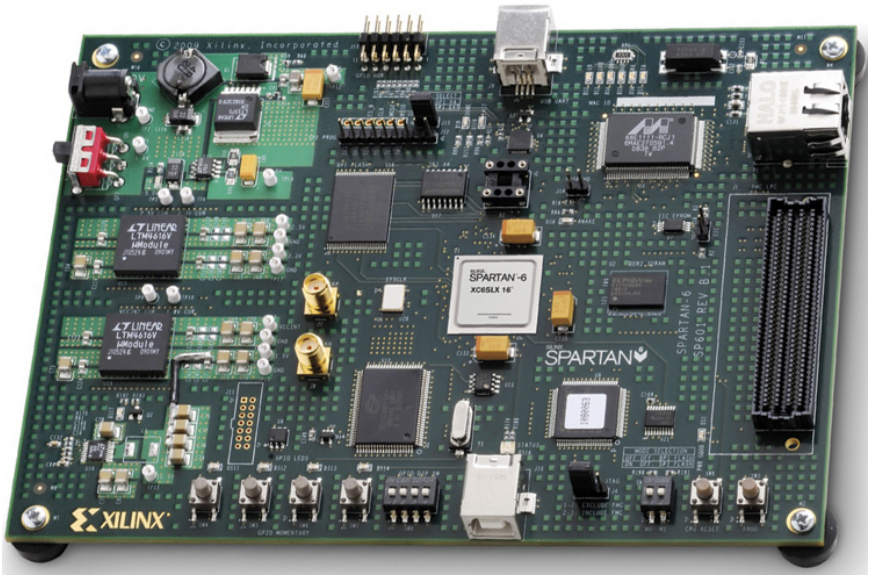


Figure 4.1: A FPGA xilinx board (Illustration from Xilinx throughout wikipedia)

FPGAs contain programmable logic components called “logic blocks”, and a hierarchy of reconfigurable interconnections that allow the blocks to be "wired together" somewhat like a one-chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory

4.1.2 GPU (Graphics Processing Unit)

In recent years researchers have found in GPU’s the alternative to run their computational models in real time. GPU has evolved into a highly parallel, multithread, multi-core processor with tremendous power and very high memory bandwidth (Figure 4.2). Modern GPU’s are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. In a personal computer, a GPU can be present on a video card, or it can be on the motherboard.

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation, exactly what graphics rendering is about and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations. The same program is executed on many data elements in parallel with high arithmetic intensity.

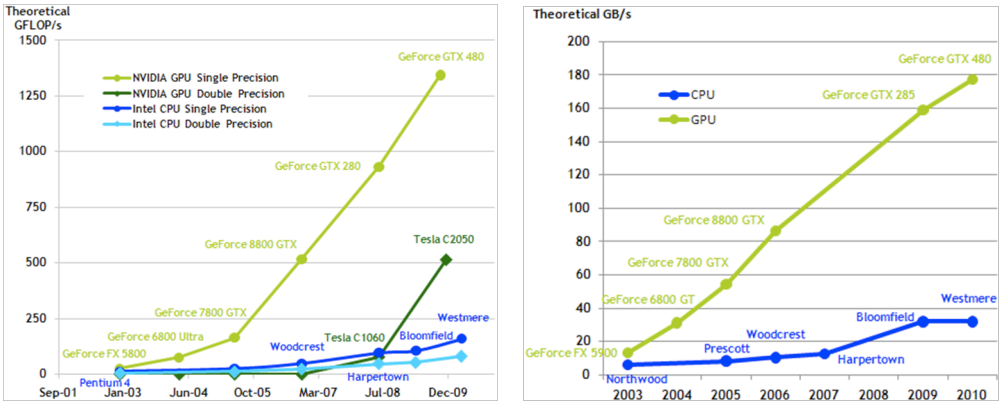


Figure 4.2: Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU (Illustration from Kirk and Hwu (2003))

4.2 Programming Languages

Behind an efficient hardware architecture exists an algorithm coded under a powerful programming tool. In this sense we consider mainly two important programming languages such as VHDL for a FPGA-based implementation and CUDA for a GPU-based implementation. We also consider another two support languages such as Handel-C and C++. Handel-C is used to test the models before to be implemented under VHDL. This language is similar to C, which makes possible a fast coding for a hardware implementation.

4.2.1 Handel-C

Handel-C is a high level programming language which targets low-level hardware, most commonly used in the programming of FPGAs. It is a rich subset of C, with non-standard extensions to control hardware instantiation with an emphasis on parallelism. Handel-C is to hardware design what the first high level programming languages were to programming CPUs. Unlike

many other design languages that target a specific architecture Handel-C can be compiled to a number of design languages and then synthesised to the corresponding hardware.

4.2.2 VHDL (VHSIC hardware description language; VHSIC: very-high-speed integrated circuit)

VHDL is a strongly typed, Ada[1]-based programming language that includes special constructs and semantics for describing concurrency at the hardware level. Programming in VHDL is quite different from programming in C because of its concurrent semantics. However, it does have several similarities with object oriented languages i.e. C++ and Java. It should help the reader to understand the basic structure of the language and its relation with hardware-specific VHDL constructs.

Structural Description (From Hauck and Dehon (2008))

1. VHDL files typically start by including the IEEE library and certain important packages like `std_logic_1164` that enable us the use of type `std_logic` and Boolean operations on it. Additional packages such as `std_logic_arith` and `std_logic_unsigned` are also included for supporting arithmetic operations.

2. The VHDL description of a hardware module requires an `entity` declaration that specifies the interface of the module with the outside world. It is an enumeration of the interface ports. The declaration also provides additional information about the ports such as their direction (in/out), data type, bit width, and endianness. An entity declaration in VHDL is analogous to an interface definition in Java or a function header declaration in C.

3. Almost all VHDL signals and ports use the data type `std_logic` and `std_logic_vector`. These data types define how VHDL models electrical behavior of signals. The vector `std_logic_vector` allows declaration of buses that are bundled together.

4. While an entity specifies the interface of a hardware module, its internal structure and function are enclosed within the `architecture` definition.

5. In a structural description of a module, the constituent submodules are declared, instantiated, and connected to each other. Each submodule

[1]Ada is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages.

needs to be first declared in the `component` declaration. This is merely a copy of the entity declaration where only the submodule interface is specified. The instantiated components are connected to each other via internal signals by a process called *port mapping*. Port mapping is performed on a signal by signal basis using the $\Rightarrow$ symbol.

6. Each component can be reused for. This is one of the benefits of a structural representation, it permits reuse of existing code for recurring design elements and helps reduce total code size.

Limitations

- VHDL syntax is verbose, extremely heavy, and requires several lines of code to describe even simple logic elements.
- Hardware needs to be described at a very low level of abstraction.
- As technology and FPGA architectures evolve, the optimal amount of pipelining required to meet the desired cycle time changes.
- Low-level descriptions also make it hard for synthesis tools to optimize and schedule logic.
- Hardware described in VHDL suffers from the additional drawback of significantly long verification times.

4.2.3 CUDA (Compute Unified Device Architecture)

CUDA is general purpose parallel computing architecture, introduced in November 2006 by NVIDIA. CUDA use a new parallel programming model and instructions set architecture that provides the parallel compute engine in GPUs to solve complex computational problems in a more efficient way than on a CPU.

The advent of multicore CPUs and GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law[2]. The challenge is to develop application software that transparently scales its parallelism to suply the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to GPUs with widely varying numbers of cores.

The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

[2]Moore's law describes a long-term trend in the history of computing hardware. The number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years

Programming Model for GPU-based implementations

The programming model for GPU-based applications is defined from a heterogeneous programming scheme. This is divided in two parts, on one hand the application uses an interface between the CPU and the GPU, usually coded in a high-level programming language such as C/C++, python, etc. On the other hand a piece of code defined as a *kernel*, which will be placed to run directly in the GPU. In order to identify which process will be executed on the CPU and which one on the GPU, there exist two qualifiers, *host* and *device*.

The `__host__` qualifier declares a function that is executed and callable only by the host (CPU).

The `__device__` qualifier declares a function that is executed and callable only by the device (GPU).

However, the `__host__` qualifier can also be used in combination with the `__device__` qualifier, in which case the function is compiled for both the host and the device. Further there exist the *global* qualifier, which makes the interface between the CPU and the GPU.

The `__global__` qualifier declares a function as being a kernel that is executed on the device and callable from the host only.

Kernels

CUDA extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different CUDA *threads*, as opposed to only once like regular C functions.

- A kernel is defined by a declaration specifier (`__global__`).
- The number of threads must be defined in the kernel call `new <<<....>>>`, where each thread could be identified in the kernel through the built-in *threadIdx* variable.

Limitations

- CUDA uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments. Fermi GPUs now have (nearly) full support of C++. Exceptions as follows:

- Code compiled for devices with compute capability 2.0 (Fermi) and greater may make use of C++ classes, as long as none of the member functions are virtual (this restriction will be removed in some future release).
- For double precision (only supported in newer GPUs like GTX 260) there are some deviations from the IEEE 754 standard: round-to-nearest-even is the only supported rounding mode for reciprocal, division, and square root. In single precision, denormals and signalling NaNs are not supported; only two IEEE rounding modes are supported (chop and round-to-nearest even), and those are specified on a per-instruction basis rather than in a control word; and the precision of division/square root is slightly lower than single precision.
- The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.
- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path.
- Unlike OpenCL, CUDA-enabled GPUs are only available from NVIDIA (GeForce 8 series and above, Quadro and Tesla).

4.3 FIXED-POINT ARITHMETIC FOR DATA REPRESENTATION

A representation data is the bridge between a mathematical model, here a spiking neuron model, and a design of reconfigurable architectures with high performance for FPGA-based implementations. This is because real values can not directly be handled by a FPGA, since it last is composed of logic blocks, which use binary logic to perform complex combinational functions. Particularly, we have chosen the fixed-point arithmetic due that consumes less area resources than floating-point arithmetic Parhami (2000). Indeed, this makes our circuitry smaller and faster than a representation in floating-point arithmetic. A value of a fixed-point data type is essentially an integer that is scaled by a specific factor determined by the type. This representation is commonly defined by two parts, integer and fractional. A fixed-point data writes:

$$\text{Fixed-point data (word size)} = \text{(int)}(\text{realvalue} * 2^{f}) \tag{4.1}$$

where f represents the number of fractional bits.

4.3.1 Resolution for the fixed-point data representation

The resolution for a fixed-point variable is set by the number of fractional bits (f) used in the fixed-point representation. The resolution ϵ is given by the next equation:

$$\epsilon = \frac{1}{2^f} \tag{4.2}$$

4.3.2 Range for the fixed-point data representation

An important point in the precision analysis is to verify that the word size will cover correctly the values that we want to represent. In order to know if we have chosen an adequate data representation we estimate the range of values that will cover our representation.

1. Considering only integer values the range is given by:

   $$\in [0, 2^b - 1]$$

   where b is the number of bits considered in the data representation.

2. Considering only positive values and a fixed-point representation $U(a, b)$, where a represents the integer part and b the fractional part, the range is given by:

   $$\in [0, 2^a - 2^{-b}]$$

3. Considering both, negative and positive values and a fixed-point representation $U(a, b)$, where a represents the integer part and b the fractional part, the range is given by:

   $$\in [-2^{M-1-b}, 2^{M-1-b} - 2^{-b}]$$

   where $M = a + b$

Since the precision analysis is carried out only on b, the value of a can be defined as follow:

$$a \geq \eta(\log_2(\text{real value} + 1)) \tag{4.3}$$

where η is a rounding variable.

4.3.3 A precision analysis of the gIF-type neuron models

The word-size of data (data representation) in the FPGA-design is an important consideration. As the number of bits being processed increases, the area of the resulting circuit and often the power consumption may increase,

at the same time that the speed in the device decreases. For this reason it is necessary that the designer calculates efficiently the number of bits required by the hardware to ensure that it can to perform certain task.

The number of bits for any algorithm can be calculated considering the range of the input data, output data and the operations involved. In this section we present a precision analysis in the design of the architecture for the gIF-type neuron models in order to calculate adequately the number of bits for its implementation.

The precision of a value describes the number of digits that are used to express that value, alternatively it describes the position at which an inexact result will be rounded. In this sense we make a precision analysis on the gIF-types neuron models. In both cases, analog and discrete we find that constant parameters (i.e. leak factor and the external current) and variables (i.e. membrane potential, synaptic weights (considering STDP, otherwise it is constant) and non-linear activation function σ in the analog neuron model) are described by real values. Further, these values are represented by a certain number of integer bits, which is an approximation of the real value. For this reason we make an analysis in terms of precision for each constant parameter and variable considered in the spiking neuron models.

- **The general constant parameters of a neural system** (N, T and D)

  A neural system is defined by two general parameters, N and T. On one hand the number of neurons N define the size of the network and the number of synaptic connections. On the other hand T that defines the simulation time. Also, in our models we consider the inter-neural transmission delays D. These parameters are defined by integer and non-negative values, for this reason they have a $U[a, 0]$ fixed-point representation as follow:

$$U_N[a, 0] \text{ with } a = \eta(log_2(N)) \tag{4.4}$$

$$U_T[a, 0] \text{ with } a = \eta(log_2(T)) \tag{4.5}$$

$$U_D[a, 0] \text{ with } a = \eta(log_2(D)) \tag{4.6}$$

- **The synaptic weights (W)**

  The values of **W** are chosen randomly from a normal distribution $\mathcal{N}(\mu, \sigma^2)$, where the variance $\sigma^2 = \frac{C}{\sqrt{N}}$ defines the activity regime (see chapter 1) of the neural network. The fact to consider a normalization

on the variance is in order to avoid cases where a simple synaptic connection can be able to produce a very strong effect on the post-synaptic neuron that makes fires the neuron or in the contrary case, where weak connections leave the neuron quiescent. The mean μ can be adjusted in order to specify the percentage of excitatory and inhibitory synaptic connections.

In a neural system, the synaptic weights are represented as a vector of $N \times N \times D$ values. Further from the viewpoint of the hardware design they can be contained in a RAM memory. Hence, each value is mapped onto the FPGA as a $U_W[a, b]$ fixed-point representation. In our neural network models we consider a fully-connected topology, it means that we have $N \times N$ connections, each modeled with D parameters.

The fixed-point data of **W** consider both, negative (inhibitory connections) and positive values (excitatory connections) and is given by

$$U_W[a, b] \tag{4.7}$$

where:

$$a = \eta(\log_2(\max(|W_{ijd} \in \mathcal{N}(\mu, \sigma^2)|) + 1))$$

and b depends on the desired precision. Further we analyze numerically the effects on the dynamics at different precision levels.

- **The leak rate (γ)**

  The leak factor γ is a constant value that solves the memory problem. Biologically, this term reflects the diffusion of ions that occurs through the membrane when some equilibrium is not reached in the cell (when the threshold has not been reached). So far, it takes into account the multiplicative effects of conductances and its value variates between 0 and 1. Moreover since γ never touches the 1 integer bits are not considered for its fixed-point representation, which writes:

$$U_\gamma[0, b] \tag{4.8}$$

- **The external current (I^{ext})**

  The external current represents an external stimulus coming from a sensorial source. In the present work we have considered it as a constant value between 0 and 1, since it is given by the next equation:

$$I^{\text{ext}} = \theta(1 - \gamma) \tag{4.9}$$

and its fixed-point representation writes:

$$U_I[0, b] \tag{4.10}$$

- **The firing state ($\mathbf{Z} = \rho(\mathbf{V})$)**

  As we described in chapter 1 the dynamics of a spiking neural network can be described by their firing state, which is commonly represented as a spike train (raster plot). Further this representation permit us to study the neural dynamics in terms of the information theory. In addition for a digital hardware implementation, this representation permits us to reduce considerably the device area consumption, since $\mathbf{Z}$ is a vector of $N \times T$ localities each represented by only 1 bit.

- **The non-linear activation function ($\sigma(V)$)**

  The gIF-type neuron model in its analog-spiking form and described by the equation 1.13 considers a non-linear activation function to perform the output of each synaptic connection in a neural network. In our case we consider the logistic sigmoid function (Figure 1.9). This form permits us to define later the same limits of $\mathbf{V}$ than in the discrete case, since its value variates between 0 and 1. Hence, the fixed-point representation writes:

$$U_\sigma[0, b] \tag{4.11}$$

- **The membrane potential (V)**

  As we described in chapter 1, the variable $\mathbf{V}$ represents the behavior of the spiking neural network, further this variable is described in two ways from its synaptic model. On one hand in equation 1.4 we have discrete-type synapses, which means that the synaptic connections depend on the firing state of the other cells. On the other hand in 1.13 we consider analog-type synapses, since them are evaluated by a sigmoid function.

  Now, we evaluate both cases for an efficient hardware implementation:

  - ***The discrete case***

    Most part of neural network models consider $\mathbf{V}$ as a vector of $N \times T$ real values, which contains the whole activity of the neural network. However, in our particular case, since there exist an one-to-one correspondence between the membrane potential and the spiking state, we only need to consider the last state of $\mathbf{V}$ as

equation 1.4 evidences. Further for a FPGA-based architecture we consider a vector $\mathbf{V}$ with N fixed-point values.

Let us to separate the equation 1.4 in three terms in order to determine the best data representation for V. We use V to define a single trajectory (the membrane potential on one neuron), further the data representation is generalized for all elements of the vector $\mathbf{V}$.

The reset mechanism is the part of the equation that evaluates the last state of the cell and is given by

$$V_i[k] = \gamma\, V_i[k-1]\,(1 - Z_i[k-1])$$

Since $(1 - Z_i[k-1])$ is a binary signal equal to 0 when $Z_i[k-1] = 1$ (which means that the neuron has fired and the membrane must be reseted) and 1 otherwise, thus a **preliminary** fixed-point representation for V is given by the product between its last state and γ. Thus, the representation is defined as follow:

1. We consider the fixed-point representations of γ and V, which have been defined as $U_\gamma(0, b)$ and $U_V(a, b)$ respectively.
2. Then, we perform the product using fixed-point arithmetic, hence we have a $U_{\gamma V}(a, b + b)$ representation.
3. Finally the $U_{\gamma V}(a, b+b)$ representation is truncated to $U_V(a, b)$.

The external stimulus can be computed at the same time that the reset mechanism in order to save one step as follow:

$$V_i[k] = \gamma\, V_i[k-1]\,(1 - Z_i[k-1]) + I_i^{\mathbf{ext}}$$

1. If $Z_i[k-1] = 1$ we reset V with $I_i^{\mathbf{ext}}$ instead of 0, where the $U_I(0, b)$ representation can be perfectly matched with the representation of V.
2. Otherwise the $I_i^{\mathbf{ext}}$ is added with the truncated value of the product between γ and V.

The synaptic activity determines the range of V and consequently its number of integer bits a in the fixed point representation. The synaptic weights have a $U_W(a, b)$ representation, where the number of fractional bits b is the same than V, but not a due that the number of integer bits in V needs to be larger in order to compute the whole synaptic activity, for this reason ever if all the synaptic weights are computed at the same time in the cell, the fixed-point representation in V must to be large enough to support it. Thus,

once that the state of the membrane has been evaluated and the external stimulus computed, the next is to add the synaptic with spiking activity in $k-d$, where $d=\{1...D\}$. From a computational viewpoint it is a sequential process, however from a hardware implementation viewpoint it is a combinational one as we probe below.

In summary the fixed-point representation of V depends on the activity of the whole neural network as we have seen. In this sense considering this analysis we can define the representation of V as follow:

$$U_V[a,b] \tag{4.12}$$

where from equations 1.9, 1.10 and 4.3 we define a range for a:

$$a \geq \eta(\log_2(\max(|V_{\min}|, V_{\max})+1)) \tag{4.13}$$

and b will depend on the neural regime as we show in the numerical results.

– ***The analog case***

The term analog refers to the kind of activation function that uses the gIF-type neuron model to compute the output inherent to each synaptic connection. In this sense the equation 1.13 describes the dynamics of a gIF-type neuron model with a non-linear activation function but with spiking activity. Its fixed-point representation is the same than that in the discrete case and given by equation 4.12. However this model needs to consider more than N values to estimate the whole neural activity due that we are considering delays ($\sigma(V_j[k-d])$). Thus, the vector $\mathbf{V}$ has $N \times D$ values with a fixed-point representation defined by equation 4.12.

4.4 FPGA-BASED ARCHITECTURES OF GIF-TYPE NEURON MODELS

The motivation to implement the proposed models on dedicated hardware, such as FPGA, has been inspired from two ideas. On one hand we consider that the gIF-type neuron models are highly feasible to large scale simulation due to the one-to-one correspondence between the membrane potential and the spiking activity. Further, the resulting spiking dynamics can be analyzed by statistical methods using the EnaS simulator[3].

[3] http://enas.gforge.inria.fr/

On the other hand we study the behavior of the FPGA-based implementations of gIF-type neuron models under different activity regimes, i.e., chaotic and periodic. The design of the FPGA-based architectures has a general and modular programming scheme that permits us to reconfigure the neural network and to evolve in a more complex systems. Also, the architectures use the previous precision analysis in order to have an efficient area distribution.

The FPGA-based architectures have been designed using the Handel-C and VHDL programming languages. The Handel-C design is not efficient compared with that in VHDL, since Handel-C code has an implicit sequential programming scheme. However we have designed it in order to perform the precision analysis and to calibrate the hardware architecture with the software parameters. Since one of the goals in this work is to analyze the behavior of the spiking neural network on hardware under different neural activity regimes, we need to consider the same parameter on both implementation, software and hardware. The Hande-C design has also permitted us to determine the best number of bits to represent our data in the FPGA device considering trivial and complex neural dynamics. Further the hardware design is implemented on VHDL in order to have a hardware architecture with high performance and best adapted to any target (FPGA device).

The next is to describe the characteristic of each design for the analog and discrete approaches (analog and discrete) of the gIF-type neuron models.

4.4.1 A FPGA-based architecture of gIF-type neural networks using Handel-C

The structure of a Handel-C code is very similar to that in C/C++ programming, which permits us to pass rapidly from a software design to a hardware design that eventually could be mapped on a FPGA device in a not efficient way. In this sense we explain now the structure of our design for the gIF-type neuron models.

The Handel-C algorithm

1. We write a C++ code that simulates a spiking neural network using either the discrete or the analog form.

    - We define the constant values of γ (float), I^{ext} (float), N (int), D (int) and T (int). Then we declare the vectors $\mathbf{V}$, $\mathbf{W}$ and $\mathbf{Z}$ ($\rho(\mathbf{V})$) using the GSL library[4]. GSL is a numerical and optimized library for C and C++ that allows one to perform operations with vector and matrices.

[4] http://www.gnu.org/software/gsl/

- We make a random initialization in $\mathbf{V}[k]$ and $\mathbf{Z}[k]$ ($\rho(\mathbf{V})$), where $k < D$. Also, we define the synaptic weights $\mathbf{W}$ using the EnaS library.
- We estimate $V_{\min}$ (Equation 1.9) and $V_{\max}$ (Equation 1.10) from the estimated values of $\mathbf{W}$.

* For the analog case we perform a C function, which computes the synaptic connection using a sigmoid function.

- We create text files containing each one a different fixed-point representation of $\mathbf{V}[k]$ and $\mathbf{W}$. Such representations are estimated using the equation 4.1.
- We perform the equations 1.4 and 1.13 using a `case` C statement.
- We compile and execute the C++ code. During the execution the text files that contain the fixed-point representations are sent to the Handel-C folder.

2. We write a Handel-C design using the same structure that the C++ code.

   - First we define the precision b that will be considered in the design.
   - We estimate γ and I^{ext} from equations 4.8 and 4.10 respectively.
   - We define the fixed-point representation of the counters for N (twice), D and T, respectively. Thus, the representations are given by the equations 4.4 (twice), 4.6 and 4.5.
   - Vectors $\mathbf{W}$ and $\mathbf{V}[k]$ are load from the text files generated in software. Further $\mathbf{Z}[k]$ is defined from $\mathbf{V}[k]$.

   * We also perform a function, which contains the sigmoid activation function to be used in the analog case.

   - Both gIF -type neuron models are implemented. However an initial signal indicates us, which synapses model will be used in the simulation.
   - We compile and execute the implementation using the Celoxica DK Design Suite.
   - The numerical results ($\mathbf{V}$ and $\mathbf{Z}$) are saved in a text file.
   - The algorithm is executed several times for different values of b and a different text file is generated for each value of b.

3. The resulting spiking dynamics in software and hardware are compared using the Victor-Purpura distance Victor and Purpura (1996), which has been implemented in EnaS. Also, we estimate the RMS error from their respective membrane potentials.

4. The steps 1, 2 and 3 are simulated for trivial and complex neural dynamics as was described in chapter 1.

5. The results on complex dynamics for both, software and hardware (at different fixed-point representations), are statistically analyzed using the Kullback-Leibler divergence[5] Cessac (2010). This function is available in EnaS.

6. We define the best fixed-point representation to design the VHDL architecture.

The implementation of the gIF-type neuron models on Handel-C and their respective version in software has permitted one to define the elements to design an efficient VHDL-based architecture that permits us to have a high performance on the FPGA-device.

4.4.2 A FPGA-based architecture of gIF-type neural networks using VHDL

We present a reconfigurable and efficient VHDL-based architecture of a gIF-type neural network, which is based on the precision analysis in 4.3 and the previous Handel-C design. This architecture permits us to switch between the analog and the discrete activation functions and is described by three main functional blocks: the first one defines the space of constants and variables. In other words the topology and parameters of the neural network are defined here. In the second one we define the gIF-type equations. The third one serves to route the resulting data to a RAM memory.

In figure 4.3 we present the general scheme of the FPGA-based architecture. The general operation begins with a signal sent from the control to the input module in order to initialize the parameters and to create the neural network. Once that the size of the network and its topology are defined the input module sends the respective parameters to each neuron. Then, from a combinational process the neurons compute the gIF-type equations and send their results to the output module, which can be either routed (only the spiking states) to the embedded RAM memory or presented to the real world.

[5]The Kullback-Leibler divergence Kullback and Leibler (1951); Kullback (1959) is a non-symmetric measure of the difference between two probability distributions P and Q. Thus this reads:

$$\begin{aligned} D_{KL}(P||Q) &= -\sum_x p(x)\log q(x) + \sum_x p(x)\log p(x) \\ &= \quad H(P,Q) \quad - \quad H(P) \end{aligned}$$

where $H(P,Q)$ is called the cross entropy of P and Q, and $H(P)$ is the entropy of P.

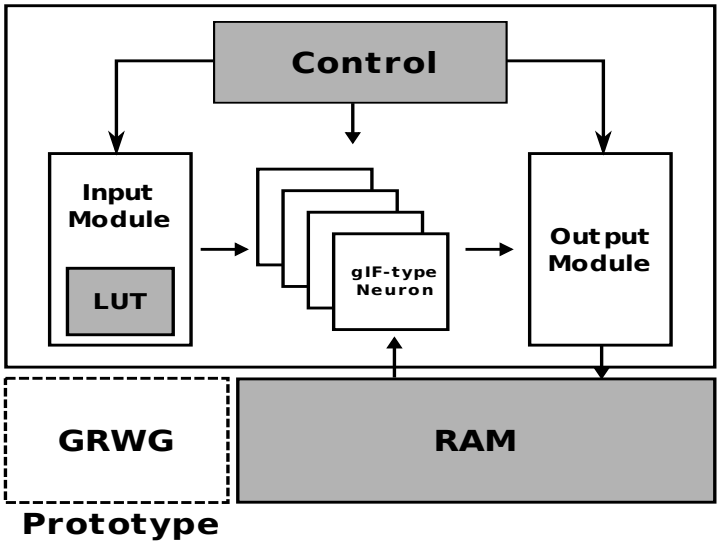


Figure 4.3: A FPGA-based architecture.

Data types package

In VHDL we have implemented a `package` with our own data types in order to reutilize them in the main modules. It is called from the other modules as follow:

```
use work.my_data_types.all
```

and contains the next data types:

```
constant Neurons : integer := N;
constant Time : integer := T;
constant Delays : integer := D;
type Nbits_Leak is range (b - 1) downto 0;
type Nbits_Current is range (b - 1) downto 0;
type Nbits_Potential is range (U_V(a, b) - 1) downto 0;
type Nbits_Weights is range (U_W(a, b) - 1) downto 0;
type Nbits_Z std_logic;
```

Further the architecture can be use of `Nbits_Potential`, `Nbits_Weights` and `Nbits_Z` to create vectors with this sizes.

Input Module

The Input Module contains a LUT with the values of the synaptic weights and is activated with a signal from the control block (Figure 4.3). This module uses the `Nbits_Weights` data type to define the size of each value in the LUT. The synaptic weights are separated and routed from this module

to each neuron in a combinational process. The LUT requires a larger vector than traditional hardware implementations, since $\mathbf{W}$ is defined by $N \times N \times D$.

In the figure 4.3 we show a block labeled as GRWG (Gaussian Random Weights Generator), which is a prototype to be further developed using the Ziggurat algorithm Zhang and Leong (2005) in order to generate the synaptic weights on-chip or as a embedded system.

gIF-type neuron module

This module can to compute the gIF-type neuron models using either the analog approach, which consider a non-linear activation function to estimate the output of each synaptic connection or the discrete approach, which takes into account the spiking state of the synaptic connections. Our method is a quite different from the most part of the works on hardware implementations of spiking neural networks, due that we consider delayed synaptic weights. It means that each synaptic weight is modeled with D parameters as was previous described. It means also that our neuron model will perform more computations than the other works. However at a network level it implies the same processing time, because the weighting sum is a combinational process.

Although this module uses all the predefined data types and operations use fixed-point arithmetic. Its operation is synchronized with the other modules. Once that the weights have been separated in the input module they are sent to each neuron, thus when the neuron receives them, it performs the membrane potential estimation. Finally this membrane potential is compared with a given threshold in order to know the firing state of the neuron.

Let us to describe the internal structure of the gIF-type neuron model considering both, the analog and the discrete approach.

Architecture with a discrete activation function

In figure 4.4 we show the internal structure of the neuron module with discrete activation function. The goal of this module is to reproduce the comportment of the membrane potential and its consequently spiking firing state, which are describe by the equation 1.4.

The spikes generation is carried out by a combinational process inside the neuron module. Hence, the process is described in three steps.

- The Reset mechanism simulates a multiplexor, which switch between two possible states. The switch is controlled by the last firing state of the neuron $Z_i[k-1]$. If $Z_i[k-1] = 1$ (neuron comes to fire a spike), a "logic 0" (neuron is reseted) is reflected in the output. Otherwise we

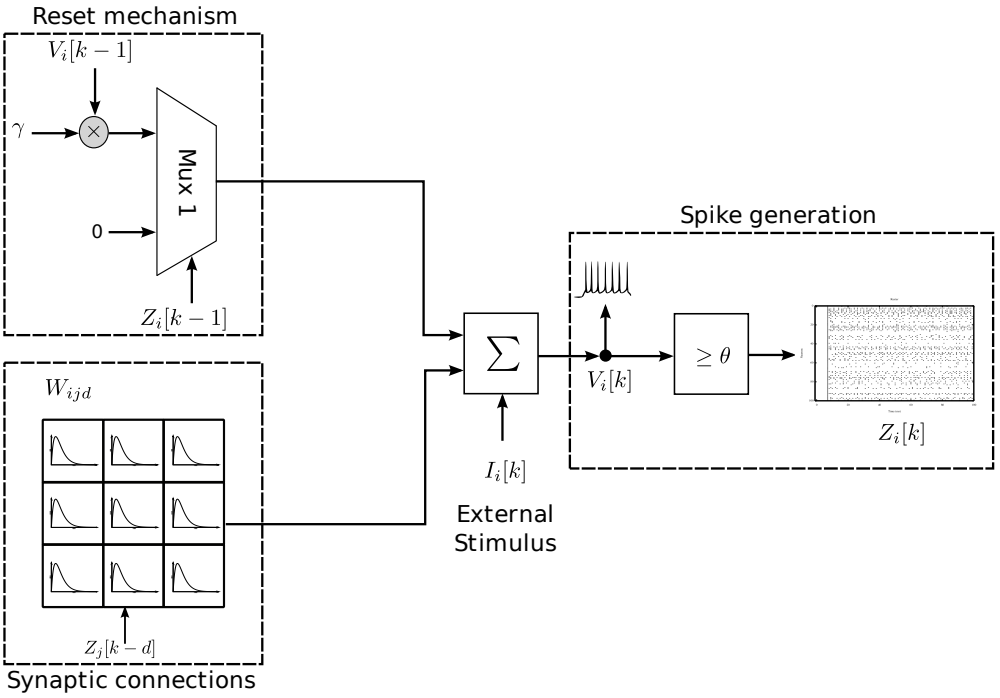


Figure 4.4: Block diagram of individual discrete-time spiking neuron

have a contraction effect in the neuron, which is given by a fixed-point value of the product between γ and $V_i[k-1]$ (see section 4.3 to more details). The output of this block goes directly to the $\sum$ block, which performs the sum of the reset state, external stimulus and the synaptic activity.

- Synaptic connections defines the activity among the neurons. We consider a fully-connected topology, which means that all neurons are connected. In addition each synaptic connection evaluates the past spiking activity of the other neurons. Here, $N \times D$ delayed weights are computed (par neuron).

- Spike generation is carried out by a comparator. Here, the membrane potential $V_i[k]$ is compared with a established firing threshold. If the membrane potential reaches this threshold a "logic 1" is set in the output, otherwise a "logic 0'.

The VHDL description

- The `entity` of the VHDL design makes use of the defined data types.

- The `port` has the inputs and the outputs declaration necessaries to perform the operations of the equation 1.4.

in The constant value of γ defined with `Nbits_Leak` bits.

in The constant value of the external stimulus I^{ext} defined with `Nbits_Current` bits.

in The **W** vector with $N \times D$ values, where each one has a size of `Nbits_Weights`.

in The **Z** vector with $N \times D$ values, where each one has a size of `Nbits_Z`.

in The the last value of the membrane potential $V_i[k-1]$, which has a size of `Nbits_Potential` bits.

out The computed membrane potential $V_i[k]$ expressed with `Nbits_Potential` bits.

out The firing state of the neuron $Z_i[k]$ expressed with `Nbits_Z` bits.

- The `architecture` contains the computational description of the discrete-time spiking neuron model. The design of this `architecture` utilizes only combinational logic, since all the terms of the weighted sum are computed at the same time thanks to the `generate` declaration, which permits us to create all the topology around the neuron.

Architecture with a non-linear activation function

As we have described throughout this chapter the unique difference between the analog and the discrete approaches is the way to compute the synaptic activity. In this sense the figure 4.5 schematize the internal structure of the hardware description of the gIF-type neuron model considering a non-linear activation function.

- The Reset mechanism operates in the same way that in the discrete case, since $\rho(V_i[k-1]) = Z_i[k]$. The multiplexor (Mux 1) selects between a "logic 0"s (reset) and a memory value given by the product of γ and $V_i[k-1]$. The selection is based in the firing state of the neuron i in the last time $k-1$. The multiplexor output will be a "logic 0" if $\rho(Z_i[k-1]) = 1$, otherwise this will be the product.

- Synaptic connections emulate real biological synapses through a mathematical model, which involves an activation function and synaptic weights. Further this model is characterized using fixed-point arithmetic to be implemented onto a FPGA. The activation function considered in this work is a sigmoid profile (Figure 1.9), which displays a history dependent progression.

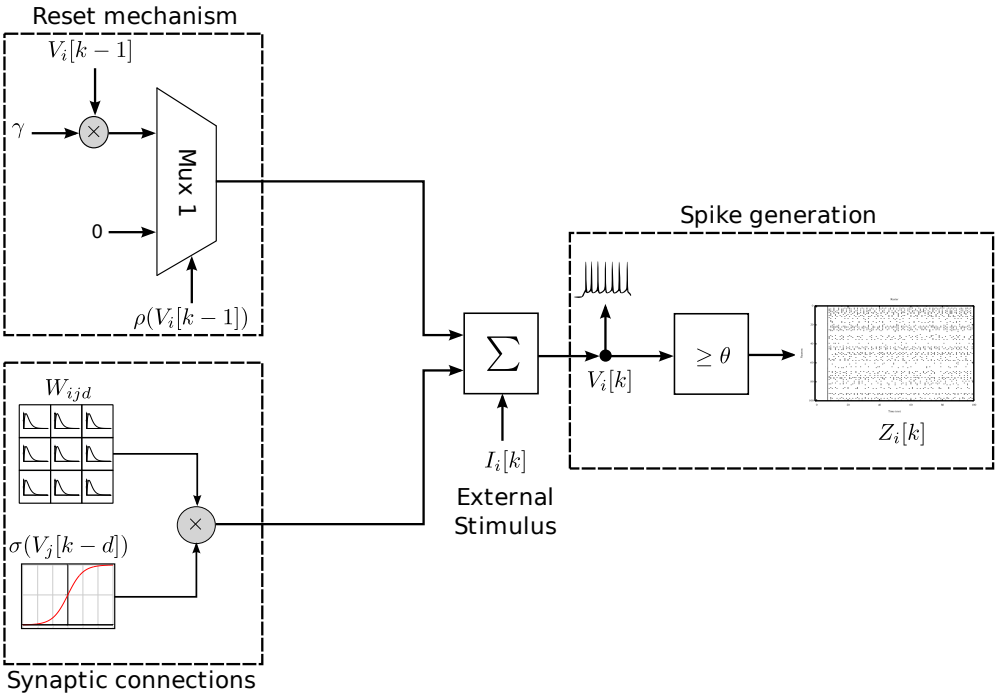


Figure 4.5: Analog-spiking neuron model

The synaptic weights are estimated from a random gaussian distribution and stored in a LUT, where each synaptic connection is modeled by D parameters.

The straightforward hardware implementation of a sigmoid function is not a feasible realization due that both, division and exponentiation in 1.14 are very demanding operations. In Tommiska (2003); Zhang et al. (1996); Myers and Hutchinson (1989); Alippi and Storti-Gajani (1991); Basterretxea and Tarela (2001); Amin and Curtis (1997) the authors present different approaches to efficiently implement a sigmoid function.

First, let us to write the differentiation of equation 1.14, which is given by

$$\frac{dy}{dx} = y(1-y)$$

where $x = V[k-d]$ and $y = \sigma(V[k-d])$.

Also, from figure 1.9 we can observe that the sigmoid function is symmetric at $(0, 0.5)$. Hence, this fact permits us to compute only a half part of the (x, y) pairs and to deduce the rest as follow:

$$y_{x>0} = 1 - y_{x\leq 0}$$

or,

$$y_{x<0} = 1 - y_{x\geq 0}$$

FPGA-based hardware designs use these assumption to approximate the sigmoid function in order to have an efficient implementation. In this sense we consider the method of Alippi and Storti-Gajani (1991) in order to implement the sigmoin function as an internal block of the neuron module. This method bases its approximation on selecting an integer set of breakpoints, and setting the y-values as power of two numbers. Thus, the approximation writes:

$$y = \frac{\frac{1}{2} + \frac{b}{4}}{2^{|a|}} \tag{4.14}$$

where a and b correspond to the integer and fractional parts respectively of the fixed-point representation of V (Equation 4.12).

The hardware implementation of this sigmoid approximation function is quite simple, since all is needed is a shift register and a counter to control it Alippi and Storti-Gajani (1991)

- Spike generation, has the same structure that the discrete-time model. Here, the membrane potential is compared with a given firing threshold. The output of this block is sent to the router module (labeled as Output Module in Figure 4.3).

Note: in both cases the external stimulus I^{ext} is a constant value defined in 4.9.

Output Module

The Output Module is a kind of router handled by the general control. The goal of this module is to bridge the spiking activity of all neurons to a RAM memory. In addition, this module has a register of N elements in the discrete model and $N \times D$ elements in the analog model, where each element has a size of `Nbits_Potential` bits. Thus, the last state of the membrane potential of each neuron is temporarily stored in this register. Further these values are used as a feedback toward the neurons.

4.5 GPU-BASED IMPLEMENTATION OF GIF-TYPE NEURON MODELS

An heterogeneous programming scheme derives in a mix of languages that are commonly used to perform the solution for a certain computational

problem. In a particular opinion a well-crafted and powerful programming scheme should be formed by Python & C++ & CUDA. This combination has inspired this section of the work, it because we needed a powerful software/hardware implementation in order to compare the performance of our FPGA-based implementation. We consider the GPU-based implementation as a combination of software and hardware due that the interface between the GPU and the programmer is carried out by the CPU using a high-level language, such as C/C++, python, etc. The main and the most considerable problem occurs here for GPU-based implementations due that the data transferring between the GPU and the CPU could be derived in a neck-bottle making slowly the communication between both devices.

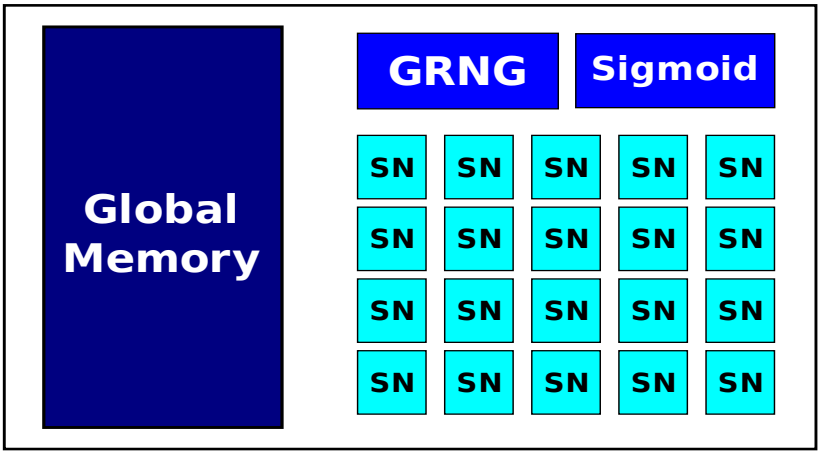


Figure 4.6: A GPU-based architecture

The GPU-based implementation can be described by the next steps:

`__host__`

The develop of a GPU-based implementation begins with an initialization on the `host` of the constant parameters, such as γ, I, N, D and T. Here, we also allocate the space memory of the vectors $\mathbf{V}$ and $\mathbf{Z}$in two ways:

- We allocate the three vectors that will be used in the `host` with the subroutine `malloc`. The vectors are labeled as `h_V` (double *) and `h_Z` (bool *), where the `h` means that will be exclusively used in the `host`.
- We allocate the three vectors that will be used in the `device` with the subroutine `cudaMalloc`. The vectors are labeled as `d_V` (double *) and `d_Z` (bool *), where the `d` means that will be exclusively used in the `device`.

Since the synaptic weights vector is defined by the qualifier `__device__`

it has no communication with the `host`.

Thus, the vectors are mapped as follow:

```
cudaMemcpy(d_V, h_V, N * T, cudaMemcpyHostToDevice)
cudaMemcpy(d_Z, h_V, N * T, cudaMemcpyHostToDevice)
```

The `cudaMemcpy` subroutine permits us to send the initial data ($V_i[d]$ and $Z_i[k]$, where $k = \{1...D\}$) from the `host` to `device` in order to simulate the gIF-type models in the GPU.

`__device__` (GRNG *kernel*)

The `device` is a CUDA function that can be only executed on the GPU. We use this programming model to implement a gaussian random number generator (GRNG) in order to generate the $N \times N \times D$ synaptic weights. This function uses the Mersenne Twister method Matsumoto and Nishimura (1998) to generate the random numbers, which is based on the Monte Carlo approaches Metropolis (1987). Then we use the Box-Muller Transform Box and Muller (1958) to map the generated random number into a normal distribution.

We have chosen the Mersenne Twister method because it presents the two main properties of the random number generators. On one hand it has a long period of $2^{19,937}$, which is enough long to generate several random values. On the other hand by its good statistical quality Knuth (1969); L'Ecuyer (2006).

The GRNG *kernel* is called from the Neuron *kernel* as a simple C function.

`__device__` (Sigmoid *kernel*)

The GPU-based architecture has another *kernel* called Sigmoid. In this kernel we have perform the equation 1.14, such function is used by the model when we consider the analog form of the gIF-type neuron models. It is also a CUDA function, which is exclusively executed in the GPU.

The exponential operation (e) in equation 1.14 is completely supported by the device using the `__expf(x)` function.

`__global__` (SN *kernel*)

The SN *kernel* is the main element of the GPU-based architecture due that it contains the description of the gIF-type neuron models. This *kernel* is defined by the `__global__` CUDA qualifier due that it is used in the `host` but described in the `device`.

The *kernel* is invoked in the `host` as follow:

```
cuSN<<<blocksPerGrid, MY_KERNEL_MAX_THREADS>>>(d_Z, d_V, leak, Iext, synapse)
```

where:

`cuSN` is the given name to the *kernel*.

Since the GPU internal structure is divided in three levels (grid, blocks and threads), we need to define them. The `blocksPerGrid` and the `MY_KERNEL_MAX_THREADS` define the GPU workspace where the network will be mapped. These parameters depend on GPU card and the number of neurons. Further this structure permits us to execute our model in parallel, where all neurons are computed at the same time. The vectors `d_V` and `d_Z` will contain the processed data in the device, in other words this variables will contain the neural activity. The `leak` and `Iext` are the leak factor and the external stimulus respectively. Finally the `synapse` parameter defines the kind of synaptic connection that will used to compute the connections in the simulation of the neural network.

`__host__`

Once that the data has been processed in the `device` it is recover by the `host` using the next subroutines:

```
cudaMemcpy(h_V, d_V, N * T, cudaMemcpyDeviceToHost)
cudaMemcpy(h_Z, d_V, N * T, cudaMemcpyDeviceToHost)
```

Note: in order to obtain the maximum performance of this approach we need consider large scale neural networks. Otherwise, we will not have an important gain on the processing speed in comparison with a classical software implementation. Besides the approach is highly suitable when learning mechanism are considered due to the high computational costs.

4.6 Numerical Results

In this section we present the numerical results of the FPGA-based implementations of the gIF-type neuron models. Our experiments are based on a precision analysis. Thus, this analysis has permitted us to reproduce spiking dynamics at different fixed-point data representations and to evaluate the behavior of these representations when they are exposed to parameters that generate complex spiking dynamics. In tables 4.1 and 4.2 we show the different precision values that we have considered to simulate our models in the FPGA. More specific, these values correspond to the fractional part b of the fixed-point representation, which has been studied in section 4.3 and that permits us to map the equations of the gIF-type neuron models onto the FPGA.

	Fixed-Point representation					
	b = 6	b = 7	b = 8	b = 9	b = 10	b = 11
Precision	0.984375	0.9921875	0.99609375	0.998046875	0.9990234375	0.99951171875

Table 4.1: Precision for different values of b, where b is the fractional part in the fixed-point representation.

	Fixed-Point representation				
	b = 12	b = 13	b = 14	b = 15	b = 32
Precision	0.99975585937	0.99987792969	0.99993896484	0.99996948242	0.99999999977

Table 4.2: Precision for different values of b, where b is the fractional part in the fixed-point representation. (Continuation)

These values allow to define the fixed-point representation for the leak factor (γ) and the external stimulus (I^{ext}), since their representations only depend on the fractional bits. However in order to define the the integer part of the fixed-point representations for $\mathbf{V}$ and $\mathbf{W}$ a numerical test have performed.

The numerical test consists of 10000 simulations in software of one of the proposed neuron models in order to obtain an average of the number of integer bits for each variable. In the case of $\mathbf{W}$ we need to take into account that their values are randomly chosen from a normal distribution $\mathcal{N}(\mu, \sigma^2)$. In this sense we estimate the best representation for a from different variances $\sigma^2 = \frac{C}{\sqrt{N}}$ as we show in table 4.3.

		Fixed-Point representation									
		C = 1	**C = 2**	**C = 3**	**C = 4**	**C = 5**	**C = 6**	**C = 7**	**C = 8**	**C = 9**	**C = 10**
N = 4	**W**	a = 1	a = 2	a = 3	a = 3	a = 4	a = 4	a = 4	a = 4	a = 5	a = 5
N = 10	**W**	a = 1	a = 2	a = 3	a = 3	a = 3	a = 4	a = 4	a = 4	a = 4	a = 4
N = 20	**W**	a = 1	a = 2	a = 2	a = 3	a = 3	a = 3	a = 3	a = 4	a = 4	a = 4
N = 50	**W**	a = 1	a = 1	a = 2	a = 2	a = 2	a = 3	a = 3	a = 3	a = 3	a = 3
N = 100	**W**	a = 1	a = 1	a = 1	a = 2	a = 2	a = 2	a = 3	a = 3	a = 3	a = 3

Table 4.3: Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$.

Moreover in the case of $\mathbf{V}$, we first estimate its minimum (Equation 1.9) and maximum (Equation 1.10) values obtained during a simulation. Then from equation 4.13 we define the number of integer bits a in order to map $\mathbf{V}$ in the FPGA device. Note that equations 1.9 and 1.10 depend on the leak

factor (γ). For this reason we have performed the numerical test for different values of γ. The results are shown in 4.4, 4.5 and 4.6. The indicative of high, medium and low resources usages are due to the value of γ, note that when it is closer to 1 the value of a is bigger than when γ is closer to 0.

- $\gamma = 0.95$ (High resources usage)

		Fixed-Point representation									
		C = 1	**C = 2**	**C = 3**	**C = 4**	**C = 5**	**C = 6**	**C = 7**	**C = 8**	**C = 9**	**C = 10**
N = 4	**V**	a = 8	a = 9	a = 9	a = 10	a = 10	a = 10	a = 10	a = 11	a = 11	a = 11
N = 10	**V**	a = 8	a = 9	a = 10	a = 10	a = 11	a = 11	a = 11	a = 11	a = 11	a = 11
N = 20	**V**	a = 9	a = 10	a = 10	a = 11	a = 11	a = 11	a = 11	a = 12	a = 12	a = 12
N = 50	**V**	a = 9	a = 10	a = 11	a = 11	a = 11	a = 12	a = 12	a = 12	a = 12	a = 12
N = 100	**V**	a = 10	a = 11	a = 11	a = 12	a = 12	a = 12	a = 12	a = 13	a = 13	a = 13

Table 4.4: Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$.

- $\gamma = 0.5$ (Medium resources usage)

		Fixed-Point representation									
		C = 1	**C = 2**	**C = 3**	**C = 4**	**C = 5**	**C = 6**	**C = 7**	**C = 8**	**C = 9**	**C = 10**
N = 4	**V**	a = 4	a = 5	a = 6	a = 6	a = 7	a = 7	a = 7	a = 7	a = 8	a = 8
N = 10	**V**	a = 5	a = 6	a = 6	a = 7	a = 7	a = 7	a = 8	a = 8	a = 8	a = 8
N = 20	**V**	a = 5	a = 6	a = 7	a = 7	a = 8	a = 8	a = 8	a = 8	a = 8	a = 9
N = 50	**V**	a = 6	a = 7	a = 7	a = 8	a = 8	a = 8	a = 9	a = 9	a = 9	a = 9
N = 100	**V**	a = 6	a = 7	a = 8	a = 8	a = 9	a = 9	a = 9	a = 9	a = 9	a = 10

Table 4.5: Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$.

- $\gamma = 0.1$ (Low resources usage)

		Fixed-Point representation									
		C = 1	**C = 2**	**C = 3**	**C = 4**	**C = 5**	**C = 6**	**C = 7**	**C = 8**	**C = 9**	**C = 10**
N = 4	**V**	a = 4	a = 4	a = 5	a = 5	a = 6	a = 6	a = 6	a = 6	a = 7	a = 7
N = 10	**V**	a = 4	a = 5	a = 6	a = 6	a = 6	a = 7	a = 7	a = 7	a = 7	a = 7
N = 20	**V**	a = 5	a = 5	a = 6	a = 6	a = 7	a = 7	a = 7	a = 7	a = 8	a = 8
N = 50	**V**	a = 5	a = 6	a = 7	a = 7	a = 7	a = 7	a = 8	a = 8	a = 8	a = 8
N = 100	**V**	a = 5	a = 6	a = 7	a = 7	a = 8	a = 8	a = 8	a = 8	a = 9	a = 9

Table 4.6: Number of integer bits a in the fixed-point representation. These values have been estimated considering different normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\sigma^2 = \frac{C}{\sqrt{N}}$.

In chapter 1 we have seen that the gIF-type neuron models present three different regimes (neural death, periodic and chaotic) in their dynamics, thus this phenomena is numerically demonstrated in this section. However in the numerical tests we focus mainly on periodic and chaotic regimes. Taking as reference the figure 1.8 the parameters have varied in our FPGA-based implementations in order to reproduce these regimes. The results that we show below demonstrate two things: first, the implementations that we have performed are able to reproduce these regimes and second, a generalization on the fixed-point representation can have drastic consequences on the desired results due that the number of bits used to reproduce trivial dynamics (periodic) is not the same than those to reproduce complex ones.

Analyzing the figures

We present a set of numerical results, where each figure contains three subfigures: a raster plot of the simulated spiking activity, a plot with the RMS error for the different precision levels and a plot with the Kullback-Leibler divergence. The results are presented according to their complexity, thus we start showing the trivial case, which correspond to periodic dynamics, then we present intermediate regimes such as: quasi-periodic and semi-chaotic, and finally we present complex dynamics (chaotic).

In order to understand the results shown in figures let us to describe each element:

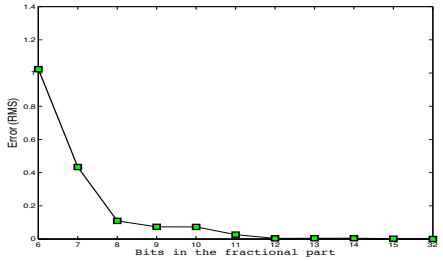
1. The raster plot represents both, the simulated dynamics in software and the desired reproduction in hardware. The red lines (|) define an initial activity, necessary since our models consider delays. The black lines (|) correspond to the generated activity by the neuron model.

2. In the RMS error plot we calculate the difference between the estimated membrane potential in software and hardware at different pre-

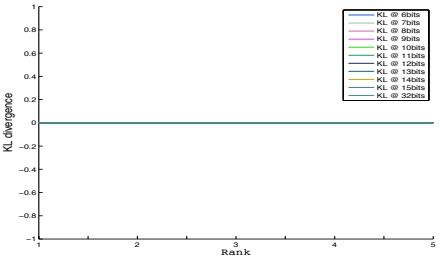
cision levels. Also, we estimate the Victor-Purpura distance between their respective spike trains, where:

- The green label (■) indicates that the spiking activity as in software ($\mathbf{Z}_S$) as in hardware ($\mathbf{Z}_H$) is the same $d(\mathbf{Z}_H, \mathbf{Z}_S) = 0$.
- The red label (■) indicates that the spiking activity in software ($\mathbf{Z}_S$) is different to that in hardware ($\mathbf{Z}_H$), $d(\mathbf{Z}_H, \mathbf{Z}_S) \neq 0$.

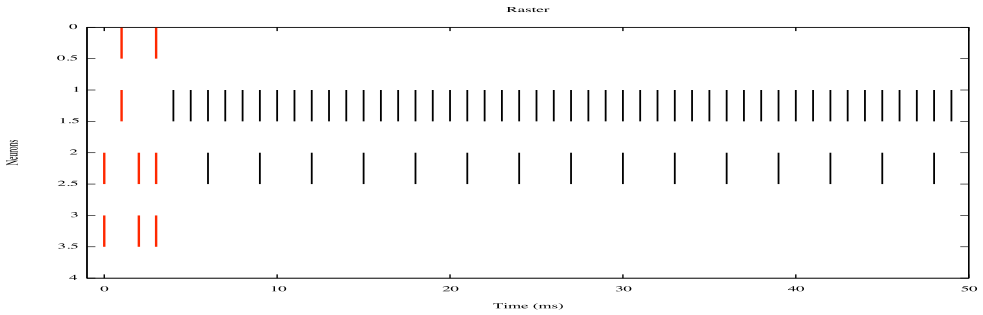
3. The KL (Kullback-Leibler) divergence plot shows a non-symetric measure of the difference between two probability distributions, $P(\mathbf{Z}_S)$ and $Q(\mathbf{Z}_H)$.



(a) RMS error

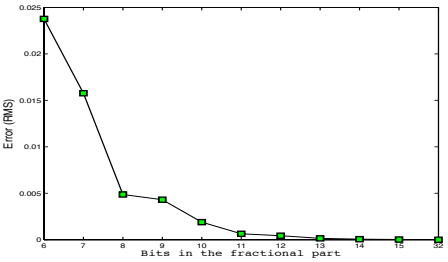


(b) Kullback-Leibler divergence

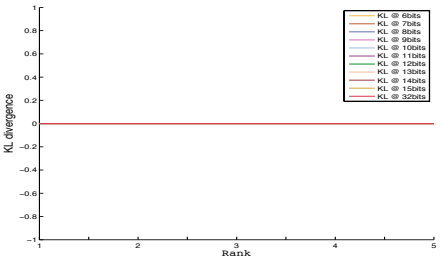


(c) Raster

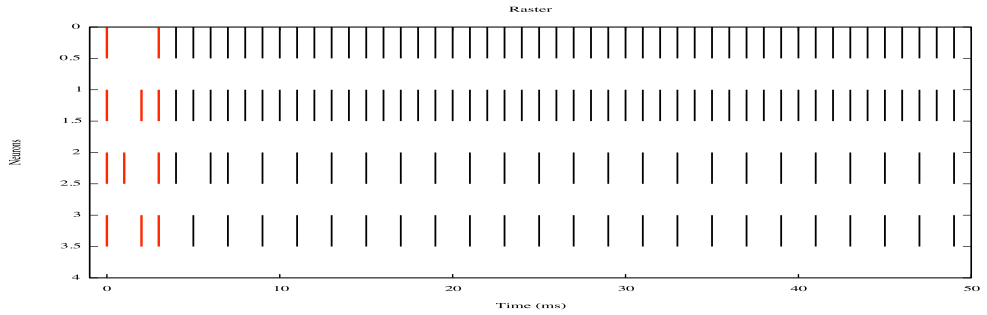
Figure 4.7: Numerical results on periodic dynamics ($N = 4$, $T = 50$).



(a) RMS error

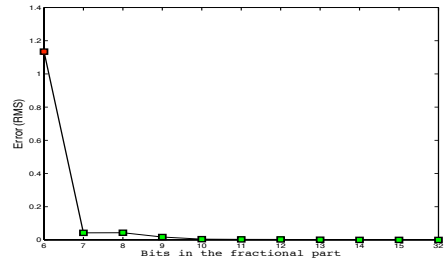


(b) Kullback-Leibler divergence

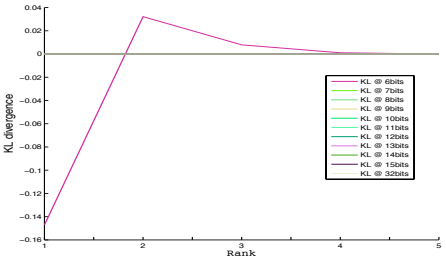


(c) Raster

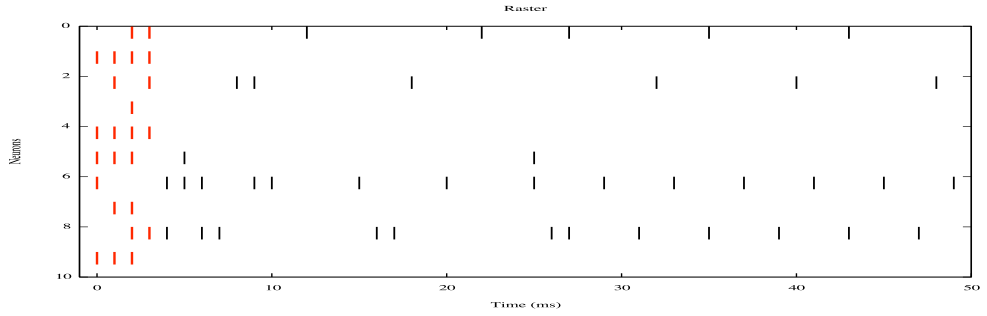
Figure 4.8: Numerical results on periodic dynamics ($N = 4$, $T = 50$).



(a) RMS error

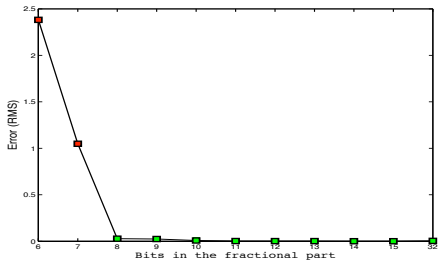


(b) Kullback-Leibler divergence

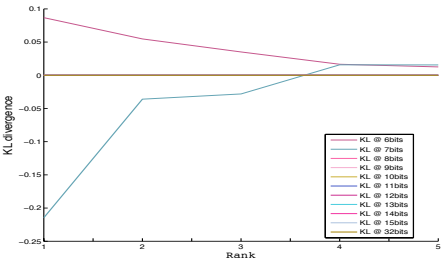


(c) Raster plot

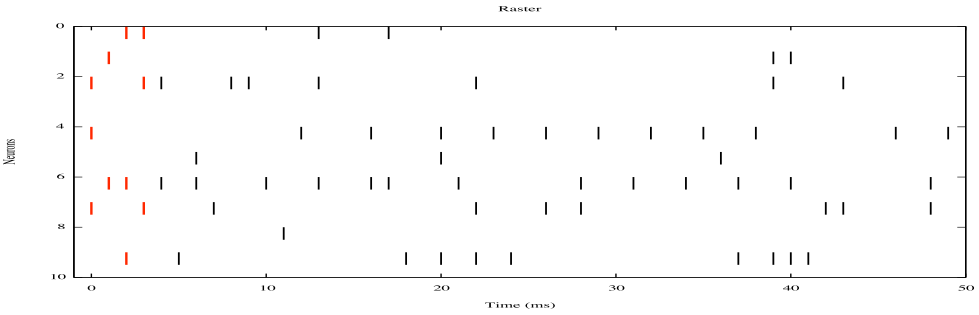
Figure 4.9: Numerical results on quasi-periodic dynamics ($N = 10$, $T = 50$).



(a) RMS error

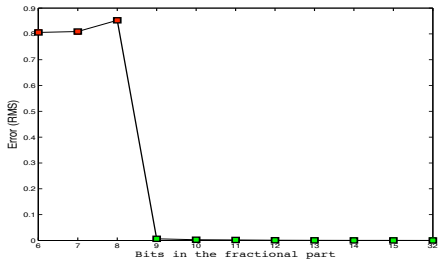


(b) Kullback-Leibler divergence

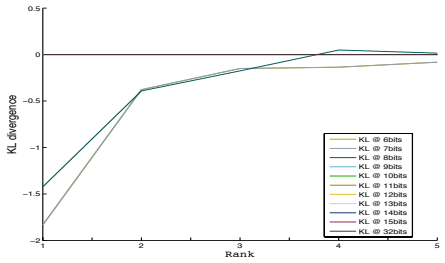


(c) Raster plot

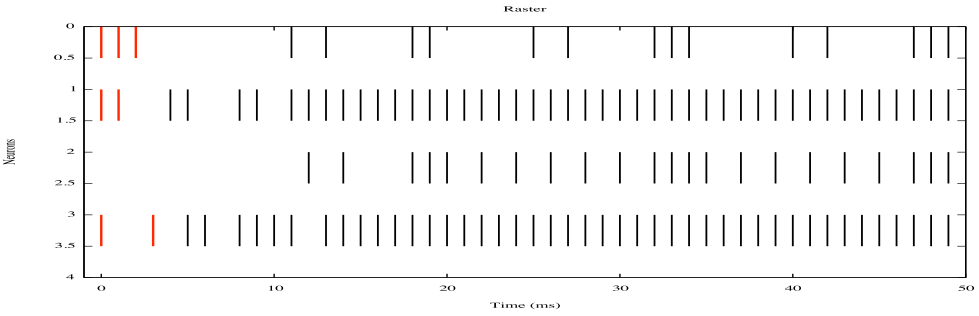
Figure 4.10: Numerical results on quasi-periodic dynamics ($N = 10$, $T = 50$).



(a) RMS error

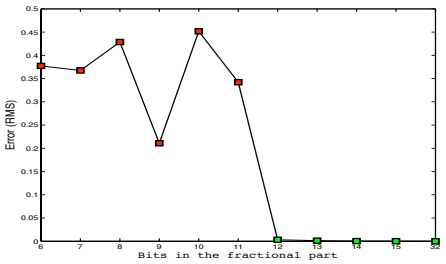


(b) Kullback-Leibler divergence

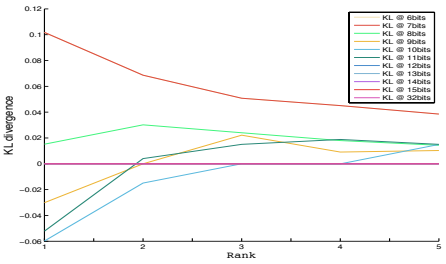


(c) Raster plot

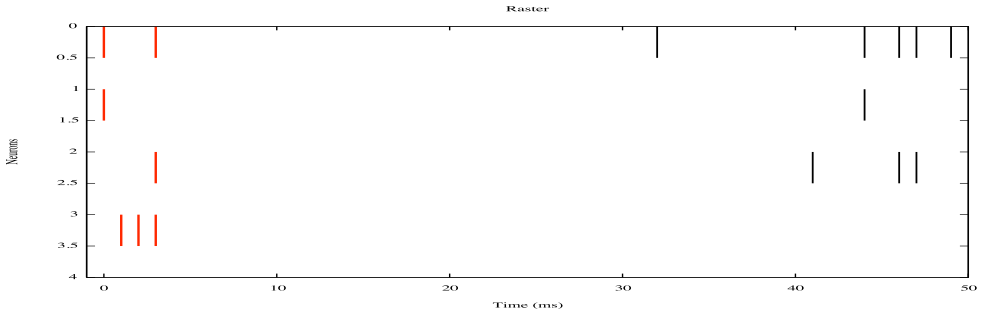
Figure 4.11: Numerical results on quasi-periodic dynamics ($N = 4$, $T = 50$).



(a) RMS error

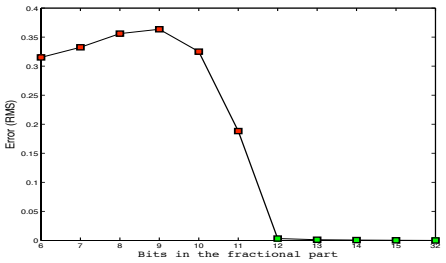


(b) Kullback-Leibler divergence

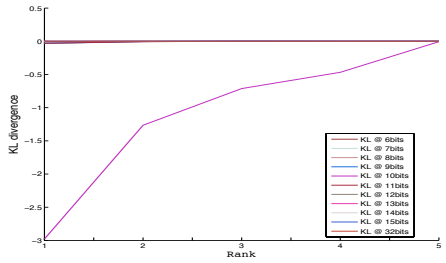


(c) Raster plot

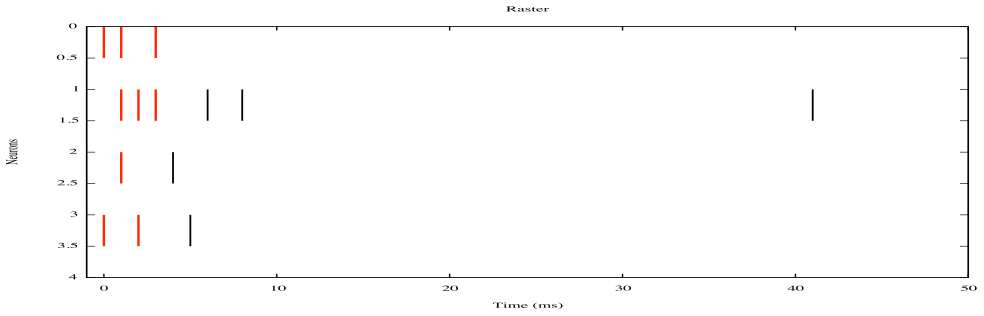
Figure 4.12: Numerical results on semi-chaotic dynamics ($N = 4$, $T = 50$).



(a) RMS error

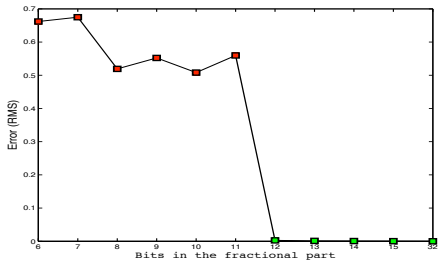


(b) Kullback-Leibler divergence

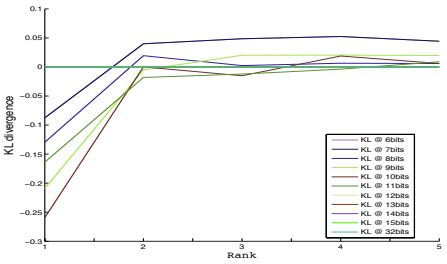


(c) Raster plot

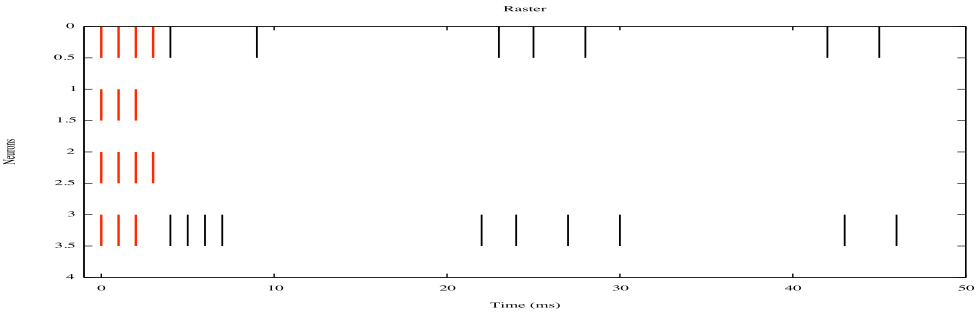
Figure 4.13: Numerical results on semi- chaotic dynamics ($N = 4$, $T = 50$).



(a) RMS error

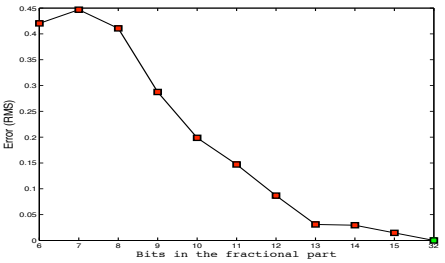


(b) Kullback-Leibler divergence

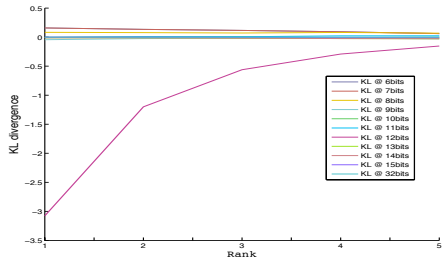


(c) Raster plot

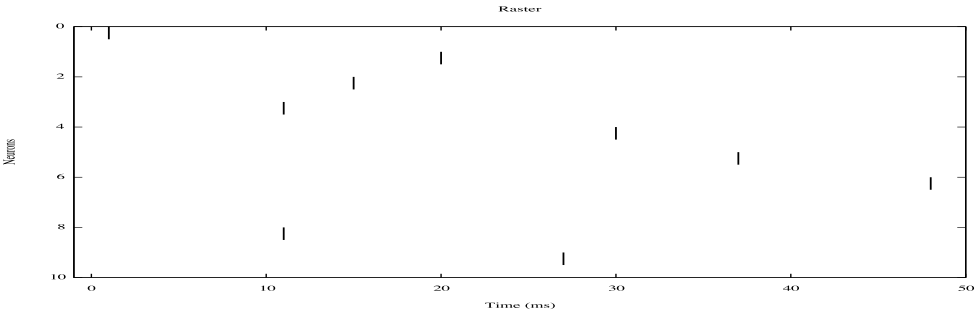
Figure 4.14: Numerical results on semi-chaotic dynamics ($N = 4$, $T = 50$).



(a) RMS error

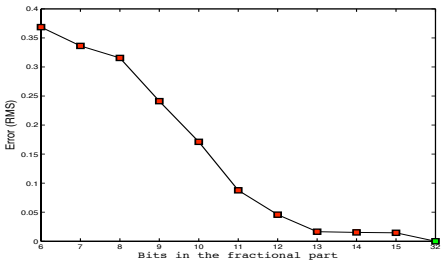


(b) Kullback-Leibler divergence

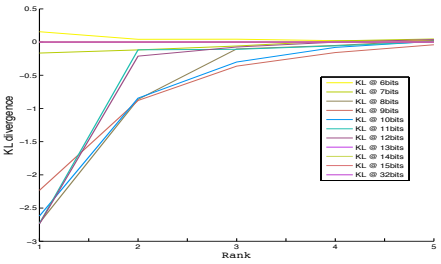


(c) Raster plot

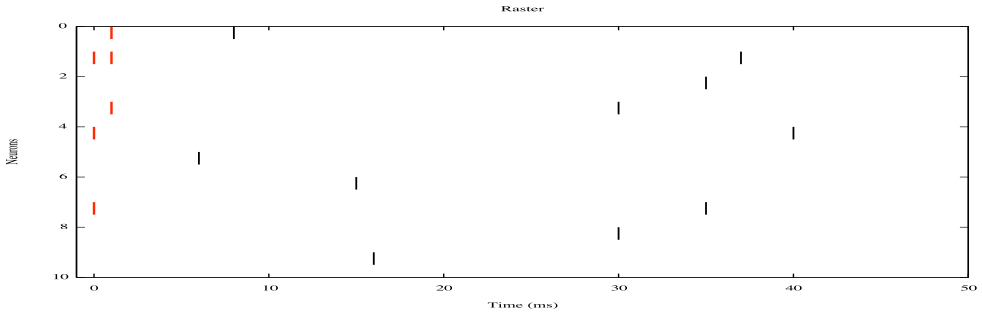
Figure 4.15: Numerical results on chaotic dynamics ($N = 10$, $T = 50$).



(a) RMS error

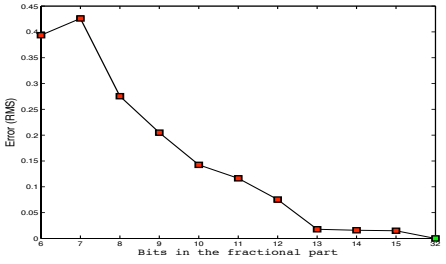


(b) Kullback-Leibler divergence

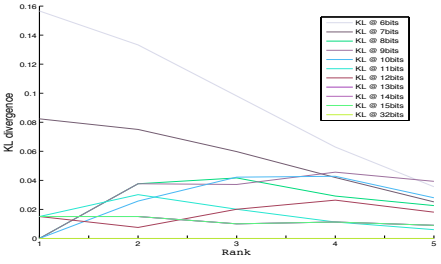


(c) Raster plot

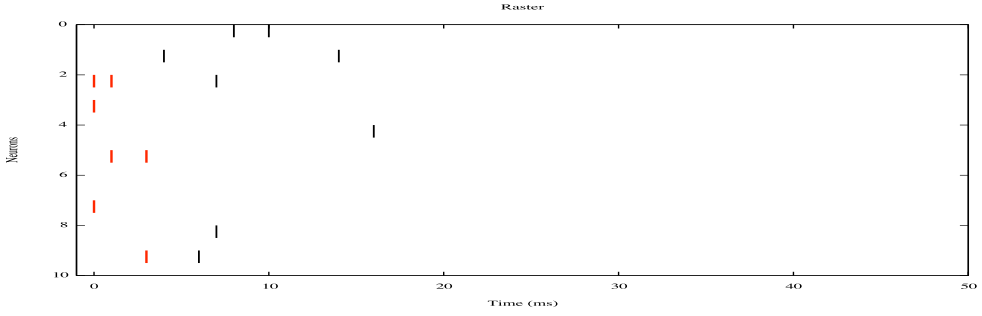
Figure 4.16: Numerical results on chaotic dynamics ($N = 10$, $T = 50$).



(a) RMS error



(b) Kullback-Leibler divergence



(c) Raster plot

Figure 4.17: Numerical results on chaotic dynamics ($N = 10$, $T = 50$).

- The **raster plots** correspond to dynamics that our FPGA-based implementations are able to reproduce.
- The **RMS error** gives us the difference between real (software) and estimated (hardware) membrane potential. However, from the RMS error we are not able to define if the spiking dynamics are the same. For this reason we have also estimated the Victor-Purpura distance between both rasters. Thus, we have results with 100 % of certainty. Moreover if we observe the evolution of these graphs we can observe an interesting phenomena:
  - When the raster plot evidences a **periodic activity**, the resources usage in the FPGA are **low**. This fact is due that in a periodic dynamic the membrane potential in the cell is constantly reseted. Thus, the error is not propagated throughout the trajectory.
  - Otherwise, when the raster plot evidences a **chaotic activity**, the resources usage in the FPGA are **high**. Contrary to the periodic case, in a chaotic dynamic the membrane potential variates constantly during a long period. Thus, each variation induces a cumulative effect on the error.
- The **Kullback-Leibler divergence** provides a "distance metric" on the space of probability distribution. From this, we can establish the minimum data representation required by the hardware implementations in order to reproduce a spiking activity. Further even if the hardware has not reproduced exactly the spiking activity than software some statistical methods can be applied to reconstruct the original one.

4.6.1 Synthesis and Performance

The synthesis of the designs has been carried out on a XC2VP100 FPGA of the Virtex II Pro family using the Xilinx ISE Project. The number of neurons that we can map onto the device is largely variable. This fact is due to the existence of different activity regimes (see figures 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16 and 4.17) in the gIF-type neuron models and also to the different normal distributions (table 4.3) that we have considered to define the synaptic weights. However, since we have developed a reconfigurable and general architecture, it can be adapted to each situation. In order to give an idea about the resources usage we evaluate the characteristics of the device and the gIF-type neuron models.

The XC2VP100 FPGA contains 44096 slices and 444 multipliers. These parameters have an important role on the number of neurons that we want to map onto the FPGA. In the case of the analog-spiking neuron model, we

can only map 30 neurons ($30 \times 30 \times 4$ synapses) due that the model incorporates multipliers to compute the synapse current. The maximal frequency is 46 MHz, it is 46000 times faster than real time, since we have established an Euler integration time step of 1 ms. Moreover the discrete-time spiking neuron model has a huge difference with the analog form of the gIF-type models. The description of this model allows a quasi-direct mapping onto the FPGA due that the synaptic activity is described by the firing state of the neurons. Thus we can map at least 1000 neurons fully-connected, it is $1000 \times 1000 \times 4$ synapses with a maximal frequency of 94 MHz, it is 94000 times faster than real time due that we consider the same Euler integration time step than the analog case. In both cases we have considered the maximum resources usages, it means that the models can reproduce either trivial dynamics (periodic) or complex dynamics (chaotic).

Now, as we described in this chapter we have also implemented a GPU kernel in order to compare the performance of the FPGA-based implementations. However, for small networks such as in the analog case there is a difference almost negligible between the simulation in classical software (C++) and the GPU kernel. This fact is due that in the data transference process from the GPU to the PC the gain of the parallel processing carried out by the device is lost. Nevertheless, if we consider the discrete-time neuron model with 1000 neurons, we gain a factor of 50 on classical software. This speeds are far from those in the FPGA, however this factor can be elevated until 1000 or more with a larger neural network or a better GPU card, since we have implemented our GPU kernel on a Alienware PC with a NVIDIA GeForge GT 335M card. The advantage of the GPU kernel is that we can consider larger networks than a FPGA for comparative hardware.

Part V

Conclusion

CHAPTER 5

CONCLUSION

Considering a deterministic time-discretized spiking network of neurons with connection weights having delays, we have been able to investigate in details to which extend it is possible to reverse-engineer the networks parameters, i.e., the connection weights. Contrary to the known NP-hard problem which occurs when weights and delays are to be calculated, the present reformulation, now expressed as a Linear-Programming (LP) problem, provides an efficient resolution and we have discussed extensively all the potential applications of such a mechanism, including regarding what is known as reservoir computing mechanisms, networks with or without a full connectivity, etc..

At the simulation level, this is a concrete instantiation of the that rasters produced by the simple model proposed here, can produce any rasters produced by more realistic models such as Hodgkin-Huxley, for a finite horizon.

At the computational level, we are here in front of a method which allows to "program" a spiking network, i.e. find a set of parameters allowing us to exactly reproduce the network output, given an input. Obviously, many computational modules where information is related to "times" and "spikes" are now easily programmable using the present method. This idea has also extended to consider both, "analog" and "spike" computations, the fact that we have studied both the unit analog state and the unit spike firing reverse-engineering problems (corresponding to the L and LP problems), tends to show that we could generalize this method to networks where both "times" and "values" have to be taken into account. The present equations are to be slightly adapted, yielding to a LP problem with both equality and inequality constraints, but the method is there.

At the modeling level, the fact that we do not only statistically reproduce the spiking output, but reproduce it *exactly*, corresponds to the computational neuroscience paradigm where "each spike matters" Guyonneau et al. (2005); Delorme et al. (2001). The debate is far beyond the scope of the present work, but interesting enough is the fact that, when considering natural images, the primary visual cortex activity seems to be very sparse

and deterministic, contrary to what happens with unrealistic stimuli Baudot (2007). This means that it is not a nonsense to address the problem of estimating a raster exactly.

Furthermore, exact input/output matching is required as a basic tool in sophisticated learning paradigms such as robust learning schemes.

As far as modeling is concerned, the most important message is in the "delayed weights design: the key point, in the present context is not to have one weight or one weight and delay but several weights at different delays". We have seen that this increases the computational capability of the network. In other words, more than the connection's weight, the connection's profile matters.

Furthermore, we point out how the related LP adjustment mechanism is distributed and has the same structure as an "Hebbian" rule. This means that the present learning mechanism corresponds to a local plasticity rule, adapting the unit weights, from only the unit and output spike-times. It has the same architecture as another spike-time dependent plasticity mechanism. However, this is supervised learning mechanisms, whereas usual STDP rules are unsupervised ones, while the rule implementations is entirely different.

To which extends this LP algorithm can teach us something about how other plasticity mechanisms is an interesting perspective of the present work. Similarly, better understanding the dynamics of the generated networks is another issue to investigate, as pointed out previously.

Although, we have studied some mechanisms in order to determine a reasonable number of hidden neurons, since we have observed that the role of these hidden layer used to span the linear space corresponding to the expected raster, as detailed in section 2.2. This opens a way, not only to find a correct set of hidden units, but a minimal set of hidden units. In this sense we have also started to explore some statistical methods Cessac et al. (2009). However, even if we have established a statistical relation between the input and the hidden neurons, the number of hidden neurons is expected to remain large. This is thus an open question to be further investigated.

Moreover in chapter 3 we have defined a theoretical framework to generalize the exact parameters estimation to an universal approximator have been pointed out by considering a mollification in the spiking metrics. Thus, this mollification allows us to make more explicit each operation when we compare two spiking dynamics. Direct applications of this idea include unsupervised or reinforcement learning schemes.

At the hardware implementation level we have pointed out two aspects: first, the different activity regimes (periodic, chaotic and intermediate) observed in the gIF-type neuron models have been reproduced by the developed

FPGA-based implementations. Second, these activity regimes have an important effect on the design process of a FPGA-based architecture due that the precision depends on the activity that you want to reproduce. Both issues have been addressed in detail in this work.

APPENDIX **A**

PUBLICATIONS OF THE AUTHOR ARISING FROM THIS WORK

Journal papers

1. H. Rostro-Gonzalez, B. Cessac and T. Vieville. *Exact spike-train reproduction with a neural network model*. Submitted to Neurocomputing

2. B. Cessac, H. Rostro, J.C. Vasquez and T. Vieville. *How Gibbs distributions may naturally arise from synaptic adaptation mechanisms. A model-based argumentation*. Journal of Statistical Physics, 136, (3), 565-602 (2009).

Conference papers

1. H. Rostro-Gonzalez, J. H. Barron-Zambrano, C. Torres-Huitzil and Bernard Girau. *Low-cost hardware implementations for discrete-time spiking neural networks*. Proceedings in Neurocomp 2010. Lyon, France.

2. B. Cessac, J.C. Vasquez, H. Nasser, H. Rostro-Gonzalez, T. Vieville and A. Palacios. *Parametric estimation of spike train statistics by Gibbs distributions: an application to bio-inspired and experimental data.* Proceedings in Neurocomp 2010. Lyon, France

3. H. Rostro, B. Cessac, J.C. Vasquez, T. Vieville. *On deterministic reservoir computing: network complexity and algorithm.* Proceedings in Neurocomp 2009. Bordeaux, France

4. J.C. Vasquez, Bruno Cessac, Horacio Rostro-Gonzalez and Thierry Vieville. *Gibbs Distributions and STDP: A case study on recurrent Neural Networks.* Proceedings in Neurocomp 2009. Bordeaux, France

5. H. Rostro-Gonzalez, B. Cessac, J.C. Vasquez and T. Vieville. *Back-engineering in spiking neural networks parameters.* Eighteenth Annual Computational Neuroscience Meeting CNS 2009. Berlin, Germany. BMC Neuroscience 2009, 10(Suppl.10): P289, BioMed Central.

6. J.C. Vasquez, B. Cessac, H. Rostro-Gonzalez and T. Vieville. *How Gibbs Distributions may naturally arise from synaptic adaptation mechanisms.* Eighteenth Annual Computational Neuroscience Meeting CNS 2009. Berlin, Germany. BMC Neuroscience 2009, 10(Suppl.10): P289, BioMed Central.

7. B. Cessac, H. Rostro, J.C. Vasquez and T. Vieville. *Statistics of spikes trains, synaptic plasticity and Gibbs distributions.* Proceedings in Neurocomp 2008. Marseille, France

8. B. Cessac, H. Rostro, J.C. Vasquez and T. Vieville. *To which extend is the neural code" a metric?.* Proceedings in Neurocomp 2008. Marseille, France

Research Reports

1. Horacio Rostro-Gonzalez, Juan Carlos Vasquez-Betancur, Bruno Cessac and Thierry Vieville. *Reverse-engineering in spiking neural networks parameters: exact deterministic parameters estimation.* INRIA Research Report 7199, February 2010.

Bibliography

Alippi, C. and G. Storti-Gajani: 1991, 'Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning'. In: *In Proc. IEEE Int. Symp. on Circuits and Systems*. pp. 1505–1508. [118, 119]

Amin, H. and K. Curtis: 1997, 'Piecewise linear approximation applied to nonlinear function of a neural network'. *IEE Proc. Circuits, Devices Sys.* **144**(6), 313–317. [118]

Aronov, D.: 2003, 'Fast algorithm for the metric-space analysis of simultaneous responses of multiple single neurons'. *Journal of Neuroscience Methods* **124**(2). [71]

Astrom, K.: 1983, 'Theory and application of adaptive control: a survey'. *Automatica* **19**, 471–486. [85]

Bartlett, P. and J. Shawe-Taylor: 1999, 'Generalization Performance of Support Vector Machines and Other Pattern Classifiers'. In: B. Schölkopf, C. Burges, and A. Smola (eds.): *Advances in Kernel Methods, Support Vector Learning*. The MIT Press, Cambridge, Chapt. 4, pp. 43–54. [8]

Basterretxea, K. and J. Tarela: 2001, 'Approximation of sigmoid function and the derivative for artificial neurons'. In: *In Mastorakis, N. (Ed.): Advances in neural networks and applications*. pp. 397–401. [118]

Baudot, P.: 2007, 'Nature is the code: high temporal precision and low noise in V1'. Ph.D. thesis, Univ. Paris 6. [7, 138]

Belatreche, A., L. Maguire, and M. McGinnity: 2006, 'Advances in Design and Application of Spiking Neural Networks'. *Soft Computing - A Fusion of Foundations, Methodologies and Applications - Fuzzy-neural computation and robotics* **11**(3). [8]

Benedetti, R. and J.-J. Risler: 1990, *Real algebraic and semi-algebraic sets*. Hermann, Paris. [73]

Bengio, Y. and Y. LeCun: 2007, 'Scaling learning algorithms towards AI'. In: L. Bottou, O. Chapelle, D. DeCoste, and J. Weston (eds.): *Large-Scale Kernel Machines*. MIT Press. [84]

Bixby, R. E.: 1992, 'Implementing the Simplex Method: The Initial Basis'. *J. on Computing* **4**(3). [36]

Bohte, S. M. and M. C. Mozer: 2007, 'Reducing the Variability of Neural Responses: A Computational Theory of Spike-Timing-Dependent Plasticity.'. *Neural Computation* **19**(2), 371–403. [6, 77]

Box, G. E. P. and M. E. Muller: 1958, 'A Note on the Generation of Random Normal Deviates'. *The Annals of Mathematical Statistics* **29**(2), 610–611. [121]

Brette, R., M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. H. Jr., M. Zirpe, T. Natschläger, D. Pecevski, G. B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Viéville, E. Muller, A. P. Davison, S. E. Boustani, and A. Destexhe: 2007, 'Simulation of networks of spiking neurons: a review of tools and strategies'. *Journal of Computational Neuroscience* **23**(3), 349–398. [70]

Cassidy, A., S. Denham, P. Kanold, and A. Andreou: 2007, 'FPGA Based Silicon Spiking Neural Array'. *IEEE* **1**, 75–80. [9]

Cessac, B.: 2008, 'A discrete time neural network model with spiking neurons. Rigorous results on the spontaneous dynamics'. *J. Math. Biol.* **56**(3), 311–345. [xi, 4, 8, 10, 17, 23, 26, 27, 34, 36, 50, 68, 70, 90]

Cessac, B.: 2010, 'A discrete time neural network model with spiking neurons II. Dynamics with noise'. *J.Math. Biol., accepted*. [4, 113]

Cessac, B., H. Paugam-Moisy, and T. Viéville: 2010, 'Overview of facts and issues about neural coding by spikes'. *J. Physiol. Paris* **104**, 5–18. [7, 70, 72]

Cessac, B., H. Rostro-Gonzalez, J. Vasquez, and T. Viéville: 2008a, 'Statistics of spikes trains, synaptic plasticity and Gibbs distributions'. In: *Neurocomp 2008*. [60]

Cessac, B., H. Rostro-Gonzalez, J. Vasquez, and T. Viéville: 2008b, 'To which extend is the "neural code" a metric ?'. In: *Neurocomp 2008*. [21]

Cessac, B., H. Rostro-Gonzalez, J. Vasquez, and T. Viéville: 2009, 'How Gibbs distribution may naturally arise from synaptic adaptation mechanisms: a model based argumentation'. *J. Stat. Phys* **136**(3), 565–602. [138]

Cessac, B. and T. Viéville: 2008, 'Introducing numerical bounds to improve event-based neural network simulation'. *Frontiers in neuroscience*. submitted. [4, 17, 20, 22]

Cessac, B. and T. Viéville: 2008, 'On Dynamics of Integrate-and-Fire Neural Networks with Adaptive Conductances'. *Frontiers in neuroscience* **2**(2). [8, 17, 22, 34, 38, 70, 89]

Chazottes, J., E. Floriani, and R. Lima: 1998, 'Relative entropy and identification of Gibbs measures in dynamical systems'. *J. Statist. Phys.* **90**(3-4), 697–725. [60]

Che, S., J. Li, J. W. Sheaffer, K. Skadron, and J. Lach: 2008, 'Accelerating Compute-Intensive Applications with GPUs and FPGAs'. In: *Proceedings of the 2008 Symposium on Application Specific Processors*. Washington, DC, USA, pp. 101–107, IEEE Computer Society. [10]

Chechik, G.: 2003, 'Spike-Timing-Dependent Plasticity and Relevant Mutual Information Maximization'. *Neural Computation* **15**(7), 1481–1510. [6, 77]

Comaniciu, D. and P. Meer: 2001, 'Mean shift: A robust approach toward feature space analysis'. *IEEE Trans. Pattern Anal. Machine Intell.* [78]

Cooper, L., N. Intrator, B. Blais, and H. Shouval: 2004, *Theory of Cortical Plasticity*. World Scientific Publishing. [6, 77]

Darst, R. B.: 1990, *Introduction to Linear Programming Applications and Extensions*. Marcel Dekker Ltd, New-York. [36]

Davé, R. N. and R. Krishnapuram: 1997, 'Robust clustering methods: A unified view.'. *IEEE Transactions on Fuzzy Systems* **5**(2), 270–293. [78, 82]

Delorme, A., L. Perrinet, and S. Thorpe: 2001, 'Network of integrate-and-fire neurons using Rank Order Coding B: spike timing dependant plasticity and emergence of orientation selectivity'. *Neurocomputing* **38**, 539–545. [6, 7, 137]

Destexhe, A.: 1997, 'Conductance-Based Integrate and Fire Models'. *Neural Computation* **9**, 503–514. [17, 28]

Dubbs, A., B. Seiler, and M. Magnasco: 2009, 'A fast Lp spike alignment metric'. *arXiv:0907.3137v2*. [72]

Fidjeland, A., E. Roesch, M. Shanahan, and W. Luk: 2009, 'NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs'. *Application-Specific Systems, Architectures and Processors, IEEE International Conference on* **0**, 137–144. [10]

Gantmatcher, F.: 1977, *Matrix Theory*. New-York: Chelsea. [49]

Gerstner, W. and W. Kistler: 2002a, *Spiking Neuron Models*. Cambridge University Press. [xi, 4, 18, 19, 89]

Gerstner, W. and W. M. Kistler: 2002b, 'Mathematical formulations of Hebbian learning'. *Biological Cybernetics* **87**, 404–415. [6, 34, 77]

Girau, B. and C. Torres-Huitzil: 2007, 'Massively distributed digital implementation of an integrate-and-fire LEGION network for visual scene segmentation'. *Neurocomputing* **70**(7-9), 1186–1197. [9]

Glackin, B., T. McGinnity, L. Maguire, Q. Wu, and A. Belatreche: 2005, 'A Novel Approach for the Implementation of Large Scale Spiking Neural Networks on FPGA'. In: *In Computational Intelligence and Bioinspired Systems. Lecture Notes in Computer Science*, Vol. 3512. pp. 552–563. [9]

Graas, E., E. Brown, and R. Lee: 2004, 'An FPGA-based approach to high-speed simulation of conductance-based neuron models'. *Neuroinformatics* **2**, 417–435. 10.1385/NI:2:4:417. [9]

Grassmann, C. and J. Anlauf: 1999, 'Fast digital simulation of spiking neural networks and neuromorphic integration with SPIKELAB'. *Int J Neural Syst.* **9**(5), 473–478. [9]

Green, P.: 1990, 'On use of the EM algorithm for penalized likelihood estimation'. *J. Roy. Statist. Soc* **52**(3), 443–452. [81]

Guyonneau, R., R. vanRullen, and S. Thorpe: 2005, 'Neurons tune to the earliest spikes through STDP.'. *Neural Computation* **17**(4), 859–879. [6, 137]

Hashimoto, S. and H. Torikai: 2009, 'A novel hybrid spiking neuron: response analysis and learning potential'. In: *Proceedings of the 15th international conference on Advances in neuro-information processing - Volume Part I*. Berlin, Heidelberg, pp. 145–152, Springer-Verlag. [9]

Hauck, S. and A. Dehon (eds.): 2008, *Reconfigurable Computing*. Elsevier - Morgan Kaufmann Publishers. [101]

Hines, M. and N. Carnevale: 1997, 'The NEURON simulation environment'. *Neural Computation* **9**(6), 1179–1209. [9]

Hornik, K., M. Stinchcombe, and H. White: 1989, 'Multilayer feedforward networks are universal approximators'. *Neural Networks* **2**, 359–366. [5, 93]

Jaeger, H.: 2003, 'Adaptive nonlinear system identification with Echo State Networks'. In: S. Becker, S. Thrun, and K. Obermayer (eds.): *NIPS*2002, Advances in Neural Information Processing Systems*, Vol. 15. pp. 593–600, MIT Press. [5, 83]

Johnston, S., G. Prasad, L. Maguire, and M. McGinnity: 2005, 'Comparative Investigation into Classical and Spiking Neuron Implementations on FPGAs'. In: *Artificial Neural Networks: Biological Inspirations ICANN 2005*, Vol. 3696 of *Lecture Notes in Computer Science*. pp. 269–274, Springer Berlin, Heidelberg. [8]

Jolivet, R., T. Lewis, and W. Gerstner: 2004, 'Generalized Integrate-and-Fire Models of Neuronal Activity Approximate Spike Trains of a Detailed Model to a High Degree of Accuracy.'. *Journal of Neurophysiology* **92**, 959–976. [28]

Kirk, D. and W. Hwu (eds.): 2003, *Programming Massivelly Parallel Processors: A hands-on Approach*. MIT Press. [xvi, 10, 100]

Knuth, D. (ed.): 1969, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley. [121]

Koch, K., J. McLean, M. Berry, P. Sterling, V. Balasubramanian, and M. Freed: 2004, 'Efficiency of Information Transmission by Retinal Ganglion Cells'. *Current Biology* **14**(17), 1523–1530. [7]

Kullback, S.: 1959, *Information Theory and Statistics*. New-York: Wiley. [113]

Kullback, S. and R. Leibler: 1951, 'On Information and Sufficiency'. *Annals of Mathematical Statistics* **22**(1), 79 – 86. [113]

Langton, C.: 1990, 'Computation at the edge of chaos'. *Physica D.* **42**. [8]

L'Ecuyer, P.: 2006, 'TestU01: A C Library for Empirical Testing of Random Number Generators'. *ACM Transactions on Mathematical Software*. [121]

Lewis, N. and S. Renaud: 2007, 'Spiking neural networks in silico: from single neurons to large scale networks'. In: *Fourth International Multi-Conference on Systems, Signals and Devices*. [9]

Maass, W.: 1997, 'Fast sigmoidal networks via spiking neurons.'. *Neural Computation* **9**, 279–304. [4]

Maass, W.: 2001, 'On the relevance of time in neural computation and learning'. *Theoretical Computer Science* **261**, 157–178. (extended version of '97, in LNAI 1316:364-384). [5]

Maass, W. and C. M. Bishop (eds.): 2003, *Pulsed Neural Networks*. MIT Press. [4]

Maass, W. and T. Natschlager: 1997, 'Networks of Spiking Neurons can Emulate Arbitrary Hopfield nets in Temporal Coding'. *Neural Systems* **8**(4), 355–372. [4]

Maass, W., T. Natschläger, and H. Markram: 2002, 'Real-time computing without stable states: A new framework for neural computation based on perturbations'. *Neural Computation* **14**(11), 2531–2560. [5, 83]

Maguire, L. P., T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin: 2007, 'Challenges for large-scale implementations of spiking neural networks on FPGAs'. *Neurocomput.* **71**, 13–29. [9]

Mahowald, M. and R. Douglas: 1991, 'A silicon neuron'. *Nature* **354**, 239–255. [9]

Markram, H., J. Lübke, M. Frotscher, and B. Sakmann: 1997, 'Regulation of synaptic efficacy by coincidence of postsynaptic AP and EPSP'. *Science* **275**(213). [28]

Masland, R. H. and P. R. Martin: 2007, 'The unsolved mystery of vision'. *Curr Biol.* **17**(15), R577–82. [7]

Matsumoto, M. and T. Nishimura: 1998, 'Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator'. *ACM Transactions on Modeling and Computer Simulation*. [121]

Maya, S., R. Reynoso, C. Torres, and M. Arias-Estrada: 2000, 'Compact Spiking Neural Network Implementation in FPGA'. In: *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*. London, UK, pp. 270–276, Springer-Verlag. [9]

McCulloch, W. and W. Pitts: 1943, 'A logical calculus of the ideas immanent in nervous activity'. *Bulletin of Mathematical Biophysics* **7**, 115–133. [4]

Metropolis, N.: 1987, 'The Beginning of the Monte Carlo Method'. *Los Alamos Science*. [121]

Misra, J. and I. Saha: 2010, 'Artificial neural networks in hardware: a survey of two decades of progress'. *Neurocomputing* **74**(1-3), 239–255. [8]

Myers, D. and R. Hutchinson: 1989, 'Efficient implementation of piecewise linear activation function for digital VLSI neural networks'. *Electronic Letters* **25**(24), 1662–1663. [118]

Nageswaran, J. M., N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum: 2009, 'A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors'. *Neural Networks* **22**(5-6), 791 – 800. Advances in Neural Networks Research: IJCNN2009, 2009 International Joint Conference on Neural Networks. [10]

Packard, N.: 1988, *Dynamic patterns in complex systems*, Chapt. Adaptation towards the edge of chaos, pp. 293–301. World Scientific. [8]

Parhami, B. (ed.): 2000, *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press. [104]

Paugam-Moisy, H., R. Martinez, and S. Bengio: 2008, 'Delay learning and polychronization for reservoir computing'. *Neurocomputing* **71**, 1143–1158. [5, 84]

Pearson, M., A. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhesphy, and M. Nibouche: 2007, 'Implementing Spiking Neural Networks for Real-Time Signal-Processing and Control Applications: A Model-Validated FPGA Approach'. *IEEE Transactions on Neural Networks* **18**, 1472–1487. [9]

Pfister, J. and W. Gerstner: 2006, 'Triplets of Spikes in a Model of Spike Timing-Dependent Plasticity'. *J. Neuroscience* **26**, 9673–9682. [28]

Politi, A. and A. Torcini: 2009, 'Stable chaos'. *http://lanl.arxiv.org/abs/0902.2545*. [8]

Renaud, S., J. Tomas, Y. Bornat, A. Daouzli, and S. Saïghi: 2007, 'Neuromimetic ICs with analog cores: an alternative for simulating spiking neural networks'. In: *Proceedings of the IEEE 2007 InternationaI Symposium on Circuits And Systems ISCAS*. [9]

Riehle, A., F. Grammont, M. Diesmann, and S. GrÃijn: 2000, 'Dynamical changes and temporal precision of synchronized spiking activity in monkey motor cortex during movement preparation'. *J. Physiol (Paris)* **94**, 569–582. [xiv, 61, 62, 63]

Rieke, F., D. Warland, R. de Ruyter van Steveninck, and W. Bialek: 1996, *Spikes, Exploring the Neural Code*. The M.I.T. Press. [7, 70]

Rostro-Gonzalez, H., B. Cessac, J. Vasquez, and T. Viéville: 2010, 'Back-engineering of spiking neural networks parameters'. Research report rr-7199, INRIA. [80, 88, 91]

Rostro-Gonzalez, H., B. Cessac, J. C. Vasquez, and T. Viéville: 2009a, 'Back-engineering of spiking neural networks parameters'. In: *Computational Neurosciences meeting (CNS)*. [80]

Rostro-Gonzalez, H., B. Cessac, J. C. Vasquez, and T. Viéville: 2009b, 'On deterministic reservoir computing: network complexity and algorithm'. In: *Neurocomp'09*. [80]

Rougier, N.: 2006, 'Dynamic Neural Field with Local Inhibition'. *Biological Cybernetics* **94**(3), 169–179. [68]

Rousseeuw, P. and A. Leroy: 1987, *Robust Regression and Outlier Detection*. John Wiley & Sons, New York. [78]

Rudolph, M. and A. Destexhe: 2006, 'Analytical Integrate and Fire Neuron models with conductance-based dynamics for event driven simulation strategies'. *Neural Computation* **18**, 2146–2210. [4, 17, 28, 89]

Saighi, S., J. Tomas, Y. Bornat, B. Belhadj, O. Malot, and S. Renaud: 2010, 'Real-time multi-board architecture for analog spiking neural networks'. In: *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium*. pp. 1939 – 1942. [9]

Schäfer, A. M. and H. G. Zimmermann: 2006, 'Recurrent Neural Networks Are Universal Approximators'. *Lecture Notes in Computer Science* **4131**, 632–640. [5, 93]

Schemmel, J., J. Fieres, and K. Meier: 2008, 'Realizing Biological Spiking Network Models in a Configurable Wafer-Scale Hardware System'. In: *IEEE International Joint Conference on Neural Networks IJCNN*. [9]

Schrauwen, B.: 2007, 'Towards Applicable Spiking Neural Networks'. Ph.D. thesis, Universiteit Gent, Belgium. [6]

Soula, H., G. Beslon, and O. Mazet: 2006, 'Spontaneous dynamics of assymmetric random recurrent spiking neural networks'. *Neural Computation* **18**(1). [i, iii, 4, 18]

Soula, H. and C. C. Chow: 2007, 'Stochastic Dynamics of a Finite-Size Spiking Neural Networks'. *Neural Computation* **19**, 3262–3292. [17]

Strata, P. and R.Harvey: 1999, 'Dale's principle'. *Brain Res. Bull.* **50**, 349âĂŤ50. [37]

Sutton, R. S. and A. G. Barto: 1998, *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA. [81]

Thomas, D. B. and W. Luk: 2009, 'FPGA accelerated simulation of biologically plausible spiking neural networks'. In: *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*. [9]

Tomas, J., Y. Bornat, S. Saïghi, T. Levi, and S. Renaud: 2006, 'Design of a modular and mixed neuromimetic ASIC'. In: *Proceedings of the 13th IEEE International Conference on Electronics, Circuits and Systems ICECS*. [9]

Tommiska, M.: 2003, 'Efficient digital implementation of a sigmoid function for reprogrammable logic'. *IEEE, Computers and Digital Techniques* **150**(6), 403–411. [118]

Toyoizumi, T., J. Pfister, K. Aihara, and W. Gerstner: 2005, 'Generalized Bienenstock-Cooper-Munro rule for spiking neurons that maximizes information transmission'. *Proceedings of the National Academy of Science* **102**, 5239–5244. [6, 77]

Toyoizumi, T., J. Pfister, K. Aihara, and W. Gerstner: 2007, 'Optimality Model of Unsupervised Spike-Timing Dependent Plasticity: Synaptic Memory and Weight Distribution'. *Neural Computation* **19**, 639–671. [6, 77]

Tropp, J.: 2004a, 'Just relax: Convex programming methods for subset selection and sparse approximation'. Technical report, Texas Institute for Computational Engineering and Sciences. [45, 86]

Tropp, J., A. Gilbert, and M. Strauss: 2006, 'Algorithms for simultaneous sparse approximation. Part I: Greedy pursuit'. *Signal Processing* **86**, 572–588. [45]

Tropp, J. A.: 2004b, 'Greed is good: Algorithmic results for sparse approximation'. *IEEE Trans. Inform. Theory* **50**, 2231–2242. [45, 86]

Tropp, J. A.: 2006, 'Algorithms for simultaneous sparse approximation. Part II: Convex relaxation'. *Sparse approximations in signal processing* **86**, 589–602. [45]

VanVreeswijk, C.: 2004, *What is the neural code?* 23 Problems in System neuroscience. van Hemmen, J.L. and Sejnowski, T.Jr. (eds), Oxford University Press. [7]

Vapnik, V.: 1998, *Statistical Learning Theory*. John Wiley. [8]

Verstraeten, D., B. Schrauwen, M. DâĂŹHaene, and D. Stroobandt: 2007, 'An experimental unification of reservoir computing methods'. *Neural Networks* **20**(3), 391–403. [5, 83, 84]

Victor, J.: 2005, 'Spike train metrics'. *Current Opinion in Neurobiology* **15**(5), 585–592. [70, 75]

Victor, J., D. Goldberg, and D. Gardner: 2007, 'Dynamic programming algorithms for comparing multineuronal spike trains via cost-based metrics and alignments'. *J. Neurosci. Meth.* **161**, 351–360. [71]

Victor, J. and K. Purpura: 1996, 'Nature and precision of temporal coding in visual cortex: a metric-space analysis.'. *J Neurophysiol* **76**, 1310–1326. [112]

Viéville, T., D. Lingrand, and F. Gaspard: 2001, 'Implementing a multi-model estimation method'. *The International Journal of Computer Vision* **44**(1). [39, 79, 82]

Vogelstein, R., U. Mallik, J. Vogelstein, and G. Cauwenberghs: 2007, 'Dynamically Reconfigurable Silicon Array of Spiking Neurons With Conductance-Based Synapses'. *Neural Networks* **18**(1), 253 – 265. [9]

Šíma, J. and J. Sgall: 2005, 'On the Nonlearnability of a Single Spiking Neuron'. *Neural Computation* **17**(12), 2635–2647. [6, 40]

Wilson, M. A., U. Bhalla, J.D.Uhley, and J. M. Bower: 1989, 'GENESIS: A system for simulating neural networks'. In: *Advances in Neural Information Processing Systems. D. Touretzky, editor. Morgan Kaufmann, San Mateo, CA*. pp. 485–492. [9]

Wörgötter, F. and B. Porr: 2005, 'Temporal sequence learning, prediction and control - A review of different models and their relation to biological mechanisms'. *Neural Comp* **17**, 245–319. [81]

Zhang, G. and P. H. W. Leong: 2005, 'Ziggurat-based hardware Gaussian random number generator'. In: *in Proc. IEEE Int. Conf. Field-Programmable Logic and its Applications, 2005*. pp. 275–280. [115]

Zhang, M., S. Vassiliadis, and J. Delgago-Frias: 1996, 'Sigmoid generators for neural computing using piecewise approximations'. *IEEE Trans. Comput.* **45**(9), 1045–1049. [118]