ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Third generation neural networks: formalization as timed automata, validation and learning

Tesi in:
Linguaggi Di Programmazione E Modelli Computazionali

Relatore:
Chiar.mo Prof. *Gianluigi Zavattaro*

Presentata da:
Giovanni Ciatto

Correlatori:
Dott.ssa *Cinzia Di Giusto*
Dott.ssa *Elisabetta De Maria*

Sessione III
A.A. 2015/2016

Abstract. In questa tesi viene mostrato come le reti neurali *spiking*, note anche come reti neurali di terza generazione, possano essere formalizzate usando reti di automi temporizzati. Tali reti neurali, a differenza di quelle di seconda generazione (e.g. reti multilivello con funzione di attivazione sigmoidale), considerano anche la dimensione temporale nell'evoluzione della loro computazione. Sono mostrate due possibili formalizzazioni, sincrona e asincrona, del modello di neurone "discrete leaky integrate and fire": in entrambi i casi, i neuroni sono modellati come automi temporizzati che restano in attesa di impulsi su un dato numero di canali di ingresso (sinapsi) per poi aggiornare il proprio potenziale tenendo conto degli input presenti e passati, opportunamente modulati dai pesi delle rispettive sinapsi e tanto più influenti quanto più recenti. Se il potenziale corrente supera una certa soglia, l'automa emette un segnale broadcast sul suo canale di uscita. Dopo ogni emissione, gli automi sono vinciolati a rimanere inattivi per un periodo *refrattario* fissato, alla fine del quale il potenziale è azzerato. Nel modello asincrono, si assume che gli impulsi in ingresso siano molto frequenti ma se ne impone l'ordinamento: non sono ammessi ingressi contemporanei. Nel modello sincrono, tutti gli impulsi ricevuti all'interno del medesimo *periodo di accumulazione* sono considerati simultanei. Una rete di neuroni è ottenuta eseguendo in parallelo più automi: questi devono condividere i canali in maniera da riflettere la struttura della rete. Le sequenze di impulsi da dare in pasto alle reti sono a loro volta speficate tramite automi temporizzati. È possibile generare tali automi per mezzo di un procedimento automatico, a partire da un linguaggio, appositamente definito, che modella sequenze, al più infinite, di impulsi e pause. Il modello sincrono è validato rispetto alla sua capacità (e incapacità) di riprodurre alcuni comportamenti (relazioni tipiche tra ingressi e uscite) ben noti in letteratura. La formalizzazione basata sugli automi temporizzati è poi sfruttata per trovare un assegnamento per i valori dei pesi sinaptici di una rete neurale in maniera da rendere quest'ultima capace di riprodurre un comportamento dato, espresso da una formula di logica temporale. Tale risultato è raggiunto per mezzo di un algoritmo che, previa identificazione degli errori commessi dai neuroni di output nel produrre l'uscita attesa, permetta di applicare delle azioni correttive sui pesi delle loro sinapsi in ingresso. Le informazioni sulle azioni correttive adeguate vengono poi propagate all'indietro verso gli altri neuroni della

rete. Questo processo è ripetuto fintanto che la rete non si dimostri capace di riprodurre il comportamento desiderato. Due sono gli approcci implementativi presentati: uno basato sulla simulazione e uno basato sul model-checking.

Keywords: Spiking neural networks, timed automata, supervised learning, CTL, model-checking

Acknowledgments

This thesis is the result of a six-months internship at the I3S laboratory of the University of Nice–Sophia Antipolis (France), which I consider a life-changing experience.

I wish to thank Prof. Gianluigi Zavattaro for offering me such a great opportunity, for his support, for his trust in me and my capabilities.

I wish to thank Prof. Cinzia Di Giusto and Prof. Elisabetta De Maria for supervising my work, supporting me in a foreign country, and for the so many things I have learned about theoretical informatics and academic life. A special acknowledgment goes to Prof. Di Giusto for her patience and perseverance, for her precious suggestions, critics, encouragements, and the for random thoughts we shared.

I wish to thank my new friends Selma Souihel and Marco Benzi for the so many times we had fun together, for their company and their support, and for sharing their cultures with me. I sincerely hope we will eventually meet again.

I wish to thank my friends and colleagues Massimo Neri and Federico Fucci, the former for indirectly being the cause of such an adventure, and the latter for writing me almost everyday keeping alive my connection with the beloved city of Cesena.

Finally, I wish to thank my parents, Anna and Carmelo, and my sister Irene, for always supporting me and my decisions both emotionally and economically. I owe them everything and it is thanks to them if I did it this far.

Giovanni Ciatto, March 16, 2017

Contents

Chapter 1

Introduction

Researchers have been trying to reproduce the behavior of the brain for over half a century: on one side they are studying the inner functioning of neurons — which are its elementary components —, their interactions and how such aspects participate to the ability to move, learn or remember, typical of living beings; on the other side they are emulating nature trying to reproduce such capabilities e.g., within robot controllers, speech/text/face recognition applications etc.

In order to achieve a complete comprehension of the brain functioning, both neurons behavior and their interaction must be studied. Historically, interconnected neurons, "neural network", have been naturally modeled as directed weighted graphs where vertices are computational units receiving inputs by a number of ingoing arcs, called *synapses*, elaborating it, and possibly propagating it over of outgoing arcs. Several inner models of the neuron behavior have been proposed: some of them make neurons behave as binary threshold gates, other ones exploit a sigmoidal transfer function, while, in a number of cases, differential equations are employed.

According to [24, 27], three different and progressive *generations* of neural networks can be recognized: (i) first generation includes discrete and threshold based models (e.g., McCulloch and Pitt's neuron [25]); (ii) second generation consists of real valued and sigmoidal-based models, which are nowadays heavily employed in machine learning related tasks because of the existence of powerful learning algorithms (e.g., error back-propagation [29]); (iii) third

generation, which is the focus of our work, consists of a number of models that, in addition to stimuli magnitude and differently from previous generations, take time into account.

Models from the third generation, also known as “spiking neural networks”, are weighted directed graphs where arcs represent synapses, weights serve as synaptic strengths, and vertices correspond to “spiking neurons”. The latter ones are computational units that may emit (or *fire*) output impulses (*spikes*) taking into account input impulses strength and their occurrence instants. Models of this sort are of great interest not only because they are closer to natural neural networks behavior, but also because the temporal dimension allows to represent information according to various *coding schemes* [27, 28]: e.g., the amount of spikes occurred within a given time window (*rate* coding), the reception/absence of spikes over different synapses (*binary* coding), the relative order of spikes occurrences (*rate rank* coding) or the precise time difference between any two successive spikes (*timing* code). A number of spiking neuron models have been proposed into the literature, having different complexities and capabilities. In [21] spiking neuron models are classified according to some *behaviors* (i.e., typical responses to an input pattern) that they should exhibit in order to be considered biologically relevant. For example the leaky-integrate-and-fire (LI&F) model [22], where past inputs relevance exponentially decays with time, is one of the most studied neuron models because of its simplicity [21, 27], while the Hodgkin-Huxley (H-H) model [18] is one of the most complex and important within the scope of computational neuroscience, being composed by four differential equations comparing the neuron to an electrical circuit. For instance, two behaviors that every model is able to reproduce are the *tonic spiking* and *integrator*: the former one describes neurons producing a periodic output if stimulated by a persistent input, the latter one illustrates how temporally closer input spikes have a greater excitatory effect on neurons potential, making them able to act as coincidence detectors. As one may expect, the more complex the model, the more behaviors it can be reproduce, at the price of greater computational cost for simulation and formal analysis; e.g., the H-H model can reproduce all behaviors, but the simulation process is really expensive even for just a few neurons being simulated for a small amount of time [21].

Our aim is to produce a neuron model being meaningful from a biological point of view but also amenable to formal analysis and verification, that could be therefore used to detect non-active portions within some network (i.e., the subset of neurons not contributing to the network outcome), to test whether a particular output sequence can be produced or not, to prove that a network may never be able to emit, or assess if a change to the network structure can alter its behavior; or investigate (new) learning algorithms which take time into account.

In this work, we take the discretized variant of LI&F introduced in [13] and we encode it into timed automata. We show how to define the behavior of a single neuron and how to build a network of neurons. Finally, we show how to verify properties of the designed system via model-checking.

Timed automata are finite state automata extended with timed behaviors: constraints are allowed limiting the amount of time an automaton can remain within a particular state, or the time interval during which a particular transition may be enabled. Timed automata networks are sets of automata running in parallel and interacting by means of *channels*.

Our modeling of spiking neural network consists of a timed automata networks where each neuron is an automaton alternating between two states: it accumulates the weighted sum of inputs, provided by a number of ingoing weighted synapses, for a given amount of time, and then, if the *potential* accumulated during the last and previous accumulation periods overcomes a given threshold, the neuron fires an output over the outgoing synapse. Synapses are channels shared between the timed automata representing neurons, while *spike* emissions are represented by *synchronizations* occurring over such channels. Timed automata can be exploited to produce or *recognize* precisely defined spike sequences, too.

The biophysical behaviors mentioned above are interpreted as *computational tree logic* (CTL) formulae and are tested in `Uppaal` [4] that provides an extended modeling language for automata, a simulator for step-by-step analysis and a subset of CTL for systems verification.

Finally, we exploit our automata-based modeling to propose a new methodology for parameter learning in spiking neural networks, namely the *advice back-propagation* (ABP) approach. In particular, ABP allows to find an as-

signment for the synaptic weights of a given neural network making it able to reproduce a given behavior. We take inspiration from SpikeProp [8], a variant of the well known back-propagation algorithm [29] used for supervised learning in second generation networks. SpikeProp reference model takes into account multi-layered cycle-free spiking neural networks and aims at training them to produce a given output sequence for each class of input sequences. The main difference with respect to our approach is that we are considering here a discrete model and our networks are not multi-layered. We also rest on Hebb's learning rule [17] and its time-dependent generalization, the spike-timing dependent plasticity (STDP) rule [30]: they both act locally, with respect to each neuron, i.e., no prior assumption on the network topology is required in order to compute the weight variations for some neuron input synapses. Differently from STDP, our approach takes into account not only the recent spikes but also some external feedback, the *advices*, in order to determine which weights should be modified and whether they must be increased or decreased. Moreover, we do not prevent excitatory synapses from becoming inhibitory (or vice versa), which is usually a constraint for STDP implementations. A general overview on spiking neural network learning approaches and open problems in this context can be found in [16].

The rest of this thesis is organized as follows. Chapter 2 exposes the theoretical background. It explains the differences between the three neural networks generations and describes our reference model, the LI&F. It provides an overview over existing learning approaches within the context of second and third neural network generations. Then, it recalls definitions of timed automata networks, CTL and model-checking. Chapter 3 shows how spiking neural networks are encoded into timed automata networks, how inputs and outputs are defined by means of an *ad-hoc* language and encoded into automata, as well. Chapter 4 describes how we validated our formalized model by providing formal proofs for the behaviors listed above. It also discuss the intrinsic properties of our model, e.g., the *maximum threshold* or the *lack of inter-spike memory*. Then, a number of extension are proposed, aiming to endow our model with further capabilities, such as the ability to emit *bursts*. In chapter 5, we introduce the *learning problem* and our ABP approach from an abstract point of view. We then provide two possible ways to realize it: the first

one is simulation-oriented and the second one is model-checking-oriented. Finally, chapter 6 summarizes our results and presents some future research directions.

Publications. This thesis has been the starting point for two scientific papers. The first one [9], describing our formalization of leaky-integrate-and-fire neural networks, has been *accepted* by the *ASSB* student workshop. The second one, concerning our approach to the learning problem, has been *submitted* to the international conference *Coordination 2017*.

Chapter 2

Theoretical background

In this chapter we present or recall a number of concepts which are discussed in the rest of this thesis.

We begin by describing a well established categorization of neural networks within three consecutive *generations*, then we recall Maass' widely general definition of spiking neural networks (i.e., a network from the third generation) and present the discrete leaky-integrate-and-fire neuron model which is extensively discussed and exploited in the following chapters. We also provide an overview about supervised and unsupervised learning within the scope of spiking neural networks and we compare learning approaches between second and third generation neural networks.

Then, we recall the definition and semantics of timed automata and timed automata networks, which compose the conceptual framework we exploited the most in our spiking neural networks formalization proposal. Finally, we briefly present the definition of the CTL temporal logics, the model-checking problem, and we provide an intuition of their semantics.

2.1 Neural networks

Neural networks are directed weighted graphs were nodes are *computational units*, also known as *neurons*, and edges represents *synapses*, i.e., connections between some neuron output and some other neuron input. Several models exist into the literature and they differ on the signals that neurons

emit/accept and on the way such signal are elaborated. An interesting classification has been proposed in [24] which distinguishes three different generations of neural networks:

1. Network models within the *first generation* handle discrete inputs and outputs and their computational units are threshold-based transfer functions; this includes McCulloch and Pitt's threshold gate [25], the perceptron [15], Hopfield networks [19] and Boltzmann machines [2].

2. *Second generation* models, instead, exploit real valued activation functions, e.g., the sigmoid function, accepting and producing real values: a well known example is the multi-layer perceptron [11, 29]. According to a common interpretation, the real-valued outputs of such networks represents the firing rates of natural neurons [27].

3. Networks from the *third generation* are known as spiking neural networks. They extend second generation models treating time-dependent and real valued signals often composed by *spike trains*. Neurons may fire output spikes according to threshold-based rules which take into account input spikes magnitude and occurrence time [27].

The core of our analysis are spiking neural networks. Because of the introduction of timing aspects (in particular, observe that information is represented not only by spikes magnitudes but also by their occurrence timings) they are considered closer to the actual brain functioning than models from previous generations.

We adopt Maass' definition (see [24] or [23]) because it is a widely general template which can be specialized in more fine-grained characterizations by providing additional constraints. Spiking neural networks are modeled as directed weighted graphs where vertices are computational units and edges represents *synapses*. The signals propagating over synapses are *trains of impulses*: *spikes*. The particular wave form of impulses must be specified by model instances. Synapses may modulate such signals according to their weight or they could introduce some propagation delay. Synapses are classified according to their weight as *excitatory*, if it is positive, or *inhibitory* if negative.

Computational units represents *neurons*, whose dynamics is governed by two variables: the *membrane potential* (or, simply, *potential*) and the *threshold*. The former one depends on spikes received by neurons over ingoing synapses, after being modulated and/or delayed. Both current and past spikes are taken into account even if old spikes contribution is lower. The latter may vary according to some rule specified by instances. The neuron outcome is controlled by the algebraic difference between the membrane potential and the threshold: it is enabled to fire (i.e., emit an output impulse over *all* outgoing synapses) only if such difference is non-negative. Immediately after each emission the neuron membrane is reset.

Another important constraint, typical of spiking neural networks, is the *refractory period*: each neuron is unable to fire for a given amount of time after each emission. Such behavior can be modeled preventing the potential to reach the threshold either by keeping the former low or the latter high.

More formally:

Definition 2.1 (Spiking neural network)**.** Let $E = \{f \mid f : \mathbb{R}_0^+ \to \mathbb{R}\}$ be the set of functions from continuous time to reals, then a *spiking neural network* is a tuple (V, A, ε), with:

- V is the set of *spiking neurons*,
- $A \subseteq V \times V$ is the set of *synapses*,
- $\varepsilon : A \to E$ is function assigning to each synapse $(u, v) \in A$ a *response function* $\varepsilon_{u,v} \in E$.

We distinguish three disjoint sets V_i of input neurons, V_{int} of intermediary neurons, and V_o of output neurons, with $V = V_i \cup V_{int} \cup V_o$.

Definition 2.2 (Spiking neuron)**.** A *spiking neuron* v is a tuple $(\theta_v, p_v, \tau_v, y_v)$, where:

- $\theta_v \in E$ is the *threshold* function,
- $p_v \in E$ is the [membrane] *potential* function,
- $\tau_v \in \mathbb{R}_0^+$ is the *refractory period* duration,

- $y_v \in E$ is the *outcome function*.

The dynamics of a neuron v is defined by means of the set of its firing times $F_v = \{t_1, t_2, \ldots\} \subset \mathbb{R}_0^+$, also called *spike train*. Such set is defined recursively: t_{i+1} is computed as a function of the value $y_v(t - t_i)$, i.e., the outcome of v since the instant t_i. For instance, a trivial model may consider $y_v(t - t_i)$ to be greater than 0 for the instant t_{spike}, defined as the smallest t such that $p_v(t - t_i) \geq \theta_v(t - t_i)$. Analogously, in a stochastic model, the value $y_v(t - t_i)$ may govern the firing probability for neuron v.

The after-spike refractory behavior is achieved by making it impossible for the potential to reach and overcome the threshold. This can be modeled in two ways: (i) making any neuron unable to reach the threshold, e.g., by constraining each threshold function θ_v such that: $\theta_v(t - t') = +\infty$ if $t - t' < \tau_v$ for each $t' \in F_v$; (ii) making any neuron ignore its inputs, e.g., by constraining each potential function p_v such that: $p_v(t-t') = 0$ if $t-t' < \tau_v$ for each $t' \in F_v$,

Each response function $\varepsilon_{u,v}$ represents the impulse propagating from neuron u to neuron v and can be used to model synapse-specific features, like delays or noises. For instance, a model may allow signals to be modulated and delayed by defining $\varepsilon_{u,v}$ as follows:

$$\varepsilon_{u,v}(t) = w_{u,v} \cdot y_u(t - d_{u,w})$$

where $w_{u,v}$ is a *synaptic weight* representing the strength of the synapse (u, v), and $d_{u,v}$ is the *propagation delay* introduced by such a synapse.

For each neuron $v \in V_{int} \cup V_o$, the potential function p_v takes into account the response function value $\varepsilon_{u,v}(t - t')$, for each previous or current firing time $t' \in F_v : t' \leq t$ and for each input synapse (u, v); so the current potential may be influenced by both the current and the previous inputs. For each neuron $v \in V_i$, the set F_v is assumed to be given as input for the network. For all neuron $v \in V_o$, the set F_v is considered an output for the network.

Such a definition is deliberately abstract because there exist into the literature a number of models fitting it, differing in the way they handle e.g., potentials, signal shapes, etc.

Some authors [21,27] classify the models presented in literature according to their *biophysical plausibility*. Estimating such a feature for a given model

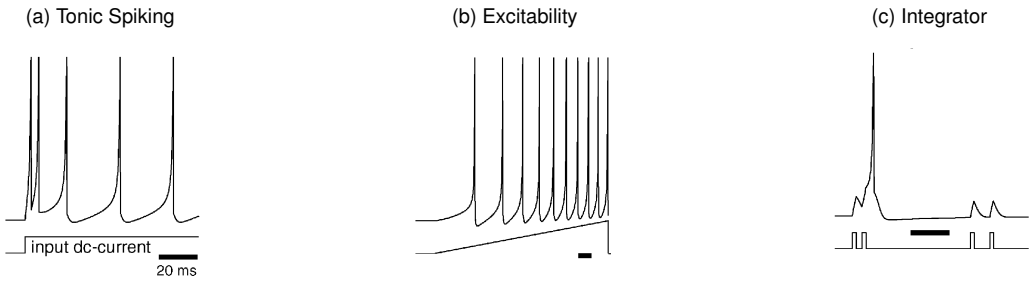


Figure 2.1: Summary and graphical representation of some of the most interesting neuron behaviors we mention within this thesis, taken from [21]. Each cell shows the neuron response (in the upper part) to a particular input current (in the lower part).

Models	tonic spiking	phasic spiking	phasic bursting	mixed mode	spike frequency adaptation	class 1 excitable	class 2 excitable	spike latency	subthreshold oscillations	resonator	integrator	rebound spike	rebound burst	threshold variability	bistability	DAP	accomodation	inhibition induced spiking	inhibition induced bursting
Integrate and Fire	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-
Integrate and Fire or Burst	+	+	+	-	+	+	-	-	-	-	+	+	+	-	+	+	-	-	?
Quadratic Integrate And Fire	+	-	-	-	-	+	-	+	-	-	+	-	-	+	+	-	-	-	-
Izhikevich's Model	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Hodgkin-Huxely	+	+	?	?	+	+	+	+	+	+	+	+	+	+	+	+	+	+	?

Figure 2.2: Comparison between several neuron models taking into account the amount of behaviors from figure 2.1 the model can reproduce. See [21] for more detailed descriptions and for references.

may be a complex task since it is not well formalized. According to Izhikevich, there exists a set of *behaviors*, some shown in figure 2.1, which a neuron may be able to reproduce. A behavior is basically a well-featured input-output relation and a model is said to be able to reproduce it if there exists at least one instance of the model presenting a comparable outcome when receiving an alike input. The author also proposes to use the amount of behaviors a model can reproduce as a measure of its biophysical plausibility.

As far as our work is concerned, the most interesting results are about the *integrate-and-fire* model capabilities. Indeeds, instances of this model should be able to reproduce the following behaviors:

Tonic spiking: as a response to a persistent input, the neuron periodically fires spikes as output.

Excitability: the emission rate of a neuron linearly increases with its input

magnitude.

Integrator: a neuron of this sort prefers high-frequency inputs: the higher the frequency the higher its firing probability; it may act as inputs coincidence detector.

2.1.1 Leaky-integrate-and-fire model

Since our aim is to define a model being simple enough to be inspectable through model-checking techniques but also complex enough to be biophysically meaningful, we focused on the leaky-integrate-and-fire (LI&F), which is one of the simplest and most studied model of biological neuron behavior (see [21] and [27]), whose original definition is traced back to [22].

We adopt the formulation proposed in [13]. It is a discretized model, amenable to formal verification, where time progresses discretely, signals are boolean-valued even if potentials are real-valued, thresholds are constant over time and potentials vary according to both the currently and previously received spikes. Synapses do not introduce any delay. They instantaneously propagate the spikes from the producing neurons to the consuming ones, modulating them according to their weight.

Definition 2.3 (LI&F network)**.** A *leaky-integrate-and-fire network* $(V,\, A,\, W)$ is a particular case of spiking neural network $(V,\, A,\, \varepsilon)$, where:

- $V = V_i \cup V_{int} \cup V_o$ is the set of neurons such that each $v \in V_{int} \cup V_o$ is a *leaky-integrate-and-fire* neuron,
- $A \subseteq V \times V$ is the set of *synapses* such that $(v,\, v) \notin A,\ \forall v \in V$,
- $W : A \to \mathbb{R}$ is the *weight* function assigning to each synapse $(u,\, v)$ a weight $w_{u,v} = W(u,\, v) \in [\,-1,\, 1\,]$ such that the response functions assigned by ε share the form:

$$\varepsilon_{u,v}(t) = w_{u,v} \cdot y_u(t)$$

Definition 2.4 (LI&F neuron)**.** A *leaky-integrate-and-fire neuron* $v = (\theta_v^{(0)},\, \lambda_v,\, p_v,\, \tau_v,\, y_v)$ is a particular case of spiking neuron, with:

- $\theta_v^{(0)} \in \mathbb{R}$ is the constant threshold value, such that $\theta_v(t) = \theta_v^{(0)},\ \forall t$,
- $\lambda \in [0, 1]$ is the *leak factor*,
- $p_v : \mathbb{N} \to \mathbb{R}$ is the *potential* function, defined as

$$p_v(t) = \sum_{(u,\, v) \in A} \varepsilon_{u,v}(t) + \lambda_v \cdot p_v(t-1) \tag{2.1}$$

  where $p_v(0) = 0$,
- $\tau_v \in \mathbb{N}^+$ is the *refractory period* duration,
- $y_v : \mathbb{N} \to \{0, 1\}$ is the *outcome function*, defined as

$$y_v(t) = \begin{cases} 1 & \text{if } p_v(t) \geqslant \theta_v \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

We refer to the value $\varepsilon_v(t) = \sum_{(u,\, v) \in A} \varepsilon_{u,v}(t)$ as the *sum of weighted inputs* of neuron v at time unit t, because, according to definition 2.3, the following equivalence holds true:

$$\varepsilon_v(t) = \sum_{(u,\, v) \in A} w_{u,v} \cdot y_u(t)$$

Thus, we can rewrite equation 2.1 in a more concise way:

$$p_v(t) = \varepsilon_v(t) + \lambda_v \cdot p_v(t-1) \tag{2.3}$$

From the point of view of the *consuming* neuron v, we say that, whenever $y_u(t) = 1$ for some u, an input spike *propagates*, *occurs* or *is received* over the synapse $(u,\, v)$. From the point of view of the *producing* neuron u, we say that, whenever $y_u(t) = 1$, the output spike *propagates*, *occurs* or *is sent* over the synapses $(u,\, v)$, for all v. Synapses do not introduce any propagation delay, but they modulate the spikes according to their weights. A synapse $(u,\, v)$ is said to be *excitatory* if $w_{u,v} \geq 0$, *inhibitory* otherwise.

Finally, let t_f be the last time unit where v emitted a spike, i.e., $y_v(t_f) = 1$, then, for a given *refractory period* $\tau_v \in \mathbb{N}$, $p_v(t_f + k) = 0,\ \forall k < \tau$. Please note that during *any* refractory period: (i) the neuron cannot increase its potential; (ii) it cannot emit any spike, since $p_v(t_f + k) < \theta_v$; (iii) any received spike is *lost*, i.e., it has no effect on neuron potential.

Remark. There exists an explicit version for equation 2.3, that is[1]:

$$p(t) = \sum_{k=0}^{t} \lambda^k \cdot \varepsilon(t-k) \tag{2.4}$$

which clearly shows how previous inputs relevance *exponentially* decays as time progresses. Such formulation is achieved as follows:

$$\begin{array}{rcccl}
p(0) & = & \varepsilon(0) & = & \lambda^0 \cdot \varepsilon(0) \\
p(1) & = & \varepsilon(1) + \lambda \cdot p(0) & = & \lambda^0 \cdot \varepsilon(1) + \lambda^1 \cdot \varepsilon(0) \\
p(2) & = & \varepsilon(2) + \lambda \cdot p(1) & = & \lambda^0 \cdot \varepsilon(2) + \lambda^1 \cdot \varepsilon(1) + \lambda^2 \cdot \varepsilon(0) \\
p(3) & = & \varepsilon(3) + \lambda \cdot p(2) & = & \lambda^0 \cdot \varepsilon(3) + \lambda^1 \cdot \varepsilon(2) + \lambda^2 \cdot \varepsilon(1) + \lambda^3 \cdot \varepsilon(0) \\
 & \vdots & & & \\
p(t) & = & \varepsilon(t) + \lambda \cdot p(t-1) & = & \sum_{k=0}^{t} \lambda^k \cdot \varepsilon(t-k)
\end{array}$$

2.2 Supervised and unsupervised learning with spiking neural networks

In this section we provide an overview about *learning* approaches within the context of (spiking) neural networks.

Neural network models often rely upon a number of parameters, e.g., in definitions 2.3 and 2.4 we introduce synaptic weights, thresholds, leak factors, and refractory periods. By "learning", we mean the process of searching for an assignment of such parameters according to a given criterion. To the best of our knowledge, all the criteria presented into the literature focus on the synaptic weights instead of the whole gamma of parameters employed by their respective reference models.

Traditionally, learning approaches are categorized as *supervised* or *unsupervised*. In a supervised learning process, the network is fed by a number of inputs and its outcome is compared with an equal number of expected outputs. In this case, parameters are varied trying to minimize the difference between the actual and the expected outputs. Well known approaches to supervised learning are, for instance: *error back-propagation*, [29], for second

[1] subscripts are omitted

generation neural networks, and its time-aware descendants, *SpikeProp* [8] and *back-propagation-through-time* (BPTT) [26], for spiking neural networks. Conversely, in an unsupervised learning process, inputs are not associated to any expected outcome, and the parameters of the network are varied in order to capture similarities or co-occurrences between patterns of inputs. Two well known rules, namely the Hebb rule [17] and its time-aware extension, *spike-timing dependent plasticity* (STDP) [30], were conceived as nature-inspired models of synaptic plasticity, i.e., synaptic strength variation rules, but they can also be considered unsupervised learning approaches, from a computational point of view.

Back-propagation and its derivates. More than two decades ago, Rumelhart et al. introduced the *back-propagation* algorithm [29], that is nowadays one of the most known and studied supervised learning approaches for networks within the second generation [27].

Back-propagation assumes the network topology to be multi-layered, non-recurrent and feed-forward, i.e., neurons are organized in *layers*. Each ingoing edge comes from a neuron within the previous layer and each outgoing edge is directed to a neuron within the next layer. This is true for all but the first layer — containing the input neurons — and last one — containing output neurons. Being a supervised approach, a number of inputs are provided to the network, and its outcome is compared with the expected outputs. An error function, measuring the squared difference between actual and expected outputs with respect to the weights of the network, is minimized by descending its gradient at each step of the algorithm, i.e., the weights are varied in order to reduce the error currently performed by the network. Sadly, it cannot guarantee to reach the global minimum of such an error function, since it may get stuck within some local minima.

Several attempts of applying similar approaches to third generation neural networks have been proposed, e.g., SpikeProp [8] or BPTT [26]. The greatest obstacle when trying to adapt back-propagation to spiking neural networks is the inherently discontinuous nature of spikes and spiking neurons with respect to time. To overcome such a limitation, spikes are modeled by impulsive but still continuous functions, asymptotically decaying to zero. For instance,

SpikeProp employs the function $\varepsilon(t) = t \cdot \nu^{-1} \cdot e^{1-t/\nu}$, where ν dictates how slowly impulses decay to zero. Another obstacle may be information representation. Back-propagation is conceived for second generation neural networks, where inputs and outputs are represented by real numbers. As we outlined in chapter 1, there exist several ways to represent information by means of spikes, e.g., temporal, rate, count or rank coding [27]. Information representation is an important trait when the actual and expected outputs come to be compared. Thus, supervised learning approaches within the scope of spiking neural networks must explicitly or implicitly assume a particular coding. For instance, SpikeProp employs temporal coding, i.e., the information carried by a spike is encoded by the time elapsed since the previous spike.

SpikeProp, too, exploits gradient descendant to minimize an error function. The learning process affects both synaptic weights and *synaptic delays*, which are part of SpikeProp reference model. Another difference with back-propagation is that time differences between the actual and expected spikes are taken into account by such a function.

Hebbs rule and STDP. The principle behind all Hebb's rule implementations is postulated in [17, p. 62]:

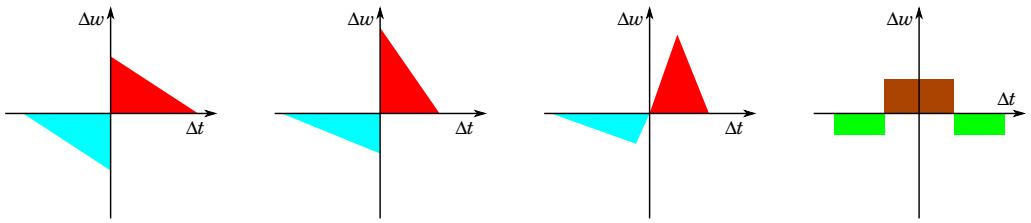
> *When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*

For first and second generation networks, the following *learning rule* is classically presented as an implementation of such a principle:

$$\Delta w_i = \eta \cdot x_i \cdot y$$

where Δw_i is the *weight variation* of the i-th input synapse for some neuron N, η is the *learning rate*, x_i is the value of the i-th input of N, and y is the output of N. The rule states the synapse connecting two neurons must be strengthened *proportionally* to the product of the outputs of the two neurons.

STDP is considered the time-dependent counterpart of Hebb's rule for spiking neural networks [27, 30]. It is strongly believed [7, 12] that the weight



(a) Greater variation for smaller Δt. Symmetric for $\Delta t > 0$ and $\Delta < 0$.

(b) Greater variation for smaller Δt. Asymmetric for $\Delta t > 0$ and $\Delta < 0$.

(c) Smaller variation for near-zero and greater Δt. Asymmetric for $\Delta t > 0$ and $\Delta < 0$.

(d) Strengthening for smaller $|\Delta t|$, weakening for greater $|\Delta t|$.

Figure 2.3: Window functions patterns for STDP, from [27]. In the three cases 2.3a, 2.3b and 2.3c, the synapse is strengthened for positive Δt and weakened otherwise.

variation of a synapse is strictly correlated with the *relative timing* of the *post-synaptic* spike with respect to the *pre-synaptic* one. From the point of view of a synapse (u, v), the pre-synaptic spike is the one emitted by neuron u at time t_{pre}, and the post-synaptic spike is the one emitted by v at time t_{post}. As summarized in [27], the general formulation of STDP is:

$$\Delta w_{u,v} = f(\Delta t)$$

where $\Delta w_{u,v}$ is the *weight variation* of synapse (u, v), $\Delta t = t_{post} - t_{pre}$, and f is a *window function*, i.e., a smoothed function matching one of the patterns in figure 2.3. Such a rule states the weight of a synapse is varied according to the difference of occurrence times between the pre- and post-synaptic spikes. According to [30], the window function f is commonly implemented by means of pairs of negative exponential functions, built in such a way to prevent excitatory synapses to become inhibitory and vice versa.

2.3 Timed automata

Timed automata [3, 5] are a powerful theoretical formalism for modeling and verification of real time systems. Next, we recall their definition and semantics, their composition into timed automata networks as well as the composed network semantics. We conclude with an overview on the extension introduced by the specification and analysis tool `Uppaal` [4] that we have employed here.

A timed automaton is a finite state machine extended with real-valued clock variables. Time progresses synchronously for all clocks, even if they can be reset independently when edges are fired. States, also called *locations*, may be enriched by *invariants*, i.e., constraints on the clock variables limiting the amount of time the automaton can remain into the constrained location. Edges are enriched too: each one may be labeled with *guards*, i.e., constraints over clocks which enable the edge when they hold, and *reset sets*, i.e., sets of clocks that must be reset to 0 when the edge is fired. Symbols, optionally consumed by edge firings, are here called *events*. More formally:

Definition 2.5 (Timed automaton)**.** Let X be a set of symbols, each identifying one clock variable, and let G be the set of all possible guards: conjunctions of predicates having the form $x \; o \; n$ or $(x - y) \; o \; n$, where $x \in X$, $n \in \mathbb{N}$ and $o \in \{>, \geqslant, =, \leqslant, <\}$. Then a timed automaton is a tuple $(L, l^{(0)}, X, I, A, E)$ where:

- L is a finite set of *locations*;
- $l^{(0)} \in L$ is the initial location;
- $I : L \to G$ is a function assigning guards to locations;
- A is a set of symbols, each identifying an event;
- $E \subseteq L \times (A \cup \{\varepsilon\}) \times G \times 2^X \times L$ is a set of edges, i.e., tuples (l, a, g, r, l') where:
  - l, l' are the source and destination locations, respectively,
  - a is an event,
  - g is the guard,
  - $r \subseteq X$ is the reset set.

In order to present de semantics of timed automata, we need to recall the definition of labeled transition systems, which are a formal way to describe formal systems semantics. They consist of directed graph where vertices are called *states*, since each of them represents a possible state of the source system, and edges are referred as *transitions*, since they represent the allowed

transitions, from a state to another, for the source system. Edges are decorated through labels representing, e.g., the action firing a particular transition, the guards enabling it or some operation to be performed on their firing.

Definition 2.6 (*Labeled* transition system)**.** Let Λ be a set of *labels*, then a labeled transition system is a tuple $\mathcal{M} = (S,\, s_0,\, \longrightarrow)$ where:

- S is a set of states,
- $s_0 \in S$ is the initial state,
- $\longrightarrow \subseteq S \times \Lambda \times S$ is a *transition relation*, i.e., the set of allowed transitions, having the form $s \xrightarrow{\lambda} s'$, where
  - $s, s' \in S$ are the source and destination state, respectively;
  - $\lambda \in \Lambda$ is a label.

For what concerns timed automaton semantics, clocks are evaluated by means of an *evaluation* function $u : X \to \mathbb{R}_0^+$ assigning a non-negative time value to each variable in X. With an abuse of notation, we will write u meaning $\{u(x) : x \in X\}$, the set containing the current evaluation for each clock; $u + d$ meaning $\{u(x) + d : x \in X\}$, for some given $d \in \mathbb{R}_0^+$, i.e., the clock evaluation where every clock is increased of d time units respect to u. Similarly, for any reset set $r \subseteq X$, we will use the notation $[r \mapsto 0]u$ to indicate the assignment $\{x_1 \mapsto 0 : x_1 \in r\} \cup \{u(x_2) : x_2 \in X - r\}$. We will then call u_0 the function such that $u_0(x) = 0\ \forall x \in X$ and $\mathbb{R}^X$ the set of all possible clocks evaluations. Finally, we will write $u \models I(l)$ meaning that, for some given location l, every invariant is satisfied by the current clock evaluation u.

Let $T = (L,\, l^{(0)},\, X,\, I,\, A,\, E)$ be a timed automaton. Then, the semantics is a *labelled transition system* $(S,\, s_0,\, \rightarrow)$ where:

- $S \subseteq L \times \mathbb{R}^X$ is the set of possible states, i.e., couples $(l,\, u)$ where l is a location and u an evaluation function;
- $s_0 \in S$ is the system initial state which by definition is $(l^{(0)},\, u_0)$;
- $\rightarrow \subseteq S \times (\mathbb{R}_0^+ \cup A \cup \{\varepsilon\}) \times S$ is a *transition relation* whose elements can be:

- **Delays:** modeling an automaton remaining into the same location for some period. This is possible only if the location invariants holds for the entire duration of such a period.
  Transitions of this sort share the form $(l,\, u) \xrightarrow{d} (l,\, u + d)$, for some $d \in \mathbb{R}_0^+$, and they are subjected to the following constraint: $(u+t) \models I(l),\ \forall t \in [0,\, d]$.
- **Event occurrences:** modeling an automaton instantaneously moving from one location to another. This is possible only if an *enabled* edge from the source location to the destination one is defined. An edge is enabled only if its guards hold and if the destination invariants keep holding after the clocks in the edge reset set have been reset.
  Transitions of this sort share the form $(l,\, u) \xrightarrow{a} (l',\, u')$, where $a \in A \cup \{\varepsilon\}$. They are subjected to the following constraint: $\exists e = (l,\, a,\, g,\, r,\, l') \in E$ such that $u \models g$ (i.e., all guards g are satisfied by the clock assignments u in l) and $u' = [r \mapsto 0]u$ (i.e., the new clock assignments u' are obtained by u resetting all clocks in r) and $u' \models I(l')$ (i.e., the new clock assignments u' satisfies all invariants of the destination state l').

Timed automata networks are a *parallel composition* of automata over a common set of clocks and communication channels obtained by means of the parallel operator $\|$. Let $\mathcal{X}$ be a set of clocks and let A_s, A_b be sets of symbols representing *synchronous* and *broadcast* communication channels respectively, such that $A_s \cap A_b = \emptyset$ and let $\mathcal{A} = \{?,\, !\} \times (A_s \cup A_b)$. Events in $\mathcal{A}$ are of two types:

- $?a$ is the event "sending/writing a message over/on channel a",
- $!a$ is the event "receiving/reading a message over/from channel a".

Let $N = T_1 \parallel \cdots \parallel T_n$ be a timed automata network where each $T_i = (L_i,\, l_i^{(0)},\, \mathcal{X},\, I_i,\, \mathcal{A},\, E_i)$ is a timed automaton. Then, its semantics is a *labelled transition system* $(S,\, s_0,\, \rightarrow)$ where:

- $S \subseteq (L_1 \times \ldots \times L_n) \times \mathbb{R}^{\mathcal{X}}$ is the set of possible states, i.e., pairs $(\mathbf{l},\, u)$ where $\mathbf{l}$ is a locations *vector* and u an evaluation function;

- $s_0 \in S$ is the system initial state which by definition is $(\mathbf{l}_0, u_0)$, with $\mathbf{l}_0 = (l_1^{(0)}, \ldots, l_n^{(0)})$;

- $\rightarrow \subseteq S \times (\mathbb{R}_0^+ \cup \mathcal{A} \cup \{\varepsilon\}) \times S$ is a *transition relation* whose elements can be:

  - **Delays:** making all automata composing the network remain in respective locations for some period. This is possible only if all invariants of every automaton hold for the entire duration of such a period.
    Transitions of this sort share the form $(\mathbf{l}, u) \xrightarrow{d} (\mathbf{l}, u + d)$, for some $d \in \mathbb{R}_0^+$, and they are subjected to the following constraint: $(u+t) \models I(\mathbf{l})$ [2] $\forall t \in [0, d]$. Note that time progresses evenly for all clocks and automata.

  - **Synchronous communications (synchronizations):** modeling a message exchange between two different automata. This can happen only if one of them, the *sender*, is enabled to write on some synchronous channel and the other one, the *receiver*, is enabled to read from the same channel. This means the sender must be within a location having an *enabled* outgoing edge decorated by $!a$, and, similarly, the receiver must be within a location having an *enabled* outgoing edge decorated by $?a$.
    Transitions of this sort are in the form $(\mathbf{l}, u) \xrightarrow{a} (\mathbf{l}', u')$, where $a \in A_s$. They are subjected to the following constraint: there exists, for two different $i, j \in \{1, \ldots, n\}$, two edges $e_i = (l_i, !a, g_i, r_i, l_i')$ and $e_j = (l_j, ?a, g_j, r_j, l_j')$ in $E_1 \cup \cdots \cup E_n$ such that $u \models (g_i \wedge g_j)$ and $u' = [(r_i \cup r_j) \mapsto 0]u$ and $u' \models I(\mathbf{l}')$, where $\mathbf{l}' = [l_i \mapsto l_i', l_j \mapsto l_j']\mathbf{l}$; so a synchronous communication makes two automata fire their edges e_i and e_j atomically. If more that a couple of automata can synchronize, one will be chosen non-deterministically.

  - **Broadcast communications:** modeling a message spreading over some channel from a sender automaton to any automaton interested in receiving messages from that channel. The main differ-

[2] with an abuse of notation we write $I(\mathbf{l})$ instead of $I(l_1) \wedge \ldots \wedge I(l_n)$, for any $\mathbf{l} = (l_1, \ldots, l_n)$

ence from synchronizations is that, here, senders can write their message even if no one is ready to receive it: thus senders cannot get stuck and massages can be lost. This transition is possible only if the sender is enabled to write on some broadcast channel a. The set of receiving automata is computed taking into account the ones being within a location having an *enabled* outgoing edge decorated by $?a$. This set must then be filtered, removing those automata which would move to a location whose invariants would be violated by some clock reset caused by this transition.
More formally, transitions of this sort share the form $(\mathbf{l}, u) \xrightarrow{a} (\mathbf{l}', u')$, where $a \in A_b$. They are subject to the following constraint: there exists in $E_1 \cup \cdots \cup E_n$

- an edge $e_i = (l_i, !a, g_i, r_i, l_i')$, for some $i \in \{1, \ldots, n\}$,
- a subset D' containing *all* edges having the form $e_j = (l_j, ?a, g_j, r_j, l_j')$ such that $u \models (g_j)$, where $i \neq j \in \{1, \ldots, n\}$,
- a subset $D = D' - \{e_t \in D' : u'' \models I(\mathbf{l}')\}$, where $u'' = [(r_t) \mapsto 0, \forall t : e_t \in D' \cup \{e_i\}]u$,

thus, for each e_k in $D \cup \{e_i\}$, $u \models (g_k)$ and $u' = [(r_k) \mapsto 0, \forall k]u$ and $u' \models I(\mathbf{l}')$, where $\mathbf{l}' = [l_k \mapsto l_k', \forall k]\mathbf{l}$. So a broadcast communication make a number of edges fire atomically and it only requires an automaton to be enabled to write. If more than one broadcast communication can occur, one is chosen non-deterministically.

– **Moves:** modeling an automaton unconstrained movement from a location to another because of an edge firing. This requires the edge guards to hold within the source state and the destination location invariants to hold after clock resets have been performed.
Such transitions have the form $(\mathbf{l}, u) \xrightarrow{\varepsilon} (\mathbf{l}', u')$, are subject to the following constraint: $\exists e = (l, \varepsilon, g, r, l')$ in $E_1 \cup \cdots \cup E_n$ such that $u \models g$ and $u' = [r \mapsto 0]u$ and $u' \models I(\mathbf{l}')$.

In the rest of this thesis, we actually exploit the definition of timed automata adopted by `Uppaal`. It provides a number of extensions that we describe informally:

- The state of the timed automata is enriched by a set of *bounded integer* or *boolean* variables. Predicates concerning such variables can be part of edges guards or locations invariants, moreover variables can be updated on edges firings but they cannot be assigned to/from clocks. So it is impossible for a variable to assume the current value of a clock and vice versa.

- Locations can be marked as *urgent* meaning that time cannot progress while an automaton remains in such a location: it is semantically equivalent to a locations labeled by the invariant $x \leq 0$ for some clock x where all ingoing edges reset x.

- Locations can also be marked as *committed* meaning that, as for urgent locations, they do not allow the time to progress and they constrain any outgoing or ingoing edge to be fired *before* any edge not involving committed locations. If more than one edge involving committed locations can fire, then one is chosen non-deterministically.

The `Uppaal` modeling language actually includes other features that were exploited. For a more detailed description consider reading [4].

When representing the edges of some timed automaton, we will indicate three sections `G`, `S` and `U` respectively containing `G`uards, communications/`S`ynchronizations and `U`pdates list, where an update can be a clock reset and/or a variable assignment.

2.4 Formal verification

In this section we recall temporal logics and model-checking, two concepts that are pervasively exploited in the rest of this thesis.

Temporal logics are extensions of the first order logic allowing to represent and reason about temporal properties of some given formal system. In this thesis, we adopt the *computational tree logics* (CTL) to express the properties of our systems, hence we now recall its syntax and provide an intuition of its semantics.

Definition 2.7 (CTL syntax)**.** Let P be the variable ranging over atomic propositions, then a CTL formula ϕ is defined by:

$$\begin{array}{rcll} \phi & = & P \mid \mathbf{true} \mid \mathbf{false} & \text{atoms} \\ & & \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \implies \phi \mid \phi \iff \phi & \text{connectives} \\ & & \mid A\psi \mid E\psi & \text{path quantifiers} \\ \\ \psi & = & X\phi \mid F\phi \mid G\phi \mid \phi U \phi & \text{state quantifiers} \end{array}$$

where $\neg$, $\wedge$, $\vee$, $\implies$ and $\iff$ are the usual logic connectives, A and E are *path quantifiers* and X, F, G and U are path-specific *state quantifiers*.

CTL formulae can only contain *couples* of quantifiers, here we give an intuition of their semantics. A formal definition can be fount in [10].

$AG\phi$ **– Always:** ϕ holds in every reachable state

$AF\phi$ **– Eventually:** ϕ will eventually hold at least in one state on every reachable path

$AX\phi$ **– Necessarily Next:** ϕ will hold in every successor state

$A(\phi_1 U \phi_2)$ **– Necessarily Until:** in every reachable path, ϕ_2 will eventually hold and ϕ_1 holds while ϕ_2 is not holding

$EG\phi$ **– Potentially always:** there exists at least one reachable path where ϕ holds in every state

$EF\phi$ **– Possibly:** there exists at least one reachable path where ϕ will eventually hold at least once

$EX\phi$ **– Possibly Next:** there exists at least one successor state where ϕ will hold

$E(\phi_1 U \phi_2)$ **– Possibly Until:** there exist at least one reachable path where ϕ_2 will eventually hold and ϕ_1 holds while ϕ_2 is not holding

The formula $AG(\phi_1 \implies AF\phi_2)$ is a common pattern used to express *liveness properties*, i.e., desirable events which will eventually occur. The formula can be read as: "ϕ_1 always leads to ϕ_2" or "whenever ϕ_1 is satisfied, then

ϕ_2 will eventually be satisfied". Formulae of this sort are sometimes written using the alternative notation $\phi_1 \rightsquigarrow \phi_2$.

Model-checking is an approach to system verification aiming to test whether a given temporal logic formula holds for a given formal system, starting from a given point in time. It generally assumes that a transition system can be build, somehow representing all possible states and all allowed transitions for the given system. The verification process usually consists into exhausting all reachable states from a given *initial state*, searching for a violation of the property. If none is found, then the property is satisfied, otherwise a counter-example, also known as *trace*, is returned, i.e., a path from the initial state to the state violating the property.

In order to test some formulae we use the `Uppaal` model checker. It employs a subset of CTL defined as follow:

$$\begin{array}{rcll} \phi & = & AG\psi \mid AF\psi \mid EG\psi \mid EF\psi & \text{quantifiers} \\ & & \mid \psi \rightsquigarrow \psi & \text{leads-to} \\ & & & \\ \psi & = & \mathbf{true} \mid \mathbf{false} \mid \mathbf{deadlock} \mid P & \text{atoms} \\ & & \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \psi \implies \psi & \text{connectives} \end{array}$$

where P, as usual, ranges over atomic propositions and $\mathbf{deadlock}$ is an atomic proposition which holds only in states having no outgoing transition.

Chapter 3

Spiking neural networks formalization

In this chapter we provide two possible encodings for a leaky-integrate-and-fire neuron into timed automata, namely, the *asynchronous* and *synchronous* encodings. Then, we define how valid input sequences should be structured, by means of a regular grammar. An encoding for regular input sequences matching such a grammar is provided, too. Consequently, we discuss how an entire neural network, comprehensive of input generators and output consumers, can be encoded into a timed automata network. In particular, we focus on the problem of realizing a specific network topology by means of channels sharing. Finally, we show how a timed automata network encoding a spiking neural network can be implemented — and therefore simulated and inspected — within the `Uppaal` framework.

3.1 Leaky-integrate-and-fire neurons as timed automata

In this section we present two possible encodings of leaky-integrate-and-fire neurons into timed automata. The first one, namely the *asynchronous* encoding, produces a reactive machine where the potential is updated as soon as an input spike is received and no two spikes can be received simultaneously.

The second one, namely the *synchronous* encoding, improves its predecessor allowing the spikes received within a given *accumulation period* to be considered simultaneous.

When defining an encoding as timed automata for LI&F, some further constraints need be added to definitions 2.3 and 2.4. Indeeds, the timed automata can only handle integer variables and do not allow to use real numbers. Thus, in what follows, we discretize the $[\,0,\,1\,]$ range splitting it into R parts, where R is a positive integer referred as *discretization granularity*. Synaptic weights are therefore integers in $\{-R,\,\ldots,\,R\}$, while potentials are integers whose value must be interpreted with respect to R. The potential update rule, shown in equation 2.3 must include the floor operator in oder to guarantee the updated potential to be an integer:

$$p(t) = \varepsilon(t) + \lfloor \lambda \cdot p(t-1) \rfloor \tag{3.1}$$

Differently from weights and potentials, leak factors are constrained to be rational numbers within the range $[\,0,\,1\,]$, so they are conveniently representable by a couple of integer numbers.

3.1.1 Asynchronous encoding as timed automata

Here we present the *asynchronous* encoding of spiking neurons into timed automata. Such an encoding does not explicitly exploit the concept of *time-quantum*. It assumes the time to be continuous and the input spikes to be strictly ordered: they can be received at any instant, but no more than one spike can be received at a time. It also assumes input spikes to be received at an almost-constant rate.

A spiking neuron can be encoded into an automaton that: (i) updates the potential whenever it receives an input spike, taking into account the previous potential value, properly decayed, (ii) if the accumulated potential overcomes the threshold, the neuron emits an output spike and resets its potential, (iii) it ignores any input spike for the whole duration of the refractory period.

Definition 3.1 (Asynchronous encoding)**.** Let $N = (\theta,\,\lambda,\,p,\,\tau,\,y)$ be a *leaky-integrate-and-fire neuron*, let m be the number of ingoing synapses of N, and

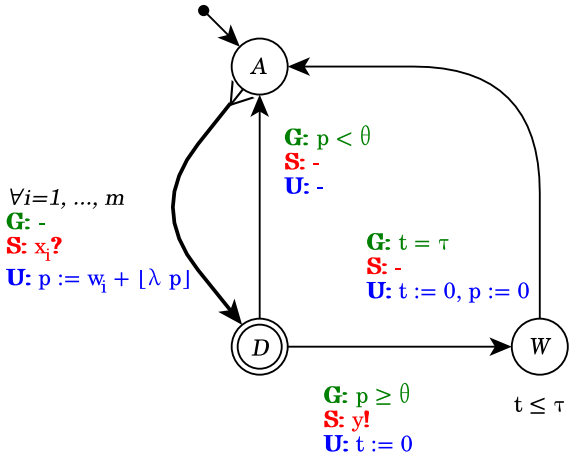


Figure 3.1: Asynchronous encoding of a leaky-integrate-and-fire neuron into a timed automaton. The initial state is **A**ccumulate, **D**ecide is a *committed* state while **W**ait is a normal state subject to the $t \leqslant \tau$. The $(\mathbf{A} \rightarrow \mathbf{D})$ edge is actually a parametric and synthetic way to represent m edges, one for each input synapse.

let $w_1, \ldots, w_m$ be the weights of such synapses, then its *asynchronous* encoding $[\![N]\!]_{asyn}$ into timed automata is a tuple $(L, \mathbf{A}, X, Inv, \Sigma, Arcs)$, where:

- $L = \{\mathbf{A}, \mathbf{D}, \mathbf{W}\}$ with $\mathbf{D}$ committed,
- $X = \{t\}$,
- $Inv = \{\mathbf{W} \mapsto (t \leq \tau)\}$
- $\Sigma = \{y\} \cup \{x_i \mid i = 1, \ldots, m\}$,
- $Arcs = \{(\mathbf{A}, \mathbf{true}, x_i?, \{p := w_i + \lfloor \lambda \cdot p \rfloor\}, \mathbf{D}) \mid \forall i = 1, \ldots, m\} \cup \{(\mathbf{D}, p < \theta, \varepsilon, \{\}, \mathbf{A}), (\mathbf{D}, p \geq \theta, y!, \{t := 0\}, \mathbf{W}), (\mathbf{W}, t = \tau, \varepsilon, \{t := 0, p := 0\}, \mathbf{A})\}$

where p and all w_i are integer variables.

Such an encoding is represented in figure 3.1 and an intuition of its behavior is described in the following. It depends on the following channels, variables and clocks:

- $x_1, \ldots, x_m$ are the broadcast channels used to receive input spikes,

- y is the output broadcast channel used to emit the output spike,
- $p \in \mathbb{N}$ is an *integer variable* holding the current potential value, which is initially 0,
- $t \in \mathbb{N}$ is a *clock*, initially set to 0.

The automaton has three locations: $\mathbf{A}$, $\mathbf{D}$ and $\mathbf{W}$, which respectively stand for $\mathbf{A}$ccumulate, $\mathbf{D}$ecide and $\mathbf{W}$ait. It can move from one location to another according following rules:

- It keeps waiting in location $\mathbf{A}$ for input spikes and whenever it receives a spike on input x_i (i.e., it *receives* on channel x_i) it moves to location $\mathbf{D}$ updating p as follows:
$$p := w_i + \lfloor \lambda \cdot p \rfloor$$
- While the neuron is in location $\mathbf{D}$ then time does not progress (since it is *committed*); from this location, the neuron moves back to $\mathbf{A}$ if $p < \theta$, or it moves to $\mathbf{W}$, firing an output spike (i.e., *writing* on y) and resetting t, otherwise.
- The neuron will remain in location $\mathbf{W}$ for an amount of time equal to τ and then it will move back to location $\mathbf{A}$ resetting both p and t.

According to definition 3.1, the neuron is a purely reactive machine: each received spike makes its potential decay by a factor λ, regardless of the time elapsed since the previous spike. If no input spike occurs, time flow has no effect on the neuron. This is undesirable because definition 2.4 clearly states that the potential should *exponentially* decay as time progresses. In order to work around such a limitation, we add the following assumption:

> Consecutive input spikes will occur with an *almost constant* frequency regardless of which synapse they come from, i.e., the time difference between one spike and its successor is considered to be *the same*

Under such an hypothesis, it is legitimate to consider the *leak factor* as a constant instead of a decreasing function of time, as for definition 3.1.

Remark. The assumptions this model relies on are maybe too strong: it does not properly handle scenarios having input spikes occurrence times with non-negligible variance and it is expected to behave poorly in such cases.

Finally, it may be noticed that a minimal automaton can be obtained collapsing locations $\mathbf{A}$ and $\mathbf{D}$. The reasons they have been kept separated are: (a) within some *model-checking* query, the presence of location $\mathbf{D}$ allows to express concepts like *"the neuron has just received a spike"* or *"the neuron is going to emit"*; (b) the presence of location $\mathbf{D}$ allows to reduce the number of required edges: without $\mathbf{D}$ we would have needed m loops on location $\mathbf{A}$ and m edges from $\mathbf{A}$ to $\mathbf{W}$, so $2m + 1$ total edges, considering the one from $\mathbf{W}$ to $\mathbf{A}$; while thanks to $\mathbf{D}$ we only need $m + 3$ edges.

3.1.2 Synchronous encoding as timed automata

We present here a second approach aimed at overcoming the limitations of the asynchronous encoding introduced above. It handles input spike co-occurrence, and time-dependent potential decay, even if no spike is received. The neuron is conceived as a synchronous and stateful machine that: (i) accumulates potential whenever it receives input spikes within a given *accumulation period*, (ii) if the accumulated potential is greater than the *threshold*, the neuron emits an output spike, (iii) it waits for *refractory period*, (iv) and *resets* to initial state. We assume that no two input spikes on the same synapse can be received within the same accumulation period (i.e., the accumulation period is shorter than the minimum refractory period of the input neurons).

Definition 3.2 (Synchronous encoding)**.** Let $N = (\theta,\, \lambda,\, p,\, \tau,\, y)$ be a *leaky-integrate-and-fire neuron*, let m be the number of ingoing synapses of N, let $w_1,\, \ldots,\, w_m$ be the weights of such synapses, and let $T \in \mathbb{N}^+$ be the duration of the *accumulation period*, then the *synchronous* encoding of N, $[\![N]\!]_{syn}$, into timed automata is a tuple $(L,\, \mathbf{A},\, X,\, Inv,\, \Sigma,\, Arcs)$, where:

- $L = \{\mathbf{A},\, \mathbf{D},\, \mathbf{W}\}$ with $\mathbf{D}$ committed,
- $X = \{t\}$,
- $Inv = \{\mathbf{A} \mapsto (t \leq T),\ \mathbf{W} \mapsto (t \leq \tau)\}$,

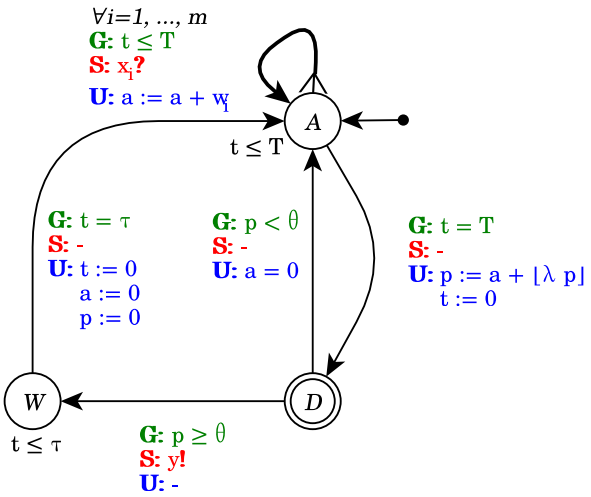


Figure 3.2: Synchronous encoding of a leaky-integrate-and-fire neuron into a timed automaton. The $(\mathbf{A} \rightarrow \mathbf{A})$ loop is actually a parametric and synthetic way to represent m edges, one for each input synapse.

- $\Sigma = \{y\} \cup \{x_i \mid i = 1, \ldots, m\}$,
- $Arcs = \{(\mathbf{A},\, t \leq T,\, x_i?,\, \{a := a + w_i\},\, \mathbf{A}) \mid \forall i = 1, \ldots, m\} \cup \{(\mathbf{A},\, t = T,\, \varepsilon,\, \{p := a + \lfloor \lambda \cdot p \rfloor, t := 0\},\, \mathbf{D}),\, (\mathbf{D},\, p < \theta,\, \varepsilon,\, \{a := 0\},\, \mathbf{A}),\, (\mathbf{D},\, p \geq \theta,\, y!,\, \{\},\, \mathbf{W}),\, (\mathbf{W},\, t = \tau,\, \varepsilon,\, \{t := 0,\, p := 0, a := 0\},\, \mathbf{A})\}$

where p, a and all w_i are integer variables.

Such an encoding is represented in figure 3.2 and an intuition of its behavior is described in the following. It depends on the following channels, variables and clocks:

- t, x_i, y and p are, respectively, a clock, the i-th input broadcast channel, the output broadcast channel and the current potential variable, as for the asynchronous encoding,
- $a \in \mathbb{N}$ is a variable holding the weighted sum of input spikes occurred within the *current* accumulation period. It is 0 at the beginning of each period.

Locations are named as for the asynchronous encoding, but here they are subject to different rules:

- The neuron keeps waiting in state $\mathbf{A}$ for input spikes while $t \leqslant T$ and whenever it receives a spike on input x_i it updates a as follows:

$$a := a + w_i$$

- When $t = T$ the neuron moves to state $\mathbf{D}$, resetting t and updating p as follows:

$$p := a + \lfloor \lambda \cdot p \rfloor$$

- Since state $\mathbf{D}$ is *committed*, it does not allow time to progress, so, from this state, the neuron can move back to $\mathbf{A}$ resetting a if $p < \theta$, or it can move to $\mathbf{W}$, firing an output spike, otherwise.

- The neuron will remain in state $\mathbf{W}$ for τ time units and then it will move back to state $\mathbf{A}$ resetting a, p and t.

The innovation here is the concept of accumulation period. According to the asynchronous encoding, two inputs cannot occur into the same instant and, above all, their relative order is the only thing that influences the neuron potential: two consecutive input spikes would have the same effect regardless of their time difference. Thanks to the accumulation period of the synchronous encoding, the time distance between two consecutive spikes can be valorized: since the firing of transition $(\mathbf{A} \rightarrow \mathbf{D})$, namely "the end of the accumulation period", is *not* governed by input spikes as for its asynchronous counterpart but only by time, the neuron potential actually decays as time progress, if no input is received.

Note that, if the assumption requiring one input not to emit more than once within the same accumulation period does not hold (i.e. inputs frequencies are *too high*), the neuron potential would increase as if the two spikes were from different synapses.

Finally, it may be noticed that, as for the asynchronous model, a minimal automaton can be obtained by removing state $\mathbf{D}$ and adding more edges.

3.2 Spiking neural networks as timed automata networks

After showing how a timed automaton can encode a neuron, the main concern is about neuron interconnection, i.e., encoding leaky-integrate-and-fire spiking neural networks into timed automata networks by means of a proper channel sharing convention. Another relevant matter covered by this section is about inputs and outputs handling, for a given network. The input neurons of a network are encoded into *input generators*, i.e., automata emitting spikes over a specific output channel. We say that such generators *feed* the network, by producing input spikes for its neurons. Analogously, the spikes produced by any output neuron of a network are consumed by an *output consumer*. We say that such automata *consume* the outcome of the network, by receiving — and possibly inspecting — the output spikes produced by its output neurons. We also define a language for *input sequences* specification and show how to encode any word from such a language into a timed automaton able to emit it. Then we introduce *non-deterministic input generators* which are useful in those contexts where random input sequences are needed. Finally, we show how *output consumers* can be used to inspect, e.g., a neuron spike frequency.

Let $S = (V, A, W)$ be a leaky-integrate-and-fire spiking neural network with $V = V_i \cup V_{int} \cup V_o$ (as remarked in definition 2.1, we distinguish between input, intermediary and output neurons), then the encoding of S into a timed automata network is given by the parallel composition of the encodings of all the neurons within the network:

$$[\![S]\!] = \Big(\Big|\Big|_{g \in V_i} [\![g]\!]\Big) \parallel \Big(\Big|\Big|_{v \in V_{int}} [\![v]\!]\Big) \parallel \Big(\Big|\Big|_{n \in V_{out}} [\![n]\!] \parallel \mathcal{O}_n\Big)$$

where $\mathcal{O}_n$ are output consumers: the network has $|V_{out}|$ automata of such a sort.

The topology of network S, expressed in A, must be reflected by the way channels are shared between the automata encoding the neurons in S. So, let $\Sigma = \{y_v \mid v \in V\}$ be the set of broadcast channels containing the output channel of each neuron composing the network, let $inputs : V \to \Sigma^*$ be the function mapping each neuron v to the tuple of channels used by $[\![v]\!]$ for

receiving spikes, let $weights : V \to \mathbb{Z}^*$ be the function mapping each neuron v to the tuple of weights used by $[\![v]\!]$, and let $output : V \to \Sigma$ be the function mapping each neuron v to the channel used by $[\![v]\!]$ for sending output spikes. Then we impose the following constraints to $[\![S]\!]$:

- $output(v) = y_v, \quad \forall v \in V$,
- $inputs(v) = \{y_v \mid \forall (u, v) \in A\}, \quad \forall v \in V_{int} \cup V_o$,
- $weights(v) = \{W(u, v) \mid \forall (u, v) \in A)\}, \quad \forall v \in V_{int} \cup V_o$,
- for each $v \in V_o$, the output channel y_v of v is the input channel of an output consumer $\mathcal{O}$.

In the rest of this thesis we may adopt the following notation to represent the interconnection of automata. Let $\mathcal{I}_1, \mathcal{I}_2, \ldots$ be input generators, let $\mathcal{N}, \mathcal{N}_1, \mathcal{N}_2, \ldots$ be some neurons encoding, and let $\mathcal{O}$ be an output consumer, then we may write:

- $(\mathcal{I}_1, \ldots, \mathcal{I}_n) \overset{\mathbf{x}}{\parallel} \mathcal{N}$, where $\mathbf{x} = (x_1, \ldots, x_n)$, meaning that each channel x_i is shared between $\mathcal{I}_i$ and $\mathcal{N}$, carrying input spikes from the former to the latter;
- $(\mathcal{N}_1, \ldots, \mathcal{N}_n) \overset{\mathbf{y}}{\parallel} \mathcal{N}$, where $\mathbf{y} = (y_1, \ldots, y_n)$, meaning that each channel y_i is shared between $\mathcal{N}_i$ and $\mathcal{N}$, carrying the output spikes of $\mathcal{N}_i$ which are received by $\mathcal{N}$;
- $\mathcal{N} \overset{y}{\parallel} \mathcal{O}$, meaning that y is a shared channel carrying the output spikes produce by $\mathcal{N}$ and consumed by $\mathcal{O}$.

3.2.1 Handling networks inputs

Here we discuss about the inputs used to feed spiking neural network. Essentially, neurons consume sequences of spikes. We propose a the grammar of a language over spikes and pauses defining how any valid input sequence may be structured. We also provide an encoding into input generators for such sequences, i.e., timed automata able to reproduce a given word over the proposed language.

Regular input sequences. Essentially, input sequences are lists of spikes and pauses. Spikes are instantaneous while pauses have a non-null duration. Sequences can be *empty*, *finite* of *infinite*. After each spike there must be a pause except when the spike is the last event of a finite sequence, i.e., there exists no sequence having two consecutive spikes. Infinite sequences are composed by two parts: a finite and arbitrary prologue and an infinite and periodic part whose period is composed by a finite sub-sequence of *spike–pause* couples which is repeated infinitely often.

Definition 3.3 (Input sequence grammar)**.** Let s, p, $]$ and $[$ be terminal symbols, let I, N, P_1, ..., P_n and P be non-terminals and let $x_1, \ldots, x_n \in \mathbb{N}^+$ be some *durations* (that are terminals) then:

$$\begin{aligned} IS &::= \Phi\, s \mid P\, \Phi\, s \\ &\mid \Phi\, \Omega^\omega \mid P\, \Phi\, \Omega^\omega \\ \Phi &::= \varepsilon \mid s\, P\, \Phi \\ \Omega &::= (s\, P_1) \cdots (s\, P_n) \\ P &::= p[N] \\ P_i &::= p[x_i] \end{aligned}$$

represents the ω*-regular expression* for valid input sequences.

The language $\mathcal{L}(IS)$ of words generated by such a grammar is the set of *valid* input sequences.

In definition 3.3, the symbol s represents a spike, p is a symbol representing a pause, and each pause is associated to a natural-valued duration, as one can notice by the productions of P and P_i. The notation $p[N]$ represents a pause whose duration is *some* number matching N, the regular expression for natural numbers, while $p[x_i]$ represents a pause whose duration is *a given number* x_i. This is important because pauses within the periodic part Ω, which is repeated infinitely often, must be the same in all repetitions. Notice that any pause within any valid input sequence is followed by a spike.

It is possible to generate a generator automaton for any regular input sequence, according to the following encoding.

Definition 3.4 (Input generator)**.** Let $I \in \mathcal{L}(IS)$ be a valid input sequence, let t be a clock and let y be a broadcast channel, then the encoding of I into a

timed automaton is a tuple $[\![I]\!] = (L(I),\ first(I),\ \{t\},\ \{y\},\ Arcs(I),\ Inv(I))$, inductively defined as follows:

- if $I := \Phi\ (\Omega)^{\omega}$
  - $L(I) = L(\Phi) \cup L(\Omega)$
  - $first(I) = first(\Phi), \quad last(I) = last(\Omega)$
  - $Arcs(I) = Arcs(\Phi) \cup Arcs(\Omega) \cup \{(last(\Phi),\ \mathbf{true},\ \varepsilon,\ \{\},\ first(\Omega))\}$
  - $Inv(I) = Inv(\Phi) \cup Inv(\Phi)$
- if $I := \Phi\ s$
  - $L(I) = L(\Phi) \cup \{\mathbf{S},\ \mathbf{E}\}$, where $\mathbf{S}$ is *urgent*
  - $first(I) = first(\Phi), \quad last(I) = \mathbf{E}$
  - $Arcs(I) = Arcs(\Phi) \cup \{(last(\Phi),\ \mathbf{true},\ \varepsilon,\ \{\},\ \mathbf{S}),\ (\mathbf{S},\ \mathbf{true},\ y!,\ \{\},\ \mathbf{E})\}$
  - $Inv(I) = Inv(\Phi)$
- if $I := p[x]\ I'$, with $I' := \Phi\ (\Omega)^{\omega}$ or $I' := \Phi\ s$ and $x \in \mathbb{N}^{+}$
  - $L(I) = \{\mathbf{P}_0\} \cup L(I')$,
  - $first(I) = \mathbf{P}_0, \quad last(I) = last(I')$
  - $Arcs(I) = Arcs(\Phi_1) \cup \{(\mathbf{P}_0,\ t \leq x,\ \varepsilon,\ \{t := 0\},\ first(I'))\}$
  - $Inv(I) = \{\mathbf{P}_0 \mapsto (t \leq x)\} \cup Inv(I')$
- if $\Phi := \varepsilon$
  - $L(\Phi) = \{\mathbf{E}\}$, where $\mathbf{E}$ is *urgent*
  - $first(\Phi) = last(\Phi) = \mathbf{E}$
  - $Arcs(\Phi) = Inv(\Phi) = \emptyset$
- if $\Phi := s\ p[x]\ \Phi'$ with $x \in \mathbb{N}^{+}$
  - $L(\Phi) = \{\mathbf{S},\ \mathbf{P}\} \cup L(\Phi')$
  - $first(\Phi) = \mathbf{S}, \quad last(\Phi) = last(\Phi')$

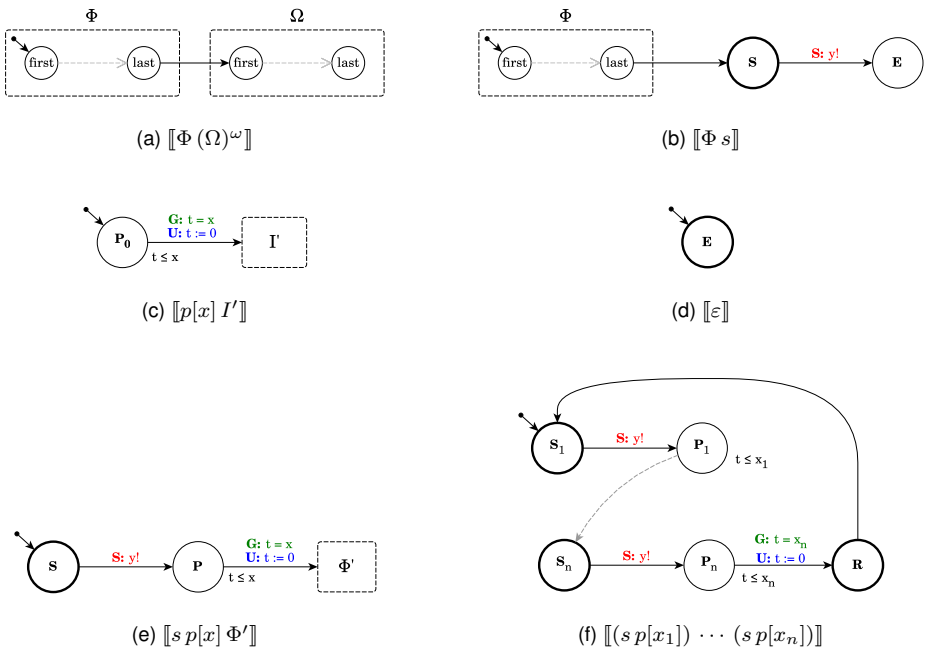


Figure 3.3: Representation of the encoding process for an input sequence

  - $Arcs(\Phi) = \{(\mathbf{S},\ \mathbf{true},\ y!,\ \{\},\ \mathbf{P}),\ (\mathbf{P},\ t = x,\ \varepsilon,\ \{t := 0\},\ first(\Phi'))\} \cup Arcs(\Phi')$
  - $Inv(\Phi) = \{\mathbf{P} \mapsto t \leq x\} \cup Inv(\Phi')$

- if $\Omega := (s\,p[x_1]) \cdots (s\,p[x_n])$ with $x_i \in \mathbb{N}^+$
  - $L(\Omega) = \{\mathbf{S}_1,\ \mathbf{P}_1,\ \ldots,\ \mathbf{S}_n,\ \mathbf{P}_n,\ \mathbf{R}\}$, where $\mathbf{R}$ and all $\mathbf{S}_i$ are *urgent*
  - $first(\Omega) = \mathbf{S}_1, \quad last(\Omega) = \mathbf{R}$
  - $Arcs(\Omega) = \{(\mathbf{S}_i,\ \mathbf{true},\ y!,\ \{\},\ \mathbf{P}_i),\ (\mathbf{P}_i,\ t = x_i,\ \varepsilon,\ \{t := 0\},\ \mathbf{S}_{i+1}) \mid i = 1,\ \ldots,\ (n-1)\} \cup \{(\mathbf{S}_n,\ \mathbf{true},\ y!,\ \{\},\ \mathbf{P}_n),\ (\mathbf{P}_n,\ t = x_n,\ \varepsilon,\ \{t := 0\},\ \mathbf{R})\} \cup \{(\mathbf{R},\ \mathbf{true},\ \varepsilon,\ \{\},\ \mathbf{S_1})\} \cup \{(\mathbf{R},\ \mathbf{true},\ \varepsilon,\ \{\},\ \mathbf{S_1})\}$
  - $Inv(\Omega) = \{\mathbf{P}_i \mapsto (t \leq x_i) \mid \forall i = 1,\ \ldots,\ n\}$

Figure 3.3 depicts the shape of input generators. Fig. 3.3a shows the generator $[\![I]\!]$ obtained from an infinite sequence $I := \Phi\,(\Omega)^\omega$: an unlabeled edge connects the last location of the finite prefix Φ to the first location of the periodic part Ω. In case of a finite sequence $I := \Phi\,s$, as shown in fig. 3.3b, the last location of finite prefix Φ is connected to an urgent location S having an

outgoing edge to the last location $\mathbf{E}$. The edge $(\mathbf{S} \to \mathbf{E})$ is the one responsible for firing the last spike of I. Fig. 3.3c shows the case of an input sequence $I := p[x]\, I'$ beginning with a pause $p[x]$. In this case, the initial location of $[\![I]\!]$ is $\mathbf{P}_0$, which produces a delay of x time units. The remainder I' of the input sequence is encoded as for the previous cases.

Fig. 3.3d shows the induction basis for the recursive encoding a sequence Φ of spike–pause couples, i.e., the case $\Phi := \varepsilon$. It is encoded as an *urgent* location $\mathbf{E}$ having no outgoing edges. Fig. 3.3e shows the case of a non-empty spike–pause couples sequence $\Phi := s\, p[x]\, \Phi'$ which is encoded as an *urgent* location $\mathbf{S}$, connected to a location, $\mathbf{P}$, which is connected to the first location of $[\![\Phi']\!]$. Each edge $(\mathbf{S} \to \mathbf{P})$ provokes a spike firing over channel y, while each location $\mathbf{P}$ introduces a delay of x.

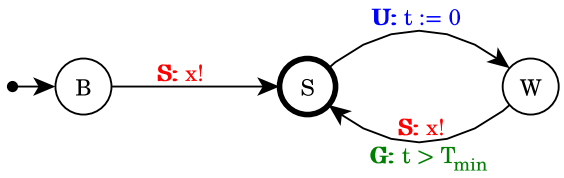
As shown in fig. 3.3f, the periodic part $\Omega := (s\, p[x_1]) \cdots (s\, p[x_n])$ is encoded by a sequence of locations $(\mathbf{S}_1, \mathbf{P}_1, \ldots, \mathbf{S}_n, \mathbf{P}_n, \mathbf{R})$, where all $\mathbf{S}_i$ and $\mathbf{R}$ are urgent. Such locations are connected as described for the previous case. An unlabeled edge $(\mathbf{R} \to \mathbf{S}_1)$ allows the periodic sequence to be generated infinitely often.

So, the generator automaton $[\![I]\!]$ will be composed by one location $\mathbf{S}$ (resp. $\mathbf{P}$) for each s (resp. $p[x]$) symbol in I. A spike is fired over channel y whenever an outgoing transition from some location $\mathbf{S}$ is fired. Similarly, each location $\mathbf{P}$ provokes a delay of x time units, where x is the duration of the pause $p[x]$ encoded by such a location.

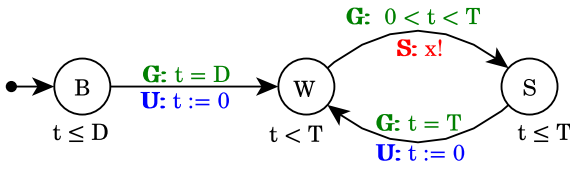
Non-deterministic input sequences. Non-deterministic input sequences are valid input sequences where the occurrence times of spikes is random but the minimum inter-spike quiescence duration is T_{min}. Such sequences can be generated by the non-deterministic generator defined in the following. This sort of automaton may be useful if no assumptions are available or desirable about the spike sequence feeding some neuron.

Definition 3.5 (Non-deterministic input generator)**.** A non-deterministic input generator I_{nd} is a tuple $(L, \mathbf{B}, X, \Sigma, Arcs, Inv)$, with:

- $L = \{\mathbf{B}, \mathbf{S}, \mathbf{W}\}$, with $\mathbf{S}$ urgent,
- $X = \{t\}$



(a) "*Non-deterministic* input generator"



(b) "*Fixed-rate* input generator"

Figure 3.4: Automata generating input sequences. Non-deterministic generators are only constrained to wait more than T_{min} time units between emissions. Fixed-rate generators are only constrained to fire exactly once for each period T.

- $\Sigma = \{x\}$
- $Arcs = \{(\mathbf{B},\, t = D,\, x!,\, \{\},\, \mathbf{S}),\, (\mathbf{S},\, \mathbf{true},\, \varepsilon,\, \{t := 0\},\, \mathbf{W}),\, (\mathbf{W},\, t > T_{min},\, x!,\, \{\},\, \mathbf{S})\}$
- $Inv = \{\mathbf{B} \mapsto (t \leq D)\}$

where $D \in \mathbb{N}$ is the initial delay and $T_{min} \in \mathbb{N}^+$ is the minimum inter-spike quiescence duration.

A representation of the structure of such an automaton is shown in figure 3.4a. Its behavior depends on the t clock and the x broadcast channel, and can be summarized as follow: it waits in location $\mathbf{B}$ an arbitrary amount of time before moving to location $\mathbf{S}$, firing its first spike over channel x. Since location $\mathbf{S}$ is *urgent*, the automaton instantaneously moves to location $\mathbf{W}$, resetting clock t. Finally, from location $\mathbf{W}$, after an arbitrary amount of time $t \in \,]\, T_{min},\, \infty\, [$, it moves to location $\mathbf{S}$, firing a spike. Notice that an initial delay D may be introduced by adding the invariant $t \leq D$ to the location $\mathbf{B}$ and the guard $t = D$ on the edge $(\mathbf{B} \to \mathbf{S})$.

Fixed-rate input sequences. Some contexts may consider input sequences having fixed *rates*, i.e., the expected amount of spikes during some given time window T is constant, even if the sequence is not formally periodic since the distribution of spikes within two different time windows may differ. Such sequences can be generated by the non-deterministic generator defined in the following.

Definition 3.6 (Fixed-rate input generator)**.** A fixed-rate input generator I_{fr} is a tuple $(L, \mathbf{B}, X, \Sigma, Arcs, Inv)$, with:

- $L = \{\mathbf{B}, \mathbf{S}, \mathbf{W}\}$
- $X = \{t\}$
- $\Sigma = \{x\}$
- $Arcs = \{(\mathbf{B},\, t = D,\, \varepsilon,\, \{t := 0\},\, \mathbf{W}),\, (\mathbf{W},\, 0 < t < T,\, x!,\, \{\},\, \mathbf{S}),\, (\mathbf{S},\, t = T,\, \varepsilon,\, \{t := 0\},\, \mathbf{S})\}$
- $Inv = \{\mathbf{B} \mapsto (t \leq D),\, \mathbf{W} \mapsto (t \leq T),\, \mathbf{S} \mapsto (t \leq T)\}$

where $D \in \mathbb{N}$ is the initial delay and $T \in \mathbb{N}^+$ is the time window size.

A representation of the structure of such an automaton is shown in figure 3.4b, and an intuition of its behavior can be summarized as follows: it waits in location $\mathbf{B}$ until clock t value equals to D, then it moves to location $\mathbf{W}$, resetting it; it waits in location $\mathbf{W}$ a non-deterministic amount of time $t_s \in]\,0,\, T\,[$ and then it moves to location $\mathbf{S}$ firing a spike over channel x; finally, it waits $T - t_s$ time units in $\mathbf{S}$ before moving back to $\mathbf{W}$.

3.2.2 Handling networks outputs

In order to have a complete modeling of a spiking neural networks, for each output neuron n we build an *output consumer*, automaton $\mathcal{O}_n$. The automaton, whose formal definition is straightforward, is shown in figure 3.5. The output consumer waits in location $\mathbf{W}$ for the corresponding output spikes on channel y and, as soon as it receives the spike, it moves to location $\mathbf{O}$. This location

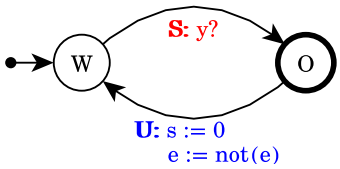


Figure 3.5: *"Output consumer"* automaton. Its initial location is **W**ait, location **O**utput is urgent, $s$ince-last-spike is a clock while $e$ven is a boolean variable.

is only needed to simplify model checking queries: if an output consumer automaton is in location $\mathbf{O}$ then its corresponding neuron has just emitted one spike. Since it is urgent, the consumer instantly moves back to location $\mathbf{W}$ resetting s, the clock measuring the elapsed time since last emission, and setting e to its negation, with e being a *boolean variable* which differentiates each emission from its successor.

3.3 Implementing spiking neural networks in `Uppaal`

In this section we briefly illustrate how our encodings of neurons and input sequences can be implemented in `Uppaal` and composed to instantiate systems made of runnable and inspectable timed automata. The `Uppaal` framework comes with a C-like *modeling* language used to define both automata behaviors and networks configurations. Our reference version of `Uppaal` is `4.1.19`. For a complete understanding of what follows, we warmly suggest to read [4].

`Uppaal` projects are defined by tree elements: a *global declarations* section, an arbitrary number of *templates*, and a *system declarations* section.

Templates are to `Uppaal` what classes are to an object-oriented language: they associate a name to a specific sort of timed automata, sharing the same structure and having a common — but still parametric — behavior. Each template must specify: (i) a *name* identifying the template withing the project, (ii) a list of *formal parameters*, (iii) a *template declarations* section, defining types, variables and functions which are "private" to the template, in the OOP sense, (iv) a *template structure* section, defining locations, edges and their guards, synchronizations and update rules. Definitions within each template declarations section can refer to the formal parameters of that template. Similarly,

```
// Type definition for rational numbers
typedef struct {
    int num;
    int den;
} ratio_t;

// Discretization granularity
const int R = 1000;

// Type definition for weights
typedef int[-R, R] weight_t;

// Synaptic weight arrays definitions (one per neuron)
weight_t w1[1] = { R };        // w1[0] = 1
weight_t w2[1] = { R / 2 };    // w2[0] = 0.5
weight_t w3[1] = { R / 2 };    // w3[0] = 0.5
weight_t w4[2] = { R / 4,      // w4[0] = 0.25
                   R / 3 };    // w4[1] = 0.33

// Output channels definitions (one per neuron)
broadcast chan y0, y1, y2, y3, y4;
```

Listing 3.1: Example of global declarations for an Uppaal project

both template declarations and the formal parameters can be referred within the template structure section.

The global declarations section contains a number of definitions which are shared between all templates.

In the system declarations section the templates are instantiated, providing them the *actual parameters* that may have been defined within the global declarations section. The last line of this section must specify which template instances will compose the *system*, i.e., the timed automata network to be run, simulated and verified.

The global declarations section. Here we define the `ratio_t` and `weight_t` data types and the discretization granularity `R`, as shown in listing 3.1. The `ratio_t` data type is used to represent fractions, i.e., rational numbers, which

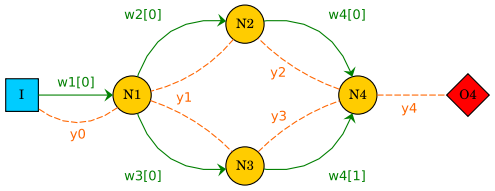


Figure 3.6: An example of spiking neural network composed by 4 neurons (yellow) forming a *diamond* structure and fed by a single input generator (azure). The output of the network is consumed by an output consumer (red). The green edges show the *logical* topology of the network, indeeds the green labels are the names of the *weights* defined in listing 3.1. The orange edges show how such a topology is achieved by means of channel sharing, indeeds the orange labels are the names of the *channels* defined in listing 3.1.

are needed to implement leak factors. As discussed at the beginning of this chapter, we discretize the $[\,0,\,1\,]$ in R parts. This implies synaptic weights, defined as reals in $[\,-1,\,1\,]$, are implemented as integers within the integer range $\{-R,\,\ldots,\,R\}$. This is reflected, in our `Uppaal` projects, by the definition of the `weight_t` data type, which is used to represent synaptic weights.

Synaptic weights arrays and output channels are declared within the global section, too. We adopt the following rule: for each neuron composing the to-be-modeled network, a weights array and a broadcast channel are declared. The size of each array is the number of inputs the corresponding neuron need to accept. Weights arrays are initialized by a `weight_t` array literal, whose values must be expressed with respect to the value of `R`. An output channel per input generator is declared too.

For instance, listing 3.1 shows the global declarations for the network composed by four neurons and an input generator shown in figure 3.6. So four weights arrays and five broadcast channels are declared. The first three neurons accept inputs from one source, while the fourth neuron accepts inputs from two sources. Thus, the length of the fourth array is 2, while for the other ones is 1.

A template for synchronous neurons with `<N>` inputs. Listing 3.2 and figure 3.7 respectively show the declarations and the structure of the `SyncNeuron <N>` template, where `<N>` is a meta-variable indicating the number of inputs accepted by the instances of such a template. Any real `Uppaal` system should

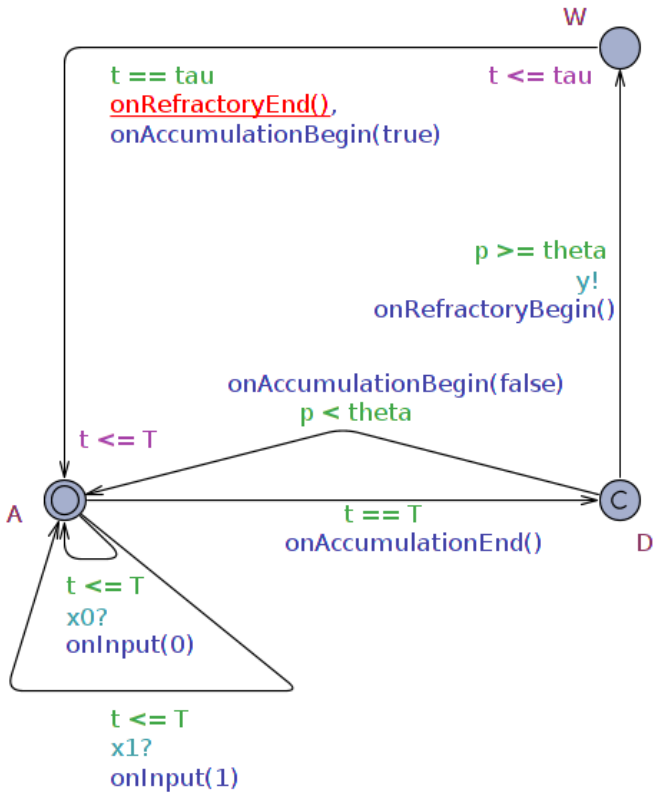


Figure 3.7: The structure of template SyncNeuron<N> for <N> $= 2$

expose as many SyncNeuron1, SyncNeuron2, ... templates as needed. The formal parameters of such a template are shown in listing 3.2: any instance of SyncNeuron<N> may differ for the duration of the accumulation or refractory period, the value of the leak factor and so on. Input and output channels, too, are formal parameters. This is necessary because the channels must be shared in order to reflect the structure of some network.

Other templates. A template AsyncNeuron<N> implementing the asynchronous encoding can be achieved similarly to the SyncNeuron<N> one.

In [1] we present additional material and further examples. E.g., we propose a template NonDeterministicInput(int D, int Tmin, broadcast chan &y) implementing a parametric non-deterministic input generator fitting definition 3.5, a template FixedRateInput(int D, int Twin, broadcast chan &y) implementing fixed-rate input generator fitting definition 3.6, and a template Output

```
//// Formal parameters ////////////////////////////////////

const int T,           // accumulation period
const int tau,         // refractory period
const int theta,       // threshold
ratio_t lambda,        // leak factor
broadcast chan &x1,    // first input channel
...
broadcast chan &x<N>, // last input channel
weight_t &w[<N>],      // vector of synaptic weights
broadcast chan &y      // output channel

//// Template declarations ///////////////////////////////

clock t;  int a = 0;  int p = 0;

void onInput(int i) { a += w[i]; }

void onAccumulationEnd() {
    p = (a * lambda.den + p * lambda.num) / lambda.den;
}

void onAccumulationBegin(bool hasEmitted) {
    t = 0;  a = 0;
}

void onRefractoryEnd() { /* empty */ }

void onRefractoryBegin() { p = 0; t = 0; }
```

Listing 3.2: The formal parameters and declaration of template SyncNeuron<N>, implementing synchronous neurons with <N> inputs, where <N> is a meta-variable indicating the number of inputs accepted by the instances of such a template.

```
// Generators (D = 5, Tmin = 1)
I = NonDeterministicInput(5, 1, y0);

// Neurons (T=1, tau=2, theta=0.5, lambda=0.25)
N1 = SyncNeuron1(1, 2, R / 2, {1, 4}, y0, w1, y1);
N2 = SyncNeuron1(1, 2, R / 2, {1, 4}, y1, w2, y2);
N3 = SyncNeuron1(1, 2, R / 2, {1, 4}, y1, w3, y3);
N4 = SyncNeuron2(1, 2, R / 2, {1, 4}, y2, y3, w4, y4);

// Output consumers
O4 = OutputConsumer(y4);

system I, N1, N2, N3, N4, O4;
```

Listing 3.3: Example of system declarations for an Uppaal project

Consumer(broadcast chan &x) implementing an output consumer. We also provide a *network description language* allowing to formally describe the topology of a spiking neural network, the parameters of each leaky-integrate-and-fire neuron composing it and the input sequences used to feed such a network. Finally, we provide a code generator able to convert a *network description* into a completely working and ready-to-use Uppaal project.

The system declaration section. This is where templates are instantiated and the actual system definition is assembled. The structure of the network is realized by providing the proper channels and weights arrays to the templates when instantiating them.

E.g., listing 3.3 shows how the templates described into the previous paragraph may be instantiated in order to implement the network shown in figure 3.6. The channel y0 is provided to both the generator I (as an output channel) and the neuron N1 (as an input channel): this is because we want to model a network having a synapse from I to N. The other synapses are implemented in an analogous manner.

Chapter 4

Validation of the synchronous model

In this chapter we focus on the *synchronous encoding* of the leaky-integrate-and-fire neuron. Within the scope of this chapter, we may concisely refer to such an encoding as "synchronous model" and to the automata achieved by means of the synchronous encoding as "synchronous neurons", i.e., instances of such a model. The notation $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$, with $\mathbf{w} = (w_1, \ldots, w_n)$, is used to indicate an instance $\mathcal{N}$ of the synchronous model having n input synapses, with the weights $w_1, \ldots, w_n$, accumulation and refractory period durations T and τ, respectively, leak factor λ, and threshold θ.

In the following sections, we analyze the intrinsic properties of the synchronous model, e.g., the *the maximum threshold* value allowing a neuron to emit, or the *lack of inter-spike memory*, preventing the behavior of a neuron from being influenced by what happened before the last spike.

According to Izhikevich [21], there exist a number of *behaviors* (i.e., typical responses to an input pattern) the integrate-and-fire model should be able to reproduce, namely *tonic spiking*, *excitability*, and *integrator*. An overview is shown in figure 2.2. We refer to such behaviors as *capabilities*, since in what follows we prove the synchronous model is able to reproduce them. Izhikevich also identifies a set of behaviors which are not expected to be reproducible by any integrate-and-fire neuron. In the last part of this chapter, we prove these *limits* to hold for the synchronous model, too. Finally, we provide, for each

non-reproducible behavior, an extension of the synchronous model allowing its instances to reproduce such a behavior.

Preliminary definitions. The following concepts are extensively used and exploited within this chapter.

Definition 4.1 (Input (sub-)sequence)**.** Let $\mathcal{I}_1, \ldots, \mathcal{I}_m$ be a *input sources* (i.e., neuron or generators) connected to some neuron $\mathcal{N}$, and let $F_i = \{t_{i,1}, t_{i,2}, \ldots\}$ be the *ordered* set of firing times of $\mathcal{I}_i$; then $I = \bigcup_{i=1}^{m} F_i$ is the *ordered* input sequence of $\mathcal{N}$. For any *continuous* interval $Q \subset \mathbb{R}_0^+$ the set $I \cap Q$ is a sub-sequence of I.

Definition 4.2 (Output (sub-)sequence)**.** Let $\mathcal{N}$ be a neuron and let $F_{\mathcal{N}} = \{t_1, t_2, \ldots\}$ be its *ordered* set of firing times; then $F_{\mathcal{N}}$ is the *ordered* output sequence of $\mathcal{N}$. For any *continuous* interval $Q \subset \mathbb{R}_0^+$ the set $F_{\mathcal{N}} \cap Q$ is a sub-sequence of $F_{\mathcal{N}}$.

Definition 4.3 (Persistent input (sub-)sequence)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, let I be its input (sub-)sequence and let t range over the accumulation periods starting instants; then I is *persistent* if and only if $card(I \cap [\, t, t + T\,[) > 0, \ \forall t$.

Definition 4.4 (Persistent excitatory/inhibitory input (sub-)sequence)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, let I be its input sub-sequence and let n range over the accumulation periods; then I is *excitatory* (resp. *inhibitory*) if and only if $A_n > 0$ (resp. $A_n < 0$) $\forall t$, where A_n is the sum of weighted inputs for the n-th accumulation period.

Definition 4.5 (Persistent constant input (sub-)sequence)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, let I be its input sub-sequence and let n range over the accumulation periods; then I is *constant* if and only if there exists some $K \in \mathbb{Z}$ such that $A_n = K, \ \forall t$.

Definition 4.6 (Periodic output (sub-)sequence)**.** Let $\mathcal{N}$ be a neuron and let $F_{\mathcal{N}} = \{t_1, t_2, \ldots\}$ be its output sequence, then $F_{\mathcal{N}}$ is *periodic* if and only if there exists some $P \in \mathbb{R}^+$ such that $t_{i+1} - t_i = P, \ \forall i$.

Definition 4.7 (Simultaneous input spikes)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, let I be its input sequence, let t range over the beginning instants of all accumulation periods and let $s_1, s_2 \in I$ be two input spikes; then s_1 and s_2 are *simultaneous* if and only if $s_1, s_2 \in [\, t, t + T \,[$ for some t.

Definition 4.8 (Consecutive input spikes)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, let I be its input sequence, let t, t' be the starting instants of some accumulation period and the next one, respectively, and let $s_1, s_2 \in I$ be two input spikes; then s_1 and s_2 are *consecutive* if and only if $s_1 \in [\, t, t + T \,[\wedge$ $s_2 \in [\, t', t' + T \,[$.

Definition 4.9 (Reset times)**.** Let $\mathcal{N}$ be a neuron and let $F_{\mathcal{N}} = \{t_1, t_2, \ldots\}$ be its output sequence, then the set of reset times of $\mathcal{N}$ is $Z_{\mathcal{N}} = \{t + \tau \mid t \in F_{\mathcal{N}}\}$.

Further notational conventions. We use calligraphic letters (e.g., $\mathcal{A}$) for automata, bold letters (e.g., $\mathbf{X}$) for locations, and lower-case italic letters (e.g., t) for variables or clocks. Within temporal logic formulae, the predicate $state_{\mathcal{A}}(\mathbf{X})$ is 1 if and only if automaton $\mathcal{A}$ is in state $\mathbf{X}$, 0 otherwise, and $eval_{\mathcal{A}}(t)$ is a function mapping a variable or clock t to the value it currently carries within the context of automaton $\mathcal{A}$: a predicate may consist of the comparison between such a value and a constant. For boolean variables we may abuse the notation writing $eval(b)$ and $\neg eval_{\mathcal{A}}(b)$ instead of $eval_{\mathcal{A}}(b) = 1$ or $eval_{\mathcal{A}}(b) = 0$, respectively.

4.1 Intrinsic properties of the synchronous model

In this section we analyze the intrinsic properties of the synchronous model, i.e., the properties deduced from its definition, which are not related to a particular behavior. For instance, we prove the existence of a minimum inter-emission period and the lack of an inter-emission memory, making the neuron reset its state as soon as it emits a spike. We also provide a rule to compute the maximum threshold value. It would be impossible to fire for a neuron having a threshold higher than such a value. Finally, we discuss the effect of spikes received over inhibitory synapses, also referred as inhibitory input spikes.

Maximum threshold. Here we show that, assuming an upper bound for the sum of ingoing synapses weights, there exists a way to compute the maximum threshold value such that, any neuron having a threshold greater than or equals to it, will never be able to fire.

Property 4.1 (Threshold-leak factor relation)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron and $a_{max} \in \mathbb{N}^+$ the maximum value of weighted inputs sum, then, if $\theta \geq \frac{a_{max}}{1-\lambda}$, the neuron is not able to fire.

Proof. Without loss of generality, we suppose that, during each accumulation period, $\mathcal{N}$ receives the maximum possible input a_{max}. Then, its potential function is:

$$p_n = a_{max} + \lfloor \lambda \cdot p_{n-1} \rfloor$$

which is always lower than or equal to its undiscretized version:

$$p_n \leq {p'}_n = a_{max} + \lambda \cdot p'_{n-1}$$

The same inequality can be written in explicit form because of equation 2.4:

$$p_n \leq {p'}_n = \sum_{k=0}^{n} a_{n-k} \cdot \lambda^k$$

and, since we assumed the neuron always receives a_{max}, a_{n-k} is constant and do not depend on k:

$$p_n \leq a_{max} \cdot \sum_{k=0}^{n} \lambda^k$$

The rightmost factor is a geometric series having a more compact representation:

$$p_n \leq a_{max} \cdot \frac{1-\lambda^n}{1-\lambda}$$

which reaches its maximum value $\frac{1}{1-\lambda}$ for $n \to \infty$, therefore:

$$p_n \leq \frac{a_{max}}{1-\lambda},\ \forall n \in \mathbb{N}$$

Thus, if $\theta \geq \frac{a_{max}}{1-\lambda}$, it is impossible for the neuron potential to reach the threshold and, consequently, the neuron cannot fire. □

Notice that, according to section 3.1, synaptic weights are never greater than an integer R, so $a_{max} = mR$ for each neuron having m ingoing synapses, even if, in the general case, we will consider $a_{max} = \sum_{i=0}^{m} w_i \leq mR$. We will say that a neuron is *firing enabled* if $\theta < \frac{a_{max}}{1-\lambda}$.

Analysis of neuron timings. We can quantify the amount of time that the neuron requires to complete an accumulate–fire–rest cycle. Such expression is useful to prove some interesting properties, e.g., here we show that there exists a minimum delay between one neuron emission and its successor.

Property 4.2 (Minimum firing period)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a *firing enabled* synchronous neuron, then the time difference between successive firings cannot be lower than $T + \tau$.

Proof. Let $A_n = \sum_{k=1}^{T} a_{k+t_0}$ be the sum of weighted inputs during the n-th *accumulation period*, then the neuron behavior can be described as follows:

$$p_n = A_n + \lfloor \lambda \cdot p_{n-1} \rfloor \tag{4.1}$$

is the potential value after the n-th accumulation period. If the neuron will eventually fire an output spike, then there exists $\hat{n} > 0$ such that:

$$\hat{n} = \underset{n \in \mathbb{N}}{arg\,min} \{p_n \mid p_n \geq \theta\} \tag{4.2}$$

i.e., the firing will occur at the end of the $\hat{n}$-th accumulation period, which means during the $\hat{t}$-th time unit since t_0, thus:

$$\hat{t} = \hat{n} \cdot T + t_0 \tag{4.3}$$

where t_0 is the *last* reset time, i.e., the last instant back in time when the neuron completed its refractory period. Then the *next* reset time t', i.e., the next instant in future when the neuron will complete its refractory period, after having emitted a spike, is:

$$t' = \hat{t} + \tau = \hat{n} \cdot T + \tau + t_0$$

At instant t', the neuron quits its refractory period, n is reset to 0, t_0 is set to t', and $\hat{n}$, $\hat{t}$ and t' must be consequently re-computed as described above.

Such a way to describe our model dynamics make it easy to express the *inter-firing period* as a function of $\hat{n}$:

$$t' - t_0 = \hat{n} \cdot T + \tau \tag{4.4}$$

So, the minimum inter-firing period is $T + \tau$ for $\hat{n} = 1$. Such a property can be verified as follows: let $\mathcal{I}$ be the non-deterministic input generator having $T_{min} = 1$ and, without loss of generality[1], initial delay $D = T + \tau$, then the timed automata network $\mathcal{I} \overset{x}{\parallel} \mathcal{N} \overset{y}{\parallel} \mathcal{O}$ satisfies the following formula:

$$AG(state_{\mathcal{O}}(\mathbf{O}) \implies eval_{\mathcal{O}}(s) \geq T + \tau) \tag{4.5}$$

where s measures the time elapsed since last firing. The formula is true because, whenever the output consumer receives a spike, the time elapsed since the previous received spike cannot be lower than $T + \tau$. □

Analysis of neuron memory. Here we discuss about the neuron capability of taking past events into account when computing its outcome. As argued above, the neuron potential is affected by every input spike it received *since the last reset time*, but every event that occurred before that instant is forgotten.

Definition 4.10 (Neuron inter-emission memory)**.** Let $\mathcal{N}$ be a neuron, let $Z_{\mathcal{N}}$ be its reset times set and let I be an input sub-sequence; then $\mathcal{N}$ has inter-emission *memory* if and only if there exist two different $t, t' \in Z_{\mathcal{N}}$ such that the output sub-sequence produced by $\mathcal{N}$ as a response to I starting from t differs from the output sub-sequence it produces as a response to I starting from t'.

Property 4.3 (Memoryless neuron)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ has not inter-emission memory.

Proof. According to definition 3.2 each reset time occurs on each $(\mathbf{W} \rightarrow \mathbf{A})$ firing. Such event makes the automaton $\mathcal{N}$ move back to its initial location while resetting clock t and variables p and a, making them equal to their starting values. So it is impossible for the neuron to behave differently if subjected to the same input sub-sequence. □

Analysis of inhibitory inputs. Here we argue about the effects of an *inhibitory* stimulation to a neuron whose potential lower than its threshold.

[1] the initial delay is required in order to make the formula hold for the first output spike too

Property 4.4 (Inhibitory effect of negative stimulations)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, let A_n be the sum of weighted inputs received during the current accumulation period and let p_{n-1} be the neuron potential at the end of the previous accumulation period, then if $p_{n-1} < \theta$ and $A_n < 0$ the neuron cannot fire at the end of the current accumulation period.

Proof. It is sufficient to prove that, under such hypotheses, $p_n < \theta$. Considering p_n definition, we can state that:

$$p_n \leq A_n + \lambda \cdot p_{n-1}$$

so, since A_n is negative, we can rewrite it as $-|A_n|$:

$$p_n + |A_n| \leq \lambda \cdot p_{n-1}$$

and then we deduce:

$$p_n < \lambda \cdot p_{n-1}$$

because $p_n < p_n + |A_n|$ and, consequently:

$$p_n \leq \frac{1}{\lambda} \cdot p_n < p_{n-1}$$

because $\lambda^{-1} \in [\,1, \infty\,[$. So finally:

$$p_n < p_{n-1} < \theta$$

□

Next we show that only positive stimulations are necessary for the neuron to produce emissions:

Property 4.5. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron such that $\theta > 0$, let A_n be the sum of weighted inputs received during the current accumulation period and let p_n be the neuron potential at the end of the current accumulation period, then $p_n \geq \theta \implies A_n > 0$.

Proof. It is sufficient to prove that, under such hypotheses, $A_n > 0$. We know that:

$$p_n = A_n + \lfloor \lambda \cdot p_{n-1} \rfloor \geq \theta$$

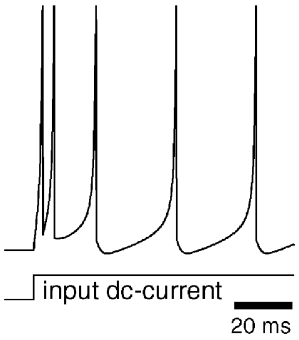


Figure 4.1: Tonic spiking representation for continuous signals from [21].

Let's analyze two sub-cases, with respect to the sign of $\lfloor \lambda \cdot p_{n-1} \rfloor - \theta$:

Consider the case: $\lfloor \lambda \cdot p_{n-1} \rfloor < \theta$. According to the initial hypothesis:

$$\lfloor \lambda \cdot p_{n-1} \rfloor < \theta \leq A_n + \lfloor \lambda \cdot p_{n-1} \rfloor$$

and, consequently:

$$0 < \theta - \lfloor \lambda \cdot p_{n-1} \rfloor \leq A_n$$

so, finally $A_n > 0$.

Consider the case: $\lfloor \lambda \cdot p_{n-1} \rfloor \geq \theta$. This means $p_{n-1} \geq \theta$, in fact:

$$p_{n-1} \geq \lambda \cdot p_{n-1} \geq \lfloor \lambda \cdot p_{n-1} \rfloor \geq \theta$$

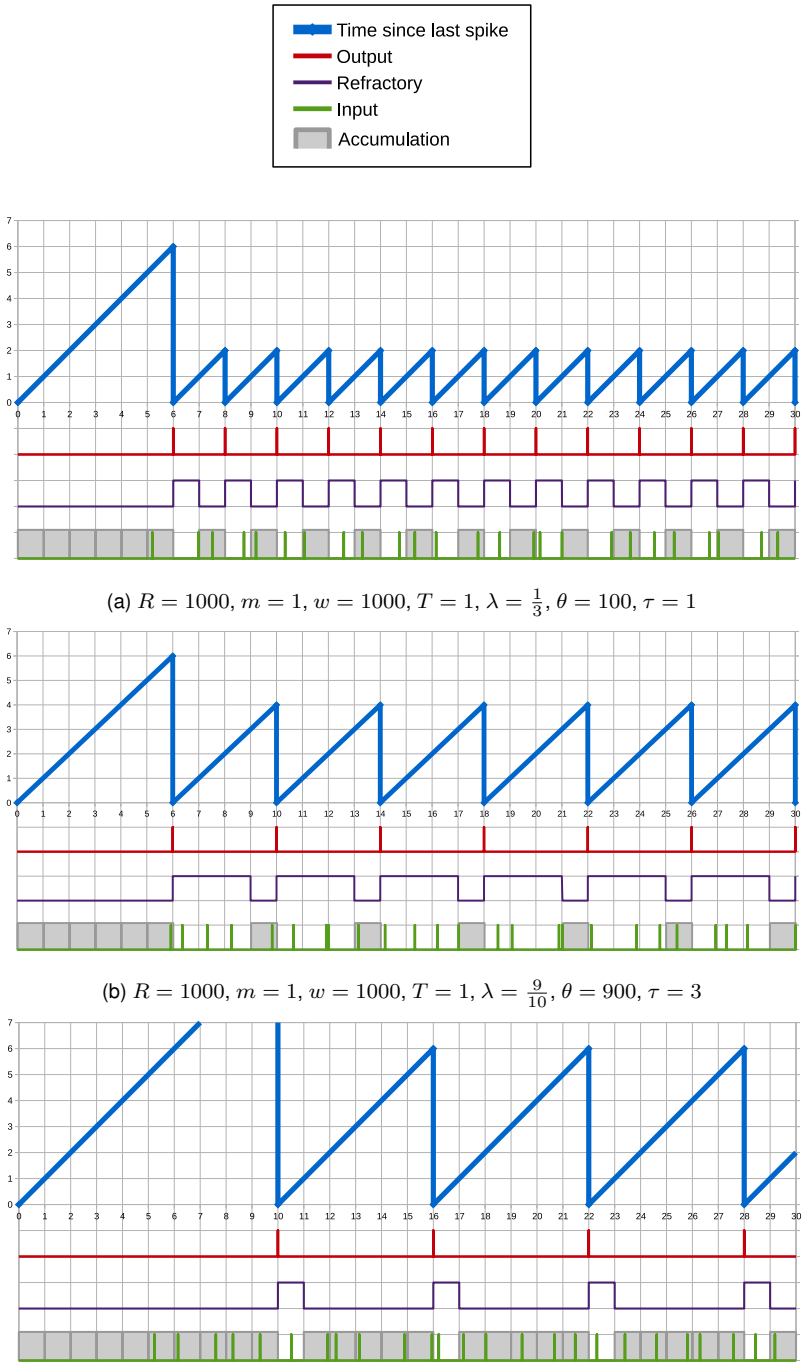
But this is absurd because if $p_{n-1} \geq \theta$, the neuron emits and resets, thus its potential in the next accumulation period is zero, which is in contradiction with the hypothesis $p_n \geq \theta > 0$. □

4.2 Capabilities of the synchronous model

In this section we prove the synchronous model is able to reproduce the behaviors that are expected to be reproducible by integrate-and-fire neurons according to [21], namely, the *tonic spiking*, *integrator*, *excitability* behaviors.

Tonic spiking. "Tonic spiking" is the behavior of a neuron producing a periodic output sub-sequence as a response to a persistent excitatory constant input sub-sequence. An example is shown in figure 4.1.

Property 4.6 (Tonic spiking)**.** Let $\mathcal{N} = (w, T, \lambda, \theta, \tau)$ be a synchronous neuron having only one ingoing excitatory synapse such that $w > 0$ and $\theta <$



(a) $R = 1000$, $m = 1$, $w = 1000$, $T = 1$, $\lambda = \frac{1}{3}$, $\theta = 100$, $\tau = 1$

(b) $R = 1000$, $m = 1$, $w = 1000$, $T = 1$, $\lambda = \frac{9}{10}$, $\theta = 900$, $\tau = 3$

(c) $R = 1000$, $m = 1$, $w = 1000$, $T = 1$, $\lambda = \frac{1}{2}$, $\theta = 1900$, $\tau = 1$

Figure 4.2: Tonic spiking simulations. Each diagram shows the time elapsed since last neuron emission (blue), the emitted spikes (red), the refractory periods (purple) and input spikes (green) within accumulation periods (gray) for three different neurons and for 30 time units. In each case, the input generator is a fixed-rate generator having initial delay 5 and time window size 1.

$w/(1-\lambda)$ and let $\mathcal{I}$ be the input source connected to $\mathcal{N}$ producing a persistent input sequence, then $\mathcal{N}$ produces a periodic output sequence.

Proof (sketch). Let $\mathcal{I}$ be the fixed-rate input generator having arbitrary initial delay D and time window size T, and let $\mathcal{O}$ be an output consumer, then the timed automata network $\mathcal{I} \overset{x}{\parallel} \mathcal{N} \overset{y}{\parallel} \mathcal{O}$ satisfies the following formulae:

$$\begin{cases} state_{\mathcal{O}}(\mathbf{O}) \wedge eval_{\mathcal{O}}(e) \rightsquigarrow state_{\mathcal{O}}(\mathbf{O}) \wedge \neg eval_{\mathcal{O}}(e) \\ state_{\mathcal{O}}(\mathbf{O}) \wedge \neg eval_{\mathcal{O}}(e) \rightsquigarrow state_{\mathcal{O}}(\mathbf{O}) \wedge eval_{\mathcal{O}}(e) \end{cases} \tag{4.6}$$

where $\mathbf{O}$ is the location that automaton $\mathcal{O}$ reaches after consuming a spike and e is boolean variable whose value changes whenever $\mathcal{O}$ moves into location $\mathbf{O}$. So, whenever automaton $\mathcal{O}$ reaches location $\mathbf{O}$ it will eventually reach it again. □

As shown in figure 4.2, if we simulate neurons having different parameters providing them the same input $\mathcal{I}$, then they keep producing a periodic outcome whose period only depends on T and τ as long as $\theta < \frac{w}{1-\lambda}$.

It should be noted that one may also find the value P of the period of some given neuron $\mathcal{N}$ by means of simulations, thus the periodic behavior can be proven by a model-checker verifying the following formula:

$$AG(state_{\mathcal{O}}(\mathbf{O}) \wedge eval_{\mathcal{N}}(f) \implies eval_{\mathcal{O}}(s) = P) \tag{4.7}$$

where s is the clock measuring the time elapsed since last spike consumed by $\mathcal{O}$, and f is a boolean variable of automaton $\mathcal{N}$ which is initially $\mathbf{false}$ and is set to $\mathbf{true}$ when edge $(\mathbf{W} \rightarrow \mathbf{A})$ fires (i.e., it indicates whether $\mathcal{N}$ has already emitted the first spike and waited the first refractory period or not).

Integrator. "Integrator" is the behavior of a neuron producing an output spike whenever it receives *at least* a specific number of simultaneous spikes from different input sources or when it receives a certain amount of *consecutive* spikes from a specific input source. So the neuron parameters can be tuned in order to *detect* (i.e., fire as a consequence of) a given number of simultaneous or consecutive spikes. An example is shown in figure 4.3.

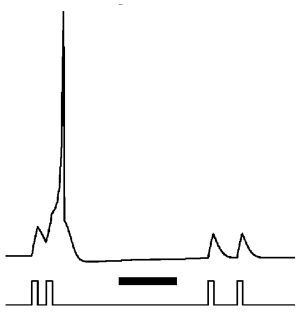


Figure 4.3: Integrator behavior representation for continuous signals, from [21].

Property 4.7 (Simultaneous integrator)**.** Let $\mathcal{N} = ((R, \ldots, R),\, T,\, \lambda,\, n,\, \tau)$ be a synchronous neuron having m synapses with maximum excitatory weight R and an integer threshold $n \leq m$, then the neuron emits if it receives a spike from at least n input sources during the same accumulation period.

Proof (sketch). Let $\mathcal{I}_1,\, \ldots,\, \mathcal{I}_m$ be non-deterministic input generators constrained to wait more than T time units between an emission and its successor, and let $\mathcal{O}$ be an output consumer, then the timed automata network $(\mathcal{I}_1,\, \ldots,\, \mathcal{I}_m) \overset{\mathbf{x}}{\parallel} \mathcal{N} \overset{y}{\parallel} \mathcal{O}$ satisfies the following formula stating that, if at least n generators are in location $\mathbf{S}$ while $\mathcal{N}$ is in $\mathbf{A}$, then $\mathcal{O}$ will eventually capture an output of $\mathcal{N}$:

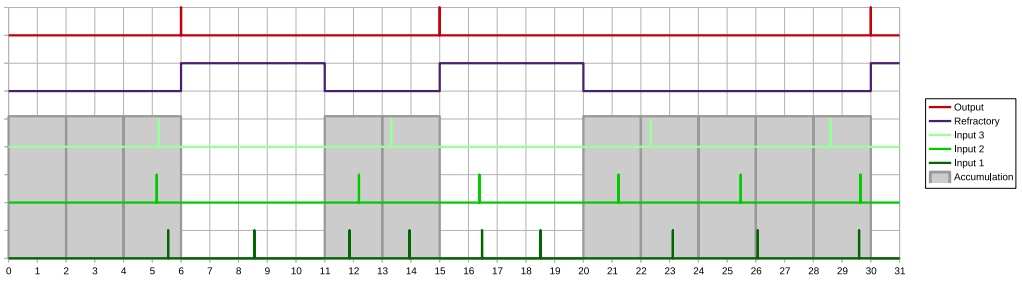
$$\left(\sum_{i=1}^{m} state_i(\mathbf{S}) \geq n\right) \wedge state_{\mathcal{N}}(\mathbf{A}) \rightsquigarrow state_{\mathcal{O}}(\mathbf{O}) \tag{4.8}$$

where $\mathbf{S}$ is the location that each automaton $\mathcal{I}_i$ reaches after producing a spike and $\mathbf{A}$ is the accumulation location of the neuron $\mathcal{N}$. □

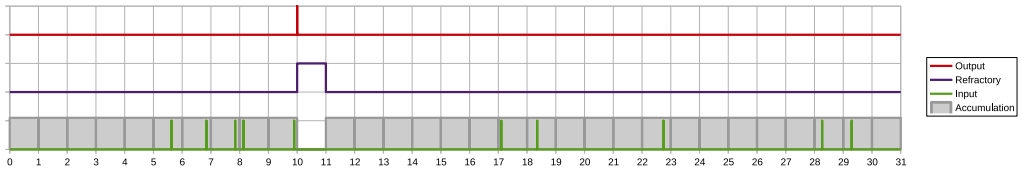
As shown in figure 4.4a, a neuron, under such hypotheses, will fire as soon as it receive n simultaneous spikes.

Notice that, since potential depends on past inputs too, the neuron may still be able to fire in other circumstances, e.g., if it keeps receiving less than n spikes for a sufficient number of accumulation periods, then it may eventually fire.

Property 4.8 (Sequential integrator)**.** Let $\mathcal{N} = (w,\, T,\, \lambda,\, \theta,\, \tau)$, be a synchronous neuron having only one ingoing synapse, such that $\theta < \frac{w}{1-\lambda}$, then



(a) $R = 1000$, $m = 3$, $w_1, w_2, w_3 = 1000$, $T = 2$, $\lambda = \frac{1}{2}$, $\theta = 3000$, $\tau = 5$



(b) $R = 1000$, $m = 1$, $w = 1000$, $T = 1$, $\lambda = \frac{1}{2}$, $\theta = 1900$, $\tau = 1$

Figure 4.4: Chart 4.4a represents the behavior of a neuron, having 3 ingoing synapses, which is able to detect the simultaneity of at least 3 inputs: whenever two or more input spikes (green lines) occur during the same accumulation period (gray), an output spike is produced (red). Chart 4.4b represents the behavior of a neuron, having a single ingoing synapse, which is able to detect a sequence of 5 consecutive input spikes.

there exists a maximal sequence of consecutive input spikes of length $\hat{n}$ that results in an output spike.

Proof (sketch). Let $\mathcal{I}$ be the fixed-rate input generator having arbitrary initial delay D and time window size T, let $\mathcal{O}$ be an output consumer, and let $\hat{n}$ be the minimum amount of consecutive input spikes required to make the potential overcome the threshold, obtained by means of simulation or by recursively computing p_n until it reaches the threshold value; then the Timed Automata Network $\mathcal{I} \overset{x}{\parallel} \mathcal{N} \overset{y}{\parallel} \mathcal{O}$ satisfies the following formula stating that, whenever $\mathcal{O}$ receives a spike, the number of consecutive spikes never greater than $\hat{n}$:

$$AG(state_{\mathcal{O}}(\mathbf{O}) \implies eval_{\mathcal{N}}(c) \leq \hat{n}) \tag{4.9}$$

where c is an integer variable of automaton $\mathcal{N}$ counting the amount of consecutive accumulation periods that received at least one spike since last emission. □

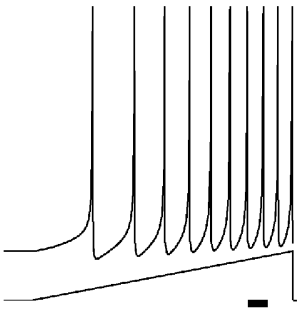


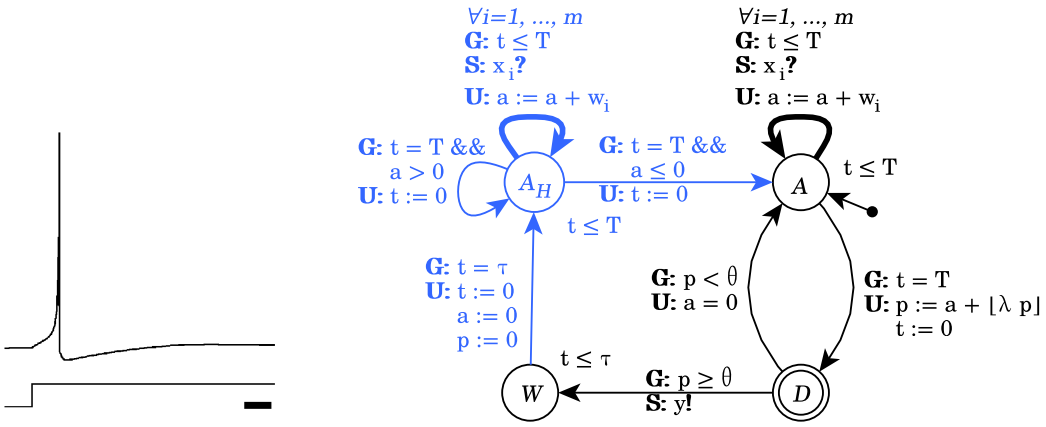
Figure 4.5: Excitability capability representation for continuous signals, from [21].

Figure 4.4b shows a simulation of a neuron able to detect 5 consecutive spikes.

Excitability. "Excitability" is the behavior of a neuron emitting sequences having a *decreasing* inter-firing period, i.e., and increasing output frequency, when stimulated by an *increasing* number of excitatory inputs. An example is shown in figure 4.5

Property 4.9 (Excitability)**.** Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a *firing enabled* synchronous neuron having m excitatory synapses, then the inter-spike period decreases as the sum of weighted input spikes increases.

Proof. If we assume the neuron is receiving an increasing number of excitatory spikes, generated by, e.g., an increasing number of input sources emitting persistent inputs, then a_t is the non-negative, non-decreasing (i.e., $a_{t+1} \geq a_t, \ \forall t$) and progressing (i.e., $\forall u \ \exists t \ : \ a_t > u$) succession representing the weighted sum of inputs within the t-th time unit. Consequently, $A_n = \sum_{k=1}^{T} a_{k+t_0}$ is the non-negative, non-decreasing and progressing succession counting the total sum of inputs within the n-th accumulation period. Since A_n is positive, according to equation 4.1 and equation 4.2, the following statement holds: if A_n increases then $\hat{n}$ decreases. Being $\hat{n}$ the only variable in equation 4.4, if $\hat{n}$ decreases then the difference $t' - t_0$ decreases too. □



(a) Phasic spiking representation for continuous signals from [21].

(b) The synchronous model edited to be able to reproduce the phasic spiking behavior. Variations with respect to the original model shown in figure 3.2 are blue-colored.

Figure 4.6: Phasic spiking: example and model variant.

4.3 Limits of the synchronous model

In this section we prove the synchronous model *is not* able to reproduce the behaviors that are expected to be *non-reproducible* by integrate-and-fire neurons according to [21]. For instance, we prove some behaviors require the neuron to have inter-emission memory or to be capable of burst production. Behaviors of this sort, e.g., phasic spiking or bursting, are clearly non-reproducible because we already proved the synchronous model to miss such requirements. We also provide a number of possible extensions to the synchronous model, each one addressing a particular behavior, making it reproducible.

Phasic spiking. "Phasic spiking" is the behavior of a neuron producing a *single* output spike on the onset of a persistent and excitatory input sub-sequence and then remaining quiescent until the end of such sub-sequence. An example is shown in figure 4.6a.

Such a behavior requires the neuron to be able to detect the onset of an excitatory input sub-sequence and, therefore, it depends on the neuron to have inter-emission memory.

Property 4.10. Let $\mathcal{N}$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *phasic spiking* behavior.

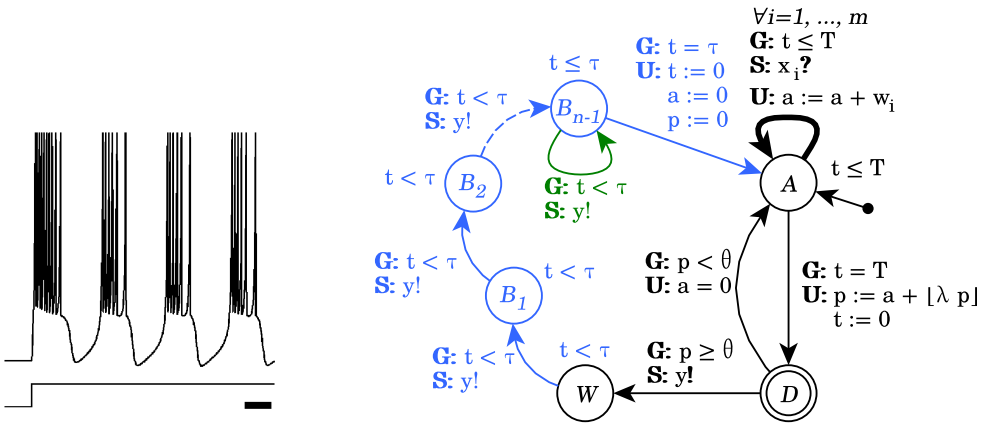
Proof. It is sufficient to prove that such a behavior requires the neuron to have inter-emission memory. In fact, the phasic spiking behavior requires the neuron to ignore any excitatory input spike occurring after its first emission. This means producing different outcomes, before and after the first emission, as a response to the same input sub-sequence, which is impossible for a memoryless neuron, as stated in property 4.3. □

The synchronous model can be edited as shown in figure 4.6b in order to make it able to reproduce such a behavior. This variant simply makes the neuron able to "remember" if it is receiving a persistent excitatory input subsequence. After each refractory period, the neuron moves to location $\mathbf{A_H}$, instead of moving back to $\mathbf{A}$. Here, accumulation periods keep repeating every T time units and weighted inputs are accumulated in variable a, as for location $\mathbf{A}$. The difference between $\mathbf{A_H}$ and $\mathbf{A}$ is that the former one *ignores* positive values of a at the end of each accumulation period. Conversely, a non-positive value of a, at the end of some accumulation period, leads the neuron back in location $\mathbf{A}$. So, such a variant of the synchronous model will fire only one spike on the onset of each persistent excitatory input sub-sequence.

Tonic bursting. A *burst* is a finite sequence of *high frequency* spikes. Some behaviors presented in [21], e.g., tonic of phasing bursting, require the neuron to be able to generate output bursts instead of single spikes. Here we formalize the "burst" concept and discuss about the tonic bursting behavior.

Definition 4.11 (Burst)**.** A spike sub-sequence is a *burst* if it is composed by a least a given number of spikes having an occurrence rate greater than $1/\tau$, where τ is the refractory period duration of the neuron generating the sub-sequence.

Definition 4.12 (Burst sequence)**.** A burst sequence is a spike sequence composed by bursts, subjected to the following constraint: the time difference between the last spike of each burst and the first spike of the next burst it greater than τ, where τ is the refractory period duration of the neuron generating the sequence.



(a) Tonic bursting representation for continuous signals from [21].

(b) The synchronous model edited to be able to reproduce the tonic bursting behavior. Variations with respect to the original model shown in figure 3.2 are blue-colored. Let n be the number of spikes a burst is composed of, then such automaton has $n-1$ locations $\mathbf{B}_i$. One may include the green-colored part to model bursts composed by *at least* n spikes.

Figure 4.7: Tonic bursting: example and model variant.

Thus, "tonic bursting" is the behavior of a neuron producing a burst subsequence as a response to a persistent and excitatory input sub-sequence. An example is shown in figure 4.7a.

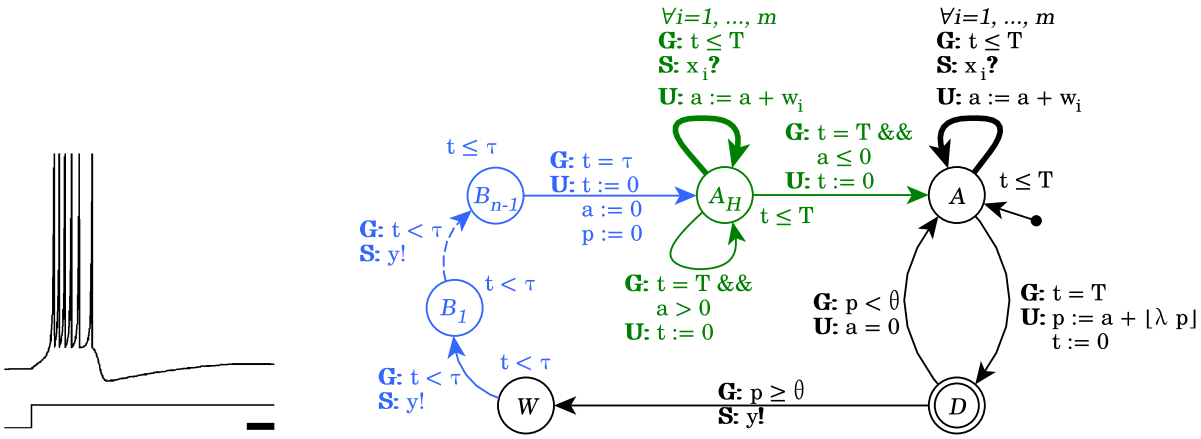
Such a behavior, differently from phasic spiking, does not require the neuron to have inter-emission memory but it requires the neuron to be able to produce bursts.

Property 4.11. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot produce bursts.

Proof. $\mathcal{N}$ cannot emit spikes having a rate greater than $1/(T+\tau)$, as stated by property 4.2, so it cannot produce bursts. □

Corollary. $\mathcal{N}$ cannot reproduce the *tonic bursting* behavior.

The synchronous model can be edited as shown in figure 4.7b in order to make it able to reproduce such a behavior. This variant simply makes the neuron emit bursts instead of spikes. The synchronous model is re-defined as a tuple $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau, n)$ where n is the number of spikes that each burst emitted by $\mathcal{N}$ will contain. If we also consider the green $(\mathbf{B}_{n-1} \to \mathbf{B}_{n-1})$ loop



(a) Phasic bursting representation for continuous signals from [21].

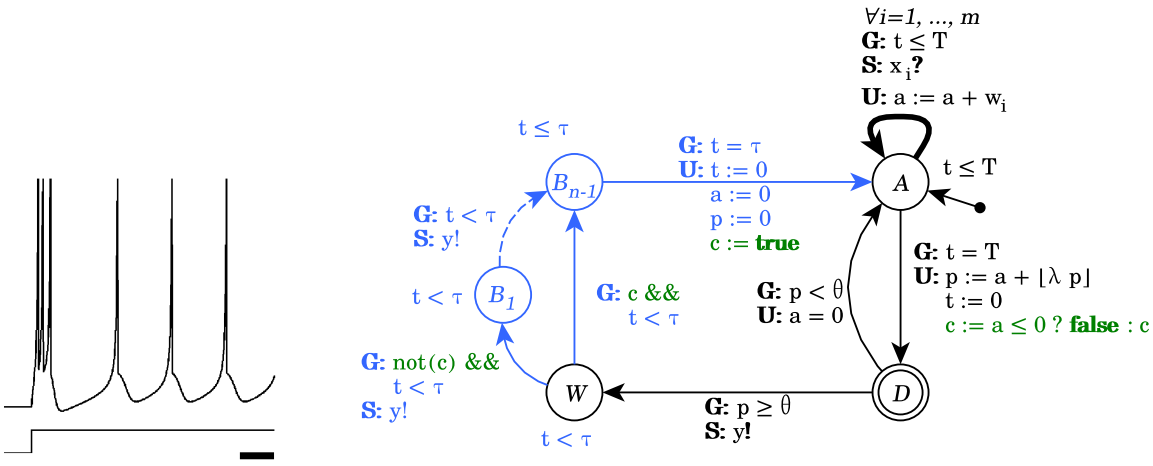
(b) The synchronous model edited to be able to reproduce the phasic bursting behavior. Variations with respect to the original model shown in figure 3.2 are colored. Note that this variant is achieved by mixing the phasic spiking (green part) and tonic bursting (blue part) variants.

Figure 4.8: Phasic bursting: example and model variant.

of figure 4.7b, then n is the *minimum* number of spikes composing a burst. The rationale of this edit is straightforward: at the end of each refractory period, the neuron must iterate over $n-1$ locations $\mathbf{B}_i$ until eventually reaching location $\mathbf{A}$, as usual. On each ingoing edge of each location $\mathbf{B}_i$, a spike is fired. Because of the invariants of locations $\mathbf{B}_i$, the entire burst emission cannot last longer than the refractory period. Please note that, if $n = 1$, then $\mathbf{B}_{n-1} \equiv \mathbf{W}$ and the automaton degenerates to the original synchronous neuron structure of figure 3.2.

Phasic bursting. "Phasic bursting" is the behavior of a neuron producing a burst *on the onset* of a persistent excitatory input sub-sequence and then remaining quiescent until the end of such sub-sequence. An example is shown in figure 4.8a.

Such a behavior, similarly to phasic spiking, requires the neuron to have inter-emission memory, in order to detect the beginning of an excitatory input sub-sequence, and, analogously to tonic bursting, depends on the neuron to be able to produce bursts.



(a) Bursting-then-spiking representation for continuous signals from [21].

(b) The synchronous model edited to be able to reproduce the bursting-then-spiking behavior. Variations with respect to the original model shown in figure 3.2 are colored. The blue-colored part is needed to produce bursts while the green-colored edits are needed to keep track of the onset of an excitatory input sub-sequence.

Figure 4.9: Bursting-then-spiking: example and model variant.

Property 4.12. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *phasic bursting* behavior.

Proof. According to property 4.11, $\mathcal{N}$ cannot produce bursts, thus it cannot reproduce the *phasic bursting* behavior. □

The synchronous model can be edited as shown in figure 4.8b in order to make it able to reproduce such a behavior. This variant simply merges the edits proposed for phasic spiking (green part) and tonic bursting (blue part).

Bursting-then-spiking. "Bursting-then-spiking" is the behavior of a neuron producing a burst *on the onset* of a persistent excitatory input sub-sequence and then producing a periodic output sub-sequence until the end of such sub-sequence. An example is shown in figure 4.9a.

Such a behavior, similarly to phasic bursting, requires the neuron to have inter-emission memory, in order to detect when a persistent subsequence is beginning, and depends on the neuron to be able to produce bursts.

Property 4.13. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *bursting-then-spiking* behavior.

Proof. According to property 4.11, $\mathcal{N}$ cannot produce bursts, thus it cannot reproduce the *bursting-then-spiking* behavior. □

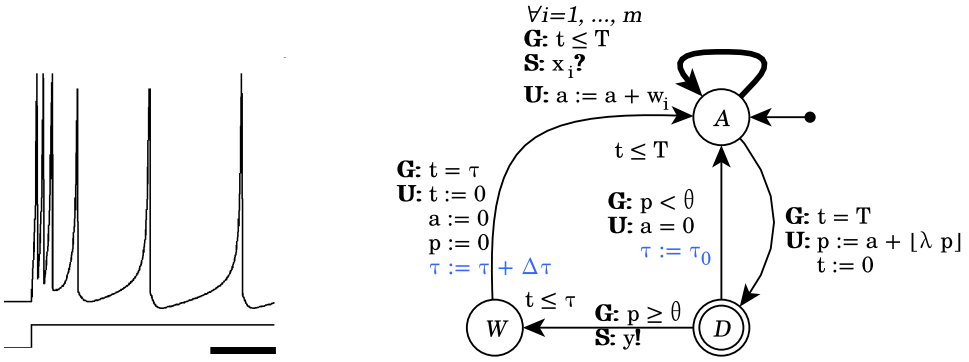
The synchronous model can be edited as shown in figure 4.9b in order to make it able to reproduce such a behavior. This variant, similarly to the one proposed for tonic bursting, comprehends locations $\mathbf{B}_1, \ldots, \mathbf{B}_{n-1}$, allowing it to produce n-bursts. Moreover, the model is extended by means of the c boolean variable keeping track of persistent excitatory input sub-sequences. More precisely, c is set at the end of each *refractory* period and it is reset at the end of any *accumulation* period where the sum of weighted input is non-positive. So, at the end of an accumulation period, if the neuron just emitted one spike and $c = \mathbf{false}$ (i.e., it's the first output spike since the beginning of the current input sub-sequence), it will emit $n - 1$ more spikes in order to compose a burst. Conversely, if $c = \mathbf{true}$, then it will not produce any further spike until the end of the next accumulation period.

Spike frequency adaptation. "Spike frequency adaptation" is the behavior of a neuron producing a decreasing-frequency output sub-sequence as a response to a persistent excitatory input sub-sequence. The inter-emission time difference increases as the time elapsed since the onset of the input sub-sequence and resets to the initial value at the end of such a sub-sequence. An example is shown in figure 4.10a.

This behavior requires the neuron to have inter-emission memory: it should be able to keep track of the time elapsed since the beginning of the input sub-sequence.

Property 4.14. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *spike frequency adaptation* behavior.

Proof. It is sufficient to prove that such behavior requires the neuron to have inter-emission memory. In fact, the spike frequency adaptation behavior requires the neuron to detect the beginning instant of an excitatory input sub-sequence and to increase the time required to fire a spike, after each emission.



(a) Spike frequency adaptation representation for continuous signals from [21].

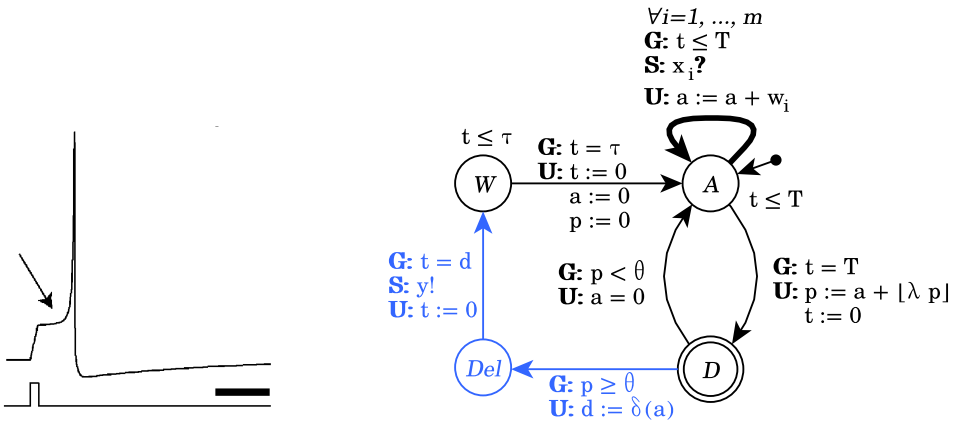
(b) The synchronous model edited to be able to reproduce the spike frequency adaptation behavior. Variations with respect to the original model shown in figure 3.2 are blue-colored.

Figure 4.10: Spike frequency adaptation: example and model variant.

This means the neuron will produce different outcomes as response to equal inputs, which is impossible for any synchronous neuron, as stated in property 4.3. □

The synchronous model can be edited as shown in figure 4.10b in order to make it able to reproduce such a behavior. This variant allows the refractory period to increase after each neuron emission thus making the output frequency decrease. More precisely, the synchronous model is re-defined as a tuple $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau_0, \Delta\tau)$ where $\tau_0 \in \mathbb{N}^+$ is the default refractory period duration and $\Delta\tau \in \mathbb{N}^+$ represents the refractory period variation, while τ is a variable of automaton $\mathcal{N}$. The initial value of variable τ is τ_0. On every firing of edge $(\mathbf{W} \rightarrow \mathbf{A})$ the variable is increased of $\Delta\tau$, while, on every firing of $(\mathbf{D} \rightarrow \mathbf{A})$, it is reset to τ_0. So, any persistent excitatory input sub-sequence will lead to an output sub-sequence having a decreasing frequency.

Spike latency. "Spike latency" is the behavior of a neuron firing delayed spikes, with respect to the instant when its potential reached or overcame the threshold. Such a delay is proportional to the strength of the signal which lead it to emission, i.e., for a synchronous neuron, the sum of weighed inputs received during the accumulation period preceding the emission. An example is shown in figure 4.11a.



(a) Spike latency representation for continuous signals from [21].

(b) The synchronous model edited to be able to reproduce the spike latency behavior. Variations with respect to the original model shown in figure 3.2 are blue-colored.

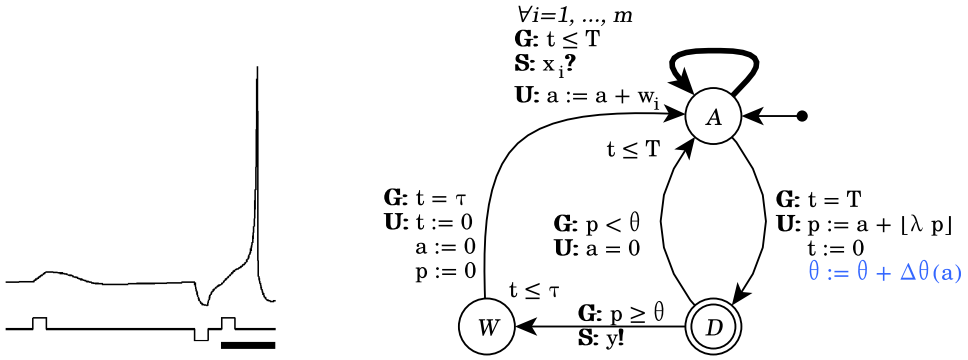
Figure 4.11: Spike latency: example and model variant.

This behavior does not require the neuron to have inter-emission memory, nevertheless it requires the neuron to be able to postpone its outcome.

Property 4.15. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *spike latency* behavior.

Proof. Let n be the first accumulation period since the last reset time where the potential reaches or overcomes θ. Then, according to definition 3.2, the neuron emission instant will be exactly $n \cdot T$: because of location $\mathbf{D}$ being *committed*, there is no way for the neuron to postpone its firing instant. □

The synchronous model can be edited as shown in figure 4.11b in order to make it able to reproduce such a behavior. The proposed variant introduces a delay between the instant the neuron reaches or overcomes its threshold and actual emission instant. Such a delay depends solely on the sum of weighted inputs from the last accumulation period. More formally, the synchronous model is re-defined as a tuple $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau, \delta)$ where $\delta : \mathbb{N} \to \mathbb{N}$ is the function computing the delay according to the sum of weighted inputs. At the end of each accumulation period, if the potential is greater than or equal to the threshold, the neuron will compute the delay duration $\delta(a)$, assigning it



(a) Threshold variability representation for continuous signals from [21]: the inhibitory spike decreases the neuron threshold making the following excitatory spike sufficient to make the neuron fire.

(b) The synchronous model edited to be able to reproduce the Threshold variability behavior. Variations with respect to the original model shown in figure 3.2 are blue-colored.

Figure 4.12: Threshold variability: example and model variant.

to an integer variable d and then wait in location $\mathbf{Del}$ for d time units before emitting a spike on channel y.

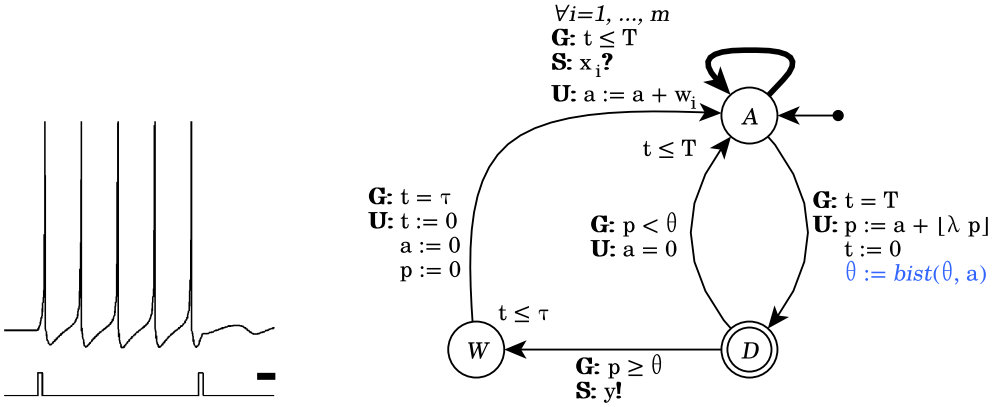
Threshold variability. "Threshold variability" is the behavior of a neuron allowing its threshold to vary according to the strength of its income. More precisely, an excitatory input will rise the threshold while an inhibitory input will decrease it. As a consequence of such a behavior, excitatory inputs may more easily lead the neuron to fire when occurring after an inhibitory input, as shown in the example of figure 4.12a.

This behavior does not require the neuron to have inter-emission memory, nevertheless it requires the neuron threshold to vary according to its inputs.

Property 4.16. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *threshold variability* behavior.

Proof. This is true by construction according to definition 3.2: the neuron threshold never changes. □

The synchronous model can be edited as shown in figure 4.12b in order to make it able to reproduce such a behavior. This variant allows the threshold to



(a) Bistability representation for continuous signals from [21].

(b) The synchronous model edited to be able to reproduce the bistability behavior. Variations with respect to the original model shown in figure 3.2 are blue-colored.

Figure 4.13: Bistability: example and model variant.

vary after each accumulation period according to the current sum of weighted inputs. More precisely, the synchronous model is re-defined as a tuple $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta_0, \tau, \Delta\theta)$ where $\theta_0 \in \mathbb{N}^+$ is the initial threshold and $\Delta\theta : \mathbb{Z} \to \mathbb{Z}$ represents the threshold variation function, while θ is a variable of automaton $\mathcal{N}$. The threshold variable initial value is θ_0. On every firing of edge $(\mathbf{A} \to \mathbf{D})$ the threshold variable is increased of $\Delta\theta(a)$, which is an integer value whose sign is opposite to the sign of a and whose magnitude is proportional to the magnitude of a, where a is the sum of weighted inputs occurred during the last accumulation period.

Bistability. "Bistability" is the behavior of a neuron alternating between two modes of operation: *periodic emission* and *quiescence*. During the former mode, it emits a periodic output sub-sequence, even if it receives no excitatory spike. During the quiescent mode, it does not emit. The neuron switches from one mode to the other every time it receives an excitatory spike. An example is shown in figure 4.13a.

Such a behavior requires the neuron to (i) be able to produce a periodic output sub-sequence, even if no excitatory spike is received, (ii) be able to *not* produce any output when no spike is received, (iii) be able to switch between

the two modes of operation when an excitatory spike is received.

Property 4.17. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *bistability* behavior.

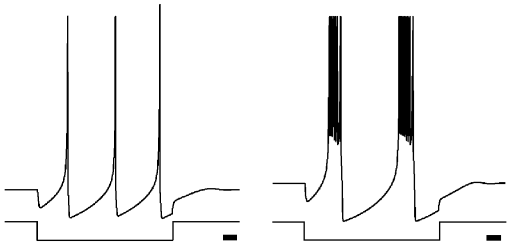
Proof. It is sufficient to prove that (i) $\mathcal{N}$ cannot produce a periodic output sub--sequence if no excitatory spike is received, or (ii) $\mathcal{N}$ cannot remain quiescent if no spike is received, or (iii) $\mathcal{N}$ cannot switch between the two modes of operation. If $\theta = 0$ and no excitatory spike is received, $\mathcal{N}$ produces a periodic output sub-sequence: it is a degenerate case of property 4.6. Conversely, if $\theta > 0$ and no input is received, then $\mathcal{N}$ remains quiescent. Since, by construction, the threshold cannot vary, there is no way for the neuron to switch between the two modes of operation. □

The synchronous model can be modified as shown in figure 4.13b in order to make it able to reproduce such a behavior. This variant simply makes its threshold switch between 0 and a positive value at the end of any accumulation period during which it received an excitatory sum of weighted inputs. A null threshold would make the neuron emit even if no input is received. Conversely, a positive threshold would prevent the neuron from emitting, if no input is received. More precisely, the synchronous model is re-defined as a tuple $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta_0, \tau)$ where $\theta_0 \in \mathbb{N}^+$ is the initial threshold value, while θ is a variable of automaton $\mathcal{N}$. On every firing of edge $(\mathbf{A} \to \mathbf{D})$, i.e., at the end of every accumulation period, the threshold value θ is computed by means of a function $bist(\cdot)$ defined as follows:

$$bist(\theta, a) = \begin{cases} 0 & \text{if } \theta > 0 \wedge a > 0 \\ \theta_0 & \text{if } \theta = 0 \wedge a > 0 \\ \theta & \text{if } a \leq 0 \end{cases}$$

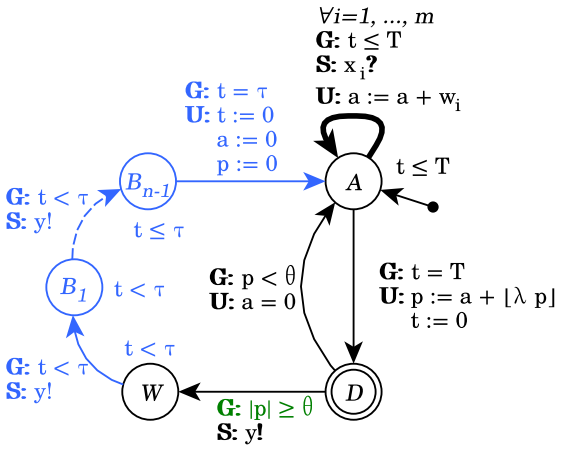
So every accumulation period where the sum of weighted inputs is positive makes the neuron threshold switch between the 0 and θ_0 values.

Inhibition-induced activities. "Inhibition-induced spiking" (resp. "bursting") is the behavior of neuron producing a spike (resp. burst) output sub-sequence



(a) Inhibition-induced spiking.

(b) Inhibition-induced bursting.



(c) The synchronous model edited to be able to produce inhibition-induced activities. Variations with respect to the original model shown in figure 3.2 are blue-colored. The blue part allows the neuron to emit bursts. The degenerate case $n = 1$ simply emits spikes. The green edit allows the neuron to after a number of inhibitory inputs.

Figure 4.14: Inhibition-induced activities: examples and model variants. Images, representing continuous signals, are from [21].

as a response to a persistent *inhibitory* input sub-sequence. Examples are shown in figure 4.14.

Both behaviors requires the neuron to be able to emit as a consequence of some inhibitory input spikes. Particularly, inhibition induced *bursting* also depends on the neuron to be able to produce bursts.

Property 4.18. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *inhibition-induced spiking* or *inhibition-induced bursting* behavior.

Proof. It is sufficient to recall that inhibitory input spikes cannot lead $\mathcal{N}$ to emit according to property 4.5. Moreover, *inhibition-induced bursting* cannot be reproduced by $\mathcal{N}$ because the latter cannot produce bursts, as stated by property 4.11. □

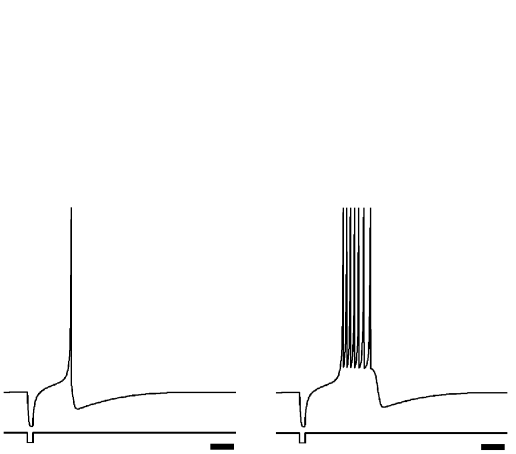
The synchronous model can be edited as shown in figure 4.14c in order to make it able to reproduce such behaviors. For what concerns the inhibition-induced spiking behavior, we propose a variant where the neuron emits whenever *the absolute value* of its potential reaches or overcomes the threshold. The inhibition-induced bursting behavior is obtained by adding locations $\mathbf{B}_1, \ldots, \mathbf{B}_{n-1}$ as described in the tonic bursting paragraph.

Rebound activities. "Rebound spike" (resp. "burst") is the behavior of a neuron producing an output spike (resp. burst) after it received an inhibitory input. Examples are shown in figure 4.15.

Similarly to inhibition-induced activities, these behaviors require the neuron to be able to emit as a consequence of an inhibitory input spike. Furthermore, rebound *bursting* also depends on the neuron to be able to produce bursts.

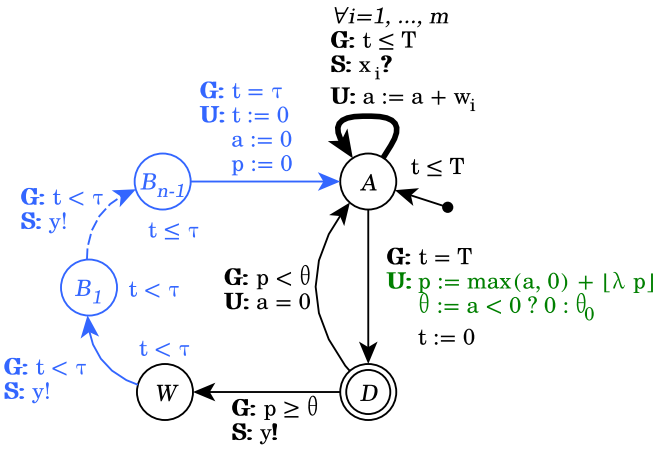
Property 4.19. Let $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta, \tau)$ be a synchronous neuron, then $\mathcal{N}$ cannot reproduce the *rebound spiking* or *rebound bursting* behavior.

Proof. It is sufficient to recall that inhibitory input spikes cannot lead $\mathcal{N}$ to emit according to property 4.5. Moreover, rebound *bursting* cannot be reproduced by $\mathcal{N}$ because the latter cannot produce bursts, as stated by property 4.11. □



(a) Rebound spike.

(b) Rebound burst.



(c) The synchronous model edited to be able to produce inhibition-induced activities. Variations with respect to the original model shown in figure 3.2 are blue-colored. The blue part allows the neuron to emit bursts. The degenerate case $n = 1$ simply emits spikes. The green edit allows the neuron to produce a rebound spike/burst after an inhibitory input.

Figure 4.15: Rebound activities: examples and model variants. Images, representing continuous signals, are from [21].

The synchronous model can be edited as shown in figure 4.15c in order to make it able to reproduce such behaviors. For what concerns the rebound spiking behavior, we propose a variant where the neuron potential is always non-negative and the threshold is set to 0 by inhibitory stimulations. We recall that a null threshold would make the neuron emit even if its potential is 0. More precisely, the synchronous model is re-defined as a tuple $\mathcal{N} = (\mathbf{w}, T, \lambda, \theta_0, \tau)$ where $\theta_0 \in \mathbb{N}^+$ is the nominal threshold value, while θ is a variable of automaton $\mathcal{N}$. On every firing of the edge $(\mathbf{A} \rightarrow \mathbf{D})$, i.e., at the end of every accumulation period, if the current sum of weighted inputs a is negative, the threshold θ is set to 0, otherwise it is set to θ_0. Thus, an inhibitory stimulus will produce a rebound spike. The rebound *bursting* behavior is obtained by adding locations $\mathbf{B}_1, \ldots, \mathbf{B}_{n-1}$ as described in the tonic bursting paragraph.

Chapter 5

Parameters tuning and learning

In this chapter we study the problem of finding the parameters assignments for a network having a given topology, allowing it to reproduce a given behavior. We focus on the case where synaptic weights are the only parameters to be computed and provide a novel methodology, namely the *advice back-propagation* (ABP) approach. The ABP method is a *supervised* approach to synaptic weights estimation relying on our formalization of the leaky-integrate-and-fire neural networks presented in chapter 3 and inspired by the supervised and unsupervised learning approaches discussed in section 2.2, in particular *SpikeProp* and *spike-timing dependent plasticity* (STDP). Similarly to SpikeProp, ABP tries to reduce the errors performed by a spiking neural network with respect to some expected output sequences. As soon as an error is detected for a given output neuron, the proper corrective action, an "advice", is sent to the neuron. Each advice can either be back-propagated to the predecessors of the neuron, or stimulate a synaptic weight variation for some ingoing synapses of the neuron. Analogously to STDP, our weights update rule takes into account the relative firing times of the predecessors and no assumption is required about the topology of the network. Finally, we illustrate two possible ways for realizing our ABP approach: the *simulation-oriented* one, leveraging an extended version of our reference model, and the *model-checking-oriented* one, employing the model-checker as a guide.

5.1 Learning problem for networks of synchronous neurons

In this section we study the *learning problem*, i.e., the problem of finding a parameter assignment making a given network able to produce a desired behavior.

The learning problem is here considered a satisfaction problem. The inputs sequences consumed by a spiking neural network S are encoded into input generators and represents the input neurons of S. The *expected* output sequences of such a network can be encapsulated within a *temporal logic* formula. Solving the learning problem means finding a parameters assignment allowing a spiking neural network encoded into a timed automata network to satisfy such a temporal logic formula, i.e., making the network behave as expected.

For instance, imagine a system $\mathcal{I} \overset{x_1}{\parallel} \mathcal{N}_1 \overset{x_2}{\parallel} \dots \overset{x_n}{\parallel} \mathcal{N}_n \overset{y}{\parallel} \mathcal{O}_n$, consisting of a series of n neurons where the first one is fed by a fixed-rate input generator $\mathcal{I}$ with arbitrary time window size. Suppose we want to allow the last neuron to eventually emit if the first one is receiving a persistent stimulation. One possible formula explicitly representing the expected output is $\phi = AF(state_{\mathcal{O}_n}(\mathbf{O}))$, where $\mathbf{O}$ is the location an output consumer reaches after receiving a spike. Notice that several sets of parameters may actually satisfy ϕ.

In what follows we will focus, for simplicity, on the *weight assignment problem*, i.e., a particular case of the *learning problem* where the following assumptions hold: (i) the network topology is supposed to be known and fixed; (ii) the leak factors, the thresholds, and the refractory periods duration are supposed to be given and fixed, for all neurons; (iii) the accumulation periods are supposed to be known and fixed to their minimal values, i.e., equal to 1. In this section we define a novel approach, namely the *advice back-propagation* (ABP), to the *weights assignment problem*, aiming to estimate the synaptic weights of some network composed by leaky-integrate-and-fire neurons encoded into timed automata, supposing the other parameters are given and fixed.

ABP is inspired by both *spike-timing dependent plasticity* (STDP) and SpikeProp. Similarly to STDP, our approach modifies the weights of some neuron ingoing synapses only exploiting *local* information and, therefore, regardless of the structure of the whole network. Differently from STDP, the ABP approach takes into account not only the recently received spikes but some external feedback, the *advice*, in order to determine which weights must be modified and whether they should increase or decrease. Moreover, ABP does not prevent excitatory synapses from becoming inhibitory (or vice versa), which is usually a constraint for STDP implementations. Analogously to SpikeProp, ABP states weights should be updated according to the time-difference between the expected and actual firing times, so they are both supervised approaches. Moreover, ABP takes into account the discrete nature of our formalization of the leaky-integrate-and-fire neuron and does not relies on synaptic delays or multi-layered network topology, which are instead assumed by SpikeProp reference model.

The information about the neuron error is carried by the *advices*. We want to minimize the likelihood of receiving again the same advice. At any given time unit, each neuron may fire at most one spike, by construction. So, for each time unit, four different scenarios are possible, with respect to some expected output:

- the neuron fires a spike, while it was supposed to do so, thus it behaved as expected,
- the neuron fires a spike, even if it was supposed to be quiescent, so it *should not have fired* but it did,
- the neuron remains quiescent, even if it was supposed to fire a spike, so it *should have fired* but it did not,
- the neuron remains quiescent, while it was supposed to do so, thus it behaved properly.

The *advice* consists of a “Should Have Fired” (SHF) or “Should Not Have Fired” (SNHF) message to be sent to each neuron not behaving as expected. Whenever a neuron receives an advice it should edit its ingoing synaptic weights accordingly and *back-propagate* the proper advice to its predecessors.

The ABP algorithm essentially lies on a traversal of the graph topology of the network, starting from the output neurons. If a neuron receives more than one advice from the same successor within the same algorithmic step, only the first advice is considered. The algorithmic step is over when the advices reaches the input neurons which simply ignore the advices without propagating them any further. The back-propagation of advices provokes several *local* weights increasing or decreasing by a constant *learning factor* ΔW, aiming at reducing the amount of errors performed by output neurons. It is a positive constant stating how much a synaptic weight may vary at each algorithmic step. It is important to choose a value which is a lot lower than 1 in order to allow the algorithm to explore a wider pool of weights assignments[1].

Algorithms 5.1 and 5.2 show the pseudo-code of two mutually recursive functions realizing the ABP approach. Let $\mathcal{N}$ be the encoding of a neuron, then if the function SHOULD-HAVE-FIRED($\mathcal{N}$) (resp. SHOULD-NOT-HAVE-FIRED($\mathcal{N}$)) is called, we say that $\mathcal{N}$ received a SHF (resp. should not have fired) advice. Moreover, let $\mathcal{N}_i$ be the i-th predecessor of $\mathcal{N}$, than we say that $\mathcal{N}_i$ fired *recently*, with respect to $\mathcal{N}$, if $\mathcal{N}_i$ fired during the current or previous accumulate-fire-rest cycle of $\mathcal{N}$. According to algorithm 5.1, whenever $\mathcal{N}$ receives a SHF advice, it should:

- *Strengthen* the weight of each ingoing *excitatory* synapse corresponding to a neuron which fired recently, because it is already contributing to make $\mathcal{N}$ fire.
- Propagate a SHF advice to each ingoing *excitatory* synapse corresponding to a neuron which *did not* fire recently, in order to encourage it to fire.
- Propagate a SNHF advice to each ingoing *inhibitory* synapse corresponding to a neuron which fired recently, in order to discourage it from firing.
- *Weaken* the weight of each ingoing *inhibitory* synapse corresponding to a neuron which *did not* fire recently, because it is not contributing to the

[1] As for other continuous value (e.g., weights and thresholds) within our conceptual framework, ΔW must be discretized taking into account the discretization granularity constant R, as discussed in section 3.1

Algorithm 5.1 Abstract ABP: *Should Have Fired* advice pseudo-code

Require: $0 < \Delta W \ll 1$ ▷ Learning factor

1: **procedure** SHOULD-HAVE-FIRED($neuron$)
2: **if** IS-VISITED($neuron$) **then**
3: **return**
4: SET-VISITED($neuron$, True)

5: **for** $i \leftarrow 1 \ldots$ COUNT-PREDECESSORS($neuron$) **do**
6: $weight_i \leftarrow$ GET-WEIGHT($neuron$, i)
7: $fired_i \leftarrow$ FIRED-RECENTLY($neuron$, i)
8: $neuron_i \leftarrow$ GET-PREDECESSOR($neuron$, i)

9: **if** $weight_i \geq 0$ **then**
10: **if** $fired_i$ **then**
11: INCREASE-WEIGHT($neuron$, i, ΔW)
12: **else**
13: SHOULD-HAVE-FIRED($neuron_i$) ▷ Advice propagation!
14: **else**
15: **if** $fired_i$ **then**
16: SHOULD-NOT-HAVE-FIRED($neuron_i$) ▷ Advice propagation!
17: **else**
18: INCREASE-WEIGHT($neuron$, i, ΔW)

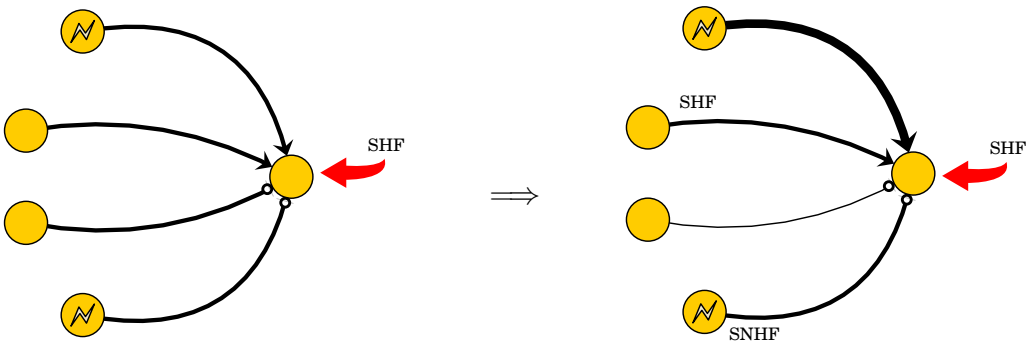
Algorithm 5.2 Abstract ABP: *Should Not Have Fired* advice pseudo-code

Require: $0 < \Delta W \ll 1$ ▷ Learning factor

1: **procedure** SHOULD-NOT-HAVE-FIRED($neuron$)
2: **if** IS-VISITED($neuron$) **then**
3: **return**
4: SET-VISITED($neuron$, True)

5: **for** $i \leftarrow 1 \ldots$ COUNT-PREDECESSORS($neuron$) **do**
6: $weight_i \leftarrow$ GET-WEIGHT($neuron$, i)
7: $fired_i \leftarrow$ FIRED-RECENTLY($neuron$, i)
8: $neuron_i \leftarrow$ GET-PREDECESSOR($neuron$, i)

9: **if** $weight_i \geq 0$ **then**
10: **if** $fired_i$ **then**
11: SHOULD-NOT-HAVE-FIRED($neuron_i$) ▷ Advice propagation!
12: **else**
13: DECREASE-WEIGHT($neuron$, i, ΔW)
14: **else**
15: **if** $fired_i$ **then**
16: DECREASE-WEIGHT($neuron$, i, ΔW)
17: **else**
18: SHOULD-HAVE-FIRED($neuron_i$) ▷ Advice propagation!



(a) Before SHF (b) After SHF

Figure 5.1: Graphical representation of the effect of a Should Have Fired (SHF) advice over a neuron. Yellow circles represent neurons while edges represent synapses. A thunder symbol is used to indicate neurons that fired recently. The thickness of the edges represent the absolute value of the weight of a synapse, while the shape of the ending discriminates between excitatory and inhibitory synapses. An arrow indicates an excitatory synapse while an empty circle indicates an inhibitory one.

firing of $\mathcal{N}$.

Figure 5.1 shows a graphical representation of the effect of a SHF advice of a neuron. The effect of a SNHF advice is dual. According to algorithm 5.2, whenever $\mathcal{N}$ receives such an advice, it should:

- *Weaken* the weight of each ingoing *excitatory* synapse corresponding to a neuron which *did not* fire recently.
- Propagate a SNHF advice to each ingoing *excitatory* synapse corresponding to a neuron which fired recently.
- Propagate a SHF advice to each ingoing *inhibitory* synapse corresponding to a neuron which *did not* fire recently.
- *Strengthen* the weight of each ingoing *inhibitory* synapse corresponding to a neuron which fired recently.

If $\mathcal{N}_i$ is an input generator it will simply ignore any advice received from $\mathcal{N}$ because the input sequences should not be altered by the learning process.

Finally, we noted that the convergence of the ABP algorithm to the desired weight assignments can be reached more quickly by introducing another learning factor δw, the *small* one. We assume that $\delta w < \Delta W$ and ΔW is now referred as the *big* learning factor.

The small learning factor is used to produce *small* weights *increasing* variations just before invoking the SHOULD-HAVE-FIRED() or SHOULD-NOT-HAVE-FIRED() procedures in algorithm 5.1 (i.e., before lines 13 and 16). It is also used to produce *small* weights *decreasing* variations just before invoking the same procedures in algorithm 5.2 (i.e., before lines 18 and 11). The original weights variations, i.e., lines 11 and 18 in algorithm 5.1, and lines 13 and 16 in algorithm 5.2, are now referred as *big* weights variations.

When it comes to implement some system taking into account such an improvement, both learning factors are realized as integers subjected to the following constraint:

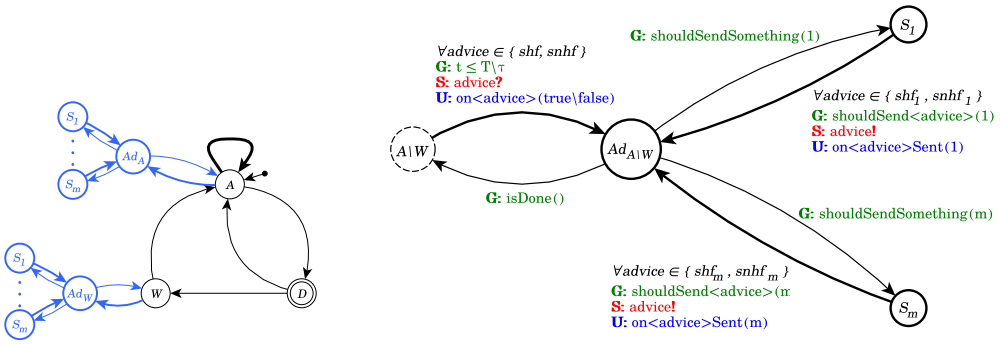
$$0 < \delta w < \Delta W \ll R$$

where R is the discretization granularity.

5.2 Simulation-oriented advice back-propagation

In what follows we propose a simulation-oriented implementation of the ABP approach where parameters tuning on the spiking neural network happens at run time. This approach requires some changes to the structure of the automata generated by the synchronous encoding of leaky-integrate-and-fire neurons. The automata encodings now have to handle SHF and SNHF advices and to be able to edit their synaptic weights accordingly. Output neurons are connected to one or more *supervisor* automata, which are essentially extended output consumers. They are built reflecting the expected outcome for the neurons they are linked to and they are responsible for analyzing the outcome of such neurons and for sending them the proper advice as soon as an error is detected. Whenever a supervisor automaton detects neurons actually learned to reproduce the proper outcome, it moves to an acceptation location. The learning process is considered over when all the supervisor automata are in the *acceptation* location. In this case the last values of all synaptic weights represent the result of the learning process. We call this implementation *simulation-oriented* because the network must be simulated until eventually achieving the result of the learning process.

The *synchronous* encoding is then extended as follows:



(a) Overview of the parts to be added to the automata attained as synchronous encoding of LI&F neurons to make them able to receive and propagate advices, during both the accumulation and refractory periods.

(b) Portion of the extended synchronous encoding responsible for handling advices. Thick-line edges represents couples of edges, one for each advice. Thick-border locations are *urgent*, i.e., they do not allow the time to progress. So the whole structure consists of $m+1$ locations and $3+3m$ edges.

Figure 5.2: Extension of the synchronous encoding making the produced automata able to handle the advices. The structure from figure 5.2b is attached to both locations $\mathbf{A}$ and $\mathbf{W}$ of the original synchronous encoding, as shown in figure 5.2a.

Definition 5.1 (*Extended* synchronous encoding)**.** Let $N = (\theta,\ \lambda,\ p,\ \tau,\ y)$ be a *leaky-integrate-and-fire neuron*, let m be the number of ingoing synapses of N, let $w_1,\ \ldots,\ w_m$ be the weights of such synapses, and let $T \in \mathbb{N}^+$ be the duration of the *accumulation period*, then the *extended* synchronous encoding of N into timed automata is a tuple $[\![N]\!]^{ext}_{syn} = (L,\ \mathbf{A},\ X,\ Inv,\ \Sigma,\ Arcs)$, where:

- $L = \{\mathbf{A}, \mathbf{W}, \mathbf{D}\} \cup L_{ext}$, where $L_{ext} = \{\mathbf{Ad_A}, \mathbf{Ad_W}\} \cup \{\mathbf{S}_i^{(\mathbf{A})}, \mathbf{S}_i^{(\mathbf{W})} \mid i = 1 \ldots m\}$, all locations in L_{ext} are urgent and $\mathbf{D}$ is committed
- $X = \{t\}$
- $\Sigma = \{y,\ shf,\ snhf\} \cup \{x_i,\ shf_i,\ snhf_i \mid i = 1 \ldots m\}$
- $Inv = \{\mathbf{A} \mapsto (t \leq T),\ \mathbf{W} \mapsto (t \leq \tau)\}$,
- $Arcs = Arcs_{old} \cup Arcs_{done} \cup Arcs_{advice} \cup \bigcup_{i=1}^{m} \left(Arcs^{(i)}_{check} \cup Arcs^{(i)}_{sent}\right)$ where :
  - $Arcs_{old} = \{(\mathbf{A},\ t \leq T,\ x_i?,\ \{a := a + w_i\},\ \mathbf{A}) \mid i = 1 \ldots m\} \cup \{(\mathbf{A},\ t = T,\ \varepsilon,\ \{p := a + \lfloor \lambda p \rfloor\},\ \mathbf{D}),\ (\mathbf{D},\ p < \theta,\ \varepsilon,\ \{a := 0\},\ \mathbf{A}),\ (\mathbf{D},\ p \geq \theta,\ y!,\ \{\},\ \mathbf{W}),\ (\mathbf{W},\ t = \tau,\ \varepsilon,\ \{a := 0,\ t := 0,\ p := 0\},\mathbf{A})\}$

- $Arcs_{done} = \{(\mathbf{Ad_A}, \text{isDone}(), \varepsilon, \{\}, \mathbf{A}), (\mathbf{Ad_W}, \text{isDone}(), \varepsilon, \{\}, \mathbf{W})\}$
- $Arcs_{advice} = \{(\mathbf{A}, t < T, shf?, \{\text{onShf}(\mathbf{true})\}, \mathbf{Ad_A}), (\mathbf{A}, t < T, shnf?, \{\text{onSnhf}(\mathbf{true})\}, \mathbf{Ad_A}), (\mathbf{W}, t < T, shf?, \{\text{onShf}(\mathbf{false})\}, \mathbf{Ad_W}), (\mathbf{W}, t < T, shnf?, \{\text{onShf}(\mathbf{false})\}, \mathbf{Ad_W})\}$
- $Arcs^{(i)}_{check} = \{(\mathbf{Ad_A}, \text{shouldSendSomething}(i), \varepsilon, \{\}, \mathbf{S}^{(\mathbf{A})}_i), (\mathbf{Ad_W}, \text{shouldSendSomething}(i), \varepsilon, \{\}, \mathbf{S}^{(\mathbf{W})}_i)\}$
- $Arcs^{(i)}_{sent} = \{(\mathbf{S}^{(\mathbf{A})}_i, \text{shouldSendShf}(i), shf!, \{\text{onShfSent}(i)\}, \mathbf{Ad_A}), (\mathbf{S}^{(\mathbf{A})}_i, \text{shouldSendSnhf}(i), snhf!, \{\text{onSnhfSent}(i)\}, \mathbf{Ad_A}), (\mathbf{S}^{(\mathbf{W})}_i, \text{shouldSendShf}(i), shf!, \{\text{onShfSent}(i)\}, \mathbf{Ad_W}), (\mathbf{W}^{(\mathbf{A})}_i, \text{shouldSendSnhf}(i), snhf!, \{\text{onSnhfSent}(i)\}, \mathbf{Ad_W})\}$

A graphical representation of such an extension is shown in figure 5.2. We now provide an intuition of the semantics of such an extended automaton. The neuron state is enriched by the following boolean variables, for $i = 1, \ldots, m$, where m is the number of ingoing synapses of the neuron:

- $fired_i$ (resp. $prevFired_i$), which is $\mathbf{true}$ if a spike was received on the i-th input channel, during the *current* (resp. *previous*) accumulate-fire-wait cycle;
- $sendShf_i$ (resp. $sendSnhf_i$), which is $\mathbf{true}$ if a SHF (resp. SNHF) advice must be propagated to the i-th predecessor;
- $ignore_i$ which is $\mathbf{true}$ if no more advices must be propagated to the i-th predecessor during the current accumulate-fire-wait cycle.

The neuron behavior is extended by means of the following rules:

- Whenever transitions $(\mathbf{D} \to \mathbf{A})$ or $(\mathbf{W} \to \mathbf{A})$ fire, (i) $prevFired_i$ is set to $fired_i$, (ii) $fired_i$ is reset to $\mathbf{false}$ and (iii) $ignore_i$ is reset to $\mathbf{false}$ too.
- On any firing of any transition $(\mathbf{A} \to \mathbf{A})$, $fired_i$ is set to $\mathbf{true}$.
- For the whole duration T (resp. τ) of each accumulation (resp. refractory) period, the neuron may receive a message over the shf or $snhf$ channels: in both cases it will move to location $\mathbf{Ad_A}$ (resp. $\mathbf{Ad_W}$), invoking either the procedure $\text{onShf}()$ or $\text{onSnhf}()$, depending on the received advice, with actual argument $\mathbf{true}$ (resp. $\mathbf{false}$);

  - procedures $\mathrm{onShf}()$ and $\mathrm{onSnhf}()$ implement Algorithms 5.1 and 5.2, respectively. The only difference from the pseudo code is that they set $sendShf_i$ or $sendSnhf_i$ to $\mathbf{true}$ instead of recursively invoking $\mathrm{onShf}()$ or $\mathrm{onSnhf}()$. The actual propagation of advices on the proper channel is described by the following rules.

- The neuron is allowed to move back to location $\mathbf{A}$ (resp. $\mathbf{W}$) from location $\mathbf{Ad_A}$ (resp. $\mathbf{Ad_W}$) as soon as the $\mathrm{isDone}()$ function evaluates to $\mathbf{true}$, i.e., when all $sendShf_i$ and all $sendSnhf_i$ variables are set to $\mathbf{false}$.

- From location $\mathbf{Ad_A}$ (resp. $\mathbf{Ad_W}$), if $\mathrm{shouldSendSomething}(i)$ evaluates to $\mathbf{true}$ for some $i \in \{1, \ldots, m\}$, i.e., any of $sendShf_i$ or $sendSnhf_i$ is $\mathbf{true}$, the neuron can move to location $\mathbf{S}_i^{(\mathbf{A})}$ (resp. $\mathbf{S}_i^{(\mathbf{W})}$).

- When moving back to location $\mathbf{Ad_A}$ (resp. $\mathbf{Ad_W}$) from location $\mathbf{S}_i^{(\mathbf{A})}$ (resp. $\mathbf{S}_i^{(\mathbf{W})}$), the neuron will propagate an advice on the shf_i channel if $sendShf_i$ is $\mathbf{true}$ or on the $snhf_i$ channel if $sendSnhf_i$ is $\mathbf{true}$. Such checks are performed within the $\mathrm{shouldSendShf}(i)$ and $\mathrm{shouldSendShf}(i)$ functions, respectively. Depending on which advice is sent, either the procedure $\mathrm{onShfSent}(i)$ or $\mathrm{onSnhfSent}(i)$ is invoked. Their aim is, respectively, to reset both $sendSnhf_i$ and $sendSnhf_i$ to $\mathbf{false}$. In both cases $ignore_i$ is reset to $\mathbf{false}$, too.

Since all locations introduced to handle the advices are *urgent*, the advice back-propagation process described above is *instantaneous*.

The learning process is led by *supervisors*, i.e., the automata in charge of back-propagating advices to the output neurons. Each supervisor is *responsible* for one or more output neurons, i.e., it can only send advices to a predefined subset of the output neurons of the network. The design of each supervisor automaton heavily depends on the specific behavior the underlying network must learn. Since behaviors are arbitrary and may differ from each other, we only provide some guidelines for the design of supervisor automata:

- In order to be able to send advices to the output neurons they are responsible for, supervisors must share their snf and $snhf$ channels.

- Supervisors could in general take into account the inner state of the network, i.e., any variable or channel of any neuron or input generator, to determinate the right advice to back-propagate.
- Supervisors should directly send advices only to output neurons, the inner ones should only be affected by the back-propagated advices.
- As soon as a supervisor detects that all the output neurons it is responsible for are able to produce the expected outcome, it should move to an *acceptation location* where no more advices are back-propagated and no more transitions are enabled.

The learning process ends when all supervisors reach their acceptation location. So, in order to solve the *weights assignment problem*, one must simulate a network composed by input generators, extended synchronous neurons and supervisors until all supervisors reach their acceptation location.

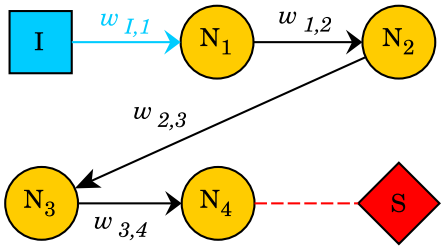
Supposing we are exploiting k supervisors $\mathcal{S}_1, \ldots, \mathcal{S}_k$ to lead the learning process, we can submit the following query to a CTL model-checker, asking it to prove that it is impossible for the learning process to end:

$$AG \neg (state_{\mathcal{S}_1}(\mathbf{Acc}_1) \wedge \ldots \wedge state_{\mathcal{S}_k}(\mathbf{Acc}_k)) \tag{5.1}$$

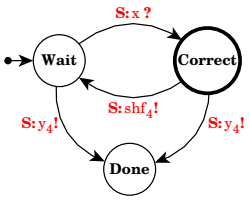
where $\mathbf{Acc}_k$ is the acceptation location of the k-th supervisor. If the model-checker provides a response, two scenarios are possible: (i) the formula is satisfied, i.e., the expected behavior *cannot* be learned by the network (according to the current definition of the supervisor automata); (ii) the formula is *not* satisfied and a trace is provided as counterexample, i.e., the expected behavior *can* be learned by the network and the provided trace represents the whole learning process. In this scenario, the solution of the weights assignment problem consists of the values of the weights into the last state of the trace.

We illustrate a few examples of learning processes exploiting the simulation-oriented ABP. The `Uppaal` systems implementing such examples and allowing to reproduce the experiments can be found in [1]. We adopt the following naming conventions:

- shf_i (resp. $snhf_i$) is the channel used to propagate a SHF (resp. SNHF) advice to neuron $\mathcal{N}_i$;



(a) The *serial* network topology



(b) The automaton describing the behavior of supervisor $\mathcal{S}$

Figure 5.3: The series of four neurons shown in fig. 5.3a can be "turned-on" by the supervisor shown in fig. 5.3b by means of the simulation-oriented ABP.

- x_j is the output channel of generator $\mathcal{I}_j$;
- $w_{i,k}$ is the weight of the synapse from $\mathcal{N}_i$ to $\mathcal{N}_k$;
- $w_{I_j,i}$ is the weight of the synapse from $\mathcal{I}_j$ to $\mathcal{N}_i$.

Example 5.1 (Turning on a series of neurons). Here we show how the simulation-oriented ABP can be used to "turn on" (i.e., to make its output neuron able to emit) a spiking neural network having the *serial* structure shown in figure 5.3a. We assume a scenario were $\mathcal{N}_1$ is fed by a fixed-rate input generator $\mathcal{I}$ but no neuron is able to emit because the weights of their input synapses are 0 and their thresholds are higher than 0. We want the network to learn the weights assignment making its output neuron $\mathcal{N}_4$ able to emit.

The learning process is led by the supervisor $\mathcal{S}$ shown in figure 5.3b. It keeps sending SHF advices to $\mathcal{N}_4$, until eventually making it able to reach its threshold and fire. More precisely, such an automaton behaves as follows:

- It begins its execution within the $\mathbf{Wait}$ location.
- In location $\mathbf{Wait}$, if the input generator $\mathcal{I}$ fires a spike over the x channel, the supervisor moves to the $\mathbf{Correct}$ location, otherwise, if the output neuron $\mathcal{N}_4$ fires a spike over the y_4 channel, it moves to the $\mathbf{Done}$ location.
- In location $\mathbf{Correct}$, the supervisor may move to the $\mathbf{Done}$ location if it receives a spike over the y_4 channel or it may move back to the $\mathbf{Wait}$ location, sending a SHF advice to $\mathcal{N}_4$ over the shf_4 channel:

  - such an advice is back-propagated making the synaptic weights of the network vary.

- Location $\mathbf{Done}$ represents the acceptation location of the supervisor: it moves there as soon as $\mathcal{N}_4$ fires a spike. Indeeds, the first spike of the output neurons proves that it is able to emit.

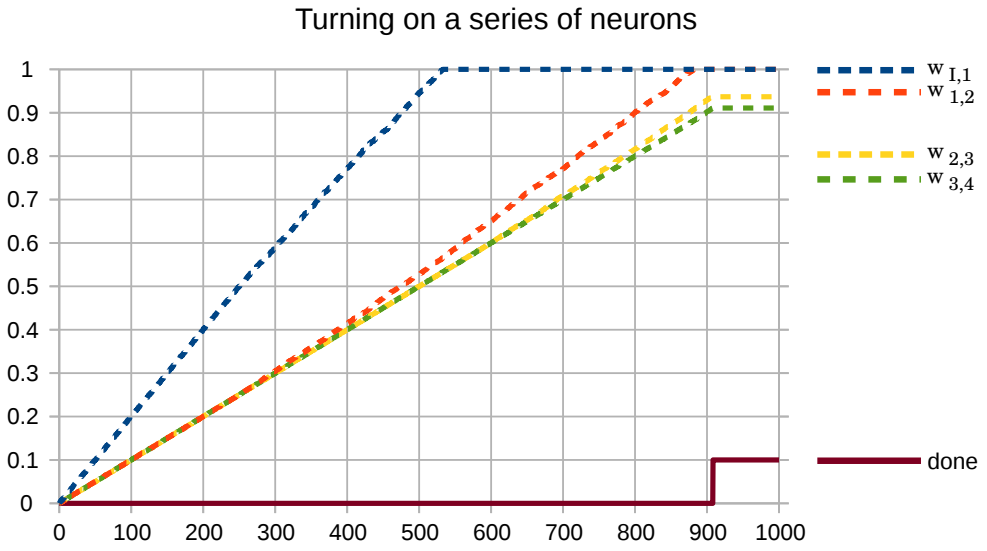


Figure 5.4: Representation of the evolution of the weights over time, for a simulation run. The parameters of all neurons $\mathcal{N}_i$ are: $T_i = 1$, $\tau_i = 1$, $\theta_i = R$ and $\lambda_i = 1/2$, where $R = 1000$ is the discretization granularity. The chosen learning rates are $\Delta W = 2$ and $\delta w = 1$. At the very beginning of the simulation, all weights are 0. The weight $w_{I,1}$, of the synapse feeding the first neuron, is the one rising the most and more quickly. It stabilizes at value 1 after 533 time units. The weight $w_{1,2}$, of the synapse feeding the second neuron, slowly stabilizes at value 1, too, after 881 time units. The weights $w_{2,3}$ and $w_{3,4}$, of the synapses connecting the first to the second and the third neuron, respectively, rise almost at the same rate. They reach the values 0.94 and 0.91, respectively, after 908 time units. At time unit 908 the supervisor $\mathcal{S}$ reaches its acceptation state (continuous red line), terminating the learning process.

Figure 5.4 shows a simulation run: the abscissas axis represents time and the ordinate axis represents the weight value. The supervisor expects to receive a spike from $\mathcal{N}_4$ every time the generator $\mathcal{I}$ fires. As soon as $\mathcal{I}$ emits, the supervisor requires a spike but, as all weights are equal to zero, no emission can happen. Thus a SHF advice is back-propagated to neurons $\mathcal{N}_4$ (and consequently to $\mathcal{N}_3$ and so on) every time $\mathcal{I}$ fires and $\mathcal{N}_4$ does not.

SHF advices initially produce *small* positive variations for all weights, except $w_{I,1}$. The weight $w_{I,1}$ raises more quickly because the predecessor of $\mathcal{N}_1$,

namely $\mathcal{I}$, is always firing recently. Around time unit 300, $w_{I,1}$ is great enough to make $\mathcal{N}_1$ able to fire, so $w_{1,2}$ begin to be affected by *big* weight variations. Around time unit 650, $w_{1,2}$ is great enough to make $\mathcal{N}_2$ able to fire, so $w_{2,3}$ begin to be affected by *big* weight variations, too. This is why they start to rise more quickly at those point.

The simulation continues until $w_{3,4}$ is great enough to enable $\mathcal{N}_4$ for firing.

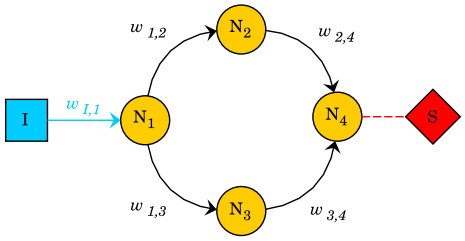


Figure 5.5: The *diamond* network topology. Such a network can be "turned-on" by the supervisor shown in fig. 5.3b by means of the simulation-oriented ABP.

Example 5.2 (Turning on a diamond structure of neurons). The following example is similar to the previous one. Here we show how the simulation-oriented ABP can also be used to "turn on" a spiking neural network having the *diamond* structure shown in figure 5.5. We assume that $\mathcal{N}_1$ is fed by a fixed-rate input generator $\mathcal{I}$ but no neuron is able to emit because the weights of their input synapses are 0 and their thresholds are higher than 0.

We want the network to learn the weights assignment making its output neuron $\mathcal{N}_4$ able to emit. The learning process is led by the supervisor $\mathcal{S}$ shown in figure 5.3b. The behavior of such a supervisor is described example 5.1.

Figure 5.6 shows a simulation run. The supervisor expects to receive a spike from $\mathcal{N}_4$ every time the generator $\mathcal{I}$ fires. As soon as $\mathcal{I}$ emits, the supervisor requires a spike but, as all weights are equal to zero, no emission can happen. Thus a SHF advice is back-propagated to neurons $\mathcal{N}_2$ and $\mathcal{N}_3$ (and consequently to $\mathcal{N}_1$) every time $\mathcal{I}$ fires and $\mathcal{N}_4$ does not.

SHF advices initially produce *small* positive variations for all weights. The variation of $w_{I,1}$ is more rapid because $\mathcal{N}_1$ receives SHF advices both from $\mathcal{N}_2$ and $\mathcal{N}_3$. Around time unit 250, $w_{I,1}$ is great enough to make $\mathcal{N}_1$ able to fire, so $w_{1,2}$ and $w_{1,3}$ begin to be affected by *big* weight variations. This is why they start to rise more quickly.

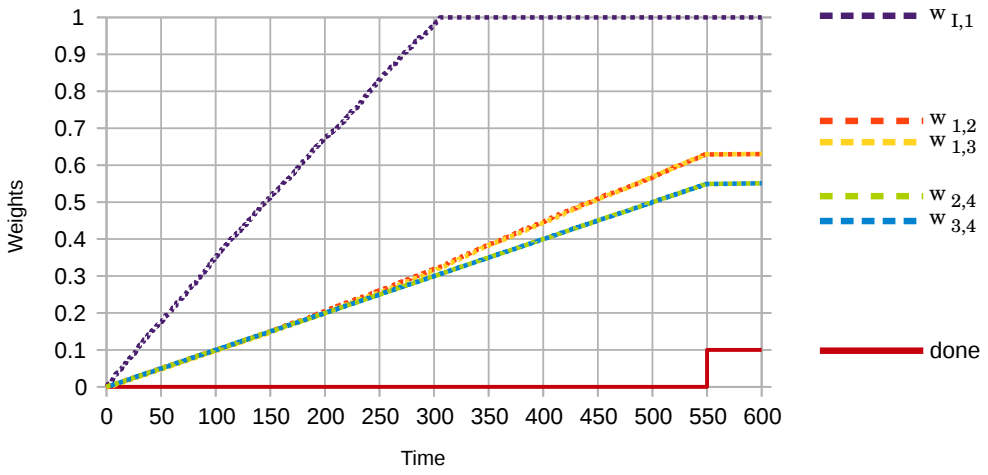


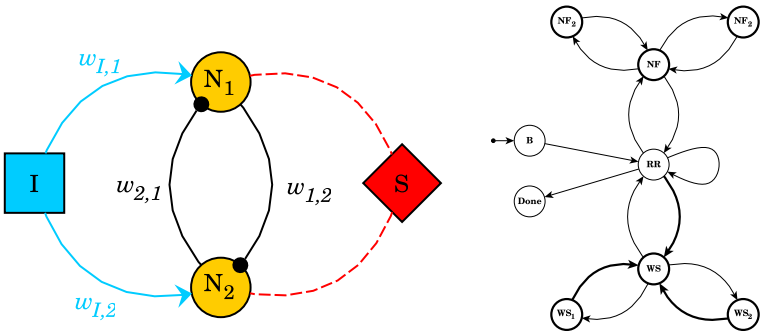
Figure 5.6: Representation of the evolution of the weights over time, for a simulation run. The parameters of all neurons $\mathcal{N}_i$ are: $T_i = 1$, $\tau_i = 1$, $\theta_i = R$ and $\lambda_i = 1/2$, where $R = 1000$ is the discretization granularity. The chosen learning rates are $\Delta W = 2$ and $\delta w = 1$. At the very beginning of the simulation, all weights are 0. The weight $w_{I,1}$, of the synapse feeding the first neuron, is the one rising the most and more quickly. It stabilizes at value 1 after 300 time units. The weights $w_{1,2}$ and $w_{1,3}$, of the synapses connecting the first to the second and the third neuron, respectively, rise almost at the same rate. They reach the values 0.63 and 0.633, respectively, after 550 time units. The weights $w_{2,4}$ and $w_{3,4}$, of the synapses connecting the second and the third to the fourth neuron, respectively, rise almost at the same rate, too. They both reach the value 0.551, after 550 time units. At time unit 550 the supervisor $\mathcal{S}$ reaches its acceptation state (continuous red line), terminating the learning process.

The simulation continues until $w_{2,4}$ and $w_{3,4}$ are great enough to enable $\mathcal{N}_4$ for firing.

Example 5.3 (Creating an oscillating negative loop). In this example we show how a network having the *loop* structure in figure 5.7a can learn to produce an oscillatory behavior by means of the simulation-oriented ABP approach. The loop topology consists of two output neurons, $\mathcal{N}_1$ and $\mathcal{N}_2$, mutually feeding each other. We assume the two neurons are fed by a fixed-rate input generator $\mathcal{I}$ but no one is able to emit because the weights of their input synapses are 0 and their thresholds are higher than 0.

The *oscillatory behavior* consists of the two neurons *eventually* beginning to *alternatively* fire a spike, in a periodic fashion. We want the network to learn the weights assignment making its output neurons $\mathcal{N}_1$ and $\mathcal{N}_2$ reproduce such a behavior.

The learning process is led by the “round robin” supervisor $\mathcal{S}$, whose struc-



(a) The *loop* network topology

(b) The structure of the *round robin* supervisor automaton $\mathcal{S}$

Figure 5.7: The loop of two neurons shown in fig. 5.7a is trained to produce an oscillation by the *round robin* supervisor, whose structure is shown in fig. 5.7b. The training process exploits the simulation-oriented ABP.

ture is shown in figure 5.7b. A complete definition of the behavior of such a supervisor is out of the scope of this example. Here we just provide an intuition of its functioning[2]. The "round robin" supervisor $\mathcal{S}$ aims to train a couple of neurons to alternatively fire a spike with a period of P time units, i.e., the two neurons should fire with period $2P$ and their phase difference should be P. It allows for an initial delay of D time units, during which no advice is back-propagated. So it begins its execution in location $\mathbf{B}$ and, after D time units, it moves to the $\mathbf{RR}$ location (for "Round-Robin"). Here it starts a periodic cycle where the duration of each iteration is P time units. It employs a variable $turn$ to keep track of which neuron should fire at the end of each period. We indicate such a neuron by $\mathcal{N}_{turn}$ and the other one by $\mathcal{N}_{other}$. For each period, $\mathcal{N}_{turn}$ is expected to fire a spike at the exact end of the period. If it does not, it receives a SHF advice, while, if it fires *before* P time units, it receives a SNHF advice. For the same period, $\mathcal{N}_{other}$ is expected to remain quiescent for the whole duration of the period. If it is not the case, it receives a SNHF advice. At the end of each period, the two neurons switch their roles. Moreover, if no error has been detected during the last period, a variable $success$ is increased, otherwise it is reset. The supervisor reaches its acceptation location, $\mathbf{Done}$, as soon as $success$ reaches the value L (for "Limit"), because the network is behaving correctly since at least $L \cdot P$ time units.

[2] a complete implementation of the "round robin" supervisor can be found in [1]

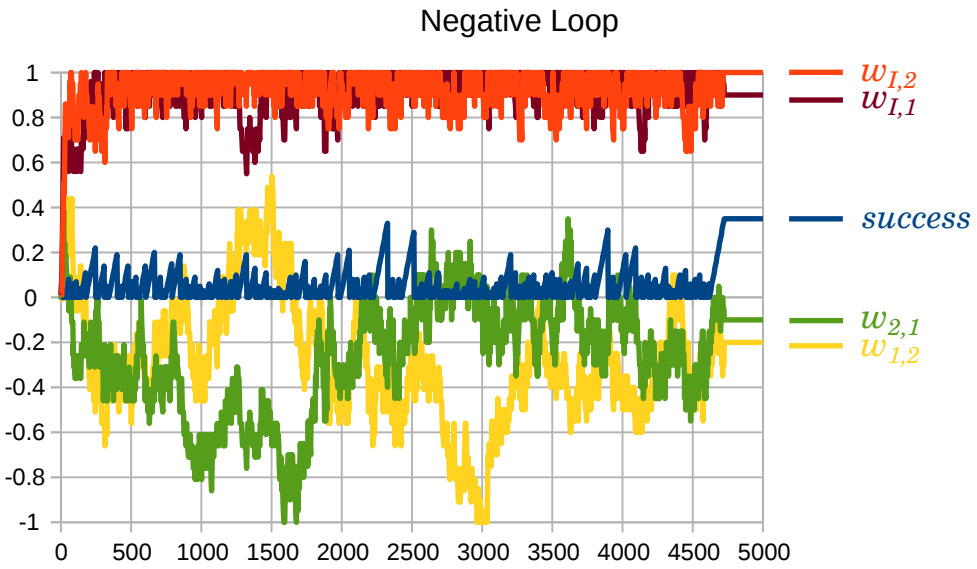


Figure 5.8: Representation of the evolution of the weights over time, for a simulation run. The parameters of all neurons $\mathcal{N}_i$ are: $T_i = 1$, $\tau_i = 1$, $\theta_i = R$ and $\lambda_i = 1/2$, where $R = 100$ is the discretization granularity. The chosen learning rates are $\Delta W = 10$ and $\delta w = 5$. The initial delay of the round-robin supervisor $\mathcal{S}$ is $D = 1$ time unit, its period duration is $P = 3$ time units, and its number of consecutive successful periods to terminate the learning process is $L = 35$. The simulation-oriented ABP required 4725 simulated time units to reach a weight assignment.

Figure 5.8 shows a simulation run. At the very beginning of the simulation, all weights are 0. The weights $w_{I,1}$ and $w_{I,2}$, of the synapses connecting the input generator to the two neurons, rapidly rise until eventually beginning an oscillation between the values $0.8 \cdot R$ and R. Conversely, the weights $w_{1,2}$ and $w_{2,1}$, of the synapses connecting the two neurons to each other, keep oscillating with a negative trend, i.e., they hold a negative value, the most of the time. The value of the $success$ variable of $\mathcal{S}$ is represented by a blue line. Such a value can be considered as a measure of the performance of the network, with respect to the desired behavior. Indeed, it controls the termination of the learning process. Such a process ends after 4725 simulated time units, when the value of the $success$ variable reaches 35 and the weights configuration is $w_{I,1} = 0.9$, $w_{I,2} = 1$, $w_{1,2} = -0.2$ and $w_{2,1} = -0.1$. Notice that in the previous steps of the simulation, where the value of $success$ variable is relatively high (e.g., time units 2325 or 2514 or 3895), the weights configuration is similar to the final one: $w_{I,1}$ and $w_{I,2}$ are positive and their absolute value is close to 1,

while $w_{1,2}$ and $w_{2,1}$ are negative and their absolute value is close to 0.

5.3 Model-checking-oriented advice back-propagation

Here we propose a model-checking-oriented implementation of the ABP approach. Such a technique consists in iterating the learning process until a CTL formula expressing the desired output of the network is verified, as shown in algorithm 5.3. We introduce the following hypotheses: (i) the network is composed by a number of input generators, neurons encoded as timed automata and output consumers sharing a common global clock which is never reset: such a clock allows to express absolute time constraints within the CTL formula defining the desired behavior; (ii) for each output consumer, there exists a clock measuring the elapsed time since the last time the neuron it is linked to received spike. The CTL formula specifying the expected outcome of the network can only contain predicates relative to the output consumers and the global clock. Each step of the algorithm consists of an invocation to the model-checker aiming to test whether the network satisfies the formula or not. If the formula is satisfied then the learning process is over. If it is not satisfied, a trace is provided as counterexample. Such a trace can be exploited to deduce the proper *corrective action* to be performed for each output neuron, i.e., whether the SHOULD-HAVE-FIRED() or SHOULD-NOT-HAVE-FIRED() procedures (or none of them) should be invoked.

Algorithm 5.3 Model-checking-oriented ABP: pseudo-code

```
1: procedure MC-ADVICE-BACKPROP(network, φ)
2:     trace ← MODEL-CHECK(network, φ)
3:     while trace ≠ ∅ do
4:         for all neuron ← GET-FAILING-NEURONS(trace, φ) do
5:             advice ← SELECT-ADVICE(neuron, trace, φ)
6:             network ← APPLY-ADVICE(neuron, advice)      ▷ SHF or SNHF
7:         trace ← MODEL-CHECK(network, φ)
```

More in detail, given a timed automata network $\mathcal{S}$ encoding some spiking neural network, we extend it by means of a global clock t_g which is never reset

by any automaton composing $\mathcal{S}$ and, for each output consumer $\mathcal{O}_k$ relative to the output neuron $\mathcal{N}_k$, we add a clock s measuring the time elapsed since the last spike consumed by $\mathcal{O}_k$.

In what follows we indicate by $state_{\mathcal{O}_k}(\mathbf{O})$ the atomic proposition evaluating to $\mathbf{true}$ if the output consumer $\mathcal{O}_k$, linked to the k-th output neuron $\mathcal{N}_k$, is in its location $\mathbf{O}$, meaning that $\mathcal{N}_k$ has just fired a spike. We also indicate by $eval_{\mathcal{O}_k}(s)$ the function returning the current value of the clock s for the automaton $\mathcal{O}_k$.

In order to make it possible to deduce the proper corrective action for a given output neuron, we impose the following constraint: the CTL formula describing the expected outcome of the network must be composed by the conjunction of sub-formulae respecting any of the following patterns:

Precise Firing. The output neuron $\mathcal{N}_k$ fires *exactly* at time t:

$$AF\,(\,t_g = t \wedge state_{\mathcal{O}_k}(\mathbf{O})\,)$$

So, if $\mathcal{N}_k$ does not fire a spike while the value of the global clock is t, the output formula is not satisfied. In this case we deduce that $\mathcal{N}_k$ should have fired, but it did not.

Weak Quiescience. The output neuron $\mathcal{N}_k$ is quiescent *exactly* at time t:

$$AG\,(\,t_g = t \implies \neg state_{\mathcal{O}_k}(\mathbf{O})\,)$$

So, if $\mathcal{N}_k$ fires a spike while the value of the global clock is t, the output formula is not satisfied. In this case we deduce that $\mathcal{N}_k$ should not have fired, but it did.

Relaxed Firing. The output neuron $\mathcal{N}_k$ fires at least once within the time window $[\,t_1,\,t_2\,]$:

$$AF\,(\,t_1 \leq t_g \leq t_2 \wedge state_{\mathcal{O}_k}(\mathbf{O})\,)$$

So, if $\mathcal{N}_k$ does not fire a spike while the value of the global clock is within the time interval $[\,t_1,\,t_2\,]$, the output formula is not satisfied. In this case we deduce that $\mathcal{N}_k$ should have fired, but it did not.

Strong Quiescience. The output neuron $\mathcal{N}_k$ remains quiescent for the whole duration of the time interval $[\,t_1,\,t_2\,]$:

$$AG\,(\,t_1 \leq t_g \leq t_2 \implies \neg state_{\mathcal{O}_k}(\mathbf{O})\,)$$

So, if $\mathcal{N}_k$ fires a spike while the value of the global clock is within the time interval $[\,t_1,\,t_2\,]$, the output formula is not satisfied. In this case we deduce that $\mathcal{N}_k$ should not have fired, but it did.

Precise Periodicity. The output neuron $\mathcal{N}_k$ will *eventually* begin to *periodically* fire a spike with exact period P:

$$\begin{aligned} AF(\,AG(\,eval_{\mathcal{O}_k}(s) \neq P \implies \neg state_{\mathcal{O}_k}(\mathbf{O})\,) \\ \wedge \\ AG(\,state_{\mathcal{O}_k}(\mathbf{O}) \implies eval_{\mathcal{O}_k}(s) = P\,)\,) \end{aligned} \quad (5.2)$$

This formula expresses a periodic behavior because the s clock is reset as soon as $\mathcal{N}_k$ fires a spike. The outer AF quantifier pair allows the periodic behavior to be arbitrary delayed. So, if $\mathcal{N}_k$ fires a spike while the s clock is different than P or it does not fire a spike while the value of the s clock is equal to P, the output formula is not satisfied. In the former case we deduce that $\mathcal{N}_k$ should not have fired, but it did. In the latter case we deduce that $\mathcal{N}_k$ should have fired, but it did not.

Relaxed Periodicity. The output neuron $\mathcal{N}_k$ will eventually begin to periodically fire a spike with a period that may vary in $[\,P_{min},\,P_{max}\,]$:

$$\begin{aligned} AF(\,AG(\,eval_{\mathcal{O}_k}(s) \notin [\,P_{min},\,P_{max}\,] \implies \neg state_{\mathcal{O}_k}(\mathbf{O})\,) \\ \wedge \\ AF(\,state_{\mathcal{O}_k}(\mathbf{O}) \implies P_{min} \leq eval_{\mathcal{O}_k}(s) \leq P_{max}\,)\,) \end{aligned} \quad (5.3)$$

This formula is analogous to the previous one, except for the interval in place of the exact values. The errors are detected as for the *precise periodicity* case.

As for future work, we intend to extend this set of patterns with new CTL formulae concerning the comparison of the output of two or more given neurons. It is important for such formlae to make it easy to understand *which*

neurons failed (i.e., did not behave as expected) and *which corrective actions* must be applied to those neurons.

Chapter 6

Conclusions and future works

In this thesis we formalized the leaky-integrate-and-fire neuron model of spiking neural networks via timed automata. We discussed properties, capabilities and limits of our formalization with respect to a number of biophisically meaningful behaviors described into the literature. Finally, we defined the learning problem and provided a novel approach to the weights assignment problem, a particular case of the former one.

Leaky-integrate-and-fire neurons are formalized as automata waiting for inputs on a number of different channels, for a fixed amount of time. When such *accumulation period* is over, the current *potential* value is computed by means of a recursive formula taking into account the current sum of weighted inputs, and the previous decayed potential value. If the current potential overcomes a given *threshold*, the automaton emits a broadcast signal over its output channel, otherwise it restarts its accumulation period. So, spikes are modeled as instantaneous communications over broadcast channels. After each emission, the automaton is constrained to remain inactive for a fixed *refractory period*, after which the potential is reset.

Spiking neural networks composed by more than one neuron can be formalized by a set of automata, one for each neuron, running in parallel and sharing channel accordingly. Part of these neurons are considered *output* neurons, i.e., the outcome of the network consists of their output spikes.

The inputs needed to feed network are defined through timed automata as well. We have provided a language and its encoding into timed automata

to model patterns of spikes and pauses and a way of modeling unpredictable sequences.

We validated our neuron model proving some characteristic properties expressed in CTL via model-checking: it is able to exhibit the *tonic spiking* behavior, i.e., it periodically emits a spike if stimulated by a persistent excitatory input; it is able to act as an *integrator* under some proper parameter settings, i.e., it can detect — meaning that it fires as a consequence of — a defined amount of simultaneous or consecutive input spikes; it is *excitable*: its output frequency increases (i.e., its inter-emission period decreases) if its total stimulus magnitude keeps increasing; there exists a way to compute the *maximum threshold*, i.e., the threshold value such that any greater or equal value would prevent the neuron from firing.

In the last part of this thesis, we face the *learning problem*, consisting of searching an assignment to *all* the parameters of some spiking neural network, in order to make it capable of reproducing a desired behavior. We chose to focus on the *weights assignment* sub-problem, for simplicity. So we assumed all parameters but the synaptic weights to be fixed and given.

Then we proposed the *advice back-propagation* (ABP) approach, which provides a solution to the weights assignment problem. It aims to detect which output neurons are not behaving as expected and to determine the right corrective action, namely the *advice*, to be performed for each one. There exists two possible advices that a neuron may “receive”: *Should Have Fired* (SHF) and *Should Not Have Fired* (SNHF), each one indicating an expected output which was not accomplished. The advices can be back-propagated to the other neurons composing the network, possibly producing a variation of the synaptic weights that should reduce the amount of errors performed by the output neurons.

We provided two possible implementations of the ABP approach: the simulation-oriented and the model-checking-oriented ones. In the former one, the neuron formalization described above is extended in order to make each neuron able to receive and forward SHF and SNHF signals. In this case, the learning process is led by some *supervisors* automata, and the network must be *simulated* until eventually reaching a solution for the weight assignment problem. In the latter implementation, no extension is required to the neuron

formalization, but some constraints are imposed to the CTL formula defining the expected behavior. Such a formula must be composed by the conjunction of some predicate *patterns*, each one allowing to define the specific behavior of any output neuron. In this case, the learning process consists of repeatedly querying the expected output formula to a CTL model-checker. If the formula is satisfied, the learning process is over, otherwise, the trace provided by the model-checker must be exploited to apply the right corrective actions to the neurons composing the network.

Technical details. In order to validate and test our timed automata networks, we pervasively employed the `Uppaal` framework. We engineered the generation of `Uppaal` systems from *network specifications*, i.e., code snippets fitting our *network description language* [1]. Such a language allows to describe the topology of a spiking neural network, the parameters of the neurons composing it, and the spike–pause sequences used to feed it. The generation process can convert a network specification into an `Uppaal` project where timed automata are realized by means of the encodings presented in this thesis. The *code generator*, needed to convert a network specification into a ready-to-use `Uppaal` system, has been implemented by means of the `Xtext` framework and the `Xtend` language [6]. Thanks to such tools, we produced both a standalone executable and a plug-in for the Eclipse IDE.

Future research directions. We consider this work as the starting point for a number of research directions.

We plan to analyze intrinsic properties, capabilities and limits of the *asynchronous* encoding, too, as we did for the *synchronous* one in chapter 4. Furthermore, it may be interesting to produce analogous formalizations for more complex spiking neuron models like, e.g., the *theta-neuron* model [14] or Izhikevich's one [20]. In a wider perspective we would also like to formalize synapses by means of timed automata. This would allow to model, e.g., propagation *delays*, which are considered an interesting feature within the scope of spiking neural networks [27].

For what concerns our novel ABP approach, we can imagine several possible improvements. For instance, we intend to define a better formalization of

the *supervisor* automaton concept, for the simulation-oriented implementation. We would also like to find more patterns describing the expected behavior of output neurons, for the model-checking-oriented approach. In both cases, we search for methods allowing to compare the output of several neurons. Finally, we plan to extend our technique in order to be able to infer not only synaptic weights but also other parameters, such as the leak factors or the firing thresholds.

Bibliography

[1] Additional material. `https://github.com/gciatto/snn_as_ta`.

[2] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. Connectionist models and their implications: Readings from cognitive science. chapter A Learning Algorithm for Boltzmann Machines, pages 285–307. Ablex Publishing Corp., Norwood, NJ, USA, 1988.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal 4.0*. Department of Computer Science, Aalborg University, Denmark, 11 2006.

[5] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[6] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013.

[7] Guo-qiang Bi and Mu-ming Poo. Synaptic modification by correlated activity: Hebb's postulate revisited. *Annual review of neuroscience*, 24(1):139–166, 2001.

[8] Sander M. Bohte, Han A. La Poutré, Joost N. Kok, Han A. La, Poutre Joost, and N. Kok. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48:17–37, 2002.

[9] Giovanni Ciatto, Elisabetta De Maria, and Cinzia Di Giusto. Spiking neural networks as timed automata. *Avancées en Biologie des Systèmes et de Synthèse* (ASSB), March 2016.

[10] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[11] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[12] Yang Dan and Mu ming Poo. Spike timing-dependent plasticity of neural circuits. *Neuron*, 44(1):23 – 30, 2004.

[13] Elisabetta De Maria, Alexandre Muzy, Daniel Gaffé, Annie Ressouche, and Franck Grammont. Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models. In *Fifth International Workshop on Hybrid Systems Biology*, Grenoble, France, October 2016.

[14] G. B. Ermentrout and N. Kopell. Parabolic bursting in an excitable system coupled with a slow oscillation. *SIAM Journal on Applied Mathematics*, 46(2):233–253, 1986.

[15] Yoav Freund and Robert E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, 1999.

[16] André Grüning and S.M. Bohte. Spiking neural networks: Principles and challenges, 2014.

[17] Donald O. Hebb. *The Organization of Behavior*. John Wiley, 1949.

[18] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.

[19] J. J. Hopfield. Neurocomputing: Foundations of research. chapter Neural Networks and Physical Systems with Emergent Collective Computational Abilities, pages 457–464. MIT Press, Cambridge, MA, USA, 1988.

[20] E. M. Izhikevich. Simple model of spiking neurons. *Trans. Neur. Netw.*, 14(6):1569–1572, November 2003.

[21] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, Sept 2004.

[22] L. Lapicque. Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. *J Physiol Pathol Gen*, 9:620–635, 1907.

[23] Wolfgang Maass. On the Computational Complexity of Networks of Spiking Neurons. Technical report, Institute for Theoretical Computer Science, Technische Universitaet Graz, 05 1994.

[24] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

[25] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[26] Michael C. Mozer. Backpropagation. chapter A Focused Backpropagation Algorithm for Temporal Pattern Recognition, pages 137–169. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[27] Hélène Paugam-Moisy and Sander Bohte. *Computing with Spiking Neuron Networks*, pages 335–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[28] Michael Recce. Pulsed neural networks. chapter Encoding Information in Neuronal Activity, pages 111–131. MIT Press, Cambridge, MA, USA, 1999.

[29] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[30] J. Sjöström and W. Gerstner. Spike-timing dependent plasticity. *Scholarpedia*, 5(2), 2010. Revision 151671.