Inferring the Synaptical Weights of Leaky Integrate and Fire Asynchronous Neural Networks: Modelled as Timed Automata

Elisabetta De Maria and Cinzia Di Giusto(✉)

Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France
cinzia.di-giusto@unice.fr

Abstract. In this work we introduce a new approach to learn the synaptical weights of neural biological networks. At this aim, we consider networks of Leaky Integrate and Fire neurons and model them as timed automata networks. Each neuron receives input spikes through its incoming synapses (modelled as channels) and computes its membrane *potential* value according to the (present and past) received inputs. Whenever the potential value overtakes a given *firing threshold*, the neuron emits a spike (modelled as a broadcast signal over the output channel of the corresponding automaton). After each spike emission, the neuron enters first an *absolute refractory period*, in which signal emission is not allowed, and then a *relative refractory period*, in which the firing threshold is higher than usual. Neural networks are modelled as sets of timed automata running in parallel and sharing channels in compliance with the network structure. Such a formal encoding allows us to propose an algorithm which automatically infers the synaptical weights of neural networks such that a given dynamical behaviour can be displayed. Behaviours are encoded as temporal logic formulae and the algorithm modifies the network weights until un assignment satisfying the specification is found.

Keywords: Neural networks · Parameter learning · Timed automata · Temporal logic · Model checking

1 Introduction

In the last decades, the study of biological neurons and their interactions has become very intensive, especially in the perspective of identifying the circuits involved in the main vital functions, such as breathing or walking, and detecting how they are modified in case of disease. The majority of the approaches aiming at exploring the brain functioning mainly relies on large-scale simulations [9]. In this paper we propose a formal approach based on the use of timed automata [2]. This formalism extends finite state automata with timed behaviours: constraints are allowed to limit the amount of time an automaton can remain within a particular state, or the time interval during which a particular transition may be

A. Cliquet jr. et al. (Eds.): BIOSTEC 2018, CCIS 1024, pp. 149–166, 2019.
https://doi.org/10.1007/978-3-030-29196-9_9

enabled. It is possible to build timed automata networks, where several automata can synchronise over *channel* communications.

As far as the modelling of neuronal networks is concerned, three different and progressive *generations* of networks can be found in the literature [24,26]. *First generation* models handle discrete inputs and outputs and their computational units are threshold-based transfer functions; they include McCulloch and Pitt's threshold gate model [25], the perceptron model [15], Hopfield networks [20], and Boltzmann machines [1]. *Second generation* models exploit real valued activation functions, e.g., the sigmoid function, accepting and producing real values: a well known example is the multi-layer perceptron [8,29]. *Third generation* networks 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 spike magnitudes and occurrence times [26].

In this work we focus on *spiking neural networks* [16]. Because of the introduction of timing aspects they are considered closer to the actual brain functioning than other generations models. Spiking neurons emit 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 behaviour, but also because the temporal dimension allows to represent information according to various *coding schemes* [26,27]: 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* coding).

Several spiking neuron models, with different capabilities and complexities, have been proposed in the literature. In [22], Izhikevich classifies spiking neuron models according to some *behaviour* (i.e., typical responses to an input pattern) that they should exhibit in order to be considered biologically relevant. The leaky integrate & fire (LI&F) model [23], where past inputs relevance exponentially decays with time, is one of the most employed neuron models because it is straightforward and easy to use [22,26]. By contrast, the Hodgkin-Huxley (H-H) model [19] is one of the most complex being composed by four differential equations comparing neurons to electrical circuits. In [22], the H-H model can reproduce all behaviours at issue, but the simulation process is really expensive even for just a few neurons being simulated for a small amount of time. Our aim is to produce a neuron model being meaningful from a biological point of view but also suitable to formal analysis and verification, that could be therefore exploited 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, to assess if a change to the network structure can alter its behaviour, or to investigate (new) learning algorithms which take time into account.

The core of our studies is the *leaky integrate & fire* (LI&F) model originally proposed in [23]. It is a computationally efficient approximation of single-compartment model [22] and is abstracted enough to be able to apply formal

verification techniques such as model-checking. Here we work on an extended version of the discretised formulation proposed in [13], which relies on the notion of logical time. Time is considered as a sequence of logical discrete instants, and an instant is a point in time where external input events can be observed, computations can be done, and outputs can be emitted. The variant we introduce here takes into account some new time-related aspects, such as the *refractory period*, a lapse of time in which the neuron cannot emit (or can only emit under some restrictive conditions).

Our modelling of spiking neural networks consists of timed automata networks where each neuron is an automaton. Each neuron receives input spikes through its incoming synapses (modelled as *channels*) and computes its membrane *potential* value according to the (present and past) received inputs. Whenever the potential value overtakes a given *firing threshold*, the neuron emits a spike (modelled as a *broadcast signal* over an output channel of the corresponding automaton).

As a central contribution, we exploit our automata-based modelling to propose a new methodology for parameter inference in spiking neural networks. In particular, our approach allows to find an assignment for the synaptical weights of a given neural network such that it can reproduce a given (expected) behaviour.

This paper is an improved and revised version of the conference paper [10]. In particular, the neuron model we introduce here is substantially different. In [10], neurons need to wait for the end of some specific accumulation periods (during which signals are received) before emitting spikes. Here, accumulation periods are removed and neurons can receive and emit signals in an *asynchronous* way. Furthermore, a unique refractory period is replaced by an *absolute refractory period*, in which signal emission is not allowed, and a *relative refractory period*, in which the firing threshold is higher than usual. This entails a new definition of Leaky Integrate and Fire neuron and a new encoding into timed automata. The examples are adapted to fit to the new model and new consistent parameters are computed. Finally, the algorithm for parameter inference is refined in order to avoid deadlock scenarios and a new simulation-oriented approach to implement this algorithm is briefly introduced (see [11] for a detailed description of this technique).

The rest of the paper is organised as follows: in Sect. 2 we recall definitions of timed automata networks, temporal logics, and model checking; in Sect. 3 we describe our reference model, the LI&F one, and its encoding into timed automata networks; in Sect. 4 we develop the parameter learning approach and we introduce a case study; in Sect. 5 we give an overview of the related work. Finally, Sect. 6 summarises our contribution and presents some future research directions.

2 Preliminaries

In this section we introduce the formalisms we adopt in the rest of the paper, that is, timed automata and temporal logics.

2.1 Timed Automata

Timed automata [2] are a powerful theoretical formalism for modelling and verifying real time systems. A timed automaton is an annotated directed (and connected) graph, with an initial node and provided with a finite set of non-negative real variables called *clocks*. Nodes (called *locations*) are annotated with *invariants* (predicates allowing to enter or stay in a location), arcs with *guards*, *communication labels*, and possibly with some variables upgrades and clock *resets*. Guards are conjunctions of elementary predicates of the form $x \ op \ c$, where $op \in \{>, \geq, =, <, \leq\}$, x is a clock, and c a (possibly parameterised) positive integer constant. As usual, the empty conjunction is interpreted as true. The set of all guards and invariant predicates will be denoted by G.

Definition 1. *A* timed automaton *TA is a tuple $(L, l^0, X, \Sigma, Arcs, Inv)$, where*

- *L is a set of locations with $l^0 \in L$ the initial one*
- *X is the set of clocks,*
- *Σ is a set of communication labels,*
- *$Arcs \subseteq L \times (G \cup \Sigma \cup U) \times L$ is a set of arcs between locations with a guard in G, a communication label in $\Sigma \cup \{\varepsilon\}$, and a set of variable upgrades (e.g., clock resets);*
- *$Inv : L \to G$ assigns invariants to locations.*

It is possible to define a synchronised product of a set of timed automata that work and synchronise in parallel. The automata are required to have disjoint sets of locations, but may share clocks and communication labels which are used for synchronisation. We restrict communications to be *broadcast* through labels $b!, b? \in \Sigma$, meaning that a set of automata can synchronise if one is emitting; notice that a process can always emit (e.g., $b!$) and the receivers ($b?$) must synchronise if they can.

Locations can be normal, urgent or committed. Urgent locations force the time to freeze, committed ones freeze time and the automaton must leave the location as soon as possible, i.e., they have higher priority.

The synchronous product $TA_1 \parallel \ldots \parallel TA_n$ of timed automata, where $TA_j = (L_j, l_j^0, X_j, \Sigma_j, Arcs_j, Inv_j)$ and L_j are pairwise disjoint sets of locations for each $j \in [1, \ldots, n]$, is the timed automaton

$$TA = (L, l^0, X, \Sigma, Arcs, Inv)$$

such that:

- $L = L_1 \times \ldots \times L_n$ and $l^0 = (l_1^0, \ldots, l_n^0)$, $X = \bigcup_{j=1}^n X_j$, $\Sigma = \bigcup_{j=1}^n \Sigma_j$,
- $\forall l = (l_1, \ldots, l_n) \in L$: $Inv(l) = \bigwedge_j Inv_j(l_j)$,
- $Arcs$ is the set of arcs $(l_1, \ldots, l_n) \xrightarrow{g,a,r} (l'_1, \ldots, l'_n)$ such that for all $1 \leq j \leq n$ then $l'_j = l_j$.

Its semantics is the one of the underlying timed automaton TA with the following notations. A location is a vector $l = (l_1, \ldots, l_n)$. We write $l[l'_j/l_j, j \in S]$

to denote the location l in which the j^{th} element l_j is replaced by l'_j, for all j in some set S. A valuation is a function ν from the set of clocks to the non-negative reals. Let $\mathbb{V}$ be the set of all clock valuations, and $\nu_0(x) = 0$ for all $x \in X$. We shall denote by $\nu \vDash F$ the fact that the valuation ν satisfies (makes true) the formula F. If r is a clock reset, we shall denote by $\nu[r]$ the valuation obtained after applying the clock reset $r \subseteq X$ to ν; and if $d \in \mathbb{R}_{>0}$ is a delay, $\nu + d$ is the valuation such that, for any clock $x \in X$, $(\nu + d)(x) = \nu(x) + d$.

The semantics of a synchronous product $TA_1 \parallel \ldots \parallel TA_n$ is defined as a timed transition system $(S, s_0, \rightarrow)$, where $S = (L_1 \times, \ldots \times L_n) \times \mathbb{V}$ is the set of states, $s_0 = (l^0, \nu_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

- (silent): $(l, \nu) \rightarrow (l', \nu')$ if there exists $l_i \xrightarrow{g,\varepsilon,r} l'_i$, for some i, such that $l' = l[l'_i/l_i]$, $\nu \vDash g$ and $\nu' = \nu[r]$,
- (broadcast): $(\bar{l}, \nu) \rightarrow (\bar{l'}, \nu')$ if there exists an output arc $l_j \xrightarrow{g_j,b!,r_j} l'_j \in Arcs_j$ and a (possibly empty) set of input arcs of the form $l_k \xrightarrow{g_k,b?,r_k} l'_k \in Arcs_k$ such that for all $k \in K = \{k_1, \ldots, k_m\} \subseteq \{l_1, \ldots, l_n\} \setminus \{l_j\}$, the size of K is maximal, $\nu \vDash \bigwedge_{k \in K \cup \{j\}} g_k$, $l' = l[l'_k/l_k, k \in K \cup \{j\}]$ and $\nu' = \nu[r_k, k \in K \cup \{j\}]$;
- (timed): $(l, \nu) \rightarrow (l, \nu + d)$ if $\nu + d \vDash Inv(l)$.

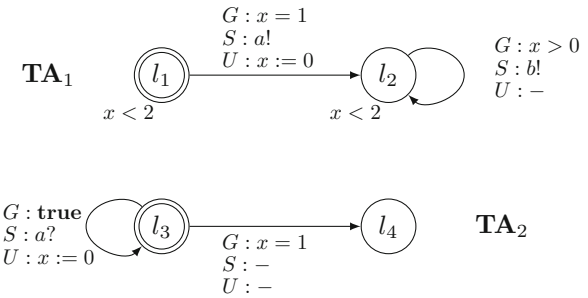
The valuation function ν is extended to handle a set of shared bounded integer variables: predicates concerning such variables can be part of edges guards or locations invariants, moreover variables can be updated on edges firings but they cannot be assigned to or from clocks.

Example 1. In Fig. 1 we consider the network of timed automata TA_1 and TA_2 with broadcast communications, and we give a possible run. TA_1 and TA_2 start in the l_1 and l_3 locations, respectively, so the initial state is $[(l_1, l_3);\ x = 0]$. A *timed* transition produces a delay of 1 time unit, making the system move to state $[(l_1, l_3);\ x = 1]$. A *broadcast* transition is now enabled, making the system move to state $[(l_2, l_3);\ x = 0]$, broadcasting over channel a and resetting the x clock. Two successive *timed* transitions (0.5 time units) followed by a *broadcast* one will eventually lead the system to state $[(l_2, l_4);\ x = 1]$. ◇

To model neural networks, we have used the specification and analysis tool `Uppaal` [4], which allows to design and simulate timed automata networks and to validate networks against temporal logic formulae. All figures depicting timed automata follow the graphic conventions of the tool (e.g., initial states are denoted with a double circle).

2.2 Temporal Logics and Model Checking

Model checking is one of the most common approaches to the verification of software and hardware (distributed) systems [7]. It allows to automatically prove whether a system verifies or not a given specification. In order to apply such a



(a) The timed automata network $TA_1 \parallel TA_2$.

$[(l_1, l_3);\ x = 0]$
↓
$[(l_1, l_3);\ x = 1]$
↓
$[(l_2, l_3);\ x = 0]$
↓
$[(l_2, l_3);\ x = 0.5]$
↓
$[(l_2, l_3);\ x = 1]$
↓
$[(l_2, l_4);\ x = 1]$

(b) A possible run.

Fig. 1. A network of timed automata with a possible run.

technique, the system at issue should be encoded as a finite transition system and the specification should be written using propositional temporal logic. Formally, a transition system over a set AP of atomic propositions is a tuple $M = (Q, T, L)$, where Q is a finite set of states, $T \subseteq Q \times Q$ is a total transition relation, and $L : Q \to 2^{AP}$ is a labelling function that maps every state into the set of atomic propositions that hold at that state.

Temporal formulae describe the dynamical evolution of a given system. The computation tree logic CTL* allows to describe properties of computation trees. Its formulas are obtained by (repeatedly) applying boolean connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$), *path quantifiers*, and *state quantifiers* to atomic formulas. The path quantifier **A** (resp., **E**) can be used to state that all the paths (resp., some path) starting from a given state have some property. The state quantifiers are **X** (next time), which specifies that a property holds at the next state of a path, **F** (sometimes in the future), which requires a property to hold at some state on the path, **G** (always in the future), which imposes that a property is true at every state on the path, and **U** (until), which holds if there is a state on the path where the second of its argument properties holds and, at every preceding state on the path, the first of its two argument properties holds. Given two formulas φ_1 and φ_2, in the rest of the paper we use the shortcut $\varphi_1 \rightsquigarrow \varphi_2$ to denote the liveness property $AG(\varphi_1 \rightarrow AF\varphi_2)$, which can be read as "$\varphi_1$ always leads to φ_2 ".

The branching time logic CTL is a fragment of CTL* that allows quantification over the paths starting from a given state. Unlike CTL*, it constrains every state quantifier to be immediately preceded by a path quantifier.

Given a transition system $M = (Q, T, L)$, a state $q \in Q$, and a temporal logic formula φ expressing some desirable property of the system, the *model checking problem* consists of establishing whether φ holds at q or not, namely, whether $M, q \models \varphi$.

3 Leaky Integrate and Fire Model and Mapping to Timed Automata

Spiking neural networks [24] have been traditionally modelled as directed graphs, where vertices represent *neurons* and oriented edges are the *synapses*. Each neuron is a computational unit whose evolution depends on time passing by and on the reception of signals through its ingoing synapses. The weight associated to each synapse defines the nature of the signal: excitatory if positive, inhibitory if negative. Input signals are, then, summed up by the neuron in a variable called the *potential*. The potential accumulated decreases as time passes by and its loss is regulated by the *leak factor*. As soon as the potential exceeds the *firing threshold*, the neuron emits a signal called *spike* over all its outgoing synapses. When the neuron fires, its potential is reset to zero and the neuron enters a special state: the *refractory period*. This period is divided into two parts, the absolute and the relative refractory period. During the former, the neuron is completely inhibited: it ignores each incoming spike and it cannot fire. During the latter, the neuron can receive spikes but its firing threshold is much higher than usual. This threshold decreases with time passing by, by a *threshold factor* η, until it reaches the normal value. This entails that, during the relative refractory period, if the neuron is stimulated enough it will fire, thus resetting the potential to zero and restarting a new refractory period (absolute and relative).

In this paper we consider discrete time. Next we give a formal definition of Spiking Integrate and Fire Neural Networks and their dynamics.

Definition 2 (Spiking Integrate and Fire Neural Network). A spiking integrate and fire neural network *is a tuple* (V, A, w)*, where:*

- V *are spiking integrate and fire neurons,*
- $A \subseteq V \times V$ *are synapses,*
- $w : A \to \mathbb{Q} \cap [-1, 1]$ *is the synapse weight function associating to each synapse* (u, v) *a weight* $w_{u,v}$*.*

Each spiking integrate and fire neuron v *is characterized by a parameter tuple* $(\theta_v, \theta'_v, \tau_v, \eta_v, \lambda_v)$*, where:*

- $\theta_v, \theta'_v \in \mathbb{N}$ *are the* firing threshold *for the normal and relative refractory period respectively,*
- $\tau_v \in \mathbb{N}^+$ *is the duration of the* absolute refractory period*,*
- $\eta_v \in \mathbb{Q} \cap [0, 1]$ *is the* threshold factor
- $\lambda_v \in \mathbb{Q} \cap [0, 1]$ *is the* leak factor*.*

The state of each neuron v is described by the tuple (s_v, p_v, f_v, t_v), where

- t_v is a timer;
- $s_v \in \{n, a, r\}$ is the state of v, n for a neuron in a normal state, a for a neuron in the absolute refractory period, and r for the relative one;
- p_v is the potential of v;
- f_v is a boolean value stating whether the neuron has fired or not.

A configuration C is the set of states of all neurons $v \in V$. The semantics of the neural network is given by the set of reachable configurations from an initial one. The initial configuration sets the state of all neurons $v \in V$ to $(n, 0, 0, 0)$. Let $C = \{(s_v, p_v, f_v, t_v) \mid v \in V\}$, then $C \to C'$ if and only if $C' = \{(s'_v, p'_v, f'_v, t'_v) \mid v \in V\}$ and $(s_v, p_v, f_v, t_v) \to_v (s'_v, p'_v, f'_v, t'_v)$ for all $v \in V$ and $\to_v$ is defined as follows:

$$\frac{p'_v < \theta_v}{(n, p_v, 0, t_v) \to_v (n, p'_v, 0, t_v + 1)} \text{ Rule 1}$$

$$\frac{p'_v \geq \theta_v}{(n, p_v, 0, t_v) \to_v (a, 0, 1, 0)} \text{ Rule 2}$$

$$\frac{t_v + 1 < \tau_v}{(a, 0, f_v, t_v) \to_v (a, 0, 0, t_v + 1)} \text{ Rule 3}$$

$$\frac{t_v + 1 \geq \tau_v}{(a, 0, 0, t_v) \to_v (r, 0, 0, 0)} \text{ Rule 4}$$

$$\frac{p'_v < \theta'_v \cdot \eta_v^{t_v+1} \qquad \theta'_v \cdot \eta_v^{t_v+1} > \theta_v}{(r, p_v, 0, t_v) \to_v (r, p'_v, 0, t_v + 1)} \text{ Rule 5}$$

$$\frac{p'_v < \theta'_v \cdot \eta_v^{t_v+1} \qquad \theta'_v \cdot \eta_v^{t_v+1} \leq \theta_v}{(r, p_v, 0, t_v) \to_v (n, p'_v, 0, 0)} \text{ Rule 6}$$

$$\frac{p'_v \geq \theta'_v \cdot \eta_v^{t_v+1}}{(r, p_v, 0, t_v) \to_v (a, 0, 1, 0)} \text{ Rule 7}$$

with $p'_v = \sum_{i=1}^{m} w_{i,v} \cdot f_i + \lambda_v \cdot p_v$ and where i is the i^{th} out of m input neuron of v, $w_{i,v}$ is the weight of the synapse connecting i and v, and f_i is the third component in the state of neuron i in configuration C.

More in detail, Rules 1 and 2 regulate the neuron when it is not in the refractory period (absolute or relative). In this case, at each instant, there are two possibilities:

Rule 1: the new potential (taking into account input spikes and the leak factor) is smaller than the firing threshold, thus the neuron remains in the normal state, it updates its potential and the timer increases of one unit,

Rule 2: the new potential is greater than the firing threshold, thus the neuron fires a spike (setting to 1 the boolean f_v), it changes its state to the absolute refractory period and resets the timer to 0.

During its absolute refractory period, the neuron ignores any received spike. The only visible change is that time passes by and the timer is incremented by one time unit (Rule 3). When the timer is greater than τ_v (the duration of the absolute refractory period), the neuron changes its state moving to the relative refractory state and resetting all the other variables of the state (Rule 4).

Last, as far as the relative refractory period is concerned, we have that its duration is determined by two parameters: θ'_v and η_v. We thus have three possible scenarios:

Rule 5: The neuron can receive spikes from its input neurons but the new potential p'_v does not exceed the firing threshold θ'_v (diminished by the threshold factor η_v) and θ'_v has not yet reached the normal firing threshold ($\theta'_v \cdot \eta_v^{t_v+1} > \theta_v$). In this case the neuron remains in the relative refractory state, it updates its potential and increases the timer.

Rule 6: In this case the neuron potential has not yet passed the firing threshold but the relative refractory period is terminated since θ'_v has been diminished until reaching θ_v. The neuron then returns in the normal state with the updated potential and the timer is reset.

Rule 7: The new potential is bigger than the firing threshold, then the neuron fires: $f'_v = 1$ and it moves to the absolute refractory state resetting to 0 both the potential and the timer.

In a spiking integrate and fire neural network, we distinguish three disjoint sets of neurons: V_i (input neurons), V_{int} (intermediary neurons), and V_o (output neurons), with $V = V_i \cup V_{int} \cup V_o$. Each input neuron receives as input an external signal. The output of each output neuron is considered as an output for the network. We have given the definition of input generators and output consumer in [11]. For the sake of this paper, it is sufficient to know that input generators are encoded as timed automata that provide trains of spike. Symmetrically, output consumers are timed automata that can in each moment receive spikes from the connected output neurons.

The encoding of neurons is as follows (it generalises the definition given in [10] by introducing the relative refractory period and removing the notion of accumulation period):

Definition 3. *Given a neuron* $v = (\theta_v, \theta'_v, \tau_v, \eta_v, \lambda_v)$ *with* m *input neurons, its encoding into timed automata is* $\mathcal{N} = (L, \mathbf{N}, X, Var, \Sigma, Arcs, Inv)$ *with:*

- $L = \{\mathbf{N}, \mathbf{F}, \mathbf{A}, \mathbf{R}\}$ *with* $\mathbf{F}$ *committed,*
- $X = \{t\}$
- $Var = \{p, f\}$
- $\Sigma = \{x_i \mid i \in [1..m]\} \cup \{x_v\}$,
- $Arcs =$
  $\{(\mathbf{N}, p + w_i \geq \theta_v, x_i?, \sim, \mathbf{F}) \mid i \in [1..m]\} \cup$
  $\{(\mathbf{N}, p + w_i < \theta_v, x_i?, \sim, \mathbf{N}) \mid i \in [1..m]\} \cup$
  $\{(\mathbf{R}, p + w_i \geq f, x_i?, \sim, \mathbf{F}) \mid i \in [1..m]\} \cup$
  $\{(\mathbf{R}, p + w_i < f, x_i?, \sim, \mathbf{R}) \mid i \in [1..m]\} \cup$

$\{(\mathbf{N}, t = 1, \sim, \{t := 0, p := p \cdot \lambda_v\}, \mathbf{N}),$
$(\mathbf{F}, \sim, x_v!, \{p := 0, t := 0\}, \mathbf{A})$
$(\mathbf{A}, t = \tau_v, \sim, \{t := 0, f := \theta'_v\}, \mathbf{R})$
$(\mathbf{R}, t = 1, \sim, \{t := 0, p := p \cdot \lambda_v, f := f \cdot \eta_v\}, \mathbf{R}),$
$(\mathbf{R}, \sim, \sim, \{t := 0\}, \mathbf{N})\}$

– $Inv(\mathbf{N}) = t < 1, \sim Inv(\mathbf{F}) = \mathbf{true},$
$Inv(\mathbf{A}) = t < \tau_v, Inv(\mathbf{R}) = t < 1 \wedge f > \theta_v.$

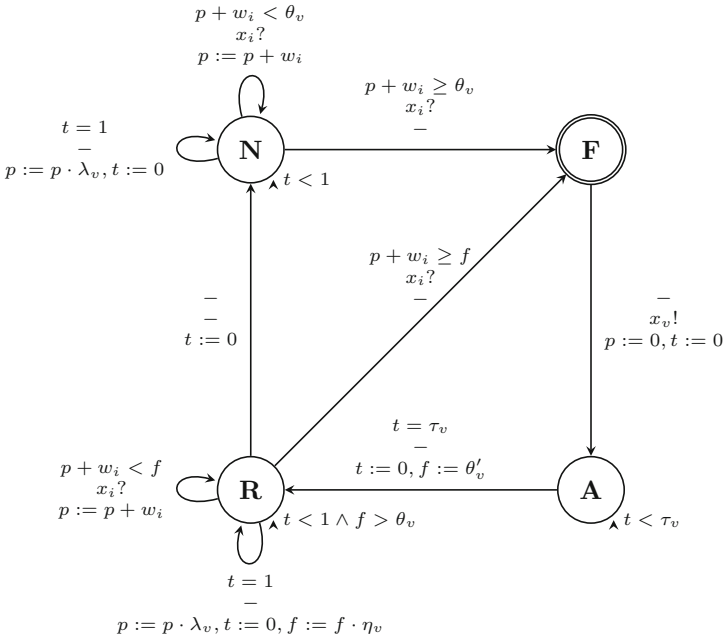


Fig. 2. Encoding of the spiking integrate and fire neuron with absolute and relative refractory period.

The automaton encoding of a neuron v is given in Fig. 2. We comment now on the definition. States $\mathbf{N}, \mathbf{F}, \mathbf{A}, \mathbf{R}$ represent, respectively, the fact that the neuron is in the normal state, it is firing, and it is in the absolute and then relative refractory period. Spikes are communicated through broadcast channels: each neuron $v \in V$ is associated to a channel x_v. The passing of time is explicit in the encoding, it is implemented by the self loop on state $\mathbf{N}$ with guard $(t = 1, -, \{p := p \cdot \lambda_v, t := 0\})$, by the similar self loop on state $\mathbf{R}$ with guard $(t = 1, -, \{p := p \cdot \lambda_v, t := 0, f := f \cdot \eta_v\})$, and by the arc from state $\mathbf{A}$ to state $\mathbf{R}$. All other actions (arcs) are meant to be instantaneous.

The rules above are encoded into arcs in the automata in the following way:

Rule 1: it is encoded by both self loops on state $\mathbf{N}$,
Rule 2: it is encoded by the arcs from $\mathbf{N}$ to $\mathbf{F}$ and from $\mathbf{F}$ to $\mathbf{A}$. Notice that, as state $\mathbf{F}$ is committed, time cannot pass by,

Rule 3: has no counterpart in the arcs, it is represented by the invariant $t < \tau_v$ on state **A**,
Rule 4: it is given by the arc from state **A** to **R**,
Rule 5: it is encoded by both self loops on state **R**,
Rule 6: it is represented by the arc from state **R** to **N**,
Rule 7: similarly as for Rule 2, it is encoded by the arcs from **R** to **F** and from **F** to **A**.

4 Parameter Inference

In this section we focus on the *Learning Problem*, which consists in determining a parameter assignment for a network with a fixed topology and a given input such that a desired output behaviour is displayed. More precisely, we examine the estimation of synaptic weights in a given spiking neural network and we leave the generalisation of our methodology to other parameters for future work.

Our technique takes inspiration from the SpikeProp algorithm [5]; in a similar way, here, the learning process is led by *supervisors*. Each output neuron $\mathcal{N}$ is linked to a supervisor. Supervisors compare the expected output behaviour with the one of the output neuron they are connected to (function EVALUATE($\mathcal{N}$) in Algorithm 1). Thus either the output neuron behaved consistently or not. In the second case, and in order to instruct the network, the supervisor back-propagates *advices* to the output neuron depending on two possible scenarios:

(i) the neuron fires a spike, but it was supposed to be quiescent,
(ii) the neuron remains quiescent, but it was supposed to fire a spike.

In the first case the supervisor addresses a *should not have fired* message (SNHF) and in the second one a *should have fired* (SHF). Then each output neuron modifies its ingoing synaptic weights and in turn behaves as a supervisor with respect to its predecessors, back-propagating the proper advice.

The advice back-propagation (ABP), Algorithm 1, is based on a depth-first visit of the graph topology of the network. Let $\mathcal{N}_i$ be the i-th predecessor of an automaton $\mathcal{N}$, then we say that $\mathcal{N}_i$ fired, if it emitted a spike during the current or previous accumulate-fire-wait cycle of $\mathcal{N}$. Thus, upon reception of a SHF message, $\mathcal{N}$ has to *strengthen* the weight of each ingoing *excitatory* synapse and *weaken* the weight of each ingoing *inhibitory* synapse. Then, it propagates a SHF advice to each ingoing *excitatory* synapse (i.e., an arc with weight greater than 0: $\text{WT} \geq 0$) corresponding to a neuron which *did not* fire recently ($\neg\text{F}(\mathcal{N})$), and symmetrically a SNHF advice to each ingoing *inhibitory* synapse ($\text{WT} < 0$) corresponding to a neuron which fired recently (see Algorithm 2 for SHF, and Algorithm 3 for the dual case of SNHF). When the graph visit reaches an input generator, it will simply ignore any received advice (because input sequences should not be affected by the learning process). The learning process ends when all supervisors do not detect any more errors.

There are several possibilities on how to implement supervisors and the ABP algorithm. We propose here two different techniques: the first one is model checking oriented while the second one is simulation oriented.

Algorithm 1. The advice back-propagation algorithm.

```
1: function ABP
2:     discovered = ∅
3:     for all 𝒩 ∈ Output do
4:         if 𝒩 ∉ discovered then
5:             discovered = discovered ∪ 𝒩
6:             if Evaluate(𝒩) = SHF then
7:                 SHF(𝒩)
8:             else if Evaluate(𝒩) = SNHF then
9:                 SNHF(𝒩)
```

Algorithm 2. Should Have Fired algorithm.

```
1: procedure Should-Have-Fired(𝒩)
2:     if 𝒩 ∈ discovered ∪ Output then
3:         return
4:     discovered = discovered ∪ 𝒩
5:     for all ℳ ∈ Pred(𝒩) do
6:         if ℳ ∉ Input then
7:             if Wt(𝒩, ℳ) ≥ 0 ∧ ¬ F(ℳ) then
8:                 SHF(ℳ)
9:             if Wt(𝒩, ℳ) < 0 ∧ F(ℳ) then
10:                SNHF(ℳ)
11:        Increase-Weight(𝒩, ℳ)
12:    return
```

As far as the first technique is concerned, it consists in iterating the learning process until a desired CTL temporal logic formula concerning the output of the network is verified. At each step of the algorithm, we make an external call to a model checker to test whether the network satisfies the formula or not. If the formula is verified, the learning process ends; otherwise, the model checker provides a trace as a counterexample. Such a trace is exploited to derive the proper corrective action to be applied to each output neuron, that is, the invocation of either the SHF procedure, or the SNHF procedure previously described (or no procedure).

In the second technique, parameters are modified during the simulation of the network. This entails that the encoding of neurons as automata needs to be adjusted in order to take care of the adaptation of such parameters. Algorithm ABP is realised by a dedicated automaton, and the role of supervisor is given to some output consumer automata. The idea is that, if an output neuron misbehaves, then its corresponding output consumer sets whether it has to be treated according to the SHF or the SNHF function. Furthermore, it signals that some adjustments on the network have to be done. Then the functions SHF or SNHF are recursively applied on the predecessors of the output neuron. Once there

Algorithm 3. Abstract ABP: Should *Not* Have Fired advice pseudo-code.

```
1: procedure SHOULD-NOT-HAVE-FIRED(neuron)
2:     if 𝒩 ∈ discovered ∪ Output then
3:         return
4:     discovered = discovered ∪ 𝒩
5:     for all ℳ ∈ PREDECESSORS(𝒩) do
6:         if ℳ ∉ Input then
7:             if WT(𝒩, ℳ) ≥ 0 ∧ F(ℳ) then
8:                 SNHF(ℳ)
9:             if WT(𝒩, ℳ) < 0 ∧ ¬ F(ℳ) then
10:                SHF(ℳ)
11:        DECREASE-WEIGHT(𝒩, ℳ)
12:    return
```

is no more neuron to whom the algorithm should be applied (for instance all neurons in the current run have been visited), the simulation is restarted in the network with the new parameters.

For a detailed description of the aforementioned techniques, the reader can refer to [11].

Example 2 (Turning on and off a diamond network of neurons). This example illustrates how the ABP algorithm can be used to make a neuron emit at least once in a spiking neural network having the *diamond* structure shown in Fig. 3. We assume that $\mathcal{N}_1$ is fed by an input generator $\mathcal{I}$ that continuously emits spikes. The neurons $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$ have the same tuple of parameters:

$$(\theta = 0.35, \theta' = 1.75, \tau = 3, \eta = 1, \lambda = \frac{7}{9})$$

while $\mathcal{N}_4$ has the parameters:

$$(\theta = 0.55, \theta' = 2.75, \tau = 3, \eta = 1, \lambda = \frac{1}{2}).$$

The initial weights are:

$w_{0,1}$	$w_{1,2}$	$w_{1,3}$	$w_{2,4}$	$w_{3,4}$
0.1	0.1	0.1	0.1	0.1

No neuron in the network is able to emit because all the weights of their input synapses are equal to 0.1 and their thresholds are 0.35.

We want the network to learn a weight assignment so that $\mathcal{N}_4$ is able to emit, that is, to produce a spike after an initial pause. At the beginning we expect no activity from neuron $\mathcal{N}_4$. As soon as the initial pause is elapsed, we require a spike but, as all weights are equal to zero, no emission can happen. Thus a SHF advice

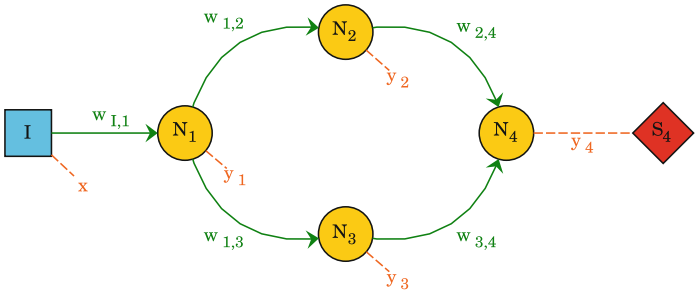


Fig. 3. A neural network with a diamond structure (Fig. 2 in [10]).

is back-propagated to the neurons $\mathcal{N}_2$ and $\mathcal{N}_3$ and consequently to $\mathcal{N}_1$. The process is then iterated until all weights stabilise and the neuron $\mathcal{N}_4$ is able to fire.

As far as the model checking approach is concerned, we want to test whether $\mathcal{N}_4$ can fire every 20 units of time. Notice that, as we cannot write recursive formulae, we only test if the formula is true for the first N times (N = 100 in our tests):

$$EG(((O4.n4 < N) \wedge (O4.O)) \Rightarrow (O4.s \leqslant 20)) \wedge EF(O4.O)$$

where $O4.n4$ is the number of spikes fired by $\mathcal{N}_4$, $O4.O$ states whether $\mathcal{N}_4$ has fired or not, and $O4.s$ is a clock signalling when $\mathcal{N}_4$ has fired. The last part of the formula ensures that the neuron fires at least once.

The next table details the steps of the model checking approach:

Step	Action	Update
1	$\mathcal{N}_4$ falsifies the formula	
2	SHF($\mathcal{N}_4$)	$w_{3,4} = 0.2$
		$w_{2,4} = 0.2$
3	SHF($\mathcal{N}_3$)	$w_{1,3} = 0.2$
4	SHF($\mathcal{N}_1$)	$w_{0,1} = 0.2$
5	SHF($\mathcal{N}_2$)	$w_{1,2} = 0.2$
6	$\mathcal{N}_4$ falsifies the formula	
7	SHF($\mathcal{N}_4$)	$w_{3,4} = 0.3$
		$w_{2,4} = 0.3$
8	SHF($\mathcal{N}_3$)	$w_{1,3} = 0.3$
9	SHF($\mathcal{N}_2$)	$w_{1,2} = 0.3$
10	$\mathcal{N}_4$ falsifies the formula	
11	SHF($\mathcal{N}_4$)	$w_{3,4} = 0.4$
		$w_{2,4} = 0.4$
12	SHF($\mathcal{N}_3$)	$w_{1,3} = 0.4$
13	SHF($\mathcal{N}_2$)	$w_{1,2} = 0.4$
14	The formula is satisfied	

For the simulation approach we obtain a similar sequence of calls to the SHF algorithm (changes in the order of calls are due to a different scheduling). The number of steps is also different as the checks on the validation of the formula are replaced by a simulation of the network. The obtained weights are only visible at the end of the simulation.

Step	Action
14	$\mathcal{N}_4$ does not behave as expected
21	SHF($\mathcal{N}_4$)
21	SHF($\mathcal{N}_2$)
21	SHF($\mathcal{N}_1$)
21	SHF($\mathcal{N}_3$)
39	$\mathcal{N}_4$ does not behave as expected
42	SHF($\mathcal{N}_4$)
42	SHF($\mathcal{N}_2$)
42	SHF($\mathcal{N}_3$)
62	$\mathcal{N}_4$ does not behave as expected
63	SHF($\mathcal{N}_4$)
63	SHF($\mathcal{N}_2$)
63	SHF($\mathcal{N}_3$)
83	$\mathcal{N}_4$ does not behave as expected
84	SHF($\mathcal{N}_4$)
488	$\mathcal{N}_4$ behaves as expected

Summing up, with the two approaches we obtain the following weights:

Method	$w_{0,1}$	$w_{1,2}$	$w_{1,3}$	$w_{2,4}$	$w_{3,4}$
Model checking	0.2	0.4	0.4	0.4	0.4
Simulation	0.2	0.4	0.4	0.5	0.5

Notice that with the simulation method we obtain a set of weights that is slightly different. This is not contradictory as the solution to the learning problem is not necessarily unique. ◇

5 Related Work

To the best of our knowledge, there are few attempts of giving formal models for LI&F. Apart from the already discussed approach of [13], where the authors model and verify LI&F networks thanks to the synchronous language Lustre, the closest related work we are aware of is [3]. In this work, the authors propose a mapping of spiking neural P systems into timed automata. The modelling is

substantially different from ours. They consider neurons as static objects and the dynamics is given in terms of evolution rules while for us the dynamics is intrinsic to the modelling of the neuron. This, for instance, entails that inhibitions are not just negative weights as in our case, but are represented as *forgetting rules*. On top of this, the notion of time is also different: while they consider durations in terms of number of applied rules, we have an explicit notion of duration given in terms of accumulation and refractory period.

As far as our parameter learning approach is concerned, we borrow inspiration from the SpikeProp rule [5], a variant of the well known back-propagation algorithm [29] used for supervised learning in second generation learning. The SpikeProp rule deals with multi-layered cycle-free spiking neural networks and aims at training networks 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 [18] and its time-dependent generalisation rule, the spike timing dependent plasticity (STDP) rule [30], which aims at adjusting the synaptical weights of a network according to the time occurrences of input and output spikes of neurons. It acts 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 the STDP, our approach takes into account not only recent spikes but also some external feedback (*advices*) in order to determine which weights should be modified and whether they must increase or decrease. 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 [17].

6 Conclusion

In this paper we formalised the LI&F model of spiking neural networks via timed automata networks. We improved the neuron model proposed in [10] by relaxing some stringent constraints related to spike emission times and by modelling a more realistic refractory period divided into absolute and relative one. We have a complete implementation of the proposed model and examples via the tool `Uppaal`, that can be found at the web pages [6] and [12]. As for future work concerning the modelling aspects, we plan to provide analogous formalisations for more complex spiking neuron models, such as the theta-neuron model [14] or the Izhikevich one [21]. We also intend to extend our model to incorporate propagation delays, which are considered relevant within the scope of spiking neural networks [26]. Our extension is intended to add some locations and clocks to model synapses. We also plan to perform a robustness analysis of the obtained model, in order to detect which neuron parameters influence most the satisfaction of some expected temporal properties.

As a salient contribution, we introduced a technique to learn the synaptical weights of spiking neural networks. At this aim, we adapted machine learning techniques to bio-inspired models, which makes our work original and complementary with respect to the main international projects aiming at understanding

the human brain, such as the Human Brain Project [9], which mainly relies on large-scale simulations.

For our learning approach, we have focussed on a simplified type of supervisors: each supervisor describes the output of a single specific neuron. However, the back-propagation algorithm still works for more complex scenarios that specify and compare the behaviour of sets of neurons. As for future work, we intend to formalise more involved supervisors, allowing to compare the output of several neurons. Moreover, to refine our learning algorithm, we could exploit some key results coming from the domain of gene regulatory networks, where some theorems linking the topology of the network and its dynamical behaviour are given [28].

To conclude, we intend to extend our methodology to the inference of other crucial parameters of neural networks, such as the leak factor or the firing threshold. We plan to consider the other parameters first one by one, and then all at the same time.

Acknowledgements. We are grateful to Laetitia Laversa for her preliminary implementation work and for her enthusiasm in collaborating with us.

References

1. Ackley, D.H., Hinton, G.E., Sejnowski, T.J.: A learning algorithm for Boltzmann machines. In: Waltz, D., Feldman, J.A. (eds.) Connectionist Models and Their Implications: Readings from Cognitive Science, pp. 285–307. Ablex Publishing Corporation, Norwood (1988)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
3. Aman, B., Ciobanu, G.: Modelling and verification of weighted spiking neural systems. Theor. Comput. Sci. **623**, 92–102 (2016). https://doi.org/10.1016/j.tcs.2015.11.005. http://www.sciencedirect.com/science/article/pii/S0304397515009792
4. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL—a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0020949
5. Bohte, S.M., Poutré, H.A.L., Kok, J.N., La, H.A., Joost, P., Kok, N.: Error-backpropagation in temporally encoded networks of spiking neurons. Neurocomputing **48**, 17–37 (2002)
6. Ciatto, G., De Maria, E., Di Giusto, C.: Additional material (2016). https://github.com/gciatto/snn_as_ta
7. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
8. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Math. Control Signals Syst. **2**(4), 303–314 (1989)
9. D'Angelo, E., et al.: The human brain project: high performance computing for brain cells HW/SW simulation and understanding. In: 2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, 26–28 August 2015, pp. 740–747. IEEE Computer Society (2015). https://doi.org/10.1109/DSD.2015.80

10. De Maria, E., Di Giusto, C.: Parameter learning for spiking neural networks modelled as timed automata. In: Anderson, P., Gamboa, H., Fred, A.L.N., i Badia, S.B. (eds.) Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2018). BIOINFORMATICS, Funchal, Madeira, Portugal, 19–21 January 2018, vol. 3, pp. 17–28. SciTePress (2018). https://doi.org/10.5220/0006530300170028
11. De Maria, E., Di Giusto, C.: Spiking neural networks modelled as timed automata with parameter learning. Research report, Université Côte d'Azur, CNRS, I3S, France, June 2018
12. De Maria, E., Di Giusto, C., Laversa, L.: Additional material (2017). https://digiusto.bitbucket.io/
13. De Maria, E., Muzy, A., Gaffé, D., Ressouche, A., Grammont, F.: Verification of temporal properties of neuronal archetypes using synchronous models. In: Fifth International Workshop on Hybrid Systems Biology, Grenoble, France (2016)
14. Ermentrout, G.B., Kopell, N.: Parabolic bursting in an excitable system coupled with a slow oscillation. SIAM J. Appl. Math. **46**(2), 233–253 (1986)
15. Freund, Y., Schapire, R.E.: Large margin classification using the perceptron algorithm. Mach. Learn. **37**(3), 277–296 (1999)
16. Gerstner, W., Kistler, W.: Spiking Neuron Models: An Introduction. Cambridge University Press, New York (2002)
17. Grüning, A., Bohte, S.: Spiking neural networks: principles and challenges (2014)
18. Hebb, D.O.: The Organization of Behavior. Wiley, New York (1949)
19. Hodgkin, A.L., Huxley, A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. J. Physiol. **117**(4), 500–544 (1952)
20. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. In: Anderson, J.A., Rosenfeld, E. (eds.) Neurocomputing: Foundations of Research, pp. 457–464. MIT Press, Cambridge (1988)
21. Izhikevich, E.M.: Simple model of spiking neurons. Trans. Neural Netw. **14**(6), 1569–1572 (2003)
22. Izhikevich, E.M.: Which model to use for cortical spiking neurons? IEEE Trans. Neural Netw. **15**(5), 1063–1070 (2004)
23. Lapicque, L.: Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. J. Physiol. Pathol. Gen. **9**, 620–635 (1907)
24. Maass, W.: Networks of spiking neurons: the third generation of neural network models. Neural Netw. **10**(9), 1659–1671 (1997)
25. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**(4), 115–133 (1943)
26. Paugam-Moisy, H., Bohte, S.: Computing with spiking neuron networks. In: Rozenberg, G., Bäck, T., Kok, J.N. (eds.) Handbook of Natural Computing, pp. 335–376. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-540-92910-9_10
27. Recce, M.: Encoding information in neuronal activity. In: Maass, W., Bishop, C.M. (eds.) Pulsed Neural Networks, pp. 111–131. MIT Press, Cambridge (1999)
28. Richard, A.: Negative circuits and sustained oscillations in asynchronous automata networks. Adv. Appl. Math. **44**(4), 378–392 (2010)
29. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. In: Anderson, J.A., Rosenfeld, E. (eds.) Neurocomputing: Foundations of Research, pp. 696–699. MIT Press, Cambridge (1988)
30. Sjöström, J., Gerstner, W.: Spike-timing dependent plasticity. Scholarpedia **5**(2), 1362 (2010)