IOP Publishing Neuromorph. Comput. Eng. 6 (2026) 014004 https://doi.org/10.1088/2634-4386/ae2cc1

NEUROMORPHIC
Computing and Engineering

PAPER

Neuromorphic complexity theory: computational models and complexity classes for spiking neural networks*

Johan Kwisthout[1], Arne Diehl[**] and Nils Donselaar[2]

Radboud University, Donders Institute for Brain, Cognition, and Behaviour, Nijmegen, The Netherlands

[1] Supported by a Grant from Intel Corporation.
[2] Supported by NWO Grant 612.001.601.

[**]Author to whom any correspondence should be addressed.

E-mail: arne.diehl@donders.ru.nl

Keywords: neuromorphic computation, spiking neural networks, structural complexity theory, energy complexity

OPEN ACCESS

RECEIVED
15 May 2025

REVISED
12 November 2025

ACCEPTED FOR PUBLICATION
15 December 2025

PUBLISHED
20 January 2026

Abstract

The last decade has seen the rise of neuromorphic architectures based on artificial spiking neural networks (SNNs), such as the SpiNNaker, TrueNorth, and Loihi systems. The massive parallelism and co-locating of computation and memory in these architectures potentially allows for an energy usage that is orders of magnitude lower compared to traditional Von Neumann architectures. However, to date a comparison with more traditional computational architectures (particularly with respect to energy usage) is hampered by the lack of a formal machine model and a computational complexity theory for neuromorphic computation. In this paper we take the first steps towards such a theory. We introduce three (increasingly complex) machine models based on SNNs where—in contrast to the familiar Turing machine—information and the manipulation thereof can be co-located in the machine. Using these models, we introduce canonical problems for our machine models, define hierarchies of complexity classes, and provide some first completeness results.

1. Introduction

Moore's law [16] states that the number of transistors in integrated circuits (ICs) doubles approximately every two years. As transistors become faster, overall IC performance has historically doubled about every 18 months, though this has come with increased energy consumption [22]. However, Moore's law appears to be slowing down [3]. Engineers from Intel and ASMC have independently reported that the trend still holds for now, but both emphasise that it is unlikely to continue indefinitely [20, 24]. Although advances in chip design and manufacturing continue to offer performance gains, they remain constrained by the limits of traditional 'Von Neumann' (VN) architectures. As VN architectures separate computation and memory by a bus, they require both data and algorithm to be transferred from memory to the CPU with every instruction cycle. This separation, known since 1978 as the VN bottleneck [2], has become increasingly problematic: while CPU speeds have grown rapidly, memory access and data transfer rates have lagged behind [10].

While more data than ever before is being generated, computing now faces fundamental limits: the end of Moore's law, performance bottlenecks in the VN architecture, and rising energy consumption with its associated carbon footprint [19]. These challenges have accelerated the development of several generations of neuromorphic hardware architectures [4, 6, 11, 15]. Inspired by the structure of the brain (largely parallel computations in neurons, low power consumption, event-driven communication via synapses) these architectures co-locate computation and memory in artificial spiking neural networks (SNNs). The spiking behaviour allows for potentially energy-lean computations [14] while still allowing

* An earlier version of this paper appeared in the *Proceedings of the Neuro-inspired Computational Elements Workshop* (NICE 2020) [12].

for in principle any conceivable computation [13]. However, we do not yet fully understand the potential and limitations of these new architectures. Benchmarking results are suggesting that event-driven information processing (e.g. in neuromorphic robotics or brain–computer-interfacing) and energy-critical applications might be suitable candidate problems, whereas 'deep' classification and pattern recognition (where SNNs are outperformed by convolutional deep neural networks) and applications that value precision over energy usage may be less natural problems to solve on neuromorphic hardware. Although several algorithms have been developed to tackle specific problems [5], there is currently no insight in the potential and limitations of neuromorphic architectures.

The emphasis on *energy* as a vital resource, in addition to the more traditional *time* and *space*, suggests that the traditional machine models in complexity theory (i.e. Turing machines and Boolean circuits) and the corresponding formal machinery (reductions, hardness proofs, complete problems etc) are ill-matched to capture the computational power of SNNs. What is lacking is a unifying computational framework and structural complexity results that can demonstrate what can and cannot be done in principle with bounded resources with respect to convergence time, network size, and energy consumption [9]. Previous work is mostly restricted to variations of Turing machine models within the VN architecture [7] or energy functions defined on threshold circuits [23] and as such unsuited for studying SNNs. This is nicely illustrated by the following quote:

> 'It is … likely that an entirely new computational theory paradigm will need to be defined in order to encompass the computational abilities of neuromorphic systems' [19, p 29]

In this paper we propose several machine models for SNN-based neuromorphic architectures and lay the foundations for a *neuromorphic complexity theory*. At this stage, we unfortunately lack any results pertaining to energy complexity specifically. However, our proposed framework turns this into an explicit dimension of analysis, and it is our hope that this will both facilitate and encourage future work towards obtaining such results. The remainder of this paper is structured as follows. In section 2 we will introduce our machine models in detail, elaborating on both standalone neuromorphic computation as well as hybrid models that combine traditional with neuromorphic computation. In section 3 we further elaborate on the resources time, space, and energy relative to our machine models. In section 4 we will explore the complexity classes associated with these machine models and derive some basic structural properties and hardness results. We conclude the paper in section 5.

2. Machine model

In order to abstract away from the actual computation on a neuromorphic device, in a similar vein as the Turing machine acts as an abstraction of computations on traditional hardware architectures, we introduce a novel notion of computation based on SNNs. While we acknowledge that not all neuromorphic architectures are spiking-based, it is the underlying principle of the most prominent architectures such as Intel's Loihi [4], the HBPs Spinnaker [6], and IBM's TrueNorth [1]. We will first elaborate on the network model and then proceed to translate that to a formal machine model.

2.1. SNN model

We model the SNN as a discrete-time dynamical system, as many of the systems that inspire our work are based on this type of SNN model. Generally speaking, a SNN mimics the dynamics of biological neural systems. It consists of discrete-time leaky integrate-and-fire (LIF) neurons, weighted and delayed synapses, and neurons with fixed spiking patterns. This framework enables the SNN to model intricate spatial arrangements and temporal dynamics, presenting a simplified model for neural processing. The core of the SNN, the discrete-time LIF neuron, replicates the membrane potential dynamics of biological neurons, updating periodically in a synchronised fashion. Its potential changes based on synaptic inputs and an intrinsic decay, mimicking a biological neuron's leaky nature. When this potential exceeds a certain threshold, the neuron fires a spike, reflecting natural neurons' all-or-nothing response. After firing a spike the neuron's membrane potential resets to the reset voltage. Figure 1 illustrates the behaviour of such a leaky integrate-and-fire neuron with multiple inputs and outputs.

In addition to LIF neurons, we also make use of programmed neurons. These neurons are set to fire at predetermined intervals which allows us to introduce 'external' stimuli or specific internal activity patterns, linking the SNN with the outside world or particular computational tasks. These neurons allow for controlled network manipulation, aiding in the exploration of network reactions to set stimuli or in achieving defined computational objectives.

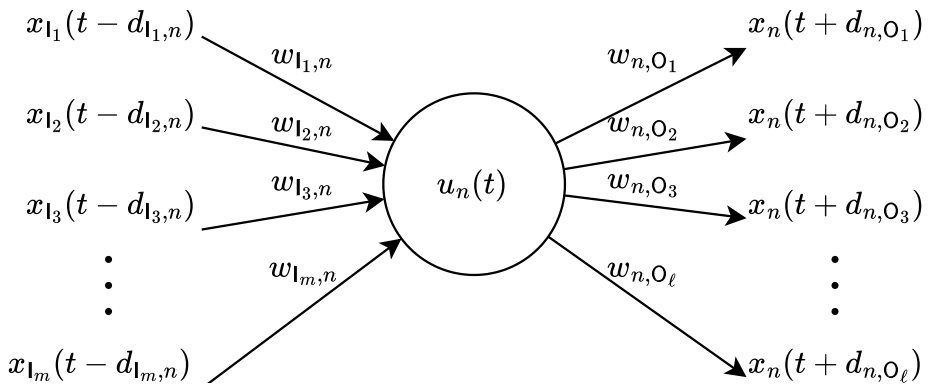


Figure 1. The figure shows the behaviour of a leaky integrate-and-fire neuron n with multiple inputs and outputs. The set of input neurons is labelled $\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_m$, each connecting to neuron n with respective weight w and delay d. On the right, the neuron n projects to the set of output neurons $\mathsf{O}_1, \mathsf{O}_2, \ldots, \mathsf{O}_\ell$ via synapses with respective weights w and delays d. Adapted with permission from [12]. Copyright © 2020 ACM.

Finally, synapses connect two neurons and are furthermore defined by their weights and delays. These synapses allow for directional neuronal communication from the presynaptic neuron to the postsynaptic neuron. Weights signify a synapse's impact on the receiving neuron, with positive weights for excitatory and negative weights for inhibitory effects. Delays introduce signal transmission times, adding temporal depth to the network's functionality. The behaviour of LIF neurons, programmed neurons and synaptic connections with weights and delays simulated in discrete time steps, gives rise to rich SNN dynamics, while offering a straightforward path to map neuronal activity to computational steps.

In formal terms, an SNN is a tuple (N, S) where N is a finite set of neurons and S is a finite set of synapses, which can be represented as directed edges between the neurons. Elements of the set N can either be LIF neurons or programmed neurons. LIF neurons are defined by their threshold T, initial voltage V, reset voltage R and leakage constant m, while programmed neurons are defined by a spike train (an arbitrary long but finite string of zeros and ones that defines the spiking behaviour for each time step). Formally, we define $\mathsf{LIF} = \mathbb{Q}_{\geqslant 0} \times \mathbb{Q}_{\geqslant 0} \times \mathbb{Q}_{\geqslant 0} \times \mathbb{Q}_{[0,1]}$, so that every LIF neuron is of some type $(T, V, R, m) \in \mathsf{LIF}$; we similarly define $\mathsf{PN} = \{0,1\}^*$ so that every programmed neuron is of some type $\sigma \in \mathsf{PN}$. Using these definitions, we say N is a set of n neurons, each of which is either of a type $\tau(i) = (T_i, V_i, R_i, m_i)$ (for a LIF neuron) or a type $\tau(i) = \sigma_i$ (for a programmed neuron).

We denote the unique synapse from neuron i to neuron j, as $s_{i,j}$. In general, a synapse is defined by a 4-tuple (i, j, d, w) which refer to the presynaptic neuron, the postsynaptic neuron, the delay and the weight of the synapse respectively. This means that formally S is a finite subset of $\mathsf{N} \times \mathsf{N} \times \mathbb{N}_{>0} \times \mathbb{Q}_{>0}$.

Our SNN model is partially based on work by Severa and colleagues at Sandia National Labs [21]. We use functions u and x to define the behaviour of the LIF neuron model. The potential function u maps pairs consisting of a LIF neuron and a timestep to a membrane voltage, while x maps pairs consisting of a neuron of any type and a timestep to a spike event. For LIF neurons, x depends on u while it does not for programmed neurons. Within a given SNN (N, S), the function $u : \mathsf{N} \times \mathbb{N} \to \mathbb{Q}_{\geqslant 0}$ (defined only for LIF neurons) is described by the following equation:

$$u(i,t) = u_i(t) = \begin{cases} V_i & \text{if } t = 0 \\ \max\left(0, R_i + \sum_{j \in N} w_{ji} x_j (t - d_{ji})\right) & \text{if } t > 0, u_i(t-1) \geqslant T_i \\ \max\left(0, m_i \cdot u_i(t-1) + \sum_{j \in \mathsf{N}} w_{ji} x_j (t - d_{ji})\right) & \text{otherwise.} \end{cases}$$

The function $x : \mathsf{N} \times \mathbb{N} \to \{0,1\}$ is formalised by the following equation:

$$x(i,t) = x_i(t) = \begin{cases} 1 & \text{if } \tau(i) \in \mathsf{LIF} \text{ and } u_i(t) \geqslant T_i \\ 1 & \text{if } \tau(i) \in \mathsf{PN} \text{ and } \sigma_i(t) = 1 \\ 0 & \text{otherwise .} \end{cases}$$

Here $\sigma_i(t)$ is the tth element of the spike train of a programmed neuron i of type σ_i.

The above definitions were chosen for their simplicity. The results in the following sections also apply to more complex neuron models, provided these can be simulated by our model with a resource

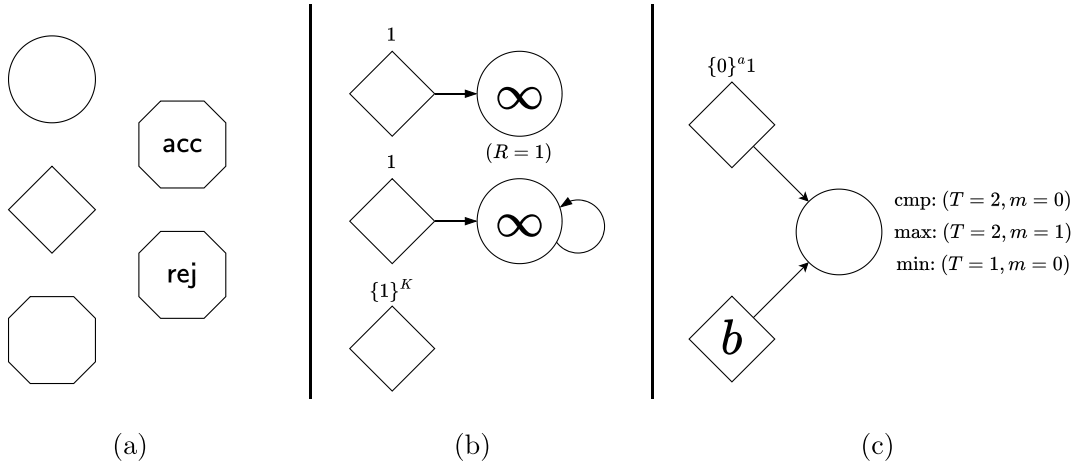


Figure 2. (a): notational conventions for (top to bottom on the left) a regular neuron, a programmed neuron, and a read-out neuron; dedicated acceptance and rejection neurons (top to bottom on the right). (b): three different ways of constructing a continuously firing neuron by (top to bottom) a single spike feeding into a neuron with $R = T = 1$, a single spike feeding into a neuron with a recurrent synapse, and an explicit spike train firing K times. (c): from top to bottom: comparing, taking max, and taking min of two natural numbers a and b encoded by spike delays. Note the shorthand notation 'b' for a programmed neuron with spike train $\{0\}^b 1$. Adapted with permission from [12]. Copyright © 2020 ACM.

overhead consistent with the respective result. Our analysis does not, however, extend to neuron models that incorporate a form of unbounded memory or stochastic processes.

2.2. Spiking machine (SM) model

Our formal machine model for a general SM now consists of a 3-tuple $\mathcal{S} = (\mathsf{N}, \mathsf{S}, \mathsf{acc})$, where (N, S) are a set of neurons and synapses respectively which together make up a SNN as described previously, and where $\mathsf{acc} \in \mathsf{N}$ indicates the acceptance neuron. This definition, though seemingly more concise since the underlying SNN contains most of the specifications, is similar to how a Turing machine $\mathcal{M}$ can be described as a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ which refers to its state space, input alphabet, tape alphabet, transition function, starting state, blank symbol and accepting states respectively (Hopcroft *et al* 2007). While this definition may seem abstract, it formally describes the structure of an ordinary SNN, viewed as a computational model with specified input and output neurons.

Having established this formal notion of a SM model, we will subsequently build on this concept by exploring certain variations of it, such as a SM with a dedicated rejection neuron. See figure 2 for notational conventions regarding the visualisation of SMs or parts thereof.

2.3. Machine models

In order to describe the computation on a Turing machine, one typically describes a generic decision problem as follows: 'Given a machine $\mathcal{M}$ and input i on its tape, does $\mathcal{M}$ accept i using resources at most R'? Following this convention, i denotes the input rather than a neuron index throughout the remainder of this paper. With this notation in place, we now let L be the language that $\mathcal{M}$ should recognise, i.e. the job of $\mathcal{M}$ is to decide whether $i \in L$. To translate such problems to a SM model, one not only needs to define the SM itself, but also the resources R that this machine may use, how the input i is encoded, and what it means for the machine to accept the input i using resources R. In the remainder of this section we will define and explore three distinct variations on the SM model.

2.3.1. Spike-input model

A straightforward mapping from this description to a SM model physically separates the machine and the input to it. We will call this approach the *spike input model* $\mathcal{S}$, where $\mathcal{S}$ has a designated programmed neuron $\mathsf{in}(\mathcal{S})$ and designated readout neuron $\mathsf{acc}(\mathcal{S})$; the input i is encoded as the programmed spike train of $\mathsf{in}(\mathcal{S})$, and we define $i \in L_{\mathcal{S}}$ to be the case precisely when $\mathsf{acc}(\mathcal{S})$ fires upon $\mathcal{S}$ receiving the input i. For convenience, an additional readout neuron $\mathsf{rej}(\mathcal{S})$ may be assumed (see discussion above) that fires if and only if $i \notin L_{\mathcal{S}}$. As we will show in section 3.1, we may extend the spike train to the singleton input neuron $\mathsf{in}(\mathcal{S})$ to a spike wave feeding into multiple neurons with only a linear increase in resource needs. This approach can graphically be depicted as in figure 3. The complexity classes associated with decision problems that can be decided using this model, where the allowed resources are described by $R_{\mathcal{S}}$, will be denoted as $\mathcal{S}(R_{\mathcal{S}})$.

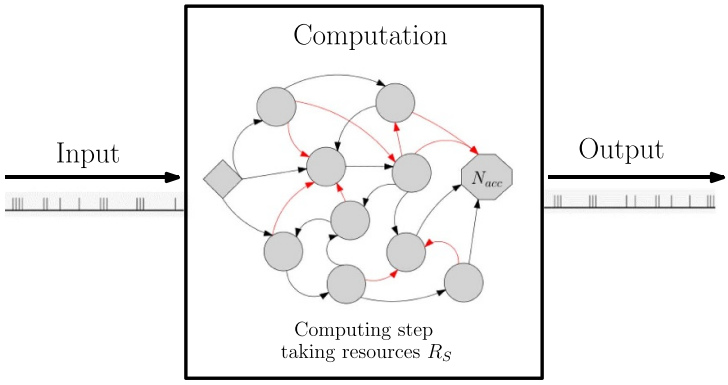


Figure 3. Schematic representation of the spike-input variant of the spiking machine model. The input spike train on the left is fed into some spiking machine $\mathcal{S}$, represented here as a black box containing a SNN. Within this box, spikes propagate from the input neuron $\mathsf{in}(\mathcal{S})$ through interconnected neurons, which may result in a spike in the readout neuron $\mathsf{acc}(\mathcal{S})$ which represents the output of the spiking machine model to the right. The diagram visualises the notion of a spiking machine as a computational unit mapping input to output.

This approach is conceptually somewhat similar to a Turing machine as it separates input and computation. Unlike a Turing machine's working tape, however, the spike train represents an unalterable read-once stream of information. We can thus think of this concept as a read-only Turing machine.

2.3.2. Pre-processing model

The simplicity of the basic spike-input model defies the basic premise that in neuromorphic architectures both the input and the algorithm operating on it are encoded in the SM's structure and parameters. That is, while the most straightforward way of encoding an input might indeed be through programming a spike train onto an input neuron, in some cases it might be more efficient to encode it otherwise, such as at the level of synaptic weights or even delays. In this approach, input and computation are no longer physically separated but are co-located within the SM.

However, when characterising the resource usage of the machine model when input and computation are co-located, we are presented with a challenge. Observe that, unlike what is the case for families of Boolean circuits (where there can be a separate circuit per input length), here potentially *every input* relates to a specific SM. This extreme non-uniformity makes the asymptotic analysis of such machines difficult, as there is no straightforward way of characterising asymptotic general behaviour. Hence, in this model we explicitly designate the *construction* of the SM as part of the computation. We will call this approach the *pre-processing model* $\mathcal{M} \circ \mathcal{S}$ described below.

In this model, a machine $\mathcal{S}_i$ may encode both the input i and the algorithm deciding whether $i \in L$. What it means to decide a problem L using a SM now becomes the following: that there is an $R_{\mathcal{M}}$-resource-bounded Turing machine $\mathcal{M}$ that generates a SM $\mathcal{S}_i$ for every input i provided on its input tape, such that $\mathcal{S}_i$ decides whether $i \in L$ using resources at most $R_{\mathcal{S}}$. $\mathcal{S}_i$ again has access to a spike input train i' (provided by $\mathcal{M}$) and accepts or rejects using designated neurons. Formally, we define a mapping $i \to (\mathcal{S}_i, i')$, characterise the mapping by resources $R_{\mathcal{M}}$, and the SNN computation by resources $R_{\mathcal{S}}$. Figure 4 depicts this approach.

Note that in this definition the workload is *shared* between the Turing machine $\mathcal{M}$ and the SM $\mathcal{S}_i$, and that the definition naturally allows for trading off generality of the SM (acceptance of different spike input trains by the same SM) and generality of the Turing machine (generating different SMs for each distinct input). Figure 5 provides a simple comparison between three implementations of the ARRAY SEARCH-problem: given an array A of integers and a number i, does A contain i? Note that in the rightmost example a 'circuit approach' is emulated, with a uniform family of SMs, one for each array length. We are not currently aware of a straightforward way to simulate the entire computation for arrays of arbitrary size in the SM, and it is unclear if the current notion even allows for this.

We can informally see the Turing machine $\mathcal{M}$ as a sort of *pre-processing* computation generating the SM $\mathcal{S}_i$ and then deferring the actual decision to accept or reject the input to this machine. We will use the notation $\mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$ to refer to the class of decision problems that can be decided in this way.

While the previous model had some conceptual similarity with a Turing machine, this model has similarities with a uniform family of Boolean circuits. In these models, a different circuit may be used for every input size $|x|$, yet, the circuit is independent of the *contents* of the input. The uniformity condition requires the existence of a resource-bounded mapping $1^n \to C_n$. However, our machine model

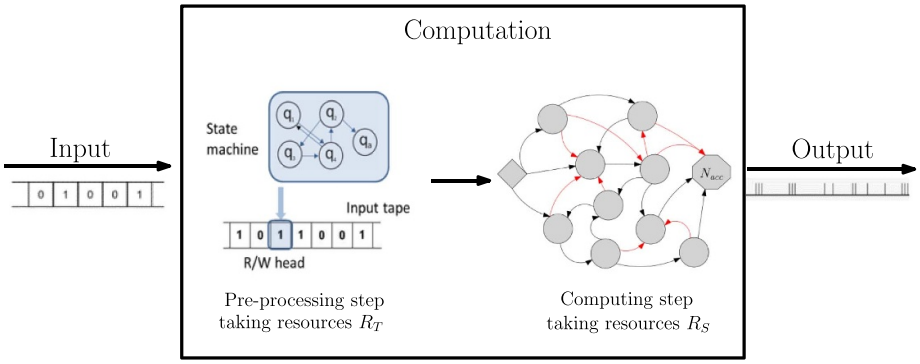
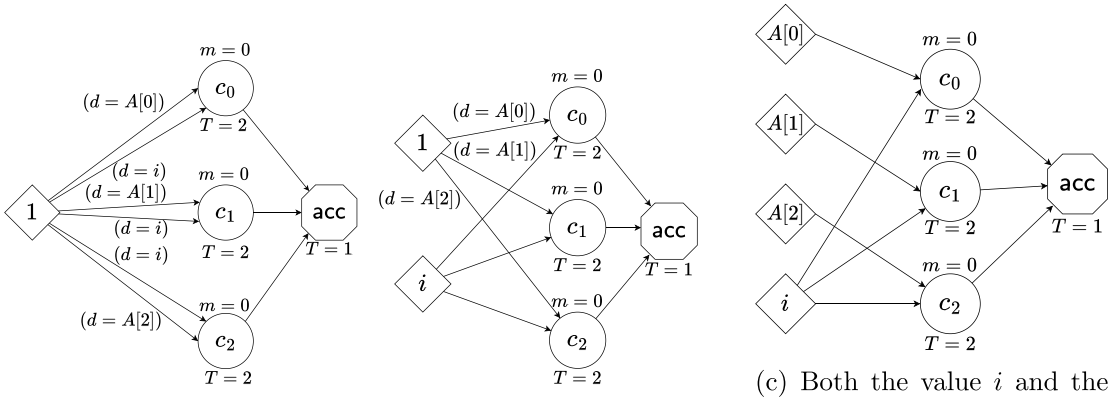


Figure 4. Schematic representation of the pre-processing machine model as a black box receiving input on the left and producing output to the right of the box. Within the box are the constituent elements of the pre-processing machine model, namely a stylised Turing machine and a SM represented by its SNN. Input to the pre-processing machine is first received by the Turing machine, which constructs the SM to the right which in which its input and computation are co-located. The execution of the SM finally yields the output of the overarching pre-processing machine model.



(a) All computation in the spiking machine

(b) The value i offered to the spiking machine as input

(c) Both the value i and the array A offered to the spiking machine as input

Figure 5. Three spiking machines designed to decide whether an array A of natural numbers contains i. Note that in figure (a) both A and i as well as the parallel comparison are encoded in the machine; in figure (b) the search value i is offered as input (using a spike train consisting of a single spike with delay i), and in figure (c) both the search value and the integers in the array are offered as input to the machine, while the size of the array is fixed. The number of spikes used in the computation is respectively $\leqslant |n| + 2$, $\leqslant |n| + 3$, and $\leqslant 2|n| + 2$; the generality of the machine increases but this comes at the price of the increasing number of spikes in the computation. Adapted with permission from [12]. Copyright © 2020 ACM.

generalises over these uniformity conditions, as we do not constrain the pre-processing to operate only on the input length[3].

It is convenient, though, to define and observe the behaviour of this constrained pre-processing. We define an $R_{\mathcal{M}}$-*uniform family* $\mathcal{S}_{\mathcal{F}}^{R_{\mathcal{M}}}(R_{\mathcal{S}})$ of SMs by constraining the $R_{\mathcal{M}}$-bounded Turing machine to realise the mapping $1^{|i|} \rightarrow \mathcal{S}_{|i|}$, where $\mathcal{S}_{|i|}$ is defined as in section 2.3.1, and assume a uniform ($R_{\mathcal{M}}$-bounded) encoding of the input i as a spike train i' feeding into $\mathcal{S}_{|i|}$. Trivially, $\mathcal{S}_{\mathcal{F}}^{R_{\mathcal{M}}}(R_{\mathcal{S}}) \subseteq \mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$.

As a final note on this model, observe that we really need the pre-processing to be part of the definition of the model for neuromorphic computation to meaningfully define resource-bounded computations, as we are allowed in principle to define a unique SM per instance i. Otherwise, the mapping from i to $\mathcal{S}_i$ could be the trivial and uninformative mapping:

$$i \rightarrow \mathcal{S}_i : \begin{cases} \mathcal{S} \text{ with } \sigma_{\mathsf{acc}(\mathcal{S})} = 1 & \text{if } i \in L \\ \mathcal{S} \text{ with } \sigma_{\mathsf{rej}(\mathcal{S})} = 1 & \text{otherwise} \end{cases}$$

[3] Note though that it is entirely possible to consider intermediate levels of generalisation between depending only on the input length and depending on the input in a fully arbitrary manner (constrained only by the bounds imposed on the computational resources). For instance, we may consider a mapping where the SM *structure* depends only on the input size or a similar feature, but the SM *parameters* are allowed to depend on the 'contents' of the input.

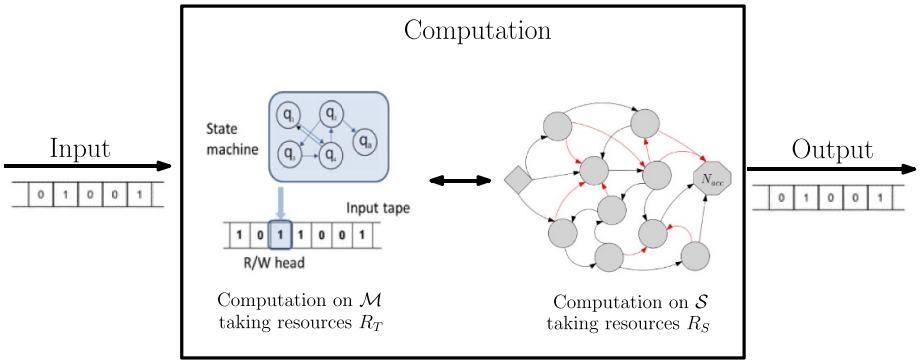


Figure 6. Interactive computational model where aspects of computation can be delegated from the Turing machine to the spiking machine, allowing for hybrid algorithms.

where σ returns the spike train string of a programmed neuron.

2.3.3. Interactive model

In addition to the pre-processing model, we can also consider a setting in which a Turing machine and a SM interact dynamically during computation, similar to how a CPU may interleave its execution with instructions handled by a coprocessor such as a GPU. In this model, we allow an *iterative* interaction between $\mathcal{M}$ and an oracle capable of deciding whether a SM $\mathcal{S}$ accepts, such that the computation carried out by $\mathcal{M}$ is interleaved with oracle calls whose results can be acted on accordingly (see figure 6).

We will formalise the interactive model of neuromorphic computation in terms of Turing machines equipped with an oracle for a specific SM. This involves augmenting a deterministic Turing machine with a *query tape*, an *oracle-query* state, and an *oracle-result* state. Now when a machine with such an oracle enters the oracle-query state with $(\mathcal{S}, i)$ on its query tape it proceeds to the oracle-result state, at which point it will replace the contents with 1 if $\mathcal{S}$ accepts and with 0 if $\mathcal{S}$ rejects (given that the query state holds a correct description of $\mathcal{S}$; the outcome otherwise returned is unspecified). We will call this approach the *interactive model* $\mathcal{M}^{\mathcal{S}}$ described below; the corresponding complexity classes for suitable choices of $R_{\mathcal{M}}$ and $R_{\mathcal{S}}$ are characterised[4] as $\mathcal{M}(R_{\mathcal{M}})^{\mathcal{S}(R_{\mathcal{S}})}$.

In section 4 we will cover the formal aspects involved in these definitions; we start in the next section by formalising the resources $R_{\mathcal{S}}$ and $R_{\mathcal{M}}$ that we wish to allocate to these machines.

3. Resources

We denote the resource constraints of the Turing machine $\mathcal{M}$ (when part of the model) with the tuple $R_{\mathcal{M}} = (\mathsf{TIME}, \mathsf{SPACE})$. We allow the decision of the SM $\mathcal{S}$ to take resources $R_{\mathcal{S}}$; this can be further specified to be a tuple $R_{\mathcal{S}} = (\mathsf{TIME}, \mathsf{SPACE}, \mathsf{ENERGY})$, referring to the number of time steps $\mathcal{S}$ may use, the total size of the SM $|\mathcal{S}|$, and total number of spikes that $\mathcal{S}$ is allowed to use, all as a function of the size of the input i, either explicated as the length of the spike train feeding directly in $\mathcal{S}$, or as the length of the input tape feeding into $\mathcal{M}$. Note that for $\mathcal{S}$ in practice $\mathsf{ENERGY} \leqslant \mathsf{TIME} \times \mathsf{SPACE}$ since any neuron can fire at most once per time step[5]. Furthermore, note that similarly SPACE is upper bounded by $R_{\mathcal{M}}$, as for example we cannot in polynomial time construct a SM with an exponential number of neurons. We assume in the remainder that the constraints can be described by their asymptotic behaviour, and in particular that they are closed under scalar and additive operations; we will describe $R_{\mathcal{M}}$ and $R_{\mathcal{S}}$ as being *well-behaved* if they adhere to this assumption. (To clarify, here we restrict ourselves to considering only deterministic resources for both $R_{\mathcal{M}}$ and $R_{\mathcal{S}}$, just as we consider only deterministic membrane potential functions.)

When the constraints $R_{\mathcal{M}}$ are such that $\mathcal{M}(R_{\mathcal{M}})$ characterises familiar complexity classes we will use the common notation for that class from here on; as an abuse of notation we will also use this notation as a shorthand for the resources $R_{\mathcal{M}}$ themselves.

[4] There is a slight technicality here: as we cannot assume that $\mathcal{S}(R_{\mathcal{S}})$ as complexity class has complete problems for all choices of $R_{\mathcal{S}}$, this notation is to be interpreted as the union of all oracles for SMs with $R_{\mathcal{S}}$ resource bounds.

[5] We thank the reviewer for pointing out that the inequality $E \leqslant T \times S$ holds only under our definition of energy complexity based on spike events. When energy is instead defined in terms of synaptic operations, the corresponding upper bound becomes $E \leqslant T \times S^2$. The results in the following sections remain unaffected by this choice of definition.

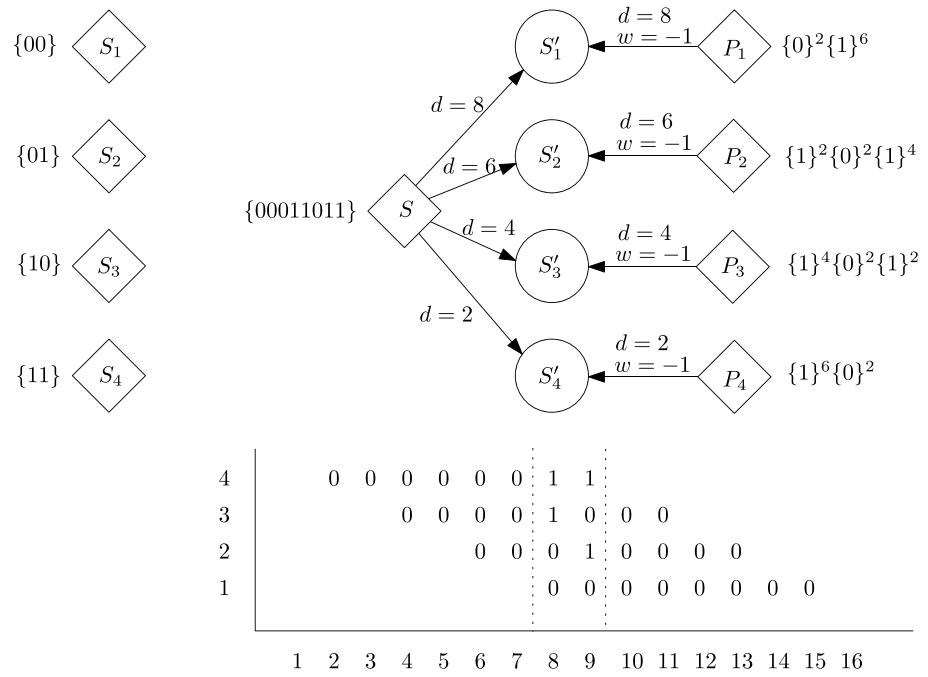


Figure 7. Example construct to separate $n = 4$ concatenated spikes trains of length $k = 2$ into separate virtual spike trains.

3.1. Spike waves vs. singleton spike train

It is often convenient to assume that multiple spike trains (or *spike waves*) can be injected simultaneously in SNNs via different input neurons. However, to keep the models of computation as simple as possible, we will in our proofs assume a single spike train to a designated input neuron. Here we will show that we can do so without loss of generality with linear delay and a linear number of additional spikes, assuming a constant number m of spike trains and a finite length k of each train. To exemplify our approach we will demonstrate a construction where a concatenated spike input stream, consisting of $m \cdot k$ spikes, can be transformed to m virtual input neurons that, after a delay of $m \cdot k$ time slices, will simultaneously produce m spike trains of length k, after which they will be silent.

We assume an input neuron S that gets as spike train $k_1 k_2 \ldots\ k_n$ that needs to be separated into n spike trains $k_1, k_2, \ldots k_n$. We introduce n neurons S_i $(i = 1 \ldots n)$ and connect S to S_i via a synapse with weight 1 and delay $i \cdot k$. We also introduce n programmed neurons P_i with programmed spike train $\{1\}^{(i-1)k}\{0\}^k\{1\}^{(n-i)k}$; each P_i is connected to S_i with weight -1 and delay $i \cdot k$. This effectively mutes the virtual input neuron for all but its designated part of the concatenated spike train; the delays ensure that all virtual input neurons spike at the same time after delay nk. In figure 7 an example is given for $k = 2$, $n = 4$, and spike wave $\{00, 01, 10, 11\}$ to the four input neurons. In the timing scheme below (where actively inhibited spikes are explicated as '0') we see the desired behaviour. Note that this construct produces, assuming a constant n, a linear amount $n^2 k$ of additional spikes and a linear delay $n \cdot k$.

3.2. Clock and meter

According to a classical and generally known result (see [8]) one can for any Turing machine $\mathcal{M}$ provide an equivalent machine $\mathcal{M}'$ which effectively has access to a *clock* and a *ruler*, such that $\mathcal{M}'$ enters the rejection state immediately when these bounds are violated, using overhead which is typically negligible from a complexity perspective. As a consequence one is often allowed to work under the assumption that Turing machines possess such clocks and rulers, as we shall also do here for convenience. To demonstrate the power of the SM model under consideration, we show that it is similarly possible to build into such a SM $\mathcal{S}$ both a *meter* to monitor energy usage as well as a *timer* which counts down the allotted time steps. However, as these components are not relevant for our subsequent results, we will not treat them as part of our baseline assumption. Given an upper bound e on the number of spikes, we can construct an energy counter neuron E, with threshold e, reset voltage e, and leakage factor m, with one synapse $s_{k,E} = (1, 1)$ for each $k \in N$, and $s_{E,\mathsf{acc}(\mathcal{S})} = (1, -\sum_j |w_{j,\mathsf{acc}(\mathcal{S})}|)$, $s_{E,\mathsf{rej}(\mathcal{S})} = (1, T_{\mathsf{rej}(\mathcal{S})} + \sum_j |w_{j,\mathsf{rej}(\mathcal{S})}|)$ where applicable. This ensures that if at some time step the permitted number

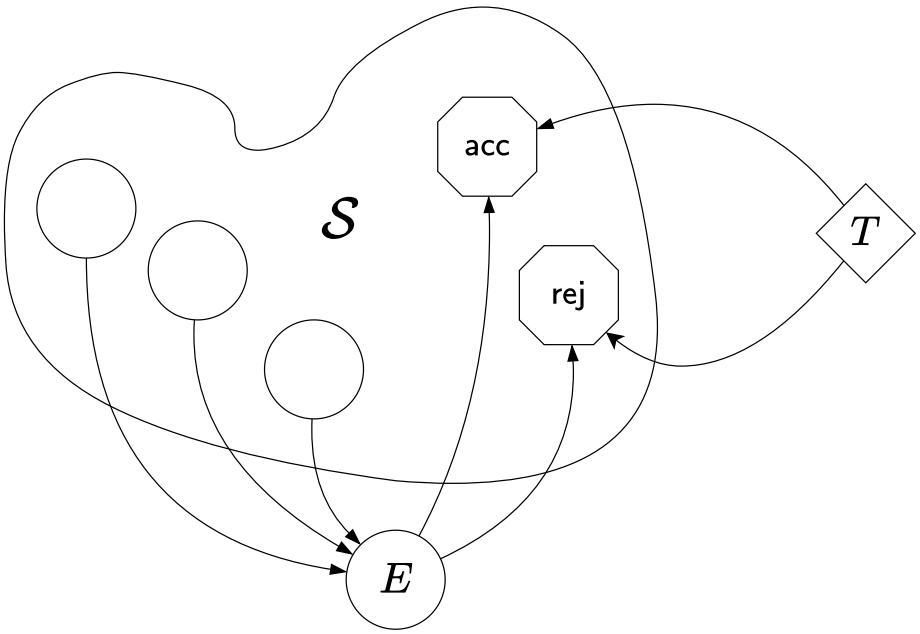


Figure 8. Adding a timer and a meter to an arbitrary spiking machine $\mathcal{S}$. Adapted with permission from [12]. Copyright © 2020 ACM.

of spikes has been reached without accepting or rejecting (which itself involves a spike from the corresponding neuron), from the next time step on the energy counter will inhibit the acceptance neuron and excite the rejection neuron if present. Along similar lines, given an upper bound t on the number of time steps, we can include a programmed timer neuron T with threshold 1, reset value 0, and leakage factor 1, which fires once at the first time step, along with synapses $s_{T,\mathsf{acc}(\mathcal{S})} = (t+1, -\sum_j |w_{j\mathsf{acc}(\mathcal{S})}|)$, and $s_{T,\mathsf{acc}(\mathcal{S})} = (t+1, T_{\mathsf{rej}(\mathcal{S})} + \sum_j |w_{j\mathsf{rej}(\mathcal{S})}|)$ where applicable (figure 8). Observe that these constructions add only two neurons, a proportionate number of synapses, and (in the presence of a rejection neuron) only a few additional spikes expended, hence the SM size and in particular its construction time remain the same asymptotically.

4. Structural complexity

In this section we explore some properties of the hierarchies of complexity classes, defined by choices for ($R_{\mathcal{M}}$ and) $R_{\mathcal{S}}$, for each of the three models of computation. We introduce complete problems and relate spiking-machine-based classes to traditional complexity classes.

4.1. Spike-input model

The computational power of the spike-input model is not trivial to establish. It appears that, due to the absence of explicit memory, there is no easy translation from a Turing machine to such a model; for starters, it is challenging to simulate the reading and writing on a working tape. In this sense, this model is likely to capture a weaker model of computation than a Turing machine equipped with a read-write head. Early work by Natschlager and Maass [17] suggests that such a model is able to represent (and learn from data) definite memory machines (themselves being specific instances of finite state machines). However, as the potential in each of the neurons can take in principle an unbounded number of values between the reset and threshold voltage, the SM can assume an infinite number of configurations and hence this model seems more powerful than a finite-state machine, unless we assume a fixed precision for the neuron and synapse parameters in which case the network effectively can be simulated by such a machine. For the purposes of this study, we restrict our model to a fixed, though arbitrary, level of precision. A more detailed analysis of how precision relates to the structural complexity of SMs could be a valuable extension, but it lies beyond the scope of this publication.

4.2. Uniform family of SMs

However, a uniform family of SMs (one machine for each input length) can easily be shown to simulate a family of Boolean circuits. It is well known that a family of Boolean circuits, with standard bounded fan-in basis, of size SIZE($t \log t$) can simulate a deterministic Turing machine taking time TIME(t) for its

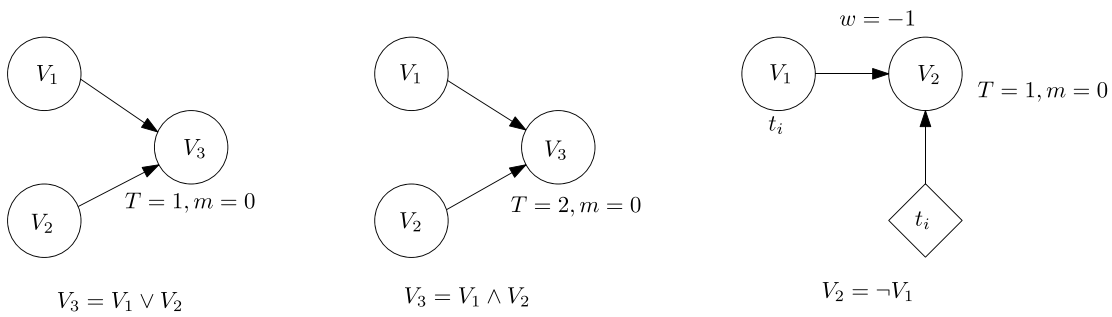


Figure 9. A spiking machine component for binary or, and, and unary negation.

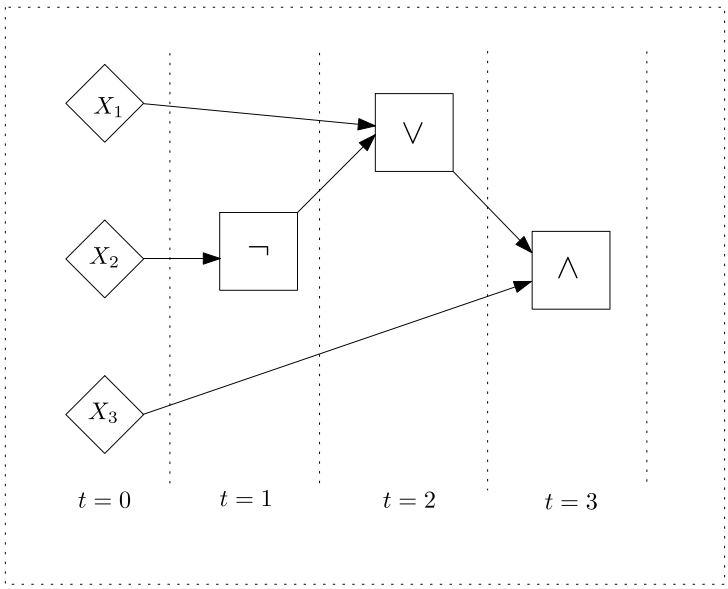


Figure 10. The spiking machine simulating $(x_1 \vee \neg x_2) \wedge x_3$.

computation, assuming that $t \geqslant |i|$, where $|i|$ is the size of the input on the tape [18, 25]. More in general this implies that any computation taking polynomial time on a deterministic Turing machine can be simulated by a (not necessarily uniform) family of circuits of polynomial size. In the remainder we show that every Boolean circuit (with binary gates $\vee$ and $\wedge$ and unary gate $\neg$) of size t can be simulated with a SM of size at most $t+1$ with time proportional to the depth of the circuit and energy proportional to its size. Figure 9 shows small SM components for each connective.

Importantly, the SM is *timed*: we assume that the input to input gates $x_1, \ldots, x_m$ is simulated by a spike wave to input neurons $X_1, \ldots, X_M$ consisting of a single spike (if the input to the corresponding gate is 1) or a silence (if the input is 0) at time $t = 0$. Spikes are delayed so that they arrive simultaneously at time t at a gate on depth t. The unary negation component uses a programmed neuron that fires at $t-1$ when the neuron simulates a gate at depth t; the input to the neuron effectively inhibits this spike, leading to the desired behaviour. Instead of one programmed neuron for each negation, we can also have a singleton neuron connecting to multiple negation simulators with the appropriate delays. In figure 10 we give an example for $(x_1 \vee \neg x_2) \wedge x_3$.

Note that this construction shows that a family $\mathcal{S}_{\mathcal{F}}(\mathsf{P}) = \bigcup_{|i|} \mathcal{S}_{|i|}(\mathsf{P})$ of polynomial-sized SMs (one machine for every input size $|i|$ to the Turing machine) can simulate any polynomially-bounded computation on such a machine, where the time needed for the simulation is upper bounded by the necessary depth of the circuit. This may be polynomial in the input size as well (unless $\mathsf{P} = \mathsf{NC}$; in this unlikely case the depth is bounded logarithmically in the input size). More specifically:

Corollary 1. $\mathsf{P} \subseteq \mathcal{S}^{\mathsf{P}}_{\mathcal{F}}(\mathsf{P})$

Note that we cannot directly prove the opposite direction $\mathcal{S}_{\mathcal{F}}(\mathsf{P}) \subseteq \mathsf{P}$ as the family of SMs is infinite. Assuming uniformity and resource-bounded realisation introduces the family $\mathcal{S}^{\mathsf{R}_{\mathcal{M}}}_{\mathcal{F}}(\mathsf{P})$ of polynomial-sized spiking networks that can be constructed by a Turing machine (on input $|i|$) using resources $\mathsf{R}_{\mathcal{M}}$; this model is likely too weak as the algorithm is uniform and limited to realise the mapping $1^{|i|} \rightarrow \mathcal{S}_{|i|}$. However, in the next section we will show that the pre-processing model gives us what is necessary to show equivalence to P.

4.3. Pre-processing model

We now take a closer look at the more generic class $\mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$, where the pre-processing is not constrained to just the mapping $1^{|i|} \rightarrow \mathcal{S}_{|i|}$. To begin with, it makes little sense to allow the pre-processing to operate with at least as much resources as the SM, since otherwise the execution of the SM can be simulated classically; this remark is illustrated in theorem 1 below. For this reason we typically choose $R_{\mathcal{M}}$ to be at most polynomial time and polynomial or even logarithmic space, corresponding to the classes P and L respectively.

Theorem 1. $\mathcal{M}(\mathsf{P}) \circ \mathcal{S}(R_{\mathcal{S}}) = \mathsf{P}$ *whenever* $R_{\mathcal{S}}$ *involves at most polynomial time constraints.*

Proof. As $\mathsf{P} \subseteq \mathcal{M}(\mathsf{P}) \circ \mathcal{S}(R_{\mathcal{S}})$ is obvious, we focus on proving the inclusion in the other direction. The crucial observation is that for a Turing machine with polynomial time constraints it is impossible to construct a larger than polynomial SM, which leads to a straightforward upper bound on the size of the SM. Recalling our earlier observation that the energy consumption of a SM is upper bounded in terms of (the product of) its size and time constraints, this implies that the SM constructed is effectively polynomially bounded (or worse) on all resources. Now it suffices to show that a deterministic Turing machine can simulate in polynomial time the execution of a SM of polynomial size for at most polynomial time. This can be done by explicitly iterating over the neurons for every time step, determining whether they fire and scheduling the transmission of this spike along the outgoing synapses, until the SM terminates or the time bounds are reached. By thus absorbing the decision procedure carried out by the SM into the classical polynomial-time computation carried out by the machine we arrive at the stated inclusion. □

One can easily see that, since P is low for itself, polynomially-bounded interactive computation does not change this result. This theorem serves as a reminder that SMs are no magical devices: while there is a potential efficiency gain, mostly in terms of energy usage relative to computations on traditional hardware (only), neuromorphic computations with at most polynomial time constraints cannot achieve more than their classical counterparts. It remains to be determined to what extent the classes $\mathcal{S}(R_{\mathcal{S}})$, $\mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$, and $\mathcal{M}(R_{\mathcal{M}})^{\mathcal{S}(R_{\mathcal{S}})}$ exhibit any hierarchical behaviour based on the constraints $R_{\mathcal{S}}$, in particular energy. We can however note that for well-defined resource constraints these classes are closed under operations such as intersection and complement, since SMs themselves are, so that decision procedures can be adjusted or combined at the level of the SM.

Observe that using different resource constraints $R_{\mathcal{M}}$ and $R_{\mathcal{S}}$ we can define a lattice of complexity classes for each computation model. It is therefore natural to consider the notions of reduction and hardness in this context, which is what we will do next. We round off this section with summarising the relation between the computational models, relative to fixed resources $R_{\mathcal{M}}$ and $R_{\mathcal{S}}$:

Theorem 2. $\mathcal{S}(R_{\mathcal{S}}) \subseteq \mathcal{S}^{R_{\mathcal{M}}}_{\mathcal{F}}(R_{\mathcal{S}}) \subseteq \mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}}) \subseteq \mathcal{M}(R_{\mathcal{M}})^{\mathcal{S}(R_{\mathcal{S}})}$

Proof. $\mathcal{S}(R_{\mathcal{S}}) \subseteq \mathcal{S}^{R_{\mathcal{M}}}_{\mathcal{F}}(R_{\mathcal{S}})$ follows as $\mathcal{S}(R_{\mathcal{S}})$, being a constant object, can be constructed independent of resource constraints. $\mathcal{S}^{R_{\mathcal{M}}}_{\mathcal{F}}(R_{\mathcal{S}}) \subseteq \mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$ holds als the former is a special case of the latter. And finally, $\mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$ is a constrained special case of $\mathcal{M}(R_{\mathcal{M}})^{\mathcal{S}(R_{\mathcal{S}})}_{1} \subseteq \mathcal{M}(R_{\mathcal{M}})^{\mathcal{S}(R_{\mathcal{S}})}$. □

4.4. Completeness for neuromorphic complexity classes

We start with observing that, assuming finite precision of the parameters, $\mathcal{S}(R_{\mathcal{S}}) \in \mathsf{DSPACE}(\mathcal{O}(1))$ as $\mathcal{S}(R_{\mathcal{S}})$ can be simulated with a deterministic finite state machine. Establishing completeness for problems inside $\mathsf{DSPACE}(\mathcal{O}(1))$ requires very specific reductions and we consider this out-of-scope for this paper.

In order to arrive at a canonical complete problem for the class $\mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$, it makes sense to consider the analogy with other models of computation, where one asks whether the given procedure (be it machine, circuit or otherwise) accepts the provided input. Since even for the class $\mathcal{M}(R_{\mathcal{M}}) \circ \mathcal{S}(R_{\mathcal{S}})$ it is not a SM but a Turing machine which controls how the input is handled, the resulting candidate for a complete problem for this class will involve the latter and not the former. This means that to distinguish this problem from its classical equivalent we must include the promise that the Turing

machine is indeed of the kind associated with the class $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$, in that it generates an $R_\mathcal{S}$-bounded SM using resources $R_\mathcal{M}$[6]. In other words, we claim that the following problem is complete under polynomial-time reductions for the promise version of the class $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$.

$\mathcal{S}(R_\mathcal{M}, R_\mathcal{S})$-Halting
Instance: Turing machine $\mathcal{M}$ along with input string i.
Promise: $\mathcal{M}$ is an $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$-machine.
Question: Does $\mathcal{M}$ accept i?

Theorem 3. *$\mathcal{S}(R_\mathcal{M}, R_\mathcal{S})$-Halting is complete under polynomial-time reductions for the promise version of $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$.*

Proof. Membership of this problem is established as follows: with a universal $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$ machine one can take the machine $\mathcal{M}$ and simulate it on the input i. If the machine $\mathcal{M}$ is indeed an $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$ machine as per the promise, then this simulation will succeed within the permitted resource bounds and we can simply return the answer given by $\mathcal{M}$. In case the promise fails to hold, we only need to ensure that the (unsuccessful) simulation does not exceed the resource bounds, since it is otherwise irrelevant which response is ultimately given. For the hardness of this problem, we observe that every problem in $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$ is by definition solvable by an $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$-machine, hence the straightforward reduction from any such problem to $\mathcal{S}(R_\mathcal{M}, R_\mathcal{S})$-Halting consists of taking the input i and passing it along to $\mathcal{S}(R_\mathcal{M}, R_\mathcal{S})$-Halting accompanied by a particular $\mathcal{M}(R_\mathcal{M}) \circ \mathcal{S}(R_\mathcal{S})$-machine which decides the problem. □

For interactive computation (viz. using a $\mathcal{M}(R_\mathcal{M})^{\mathcal{S}(R_\mathcal{S})}$ machine), we can similarly as in theorem 3 show that $\mathcal{S}(R_\mathcal{M}, R_\mathcal{S})$-Halting is complete for the promise version of $\mathcal{M}(R_\mathcal{M})^{\mathcal{S}(R_\mathcal{S})}$. The argument is identical as in theorem 3 for both membership and hardness.

Corollary 2. *$\mathcal{S}(R_\mathcal{M}, R_\mathcal{S})$-Halting is complete under polynomial-time reductions for the promise version of $\mathcal{M}(R_\mathcal{M})^{\mathcal{S}(R_\mathcal{S})}$.*

However, for particular assignments of $R_\mathcal{M}$ we can actually replace the Turing machine with a SM and still end up with a complete (promise) problem. We will illustrate this construction for $R_\mathcal{M}$ corresponding to L, using logspace reductions as part of the hardness proof.

$\mathcal{S}(\mathsf{L}, R_\mathcal{S})$-Network Halting
Instance: Machine $\mathcal{S}$ along with input string i.
Promise: $\mathcal{S}$ terminates within resource bounds $R_\mathcal{S}$ expressed as a function of $|i|$.
Question: Does $\mathcal{S}$ accept?

Theorem 4. *$\mathcal{S}(\mathsf{L}, R_\mathcal{S})$-Network Halting is complete under logspace reductions for the promise version of $\mathcal{M}(\mathsf{L} \circ \mathcal{S}(R_\mathcal{S}))$.*

Proof. Membership follows from the observation that a Turing machine can simply ignore the input string $|i|$ and return $\mathcal{S}$, which by definition is an $R_\mathcal{S}$-constrained SM which accepts precisely whenever $\mathcal{S}$ does. To prove hardness we reduce $\mathcal{S}(R_\mathcal{M}, R_\mathcal{S})$-Halting to $\mathcal{S}(\mathsf{L}, R_\mathcal{S})$-Network Halting using logspace reductions. Let $(\mathcal{M}, i)$ be an instance of the former. By simulating the application of $\mathcal{M}$ on i and replacing it with the resulting machine $\mathcal{S}_i$ (which by the promise can be done using logarithmic space), we obtain an instance $(\mathcal{S}_i, i)$ of $\mathcal{S}(\mathsf{L}, R_\mathcal{S})$-Network Halting where the promise for $\mathcal{S}_i$ is inherited from that for $\mathcal{M}$ and the decision of $\mathcal{S}_i$ is that of $\mathcal{M}$ on i by definition[7]. □

This completeness result shows that for those choices of $R_\mathcal{M}$ that we were likely to consider anyways (cf the remark at the beginning of this section) we are justified in taking SMs, and by extension SNNs, as computationally primitive in a sense relevant for our treatment.

5. Conclusion

In this paper we proposed a machine model to assess the potential of neuromorphic architectures with energy as a vital resource in addition to time and space. We introduced a hierarchy of computational

[6] This construction is similar to the one required for the class BPP associated with probabilistic Turing machines, where we cannot verify and therefore must take for granted that the machine always satisfies the bounded probability requirement.
[7] An earlier version noted that this argument works for linear time as well, but on closer inspection this may fail to be the case since having to simulate $\mathcal{M}$ on i by a universal Turing machine will likely add a logarithmic overhead to the linear time of $\mathcal{M}$, which means that the reduction itself would not be linear time.

complexity classes relative to these resources and provided some first structural results and canonical complete problems for these classes. It is important to note that these completeness results are largely independent of the exact definition of a SNN model that is used, for example, whether neuron potentials can be negative or whether there is a refractory period of the neurons. It is still an open question whether variations therein may have consequences for the complexity level.

We already hinted at other future structural complexity work, most urgently on the role of energy as explicit dimension of analysis. Particular challenges include identifying general examples of an asymptotic trade-off between time and energy, and determining whether there exists an energy analogue of the time complexity hierarchy. Of further importance is a notion of amortisation of resources that is crucial when considering *local changes* to the SM, such as adapting the weights when learning, or when using a SM with a set of spike trains rather than recreating everything from scratch.

In addition to furthering our understanding of the structural aspects, the more applied work of populating the associated classes with natural problems using neuromorphic algorithms, along with deriving concrete hardness results for these problems, should be high on the agenda for the neuromorphic research community.

Data availability statement

No new data were created or analysed in this study.

Acknowledgments

The authors are grateful to the reviewers of an earlier conference version of this paper [12] for their useful feedback.

ORCID iDs

Johan Kwisthout 0000-0003-4383-7786
Arne Diehl 0000-0001-9702-1083

References

[1] Akopyan F *et al* 2015 Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **34** 1537–57
[2] Backus J 1978 Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs *Commun. ACM* **21** 613–41
[3] Cross T 2016 *After Moore's Law* (Technology Quarterly by The Economist)
[4] Davies M *et al* 2018 Loihi: a neuromorphic manycore processor with on-chip learning *IEEE Micro* **38** 82–99
[5] Davies M, Wild A, Orchard G, Sandamirskaya Y, Guerra G A F, Joshi P, Plank P and Risbud S R 2021 Advancing neuromorphic computing with Loihi: A survey of results and outlook *Proc. IEEE* **109** 911–34
[6] Furber S and Temple S 2007 Neural systems engineering *J. R. Soc. Interface* **4** 193–206
[7] Graves A, Wayne G and Danihelka I 2014 Neural Turing machines (arXiv:1410.5401)
[8] Hartmanis J and Stearns R 1965 On the computational complexity of algorithms *Trans. Am. Math. Soc.* **117** 285–306
[9] Haxhimusa Y, van Rooij I, Varma S and Wareham H T 2014 Resource-bounded problem solving (dagstuhl seminar 14341) *Dagstuhl Reports* vol 4
[10] Hennessy J L and Patterson D A 2011 *Computer Architecture: a Quantitative Approach* 5th edn (Morgan Kaufmann)
[11] Indiveri G *et al* 2011 Neuromorphic silicon neuron circuits *Front. Neurosci.* **5** 73
[12] Kwisthout J and Donselaar N 2020 On the computational power and complexity of spiking neural nets *Proc. the Neuro-Inspired Computational Elements Workshop (ICPS Series)* vol **4** pp 1–7
[13] Maass W 1996 Lower bounds for the computational power of networks of spiking neurons *Neural Comput.* **8** 1–40
[14] Maass W 2014 Noise as a resource for computation and learning in networks of spiking neurons *Proc. IEEE* **102** 860–80
[15] Mead C 1990 Neuromorphic electronic systems *Proc. IEEE* **78** 1629–36
[16] Moore G E 1975 Progress in digital integrated electronics *Proc. 1975 Int. Electron Devices Meeting* ed Holton W
[17] Natschläger T and Maass W 2002 Spiking neurons and the induction of finite state machines *Theor. Comput. Sci.* **287** 251–65
[18] Pippenger N and Fischer M 1979 Relations among complexity measures *J. ACM* **26** 361–81
[19] Potok T, Schuman C, Patton R, Hylton T, Li H and Pino R 2016 Neuromorphic computing: Architectures, models, and applications. A Beyond-CMOS approach to future computing. *DoE Workshop Report. Technical Report (Oak Ridge National Laboratory)*.
[20] Radosavljevic M and Kavalieros J 2022 Taking moore's law to new heights: When transistors can't get any smaller, the only direction is up *IEEE Spectr.* **59** 32–37
[21] Severa W, Parekh O, Carlson K, James C and Aimone J 2016 Spiking network algorithms for scientific computing *Proc. IEEE Int. Conf. on Rebooting Computing (ICRC)*
[22] Shalf J M and Leland R 2015 Computing beyond Moore's Law *Computer* **48** 14–23
[23] Uchizawa K, Nishizeki T and Takimoto E 2009 Energy complexity and depth of threshold circuits *Proc. 17th Int. Conf. on Fundamentals of Computation Theory*, ed M Kutylowski, W Charatonik and M Gebala pp 335–45
[24] Van Schoot J 2023 The moore's law machine: The next trick to tinier transistors is high-numerical-aperture euv lithography *IEEE Spectr.* **60** 44–48
[25] Vollmer H 2010 *Introduction to Circuit Complexity: A Uniform Approach* 1st edn (Springer)