Turing Complete Neural Computation Based on Synaptic Plasticity

Jérémie Cabessa^{1,2,*}

 Laboratory of Mathematical Economics and Applied Microeconomics (LEMMA), University Paris 2 – Panthéon-Assas, 75005 Paris, France
 Institute of Computer Science, Czech Academy of Sciences
 18207 Prague 8, Czech Republic

* E-mail: jeremie.cabessa@u-paris2.fr

Abstract

In neural computation, the essential information is generally encoded into the neurons 12 via their spiking configurations, activation values or (attractor) dynamics. The synapses 13 and their associated plasticity mechanisms are, by contrast, mainly used to process 14 this information and implement the crucial learning features. Here, we propose a novel 15 Turing complete paradigm of neural computation where the essential information is 16 encoded into discrete synaptic states, and the updating of this information achieved 17 via synaptic plasticity mechanisms. More specifically, we prove that any 2-counter 18 machine—and hence any Turing machine—can be simulated by a rational-weighted 19 recurrent neural network employing spike-timing-dependent plasticity (STDP) rules. 20 The computational states and counter values of the machine are encoded into discrete 21 synaptic strengths. The transitions between those synaptic weights are then achieved 22 via STDP. These considerations show that a Turing complete synaptic-based paradigm 23 of neural computation is theoretically possible and potentially exploitable. They support 24 the idea that synapses are not only crucially involved in information processing and 25 learning features, but also in the encoding of essential information. This approach 26 represents a paradigm shift in the field of neural computation. 27

Introduction

How does the brain compute? How do biological neural networks encode and process information? What are the computational capabilities of neural networks? Can neural networks implement abstract models of computation? Understanding the computational and dynamical capabilities of neural systems is a crucial issue with significant implications in computational and system neuroscience, artificial intelligence, machine learning, bioinspired computing, robotics, but also theoretical computer science and philosophy.

10

11

In 1943, McCulloch and Pitts proposed the concept of an *artificial neural network* (ANN) as an interconnection of neuron-like logical units [1]. This computational model significantly contributed to the development of two research directions: (1) Neural Computation, which studies the processing and coding of information as well as as the computational capabilities of various kinds of artificial and biological neural models; (2) Machine Learning, which concerns the development and utilization of neural network algorithms in Artificial Intelligence (AI).

The proposed study lies within the first of these two approaches. In this context, the 42 computational capabilities of diverse kinds of neural networks have been shown to range 43 from the finite automaton degree [1-3] up to the Turing [4] or even to the super-Turing 44 levels [5-7] (see [8] for a survey of complexity theoretic results). In short, Boolean 45 recurrent neural networks are computationally equivalent fo finite state automata; analog 46 neural networks with rational synaptic weights are Turing complete; and analog neural 47 nets with real synaptic weights as well as evolving neural nets are capable of super-Turing 48 capabilities (cf. Table 1). These theoretical results have later been improved, motivated 49 by the possibility to implement finite state machines on electronic hardwares (see for 50 instance [9-13]). Around the same time, the computational power of *spiking neural* 51 *networks* (instead of sigmoidal ones) has also been extensively studied [14, 15]. More 52 recently, the study of *P* systems—parallel abstract models of computation inspired 53 from the membrane structure of biological cells—has become a highly active field of 54 research [16–18]. 55

	BOOLEAN	Sigmoid				
	STATIC	STATIC	BI-VALUED EVOLVING	Evolving		
Q	FSA	TM	TM/poly(A)	TM/poly(A)		
	REG	Р	P/poly	P/poly		
R	FSA	TM/poly(A)	$\mathrm{TM}/\mathrm{poly}(\mathrm{A})$	$\mathrm{TM}/\mathrm{poly}(\mathrm{A})$		
	REG	P/poly	P/poly	P/poly		

Table 1. Computational power of various models of recurrent neural networks. FSA, TM and TM/poly(A) stand for finite state automata, Turing machines and Turing machines with polynomial advice (which are super-Turing), respectively. **REG**, **P** and **P/poly** are the complexity classes decided in polynomial time by these three models of computation. The results in the case of classical computation can be found in [1-7, 19-24]. Results in alternative infinite computational frameworks have also been obtained [25–35].

Concerning the second direction, Turing himself brilliantly anticipated the two concepts of *learning* and *training* that would later become central to machine learning [36]. 57 These ideas were realized with the introduction of the *perceptron* [37], which gave 58 rise to the algorithmic conception of learning [38-40]. Despite some early limitation 59 issues [41], the development of artificial neural networks has steadily progressed since 60 then. Nowadays, artificial neural networks represent a most powerful class of algorithms 61 in machine learning, thanks to their highly efficient training capabilities. In particular, 62 deep learning methods—multilayer neural networks that can learn in supervised and/or 63 unsupervised manners—have achieved impressive results in numerous different areas 64 (see [42] for a brilliant survey and the references therein). 65

These approaches share a common and certainly sensible conception of neural com-66 putation that could be qualified as a neuron-based computational framework. According 67 to this conception, the essential information is encoded into the neurons, via their 68 spiking configurations, activation values or (attractor) dynamics. The synapses and 69 their associated plasticity mechanisms are, by contrast, essentially used to process this information and implement the crucial learning features. For instance, in the simulation 71 of abstract machines by neural networks, the computational states of the machines are 72 encoded into activation values or spiking patterns of neurons [8]. Similarly, in most if not 73 all deep learning algorithms, the input, output and intermediate information is encoded 74 into activation values of input, output and hidden (layers of) neurons, respectively [42]. 75 But what if the synaptic states would also play a crucial role in the encoding of informa-76 tion? What if the role of the synapses would not only be confined to the processing of 77 information and learning processes, as crucial as these features might be? In short, what 78 about a synaptic-based computational framework? 79

In biology, the various mechanisms of *sunaptic plasticity* provide "the basis for 80 most models of learning, memory and development in neural circuits" [43]. Spike-81 timing-dependent plasticity (STDP) refers to the biological Hebbian-like learning process 82 according to which the synapses' strengths are adjusted based on the relative timings 83 of the presynaptic and postsynaptic spikes [38,44,45]. It is widely believed that STDP 84 "underlies several learning and information storage processes in the brain, as well as the 85 development and refinement of neuronal circuits during brain development" (see [46] 86 and the references therein). In particular, fundamental neuronal structures like synfire 87 chains [47–51] (pools of successive layers of neurons strongly connected from one stratum 88 to the next by excitatory connections), synfire rings [52] (looping synfire chains) and 89 polychronous groups [53] (groups of neurons capable of generating time-locked repro-90 ducible spike-timing patterns), have all been observed to emerge in self-organizing neural 91 networks employing various STDP mechanisms [52–55]. On another level, regarding 92 STDP mechnisms, it has been shown that synapses might change their strengths by 93 jumping between discrete mechanistic states, rather than by simply moving up and down 94 in a continuum of efficacy [56]. 95

Based on these considerations, we propose a novel Turing complete synaptic-based 96 paradigm of neural computation. In this framework, the essential information is encoded 97 into discrete synaptic states instead of neuronal spiking patterns, activation values or 98 dynamics. The updating of this information is then achieved via synaptic plasticity mech-99 anisms. More specifically, we prove that any 2-counter machine—and hence any Turing 100 machine—can be simulated by a rational-weighted recurrent neural network subjected 101 to STDP. The computational states and counter values of the machine are encoded 102 into discrete synaptic strengths. The transitions between those synaptic weights are 103 achieved via STDP. These results show that a Turing complete synaptic-based paradigm 104 of computation is theoretically possible and potentially exploitable. They support the 105 idea that synapses are not only crucially involved in information processing and learning 106 features, but also in the encoding of essential information in the brain. This approach 107 represents a paradigm shift in the field of neural computation.

The possible impacts of these results are both practical and theoretical. In the 109 field of neuromorphic computing, our synaptic-based paradigm of neural computation 110 might lead to the realization of novel analog neuronal computers implemented on VLSI 111 technologies. Regarding AI, our approach might lead to the development of new machine 112 learning algorithms. On a conceptual level, the study of neuro-inspired paradigms of 113 abstract computation might improve the understanding of both biological and artificial 114 intelligences. These aspects are discussed in the conclusion. 115

Materials and methods

Recurrent Neural Networks

A rational-weighted recurrent neural network (RNN) \mathcal{N} consists of a synchronous network of neurons connected together in a general architecture. The network is composed of Minput neurons $(u_i)_{i=1}^M$ and N internal neurons $(x_i)_{i=1}^N$. The dynamics of network \mathcal{N} is computed as follows: given the activation values of the input neurons $(u_j(t))_{j=1}^M$ and internal neurons $(x_j(t))_{j=1}^N$ at time step t, the activation values of the internal neurons $(x_i(t+1))_{i=1}^N$ at time step t+1 are given by the following equations:

$$x_i(t+1) = f\left(\sum_{j=1}^N a_{ij}(t) \cdot x_j(t) + \sum_{j=1}^M b_{ij}(t) \cdot u_j(t) + c_i(t)\right), \text{ for } i = 1, \dots, N$$
(1)

where $a_{ij}(t), b_{ij}(t) \in \mathbb{Q}$ are the rational weights of the synaptic connections from x_j to x_i and u_j to x_i at time t, respectively, $c_i(t) \in \mathbb{Q}$ is the rational bias of cell x_i at time t, and f is either the hard-threshold activation function θ or the linear sigmoid activation function σ defined by

$$\theta(x) = \begin{cases} 0 & \text{if } x < 1 \\ 1 & \text{if } x \ge 1 \end{cases} \qquad \sigma(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \le x \le 1 \\ 1 & \text{if } x > 1. \end{cases}$$

A neuron is called *Boolean* or *analog* depending on whether its activation value is 124 computed by the function θ or σ , respectively. Input neurons $(u_i)_{i=1}^M$ are all Boolean. 125 The input state and intermediate of N at time t are the materia.

The *input state* and *internal state* of \mathcal{N} at time t are the vectors

$$\mathbf{u}(t) = (u_1(t), \dots, u_M(t))^T \in \mathbb{B}^M$$
$$\mathbf{x}(t) = (x_1(t), \dots, x_N(t))^T \in \mathbb{Q}^N$$

For any Boolean input stream $u = \mathbf{u}(0)\mathbf{u}(1)\mathbf{u}(2)\cdots$, the computation of \mathcal{N} over input u is the sequence of internal states $\mathcal{N}(u) = \mathbf{x}(0)\mathbf{x}(1)\mathbf{x}(2)\ldots$, where $\mathbf{x}(0) = \mathbf{0}$ and the components of $\mathbf{x}(t)$ are given by Eq (1), for each t > 0. A simple recurrent neural network is illustrated in Fig 1.

108

116



Fig 1. A recurrent neural network. The network contains two input cells u_1, u_2 and three internal cells x_1, x_2, x_3 . Excitatory and inhibitory connections are represented as red and blue arrows, respectively. Cells u_1, u_2, x_1, x_2 are Boolean (activation function θ) whereas x_3 is analog (activation function σ). Over the Boolean input $u = (1, 1)^T (1, 0)^T (0, 1)^T$, the network's computation is $\mathcal{N}(u) = (0, 0, 0)^T (0, 1, 0)^T (1, 0, 0)^T (0, 1, 1)^T (0, 0, 0.25)^T (0, 0, 0.625)^T \cdots$.

A spike-timing dependent plasticity (STDP) rule modifies the synaptic weights $a_{ij}(t)$ according to the spiking patterns of the presynaptic and postsynaptic cells x_j and x_i [45]. Here, we consider two STDP rules. The first one is a classical generalized Hebbian rule [38]. It allows the synaptic weights to vary across finitely many values comprised between two bounds a_{min} and a_{max} ($0 < a_{min} < a_{max} < 1$). The rule is given as follows: 134

$$a_{ij}(t+1) = \begin{cases} a_{min}, \text{ if } R(t+1) < a_{min} \\ a_{max}, \text{ if } R(t+1) > a_{max} \\ R(t+1), \text{ otherwise} \end{cases}$$
(2)

$$R(t+1) := a_{ij}(t) + \eta \cdot \left(\lfloor x_i(t+1) \rfloor \cdot \lfloor x_j(t) \rfloor - \lfloor x_i(t) \rfloor \cdot \lfloor x_j(t+1) \rfloor \right)$$

where |x| denotes the floor of x (the greatest integer less than or equal to x) and 135 $\eta > 0$ is the *learning rate*. Accordingly, the synaptic weight $a_{ij}(t)$ is incremented 136 (resp. decremented) by η at time t+1 if the presynaptic cell x_i spikes 1 time step before 137 (resp. after) the postsynaptic cell x_i . The floor function is used to truncate the activation 138 values of analog neurons to their integer part, if needed. The synaptic weights enabled 139 by this rule is illustrated in Fig 2. In the sequel, this STDP rule will be used to encode 140 the transitions between the finitely many computational states of the machine to be 141 simulated. 142

The second rule is an adaptation to our context of a classical Hebbian rule. It allows the synaptic weights to vary across the infinitely many values of the sequence 144

$$\beta = \left(1 - \frac{1}{2^k}\right)_{k=0}^{\infty} = (0.0, 0.5, 0.75, 0.875, 0.9375, \dots)$$

The rule is given as follows:

$$a_{ij}(t+1) = \begin{cases} a_{ij}(t) + \frac{1}{2}(1 - a_{ij}(t)) & \text{if } x_i(t+1) \cdot x_j(t) - x_i(t) \cdot x_j(t+1) = 1\\ \max(a_{ij}(t) - (1 - a_{ij}(t)), 0) & \text{if } x_i(t+1) \cdot x_j(t) - x_i(t) \cdot x_j(t+1) = -1 \\ a_{ij}(t) & \text{if } x_i(t+1) \cdot x_j(t) - x_i(t) \cdot x_j(t+1) = 0 \end{cases}$$
(3)

As for the previous one, the synaptic weight $a_{ij}(t)$ is incremented (resp. decremented) at 147 time t+1 if the presynaptic cell x_i spikes 1 time step before (resp. after) the postsynaptic 148 cell x_i . But in this case, the synaptic weight varies across the infinitely many successive 149 values of the sequence β . For instance, if $a_{ij}(t) = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 0.875$ is incremented 150 (resp. decremented) by the STDP rule, then $a_{ij}(t+1) = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = 0.9375$ 151 (resp. $a_{ij}(t+1) = \frac{1}{2} + \frac{1}{4} = 0.75$). Here, the floor functions are removed, since this rule 152 will only be applied to synaptic connections between Boolean neurons. The synaptic 153 weights enabled by this rule is illustrated in Fig 2. In the sequel, this STDP rule will be 154 used to encode the variations among the infinitely many possible counter values of the 155 machine to be simulated. 156



Fig 2. Synaptic weights enabled by to the two STDP rules. The red curve displays the finitely many possible synaptic weights enabled by the first STDP rule (Eq (2)), where $a_{min} = 0.1$, $a_{max} = 1$ and $\eta = 0.1$. These are the successive values of the sequence (0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0). The blue curve displays the first elements of the infinitely many synaptic weights enabled by the second STDP rule (Eq (3)). These are the successive values of the sequence $\beta = (0.0, 0.5, 0.75, 0.875, 0.9375, \ldots)$.

Finite State Automata

A deterministic finite state automaton (FSA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where:

- $Q = \{q_0, \ldots, q_{n-1}\}$ is a finite set of *computational states*;
- Σ is an alphabet of *input symbols*;
- $\delta: Q \times \Sigma \to Q$ is a transition function;
- $q_0 \in Q$ is the *initial state*;

157

159

160

162

145

• $F \subseteq Q$ is the set of *final states*.

Each transition $\delta(q, a) = q'$ signifies that if the automaton is state $q \in Q$ and reads input symbol $a \in \Sigma$, then it will move to state $q' \in Q$. For any input $w = a_0 a_1 \cdots a_p \in \Sigma^*$, the *computation* of \mathcal{A} over w is the finite sequence

$$\mathcal{A}(w) = \left((q_{i_0}, a_0, q_{i_1}), (q_{i_1}, a_1, q_{i_2}), \dots, (q_{i_p}, a_p, q_{i_{p+1}}) \right)$$

such that $q_{i_0} = q_0$ and $\delta(q_{i_k}, a_k) = q_{i_{k+1}}$, for all $k = 0, \ldots, p$. Such a computation is usually denoted as

$$\mathcal{A}(w): q_0 \xrightarrow{a_0} q_{i_1} \xrightarrow{a_1} q_{i_2} \cdots q_{i_p} \xrightarrow{a_p} q_{i_{p+1}}.$$

Input w is said to be *accepted* (resp. *rejected*) by automaton \mathcal{A} if the last state $q_{i_{p+1}}$ of computation $\mathcal{A}(w)$ belongs (resp. does not belong) to the set of final states F. The set of all inputs accepted by \mathcal{A} is the *language* recognized by \mathcal{A} . Finite state automata recognize the class of *regular* languages. A finite state automaton is generally represented as a directed graph, as illustrated in Fig 3.



Fig 3. A finite state automaton. The nodes and edges of the graph represent the states and transitions of the automaton, respectively. Initial and final states are represented with an incoming arrow and a double-circle, respectively. An edge from state q to q' labelled by arepresents the transition relation $\delta(q, a) = q'$. This automaton recognizes the language $\{0^m 1^n : m, n > 0\}$, i.e., the sequences of bits beginning with a strictly positive number of 0's and ending with a strictly positive number of 1's.

Counter Machines

A *counter machine* is a finite state automaton provided with additional counters [57]. ¹⁷⁰ The counters are used to store integers. They can be pushed (incremented by 1), popped ¹⁷¹ (decremented by 1) or kept unchanged. At each step, the machine determines its next ¹⁷² computational state according to its current input symbol, computational state and ¹⁷³ counters' states, i.e., if counters are zero or non-zero. ¹⁷⁴

Formally, a deterministic k-counter machine (CM) is a tuple $C_k = (Q, \Sigma, C, O, \delta, q_0, F)$, ¹⁷⁵ where:

- $Q = \{q_0, \ldots, q_{n-1}\}$ is a finite set of *computational states*;
- Σ is an alphabet of *input symbols* not containing the *empty symbol* ϵ (recall that the empty symbol satisfies $\epsilon w = w\epsilon = w$, for any string $w \in \Sigma^*$); 179

7/39

- C = {⊥, ⊤} is the set of *counter states*, where ⊥, ⊤ represent the *zero* and *non-zero* 180 states, respectively;
 N is the set of *counter values* (doesn't need to be hold in the tuple C_k);
- $O = \{push, pop, -\}$ is the set of *counter operations*;
- $\delta: Q \times \Sigma \cup \{\epsilon\} \times C^k \to Q \times O^k$ is a (partial) transition function; 184
- $q_0 \in Q$ is the *initial state*; 185
- $F \subseteq Q$ is the set of *final states*.

The value and state of counter j are denoted by c_j and \bar{c}_j , respectively, for $j = 1, \ldots, k$. (In the sequel, certain cells will also be denoted by c_j 's and \bar{c}_j 's. The use of same notations to designate counter's values or states and specific cells will be clear from the context.) The "bar function" $(c \mapsto \bar{c})$ retrieves the counter's state from its value. It is naturally defined by $\bar{c}_j = \bot$ if $c_j = 0$ and $\bar{c}_j = \top$ if $c_j > 0$. The value of counter j after application of operation $o_j \in O$ is denoted by $o_j(c_j)$. The counter operations influence their values in the following natural way:

- If $o_j = push$, then $o_j(c_j) = c_j + 1$;
- If $o_j = pop$, then $o_j(c_j) = \max(c_j 1, 0)$;
- If $o_j = -$, then $o_j(c_j) = c_j$.

Each transition $\delta(q, a, \bar{c}_1, \dots, \bar{c}_k) = (q', o_1, \dots, o_k)$ signifies that if the machine is 197 state $q \in Q$, reads the regular or empty input symbol $a \in \Sigma \cup \{\epsilon\}$ and has its k counter 198 being in states $\bar{c}_1, \ldots, \bar{c}_k \in C$, then it will move to state $q' \in Q$ and perform the k counter 199 operations $o_1, \ldots, o_k \in O$. Depending on whether $a \in \Sigma$ or $a = \epsilon$, the corresponding 200 transition is called a *regular transition* or an ϵ -transition, respectively. We assume that 201 δ is a partial (rather than a total) function. Importantly, the determinism is expressed 202 by the fact that the machine can never face a choice between either a regular or an 203 ϵ -transition, i.e., for any $q \in Q$, any $a \in \Sigma$ and any $\bar{c}_1, \ldots, \bar{c}_k \in C$, if $\delta(q, a, \bar{c}_1, \ldots, \bar{c}_k)$ is 204 defined, then $\delta(q, \epsilon, \bar{c}_1, \ldots, \bar{c}_k)$ is undefined [57]. 205

For any input $w = a_0 a_1 \cdots a_p \in \Sigma^*$, the *computation* of a k-counter machine \mathcal{C}_k over input w can be described as follows. For each successive input symbol $a_i \in \Sigma$, before trying to process a_i , the machine first tests if an ϵ -transition is possible. If this is the case, it performs this transition. Otherwise, it tests if the regular transition associated with a_i is possible, and if so, performs it. The deterministic condition ensures that a regular and an ϵ -transition are never possible at the same time. When no more transition can be performed, the machine stops.

For any input $w = a_0 a_1 \cdots a_p \in \Sigma^*$, the *computation* of \mathcal{C}_k over w is the unique finite or infinite sequence of states, symbols and counter values encountered by \mathcal{C}_k while reading the successive bits of w possibly interspersed with ϵ symbols. The formal definition involves the following notions. An instantaneous description of C_k is a tuple $(q, w, c_1, \ldots, c_k) \in Q \times \Sigma^* \times \mathbb{N}^k$. For any empty or non-empty symbol $a' \in \Sigma \cup \{\epsilon\}$ and any $w \in \Sigma^*$, the relation " \vdash " over the set of instantaneous descriptions is defined as follows:

$$(q, a'w, c_1, \dots, c_k) \vdash (q', w, c'_1, \dots, c'_k)$$
 iff $\delta(q, a', \bar{c}_1, \dots, \bar{c}_k) = (q', o_1, \dots, o_k)$
and $c'_1 = o_1(c_1), \dots, c'_k = o_k(c_k)$

Note that depending on whether $a' = \epsilon$ or $a' \in \Sigma$, the relation " \vdash " is determined by an ϵ -transition or a regular transition, respectively. (Note also that when $a' = \epsilon$, one has $a'w = \epsilon w = w$, and in this case, the relation " \vdash " keeps w unchanged.)

For any input $w = a_0 a_1 \cdots a_p \in \Sigma^*$, the determinism of \mathcal{C}_k ensures that there is a unique finite or infinite sequence of instantaneous descriptions

$$((q_{n_i}, w_i, c_{1,i}, \dots, c_{k,i}))_{i=0}^l, \ l \in \mathbb{N} \cup \{\infty\}$$

such that $(q_{n_0}, w_0, c_{1,0}, \ldots, c_{k,0}) = (q_0, w, 0, \ldots, 0)$ is the initial instantaneous description, and $(q_{n_i}, w_i, c_{1,i}, \ldots, c_{k,i}) \vdash (q_{n_{i+1}}, w_{i+1}, c_{1,i+1}, \ldots, c_{k,i+1})$, for all i < l. Then, the computation of \mathcal{C}_k over w, denoted by $\mathcal{C}_k(w)$, is the finite or infinite sequence defined by 225

$$\mathcal{C}_{k}(w) = \left(\left(q_{n_{i}}, a'_{i}, c_{1,i}, \dots, c_{k,i} \right) \right)_{i=0}^{l}, \quad l \in \mathbb{N} \cup \{ \infty \}$$
(4)

where $a'_i = \epsilon$ if $w_i = w_{i+1}$ (case of an ϵ -transition), and a'_i is the first bit of w_i otherwise 226 (case of a regular transition), for all i < l. Note that the computation over w can take 227 longer than |w| = p + 1 steps, even be infinite, due to the use of ϵ -transitions. The input 228 $w \in \Sigma^*$ is said to be *accepted* by \mathcal{C}_k if the computation of the machine over w is finite, 229 consumes all letters of w and stops in a state of F, i.e., if $a'_l = \epsilon$ and $q_{n_l} \in F$. It is 230 *rejected* otherwise. The set of all inputs accepted by \mathcal{C}_k is the *language* recognized by \mathcal{C}_k . 231

It is known that 1-counter machines are strictly more powerful than finite state 232 automata, and k-counter machines are computationally equivalent to Turing machines 233 (Turing complete), for any $k \ge 2$ [57]. However, the class of k-counter machines that do not make use of ϵ -transitions is not Turing complete. For this reason, the simulation 236 of ϵ -transitions by our neural networks will be essential towards the achievement of 236 Turing-completeness. 237

A k-counter machine can also be represented as a directed graph, as illustrated ²³⁸ in Fig 4. The 2-counter machine of Fig 4 recognizes a language that is recursively ²³⁹ enumerable but not context-free, i.e., it can be recognized by some Turing machine, yet ²⁴⁰ by no pushdown automaton. Note that this 2-counter machine contains ϵ -transitions. ²⁴¹

Results

242

We show that *any* k-counter machine can be simulated by a recurrent neural network composed of Boolean and analog neurons, and using the two STDP rules described by Eq (2)–(3). In this computational paradigm, the states and counter values of the machine 245



Fig 4. A 2-counter machine. The nodes and edges of the graph represent the states and transitions of the machine, respectively. An edge from q to q' labelled by $a, \bar{c}_1, \bar{c}_2 \rightarrow o_1, o_2$ represent the transition $\delta(q, a, \bar{c}_1, \bar{c}_2) = (q', o_1, o_2)$. In other words, if the machine is in computational state q, reads input a and has counter states \bar{c}_1, \bar{c}_2 , then it will move to computational state q' and performs counter operations o_1, o_2 . This 2-counter machine recognizes the language $\{0^n 1^n 0^n : n > 0\}$, i.e., the sequences of bits beginning with a strictly positive number of 0's followed by the same number of 1's and followed by the same number of 0's again.

are encoded into specific synaptic weights of the network. The transitions between those states and counter values are reflected by an evolution of the corresponding synaptic weights. Since 2-counter machines are computationally equivalent to Turing machines, these results show that the proposed STDP-based recurrent neural networks are Turing complete.

Construction

We provide an algorithmic construction which takes the description of a k-counter machine C_k as input and provides a recurrent neural network \mathcal{N} that simulates C_k as output. The network \mathcal{N} is constructed by assembling several modules together: an *input encoding module*, an *input transmission module*, a state module, k counter modules and several detection modules. These modules are described in detail in the sequel. The global behaviour of \mathcal{N} can be summarized as follows.

- 1. The computational state and k counter values of C_k are encoded into specific synaptic weights belonging to the state module and counter modules of \mathcal{N} , respectively. 259
- 2. At the beginning of the simulation, \mathcal{N} receives its input stream via successive 260 activations of its *input cells* belonging to the *input encoding module*. Meanwhile, 261 this module encodes the whole input stream into a single rational number, and 262 stores this number into the activation value of a sigmoid neuron. 263
- 3. Then, each time the so-called *tic* cell of the *input encoding module* is activated, \mathcal{N}_{264} triggers the simulation of one computational step of \mathcal{C}_k .
 - (a) First, it attempts to simulate an ϵ -transition of C_k by activating the cell u_ϵ of the *input transmission module*. If such a transition is possible in C_k , then \mathcal{N} is simulates it.
 - (b) Otherwise, a signal is sent back the *input encoding module*. This module then retrieves the last input bit *a* stored in its memory, and attempts to simulate the regular transition of C_k associated with *a* by activating the cell u_a of the *input transmission module*. If such a transition is possible in C_k , then \mathcal{N} simulates it.
- 4. The network \mathcal{N} simulates a transition of \mathcal{C}_k as follows: first, it retrieves the current computational state and k counter values of \mathcal{C}_k encoded into k + 1 synaptic weights by means of its *detection modules*. Based on this information, it sends specific signals to the *state module* and *counter modules*. These signals update specific synaptic weights of these modules in such a way to encode the new computational state and counter values of \mathcal{C}_k .

The general architecture of \mathcal{N} is illustrated in Fig 5. The general functionalities of the modules are summarized in Table 2. The following sections are devoted to the detailed description of the modules, as well as to the proof of correctness of the construction.

Module	Role	
INPUT ENCODING	 Store the successive input bits into a "stack". Implement a "tic mechanism" which triggers the simulation of one computational step of the machine. 	
INPUT PROCESSING	• Transmit the successive input bits to the network.	
STATE	Encode the successive computational states of the machine into an evolving synaptic weight.Simulate the change in computational states of the machine throughout the computation.	
COUNTER	 NTER Encode the successive counter values of the machine into evolving synaptic weights. Simulate the change in the counter values of the machine through the computation. 	
DETECTION	 • Retrieve the current computational and counter states of the machine. • Use this information to simulate the next transition of the machine. 	

Table 2. Modules composing the STDP-based recurrent neural network that simulates a k-counter machine.



Fig 5. STDP-based recurrent neural network simulating a k-counter machine. The network is obtained by the construction given in Algorithm 1. It is composed of 1 input encoding module. 1 input transmission module, 1 state module, k counter modules, and at most $|Q| \cdot |\Sigma \cup {\epsilon}| \cdot 2k = 6nk$ detection modules, all interconnected together in a precise way. According to this construction, the computational state and counter values of the machine are encoded into specific synaptic weights of the state and counter modules, respectively (red dashed arrow). The synaptic connections provoking changes in these specific weights are depicted in boldface.

Stack encoding. In the sequel, each binary input stream will be piled up into a ²⁸³ "binary stack". In this way, the input stream can be stored by the network, and then ²⁸⁴ processed bit by bit at successive time steps interspersed by constant intervals. The ²⁸⁵ construction of the stack is achieved by "pushing" the successive incoming bits into ²⁸⁶ it. The stack is encoded as a rational number stored in the activation value of one (or ²⁸⁷ several) analog neurons. The pushing and popping stack operations can be simulated by ²⁸⁸ simple analog neural circuits [4]. We now present these notions in detail. ²⁸⁹

A binary stack whose elements from top to bottom are $\gamma_1, \gamma_2, \ldots, \gamma_p \in \{0, 1\}$ is represented by the finite string $\gamma = \gamma_1 \gamma_2 \cdots \gamma_p \in \{0, 1\}^*$. The stack γ whose top element has been popped is denoted by $pop(\gamma) = \gamma_2 \cdots \gamma_p$, and the stack obtained by pushing element $\alpha \in \{0, 1\}$ into γ is denoted by $push(\alpha, \gamma) = \alpha \gamma_1 \gamma_2 \ldots \gamma_p$ (α is now the top element). For instance, if $\gamma = 0110$, then $pop(\gamma) = 110$, $push(0, \gamma) = 00110$ and $push(1, \gamma) = 10110$.

In our context, any stack $\gamma = \gamma_1 \gamma_2 \cdots \gamma_p \in \{0,1\}^*$ is encoded by the rational number $\bar{r}_{\gamma} := \sum_{i=1}^{n} \frac{2\gamma_i + 1}{4^i} \in [0,1]$ [4]. Hence, the top element γ_1 of γ can be retrieved by the operation $top(\gamma) = \sigma(4\bar{r}_{\gamma} - 2) \in \{0,1\}$, where σ is the linear sigmoid function defined previously. The encodings of $push(0,\gamma)$ and $push(1,\gamma)$ are given by $\sigma(\frac{\bar{r}_{\gamma}}{4} + \frac{1}{4})$ and $\sigma(\frac{\bar{r}_{\gamma}}{4} + \frac{3}{4})$, respectively. The encoding of $pop(\gamma)$ is given by $\sigma(4\bar{r}_{\gamma} - 2top(\gamma) - 1)$. As an illustration, the stack $\gamma = 0110$ is encoded by $\bar{r}_{\gamma} = \frac{1}{4} + \frac{3}{16} + \frac{3}{64} + \frac{1}{256}$. The top element of γ is $top(\gamma) = \sigma(1 + \frac{3}{4} + \frac{3}{16} + \frac{1}{64} - 2) = 0$. The encodings of $push(0, \gamma)$ and $push(1, \gamma)$ are $\frac{1}{4} + \frac{1}{16} + \frac{3}{64} + \frac{3}{256} + \frac{1}{1024}$ and $\frac{3}{4} + \frac{1}{16} + \frac{3}{64} + \frac{3}{256} + \frac{1}{1024}$, which represents the stacks 00110 and 10110, respectively. The encoding of $pop(\gamma)$ is $\sigma(1 + \frac{3}{4} + \frac{3}{16} + \frac{1}{64} - 2 \cdot 0 - 1) = \frac{3}{4} + \frac{3}{16} + \frac{1}{64}$, which represents to the stack 110. These four operations can be implemented by simple neural circuits.

Input encoding module. The *input encoding module* is used for two purposes: pile 307 up the successive input bits into a stack, and implement a "tic mechanism" which triggers 308 the simulation of one computational step of the counter machine by the network. These 309 two processes are described in detail below. This module (the most intricate one) has 310 been designed on the basis of the previous considerations about stack encoding, involving 311 neural circuits that implement the "pop", "top" and "pop" operations. It is composed 312 of 31 cells in_0 , in_1 , end, tic, $c_1, \ldots, c_{20}, d_1, \ldots, d_7$, some of which being Boolean and 313 others analog, as illustrated in Fig 6. It is connected to the *input transmission module* 314 and the *detection modules* described below. 315

The three Boolean cells in_0 , in_1 and end are input cells of the network. They are used to transmit the successive inputs bits to the network. The transmission of input 0 or 1 is represented by a spike of cell in_0 or in_1 , respectively. At the end of the input stream, cell end spikes to indicate that all inputs have been processed.

The activity of this module, illustrated in Fig 7, can be described as follows. Suppose 320 that the input stream $a_1 \cdots a_p$ is transmitted to the network. While the bits a_1, \ldots, a_p 321 are being received, the module builds the stack $\gamma = a_1 \cdots a_p$, and stores its encoding 322 \bar{r}_{γ} into the activation values of an analog neuron. To achieve this, the module first 323 pushes every incoming input a_i into a stack γ' (first 'push' circuit in Fig 6). Since 324 pushed elements are by definition added on the top of the stack, γ' consists of elements 325 a_1,\ldots,a_p in reverse order, i.e., $\gamma'=a_p\cdots a_1$. The encoding $\bar{r}_{\gamma'}$ of stack γ' is stored 326 in cell c_1 . Then, the module pops the elements of γ' from top to bottom (first 'pop' 327 circuit in Fig 6), and pushed them into another stack γ (second 'push' circuit in Fig 6). 328 After completion of this process, γ consists of elements a_1, \ldots, a_p in the right order, i.e., 329 $\gamma = a_1 \cdots a_p$. The encoding \bar{r}_{γ} of stack γ is stored in cell c_{14} . 330

The Boolean cell *tic* is also an input cell. Each activation this cell triggers the 331 simulation of one computational step of the counter machine by the network. When the 332 tic cell spikes, it sends a signal to cell u_{ϵ} of the next input transmission module. The 333 activation of u_{ϵ} attempts to launch the simulation of an ϵ -transition of the machine. If, 334 according to the current computational and counter states of the machine, an ϵ -transition 335 is possible, then the network simulates it via its other modules, and at the same time, 336 sends an inhibitory signal to c_{15} . Otherwise, after some delay ('delays' circuit in Fig 6), 337 cell c_{15} is activated. This cell triggers a sub-circuit that pops the current stack γ (second 338 'pop' circuit in Fig 6) and transmits its top element $a \in \{0, 1\}$ to cell u_a of the next input 339 transmission module. Then, the activation of u_a launches the simulation of a regular 340 transition of the machine associated with input symbol a, via the other modules of the 341 network.

The module is composed of several sub-circuits that implement the top(), push() and pop() operations described previously, as shown in Fig 6. An input encoding module is denoted as $input_encoding_module()$.



Fig 6. Input encoding module. This module piles up the successive incoming input bits into a stack and implement the "tic mechanism", which triggers the simulation of one computational step of the counter machine. It is composed of 31 Boolean and analog cells (depicted in white/blue and grey, respectively) $in_0, in_1, end, tic, c_1, \ldots, c_{20}, d_1, \ldots, d_7$. First of all, at successive time steps, cell in_0 or in_1 spikes depending on whether input 0 or 1 is received. Then, cell end spikes to indicate that all input bits have been processed. Meanwhile, the successive bits are pushed into a stack γ' whose encoding is hold by c_1 (first 'push' circuit). After all bits have been pushed, γ' contains all input bits in reverse order. Subsequently, c_2, \ldots, c_7 pop every element of γ' (first 'pop' circuit). Cell c_8 or c_9 spikes iff the popped element is a 0 or a 1, respectively. Afterwards, cells c_{10}, c_{11} push these elements back into a new stack, in order to build the reversed stack γ (second 'push' circuit). The encoding of γ is transferred to and hold by c_{12} and c_{13} at alternating time steps ('copy' circuit), and then hold by c_{14} at every time step. After completion of this process, γ contains all input bits in the original order. Besides this, each time the *tic* cells spikes, it triggers the simulation of one computational step of the counter machine by the network. First, it attempts to simulatate an ϵ -transition by activating cell u_{ϵ} of the next module. If this simulation step fails, cell c_{15} is activated after some delay ('delays' circuit), which represents a signal telling that the top element of stack γ , instead of ϵ , has to be given as next input symbol. In this case, $c_{14}, c_{16}, c_{17}, c_{18}$ pop γ (second 'pop' circuit) and transmit its top element, 0 or 1, to cell c_{19} or c_{20} , respectively. Cell c_{19} or c_{20} then activates cell u_0 or u_1 of the next module, respectively, triggering the simulation of a regular transition.

Input transmission module. The *input transmission module* is used to transmit to the network the successive input bits sent by the previous input encoding module. The module simply consists of 3 Boolean input cells u_0 , u_1 , u_{ϵ} followed by 3 layers of Boolean delay cells, as illustrated in Fig 8. It is connected to the *input encoding module* described 349



Fig 7. Example of activity of the input encoding module. The lower graph is a raster plot displaying the cells' activities. Activation values between 0 and 1 (of sigmoid neurons) are not represented, only spikes are. In this simulation, the input stream 001101 and the "end of input" signal are transmitted via cells in_0, in_1, end at successive time steps $0, 1, 2, \ldots, 7$ (blue pattern). The successive input bits are first piled up in reverse order into a stack γ' whose encoding is stored as the activation value of c_1 , and then piled up again in the right order into a stack γ whose encoding is stored as the activation value of c_{14} . The activation values of c_1 and c_{14} over time are represented by the orange and red curves in the upper graph, respectively. Then, the *tic* cell spikes every 15 time steps from t = 20 onwards (blue pattern). Each such spike triggers the sub-circuit that pops stack γ and outputs its top element, 0 or 1, by activating cell c_{19} or c_{20} 10 time steps later, respectively. We see that the successive input bits, namely 0, 0, 1, 1, 0, 1, 0 (blue pattern), are correctly output by cells c_{19} or c_{20} (red pattern).

above, and to the state module, counter modules and detection modules described below. The activation of cell u_0 , u_1 or u_{ϵ} simulates the reading of input symbol 0, 1 or ϵ by the counter machine, respectively. Each time such a cell is activated, the information propagates along the delay cells of the corresponding row. An input transmission module is denoted as *input_transmission_module()*. 350

State module. In our model, the successive computational states of the counter machine are encoded as rational numbers, and stored as successive weights of a designated synapse $w_s(t)$ (subscript *s* refers to 'state'). More precisely, the fact that the machine is in state q_k is encoded by the rational weight $w_s(t) = a_{min} + k \cdot \eta$, for k = 0, ..., n-1, where a_{min} and η are parameters of the STDP rule given by Eq (2). Hence, the change in computational state of the machine is simulated by incrementing or decrementing $w_s(t)$ in a controlled manner. This process is achieved by letting $w_s(t)$ be subjected to



Fig 8. Input transmission module. This module transmits the successive inputs bits to the network. It is composed of three Boolean input cells u_0 , u_1 , u_{ϵ} (in blue) followed by 3 layers of Boolean delay cells connected in a parallel way via excitatory connections of weights 1. The activation of cells u_0 , u_1 or u_{ϵ} simulates the reading of input symbols 0, 1 or ϵ by the counter machine, respectively.

the STDP rule of Eq (2), and by triggering specific spiking patterns of the presynaptic and postsynaptic cells of $w_s(t)$.

The state module is designed to implement these features. It is composed of a Boolean 364 presynaptic cell pre_s connected to an analog postsynaptic cell $post_s$ by a synapse of 365 weight $w_s(t)$, as well as of 6(n-1) Boolean cells $c_1, \ldots, c_{3(n-1)}$ and $\bar{c}_1, \ldots, \bar{c}_{3(n-1)}$ (for 366 some n to be specified), as illustrated in Fig 9. The synaptic weight $w_s(t)$ is subjected to 367 the STDP rule of Eq (2), and has an initial value of $w_s(0) = a_{min}$. The architecture of 368 the module ensures that the activation of cell c_{3k+1} or \bar{c}_{3k+1} triggers successive specific 369 spiking patterns of pre_s and $post_s$ which, according to STDP (Eq (2)), increments or 370 decrements $w_s(t)$ by $(n-1-k) \cdot \eta$, for any $0 \le k \le n-2$, respectively (for instance, if 371 k = 0, then $w_s(t)$ is incremented or decremented by $(n-1) \cdot \eta$, whereas if k = n-2, then 372 $w_s(t)$ is only incremented or decremented by $1 \cdot \eta$). The module is linked to the *input* 373 transmission module described above and to the detection modules described below. 374

The activity of this module, illustrated in Fig 10, can be described as follows. Suppose 375 that at time step t, one has $w_s(t) = v$ and one wishes to increment (resp. decrement) 376 $w_s(t)$ by $(n-1-k) \cdot \eta$, where η is the learning rate of the STDP rule of Eq (2) and 377 $0 \le k \le n-2$. To achieve this, we activate the cell c_{3k+1} (resp. cell \bar{c}_{3k+1}) (a blue cell 378 of Fig 9). The activation of c_{3k+1} (resp. cell \bar{c}_{3k+1}) launches a chain of activations of 379 the next cells (red events in Fig 10), which, according to the connectivity of the module, 380 induces k successive pairs of spikes of pre_s followed by $post_s$ (resp. $post_s$ followed by 381 pre_s) (blue events in Fig 10). Thanks to the STDP rule of Eq (2), these spiking patterns 382 increment (resp. decrement) k times the value of $w_s(t)$ by an amount of η . A state 383 module with 6(n-1) + 2 cells is denoted as $state_module(n-1)$. 384

Counter module. In our model, the successive counter values of the machine are encoded as rational numbers and stored as successive weights of designated synapses $w_{c_j}(t)$, for $j = 1, \ldots, k$ (subscript c_j refers to 'counter j'). More precisely, the fact that counter j has a value of $n \ge 0$ at time t is encoded by the synaptic weight $w_{c_j}(t)$ having the rational value $r_n := \sum_{i=1}^n \frac{1}{2^i}$ (with the convention that $r_0 := 0$). Then, the "push" (incrementing the counter by 1) and "pop" (decrementing the counter by 1) operations are simulated by incrementing or decrementing $w_{c_j}(t)$ appropriately.



Fig 9. State module. This module is used to simulate the successive computational states of the counter machine. It is composed of a Boolean cell pre_s connected to an analog cell $post_s$ via a synaptic connection of weight $w_s(t)$ (dashed red arrow) subjected to the first STDP rule given by Eq (2), as well as of 6n Boolean cells $c_1, \ldots, c_{3(n-1)}$ and $\bar{c}_1, \ldots, \bar{c}_{3(n-1)}$. The latter cells project onto pre_s and $post_s$ via excitatory and inhibitory synapses. To increment (resp. decrement) the value of $w_s(t)$ by $(n-1-k) \cdot \eta$ (where η is the learning rate of the STDP rule of Eq (2)), it suffices to activate the blue cell c_{3k+1} (resp. cell \bar{c}_{3k+1}), where $0 \le k \le n-2$.

The k counter modules are designed to implement these features. Each counter module is composed of 12 Boolean cells push, pop, test, = $0, \neq 0, pre_c, post_c, c_1, c_2, c_3, c_4, c_5$, as illustrated in Fig 11. The presynaptic and postsynaptic cells pre_c and $post_c$ are connected by a synapse of weight $w_c(t)$ subjected to the second STDP rule given by Eq (3) and having an initial value of $w_c(0) = 0$. Accordingly, the values of $w_c(t)$ may vary across the elements of the infinite sequence $\beta = (1 - \frac{1}{2^k})_{k=0}^{\infty} = (0, 0.5, 0.75, 0.875, 0.9375, ...)$. The module is connected to the *input transmission module* described above and to detection modules described below.

The activity of this module, illustrated in Fig 12, can be described as follows. Each 400 activation of the *push* (resp. *pop*) cell (blue events in Fig 12) propagates into the circuit 401 and results 2 time steps later in successive spikes of the pre_c and $post_c$ cells (resp. $post_c$ 402 and pre_c cells), which, thanks to the STDP rule of Eq (3), increment (resp. decrement) 403



Fig 10. Example of activity of the state module. The lower graph is a raster plot displaying the cells' activities. When cell c_1 (resp. \bar{c}_1) spikes, it launches a chain of activations of the next cells c_2, \ldots, c_9 (resp. $\bar{c}_2, \ldots, \bar{c}_9$). These activations (red events) induce spiking patterns of the cells pre_s and $post_s$ (blue events), which thanks to the STDP rule of Eq (2), increment (resp. decrement) the synaptic weight $w_s(t)$ by steps of η ($\eta = 0.1$ here). The value of $w_s(t)$ over time is represented in the upper plot (red curve).

the value of $w_c(t)$ (red curve in Fig 12). The activation of the test cell (blue events in 404 Fig 12) results 4 time steps later in the spike of the Boolean cell '= 0' or ' \neq 0' (red 405 events in Fig 12), depending on whether $w_c(t) = 0$ or $w_c(t) \neq 0$, respectively. During this 406 process, the value of $w_c(t)$ is first incremented (2 time steps later) and then decremented 407 (2 time steps later again) back to its original value. In other words, the testing procedure 408 induces a back and forth fluctuation of $w_c(t)$, without finally modifying it from its initial 409 value (this fluctuation is unfortunately unavoidable). A counter module is denoted as 410 counter_module(). 411

Detection modules. Detection modules are used to retrieve—or detect—the current 412 computational and counter states of the machine being simulated. This information 413 is then employed to simulate the next transition of the machine. More precisely, each 414 input symbol $a \in \Sigma \cup \{\epsilon\}$, computational state $q \in Q$ and counter states $\bar{c}_1, \ldots, \bar{c}_k \in C$ 415 of the machine are associated with a corresponding detection module. This module is 416 activated if and only if the current input bit processed by the network is precisely a, the 417 current synaptic weights $w_s(t)$ corresponds to the encoding of the computational state q, 418 and the current synaptic weights $w_{c_1}(t), \ldots, w_{c_k}(t)$ are the encodings of counter values 419 with corresponding counter states $\bar{c}_1, \ldots, \bar{c}_k$. Afterwards, the detection module sends 420 suitable activations to the state and counter modules so as to simulate the next transition 421 $\delta(q, a, \bar{c}_1, \dots, \bar{c}_k) = (q', o_1, \dots, o_k)$ of the machine. Formally, a detection module detects 422 if the activation value of cell $post_s$ of the state module is equal to a certain value v, 423 together with the fact that k signals from cells = 0 or $\neq 0$ of the k counter modules are 424



Fig 11. Counter module. This module is used to simulate one counter of a k-counter machine. It is composed of 12 Boolean cells: push, pop, test, = 0, $\neq 0$, pre_c , $post_c$ (in blue), c_1 , c_2 , c_3 , c_4 , c_5 . The presynaptic and postsynaptic cells pre_c and $post_c$ are connected by a synapse of weight $w_c(t)$ (dashed red arrow) subjected to the second STDP rule given by Eq (3). The activation of the *push* or *pop* cell increments or decrements the value of $w_c(t)$, respectively. The activation of the *test* cell results in the activation of the cell '= 0' or ' \neq 0', depending on whether w(t) = 0 or $w(t) \neq 0$, respectively.

correctly received. The module is composed of 4 Boolean cells connected in a feedforward manner, as illustrated in Fig 13. It is connected to the *input transmission module*, the *state module* and the *counter modules* described above.

The activity of this module, illustrated in Fig 14, can be described as follows. Suppose 428 that at time step t, cell c_1 is spiking and cell $post_s$ has an activation value of v (with 429 $0 \le v \le 1$). Then, at time t+1, both c_2 and c_3 spike (since they receive signals of 430 intensity 1). At next time t+2, two signals of intensities $\frac{1}{k+2}$ are transmitted to c_4 . 431 Suppose that at this same time step, c_4 also receives k signals from the counter modules. 432 Then, c_4 receives k+2 signals of intensities $\frac{1}{k+2}$, and hence spikes at time t+3 (case 433 1 of Fig 14). By contrast, if at time step t, c_1 is spiking and $post_s$ has an activation 434 value of v' > v (resp. v' < v), then at time t + 1 only c_2 (resp. c_3) spikes. Hence, at time 435 $t+2, c_4$ receives less than k+2 signals of intensities $\frac{1}{k+2}$, and thus stays quiet (cases 3) 436 and 4 of Fig 14). Consequently, the 'detection cell' c_4 (blue cell of Fig 13) spikes if and 437 only if $post_s$ has an exact activation value of v and c_4 receives exactly k signals from its 438 afferent connections. A detection module involving weights $1 - v, 1 + v, \frac{1}{k+2}$ is denoted 439 as $detection_module(v, k)$. 440

Assembling the modules. Any given k-counter machine $C_k = (Q, \Sigma, C, O, \delta, 0, F)$ (where $\Sigma = \{0, 1\}$ and $Q = \{0, ..., n-1\}$) can be simulated by a recurrent neural network \mathcal{N} subjected to the STDP rules given by Eq (2)–(3). The network is obtained 443



Fig 12. Example of activity of the counter module. The lower graph is a raster plot displaying the cells' activities. Cells push, push, test, pop, test, pop, pop, test are activated at successive time steps 0, 10, 20, 30, 40, 50, 60, 70 (blue pattern). The upper curve shows the fluctuation of the synaptic weight $w_c(t)$, which encodes the change in the counter value over time. Note that the activations of the push and pop cells correctly increment and decrement the value of $w_c(t)$, respectively. At time 60, when $w_c(t) = 0$ (counter is zero), the pop signal has no more effect on its value. Moreover, test queries are performed at times 20, 40 and 70 and their answers given by the activities of cells '= 0' and ' \neq 0' (red pattern) at time 24, 44 and 74, respectively. Note that cells '= 0' and ' \neq 0' provide correct answers to whether the value of $w_c(t)$ is 0 or not. Finally, note that whenever $w_c(t) \neq 0$, each testing procedure induces a fluctuation of $w_c(t)$ (peaks of the red curve), without finally modifying its initial value.



Fig 13. Detection module. This module is used to detect if the activation value of $post_s$ is equal to v together with the fact that k signals from the counter modules are correctly received. If these conditions are fulfilled, the 'detection cell' c_4 spikes, which triggers the simulation of the next transition of the machine. It is composed of 4 Boolean cells c_1 , c_2 , c_3 , c_4 connected in a feedforward way.



Fig 14. Examples of activity of the detection module. The detection module, composed of the cells c_1, c_2, c_3, c_4 , receives activations from the state module via the cell $post_s$, as well as from 2 counter modules via the cells ext_1, ext_2 . The module detects whether the activation value of $post_s$ is equal to v = 0.3 together with the fact that both ext_1, ext_2 have been activated. In case 1, these conditions are fulfilled, and thus the 'detection cell' c_4 spikes (bold spike at t = 3). In all other cases, the required conditions are not fulfilled: either only one external activation is received (ext_2 has not spiked, case 2), or the activation value v of $post_s$ satisfies v > 0.3 (thus c_3 is not spiking, case 3) or v < 0.3 (thus c_2 is not spiking, case 4). In each case, the 'detection cell' c_4 does not spike.

by a suitable assembling of the modules described above. The architecture of $\mathcal N$ is 444 illustrated in Fig 5, and its detailed construction is given by Algorithm 1. In short, 445 the network \mathcal{N} is composed of 1 input encoding module (line 1), 1 input transmission 446 module (line 2), 1 state module (line 3), k counter modules (lines 4-6) and at most 447 $|Q| \cdot |\Sigma \cup \{\epsilon\}| \cdot 2^k = 3n2^k$ detection modules (lines 7–11). The modules are connected 448 together according to the patterns described in lines 12–47. This makes a total of $\mathcal{O}(n2^k)$ 449 cells and $\mathcal{O}(nk2^k)$ synapses, which, since the number of counters k is fixed, corresponds 450 to $\mathcal{O}(n)$ cells and $\mathcal{O}(n)$ synapses. A recurrent neural networks obtained via Algorithm 1 451 is referred to as an STDP-based RNN. 452

Turing completeness

We now prove that any k-counter machine is correctly simulated by its corresponding 454 STDP-based RNN given by Algorithm 1. Since 2-counter machines are Turing complete, 455 then so is the class of STDP-based RNNs. Towards this purpose, the following definitions 456 need to be introduced. 457

Let \mathcal{N} be an STDP-based RNN. The input cells of \mathcal{N} are the cells in_0, in_1, end, tic of the input encoding module (cf. Figure 6, four blue cells of the first layer). Thus, inputs of \mathcal{N} are vectors in \mathbb{B}^4 whose successive components represent the spiking configurations of cells in_0, in_1, end , and tic, respectively. In order to describe the input streams of \mathcal{N} , we consider the following vectors of \mathbb{B}^4 :

$$\mathbf{0} := \begin{pmatrix} 1\\0\\0\\0 \end{pmatrix}, \ \mathbf{1} := \begin{pmatrix} 0\\1\\0\\0 \end{pmatrix}, \ \mathbf{end} := \begin{pmatrix} 0\\0\\1\\0 \end{pmatrix}, \ \mathbf{tic}_{\mathbf{i}} := \begin{pmatrix} 0\\0\\0\\1 \end{pmatrix}, \text{for all } i \ge 0 \text{ and } \mathbf{\emptyset} := \begin{pmatrix} 0\\0\\0\\0\\0 \end{pmatrix}.$$

According to these notations, the input stream 0011endØØtic corresponds to the 463

Algorithm 1 Procedure which takes a k-counter machine as input and builds an STDP-based RNN that simulates it.

O T	Di -based ititit that sindlates it.						
Re	quire: k-counter machine $C_k = (Q, \Sigma, C, O, \delta, 0, F)$, where	$\Sigma = \{0, 1\}$ at	nd $Q = \{0, \dots, n-1\}$				
	// note: comp	utational sta	ates are represented as integers				
	•	// *** I	NSTANTIATION OF THE MODULES ***				
1:	$IN1 \leftarrow input_encoding_module()$		<pre>// input encoding module</pre>				
2:	$IN2 \leftarrow input_transmission_module()$		// input transmission module				
3.	$ST \leftarrow state module(n-1)$		// state module (where $n = Q $)				
4.	for all $i = 1$ k do	,	, state medale ("mole it deli				
5.	$C(i) \leftarrow counter module()$		l/k counter modules				
о. 6.	end for		// % counter modules				
7.	for all tuple $(i, q, \bar{c}, \dots, \bar{c}) \in O \times \Sigma \cup \{c\} \times C^k$ do						
0.	if $\delta(i, q, \bar{c}_1, \dots, \bar{c}_k) \in Q \times D \in \{0\} \times O$ up						
0.	$\operatorname{DET}(i, a, \overline{a}, \dots, \overline{a})$ is defined then $\operatorname{DET}(i, a, \overline{a}, \dots, \overline{a}) \neq detection modulo(a, \dots, i)$	(n, h)	// detection modules				
9:	$DEI(i, a, c_1, \ldots, c_k) \leftarrow aeieciion_module(a_{min} + i)$	$\cdot \eta, \kappa)$	// detection modules				
10:	end n						
11:	end for	//	CONNECTION DETUEEN MODILIEC Atta				
	IND CINI 4 CONDUCTING	// ***	CUNNECTION BETWEEN MUDULES ***				
12:	connect c_{19} of IN1 to u_0 of IN2: weight 1	// input	encoding to input transmission				
13:	connect c_{20} of IN1 to u_1 of IN2: weight 1						
14:	connect <i>tic</i> of IN1 to u_{ϵ} of IN2: weight 1						
15:	for all $j = 1, \ldots, k$ do	//	input transmission to counters				
16:	connect u_0, u_1, u_{ϵ} of IN2 to test of $C(j)$: weight 1						
17:	end for						
18:	connect $d_{2,0}, d_{2,1}, d_{2,\epsilon}$ of IN2 to pre_s of ST: weight 1		// input transmission to state				
19:	for all tuple $(i, a, \overline{c}_1, \dots, \overline{c}_k) \in Q \times \Sigma \cup \{\epsilon\} \times C^k$ do						
20:	if $\delta(i, a, \overline{c}_1, \dots, \overline{c}_k) = (i', o_1, \dots, o_k)$ then						
21:	connect $d_{3,a}$ of IN2 to c_1 of $\text{DET}(i, a, \bar{c}_1, \ldots, \bar{c}_k)$: w	reight 1 // :	input transmission to detection				
22:	connect post of ST to c_2 of DET $(i, a, \bar{c}_1, \ldots, \bar{c}_k)$: we	eight 1	<pre>// state to detection</pre>				
23:	connect post of ST to c_3 of DET $(i, a, \bar{c}_1, \ldots, \bar{c}_k)$: we	eight -1					
24:	if $a == \epsilon$ then		<pre>// detection to input encoding</pre>				
25:	connect c_4 of $DET(i, a, \overline{c}_1, \ldots, \overline{c}_k)$ to c_{15} of IN1:	weight -1					
26:	end if						
27:	if $i' - i > 0$ then		<pre>// detection to state</pre>				
28:	connect c_4 of $\text{DET}(i, a, \overline{c}_1, \dots, \overline{c}_k)$ to $c_{3((n-1)-(i)}$	(-i))+1 of ST	: weight 1				
29:	else if $i' - i < 0$ then	.,,,,					
30:	connect c_4 of DET $(i, q, \bar{c}_1, \dots, \bar{c}_h)$ to $\bar{c}_{q_1}(\dots, q_h)$	(in the of ST	: weight 1				
31:	end if	-i))+1 ~-					
32.	for all $i = 1, \dots, k$ do		// detection to counters				
33.	if $a_i = nush$ then		,, 200000101 00 002001012				
34.	connect c_4 of DET $(i, a, \bar{c}_1, \dots, \bar{c}_k)$ to cell must	of C(i): wei	ght 1				
35	else if $a_i == non$ then	0 01 0 (j). Wei	8				
36.	connect c_4 of DET $(i \ a \ \bar{c}_1 \ \bar{c}_1)$ to cell pop	of C(i) weigh	ht 1				
37.	end if	or e(j). weigi					
201.	end for						
20.	end if						
40.	for all $i = 1$ k do		// counters to detection				
40:	if $\bar{a}_1 = 1, \dots, n$ do		// counters to detection				
41:	$c_{j} = \pm t_{i}$ end $c_{j} = \pm c_{i}$ end c_{i} of $C(i)$ to c_{i} of $DET(i, a, \bar{a})$), weight 1					
42:	$= 0 \text{ of } O(f) \text{ to } c_4 \text{ of } D E I(i, a, c_1, \dots, c_k)$	k). Weight k +	2				
43:	else il $c_j == +$ then	×					
44:	connect $\neq 0$ or $\mathcal{O}(j)$ to c_4 of DET (i, a, c_1, \ldots, c_k)	$\frac{1}{k+1}$: weight $\frac{1}{k+1}$	2				
45:	end if						
46: end for							
47: end for							

following sequence of vectors provided at successive time steps

464

i.e., to the successive spikes of cells in_0 , in_0 , in_1 , in_1 , end, followed by two times steps during which all cells are quiet, followed by a last spike of the cell tic.

For any binary input $w = a_0 \cdots a_p \in \Sigma^*$, let $u_w \in (\mathbb{B}^4)^*$ be the corresponding input stream of \mathcal{N} defined by

$$u_w = \mathbf{a_0} \cdots \mathbf{a_p} \mathbf{end} \underbrace{\emptyset \cdots \emptyset}_{K'_{p+1}} \operatorname{tic}_{\mathbf{0}} \underbrace{\emptyset \cdots \emptyset}_{K} \operatorname{tic}_{\mathbf{1}} \underbrace{\emptyset \cdots \emptyset}_{K} \operatorname{tic}_{\mathbf{2}} \cdots$$

where $\mathbf{a_i} = \mathbf{0}$ if $a_i = 0$ and $\mathbf{a_i} = \mathbf{1}$ if $a_1 = 0$, for $i = 0, \dots, p$. In other words, the input 467 stream u_w consists of successive spike from cells in_0 and in_1 (inputs $\mathbf{a}_0 \cdots \mathbf{a}_p$), followed 468 by one spike from cell end (input end), followed by K'_{p+1} time steps during which 469 nothing happens (inputs $\emptyset \cdots \emptyset$), followed by successive spikes from cell *tic*, interspersed 470 by constant intervals of K time steps during which nothing happens (input blocks 471 $\operatorname{tic}_{\mathbf{i}} \emptyset \cdots \emptyset$). The value of K'_{p+1} is chosen such that, at time step p+2+K', the p+1472 successive bits of u_w are correctly stored into cell c_{14} of the input encoding module. The 473 value of K is chosen such that, after each spike of the tic cell, the updating of the state 474 and counter modules can be achieved within K time steps. Taking $K'_{p+1} \geq 3(p+1) + 4$ 475 and $K \ge 17 + 3(n-1)$ (where n = |Q|) satisfies these requirements. Note that K'_{p+1} 476 depends on the input length, while K is constant (for a given counter machine). An input 477 stream of this form is depicted by the 4 bottom lines of Fig 15 (in this case $K'_{n+1} = 23$ 478 and K = 29). Besides, for each $i \ge 0$, let t_i be the time step at which **tic**_i occurs. For 479 instance, in Fig 15, one has $t_0 = 30, t_1 = 60, t_2 = 90, t_3 = 120, \dots$ Let also 480

$$w_{s,i} := w_s(t_i - 1)$$

$$w_{c_{1,i}} := w_{c_1}(t_i - 1), \dots, w_{c_{k,i}} := w_{c_k}(t_i - 1)$$

be the synaptic weights $w_s(t), w_{c_1}(t), \ldots, w_{c_k}(t)$ of the state and counter modules at time step $t_i - 1$ (i.e., 1 time step before **tic**_i has occurred), with the assumption that

$$(w_{s,0}, w_{c_1,0}, \dots, w_{c_k,0}) = (a_{min}, 0, \dots, 0).$$

For example, in Fig 15, the values of $w_s(t)$, $w_{c_1}(t)$, $w_{c_2}(t)$ over time are represented by the upper red and orange curves (pay attention to the different left-hand and right-hand scales associated to these curves): one has $(w_{s,0}, w_{c_1,0}, w_{c_2,0}) = (0.1, 0, 0)$, $(w_{s,1}, w_{c_1,1}, w_{c_2,1}) = (0.3, 0, 0)$, $(w_{s,2}, w_{c_1,2}, w_{c_2,2}) = (0.3, 0.5, 0.5)$, $(w_{s,3}, w_{c_1,3}, w_{c_2,3}) = (0.3, 0.75, 0.75)$, etc. Furthermore, let $a''_i \in \Sigma \cup \{\epsilon\} \cup \{\emptyset\}$ be defined by

$$a_i'' = \begin{cases} a & \text{if cell } c_4 \text{ of one and only one detection module } \text{DET}(q, a, \bar{c}_1, \dots, \bar{c}_k) \\ & \text{spikes between } t_i \text{ and } t_{i+1}, \text{ for some } q \in Q, \ a \in \Sigma \cup \{\epsilon\}, \ \bar{c}_1, \dots, \bar{c}_k \in C \\ \emptyset & \text{otherwise} \end{cases}$$

In other words, a_i'' is the *input symbol* (possibly ϵ) processed by \mathcal{N} between t_i and t_{i+1} . For instance, in Fig 15, the successive input bits processed by the network are displayed by the spiking patterns of the cells u_{ϵ}, u_0, u_1 : one has $a_0'' = \epsilon$ (only u_{ϵ} spikes between t_0 and t_1), $a_1'' = 0$ (both u_{ϵ} and then u_0 spike between t_1 and t_2 , but only u_0 leads to the activation of a detection module, even if this is not represented), $a_2'' = 0$ (u_0 spikes after u_{ϵ} between t_2 and t_3), $a_3'' = 1$ (u_1 spikes after u_{ϵ} between t_3 and t_4), etc.

Now, for any input stream u_w , the *computation* of \mathcal{N} over u_w is the sequence

$$\mathcal{N}(u_w) = \left(\left(w_{s,i}, a_i'', w_{c_{1,i}}, \dots, w_{c_{k,i}} \right) \right)_{i=0}^{l_2}, \ l_2 \in \mathbb{N} \cup \{\infty\}$$
(5)

where $l_2 = \min\{t_i : a_i'' = \emptyset, i \ge 0\}$. In other words, the computation of \mathcal{N} over u_w is the sequence of successive values of $w_s(t), a_i'', w_{c_1}(t), \ldots, w_{c_k}(t)$, which are supposed to encode the successive states, input symbols and counter values of the machine to be simulated, respectively.

According to these considerations, we say that C_k is simulated in real time by \mathcal{N} , or equivalently that \mathcal{N} simulates C_k in real time, if and only if, for any input $w \in \Sigma^*$ with corresponding input stream $u_w \in (\mathbb{B}^4)^*$, the computations of C_k over w (Eq (4)) and of \mathcal{N} over u_w (Eq (5)) 495

$$\mathcal{C}_{k}(w) = ((n_{i}, a'_{i}, c_{1,i}, \dots, c_{k,i}))_{i=0}^{l_{1}}$$

$$\mathcal{N}(u_{w}) = ((w_{s,i}, a''_{i}, w_{c_{1,i}}, \dots, w_{c_{k,i}}))_{i=0}^{l_{2}}$$

satisfy the following conditions:

$$w_{s,i} = a_{min} + n_i \cdot \eta \qquad \text{state condition} \tag{6}$$

$$a_i'' = a_i'$$
 symbol condition (7)

$$w_{c_{j,i}} = r_{c_{j,i}}$$
 for all $j = 1, \dots, k$ counter values condition (8)

for all $i = 0, \ldots, l_1$, which implicitly implies that $l_2 \ge l_1$ (recall that $r_0 := 0$ and $r_n := \sum_{i=1}^n \frac{1}{2^i}$, for all n > 0). In other words, \mathcal{C}_k is simulated by \mathcal{N} iff, on every input, 497 the computations of \mathcal{C}_k is perfectly reflected by that of \mathcal{N} : the sequence of input symbols 498 processed by \mathcal{C}_k and \mathcal{N} coincide (Condition (7)), and the successive computational 499 states and counter values of \mathcal{C}_k are properly encoded into the successive synaptic weights 500 of $w_s(t), w_{c_1}(t), \ldots, w_{c_k}(t)$ of \mathcal{N} , respectively (Conditions (6) and (8)). According to 501 these considerations, each state $n_i \in \mathbb{N}$ and counter value $c_{j,i} \in \mathbb{N}$ of \mathcal{C}_k is encoded 502 by the synaptic value $w_s(t_i-1) = a_{min} + n_i \cdot \eta \in \mathbb{Q}$ and $w_{c_i}(t_i-1) = r_{c_{i,i}} \in \mathbb{Q}$, for 503 $j = 1, \ldots, k$, respectively. The real time aspect of the simulation is ensured by the fact 504 that the successive time steps $(t_i)_{i>0}$ involved in the computation $\mathcal{N}(w)$ are separated 505 by a constant number of time steps K > 0. This means that the transitions of \mathcal{C}_k are 506 simulated by \mathcal{N} in fixed amount of time. 507

We now show that, in this precise sense, any k-counter machine is correctly simulated tits corresponding STDP-based recurrent neural network. 509

Theorem 1. Let C_k be a k-counter machine and \mathcal{N} be the STDP-based RNN given by Algorithm 1 applied on C_k . Then, C_k is simulated in real time by \mathcal{N} .

Proof. Let $w = a_0 \cdots a_p \in \Sigma^*$ be some input and $u_w \in (\mathbb{B}^4)^*$ be its corresponding input stream. Consider the two computations of \mathcal{C}_k on w (Eq (4)) and of \mathcal{N} on u_w (Eq (5)), respectively:

$$\mathcal{C}_{k}(w) = \left((n_{i}, a'_{i}, c_{1,i}, \dots, c_{k,i}) \right)_{i=0}^{l_{1}} \\ \mathcal{N}(u_{w}) = \left((w_{s,i}, a''_{i}, w_{c_{1,i}}, \dots, w_{c_{k,i}}) \right)_{i=0}^{l_{2}}$$

We prove by induction on i that $C_k(w)$ and $\mathcal{N}(u_w)$ satisfy Conditions (6)–(8), for all 515

 $i=0,\ldots,l_1.$

By definition, the first elements of $C_k(w)$ and $\mathcal{N}(u_w)$ are

$$(n_0, a'_0, c_{1,0}, \dots, c_{k,0}) = (0, a'_0, 0, \dots, 0)$$

$$(w_{s,0}, a''_0, w_{c_1,0}, \dots, w_{c_k,0}) = (a_{min}, a''_0, 0, \dots, 0).$$

Hence, Conditions (6) and (8) are satisfied for i = 0, i.e.,

$$w_{s,0} = a_{min} + n_0 \cdot \eta$$
 and $w_{c_{j,0}} = r_{c_{j,0}}$ for all $j = 1, \dots, k.$ (9)

We now prove Condition (7) for i = 0. Towards this purpose, the following observa-519 tions are needed. By construction and according to the value of K_{p+1} , at time $t_0 - 1$, cell 520 c_{14} of the input encoding module IN1 holds the encoding of the whole input $w = a_0 \cdots a_p$ 521 (the latter being considered as a stack). The top element of this stack is a_0 . Besides, 522 according to Relations (9) and Algorithm 1 (lines 22-23 and 40-46), only the detection 523 modules $\text{DET}(n_0, a, \bar{c}_{1,0}, \dots, \bar{c}_{k,0})$, where $a \in \Sigma \cup \{\epsilon\}$, are susceptible have their cell 524 c_4 activated between t_0 and t_1 (indeed, only these modules are capable of "detecting" 525 the current synaptic value $a_{min} + n_0 \cdot \eta$ and counters states $c_{1,0}, \ldots, c_{k,0}$ involved in 526 Relations (9)). 527

Now, consider the symbol $a'_0 \in \Sigma \cup \{\epsilon\}$. Then either $a'_0 \in \Sigma$ or $a'_0 = \epsilon$. As a 528 first case, suppose that $a'_0 \in \Sigma$. Since $a'_0 \neq \epsilon$ and a'_0 is the first symbol processed by 529 \mathcal{C}_k during its computation over input $w = a_0 \cdots a_p$ (cf. Eq (4)), one necessarily has 530 $a'_0 = a_0$. Thus, $\delta(n_0, a'_0, \bar{c}_{1,0}, \dots, \bar{c}_{k,0}) = \delta(n_0, a_0, \bar{c}_{1,0}, \dots, \bar{c}_{k,0})$, and the determinism of 531 \mathcal{C}_k ensures that $\delta(n_0, \epsilon, \overline{c}_{1,0}, \ldots, \overline{c}_{k,0})$ is undefined. According to Algorithm 1 (lines 7–11), 532 the module $\text{DET}(n_0, a_0, \bar{c}_{1,0}, \dots, \bar{c}_{k,0})$ is instanciated, whereas $\text{DET}(n_0, \epsilon, \bar{c}_{1,0}, \dots, \bar{c}_{k,0})$ 533 is not. Hence, the dynamics of \mathcal{N} between t_0 and t_1 goes as follows. At time t_0 , the 534 cell tic of IN1 sends a signal to u_{ϵ} of IN2 (Algorithm 1, line 14) which propagates 535 to the detection modules associated to symbol ϵ (Algorithm 1, line 21). Since the 536 module $\text{DET}(n_0, \epsilon, \bar{c}_{1,0}, \ldots, \bar{c}_{k,0})$ does not exist, it can certainly not be activated, and 537 thus, the cell c_{15} of IN1 will not be inhibited in return (Algorithm 1, line 24–26). The 538 spike of c_{15} will then trigger the sub-circuit of IN1 that pops the top element of the 539 stack currently encoded in c_{14} , namely, the symbol a_0 . This triggers the activation 540 of c_{19} or c_{20} of IN1 depending on whether $a_0 = 0$ or $a_0 = 1$. This activity then 541 propagates to cells u_{a_0} and next d_{3,a_0} of IN2 (Algorithm 1, lines 12–13). It propagates 542 further to the detection modules of the form $DET(\cdot, a_0, \cdot, \ldots, \cdot)$, and in particular to 543 $\text{DET}(n_0, a_0, \bar{c}_{1,0}, \dots, \bar{c}_{k,0})$ (Algorithm 1, line 21). According to Relations (9), the cell c_4 544 of $\text{DET}(n_0, a_0, \bar{c}_{1,0}, \dots, \bar{c}_{k,0})$, and of this module only, will be activated, since it is the 545 only module of this form capable of detecting the current weight $w_s(t) = a_{min} + \eta \cdot n_0$ 546 as well as the current counter states $\bar{c}_{1,0}, \ldots, \bar{c}_{k,0}$ (Algorithm 1, lines 22–23 and 40–46). 547 This amounts to saying that the symbol a_0'' processed by \mathcal{N} between t_0 and t_1 is equal to 548 a_0 . Therefore, $a''_0 = a_0 = a'_0$. This shows that in this case, Condition (7) holds for i = 0. 549 As a second case, suppose that $a'_0 = \epsilon$. It follows that $\delta(n_0, a'_0, \bar{c}_{1,0}, \dots, \bar{c}_{k,0}) =$ 550

516 517

 $\delta(n_0, \epsilon, \bar{c}_{1,0}, \ldots, \bar{c}_{k,0})$, and by Algorithm 1 (lines 7–11), the module DET $(n_0, \epsilon, \bar{c}_{1,0}, \ldots, \bar{c}_{k,0})$ ⁵⁵¹ is instanciated. Consequently, the dynamics of \mathcal{N} between t_0 and t_1 goes as follows. At time t_0 , the cell *tic* of IN1 sends a signal to u_{ϵ} of IN2 (Algorithm 1, line 14) which propagates to the module DET $(n_0, \epsilon, \bar{c}_{1,0}, \ldots, \bar{c}_{k,0})$ (Algorithm 1, line 21). By Relations (9), the cell c_4 of this detection module, and of only this one, will be activated (Algorithm 1, lines 22–23 and 40–46). This amounts to saying that $a''_0 = \epsilon = a'_0$. Therefore, in this case also, Condition (7) holds for i = 0.

For the induction step, let $m < l_1$, and suppose that Conditions (6)–(8) are satisfied for all $i \leq m$. Let also $o_{1,m+1}, \ldots, o_{k,m+1} \in O$ be the counter operations such that

$$\delta(n_m, a'_m, \bar{c}_{1,m}, \dots, \bar{c}_{k,m}) = (n_{m+1}, o_{1,m+1}, \dots, o_{k,m+1}).$$
(10)

By definition of the sequence $\mathcal{C}_k(w)$, $a'_m \in \Sigma \cup \{\epsilon\}$ and the counter operations satisfy 560

$$c_{1,m+1} = o_{1,m+1}(c_{1,m}), \dots, c_{k,m+1} = o_{k,m+1}(c_{k,m}).$$
(11)

By the induction hypothesis (Condition (7)), $a''_m = a'_m$. The definition of a''_m ensures that the cell c_4 of one and only one detection module $\text{DET}(q, a''_m, \bar{c}_1, \ldots, \bar{c}_k)$ is activated between time steps t_m and t_{m+1} , for some $q \in Q$ and some $\bar{c}_1, \ldots, \bar{c}_k \in C$. But by the induction hypotheses (Conditions (6) and (8)), at time step $t_m - 1$, one has

$$w_{s,m} = a_{min} + n_m \cdot \eta \tag{12}$$

$$w_{c_j,m} = r_{c_{j,m}}, \text{ for all } j = 1, \dots, k.$$
 (13)

Hence, Relations (12)–(13) and Algorithm 1 (lines 22–23 and 40–46) ensure that the module $\text{DET}(n_m, a''_m, \bar{c}_{1,m}, \dots, \bar{c}_{k,m})$, and only this one, has its cell c_4 activated between time steps t_m and t_{m+1} . By Relation (10), the cell c_4 of this detection module is connected to cell

$$c_{3((n-1)-(n_{m+1}-n_m))+1} \quad \text{if} \quad n_{m+1}-n_m > 0 \quad \text{or}$$

$$\bar{c}_{3((n-1)-(n_{m+1}-n_m))+1} \quad \text{if} \quad n_{m+1}-n_m < 0$$

of the state module ST (Algorithm 1, lines 27–31). Hence, the activation of this detection module between t_m and t_{m+1} induces subsequent spiking patterns of the state module which, by construction, increments (if $n_{m+1} - n_m > 0$) or decrements (if $n_{m+1} - n_m < 0$) the synaptic weight $w_s(t)$ by $|n_{m+1} - n_m| \cdot \eta$, and hence, changes it from its current value $a_{min} + n_m \cdot \eta$ (cf. Eq (12)) to the new value $a_{min} + n_m \cdot \eta + (n_{m+1} - n_m) \cdot \eta = a_{min} + n_{m+1} \cdot \eta$. Note that each spiking pattern takes 3 time steps, and hence, the updating of $w_s(t)$ takes at most 3(n-1) time steps, where n is the number of states of C_k (the longest update being when $|n_{m+1} - n_m| = n - 1$, which takes 3(n-1) time steps). Therefore, at time $t_{m+1} - 1$, one has

$$w_{s,m+1} = a_{min} + n_{m+1} \cdot \eta.$$

This shows that Condition (6) is satisfied for i = m + 1.

Similarly, by Relation (10), the cell c_4 of the module $\text{DET}(n_m, a''_m, \bar{c}_{1,m}, \ldots, \bar{c}_{k,m})$ is connected to cells *push* or *pop* of the counter module C(j) depending on whether $o_{j,m+1} == push$ or $o_{j,m+1} == pop$, respectively, for $j = 1, \ldots, k$ (Algorithm 1, lines 32–38). Hence, the activation of the detection module $\text{DET}(n_m, a''_m, \bar{c}_{1,m}, \ldots, \bar{c}_{k,m})$ between t_m and t_{m+1} induces subsequent activations of the counter modules which, by construction, change the synaptic weights $w_{c_j}(t)$ from their current value $r_{c_{j,m}}$ to $r_{o_{j,m+1}(c_{j,m})}$, for $j = 1, \ldots, k$. Note that the updating of each $w_{c_j}(t)$ takes only 3 time steps. Consequently, at time $t_{m+1} - 1$, one has

$$w_{c_{j,m+1}} = r_{o_{j,m+1}(c_{j,m})}, \text{ for } j = 1, \dots, k$$

By Relation (11), these equations can be rewritten as

$$w_{c_{j,m+1}} = r_{c_{j,m+1}}, \text{ for } j = 1, \dots, k.$$

This shows that Condition (8) is satisfied for i = m + 1.

We now show that Condition (7) holds for i = m + 1. By the induction hypothesis, one has $(a'_i)_{i=0}^m = (a''_i)_{i=0}^m$. We must prove that $a'_{m+1} = a''_{m+1}$. By definition, elements from $(a'_i)_{i=0}^m$ and $(a''_i)_{i=0}^m$ belong to $\Sigma \cup \{\epsilon\}$. Let $(a'_{i_j})_{j=0}^{p_1}$ (with $p_1 \leq m$) and $(a''_{i_j})_{j=0}^{p_2}$ (with $p_2 \leq m$) be the subsequences formed by the non-empty symbols of $(a'_i)_{i=0}^m$ and $(a''_i)_{i=0}^m$, respectively. The induction hypothesis ensures that $p_1 = p_2 = p'$ and

$$(a'_{ij})_{j=0}^{p'} = (a''_{ij})_{j=0}^{p'}.$$
(14)

Moreover, by definition again, $(a'_i)_{i=0}^m$ is the sequence of empty and non-empty symbols processed by C_k during the m + 1 first steps of its computation over input $w = a_0 \cdots a_p$ (cf. Eq (4)). Hence, the subsequence of its non-empty symbols $(a'_{i_j})_{j=0}^{p'}$ corresponds precisely to the p' + 1 successive letters of w, i.e.,

$$(a'_{ij})_{j=0}^{p'} = (a_i)_{i=0}^{p'}.$$
(15)

The fact that ϵ symbols of $(a'_i)_{i=0}^m$ vanish within concatenation together with Relation (15) yield the following equalities

$$a'_{0}\cdots a'_{m} = a'_{i_{0}}\cdots a'_{i_{p'}} = a_{0}\cdots a_{p'}.$$
(16)

Also, Relations (14) and (15) directly imply

$$(a_{i_j}')_{j=0}^{p'} = (a_i)_{i=0}^{p'}.$$
(17)

Besides, as already mentioned, at time $t_0 - 1$, cell c_{14} of module IN1 holds the encoding of input $w = a_0 \cdots a_p$ (considered as a stack). Between times t_0 and $t_{m+1} - 1$, the elements of $(a_i'')_{i=0}^m$ are successively processed by \mathcal{N} (cf. Eq (5)). During this time interval, 585

582

569

the successive non-empty symbols of $(a_i'')_{i=0}^m$, i.e., the elements of $(a_{i_j}'')_{j=0}^{p'} = (a_i)_{i=0}^{p'}$ (cf. Relation (17)), are successively popped from $w = a_0 \cdots a_p$ and the remaining string stored in cell c_{14} . Consequently, at time $t_{m+1} - 1$, cell c_{14} holds the encoding of the remaining string $a_{p'+1} \cdots a_p$, and thus, its top element is $a_{p'+1}$.

From this point onwards, the proof of Relation (7) for the case i = 0 can be adapted to the present situation. In short, consider $a'_{m+1} \in \Sigma \cup \{\epsilon\}$. Then either $a'_{m+1} \in \Sigma$ or $a'_{m+1} = \epsilon$. Note that in case $a'_{m+1} \in \Sigma$, Relation (16) ensures that $a'_{m+1} = a_{p'+1}$. Taking this fact into account and replacing variables $a_0, a'_0, a''_0, n_0, \bar{c}_{1,0}, \ldots, \bar{c}_{k,0}$ of the previous argument by $a_{m+1}, a'_{m+1}, a''_{m+1}, n_{m+1}, \bar{c}_{1,m+1}, \ldots, \bar{c}_{k,m+1}$, respectively, leads to $a''_{m+1} = a'_{m+1}$. Therefore, Condition (7) holds for i = m + 1.

Finally, we show that STDP-based RNNs are Turing complete. Let \mathcal{N} be an STDPbased RNN. Let also $acc, rej \in \mathbb{Q}$ be two specific values for $w_s(t)$. For any binary input $w = a_0 \cdots a_p \in \Sigma^*$, we say that w is *accepted* (resp. *rejected*) by \mathcal{N} if the sequence $\mathcal{N}(u_w)$ is finite, and its last element $(w_{s,l_2}, a''_{l_2}, w_{c_{1,l_2}}, \ldots, w_{c_{k,l_2}})$ satisfies $a''_{l_2} = \epsilon$ and $w_{s,l_2} = acc$ (resp. $w_{s,l_2} = rej$). The language recognized by \mathcal{N} , denoted by $L(\mathcal{N})$, is the set of inputs accepted by \mathcal{N} . A language $L \subseteq \Sigma^*$ is recognizable by some STDP-based RNN if there exists some STDP-based RNN \mathcal{N} such that $L(\mathcal{N}) = L$.

Corollary 1. Let $L \subseteq \Sigma^*$ be some language. The language L is recognizable by some Turing machine if and only if L is recognizable by some STDP-based RNN.

Proof. Suppose that L is recognizable by some STDP-based RNN \mathcal{N} . The construction described in Algorithm 1 ensures that \mathcal{N} can be simulated by some Turing machine \mathcal{M} . Hence, L is recognizable by some Turing machine \mathcal{M} . Conversely, suppose that L is recognizable by some Turing machine \mathcal{M} . Then L is also recognizable by some 2-counter machine \mathcal{C}_2 [57]. By Theorem 1, L is recognizable by some STDP-based RNN \mathcal{N} . \Box

Simulations

We now illustrate the correctness of our construction by means of computer simulations. 611

First, let us recall that the 2-counter machine of Fig 4 recognizes the recursively 612 enumerable (but non context-free and non regular) language $\{0^n 1^n 0^n : n > 0\}$, i.e., 613 the sequences of bits beginning with a strictly positive number of 0's followed by the 614 same number of 1's and followed again by the same number of 0's. For instance, inputs 615 $w_1 = 001100$ and $w_2 = 0011101$ are respectively accepted and rejected by the machine. 616 Based on the previous considerations, we implemented an STDP-based RNN simulating 617 this 2-counter machine. The network contains 390 cells connected together according to 618 the construction given by Algorithm 1. We also set $a_{min} = \eta = 0.1$ in the STDP rule 619 of Eq (2). Two computations of this network over an accepting and a rejecting input 620 stream are illustrated in Fig 15 and Fig 16. These simulations illustrate the correctness 621 of the construction described in Algorithm 1. 622

More specifically, the computation of the network over the input stream

$$u_{w_1} = \mathbf{001100} \operatorname{end} \underbrace{\emptyset \cdots \emptyset}_{K'_6 = 23} \operatorname{tic}_0 \underbrace{\emptyset \cdots \emptyset}_{K = 29} \operatorname{tic}_1 \underbrace{\emptyset \cdots \emptyset}_{K = 29} \operatorname{tic}_2 \cdots$$

which corresponds to the encoding of $w_1 = 001100$, is displayed in Fig 15. In this case, taking K = 17 + 3(5 - 1) = 29 suffices for the correctness of the simulation (since the largest possible state update, in terms of the states' indices, is a change from q_5 to q_1). The lower raster plot displays the spiking activities of some of the cells of the network belonging to the input encoding module (in_0, in_1, end, tic) , the input transmission module (u_0, u_1, u_{ϵ}) , the state module $(pres_s, post_s)$ and the two counter modules $(push, pop, test, pre_{c_k}, post_{c_k}, = 0, \neq 0$, for k = 1, 2).

From time step t = 0 to t = 6, the encoding of the input stream 001100 is transmitted 630 to the network via activations of cells in_0 , in_1 and end (blue pattern). Between t = 6631 and t = 30, the input pattern is encoded into activation values of sigmoid cells in the 632 input encoding module, as illustrated in Fig 7. From t = 30 onwards, the *tic* cell is 633 activated every 30 time steps in order to trigger the successive computational steps of the 634 network. Each spike of the *tic* cell induces a subsequent spike of u_{ϵ} one time step later. 635 At this moment, the network tries to simulate an ϵ -transition of the counter machine. 636 If such a transition is possible, the network performs it: this is the case at time steps 637 t = 31, 181. Otherwise, the input encoding module retrieves the next input bit to be 638 processed, and activates the corresponding cell u_0 or u_1 (blue pattern): this is the case 639 at time steps t = 71, 101, 131, 161, 221, 251. In Fig 15 (cells u_0, u_1, u_{ϵ}), we can see that 640 on this input stream, the network processes the sequence of input symbols $\epsilon 0011\epsilon 00$. 641

Every time the network receives an input symbol (ϵ , 0 or 1), it simulates one transition 642 of the counter machine associated to this input. The successive computational states of 643 the machine are encoded into the successive values taken by $w_s(t)$ (cf. Fig 15, red curve 644 in the upper graph). The changes in these synaptic weights are induced by the spiking 645 patterns of cells pre_s and $post_s$ (red patterns). The successive counter states of the 646 machine, i.e., 'zero' or 'non-zero', are given by the activations of cells '= 0' or ' \neq 0' of 647 the counter modules, respectively (black patterns). The consecutive counter operations 648 are given by the activations of cells *push*, *pop* and *test* (black patterns). The successive 649 counter values of the machine are encoded into the successive values taken by $w_{c_1}(t)$ and 650 $w_{c_2}(t)$ (orange curves of the upper graph). The changes in these synaptic weights are 651 induced by the spiking pattern of cells pre_{c_i} and $post_{c_i}$, for j = 1, 2 (orange patterns). 652 The pics along these curves are caused by the testing procedures which increment and 653 decrement back the values of the synapses without finally modifying their current values 654 (cf. description of the counter module). 655

The computation of the network over input stream u_{w_1} can be described by the successive synaptic weights $(w_s(t), w_{c_1}(t), w_{c_2}(t))$ at time steps t = 30k, for $1 \le k \le 10$.

In this case, one has

$$\begin{pmatrix} w_s(t) \\ w_{c_1}(t) \\ w_{c_2}(t) \end{pmatrix} = \begin{pmatrix} 0.1 \\ 0.0 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.0 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.5 \\ 0.5 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.6 \\ 0.75 \\ 0.5 \end{pmatrix} \begin{pmatrix} 0.6 \\ 0.75 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.4 \\ 0.75 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.4 \\ 0.5 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.4 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.4 \\ 0.0 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.4 \\ 0.0$$

Recall that state n and counter value x of C_k are encoded by the synaptic weights $w_s(t) = a_{min} + n \cdot \eta$ and $w_c(t) = r_x$ in \mathcal{N} , respectively. Accordingly, the previous values correspond to the encodings of the following states and counter values (q, c_1, c_2) of the counter machine:

$$\begin{pmatrix} q\\c_1\\c_2 \end{pmatrix} = \begin{pmatrix} 0\\0\\0 \end{pmatrix} \begin{pmatrix} 2\\0\\0 \end{pmatrix} \begin{pmatrix} 2\\1\\1 \end{pmatrix} \begin{pmatrix} 2\\2\\2 \end{pmatrix} \begin{pmatrix} 5\\2\\1 \end{pmatrix} \begin{pmatrix} 5\\2\\0 \end{pmatrix} \begin{pmatrix} 3\\2\\0 \end{pmatrix} \begin{pmatrix} 3\\1\\0 \end{pmatrix} \begin{pmatrix} 3\\0\\0 \end{pmatrix} \begin{pmatrix} 4\\0\\0 \end{pmatrix}$$

These are the correct computational states and counter values encountered by the machine along the computation of input $w_1 = 001100$ (cf. Fig 4). Therefore, the network simulates the counter machine correctly. The fact that the computations of the machine and the network terminate in state 4 and with $w_s(t) = 0.5 = 0.1 + 4 \cdot \eta$, respectively, means that inputs w_1 and u_{w_1} are accepted by both systems.

As another example, the computation of the network over the input stream

$$u_{w_2} = \mathbf{0011101} \operatorname{end} \underbrace{\emptyset \cdots \emptyset}_{23} \operatorname{tic}_{\mathbf{0}} \underbrace{\emptyset \cdots \emptyset}_{29} \operatorname{tic}_{\mathbf{1}} \underbrace{\emptyset \cdots \emptyset}_{29} \operatorname{tic}_{\mathbf{2}} \cdots$$

which corresponds to the encoding of $w_2 = 0011101$, is displayed in Fig 16 (cells u_0, u_1, u_{ϵ}). We see that on this input stream, the network processes the sequence of input symbols $\epsilon 0011\epsilon 101$. The successive synaptic weights $(w_s(t), w_{c_1}(t), w_{c_2}(t))$ at time steps t = 30k, for $1 \le k \le 10$ are

$$\begin{pmatrix} w_s(t) \\ w_{c_1}(t) \\ w_{c_2}(t) \end{pmatrix} = \begin{pmatrix} 0.1 \\ 0.0 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.0 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.5 \\ 0.5 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.6 \\ 0.75 \\ 0.5 \end{pmatrix} \begin{pmatrix} 0.6 \\ 0.75 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.4 \\ 0.75 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.0 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.75 \\ 0.75 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.75 \\ 0.$$

These values correspond to the encodings of the following states and counter values (q, c_1, c_2) of the counter machine:

$$\begin{pmatrix} q\\c_1\\c_2 \end{pmatrix} = \begin{pmatrix} 0\\0\\0 \end{pmatrix} \begin{pmatrix} 2\\0\\0 \end{pmatrix} \begin{pmatrix} 2\\1\\1 \end{pmatrix} \begin{pmatrix} 2\\2\\2 \end{pmatrix} \begin{pmatrix} 5\\2\\1 \end{pmatrix} \begin{pmatrix} 5\\2\\0 \end{pmatrix} \begin{pmatrix} 3\\2\\0 \end{pmatrix} \begin{pmatrix} 1\\2\\0 \end{pmatrix} \begin{pmatrix} 1\\2\\0$$

These are the correct computational states and counter values encountered by the machine working over input $w_2 = 0011101$ (cf. Fig 4). Therefore, the network simulates the counter machine correctly. The fact that the computations of the machine and the network terminate in state 1 and with $w_s(t) = 0.2 = 0.1 + 1 \cdot \eta$, respectively, means that inputs w_1 and u_{w_1} are rejected by both systems.

Discussion

666

We proposed a novel Turing complete paradigm of neural computation where the essential information is encoded into discrete synaptic levels rather than into spiking configurations, 668



Fig 15. Simulation 1. Computation of the STDP-based RNN simulating the 2-counter machine of Fig 4 over input 001100. The lower graph is a raster plot displaying the spiking patterns of some of the cells of the network belonging to the input encoding module (cells in_0, in_1, end, tic), the input transmission module (cells u_0, u_1, u_{ϵ}), the state module (cells $pres_s, post_s$) and the two counter modules (cells $push, pop, test, pre_{c_k}, post_{c_k}, = 0, \neq 0$, for k = 1, 2). The upper graph displays the evolution of the synaptic weights $w_s(t)$ (red curve) and $w_{c_1}(t), w_{c_2}(t)$ (orange curves) over time. The red curve is displayed relatively to the left-hand scale (ranging from 0 to 1). The two orange curves are are displayed relatively to the upper and lower right-hand scales, respectively (both ranging from 0 to 1). The evolution of $w_s(t)$ and $w_{c_1}(t), w_{c_2}(t)$ (red and orange curves) represent the encodings of the successive states and counter values of the 2-counter machine, respectively.

activation values or (attractor) dynamics of neurons. More specifically, we showed that 669 any 2-counter machine—and thus any Turing machine—can be simulated by a recurrent 670 neural network subjected to two kinds of spike-timing-dependent plasticity (STDP) 671 mechanisms. The finitely many computational states and infinitely many counter values 672 of the machine are encoded into finitely and infinitely many synaptic levels, respectively. 673 The transitions between states and counter values are achieved via the two STDP rules. 674 In short, the network operates as follows. First, the input stream is encoded and stored 675 into the activation value of a specific analog neuron. Then, every time a *tic* input 676 signal is received, the network tries to simulate an ϵ -transition of the machine. If such a 677 transition is possible, the network simulates it. Otherwise, the network retrieves from its 678 memory the next input bit to be processed, and simulates a regular transition associated 679 with this input. These results have been illustrated by means of computer simulations. 680 An STDP-based recurrent neural network simulating a specific 2-counter machine has 681 been implemented and its dynamics analyzed. 682

We emphasize once again that the possibility to simulate ϵ -transitions is (unfortunately) necessary to the achievement of Turing completeness. Indeed, it is well-known that the class of k-counter machines that do not make use of ϵ -transitions is not Turing



Fig 16. Simulation 2. Computation of the STDP-based RNN simulating the 2-counter machine of Fig 4 over input 0011101.

complete, for any k > 0. For instance, the language $L = \{w \# w : w \in \{0,1\}^*\}$ (the strings 686 of bits separated by a symbol # whose prefix and suffix are the same), is recursively 687 enumerable, but cannot be recognized by a k-counter machine without ϵ -transitions. The 688 input encoding module, as intricate as it is, ensures the implementation of this feature. It 689 encodes and stores the incoming input stream so as to be able to subsequently intersperse 690 the successive regular transitions (associated to regular input symbols) with ϵ -transitions 691 (associated to ϵ symbols). By contrast, a k-counter machine without ϵ -transitions could 692 be simulated by an STDP-based neural network working in an online fashion. The 693 successive input symbols would be processed as they arrive, and a regular transition be 694 simulated for each successive symbol. An STDP-based neural net (as described in Fig 5) 695 without input encoding module could simulate a k-counter machine without ϵ -transitions. 696 One would just need to add sufficiently many delay layers to its input transmission 697 module in order to have enough time to emulate each regular transition. 698

In the present context, the STDP-based RNNs are capable of simulating Turing 699 machines working in the *accepting mode* (i.e., machines that provide accepting or rejecting 700 decisions of their inputs by halting in an accepting or a rejecting state, respectively). But 701 it would be possible to adapt the construction to simulate Turing machines working also 702 in the *generative mode* (i.e., machines that write the successive words of a language on 703 their output tape, in an enumerative way). To this end, we would need to simulate the 704 program and work tape of \mathcal{M} by an STDP-based RNN \mathcal{N} (as described in Theorem 1), 705 and the output tape of \mathcal{M} by an additional neural circuit \mathcal{N}_{out} plugged to \mathcal{N} . Broadly 706 speaking, the simulation process could be achieved as follows: 707

• Every non-output move of $\mathcal M$ is simulated by the STDP-based RNN $\mathcal N$ in the $_{708}$

usual way (cf. Theorem 1).

- Every time \mathcal{M} is generating a new word $w = a_1 \cdots a_n$ on its output tape, use the recircuit \mathcal{N}_{out} to build step by step the encoding $\bar{r}_w = \sum_{i=1}^n \frac{2a_i+1}{4^i} \in [0,1]$ of w and store this value in a designated neuron c (as described in the paragraph "Input encoding module").
- When \mathcal{M} has finished generating w, use the circuit \mathcal{N}_{out} to transfer the value \bar{r}_w of c to another neuron c', to set the activation value of c back to 0, and to output the successive bits of w by popping the the stack \bar{r}_w stored in c' (again, as described in the paragraph "Input encoding module").

In this way, the STDP-based RNN \mathcal{N} plugged to the circuit \mathcal{N}_{out} could work as a language generator: it outputs bit by bit the successive words of the language L generated by \mathcal{M} . The implementation of the circuit \mathcal{N}_{out} is along the lines of what is described in the paragraph "input encoding module".

Concerning the complexity issue, our model uses $\mathcal{O}(n)$ neurons and $\mathcal{O}(n)$ synapses to 722 simulate a counter machine with n states. Moreover, the simulation works in real-time, 723 since every computational step of the counter machine can be simulated in a fixed amount 724 of 17 + 3(n-1) time steps (17 time steps to transmit the next input bit up to the end of 725 the detection modules, and at most 3(n-1) time steps to perform the state and counter 726 updates). In the context of rational-weighted sigmoidal neural networks, the seminal 727 result from Siegelmann and Sontag uses 886 Boolean and analog neurons to simulate 728 a universal Turing machine [4]. Recent results show that Turing completeness can be 729 achieved with a minimum of 3 analog neurons only, the other ones being Boolean [58]. 730 As for spiking neural P systems, Turing universality can be achieved with 3 or 4 neurons 731 only, but this comes at the price of exponential time and space overheads (see [59], Table 732 1). In our case, the complexity of Turing universality is expected to be investigated in 733 detail in a future work. 734

Regarding synaptic-based computation, a somehow related approach has already been 735 pursued in the P system framework with the consideration of spiking neural P systems 736 with rules on synapses [60]. In this case, synapses are considered as computational units 737 triggering exchanges of spikes between neurons. The proposed model is shown to be 738 Turing universal. It is claimed that "placing the spiking and forgetting rules on synapses 739 proves to be a powerful feature, both simpler proofs and smaller universal systems are 740 obtained in comparison with the case when the rules are placed in the neurons" [60]. In 741 this context however, the information remains encoded into the number of spikes hold 742 by the neurons, referred to as the "configuration" of the system. By contrast, in our 743 framework, the essential information—the computational states and counter values—is 744 encoded into discrete synaptic levels, and their updates achieved via synaptic plasticity 745 rules. 746

As already mentioned, it has been argued that in biological neural networks "synapses 747 change their strength by jumping between discrete mechanistic states rather than by 748 simply moving up and down in a continuum of efficacy" [56]. These considerations 749 represent "a new paradigm for understanding the mechanistic underpinnings of synaptic 750 plasticity, and perhaps also the roles of such plasticity in higher brain functions" [56]. In 751 addition, "much work remains to be done to define and understand the mechanisms and 752 roles these states play" [56]. In our framework, the computational states and counter 753 values of the machine are encoded into discrete synaptic states. However, the input 754 stream to be processed is still encoded into the activation value of a specific analog 755 neuron. It would be interesting to develop a paradigm where this feature also is encoded 756 into synapses. Moreover, it would be interesting to extend the proposed paradigm of 757 computation to the consideration of more biological STDP rules. 758

It is worth noting that synaptic-based and neuron-based computational paradigms are not opposite conceptions, but intertwined processes instead. Indeed, changes in synaptic states are achieved via the elicitation of specific neuronal spiking patterns (which modify the synaptic strengths via STDP). The main difference between these two conceptions is whether the essential information is encoded and memorized into synaptic states or into spiking configurations, activation values or (attractor) dynamics of neurons. 769

In biology, real brain circuits do certainly not operate by simulating abstract finite 765 state machines. And with our work, we do intend to argue in this sense. Rather, our 766 intention is to show that a bio-inspired Turing complete paradigm of abstract neural 767 computation—centered on the concept of synaptic plasticity—is not only theoretically 768 possible, but also potentially exploitable. The idea of representing and storing essential 769 information into discrete synaptic levels is, we believe, novel and worthy of consideration. 770 It represents a paradigm shift in the field of neural computation. 771

Finally, the impacts of the proposed approach are twofold. From a practical perspec-772 tive, contemporary developments in neuromorphic computing provide the possibility to 773 implement neurobiological architectures on very-large-scale integration (VLSI) systems, 774 with the aim of mimicking neuronal circuits present in the nervous system [61, 62]. The 775 implementation of our model on VLSI technologies would lead to the realization of new 776 kinds of analog neuronal computers. The computational and learning capabilities of 777 these neural systems could then be studied directly from the hardware point of view. 778 And the integrated circuits implementing our networks might be suitable for specific 779 applications. Besides, from a Machine Learning (ML) perspective, just as the dynamics 780 of biological neural nets inspired neuronal-based learning algorithms, in this case also, 781 the STDP-based recurrent neural networks might eventually lead to the development of 782 new ML algorithms. 783

From a theoretical point of view, we hope that the study of neuro-inspired paradigms 784 of abstract computation might contribute to the understanding of both biological and 785 artificial intelligences. We believe that similarly to the foundational work from Turing, 786 which played a crucial role in the practical realization of modern computers, further 787 theoretical considerations about neural- and natural-based models of computation shall 788 contribute to the emergence of novel computational technologies, and step by step, open 789 the way to the next computational generation. 790

Supporting information

S1 Files. Python code. All python scripts generating the results of the paper are provided in an attached zip folder files.zip. The description of the different files is given in Read_me.txt. 794

Acknowledgments

Supports from DARPA – Lifelong Learning Machines (L2M) program, cooperative agreement No. HR0011-18-2-0023, as well as from the Czech Science Foundation, grant No. GA19-05704S are gratefully acknowledged. We warmly thank Brigitte Quenet for insightful discussions about synaptic computation as well as for the implementation of the counter module. We also thank Hélène Oppenheim-Gluckman for precious (silent) advices.

References

- 1. McCulloch WS, Pitts W. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysic. 1943;5:115–133.
- Kleene SC. Representation of events in nerve nets and finite automata. In: Shannon C, McCarthy J, editors. Automata Studies. Princeton, NJ: Princeton University Press; 1956. p. 3–41.
- 3. Minsky ML. Computation: finite and infinite machines. Englewood Cliffs, N. J.: Prentice-Hall, Inc.; 1967.
- Siegelmann HT, Sontag ED. On the computational power of neural nets. J Comput Syst Sci. 1995;50(1):132–150.
- Siegelmann HT, Sontag ED. Analog computation via neural networks. Theor Comput Sci. 1994;131(2):331–360.
- Cabessa J, Siegelmann HT. Evolving recurrent neural networks are super-Turing. In: Proceedings of IJCNN 2011. IEEE; 2011. p. 3200–3206.
- Cabessa J, Siegelmann HT. The Super-Turing Computational Power of plastic Recurrent Neural Networks. Int J Neural Syst. 2014;24(8).
- Síma J, Orponen P. General-Purpose Computation with Neural Networks: A Survey of Complexity Theoretic Results. Neural Computation. 2003;15(12):2727– 2778.
- 9. Elman JL. Finding Structure in Time. Cognitive Science. 1990;14(2):179–211.
- Pollack JB. The Induction of Dynamical Recognizers. Machine Learning. 1991;7:227–252.

791

- Indyk P. Optimal Simulation of Automata by Neural Nets. In: Mayr EW, Puech C, editors. STACS. vol. 900 of Lecture Notes in Computer Science. Springer; 1995. p. 337–348.
- 12. Horne BG, Hush DR. Bounds on the complexity of recurrent neural network implementations of finite state machines. Neural Networks. 1996;9(2):243–252.
- Siegelmann HT. Recurrent Neural Networks and Finite Automata. Computational Intelligence. 1996;12:567–574.
- Maass W. Computing with Spiking Neurons. In: Maass W, Bishop CM, editors. Pulsed Neural Networks. Cambridge, MA, USA: MIT Press; 1999. p. 55–85.
- Maass W, Bishop CM, editors. Pulsed Neural Networks. Cambridge, MA, USA: MIT Press; 1999.
- 16. Păun G. Computing with Membranes. J Comput Syst Sci. 2000;61(1):108-143.
- 17. Păun G. Membrane Computing. An Introduction. Berlin: Springer-Verlag; 2002.
- 18. The P Systems Webpage;. Available from: http://ppage.psystems.eu/.
- Neumann Jv. The computer and the brain. New Haven, CT, USA: Yale University Press; 1958.
- Kilian J, Siegelmann HT. The dynamic universality of sigmoidal neural networks. Inf Comput. 1996;128(1):48–56.
- 21. Hyötyniemi H. Turing machines are recurrent neural networks. In: Alander J, Honkela T, M J, editors. STeP '96 - Genes, Nets and Symbols; Finnish Artificial Intelligence Conference, Vaasa 20-23 Aug. 1996. Vaasa, Finland: University of Vaasa, Finnish Artificial Intelligence Society (FAIS); 1996. p. 13–24.
- Balcázar JL, Gavaldà R, Siegelmann HT. Computational power of neural networks: a characterization in terms of Kolmogorov complexity. IEEE Transactions on Information Theory. 1997;43(4):1175–1183.
- 23. Neto JaPG, Siegelmann HT, Costa JF, Araujo CPS. Turing Universality of Neural Nets (Revisited). In: EUROCAST '97: Proceedings of the A Selection of Papers from the 6th International Workshop on Computer Aided Systems Theory. London, UK: Springer-Verlag; 1997. p. 361–366.
- 24. Siegelmann HT. Neural networks and analog computation: beyond the Turing limit. Cambridge, MA, USA: Birkhauser Boston Inc.; 1999.
- 25. Cabessa J, Duparc J. Expressive Power of Non-deterministic Evolving Recurrent Neural Networks in Terms of Their Attractor Dynamics. In: Calude CS, Dinneen MJ, editors. Unconventional Computation and Natural Computation 14th International Conference, UCNC 2015, Auckland, New Zealand, August 30 -

September 3, 2015, Proceedings. vol. 9252 of Lecture Notes in Computer Science. Springer; 2015. p. 144–156.

- 26. Cabessa J, Duparc J. Expressive Power of Nondeterministic Recurrent Neural Networks in Terms of their Attractor Dynamics. IJUC. 2016;12(1):25–50.
- 27. Cabessa J, Finkel O. Expressive Power of Evolving Neural Networks Working on Infinite Input Streams. In: Klasing R, Zeitoun M, editors. Fundamentals of Computation Theory - 21st International Symposium, FCT 2017, Bordeaux, France, September 11-13, 2017, Proceedings. vol. 10472 of Lecture Notes in Computer Science. Springer; 2017. p. 150–163.
- 28. Cabessa J, Siegelmann HT. The Computational Power of Interactive Recurrent Neural Networks. Neural Computation. 2012;24(4):996–1019.
- 29. Cabessa J, Villa AEP. The expressive power of analog recurrent neural networks on infinite input streams. Theor Comput Sci. 2012;436:23–34.
- Cabessa J, Villa AEP. The Super-Turing Computational Power of Interactive Evolving Recurrent Neural Networks. In: et al VM, editor. Proceedings of ICANN 2013. vol. 8131 of Lecture Notes in Computer Science. Springer; 2013. p. 58–65.
- Cabessa J, Villa AEP. Interactive Evolving Recurrent Neural Networks Are Super-Turing Universal. In: et al SW, editor. Proceedings of ICANN 2014. vol. 8681 of Lecture Notes in Computer Science. Springer; 2014. p. 57–64.
- Cabessa J, Villa AEP. Computational capabilities of recurrent neural networks based on their attractor dynamics. In: 2015 International Joint Conference on Neural Networks, IJCNN 2015, Killarney, Ireland, July 12-17, 2015. IEEE; 2015. p. 1–8.
- 33. Cabessa J, Villa AEP. Recurrent neural networks and super-Turing interactive computation. In: Koprinkova-Hristova P, Mladenov V, Kasabov KN, editors. Artificial Neural Networks: Methods and Applications in Bio-/Neuroinformatics. Springer; 2015. p. 1–29.
- 34. Cabessa J, Villa AEP. On Super-Turing Neural Computation. In: Liljenström H, editor. Advances in Cognitive Neurodynamics (IV): Proceedings of the Fourth International Conference on Cognitive Neurodynamics - 2013. Dordrecht: Springer Netherlands; 2015. p. 307–312.
- Cabessa J, Villa AEP. Expressive power of first-order recurrent neural networks determined by their attractor dynamics. Journal of Computer and System Sciences. 2016;82(8):1232–1250.
- Turing AM. Intelligent Machinery. Teddington, UK: National Physical Laboratory; 1948.

- Rosenblatt F. The perceptron: A perceiving and recognizing automaton. Ithaca, New York: Cornell Aeronautical Laboratory; 1957. 85-460-1.
- Hebb DO. The organization of behavior : a neuropsychological theory. John Wiley & Sons Inc.; 1949.
- Rosenblatt F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review. 1958;65(6):386–408.
- Widrow B. The Speed of Adaption in Adaptive Control Systems. In: American Rocket Society (ARS) Guidance, Control and Navigation Conference Proceedings; 1961. p. 1933–1961.
- Minsky ML, Papert S. Perceptrons: An Introduction to Computational Geometry. Cambridge, MA, USA: MIT Press; 1969.
- Schmidhuber J. Deep learning in neural networks: An overview. Neural Networks. 2015;61:85–117.
- Abbott LF, Nelson SB. Synaptic plasticity: taming the beast. Nat Neurosci. 2000;3 Suppl.:1178–1183.
- 44. Markram H, Lübke J, Frotscher M, Sakmann B. Regulation of Synaptic Efficacy by Coincidence of Postsynaptic APs and EPSPs. Science. 1997;275(5297):213–215.
- Caporale N, Dan Y. Spike timing-dependent plasticity: a Hebbian learning rule. Annu Rev Neurosci. 2008;31:25–46.
- Sjöström J, Gerstner W. Spike-timing dependent plasticity. Scholarpedia. 2010;5(1):1362.
- 47. Abeles M. Local Cortical Circuits. An Electrophysiological Study. vol. 6 of Studies of Brain Function. Berlin Heidelberg New York: Springer-Verlag; 1982.
- Abeles M. Corticonics: Neuronal Circuits of the Cerebral Cortex. 1st ed. Cambridge University Press; 1991.
- 49. Abeles M. Time Is Precious. Science. 2004;304(5670):523-524.
- Ikegaya Y, Aaron G, Cossart R, Aronov D, Lampl I, Ferster D, et al. Synfire Chains and Cortical Songs: Temporal Modules of Cortical Activity. Science. 2004;304(5670):559–564.
- Mainen ZF, Sejnowski TJ. Reliability of spike timing in neocortical neurons. Science. 1995;268(5216):1503–1506.
- Zheng P, Triesch J. Robust development of synfire chains from multiple plasticity mechanisms. Front Comput Neurosci. 2014;8(66).

- Izhikevich EM. Polychronization: computation with spikes. Neural Computation. 2006;18(2):245–82.
- Szatmáry B, Izhikevich EM. Spike-Timing Theory of Working Memory. PLoS Computational Biology. 2010;6(8):e1000879.
- Jun JK, Jin DZ. Development of Neural Circuitry for Precise Temporal Sequences through Spontaneous Activity, Axon Remodeling, and Synaptic Plasticity. PLOS ONE. 2007;2(8):1–17.
- Montgomery JM, Madison DV. Discrete synaptic states define a major mechanism of synapse plasticity. Trends in Neurosciences. 2004;27(12):744 –750.
- Hopcroft JE, Motwani R, Ullman JD. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 2006.
- Šíma J. Three Analog Neurons Are Turing Universal. In: Fagan D, Martín-Vide C, O'Neill M, Vega-Rodríguez MA, editors. Theory and Practice of Natural Computing - 7th International Conference, TPNC 2018, Dublin, Ireland, December 12-14, 2018, Proceedings. vol. 11324 of Lecture Notes in Computer Science. Springer; 2018. p. 460–472.
- 59. Neary T. Three small universal spiking neural P systems. Theor Comput Sci. 2015;567:2–20.
- Song T, Pan L, Păun G. Spiking neural P systems with rules on synapses. Theoretical Computer Science. 2014;529:82–95.
- Mead C. Neuromorphic electronic systems. Proceedings of the IEEE. 1990;78(10):1629–1636. doi:10.1109/5.58356.
- Monroe D. Neuromorphic Computing Gets Ready for the (Really) Big Time. Commun ACM. 2014;57(6):13–15. doi:10.1145/2601069.