Finite-state computation in analog neural networks: steps towards biologically plausible models?

Mikel L. Forcada and Rafael C. Carrasco

Departament de Llenguatges i Sistemes Informàtics,
Universitat d'Alacant,
E-03071 Alacant, Spain.
E-mail: {mlf,carrasco}@dlsi.ua.es

In Wermter, S., Austin, J., Willshaw, D., eds. (2001) *Emergent Neural Computational Models Based on Neuroscience* (Heidelberg: Springer-Verlag), p. 482-486.

Abstract. Finite-state machines are the most pervasive models of computation, not only in theoretical computer science, but also in all of its applications to real-life problems, and constitute the best characterized computational model. On the other hand, neural networks —proposed almost sixty years ago by McCulloch and Pitts as a simplified model of nervous activity in living beings— have evolved into a great variety of so-called *artificial neural networks*. Artificial neural networks have become a very successful tool for modelling and problem solving because of their built-in learning capability, but most of the progress in this field has occurred with models that are very removed from the behaviour of real, i.e., biological neural networks. This paper surveys the work that has established a connection between finite-state machines and (mainly discrete-time recurrent) neural networks, and suggests possible ways to construct finite-state models in biologically plausible neural networks.

1 Introduction

Finite-state machines are the most pervasive models of computation, not only in theoretical computer science, but also in all of its applications to real-life problems (natural and formal language processing, pattern recognition, control, etc.), and constitute the best characterized computational model. On the other hand, neural networks, —proposed almost sixty years ago by McCulloch and Pitts [1] as a simplified model of nervous activity in living beings—, have evolved into a great variety of so-called *artificial neural networks*. Artificial neural networks have become a very successful tool for modelling and problem solving because of their built-in learning capability, but most of the progress in this field has occurred with models that are very removed from the behaviour of real, i.e., biological neural networks.

This paper surveys the work that has established a connection between finite-state machines and (mainly discrete-time recurrent) neural networks, and reviews possible ways to construct finite-state models in biologically plausible neural networks. The paper is organized as follows: section 2 describes the simultaneous inception of discrete-state discrete-time neural net models and finite-state machines; section 3 describes the relation between continuous-state discrete-time neural networks and finite-state machines; section 4 moves on to a more realistic model, namely, continuous-state continuous-time neural nets; spiking neurons as a biologically plausible continuous-time continuous-state model of finite-state computation is discussed in section 5. Finally, concluding remarks are presented in section 6.

2 The early days: discrete-time, discrete-state models

2.1 McCulloch-Pitts nets

The fields of neural networks and finite-state computation started indeed simultaneously: when McCulloch and Pitts [1] formulated mathematically the behaviour of ensembles of neurons (after a number of simplifying assumptions such as the discretization of time and signals), they defined what we currently know as a finite-state machine (FSM). McCulloch & Pitts' simplified neurons work on *binary* signals and in discrete time; they receive one or more input signals and produce an output signal as follows:

- input and output signals can be *high* or *low*;
- inputs can be *excitatory* or *inhibitory*;
- the neuron has an integer activation threshold A;
- the neuron's output is high at time $t+1$ if, at time t,
  - more than A excitatory inputs are high and
  - no inhibitory input is high

  and it is low otherwise.

McCulloch & Pitts' neuron may be easily shown to be equivalent to the current formulation of a *linear threshold unit*:

- input (u_i) and output (y) signals are 0 (*low*) or 1 (*high*);
- the neuron has real weights W_i, one for each input signal, and a real bias b;
- the neuron's output at time $t+1$, $y[t+1]$ is given by

$$y[t+1] = \theta\left(\sum_i W_i u_i[t] + b\right) \tag{1}$$

  where $\theta(x)$ is the step function[1] (time indices t, $t+1$ may be dropped in some applications).

[1] The step function is defined as follows: $\theta(x) = 1$ if $x \geq 0$, 0 otherwise.

A McCulloch-Pitts net is a finite set of interconnected McCulloch-Pitts neurons. Some neurons receive external inputs; they are called *inputs to the net*. Other neurons compute the *outputs of the net*. The remaining neurons are called *hidden neurons*. Connections between neurons may form cycles (*i.e.*, the net may be *recurrent*). A McCulloch-Pitts net may be seen as a *discrete-time sequence processor* which turns a sequence of binary input vectors $\mathbf{u}[t]$ into a sequence of binary output vectors $\mathbf{y}[t]$. The *state* of a McCulloch-Pitts at time t is the vector of the outputs at time t of its recurrent neurons (i.e., those involved in cycles, if any). Therefore, one may see a a McCulloch-Pitts recurrent net as a finite-state machine[2] $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_I)$, because

- The net may be found at any time t in a state from a finite set $Q = \{\text{high}, \text{low}\}^{n_X}$, where n_X is the number of recurrent units.
- The vector of inputs at time t takes values from a finite set $\Sigma = \{\text{high}, \text{low}\}^{n_U}$, where n_U is the number of inputs to tne net. The set Σ is finite and may be seen as an alphabet.
- The vector of outputs at time t takes values from a finite set $\Gamma = \{\text{high}, \text{low}\}^{n_Y}$, where n_Y is the number of output signals going out from the network. This set is finite and may also be seen as an alphabet.
- The state of the net at time $t+1$ is a function δ of inputs and states at time t which is defined by the architecture of the net.
- The output of the net at time $t+1$ is a function λ of inputs and states at time t which is defined by the architecture of the net.
- The initial state $q_I \in Q$ is formed by the outputs of neurons and by the inputs to the net at time $t = 0$.

2.2 Regular sets

Later, Kleene [3] formalized the sets of input sequences that led a McCulloch-Pitts network to a given state. He called them *regular events*; we currently know them as *regular sets* or *regular languages*. Nowadays, computer scientists relate regular sets to finite-state machines, not to neural nets [2, p. 28].

2.3 Constructing finite-state machines in McCulloch-Pitts nets

Minsky [4] showed that any FSM can be simulated by a discrete-time recurrent neural net (DTRNN) using McCulloch-Pitts units; the construction used a number of neurons proportional to the number of states in the automaton; more recently, Alon et al. [5], Indyk [6], and and Horne and Hush [7] have established better (sublinear) bounds on the number of discrete neurons necessary to simulate an finite state machine.

[2] Such as Mealy and Moore machines, see [2, p. 42].

3 Relaxing the discrete-state restriction: sigmoid discrete-state recurrent neural networks

Discrete neurons (taking values, for example, in $\{0, 1\}$) are a very rough model of neural activity, and, on the other hand, error functions for discrete networks are not continuous with respect to the values of weights, which is crucial for the application of learning algorithms such as those based in gradient descent. Researchers have therefore also shown interest in neural networks containing analog units with continuous, real-valued activation functions g such as the logistic sigmoid $g_L(x; \beta) = 1/(1 + \exp(-\beta x))$, with β a positive number called the *gain* of the sigmoid. A logistic neuron has the form

$$y[t+1] = g_L\left(\sum_i W_i u_i[t] + b; \beta\right); \tag{2}$$

this relaxes the discrete-signal restriction. The logistic function has the following limiting properties: $\lim_{x\to\infty} g_L(x; \beta) = 1$ and $\lim_{x\to-\infty} g_L(x; \beta) = 0$, and $\lim_{\beta\to\infty} g_L(x; \beta) = \theta(x)$, where θ is the step function. The last property has the consequence that a logistic neuron with infinite gain is equivalent to a linear threshold unit. In summary, these are the advantages of relaxing the discrete-state restriction:

- Differentiability allows for gradient-descent learning
- Graded response gives a better model of some biological processes
- May, in principle, emulate discrete-state behaviour (the same as digital computers are built from analog units as transistors and diodes).

The use of analog units turns the state of the network into a real-valued vector. In principle, neural networks having neurons with real-valued states should be able to perform not only finite-state computation but also more advanced computational tasks.

3.1 Neural state machines

A *neural state machine*, by analogy with a FSM $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_I)$, is $N = (X, U, Y, \mathbf{f}, \mathbf{h}, \mathbf{x}_0)$ with

- $X = [0, 1]^{n_X}$ (for n_X state neurons);
- $U = [0, 1]^{n_U}$ (for n_U input signals);
- $Y = [0, 1]^{n_Y}$ (for n_Y output neurons);
- $\mathbf{f} : X \times U \to X$ and $\mathbf{h} : X \times U \to Y$ are computed by feedforward neural nets;
- $\mathbf{x}_0 \in [0, 1]^{n_X}$ is the initial state.

(the use of the $[0, 1]$ interval is consistent with the use of the logistic sigmoid function, but other choices are possible). This construction is usually called a *discrete-time recurrent neural net* (DTRNN).

Similarly to FSM, we can divide neural-state machines in *neural Mealy machines* and *neural Moore machines* [2, p. 42]. In neural Mealy machines the output is a function of the previous state and the current input. Figure 1 shows a block diagram of a neural Mealy machine. Robinson and Fallside's [8] *recurrent*

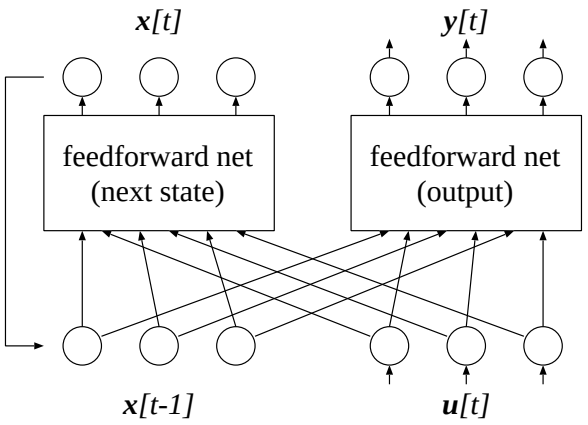


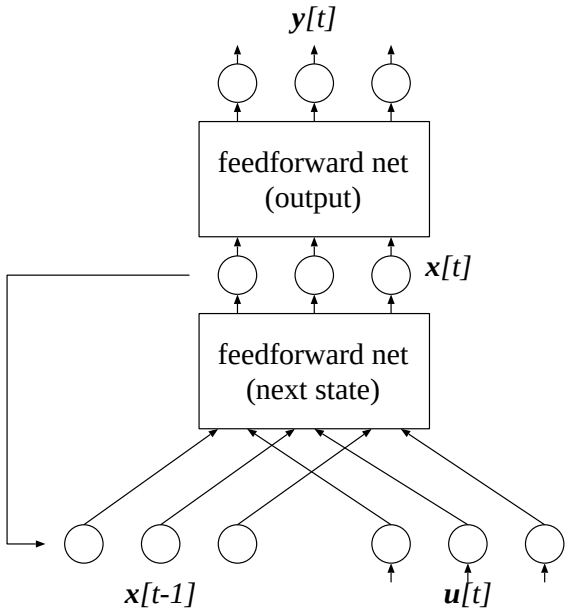
Fig. 1. A neural Mealy machine

error propagation nets, used for speech recognition, are an example of a neural Mealy machine. In a neural Moore machine, the output is simply a function of the state just computed ($\mathbf{h} : X \rightarrow Y$), see figure 2. Elman's *simple recurrent net* [9] is an example of a neural Moore machine.

3.2 Learning finite-state behaviour in sigmoid DTRNN

Under this intuitive assumption, a number of researchers set out to test whether sigmoid DTRNN could learn FSM behaviour from samples [10–20]. Sigmoid DTRNN may be trained to be FSM as follows:

- Define a *learning set* (input and output strings).
- Choose a suitable architecture (some DTRNN architectures are incapable of representing all FSM [21, 22])
- Decide on an encoding for inputs.
- Define output targets for each symbol and establish tolerances.
- Initialize weights, biases and initial states adequately.
- Use a suitable learning algorithm to vary weights, biases, and, optionally, initial states until the net outputs the correct output string (within tolerance) for each input string in the *learning set*.
- If a *test set* has been set aside, check the network's *generalization* behaviour.
- If a FSM is needed, *extract* (see section 3.3) one from the dynamics of the network (even if dynamics is not finite-state!).



-

Fig. 2. A neural Moore machine

However, a number of open questions arise when training DTRNN to behave as FSM, among which

- How does one choose n_X? This introduces a definite inductive bias: too small a value may yield an incapable DTRNN; too big a value may hamper its generalization ability.
- Will the DTRNN exhibit finite-state behaviour? A continuous-state DTRNN has an infinite number of states available and therefore has no bias toward discrete-state behaviour.
- Will it indeed learn? A DTRNN may be computationally capable to perform a task but *learning* that task from examples may not be easy or even possible.
- As with all neural networks, learning may get trapped in undesirable local minima of the error function.

The results obtained so far by the researchers cited show that, indeed, DTRNN can learn FSM-like behaviour from samples, althogh some problems persist.

One such problem is called *instability*: after learning, FSM-like behaviour is observed only for for short input sequences but degrades with sequence length; indeed, we will say that a DTRNN shows *stable* FSM behaviour when outputs are within tolerance of targets for input strings of *any* length. As will be explained later, DTRNN may be *constructed* to emulate finite-state machines; stable DTRNN constructed in that way have high weights, which in turn has

the consequence that the error function has very small gradients; this may be part of the explanation why stable behaviour is hard to learn.

Another related problem occurs when the task to be learned has long-term dependencies, that is, when late outputs depend on very early inputs; when this is the case, gradient-descent algorithms have trouble relating late contributions to the error to small changes in the state of neurons in early stages of the processing; these problems have been studied in detail by Bengio et al. [23].

3.3 Extracting finite-state machines from trained networks

Once successfully trained, specialized algorithms may *extract* FSM from the dynamics of the DTRNN; some use a straightforward equipartition of neural state space followed by a branch-and-bound algorithm [12], or a clustering algorithm [10, 16, 20]. Very often, the finite-state automaton extracted behaves correctly for strings of any length, even better than the original DTRNN. But automaton extraction algorithms have been criticised [24, 25] in the sense that FSM extraction may not reflect the actual computation performed by the DTRNN. More recently, Casey [26] has shown that DTRNN can indeed "organize their state space to mimic the states in the [...] state machine that can perform the computation" and be trained or programmed to behave as FSM. Also recently, Blair and Pollack [27] presented an increasing-precision dynamical analysis that identifies those DTRNNs that have actually learned to behave as FSM.

3.4 Programming sigmoid DTRNN to behave as finite-state machines

Finally, some researchers have set out to study whether it is possible to program a sigmoid-based DTRNN so that it behaves as a given FSM, that is, they have tried to formulate sets of rules for choosing the weights and initial states of the DTRNN based on the transition function and the output function of the corresponding FSM. In summary, to simulate a FSM in a sigmoid DTRNN one has to decide:

- How to encode the input symbols $\sigma_k \in \Sigma$ as vectors $\mathbf{u}_k \in U$. The usual choice is a (*one-hot*) encoding: $n_U = |\Sigma|$, $(\mathbf{u}_k)_i = \delta_{ik}$, where δ_{ik} is Kronecker's delta, defined as $\delta_{ik} = 1$ if $i = k$ and 0 otherwise.
- How to interpret outputs as symbols $\gamma_m \in \Gamma$. For example, nonempty disjoint regions $Y_m \subseteq Y$ (defined *e.g.* by a tolerance around suitable target values) may be assigned to each $\gamma_m \in \Gamma$.
- How many state neurons n_X to use (this will be discussed later).
- Which neural architecture to use.
- The values for weights.
- A value for the initial state $\mathbf{x}_0$ (for q_I).

As has been said in section 3.2, not all DTRNN architectures can emulate all FSM [21, 22].

When does a DTRNN behave as a FSM? A DTRNN $N = (X, U, Y, \mathbf{f}, \mathbf{h}, \mathbf{x}_0)$ is said to behave as a FSM $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_I)$ when (Casey 1996):

1. States $q_i \in Q$ are assigned nonempty disjoint regions $X_i \subseteq X$ such that the DTRNN N is said to be in state q_i at time t when $\mathbf{x}[t] \in X_i$.
2. The initial state of the DTRNN N, belongs to the region assigned to state q_I, that is, $\mathbf{x}_0 \in X_I$.
3. $\mathbf{f}_k(X_j) \subseteq X_i \quad \forall q_j \in Q, \sigma_k \in \Sigma : \delta(q_j, \sigma_k) = q_i$, where $\mathbf{f}_k(A) = \{\mathbf{f}(\mathbf{x}, \mathbf{u}_k) : \mathbf{x} \in A\}$ for short (see figure 3).

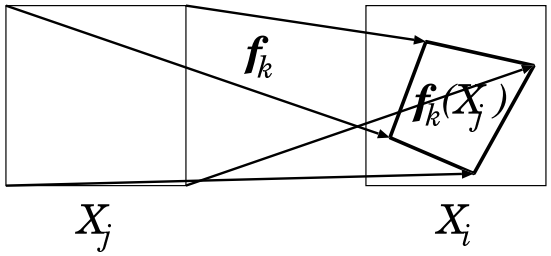


Fig. 3. Correctness of the next-state function of a FSM as computed by a DTRNN. The transition $\delta(q_j, \sigma_k) = q_i$ is illustrated.

4. $\mathbf{h}_k(X_j) \subseteq Y_m \quad \forall q_j \in Q, \sigma_k \in \Sigma : \lambda(q_j, \sigma_k) = \gamma_m$, where $\mathbf{h}_k(A) = \{\mathbf{h}(\mathbf{x}, \mathbf{u}_k) : \mathbf{x} \in A\}$ for short (see figure 4).

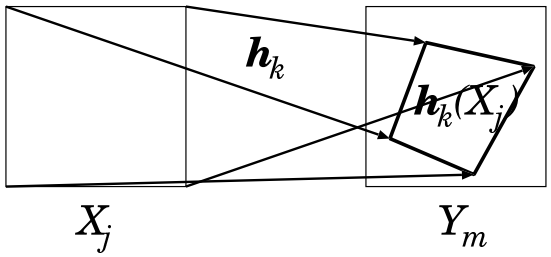


Fig. 4. Correctness of the output function of a FSM as computed by a DTRNN. The production of output $\lambda(q_j, \sigma_k) = \gamma_m$ is illustrated.

Omlin and Giles [28] have proposed an algorithm for encoding deterministic finite-state automata (DFA, a class of FSM) in second-order recurrent neural networks which is based on a study of the fixed points of the sigmoid function. Alquézar and Sanfeliu [29] have generalized Minsky's [4] result to show that DFA may be encoded in Elman [9] nets with rational (not real) sigmoid transfer functions. Kremer [30] has recently shown that a single-layer first-order sigmoid

DTRNN can represent the state transition function of any finite-state automaton. Frasconi et al. [31] have shown similar encodings for radial-basis-function DTRNN. All of these constructions use a number of hidden units proportional to the number of states in the FSM. More recently, Šíma [32] has shown that the behaviour of any discrete-state DTRNN may be stably emulated by a continuous-state DTRNN using activation functions in a very general class which includes sigmoid functions. In a more recent paper, Šíma and Wiedermann [33] show that any regular language may be more efficiently recognized by a DTRNN having threshold units. Combining both results, one concludes that sigmoid DTRNN can act as DFA accepting any regular language.

Recently, Carrasco et al. [22] (see also [34, 35]) have expanded the current results on stable encoding of FSM on DTRNN to a larger family of sigmoids, a larger variety of DTRNN (including first- and second-order architectures), and a wider class of FSM architectures (DFA and Mealy and Moore FSM), by establishing a simplified procedure to prove the stability of a devised encoding and to obtain weights as small as possible. Small weights are of interest if encoding is used to inject partial *a priori* knowledge into the DTRNN before training it through gradient descent.

One of Carrasco et al.'s [22] constructions is explained here in more detail: the encoding of a Mealy machine in a second-order DTRNN; this construction is similar to the one proposed earlier by Omlin and Giles [28]. It uses a one-hot encoding for inputs and a "one-hot" interpretation for outputs, and $n_X = |Q|$ state units (DTRNN is in state q_i at time t if $x_i[t]$ is high and the $x_j[t], j \neq i$, are low). Then the initial state is chosen so that $x_I[0] = 1$ and all other $x_i[0] = 0$. A single-layer *second-order* neural net [12] is used for both δ and λ as follows:

– *Next state*: for each $i = 1, 2, \ldots n_X$,

$$x_i[t] = g\left(\sum_{j=1}^{n_X}\sum_{k=1}^{n_U} W_{ijk}^{xxu} x_j[t-1]u_k[t]\right) \tag{3}$$

– *Output*: for each $i = 1, 2, \ldots n_Y$,

$$y_i[t] = g\left(\sum_{j=1}^{n_X}\sum_{k=1}^{n_U} W_{ijk}^{yxu} x_j[t-1]u_k[t]\right), \tag{4}$$

where $g(x) = g_L(x; 1)$. The next-state weights are chosen as follows: $W_{ijk}^{xxu} = H$ if $\delta(q_j, \sigma_k) = q_i$ and $-H$ otherwise ($\sigma_k \in \Sigma$, $q_i, q_j \in Q$). The output weights are $W_{ijk}^{yxu} = H$ if $\lambda(q_j, \sigma_k) = \gamma_i$ and $-H$ otherwise ($\gamma_i \in \Gamma$). It may also be said that weights are $+1$ and -1 and that the gain of the sigmoid is $\beta = H$ (see section 3). Now, the key is to choose H high enough to ensure correct output for strings of *any* length; Carrasco et al. [22] obtain the lowest possible values for H by mathematical induction after a worst-case study; the results are shown in table 1. Encoding results such as this are very important: they show that discrete-state behaviour may be obtained with continuous-state units having a finite sigmoid "gain" H (discrete-state behaviour is always guaranteed by an infinite gain, see section 3).

$\|Q\|$	H
2	2^+
3	3.113
6	4.181
10	4.863
30	6.224

Table 1. Values of the weight parameter H for stable finite-state behaviour of a DTRNN as a function of the number of states $|Q|$ of the FSM (the value 2^+ indicates any value infinitesimally larger than 2)

4 Relaxing the discrete-time restriction

All of the encodings discussed are for finite-state machines in *discrete-time* recurrent neural networks which assume the existence of a non-neural external clock which times their behaviour and a non-neural storage or memory for the previous state of the network, which is needed to compute the next state from the inputs. However, real (biological) neural networks are physical systems that operate in continuous time and should contain, if involved in the emulation of finite-state behaviour, neural mechanisms for synchronization and for memory.

Clocks and memories are indeed present in digital computers that, on the one hand, show discrete-time behaviour, but, on the other, are built from analog transistors and diodes having continuous-time dynamics. Therefore, it should in principle be possible to build a more natural model of finite-state computation based on *continuous-time* recurrent neural networks (CTRNN). CTRNN are a class of networks whose inputs and outputs are functions of a continuous-time variable and whose neurons have a temporal response that is described as a differential equation in time (for an excellent review on CTRNN, see [36]). We are not aware of any attempt to describe the finite-state computational behaviour of CTRNN.

The continuous-time version of the sigmoid unit may be written as follows [37, 38]:

$$\tau \frac{dy}{dt} = -y + g\left(\sum_i W_i u_i + b\right), \tag{5}$$

where τ is the unit's time constant (dynamics). The stationary (infinite-time) state is, obviously, that of a discrete-time or instantaneous neuron

$$y = g\left(\sum_i W_i u_i + b\right). \tag{6}$$

CTRNN may be constructed and trained [36] to process a *continuous input signal* into a *continuous output signal*:

$$\mathbf{u}(t) \to \mathbf{y}(t). \tag{7}$$

In particular, CTRNN may also be built to act as a memory (bistable device) or as a clock (oscillator). Therefore, CTRNN may emulate DTRNN (using CTRNN clocks and CTRNN memories), if inputs follow a precise chronogram (as they do in digital computers). As a corollary, CTRNN may be programmed to emulate FSM behaviour.

Nevertheless, a note regarding biological plausibility is in order. FSM behaviour is often a discrete-time simplification of continuous-time behaviour (e.g., speech, vision, etc.), which could be treated directly using CTRNN without constructing a discrete-time computational model. Indeed, while discrete-state discrete-time RNN were characterized in the 50's as finite-state machines, to our knowledge, the continuous-time continuous-state computational models represented by CTRNN have not been characterized theoretically.

5 Biologically inspired models: spiking or integrate-and-fire neurons

5.1 Spiking or integrate-and-fire neurons

DTRNN and CTRNN are usually formulated in terms of sigmoid neurons transmitting amplitude-modulated signals, but most real neurons may be seen as using some kind of temporal encoding:

- trains of impulses (spikes) are sent,
- they are received and integrated by each neuron (possibly with some leakage),
- the neuron fires (sends an impulse) when the integral reaches a certain threshold.

These are the rudiments of *integrate-and-fire* or *spiking neuron* models. The computational capabilities of recurrent neural networks containing these biologically-motivated neuron models have yet to be explored. This opens a wide field for future interaction between theoretical computer scientists and neuroscientists.

5.2 Equivalence to sigmoid units and implications

While integrate-and-fire or spiking models may be seen as very different from the sigmoid neurons discussed so far, they are not. The correspondence is, however, not straightforward. Biological neural systems perform visual processing (10 synaptic stages) in $\sim$ 100 ms [39]. But real neurons are very *slow*: firing frequencies are in the range of 100 Hz. Therefore, direct temporal encoding of analog variables such as the output of a continuous-state sigmoid unit are not biologically plausible. A variety of solutions have been proposed. One is called the *space-rate code*: analog variables in $[0, 1]$ are encoded as the fraction of neurons in a pool of neurons which fire within a given time interval. Fast analog space-rate computation (e.g. of sigmoids) and temporal processing (e.g. band-pass filtering) has been demonstrated through simulation [40]. This suggests that

finite-state machines may be indirectly encoded in integrate-and-fire networks by implementing a sigmoid DTRNN as a CTRNN and then converting the CTRNN into an integrate-and-fire network.

5.3 Direct encoding of finite-state machines in integrate-and-fire networks

A more direct, but related approach to finite-state behaviour in networks of spiking neurons has been recently proposed by Wennekers [41]. Wennekers' spiking neurons form *synfire chains*, that is, sequences of events in which a particular subset of integrate-and-fire neurons *syn*chronizedly *fire* in response to the synchronized firing of another (or the same) subset of neurons after an approximately constant delay determined by the characteristics of the connections and the neurons themselves. Each state in the finite-state machine is associated with a particular subset of neurons and the network is said to be in a certain state when the corresponding subset is synchronizedly firing. State transitions are gated by special subsets which fire only when a certain external input is present and a certain state was firing one delay unit before.

6 Concluding remarks

Our survey shows that

1. Very useful finite-state computation models stem from McCulloch & Pitts' [1] idealized discrete-signal discrete-time recurrent neural network models. These models are present in many successful theoretical and practical computational solutions.
2. Continuous-state discrete-time models are more convenient in learning settings and are shown to be able to stably emulate finite-state behaviour. These networks use non-neural (external) devices such as clocks and memories.
3. Continuous-time RNN may, in principle, emulate clocks, memories and, therefore, they should be capable of emulating FSM behaviour, although we are not aware of any work about this aspect.
4. Theoretical models of the computational capabilities of continuous-time continuous-state recurrent neural networks such as those available for discrete-time discrete-state RNN (i.e. finite-state machines) are still missing.
5. The study of more biologically plausible models of finite-state computation —e.g., integrate-and-fire neurons– has just started.

However, an important question remains unanswered: is finite-state computation still a relevant model of biological information processing or is it a computationally convenient simplification of this behaviour?

Acknowledgements: The authors thank Juan Antonio Pérez-Ortiz for comments on this manuscript and acknowledge the support of the Spanish Comisión Interministerial de Ciencia y Tecnologia through grant TIC97-0941.

References

1. W. S. McCulloch and W. H. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.
2. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison–Wesley, Reading, MA, 1979.
3. S.C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.
4. M.L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1967. Ch: Neural Networks. Automata Made up of Parts.
5. N. Alon, A. K. Dewdney, and T. J. Ott. Efficient simulation of finite automata by neural nets. *Journal of the Association of Computing Machinery*, 38(2):495–514, 1991.
6. P. Indyk. Optimal simulation of automata by neural nets. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 337–348, Berlin, 1995. Springer-Verlag.
7. B. G. Horne and D. R. Hush. Bounds on the complexity of recurrent neural network implementations of finite state machines. *Neural Networks*, 9(2):243–252, 1996.
8. Tony Robinson and Frank Fallside. A recurrent error propagation network speech recognition system. *Computer Speech and Language*, 5:259–274, 1991.
9. J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.
10. A. Cleeremans, D. Servan-Schreiber, and J. L. McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381, 1989.
11. Jordan B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:227–252, 1991.
12. C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee. Learning and extracted finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405, 1992.
13. R. L. Watrous and G. M. Kuhn. Induction of finite-state languages using second-order recurrent networks. *Neural Computation*, 4(3):406–414, 1992.
14. Arun Maskara and Andrew Noetzel. Forcing simple recurrent neural networks to encode context. In *Proceedings of the 1992 Long Island Conference on Artificial Intelligence and Computer Graphics*, 1992.
15. A. Sanfeliu and R. Alquézar. Active grammatical inference: a new learning methodology. In Dov Dori and A. Bruckstein, editors, *Shape and Structure in Pattern Recognition*, Singapore, 1994. World Scientific. Proceedings of the IAPR International Workshop on Structural and Syntactic Pattern Recognition SSPR'94 (Nahariya, Israel).
16. P. Manolios and R. Fanelli. First order recurrent neural networks and deterministic finite state automata. *Neural Computation*, 6(6):1154–1172, 1994.
17. M. L. Forcada and R. C. Carrasco. Learning the initial state of a second-order recurrent neural network during regular-language inference. *Neural Computation*, 7(5):923–930, 1995.
18. Peter Tiňo and Jozef Sajda. Learning and extracting initial Mealy automata with a modular neural network model. *Neural Computation*, 7(4), July 1995.
19. R. P. Ñeco and M. L. Forcada. Beyond Mealy machines: Learning translators with recurrent neural networks. In *Proceedings of the World Conference on Neural Networks '96*, pages 408–411, San Diego, California, September 15–18 1996.

20. Marco Gori, Marco Maggini, E. Martinelli, and G. Soda. Inductive inference from noisy examples using the hybrid finite state filter. *IEEE Transactions on Neural Networks*, 9(3):571–575, 1998.
21. M.W. Goudreau, C.L. Giles, S.T. Chakradhar, and D. Chen. First-order vs. second-order single layer recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(3):511–513, 1994.
22. Rafael C. Carrasco, Mikel L. Forcada, M. Ángeles Valdés-Muñoz, and Ramón P. Ñeco. Stable encoding of finite-state machines in discrete-time recurrent neural nets with sigmoid units. *Neural Computation*, 12, 2000. In press.
23. Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.
24. J.F. Kolen and Jordan B. Pollack. The observer's paradox: apparent computational complexity in physical systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 7:253–277, 1995.
25. J. F. Kolen. Fool's gold: Extracting finite state machines from recurrent network dynamics. In J. D. Cowan, G. Tesauro, , and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 501–508, San Mateo, CA, 1994. Morgan Kaufmann.
26. M. Casey. The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6):1135–1178, 1996.
27. A. Blair and J. B. Pollack. Analysis of dynamical recognizers. *Neural Computation*, 9(5):1127–1142, 1997.
28. C. W. Omlin and C. L. Giles. Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM*, 43(6):937–972, 1996.
29. R. Alquézar and A. Sanfeliu. An algebraic framework to represent finite state automata in single-layer recurrent neural networks. *Neural Computation*, 7(5):931–949, 1995.
30. Stefan C. Kremer. *A Theory of Grammatical Induction in the Connectionist Paradigm*. PhD thesis, Department of Computer Science, University of Alberta, Edmonton, Alberta, 1996.
31. Paolo Frasconi, Marco Gori, Marco Maggini, and Giovanni Soda. Representation of finite-state automata in recurrent radial basis function networks. *Machine Learning*, 23:5–32, 1996.
32. Jiří Šíma. Analog stable simulation of discrete neural networks. *Neural Network World*, 7:679–686, 1997.
33. Jiří Šíma and Jiří Wiedermann. Theory of neuromata. *Journal of the ACM*, 45(1):155–178, 1998.
34. Ramón P. Ñeco, Mikel L. Forcada, Rafael C. Carrasco, and M. Ángeles Valdés-Muñoz. Encoding of sequential translators in discrete-time recurrent neural nets. In *Proceedings of the European Symposium on Artificial Neural Networks ESANN'99*, pages 375–380, 1999.
35. Rafael C. Carrasco, Jose Oncina, and Mikel L. Forcada. Efficient encodings of finite automata in discrete-time recurrent neural networks. In *Proceedings of ICANN'99, International Conference on Artificial Neural Networks*, 1999. (in press).
36. B. A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: a survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, 1995.
37. F. J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, 1987.

38. L.B. Almeida. Backpropagation in perceptrons with feedback. In R. Eckmiller and Ch. von der Malsburg, editors, *Neural Computers*, pages 199–208, Neuss 1987, 1988. Springer-Verlag, Berlin.
39. S. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381:520–522, 1996.
40. Thomas Natschläger and Wolfgang Maass. Fast analog computation in networks of spiking neurons using unreliable synapses. In *Proceedings of ESANN'99, European Symposium on Artificial Neural Networks*, pages 417–422, 1999.
41. Thomas Wennekers. Synfire graphs: From spike patterns to automata of spiking neurons. Technical Report Ulmer Informatik-Berichte Nr. 98–08, Universität Ulm, Fakultät für Informatik, 1998.