Injecting Nondeterministic Finite State Automata into Recurrent Neural Networks

Paolo Frasconi   Marco Gori   Giovanni Soda

Dipartimento di Sistemi e Informatica
Via di Santa Marta 3 - 50139 Firenze - Italy
Tel. (+39) 55-4796265 - Fax (+39) 55-4796363
e-mail : {frasconi,marco}@ingfi1.cineca.it

Abstract

In this paper we propose a method for injecting time-warping nondeterministic finite state automata into recurrent neural networks. The proposed algorithm takes as input a set of automata transition rules and produces a recurrent architecture. The resulting connection weights are specified by means of linear constraints. In this way, the network is guaranteed to carry out the assigned automata rules, provided the weights belong to the constrained domain and the inputs belong to an appropriate range of values, making possible a boolean interpretation. In a subsequent phase, the weights can be adapted in order to obtain the desired behavior on corrupted inputs, using learning from examples. One of the main concerns of the proposed neural model is that it is no longer focussed exclusively on learning, but also on the identification of significant architectural and weight constraints derived systematically from automata rules, representing the partial domain knowledge on a given problem.

I Introduction

The ability of learning from examples is certainly the most appealing feature of neural networks. In the last few years, several researchers have used connectionist models for solving different kinds of problems ranging from robot control to pattern recognition. Coping with optimization of functions with several thousands of variables is quite common. Surprisingly, in many practical cases, global or near-global optimization is attained with rough numerical methods. For example, successful applications of neural nets for recognition of handwritten characters (e.g.: [18]) and for phoneme discrimination (e.g. [31]) have been proposed which do not report serious convergence problems.

Some attempts to understand the theoretical reasons for the successes and failures of supervised learning schemes, and particular of Backpropagation, [19], [27],[23],[34] have been carried out [16],[11] which explain when such schemes are likely to succeed. These results should not lead us to believe that tabula rasa learning schemes, based exclusively on learning from examples, are likely to succeed in any practical application. On the contrary, as stated by Minsky and Papert, "...significant learning at significant rate presupposes some significant prior structure. Simple learning schemes based on adjusting coefficients can indeed be practical and valuable when the partial functions are reasonably matched on the task,..." ([22], p. 16). Recently, Geman *et. al.* [12] have used the statistical viewpoint to highlight strengths and weaknesses of neural models. In that framework the use of *nonparametric statistical inference* lead them to formulate the *bias/variance dilemma*. They conclude that "Important properties must be built-in or hard-wired, perhaps to be *tuned* later by experience, but not learned in any statistical meaningful way".

The ideas reported in this paper can be traced back to our belief in "biased models" for solving most meaningful practical problems. When assuming this viewpoint, neural modeling is no longer focussed exclusively on learning, but also on the identification of significant architectural and weight constraints. Following the guidelines of many researchers (see next section), basically, we suggest to represent the partial domain knowledge on a given problem in terms of weight constraints, and to complete it by learning from examples.

In order to conceive models of this kind, one has to specify the kind of prior knowledge to be used. A quite common situation is that this knowledge is more or less affected by uncertainty and, therefore, it should be embedded carefully into a connectionist model in order not to force artificial behaviors. The uncertainty can be dealt with naturally by the network, relying on its learning capabilities. Instead of developing representations from tabula rasa, however, the role of learning becomes that of refining prior knowledge, i.e. the *learning space*[1] turns out to be severely constrained by the prior knowledge in a restricted domain. In this way, the learning algorithm becomes tightly-related to rule-embedding.

In the choice of the more appropriate model for a given problem, one can decide the degree of prior knowledge biasing. The kind of knowledge to be embedded into a network was suggested by the kind of applications we are mainly interested in. Basically, we want our connectionist models to be able to deal with the temporal structure typical of Automatic Speech Recognition and Understanding problems.

The need of approaching similar problems requires the knowledge model to be flexible enough for dealing with the various levels of the recognition and understanding processes. For example, while the prior knowledge may play a considerable role at phonetic level, there is no doubt that the absence of any form of model biasing has no chances to succeed in performing large lexicon word recognition. When moving from lower to higher level tasks the kind of prior knowledge changes significantly.

Quite a common aspect of all the speech processing levels is that we must deal with temporal phenomena, which appear progressively symbolic when moving toward the higher levels of processing. Although these sequential phenomena all exhibit some interesting regularity, their

[1] We call *learning space* the set of network weights under which learning takes place.

inherent stochastic nature makes difficult any form of deterministic modeling. For example a very serious problem is that the duration of the transients from one phoneme to the following one is largely variable, depending on the phonemes involved. As a result, the lexical relationship between two subsequent phonemes in a word may be very clear, but it may be obscure at "frame level" because of the uncertainty due to the transient duration. Moreover, we must always deal with the uncertainty due to the possible error at any level. At the frame level, a phonetic coder which responds with the string "...aaaauaaaa..." is very likely performing the recognition of the vowel "a" and the "u" is likely to be a "noisy symbol". These analyses, which are more developed in [10], suggest the representation of the partial domain knowledge by *Nondeterministic Finite State Automata (NFA)*. The availability of *partial* domain knowledge is translated into the *partial* definition of the state transitions of an NFA. Our preliminary experiments in Isolated Word Recognition for Italian, however, indicate that the most important part which needs to be modeled by NFA is that involving the already discussed uncertainty arising from the undefined duration of phonemes at frame level (*time-warping phenomena*). As for the lexical rules composing the words, we may place plenty of deterministic rules [2] for describing even very rare transitions, and learning from examples those which are better fitted with the given data. This requires the use of the kind of architectures referred to as *K-L* [10].

Since we are mainly interested in modeling the time-warping phenomena, we introduce the *time-warping nondeterministic automata* concept [3]. Basically, these are nondeterministic automata in which the only form of "uncertainty" (undefined state transitions) is limited to the number of steps needed for performing "actual state transitions".

Recurrent neural networks capable of exhibiting this form of nondeterministic behavior were first introduced in [9],[10] but, as pointed out by Giles *et. al* [14], the proposed algorithm for setting up the weight constraints assumed quite a limited hypothesis on the kind of architectures. They point out that there exist simple finite state automata which can not be represented with first-order, fully-recurrent architectures without additional layers of weights. In this paper, we remove this limitation and extend that algorithm, thus providing a tool for effectively *compiling* any time-warping regular automata into a recurrent neural network (RNN). This extension was motivated theoretically, but also by the need of experimenting these ideas massively in the field of Automatic Speech Recognition. The process of compiling the assigned "program" lead us to create a recurrent architecture and a set of weight constraints. As suggested in [10], this network can be subsequently integrated into a more complex neural architecture. Learning from examples can be used to globally optimize the resulting system which contains both free and constrained connection weights.

Basically, the algorithm proposed in this paper, allocates a set of self-recurrent neurons to encode the automata states. Before this step, a slight modification to the original automaton is needed, in order to guarantee that each pair of consecutive states has unitary Hamming distance

[2] Unlikely the time-warping phenomena, these rules can be handled quite well in practice [10].

[3] A formal definition of these automata is given in section III.

in the code string (see also [10] for a discussion of this coding requirement). Thereafter, a set of neurons is allocated to compute the state transition function in a boolean-like fashion. An initial weight vector and a set of linear constraints on the weight values is computed by the algorithm.

The resulting network requires that the inputs belong to an appropriate range of values, allowing to give them a boolean interpretation. The actual size of the correct input space can be programmed in the construction algorithm. For each weight vector which satisfies the set of linear constraints, and for each input sequence which lies in the correct input space, the recurrent net can be shown to carry out one of the computations of the assigned nondeterministic automaton.

After the proposed network generation method has been applied, the weights can be moved in the allowed space in order to select the particular computation which is more appropriate to the available training examples. This can be accomplished by using constrained optimization techniques (see e.g. [20]).

The paper is organized as follows. In the next section we give some reference to related work. In section III, we define the time-warping automata concept formally, while in section IV, the Recurrent Neural Network used is described together with some useful properties. In section V, we discuss input and state coding. In section VI, we propose an algorithm for transforming the assigned automaton into a more adequate form for realization, which is described in section VII. In section VIII, the theory proposed is illustrated with an example and, finally, some conclusions are drawn in section IX.

II Related work

Neural networks have been suggested for implementing automata a long time ago. The research in this field can be traced back to early works by McCulloch and Pitts [21]. The recent renewal of interest in Neural Networks has led several researchers to carry out detailed analyses on this and related subjects (e.g. [15],[7]). The learning capability of many recently proposed models has been particularly stressed. The availability of new connectionist procedures and of more powerful computers have allowed many researchers to investigate the possibility of learning interesting sequential behaviors from examples. For example, very interesting results have been obtained in language processing (e.g. [6],[5]) using Elman's net. Successful experiments of inferring unknown grammars from examples have been also carried out with recurrent networks which are very well suited for this kind of problem. In [35], Williams and Zipser have reported some experiments where a first-order recurrent network succeed in simple sequence recognition tasks. More recently, Giles *et al.* [13], and Watrous *et al.* [32] have proposed the learning of Finite State Automata with second-order recurrent nets. Sun *et al.* [28] have explored the use of external memory stacks for learning context-free grammars from examples. These investigations are very interesting, since they show that connectionist models can learn rules only relying on the presentation of examples. However, the use of "tabula rasa" neural networks is likely to fail for problems with significant structure. In order to deal with these problems many proposals have been suggested, in the attempt to make the neural networks somewhat biased by the domain knowledge.

For example, le Cun [18] has proposed the use of receptive fields in feedforward nets for applications to handwritten character recognition. The implementation of that concept results in special equality constraints on the weights. Perantonis *et al.* [26] have introduced a method for reducing the number of weights of a third-order network used for pattern recognition by imposing invariance under translation, rotation, and scaling. For incorporating the temporal properties of speech signals, Waibel [31] *et. al.* have proposed the Time Delay Neural Networks (TDNN), where proper weight constraints make it possible to deal effectively with the temporal nature of speech. Bengio *et al.* [3] have used special recurrent architectures, with only self-loop recurrent connections, in order to introduce a natural "forgetting behavior" into the network. This turns out to be useful, since only recent information is likely to affect phoneme recognition.

Although for different problems and for different network architectures, these papers report attempts to incorporate domain knowledge in terms of weight constraints, which, in turn, lead to reduce the number of free parameters to learn. The use of prior knowledge on a given problem may be exploited at several different levels. This knowledge may be more or less precise, but is certainly a piece of information for the construction of the mapping a neural network is commonly used for. It would be inefficient to take blind examples without exploiting what we already know. Learning from partial information is referred to as "learning from hints". Although the availability of that information does not overcome the theoretical limitations arising from computational complexity arguments [1], there is no doubt about the actual role of learning from hints in practice. Interesting "hints" for solving special problems have been proposed by several researchers (e.g.: [2],[14],[24]). In particular, in [14],[24], the learning of Finite State Automata is proposed using hints which are placed in a very elegant way using second-order recurrent networks. Towell *et al.* [30], [29] conceive the integration of rule-based knowledge with neural networks as a three-step process: insertion of the knowledge into the network, refinement by learning from examples, and extraction of the knowledge after learning refinement. In many cases, this final process is likely to show that the learning refinement may destroy the prior knowledge injected into the network. In the approach proposed in this paper, the learning from examples provides a congruent completion of the injected prior knowledge, and not its - though partial - destroying.

III Finite state automata

The aim of this section is to give a detailed formal definition of the class of automata that concern this paper. Let us begin by recalling some fundamental notions from classical automata theory [17]. A deterministic finite automaton (DFA) $\mathcal{A}$ is a quintuple:

$$\mathcal{A} \doteq \{Q, \Sigma, \Phi, R, F\} \tag{1}$$

where Q is a finite set of states, Σ is the input finite alphabet, $R \in Q$ is the initial state, $F \subset Q$ is the subset of accepting states, and Φ is a *total* function [4] known as state transition function (STF). The cardinalities of Q and Σ are denoted $|Q| = n$ and $|\Sigma| = m$. Given the input symbol $\sigma(t) \in \Sigma$ and the present state $q(t) \in Q$, the next state $q(t+1)$ can be computed by using the STF:

$$\Phi : Q \times \Sigma \longmapsto Q, \qquad q(t+1) = \Phi(q(t), \sigma(t)). \tag{2}$$

The sequence of input symbols $w = \sigma(1), \ldots, \sigma(T)$ is called input string.

The computation of the automaton starts in the state $q(1) = R$ and ends for $t = T$. The state $q(T)$ is called final state. The computation of an automaton determines the acceptability of the input string. The string is accepted if the final state $q(T) \in F$; otherwise it is rejected. The set of all the strings obtained as sequences of symbols in Σ is known as *closure* of Σ, denoted Σ^+. The union of Σ and the empty string is denoted Σ^*. The automaton defines a partition of Σ^+ into a set of recognized strings and a set of rejected strings. The set of strings recognized by $\mathcal{A}$ is known as *language* of $\mathcal{A}$, denoted $L(\mathcal{A})$. Two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be *equivalent* if they recognize the same language, i.e. $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

A more general class of automata can be defined to include nondeterministic computation. In a nondeterministic finite automaton (NFA) the state transition function maps a combination of state and input to a set of states. Formally:

$$\Phi_n : Q \times \Sigma \longmapsto \mathcal{P}(Q) \tag{3}$$

being $\mathcal{P}(Q)$ the set of all subsets of Q. Throughout the paper we will use the subscript n when referring to nondeterministic automata, and the subscript d for deterministic ones. For an NFA, the same input string may produce different computations and, therefore, different state sequences. A string $w \in \Sigma^+$ is recognized by an NFA if there exists at least one computation which ends in an accepting state.

In this paper we consider NFAs. More precisely, we assume the following kind of nondeterministic computation:

Definition 1 (Time-warping NFAs) *Let $\mathcal{A}_n$ be an NFA. We call $\mathcal{A}_n$ a* time-warping nondeterministic finite automaton *if, for any symbol $\sigma \in \Sigma$*

$$\begin{aligned} &q \in \Phi_n(q, \sigma) \quad \textit{and} \\ &|\Phi_n(q, \sigma)| \leq 2 \quad \forall q \in Q, \end{aligned} \tag{4}$$

where $|\cdot|$ denotes cardinality.

[4] A function is said to be total if it is defined for each element of its domain.

■

The language recognized by these automata comprises strings in which the number of consecutive repetitions of any symbol is not relevant. Formally, consider two strings $v, w \in \Sigma^*$ and a symbol $a \in \Sigma$. If $vaw \in L(\mathcal{A}_n)$, then $vaaw \in L(\mathcal{A}_n)$. Automata of this kind are useful to model time warping phenomena, which are relevant in temporal pattern recognition problems such as speech recognition [9],[10].

For this class of NFA we can use an equivalent representation by means of a DFA:

Definition 2 (Deterministic reduction and nondeterministic extension) *Let*

$$\mathcal{A}_n = (Q, \Sigma, \Phi_n, S, F)$$

be an NFA. Then the deterministic finite automaton $\mathcal{A}_d = (Q, \Sigma, \Phi_d, S, F)$ *having*

$$\Phi_d(q_i, \sigma_l) = \begin{cases} q_k & if \quad \Phi_n(q_i, \sigma_l) = \{q_i, q_k\} \\ q_i & if \quad \Phi_n(q_i, \sigma_l) = \{q_i\} \end{cases} \qquad \forall \sigma_l \in \Sigma \tag{5}$$

is called deterministic reduction *of* $\mathcal{A}_n$*.*

Let $\mathcal{A}_d = (Q, \Sigma, \Phi_d, S, F)$ *be a deterministic finite automaton. The* time-warping nondeterministic extension *of* $\mathcal{A}_d$ *is the NFA* $\mathcal{A}_n = (Q, \Sigma, \Phi_n, S, F)$ *having*

$$\Phi_n(q_i, \sigma_l) = \Phi_d(q_i, \sigma_l) \cup \{q_i\} \qquad \forall \sigma_l \in \Sigma. \tag{6}$$

■

Notice that assigned an arbitrary DFA $\mathcal{A}_d$, the suggested nondeterministic extension $\mathcal{A}_n$ always exists. The main goal of this paper is to show how any time-warping NFA can be implemented by means of a recurrent neural network.

IV Neural network model: some useful switching properties

A Formalism for the RNN

The model considered in this paper is a first order Multi-Delay RNN. This neural network architecture can be mapped onto an oriented graph $\mathcal{G} \doteq \{\mathcal{U} \cup \mathcal{N}, \mathcal{W}\}$, where $\mathcal{U}$ is the set of input nodes, $\mathcal{N}$ is a set of sigmoidal units and $\mathcal{W} \in (\mathcal{N} \times \mathcal{U}) \cup \mathcal{N}$ is a set of arcs. The generic arc $v_{ij}^{(d)} \in \mathcal{W}$ connects node $j \in \mathcal{U} \cup \mathcal{N}$ to node $i \in \mathcal{N}$ with delay d. The corresponding connection weight is denoted $w_{ij}^{(d)}$. The delay d can be any non-negative integer. In particular it can be zero; in that case the connection is said *static*. Denote with $\mathcal{G}^{(d)}$ the (possibly not connected) graph obtained removing from $\mathcal{G}$ all the arcs $v_{ij}^{(r)}$ with delay $r \neq d$. The only topological constraint we assume is the following:

Hypothesis 1 *The sub-graph* $\mathcal{G}^{(0)}$ *has no cycles.*

The computation performed by the Multi-Delay RNN is based on the following equations:

$$\begin{aligned} a_i(t) &= \sum_{d=0}^{D} \sum_{j \in \mathcal{S}_i^{(d)}(\mathcal{N})} w_{ij}^{(d)} x_j(t-d) + \sum_{d=0}^{D} \sum_{j \in \mathcal{S}_i^{(d)}(\mathcal{U})} w_{ij}^{(d)} u_j(t-d) \\ x_i(t) &= f(a_i(t)) = \tanh\left(\frac{a_i(t)}{2}\right) \end{aligned} \tag{7}$$

where $a_i(t)$ is the activation of i^{th} neuron at time t, $x_i(t)$ is its squashed output, and $u_j(t)$ is the j^{th} external input. $\mathcal{S}_i^{(d)}(\mathcal{N})$ is the set of network neurons connected to i with delay d, and $\mathcal{S}_i^{(d)}(\mathcal{U})$ is the set of external inputs connected to i with delay d. In this paper we assume only instantaneous connections and one time step delayed connections, i.e. $D = 1$ in eq. (7).

Remark 1 The above neural network model uses a sigmoidal squash function. Hence, the outputs of neurons are real values. This choice has origin in the precise intent of using learning from examples on the network initialized with automata rules. Supervised learning in this kind of networks can be accomplished by defining a standard quadratic error function and using Backpropagation through both space and time (see e.g. [25], [33], [35]). On the other hand, given the symbolic nature of the automaton, it is sometimes useful to be able to interpret the continuous outputs of the neurons as boolean values. For this purpose, consider the following definition:

Definition 3 (Boolean quantization function) *Let*

$$\rho^-, \rho^+ \in [0, 1)$$

be two threshold values. The boolean-valued function Γ *defined as* [5]*:*

$$\begin{aligned} &\Gamma : [-1, 1] \longmapsto \{-1, +1\} : \\ &\Gamma(x; \rho^-, \rho^+) = \begin{cases} -1 & if \;\; x < -\rho^- \\ +1 & if \;\; x > \rho^+ \end{cases} \end{aligned} \tag{8}$$

is called a boolean quantization function of x with thresholds ρ^-, ρ^+.

■

A boolean quantization function is not defined in $[-\rho^-, \rho^+]$. For values belonging to that interval no boolean interpretation is given.

B Latching properties of multi-delay recurrent nets

Let us focus on dynamic neurons, i.e. neurons having a self-loop connection, weighted by $w_{ii}^{(1)} \neq 0$. For these neurons, it is convenient to distinguish two contributions to the activation $a_i(t)$: the *self-loop contribution* and the *generalized input* $\mathcal{I}_i(t)$:

$$a_i(t) = w_{ii}^{(1)} x_i(t-1) + \mathcal{I}_i(t) \tag{9}$$

[5] In this paper we use the notation -1 and $+1$ for the two boolean constants.

$$\mathcal{I}_i(t) \doteq \sum_{d=0}^{1} \sum_{j \in \mathcal{S}_i^{(d)}(\mathcal{N})} (1-\delta_{ij}) w_{ij}^{(d)} x_j(t-d) + \sum_{d=0}^{1} \sum_{j \in \mathcal{S}_i^{(d)}(\mathcal{U})} w_{ij}^{(d)} u_j(t-d) \tag{10}$$

being $\delta_{ij} = 1$ iff $i = j$.

Dynamic neurons have interesting properties concerning the possibility of latching the information represented by their activation. Information latching was introduced in [8] for Local Feedback Multi-Layered Networks. This "long term" memory capability of recurrent networks is essential in order to obtain finite state machine behaviors. In [10] information latching was investigated for implementing a simple class of automata, bearing a "chain-like structure". In that paper the definition of information latching was based on the sign of the activation. We extend here that concept, using boolean quantization instead of the sign function.

Definition 4 (ρ-quantized information latching) *We say that a generic dynamic neuron i latches the information at t_0 if, given $\rho \in [0,1)$, the following relationship holds:*

$$\Gamma\left(x_i(t); \rho, \rho\right) = \Gamma\left(x_i(t_0); \rho, \rho\right) \qquad \forall t \geq t_0, \tag{11}$$

being Γ a boolean quantization function.

■

We say that a *state switching* occurs for the dynamic neuron i when the ρ-quantized information $\Gamma\left(x_i(t); \rho, \rho\right)$, latched by the neuron i, changes in a finite number of steps. In the following theorem we indicate under which conditions ρ-quantized information latching and state switching occur:

Theorem 1 *Let i be a generic neuron having $w_{ii}^{(1)} \neq 0$. Depending on $\mathcal{I}_i(t)$, it follows that:*

1. *if $\mathcal{I}_i(t) = 0$ then ρ-quantized information latching occurs provided that*

$$w_{ii}^{(1)} > w_\rho \doteq \frac{1}{\rho} \ln \left(\frac{1+\rho}{1-\rho} \right); \tag{12}$$

2. *if $w_{ii}^{(1)} > w_\rho^*$ then the latching condition also holds if $|\mathcal{I}_i(t)| < \mathcal{I}_i^* \;\; \forall t, t_0 : t > t_0$, being*

$$w_\rho^* = \frac{2}{1-\rho^2}, \tag{13}$$

$$\mathcal{I}_i^* = \sqrt{w_{ii}^{(1)}(w_{ii}^{(1)} - 2)} + \ln \left(w_{ii}^{(1)} - 1 - \sqrt{w_{ii}^{(1)}(w_{ii}^{(1)} - 2)} \right); \tag{14}$$

3. *if $w_{ii}^{(1)} > w_\rho^*$ then a state switching occurs in a finite number of steps if*

$$\begin{array}{ll} \mathcal{I}_i(t) > \mathcal{I}_i^* & \textit{low to high switching} \\ \mathcal{I}_i(t) < -\mathcal{I}_i^* & \textit{high to low switching.} \end{array} \tag{15}$$

Proof

See Appendix. ■

Notice that for $\rho \to 0$ we have $w_\rho \to 2$ and $w^*_\rho \to 2$ (cf. [10]). As for the case of activation sign information latching, the actual value of $w^{(1)}_{ii}$ affects the duration of the switching transient L_i: the greater is $w^{(1)}_{ii}$, the longer is the transient. A first slight difference in the case of ρ-quantized information latching is that the transient lasts until $x_i(t)$ reaches the boolean threshold value ρ. A more remarkable difference is that, during the transient, the boolean quantization function may be undefined. This can represent an effective problem if the information stored into memory neurons encodes the internal states of an automaton. As it is show in section VII, the method we propose circumvents this inconvenience. Notice that the variable duration of the switching transients L_i is the basic element which gives our recurrent networks a time-warping NFA behavior.

C Neural logic operators

Besides memory elements, the implementation of automata requires a feedforward network which carries out the computation of the state transition function. In this subsection we focus on the basic components of this portion of the global architecture. In particular, we describe a method for computing boolean functions with sigmoidal units. Again, our main concern is to provide a realization featuring continuous values which can be given a boolean interpretation. To this purpose we introduce the concept of boolean-like functions, taking advantage of the previously defined boolean quantization function.

Definition 5 (Boolean-like AND functions) *Let ρ, ρ_0^+, ρ_0^- $\in [0,1)$ be three quantization thresholds. A real function*

$$\mathcal{B}_\wedge : \{[-1,0) \cup (\rho,1]\}^n \longmapsto [-1,-\rho_0^-) \cup (\rho_0^+,1] :$$

$$y = \mathcal{B}_\wedge(x_1, x_2, \ldots, x_n; \rho, \rho_0^-, \rho_0^+)$$

is called a boolean-like AND function if

$$\Gamma(y;\rho_0^-,\rho_0^+) = \bigwedge_{i=1}^{n} \Gamma(x_i;0,\rho). \tag{16}$$

■

A similar definition can be given for OR functions:

Definition 6 (Boolean-like OR functions) *Let ρ, ρ_0^+, ρ_0^- $\in [0,1)$ be three quantization thresholds. A real function*

$$\mathcal{B}_\vee : \{[-1,-\rho) \cup (0,1]\}^n \longmapsto [-1,-\rho_0^-) \cup (\rho_0^+,1] :$$

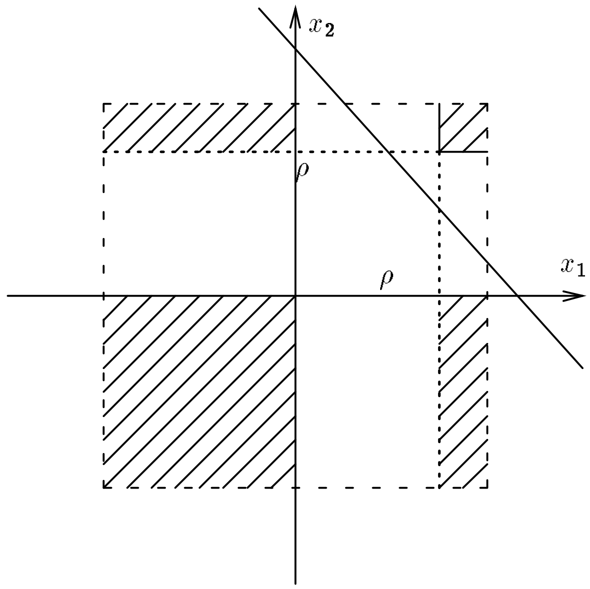


Figure 1: Example of domain (dashed region) for a boolean-like AND function of two variables. The line corresponds to a weight vector which satisfies inequalities (20).

$$y = \mathcal{B}_{\vee}(x_1, x_2, \ldots, x_n; \rho, \rho_0^-, \rho_0^+)$$

is called a boolean-like OR function if

$$\Gamma(y; \rho_0^-, \rho_0^+) = \bigvee_{i=1}^{n} \Gamma(x_i; -\rho, 0). \tag{17}$$

■

In the above definitions, ρ influences the boolean interpretation of the input variables; therefore it is called input quantization threshold. Similarly, ρ_0^- and ρ_0^+ are called output quantization thresholds. In the proposed framework we also assume that the NOT operator is implemented by sign inversion.

These boolean-like functions can be computed by a single sigmoidal neuron with n inputs. Such neurons will be referred to as *neural logic gates.* Let us begin considering the case of AND neural gates. Denote with $w_0, w_1, \ldots, w_n$ the weights of this neuron, being w_0 the bias. The activation is computed as:

$$a = w_0 + \sum_{i=1}^{n} w_i x_i \tag{18}$$

and the neuron output is computed as

$$y = f(a) = \tanh\left(\frac{a}{2}\right). \tag{19}$$

Theorem 2 *Assume a vector of weights $w_i > 0, i = 1 \dots n$. A single neuron which computes its output according to eqs.(18-19) implements a boolean-like AND function with input threshold ρ and output thresholds ρ^+, ρ^- if the weights satisfy the following $n+1$ linear constraints:*

$$\begin{aligned} & w_0 + \sum_{i=1}^{n} w_i \rho \geq f^{-1}(\rho_0^+) \\ & w_0 + \sum_{i=1, i \neq j}^{n} w_i \leq -f^{-1}(\rho_0^-) \qquad \forall j = 1 \dots n. \end{aligned} \tag{20}$$

In particular, the weights vector defined as follows

$$\begin{aligned} & w_1 = w_2 = \dots = w_n = \frac{f^{-1}(\rho_0^+) + f^{-1}(\rho_0^-)}{n\rho - n + 1} \\ & w_0 = -\frac{n\rho f^{-1}(\rho_0^-) + (n-1) f^{-1}(\rho_0^+)}{n\rho - n + 1} \end{aligned} \tag{21}$$

satisfies inequalities (20).

Proof
See Appendix. ■

Remark 2 It can be seen that if the linear constraints (20) are satisfied with $w_i > 0$, for $i = 1 \dots n$ (as required by Theorem 2), then the input threshold ρ must belong to the following interval:

$$\frac{n-1}{n} < \rho < 1. \tag{22}$$

This condition arises from the domain we have chosen defining our boolean-like functions. Actually, it imposes a limitation on the fan-in of the gate when the input threshold ρ is fixed. This restriction could be overcome by using multiple gates, adding a layer of neurons.

In our case it is also important to consider the realization of boolean-like AND functions having some of their inputs negated. Such functions can be easily implemented by changing the sign of the weights associated to the negated input. As a special case, a boolean-like OR function can be implemented from an AND one, by using the De Morgan law. As a first step all the inputs must be negated, i.e. the sign of all the weights $w_1, \dots w_n$ has to be changed. Then, the output must be negated. To this purpose the following relationship can be used:

$$\Gamma(y; \rho_0^-, \rho_0^+) = \overline{\Gamma(-y; \rho_0^+, \rho_0^-)}. \tag{23}$$

The sign inversion needed for implementing the NOT operator can be moved to each of the weights $w_0, \dots w_n$. Therefore, since there are two sign inversions, weights $w_1, \dots w_n$ in (21) do not change. Instead, the bias w_0 becomes

$$w_0 = \frac{n\rho f^{-1}(\rho_0^+) + (n-1) f^{-1}(\rho_0^-)}{n\rho - n + 1}. \tag{24}$$

The constraints which must be satisfied for a boolean-like OR function are:

$$
\begin{aligned}
& w_0 - \sum_{i=1}^{n} w_i \rho \leq -f^{-1}(\rho_0^-) \\
& w_0 - \sum_{i=1, i \neq j}^{n} w_i \geq f^{-1}(\rho_0^+) \quad \forall j = 1 \ldots n.
\end{aligned}
\tag{25}
$$

V Input and state representations

The approach we are pursuing for the realization of automata is quite different from their classical implementation by means of software or hardware logic devices. The automaton operates in a *symbolic domain*, defined by a finite alphabet and a finite set of states. On the contrary, the RNN model we use receives continuous inputs and computes continuous values by means of its neurons. So, an effective method is required to provide a *representation* of state and input symbols using vectors of real elements. We assume the network input is given by a vector $\mathbf{u} \in [-1, 1]^m$. We also assume n neurons of the networks are used to encode the automata states. Their outputs form a vector $\mathbf{x} \in [-1, +1]^n$. In order to give these vectors a symbolic interpretation, two quantization functions can be used, which map $\mathbf{u}$ and $\mathbf{x}$ onto a symbol $\sigma \in \Sigma$ and a state $q \in Q$, respectively.

These quantization functions are defined on the basis of a binary coding of the states and the input symbols, respectively. Given a finite set of symbols A, a binary coding function $\mathcal{C}$ is a mapping which transforms each element of A onto a string of boolean values:

$$
\mathcal{C} : A \longmapsto \{-1, +1\}^k \tag{26}
$$

where k is the number of bits used for coding, usually function of the cardinality of A. In our case we use n bits to encode the n states and m bits to encode the m input symbols. In this way a direct correspondence is assumed between a neuron and a bit in the coding string. In the approach we propose in this paper, the exclusive coding is chosen for both inputs and states. The coding vector for each input symbol $\sigma_l \in \Sigma$, denoted $\mathbf{u}_l^b \doteq \mathcal{C}_{\Sigma}(\sigma_l)$ is defined as:

$$
u_{lj}^b = \begin{cases} +1 & \text{if } \; j = l \\ -1 & \text{if } \; j \neq l \end{cases} \qquad l, j = 1 \ldots m \tag{27}
$$

Similarly, the boolean coding vector $\mathbf{x}_i^b \doteq \mathcal{C}_Q(q_i)$ for a state $q_i \in Q$ is defined as:

$$
x_{ij}^b = \begin{cases} +1 & \text{if } \; j = i \\ -1 & \text{if } \; j \neq i \end{cases} \qquad i, j = 1 \ldots m. \tag{28}
$$

Now we can use the boolean codes and Definition 3 to formally define the quantization functions. Given the real vector $\mathbf{u} = [u_1, \cdots, u_m]'$, the input quantization function is defined as:

$$
\begin{aligned}
& \mathcal{Q}_{\Sigma} : [-1, 1]^m \longmapsto \Sigma \cup \{\lambda\} \\
& \mathcal{Q}_{\Sigma}(\mathbf{u}) = \begin{cases} \sigma_l & \text{if } \;\; \Gamma(u_j; \rho_1, \rho_1) = u_{lj}^b \quad \forall j = 1 \ldots m \\ \lambda & \text{otherwise} \end{cases}
\end{aligned}
\tag{29}
$$

where $\rho_1 \in [0,1)$ is an appropriate threshold value. Similarly, the state quantization function is defined as:

$$\mathcal{Q}_Q : [-1,1]^n \rightarrow Q \cup \{\mu\}$$
$$\mathcal{Q}_Q(\mathbf{x}) = \begin{cases} q_i & \text{if} \quad \Gamma(x_j;\rho,\rho) = x^b_{ij} \quad \forall i = 1 \dots n \\ \mu & \text{otherwise.} \end{cases} \tag{30}$$

where $\rho \in [0,1)$ is an appropriate threshold value. The ranges of the two above quantization functions contain two *nil* elements, λ and μ.

The transition function depends on a state/input pair. So, a basic requirement of the implementation is that of detecting such a pair, in order to produce the appropriate computation (i.e. state switching). According to the above scheme, let us define:

$$\mathbf{z}(t) \doteq [x_1(t), \cdots, x_n(t), u_1(t), \cdots u_m(t)]' \in [-1,1]^{n+m}. \tag{31}$$

The information contained in this vector defines the computation which must be carried out. We call it the *rule vector*. We also define the *binary rule vector* as

$$\mathbf{z}^b(t) \doteq [x^b_1(t), \cdots, x^b_n(t), u^b_1(t), \cdots u^b_m(t)]' \in \{-1,1\}^{n+m}. \tag{32}$$

Once a coding function has been defined for inputs and states, the automata rules can be rewritten as switching rules for each bit in the state binary vector $\mathbf{x}^b$. This concept is formalized in the following definition:

Definition 7 (Binary excitation function) *Let $\mathcal{A}_d$ be a DFA and consider two coding functions $\mathcal{C}_Q$ and $\mathcal{C}_\Sigma$. Define*

$$\begin{aligned} \mathbf{x}^b(t) &\doteq \mathcal{C}_Q\left(q(t)\right) \\ \mathbf{u}^b(t) &\doteq \mathcal{C}_\Sigma\left(\sigma(t)\right) \\ \mathbf{x}^b(t+1) &\doteq \mathcal{C}_Q\left(\Phi_d\left(q(t),\sigma(t)\right)\right). \end{aligned}$$

A function

$$E : \{-1,+1\}^n \times \{-1,+1\}^m \longmapsto \{-1,0,+1\}^n \tag{33}$$

is called a binary excitation function *for $\mathcal{A}_d$ if the following relationships hold for each component $j = 1 \dots n$:*

$$\begin{array}{llll} E_j\left(\mathbf{x}^b(t),\mathbf{u}^b(t)\right) = +1 & \textit{if} \;\; x^b_j(t) = -1 & \textit{and} & x^b_j(t+1) = +1; \\ E_j\left(\mathbf{x}^b(t),\mathbf{u}^b(t)\right) = -1 & \textit{if} \;\; x^b_j(t) = +1 & \textit{and} & x^b_j(t+1) = -1; \\ E_j\left(\mathbf{x}^b(t),\mathbf{u}^b(t)\right) \neq -1 & \textit{if} \;\; x^b_j(t) = +1 & \textit{and} & x^b_j(t+1) = +1; \\ E_j\left(\mathbf{x}^b(t),\mathbf{u}^b(t)\right) \neq +1 & \textit{if} \;\; x^b_j(t) = -1 & \textit{and} & x^b_j(t+1) = -1. \end{array} \tag{34}$$

■

This function can be interpreted as follows: its j^{th} component $E_j\left(\mathbf{x}^b(t),\mathbf{u}^b(t)\right)$ is either zero or the value of the j^{th} bit in the next state code; it can be zero only if that bit is not required to switch in the state transition occurring at time t.

VI Adapting the original NFA for the implementation

As pointed out in section IV, the special form of time-warping nondeterministic behavior can be provided by sigmoidal neurons with self-loop connections which allow us to control state switching with variable duration. In the approach we propose, the assigned NFA $\mathcal{A}_n$ is not implemented directly. Instead, an NFA $\mathcal{A}'_n$ is derived from $\mathcal{A}_n$. The two automata are essentially equivalent, apart from a slight difference in modeling the input symbol duration. We begin this section explaining why this intermediate step is required. Thereafter we describe an algorithm for devising $\mathcal{A}'_n$ from the assigned NFA $\mathcal{A}_n$ and we state a formal relationship between the two automata. Because of the special form of nondeterministic behavior we consider, the reduction of $\mathcal{A}_n$ to $\mathcal{A}'_n$ is based on their associated deterministic reductions $\mathcal{A}_d$ and $\mathcal{A}'_d$.

As we have seen in section IV, state encoding neurons switch under the control of their generalized inputs $\mathcal{I}_i(t)$. This switching is used to perform state transition. Hence, the rules governing neuron switching must reflect the state transition function Φ. Said another way, the generalized input $\mathcal{I}_i(t)$ of neuron encoding state i is the continuous domain counterpart of the excitation function $E_i(t)$ defined in the symbolic domain. Since $E_i(t)$ depends on $\mathbf{z}^b(t-1)$, $I_i(t)$ should depend on the continuous rule vector $\mathbf{z}(t-1)$.

An important point of the proposed implementation is the transition duration, which is controlled by $w_{ii}^{(1)}$ and the absolute value of $\mathcal{I}_i(t)$ (see also [10]). Clearly, the value of $\mathcal{I}_i(t)$ must satisfy the condition expressed by point 3 of Theorem 1 during the time interval necessary to complete the transition. Concerning the external inputs, it is a basic assumption that they are nearly constant during the time necessary to complete the transition. On the other hand, it is a design issue to guarantee that inequalities (15) are not affected by the change in the state components of $\mathbf{z}(t-1)$, during the transition. To accomplish this, two steps must be taken.

First, we compute $\mathcal{I}_i(t)$ as function of all the components of $\mathbf{z}(t-1)$, but the one corresponding to the i^{th} state. Formally, we define the *reduced rule vector* as:

$$\begin{aligned} \mathbf{z}_i &= [u_1, \cdots, u_m, x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n]' \in [-1,1]^{n+m-1} \\ i &= 1 \ldots n. \end{aligned} \tag{35}$$

The binary counterpart $\mathbf{z}_i^b$ is defined in a similar way. This prevents a direct dependence of $\mathcal{I}_i(t)$ upon the output $x_i(t-1)$.

Second, in order to control the switching conditions shown in Theorem 1, the adoption of state codings with Hamming distance equal one between any pair $(q(t), q(t+1))$ of the STF turns out to be useful. In so doing, during state transition, we must drive the change of only one bit in the state portion of binary rule vector. Unfortunately, it is not possible to encode any set of state symbols with the requirement that each pair of codes have unitary Hamming distance [6].

[6] Consider the boolean n-hypercube: the coding maps each symbol into a vertex; two codes have unitary Hamming distance if they are adjacent in the hypercube.

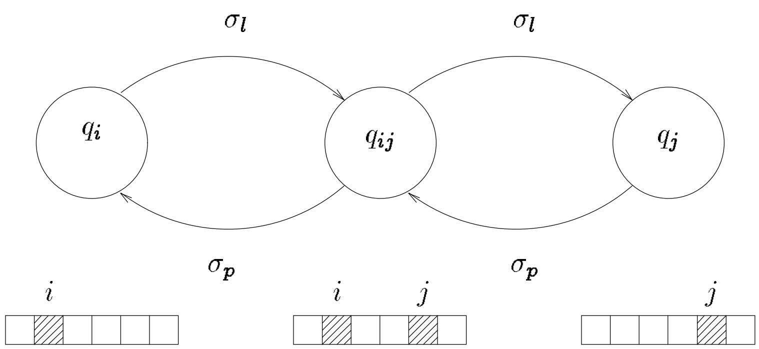


Figure 2: Example of state transition using a temporary intermediate state. State codes are depicted below the diagram.

By comparison, in the approach proposed in [10] the Hamming distance from consecutive states was always one, because of the state coding assumption. This essentially depended on the *chain-like* hypotheses assumed for the transition function. In a chain-like automaton, once a state is exited, it cannot be reached again. In the present case, no restrictions are imposed on the transition function of the deterministic reduction of the automaton to be implemented.

The unitary Hamming distance requirement can be fulfilled by introducing a temporary state for each pair of the original automata states. The code for the temporary state is obtained by or-ing the codes of the two original states involved in the transition. The automata state transition is split in two transitions. The first one leads the automaton in the temporary state; the second one leads the automaton in the original next state. This method is illustrated in Fig. 2. Notice that, because of the coding assumptions, we have $q_{ij} = q_{ji}$ for the modified automaton. In Fig. 3 we give an example of the introduction of temporary states.

Temporary states make it possible to perform state transition by involving the switching of one neuron at a time. However, a couple of difficulties may arise. In the following subsection we explain how to preserve the basic features of the original automaton, in spite of the presence of temporary states.

A Mutually consecutive states

A first problem may arise if $\mathcal{A}_d$ contains two states which are mutually consecutive for a same input symbol, i.e.

$$\begin{array}{l} \exists\sigma \in \Sigma, \exists q_i, q_j \in Q \;\; : \\ \begin{cases} \Phi_d(q_i, \sigma) = q_j & \text{and} \\ \Phi_d(q_j, \sigma) = q_i. \end{cases} \end{array} \tag{36}$$

At this level we are dealing with a DFA. Therefore we must guarantee that the introduction of temporary states does not produce ambiguity. Since $q_{ij} = q_{ji}$, indetermination may arise if

Φ_d	σ_1	σ_2	σ_3	σ_4
0001 (q_1)	q_1	q_2	q_3	q_2
0010 (q_2)	q_4	q_2	q_2	q_4
0100 (q_3)	q_3	q_4	q_4	q_2
1000 (q_4)	q_3	q_2	q_4	q_4

Φ_d	σ_1	σ_2	σ_3	σ_4
0001 (q_1)	0001	0011	0101	0011
0010 (q_2)	1010	0010	0010	1010
0100 (q_3)	0100	1100	1100	0110
1000 (q_4)	1100	1010	1000	1000
0011 (q_{12})	-	0010	-	0010
0101 (q_{13})	-	-	0100	-
1010 (q_{24})	1000	0010	-	1000
1100 (q_{34})	0100	1000	1000	-
0110 (q_{23})	-	-	-	0010

Figure 3: Example of introduction of temporary states. On the left: the state transition table for the assigned DFA $\mathcal{A}_d$; on the right, the modified DFA $\mathcal{A}_d$ with temporary states. Coding vectors are also shown. For better readability we use here the boolean constants 0 and 1 instead of -1 and $+1$.

Φ_d	σ_1	σ_2	σ_3
$\mathbf{q_1}$	q_2	q_3	$\mathbf{q_3}$
q_2	q_2	q_1	q_3
$\mathbf{q_3}$	q_2	q_3	$\mathbf{q_1}$

Φ_d	σ_1	σ_2	σ_3
q_1	q_2	q_3	q_4
q_2	q_2	q_1	q_3
q_3	q_2	q_3	q_1
q_4	q_2	q_4	q_5
q_5	q_2	q_3	q_3

Figure 4: Example of automaton having mutually consecutive states. On the right an equivalent automaton is shown, which has not mutually exclusive states. Concerning acceptance, the state q_4 is equivalent to q_3, and q_5 is equivalent to q_1.

the condition (36) occurs. For example consider the automaton in Fig. 4. If temporary states are added to that automaton, q_{13} would have two next-states for the input symbol σ_3. This ambiguity can be removed augmenting the number of states of the automaton. Figure 4 provides an example of such an operation, which is formalized in step 2 of Algorithm 1.

B Effects on state duration

Because of temporary states, no transition of the original automaton can be carried out in a single time step. However, variable length transition is a basic feature of the model, so this should not be considered as a serious drawback [7]. Moreover, it is possible to arrange the network to perform fast transitions when exiting from temporary states. In fact, any transition is produced

[7] If the application requires single step transitions, this drawback can be removed by duplicating the input frequency.

by changing one bit in the state encoding set of neurons. The switching is from low to high when entering a temporary state, and from high to low when exiting. On the other hand, neuron switching speed is controlled by the absolute value of the generalized input. Then, if we modify eq. (15) in this way:

$$\begin{aligned} \mathcal{I}_i(t) &> K^+ \mathcal{I}_i^* \\ \mathcal{I}_i(t) &< -K^- \mathcal{I}_i^*, \end{aligned} \tag{37}$$

we can separately adjust switching speed for the two cases, according to the values of the two constants K^+ and K^-. Choosing constant K^- large enough, we can assure exiting from temporary states in just one step. In so doing, the influence of temporary states in modeling state duration can be kept to a minimum.

C Avoiding getting stuck in temporary states

Consider temporary state q_{ij} associated to the generic pair of states q_i and q_j. The next state is only defined for those inputs which produce a transition from q_i to q_j or viceversa. In the other cases no state transition is explicitly required (see dashes in Fig. 3). So we could simply set the value of the excitation function to zero in those cases. This apparently natural solution has a serious drawback. Suppose, while in the state q_{ij}, the input symbol which started the transition disappears. If no transition is specified, the automaton gets stuck until a new symbol arrives which has a defined new state for that row. This can dramatically modify the behavior of the automaton and must be avoided by imposing escaping rules from temporary states, even if the incoming symbol is not one of those which began the transition. Since Hamming distance between consecutive states must be one, there are only two possibilities. We can either decide to complete the transition anyhow, or decide to go back to the starting state. The relevance of the decision should be evaluated bearing in mind the effects on state duration modeling. In one case we permit a transition even if the symbol lasts a few steps less than the required number. In the other case we deny the transition because the symbol duration is too short. Since we are dealing with a case at the border of the correct duration, the influence is modest and the two alternatives can be assumed equally acceptable from the point of view of state duration modeling.

D Translation of the assigned automaton

We can now define an algorithm which modifies the assigned NFA $\mathcal{A}_n$ and generates a target NFA $\mathcal{A}'_n$ which is directly suitable for the neural network implementation.

Algorithm 1 *Given the NFA $\mathcal{A}_n = \{\Sigma, Q, \Phi_n, R, F\}$, an NFA $\mathcal{A}'_n$ is derived according to the following rules:*

1. *Initialize $\mathcal{A}'_d$ as the deterministic reduction of $\mathcal{A}_n$.*

2. (Avoid mutually consecutive states):

For $i = 1 \ldots n$

For l*=1* $\ldots m$

Denote $q_j = \Phi'_d(q_i, \sigma_l)$*;*

if $\Phi'_d(q_j, \sigma_l) = q_i$ *then*

2.1 $Q' \leftarrow Q' \cup \{q_{n+1}, q_{n+2}\}$;

2.2 *for* $k = 1 \ldots m$*:*

- *if* $\Phi'_d(q_i, \sigma_k) = q_j$ *and* $\Phi'_d(q_j, \sigma_k) = q_i$ *then*

$$\begin{aligned} &\Phi'_d(q_i, \sigma_k) \leftarrow q_{n+1}; \\ &\Phi'_d(q_{n+1}, \sigma_k) \leftarrow q_{n+2}; \\ &\Phi'_d(q_{n+2}, \sigma_k) \leftarrow q_j; \end{aligned} \tag{38}$$

- *else*

$$\begin{aligned} \Phi'_d(q_{n+1}, \sigma_k) &\leftarrow \begin{cases} q_{n+1} & \textit{if} \quad \Phi'_d(q_j, \sigma_k) = q_j \\ \Phi'_d(q_j, \sigma_k) & \textit{otherwise;} \end{cases} \\ \Phi'_d(q_{n+2}, \sigma_k) &\leftarrow \begin{cases} q_{n+2} & \textit{if} \quad \Phi'_d(q_i, \sigma_k) = q_i \\ \Phi'_d(q_i, \sigma_k) & \textit{otherwise;} \end{cases} \end{aligned} \tag{39}$$

2.3 *if* $q_j \in F$ *then* $F \leftarrow F \cup \{q_{n+1}\}$;

if $q_i \in F$ *then* $F \leftarrow F \cup \{q_{n+2}\}$;

3. (Introduce temporary states):

For each pair of states $q_i, q_j \in Q, i \neq j \;:\; \exists \overline{\sigma} \in \Sigma : \;\Phi_d(q_i, \overline{\sigma}) = q_j$

3.1 *if* $q_{ij} \notin Q'$ *then add a new state* q_{ij}*:* $Q' \leftarrow Q' \cup \{q_{ij}\}$;

3.2 *modify* Φ'_d *as follows:*

$$\begin{aligned} \Phi'_d(q_i, \overline{\sigma}) &\leftarrow q_{ij}; \\ \Phi'_d(q_{ij}, \overline{\sigma}) &\leftarrow q_j; \end{aligned} \tag{40}$$

and add escape rules: $\forall \sigma \neq \overline{\sigma}$ *choose one of:*

$$\begin{aligned} \Phi'_d(q_{ij}, \sigma) &\leftarrow q_i \quad \textit{or} \\ \Phi'_d(q_{ij}, \sigma) &\leftarrow q_j; \end{aligned} \tag{41}$$

4. *Set* $\mathcal{A}'_n$ *to the nondeterministic extension of* $\mathcal{A}'_d$*.*

For the new automaton $\mathcal{A}'_n$, we derive a new state coding function $\mathcal{C}'_Q()$ from $\mathcal{C}_Q()$, as follows:

$$\begin{aligned} \mathcal{C}'_Q(q_i) &= \mathcal{C}_Q(q_i) \quad \forall\; q_i \in Q \\ \mathcal{C}'_Q(q_{ij}) &= \mathcal{C}_Q(q_i) \vee \mathcal{C}_Q(q_j) \quad \forall q_{ij} \;\in Q' - Q. \end{aligned} \tag{42}$$

The new automaton $\mathcal{A}'_n$ is strictly related to $\mathcal{A}_n$, as stated in the following theorem:

Theorem 3 *Let $\mathcal{A}_n$ be an NFA and let $\mathcal{A}'_n$ be the NFA obtained from $\mathcal{A}_n$ by using Algorithm 1. Given a string v of length T*

$$v = \sigma(1)\sigma(2)\cdots\sigma(T) \in \Sigma^*$$

consider the string w of length $2T$ defined as follows:

$$w = \sigma(1)\sigma(1)\sigma(2)\sigma(2)\cdots\sigma(T)\sigma(T).$$

Then we have

$$v \in L(\mathcal{A}_n) \;\; \text{if and only if} \;\; w \in L(\mathcal{A}'_n). \tag{43}$$

Proof
See Appendix. ■

E Reduced rule vectors

The generalized input $\mathcal{I}_i(t)$ must be computed according to the switching rules for the bit x_i^b, and these rules depend on the present state. On the other hand, according to the considerations at the beginning of this section, we want to compute $\mathcal{I}_i(t)$ as a function of the reduced rule vector $\mathbf{z}_i(t)$. This is necessary in order to avoid direct dependence of $\mathcal{I}_i(t)$ on $x_i(t)$. Now we show that the switching rules for neuron i can be computed without using the i^{th} bit of the state coding string. A constructive proof will be given in the symbolic domain, using the symbolic counterpart E_i of the generalized input.

For each $i = 1 \ldots n$ define the set of pairs

$$\mathcal{H}_i \doteq \left\{(j,l) : q(t) = q_j, \sigma(t) = \sigma_l, x_i^b(t) = -1, x_i^b(t+1) = +1\right\}. \tag{44}$$

From (44) and Definition 7 it follows that

$$(j,l) \in \mathcal{H}_i \quad \Longrightarrow \quad E_i\left(\mathcal{C}'_Q(q_j), \mathcal{C}_\Sigma(\sigma_l)\right) = +1. \tag{45}$$

Similarly, define

$$\mathcal{L}_i \doteq \left\{(j,l) : q(t) = q_{ij}, \sigma(t) = \sigma_l, x_i^b(t) = +1, x_i^b(t+1) = -1\right\}. \tag{46}$$

We have

$$(j,l) \in \mathcal{L}_i \quad \Longrightarrow \quad E_i\left(\mathcal{C}'_Q(q_{ij}), \mathcal{C}_\Sigma(\sigma_l)\right) = -1. \tag{47}$$

Now, for each pair $(j,l) \in \mathcal{H}_i$ introduce the boolean variables:

$$R_{ij,l}^b \doteq \left(\bigwedge_{k=1, k\neq i, k\neq j}^{n} \overline{x}_k^b\right) \wedge x_j^b \wedge u_l^b. \tag{48}$$

Then for each pair $(j,l) \in \bigcup_{k=1}^{n} \mathcal{L}_k$ introduce the boolean variables:

$$P^b_{j,l} \doteq x^b_j \wedge u^b_l. \tag{49}$$

Finally, for each $i = 1 \dots n$ introduce

$$\begin{aligned} H^b_i &\doteq \bigvee_{(j,l)\in\mathcal{H}_i} R^b_{ij,l}; \\ L^b_i &\doteq \bigvee_{(j,l)\in\mathcal{L}_i} P^b_{j,l}. \end{aligned} \tag{50}$$

The meaning of the variables defined in (50) is the following:

- if $H^b_i = +1$ and $L^b_i = -1$ then x^b_i must switch from low to high;
- if $H^b_i = -1$ and $L^b_i = +1$ then x^b_i must switch from high to low;
- if $H^b_i = -1$ and $L^b_i = -1$ then no switch is required for the bit x^b_i.

As a result, the information encoded in the variables H^b_i and L^b_i, $i = 1 \dots n$ is sufficient for determining the next state of the automaton. Moreover H^b_i and L^b_i are computed without using x^b_i. This proves the following

Theorem 4 *Consider the time-warping NFA $\mathcal{A}'_n$ obtained from Algorithm 1 and let $\mathcal{A}'_d$ be its deterministic reduction. Also suppose that the binary encoding functions $\mathcal{C}'_Q$ (42) are used. Let E be a function having components*

$$E_i(\mathbf{x}^b, \mathbf{u}^b) = \begin{cases} +1 & \textit{if} \quad H^b_i = +1 \quad \textit{and} \quad L^b_i = -1 \\ -1 & \textit{if} \quad H^b_i = -1 \quad \textit{and} \quad L^b_i = +1 \\ 0 & \textit{if} \quad H^b_i = -1 \quad \textit{and} \quad L^b_i = -1 \end{cases} \tag{51}$$

where H^b_i and L^b_i are computed according to eqs.(48 - 50). Then E_i does not depend on x^b_i and E is a binary excitation function *for $\mathcal{A}'_d$, according to Definition 7. Moreover, $H^b_i \wedge L^b_i = -1$, $\forall\ i = 1 \dots n$.*

■

VII Injection of NFA into the recurrent network

In this section we show how an NFA can be injected into a recurrent network. The algorithm takes as input the DFA $\mathcal{A}'_d$, obtained as deterministic reduction of $\mathcal{A}'_n$, which in turn is derived from the assigned NFA $\mathcal{A}_n$ by using Algorithm 1. It produces as output a recurrent neural network architecture, intended as a collection of units and connections among units. The weights are specified by means of linear constraints. An initial set of weights is also computed, which satisfies those constraints. Finally a proof is given, which guarantees that the resulting RNN is actually an implementation of the NFA $\mathcal{A}'_n$.

A Network generation algorithm

This subsection is devoted to describe the algorithm which generates the RNN.

Algorithm 2

1. *Assign an input neuron U_l for each external input $l = 1 \dots m$; generate a special input neuron 0, used to provide biases; initialize $\mathcal{G}$ as follows:*

$$\begin{aligned} &\mathcal{U} \leftarrow \{0, U_1, U_2, \dots, U_m\}; \\ &\mathcal{N} \leftarrow \emptyset; \\ &\mathcal{W} \leftarrow \emptyset. \end{aligned} \tag{52}$$

2. *For $l = 1 \dots m$:*

2.1 *Generate a neuron V_l and connect it to all input neurons:*

$$\begin{aligned} &\mathcal{N} \leftarrow \mathcal{N} \cup \{V_l\}; \\ &S^{(0)}_{V_l} \leftarrow \{0, U_1, \cdots, U_m\}; \\ &\mathcal{W} \leftarrow \mathcal{W} \cup \{V_l\} \times S^{(0)}_{V_l}. \end{aligned} \tag{53}$$

2.2 *Initialize the input weights of V_l in order to obtain a boolean-like AND function — see eq. (21) —, using ρ_1 as input quantization threshold and ρ as output quantization threshold. The neuron V_l must implement the logic function*

$$V^b_l = \left(\bigwedge_{j=1, j \neq l}^{m} \overline{u^b_j} \right) \wedge u^b_l. \tag{54}$$

2.3 *Set constraints on the above weights according to eqs. (20).*

3. *For $i = 1 \dots n$:*

3.1 *Generate three neurons: i used as component of the state coding vector; H_i and L_i used to generate the generalized input for neuron i:*

$$\begin{aligned} &\mathcal{N} \leftarrow \mathcal{N} \cup \{i, H_i, L_i\}; \\ &S^{(1)}_i \leftarrow \{0, i, H_i, L_i\}; \\ &\mathcal{W} \leftarrow \mathcal{W} \cup \{i, H_i, L_i\} \times S^{(1)}_i. \end{aligned} \tag{55}$$

3.2 *Chose a value ρ to be used as quantization threshold for the information latched by the above introduced neurons i. Then choose a value $w^{(1)}_{ii} > w^*_\rho$ according to eq. (13).*

3.3 *During learning the weight $w^{(1)}_{i,i}$ is fixed to the value assigned in the previous step. Concerning the other weights set constraints according to the following equations (see also Lemma 2 and Theorem 1):*

$$\begin{aligned} &w^{(0)}_{i,0} + \rho^+_3 w^{(1)}_{i,H_i} - \rho^-_3 w^{(1)}_{i,L_i} > \mathcal{I}^*_i \\ &w^{(0)}_{i,0} - \rho^-_3 w^{(1)}_{i,H_i} + \rho^+_3 w^{(1)}_{i,L_i} < -\mathcal{I}^*_i \\ &w^{(0)}_{i,0} - \rho^-_3 w^{(1)}_{i,H_i} - w^{(1)}_{i,L_i} < \mathcal{I}^*_i \\ &w^{(0)}_{i,0} - w^{(1)}_{i,H_i} - \rho^-_3 w^{(1)}_{i,L_i} > -\mathcal{I}^*_i. \end{aligned} \tag{56}$$

3.4 *Set the weights* $w^{(0)}_{i,0}, w^{(1)}_{i,H_i} w^{(1)}_{i,L_i}$ *to an initial value which satisfies the above constraints. This can be obtained easily by linear programming.*

4. *For* $i = 1 \ldots n$ *:*

4.1 *For each pair* $(j, l) \in \mathcal{H}_i$ *— cf. eq(44):*

4.1.1 *Generate a new neuron* $R_{ij,l}$

$$\mathcal{N} \leftarrow \mathcal{N} \cup \{R_{ij,l}\}; \tag{57}$$

4.1.2 $\forall k \neq i$ *connect each neuron* k *to* $R_{ij,l}$*; also connect* V_l *to* $R_{ij,l}$*:*

$$\begin{aligned} &S^{(0)}_{R_{ij,l}} \leftarrow \{0, 1, \ldots, i-1, i+1, \ldots, n, V_l\}; \\ &\mathcal{W} \leftarrow \mathcal{W} \cup \{R_{ij,l}\} \times S^{(0)}_{R_{ij,l}}. \end{aligned} \tag{58}$$

4.1.3 *Initialize weights towards* $R_{ij,l}$ *in order to obtain a boolean-like AND function — see eq. (21) —, using* ρ *as input quantization threshold and* ρ_2 *as output quantization threshold. The neuron* $R_{ij,l}$ *must implement the logic function:*

$$R^b_{ij,l} \doteq \left(\bigwedge_{k=1, k\neq i, k\neq j}^{n} \overline{x}^b_k \right) \wedge x^b_j \wedge V^b_l. \tag{59}$$

4.1.3 *Set constraints on the above weights, according to inequalities (20).*

5. *For each pair* $(j, l) \in \bigcup_{i=1}^{n} \mathcal{L}_i$ *— cf. (46):*

5.1 *Generate a new neuron* $P_{j,l}$

$$\mathcal{N} \leftarrow \mathcal{N} \cup \{P_{j,l}\}. \tag{60}$$

5.2 *Connect bias,* j *and* V_l *to* $P_{j,l}$*:*

$$\begin{aligned} &S^{(0)}_{P_{j,l}} \leftarrow \{0, j, V_l\} \\ &\mathcal{W} \leftarrow \mathcal{W} \cup \{0, j, V_l\} \times S^{(0)}_{P_{j,l}}. \end{aligned} \tag{61}$$

5.3 *Initialize the weights towards* $P_{j,l}$ *in order to obtain a boolean-like AND function — see eq. (21) —, using* ρ *as input quantization threshold and* ρ_2 *as output quantization threshold. The neuron* $P_{j,l}$ *must implement the logic function:*

$$P^b_{j,l} = x^b_j \wedge V^b_l. \tag{62}$$

6. *For* $i = 1 \ldots n$*:*

6.1 *For each pair* $(j, l) \in \mathcal{H}_i$*:*

6.1.1 *connect $R_{i,j,l}$ to H_i:*

$$\begin{array}{l} S^{(0)}_{H_i} \leftarrow \{R_{i,j,l}\} \ \ \forall\ (j,l) : E_i(q_j, \sigma_l) = +1; \\ \mathcal{W} \leftarrow \mathcal{W} \cup \{R_{i,j,l}\} \times S^{(0)}_{H_i}. \end{array} \tag{63}$$

6.1.2 *Initialize the weights towards H_i in order to obtain a boolean-like OR function using ρ_2 as input quantization threshold and ρ_3 as output quantization threshold. The neuron H_i must implement the logic function:*

$$H^b_i = \bigvee_{(j,l)\in\mathcal{H}_i} R^b_{i,j,l}. \tag{64}$$

6.1.3 *Set constraints on the above weights, according to inequalities (25).*

7. *For $i = 1 \ldots n$:*

7.1 $\forall (j,l) \in \mathcal{L}_i$ *connect $P_{j,l}$ to L_i:*

$$\begin{array}{l} S^{(0)}_{L_i} \leftarrow \{P_{j,l}\} \ \ \forall (j,l) : E_i(q_j, \sigma_l) = -1; \\ \mathcal{W} \leftarrow \mathcal{W} \cup \{P_{j,l}\} \times S^{(0)}_{L_i}. \end{array} \tag{65}$$

7.2 *Initialize the weights towards L_i in order to obtain a boolean-like OR function using ρ_2 as input quantization threshold and ρ_3 as output quantization threshold. The neuron L_i must implement the logic function:*

$$L^b_i = \bigvee_{(j,l)\in\mathcal{L}_i} P^b_{j,l}. \tag{66}$$

7.3 *Set constraints on the above weights, according to inequalities (25).*

Some points need to be explained in order to clarify how the above algorithm works. Two kinds of neurons are allocated: neurons used to encode and store the state information, and neurons used to compute, in a boolean-like fashion, the state transition function. All the outputs of these neurons can be given a boolean interpretation by using quantization thresholds, according to Definition 3. Algorithm 2 uses five different quantization thresholds. Here we explain how to chose appropriate values for these thresholds.

Threshold ρ_1 is used for the input to neurons V_l. These neurons have been introduced to protect the network against unpredictable behaviors which could arise in presence of arbitrary external input (also cf. Lemma 3). According to (22):

$$\frac{m-1}{m} < \rho_1 < 1. \tag{67}$$

Threshold ρ is used for the input of neurons $R_{ij,l}$ and $P_{i,l}$. They are fed by neurons encoding state and by neurons V_l. Hence, the same value ρ must also be used for the output of neurons

Neurons	Number	Number of weights
V_l	m	$m(m+1)$
$R_{ik,l}$	K	$K(n+1)$
$P_{i,l}$	X	$3X$
H_i	n	$K+n$
L_i	n	$X+n$
state	n	$4n$

Table 1: Complexity of the recurrent neural network generated by Algorithm 2.

encoding state and for neurons V_l. The value of ρ is restricted by the fan-in of neurons $R_{ij,l}$, and therefore

$$\frac{n-1}{n} < \rho < 1. \tag{68}$$

The output threshold for neurons $R_{ij,l}$ and $P_{i,l}$ is ρ_2. This value must be chosen according to the limitation coming from the maximum fan-in of their destination neurons (i.e. the OR neurons). Actually, a different value of ρ_2 could be adopted for each of these neurons. Finally, the values ρ_3^- and ρ_3^+ must be chosen for guaranteeing that the linear constraints (56) admit a feasible solution.

B Neural network complexity

We now wish to provide an upper bound on the complexity of the RNN graph obtained by using Algorithm 2. We are interested in the number of neurons $|\mathcal{N}|$ and the number of weights $|\mathcal{W}|$. These numbers will be estimated as a function of the following parameters :

- the number of states $n = |Q|$ in the original automaton $\mathcal{A}_n$ [8];
- the number of input symbols $m = |\Sigma|$;
- the number of neurons $R_{ik,l}$, which is equal to the number K of transitions involving a state change in the deterministic reduction $\mathcal{A}_d$ of the assigned NFA $\mathcal{A}_n$:

$$K \doteq |\{(i,j) : \Phi_d(q_i, \sigma_l) \neq q_i\}|\,; \tag{69}$$

- the number X of neurons $P_{j,l}$, which is equal to the number of elements in the union of the sets L_i, $i = 1 \ldots n$ defined in eq. (46).

According to Table 1, we have:

$$|\mathcal{N}| = m + K + X + 3n \tag{70}$$

[8] In this context n includes the eventual additional states added by step 1 of Algorithm 1, used to remove mutually consecutive states.

and

$$|\mathcal{W}| = m(m+1) + K(n+1) + 4X + K + 6n. \tag{71}$$

The numbers K and X depend on the original state transition function. Assuming the worst case we have:

$$\begin{aligned} K &\leq m(n-1) \\ X &\leq m(n-1) \end{aligned} \tag{72}$$

and then

$$\begin{aligned} |\mathcal{N}| &\leq 2mn - m + 3n \\ |\mathcal{W}| &\leq m(n^2 + m + 5n - 5) + 6n. \end{aligned} \tag{73}$$

However, the estimation in eq. (73) is very pessimistic since it assumes that for each state there exists a transition towards each other state.

C Effectiveness of the implementation

We now discuss some properties of the neural network architecture obtained so far. Besides the RNN topology, this algorithm defines a region Ω of weight space, bounded by linear surfaces. We call each weight vector belonging to Ω a *feasible weight vector.*

Our first concern is to show how the computation of the RNN is related to the computation of the deterministic reduction $\mathcal{A'}_d$ of $\mathcal{A'}_n$. This is stated in the following theorem.

Theorem 5 *Consider a time-warping NFA $\mathcal{A}_n$ and its deterministic reduction $\mathcal{A}_d$. Let $\mathcal{A'}_d$ be the deterministic reduction of the NFA $\mathcal{A'}_n$, obtained by applying Algorithm 1. Consider the RNN architecture obtained from $\mathcal{A'}_d$ by using Algorithm 2. Denote with $\mathbf{u}(t)$ the external input vector to RNN, and with $\mathbf{x}(t)$ the vector of outputs of state encoding neurons. Then, for each feasible weight vector there exists a* finite *time interval ΔT such that:*

$$\begin{aligned} \textit{if} \quad & \mathcal{Q}_Q(\mathbf{x}(\bar{t})) = q_k \in Q \qquad \textit{and} \\ & \mathcal{Q}_\Sigma(\mathbf{u}(\bar{t}+t)) = \sigma_l \in \Sigma \\ & \forall t = 0, 1, \ldots, \Delta T - 1 \end{aligned} \tag{74}$$

then

$$\mathcal{Q}_Q(\mathbf{x}(\bar{t} + \Delta T)) = \Phi_d(q_k, \sigma_l). \tag{75}$$

Proof

See Appendix. ■

The interpretation of the theorem is the following: if the RNN initial conditions encode a state of the assigned DFA $\mathcal{A}_d$ and if the input is kept constant to a value encoding a valid input symbol, then the computation of the RNN leads, in a finite number of steps, to an encoded representation of the correct next-state of $\mathcal{A}_d$.

The actual value of ΔT in the above theorem depends on the choice of weight vector in Ω. In particular, it strongly depends on the weights appearing in the expressions of the generalized inputs of state encoding neurons. The switching speed, and hence the state transition speed, depends in fact on the absolute value of $\mathcal{I}_i$. If this value is slightly greater than $\mathcal{I}_i^*$ a very slow state transition occurs [10].

A remarkable consequence of these effects on state transition duration is that once weights are fixed, the computation carried out by the RNN, for a given input sequence, is only one of the computations of the NFA. Said another way, the language recognized by the RNN is a subset of the language recognized by $\mathcal{A'}_n$. If we use learning by example to refine the weights, this language is optimally selected according to the training sequences. At the same time, by constraining the weights into the feasible region, we can guarantee that the RNN rejects each input sequence that would be rejected by $\mathcal{A'}_n$.

Another important issue of this implementation is the behavior with respect to a continuous valued input vector. Theorem 5 assumes the input vectors actually represent input symbols, according to the definition of input quantization function. Now we want to give more insight about the behavior of the RNN for a generic input vector.

An important concern is to guarantee that no input vector could lead the network into an undefined state. This is stated precisely by the following theorem:

Theorem 6 *Consider the RNN architecture obtained from $\mathcal{A'}_d$ by using Algorithm 2. Let $\mathbf{s}(t) \in \{-1, +1\}^n$ be a boolean vector defined as:*

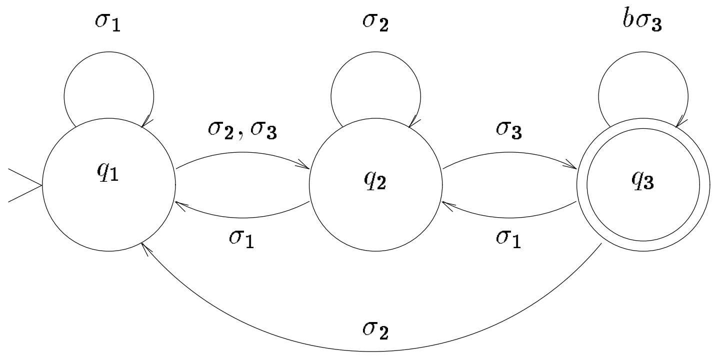
$$s_i(t) \doteq \begin{cases} +1 & \text{if } x_i(t) > 0 \\ -1 & \text{if } x_i(t) \leq 0 \end{cases} \qquad i = 1 \ldots n \tag{76}$$

For each feasible weight vector, for each time $t = 1 \ldots T$ and for each input vector $\mathbf{u}(t)$, there exists a state $q \in \mathcal{Q'}$ such that $\mathbf{s}(t) = \mathcal{C'}_Q(q)$.

Proof

See Appendix. ■

The above theorem guarantees that if we use "sign quantization" for a symbolic interpretation, then any reachable state belongs to $\mathcal{Q'}$. Therefore, the state can not be trapped into undefined states, no matter what the inputs are. This behavior is particularly desirable as a reaction to noisy inputs. Moreover, as long as such inputs disappear, a correct state is reached in a few steps also assuming the boolean quantization functions defined in section IV. It turns out that an inportant feature of the realization is the rejection of meaningless inputs. The actual rejected region in the input space depends on the choice of weight vector in Ω. In particular, it strongly depends on the weights of the neurons V_l, implementing boolean-like AND functions. If we use learning from examples to refine the weights, the rejected input region is optimally selected according to the training sequences. At the same time, constraining the weights in the feasible region, we are guaranteed that the RNN does not lose input vectors that, according to the input quantization function, define valid input symbols.



Φ'_d	σ_1	σ_2	σ_3
q_1	q_1	q_{12}	q_{12}
q_2	q_{12}	q_2	q_{23}
q_3	q_{23}	q_{31}	q_3
q_{12}	q_1	q_2	q_2
q_{23}	q_2	$\mathbf{q_2}$	q_3
q_{31}	$\mathbf{q_1}$	q_1	$\mathbf{q_3}$

Figure 5: a) State diagram of the example time-warping NFA $\mathcal{A}_n$. b) Tabular representation of the state transition function of the deterministic reduction of $\mathcal{A'}_n$. States in boldface have been introduced as escape rules from temporary states.

VIII An example of the proposed implementation

In this section we provide an example to help the reader better understand the algorithm. A simple automaton $\mathcal{A}_n$ with three states q_1, q_2, q_3 and three input symbols $\sigma_1, \sigma_2, \sigma_3$ is considered. Figure 5a depicts a graphic representation of the state transition function of the deterministic reduction of $\mathcal{A}_n$. As a first step we apply Algorithm 1, in order to obtain the NFA $\mathcal{A'}_n$ with temporary states. In this particular case step 1 of that algorithm is not required. The state transition function of the deterministic reduction of $\mathcal{A'}_n$ is depicted in Figure 5b.

Then, Algorithm 2 is used. The resulting RNN is depicted in Figure 6. The three neurons with self-loops $(1, 2, 3)$ are used to encode states. Each of these three neurons receives its generalized input through a pair of neurons L_i and H_i, which are responsible for producing the appropriate switching. For clarity, the boolean expressions of the switching rules for each state encoding neuron are reported:

$$H_1^b(t) = \left(x_2^b(t-1) \wedge \overline{x_3^b(t-1)} \wedge V_1^b(t-1)\right) \vee \left(x_3^b(t-1) \wedge \overline{x_2^b(t-1)} \wedge V_2^b(t-1)\right) \tag{77}$$

$$L_1^b(t) = \left(x_2^b(t-1) \wedge V_2^b(t-1)\right) \vee \left(x_2^b(t-1) \wedge V_3^b(t-1)\right) \vee \left(x_3^b(t-1) \wedge V_3^b(t-1)\right) \tag{78}$$

$$\begin{aligned} H_2^b(t) = {} & \left(x_1^b(t-1) \wedge \overline{x_3^b(t-1)} \wedge V_2^b(t-1)\right) \vee \left(x_1^b(t-1) \wedge \overline{x_3^b(t-1)} \wedge V_3^b(t-1)\right) \vee \\ & \left(x_3^b(t-1) \wedge \overline{x_1^b(t-1)} \wedge V_1^b(t-1)\right) \end{aligned} \tag{79}$$

$$L_2^b(t) = \left(x_1^b(t-1) \wedge V_1^b(t-1)\right) \vee \left(x_3^b(t-1) \wedge V_3^b(t-1)\right) \tag{80}$$

$$H_3^b(t) = \left(x_2^b(t-1) \wedge \overline{x_1^b(t-1)} \wedge V_3^b(t-1)\right) \tag{81}$$

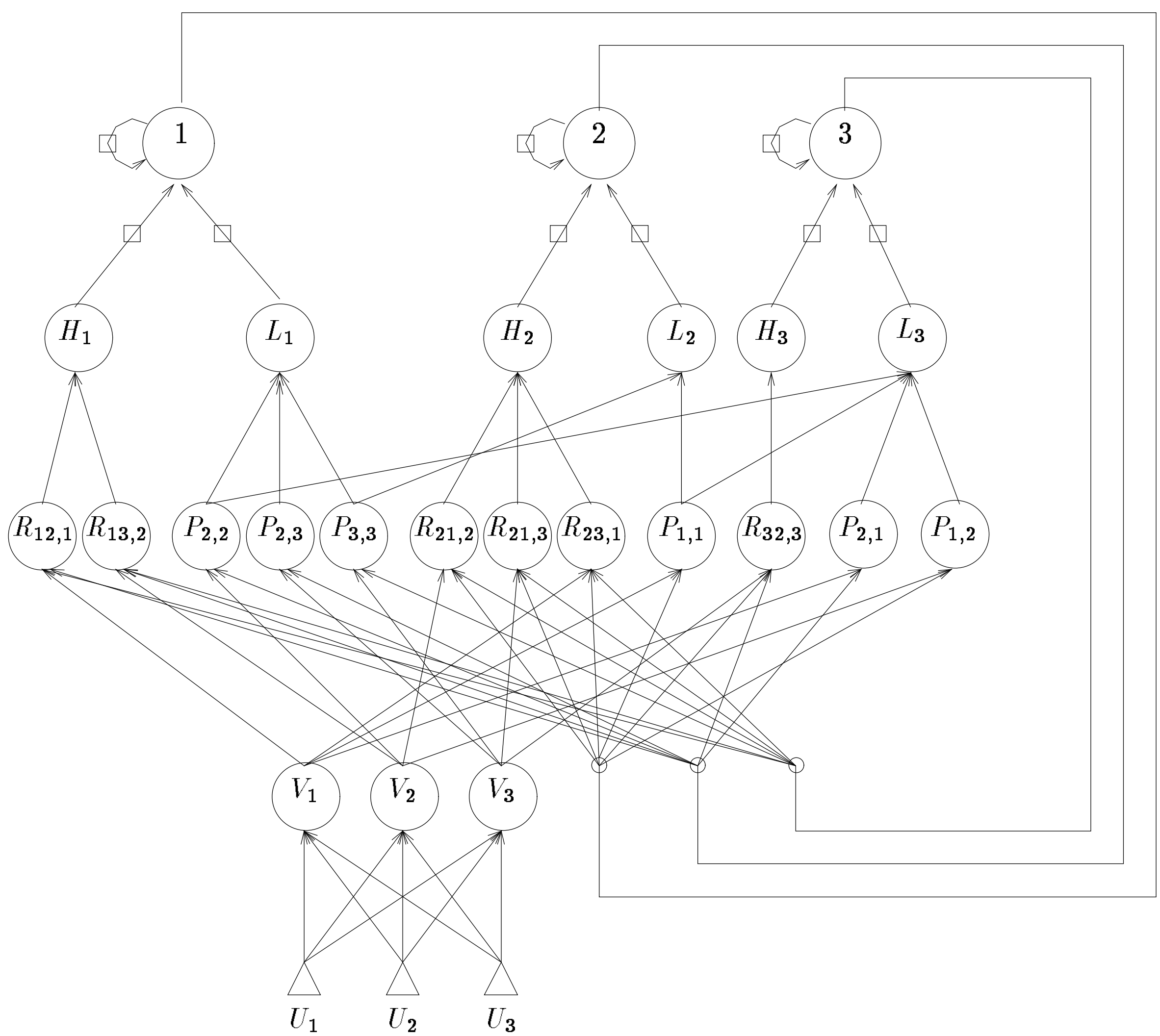


Figure 6: Recurrent Neural Network architecture produced as output of Algorithm 2, applied to the example. For the sake of simplicity, bias connections are not shown. Time delays are depicted as small squares on the connections. Small triangles are used to depict external inputs.

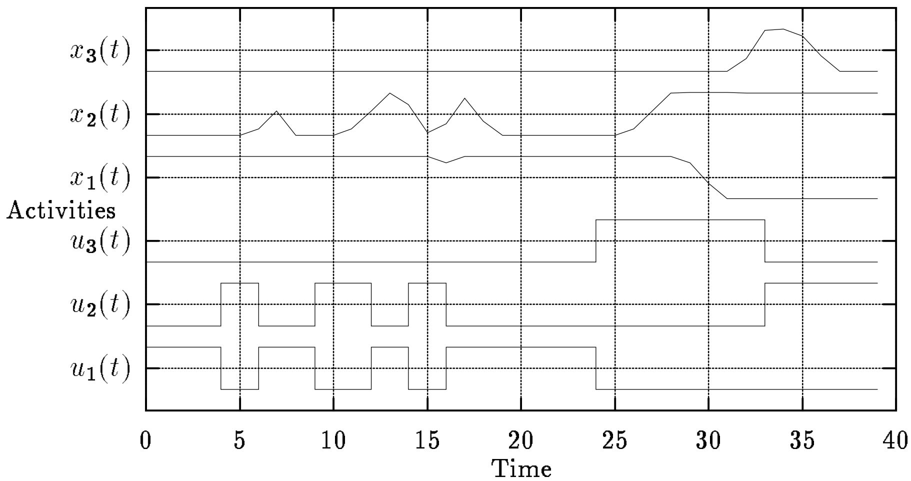


Figure 7: Evolution of state encoding neurons in response to the input sequence encoding the example string. The weights are fixed to values yielding "slow" transitions.

$$
\begin{aligned}
L_3^b(t) \quad = \quad & \left(x_2^b(t-1) \wedge V_1^b(t-1)\right) \vee \left(x_2^b(t-1) \wedge V_2^b(t-1)\right) \vee \left(x_1^b(t-1) \wedge V_1^b(t-1)\right) \vee \\
& \left(x_1^b(t-1) \wedge V_2^b(t-1)\right). \qquad (82) \\
& \qquad (83)
\end{aligned}
$$

The R and L neurons in Figure 6 implement boolean-like AND functions. Each of them is devoted to compute a boolean-like function for one of the minterms occurring in eqs. (77–83). External inputs do not feed directly these neurons. Instead they are *pre-filtered* by a fully connected layer of neurons (V_1, V_2, V_3), shown in the lower part of Figure 6.

Finally we discuss some numerical simulations showing the behavior of the RNN. In particular, our intention is to demonstrate how the weight change (according to the linear constraints) can actually modify the rate of the dynamics, which is reflected in a change in the number of steps required to perform a state transition. For the sake of simplicity we use here binary input values. Therefore, the different behaviors due to changing the input weights are not shown here.

Figures 7, 8 show the time evolution for external inputs and for the outputs of the state encoding neurons. In both cases the same input string is used; the duration of the symbols was randomly chosen The two networks only differ in the choice of the weights inside the feasible region Ω. The two choices actually produce different computations. In Figure 7 the final state is q_2, whereas in Figure 7 the final state is q_1. It is easy to recognize that the transitions for the former network are slower. In particular notice that at time $t \approx 30$ a transition towards q_3 begins, but it is completed only in the second network. Although this is a very simple example,

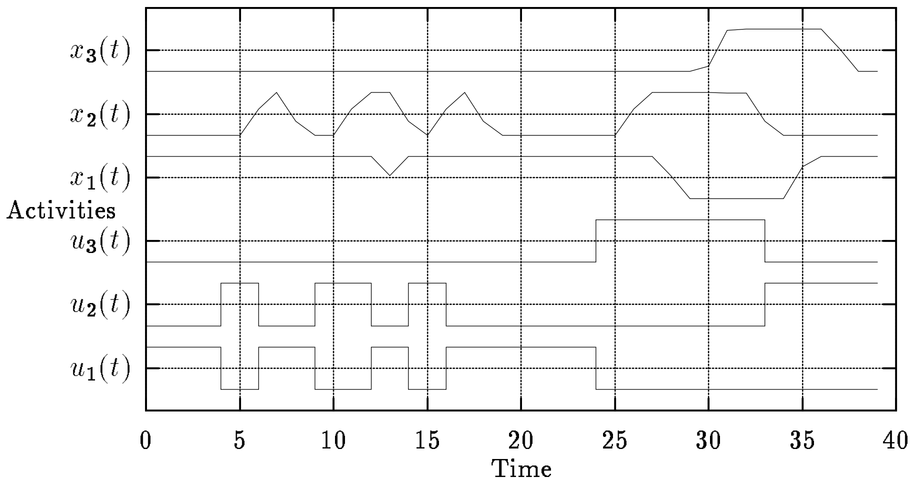


Figure 8: Evolution of state encoding neurons in response to the input sequence encoding the same example string. Weight are fixed to values yielding faster transitions.

it should clarify how moving weights inside Ω can actually modify the behavior of the network.

IX Conclusions

In this paper we have shown how a special kind of NFA can be injected into a RNN in terms of a set of linear constraints in weight space. Instead of learning in the whole weight space, the given rules - assigned by the NFA - allow us to restrict the search for the optimal solution into a special domain where the rules hold. In so doing, we continue to stress the importance of learning from examples, which is certainly one of the most attractive features of neural nets, but using a model biased by the prior rule-based knowledge. These rules, given in terms of the *partial* definition of the transition function of an NFA, represent the *partial* domain knowledge on a given problem.

Bearing in mind the knowledge typical of many speech processing tasks, we propose a special kind of NFA, which is well suited for modeling the time warping phenomena of the speech. Moreover, the realization we provide by RNN of a given automaton is consistent not only for booleans, but also for continuous inputs belonging to given ranges. What is mainly different with respect to other referred approaches [14], [24], [30], [29] is that the given rules are not destroyed during the learning process. We consider it very important, since it allows us to obtain a very clear interpretation of what the net learned from examples. It is also worth mentioning that the architecture and the constraints provided by the proposed realization guarantee that the network will not be trapped in undefined configurations also in presence of noisy inputs. This turns out to be very useful for designing the K block of $K - L$ nets [10].

Appendix

Proof of Theorem 1

The proof uses some results which can be found in [8] and [10], and hence not repeated here. Actually, in [10] a proof of this theorem is given for the special case $\rho \to 0$. Note that for $\rho \to 0$ we have $w_\rho \to 2$, and $w_\rho^* \to 2$.

In [10] it is proved that for $w_{ii}^{(1)} > 2$ and $\mathcal{I}_i(t) = 0$ the equation

$$x_i(t) = f\left(w_{ii}^{(1)} x_i(t-1)\right)$$

has two asymptotically stable equilibrium points x^+ and x^-. Because of symmetry of sigmoid function $x^+ = -x^-$. Keeping fixed x^+, the corresponding weight value is

$$w_{ii}^{(1)} = \frac{1}{x^+} f^{-1}(x^+).$$

Hence, if $w_{ii}^{(1)} > w_\rho$ we have

$$\frac{1}{x^+} f^{-1}(x^+) > \frac{1}{\rho} f^{-1}(\rho)$$

and therefore $x^+ > \rho$. By symmetry we also have $x^- < -\rho$.

Now let us consider the case of non-zero generalized input. Equation (9) defines a line whose slope depends on $w_{ii}^{(1)}$ and whose intercept depends on $\mathcal{I}_i$. In [10] it is proven that the latching condition holds for $\rho \to 0$, provided that this line has three intersections with the sigmoid $f(a_i)$. Then, our concern here is just to show that the absolute minimum x^* of neuron's output during latching satisfies the boolean quantization requirements. This minimum occurs when the input line is tangent to the sigmoid function:

$$x^* = \sqrt{\frac{w_{ii}^{(1)} - 2}{w_{ii}^{(1)}}}.$$

Hence, if $w_{ii}^{(1)} > w_\rho^*$ we have $x^* > \rho$. The absolute maximum of generalized input $\mathcal{I}_i^*$ (see eq. (14)) can be easily obtained from the tangency condition.

The third statement of this theorem does not depend on ρ and, hence, the proof is identical to that in [10]. ■

Proof of Theorem 2

We begin by observing that the relationship

$$\Gamma(y; \rho_0^-, \rho_0^+) = \bigwedge_{i=1}^{n} \Gamma(x_i; 0, \rho) \tag{84}$$

is implied by the following two conditions:

$$\begin{cases} y > \rho_0^+ & \text{if} \quad x_i > \rho \ \ \forall i, 1 \le i \le n \\ y < -\rho_0^- & \text{if} \quad \exists i, 1 \le i \le n \quad : x_i < 0. \end{cases} \tag{85}$$

Therefore it suffices to prove that the hypotheses of the theorem imply (85). Since all the weights $w_i, i = 1 \ldots n$ are positive, if $x_i > \rho, i = 1 \ldots n$ we have

$$f^{-1}(y) > w_0 + \sum_{i=1}^{n} w_i \rho \ge f^{-1}(\rho_0^+) \tag{86}$$

where the last inequality comes from the first of (20). Similarly, if $\exists i : x_i < 0$ then

$$f^{-1}(y) < w_0 + \sum_{j=1 j \ne i}^{n} w_i \le -f^{-1}(\rho_0^-) \tag{87}$$

where once again the last inequality comes from (20). Finally, it can be easily verified by substitution that

$$\begin{aligned} w_0 + \sum_{i=1}^{n} w_i \rho &= f^{-1}(\rho_0^+) \\ w_0 + \sum_{j=1, j \ne i}^{n} w_i &= -f^{-1}(\rho_0^-) \end{aligned} \tag{88}$$

when the weights are set to the values proposed in eqs. (21). ■

Proof of Theorem 3

Step 1 of Algorithm 1 does not change the language of the automaton. Hence we only focus on the effects of step 2. We will prove by induction on t that there exist two computations of the two automata such that

$$q(t) = q'(2t - 1). \tag{89}$$

The automata start their computation in the same state $q(1) = q'(1) = R$. This is the base step of induction. Now suppose $q(t) = q'(2t-1) = q_i$. Let $q(t+1) = q_j \in \Phi_n(q(t), \sigma(t))$; By eq. (40) we have $q_{ij} \in \Phi'_n(q_i, \sigma(2t-1))$. By hypothesis, for symbols in w we have $\sigma(2t) = \sigma(2t-1)$. Then by eq. (40) $q_j \in \Phi'_n(q_{ij}, \sigma(2t))$, that is there exists a computation s.t. $q'(2t+1) = q_j = q(t+1)$. Consequently, eq. (89) holds for each t; in particular it holds for $t = T$ and then there exist two computations s.t. $q(T) = q'(2T-1)$. On the other hand there is a computation of $\mathcal{A}'_n$ s.t. $q'(2T) \in \Phi'_n(q'(2T-1), \sigma(T-1))$ and then $q(T) = q'(2T)$. ■

Lemmas for Theorem 5

Lemma 1 *Consider a neuron i having $w_{ii}^{(1)} \ne 0$, latched in the low or high state at $t = t_0$. Suppose the hypotheses assumed in point 3 of Theorem 1 hold, so that a switching occurs. In particular, assume*

$$\begin{aligned} &w_{ii}^{(1)} = \frac{2\alpha}{1-\rho^2} \\ &\alpha > 1. \end{aligned} \tag{90}$$

Let t_1 be the time step when the sign of the x_i changes. Then, there exists α in (90) such that $\forall \rho \in (0,1)$ the duration of the transient required for state switching is less than or equal to two steps, i.e.:

$$\Gamma(x_i(t_1+2);\rho,\rho) = -\Gamma(x_i(t_0);\rho,\rho). \tag{91}$$

Proof

We assume a low to high switching, i.e. $x_i(t_1) = \epsilon > 0$ (the high to low case can be similarly treated). Consider the output of neuron i at time $t_1 + 2$:

$$x_i(t_1+2) = f\left(w_{ii}^{(1)} f(\mathcal{I}_i(t_1+1) + w_{ii}^{(1)}\epsilon) + \mathcal{I}_i(t_1+2)\right). \tag{92}$$

Since $w_{ii}^{(1)} > 0$, $\mathcal{I}_i(t_1+1) > \mathcal{I}_i^*$ and $\mathcal{I}_i(t_1+2) > \mathcal{I}_i^*$, we have:

$$x_i(t_1+2) > f\left(w_{ii}^{(1)} f(\mathcal{I}_i^*) + \mathcal{I}_i^*\right). \tag{93}$$

Now consider the function

$$J(\rho) \doteq f\left(w_{ii}^{(1)} f(\mathcal{I}_i^*) + \mathcal{I}_i^*\right) - \rho. \tag{94}$$

Suffice it to prove that $J(\rho) > 0$ for $\rho \in (0,1)$. On the other hand, given the dependence of $\mathcal{I}_i^*$ on $w_{ii}^{(1)}$ — cf. eq. (14) —, and since $w_{ii}^{(1)}$ depends linearly on α, it is easy to see that for each ρ, choosing α big enough we can obtain $J(\rho) > 0$. ■

Lemma 2 *Consider the generalized input of a generic state encoding neuron i*

$$\mathcal{I}_i(t) = w_{i,0}^{(0)} + w_{i,L_i}^{(1)} x_{L_i}(t-1) + w_{i,H_i}^{(1)} x_{H_i}(t-1). \tag{95}$$

If the linear constraints (56) are satisfied, then the following inequalities hold concerning the generalized input:

$$\begin{array}{lll}
\mathcal{I}_i(t) > \mathcal{I}_i^* & \textit{if} & \left\{\begin{array}{ll} x_{H_i}(t) > \rho_3^+ & \textit{and} \\ x_{L_i}(t) < -\rho_3^- & \end{array}\right. \\
\mathcal{I}_i(t) < -\mathcal{I}_i^* & \textit{if} & \left\{\begin{array}{ll} x_{L_i}(t) > \rho_3^+ & \textit{and} \\ x_{H_i}(t) < -\rho_3^- & \end{array}\right. \\
|\mathcal{I}_i(t)| < \mathcal{I}_i^* & \textit{if} & \left\{\begin{array}{ll} x_{L_i}(t) < -\rho_3^- & \textit{and} \\ x_{H_i}(t) < -\rho_3^-. & \end{array}\right.
\end{array} \tag{96}$$

Proof

The proof of this lemma trivially follows from linear constraints (56). ■

Lemma 3 *For each input $\mathbf{u}(t) \in [-1,1]^m$ of the RNN created by algorithm 2, at most one of the neurons $V_l, l = 1\ldots m$ has positive output.*

Proof

For each input vector $\mathbf{u}(t)$ consider the sign vector

$$\mathbf{s}(t) \in \{-1,+1\}^m : \quad s_i(t) = \mathrm{sign}(u_i(t))$$

and for each neuron V_l consider the sign vector

$$\mathbf{s}_l \in \{-1,+1\}^m : \quad s_{il} = \begin{cases} +1 & \text{if } i = l \\ -1 & \text{if } i \neq l \end{cases}$$

From Definition 5 and from step 2 of Algorithm 2 it follows that

$$\mathbf{s}(t) \neq \mathbf{s}_l \Longrightarrow x_{V_l}(t) < -\rho_2^-$$

and consequently at most one of the m neurons V_l can have positive output at time t. ■

Proof of Theorem 5

Suppose $\mathcal{Q}_Q(\mathbf{x}(\bar{t})) = q_k$, $\mathcal{Q}_\Sigma(\mathbf{u}(\bar{t})) = \sigma_l$, and $\Phi_d(q_k, \sigma_l) = q_i$. According to the generation algorithm, neurons H_j and L_j, $j = 1 \ldots n$ implements the boolean-like logic functions which compute the variables H_j^b and L_j^b defined in equations (50). Then, from Theorem 4 and from the theory on boolean-like logic functions (see subsection C in section IV) it follows for $t = \bar{t}$:

$$\begin{cases} x_{H_i}(\bar{t}) \geq \rho_3^+ \\ x_{L_i}(\bar{t}) \leq -\rho_3^- \end{cases} \tag{97}$$

and

$$\begin{cases} x_{H_j}(\bar{t}) < 0 \\ x_{L_j}(\bar{t}) < 0 \end{cases} \qquad \forall j \neq i \tag{98}$$

and therefore, by Lemma 2:

$$\begin{array}{l} \mathcal{I}_i(\bar{t}+1) > \mathcal{I}_i^* \quad \text{and} \\ |\mathcal{I}_j(\bar{t}+1)| < \mathcal{I}_j^* \quad \forall j \neq i. \end{array} \tag{99}$$

Then, from Theorem 1 it follows that:

- switching transient begins for neuron i (from low to high);
- no switching occurs for each neuron $j \neq i$.

We now prove that during the next time steps, until $x_i > 0$, inequalities (97,98) are still valid. Neurons H_j, L_j in (98) compute a boolean-like OR function. Then, by definition 6, they can only change their sign if at least one of their inputs becomes greater than $-\rho_2$. On the other hand, such inputs come from neurons which compute a boolean-like AND function. Then the above mentioned threshold ρ_2 is exceeded only if a change of sign occurs in a state encoding neuron — external input cannot change sign by hypothesis. The same considerations apply to the other two neurons H_i and L_i.

Denote with t_1 the time when neuron's i output crosses zero. Before t_1 each neuron $j, j \neq i$ latches its ρ-quantized information, whereas neuron i is moving towards the high value. After

t_1 we cannot anymore give guarantees about inequalities (97). Let us consider the possibility that neuron i loses its excitation signal before it completes the low to high switching. We show now that this cannot happen. In fact, the effect of the change of sign of $x_i(t_1)$ cannot affect the generalized input of x_i itself before 2 time steps. This is a direct consequence of the time delays in the non-local path looping on x_i. Then, applying Lemma 1 we conclude that: for $t = t_1 + 2$ we have:

$$x_i(t_1 + 2) > \rho. \tag{100}$$

With considerations similar to that used at the beginning of this proof, we obtain:

$$\begin{cases} x_{H_k}(t_1 + 2) \leq \rho_3^+ \\ x_{L_i}(t_1 + 2) \geq -\rho_3^- \end{cases} \tag{101}$$

and

$$\begin{cases} x_{H_j}(t_1 + 2) < 0 \\ x_{L_j}(t_1 + 2) < 0 \end{cases} \qquad \forall j \neq k \tag{102}$$

and therefore, by Lemma 2:

$$\begin{array}{ll} \mathcal{I}_k(t_1 + 3) < -\mathcal{I}_k^* & \text{and} \\ |\mathcal{I}_j(t_1 + 3)| < \mathcal{I}_j^* & \forall j \neq k. \end{array} \tag{103}$$

Then, from Theorem 1 it follows that:

- switching transient begins for neuron k (from high to low);
- no switching occurs for each neuron $j \neq k$.

By using the same arguments concerning the transition from low to high of neuron i, it can be proved that inequalities (101,102) are still valid until $x_k < 0$. Denoting with t_2 the time when neuron's k output crosses zero. Also denote $\Delta t = t_2 + 2 - \bar{t}$. Then the proof of the theorem follows applying again Lemma 1. ■

Lemmas for Theorem 6

Lemma 4 *Suppose $x_i(t) > 0$ and $x_k(t) < 0$ for $k \neq i$. Then, regardless of the input vector $\mathbf{u}(t)$, we have $x_i(t + 1) > 0$.*

Proof

Let $P_{j,l}$ be a generic neuron connected to L_i. This neuron compute the boolean-like AND function of $x_j(t)$ and $x_{V_l}(t)$, where $j \neq i$ by construction. Since for $j \neq i$ $x_j(t) < 0$, we have $x_{P_{j,l}} < -\rho_2^-$. Then it follows that $x_{L_1}(t) < 0$ and then, by Lemma 2, $\mathcal{I}_i(t) > -\mathcal{I}_i^*$. Finally, by Theorem 1, it follows that $x_i(t + 1) > 0$ ■

Lemma 5 *Suppose $x_{V_l}(t) > 0$, $x_i(t) > 0$, and $x_k(t) < 0$ for $k \neq i$. Suppose also $\Phi_d'(q_i, \sigma_l) = q_{ir}$. Then $x_j(t + 1) < 0$ for each $j \neq r, j \neq i$.*

Proof

Consider the neurons $R_{jk,p}$ for $j \neq i$. These neurons compute the boolean-like AND functions defined in eq. (59). By Lemma 3, only one neuron V_l has positive output, and consequently $x_{R_{jk,p}} < 0$ for each $p \neq l$. Moreover, since $x_k(t) < 0$ for $k \neq i$, we have $x_{R_{jk,p}} < 0$ for $k \neq i$. Now fix i and l and focus on neurons $R_{ji,l}$; if $\Phi'_d(q_i, \sigma_l) = q_{ir}$, then only $R_{ri,l}$ was allocated by Algorithm 2. This means that $x_{H_j}(t) < 0$ for each $j \neq r$. Then, by Lemma 2, $\mathcal{I}_j(t) < \mathcal{I}^*_j$ and by Theorem 1 $x_j(t+1) < 0$ for each $j \neq r$, $j \neq i$. ∎

Lemma 6 *Suppose $x_i(t) > 0$, $x_j(t) > 0$, and $x_k(t) < 0$ for $k \neq i$. Then $x_i(t+1) > 0$ or $x_j(t+1) > 0$*

Proof

According to Lemma 3, there is only one neuron V_l such that $x_{V_l}(t) > 0$. Moreover $x_k(t) < 0$ for $k \neq i$ and $k \neq j$. Then, $x_{P_{k,p}}(t) < 0$ for $k \neq i, k \neq j, p \neq l$. Concerning the state transition function, we can only have $\Phi'_d(qij, \sigma_l) = q_i$ or $\Phi'_d(qij, \sigma_l) = q_j$, and then $x_{P_{i,l}}(t)$ and $x_{P_{j,l}}(t)$ cannot be both positive. Then, the same arguments used in Lemma 4 can be used to conclude that $x_i(t+1) > 0$ or $x_j(t+1) > 0$. ∎

Lemma 7 *Suppose $x_i(t) > 0$, $x_j(t) > 0$, and $x_k(t) < 0$ for $k \neq i$. Then $x_k(t+1) < 0$ for $k \neq i, k \neq j$.*

Proof

Consider all the neurons $R_{rk,p}$. These neurons compute the boolean-like AND functions defined in eq. (59). Since $n-2$ state neurons have positive output, none of the neurons $R_{rk,p}$ can have positive output. Then, for each $r = 1 \ldots n$ we have $x_{H_r}(t) < 0$ and by Lemma 2 $\mathcal{I}_r(t) < \mathcal{I}^*_r$. Finally, by Theorem 1 $x_r(t+1) < 0$ for each $r \neq i$, $r \neq j$. ∎

Proof of Theorem 6

We prove the theorem by induction on t. The initial state vector $\mathbf{x}(1)$ has only one positive component, corresponding to the initial state R of the automaton, and then $\mathbf{s}(1) = \mathcal{C}'(R)$; this is the base step of induction. Suppose the sign structure of $x(t)$ matches a defined state binary coding, that is there exists a state $q \in Q'$ such that $\mathbf{s}(t) = \mathcal{C}'(q)$. Now consider a generic time t and suppose there exists a state $q \in \mathcal{Q}'$ such that $\mathbf{s}(t) = \mathcal{C}'_Q(q)$. Denote with $|\mathbf{s}(t)|$ the number of positive elements of $\mathbf{s}(t)$. Because of the coding, we can only have $|\mathbf{s}(t)| = 1$ or $|\mathbf{s}(t)| = 2$. If $|\mathbf{s}(t)| = 1$ then Lemma 4 and Lemma 5 guarantee that there exists a state $p \in Q'$ such that $\mathbf{s}(t+1) = \mathcal{C}'(p)$. The same is true if $|\mathbf{s}(t)| = 2$, as can be deduced by using Lemma 6 and Lemma 7. ∎

Acknowledgements

We would like to thank Yoshua Bengio for his useful comments on this paper. This research was supported by MURST 40%.

References

[1] Y.S. Abu-Mostafa, "Learning from Hints in Neural Networks," *Journal of Complexity* vol. 6, pp. 192-198 (1990)

[2] K.A. Al-Mashouq, I.S. Reed, "Including Hints in Training Neural Nets," *Neural Computation*, vol. 3, no. 4, p. 418, (1991)

[3] Y. Bengio, M. Gori, and R. De Mori, "Learning the dynamic nature of speech with backpropagation for sequences", Pattern Recognition Letters 13 (1992) 375-385

[4] A. Blum and R. Rivest, "Training in 3-node neural net is NP-complete", in: D.S. Touresky Ed. *Advance in Neural Information Processing Systems* I, San Mateo, CA: Morgan Kaufman, 1989, pp. 494–501.

[5] A. Cleeremans, D. Servan-Schreiber, and J.L. McClelland, "Finite State Automata and Simple Recurrent Networks," *Neural Computation*, vol. 1, no. 3, pp. 372–381, 1989.

[6] J.L. Elman, "Finding structure in time," *Cognitive Sciences*, vol. 14, pp. 179–211, 1990.

[7] F. Fogelman-Soulie, Y. Robert, and M Tchuente Eds, "Automata networks in computers science," Manchester University Press, 1987

[8] P. Frasconi, M. Gori, and G. Soda, "Local Feedback Multi-Layered Networks," *Neural Computation* vol. 4, no. 1, January 1992, pp. 120–130, 1992.

[9] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "A Unified Approach for Integrating Explicit Knowledge and Learning by Example in Recurrent Networks", Proceedings of IEEE-IJCNN91, Seattle, I 811–816, July 8–12 1991.

[10] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "Unified Integration of Explicit Rules and Learning by Example in Recurrent Networks," *IEEE Trans. on Knowledge and Data Engineering*, (to appear).

[11] P. Frasconi, M. Gori, and A. Tesi, "Successes and Failures of Backpropagation: a Theoretical Investigation," *Progress in Neural Networks*, Vol. 5, Ablex Publishing, O. Omidvar Ed. (to appear).

[12] S. Geman, E. Bienenstock, and R. Dourstat, "Neural Networks and the Bias/Variance Dilemma", Neural Computation, Vo. 4, No. 1, January 1992, pp. 1-58

[13] C.L. Giles, C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, and Y.C. Lee, "Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks," *Neural Computation*, Vol. 4, No. 3, May 1992 pp. 393–405.

[14] C.L. Giles and C.W. Omlin., "Inserting Rules into Recurrent Neural Networks", Proc. of IEEE Workshop on Signal Processing and Neural Network (to appear)

[15] E. Goles and S. Martinez, "Neural and Automata Networks," Kluwer Academic Publishers, Dordrecht, Boston, London 1990

[16] M. Gori and A. Tesi, "On the Problem of Local Minima in BackPropagation", *IEEE Trans. Pattern Anal. and Machine Intell.*, vol. 14, no. 1, pp. 76–86, 1991.

[17] J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Mass., 1979.

[18] Y. le Cun, "Generalization and Network Design Strategies", in *Proceedings of Connectionism in Perspective*, Elsevier Publishers P. V., North Holland, 1989.

[19] Y. le Cun, "Learning Processes in Asymmetric Threshold Networks," in F. Fogelman-Soulie, E. Bienenstock, and G. Weisbuch Eds, Disordered Systems and Biological Organization, Springer Verlag, pp. 233-240, Les Houches, France, 1986.

[20] D.G. Luemberger, "Introduction to Linear and Nonlinear Programming", Addison-Wesley Publishing Corporation, 1965

[21] W.S. McCulloch, and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bullettin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.

[22] M.L. Minsky, and S.A. Papert, *Perceptrons - Expanded Edition*, MIT Press, 1988.

[23] D.B. Parker, "Learning Logic", Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, April 1985.

[24] C.W. Omlin and C.L. Giles, "Training Second-Order Recurrent Neural Networks using Hints," in: D. Sleeman and P. Edwards eds. *Proc. of the Ninth International Conference on Machine Learning*, 1992.

[25] B.A. Pearlmutter, "Learning State Space Trajectories in Recurrent Neural Networks," *Neural Computation* vol. 1, no. 2, pp. 263–269, 1989.

[26] S.J. Perantonis and P.J.G. Lisboa, "Translation, Rotation, and Scale Invariant Pattern Recognition by High-Order Neural Networks and Moment Classifiers," *IEEE Trans. on Neural Networks*, vol. 3, no. 2, 1992, pp. 241-251

[27] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing. Exploration in the microstructure of Cognition*, vol. 1: Foundations, MIT Press, 1986.

[28] G.Z. Sun, H.H. Chen, C.L. Giles, Y.C. Lee, and D. Chen, "Neural networks with external memory stack that learn context-free grammars from examples," Proc. of the 1990 Conference on Information Sciences and Systems.

[29] G.G. Towell, and J.W. Shavlik, "The Extraction of Refined Rules from Knowledge-Based Neural Networks", Machine Learning Research Group Working Paper 91-4.

[30] G.G. Towell, J.W. Shavlik, and M.O. Noordewier, "Refinement of Approximately Correct Domain Theories by Knowledge-Based Neural Networks," Proc. of the Eigth National Conference on Artificial Intelligence, Boston, MA, p. 861, 1990.

[31] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, K. Lang, "Phoneme Recognition Using Time-Delay Neural Networks," *IEEE Trans. on ASSP*, vol. 37, no. 3, March 1989.

[32] R.L. Watrous and G.M. Kuhn, "Induction of Finite-State Languages Using Second-Order Recurrent Networks," *Neural Computation* Vol. 4, No. 3, May 1992, pp.406-414.

[33] R.L. Watrous, "Speech Recognition Using Connectionist Networks", Ph.D. Thesis, University of Pennsylvania, Philadelphia, November 1988

[34] P.J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences", PhD dissertation, Committee on Applied Mathematics, Harvard University, Cambridge, MA, November 1974.

[35] R.J. Williams, and D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

[36] R.J. Williams and J. Peng, "An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Networks Trajectories," *Neural Computation*, vol. 2, no. 4, pp. 490–501, 1990.