Parallel Computing Model of Nondeterministic Finite Automata via Artificial Neural Networks

Chun-Hsien Chen[1] and Her-Kun Chang[2]
Department of Information Management
Chang Gung University
259 Wen-Hwa 1st Road, Kwei-Shan
Tao-Yuan, Taiwan, R.O.C.
e-mail : [1]cchen@mail.cgu.edu.tw, [2]hkchang@mail.cgu.edu.tw

Abstract

Artificial neural networks (ANNs), due to their inherent parallelism, offer an attractive paradigm for efficient implementations of functional modules for symbolic computations intensively involving content-based pattern matching. This paper explores how to exploit the inherent parallelism and versatile representation in ANNs to reduce the operation and implementation time overhead of nondeterministic finite automata (NFAs). NFAs are a basic model of symbolic computing in computer science, and they thus provide a typical model suitable for the exploration of parallel symbolic computing via ANNs. For every NFA, a recurrent neural network (RNN) can be systematically synthesized to concurrently track at each time step all the states reached by the possible nondeterministic moves of the NFA. Such a concurrent breadth-first tracking is facilitated by two types of parallel symbolic computations executed by the proposed RNN. One is parallel content-based pattern matching, and the other is parallel union operation of sets.

Keywords : artificial neural network, nondeterministic finite automata, parallel nondeterministic computing

1 Introduction

Artificial neural networks offer an attractive paradigm for a variety of applications in computer science and engineering, artificial intelligence, and cognitive modeling for various reasons mainly including their learning as well as generalization capabilities, potential for fault tolerance, and inherent parallelism. This paper presents one of the efforts in integrating neural network and symbolic computing by taking advantage of the inherent parallelism in ANNs. Despite the success in the application of ANNs to a broad range of tasks in pattern classification, control, function approximation, and system identification, their use in symbolic computing tasks (e.g., storage and retrieval of records in large databases and knowledge bases, language processing, etc.) is only beginning to be explored [3, 4, 7, 8, 9, 10, 11, 15, 17, 19, 20, 21, 22, 24].

This paper explores how to exploit the inherent parallelism in ANNs to reduce the operational and implementational time overhead of NFAs. For every given NFA, a recurrent neural network can be synthesized to deterministically track in linear time the nondeterministic computing of the NFA. Although the concept of nondeterministicism embedded in NFAs provides an elegantly simple and intuitive description for sequence processing, it results in much computation and implementation overhead in current computer systems. Thus the concept of nondeterministicism in NFAs, which plays a central role in both the theory of languages and the theory of computation [12], provides a typical model suitable for the exploration of parallel symbolic computing via neural networks. The reduced operation time of NFAs realized by the proposed RNN is due to the parallel operations of the neural assemblies in the RNN. The proposed RNN is assembled from two kinds of neural assemblies. One computes a logic `AND`, and the other computes a logic `OR`. The RNN acts like a cost-effective SIMD computer system dedicated to two types of relaid parallel symbolic computations. One is parallel content-based pattern matching, and the other is parallel union operation of sets.

It is well known that NFAs and derterministic finite automata (DFAs) are equivalent, and every NFA can be converted into its equivalent DFA [12]. NFAs seem to be of no practical interest in direct application implementations since they are embedded with nondeterministicism and don't correspond naturally to deterministic algorithms. But, NFAs have a variety of practical applications in computer science, linguistics, systems modeling and control, artificial intelligence, and structural pattern recognition since NFAs are simpler and more intuitive to define than their equivalent DFAs due to the powerful concept of nondeterministicism embedded in NFAs, especially for pattern matching [25]. NFAs are rarely implemented directly in conventional computer systems because the nondeterministicism in NFAs causes poor performance and implementation overhead. Usually, they are converted into their equivalent DFAs for implementation. Therefore, for the purpose of syntax analysis on regular languages, an NFA could be constructed for a given language first, and then its equivalent DFA is implemented to recognize the language. The direct construction of an NFA is as simple as that of a DFA using the proposed RNN (see Section 3). In that way, the power of nondeterministicism in NFAs is retained, and there is no need to convert an NFA into its equivalent DFA for implementation. Since every DFA is an NFA, the proposed RNN can be used as a general neural architecture for realizing finite automata including DFAs and NFAs.

The rest of the paper is organized as follows. Section 2 reviews some of the key concepts and develops definitions that will be used in the rest of the paper. Section 3 develops the theoretical foundation for direct implementation of NFAs using RNNs. Section 4 concludes with a summary and a brief discussion.

2 Review and Definitions

2.1 Perceptrons

A 1-layer Perceptron has n input neurons, m output neurons and one layer of connection weights. The output y_i of output neuron i, $1 \leq i \leq m$, is given by $y_i = f_h(\sum_{j=1}^{n} w_{ij}x_j - \theta_i)$. w_{ij} denotes the weight on the connection from input neuron j to output neuron i, θ_i is the threshold of output neuron i, x_j is the value at input neuron j, and f_h is *binary hardlimiter* function, where

$$f_h(x) = \begin{cases} 1 & \text{if x} > 0 \\ 0 & \text{otherwise} \end{cases}$$

It is well known that such a 1-layer Perceptron can only implement linearly separable functions mapping from $\mathbf{R}^n$ to $\{0,1\}^m$ [16]. The connection weight vector $w_i =< w_{i1}, ..., w_{in} >^T$ can be viewed as defining a linear hyperplane H_i which linearly separates all the n-dimensional vectors into two sets, where $[\cdot]^T$ denotes the *transpose* of a vector or a matrix.

A 2-layer Perceptron has one layer of k hidden neurons (and hence two layers of connection weights with each hidden neuron being connected to every input neuron and every output neuron). Note that there is no connection from input neurons to output neurons in a 2-layer Perceptron. In this paper, every hidden neuron and output neuron in the 2-layer Perceptron use binary hardlimiter function f_h as activation function and produce binary outputs; its weights are restricted to values from $\{-1, 0, 1\}$; and it uses integer thresholds. It is known that such a 2-layer Perceptron can realize arbitrary binary mappings [1, 2].

2.2 Finite Automata

Deterministic Finite Automata

A *deterministic finite automaton* M_{DFA} is a 5-tuple $(Q, \Gamma, \delta, q_0, F)$ [12], where Q is a finite non-empty set of *states*, Γ is a finite non-empty *input alphabet*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final* or *accepting states*, and δ is the *transition function* mapping from $Q \times \Gamma$ to Q. A finite automaton is deterministic if there is at most one transition that is applicable for each combination of state and input symbol. An input string is *accepted* by M_{DFA} if the computation on the input string by M_{DFA} terminates in an accepting state; otherwise it is rejected. The set of strings accepted by M_{DFA} in Γ^* is denoted as $L(M_{DFA})$, called *the language accepted by* M_{DFA}.

Nondeterministic Finite Automata

A *nondeterministic finite automaton* M_{NFA} is a 5-tuple $(Q, \Gamma, \delta', q_0, F)$ [12], where Q, Γ, q_0, and F have same meaning as for a DFA, but δ' is a mapping from $Q \times \Gamma$ to 2^Q. Note that 2^Q is the power set of Q and $\delta'(q, a)$ is the set of all states p such that there is a transition, denoted as (q, a, p), from q to p on an input symbol a. Also note that there could be more than one transition which is applicable for each combination of state and input symbol in an NFA, and $|\delta'(q, a)|$ is bounded by $|Q|$, where $|A|$ denotes the cardinality of set A. An input string is *accepted* by M_{NFA} if there is a computation on the input string by M_{NFA} which processes the entire input string and halts in an accepting state; otherwise it is rejected. The set of strings accepted by M_{NFA} in Γ^* is denoted as $L(M_{NFA})$, called *the language accepted by* M_{NFA}.

Simplicity and Intuitiveness of NFA

It is known that NFA and DFA are equivalent [12]. Two automata are said *equivalent* if they accept the same language. Any language accepted by an NFA can also be accepted by a DFA, and every NFA can be converted into an equivalent DFA [12]. However, an NFA is usually simpler and more intuitive to define than its equivalent DFA due to the powerful concept of nondeterministicism embedded in NFAs, especially for pattern matching [25]. Figures 1.a and 1.b respectively show the state diagrams of an NFA and its equivalent DFA. Both of them accept any input string containing a sub-string `abaa` [25]. These two automata are equivalent, but apparently the representation of the NFA is simpler and more intuitive than that of the DFA in this case (in terms of the representation of state diagram). The state diagram of an NFA or a DFA is a labeled directed graph in which the nodes denote the states of the NFA or DFA, and the arcs are obtained from the transition function. An arc from node q_i to q_j is labeled a if $\delta(q_i, a) = q_j$ for a DFA or $q_j \in \delta(q_i, a)$ for an NFA, and the transition (q_i, a, q_j) is a *fan-in transition* of state q_j on input symbol a. From the examples, it is noted that an NFA search for a solution along all reachable paths in parallel conceptually in the state space. Note also that, if an NFA has Q states, then the number of possible states of its equivalent DFA could be as large as $2^{|Q|}$ and the number of possible transitions in the DFA could also be the same order. For example, if an NFA has 20 states, then its equivalent DFA has $2^{20} \approx 1 \times 10^6$ states in the worst case. Therefore, it may take much overhead to implement NFAs deterministically in current computer systems.

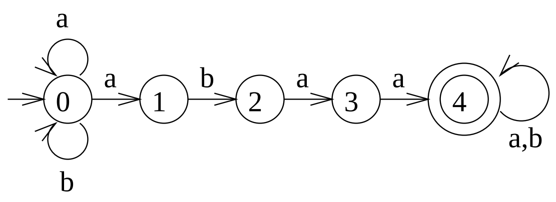

Figure 1.a. The state diagram of an NFA that accepts any input string having a sub-string `abaa`.

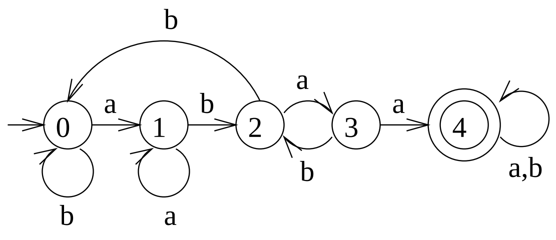

Figure 1.b. The state diagram of a DFA that accepts any input string having a sub-string `abaa`.

2.3 Parallel Computing Model of NFAs via RNNs

The deterministic and linear-time operation of a given NFA directly constructed on the proposed RNN architecture can be modeled conventionally by its equivalent DFA which is derived according to *Subset Construction* algorithm [25] whose main idea is to concurrently track all the possible states that can be reached at each move of an NFA. Such a tracking in the operations of an NFA computes all the possible paths at each move and induces much overhead in single-CPU computer systems, whereas the proposed RNN efficiently computes in parallel the set of reachable states at each move by exploiting the inherent parallelism in ANNs. In Subset Construction algorithm, for a given NFA $M_{NFA} = (Q, \Gamma, \delta', q_0, F)$, a DFA $M''_{DFA} = (2^Q, \Gamma, \delta'', Q_0, F'')$ is defined from M_{NFA} such that $L(M_{NFA}) = L(M''_{DFA})$, where $Q_0 = \{q_0\}$, $F'' = \{K \mid K \subseteq Q \ \& \ K \cap F \neq \emptyset\}$, and $\delta'' : 2^Q \times \Gamma \to 2^Q$ is defined by

$$Q_j = \delta''(Q_i, a), \ \texttt{if} \ Q_j = \cup_{q \in Q_i} \delta'(q, a) \quad \texttt{for all} \ Q_i \subseteq Q \ \& \ a \in \Gamma \quad (1)$$

One main problem with Subset Construction algorithm, which views Q_i's as individual states in implementation, is the exponential increase in the number of states ($O(2^{|Q|})$) and the number of possible transitions defined in transition function δ'' ($O(2^{2|Q|} \times |\Gamma|)$). This situation can often be somewhat alleviated by *Iterative Subset Construction* algorithm [25] which only includes the states that can be reached from initial state q_0. Let $M^*_{DFA} = (Q^*, \Gamma, \delta^*, Q_0, F^*)$ be defined from M_{NFA} according to Iterative Subset Construction such that $L(M_{NFA}) = L(M^*_{DFA})$. Since Iterative Subset Construction eliminates the states which can not be reached from initial state q_0, M^*_{DFA} is smaller than M''_{DFA} in terms of the number of defined states and transitions.

The major difficulty with both Subset Construction algorithms in implementation is the bookkeeping of δ'', F'', Q^*, δ^*, and F^*, which are derived from M_{NFA}. The direct realization of a given NFA by the proposed RNN is free of the problem since the transition function module of the RNN captures the regularity of δ'' in expression (1) via δ' without actually knowing the mappings in δ'' in advance (see Section 3). Such simplification is partly facilitated by representationally viewing Q_i's as individual sets of states denoted by neural localist representation. The transition function module of the proposed RNN realizes not only δ'' but also δ^*. Since the proposed RNN always starts from initial state q_0 for any input string, the states which can not be reached from q_0 will not appear in the transition function module of the RNN during input processing, i.e., only the states in Q^* will appear in the transition function module during input processing. Hereafter, only M^*_{DFA} instead of $M^{''}_{DFA}$ is discussed.

Let 0,1,...,t,t+1,... denote a succession of points along the discrete time line. Then, for an NFA, let us call $Q_{act}(0) = Q_0$ the *initial set of active states*, $Q_{act}(t)$ the *current set of active states* which corresponds to the set of reachable states from q_0 after t (current time) moves by M_{NFA} (M^*_{DFA}), and $Q_{act}(t+1)$ the *next set of active states* which corresponds to the set of reachable states from q_0 after $t+1$ moves by M_{NFA} (M^*_{DFA}). $Q_{act}(t)$ and $Q_{act}(t+1)$ are derived recursively from $Q_{act}(0)$ by expression $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q,a)$ along the processing of the input string, where a is input symbol at time t. $Q_{act}(t)$ is bounded in a way that $Q_{act}(t) \subseteq Q$ for $t \geq 0$, i.e., all the sets of reachable states from initial state along the processing of the input string are bounded by a same set of states. Therefore, the number of reachable states at each step does not proliferate indefinitely or exponentially along the processing of the input string, and thus the nondeterministicism shown along the processing of the input string is *globally bounded.* The proposed RNN directly realizes a given NFA M_{NFA} without a need to convert the given NFA into its equivalent DFA M^*_{DFA}, and it concurrently tracks all the possible paths along the processing of the input string in the NFA by simulating the deterministic move of the M^*_{DFA}. According to Iterative Subset Construction and expression (1), the transition function δ^* and the moving of M^*_{DFA} can be characterized by

$$\forall\, a \in \Gamma\, \forall\, t \geq 0\ [Q_{act}(t+1) = \delta^*(Q_{act}(t), a)],$$
$$\textit{where } Q_{act}(0) = \{q_0\} \textit{ and}$$
$$Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q,a) \quad (2)$$

For an input string, the recursive evaluation of $Q_{act}(t+1)$ along the moves of M^*_{DFA} (M_{NFA}) involves two kinds of repetitive symbolic computations. One computes the sets of reachable states from every state in $Q_{act}(t)$, and the other computes the union of the sets of the reachable states. The former is computed by the first layer of the transition function module of the proposed RNN by way of parallel content-based pattern matching and the latter by the second layer by way of logic `OR` operations. In applications of realizing an NFA by the proposed RNN, a special symbol $\$ \notin \Gamma$ might need to be appended at the end of the input string to acknowledge the end of input. When the $\$$ is encountered, the RNN terminates input processing and tests the acceptance of the input string.

3 Realization of NFAs by RNNs

This section develops the theoretical foundation and property for the neural assemblies used to assemble the proposed RNN for an NFA (RNN NFA). First, the representation in the proposed RNN NFA is described. Then, a systematical method is proposed to assemble the proposed RNN NFA using the developed neural assemblies. The following theorem and its proof facilitate the development of theoretical foundation for two kinds of neural assemblies used to assemble the proposed RNN NFA.

Theorem 1 : Any single binary vertex is linearly separable from *all* other binary vertices of same dimension.

This theorem has been proved by [1, 13, 23] by finding a hyperplane linearly separating a given binary vertex from all other vertices in a binary hypercube. [1] proves Theorem 1 by examining the spatial distribution and linear separability of binary vertices from geometrical perspective to locate a set of hyperplanes which linearly separate a binary vertex from all other vertices in a binary hypercube. The proof allows locating a separating hyperplane that can be efficiently implemented in a 1-layer Perceptron with one output neuron. The computations in the Perceptron only involve integer processing. [1] proposes that for a given n-dimensional binary vertex v, all n-dimensional binary vertices can be partitioned into $n+1$ parallel layers in geometrical space according to their Hamming distance to the given binary vertex v.

Let v be a binary vector of dimension n, i.e., $v = < v_1, ..., v_n >^T$ where $v_i \in \{0,1\}$ for $1 \leq i \leq n$. Then v can be viewed as a binary pattern or a binary vertex of an n-dimensional hypercube. Hereafter, $< v_1, ..., v_n >$ is also used to denote an n-dimensional vector (vertex). Now consider a binary vector v^* of dimension m, where $m \geq n$. For the purpose of constructing the proposed RNN NFA, only the values of certain n components of vector v^* are of interest (see Sections 3.2 and 3.3); i.e., for two given binary vectors v and v^* of dimension n and m respectively, only whether $v^*_{j_1} = v_1, ..., v^*_{j_i} = v_i, ..., v^*_{j_n} = v_n$ are concerned, where $1 \leq n \leq m$ and $1 \leq j_1 < j_2 < \cdots < j_n \leq m$. Let us call $J^n_k = \{j_1, j_2, ..., j_n\}$ the *interest set* J^n_k

and $v^*(J_k^n) =< v_{j_1}^*, v_{j_2}^*, ..., v_{j_n}^* >$ the ordered J_k^n-set partial vector of the binary vector v^*, where k is a natural number. Note that at most 2^m interest sets can be defined concurrently for a given problem in an m-dimensional binary space.

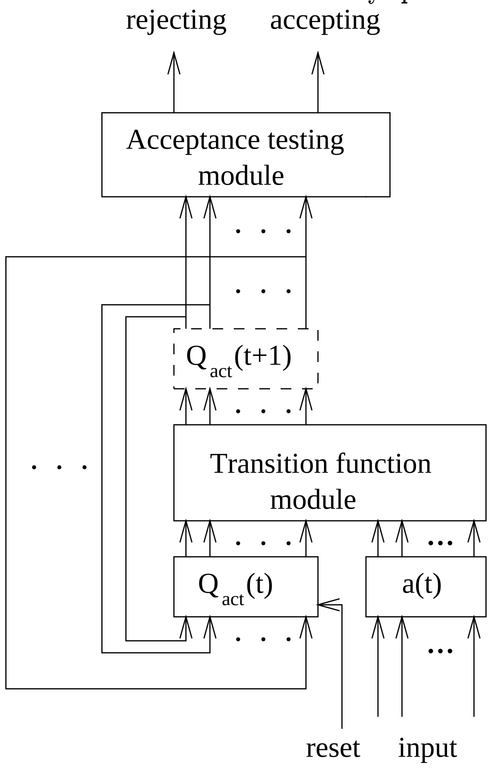

Figure 2. The block architecture of the proposed recurrent neural network for concurrently tracking all the nondeterministic computations of a given NFA. The dotted box labeled with $Q_{act}(t+1)$ exists only logically but not physically.

Figure 2 shows the partially recurrent neural network architecture for concurrently tracking all the nondeterministic computations of a given NFA. The entire architecture essentially consists of one *transition function module*, one *acceptance testing module*, one *end-of-input testing module* which is not shown in the figure, three buffers, and recurrent links from the output neurons of the transition function module to the buffer storing $Q_{act}(t)$ (which could be part of the input neuron layer of the transition function module. It depends on applications implemented). The transition function module is a 2-layer Perceptron, and the acceptance testing module as well as the end-of-input testing module are 1-layer Perceptrons. One buffer stores current set of active states $Q_{act}(t)$, another buffer (which could be the other part of the input neural layer of the transition function module. It depends on applications implemented) stores current input symbol $a(t)$, and the next set of active states $Q_{act}(t+1)$ is represented by the other buffer which exists only logically but not physically. The first two buffers are under centralized synchronization control which enforces discrete time 0,1,...,t,$t+1$,.... The link "`reset`" resets the RNN NFA to its initial set of active states.

3.1 Symbolic Representation in the Transition Function Module

How the transition function module of the proposed RNN NFA realizes the transition function δ^* of the DFA M_{DFA}^* plays a central role for the realization of a given NFA via RNN. This subsection follows the notations described in Section 2. The symbolic representations in the transition function module are described as follows.

- The output from every neuron and the input to every input neuron are binary values.
- The input neurons are divided into two groups. One group uses distributed representation, and the other group uses local representation. Th former group has no recurrent connection and denotes the binary-coded current input symbol. There are $\lceil \log(\mid \Gamma \mid +1) \rceil$ such input neurons, where $\lceil \cdot \rceil$ denotes the ceiling integer value of a real value. The latter group has recurrent connections and denotes the current set of active states $Q_{act}(t)$. There are $\mid Q \mid$ such input neurons, the ith neuron of which denotes whether state q_{i-1} is in $Q_{act}(t)$. If the value at the ith neuron of this group is 1, then state q_{i-1} is in $Q_{act}(t)$. Otherwise state q_{i-1} is not in $Q_{act}(t)$. In this group, the ith input neuron has a recurrent link from the ith output neuron. The input layer denotes $Q_{act}(t) \times a(t)$.
- The output layer uses local representation, and the output neurons together denote the next set of active states $Q_{act}(t+1)$. There are $\mid Q \mid$ output neurons, the ith neuron of which denotes whether state q_{i-1} is in $Q_{act}(t+1)$. If the value at the ith neuron is 1, then state q_{i-1} is in $Q_{act}(t+1)$. Otherwise state q_{i-1} is not in $Q_{act}(t+1)$. The output neurons along with their associated 2nd-layer connections operate together to compute the next set of active states according to expression $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q,a)$ (the union of the sets of reachable states entered from every state in the current set of active states on current input symbol).
- The hidden neurons along with their 1st-layer associated connections are used to recognize

(identify) the applicable transitions defined by the transition function of an NFA. The hidden layer uses local representation, and one hidden neuron is used for one uniquely defined transition. The number of activated hidden neurons at each move of the proposed RNN NFA equals $\Sigma_{q \in Q_{act}(t)} \mid \delta'(q, a(t)) \mid$. The activated hidden neurons in turn activate some of the output neurons.

- Every transition defined by the transition function δ^* (expression (2)) is represented as an ordered binary mapping pair $< Q_{act}(t) \times a(t), Q_{act}(t+1) >$ by the input and output neurons of the transition function module.
- The recurrent connections from the output neurons to some of the input neurons facilitate the continuous execution of the proposed RNN NFA.

3.2 The First Neural Layer of the Transition Function Module

This section develops the theoretical foundation for explaining how the hidden neurons along with their associated 1st-layer connections of the transition function module in the proposed RNN are used to concurrently identify multiple sub-patterns contained in an input pattern. Every hidden neuron and its associated 1st-layer connections compose a basic neural assembly which is used to identify if an input pattern contains a sub-pattern of interest. The first neural layer of the transition function module consists of a fixed amount of such neural assemblies which operate in parallel and independently.

Neural Assembly for Recognition of Partial Pattern

Let $v =< v_1, ..., v_n >$ be a binary vertex of dimension n, where $v_i \in \{0, 1\}$ for $1 \le i \le n$. Let $H_S^{n,v}$ be an n-dimensional separating hyperplane which linearly separate vertex v from all other n-dimensional vertices. Among a set of possible expressions for $H_S^{n,v}$, [1] chooses

$$H_S^{n,v} \equiv (\sum_{i=1}^{n}(2v_i - 1)x_i) - (\| v \|^2 - 1) = 0 \qquad (3)$$

where $\| \cdot \|$ denotes the length of a vector. In a binary space, $\| \cdot \|^2$ equals the number of 1's in a binary vector, and thus $\| v \|^2 = \sum_{i=1}^{n} v_i$. The separating hyperplane $H_S^{n,v}$ for identifying binary vector (binary pattern) v can be implemented in a 1-layer Perceptron with one output neuron by setting:

- $\sum_{i=1}^{n} v_i - 1$ as the threshold of the output neuron, and
- $2v_i - 1$ as the weight of the connection from the ith input neuron to the output neuron.

Note that the value of $2v_i - 1$ is either 1 or -1, that of x_i is either 1 or 0, that of $(2v_i - 1)x_i$ is either 1, 0 or -1, and that of $\sum_{i=1}^{n} v_i$ is an integer. Since only integer computations are involved in the hardware implementation of this Perceptron, the computations in the Perceptron are relatively efficient.

Let v^* be an m-dimensional binary vector and an interest set J_k^n be chosen for $1 \le j_1 < j_2 < \cdots < j_n \le m$ such that $v^*_{j_1} = v_1, ..., v^*_{j_i} = v_i, ..., v^*_{j_n} = v_n$, where $m \ge n$ and $1 \le k \le 2^m$. In the following, the expression for the separating hyperplane $H_S^{n,v}$ in an n-dimensional binary space is re-defined as a separating hyperplane H_S^{m,v^*,J_k^n} in an m-dimensional binary space to recognize the m-dimensional binary vectors that contain the ordered J_k^n-set partial vectors equaling the given n-dimensional binary vector $v(= v^*(J_k^n))$:

$$\begin{aligned} H_S^{m,v^*,J_k^n} \equiv\ & (\sum_{i \in J_k^n}^{m}(2v_i^* - 1)x_i) + (\sum_{j \notin J_k^n}^{m} 0 \cdot x_j) - \\ & (\sum_{i \in J_k^n}^{m} v_i^* - 1) = 0 \qquad (4) \end{aligned}$$

According to expression (4), the separating hyperplane H_S^{m,v^*,J_k^n} can be implemented in a 1-layer Perceptron with one output neuron by setting:

- $2v_i^* - 1$ as the weight of the connection from the ith input neuron ($i \in J_k^n$) to the output neuron,
- 0 as the weight of the connections from the non-J_k^n-set input neurons to the output neuron, and
- $\sum_{i \in J_k^n}^{m} v_i^* - 1$ $(= \sum_{k=1}^{n} v_k - 1)$ as the threshold of the output neuron.

The values from those $j \notin J_k^n$ input neurons will not affect the incoming summation value at the output neuron since the weights on the connections from those input neurons are set as 0. They altogether act as a *don't-care* filter,

3.3 The Second Neural Layer of the Transition Function Module

This section develops the theoretical foundation for explaining how the output neurons along with their associated 2nd-layer connections of the transition function module operate in parallel to compute $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q, a)$ (the union of the sets of reachable states entered from the states in the current set of active states on current input symbol). Such a concurrent union operation of sets are computed by a neural layer which is assembled from a fixed amount of neural assemblies. Each of the assemblies computes a logic `OR` operation in parallel with each other on shared inputs. Every output neuron and its associated 2nd-layer connections compose such a basic neural assembly.

Neural Assembly for Executing a Logic `AND`

Consider an n-variable Boolean expression in the following form (a conjunction clause of no negated Boolean variable):

$$v_1 \wedge\ v_2 \cdots \wedge v_n \tag{5}$$

where $\wedge$ is a logical connective `AND` and v_i is a Boolean variable for $1 \leq i \leq n$. The value of v_i is either 0 or 1, with 0 denoting *false* and 1 *true*. Let $v =< v_1, ..., v_n >$, $C^n(v) = v_1 \wedge\ v_2 \cdots \wedge v_n$, and $< 1^n >$ denote an n-dimensional binary vertex of all ones. Then, it can be derived that

$$C^n(v) = \begin{cases} 1 & \text{if } v =< 1^n > \\ 0 & \text{otherwise} \end{cases}$$

The function $C^n(v)$ is a logic `AND` function which can be realized by a 1-layer Perceptron that implements the hyperplane $H_S^{n,<1^n>}$. $H_S^{n,<1^n>}$ linearly separates binary vertex $< 1^n >$ from all other n-dimensional binary vertices according to expression (3). Let $H_{AND}^{n,<1^n>}$ be used for $H_S^{n,<1^n>}$. Then, according to expression (3), the separating hyperplane for identifying binary vertex $< 1^n >$ is

$$\begin{aligned} H_{AND}^{n,<1^n>} &\equiv H_S^{n,<1^n>} \\ &\equiv (\sum_{i=1}^{n} x_i) - (n-1) = 0 \end{aligned} \tag{6}$$

i.e., a logic `AND` function of n Boolean variables (more specifically, a conjunction clause of no negated Boolean variable) can be realized by a 1-layer Perceptron with n input neurons, one output neuron, and

- $n-1$ as the threshold of the output neuron, as well as
- 1 as the weight of the connection from every input neuron to the output neuron.

Neural Assembly for Executing a Logic `OR`

Consider an n-variable Boolean expression in the following form (a disjunction clause of no negated Boolean variable) :

$$v_1 \vee\ v_2 \cdots \vee v_n \tag{7}$$

where $\vee$ is a logical connective `OR` and v_i is a Boolean variable for $1 \leq i \leq n$. The value of v_i is either 0 or 1. [1] proposes that for a given n-dimensional binary vertex $u =< u_1, ..., u_n >$, all n-dimensional binary vertices can be partitioned into $n+1$ parallel layers according to their Hamming distance p to the given binary vertex u. Those $n+1$ layers are respectively on $n+1$ parallel n-dimensional hyperplanes $H_p^{n,u}$'s, $0 \leq p \leq n$, where

$$\begin{aligned} H_p^{n,u} &\equiv (\sum_{i=1}^{n}(2u_i - 1)x_i) - (\| u \|^2 - p) = 0 \\ &\equiv (\sum_{i=1}^{n}(2u_i - 1)x_i) - \\ &\quad (\sum_{i=1}^{n} u_i) + p = 0 \end{aligned} \tag{8}$$

and u is the only vertex of the first layer which is on $H_0^{n,u}$, where

$$H_0^{n,u} \equiv (\sum_{i=1}^{n}(2u_i - 1)x_i) - \sum_{i=1}^{n} u_i = 0 \tag{9}$$

Let $\overline{u} =< \overline{u}_1, ..., \overline{u}_n >$ be the complement vertex of binary vertex u, i.e., $u_i + \overline{u}_i = 1$ for $1 \leq i \leq n$. Then $\overline{u}$ is the only vertex of the (n+1)th layer which is on $H_n^{n,u}$, where

$$\begin{aligned} H_n^{n,u} &\equiv (\sum_{i=1}^{n}(2u_i - 1)x_i) - \\ &\quad (\sum_{i=1}^{n} u_i) + n = 0 \end{aligned} \tag{10}$$

The hyperplane $H_n^{n,u}$ can be implemented by a 1-layer Perceptron to separate binary vertex $\overline{u}$ from all other n-dimensional binary vertices. Let $u =< 1^n >$, then $\overline{u} =< 0^n >$. Let $D^n(v) = v_1 \vee\ v_2 \cdots \vee v_n$. Then it can be derived that

$$D^n(v) = \begin{cases} 0 & \text{if } v =< 0^n > \\ 1 & \text{otherwise} \end{cases}$$

The function $D^n(v)$ is a logic `OR` function which can be realized by a 1-layer Perceptron that implements the hyperplane $H_n^{n,<1^n>}$ to separate binary vertex $< 0^n >$ from all other n-dimensional binary vertices according to expression (10). Let $H_{OR}^{n,<1^n>}$ be used for $H_n^{n,<1^n>}$. Then the separating hyperplane is

$$\begin{aligned} H_{OR}^{n,<1^n>} &\equiv H_n^{n,<1^n>} \\ &\equiv \sum_{i=1}^{n} x_i - (\sum_{i=1}^{n} 1) + n = 0 \end{aligned}$$

$$\equiv \quad (\sum_{i=1}^{n} x_i) - 0 = 0 \qquad (11)$$

i.e., a logic OR function of n Boolean variables (more specifically, a disjunction clause of no negated Boolean variable) can be realized by a 1-layer Perceptron with n input neurons, one output neuron, and

- 0 as the threshold of the output neuron, as well as
- 1 as the weight of the connection from every input neuron to the output neuron.

In a space of m Boolean variables, to deal with the logic OR functions of n Boolean variables (called as n-out-of-m-variable disjunction clauses of no negated Boolean variable), where $m \geq n$, the expression for the separating hyperplane $H_{OR}^{n,u}$ needs to be extended from an n-dimensional binary space to an m-dimensional binary space to recognize the m-dimensional binary patterns that contain an ordered partial pattern not equaling the n-dimensional binary vector $\overline{u}$ for a certain interest set J_k^n. Suppose $u^* =< u_1^*, u_2^*, ..., u_m^* >$ is a binary vector of dimension m, where $m \geq n$. And assume an interest set $J_k^n = \{j_1, j_2, ..., j_n\}$ is defined, where $1 \leq j_1 < j_2 < \cdots < j_n \leq m$. Define $D^{m,J_k^n}(u^*) = u_{j_1}^* \vee u_{j_2}^* ... \vee u_{j_n}^*$. Then,

$$D^{m,J_k^n}(u^*) = \begin{cases} 0 & \text{if } u^*(J_k^n) =< 0^n > \\ 1 & \text{otherwise} \end{cases}$$

$H_{OR}^{n,<1^n>}$ is re-defined as $H_{OR}^{m,<1^m>,J_k^n}$ in an m-dimensional binary space as follows :

$$H_{OR}^{m,<1^m>,J_k^n} \equiv (\sum_{i \in J_k^n}^{m} x_i) - (\sum_{j \notin J_k^n}^{m} 0 \cdot x_j) - 0 = 0 \qquad (12)$$

i.e., the logic OR function $D^{m,J_k^n}(u^*) = u_{j_1}^* \vee u_{j_2}^* ... \vee u_{j_n}^*$ (more specifically, an n-out-of-m-variable disjunction clause of no negated Boolean variable) can be realized by a 1-layer Perceptron with m input neurons, one output neuron, and

- 0 as the threshold of the output neuron,
- 1 as the weight of the connection from the j_ith input neuron to the output neuron, where $j_i \in J_k^n$ for $1 \leq i \leq n$, and
- 0 as the weight of the connections from the non-J_k^n-set input neurons to the output neuron.

Such a neural assembly (a 1-layer Perceptron) can recognize all the $2^{m-n} \cdot (2^n - 1)$ m-dimensional binary patterns that let D^{m,J_k^n} produce 1, and such a neural assembly can check whether any of certain n neurons among m neurons is activated, where $m \geq n$. In the proposed RNN NFA, every such neural assembly is used for one particular fan-in transition of a certain state of an NFA to check whether the transition is applicable on current input symbol.

3.4 Assembling the Proposed RNN NFA

The following subsections describe how to integrate the neural assemblies developed in Sections 3.2 and 3.3 to directly construct an RNN NFA for a given NFA. In the meanwhile, it is also described for what each layer of neurons in the modules of the proposed RNN NFA represents symbolically .

Neural Representation in the Transition Function Module

Let $n_T = \Sigma_{q \in Q} \Sigma_{a \in \Gamma} \mid \delta'(q,a) \mid$, $n_I = \lceil \log(\mid \Gamma \mid +1) \rceil$, and $n_A = \mid Q \mid$ be respectively the total number of defined transitions of a given NFA, the number of input neurons used for denoting current input symbol, and the number of input neurons used for denoting the current set of active states in the transition function module of the proposed RNN NFA. Then the transition function module has $(n_A + n_I)$ input neurons, n_T hidden neurons, and n_A output neurons. The transition function module is constructed directly from the symbolic function δ' of the given NFA M_{NFA} as follows.

The hidden neurons along with their associated 1st-layer connections are used to identify the transitions applicable by the current set of active states on current input symbol. One hidden neuron is used for one uniquely defined transition in the given NFA. The number of hidden neurons activated by current set of active states $Q_{act}(t)$ on current input symbol a equals $\Sigma_{q \in Q_{act}(t)} \mid \delta'(q,a) \mid$.

Let binary vectors $u =< u_1, ..., u_{n_A+n_I} >$ and $v =< v_1, ..., v_{n_A} >$ respectively denote the ordered values at input neurons and output neurons in the transition function module. The first n_A components of vector u, being $< u_1, ..., u_{n_A} >$, together represent $Q_{act}(t)$; and the last n_I components of vector u, being $< u_{n_A+1}, ..., u_{n_A+n_I} >$, together represent current input symbol a. The vectors u and v respectively represent $Q_{act}(t) \times a$ and $Q_{act}(t+1)$ for the given NFA. Let $J_i^{n_I+1} = \{i+1, n_A+1, n_A+2, ..., n_A+n_I\}$ be an interest set for $0 \leq i \leq n_A - 1$. Totally, n_A interest sets $J_0^{n_I+1}$, $J_1^{n_I+1}$, ..., $J_{n_A-1}^{n_I+1}$ are defined. Each of them contains $n_I + 1$ elements. Let current input symbol a be encoded as a binary vector $< a_1, ..., a_{n_I} >$, where $a_k \in \{0, 1\}$ for $1 \leq k \leq n_I$. If $u_{i+1} = 1$, then q_i is in $Q_{act}(t)$ and $u(J_i^{n_I+1}) =< u_{i+1}, u_{n_A+1}, ..., u_{n_A+n_I} >=< 1, a_1, ..., a_{n_I} >$ denotes $\{q_i\} \times a$, where $0 \leq i \leq n_A - 1$.

Settings of Connection Weights in the Transition Function Module

The realization of the symbolic function δ^* by the transition function module is the heart for the construction of the proposed RNN NFA. Note that every transition defined in expression (2) is represented as an ordered binary mapping pair $< Q_{act}(t) \times a(t), Q_{act}(t+1) >$ at the input and output layers of the transition function module, and such mappings are achieved by capturing the regularity in expression $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta^{'}(q, a(t))$ using a 2-layer Perceptron.

Suppose $q_i, q_j \in Q$, $a \in \Gamma$, and $q_j \in \delta^{'}(q_i, a)$, where $0 \leq i, j \leq n_A - 1$. In order to implement the transition (q_i, a, q_j) in the transition function module, the module has to be able to recognize the input $\{q_i\} \times a$. Therefore, an interest set $J_i^{n_I+1} = \{i+1, n_A+1, n_A+2, ..., n_A+n_I\}$ is used for the identification of the ordered $J_i^{n_I+1}$-set partial input vector $u(J_i^{n_I+1}) =< 1, a_1, ..., a_{n_I} >$ which denotes $\{q_i\} \times a$.

According to expressions (4), (12) and their corresponding Perceptron implementation, a hidden neuron h is created, and its associated connection weights as well as threshold are set for the transition (q_i, a, q_j) of the given NFA M_{NFA} in the transition function module as follows :

1. In the 1st-layer connections, according to expression (4),
   - 1 is set as the weight of the connection from the $(i+1)$th input neuron to the hidden neuron h,
   - $2a_k - 1$ is set as the weight of the connection from the $(n_A + k)$th input neuron to the hidden neuron for $1 \leq k \leq n_I$, and
   - 0 is set as the weights of the connections from other input neurons (which are not in $J_i^{n_I+1}$) to the hidden neuron.
2. The threshold of the hidden neuron is set as $\sum_{k=1}^{n_I} a_k$.
3. In the 2nd-layer connections, according to expression (12),
   - 1 is set as the weight of the connection from the hidden neuron to the $(j+1)$th output neuron, and
   - 0 is set as the weights of the connections from the hidden neuron to other output neurons.
4. The thresholds of all output neurons are set as 0 in the transition function module.

In the above representations, if $q_i \in Q_{act}(t)$ and a is current input symbol, then $u_{i+1} = 1$ and $u_{n_A+k} = a_k$ for $1 \leq k \leq n_I$ at time t. Therefore, in the proposed transition function module, the partial input pattern $u(J_i^{n_I+1}) =< 1, a_1, ..., a_{n_I} >$ is identified, the hidden neuron h is activated, and in turn the $(j+1)$th output neuron is activated (i.e., $v_{j+1} = 1$ at time $t+1$ and $q_j \in Q_{act}(t+1)$) according to above settings. So, the transition (q_i, a, q_j) is realized in the transition function module.

Parallel Symbolic Computation in the Transition Function Module

The realization of every move (the move from $Q_{act}(t)$ to $Q_{act}(t+1)$) of M^*_{DFA} in the transition function module of the proposed RNN NFA can be reasoned in two steps as follows :

1. Every hidden neuron h and its associated 1st-layer connections serve as a neural assembly for recognizing a certain ordered $J_i^{n_I+1}$-set partial input pattern. Every such neural assembly checks for a certain transition (q_i, a, q_j) $(0 \leq i, j \leq n_A - 1)$ whether current input vector u contains the ordered $J_i^{n_I+1}$-set partial pattern $u(J_i^{n_I+1}) =< 1, a_1, ..., a_{n_I} >$ (denoting $\{q_i\} \times a$ and $q_i \in Q_{act}(t)$) according to expression (4). If it is, the hidden neuron h is activated by the partial input pattern $\{q_i\} \times a$ at time t and the transition (q_i, a, q_j) is applied. Totally there are n_T such neural assemblies operating in parallel to identify all the possible transitions which are applicable by the states in $Q_{act}(t)$ on current input symbol, and thus they work like a simplified and cost-effective SIMD computer system dedicated to parallel partial pattern matching.
2. Every output neuron along with their associated 2nd-layer connections operate together as a neural assembly to compute a logic `OR`. Every such neural assembly is used to check for a certain state q_j whether any of its fan-in transitions is applied by $Q_{act}(t)$ on current input symbol a. If it is, then output neuron $j+1$ is activated and q_j is in $Q_{act}(t+1)$. Totally there are n_A such neural assemblies (output neurons) which share their input neurons, and operate in parallel to

compute the next set of active states $Q_{act}(t+1)$ according to expression $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta^{'}(q, a)$. Such neural assemblies work together to compute $Q_{act}(t+1)$ like a simplified and cost-effective SIMD computer system dedicated to parallel union computations of sets.

When the representations of $Q_{act}(t)$ and $Q_{act}(t+1)$ are viewed locally (i.e., they are viewed as a set of states respectively), the transition function module realizes the transition function $\delta^{'}$ of the given NFA M_{NFA} if it is restricted that $| Q_{act}(t) | = 1$. Note that $\delta^{'}$ maps from $Q \times \Gamma$ to 2^Q. Such local representations facilitate the parallel recognition of all the transitions applicable by the states in $Q_{act}(t)$ on current input symbol even if $| Q_{act}(t) | \geq 1$. Thus the representations facilitate the concurrent tracking of all possible nondeterministic paths at each move of an NFA. When the representations of $Q_{act}(t)$ and $Q_{act}(t+1)$ are viewed distributedly (i.e., they are viewed as a single state respectively), the transition function module realizes the transition function δ^* of M^*_{DFA}. It means that the transition function module in the proposed RNN concurrently realizes the transition functions $\delta^{'}$ and δ^*. Such a concurrent realization facilitates not only the direct construction of the transition function module from the transition function $\delta^{'}$ but also the linear operation time of the proposed RNN NFA for the processing of input strings.

Settings of Connection Weights in the Acceptance Testing Module

The acceptance testing module of the proposed RNN NFA tests whether an input string is accepted by the RNN NFA at the end of input processing. It is a 1-layer Perceptron which has n_A input neurons and an output neuron.

The output neuron tests whether $Q_{act}(t+1) \in F^*$ by checking whether any state of F is in $Q_{act}(t+1)$ at the end of input processing. Such a test can be characterized by a logic `OR` operation (expression (7)) on the values of the neurons denoting accepting states, and hence it can be realized by a Perceptron according to expression (12) with input neuron v_i denoting whether state q_{i-1} is in F. The connection weights and threshold of the accepting neuron are set as follows:

- If $q_i \in F$, then the connection weight from the $(i+1)$th input neuron to the output neuron is set as 1 for $0 \leq i \leq n_A - 1$. Otherwise it is set as 0.
- The threshold of the output neuron is set as 0.

The End-of-Input Testing Module

In the proposed RNN NFA, an *end-of-input testing module* is used to test the end of input string. The end-of-input testing module is a neural assembly (a 1-layer/1-output Perceptron) realizing a logical `AND` to recognize the *end-of-input symbol* \$ which is encoded as binary vector $< 1^{n_I} >$. By expression (6) and its corresponding Perceptron implementation, all connection weights are set as 1 and the threshold at output neuron is set as $(n_I - 1)$ in the 1-layer/1-output Perceptron to recognize \$. The end-of-input testing module is not shown in the proposed RNN NFA (Figure 2).

3.5 Operation Time Complexity of the Proposed RNN NFA

The time complexity of processing an input string of length n by an NFA directly implemented in single-processor computer systems is $O(m^2 n)$ [18], where m is number of states in the NFA. The proposed RNN NFA concurrently tracks all possible nondeterministic transitions along the processing of an input string in a given NFA by exploiting the inherent parallelism in ANNs. In such a parallel computation, the proposed RNN NFA retains not only the power of nondeterministicism in NFAs but also the advantage of their equivalent DFAs which run deterministically in linear time proportional to the length of input strings. Since the transition function module in the proposed RNN NFA realizes both the transition functions $\delta^{'}$ and δ^*, the time complexity of processing an input string by such a parallel and deterministic computation in the proposed RNN is linearly proportional to the length of the input string, i.e., for an input string of length n the processing time complexity in the proposed RNN NFA is $O(n)$. Therefore the computation overhead of input processing due to the nondeterministicism in NFA can be removed by taking advantage of the inherent parallelism in ANNs as shown by the proposed RNN NFA.

4 Summary and Discussion

Artificial neural networks, due to their inherent parallelism, offer an attractive paradigm for efficient implementations of functional modules for parallel symbol processing. This paper has proposed to exploit the inherent parallelism of neural networks to directly construct NFAs for efficient recognition of regular languages using a class of partially recurrent neural networks. The time complexity of processing an input string of length

n by an NFA implemented in single-processor computer systems is $O(m^2n)$ [18], where m is number of states in the NFA. On the other hand, the corresponding time complexity in the proposed RNN is $O(n)$, i.e., the computation overhead due to the nondeterministicism embedded in NFAs can be removed by exploiting the inherent parallelism in ANNs. From the standpoint of exploiting parallel computing to realize a given NFA, a multiple-processor computer system can approximately do the same as the proposed RNN, but the runtime communication and coordination overhead required by a multiple-processor computer system would make the corresponding time complexity of input processing higher. The overhead gets worse as the number of processors in the computer system increases. Since every DFA is an NFA, the proposed RNN can serve as a more general architecture than that proposed in [1] for constructing finite automata including DFAs and NFAs.

This paper has presented one of the attempts in exploring the inherent parallelism of neural networks to efficiently handle symbol processing. Such attempts are based on a goal that the constructed neural networks for symbol processing be programmed (assembled) from a basic set of neural assemblies for basic operations such as logic `AND` and `OR` operations which have been long used to realize many complex computations via a variety of systematical assembling methods. Such a concept is also demonstrated by [14] which assembles Elman-style RNNs [6] to realizes DFAs from certain `AND` and `OR` neural assemblies. Although many complex computations can be realized by ANNs in terms of *brute-force* binary mapping [4], they can be realized more space-savingly (neuron-savingly) in terms of their embedded regularities which in turn can be translated into logic `AND`s and `OR`s.

In conventional computer systems, many practical computer programs are usually large and built from a set of utility programs (or objects/classes in object-oriented paradigms). Such programs allow code reuse and fast implementation with less errors. The same idea is to be applied in constructing practical neural networks of large size. This paper has constructed a given NFA using a partially RNN which is assembled from basic neural assemblies that realize respectively a logic `AND` and a logic `OR` operations on Boolean variables. Our other attempts include neural networks designed respectively for deterministic finite automata [1], simple database query processing [3], syntax analysis [4], and information retrieval [5]. It is expected to see that the same idea can be applied in constructing neural networks for a variety of important applications in the future.

Acknowledgements

The author would like to acknowledge the support of the National Science Council (through the grant NSC90-2213-E-182-005) and the Chang Gung University during this research.

References

[1] Chen, C. and Honavar, V., Neural Network Automata, *Proceedings of World Congress on Neural Networks*, vol. 4, pp. 470-477, San Diego, CA, June 1994.

[2] Chen, C. and Honavar, V., A Neural Architecture for Content as well as Address-Based Storage and Recall: Theory and Applications, *Connection Science*, vol. 7, no. 3 & 4, pp. 281-300, 1995.

[3] Chen, C. and Honavar, V., A Neural Network Architecture for High-Speed Database Query Processing System, *Microcomputer Applications*, vol. 15, no. 1, pp. 7-13, 1996.

[4] Chen, C. and Honavar, V., A Neural-Network Architecture for Syntax Analysis, *IEEE Transactions on Neural Networks*, vol. 10, no.1, pp. 94-114, January 1999.

[5] Chen, C. and Honavar, V., Neural Architectures for Information Retrieval and Query Processing, In: *A Handbook of Natural Language Processing: Techniques and Applications for the Processing of Language as Text*, R. Dale, H. Moisl, and H. Somers (Ed.), Marcel Dekker, New York, July 2000.

[6] Elman, J. L., Finding Structure in Time, *Technical report, Center for Research in Language (CRL) UCSD*, April 1988.

[7] Giles, C. L. et al., Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks, *Neural Computation*, vol. 4, no. 3, pp. 393-405, 1992.

[8] Goonatilake, S. and Khebbal, S. (Ed.), Intelligent Hybrid Systems, *Wiley*, London, 1995.

[9] Honavar, V., Symbolic Artificial Intelligence and Numeric Artificial Neural Networks: Toward A Resolution of the Dichotomy, in *Computational Architectures Integrating Symbolic and Neural Processes*, Sun R. and Bookman L. (Ed.), pp. 351-388, Kluwer Academic Publishers, Norwell, MA, 1995.

[10] Honavar, V. and Uhr, L. (Ed.), Artificial Intelligence and Neural Networks: Steps Toward Principled Integration, *Academic Press*, New York, 1994.

[11] Honavar, V. and Uhr, L., Integrating Symbol Processing Systems and Connectionist Networks, in *Intelligent Hybrid Systems*, Goonatilake S. and Khebbal S. (Ed.), pp. 177-208, Wiley, London, 1995.

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

[13] Huang, S. C. and Huang, Y. F., Bounds on the Number of Hidden Neurons in Multilayer Perceptrons, *IEEE Transactions on Neural Networks*, vol. 2, no. 1, pp.47-55, January 1991.

[14] Kremer, S. C., On the Computational Power of Elman-Style Recurrent Networks, *IEEE Transactions on Neural Networks*, vol. 6, no. 4, pp.1000-1004, July 1995.

[15] Levine, D. and Aparicio IV, M. (Ed.), Neural Networks for Knowledge Representation and Inference, *Lawrence Erlbaum Associates*, Hillsdale, NJ, 1994.

[16] Minsky, M., Computation: Finite and Infinite Machines, *Prentice Hall*, Englewood Cliffs, NJ, 1969.

[17] Omlin, C. W. and Giles, C. L., Constructing Deterministic Finite-State Automata in Sparse Recurrent Neural Networks, *IEEE International Conference on Neural Networks*, vol. 3, pp. 1732- 1737, Orlando, FL, June 1994.

[18] Sedgewick, R., Algorithms, 2nd ed., *Addison-Wesley*, 1988.

[19] Shastri, L., A Connectionist Approach to Knowledge Representation and Limited Inference, *Cognitive Science*, 12, pp. 331-392, 1988.

[20] Smolensky, P., On Variable Binding and Representation of Symbolic Structure, *Tech Report*, University of Colorado, Boulder, CO, 1987.

[21] Sun, R., Logics and Variables in Connectionist Models: A Brief Overview, *Symbolic Processors and Connectionist Networks for Artificial Intelligence and Cognitive Modeling*, Academic Press, New York, 1994.

[22] Sun, R. and Bookman, L. (Ed.), Computational Architectures Integrating Symbolic and Neural Processes, *Kluwer Academic Publishers*, Norwell, MA, 1995.

[23] Tan, S. and Vandewalle, J., Efficient Algorithm for the Design of Multilayer Feedforward Neural Networks, *IEEE International Joint Conference on Neural Networks*, vol. 2, pp. 190-195, Baltimore, MD, 1992.

[24] Uhr, L. and Honavar, V., Artificial Intelligence and Neural Networks: Steps Toward Principled Integration, in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar V. and Uhr L. (Ed.), pp. xvii-xxxii, Academic Press, New York, 1994.

[25] Wood, D., Theory of Computation, *John Wiley & Sons*, New York, 1987.