Neural Network Automata

by

Chun-Hsien Chen and Vasant Honavar
Computer Science Department
Iowa State University
Ames, Iowa 50011

Abstract

Artificial neural networks (ANN), due to their inherent parallelism and potential fault tolerance offer an attractive paradigm for robust and efficient implementations of functional modules for symbol processing. This paper presents designs of such ANN modules for simulating deterministic finite automata (DFA) and deterministic pushdown automata (DDPA). The designs use an implementation of a class of partially recurrent ANN (modified Elman networks) constructed using a general-purpose binary mapping module (BMP) which in turn is synthesized from multi–layer perceptrons. The paper also discusses some relevant mathematical properties of multi-layer perceptrons that facilitate automated synthesis of BMP modules and points out several potential applications of ANN implementations of DFA and DDPA.

1. Introduction

Artificial neural networks (ANN), due to their inherent parallelism and fault tolerance offer an attractive paradigm for fast and robust implementations of functional modules for a variety of applications. Of particular interest are ANN modules for symbol processing [2], [12], [13] & [14]. This paper focuses on the efficient designs for neural network automata for regular and context-free language recognition. Such modules have a variety of practical applications in computer science, linguistics, systems modeling and control, artificial intelligence, and structural pattern recognition. The proposed designs are facilitated by the binary mapping properties of perceptrons. The rest of the paper is organized as follows:

- Section 2 reviews some of the key concepts and definitions that will be used in the rest of the paper.
- Section 3 presents a technique for automating the synthesis of a general–purpose *binary mapping perceptron* (BMP) module.
- Section 4 describes efficient ANN implementations of deterministic finite state automata (NNDFA) and deterministic pushdown automata (NNDDPA).
- Section 5 concludes with a summary of the key points of the paper and a brief discussion of related ongoing research.

2. Review

2.1 Perceptron

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

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

It is well known that such a 1-layer perceptron can implement only linearly separable functions from $\mathbf{R}^n$ to $[0,1]^m$ [Minsky & Papert, 1969]. A 2-layer Perceptron has one hidden layer of k hidden neurons (and hence 2 layers of weights with each hidden neuron being connected to every input neuron as well as every output neuron). In this paper, we use 2-layer perceptrons in which each hidden neuron uses a hard-limiter function (as described above). The output of a neuron i in the output layer is given by $y_i = f_o(\sum_{l=1}^{k} w_{il}z_l)$ where z_l is the output of hidden neuron l.

2.2 Binary Mapping Function

Given a set U of k input binary vectors u_1,...,u_k of dimension m and a set V of k output binary vectors v_1,...,v_k of dimension n, want to find a binary mapping function $g : U \rightarrow V$ such that $g(u_i) = v_i$ for $1 \leq i \leq k$. In this paper we restrict function g to be a many-to-one function.

The realization of binary mapping function by neural networks has been investigated in a number of different contexts by several researchers over last few years [Baum 1988, Grossberg 1988, Hao, Tan & Vandewall 1990 & 1992, Kwon 1992, Ramacher & Wesseling 1989, and Ruján 1989].

2.3 Deterministic Finite Automata (DFA)

A *finite automaton* M_{FA} is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ [Hopcroft & Ullman, 1979], where Q is a finite set of *states*, Σ is a finite *input alphabet*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and δ is the *transition function* mapping from $Q \times \Sigma$ 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. We denote a DFA by M_{DFA}.

2.4 Deterministic Pushdown Automata (DPDA)

A *pushdown automaton* M_{PDA} is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ [Hopcroft & Ullman, 1979], where Q is a finite set of *states*, Σ is a finite *input alphabet*, Γ is a finite *stack alphabet*, $q_0 \in Q$ is the *initial state*, $Z_0 \in \Gamma$ is a particular stack symbol called the *stack start symbol*, $F \subseteq Q$ is the set of *final states*, and δ is the *transition function* mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$. A pushdown automaton is deterministic if there is at most one transition that is applicable for each combination of state, input symbol and stack top symbol. We denote a DPDA by M_{DPDA}.

3. Binary Mapping Perceptron Module (BMP)

3.1 Linearly Separable Binary Vectors

Theorem 1 : Any single binary pattern is is linearly separable from the set of *all* other binary patterns of same dimension.

This theorem has been proved by Huang & Huang [1991] and Tan & Vandewalle [1992] by finding a linear hyperplane separating a binary vertex from all other vertices in a binary hypercube. Here we examine the spatial distribution and linear separability of binary vertices in a binary hypercube from geometrical perspective. We prove Theorem 1 by locating a set of linear hyperplanes which separate a binary vertex from all other vertices in a binary hypercube. This proof allows us to locate a separating hyperplane that permits an efficient hardware implementation of the corresponding 1-layer Perceptron. Such perceptrons can be used to synthesize a BMP module for any desired binary mapping.

Proof : Let v be a binary vector of dimension n, i.e., $v =< v_1, ..., v_n >$ where $v_i \in \{0, 1\}$ for $1 \leq i \leq n$. Then we can view v as a binary vertex of an n-dimensional hypercube. Hereafter, we will use binary vertex and binary vector interchangeably. Figure 1 shows a 3-dimensional hypercube.

Let $\overline{v} =< \overline{v}_1, ..., \overline{v}_n >$ be the complement binary vector of v, i.e., $v_i + \overline{v}_i = 1$ for $1 \leq i \leq n$; and $v - \overline{v} = v^{ref_v} =< v_1^{ref_v}, ..., v_n^{ref_v} >$. Note that $v_i^{ref_v} \in \{1, -1\}$ for $1 \leq i \leq n$. Let S_k^v be the set of n-dimensional binary vertices which are located at a Hamming distance equal to k from the vertex v; $0 \leq k \leq n$. Then every binary vertex $u \in S_k^v$ falls on a hyperplane H_k^v which is perpendicular to the vector v^{ref_v}. Let $H^v = \{H_k^v; 0 \leq k \leq n\}$. Then the hyperplanes in H^v are mutually parallel. Figure 2 shows this situation.

Let u be a binary vertex in S_k^v, $u - \overline{v} = u^{ref_v} =< u_1^{ref_v}, ..., u_n^{ref_v} >$ and l_u^v be the length of the projection of u^{ref_v} onto the direction of v^{ref_v}. Note that $u_i^{ref_v} = 0$ or 1 if $v_i^{ref_v} = 1$, and $u_i^{ref_v} = 0$ or -1 if $v_i^{ref_v} = -1$ for $1 \leq i \leq n$.

Note also that there are k components $u_i^{ref_v}$ of u^{ref_v} such that $u_i^{ref_v} = 0$ and (n - k) components of u^{ref_v} such that $u_j^{ref_v} = 1$ $or -1$, where $1 \leq j \leq n$. Then

$$l_u^v \quad = \quad \frac{1}{|v^{ref_v}|} v^{ref_v} (u^{ref_v})^T \tag{1}$$

$$= \frac{1}{|v^{ref_v}|}\sum_{i=1}^{n} v_i^{ref_v} u_i^{ref_v} \tag{2}$$

$$= \frac{1}{|v^{ref_v}|}\Big(\sum_{u_i^{ref_v}=1 or -1}^{n-k} v_i^{ref_v} u_i^{ref_v} + \sum_{u_i^{ref_v}=0}^{k} v_i^{ref_v} u_i^{ref_v}\Big) \tag{3}$$

$$= \frac{1}{\sqrt{n}}(n-k) \tag{4}$$

$where (u^{ref_v})^T$ is the *transpose* of u^{ref_v}. Thus, $\forall u \in S_k^v$ the length of the projection of u^{ref_v} in the direction of v^{ref_v} is $(n-k)/\sqrt{n}$. That is, all binary vertices in S_k^v lie on the same hyperplane H_k^v which is perpendicular to the vector v^{ref_v} and has a distance (k - n)/$\sqrt{n}$ to vertex $\overline{v}$. Note that every hyperplane in the set H^v of hyperplanes H_k^v's, $1 \le k \le n$, is parallel to every other hyperplane in the set. Among them, H_1^v is the hyperplane which is closest to the binary vertex v in the n-dimensional space. So there exists at least one separating hyperplane H_S^v between H_1^v and the binary vertex v (v is on H_0^v), which linearly separates the binary vertex v from all other n-dimensional binary vertices which are all on the other H_k^v's, where $1 \le k \le n$. .... □□

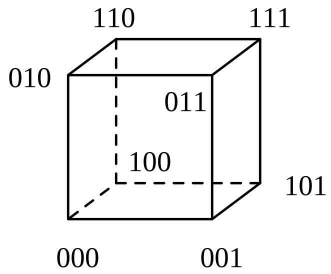

Figure 1. A 3-dimensional hypercube

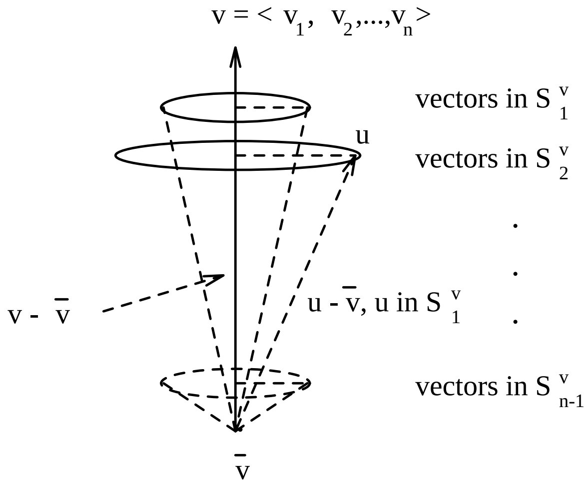

Figure 2. Parallel-layered partitions of n-dimensional vertices and their reference vector v - $\overline{v}$

The ability of hyperplane H_S^v separating binary vertex v from all other binary vertices in a same dimensional hypercube can be implemented by an 1-layer Perceptron with 1 output neuron. Now let us find the expression defining the separating hyperplane H_S^v. Let x be a point on hyperplane H_S^v. Then

$$(n-1)/\sqrt{n} < \frac{1}{|v^{ref_v}|} v^{ref_v} (x^{ref_v})^T < (n-0)/\sqrt{n} \tag{5}$$

$$\Rightarrow \quad (n-1) < (v-\overline{v})(x-\overline{v})^T < n \tag{6}$$

Equation (5) follows from the fact that H_S^v is in between H_0^v and H_1^v. So the projection length of x^{ref_v} in the direction of v^{ref_v} is less than the projection length of the reference vector of any binary vertex on H_1^v and is greater than that of the reference vector of any binary vertex on H_0^v. Equation (6) says that there are infinite number of such separating hyperplanes H_S^v's that point x could be on. To efficiently implement the separating hyperplane H_S^v in a 1-layer Perceptron, let us choose $(v-\overline{v})(x-\overline{v})^T = n-1$ as the expression defining the separating hyperplane H_S^v. Note that in this case $(v-\overline{v})(w-\overline{v})^T \le (n-1)$ for any binary vertex w of dimension n except binary vertex v. Then

$$H_S^v \equiv (v-\overline{v})(x-\overline{v})^T = (n-1) \tag{7}$$

$$
\begin{aligned}
&\equiv \quad (v-\overline{v})x^T-(v-\overline{v})\overline{v}^T-(n-1)=0 && (8)\\
&\equiv \quad (v-\overline{v})x^T-(n-|\overline{v}|^2-1)=0, \quad \textit{note} \ |v|^2+|\overline{v}|^2=n && (9)\\
&\equiv \quad (v-\overline{v})x^T-(|v|^2-1)=0 && (10)\\
&\equiv \quad (v_1-\overline{v}_1)x_1+...+(v_n-\overline{v}_n)x_n-(|v|^2-1)=0 && (11)\\
&\equiv \quad (2v_1-1)x_1+...+(2v_n-1)x_n-(|v|^2-1)=0 && (12)\\
&\equiv \quad (\sum_{i=1}^{n}(2v_i-1)x_i)-(|v|^2-1)=0 && (13)
\end{aligned}
$$

So the separating hyperplane H_S^v can be efficiently implemented by a 1-layer Perceptron whose output neuron has a threshold of $|v|^2-1$ and the connection weights on the input links are given by $2v_i-1$ for $i \leq i \leq n$.

Note that the value of $2v_i-1$ is either 1 or -1, x_i is either 1 or 0, $(2v_i-1)x_i$ is either 1, 0 or -1, and $|v|^2$ is integer. So in the hardware implementation of this Perceptron, only integer adder is needed. Note also that the maximal summation value at output neuron is 1 and minimal value is $-(n-1)$, since the separating hyperplane H_S^v is defined as $(v-\overline{v})(x-\overline{v})^T-(n-1)=0$, the maximal value of $(v-\overline{v})(x-\overline{v})^T$ is n, and the minimal value of $(v-\overline{v})(x-\overline{v})^T$ is 0. If zero is used as the initial value in the summation operation at the output neuron, -n is the minimal and n the maximal value appearing in the whole summation process. If $-(|v|^2-1)$ is used as the initial value in the summation operation at the output neuron, $-(n-1)$ is the minimal and 1 the maximal value appearing in the whole summation process. In both cases, an m-bit integer adder is needed where $m=\lceil\log_2(2n+1)\rceil$ for the former and $m=\lceil\log_2(n+1)\rceil$ for the latter. Furthermore, the hardware complexity of the the latter implementation is lower since the summation operations executed at the output neuron only involve *increment* and *decrement* operations. So, in the latter case the integer adder could be replaced by a simpler module, *counter*, to execute the summation operation.

3.2 Synthesis of General-purpose Binary Mapping Perceptron Module

Given a set A of k input binary vectors a_1,...,a_k of dimension m, where $a_h=<a_{h1},...,a_{hm}>$ and $a_{hi}\in\{0,1\}$ for $1\leq h\leq k$ & $1\leq i\leq m$, and a set D of k desired output binary vectors d_1,...,d_k of dimension n, where $d_h=<d_{h1},...,d_{hn}>$ and $d_{hj}\in\{0,1\}$ for $1\leq h\leq k$ & $1\leq j\leq n$. No two a_i's are equal. Let $\mathbf{B}^m$ denotes the universe of m-dimensional binary vectors and 0^n the n-dimensional binary vector of all zeros. We want to find a binary mapping function $g:\mathbf{B}^m\rightarrow(D\cup\{0^n\})$ such that $g(a_h)=d_h$ for $1\leq h\leq k$ and $g(x)=0^n$ for $x\in(\mathbf{B}^m-A)$. Note that we restrict function g to be a many-to-one function.

A 2-layer BMP module for any desired binary mapping function g can be synthesized using the 1-layer Perceptron (defined in section 3.1) as follows: The BMP module has m input, k hidden and n output neurons. For each binary mapping ordered pair (a_h, d_h), where $1\leq h\leq k$, we create a hidden neuron h with threshold $|a_h|^2-1$. The connection weight from input neuron i to this hidden neuron is $2a_{hi}-1$ $(=a_{hi}-\overline{a}_{hi})$ and that from this hidden neuron to output neuron j is d_{hj}. The threshold for each of the output neurons is set to zero. Figure 3 shows the implementation. The activation function f_o at output neuron is identity function, i.e., $f_o(x)=I(x)=x$ and that f_h at hidden node is hardlimiter function, i.e.,

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

Note that for input binary vector a_h, only the hidden neuron h produces an output of 1, and the output values from all other hidden neurons are 0 in our BMP module. So the value at output neuron j is d_{hj}, and hence the output binary vector will be $<d_{h1},...,d_{hn}>=d_h$. Since only one of the hidden neurons has output value 1 and others have output value 0 in the computation at hidden layer, the computation at output layer can be seen as being *enabled* by one of the hidden neurons. So, in hardware implementation it is not necessary to use summation operation to compute the output value from each output neuron. (Note that in this case we have to use *preset* and *synchronization control* to handle case when $g(x)=0^n$ for $x\in(\mathbf{B}^m-A)$.)

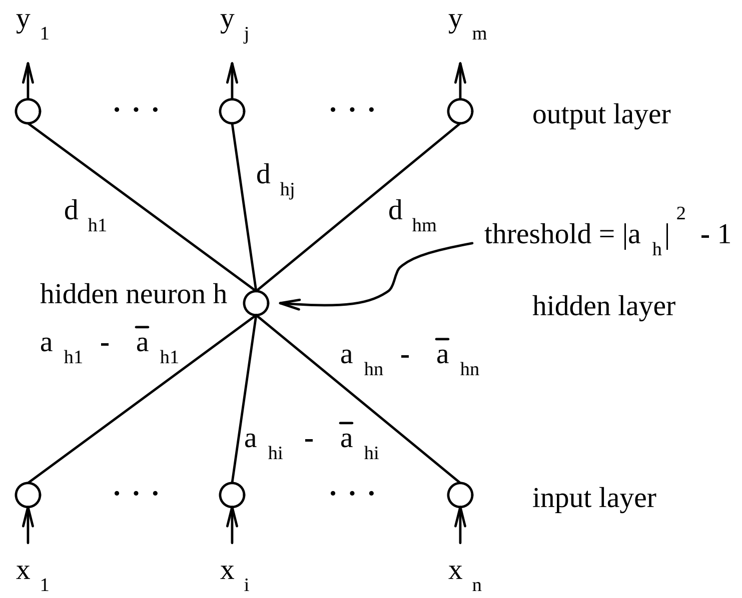

Figure 3. Perceptron implementation for binary mapping pair (a_h, d_h)

4. Neural Network Automata

The prerequisites for a neural network implementation of automata are: an encoding for the set Q of finite states, an encoding for alphabets including the set Σ of input alphabet, the set Γ of stack alphabet (only for PDA) and tape alphabet (only for Turing Machines), a representation for transition function, and a neural network architecture for continuous execution of an automaton. In this paper we restrict ourselves to neural network implementations of DFA and PDA. In this case, the first two requirements can be met by a binary coding, the third by using a binary mapping Perceptron module, and the fourth by adding a synchronization mechanism in modified Elman networks [Elman, 1988].

Let us encode the set Q of finite states into a set S of binary values of dimension m, the set Σ of input alphabet into a set I of of binary values of dimension n and the set Γ of stack alphabet into a set K of binary values of dimension k. Let $0^m \in S$ denote the encoded binary value of *dead state* (*garbage state*), a state which is not a final state and has transitions to itself on all input symbols. To standardize the implementation we let $1^m \in S$ denote the encoded binary value of initial state. Then $m = \lceil \log_2(|Q|+1) \rceil$, $n = \lceil \log_2 |\Sigma| \rceil$, and $k = \lceil \log_2 |\Gamma| \rceil$, where $|Z|$ denotes the cardinality of set Z. In the BMP module simulating the transition function, the number of needed hidden nodes equals the number of different transitions of the transition function.

4.1 NN Deterministic Finite Automata (NN DFA)

Figure 4 shows the neural network architecture for simulating a DFA. Let 0,1,2,...,t denote a succession of points along the discrete time line. $State(t)$ denotes current state and $state(t+1)$ denotes next state. $Input(t)$ denotes current input symbol. This NN DFA module consists of two BMP modules, one *accepting state trapping module* (AST module) and three buffers. One buffer stores current state $state(t)$, another stores input symbol $input(t)$ and the other stores next state $state(t+1)$. The first two buffers operate under synchronization control. The *reset* link resets the NN DFA to initial state. The synchronization control enforces discrete time 0,1,...,t.

BMP module 1, called *transition BMP module*, simulates the transition function of a DFA. The transition BMP module represents each state transition as an ordered pair of binary vectors. Suppose $p, q \in Q, a \in \Sigma, \delta(p,a) = q$ is a valid transition, and p, q and a are encoded as binary vectors such that $p = <p_1, ..., p_m>, q = <q_1, ..., q_m>$ and $a = <a_1, ..., a_n>$ where $p_i, q_i, a_j \in \{0,1\}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. Then the state transition $\delta(p,a) = q$ is a binary mapping ordered pair $(<p_1, ..., p_m, a_1, ..., a_n>, <q_1, ..., q_m>)$ implemented in transition BMP module using the algorithm for the synthesis of a BMP module (see section 3.2). $<p_1, ..., p_m>$ represents the encoded binary value of current state (state(t)), $<a_1, ..., a_n>$ represents the encoded binary value of current input symbol (input(t)), and $< q_1, ..., q_m >$ represents the encoded binary value of next state (state(t+1)). The AST module is optional and can be implemented by AND/OR gates or a BMP module. It enables BMP module 2 to produce an output only when the NN DFA goes into an accepting state. There might need a connection from the AST module to upper-layer control to tell a rejection when the AST module traps rejecting states, i.e., when this NN DFA goes into rejecting states. If the AST module is omitted, this NN DFA simulates a *Moore machine* (Hopcroft & Ullman, 1979), a super-class automaton of finite automaton which produces output depending on current state. BMP module 2 is optional, and it allows the output of the NN DFA to be remapped from the output of BMP module 1. Note that any unspecified transition will automatically have the next state coded as 0^m as a consequence of our design of the transition BMP module. This simplifies the implementation of DFA, since any transition to rejecting state does not need to be implemented using a hidden neuron in the transition BMP module.

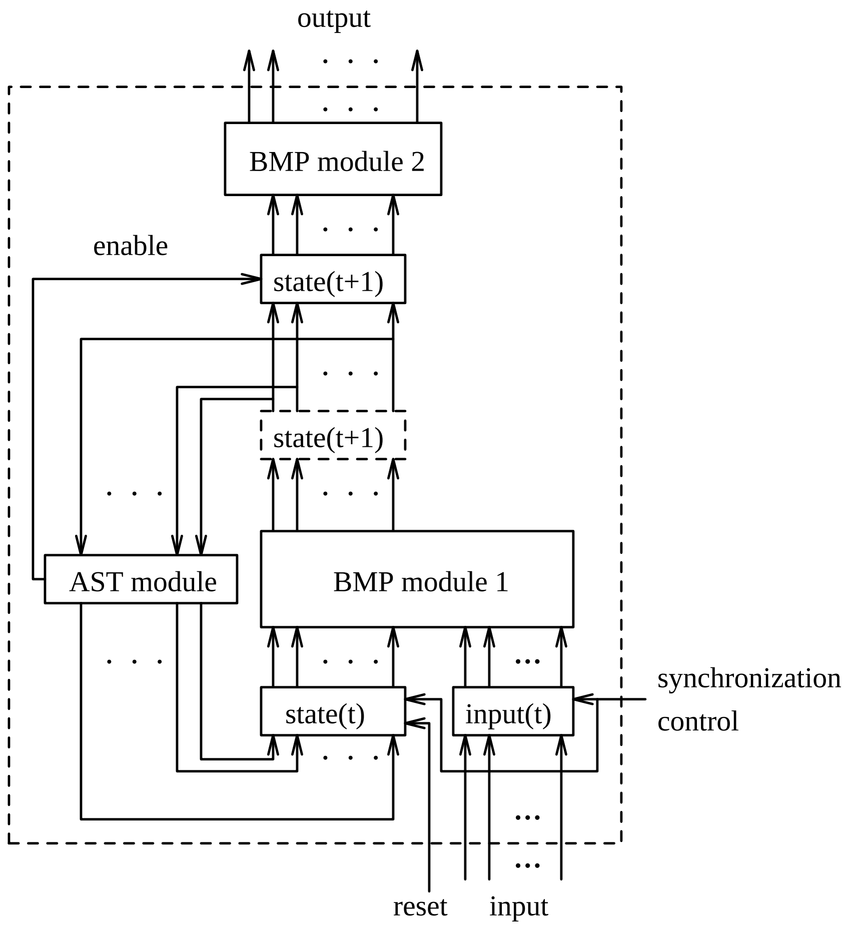

Figure 4. Neural network DFA. The dotted box labeled with state(t+1) exists only logically but not physically.

4.2 NN Deterministic Pushdown Automata (NN DPDA)

Figure 5 shows the neural network architecture for simulating a DPDA. $State(t)$ denotes current state and $state(t+1)$ denotes next state. This NN DPDA module consists of three BMP modules, one AST module, one stack mechanism module and four buffers. One buffer stores current state $state(t)$, one stores input symbol $input(t)$, another stores stack top symbol $stack_{top}$ and the other stores next state $state(t+1)$. The first three buffers operate under synchronization control. The *reset* link resets the NN DPDA to initial state. The synchronization control enforces discrete time 0,1,...,t.

BMP module 1 (the transition BMP module) simulates the transition function of a DPDA. Each state transition is coded as an ordered pair of binary mapping vectors.

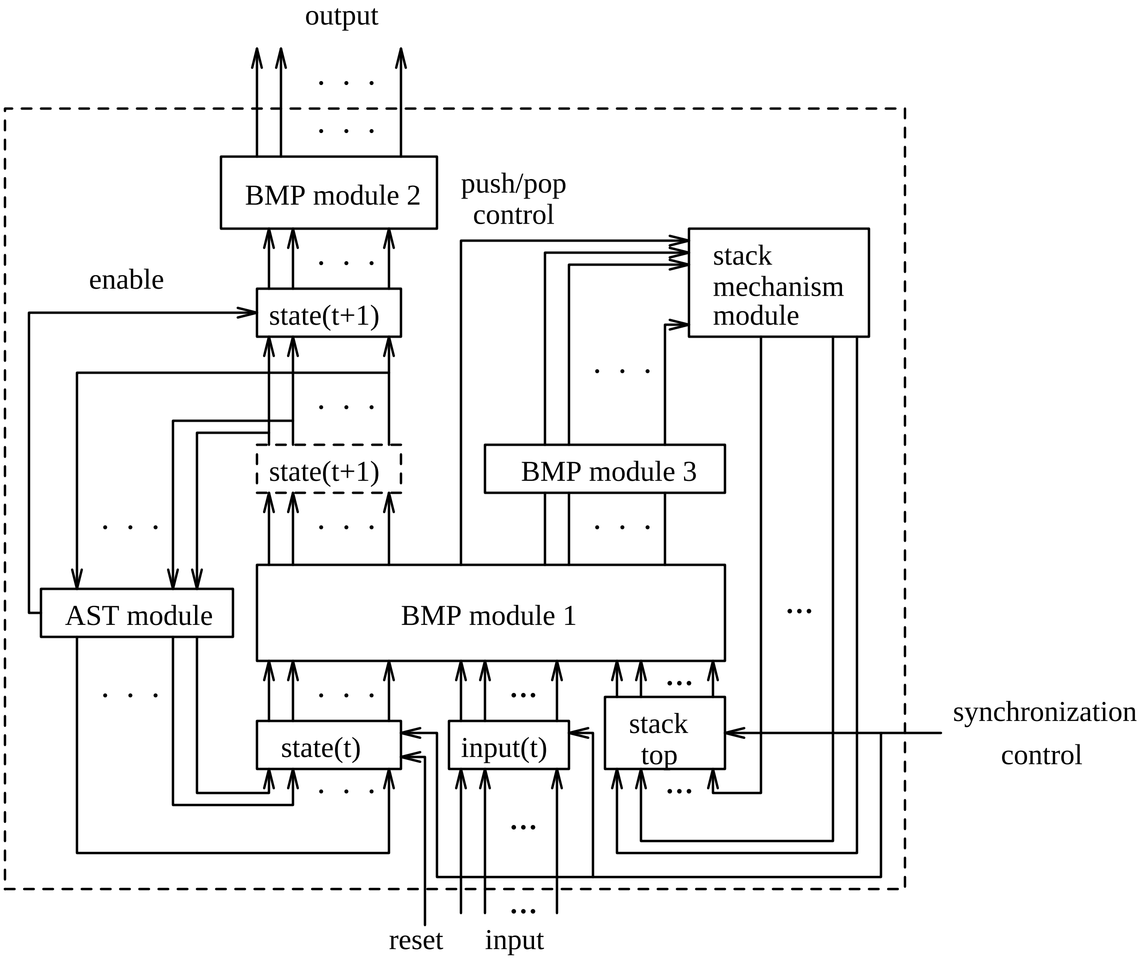

Figure 5. Neural network DPDA. The dotted box labeled with state(t+1) exists only logically but not physically.

Suppose $p, q \in Q, a \in (\Sigma \cup \{\epsilon\}), \alpha, \beta \in \Gamma, s \in \{pop,\ push\}, \delta(p, a, \alpha) = [q, s, \beta]$ is a valid transition, and p, q, a, α, β and s are encoded into binary vectors such that $p =< p_1, ..., p_m >$ $, q =< q_1, ..., q_m >$, $a =< a_1, ..., a_n >$, $\alpha =< \alpha_1, ..., \alpha_k >, \beta =< \beta_1, ..., \beta_k >$ and $s =< s_1 >$ where $p_i, q_i, a_j, \alpha_l, \beta_l, s_1 \in \{0, 1\}$ for $1 \leq i \leq m$, $1 \leq j \leq n$, and $1 \leq l \leq k$. Note that our representation of a transition of a DPDA is different from the conventional representation in that we express pop/push action explicitly. Then transition $\delta(p, a, \alpha) = [q, s, \beta]$ is represented as the binary mapping ordered pair $(< p_1, ..., p_m, a_1, ..., a_n, \alpha_1, ..., \alpha_k >, < q_1, ..., q_m, s_1, \beta_1, ..., \beta_k >)$ to be implemented in the transition BMP module. $< p_1, ..., p_m >$ represents the encoded binary value of current state $state(t)$, $< a_1, ..., a_n >$ represents the encoded binary value of current input symbol $input(t)$, $< \alpha_1, ..., \alpha_k >$ represents stack top symbol $stack_{top}$, $< q_1, ..., q_m >$ represents the encoded binary value of next state $state(t + 1)$, $< s_1 >$ represents the pop/push action in the transition, and $< \beta_1, ..., \beta_k >$ represents the produced stack symbols in the transition. There is a *push/pop* connection from BMP module 1 to the stack mechanism module. This link informs the stack mechanism module whether to pop or push. Stack mechanism module can be implemented by sequence memory, memory read-write head and buffer, and a register (*counter*) for storing pointer pointing to the stack top. A push action increments the pointer (counter) and a pop action decrements the pointer (counter). The special symbol $stack_{bottom}$ is stored at the beginning of the memory. The AST module is optional and can be implemented by AND/OR gates or a BMP module. It enables BMP module 2 to produce output only when the NN DPDA goes into accepting states. There might need a connection from the AST module to upper-layer control to tell a rejection when the AST module traps rejecting states, i.e., when this NN DPDA goes into rejecting states. BMP module 2 is optional, and it allows the output from the NN DPDA to be remapped from the output of BMP module 1. BMP module 3 is optional and provides remapping of stack symbol produced from BMP module 1. Note that any unspecified transition will have the next state 0^m given our implementation of the transition BMP module.

5. Discusion

In this paper we present an algorithm for automating the synthesis of general-purpose binary mapping module in the form of a 2-layer Perceptron. We also present efficient NN architectures for simulation of finite state and pushdown automata. Such designs can be used in a variety of tasks that require the integration of symbol processing capabilities into neural networks (for examples of such applications, see Shastri [12], Smolensky [13], Sun [14].) Other applications of NN automata include efficient hardware implementations of lexical analyzers and parsers used in compilers (Chen & Honavar [2]) and natural language processing, and language recognizers used in syntactic pattern recognition.

References

[1] E. B. Baum, On the Capabilities of Multilayer Perceptron, *Journal of Complexity*, vol. 4, pp. 193-215, 1988.

[2] C. Chen and V. Honavar, Modularized Neural Networks for Lexical Analyzer and Grammar Parser. *To appear.*

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

[4] S. Grossberg, Nonlinear neural networks: Principles, mechanisms, and architectures, *Neural Networks*, vol. 1, pp. 17-61, 1988.

[5] J. Hao, S. Tan and J. Vandewalle, A Geometric Approach to the Structural Synthesis of Multilayer Perceptron Neural Networks, *Proceeding of 1990 International Joint Conference on Neural Networks*, vol. 2, pp. 881-885.

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

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

[8] M. Minsky and S. Papert, Perceptrons: An Introduction to Computational Geometry, MIT Press, 1969.

[9] T. M. Kwon, A Guaranteed Training of Binary Pattern Mappings Using Gaussian Perceptron Networks, *IEEE International Joint Conference on Neural Networks*, vol. 3, pp. 614-619, Baltimore, 1992.

[10] U. Ramacher and M. Wesseling, A Geometrical Approach to Neural Network Design, *Proceeding of International Joint Conference on Neural Networks*, vol. 2, pp. 147-153, 1989.

[11] P. Ruján, A Geometric Approach to Learning in Neural Networks, *Proceeding of International Joint Conference on Neural Networks*, vol. 2, pp. 105-109, 1989.

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

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

[14] R. Sun, 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.

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