Neural architectures for database query processing, syntax analysis, knowledge representation, and inference

by

Chun-Hsien Chen

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Major Professor: Vasant Honavar

Iowa State University

Ames, Iowa

1998

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Chun-Hsien Chen

has met the dissertation requirements of Iowa State University

Committee Member

Committee Member

Committee Member

Committee Member

Major Professor

For the Major Program

For the Graduate College

TABLE OF CONTENTS

ACKNOWLEDGEMENTS x

ABSTRACT xi

1 INTRODUCTION 1

1.1 Artificial Neural Networks 3

1.1.1 Artificial neural units 3

1.1.2 Activation functions 4

1.1.3 Types of artificial neural networks and their computational capabilities 5

1.1.4 Implementation of artificial neural networks 6

1.2 A Brief Review of Artificial Neural Networks 6

1.3 An Overview of the Dissertation 9

2 A NEURAL MEMORY FOR CONTENT AS WELL AS ADDRESS-BASED STORAGE AND RECALL 13

2.1 Introduction 13

2.1.1 Information retrieval and binary mapping 14

2.1.2 Associative memory (Content-addressed memory) 15

2.1.3 Address-based memory 18

2.1.4 Perceptrons 19

2.2 Multi-layer Perceptrons as Neural Memories 19

2.2.1 The application of linear separability of binary vertices in pattern classification 20

2.2.2 Best match: pattern classification with precision control 23

2.2.3 Storage capacity 24

2.2.4 Synthesis of associative and address-based memories 24
2.2.5 Exact match: binary mapping Perceptron (BMP) module 27
2.2.6 Conversion between memory models using bipolar and binary inputs . . 28
2.3 Properties of the Proposed Neural Associative Memory 32
2.3.1 Partial match: associative recall from a partially specified input 33
2.3.2 Multiple associative recalls 35
2.3.3 Fault tolerance 38
2.4 Summary and Discussion 40

3 NEURAL ARCHITECTURES FOR INFORMATION RETRIEVAL AND DATABASE QUERY PROCESSING 42

3.1 Introduction 42
3.1.1 Information retrieval in neural associative memories 44
3.2 Query Processing Using Neural Associative Memories 45
3.2.1 Realization of lexical access for a machine-readable lexicon using a neural associative memory 45
3.2.2 Realization of a library query system using a neural associative memory 49
3.2.3 The implementation of case insensitive pattern matching 52
3.3 Comparison with Other Database Query Processing Techniques 53
3.3.1 Performance of current electronic realization for neural networks 53
3.3.2 Analysis of query processing in conventional computer systems 54
3.4 Summary and Discussion 58

4 NEURAL ARCHITECTURES FOR ELEMENTARY LOGICAL INFERENCE 59

4.1 Introduction 59
4.2 Neural Assemblies for the Recognition of Partial Patterns 60
4.2.1 A neural assembly for inclusive recognition 61
4.2.2 A neural assembly for exclusive recognition 62
4.3 A Neural Assembly for Executing a Logical `AND` (`AND` Neural Assembly) 64

4.4 Neural Assemblies for Executing Logic ORs (OR Neural Assemblies) 67
4.4.1 A general OR neural assembly 67
4.4.2 A monotone OR neural assembly 69
4.5 A Neural Architecture for Realizing DNF Boolean Functions 70
4.6 Summary and Discussion 71
5 NEURAL ARCHITECTURES FOR SEQUENCE PROCESSING 73
5.1 Introduction 73
5.1.1 Symbolic functions and binary mappings 74
5.2 Neural Network Design for Deterministic Finite Automata (NN DFA) 76
5.2.1 Deterministic finite automata (DFA) 76
5.2.2 Architecture of NN DFA 77
5.3 Neural Network Design for Deterministic Pushdown Automata (NN DPDA) . . 80
5.3.1 Deterministic pushdown automata (DPDA) 80
5.3.2 Architecture of NN DPDA 81
5.4 Neural Network Design for Stack (NN Stack) 84
5.4.1 Symbolic representation of stack 84
5.4.2 Architecture of NN Stack 85
5.4.3 NN Stack in action 89
5.5 Neural Network Design for Nondeterministic Finite Automata (NN NFA) . . . 90
5.5.1 Nondeterministic finite automata (NFA) 91
5.5.2 Model for concurrently tracking all the possible nondeterministic moves in the operation of an NFA using RNN 92
5.5.3 Architecture of NN NFA 95
5.5.4 Proof of correctness 103
5.5.5 NN NFA in Action 105
5.6 Summary and Discussion 106
6 NEURAL ARCHITECTURES FOR SYNTAX ANALYSIS 107
6.1 Introduction 107

6.1.1 Review of related research on neural architectures for syntax analysis . 108
6.2 Neural Network Design for a Lexical Analyzer (NNLexAn) 111
6.2.1 Neural network design for a word segmenter (NNSeg) 112
6.2.2 Neural network design for a word lookup table (NNLTab) 114
6.3 A Modular Neural Architecture for LR Parser (NNLR Parser) 115
6.3.1 Representation of parse table 115
6.3.2 Representation of parsing moves and parse trees 119
6.3.3 Architecture of NNLR parser 120
6.3.4 NNLR Parser in action 122
6.4 Performance Analysis 127
6.4.1 Performance analysis of lexical analyzer 128
6.4.2 Performance analysis of LR parser 130
6.5 Summary and Discussion 132
7 CONCLUSION 136
BIBLIOGRAPHY 137

LIST OF FIGURES

Figure 1.1 A typical computing unit of an ANN 4

Figure 2.1 Examples of memory pattern, noisy patterns, and partial pattern . . . 17

Figure 2.2 The spatial distribution of a 3-dimensional and an n-dimensional binary hypercubes 22

Figure 2.3 The setting of connection weights and hidden node threshold in the proposed neural memory (a 2-layer Perceptron with binary input) for a given associated memory pair 26

Figure 2.4 The settings of connection weights and hidden node threshold in the proposed BMP module for an associated binary mapping ordered pair 28

Figure 2.5 The setting of connection weights and hidden node threshold in the proposed neural memory (a 2-layer Perceptron with bipolar input) for a given associated memory pair. 32

Figure 3.1 A modular design of the proposed ANN memory for easy expansion. This 1-dimensional array structure can be easily extended to *2* or *3*-dimensional array structures. 50

Figure 4.1 A 1-layer Perceptron which recognizes all the 5-dimensional binary patterns that contain the partial pattern *<1,?,0,?,1>*, where *?* denotes don't care 63

Figure 4.2 A 1-layer Perceptron which recognizes all the 5-dimensional binary patterns that don't contain the partial pattern *<1,?,0,?,1>*, where *?* denotes don't care 65

Figure 4.3 An `AND` neural assembly which realizes the logical `AND` function *C(v)* . 66

Figure 4.4 An `OR` neural assembly which realizes the logical `OR` function *D(v)* . . . 69

Figure 4.5 An neural architecture which realizes the DNF Boolean function *E(v)* . 72

Figure 5.1 The proposed modular neural network architecture for DFA 79

Figure 5.2 The proposed modular neural network architecture for DPDA 83

Figure 5.3 The proposed neural network architecture for stack mechanism 86

Figure 5.4 The state diagram of an NFA that accepts any input string containing the sub-string `abaa` 93

Figure 5.5 The state diagram of a DFA that accepts any input string containing the sub-string `abaa` 93

Figure 5.6 The proposed recurrent neural network architecture for concurrently tracking all the nondeterministic computations of a given NFA 96

Figure 6.1 The simplified state diagram of a DFA which recognizes keywords : `begin, end, if, then`, and `else` 112

Figure 6.2 The state diagram of a DFA which simulates a simple word segmenter carving continuous input stream of characters into words including integer constants, keywords and identifiers 113

Figure 6.3 The proposed neural network architecture for LR(1) parser 118

Figure 6.4 The state diagram of the DFA M_{L_1} for the lexical analyzer L_1 126

LIST OF TABLES

Table 2.1 The corresponding maximal storage capacity of a 1-layer Perceptron with 100 input neurons for classifying binary patterns for a range of allowable noise levels 25

Table 3.1 A comparison of the estimated performance of the proposed neural associative memory with that of other techniques commonly used in conventional computer systems for locating a record pointer in key-based organizations 56

Table 3.2 A comparison of the capabilities of the proposed neural associative memory with those of other techniques commonly used in conventional computer systems for exact match and partial match 56

Table 6.1 The parse table of the LR(1) parser for grammar G_1 124

Table 6.2 The transition function δ_{L_1} of the DFA M_{L_1} 126

Table 6.3 Moves of the LR(1) parser for grammar G_1 on input string I×I+I . . . 127

Table 6.4 A comparison of the estimated performance of the proposed NNLR Parser with that of conventional computer systems for syntax analysis 132

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to Dr. Vasant Honavar for his support, assistance, and guidance, as well as challenging and inspiring criticism in the preparation of this dissertation and in my studies at Iowa State University. I would also like to thank my Graduate Committee: Dr. Tom Barta, Dr. Julie Dickerson, Dr. Les Miller, and Dr. Johnny Wong for their helpful advice, comments, and suggestions.

I am deeply indebted to the Extended and Continuing Education Office, Iowa State University, for the warm friendship of the staff, the valuable work experience, and the timely financial support without which my study for a Ph.D. would not be possible. I am grateful to Russell Meier and the Artificial Intelligence Research Group: Karthik Balakrishnan, Armin Mikler, Rajesh Parekh and Jihoon Yang for their friendship and inspiring discussion. I am also grateful to my friends in Ames, Iowa, for keeping me company at my leisure time and for nourishing me with friendship.

Also, I would like to express my deepest gratitude to my parents for their everlasting support and love.

ABSTRACT

Artificial neural networks, due to their inherent parallelism, potential for fault tolerance, and adaptation through learning, offer an attractive computational paradigm for a variety of applications in computer science and engineering, artificial intelligence, robotics, and cognitive modeling. Despite the success in the application of ANN to a broad range of numeric tasks in pattern classification, control, function approximation, and system identification, the integration of ANN and symbolic computing is only beginning to be explored. This dissertation explores to integrate ANN and some essential symbolic computations for content-based associative symbolic processing. This offers an opportunity to explore the potential benefits of ANN's inherent parallelism in the design of high performance computing systems for real time content-based symbolic processing applications. We develop methods to systematically design massively parallel architectures for pattern-directed symbol processing using neural associative memories as key components. In particular, we propose neural architectures for content-based as well as address-based data storage and recall, information retrieval and database query processing, elementary logical inference, sequence processing, and syntax analysis. Their potential advantages over conventional serial computer implementations of the same functions are examined in the dissertation.

1 INTRODUCTION

The goal of artificial intelligence (AI), broadly interpreted, is to understand and engineer intelligent systems. It is often suggested that traditionally serial symbol processing systems of AI and inherently massively parallel artificial neural networks (ANN) offer two radically, perhaps even irreconcilably different paradigms for modelling minds and brains — both artificial as well as natural [130, 160]. AI has been successful in applications such as theorem proving, knowledge–based expert systems, mathematical reasoning, syntax analysis, and related applications which mainly involve systematic symbol manipulation. On the other hand, ANN have been particularly successful in applications such as pattern recognition, function approximation, and nonlinear control [60, 150] which involve primarily numeric computation. Meyerowitz has suggested that the design of neural architectures capable of supporting dynamic representations for symbol manipulation is one of the grand challenges of neural network research [113]. As shown by Church, Kleene, McCulloch, Post, Turing, and others through their work on the theory of Computation [117, 100], both AI and ANN represent particular realizations of a universal (Turing-equivalent) model of computation [185]. Thus, despite assertions by some to the contrary, any task that can be realized by one can, in principle, be accomplished by the other. However, most AI systems have been traditionally programmed in languages that were influenced by Von Neumann's design of a serial stored program computer. ANN systems on the other hand, have been inspired by (albeit overly simplified) models of biological neural networks. They represent different commitments regarding the architecture and the primitive building blocks used to implement the necessary computations. Thus they occupy different regions characterized by possibly different cost–performance tradeoffs in a much larger space of potentially interesting designs for intelligent systems. Recently, several researchers have begun

to explore previously unexplored parts of this design space.

Given the reliance of both traditional AI and ANN on essentially equivalent formal models of computation, a central issue in design and analysis of intelligent systems has to do with the identification and implementation, under a variety of design, cost, and performance constraints, of a suitable subset of Turing-computable functions that adequately model the desired behaviors. Today's AI and ANN systems each demonstrate at least one way of performing a certain task (e.g., logical inference, pattern recognition, syntax analysis) naturally and thus pose the interesting problem for the other of doing the same task, perhaps more elegantly, efficiently, robustly, or cost-effectively than the other. In this context, it is beneficial to critically examine the often implicit and unstated assumptions on which current AI and ANN systems are based and to identify alternative (and potentially better) approaches to designing such systems. Massively parallel symbol processing architectures for AI systems or highly structured (as opposed to homogeneous, fully connected) ANN are just two examples of a wide range of approaches to designing intelligent systems [185, 72, 73]. Of particular interest are alternative designs (including synergistic hybrids of ANN and AI designs) for intelligent systems [47, 65, 70, 72, 73, 99, 136, 172, 179, 185]. Examples of such systems include: neural architectures for information retrieval and database query processing [23, 24], generation of context-free languages [187], rule-based inference [5, 31, 141, 167, 176], computer vision [11, 119], natural language processing [14, 32], learning [46, 69, 168], and knowledge-based systems [94, 145]. We strongly believe that a judicious and systematic exploration of the design space of such systems is essential for understanding the nature of key cost–performance tradeoffs in the synthesis of intelligent systems.

This dissertation explores to integrate ANN and some essential symbolic computations for content-based associative symbolic processing. This offers an opportunity to explore the potential benefits of ANN's inherent parallelism in the design of high performance computing systems for real time content-based symbolic processing applications.

1.1 Artificial Neural Networks

ANN are biologically inspired by the neural systems of human brain which are massively parallel interconnected networks of hierarchically organized nerve cells (neurons). ANN are extremely simplified models of biological neural systems in many aspects such as the structure of basic computational units, the mechanism for information processing, network architecture, etc. Compared to most current digital computer systems, ANN are particularly well-suited for pattern-directed problems – pattern completion, pattern classification and pattern association [29] which arise frequently in applications such as language processing, speech recognition, and pattern recognition.

It is worth mentioning that the primary goal of ANN research (unlike neural modelling or computational neuroscience research) is not to discover a computational model for the detailed processes of human brain but to technologically pursue a computing paradigm which can effectively realize and efficiently perform high-level intelligent processes.

1.1.1 Artificial neural units

A typical computing unit (node) in an ANN has n input and m output connections, each of which has an associated weight. The node computes the weighted sum on the inputs, compares the sum to its node threshold, and produces its output based on an activation function. A commonly used activation function is threshold function. The resulting output is sent along the output connections to other nodes. The output of such a node used in this dissertation is defined by

$$y = f(s) \quad and \quad s = \sum_{i=1}^{n} w_i x_i - \theta \tag{1.1}$$

where x_i is the value of input i, w_i is the associated weight on input connection i, θ is the node threshold, y is the output value, and f is the activation function. Figure 1.1 shows such a node.

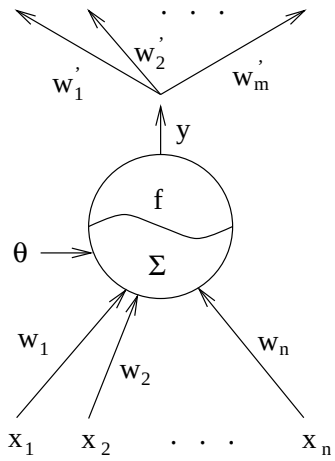

Figure 1.1 A typical computing unit of an ANN

1.1.2 Activation functions

The types of activation functions used by an ANN affect its expressiveness, computational capabilities, and performance. Several typical activation functions are *linear, binary sigmoidal, bipolar sigmoidal, binary hardlimiter, bipolar hardlimiter, gaussian, and ramp* [56, 102] defined as follows:

linear : $f_L(s) = cs$, where $s = \sum_{i=1}^{n} w_i x_i - \theta$ and $c > 0$

binary sigmoidal : $f_S(s) = \frac{1}{1+e^{-cs}}$, where $s = \sum_{i=1}^{n} w_i x_i - \theta$ and $c > 0$

bipolar sigmoidal : $\breve{f}_S(s) = \frac{1-e^{-cs}}{1+e^{-cs}}$, where $s = \sum_{i=1}^{n} w_i x_i - \theta$ and $c > 0$

binary hardlimiter : $f_H(s) = \begin{cases} 1 & \text{if s} \geq 0 \\ 0 & \text{otherwise} \end{cases}$, where $s = \sum_{i=1}^{n} w_i x_i - \theta$

bipolar hardlimiter : $\breve{f}_H(s) = \begin{cases} +1 & \text{if s} \geq 0 \\ -1 & \text{otherwise} \end{cases}$, where $s = \sum_{i=1}^{n} w_i x_i - \theta$

gaussian : $f_G(s) = e^{\frac{-s}{2\sigma^2}}$, where $s = \sum_{i=0}^{n}(w_i - x_i)^2$

ramp : $f_R(s) = \begin{cases} +1 & \text{if cs} > +1 \\ cs & \text{if } |\text{cs}| \leq 1 \\ -1 & \text{if cs} < \text{-1} \end{cases}$, where $s = \sum_{i=1}^{n} w_i x_i - \theta$ and $c > 0$

In above equations, c is activation gain. Note that most of the activation functions produce output in the range of $[0, 1]$ for binary signals, and $[-1, 1]$ for bipolar signals.

This dissertation uses two types of threshold functions: *binary hardlimiter* and *bipolar hardlimiter*. Their simplicity allows simple and efficient hardware implementation of such threshold functions.

1.1.3 Types of artificial neural networks and their computational capabilities

ANN can be mainly classified into three basic categories : *feedforward networks*, *feedback networks* and *recurrent networks* [28, 56, 131] according to their architectures, functionalities, and signal propagation direction of their connections. The output of a feedforward network is a function of current input, and its connections are unidirectional. The output of a feedback network is a function of current input (and past inputs in some cases), and its connections are not necessarily unidirectional. The output of a recurrent network is a function of current and past inputs, and its connections are unidirectional. Architecturally, a recurrent network is a feedforward networks with recurrent connections, but it is a feedback network functionally. Since the output of both feedback networks and recurrent networks can be a function of past inputs, and thus they are suitable for sequence processing. Mathematically, the computing of feedforward networks approximates a function mapping, and that of feedback networks approximates finite state machines, pushdown automata, or Turing machines.

Typically, a feedforward network and a recurrent network has a layer of input neurons to receive input, a layer of output neurons to produce output, and often layers of hidden neurons to extend the computing capability of the network. Usually, the neurons of a feedback net-

work are classified into input, hidden, and output neurons functionally but not architecturally. *Perceptrons* [155] and *multi-layer Perceptron* [156] are two examples of feedforward networks. *Elman network* [33] and *Jordan network* [81] are two examples of recurrent networks. *Hopfield networks* [75] and *BAM* [90] two examples of feedback networks.

1.1.4 Implementation of artificial neural networks

Due to the computations required by enormous neural nodes to calculate their thresholded activation and weighted sum on the inputs from their associated input connections in an ANN, ANN systems generally require more intensive computational power but simpler types of computations than current computer systems do. There are many technologies available for implementing ANN, mainly including software simulation which is the most widely used due to the fact that digital computer systems are highly available for writing and testing the simulation programs, electronic hardware (digital VLSI, analog VLSI, hybrid of digital and analog VLSI, etc.) realization which potentially owns both the benefits of high performance and cost-effectiveness currently due to the fact that VLSI provides relatively high performance and is extensively used in current computer systems, optical computing which potentially has the highest performance because it computes at the speed of light, and biological implementation which is biologically closer to biological nervous systems.

1.2 A Brief Review of Artificial Neural Networks

Since the resurgence of research on ANN in 1980s, ANN have attracted much interest of many researchers from various science and engineering disciplines, which is shown by the explosive amount of applications and published technical papers on ANN in 1980s and 1990s. It is beyond the scope of this dissertation to review in detail the rich literature in every research area of ANN. Instead, this section will only briefly review up to late 1980s several representative concepts and landmarks on the common research ground of ANN. Reviewed in much more detail in the related chapters is the literature specific to the research topics covered by this dissertation which explores methods for systematically designing neural architectures for associative memories, database query processing, elementary logical inference, sequence processing,

and syntax analysis. The reference book [189] which provides more than 4000 references is a good source of research material for facilitating a general and in-depth understanding of ANN research.

Two of ANN research problem domains for which few conventional computing solutions exist are

- *associative memories* which are anticipated to provide the same advantageous capability as human memory does and are currently mainly used in the applications of pattern classification based on their capability for best match and partial match, and
- *learning* which is anticipated to be used as an efficient and cost-effective alternative to knowledge engineering for automated knowledge acquisition without intensive programming.

Following brief review on ANN literature mainly proceeds along these two intermingled themes which have driven the development of new ANN architectures, models, and algorithms for information processing. The development of formal mathematical models for ANN can be traced back to early 1940s in the work by McCulloch and Pitts [108], which showed that any logical proposition can be represented by a network of interconnected neurons of two states if enough neurons are provided. The computational capability of McCulloch-Pitts neural networks was proved to be equivalent to Turing machines [183] which are the essential model of symbolic computation and can perform any computation that can be described by a finite program in any general purpose language [26].

In 1949, Hebb proposed the first learning rule for neurons [63]. In late 1950s and early 1960s, Rosenblatt introduced a class of neural networks [155], called *perceptrons*, which can learn to classify patterns through *supervised learning*. Rosenblatt's work helped produce a large amount of research activities in this early ANN research era. In 1969, Minsky and Papert showed in their landmark book *Perceptrons* that the computational power of perceptron's single-layer learning algorithm is only able to solve linearly separable problem but not a large class of other problems [118]. With the misinterpretation of such a result, research funding and interest in ANN drastically dropped in the following 1970s. In the dark ages of the 1970s, the

dedicated and everlasting efforts of Amari [7, 8, 9], Anderson [10], Fukushima [39], Grossberg [51, 52, 53], Kohonen [84, 85], and many other researchers ultimately brought in the renascence of ANN in 1980s.

The tremendous resurgence of ANN research interest in 1980s was mainly due to the invention of Hopfield networks [75] which can serve as content-addressable memory or solve combinatorial optimization problems [76], and the introduction of Backpropagation learning algorithm [156] which overcomes the limitation of perceptron's single-layer learning algorithm in linearly separable problems and can be exploited to train multi-layer perceptron to solve nonlinearly separable problems. Since then, Backpropagation multi-layer perceptron has been successfully applied in a variety of applications and has became the most widely used neural network paradigm. The Backpropagation learning algorithm were independently derived by Werbos [193], Parker [138, 139], and LeCun [98], but its popularity was mainly due to the effort of Rumelhart, McClelland, and the PDP Group. Other representative ANN models in the bright 1980s, to name a few, include Hinton, Sejnowski, and Ackley's Boltzmann machine model [1, 67] which can be used to find the global optimum solution for a given problem; Kohonen's Self-Organizing Feature Map [86] which can be trained without supervision to find the organization of relationships among training patterns; Kosko's BAM [89, 90] which can serve as hetero-associative memory and temporal associative memory; Carpenter and Grossberg's ART networks [16, 17, 18] which can be typically used to cluster training patterns via unsupervised training; Radial Basis Function method [114, 146, 147] which was originally used for function interpolation and was also applied to other applications [128, 149]; Hecht-Nielsen's Counterpropagation network [64] which has both supervised as well as unsupervised training stages and can be trained to perform pattern mapping, data compression and associative recall; Fukushima, Miyake, and Ito's Neocognitron [40, 41] which can be trained with supervision to recognize handwritten characters; and recurrent neural networks [33, 81, 144] which allow recursive processing on input string of variable length. A more detailed taxonomy of most neural network architectures and learning algorithms can be found in [56, 102, 112].

1.3 An Overview of the Dissertation

Artificial neural networks, due to their inherent parallelism, potential for fault tolerance, and adaptation through learning, offer an attractive computational paradigm for a variety of applications in computer science and engineering, artificial intelligence, robotics, and cognitive modeling. Despite the success in the application of ANN to a broad range of numeric tasks in pattern classification, control, function approximation, and system identification, the integration of ANN and symbolic computing is only beginning to be explored [22, 23, 47, 70, 72, 73, 99, 145, 172, 179, 185] and is currently viewed as one of important research goals in massively parallel computing and artificial intelligence [65].

Pattern-directed associative processing relies on associative pattern matching and retrieval, is central to many problem solving paradigms in AI (e.g., knowledge based expert systems, case based reasoning) as well as computer science (e.g., database query processing, information retrieval) [54, 97, 181], and dominates the computational requirements of many applications in AI and computer science [55, 97, 127]. This dissertation proposes methods to systematically design massively parallel architectures for pattern-directed symbol processing using neural associative memories as key components. In particular, we propose neural architectures for content-based as well as address-based data storage and recall, database query processing, elementary logical inference, sequence processing, and syntax analysis. Their potential advantages over conventional serial computer implementations of the same functions are examined in the dissertation.

Chapter 2 proposes an approach for the design of a neural memory which supports both content-based (associative) and address-based data storage and retrieval. The proposed neural associative memory allows efficient access of stored data by way of massively parallel best match, partial match and exact match. When used as a content-addressed memory, the proposed neural memory supports recall from partial input patterns, (sequential) multiple recalls, fault-tolerance, precision control and sorted extraction of all stored memory patterns. When used as an address-based memory, the memory module can provide working space for dynamic representations for symbol processing and shared message-passing among neural network mod-

ules within an integrated neural network system. It also provides for real-time update of memory contents by one-shot learning without interference with other stored patterns.

The pattern matching and retrieval process in the proposed neural associative memory which provides massive communication bandwidth and processing units respectively via its massive connections and nodes to match a given pattern with all stored patterns in parallel within one step can be far more efficient (in terms of computation time) than that in a key-based organization of the sort used in conventional computer systems. Chapter 3 takes advantage of this fact to explore the potential benefits of the proposed neural associative memory in the implementation of efficient, noise-tolerant information retrieval and query module in large database systems.

Since most of current digital computer systems store data using address-based memories which are accessed via shared buses, the retrieval of a desired data item satisfying certain criteria (patterns) from a set of candidate data items stored in the memories is inherently sequential and requires certain data organization, which is manipulated and interpreted by a relatively complex program(s), to provide appropriate performance. Although parallel pattern matching can be achieved by current digital computer systems when the systems are provided with multiple processors and memory buses, it would not be cost effective to dedicate such systems to applications which mainly involve intensive pattern matching. The proposed neural associative memory is a cost effective SIMD computer dedicated to pattern association. Therefore, such SIMD capability of the proposed neural associative memory is further explored for relational database queries. The potential merits of ANN's inherent parallelism and noise-tolerance for database query processing are demonstrated by comparing the estimated performance of the proposed neural architecture with that of other techniques commonly used in conventional computer systems for database query processing.

Chapter 4 explores how neural architectures for binary pattern recognition can be extended for elementary logical inference. The proposed neural assemblies for propositional logic are based on geometrical/mathematical analysis. Logical operations such as `AND` and `OR` are realized by neural assemblies for the recognition of binary subpatterns. It is known that any proposition

(or equivalently a Boolean function) can be represented in DNF, and hence can be realized by a 2-layer neural architecture assembled using the proposed `AND` and `OR` neural assemblies. Since logical `AND`, logical `OR`, as well as DNF representation are essential to logical inference and Boolean functions are basic to many applications in science and engineering, we expect the proposed neural assemblies would find use in the construction of modular neural networks for a variety of applications. For instance, Chapter 5 illustrates their use in an neural architecture for sequence processing.

Chapter 5 proposes methods for systematic design of neural architectures for sequence processing, which are used as building blocks to systematically assemble neural architectures for syntax analysis in Chapter 6. Basically, memories and sequence processing mechanisms (with flow control capability) compose current digital computer systems which are driven by sequences of binary codes which are translated from sequences of symbolic program representation that humans can efficiently and effectively read, write, and reason on. Therefore, a computing system integrated from the proposed neural architectures for memories and sequence processing is expected to possess computation capability corresponding to that of current digital computer systems.

Chapter 6 explores the advantages of ANN's inherent parallelism and associative processing capability in the design of modular neural architectures for syntax analysis using a pre-specified grammar — a prototypical symbol processing task. A more general goal of this chapter is to explore the systematical design of massively parallel architectures for symbol processing using the neural associative memory proposed in Chapter 2 and the neural architectures for sequence processing proposed in Chapter 5 as key components.

Since each component in the proposed neural architectures for syntax analysis computes a well-defined symbolic function, it facilitates the systematic synthesis as well as analysis of the resulting symbolic computation at a fairly abstract (symbolic) level. This facilitates rapid design and test of other provably correct prototypes of modular neural architectures for complex symbolic processing using simpler building blocks by way of recursion, composition of elementary symbolic functions, and data representation manipulated by them. The elementary

symbolic functions are represented in terms of binary mappings which are realized provably correctly by basic neural modules using one-shot learning.

Chapter 6 concludes with a summary of the key contributions of this dissertation.

2 A NEURAL MEMORY FOR CONTENT AS WELL AS ADDRESS-BASED STORAGE AND RECALL

2.1 Introduction

This chapter presents an approach to design of a neural architecture for both associative (content-addressed) and address-based memories. Several interesting properties of the memories are mathematically analyzed in detail such that it is known that by systematically adjusting the node thresholds and connection weights, the same proposed neural architecture can serve as memories with precision control to perform best match, exact match and partial match which are main knowledge retrieval techniques extensively used in numerous artificial intelligence systems [191]. When used as an associative memory, the proposed neural architecture supports recall from partial input patterns, (sequential) multiple recalls and fault tolerance. When used as an address-based memory, the memory can provide working space for dynamic representations for symbol processing and shared message-passing among neural network modules within an integrated neural network system. It also provides for real-time update of memory contents by one-shot learning without interference with other stored patterns.

It is generally agreed that artificial neural networks (ANN) have demonstrated success in *low-level* perceptual tasks (e.g., signal processing, pattern recognition) [62, 93, 111, 113]. However, despite their generality (as computational models) and despite the potential advantages of using them as components in general-purpose artificial intelligence systems which usually involve content-based or memory-based knowledge storing and retrieving [47, 70, 72, 73, 99, 173, 179], detailed design and performance tradeoffs in integrated systems of this sort are yet to be fully understood and working prototypes of such systems are only beginning to be developed. Towards this end, an innovative design and careful analysis of neural associative memories with

emphasis on problems and prospects of integrating them into larger systems that combine the advantages of both traditional symbol processing and neural network approaches to artificial intelligence is needed.

A particular class of neural memories built from threshold logic units (Perceptrons or McCulloch-Pitts neurons) is explored from a geometrical/mathematical perspective in this chapter. This analysis provides mathematical foundations for understanding several interesting properties of such memories including: auto as well as hetero-associative recall from partially specified patterns, (sequential) sorted recall of multiple stored patterns with different degrees of match with an input pattern, incremental learning, fault tolerance, and address-based storage and recall (mimicking the behavior of memories used in conventional digital computers). The mathematical analysis also suggests efficient hardware realizations of such memories. This chapter is organized as follows:

- Section 2.1 reviews associative memory, address-based memory, and key properties of multi-layer Perceptrons which form the basis of the proposed neural memories.
- Section 2.2 develops the theoretical foundations and examines the storage capacity of the proposed binary/bipolar neural memories through an investigation of the spatial distribution and linear separability of vertices in binary/bipolar hypercubes from a geometric perspective.
- Section 2.3 explores several interesting properties of the proposed memory modules including: recall from partially specified input patterns, (sequential) multiple recalls, and fault tolerance by examining and extending the physical meanings of the settings of connection weights and neuron thresholds in the proposed neural memories.
- Section 2.4 concludes with a summary of the chapter and a brief discussion of related researches.

2.1.1 Information retrieval and binary mapping

In general, most classification and information retrieval problems using discrete input/output values can be viewed in terms of a binary random mapping f_I, where f_I is rigidly defined from

a set U of k distinct binary input vectors u_1, ..., u_k of dimension n to a set V of k binary output vectors v_1, ..., v_k of dimension m such that $f_I : U \rightarrow V$ and

$$f_I(u_i) = v_i \text{ for } 1 \leq i \leq k \tag{2.1}$$

Note that f_I is a partial function.

2.1.2 Associative memory (Content-addressed memory)

Since the resurgence of ANN in 1980s, ANN have been applied in many science and engineering disciplines. This is shown by the explosive growth in the number of published technical papers on ANN in 1980s and 1990s. In particular, neural architectures for associative memories have been the subject of considerable research, because of their potential applications in several areas of artificial intelligence, computer science, and cognitive modelling.

The term *associative memory* (AM) or *content-addressed memory* refers to a memory system where recall of a stored pattern is accomplished by providing a noisy or partially specified input pattern. Examples of such memory models include Hopfield networks [75], correlation matrix memories [84], bidirectional associative memories [90], among others [9, 59, 91, 125]. A precise definition of binary/bipolar associative memories follows:

Let $D_H(u, u')$ denote the Hamming distance between binary (bipolar) vectors u and u'. *Hamming distance* is the number of bits that differ between two binary (bipolar) vectors. Suppose we are given a set U of k binary input vectors u_1, ..., u_k of dimension n and a set V of k desired binary output vectors v_1, ..., v_k of dimension m. Then the task is to design an associative memory that can store each of the input-output pattern pairs.

In many applications, it is useful to be able to control the degree of mismatch that is tolerated during information retrieval. This is accomplished by introducing the concept of *precision control* in associative memory as follows: Define $U_i^n(p_i) = \{u | u \in \mathbf{B}^n \ \& \ D_H(u, u_i) \leq p_i\}$, $1 \leq i \leq k$, i.e., $U_i^n(p_i)$ is the set of n-dimensional binary vectors which have Hamming distance less than or equal to p_i away from the given n-dimensional binary vector u_i, where $\mathbf{B}^n$ is the *universe* of n-dimensional binary vectors, and p_i is called *allowable precision level* and is an adjustable integer parameter.

Information retrieval in a binary associative memory can be specified in terms of a binary *associative mapping* $f_A : U^n \rightarrow V$ as follows:

$$f_A(x) = v_i \text{ if } x \in U_i^n(p_i),\ 1 \leq i \leq k \tag{2.2}$$

where $U^n = \cup_{i=1}^{k} U_i^n(p_i) = U_1^n(p_1) \cup U_2^n(p_2) \ldots \cup U_k^n(p_n)$ and conventionally $U_i^n(p_i) \cap U_j^n(p_j) = \emptyset$ for $i \neq j$, $1 \leq i, j \leq k$. For example, if such a memory is used to store and recall uppercase English characters, then $U = V$ and $u_i = v_i$, $1 \leq i \leq 26$. Suppose the allowable precision levels (i.e., all of the p_is) are set equal to (Hamming distance) 4. Then in Figure 2.1, the noisy input patterns 1 and 2 would result in the recall of the stored memory pattern **T**. Multiple recalls are possible in the proposed neural memory when $\exists i \neq j$ such that $U_i^n(p_i) \cap U_j^n(p_j) \neq \emptyset$ in which case f_A is a *one-to-many* mapping. Most conventional associative memory models seldom tackle the problem of multiple recalls.

Note that $f_I \subseteq f_A$ if functions f_I (expression 2.1) and f_A are viewed as sets of input-output ordered pairs of the functions f_I and f_A respectively. That is,

$$f_I \equiv \{(x, f_I(x)) | x \in U\}$$
$$f_A \equiv \{(x, f_A(x)) | x \in U^n\}$$

The partial function f_A may be extended to a full function $\hat{f_A} : \mathbf{B}^n \rightarrow (V \cup \{< 0^m >\})$ for binary associative (information retrieval) memory as follows:

$$\hat{f_A}(x) = \begin{cases} f_A(x) & \text{if } x \in U^n \\ < 0^m > & \text{if } x \in (\mathbf{B}^n - U^n) \end{cases} \tag{2.3}$$

where $< 0^m >$ is the m-dimensional binary vector of all zeros and denotes a value which is *undefined*.

Content-addressed memories can be divided into two categories: *auto-associative memories* (used primarily for reconstructing a pattern from a noisy or partially specified pattern) and *hetero-associative* memories which can be used to store associated pattern pairs so that when an input pattern is provided, the associated pattern is retrieved. The types of pattern associations that can be stored in neural associative memories depend on various factors such as: the choice of neural network architecture, the choice of activation functions computed by the neurons,

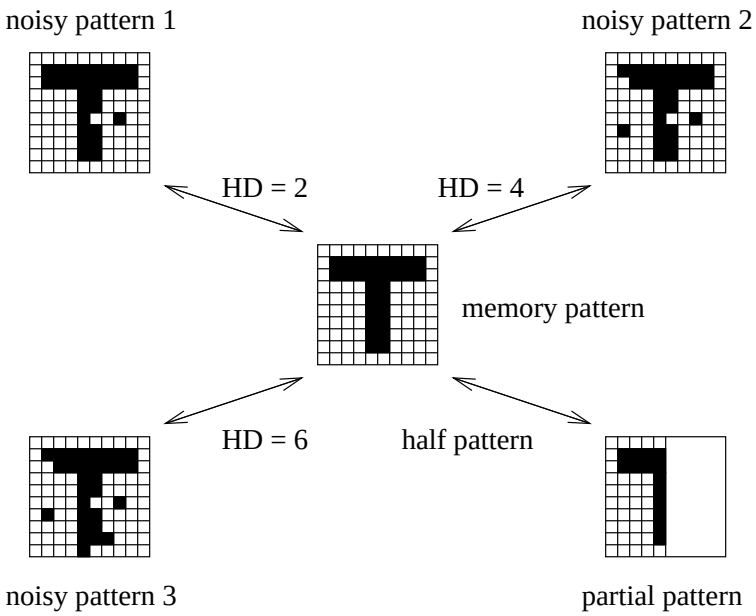

Figure 2.1 Examples of memory pattern, noisy patterns, and partial pattern

and the algorithm used to set up the parameters (thresholds and weights) associated with the neurons and connections. Thus, a *linear associative memory* with n input neurons can store and recall perfectly at most n pattern associations. Similar storage capacity results are known for several content-addressed memory models such as the *Hopfield* network [75, 109], *bi-directional associative memories* [90], correlation matrix memories [84], etc. A variety of associative memory models are discussed in [62, 93]. As already pointed out, many simple content-addressed memory models studied in the literature are incapable of stable storage and recall of associations between arbitrary pairs of patterns (except under certain restricted circumstances). In such models, whether a pattern can be associated with another critically depends on how the two patterns are coded as bit vectors as well as on all the other pattern associations that have already been stored in memory. The ability to reliably store and recall associations between arbitrary patterns is regarded by many to be a prerequisite for higher level cognitive activity (e.g., logical inference) [35]. The associative memory model proposed in this chapter is designed to reliably store and recall associations between arbitrary pairs of patterns.

2.1.3 Address-based memory

Address-based memory is extensively used for storing both data as well as programs in current computer systems. In cognitive models and artificial intelligence programs based on *Von Neumann* model of computation, i.e., models within the so–called symbolic paradigm [126], address-based memory often serves as the working memory (or scratch-pad) for storing intermediate results during the execution of a program. On the surface, storage and recall of patterns using addresses appear to be very different in spirit from the recall of patterns based on their content (as judged by its similarity to a stored pattern). Indeed, many authors have suggested this to be a primary difference between neural networks (or connectionist models) and traditional artificial intelligence systems. However, this perceived difference is rather superficial given the demonstrable Turing-equivalence of sufficiently powerful neural network models [69, 70]. Therefore, it is rather straightforward to design neural memories capable of address-based storage and recall of patterns as the following discussion illustrates.

A mathematical model for information retrieval in address-based memory can be formulated in terms of a binary random mapping f_I (expression 2.1) by extending the partial function f_I to a full function $\hat{f}_I : \mathbf{B}^n \rightarrow (V \cup \{<0^m>\})$ for address-based (information retrieval) memory as follows:

$$\hat{f}_I(x) = \begin{cases} f_I(x) & \text{if } x \in U \\ <0^m> & \text{if } x \in (\mathbf{B}^n - U) \end{cases} \tag{2.4}$$

$\hat{f}_I$ maps from the set of n-bit binary addresses to the set of m-bit binary values. The retrieved value (or content of a memory address) is undefined if no pattern has been stored at the corresponding address.

It is well known (in the literature on the design of memory systems for digital computers) that this approach to address-based memory design is not necessarily the most efficient for large address spaces. In this case, hierarchical memory organization using multiple levels of address decoding and multiple memory modules of the type specified above is a more practical alternative [171].

2.1.4 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 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 *binary hardlimiter* function, where

$$f_H(x) = \begin{cases} 1 & \text{if x} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

It is well known that such a 1-layer Perceptron can implement only linearly separable functions from $\mathbf{R}^n$ to $\{0,1\}^m$ [118]. We can see the connection weight vector $w_i =< w_{i1}, ..., w_{in} >^T$ and the node threshold θ_i as defining a linear hyperplane H_i which partitions the n-dimensional pattern space into two half-spaces, 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 as well as every output neuron). In this chapter, we use 2-layer Perceptron in which each hidden neuron uses binary hardlimiter function f_H as activation function. The output of output neuron i is given by $y_i = f(\sum_{l=1}^{k} w_{il}z_l - 1)$; where z_l is the output of hidden neuron l, f is binary hardlimiter function f_H in the model using binary output, and f is bipolar hardlimiter function $\breve{f}_H$ in the model using bipolar output. (The thresholds of all output neurons are set to 1). The *bipolar hardlimiter* function $\breve{f}_H$ is defined as

$$\breve{f}_H(x) = \begin{cases} 1 & \text{if x} \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.6}$$

2.2 Multi-layer Perceptrons as Neural Memories

This section describes the synthesis of a binary address-based memory or a binary associative memory using a 2-layer Perceptron. The binary address-based memory has a storage capacity of $N = 2^n$ while the binary associative memory has a storage capacity $N = \lfloor 2^n / \sum_{i=0}^{p} C(n,i) \rfloor$, where n is the number of input neurons and p is the adjustable precision

level (allowable noise level) measured in terms of Hamming distance. A hidden neuron is used for each stored associative pair of input and output patterns. The numbers of input and output neurons are fixed in these models. In the case of associative memory, this amounts to fixing the dimensionality of input and output patterns; while in the address based memory, it is tantamount to fixing the maximum size of the address space and the dimensionality of the patterns stored in memory.

2.2.1 The application of linear separability of binary vertices in pattern classification

Note that every n-dimensional binary vector is a binary vertex of an n-dimensional hypercube. Hereafter, we will use the terms *binary vertex* and *binary vector* interchangeably. The following theorem and its proof facilitate the systematic synthesis of the proposed neural memories.

Theorem 2.1: Let u be a binary vector of dimension n, i.e., $u =< u_1, ..., u_n >^T$ where $u_i \in \{0, 1\}$ for $1 \leq i \leq n$. Let $\overline{u} =< \overline{u}_1, ..., \overline{u}_n >^T$ be the complement of the binary vector u. That is, $u_i + \overline{u}_i = 1$ for $1 \leq i \leq n$. Let $u - \overline{u} = u^{ref_u} =< u_1^{ref_u}, ..., u_n^{ref_u} >^T$. Note that $u_i^{ref_u} \in \{1, -1\}$ for $1 \leq i \leq n$. Let us call u^{ref_u} the *reference vector*. Let S_p^u be the set of n-dimensional binary vertices which are at a Hamming distance p away from vertex u, $0 \leq p \leq n$. Then every binary vertex $x \in S_p^u$ falls on an n-dimensional linear hyperplane $H_p^{n,u}$ which is perpendicular to the reference vector u^{ref_u}. Furthermore, if $H^{n,u} = \{H_p^{n,u}; 0 \leq p \leq n\}$, the n-dimensional linear hyperplanes in $H^{n,u}$ are mutually parallel.

Proof: Let x be a binary vertex in S_p^u, $x - \overline{u} = x^{ref_u} =< x_1^{ref_u}, ..., x_n^{ref_u} >^T$ and l_x^u be the length of the projection of x^{ref_u} onto the reference vector u^{ref_u}. Note that $x_i^{ref_u} = 0$ or 1 if $u_i^{ref_u} = 1$, and $x_i^{ref_u} = 0$ or -1 if $u_i^{ref_u} = -1$ for $1 \leq i \leq n$. Note also that there are p components $x_i^{ref_u}$ of x^{ref_u} such that $x_i^{ref_u} = 0$ and $(n - p)$ components $x_j^{ref_u}$ of x^{ref_u} such that $x_j^{ref_u} = 1$ *or* -1, where $1 \leq i, j \leq n$. Let $\| \cdot \|$ denote the length of a vector. Then

$$l_x^u = \frac{1}{\|u^{ref_u}\|}(u^{ref_u})^T x^{ref_u} \tag{2.7}$$

$$= \frac{1}{\|u^{ref_u}\|}\sum_{i=1}^{n} u_i^{ref_u} x_i^{ref_u} \tag{2.8}$$

$$= \frac{1}{\|u^{ref_u}\|}\Big(\sum_{x_j^{ref_u}=1 or -1}^{n-p} u_j^{ref_u} x_j^{ref_u} + \sum_{x_i^{ref_u}=0}^{p} u_i^{ref_u} x_i^{ref_u}\Big) \tag{2.9}$$

$$= \frac{1}{\|u^{ref_u}\|}(n-p) \tag{2.10}$$

$$= \frac{1}{\sqrt{n}}(n-p) \tag{2.11}$$

Thus, $\forall x \in S_p^u$, the length of the projection of x^{ref_u} onto the reference vector u^{ref_u} is $(n-p)/\sqrt{n}$. That is, all binary vertices in S_p^u lie on the same n-dimensional linear hyperplane $H_p^{n,u}$ which is perpendicular to the reference vector u^{ref_u} and located at a distance of $(n-p)/\sqrt{n}$ from the vertex $\overline{u}$, that is, a distance $p/\sqrt{n}$ to vertex u. Hence, every hyperplane $H_p^{n,u} \in H^{n,u}$, $0 \le p \le n$, is parallel to every other hyperplane in $H^{n,u}$. There are n+1 such mutually parallel linear hyperplanes $H_p^{n,u}$'s (Figure 2.2). Among them, u is on $H_0^{n,u}$ and $\overline{u}$ is on $H_n^{n,u}$. The hyperplanes have same normal vector $(u-\overline{u})/\sqrt{n}$. ..... $\Diamond$

The expression defining the n-dimensional linear hyperplane $H_p^{n,u}$, $0 \le p \le n$, is

$$H_p^{n,u} \equiv (\sum_{i=1}^{n}(2u_i - 1)x_i) - (\sum_{i=1}^{n} u_i - p) = 0 \tag{2.12}$$

which can be derived as follows. Let x be a binary vertex on hyperplane $H_p^{n,u}$, where $0 \le p \le n$. From expressions 2.7 and 2.11, we have:

$$l_x^u = \frac{1}{\|u^{ref_u}\|}(u^{ref_u})^T x^{ref_u} = \frac{1}{\sqrt{n}}(u-\overline{u})^T(x-\overline{u}) = \frac{1}{\sqrt{n}}(n-p) \tag{2.13}$$

Thus,

$$(u-\overline{u})^T(x-\overline{u}) = (n-p) \tag{2.14}$$

So the defining expression of the hyperplane $H_p^{n,u}$, $0 \le p \le n$, is given by:

$$H_p^{n,u} \equiv (u-\overline{u})^T(x-\overline{u}) = (n-p) \tag{2.15}$$

$$\equiv (u-\overline{u})^T x - (u-\overline{u})^T\overline{u} - (n-p) = 0, \quad note \;\; u^T\overline{u} = 0 \tag{2.16}$$

$$\equiv (u-\overline{u})^T x - (n - \|\overline{u}\|^2 - p) = 0, \quad note \;\; \|u\|^2 + \|\overline{u}\|^2 = n \tag{2.17}$$

$$\equiv (u-\overline{u})^T x - (\|u\|^2 - p) = 0 \tag{2.18}$$

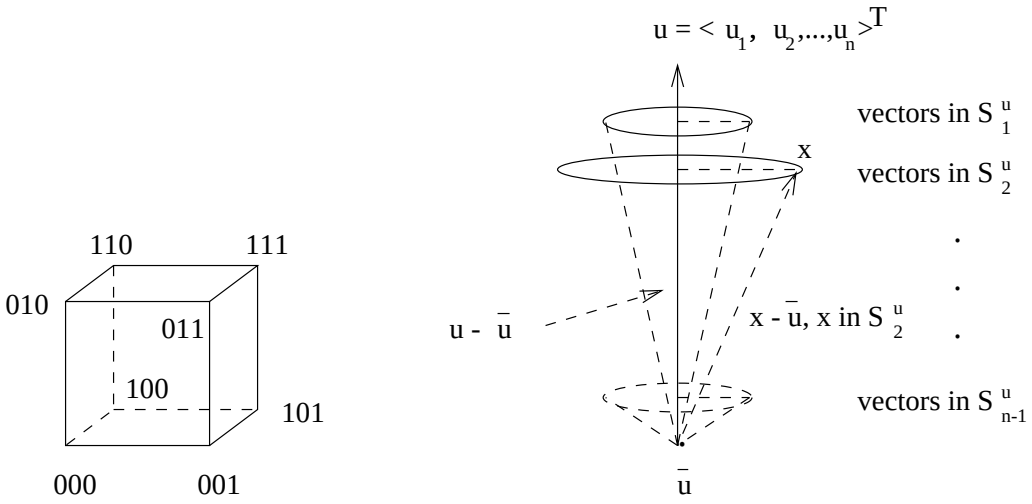

Figure 2.2 The spatial distribution of a 3-dimensional and an n-dimensional binary hypercubes

$$\equiv \quad (u_1 - \overline{u}_1)x_1 + ... + (u_n - \overline{u}_n)x_n - (\|u\|^2 - p) = 0 \tag{2.19}$$

$$\equiv \quad (2u_1 - 1)x_1 + ... + (2u_n - 1)x_n - (\|u\|^2 - p) = 0 \tag{2.20}$$

$$\equiv \quad (\sum_{i=1}^{n}(2u_i - 1)x_i) - (\|u\|^2 - p) = 0 \tag{2.21}$$

$$\equiv \quad (\sum_{i=1}^{n}(2u_i - 1)x_i) - (\sum_{i=1}^{n} u_i - p) = 0 \tag{2.22}$$

Note that $\|u\|^2 = \sum_{i=1}^{n} u_i$ since u is a binary vector. From above, it is known that the expressions defining the n+1 n-dimensional mutually parallel linear hyperplanes $H_p^{n,u}$'s, $0 \leq p \leq n$, have same coefficients but different constant terms. Every hyperplane $H_p^{n,u}$, where $0 \leq p \leq n$, can serve as a linear separating hyperplane to partition all n-dimensional binary vertices into two sets. Such a linear separating hyperplane can be efficiently implemented for 2-class pattern classification by a 1-layer Perceptron with one output neuron. The output neuron has a threshold of $\sum_{i=1}^{n} u_i - p$ and the connection weight on the link from input neuron i is given by $2u_i - 1 (= u_i - \overline{u}_i)$ for $1 \leq i \leq n$, where n is the number of input neurons.

When the separating hyperplane $H_p^{n,u}$ is realized by a 1-layer, 1-output Perceptron, the value of $2u_i - 1$ is either 1 or -1, x_i is either 1 or 0, and $(2u_i - 1)x_i$ can therefore be 1, 0 or -1; and $\sum_{i=1}^{n} u_i$ is integer. Also note that the maximum activation of the 1-layer Perceptron

is p and minimum value is $-(n-p)$; since the separating hyperplane $H_p^{n,u}$ is defined as $(u-\overline{u})(x-\overline{u})^T-(n-p)=0$, the maximum value of $(u-\overline{u})(x-\overline{u})^T$ is n when $x=u$, and the minimum value of $(u-\overline{u})(x-\overline{u})^T$ is 0 when $x=\overline{u}$.

2.2.2 Best match: pattern classification with precision control

Since each binary vertex of dimension n on hyperplane $H_p^{n,u}$ is Hamming distance p away from vertex u, there are $N_p = C(n,p) = \frac{n!}{(n-p)!p!}$ such binary vertices of dimension n on hyperplane $H_p^{n,u}$, where $0 \le p \le n$. The *separating hyperplane* $H_p^{n,u}$ partitions all the binary vertices of a binary n-hypercube into two sets. One set contains $N_A = \sum_{i=0}^{p} N_i = \sum_{i=0}^{p} C(n,i)$ binary vertices that are at a Hamming distance less than or equal to p away from vertex u, and the other contains $N_B = \sum_{i=p+1}^{n} N_i = \sum_{i=p+1}^{n} C(n,i)$ binary vertices that are at a Hamming distance more than p away from vertex u. Let us call the former partition the *associative partition* (denoted by α_p^u) of u, the vertex u the *center* of that associative partition, and p the *radius* of the associative partition. Note that both $H_p^{n,u}$ and α_p^u are defined by the given binary exemplar pattern u and its precision level p.

In theory, an n-hypercube can be almost equally partitioned by $N = \lfloor 2^n / \sum_{i=0}^{p} C(n,i) \rfloor$ such associative partitions as α_p^u with each associative partition containing $\sum_{i=0}^{p} C(n,i)$ n-dimensional binary vertices which are all at a Hamming distance less than or equal to p away from their corresponding partition center. The partition centers correspond to the given binary exemplar patterns.

We say that an associative partition $\alpha_{p_i}^{\nu_i}$ is not *isolated* from another associative partition $\alpha_{p_j}^{\nu_j}$ $(i \ne j)$ if $\alpha_{p_i}^{\nu_i} \cap \alpha_{p_j}^{\nu_j} \ne \emptyset$. Thus, if two associative partitions are not isolated from each other, they overlap and as a result, there is at least one binary vector that is a member of both partitions. The separating hyperplanes (or equivalently, associative partitions) can be implemented in a 1-layer Perceptron with N output neurons to recognize N patterns with precision level (allowable noise level) up to Hamming distance p_i for exemplar pattern ν_i, $1 \le i \le N$, provided the precision levels (p_is) are chosen to ensure that each associative partition is *isolated* from every other. When $x \in \alpha_{p_i}^{\nu_i}$ is fed into the 1-layer Perceptron, the

output neuron that corresponds to the separating hyperplane $H_{p_i}^{n,\nu_i}$ is activated to produce an output of 1. In order to ensure that the associative partitions corresponding to two exemplar patterns ν_i and ν_j are isolated from each other, $D_H(\nu_i, \nu_j)$ has to be greater than $(p_i + p_j)$ where p_i and p_j are the allowable precision levels. Otherwise, the associative partitions of ν_i and ν_j would overlap with each other, and when an input pattern x, where $D_H(x, \nu_i) \leq p_i$ and $D_H(x, \nu_j) \leq p_j$, is fed into the 1-layer Perceptron, the output neurons for the two exemplar patterns ν_i and ν_j will produce 1 as their outputs. In this case, the input pattern x cannot be unambiguously classified as it falls in the region of overlap between the associative partitions $\alpha_{p_i}^{\nu_i}$ and $\alpha_{p_j}^{\nu_j}$.

2.2.3 Storage capacity

Suppose input patterns are 10×10 arrays of binary pixels (see Figure 2.1). Then 100 input neurons are required to implement such a 1-layer Perceptron for pattern classification. The number of possible input patterns is $2^{100} \approx 10^{30}$. An output neuron is needed for each distinct exemplar pattern. Table 2.1 shows the corresponding maximal storage capacity of the 1-layer Perceptrons designed for a range of different allowable noise levels. Table 2.1 also suggests that a 1-layer Perceptron with n input neurons has very high storage capacity for classifying binary patterns and that the allowable precision (noise) levels of less than 30% are desirable for reliable classification.

2.2.4 Synthesis of associative and address-based memories

Given a set U of k distinct binary input vectors u_1, ..., u_k of dimension n, where $u_i =< u_{i1}, ..., u_{in} >^T$ and $u_{ig} \in \{0, 1\}$ for $1 \leq i \leq k$ & $1 \leq g \leq n$; and a set V of k desired binary (bipolar) output vectors v_1, ..., v_k of dimension m, where $v_i =< v_{i1}, ..., v_{im} >^T$ and $v_{ih} \in \{0, 1\}$ (or $\{-1, 1\}$) for $1 \leq i \leq k$ & $1 \leq h \leq m$. Assume the Hamming distance between any two binary vectors in U is at least $2p + 1$, where $p \in \mathbf{N}$. This ensures that all associative partitions

Table 2.1 The corresponding maximal storage capacity of a 1-layer Perceptron with 100 input neurons for classifying binary patterns for a range of allowable noise levels

allowable noise	maximal capacity
0%	$N = \lfloor 2^{100} / \sum_{i=0}^{100\times 0} C(100,i) \rfloor \approx 1.0 \times 10^{30}$
10%	$N = \lfloor 2^{100} / \sum_{i=0}^{100\times 0.1} C(100,i) \rfloor \approx 5.0 \times 10^{16}$
20%	$N = \lfloor 2^{100} / \sum_{i=0}^{100\times 0.2} C(100,i) \rfloor \approx 1.4 \times 10^{9}$
30%	$N = \lfloor 2^{100} / \sum_{i=0}^{100\times 0.3} C(100,i) \rfloor \approx 2.4 \times 10^{4}$
40%	$N = \lfloor 2^{100} / \sum_{i=0}^{100\times 0.4} C(100,i) \rfloor \approx 38$
50%	$N = 2$

would be isolated with the precision level being set at p.

We can now design a neural architecture for information retrieval using address-based memory, denoted by function $\hat{f}_I$ (defined by expression 2.4), or associative memory, denoted by function $\hat{f}_A$ (defined by expression 2.3). For this purpose, a memory module of a 2-layer Perceptron can be synthesized using the 1-layer Perceptrons, proposed for pattern classification in Section 2.2.1, as follows:

The *memory module* (using binary input) has n input, k hidden, and m output neurons. For each associative ordered pair (u_i, v_i), where $1 \leq i \leq k$, we create a hidden neuron i with threshold $\sum_{j=1}^{n} u_{ij} - p_i$ (see Figure 2.3), where $p_i \in \mathbf{N}$ and $p_i \leq p$ is the adjustable precision level for that associative pair. The connection weight from input neuron g to hidden neuron i is $2u_{ig} - 1$ $(= u_{ig} - \overline{u}_{ig})$ and that from hidden neuron i to output neuron h is v_{ih}. The threshold for each of the output neurons is set to 1. The activation functions at hidden neurons are binary hardlimiter function f_H. The activation functions at output neurons are binary hardlimiter function f_H (expression 2.5) if the desired output of output neurons is binary. The activation functions at output neurons are bipolar hardlimiter function $\breve{f}_H$ (expression 2.6) if the desired output of output neurons is binary.

Since input is binary, the weights in the 1st-layer connections of the memory module are either 1 or -1. A bit of an input pattern that is wrongly on (with respect to a stored pattern), contributes -1 to the activation of the corresponding hidden neuron and a bit of an input pattern that is rightly on (with respect to a stored pattern) contributes +1 to the activation

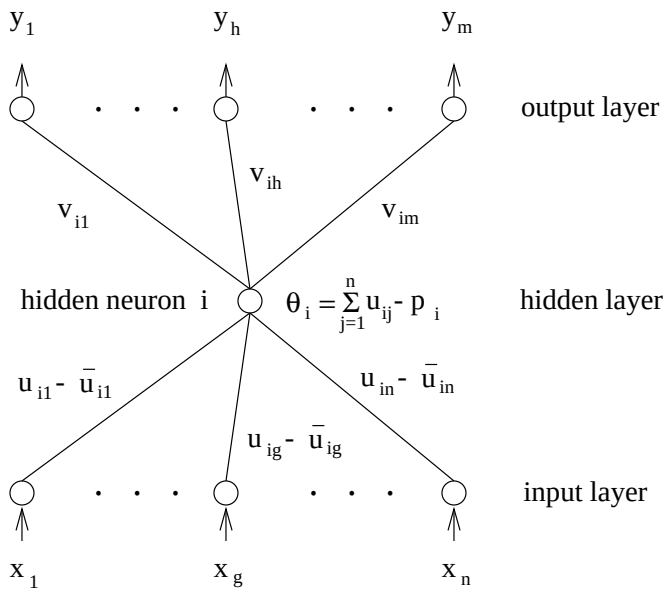

Figure 2.3 The setting of connection weights and hidden node threshold in the proposed neural memory (a 2-layer Perceptron with binary input) for a given associated memory pair

of the corresponding hidden neuron. A bit of an input pattern that is (rightly or wrongly) off (with respect to a stored pattern) contributes 0 to the activation of the corresponding hidden neuron. Each hidden neuron sums up the contributions to its activation from its 1st-layer connections, compares the result with its threshold (which equals the number of 1 in the stored memory pattern minus its desired precision level), and produces output value 1 if its activation exceeds or equals its threshold. If one of the hidden neurons is turned on, one of the stored memory patterns will be *recalled* by that hidden neuron. Note that an input pattern is matched against all the stored memory patterns *in parallel*. If the time delay for computing the activation at a neuron is fixed, the time complexity for such a pattern matching process is $O(1)$. Note that this is attained at the cost of a hidden neuron (and its connections) for each stored association.

Since all the associative partitions are isolated from each other, when the memory module is presented with a binary input vector $x \in \alpha_{p_i}^{u_i}$, only the hidden neuron i produces an output of 1 and the output values from all other hidden neurons are 0. So the value at output neuron

j is v_{ij}, and hence the output binary vector will be $< v_{i1}, ..., v_{im} >^T = v_i$. Since for each memory association pair a hidden neuron is created and its creation or deletion is independent of other stored associative pairs, this particular design of associative memory lends itself to rapid *one-shot incremental learning* with no interference with previously stored associations.

It is also worth pointing out that exactly the same network architecture can be used to realize both associative as well as address-based memory. If p_i is set as 0, $\mid \alpha_{p_i}^{u_i} \mid = 1$ and the memory module functions as an address-based memory when 2^n hidden neurons are used to resolve all possible addresses; and if $1 \leq p_i \leq p$, $\mid \alpha_{p_i}^{u_i} \mid > 1$ and it can be used as an associative memory with adjustable precision control. ($\mid A \mid$ denotes the cardinality of a set A). Address-based memory, extensively used in current computer system, can serve as working space of dynamic representations for symbol processing and shared message-passing space among neural network modules in an integrated neural network system. As working space for symbol manipulation, neural memories have to allow run-time update without learning and do not degrade when the number of stored memory patterns increases. Note that the proposed neural address-based memory has these two properties.

2.2.5 Exact match: binary mapping Perceptron (BMP) module

Let U be a set of k distinct binary input vectors u_1, ..., u_k of dimension n, where $u_i =< u_{i1}, ..., u_{in} >^T$ and $u_{ig} \in \{0, 1\}$ for $1 \leq i \leq k$ & $1 \leq g \leq n$; and V be a set of k desired binary output vectors v_1, ..., v_k of dimension m, where $v_i =< v_{i1}, ..., v_{im} >^T$ and $v_{ih} \in \{0, 1\}$ for $1 \leq i \leq k$ & $1 \leq h \leq m$. Consider a binary mapping function $f_{BMP} : \mathbf{B}^n \rightarrow (V \cup \{<0^m>\})$ defined as follows:

$$f_{BMP}(x) = \begin{cases} v_i & \text{if } x = u_i, 1 \leq i \leq n \\ <0^m> & \text{if } x \in (\mathbf{B}^n - U) \end{cases} \tag{2.23}$$

where $\mathbf{B}^n$ is the n-dimensional binary space. A BMP module for the binary mapping function f_{BMP} can be synthesized using a 2-layer Perceptron as follows: The BMP module (see Figure 2.4) has n input, k hidden and m output neurons. For each binary mapping ordered pair (u_i, v_i), where $1 \leq i \leq k$, we create a hidden neuron i with threshold $\sum_{j=1}^{n} u_{ij}$. The connection weight from input neuron g to hidden neuron i is $2u_{ig} - 1$ $(= u_{ig} - \overline{u}_{ig})$ and that from hidden

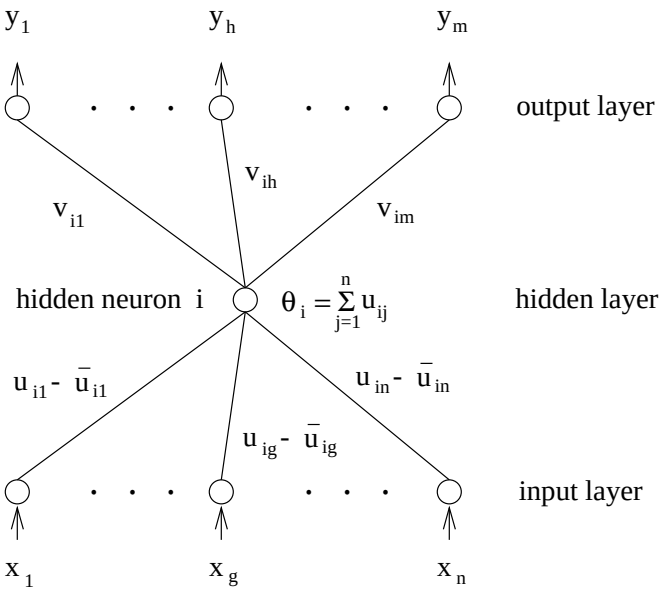

Figure 2.4 The settings of connection weights and hidden node threshold in the proposed BMP module for an associated binary mapping ordered pair

neuron i to output neuron h is v_{ih}. The threshold for each of the output neurons is set to 1. The activation functions at hidden and output neurons are binary hardlimiter function f_H.

Note that for the binary input vector u_i, only the hidden neuron i outputs a 1, and the rest of the hidden neurons output 0. Thus the output of the jth output neuron is v_{ij}, and so the binary output vector is $< v_{i1}, ..., v_{im} >= v_i$. While for an input vector $x \notin U$, no hidden neuron is activated and the output is $<0^m>$.

2.2.6 Conversion between memory models using bipolar and binary inputs

Much of the analysis in previous subsections assumed binary input patterns. It turns out that the use of *bipolar* instead of *binary* inputs simplifies the implementation of the proposed associative memory design especially when recall from partially specified input patterns is desired (see Section 2.3 for details). This subsection explores the relationship between memory models using binary and bipolar inputs and the conversion between the two.

The spatial distribution and geometrical characteristics of bipolar vertices in a bipolar hypercube is very similar to those of binary vertices in a binary hypercube, except that the distance between any two bipolar vertices is 2 times of that between their corresponding binary vertices. Given a bipolar vertex u, there also exist n+1 mutually parallel linear hyperplanes which have similar features described in Theorem 2.1. The expression defining $H_p^{n,u}$ in an n-dimensional bipolar hypercube is

$$H_p^{n,u} \equiv (\sum_{i=1}^{n} u_i x_i) - (n - 2p) = 0 \tag{2.24}$$

which will be derived in the following. All notations here are same as those in Section 2.2.4, with one exception: bipolar vector (vertex) is used in place of binary vector (vertex). In this case $u_i \in \{-1, 1\}$; $u_i + \overline{u}_i = 0$; $u_i^{refu} \in \{2, -2\}$; $x_i^{refu} = 0$ or 2 if $u_i^{refu} = 2$, and $x_i^{refu} = 0$ or -2 if $u_i^{refu} = -2$; there are p components x_i^{refu} of x^{refu} such that $x_i^{refu} = 0$, and $(n - p)$ components x_j^{refu} of x^{refu} such that $x_j^{refu} = 2 \; or \; -2$. Then

$$\begin{aligned}
l_x^u &= \frac{1}{\|u^{refu}\|}(u^{refu})^T x^{refu} && (2.25)\\
&= \frac{1}{\|u^{refu}\|}\sum_{i=1}^{n} u_i^{refu} x_i^{refu} && (2.26)\\
&= \frac{1}{\|u^{refu}\|}(\sum_{x_j^{refu}=2 or -2}^{n-p} u_j^{refu} x_j^{refu} + \sum_{x_i^{refu}=0}^{p} u_i^{refu} x_i^{refu}) && (2.27)\\
&= \frac{1}{\|u^{refu}\|} \times 4(n - p) && (2.28)\\
&= \frac{1}{2 \times \sqrt{n}} \times 4(n - p) && (2.29)\\
&= \frac{2}{\sqrt{n}}(n - p) && (2.30)
\end{aligned}$$

From expressions 2.25 and 2.30 we have:

$$l_x^u = \frac{1}{\|u^{refu}\|}(u^{refu})^T x^{refu} = \frac{1}{2\sqrt{n}}(u - \overline{u})^T(x - \overline{u}) = \frac{2}{\sqrt{n}}(n - p) \tag{2.31}$$

Thus,

$$(u - \overline{u})^T(x - \overline{u}) = 4(n - p) \tag{2.32}$$

So the hyperplane $H_p^{n,u}$ is given by

$$H_p^{n,u} \equiv (u - \overline{u})^T(x - \overline{u}) = 4(n - p) \tag{2.33}$$

$$\equiv \quad (u - \overline{u}^T)x - (u - \overline{u})^T\overline{u} - 4(n-p) = 0 \tag{2.34}$$

$$\equiv \quad (u - \overline{u})^T x - (u^T\overline{u} - \|\overline{u}\|^2 + 4n - 4p) = 0,\ note\ u^T\overline{u} = -n \tag{2.35}$$

$$\equiv \quad (u - \overline{u})^T x - (-n - n + 4n - 4p) = 0 \tag{2.36}$$

$$\equiv \quad (u - \overline{u})^T x - (2n - 4p) = 0 \tag{2.37}$$

$$\equiv \quad (u_1 - \overline{u}_1)x_1 + ... + (u_n - \overline{u}_n)x_n - (2n - 4p) = 0 \tag{2.38}$$

$$\equiv \quad 2u_1x_1 + ... + 2u_nx_n - (2n - 4p) = 0 \tag{2.39}$$

$$\equiv \quad u_1x_1 + ... + u_nx_n - (n - 2p) = 0 \tag{2.40}$$

$$\equiv \quad (\sum_{i=1}^{n} u_ix_i) - (n - 2p) = 0 \tag{2.41}$$

Every hyperplane $H_p^{n,u}$, where $0 \leq p \leq n$, can serve as a linear separating hyperplane to partition all n-dimensional bipolar vertices into two sets. Such a linear separating hyperplane can be efficiently implemented for pattern classification by a 1-layer Perceptron with one output neuron. The output neuron has a threshold of $n - 2p$ and the connection weight on the link from input neuron i is given by u_i for $1 \leq i \leq n$, where n is the number of input neurons.

Since input is bipolar, the connection weight in the 1-layer Perceptron is either 1 or -1. The connection weight of 1 matches the corresponding bit of an input pattern if it is *on* while a connection weight of -1 matches the corresponding bit of an input pattern if it is *off*. A match contributes 1 unit to the activation of the corresponding hidden neuron while a mismatch contributes -1 unit. Each hidden neuron sums up the contributions to its activation from each of its input links, compares it with its threshold and activates the corresponding output neuron if the degree of match (similarity measurement) for the entire pattern exceeds or equals the threshold.

Note that the value passed from each connection is either 1 or -1 , compared to the three values $\{1, 0, -1\}$ in the binary model. This property can further simplify the hardware implementation requirement for a 1-layer Perceptron using bipolar (as opposed to binary) input.

Based on this 1-layer Perceptron and the method described in previous subsections for setting the weights of the second layer connections, the synthesis of a memory module (using

bipolar input, see Figure 2.5) of 2-layer Perceptron is rather straightforward given a set of desired pattern associations.

The *memory module* (using bipolar input) has n input, k hidden, and m output neurons. For each associative ordered pair (u_i, v_i), where $1 \leq i \leq k$, we create a hidden neuron i with threshold $n - 2p_i$, where $p_i \in \mathbf{N}$ and $p_i \leq n$ is the adjustable precision level for that associative pair. The connection weight from input neuron g to hidden neuron i is u_{ig} and that from hidden neuron i to output neuron h is v_{ih}. The threshold for each of the output neurons is set to 1. The activation functions at hidden neurons are binary hardlimiter function f_H. The activation functions at output neurons are binary hardlimiter function f_H if the desired output of output neurons is binary. The activation functions at output neurons are bipolar hardlimiter function $\breve{f}_H$ if the desired output of output neurons is binary.

It is worth pointing out that the bipolar associative neural memory model derived here turns out to be exactly equivalent to the memory model proposed by [59] which uses real-value neuron thresholds and proves the effectiveness of the bipolar memory model by algebra based on using Hamming distance as difference measurement between input pattern and memory patterns. In this subsection, the spatial distribution and linear separability of bipolar vertices in a bipolar hypercube is examined from a geometrical perspective to locate a set of mutually parallel linear hyperplanes which respectively separate nicely all the bipolar vertices into two sets to facilitate the design of bipolar neural memories. The linear separating hyperplanes can be efficiently implemented in a 1-layer Perceptron with connection weights and neuron thresholds of integer values.

Some notable differences between the binary and bipolar associative memory models developed above are:

- The binary model uses binary hardlimiter as the activation function at both hidden and output neurons, so does the bipolar model if the associated output is binary. If the associated output is bipolar, binary and bipolar hardlimiters (respectively) are used as activation functions at hidden and output neurons.

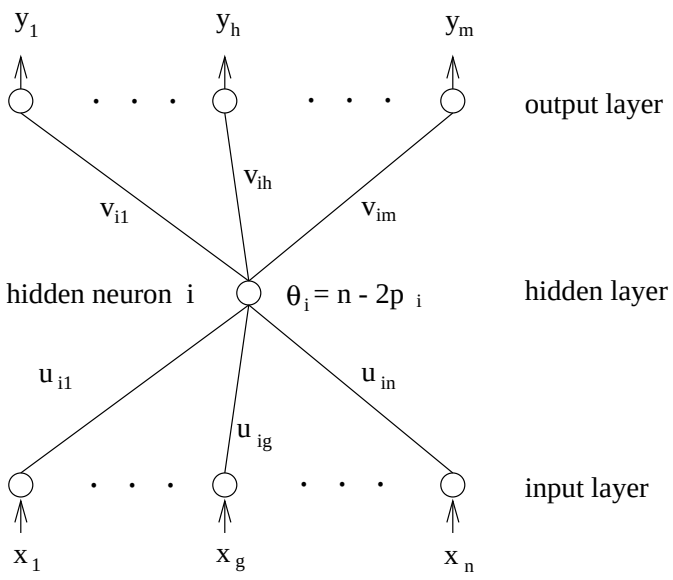

Figure 2.5 The setting of connection weights and hidden node threshold in the proposed neural memory (a 2-layer Perceptron with bipolar input) for a given associated memory pair.

- Threshold setting for a hidden neuron in the binary model equals the number of *on* bits of the corresponding memory pattern minus the desired precision level (measured in Hamming distance), which is not independent of the corresponding memory pattern; whereas threshold setting for a hidden neuron in the bipolar model equals the number of input neurons minus twice the value of the desired precision level, which is independent of all the memory patterns. This has a special advantage when the associative memory is used to recall a pattern based on a partially specified input (see Section 2.3 for details).

2.3 Properties of the Proposed Neural Associative Memory

The following three subsections explore and develop mathematical models for several interesting properties of the proposed bipolar neural associative memory including: recall from partially specified input patterns, (sequential) multiple recalls, and fault tolerance. A set U of k bipolar input vectors u_1, ..., u_k of dimension n and a set V of k desired binary/bipolar output vectors v_1, ..., v_k of dimension m are given. In the discussion that follows, we assume

that the m-dimensional null pattern (a vector of all 0s in the binary case or a vector of all -1s in the bipolar case) is excluded from V.

2.3.1 Partial match: associative recall from a partially specified input

This subsection examines the problem of recall from a partially specified bipolar input pattern. The analysis that follows assumes that the unavailable components of a bipolar partial input pattern have a default value of 0. Thus, a bipolar partial input pattern is *completed* by filling in a 0 for each of the unavailable components (the 0s as a whole also can serve as noise mask or filter). This makes it possible to handle the problem of associative recall from partially specified input pattern in a manner that is analogous to that of recall from completely specified input pattern. The 1st-layer connections of the neural memory module perform similarity measurements on the available components of a partial input pattern, ignore the similarity measurements on the unavailable components, and pass the similarity measurements to the corresponding hidden neurons to decide whether to activate a corresponding hidden neuron.

Let $\dot{u}$ be a partially specified n-dimensional bipolar pattern with the values of some of its components being unknown. Define

- $bits(\dot{u})$: a function which counts the number of components with known values (+1 or -1) of bipolar partial pattern $\dot{u}$
- $pad0(\dot{u})$: a function which pads the unavailable bits of bipolar partial pattern $\dot{u}$ with 0s
- $u \odot v$: a binary predicate which tests whether "*u is a partial pattern of v*", where "*u is a partial pattern of v*" means that the values of available components of u are same as those of their corresponding components in v

For example, let $\dot{u} = <?, -1, 1, 1, ?>$ be a 5-dimensional partial pattern whose first and fifth components have unspecified values. Then $bits(\dot{u}) = 3$, $pad0(\dot{u}) = <0, -1, 1, 1, 0>$ and $\dot{u} \odot <1, -1, 1, 1, 1>$ is *true*. (Note that this definition of a partial pattern respects the positions of the components and does not accommodate shifts or translation).

Let $\dot{D}_H(\dot{u}, \dot{v})$ denote the Hamming distance between two bipolar partial patterns $\dot{u}$ and $\dot{v}$ which have same corresponding unavailable components. If $bits(\dot{u}) = j$, $pad0(\dot{u})$ is a *padded*

j-bit partial pattern derived from partially specified pattern $\dot{u}$. Define $\dot{U}^j_{P_i} = \{\dot{u} \mid bits(\dot{u}) = j \ \& \ \dot{u} \odot u_i\}, 1 \leq j \leq n \ \& \ 1 \leq i \leq k$, i.e., $\dot{U}^j_{P_i}$ is the set of partial patterns, with j available bits, of a bipolar pattern u_i. Define $\ddot{U}^j_{P_i}(p_i) = \{pad0(\ddot{u}) \mid \exists \dot{u},\ \dot{u} \in \dot{U}^j_{P_i} \ \& \ \dot{D}_H(\ddot{u}, \dot{u}) \leq \lfloor j/n \rfloor \times p_i\}$, $1 \leq j \leq n \ \& \ 1 \leq i \leq k$, i.e., $\ddot{U}^j_{P_i}(p_i)$ is the set of padded j-bit partial patterns which are at Hamming distance less than or equal to $\lfloor j/n \rfloor \times p_i$ to any one of the j-bit partial patterns of full pattern u_i.

Practical applications may place limits on the range of usable settings of p_i (allowable noise level) (see Section 2.2 for details). It may also be necessary to limit recall from partial input pattern to cases in which a sufficiently large number of bits in the input pattern, say $j \geq c$, have available values. For instance, when dealing with patterns of 10×10 pixels, we may set $p_i = 0.3 \times 100 = 30$ and require that at least 40% of the pixels be available in the input pattern.

To simplify matters in what follows, we use the same precision level p for each stored pattern. That is, $p_i = p, 1 \leq i \leq k$. However, note that particular applications may require the use of different values of p_i under different circumstances. For example, punctuation symbols and letters of the alphabet may need different values of p_i for successful recognition in recognizing printed English characters.

Let $\ddot{U}^{c\sim n}_{P_i}(p) = \cup^n_{j=c} \ddot{U}^j_{P_i}(p)$; and $\ddot{U}^{c\sim n}_{P}(p) = \cup^k_{i=1} \ddot{U}^{c\sim n}_{P_i}(p)$. Let $f_P : \ddot{U}^{c\sim n}_{P}(p) \rightarrow V$ denote the function of recall from padded bipolar partial pattern. Then f_P is defined as follows:

$$f_P(x) = v_i; \ \text{ if } \ x \in \ddot{U}^{c\sim n}_{P_i}(p), 1 \leq i \leq k \tag{2.42}$$

f_P is a partial function and is extended to a function $\hat{f_P} : \ddot{\mathbf{B}}^{c\sim n} \rightarrow (V \cup \{<(-1)^m>\})$ for recall from padded bipolar partial pattern using associative memory as follows:

$$\hat{f_P}(x) = \begin{cases} f_P(x) & \text{if } x \in \ddot{U}^{c\sim n}_{P}(p) \\ <(-1)^m> & \text{if } x \in (\ddot{\mathbf{B}}^{c\sim n} - \ddot{U}^{c\sim n}_{P}(p)) \end{cases} \tag{2.43}$$

where $\ddot{\mathbf{B}}^{c\sim n}$ is the *universe* of n-dimensional vectors each of whose components is 1, 0, or -1 and which have at least c non-zero components (corresponding to the available bits) and thus at most $(n - c)$ zeros for the unavailable bits (as a result of padding).

It is easy to see that if the 1st-layer connection weights in the bipolar neural memory (described in Section 2.2) were set up using only a part of a complete memory pattern, the

connection weights set for the available components of its partial pattern would be same as those that would have been obtained if the complete memory pattern were used in establishing the weights. The threshold setting for a hidden neuron in the bipolar memory equals the number of input neurons minus twice the value of the desired precision level, which is independent of all stored memory patterns but depends on the dimensionality of input pattern. Hence the bipolar neural memory module designed for recall from a fully specified input pattern can be used for associative recall from a partially specified input pattern by only adjusting the thresholds of the hidden neurons as follows: *multiply the threshold of each hidden neuron by the ratio of the number of available components of a partial input pattern to that of a complete pattern. That is, reduce the threshold of each hidden neuron i from $(n - 2p_i)$ to $(n - 2p_i) \times n_a/n$, where $n_a \leq n$ is the number of the available bits of a partial input pattern.*

Note that p_i is the precision level for memory pattern i in the problem of recall from full pattern and $(n - 2p_i) \times n_a/n = n_a - 2(p_i \times n_a/n)$ is the new threshold for recall from a partial pattern. The expression for the new threshold is similar to that for old threshold. In the new threshold n_a equals the number of available bits of a partial input pattern and $p_i \times n_a/n$ is the new precision level. In the interest of efficiency of a hardware realization, it is desirable to use $\lceil p_i \times n_a/n \rceil$ or $\lfloor p_i \times n_a/n \rfloor$ as the new precision level, where $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ respectively denote the integer ceiling and floor of a real value.

2.3.2 Multiple associative recalls

The memory retrieval process in the neural memory described in Section 2.2 can be viewed as a two-stage process: *identification* and *recall*. During identification of an input pattern, the 1st-layer connections perform similarity measurements and sufficiently activate zero or more hidden neurons so that they produce outputs of 1. The actual choice of hidden neurons to be turned on is a function of the 1st-layer weights, the input pattern, and the threshold settings of the hidden neurons. During recall, if only one hidden neuron is turned on, one of the stored memory patterns will be *recalled* by that hidden neuron along with its associated 2nd-layer connections. Without any additional control, if multiple hidden neurons are enabled, the

corresponding output pattern will be a superposition of the output patterns associated with each of the activated hidden neurons. With the addition of appropriate control circuitry, this behavior can be modified to yield sequential recall of more than one stored pattern. This has a number of practical applications such as information retrieval, database query processing [23, 24] (see Chapter 3), knowledge-based diagnosis systems, etc. This has the effect of searching through memory for patterns that are sufficiently close to a given input pattern and then recall them one after another.

Multiple recalls are possible if some of the associative partitions realized in the memory module are *not isolated* (see Section 2.2 for details). An input pattern (a bipolar vertex) located in a region of overlap among several partitions is close enough to the corresponding partition centers (stored memory patterns) at the same time and hence can turn on more than one hidden neuron. The following explores this phenomenon in more detail.

Define $\dot{U}_i^n(p_i) = \{u \mid u \in \dot{\mathbf{B}}^n \ \& \ D_H(u, u_i) \leq p_i\}$, $1 \leq i \leq k$, where $\dot{\mathbf{B}}^n$ is the universe of n-dimensional bipolar vectors; i.e., $\dot{U}_i^n(p_i)$ to be the set of n-dimensional bipolar vectors which have Hamming distance less than or equal to p_i away from the given n-dimensional bipolar vector u_i, where p_i is a specified precision level. Let $p_i = p, 1 \leq i \leq k$.

Define $f_M : \dot{U}^n(p) \rightarrow (2^V - \emptyset)$ as follows:

$$f_M(x) = \{v_i \mid x \in \dot{U}_i^n(p),\ 1 \leq i \leq k\} \tag{2.44}$$

where $\dot{U}^n(p) = \dot{U}_1^n(p) \cup \dot{U}_2^n(p) \ldots \cup \dot{U}_k^n(p)$, $\dot{U}_i(p) \cap \dot{U}_j(p) \neq \emptyset$ for some $i \neq j$, and 2^V is the *power set* of V (i.e., the set of all subsets of V). The output of f_M is a set of bipolar vectors that correspond to the set of patterns that should be recalled given the bipolar input vector x.

f_M is a partial function and is extended to a full function $\hat{f}_M : \dot{\mathbf{B}}^n \rightarrow (2^V \cup \{<(-1)^m>\} - \emptyset)$ to describe multiple recall in the neural associative memory as follows:

$$\hat{f}_M(x) = \begin{cases} f_M(x) & \text{if } x \in \dot{U}^n(p) \\ \{<(-1)^m>\} & \text{if } x \in (\dot{\mathbf{B}}^n - \dot{U}^n(p)) \end{cases} \tag{2.45}$$

Recall of multiple patterns is likely to be all the more useful when the input pattern is only partially specified. The following extends the mathematical model for multiple recall outlined above to deal with recall from a partially specified bipolar input pattern.

Let $f_{MP} : \ddot{U}_P^{c\sim n}(p) \rightarrow (2^V - \emptyset)$ be a function defined as follows:

$$f_{MP}(x) = \{v_i \mid x \in \ddot{U}_{P_i}^{c\sim n}(p),\ 1 \leq i \leq k\} \tag{2.46}$$

where $\ddot{U}_{P_i}^h(p) \cap \ddot{U}_{P_j}^h(p) \neq \emptyset$ for some h's and $i \neq j$'s, $c \leq h \leq n$ and $1 \leq i,\ j \leq k$. f_{MP} is a partial function and is extended to a function $\hat{f}_{MP} : \ddot{\mathbf{B}}^{c\sim n} \rightarrow (2^V \cup \{< (-1)^m >\} - \emptyset)$ for multiple recalls from padded bipolar partial patterns in associative memory as follows:

$$\hat{f}_{MP}(x) = \begin{cases} f_{MP}(x) & \text{if } x \in \ddot{U}_P^{c\sim n}(p) \\ < (-1)^m > & \text{if } x \in (\ddot{\mathbf{B}}^{c\sim n} - \ddot{U}_P^{c\sim n}(p)) \end{cases} \tag{2.47}$$

Another interesting property of the proposed ANN memory is that it allows *sorted multiple recall* described as follows: If the input pattern is held constant and the thresholds of all hidden neurons are decremented at each time step, then gradually more and more hidden neurons will be turned on. Decrementing the threshold of a hidden neuron results in an enlargement of the corresponding associative partition in a geometrical sense, and hence more and more partitions will overlap at the input vector (vertex) from iteration to iteration. In the absence of any other control circuits, the recalled pattern will be a superposition of the outputs resulting from all of the hidden neurons that are enabled at any time step. However, in many applications, we need different patterns to be recalled individually. This can be accomplished by adding a habituation mechanism that forces a hidden neuron to turn itself off automatically after it has been on for one time step unless a new input pattern is presented. This results in a serialized or sequential recall of patterns in increasing order of dissimilarity (as measured by Hamming distance) from the input pattern. Alternatively, one can perform a *set difference* operation on the hidden neuron outputs from every pair of consecutive time steps before allowing the hidden neurons to influence the 2nd-layer connections and the output neurons. It is rather straightforward to implement such set-theoretic operations using neural networks [25].

As already pointed out, the ability to perform multiple recalls is more likely to be useful when dealing with partially specified input patterns. Such information retrieval applications of practical interest include: database lookup using keywords, diagnosis of diseases or faults from a partially specified set of symptoms or test results, and DNA sequence recognition from available DNA segments.

2.3.3 Fault tolerance

This section discusses the performance of the proposed neural memory in the presence of two basic types of faults — *connection fault* and *neuron fault*. In the discussion that follows, it is assumed that:

- When a connection fails and stops passing a value, it is assumed that *default value 0* is passed from that faulty connection.
- When an input or hidden neuron fails and stops functioning, it is assumed that *default value 0* is passed along each of its outgoing connections.
- When an output neuron fails, it is assumed that *default value* -1 (or 0 for binary output) is produced by that output neuron.

2.3.3.1 Connection fault

First we note that a single fault in a 1st-layer connection has less deleterious effect on the performance of the memory than that caused by a noisy bit in an input pattern. This is because a faulty 1st-layer connection will adversely affect only one of the similarity measurements between the input pattern and the stored memory patterns whereas a noisy bit of an input pattern affects all the similarity measurements.

For example, assume $u_i =< 1, -1, 1, -1, 1 >^T$ and $u_j =< 1, 1, 1, 1, 1 >^T$ are two of the memory patterns stored by hidden neurons i, j and their respective connections in an auto-associative memory. Note that $D_H(u_i, u_j) = 2$. Let w_i^1 denote the weight vector of the 1st-layer connections connected to hidden neuron i. Suppose the precision levels $p_i = p_j = 1$. Then $w_i^1 =<1, -1, 1, -1, 1>^T$, $w_j^1 =<1, 1, 1, 1, 1>^T$, and the thresholds at hidden neurons i and j are $\theta_i = \theta_j = 3$ in the neural memory according to expression 2.41. Suppose a noisy input pattern $x =<1, -1, 1, 1, 1>^T$ is fed into the neural memory module. Then both hidden neurons i and j are activated and as a result, the output is a superposition of memory patterns u_i and u_j. Suppose the connection from the second input neuron to hidden neuron i is faulty, which causes $w_i^1 =<1, 0, 1, -1, 1>^T$ equivalently. Assumes that w_j^1 is unaffected by the connection

fault. When the noisy input pattern x is fed into the neural memory module, the summation values at hidden neuron i and j are -1 and 0 respectively. Only hidden neuron j is activated and memory pattern u_j is recalled.

Since each of the 2nd-layer connections emanating from a hidden unit stores 1 bit of the corresponding stored memory pattern, a faulty 2nd-layer connection corrupts at most 1 bit of the recalled memory pattern. Suppose the connection from hidden neuron j to the third output neuron is faulty. When only hidden neuron j is activated to recall memory pattern u_j, a default value 0 is passed from that faulty connection to the third output neuron under the assumption of connection fault. The recalled output is $<1, 1, -1, 1, 1>^T$ which is one Hamming distance from memory pattern u_j. When only hidden neuron i is activated to recall memory pattern u_i with a connection fault from hidden neuron i to the second output neuron, the recalled output is $<1, -1, 1, -1, 1>^T$ which equals u_i

2.3.3.2 Neuron fault

A fault in one of the input neurons has less of an adverse effect on the performance of the memory than 1 bit of noise in the input pattern. However, it is easy to see that an input neuron fault is more serious than a fault in a single 1st-layer connection. This is because a fault in one of the input neuron adversely affects each of the stored memory patterns. For example, suppose u_i, u_j and x are as before. Let us consider following three cases:

- **Case 1**: one of the first, third, or fifth input neurons is faulty. When x is fed into the neural memory module, the summation values at hidden neurons i and j are both -1. No memory pattern is recalled.
- **Case 2**: the second input neuron is faulty. Then the summation values at hidden neuron i and j are -1 and 1 respectively. The hidden neuron j is activated to recall memory pattern u_j.
- **Case 3**: the fourth input neuron is faulty. Then the summation values at hidden neuron i and j are 1 and -1 respectively. The hidden neuron i is activated to recall memory pattern u_i.

If a hidden neuron is faulty, the memory pattern associated with that hidden neuron can not be recalled. If an output neuron is faulty, the corresponding bit of all recalled memory patterns will have value -1. So the recalled memory patterns having value 1 at that corresponding bit are corrupted by one bit of noise.

2.4 Summary and Discussion

This chapter has discussed the analysis and synthesis of a neural memory for both address-based as well as associative (content-based) storage and recall of patterns. When used as content-addressed memory, the proposed ANN memory allows adjustable precision and sorted extraction of all stored memory patterns, has high potential storage capacity, and exhibits several interesting properties: recall from partial pattern, multiple recall and fault tolerance. It also lends itself to one-shot incremental learning without interference with previously memorized patterns. A detailed mathematical analysis of the properties of the proposed neural memory architecture is presented. Address-based memory can serve as working space of dynamic representations for symbol processing and shared message-passing space among neural network modules in an integrated neural network system. It provides for reliable content modification in real time, a necessary feature for symbol processing applications.

The pattern matching process of the proposed content-addressed memory in which data parallelism is achieved and all memory patterns are compared with input pattern in parallel within one step can be far more efficient (in terms of computation time) than searching for data in a key-based organization of the sort commonly used in conventional computer systems [23]. With the need for real-time response in language translation and with the increased number of users as well as increased use of large networked databases over the Internet, efficient architectures for high-speed table lookup, message routing and database query processing have assumed great practical significance. Extensions of the proposed ANN memory architecture for efficiently handling database queries and syntax analysis are proposed in Chapters 3 and 6 (also see [22, 23]) respectively.

It is worth mentioning that the proposed neural memory supports realization of many-to-

one binary random mappings which is extensively used in the design of digital logic devices such as logic circuitries of AND/OR (or NAND/NOR) gates. The design and optimization of such circuitry has always been one of the key research problems in the VLSI research community and industry. The hardware realization of the proposed neural architecture provides same flexibility as PLA (programmable logic array) and thus higher abstraction level than AND/OR gates for logic functions (many-to-one binary mappings). The anticipated performance of hardware realizations of the proposed memory architecture is evaluated in Section 3.3. If the hardware realization provides for run-time loading of connection weights and neuron thresholds under software control, it provides for an efficient, time-saving, and error-preventing alternative to the implementation of PLA and combinational circuitry of AND/OR gates for logic circuitry.

3 NEURAL ARCHITECTURES FOR INFORMATION RETRIEVAL AND DATABASE QUERY PROCESSING

3.1 Introduction

This chapter explores the application of neural associative memory to efficient implementation of noise-tolerant information retrieval and query module in large database systems. Based on the neural associative memory proposed in Chapter 2, a library query system and a query system for text-based machine-readable lexicon are explored respectively by exploiting the capability of neural associative memory for massively parallel associative pattern matching and retrieval. The performance of the ANN-based database query module is analyzed and compared with other techniques commonly used in current computer systems. The results of this analysis suggest that the proposed ANN design offers an attractive approach for the realization of query modules in large database and knowledge base systems, especially for information retrieval based on partial matches.

Artificial neural networks offer an attractive computational model for a variety of applications in pattern classification, language processing, complex systems modelling, control, optimization, prediction and automated reasoning for a variety of reasons including: potential for massively parallel, high-speed processing, resilience in the presence of faults (failure of components) and noise. Despite a large number of successful applications of ANN in aforementioned areas, their use in complex symbolic computing tasks (including storage and retrieval of records in large databases, and inference in deductive knowledge bases) is only beginning to be explored [21, 22, 23, 24, 47, 72, 99, 179].

Database query entails a process of content-based table lookup (associative search and retrieval) which is used in a wide variety of computing applications. Examples of such lookup

tables include: *routing tables* used in routing of messages in communication networks, *symbol tables* used in compiling computer programs written in high level languages, knowledge bases which store facts and rules in relational form, fact and rule tables used in unification process of logic programming systems, keyword tables (inverted and signature files) used in information retrieval applications [37], and machine-readable lexicons, dictionary as well as varieties of tables used in *memory-based parsing* [82] for natural language processing. In such tables, every table entry is an associated input-output ordered pair. As the number of table entries and the occurrence of partially specified inputs increase, the delay of locating an associative table entry can become a severe bottleneck in large-scale information processing tasks which involve extensive associative table lookup. Therefore, many researchers have explored to augment conventional database systems with subsystems which effectively exploit associative processing to enhance the performance of the systems [30, 101, 121, 135, 157, 162, 196]. Many applications require associative table lookup mechanism or query processing system to be capable of retrieving items based on *partial matches* (some features of the input are noisy or missing) or retrieval of *multiple* records matching the specified query criteria. This capability is computationally rather expensive in many current computer systems. The ANN-based approach to database query processing that is proposed in this chapter exploits the fact that an associative table lookup task can be viewed at an abstract level in terms of *associative pattern matching and retrieval* which can be efficiently realized using neural associative memories. The rest of the chapter is organized as follows:

- The rest of Section 3.1 briefly discusses how to represent symbolic information in terms of binary codings to facilitate symbolic information manipulation on the proposed neural associative memory which operates on bipolar/binary values.

- Section 3.2 explores information retrieval and query processing using neural associative memory. ANN designs are developed respectively for a library query system and a query system for text-based machine-readable lexicon by taking advantage of the capability of the proposed neural associative memory for massively parallel pattern matching and retrieval.

- Section 3.3 compares the performance of the proposed ANN-based query processing system with that of several commonly used techniques.
- Section 3.4 concludes with a summary.

3.1.1 Information retrieval in neural associative memories

Most database systems store (symbolic) data in the form of structured records. When a *query* is made, the database system searches and retrieves records that match the user's query criteria which typically only partially specify the contents of records to be retrieved. Also, there are usually multiple records that match a query (e.g., books written by a particular author in a library or the lexical specifications of the words matching the partially specified input pattern `ma?e` in a machine-readable lexicon, where the symbol `?` means the English letter at that position is unavailable). Thus, query processing in a database can be viewed as an instance of the task of recall of multiple stored patterns given a partial specification of the patterns to be recalled. The proposed neural associative memory which is capable of massively parallel best match, exact match, and partial match and recall of binary (bipolar) patterns can serve to efficiently handle information retrieval and query processing in large database systems.

The proposed neural associative memory operates on binary (bipolar) values. Since humans find it difficult to work with binary codes, we use symbolic representations when the neural associative memory is used for information storage and retrieval. The translation from symbolic representations to binary codings can be done automatically and is not discussed here.

In general, symbolic information retrieval (lookup) from a table can be viewed in terms of a binary random mapping $f_I : U \rightarrow V$, defined in expression 2.1. A binary vector $u_i \in U$ can be used to represent an ordered set of r binary-coded symbols from symbol sets $\Gamma_1, \Gamma_2, \cdots, \Gamma_r$ respectively (i.e., $\exists \alpha_1 \in \Gamma_1, ..., \alpha_r \in \Gamma_r \; s.t. \; u_i = \alpha_1 \cdot \alpha_2 \cdot ... \cdot \alpha_r$, where $\cdot$ denotes the concatenation of two binary codes), and a binary vector $v_i \in V$ can be used to represent an ordered set of t symbols from symbol sets $\Delta_1, \Delta_2, \cdots, \Delta_t$ respectively, where $1 \leq i \leq k$. In the context of Section 3.2, every Γ_i denotes the set of ASCII-coded English letters, r is the length (in

number of English letters) of the input, $t = 1$, and Δ_1 is the set of M-bit record pointers, where $1 \leq i \leq r$. Let $|U| = |\Gamma_1||\Gamma_2| \cdots |\Gamma_r|$. Then, f_I defines a symbolic mapping function $f_S : \Gamma_1 \times \Gamma_2 \cdots \times \Gamma_r \rightarrow \Delta_1 \times \Delta_2 \cdots \times \Delta_t$. In this case, the I/O mapping of symbolic function f_S (information retrieval from a symbolic table, given a query criterion) can be viewed in terms of the binary (bipolar) mapping operations of f_I which is realized by the proposed neural associative memory.

3.2 Query Processing Using Neural Associative Memories

This section describes the use of the neural associative memory described in Chapter 2 to implement high-speed database query systems. An ANN-based library query system and an ANN-based query system for a text-based machine-readable lexicon for natural language processing (NLP) are presented respectively to illustrate the key concepts. As the quantity of entries (records) of a database increases, the cost of locating an entry can become a significant cost for real-time, large-scale machine processing of text and for a library system with huge stored volumes and large users. For example, the library at Iowa State University has over 2 million volumes, and the number of words a native English speaker knows is estimated to be between 50,000 and 250,000 [4]. In the proposed ANN-based query systems, such a cost can be reduced significantly by taking advantage of the capability of the proposed neural associative memory for massively parallel associative pattern matching and retrieval.

3.2.1 Realization of lexical access for a machine-readable lexicon using a neural associative memory

In the analysis, interpretation, and generation of natural languages, the lexicon is one of the central components of many NLP applications. Basically, the lexical specification for a word in a lexicon contains phonological, morpho-syntactic, syntactic, semantic, and other fields [58]. Each field may contain several sub-fields. In a lexical database which realizes a machine-readable lexicon for real-time NLP, the lengths of the fields and sub-fields are usually fixed to allow efficient random access to them. This is where a computational lexicon is distinguished from a dictionary in which the format of lexical entries is mostly irregular and hence the access

of the lexical fields for a word (lexeme) is sequential. Typically, a dictionary contains much free text including definitions, examples, cross-reference words, and others.

Generally, there are two basic conceptions about the form of the items which serve as access keys in a lexicon. One is *minimal listing hypothesis* [15] which only lists lexemes and results in a *root lexicon*. A lexeme may have several variants, e.g., in English, the words: `produces, produced, producing, producer, productive` and `production` are variants of the lexeme `produce`, and the words: `shorter, shortest` and `shortly` are variants of the lexeme `short`. The other is *full listing hypothesis* which lists all possible words of a language and results in a *full-form lexicon*. A root lexicon is more compact and requires a rule system to process the variants of lexemes, while a full-form lexicon is more computationally efficient in terms of lexical access and more user-friendly in terms of lexicon editing and extension [58]. Therefore, a hybrid of the two conceptions is often adopted in many computational lexicon applications. In the following, the term *access key* is used to stand for either word or lexeme in a computational lexicon no matter whether it is a root or full-form lexicon.

There are several models of lexical access in a computational lexicon. Our ANN-based query system for NLP lexicon is based on the search model of lexical access (indirect access) [36, 58]. In such a model, a text-based computational lexicon which associates every access key with its lexical specification contains two organizations: one is called *master file* which stores entries of lexical specifications, and the other is called *access file* which consists of pairs of (<access key>, <lexical pointer>). The access keys are organized to allow location of desired access keys and their associated lexical pointers efficiently. The lexical pointers point to the lexical specifications of their corresponding entries in the master file. The process of lexical access in the search model is similar to that of locating a book in a library. To locate a book from a collection of shelves (the master file) in a library, the book catalog (the access file) is searched using author name(s) and/or book title to find the call number (a pointer indicating the location) of a desired book.

A noise-tolerant neural associative memory which can efficiently support the process of search and retrieval of desired lexical pointers for a text-based machine-readable English lexicon

is designed as follows. Suppose English letters of the lexicon are represented using 8-bit ASCII codes (extended to 8 bits by padding each 7-bit ASCII code with a leading *0*). Assume the maximal length of an English word is L letters. Since each letter is represented by an 8-bit ASCII code, $8L$ input neurons are used in the ANN memory. Each binary bit x_b of the ASCII input is converted into a bipolar bit x_p by expression $x_p = 2x_b - 1$ before it is fed into the ANN memory to execute a query. (This is motivated by the relative efficiency of the hardware implementations of binary and bipolar neural associative memories – see Chapter 2 for details). Let the output (the lexical pointer) be represented as an M-bit binary vector which can access at most 2^M lexical specifications in the lexical database. So, the ANN memory uses M output neurons.

For every associative ordered pair of an access key and a lexical pointer, a hidden neuron is used in the ANN memory. Suppose there are k such pairs. Then, the ANN memory uses k hidden neurons. Every access key is represented by padding its corresponding English word with trailing `space`s and each binary bit x_b of every access key is converted into a bipolar bit x_p by expression $x_p = 2x_b - 1$ to be stored in the ANN memory. For example, if an English word has j letters ($j \leq L$), then the first j letters of its corresponding access key are from the English word and the last $L - j$ letters of the access key are all `space`s. The reason for such padding will become obvious in the coming examples. The ASCII code for the special symbol `space` is $20_{16} = 0010\ 0000_2$. During storage of an associated pair, the connection weights are set as explained in Section 2.2.6. Note that the input and output of an associated pair are represented in bipolar and binary values respectively. During recall, the thresholds of hidden neurons are adjusted for each query as outlined in Section 2.3.1 (where for each query, the value of n_a can be set either by centralized check on the number of letters of the input access key, or distributed circuitry embedded in input neurons). The precision level p is set at 0 for this associative ANN memory.

3.2.1.1 Examples of query processing in the neural lexicon

The following examples illustrate how the proposed ANN memory for NLP lexicon retrieves desired lexical pointers by processing a query which may contain a partially specified input (target access key).

- **Example 1** (exact match) : Suppose the lexical pointer of the word `product` is to be retrieved from the ANN memory. Then, the first 7 letters of the target access key to be searched are `p, r, o, d, u, c` and `t`, and the last $L-7$ letters are `space`s. In this case, no letter of the target access key is unavailable. Therefore, in the ANN memory, the threshold set at all hidden neurons is $L \times 8 = 8L$. Suppose a hidden neuron i is used for the association of this access key and its associated lexical pointer. When the target access key is presented to the ANN memory, only hidden neuron i has net input of 0 and other hidden neurons have net input less than 0 (see Section 2.2 for details). So, hidden neuron i is activated to recall the desired lexical pointer using the weights on the 2nd-layer connections associated with hidden neuron i.

- **Example 2** (prefix match) : Suppose the lexical pointer(s) of the word(s) matching the pattern `product`$*$ is to be retrieved from the ANN memory, where the symbol $*$ means the trailing English letters starting from that position are unavailable. In this case, the last $L-7$ letters of the target access key are viewed as unavailable, only the first 7 letters are available, and its first 7 letters are `p, r, o, d, u, c` and `t`. Therefore, in the ANN memory, only the first $7 \times 8 = 56$ input neurons have input value either 1 or -1, the other input neurons are fed with 0, and the threshold set at all hidden neurons is $7 \times 8 = 56$. Suppose, in the lexicon, `product, production, productive, productively, productiveness` and `productivity` are the words the first 7 letters of which match the pattern `product`$*$. In this case, six hidden neurons are used for the associations of these six access keys and their lexical pointers respectively in the ANN memory. When the partially specified target access key is presented to the ANN memory, only these six hidden neurons have net input of 0 and other hidden neurons have net input less than 0. So, these six hidden neurons get activated one at a time to sequentially recall

the associated lexical pointers using the weights on the 2nd-layer connections respectively associated with these six hidden neurons.

- **Example 3** (partial match) : Suppose the lexical pointer(s) of a noisy 7-letter word `pro??ct` is to be retrieved from the ANN memory, where the symbol `?` means the English letter at that position is unavailable. In this case, 2 of the letters (the $4th$ and $5th$ letters) of the target access key are viewed as unavailable, its first 3, $6th$ and $7th$ letters are `p, r, o, c` and `t` respectively, and the last $L-7$ letters are `space`s. Therefore, in the ANN memory, the input neurons representing the $4th$ and $5th$ input letters are fed with 0, other input neurons have input value either 1 or -1, and the threshold set at all hidden neurons is $(L-2)\times 8 = 8(L-2)$. Suppose, in the lexicon, `product`, `project`, and `protect` are the only 7-letter words which match the pattern `pro??ct`. Therefore, three hidden neurons are used for the associations of these three access keys and their lexical pointers respectively in the ANN memory. When the partially specified target access key is presented to the ANN memory, only these three hidden neurons have net input of 0 and other hidden neurons have net input less than 0. So, these three hidden neurons get activated one at a time to sequentially recall the associated lexical pointers using the weights on the 2nd-layer connections respectively associated with them.

The large number of hidden neurons in such an ANN module poses a problem for hardware realization because of the large fan-out for input neurons and large fan-in for output neurons. One solution to this problem is to divide the whole module into several sub-modules which contain same number of input, hidden, and output neurons. These sub-modules are linked together by shared input and output bus (see Figure 3.1). Such a bus topology also makes it possible to easily expand the size of the ANN memory. The *1*-dimensional array structure shown in Figure 3.1 can be easily extended to *2* or *3*-dimensional array structures.

3.2.2 Realization of a library query system using a neural associative memory

A neural associative memory that can be used to support a library system queried by name can be designed as follows: Suppose the input is a name (provided in a format with last name

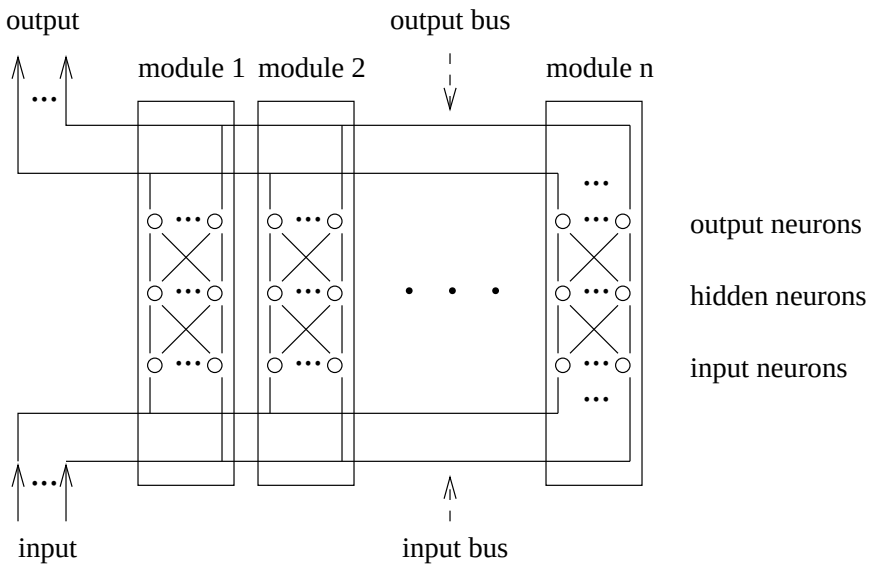

Figure 3.1 A modular design of the proposed ANN memory for easy expansion. This 1-dimensional array structure can be easily extended to *2* or *3*-dimensional array structures.

followed by first name) of an author, and characters that appear in the name are represented using 8-bit ASCII codes. Assume the length of both last and first name are truncated to at most L characters each. Since each ASCII code consists of 8 binary bits, $16L$ input neurons are used in the ANN memory. The first $8L$ input neurons are for last name and the last $8L$ neurons for first name. Each binary bit x_b of the ASCII input is converted into a bipolar bit x_p by expression $x_p = 2x_b - 1$ before it is fed into the ANN memory module for database queries.

Let output be a M-dimensional binary vector pointing to a record in the library database that contains information about a volume (or the binary vector can encode information about a volume directly). The output binary vector in turn can therefore be used to locate the title, author, call number and other relevant information about a volume. Using M output neurons, we can access at most 2^M records from the library database. Each hidden neuron in the associative memory module is used to realize an ordered pair associating an author's name with an M-bit pointer that points to a record which contains information about a corresponding volume. The last and first names of an author of an associated pair are represented by padding

the names with trailing `space`s and each binary bit x_b of the padded names is converted into a bipolar bit x_p by expression $x_p = 2x_b - 1$ to be stored in the ANN memory. For example, if *Smith John* is the name part of an associated pair, the first 5 letters for the last name part of the associated pair are `Smith` and the other $L - 5$ letters are `space`s, and the first 4 letters for the first name part of the associated pair are `John` and the other $L - 4$ letters are `space`s. During storage of an associated pair, the connection weights are set as explained in Section 2.2.6. Note that the input and output of an associated pair are represented in bipolar and binary values respectively. During recall, the thresholds of hidden neurons are adjusted for each query as outlined in Section 2.3.1. The precision level p is set at 0 for this associative ANN memory module.

The following cases illustrate how the ANN-based library query system retrieves desired record pointers by processing a query which may contain a partially specified input.

- **Case 1** : Suppose a user enters "*Smith* $*$" to search for the books written by authors with last name *Smith*. In this case, that part of input for first name is viewed as unavailable, the first 5 letters for the part of input for last name are `S, m, i, t,` and `h`, and the other $L-5$ letters are `space`s. Therefore, in the ANN memory, the first $8 \times L = 8L$ input neurons have input value either 1 or -1, the last $8 \times L = 8L$ input neurons which together represent the part of input for first name are fed with 0, and the threshold set at all hidden neurons is $8 \times L = 8L$. Suppose the library database contains k volumes written by authors with last name *Smith*. In this case, the ANN memory module contains k hidden neurons for these k volumes (one for each volume written by an author whose last name is *Smith*). During the recall process all these hidden neurons will have net input of 0 and other hidden neurons have net input less than 0 (see Chapter 2 for details). The neurons with non-negative net input get activated one at a time to sequentially recall the desired M-bit pointers pointing to the books written by authors with the specified last name.

- **Case 2** : suppose a user enters "$*$ *John*" to search for the books written by authors with first name *John*. In this case, that part of input for last name is viewed as unavailable,

the first 4 letters for the part of input for first name are `J, o, h,` and `n`, and the other $L-4$ letters are `spaces`. Therefore, in the ANN memory, the last $8 \times L = 8L$ input neurons have input value either 1 or -1, the first $8 \times L = 8L$ input neurons which together represent the part of input for last name are fed with 0, and the threshold set at all hidden neurons is $8 \times L = 8L$. The recall of the associated pointers proceeds as in Case 1.

- **Case 3** : Suppose a user enters "*Smith J**" to search for the books written by authors with last name called *Smith* and first name beginning with a *J*. In this case, the rest of the letters of first name is viewed as unavailable, the first 5 letters for the part of input for last name are `S, m, i, t,` and `h`, and the other $L-5$ letters are `spaces`. Therefore, in the ANN memory, the first $8 \times (L+1) = 8(L+1)$ input neurons have input value either 1 or -1, the last $8 \times (L-1) = 8(L-1)$ input neurons are fed with 0, and the threshold set at all hidden neurons is $8 \times (L+1) = 8(L+1)$. The recall of the associated pointers proceeds as in Case 1.

3.2.3 The implementation of case insensitive pattern matching

It is rather straightforward to modify the proposed ANN-based query system to make it case-insensitive. The following shows ASCII codes of English letters, which are denoted in hexadecimal and binary codes.

$$A = 41_{16} = 0100\ 0001_2, \ldots, Z = 5A_{16} = 0101\ 1010_2$$

$$a = 61_{16} = 0110\ 0001_2, \ldots, z = 7A_{16} = 0111\ 1010_2$$

The binary codes for the capital case and small case of every same English letter only differ at the $3rd$ bit counted from left hand side. If that bit is viewed as "*don't care*" (or unavailable), this query system will be case insensitive. This effect can be achieved by treating the corresponding input value as though it was unavailable.

3.3 Comparison with Other Database Query Processing Techniques

This section compares the anticipated performance of the proposed neural architecture for database query processing with other approaches that are widely used in current computer systems. Such a comparison takes into account the performance of hardware used in these systems and the process used for locating data items. It is assumed that the systems have comparable I/O characteristics which are not discussed here. First, the performance of the proposed neural network is estimated, based on current CMOS technology for realizing neural networks. Next, the operation of conventional database systems is examined, and their performance is estimated and compared to that of the proposed neural architecture.

3.3.1 Performance of current electronic realization for neural networks

Electronic hardware realizations of ANN have been explored by several authors [49, 50, 57, 103, 106, 107, 120, 152, 184, 190]. Such implementations typically employ CMOS analog, digital, or hybrid (analog/digital) electronic circuits. Analog circuits typically consist of processing elements for multiplication, summation and thresholding. Analog CMOS technology is attractive for realization of ANN because it can yield compact circuits that are capable of high-speed asynchronous operation [48]. [184] reports a measured propagation delay of 104ns in a digital circuit with each synapse containing an 8-bit memory, an 8-bit subtractor and an 8-bit adder. [50] reports throughput at the rate of 10MHz (or equivalently, delay of 100ns) in a Hamming Net pattern classifier using analog circuits. [106] describes a hybrid analog-digital design with 5-bit (4 bits + sign) binary synapse weight values and current-summing circuits that is used to realize a 2-layer feed-forward ANN with a network computation delay of less than 20ns.

The 1st-layer and 2nd-layer subnetworks of the proposed neural architecture for database query processing are very similar to the 1st-layer subnetwork of a Hamming Net respectively, and the neural architecture with 2 connection layers in the proposed ANN is exactly same as that implemented by [106] except [106] uses discretized inputs, 5-bit synaptic weights, and sigmoid-like activation function. The proposed ANN uses bipolar inputs, weights in $\{-1, 0, 1\}$

and binary hardlimiter as activation function. Hence the computation delay of the proposed ANN can be expected to be at worst of the order of 100 ns and at best 20 ns given the current CMOS technology for realizing ANN.

The development of specialized hardware for implementation of ANN is still in its early stages. Conventional CMOS technology that is currently the main technology for VLSI implementation of ANN is known to be slow [92, 104]. Other technologies, such as BiCMOS, NCMOS [92], pseudo-NMOS logic, standard N-P domino logic, and quasi N-P domino logic [104], may provide better performance for the realization of ANN. Thus, the performance of the hardware implementation of ANN is likely to improve with technological advances in VLSI.

3.3.2 Analysis of query processing in conventional computer systems

Accessing information based on a key is central to information retrieval systems [37, 157, 158] and database systems [186]. In relational database systems implemented on conventional computer systems, given the value for a key, a record is located efficiently by using key-based organizations including *hashing*, *index-sequential access files* and *B-trees* [186]. Such a key-based organization usually contains two data structures : *index files(s)* and *master file*. In an index file, every key is organized and usually associated with a record pointer which points to a corresponding record in the master file which is typically stored in secondary storage devices like hard disks for large databases. Conventionally, estimated cost of locating a record is based on the number of physical block accesses of secondary storage devices [186] since the access latency with current cost-effective disk systems is around 5$\sim$10 ms (*millisecond*) and every one of the repetitive search steps which together facilitate locating a desired record pointer from index files (loaded into the main memory) takes only several CPU clock cycles. The clock cycle of current cost-effective CPUs is around 2$\sim$10 ns. With the large number of entries in index files of large databases and with the low price of current memory chips, master files of databases for real-time applications tend to be loaded into the main memory to avoid accessing records from low-speed secondary storage devices (compared to memory chips) and thus the cost of locating a desired record pointer can become a dominant cost for record retrieval in

large databases.

The following compares the anticipated performance of the proposed neural associative memory with other approaches that are widely used in current computer systems for locating a record pointer associated with a given key. In the following analysis, it is assumed that all program and index files for processing queries using current computer systems are pre-loaded into the main memory. The effect of data dependency among instructions which offsets pipeline and superscalar effects and thus much reduces the average performance of current computer systems is not considered here.

To simplify the comparison, it is assumed that each instruction on a conventional computer takes τ ns on an average. For instance, on a relatively cost-effective 100 MIPS processor, a typical instruction would take 10 *ns* (The MIPS measure for speed combines clock speed, effect of caching, pipelining and superscalar design into a single figure for speed of a microprocessor). Similarly, we will assume that a single identification and recall operation cycle by a neural associative memory takes α ns. Assuming hardware implementation based on current CMOS VLSI technology, α is around 20$\sim$100 *ns*. Table 3.1 summarizes from following analysis the estimated performance of the proposed neural associative memory and other techniques commonly used in conventional computer systems for locating a desired record pointer. The summary assumes that the value of the key is given, the data structures and programs are loaded into the main memory of the computer systems used, index search occurs in a balanced binary tree of $(2^M - 1)$ records, and partial match occurs in a *k-d-tree* of N records. L is the total number of bytes of a key, n is the data bus width of the computer systems used, h is the average number of executed instructions in a hashing cycle, τ is the average time delay for executing an instruction, b is the average number of executed instructions in a comparison cycle for every n bits in a binary search cycle, α is the time delay of the proposed neural memory, K is the number of index fields used in the *k-d-tree*, and J is the number of index fields specified in a query criterion. Table 3.2 summarizes the capabilities of the proposed neural associative memory and other techniques commonly used in conventional computer systems for exact match, prefix match and partial match mentioned in Section 3.2.1.

Table 3.1 A comparison of the estimated performance of the proposed neural associative memory with that of other techniques commonly used in conventional computer systems for locating a record pointer in key-based organizations

Method	*Estimated time (ns)*
hashing	$\lceil 8L/n \rceil\ h\ \tau$
index search	$(M-1)\lceil 4L/n \rceil\ b\ \tau$
ANN memory	α
k-d-tree (partial match)	$O(N^{(K-J)/K})$

Table 3.2 A comparison of the capabilities of the proposed neural associative memory with those of other techniques commonly used in conventional computer systems for exact match and partial match

Method	*Exact match*	*Prefix match*	*Partial match*
hashing	efficient	unable	unable
index search	efficient	efficient	inefficient
ANN memory	efficient	efficient	efficient
k-d-tree	satisfactory	satisfactory	inefficient

3.3.2.1 Analysis of locating a record pointer using hashing functions

Hashing structure is the fastest of all key-based searching techniques for locating a record pointer for a single record. However, although it is effective in locating a single record by exact match (e.g., example 1 of Section 3.2.1), it is inefficient at or incapable of locating related records in response to a partially specified input (e.g., examples 2 and 3 of Section 3.2.1). Let us consider the time needed for locating a record pointer using a hash function in current computer systems. Commonly used hash functions are based on multiplication, division and addition operations [87, 163]. In hardware implementation addition is faster than multiplication which in turn is far faster than division. Assume that computing a hashing function on a key with a length of L bytes (characters) takes $\lceil 8L/n \rceil$ cycles using a processor with an n-bit data bus and every cycle takes h instructions. Then, the estimated computation time for locating a record pointer is $\lceil 8L/n \rceil\ h\ \tau$. Other overheads in computing a hashing function in such systems include the time for handling the potential problem of *collisions* in hash functions. If a single-CPU 100 MIPS processor with a 32-bit data bus is used, it is

expected that the total computation time for locating a record pointer will typically be in excess of $100ns$ (If $L = 15$ and $h = 5$, the total computation time is $\lceil 8 \times 15/32 \rceil \times 5 \times 10\ ns = \lceil 120/32 \rceil \times 50\ ns = 200\ ns$).

3.3.2.2 Analysis of locating a record pointer using index search

A perfectly balanced binary search tree is another popular, efficient data structure used in conventional database systems to locate a single record by exact match (e.g., example 1 of Section 3.2.1) or several related records by partial match (e.g., example 2 but not example 3 of Section 3.2.1). Assume every non-terminal node in a perfectly balanced binary search tree links two child subtrees and there are $(2^M - 1)$ nodes in the tree. Assume the length of the index key is L bytes (characters). The average number of nodes visited for locating a desired key would be $\frac{2^M(M-1)+1}{2^M-1} \approx M - 1$. On an average, every visit takes $\lceil (\frac{1}{2} \times 8L)/n \rceil = \lceil 4L/n \rceil$ comparison cycles for a processor with an n-bit data bus. Suppose every comparison cycle takes b instructions. Then, the estimated computation time for locating a desired record pointer is $(M - 1)\lceil 4L/n \rceil\ b\ \tau$. If $L = 15$, and a 100 MIPS processor with a 32-bit data bus is used, the comparison cycle for every 32 bits takes 5 instructions on average, and there are $2^{16} - 1 = 65,535$ records (the number of words a native English speaker knows is estimated to be between 50,000 and 250,000 [4]), then the overhead for locating a desired record pointer is about $(16 - 1) \times \lceil 4 \times 15/32 \rceil \times 5 \times 10\ ns = 1500\ ns$ which compares unfavorably with 100 ns. Note that this is only the cost of locating a record pointer for a single record. The cost of locating several record pointers of related records using user-entered data in an index file containing multiple index fields is examined in next section.

3.3.2.3 The cost of partial-match queries

One of the most commonly used data structures for processing partial-match queries on multiple index fields is *k-d-tree* [12]. It can provide approximately satisfactory performance for locating a single record by exact match or several related records by partial match. In the worst case, the number of visited nodes in an ideal *k-d-tree* of N nodes (one for each record

stored) for locating the desired record pointers for a partial-match query is

$$\frac{(J+2)2^{K-J-1}-1}{2^{K-J}-1}[(N+1)^{(K-J)/K}-1] \approx O(N^{(K-J)/K}) \tag{3.1}$$

where K is the number of index fields used to construct the *k-d-tree*, and J out of K index fields are explicitly specified by a user query. For typical values of N, K, and J, the performance of such systems is far worse than that of the proposed ANN based model according to expression 3.1.

3.4 Summary and Discussion

Artificial neural networks, due to their inherent parallelism and potential for noise tolerance, offer an attractive paradigm for efficient implementations of a broad range of information processing tasks. In this chapter, we have explored the use of artificial neural networks for pattern-based (key-based) query processing in large databases. The use of the proposed approach was demonstrated using the examples of a library query system and a query system for text-based machine-readable lexicon used in natural language processing. The performance of a CMOS hardware realization of the proposed neural associative memory for database query processing system was estimated and compared with that of other approaches which are widely used in conventional databases implemented on current computer systems. The comparison shows that ANN architectures for query processing offer an attractive alternative to conventional approaches, especially for dealing with partial-match queries in large databases. With the need for real-time response in language translation and with the explosive growth of the Internet as well as increased use of large networked databases over the Internet, efficient architectures for high-speed information retrieval, associative table lookup, message routing and database query processing have assumed great practical significance.

4 NEURAL ARCHITECTURES FOR ELEMENTARY LOGICAL INFERENCE

4.1 Introduction

Inference often involves tasks which look for interesting situations occurring as patterns in the input or memory data to solve questions such as *"What is the most likely answer?", "Is there sufficient evidence to adopt a conclusion or is more evidence needed?"* [42, 61, 191], etc. Such tasks are important for inference from partial information, and they generally involve a process of pattern recognition by way of best, partial, and/or exact matches. Artificial neural networks, due to their inherent massive parallelism, potential for fault tolerance and adaptation capability through learning, have attracted extensive interest for robust and efficient implementations of logical inference systems. Many of the systems proposed in the literature are motivated by the need for massively parallel architecture for AI applications, and some of them are proposed to model human cognitive processes robustly. In particular, they explore neural mechanisms for variable binding to facilitate complex reasoning based on predicate logic [5, 31, 95, 175, 176]; and connectionist realizations of production system [182], expert systems [42, 43], hybrid knowledge processing system [136], semantic networks [68, 167], frame representation [79], planning [145, 188], nonmonotonic reasoning [142], legal reasoning [148], commonsense reasoning [177, 178], and logical theorem proving [141]. This chapter explores how neural architectures for binary partial pattern recognition can be extended for elementary logical inference based on propositional logic. The proposed neural architectures, like the ones proposed in Chapters 2 and 3 for associative memory and query processing, exploit the massively parallel computational capability of artificial neural networks.

Propositional logic, which typically operates on propositions and logical connectives: AND,

OR, as well as negation, is basic to logical inference. For this reason, it is customary to use propositional logic for demonstrating the feasibility of new tools for logical inference. This chapter proposes a method based on geometrical/mathematical analysis to systematically design neural architectures for realizing logical ANDs, logical ORs, and DNF (Disjunctive Normal Form) propositions (sum of products). A DNF proposition is a disjunction of conjunctions. The evaluation of a conjunction corresponds to that of a logical AND function, and the evaluation of a disjunction corresponds to that of a logical OR function. The evaluation of logical AND and OR functions can be respectively realized by the AND and OR neural assemblies proposed in this chapter through a process of pattern recognition. It is known that any proposition can be represented in DNF. Therefore, any proposition can be realized by a 2-layer neural architecture assembled from an OR neural assembly and a fixed number of AND neural assemblies. The rest of the chapter is organized as follows:

- Section 4.2 develops two types of neural assemblies for the recognition of binary partial patterns.
- Section 4.3 develops a general AND neural assembly which can be used to realize any arbitrary logical AND function of a finite number of Boolean variables.
- Section 4.4 develops a general OR neural assembly which can be used to realize any arbitrary logical OR function of a finite number of Boolean variables. Then, a monotone OR neural assembly is derived.
- Section 4.5 discusses how to use AND and OR neural assemblies to realize arbitrary Boolean functions.
- Section 4.6 concludes with a summary of the chapter and a brief discussion.

4.2 Neural Assemblies for the Recognition of Partial Patterns

This section develops two types of neural assemblies for the recognition of binary partial patterns. One type of the assemblies is used for the recognition of patterns which contain a

specific sub-pattern, and the other is used for the recognition of patterns which don't contain a specific sub-pattern. Let us call the former the *neural assembly for inclusive recognition*, and the latter, the *neural assembly for exclusive recognition*. The two assemblies are used to build the `AND` neural assembly proposed in Section 4.3 and the `OR` neural assembly proposed in Section 4.4 respectively.

4.2.1 A neural assembly for inclusive recognition

Let $u =< u_1, ..., u_n >$ be a binary vertex (vector) of dimension n, where $u_i \in \{0, 1\}$ for $1 \leq i \leq n$. Let $H_S^{n,u}$ be an n-dimensional separating hyperplane which can be used to implement a 1-layer Perceptron to distinguish the vertex u from all other n-dimensional vertices. According to expression 2.12, among a set of possible expressions for $H_S^{n,u}$, we choose:

$$H_S^{n,u} \equiv \sum_{i=1}^{n}(2u_i - 1)x_i - \sum_{i=1}^{n} u_i = 0 \tag{4.1}$$

Therefore, in the n-input, 1-output Perceptron implemented to recognize the vertex u

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

Now consider a binary vector $\tilde{u}$ of dimension m, where $m \geq n$. Suppose, in a system of m variables, only the values of n of m components of vector $\tilde{u}$ are of interest. For two given binary vectors u and $\tilde{u}$ of dimensions n and m respectively, only whether $\tilde{u}_{j_1} = u_1, ..., \tilde{u}_{j_i} = u_i, ..., \tilde{u}_{j_n} = u_n$ are concerned, where $1 \leq n \leq m$ and $1 \leq j_1 < j_2 < \cdots < j_n \leq m$. Let us call $J^n = \{j_1, j_2, ..., j_n\}$ the *interest set* J^n, and $\tilde{u}(J^n) =< \tilde{u}_{j_1}, \tilde{u}_{j_2}, ..., \tilde{u}_{j_n} >$ the J^n*-set partial vector* of the binary vector $\tilde{u}$. Note that several interest sets could be defined concurrently for a given problem in an m-dimensional binary space. In the following, the expression for the separating hyperplane $H_S^{n,u}$ (expression 4.1) in an n-dimensional binary space is re-defined as a separating hyperplane H_S^{m,u,J^n} in an m-dimensional binary space to implement a 1-layer Perceptron to recognize all the 2^{m-n} m-dimensional binary vectors whose J^n-set partial vectors

equal the given n-dimensional binary vector $u = \tilde{u}(J^n) =< \tilde{u}_{j_1}, \tilde{u}_{j_2}, ..., \tilde{u}_{j_n} >=< u_1, u_2, ..., u_n >$:

$$H_S^{m,u,J^n} \equiv \sum_{j_i \in J^n}^{m} (2u_i - 1)x_{j_i} + \sum_{i \notin J^n}^{m} 0 \cdot x_i - \sum_{k}^{n} u_k = 0 \tag{4.2}$$

Let w_i be the connection weight from the ith input neuron to the output neuron and θ be the threshold of the output neuron in a 1-layer, 1-output Perceptron. Then, in the m-input, 1-output Perceptron,

- $\forall j_i \in J^n \ \& \ 1 \leq i \leq n, w_{j_i} = 2u_i - 1$,
- $\forall i \notin J^n \ \& \ 1 \leq i \leq m, w_i = 0$, and
- $\theta = \sum_{k=1}^{n} u_k$.

The values from those input neurons which are not in the interest set J^n will not affect the net input of the output neuron since the weights on the connections from those input neurons are set as 0. These connections together act as a *don't-care* filter.

For example, suppose one wants to use a 1-layer Perceptron to recognize all the 5-dimensional binary vertices whose 1st, 3rd, and 5th components are 1, 0, and 1 respectively. Then, the corresponding interest set would be $J^5 = \{1, 3, 5\}$, and the implemented Perceptron is shown in Figure 4.1.

4.2.2 A neural assembly for exclusive recognition

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 (Theorem 2.1). Those $n+1$ layers are respectively on $n+1$ mutually parallel n-dimensional hyperplanes $H_p^{n,u}$'s (expression 2.12), $0 \leq p \leq n$, where

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

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

$$H_0^{n,u} \equiv \sum_{i=1}^{n}(2u_i - 1)x_i - \sum_{i=1}^{n} u_i = 0 \tag{4.3}$$

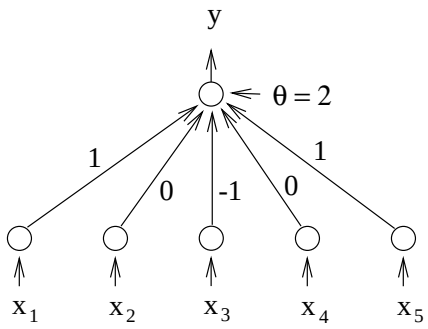

Figure 4.1 A 1-layer Perceptron which recognizes all the 5-dimensional binary patterns that contain the partial pattern *<1,?,0,?,1>*, where *?* denotes don't care

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

$$H_n^{n,u} \equiv \sum_{i=1}^{n}(2u_i - 1)x_i - (\sum_{i=1}^{n} u_i - n) = 0 \tag{4.4}$$

The hyperplane $H_{n-1}^{n,u}$ which is defined as

$$H_{n-1}^{n,u} \equiv \sum_{i=1}^{n}(2u_i - 1)x_i - (\sum_{i=1}^{n} u_i - n + 1) = 0 \tag{4.5}$$

can be used to implement a 1-layer Perceptron to distinguish the binary vertex $\overline{u}$ from all other n-dimensional binary vertices (i.e. the Perceptron recognizes all the n-dimensional binary vertices which are not $\overline{u}$) by setting

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

in the n-input, 1-output Perceptron.

Now consider a binary vector $\tilde{u}$ of dimension m, where $m \geq n$. Suppose, only the values of n of m components of vector $\tilde{u}$ are of interest and an interest set $J^n = \{j_1, j_2, ..., j_n\}$ is defined. The expression for the hyperplane $H_{n-1}^{n,u}$ in an n-dimensional binary space is re-defined as a hyperplane H_{n-1}^{m,u,J^n} in an m-dimensional binary space to implement a 1-layer Perceptron to recognize all the 2^{m-n} m-dimensional binary vectors whose J^n-set partial vectors don't equal

the n-dimensional binary vector $\overline{u}$. Then,

$$H_{n-1}^{m,u,J^n} \equiv \sum_{j_i \in J^n}^{m} (2u_i - 1)x_{j_i} + \sum_{i \notin J^n}^{m} 0 \cdot x_i - (\sum_{k}^{n} u_k - n + 1) = 0 \tag{4.6}$$

and, in the 1-layer, m-input, 1-output Perceptron,

- $\forall j_i \in J^n \ \& \ 1 \leq i \leq n, w_{j_i} = 2u_i - 1$,
- $\forall i \notin J^n \ \& \ 1 \leq i \leq m, w_i = 0$, and
- $\theta = \sum_{k=1}^{n} u_k - n + 1$.

For example, suppose one wants to use a 1-layer Perceptron to recognize all the 5-dimensional binary vertices whose 1st, 3rd, and 5th components are not 1, 0, and 1 respectively. Then, the corresponding interest set would be $J^5 = \{1, 3, 5\}$, and the implemented Perceptron is shown in Figure 4.2.

4.3 A Neural Assembly for Executing a Logical AND (AND Neural Assembly)

This section develops an AND neural assembly which can realize any arbitrary logical AND function of a finite number of Boolean variables. First, we develop notations to represent Boolean variables (atomic propositional variables) and logical AND expressions to facilitate such a realization. Let $\neg v_i$ be the negation of the Boolean variable v_i. Further, let $\neg v_i$ be denoted by v_i^0, and v_i by v_i^1. Then a logical expression $v_1 \wedge \neg v_2 \wedge v_3$ (a conjunction of three Boolean variables) can be denoted as $v_1^1 \wedge v_2^0 \wedge v_3^1$. Let $v =< v_1, ..., v_n >$ and $z =< z_1, ..., z_n >$, where $v_i, z_i \in \{0, 1\}$ for $1 \leq i \leq n$. Then, for a logical AND function denoted by $C^{n,z}(v) = v_1^{z_1} \wedge \ v_2^{z_2} \cdots \wedge v_n^{z_n}$, we have

$$C^{n,z}(v) = \begin{cases} 1 & \text{if } v = z \\ 0 & \text{if } v \text{ is any other } n\text{-dimensional binary vertex} \end{cases} \tag{4.7}$$

The evaluation of the logical AND function $C^{n,z}(v)$ can be viewed as a process of binary pattern recognition. Thus it can be realized by a 1-layer Perceptron that implements the

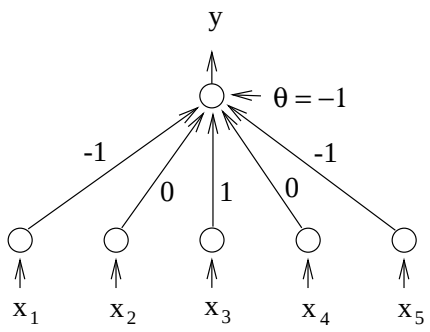

Figure 4.2 A 1-layer Perceptron which recognizes all the 5-dimensional binary patterns that don't contain the partial pattern *<1,?,0,?,1>*, where *?* denotes don't care

hyperplane $H_S^{n,z}$ to recognize the binary vertex z. Let $H_{AND}^{n,z}$ be used for $H_S^{n,z}$. Then, according to expression 4.1 and its associated Perceptron implementation,

$$H_{AND}^{n,z} \equiv H_S^{n,z} \equiv \sum_{i=1}^{n}(2z_i - 1)x_i - \sum_{i=1}^{n} z_i = 0 \tag{4.8}$$

and the logical `AND` function $C^{n,z}(v)$ can be realized by a 1-layer Perceptron with n input neurons and one output neuron. The corresponding Perceptron has

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

For example, suppose $v =< v_1, v_2, v_3 >$ and $C(v) = v_1 \wedge \neg v_2 \wedge v_3 = v_1^1 \wedge v_2^0 \wedge v_3^1$. Then, we have

$$C(v) = \begin{cases} 1 & \text{if } v =< 1,0,1 > \\ 0 & \text{if } v \text{ is any other } n\text{-dimensional binary vertex} \end{cases} \tag{4.9}$$

and the corresponding Perceptron which realizes the logical `AND` function $C(v)$ is shown in Figure 4.3.

In order to be able to realize all possible logical `AND` functions in a system of m Boolean variables using their corresponding 1-layer Perceptrons, the expression for the separating hyperplane $H_{AND}^{n,z}$ is extended from an n-dimensional binary space to an m-dimensional binary space to recognize all the m-dimensional binary patterns whose partial patterns equal the

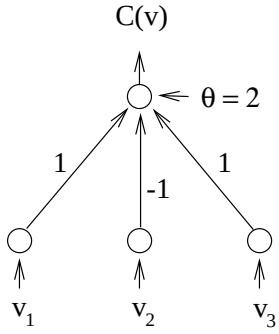

Figure 4.3 An AND neural assembly which realizes the logical AND function $C(v)$

n-dimensional binary vector z for certain interest set J^ns, where $m \geq n$. Suppose $\tilde{v} =< \tilde{v}_1, \tilde{v}_2, ..., \tilde{v}_m >$ is a binary vertex of dimension m. We define an interest set $J^n = \{j_1, j_2, ..., j_n\}$, $1 \leq j_1 < j_2 < \cdots < j_n \leq m$. Let $C^{m,z,J^n}(\tilde{v}) = \tilde{v}_{j_1}^{z_1} \wedge \tilde{v}_{j_2}^{z_2} \cdots \wedge \tilde{v}_{j_n}^{z_n}$. Then

$$C^{m,z,J^n}(\tilde{v}) = \begin{cases} 1 & \text{if } \tilde{v}(J^n) = z \\ 0 & \text{if } \tilde{v}(J^n) \text{ is any other } n\text{-dimensional binary vector} \end{cases} \tag{4.10}$$

The logical AND function $C^{m,z,J^n}(v)$ can be realized by a 1-layer Perceptron that implements the hyperplane H_S^{m,z,J^n} to recognize all the m-dimensional binary vectors whose J^n-set partial vectors equal to the n-dimensional binary vector z. Let H_{AND}^{m,z,J^n} be used for H_S^{m,z,J^n}. Then, according to expression 4.2 and its associated Perceptron implementation,

$$H_{AND}^{m,z,J^n} \equiv \sum_{j_i \in J^n}^{m} (2z_i - 1)x_{j_i} + \sum_{i \notin J^n}^{m} 0 \cdot x_i - \sum_{k}^{n} z_k = 0 \tag{4.11}$$

and the logical AND function $C^{m,z,J^n}(\tilde{v}) = \tilde{v}_{j_1}^{z_1} \wedge \tilde{v}_{j_2}^{z_2} \cdots \wedge \tilde{v}_{j_n}^{z_n}$ can be realized by a 1-layer Perceptron with m input neurons and one output neuron. In the Perceptron,

- $\forall j_i \in J^n \ \& \ 1 \leq i \leq n, w_{j_i} = 2z_i - 1$,
- $\forall i \notin J^n \ \& \ 1 \leq i \leq m, w_i = 0$, and
- $\theta = \sum_{k=1}^{n} z_k$.

Such an AND neural assembly will be used as a building block to assemble the neural architectures for realizing Boolean functions represented in DNF representation (see Section 4.5). Examples of such a neural assembly will be shown in Section 4.5 to assemble a neural architecture which realizes a given DNF Boolean function.

4.4 Neural Assemblies for Executing Logic ORs (OR Neural Assemblies)

This section develops OR neural assemblies which can realize any arbitrary logical OR functions of a finite number of Boolean variables. First, a general OR neural assembly is described. The assembly will be used a building block to assemble the neural architecture proposed in Section 4.5 for realizing DNF Boolean functions. Then, a monotone OR neural assembly is derived from the general OR neural assembly. The monotone OR neural assembly will be used as a building block to assemble the neural architecture proposed in Section 5.5 for realizing NFA.

4.4.1 A general OR neural assembly

This subsection investigates how a 1-layer Perceptron can realize a general logical OR function which contains negated Boolean variables. The notations used here follows that in Section 4.3. Let $D_G^{n,z}(v) = v_1^{z_1} \vee v_2^{z_2} \cdots \vee v_n^{z_n}$, where $\vee$ is a logical connective OR, v_i's are Boolean variables, $v =< v_1, ..., v_n >$, $z =< z_1, ..., z_n >$, and $v_i, z_i \in \{0, 1\}$ for $1 \leq i \leq n$. Then, we have

$$D_G^{n,z}(v) = \begin{cases} 0 & \text{if } v =< \overline{z}_1, \overline{z}_2, ..., \overline{z}_n > \\ 1 & \text{if } v \text{ is any other } n\text{-dimensional binary vertex} \end{cases} \tag{4.12}$$

The logical OR function $D_G^{n,z}(v)$ can be realized by a 1-layer Perceptron that implements the hyperplane $H_{n-1}^{n,z}$ to recognize all n-dimensional binary vertices which are not $\overline{z}$. Let $H_{OR}^{n,z}$ be used for $H_{n-1}^{n,z}$. Then, according to expression 4.5 and its associated Perceptron implementation,

$$H_{OR}^{n,z} \equiv H_{n-1}^{n,z} \equiv \sum_{i=1}^{n}(2z_i - 1)x_i - (\sum_{i=1}^{n} z_i - n + 1) = 0 \tag{4.13}$$

and, in the n-input, 1-output Perceptron which realizes $D_G^{n,z}(v)$,

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

For example, suppose $v =< v_1, v_2, v_3 >$ and $D(v) = v_1 \vee \neg v_2 \vee v_3 = v_1^1 \vee v_2^0 \vee v_3^1$. Then, we have

$$D(v) = \begin{cases} 0 & \text{if } v =< \overline{1}, \overline{0}, \overline{1} > \\ 1 & \text{if } v \text{ is any other } n\text{-dimensional binary vertex} \end{cases} \tag{4.14}$$

and the corresponding Perceptron which realizes the logical OR function $D(v)$ is shown in Figure 4.4.

In order to be able to realize all the possible logical OR functions in a system of m Boolean variables using their corresponding 1-layer Perceptron implementations, the expression for the separating hyperplane $H_{OR}^{n,z}$ is extended from an n-dimensional binary space to an m-dimensional binary space to recognize all the m-dimensional binary patterns whose partial patterns don't equal the n-dimensional binary vector $\overline{z}$ for certain interest set J^ns, where $m \geq n$. Suppose $\tilde{v} =< \tilde{v}_1, \tilde{v}_2, ..., \tilde{v}_m >$ is a binary (Boolean) vertex of dimension m. We define an interest set $J^n = \{j_1, j_2, ..., j_n\}$, $1 \leq j_1 < j_2 < \cdots < j_n \leq m$. Let $D_G^{m,z,J^n}(\tilde{v}) = \tilde{v}_{j_1}^{z_1} \vee \tilde{v}_{j_2}^{z_2} \cdots \vee \tilde{v}_{j_n}^{z_n}$. Then

$$D_G^{m,z,J^n}(\tilde{v}) = \begin{cases} 0 & \text{if } \tilde{v}(J^n) = \overline{z} \\ 1 & \text{if } \tilde{v}(J^n) \text{ is any other } n\text{-dimensional binary vector} \end{cases} \tag{4.15}$$

The logical OR function $D_G^{m,z,J^n}(\tilde{v})$ can be realized by a 1-layer Perceptron that implements the hyperplane H_{n-1}^{m,z,J^n} to recognize the m-dimensional binary vectors whose J^n-set partial vectors don't equal to the n-dimensional binary vector $\overline{z}$. Let H_{OR}^{m,z,J^n} be used for H_{n-1}^{m,z,J^n}. Then, according to expression 4.6 and its associated Perceptron implementation,

$$H_{OR}^{m,z,J^n} \equiv \sum_{j_i \in J^n}^{m} (2z_i - 1)x_{j_i} + \sum_{i \notin J^n}^{m} 0 \cdot x_i - (\sum_{k}^{n} z_k - n + 1) = 0 \tag{4.16}$$

and, in the m-input, 1-output Perceptron which realizes $D_G^{m,z,J^n}(\tilde{v})$,

- $\forall j_i \in J^n \ \& \ 1 \leq i \leq n, w_{j_i} = 2z_i - 1$,
- $\forall i \notin J^n \ \& \ 1 \leq i \leq m, w_i = 0$, and

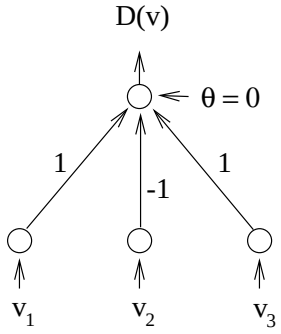

Figure 4.4 An OR neural assembly which realizes the logical OR function $D(v)$

- $\theta = \sum_{k=1}^{n} z_k - n + 1.$

Figure 4.2 is a corresponding Perceptron implementation which realize the logical OR function $y = D(x) = \neg x_1 \vee x_3 \vee \neg x_5 = x_1^0 \vee x_3^1 \vee x_5^0$ in a system which contains Boolean variables x_1, x_2, x_3, x_4, and x_5; where $x =< x_1, ..., x_5 >$.

4.4.2 A monotone OR neural assembly

Consider an n-variable Boolean expression represented as a monotone disjunction :

$$v_1 \vee \ v_2 \cdots \vee v_n \tag{4.17}$$

where v_i's are Boolean variables. Monotone disjunctions are simply disjunctions which don't contain negated Boolean variables. For example, $v_1 \vee v_2$ is a monotone disjunction, but $\neg v_1 \vee v_2$ is not a monotone disjunction. Let $v =< v_1, ..., v_n >$ and $D_M^n(v) = v_1^1 \vee \ v_2^1 \cdots \vee v_n^1$. Then, we have

$$D_M^n(v) = \begin{cases} 0 & \text{if } v =< \overline{1}^n > \\ 1 & \text{if } v \text{ is any other } n\text{-dimensional binary vertex} \end{cases} \tag{4.18}$$

The logical monotone OR function $D_M^n(v)$ is a special case of a general logical OR function $D_G^{n,z}(v)$ with $z =< 1^n >$. Then, according to the Perceptron implementation for $D_G^{n,z}(v)$ (which corresponds to expression 4.13), the logical monotone OR function $D_M^n(v)$ can be

realized by an n-input, 1-output Perceptron with the threshold of the output neuron being set as 1, and the connection weight from every input neuron to the output neuron being set as 1.

In order to be able to realize all the possible logical monotone `OR` functions in a system of m Boolean variables using their corresponding 1-layer Perceptron implementations, where $m \geq n$. The logical monotone `OR` function $D_M^n(v)$ is extended from an n-dimensional Boolean space to an m-dimensional Boolean space, where $m \geq n$. Suppose $\tilde{v} =< \tilde{v}_1, \tilde{v}_2, ..., \tilde{v}_m >$ is a Boolean vector of dimension m. Assume an interest set $J^n = \{j_1, j_2, ..., j_n\}$, $1 \leq j_1 < j_2 < \cdots < j_n \leq m$, is defined. Define $D_M^{m,J^n}(\tilde{v}) = \tilde{v}_{j_1}^1 \vee \tilde{v}_{j_2}^1 \cdots \vee \tilde{v}_{j_n}^1$. Then

$$D_M^{m,J^n}(\tilde{v}) = \begin{cases} 0 & \text{if } \tilde{v}(J^n) =< \overline{1}^n > \\ 1 & \text{if } \tilde{v}(J^n) \text{ is any other } n\text{-dimensional binary vector} \end{cases} \tag{4.19}$$

The logical monotone `OR` function $D_M^{m,J^n}(\tilde{v})$ is a special case of a general logical `OR` function $D_G^{m,z,J^n}(\tilde{v})$ with $z =< 1^n >$. Then, according to the Perceptron implementation for $D_G^{m,z,J^n}(\tilde{v})$ (which corresponds to expression 4.16), the logical monotone `OR` function $D_M^{m,J^n}(v)$ can be realized by an m-input, 1-output Perceptron with

- $\forall j_i \in J^n \ \& \ 1 \leq i \leq n, w_{j_i} = 1$,
- $\forall i \notin J^n \ \& \ 1 \leq i \leq m, w_i = 0$, and
- $\theta = 1$.

Such a general `OR` neural assembly will be used as a building block to assemble the neural architectures for realizing NFA in Section 5.5.

4.5 A Neural Architecture for Realizing DNF Boolean Functions

Let $\neg C_i$ be the negation of the conjunction C_i. Further, let $\neg C_i$ be denoted by $(C_i)^0$, and C_i by $(C_i)^1$. Let $v =< v_1, ..., v_m >$ and C_i be defined on v for $1 \leq i \leq n$. Then, a DNF Boolean function $D_{DNF}^m(v) = C_1^{z_1} \vee \ C_2^{z_2} \cdots \vee C_n^{z_n}$ can be realized by a 2-layer Perceptron. The first layer of the Perceptron consists of n m-input `AND neural assemblies` defined by expression 4.11, and the second layer is an n-input `OR` neural assembly defined by expression 4.13. Each of the n `AND` neural assemblies is used to realize a conjunction C_i, where $1 \leq i \leq n$.

For example, let $v =< v_1, v_2, v_3, v_4, v_5 >$, $J_1^5 = \{1, 2, 3\}$, and $J_2^5 = \{3, 4, 5\}$. Then,

$$E(v) = (v_1 \wedge \neg v_2 \wedge v_3) \vee \neg(v_3 \wedge v_4 \wedge v_5) \tag{4.20}$$

$$= (v_1^1 \wedge v_2^0 \wedge v_3^1)^1 \vee (v_3^1 \wedge v_4^1 \wedge v_5^1)^0 \tag{4.21}$$

$$= (C^{5,<1,0,1>,J_1^5}(v))^1 \vee (C^{5,<1,1,1>,J_2^5}(v))^0 \tag{4.22}$$

where $C^{5,<1,0,1>,J_1^5}(v) = v_1 \wedge \neg v_2 \wedge v_3$ and $C^{5,<1,1,1>,J_2^5}(v) = v_3 \wedge v_4 \wedge v_5$. The corresponding 2-layer Perceptron which realize the DNF Boolean function $E(v)$ is shown in Figure 4.5.

4.6 Summary and Discussion

Artificial neural networks, due to their inherent massive parallelism, potential fault tolerance and adaptation capability through learning, offer an alternative paradigm for robust and efficient implementations of logical inference systems. In this chapter, a method based on geometrical/mathematical analysis has been proposed for systematically designing neural architectures for elementary logical inference. Particularly, neural architectures for realizing logical `AND`s, logical `OR`s, and DNF propositions have been synthesized by way of binary pattern recognition.

The input to a Boolean function can be represented as a binary (bipolar) code. Therefore, the evaluation of a Boolean function can be viewed as a process of binary (bipolar) pattern recognition. It is known that every Boolean function can be represented as a DNF expression [43]. A DNF expression is a disjunction of conjunctions. The evaluations of conjunction and disjunction can be realized by the proposed `AND` and `OR` neural assemblies respectively. Hence, any Boolean function (except the constant 0) can be realized by a 2-layer neural architecture (Perceptron) assembled from a fixed number of `AND` and `OR` neural assemblies. Besides, Perceptrons have space and speed advantages over DNF representations for representing and evaluating Boolean functions (see [43] for details). Since logical `AND`, logical `OR`, as well as DNF representation are essential to logical inference and Boolean functions are basic to many applications in science and engineering, we expect the proposed neural assemblies would find use in the construction of modular neural networks for a variety of applications. But, in or-

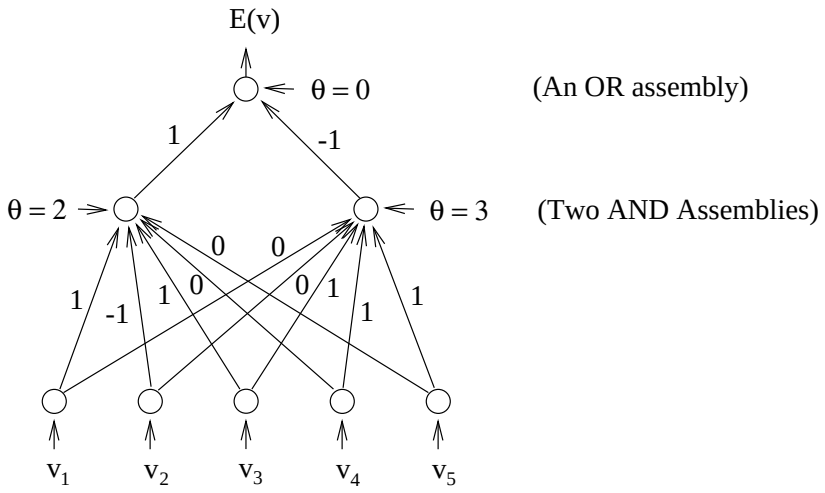

Figure 4.5 An neural architecture which realizes the DNF Boolean function *E(v)*

der to apply neural networks to applications involving more complex logical inference, neural networks would need to be able to do variable binding, logical proof, unification, resolution, etc.

It is worth pointing out that the derivation of `AND` and `OR` neural assemblies which operate on bipolar values is straightforward given the methods proposed in this chapter and the method proposed in Section 2.2.6 for the conversion between models using bipolar and binary inputs. We expect that the resulting bipolar `AND` and `OR` neural assemblies will be exactly equivalent to those proposed in [43]. Since an input value 0 can be used to stand for `unknown` in bipolar model which denotes `true` by 1 and `false` by -1, bipolar model is more flexible than binary model which denotes `true` by 1 and `flase` by 0.

5 NEURAL ARCHITECTURES FOR SEQUENCE PROCESSING

5.1 Introduction

Artificial neural networks (ANN), due to their inherent parallelism, offer an attractive paradigm for efficient implementations of functional modules for symbol processing. This chapter focuses on systematic designs for neural network architectures for sequence processing which is essential to many practical applications involving symbol processing in computer science, linguistics, systems modeling and control, artificial intelligence, and structural pattern recognition.

The capabilities of neural network models (in particular, recurrent networks of threshold logic units or McCulloch-Pitts neurons) in processing and generating sequences (strings defined over some finite alphabet) and hence their formal equivalence with finite state automata or regular language generators/recognizers have been known for several decades [83, 108, 117]. More recently, recurrent neural network realizations of finite state automata for recognition and learning of finite state (regular) languages have been explored by numerous authors [6, 20, 33, 38, 45, 44, 77, 122, 129, 132, 133, 134, 159, 166, 192]. There has been considerable work on extending the computational capabilities of recurrent neural network models by providing some form of external memory in the form of a tape [194] or a stack [13, 27, 66, 116, 123, 144, 161, 169, 174, 197]. To the best of our knowledge, to date, most of the research on neural architectures for sequence processing has focused on the investigation of neural networks that are designed to *learn* to handle sequence processing.

This chapter presents designs of several modular ANN modules for basic sequence processing. The ANN modules which are used as building blocks for the neural architectures proposed in Chapter 6 for syntax analysis include neural network architectures for realizing determin-

istic finite automata, stacks, and deterministic pushdown automata. These ANN modules are systematically synthesized from the BMP modules proposed in Section 2.2. Besides, neural network architecture for realizing nondeterministic finite automata is proposed to explore the potential benefits of ANN in the design of high performance systems for parallel symbolic computing applications. The rest of the chapter is organized as follows:

- The rest of Section 5.1 briefly discusses how to represent symbolic functions in terms of binary mappings to facilitate symbolic information manipulation via the proposed BMP module which operates on binary values.
- Sections 5.2, 5.3, 5.4 and 5.5 respectively explore the systematic synthesis of neural network architectures for realizing deterministic finite automata, deterministic pushdown automata, stack and nondeterministic finite automata.
- Section 5.6 concludes with a summary and a brief discussion.

5.1.1 Symbolic functions and binary mappings

In general, most of simple, non-recursive symbolic functions and table lookup functions can be viewed in terms of a binary random mapping $f_I : U \rightarrow V$ (expression 2.1). For example, f_I may define a symbolic mapping function $f_S : \Gamma_1 \times \Gamma_2 \cdots \times \Gamma_r \rightarrow \Delta_1 \times \Delta_2 \cdots \times \Delta_t$ as described in Section 3.1.1. In this case, the operations of f_S on its associated symbols can be viewed in terms of the binary mapping operations of f_I which in turn can be realized by a BMP module proposed in Section 2.2.5.

Therefore, modular neural network modules for complex symbol processing can be synthesized through a *composition* of appropriate primitive symbolic functions which are directly realized by suitable BMP modules. Two of basic ways of recursively composing composite symbolic functions from component symbolic functions (which may themselves be composite functions or primitive functions) are discussed here. Let f and g be two symbolic functions defined as follows:

$$f : \Gamma_1 \times \Gamma_2 \cdots \times \Gamma_r \rightarrow \Delta_1 \times \Delta_2 \cdots \times \Delta_s \tag{5.1}$$

$$g : \Delta_1 \times \cdots \times \Delta_s \rightarrow \Lambda_1 \times \cdots \times \Lambda_t \tag{5.2}$$

The composition of f and g is denoted by $g \circ f$ such that

$$g \circ f : \Gamma_1 \times \cdots \times \Gamma_r \rightarrow \Lambda_1 \times \cdots \times \Lambda_t \tag{5.3}$$

and for every $(\alpha_1, \cdots, \alpha_r)$ in $\Gamma_1 \times \cdots \times \Gamma_r$

$$g \circ f(\alpha_1, \cdots, \alpha_r) = g(f(\alpha_1, \cdots, \alpha_r)) \tag{5.4}$$

Suppose f_i is a symbolic function such that

$$f_i : \Gamma_1 \times \cdots \times \Gamma_r \rightarrow \Delta_i \text{ for } 1 \leq i \leq s \tag{5.5}$$

The composition c of symbolic functions g, f_1, ..., f_s is defined as:

$$c : \Gamma_1 \times \cdots \times \Gamma_r \rightarrow \Lambda_1 \times \cdots \times \Lambda_t \tag{5.6}$$

and for every $(\alpha_1, \cdots, \alpha_r)$ in $\Gamma_1 \times \cdots \times \Gamma_r$

$$c(\alpha_1, \cdots, \alpha_r) = g(f_1(\alpha_1, \cdots, \alpha_r), \cdots, f_s(\alpha_1, \cdots, \alpha_r)) \tag{5.7}$$

The recursive processing of input strings of variable length (of the sort needed in lexical analysis and parsing) can be handled by composite functions $\hat{f} : \Gamma^* \rightarrow \Delta^*$, $\hat{g} : \Lambda \times \Gamma^* \rightarrow \Lambda$, and $\hat{c} : \Lambda \times \Gamma^* \rightarrow \Lambda \times \Delta^*$ which are respectively realized by the modular recurrent neural architectures proposed in this chapter and Chapter 6, where Γ^* (Δ^*) denotes the set of all strings over the alphabet Γ (Δ). Here, function $\hat{f}$ denotes the recursive processing of input strings of variable length by a parser or a lexical analyzer (see Chapter 6); function $\hat{g}$ denotes the recursive evaluation of input strings of variable length by the extended transition function of a DFA (see Section 5.2); and function $\hat{c}$ denotes the recursive parsing of syntactically tagged input tokens by the extended transition function of an LR(1) parser (see Chapter 6). The functions $\hat{f}$, $\hat{g}$, and $\hat{c}$ that process input strings of variable length can be composed using symbolic functions f, g, c, output selector function, and string concatenation function by recursion on the length of the input string (See Section 5.2 for an example). Other recursive symbolic functions can also be composed using composition and recursion [140, 154, 195].

The operation of a desired composite function on its symbolic input (string) can be fully characterized analytically in terms of its component symbolic functions on their respective symbolic inputs and outputs. The component symbolic functions are either composite functions of other symbolic functions or primitive symbolic functions which are realized directly by appropriate BMP modules. This makes it possible to systematically (and provably correctly) synthesize any desired symbolic function using BMP modules. (Such designs often require recurrent links for realizing recursive functions such as the extended transition function $\hat{\delta}$ of a DFA or a more complex recursive function as we shall see later and in Chapter 6).

5.2 Neural Network Design for Deterministic Finite Automata (NN DFA)

Deterministic finite automata (finite state machines) are a basic computing model which is essential to many science and engineering applications involving sequence processing. This section first briefly reviews the symbolic computing model for deterministic finite automata and then presents a method to systematically design neural network architectures for realizing deterministic finite automata [20].

5.2.1 Deterministic finite automata (DFA)

A *deterministic finite automaton* is a 5-tuple $M_{DFA} = (Q, \Gamma, \delta, q_0, F)$ [74], 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 $\delta : Q \times \Gamma \rightarrow Q$ is the *transition function*. A finite automaton is deterministic if there is at most one transition that is applicable for each pair of state and input symbol.

The extended transition function $\hat{\delta}$ of a DFA with transition function δ is a mapping from $Q \times \Gamma^*$ to Q defined by recursion on the length of the input string as follows :

- **Basis** : $\hat{\delta}(q_i, \epsilon) = q_i$, where ϵ is empty string.
- **Recursive step** : $\hat{\delta}(q_i, ua) = \delta(\hat{\delta}(q_i, u), a)$ for all input symbols $a \in \Gamma$ and strings $u \in \Gamma^*$.

The computation of the machine M_{DFA} in state q_i with string w halts in state $\hat{\delta}(q_i, w)$. The evaluation of the function $\hat{\delta}(q_0, w)$ simulates the repeated application of the transition

function δ required to process the string w from initial state q_0. A string w is accepted by M_{DFA} if $\hat{\delta}(q_0, w) \in F$; otherwise it is rejected. The set of strings accepted by M_{DFA} is denoted as $L(M_{DFA}) = \{w|\hat{\delta}(q_0, w) \in F\}$, called *the language* of M_{DFA}.

A *Mealy machine* is a DFA augmented with an output function. It is defined by a 6-tuple $M_{Mealy} = (Q, \Gamma, \Delta, \delta, \lambda, q_0)$ [74], where Q, Γ, δ, and q_0 are as in the DFA M_{DFA}, Δ is a finite non-empty *output alphabet*, and λ is output function mapping from $Q \times \Gamma$ to Δ. $\lambda(q, a)$ is the output associated with the transition from state q on input symbol a. The output of M_{Mealy} responding to input string $a_1 a_2 \cdots a_n$ is output string $\lambda(q_0, a_1)\lambda(q_1, a_2) \cdots \lambda(q_{n-1}, a_n)$, where q_0, q_1, ..., q_n is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$.

5.2.2 Architecture of NN DFA

A partially recurrent neural network architecture can be used to realize a DFA as shown in [20]. Its central concept is to use a BMP module to realize the transition function of a DFA. The neural representation in the BMP module is described as follows.

- The input neurons are divided into two groups. One group of input neurons has no recurrent connections and receives the binary coded current input symbol. There are $n = \lceil \log_2 |\Gamma| \rceil$ such input neurons. The second group has $m = \lceil \log_2(|Q| + 1) \rceil$ input neurons and holds the current state (coded in binary). Each input neuron in this group has a recurrent connection from the corresponding output neuron.

- The output neurons together hold the next state (coded in binary). There are $m = \lceil \log_2(|Q| + 1) \rceil$ output neurons.

- Every transition is represented as an ordered pair of binary codes. For each such ordered pair, a hidden neuron and its associated connections are used to realize the ordered pair in terms of binary mapping. Thus the number of required hidden neurons equals the number of valid transitions in the transition function. For example, suppose $p, q \in Q, a \in \Gamma, \delta(p, a) = q$ is a valid transition, and p, q as well as a are encoded as binary codes 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 transition $\delta(p, a) = q$ is represented as a binary

mapping ordered pair ($< p_1, ..., p_m, a_1, ..., a_n >, < q_1, ..., q_m >$) implemented by a BMP module (See Section 2.2).

- An explicit synchronization mechanism is used to support the repetitive evaluation of the transition function δ on input string of variable length.

The transition function of a DFA can be represented as a 2-dimensional table with current state and current input symbol as indices. The operation of such a DFA involves repetitive lookup of the value for next state from the table using current state and current input symbol at each move until an error state or an accepting state is reached. Such a repetitive table lookup process involves content-based pattern matching and retrieval wherein the indices of the table are used as input patterns to retrieve the next state. This process can exploit the massively parallel associative processing capabilities of the neural associative memory proposed in Chapter 2.

Figure 5.1 shows the neural network architecture for realizing a DFA. Let 0, 1, 2, ..., t denote a succession of points along the discrete time line. The current and next states are denoted by $state(t)$ and $state(t+1)$ respectively. The current input symbol is denoted by $input(t)$. 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 $state(t+1)$ which exists only logically but not physically. The first two buffers operate under synchronization control which enforces discrete time 0, 1, ..., t. The *reset* link resets the NN DFA to initial state.

BMP module 1, called *NN DFA transition module*, realizes the transition function of a DFA. BMP module 2 is optional, and it allows the output of the NN DFA to be remapped from the output of BMP module 1. The AST module is optional and can be implemented by a BMP module. It enables BMP module 2 to produce an output only when the NN DFA goes into an accepting state. A connection from the AST module to upper-layer control would be needed to alert it when the AST module traps a rejecting state, i.e., when this NN DFA goes into a rejecting state. Let $< 0^m >$ 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. Note that any

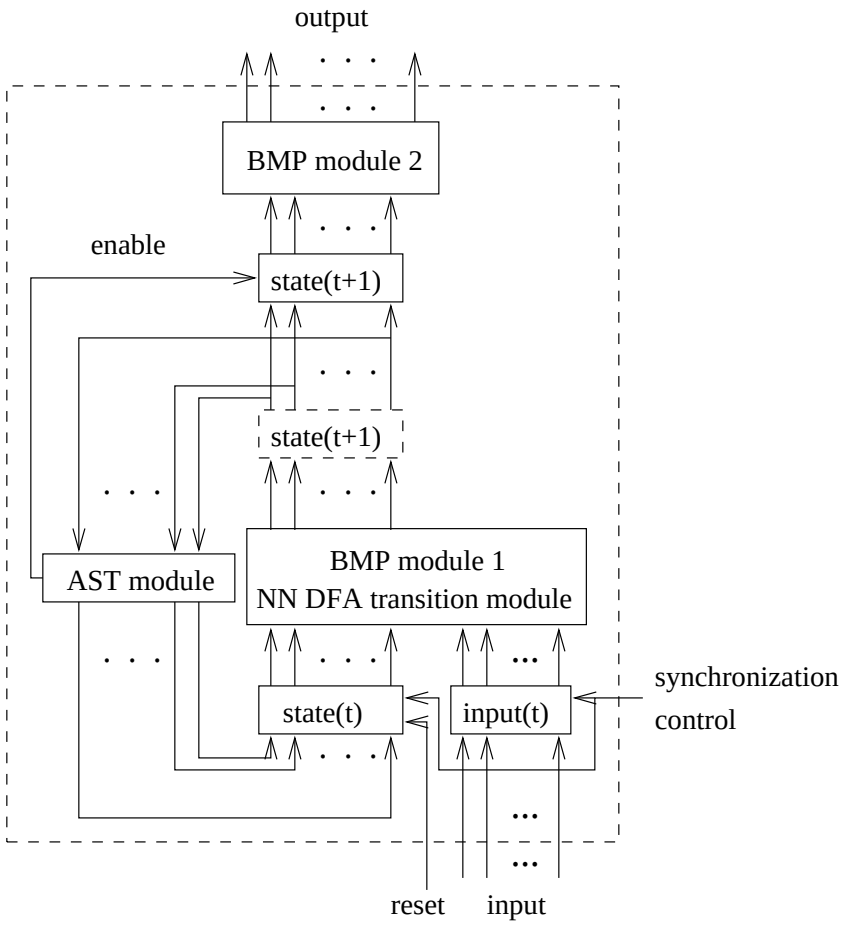

Figure 5.1 The proposed modular neural network architecture for DFA

unspecified transition will automatically have the next state coded as $< 0^m >$ as a consequence of our design of a BMP module (see Section 2.2.5). This simplifies the implementation of a DFA, since any transition to rejecting state does not need to be implemented using a hidden neuron in the NN DFA transition module.

5.3 Neural Network Design for Deterministic Pushdown Automata (NN DPDA)

The capability of DFA is limited to recognition and production of the set of regular languages, the simplest class of languages in Chomsky hierarchy [74]. The capability of DFA can be extended by adding a *stack*. The resulting automata can recognize the set of deterministic context-free languages (DCFL), a more complex and widely used class of languages in Chomsky hierarchy [74]. This section describes a method to systematically synthesize neural network architectures for deterministic pushdown finite automata [20].

5.3.1 Deterministic pushdown automata (DPDA)

A *pushdown automaton* M_{PDA} is a 7-tuple $(Q, \Gamma, \Delta, \delta, q_0, \perp, F)$ [74], where Q is a finite set of *states*, Γ is a finite *input alphabet*, Δ is a finite *stack alphabet*, $q_0 \in Q$ is the *initial state*, $\perp \in \Delta$ is a particular stack symbol called *stack start symbol*, $F \subseteq Q$ is the set of *final states*, and δ is the *transition function* mapping from $Q \times (\Gamma \cup \{\epsilon\}) \times \Delta$ to $Q \times \Delta^*$. 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}. An input string is accepted if the automaton processes the entire string and ends in an accepting state with an empty stack.

For the need of implementing a DPDA in a neural network, we let δ map from $Q \times (\Gamma \cup \{\epsilon\}) \times \Delta$ to $Q \times \{\texttt{pop}, \texttt{push}, \texttt{noop}\} \times (\Delta \cup \{*\})$ to allow stack operation being expressed explicitly during the computation of a DPDA, where $*$ denotes a `don't care` value, {`pop, push, noop`} is the set of possible stack operations, and `noop` denotes no operation.

5.3.2 Architecture of NN DPDA

A partially recurrent neural network architecture can be used to realize a DPDA as shown in [20]. Its central concept is to use a BMP module to realize the transition function of a DPDA. The neural representation in the BMP module is described as follows.

- The input neurons are divided into three groups. The first group has $m = \lceil \log_2(|Q|+1) \rceil$ neurons and holds the binary-coded current state. Each input neuron in this set has a recurrent connection from the corresponding output neuron. The second group receives the binary coded current input symbol and has $n = \lceil \log_2 |\Gamma| + 1 \rceil$ neurons. The third group receives the binary coded stack top symbol and has $k = \lceil \log_2 |\Delta| + 1 \rceil$ neurons. The last two groups have no recurrent connections.

- The output neurons are divided into three groups. The first group represents the binary-coded next state and has $m = \lceil \log_2(|Q|+1) \rceil$ neurons. The second group has two neurons and represents the binary-coded stack operation. The third group has $k = \lceil \log_2 |\Delta| + 1 \rceil$ neurons and represents the binary-coded stack symbol to be pushed into the stack or a `don't care` (denoted as $*$) when the stack action to be performed is a `pop`.

- Every transition is represented as an ordered pair of binary codes. For each such ordered pair, a hidden neuron and its associated connections are used to realize the ordered pair in terms of binary mapping. Thus the number of required hidden neurons equals the number of valid transitions in the transition function. For example, suppose $p, q \in Q, a \in (\Gamma \cup \{\epsilon\}), \alpha, \beta \in \Delta, s \in \{\texttt{pop}, \texttt{push}, \texttt{noop}\}, \delta(p, a, \alpha) = (q, s, \beta)$ 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 $s =< s_1, s_2 >$, where $p_i, q_i, a_j, \alpha_l, \beta_l, s_1, s_2 \in \{0, 1\}$ 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 stack pop/push action explicitly. Stack `push`, `pop`, and `noop` actions are denoted by $s =< 0, 1 >$, $s =< 1, 0 >$, and $s =< 0, 0 >$ respectively. Then the transition $\delta(p, a, \alpha) = (q, s, \beta)$ is represented as the binary mapping ordered pair

$(< p_1, ..., p_m, a_1, ..., a_n, \alpha_1, ..., \alpha_k >, < q_1, ..., q_m, s_1, s_2, \beta_1, ..., \beta_k >)$ to be implemented by a BMP module (See Section 2.2.5).

- An explicit synchronization mechanism is used to support the repetitive evaluation of the transition function on input string of variable length.

The transition function of a DPDA can be represented as a 3-dimensional table with current state, current input symbol, and stack top symbol as indices. The operation of such a DPDA involves repetitive lookup of the value for next state from the table using current state, current input symbol and stack top symbol at each move until an error state or an accepting state is reached. Such a repetitive table lookup process involves content-based pattern matching and retrieval wherein the indices of the table are used as input patterns to retrieve the next state. This process can exploit the massively parallel associative processing capabilities of the neural associative memory proposed in Chapter 2.

Figure 5.2 shows the proposed modular neural network architecture for realizing a DPDA. The current and next states are denoted by $state(t)$ and $state(t+1)$ respectively. 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 current input symbol $input(t)$, another stores stack top symbol $stack_{top}$, and the other stores next state $state(t+1)$ which exists only logically but not physically. The first three buffers operate under synchronization control which enforces discrete time 0, 1, ..., t. The *reset* link resets the NN DPDA to initial state.

BMP module 1, called *NN DPDA transition module*, realizes the transition function of a DPDA. Each state transition is coded as an ordered pair of binary mapping codes. There are two *push/pop* connections from the NN DPDA transition module to the stack mechanism module. These links inform the stack mechanism module whether to pop or push. The AST module is optional and can be implemented by a BMP module. It enables BMP module 2 to produce output only when the NN DPDA goes into an accepting state. A connection from the AST module to upper-layer control would be needed to alert it when the AST module traps a rejecting state, i.e., when this NN DPDA goes into a rejecting state. BMP module 2

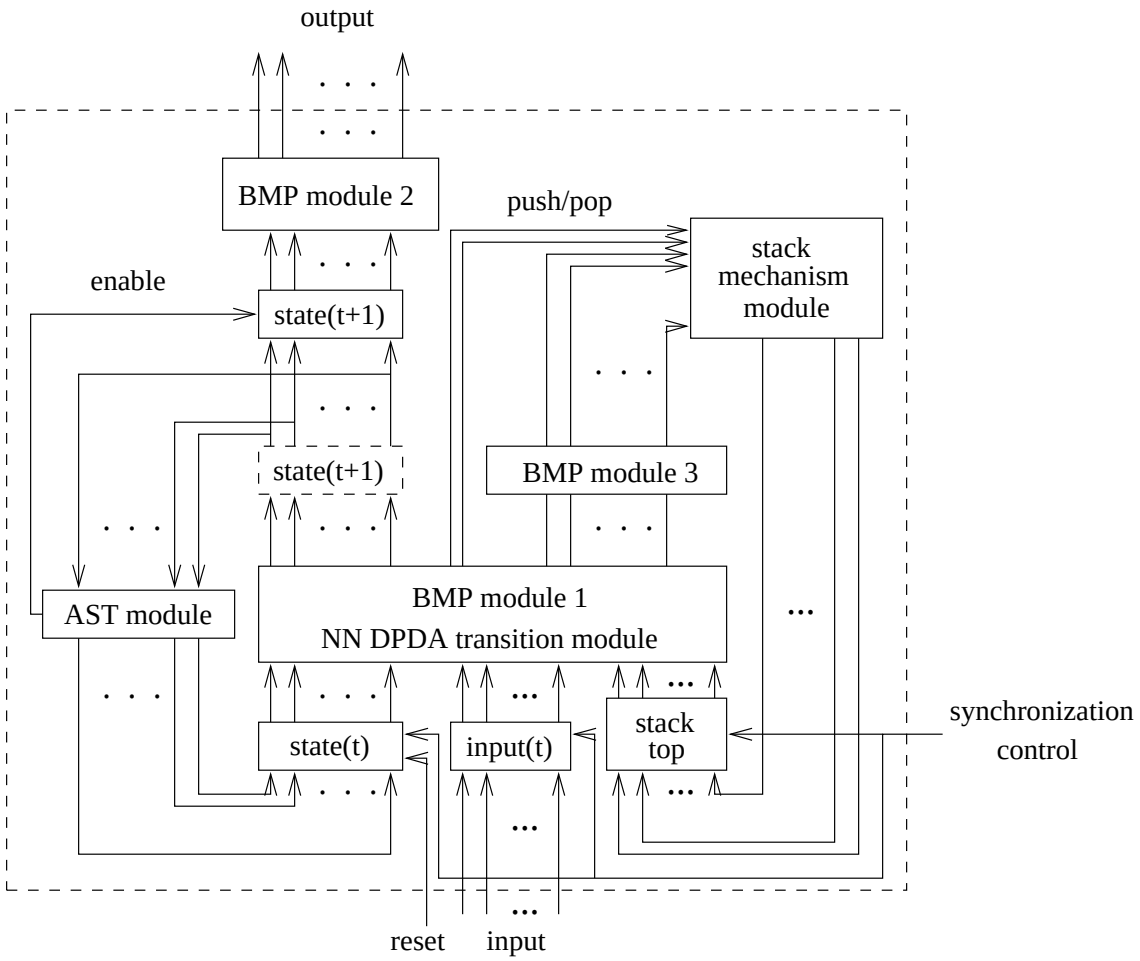

Figure 5.2 The proposed modular neural network architecture for DPDA

is optional, and it allows the output from the NN DPDA to be remapped from the output of the NN DPDA transition module. BMP module 3 is optional and provides remapping of stack symbol produced from the NN DPDA transition module. Note that any unspecified transition will have the next state $< 0^m >$ given our implementation of a BMP module.

5.4 Neural Network Design for Stack (NN Stack)

This section first briefly discuss the symbolic computing model for stack and then presents a method to systematically design neural network architectures for realizing stacks [22].

5.4.1 Symbolic representation of stack

A stack can be coded as a string over a stack alphabet, with its top element at one end of the string and its bottom element at the other end. `Pop` and `push` are the main actions of a stack. In the implementation of a stack, these actions can be performed by a DFA which is augmented with memory to store stack symbols which are accessed sequentially using a *stack top pointer* (SP) which points to the top symbol of the stack. The stack top pointer is maintained by the current state of the DFA, and the current action of the stack by the input to the DFA. Let $A =$ { `pop, push, noop` } be the set of possible stack actions, C the set of possible stack configurations (contents), S the set of stack symbols, $P = \{0, 1, 2, ..., n\}$ the set of possible positions of stack top pointer, and n the maximal depth (capacity) of a given stack. Let $\perp$ be stack bottom symbol and $c \cdot s$ denote the stack configuration after a stack symbol s is pushed onto the stack configuration c. Note that $C = \{\alpha \mid \alpha \in \perp \cdot S^* \; and \; \mid \alpha \mid \leq n\}$, where $\mid \alpha \mid$ denotes the number of stack symbols in the stack configuration α. Assume that the value of stack top pointer doesn't change on a `noop` action, and it is incremented on a `push` action and decremented on a `pop` action. The operation of a stack and the retrieval of stack top symbol from a stack can be characterized by the symbolic functions $f_{Stack} : A \times S \times C \times P \rightarrow C \times P$ and $f_{Top} : C \times P \rightarrow S \cup \{\perp\}$ respectively. They are defined as follows.

$$f_{Stack}(\texttt{push}, s, c, p) = \begin{cases} (c \cdot s, p+1) & \text{if } s \in S,\ c \in C, \\ & \quad p \in P, \text{ and } p \leq n-1 \\ \texttt{error} & \text{otherwise} \end{cases} \tag{5.8}$$

$$f_{Stack}(\texttt{pop}, *, c, p) = \begin{cases} (c', p-1) & \text{if } c \in C \text{ and } c = c' \cdot s \text{ for} \\ & \quad \text{some } s \in S \text{ and some } c' \in C; \\ & \quad \text{and } p \in P \text{ and } p \geq 1 \\ \texttt{error} & \text{otherwise} \end{cases} \tag{5.9}$$

$$f_{Stack}(\texttt{noop}, *, c, p) = \begin{cases} (c, p) & \text{if } c \in C \text{ and } p \in P \\ \texttt{error} & \text{otherwise} \end{cases} \tag{5.10}$$

$$f_{Top}(c, p) = \begin{cases} \perp & \text{if } c = \perp \text{ and } p = 0 \\ s & \text{if } c \in C \text{ and } c = c' \cdot s \text{ for some } s \in S \\ & \quad \text{and some } c' \in C; \text{ and } p \in P \text{ and } \mid c \mid = p \\ \texttt{error} & \text{otherwise} \end{cases} \tag{5.11}$$

where $*$ stands for a `don't care`.

5.4.2 Architecture of NN Stack

This subsection discusses the neural network realization of a stack in terms of symbolic functions f_{Stack} and f_{Top}. A design for NN Stack obtained by adding a *write control module* to an NN DFA is shown in Figure 5.3. (The use of such a circuit might be considered by some to be somewhat unconventional given the implicit assumption of lack of explicit control in many neural network models. However, the operation of most existing neural networks implicitly assume at least some form of control. Given the rich panoply of controls found in biological neural networks, there is no reason not to build in a variety of control and coordination structures into neural networks whenever it is beneficial to do so [71]). NN Stack has an n-bit binary output corresponding to the element popped from the stack, and four sets of binary inputs:

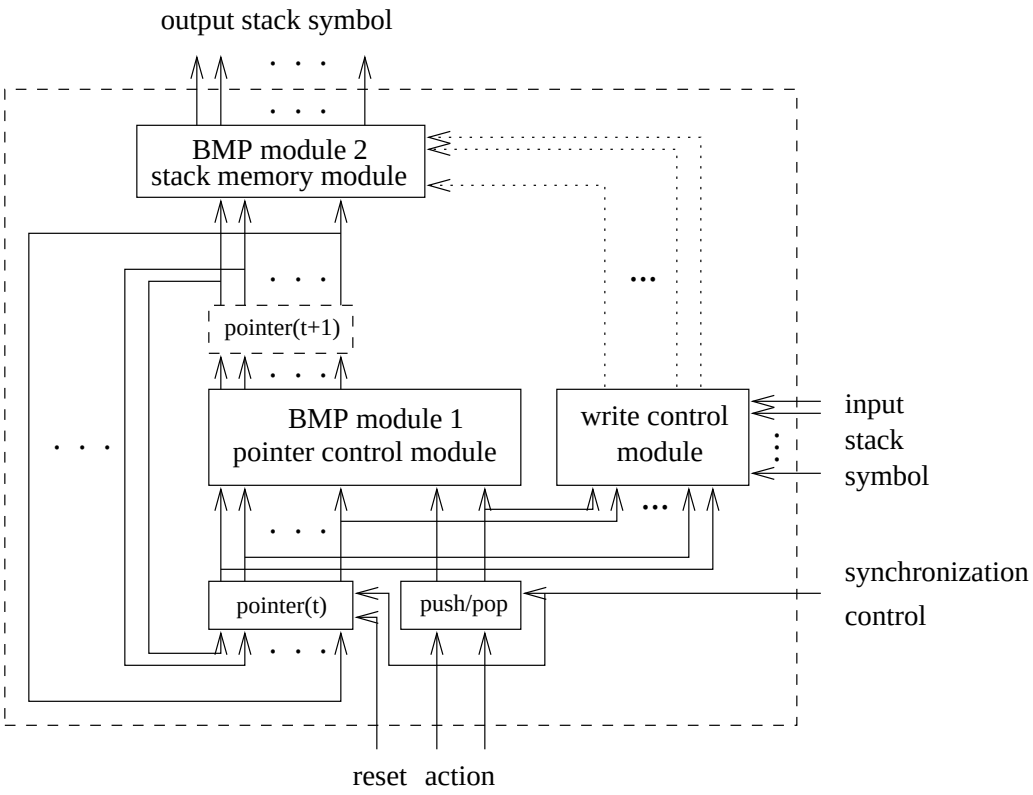

Figure 5.3 The proposed neural network architecture for stack mechanism

- *Reset* which is a 1-bit signal which resets *pointer*(t) (current SP) to point to the bottom of the stack at the beginning.
- *Synchronization control* which is a 1-bit signal that synchronizes NN Stack with the discrete time line, denoted by $0, 1, \cdots, t, t+1, \cdots$.
- *Action* which is a 2-bit binary code so that
  - 01 denotes `push`.
  - 10 denotes `pop`.
  - 00 denotes `no action`.
- *Input stack symbol* which is an n-bit binary code for the symbol to be pushed onto the stack during a stack operation.

An NN Stack consists of a *pointer control module*, a *stack memory module*, a *write control module* and two *buffers*. The first buffer stores current SP value (*pointer*(t)) and the second stores the current stack action (*push/pop*). In Figure 5.3, the dotted box labeled with *pointer*(t+1) exists only logically but not physically, and *pointer*(t) and *pointer*(t+1) respectively denote SP before and after a stack action. SP is coded into an m-bit binary number.

5.4.2.1 Pointer control module

The pointer control module (BMP module 1) realizes a symbolic function $f_{PControl} : A \times P \rightarrow P$ and controls the movement of SP which is incremented on a `push` and decremented on a `pop`. The pointer control module uses $m+2$ input, 3×2^m hidden, and m output neurons. m of the input neurons represent *pointer*(t) (current SP value), and the remaining 2 input neurons encodes the stack action. There are 2^m possible SP values. The m output neurons represent *pointer*(t+1) (the SP value after a stack action). Each change in SP value can be realized by a binary mapping (with one hidden neuron per change). Since `noop` (*no action*) is one of legal stack actions, 3×2^m hidden neurons are used in the pointer control module.

5.4.2.2 Stack memory module

The stack memory module (BMP module 2) realizes the symbolic function f_{Top}. It uses m input neurons, n output neurons, and 2^m hidden neurons which together allow storage of 2^m stack symbols at 2^m SP positions. The stack symbols stored in stack memory module are accessed through *pointer*(t+1) (the output of the pointer control module). Note that the BMP module 2 uses its 2nd-layer connections associated with a hidden neuron to store a symbol (see Chapter 2).

5.4.2.3 Write control module

The write control module (plus stack memory module) realizes a symbolic function f_{SWrite} : $A \times S \times C \times P \rightarrow C$. Physically, it receives m binary inputs from the buffer labeled with *pointer*(t) (denoting current SP), 1 binary input from the second output line of the buffer labeled with `push/pop` (denoting current stack action), and n binary inputs (denoting the stack symbol to be pushed onto the stack) from environment. Stack memory module is used to store current stack configuration. The module does nothing when a *pop* is performed. The n dotted output lines from the write control module write the n-bit binary-coded stack symbol into n of the 2nd-layer connections associated with a corresponding hidden neuron in the stack memory module when a `push` is performed. The hidden neuron and its n associated connections are located by using current SP value (*pointer*(t)). (The processing of stack *overflow* and *underflow* is not discussed here. It has to be taken care of by appropriate error handling mechanisms).

5.4.2.4 Timing considerations

The proposed design for NN Stack shown in Figure 5.3 is based on the assumption that the write control module finishes updating the 2nd-layer connection weights associated with a hidden neuron of stack memory module before the signals from pointer control module are passed to stack memory module during a `push` stack action. If this assumption fails to hold, the original design needs to be modified by adding: n links from input stack symbol (buffer)

to output stack symbol (buffer); an inhibition latch, which is activated by the leftmost output line of the *push/pop* buffer, on the links to inhibit signal passing from input stack symbol (buffer) to output stack symbol (buffer) at a `pop` operation; a second inhibition latch, which is activated by the rightmost output line of the *push/pop* buffer between pointer control module and stack memory module to inhibit signal transmission between these two modules at a `push` operation.

5.4.3 NN Stack in action

This subsection symbolically illustrates how the modules of NN Stack together realize a stack by considering several successive stack actions. Symbolic function f_{Stack} is a composition of symbolic functions $f_{PControl}$ and f_{SWrite} such that $\forall(a, s, c, p) \in A \times S \times C \times P$, $f_{Stack}(a, s, c, p) = (f_{SWrite}(a, s, c, p), f_{PControl}(a, p))$. Consider the following sequence of stack operations:

1. At time $= t_1$, suppose the value of stack top pointer (current SP value) is 4 and the stack action to be performed is a `push` on a stack symbol `a`. Let c_{t_1} be current stack configuration. At this time step, NN Stack computes $f_{Stack}(\texttt{push}, \texttt{a}, c_{t_1}, 4) = (c_{t_1} \cdot \texttt{a}, 5)$ and $f_{Top}(c_{t_1} \cdot \texttt{a}, 5) = \texttt{a}$, i.e.,

   - the pointer control module computes $f_{PControl}(\texttt{push}, 4) = 5$,
   - the write control module (plus stack memory module) computes $f_{SWrite}(\texttt{push}, \texttt{a}, c_{t_1}, 4) = c_{t_1} \cdot \texttt{a}$, and
   - the stack memory module computes $f_{Top}(c_{t_1} \cdot \texttt{a}, 5) = \texttt{a}$.

2. At time $= t_1 + 1$, suppose the stack action to be performed is a `push` on a stack symbol `b`. At this time step, NN Stack computes $f_{Stack}(\texttt{push}, \texttt{b}, c_{t_1} \cdot \texttt{a}, 5) = (c_{t_1} \cdot \texttt{a} \cdot \texttt{b}, 6)$ and $f_{Top}(c_{t_1} \cdot \texttt{a} \cdot \texttt{b}, 6) = \texttt{b}$, i.e.,

   - the pointer control module computes $f_{PControl}(\texttt{push}, 5) = 6$,
   - the write control module (plus stack memory module) computes $f_{SWrite}(\texttt{push}, \texttt{b}, c_{t_1} \cdot \texttt{a}, 5) = c_{t_1} \cdot \texttt{a} \cdot \texttt{b}$, and

- the stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = \mathtt{b}$.

3. At time $= t_1 + 2$, suppose the stack action to be performed is a `pop`. At this time step, NN Stack computes $f_{Stack}(\mathtt{pop}, *, c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = (c_{t_1} \cdot \mathtt{a}, 5)$ and $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$, i.e.,

   - the pointer control module computes $f_{PControl}(\mathtt{pop}, 6) = 5$,
   - the write control module does nothing, and
   - the stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$.

5.5 Neural Network Design for Nondeterministic Finite Automata (NN NFA)

This section explores how to exploit the inherent parallelism and versatile representation in ANN to reduce the operational and implementational time overhead of nondeterministic finite automata (NFA) which are a basic model of symbolic computing in computer science and provide a typical model suitable for the exploration of parallel symbolic computing via ANN. A recurrent neural network (RNN) is systematically synthesized to concurrently track all the possible nondeterministic computations of a given NFA. Such a concurrent breadth-first tracking is facilitated by two types of parallel symbolic computations executed by the proposed RNN. One of the types is parallel content-based pattern matching, and the other is parallel union operations of sets. The RNN acts like a cost-effective SIMD computer system dedicated to the two types of parallel symbolic computations. The proposed RNN is provably correctly assembled from two kinds of neural assemblies. One of the neural assemblies computes a logical `AND`, and the other computes a logical `OR`.

Although the concept of nondeterministicism embedded in NFA provides an elegantly simple and intuitive description for sequence processing, it results in much computational and implementational overhead in single-CPU computer systems. Thus the concept of nondeterministicism in NFA, which plays a central role in both the theory of languages and the theory of computation [74], provides a typical model suitable for the exploration of parallel symbolic

computing via neural networks. The reduced operation time complexity of NFA realized by the proposed RNN is due to the parallel operations of the neural assemblies in the RNN.

It is well known that DFA and NFA are equivalent, and every NFA can be converted into its equivalent DFA [74]. NFA 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, NFA have a variety of practical applications in computer science, linguistics, systems modeling and control, and artificial intelligence; and NFA are simpler and more intuitive to design than their equivalent DFA for a given task due to the powerful concept of nondeterministicism embedded in NFA, especially for pattern matching [195]. NFA are rarely directly implemented in conventional computer systems because the nondeterministicism in NFA causes operational and implementational overhead. Usually, they are converted into their equivalent DFA for implementation. So, for 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 in which the power of nondeterministicism in NFA is retained, and there is no need to convert an NFA into its equivalent DFA before its construction. Note also that every DFA is an NFA. Therefore, the proposed RNN can be used as a general neural architecture for realizing finite automata including DFA and NFA.

5.5.1 Nondeterministic finite automata (NFA)

A *nondeterministic finite automaton* M_{NFA} is a 5-tuple $(Q, \Gamma, \delta', q_0, F)$ [74], 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}.

5.5.1.1 Advantages of NFA for applications

It is well known that NFA and DFA are equivalent [74]. 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 [74]. However, an NFA is usually simpler and more intuitive to design than its equivalent DFA for a given language due to the powerful concept of nondeterministicism inherent in NFA. Figures 5.4 and 5.5 respectively show the state diagrams of an NFA and its equivalent DFA. Both of them accept input strings that contain the sub-string `abaa` [195]. These two automata are equivalent, but apparently the language the NFA accepts is much easier to understand. 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 their transition functions. 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* for state q_j on input symbol a. Note 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.

5.5.2 Model for concurrently tracking all the possible nondeterministic moves in the operation of an NFA using RNN

The deterministic and linear-time operation of a given NFA which is realized by the proposed RNN can be modeled conventionally by its equivalent DFA which is constructed according to *Subset Construction* algorithm [195]. The main idea of Subset Construction is to concurrently track all the possible states that can be reached at each step of an NFA. In the computation of an NFA, tracking all the reachable states at each step induces much overhead in single-CPU computer systems, whereas the proposed RNN efficiently computes in parallel, all the reachable states at each step by exploiting the parallelism of ANN. In Subset Construction, 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

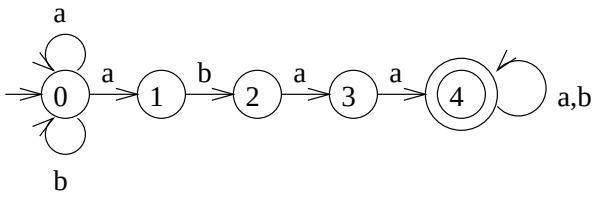

Figure 5.4 The state diagram of an NFA that accepts any input string containing the sub-string `abaa`

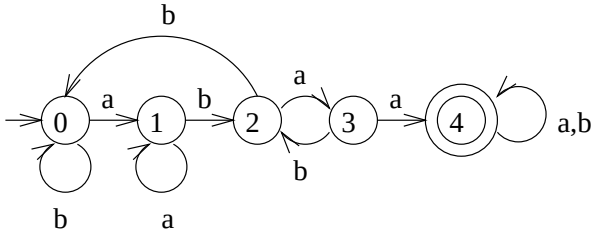

Figure 5.5 The state diagram of a DFA that accepts any input string containing the sub-string `abaa`

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 $\delta'' : 2^Q \times \Gamma \rightarrow 2^Q$ is defined by

$$Q_j = \delta''(Q_i, a), \texttt{ if } Q_j = \cup_{q \in Q_i} \delta'(q, a) \texttt{ for all } Q_i \subseteq Q \ \& \ a \in \Gamma \tag{5.12}$$

One main problem with Subset Construction, which views every Q_i as an individual state 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* [195] 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 drawback of both subset algorithms is the bookkeeping overhead associated with maintaining δ'', F'', Q^*, δ^*, and F^*, which are derived from M_{NFA}. The direct realization of a given NFA by the proposed RNN avoids this problem since the transition function module of

the proposed NN NFA captures the regularity of δ'' in expression 5.12 via δ' without actually knowing in advance the legal transitions defined by δ''. Such simplification is partly facilitated by representationally viewing every Q_i as an individual set of states denoted by localist neural representation. The transition module of the proposed NN NFA 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 module of the proposed NN NFA during input processing, i.e., only the states in Q^* will appear in the proposed transition module during input processing. Hereafter, we only discuss M^*_{DFA} instead of M''_{DFA}.

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 by M_{NFA} (M^*_{DFA}) at time t (current time), and $Q_{act}(t+1)$ the *next set of active states* which corresponds to the set of reachable states from q_0 at time $t+1$. $Q_{act}(t)$ and $Q_{act}(t+1)$ are derived recursively from $Q_{act}(0)$ by expression $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q, a)$ during 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 during the processing of the input string are bounded by a same set of states, the number of reachable states at each step does not proliferate indefinitely or exponentially during the processing of the input string, and thus the nondeterministicism shown during the processing of the input string is *globally bounded*. The proposed RNN directly constructs a given NFA M_{NFA} without the need to convert the given NFA into its equivalent DFA M^*_{DFA}. It concurrently tracks all the possible moves during 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 5.12, the transition function δ^* and every move of M^*_{DFA} can be characterized by

$$\forall a \in \Gamma \forall t \geq 0 \, [Q_{act}(t+1) = \delta^*(Q_{act}(t), a)] \tag{5.13}$$

where $Q_{act}(0) = \{q_0\}$ and $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q, a)$

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 of which computes the

sets of reachable states from every state in $Q_{act}(t)$ and the other of which computes the union of the sets of the reachable states. In the proposed NN NFA, the former is computed by the first layer of the transition module of the proposed NN NFA by parallel content-based pattern matching and the later by the second layer by parallel logical `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.

5.5.3 Architecture of NN NFA

This subsection describes the symbolic and neural representations in the proposed NN NFA, and presents a method for assembling the proposed NN NFA using the neural assemblies proposed in Chapter 4

Figure 5.6 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 *NN NFA transition 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 NN NFA transition module to the buffer storing $Q_{act}(t)$ (which could be part of the input neurons of the NN NFA transition module, depending on the applications implemented). One buffer stores current set of active states $Q_{act}(t)$, another buffer (which could also be part of the input neurons of the NN NFA transition module, depending on applications implemented) stores current input symbol $a(t)$, and the other buffer (which exists only logically but not physically) represents the next set of active states $Q_{act}(t+1)$. The first two buffers are under centralized synchronization control which enforces discrete time 0, 1, ..., t, t+1, ... The link "`reset`" resets the NN NFA to its initial set of active states.

5.5.3.1 Symbolic representation in the NN NFA transition module

The realization of the symbolic function δ^* by the NN NFA transition module is central to the construction of the proposed NN NFA. The notations used here follow those described in

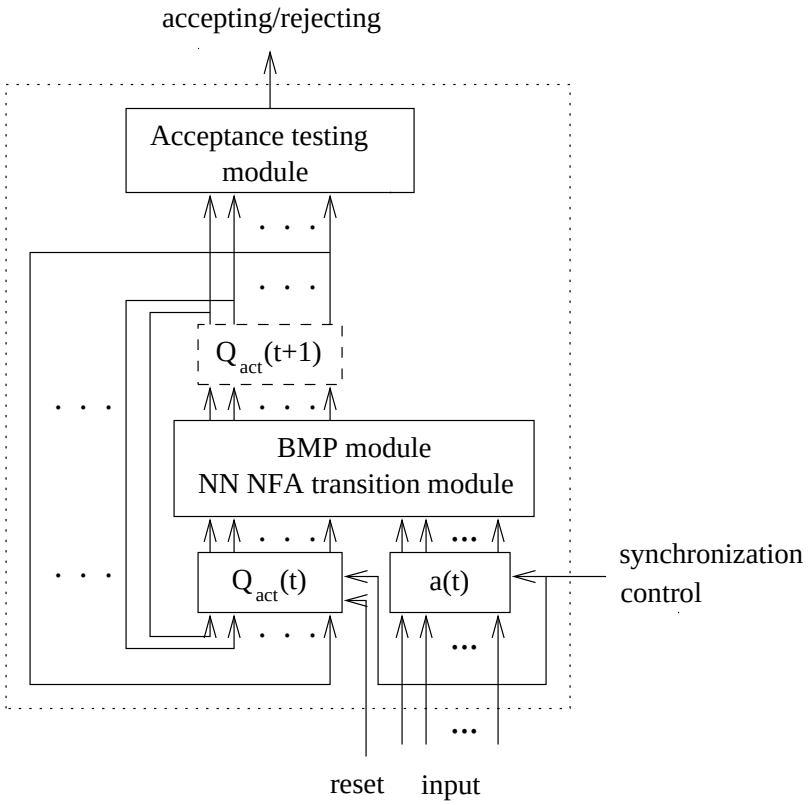

Figure 5.6 The proposed recurrent neural network architecture for concurrently tracking all the nondeterministic computations of a given NFA

previous subsections. The symbolic representations of the NN NFA transition module which is a 2-layer Perceptron are described as follows.

- The output from every neuron and the input to every input neuron are binary value.

- Every transition defined by the transition function δ^* (expression 5.13) is represented as an ordered binary mapping pair $< Q_{act}(t) \times a(t), Q_{act}(t+1) >$ stored in the NN NFA transition module.

- The input neurons together denote $Q_{act}(t) \times a(t)$ and are divided into two groups. One group uses a distributed representation and the other uses a local representation. The former group has no recurrent connection and denotes the binary-coded current input symbol $a(t)$. There are $\lceil \log(\mid \Gamma \mid +1) \rceil$ such input neurons. 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 hidden neurons along with their associated 1st-layer connections are used to concurrently recognize all the applicable transitions for the states in the current set of active states on current input symbol. The hidden layer uses a local representation, and one hidden neuron is used for one uniquely defined transition. The number of activated hidden neurons at each step of the proposed NN 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 which together denote the next set of active states.

- The output layer uses a 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 (which get their input from

the hidden neurons) 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 reached by the states in the current set of active states on current input symbol).

- The recurrent connections from the output neurons to part of the input neurons facilitate the continuous execution of the proposed NN NFA.

5.5.3.2 Neural representation in the NN NFA transition 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 NN NFA transition module. Then the NN NFA transition module has $(n_A + n_I)$ input neurons, n_T hidden neurons, and n_A output neurons. The hidden neurons along with their associated 1st-layer connections are used to identify all the transitions applicable for the states in 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 NN NFA transition module is constructed directly from the transition function δ' of the given NFA M_{NFA}.

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 NN NFA transition 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. 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 $0 \leq i \leq n_A - 1$.

5.5.3.3 Parallel symbolic computations in the NN NFA transition module

The realization of every move (the move from $Q_{act}(t)$ to $Q_{act}(t+1)$) of M^*_{DFA} in the NN NFA transition module can be reasoned in two steps. The first step is computed by the hidden neurons which serve as parallel recognizers of multiple input sub-patterns, and the second step is computed by the output layer which serves as a parallel union operator of sets.

1. The hidden layer consists of a fixed number of neural assemblies which operate in parallel and independently of each other. Thus the hidden neurons serve as parallel recognizers of multiple sub-patterns contained in the input. Such a neural assembly (equivalent to an `AND` neural assembly) for partial pattern recognition is proposed in Section 4.2. Each hidden neuron h and its associated 1st-layer connections serve as a neural assembly for recognizing a certain $J_i^{n_I+1}$-set partial input pattern. Each 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 $J_i^{n_I+1}$-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 4.2. 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 for the states in $Q_{act}(t)$ on current input symbol, and thus they operate together like a simplified, cost-effective SIMD computer system dedicated to parallel partial pattern matching.

2. The output layer consists of a fixed number of the monotone `OR` neural assemblies proposed in 4.4. Each of the assemblies computes a logical `OR` operation in parallel with each other on shared inputs. Each output neuron and its associated 2nd-layer connections compose such a neural assembly. Each such neural assembly is used to check for a certain state q_j whether any of its fan-in transitions is applicable for the states in $Q_{act}(t)$ 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, 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 operate

together to compute $Q_{act}(t+1)$ like a simplified, 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., each of them is viewed as a set of states), the NN NFA transition module realizes the transition function δ' of the given NFA M_{NFA} if it is restricted that $\mid Q_{act}(t) \mid = 1$. Note that δ' maps from $Q \times \Gamma$ to 2^Q. Such local representations facilitate the parallel recognition of all the transitions applicable for the states in $Q_{act}(t)$ on current input symbol even if $\mid Q_{act}(t) \mid \geq 1$. Thus the representations facilitate the concurrent tracking of all the 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., each of them is viewed as a state), the NN NFA transition module realizes the transition function δ^* of M^*_{DFA}. It means that the NN NFA transition module concurrently realizes the transition functions δ' and δ^*. Such a concurrent realization facilitates not only the direct construction of the NN NFA transition module from the transition function δ' but also the linear operation time complexity of the proposed NN NFA for the processing of input strings.

5.5.3.4 Settings of connection weights and thresholds in the NN NFA transition module

Note that every transition defined by expression 5.13 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 NN NFA transition 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 identify the transition (q_i, a, q_j) which is applicable on an input containing the partial input pattern $\{q_i\} \times a$ in the NN NFA transition module, the 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 $J_i^{n_I+1}$-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 4.2, 4.16 and their corresponding Perceptron implementation, a hidden neuron h is created, and its associated connection weights as well as threshold are set

for every transition (q_i, a, q_j) of the given NFA M_{NFA} in the NN NFA transition module as follows :

1. In the 1st-layer connections, according to expression 4.2,
   - the connection weight from the $(i+1)$th input neuron to the hidden neuron h is set to 1,
   - the connection weight from the (n_A+k)th input neuron to the hidden neuron is set to $2a_k - 1$ for $1 \leq k \leq n_I$, and
   - the connection weights from other input neurons (which are not in $J_i^{n_I+1}$) to the hidden neuron are set to 0.
2. The threshold of the hidden neuron is set to $\sum_{k=1}^{n_I} a_k + 1$.
3. In the 2nd-layer connections,
   - the connection weight from the hidden neuron to the $(j+1)$th output neuron is set to 1, and
   - the connection weights from the hidden neuron to other output neurons are set to 0.
4. The thresholds of all output neurons are set to 1 in the NN NFA transition module.

Therefore, if $q_i \in Q_{act}(t)$ and a is current input symbol; then $u_{i+1} = 1$, $u_{n_A+k} = a_k$ for $1 \leq k \leq n_I$ at time t, an input containing sub-pattern $u(J_i^{n_I+1}) =< 1, a_1, ..., a_{n_I} >$ is identified by hidden neuron h and its associated 1st-layer connections, 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 thus $q_j \in Q_{act}(t+1)$) according to above settings. So, the transition (q_i, a, q_j) is applied in the NN NFA transition module.

5.5.3.5 Settings of connection weights and thresholds in the acceptance testing module

The acceptance testing module of the proposed NN NFA tests whether an input string is

accepted by the NN 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 monotone logical `OR` operation (expression 4.17) on the values of the neurons denoting accepting states, and hence it can be realized by a Perceptron according to expression 4.16 with 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 to 1 for $0 \leq i \leq n_A - 1$. Otherwise it is set to 0.
- The threshold of the output neuron is set to 1.

5.5.3.6 The end-of-input testing module

In the proposed NN 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) which recognizes the *end-of-input symbol* \$ that is encoded as binary vector $< 1^{n_I} >$. By expression 4.1 and its corresponding Perceptron implementation, all connection weights are set to 1 and the threshold at output neuron is set to n_I in the 1-layer/1-output Perceptron to recognize \$. The end-of-input testing module is not shown in Figure 5.6.

5.5.3.7 Operation time complexity of the proposed NN NFA

The time complexity of processing an input string of length n by an NFA directly implemented in single-processor computer systems is $O(m^2 n)$ [163], where m is the number of states in the NFA. The proposed NN NFA concurrently tracks all the possible nondeterministic transitions during the processing of an input string for a given NFA by exploiting the inherent parallelism in ANN. In such a computation, the proposed NN NFA retains the powerful concept of nondeterministicism of NFA, and it also has the advantage of DFA which run in linear time proportional to the length of the input string. Since the NN NFA transition module

realizes both the transition functions δ' and δ^*, the time complexity of processing an input string by such a parallel and deterministic computation in the proposed NN NFA 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 NN NFA is $O(n)$. Therefore the computational overhead of input processing due to the nondeterministicism in NFA can be eliminated by taking advantage of the inherent parallelism of ANN as shown by the proposed NN NFA.

5.5.4 Proof of correctness

This subsection proves the correctness in the construction of the proposed NN NFA for a given NFA.

Theorem 5.1 : The proposed NN NFA can correctly realize a given NFA $M_{NFA} = (Q, \Gamma, \delta', q_0, F)$.

Proof : The theorem is proved by showing that the NN NFA transition module of the proposed NN NFA correctly realizes the transition function δ^* of M^*_{DFA} which is re-defined from the given NFA M_{NFA} according to Iterative Subset Construction (please refer to [195] for the proof of equivalence between a given NFA M_{NFA} and its equivalent DFA M^*_{DFA}). Note that every transition defined by the transition function δ^* 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 NN NFA transition module. The mappings are implemented in the NN NFA transition module without knowing in advance every possible mapping pair (legal transition) in transition function δ^*. Instead, they are realized 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, the first layer of which consists of `AND` neural assemblies and the second layer of which consists of `OR` neural assemblies.

All the notations and representations here follow previous subsections. The transition function δ^* is defined by expression 5.13 as follows :

$$\forall a \in \Gamma \forall t \geq 0 \; [Q_{act}(t+1) = \delta^*(Q_{act}(t), a)]$$

where $Q_{act}(0) = \{q_0\}$ and $Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q, a)$. Since $Q_{act}(t+1)$ is computed from $Q_{act}(t)$ and a, the above expression can be denoted as following :

$$\forall a \in \Gamma \; \forall t \geq 0 \; [Q_{act}(t+1) = \cup_{q \in Q_{act}(t)} \delta'(q, a) = \{q_j \mid q_j \in \delta'(q_i, a) \; \& \; q_i \in Q_{act}(t)\}] \quad (5.14)$$

where $Q_{act}(0) = \{q_0\}$. Expression 5.14 is equivalent to

$$\forall a \in \Gamma \; \forall t \geq 0 \; [\forall i \; \forall j \; q_i \in Q_{act}(t) \; \& \; q_j \in \delta'(q_i, a) \Longrightarrow q_j \in Q_{act}(t+1)\,] \qquad (5.15)$$

where $Q_{act}(0) = \{q_0\}$. We want to show that the NN NFA transition module realizes the transition function δ^* of M^*_{DFA} by proving that expression 5.15 is preserved by the NN NFA transition module.

The NN NFA transition module is represented in such a way (see Section 5.5.3) that the conditions $a \in \Gamma$ and $0 \leq i, j \leq n_A - 1$ always hold in expression 5.15. Let $u =< u_1, ..., u_{n_A+n_I} >$ and $v =< v_1, ..., v_{n_A} >$ be respectively the binary input value and output value of the NN NFA transition module, and $a \in \Gamma$ be current input symbol encoded as a binary value $< a_1, ..., a_{n_I} >$, where $a_k \in \{0, 1\}$ for $1 \leq k \leq n_I$. Let $w^1_{k,i}$, $w^2_{j,k}$, θ^1_k, and θ^2_j respectively denote the 1st-layer connection weight from input neuron i to hidden neuron k, the 2nd-layer connection weight from hidden neuron k to output neuron j, the threshold of hidden neuron k, and the threshold of output neuron j in the NN NFA transition module, where $1 \leq i \leq n_A + n_I$, $1 \leq k \leq n_T$ and $1 \leq j \leq n_A$.

For all i and j $(0 \leq i, j \leq n_A - 1)$, if $q_j \in \delta'(q_i, a)$, a hidden neuron h with an interest set $J_i^{n_I+1} = \{i+1, n_A+1, n_A+2, ..., n_A+n_I\}$ should have been created by the proposed method presented in Section 5.5.3 for the transition (q_i, a, q_j) for recognizing the $J_i^{n_I+1}$-set partial input value $u(J_i^{n_I+1}) =< 1, a_1, ..., a_{n_I} >$, denoting $\{q_i\} \times a$, by following settings :

- $\theta^1_h = \sum_{k=1}^{n_I} a_k + 1$,
- $w^1_{h,i+1} = 1$
- $w^1_{h,k} = 0$ for $1 \leq k \leq n_A \; \& \; k \neq (i+1)$ (where $k \notin J_i^{n_I+1}$),
- $w^1_{h,k} = 2a_k - 1$ for $n_A + 1 \leq k \leq n_A + n_I$ (where $k \in J_i^{n_I+1}$),
- $\theta^2_j = 1$,
- $w^2_{j+1,h} = 1$, and
- $w^2_{m,h} = 0$ for $1 \leq m \leq n_A \; \& \; m \neq (j+1)$.

For any moment $t \geq 0$, if $q_i \in Q_{act}(t)$ (i.e. $u_{i+1} = 1$), the $J_i^{n_I+1}$-set partial input vector $u(J_i^{n_I+1}) =< 1, a_1, ..., a_{n_I} >$, denoting $\{q_i\} \times a$, is recognized and hidden neuron h is activated when the input u denoting $Q_{act}(t) \times a$ is fed into the NN NFA transition module. The hidden neuron h in turn activates output neuron $j+1$. So $v_{j+1} = 1$ at time $t+1$, and thus $q_j \in Q_{act}(t+1)$. Therefore expression 5.15, i.e., expression 5.13, is preserved by the NN NFA transition module. .....$\Diamond$

5.5.5 NN NFA in Action

This subsection constructs an NN NFA transition module of the proposed NN NFA for the NFA defined in Figure 5.4 (see Section 5.5.1). In the NFA, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, q_0 is initial state, $\Gamma = \{\mathtt{a}, \mathtt{b}\}$, and $F = \{q_4\}$. The transitions defined in the NFA are (q_0, a, q_0), (q_0, b, q_0), (q_0, a, q_1), (q_1, b, q_2), (q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_4), and (q_4, b, q_4). Then $n_T = 8$, $n_I = 2$, and $n_A = 5$. The NN NFA transition module has 7 input, 8 hidden, and 5 output neurons. Let $w^1_{k,i}$, $w^2_{j,k}$, θ^1_k, and θ^2_j respectively denote the 1st-layer connection weight from input neuron i to hidden neuron k, the 2nd-layer connection weight from hidden neuron k to output neuron j, the threshold of hidden neuron k, and the threshold of output neuron j in the NN NFA transition module, where $1 \leq i \leq 7$, $1 \leq k \leq 8$ and $1 \leq j \leq 5$. Let input symbol $\mathtt{a}$ be encoded as $< 0, 0 >$ and $\mathtt{b}$ as $< 0, 1 >$. Then the connection weights and neuron thresholds of the NN NFA transition module are set as follows :

$\theta^1_1 = 1$, $w^1_{1,1} = 1$, $w^1_{1,2} = 0$, $w^1_{1,3} = 0$, $w^1_{1,4} = 0$, $w^1_{1,5} = 0$, $w^1_{1,6} = -1$, $w^1_{1,7} = -1$,

$\theta^1_2 = 2$, $w^1_{2,1} = 1$, $w^1_{2,2} = 0$, $w^1_{2,3} = 0$, $w^1_{2,4} = 0$, $w^1_{2,5} = 0$, $w^1_{2,6} = -1$, $w^1_{2,7} = 1$,

$\theta^1_3 = 1$, $w^1_{3,1} = 1$, $w^1_{3,2} = 0$, $w^1_{3,3} = 0$, $w^1_{3,4} = 0$, $w^1_{3,5} = 0$, $w^1_{3,6} = -1$, $w^1_{3,7} = -1$,

$\theta^1_4 = 2$, $w^1_{4,1} = 0$, $w^1_{4,2} = 1$, $w^1_{4,3} = 0$, $w^1_{4,4} = 0$, $w^1_{4,5} = 0$, $w^1_{4,6} = -1$, $w^1_{4,7} = 1$,

$\theta^1_5 = 1$, $w^1_{5,1} = 0$, $w^1_{5,2} = 0$, $w^1_{5,3} = 1$, $w^1_{5,4} = 0$, $w^1_{5,5} = 0$, $w^1_{5,6} = -1$, $w^1_{5,7} = -1$,

$\theta^1_6 = 1$, $w^1_{6,1} = 0$, $w^1_{6,2} = 0$, $w^1_{6,3} = 0$, $w^1_{6,4} = 1$, $w^1_{6,5} = 0$, $w^1_{6,6} = -1$, $w^1_{6,7} = -1$,

$\theta^1_7 = 1$, $w^1_{7,1} = 0$, $w^1_{7,2} = 0$, $w^1_{7,3} = 0$, $w^1_{7,4} = 0$, $w^1_{7,5} = 1$, $w^1_{7,6} = -1$, $w^1_{7,7} = -1$,

$\theta^1_8 = 2$, $w^1_{8,1} = 0$, $w^1_{8,2} = 0$, $w^1_{8,3} = 0$, $w^1_{8,4} = 0$, $w^1_{8,5} = 1$, $w^1_{8,6} = -1$, $w^1_{8,7} = 1$,

$\theta^2_j = 1$ for $1 \leq j \leq 5$,

$w^2_{1,1} = 1$, $w^2_{1,2} = 1$, $w^2_{1,3} = 0$, $w^2_{1,4} = 0$, $w^2_{1,5} = 0$, $w^2_{1,6} = 0$, $w^2_{1,7} = 0$, $w^2_{1,8} = 0$,
$w^2_{2,1} = 0$, $w^2_{2,2} = 0$, $w^2_{2,3} = 1$, $w^2_{2,4} = 0$, $w^2_{2,5} = 0$, $w^2_{2,6} = 0$, $w^2_{2,7} = 0$, $w^2_{2,8} = 0$,
$w^2_{3,1} = 0$, $w^2_{3,2} = 0$, $w^2_{3,3} = 0$, $w^2_{3,4} = 1$, $w^2_{3,5} = 0$, $w^2_{3,6} = 0$, $w^2_{3,7} = 0$, $w^2_{3,8} = 0$,
$w^2_{4,1} = 0$, $w^2_{4,2} = 0$, $w^2_{4,3} = 0$, $w^2_{4,4} = 0$, $w^2_{4,5} = 1$, $w^2_{4,6} = 0$, $w^2_{4,7} = 0$, $w^2_{4,8} = 0$,
$w^2_{5,1} = 0$, $w^2_{5,2} = 0$, $w^2_{5,3} = 0$, $w^2_{5,4} = 0$, $w^2_{5,5} = 0$, $w^2_{5,6} = 1$, $w^2_{5,7} = 1$, $w^2_{5,8} = 1$.

Note that this NN NFA starts from initial state $Q_{act}(0) = \{q_0\}$ which is encoded as $< 1, 0, 0, 0, 0 >$.

5.6 Summary and Discussion

In conventional computer systems, computer programs for real world applications are usually large and complex. They are typically built from a set of pre-defined modules (or objects/classes in object-oriented paradigms). Such modules allow code reuse and rapid implementation with fewer errors. We advocate a similar approach to the construction of complex neural networks. In this chapter, we have constructed a given DFA using an RNN which in turn is assembled from BMP modules and recurrent links; a stack using an RNN which is assembled from BMP modules, recurrent links and a write control module; a given DPDA using an RNN which is assembled from BMP modules, an NN Stack and recurrent links; and a given NFA using an RNN which is assembled from basic neural assemblies that realize logical `AND` and `OR` operations on Boolean variables. The proposed NN NFA demonstrates the potential benefits of ANN in the design of high performance systems for parallel symbolic computing applications. Our other attempts for more complex symbolic processing includes neural networks designed respectively for simple database query processing (see Chapter 3) and syntax analysis (see Chapter 6). We expect a similar approach to be applicable in the construction of neural networks for a variety of important applications.

6 NEURAL ARCHITECTURES FOR SYNTAX ANALYSIS

6.1 Introduction

This chapter explores the synthesis of neural architectures for syntax analysis using pre-specified grammars — a prototypical symbol processing task with applications in interactive programming environments (using interpreted languages such as LISP and JAVA), analysis of symbolic expressions (e.g., in real-time knowledge based systems and database query processing), and high-performance compilers. This chapter does not address machine learning of unknown grammars (which finds applications in tasks such as natural language acquisition).

A more general goal of this chapter is to explore the design of massively parallel architectures for symbol processing using the neural associative memories proposed in Chapter 2 as key components. Pattern-directed associative inference is an essential part of most AI systems [54, 97, 181] and dominates the computational requirements of many AI applications [55, 97, 127].

The proposed high performance neural architectures for syntax analysis are systematically (and provably correctly) synthesized through composition of the necessary symbolic functions using a set of component symbolic functions each of which is realized using a neural associative processor (memory). It takes advantage of massively parallel pattern matching and retrieval capabilities of neural associative processors (memories) to speed up syntax analysis for real-time applications. The rest of the chapter is organized as follows:

- The remainder of Section 6.1 reviews related research on neural architectures for syntax analysis.

- Sections 6.2 and 6.3 respectively develop modular neural network architectures for lexical

analysis and parsing.

- Section 6.4 compares the estimated performance of the proposed neural architectures for syntax analysis (based on current CMOS VLSI technology) with that of commonly used approaches to syntax analysis in conventional computer systems that rely on inherently sequential index or matrix structure for pattern matching.
- Section 6.5 concludes with a summary and discussion.

6.1.1 Review of related research on neural architectures for syntax analysis

To the best of our knowledge, to date, most of the research on neural architectures for syntax analysis has focused on the investigation of neural networks that are designed to *learn* to parse particular classes of syntactic structures (e.g., strings from deterministic context-free languages (DCFL) or natural language sentences constructed using limited vocabulary). Notable exceptions are: connectionist realizations of Turing Machines (wherein a stack is simulated using binary representation of a fractional number) [143, 169]; a few neural architectures designed for parsing based on a known grammar [34, 164]; and neural network realizations of finite state automata [20, 134]. Nevertheless, it is informative to examine the various proposals for neural architectures for syntax analysis (regardless of whether the grammar is preprogrammed or learned). The remainder of this subsection explores some of the proposed architectures in the literature for syntax analysis in terms of how each of them addresses the key subtasks of syntax analysis.

[34] proposes a neural network to parse input strings of fixed maximum length for known context-free grammars (CFG). The whole input string is presented at one time to the neural parser which is a layered network of logical AND and OR nodes with connections set by an algorithm based on CYK algorithm [74].

PARSEC [80] is a modular neural parser consisting of six neural network modules. It transforms a semantically rich and therefore fairly complex English sentence into three output representations produced by its respective output modules. The three output modules are *role labeler* which associates case-role labels with each phrase block in each clause, *interclause*

labeler which indicates subordinate and relative clause relationships, and *mood labeler* which indicates the overall sentence mood (declarative or interrogative). Each neural module is trained individually by a variation of Backpropagation algorithm. The input is a sequence of syntactically as well as semantically tagged words in the form of binary vectors and is sequentially presented to PARSEC, one word at a time. PARSEC exploits generalization as well as noise tolerance capabilities of neural networks to reportedly attain 78% correct labeling on a test set of 117 sentences when trained with a training set of 240 sentences. Both the test and training sets were based on conference registration dialogs from a vocabulary of about 400 words.

SPEC [116] is a modular neural parser which parses variable-length sentences with embedded clauses and produces case-role representations as output. SPEC consists of a *parser* which is a simple recurrent network, a *stack* which is realized using a recursive auto-associative memory (RAAM) [144], and a *segmenter* which controls the *push/pop* operations of the *stack* using a 2-layer Perceptron.

RAAM has been used by several researchers to implement stacks in connectionist designs for parsers [13, 66, 116]. A RAAM is a 2-layer Perceptron with recurrent links from hidden neurons to part of input neurons and from part of output neurons to hidden neurons. The performance of a RAAM stack is known to degrade substantially with increase in depth of the stack, and the number of hidden neurons needed for encoding a stack of a given depth has to be determined through a process of trial and error [116]. A RAAM stack has to be trained for each application. Other drawbacks associated with the use of RAAM as a stack are discussed in [174].

Each module of SPEC is trained individually using Backpropagation algorithm to approximate a mapping function as follows: Let Q be a finite non-empty set of *states*, Γ a finite non-empty *input alphabet*, V_{CRV} a finite non-empty set of case-role vectors, $A = \{output, push, pop\}$ the set of stack actions, and V_{stack} the set of compressed stack representations at the hidden layer of a RAAM. Then the first and second connection layers of the *parser* approximate the transition function of a DFA (see Section 5.2) $f_{P1} : \Gamma \times Q \rightarrow Q$ and a symbolic

mapping function $f_{P2} : Q \rightarrow V_{CRV}$ respectively; the *segmenter* approximates a symbolic function $f_S : \Gamma \times Q \rightarrow Q \times A$; and the first and second connection layers of the RAAM approximate the *push* function $f_{push} : V_{stack} \times Q \rightarrow V_{stack}$ and the *pop* function $f_{pop} : V_{stack} \rightarrow V_{stack} \times Q$ of a RAAM stack respectively. The input string is sequentially presented to SPEC and is a sequence of syntactically untagged English words represented as fixed-length distributive vectors of gray-scale values between 0 and 1. The emphasis of SPEC was on exploring the generalization as well as noise tolerance capabilities of a neural parser. SPEC uses implicit central control to integrate its different modules and reportedly achieves 100% generalization performance on a whole test set of 98100 English relative clause sentences with up to 4 clauses. Since the words (terminals) in the CFG which generates the test sentences are not pre-translated by a lexical analyzer into syntactically tagged tokens, the number of production rules and terminals tend to increase linearly with the size of the vocabulary in the CFG. Augmenting SPEC with a lexical analyzer offers a way around this problem.

[27, 174, 197] propose higher-order recurrent neural network equipped with an external stack to learn to recognize deterministic CFG, i.e., to learn to simulate a deterministic pushdown automata (DPDA). [27, 174] use an analog network coupled with a *continuous* stack and use a variant of a real time recurrent network learning algorithm to train the network. [197] uses a discrete network coupled with a *discrete* stack and employs a pseudo-gradient learning method to train the network. The input to the network is a sequentially presented, unary-coded string of variable length. Let Q be a finite non-empty set of *states*, Γ a finite non-empty *input alphabet*, Λ a finite non-empty *stack alphabet*, $A = \{push,\ pop,\ no\text{-}operation\}$ the set of stack actions, and *Boolean* the set $\{false,\ true\}$. These recurrent neural networks approximate the transition function of a DPDA, i.e., $f_{DPDA} : Q \times \Gamma \times \Lambda \rightarrow Q \times \Lambda \times A$. The networks are trained to approximate a language recognizer function $f_L : \Gamma^* \rightarrow Boolean$. Strings generated from CFG including *balanced parenthesis grammar*, $a^n b^n$, $a^{m+n} b^m c^n$, $a^n b^n c b^m a^m$, *postfix grammar*, and/or *palindrome grammar* were used to evaluate the generalization performance of the proposed networks.

The proposed neural architecture for syntax analysis is composed of neural network mod-

ules for stack, lexical analysis, parsing, and parse tree construction. It differs from most of the neural network realizations of parsers in that it is systematically assembled using neural associative processors (memories) as primary building blocks. It is able to exploit massively parallel content-based pattern matching and retrieval capabilities of neural associative processors (memories). This offers an opportunity to explore the potential benefits of massively parallel pattern matching in the design of high performance computing systems for real time symbol processing applications.

6.2 Neural Network Design for a Lexical Analyzer (NNLexAn)

A lexical analyzer is defined by a recursive symbolic function $\hat{f}_{LexAn} : \Gamma^*\$ \rightarrow \Delta^*\$$. Γ is the input alphabet, \$ is a special symbol denoting "*end of input*", and Δ is the set of lexical tokens. $\Gamma^*\$$ (or $\Delta^*\$$) denotes the set of strings obtained by adding the suffix \$ to each of the strings over the alphabet Γ (or Δ). The conventional approach to implementing a lexical analyzer using a DFA (in particular, a Mealy machine) can be realized quite simply using an NN DFA [20]. However, a major drawback of this approach is that all legal transitions have to be exhaustively specified in the DFA. For example, Figure 6.1 shows a simplified state diagram without all legal transitions specified for a lexical analyzer which recognizes keywords of a programming language : `begin`, `end`, `if`, `then`, and `else`.

Suppose the lexical analyzer is in a state that corresponds to the end of a keyword. Then its current state would be state *7, 11, 15, 18*, or *23*. If the next input character is *b*, there should be legal transitions defined from those states to state *2*. That is the same for states *8, 16* and *19* in order to handle the next input characters *e*, *i*, and *t*. Thus, this extremely simple lexical analyzer with 22 explicitly defined legal transitions has 20 unspecified transitions. The realization of such a simple 5-word (23-state) lexical analyzer by an NN DFA requires 20+22=42 hidden neurons. Additional transitions have to be defined in order to allow multiple blanks between two consecutive words in the input stream, and for error handling. These drawbacks are further exacerbated in applications involving languages with large vocabularies.

A better alternative is to use a Dictionary (or a database) to serve as a lexicon. The

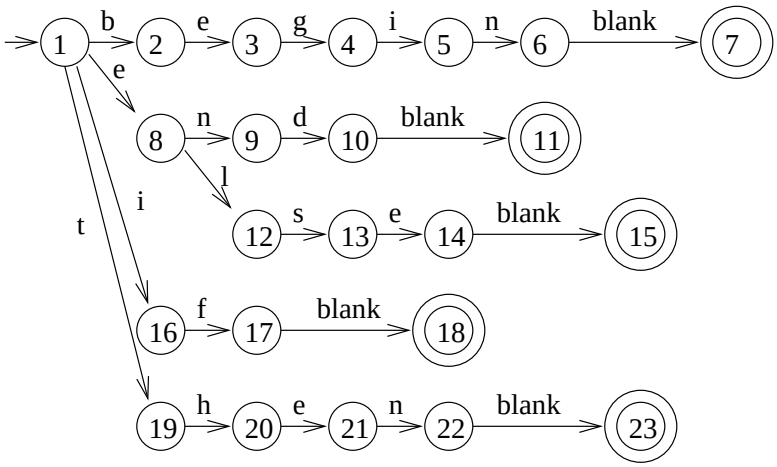

Figure 6.1 The simplified state diagram of a DFA which recognizes keywords : `begin, end, if, then`, and `else`

proposed design for NNLexAn consists of a *word segmenter* for carving an input stream of characters into a stream of words, and a *word lookup table* for translating the carved words of variable length into syntactically tagged tokens of fixed length. The syntactically tagged tokens of fixed length are to be used as single logical units in parsing. Such a translation can be realized by a simple query to a database using a key. Such database query processing can be efficiently implemented using neural associative memories (see Chapter 2).

6.2.1 Neural network design for a word segmenter (NNSeg)

In program translation, the primary function of a word segmenter is to identify *illegal words* and to group input stream into *legal words* including keywords, identifiers, constants, operators, and punctuation symbols. A word segmenter can be defined by a recursive symbolic function $\hat{f}_{WordSeg} : \Gamma^*\$ \rightarrow \Lambda^*\$$, where Γ is the input alphabet, \$ is a special symbol denoting "*end of input*", and Λ is the set of legal words. $\Gamma^*\$$ (or $\Lambda^*\$$) denotes the set of strings obtained by adding the suffix \$ to each of the strings over the alphabet Γ (or Λ).

Figure 6.2 shows the state diagram of a DFA simulating a simple word segmenter which carves continuous input stream of characters into integer constants, keywords, and identifiers. Both the keywords and identifiers are defined as strings of English characters. For simplicity,

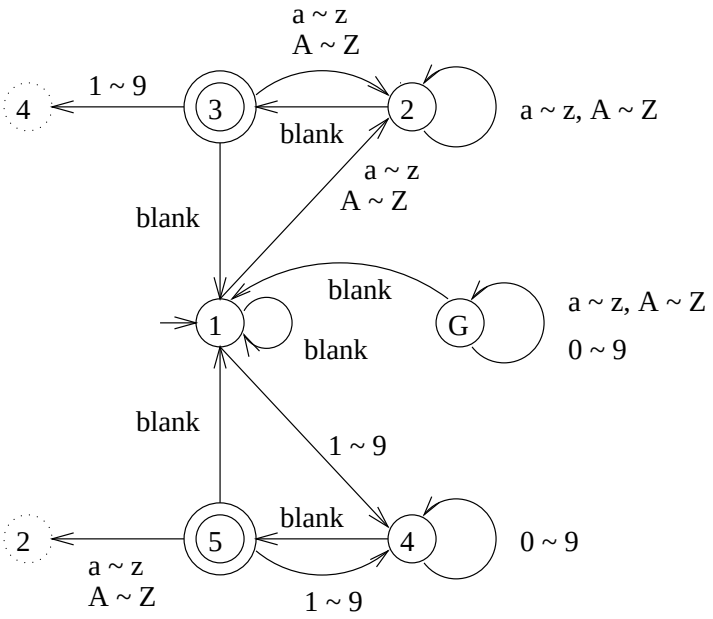

Figure 6.2 The state diagram of a DFA which simulates a simple word segmenter carving continuous input stream of characters into words including integer constants, keywords and identifiers

the handling of *end-of-input* is not shown in the figure. The word segmenter terminates processing upon encountering the end-of-input symbol $. Each time when the word segmenter goes into an accepting state, it instructs the word lookup table to look up a word that has been extracted from the input stream and stored in a buffer.

Since syntax error handling is not discussed here, it may be assumed that any illegal word is discarded by the word segmenter and is also discarded from the buffer which temporarily stores the illegal word being extracted from the input stream. Such a word segmenter can also be realized by an NN DFA. Since any undefined (un-implemented) transition moves into a binary-coded state of all zeros automatically in an NN DFA, it would be expedient to encode the garbage state (state G in Figure 6.2) using a string of all zeros. Although the most straightforward implementation of NN DFA (see Section 5.2) uses one hidden neuron per transition, one can do better. In Figure 6.2 the 10 transitions from state 4 on ASCII-coded input symbols 0,1,...,9 can be realized by only two hidden neurons in an NN DFA using *partial pattern recognition* (see Chapters 2 and 3). Other transitions on input symbols 0,1,...,9,

a,b,...,z, and A,B,...,Z can be handled in a similar fashion.

6.2.2 Neural network design for a word lookup table (NNLTab)

During lexical analysis in program compilation or similar applications, each word of variable length (extracted by the word segmenter) is translated into a *token* of fixed length. Each such token is treated as a single logical entity: an identifier, a keyword, a constant, an operator or a punctuation symbol. Such a translation can be defined by a simple symbolic function $f_{WordTran} : \Lambda \cup \{\$\} \rightarrow \Delta \cup \{\$\}$. Here, Λ, \$, and Δ denote the same entities as in the definitions of $\hat{f}_{WordSeg}$ and $\hat{f}_{LexAn}$ above. Note that $f_{WordTran}$ can be realized by a BMP module. In other lexical analysis applications, a word may be translated into a token having two subparts: category code denoting the syntactic category of a word, and feature code denoting the syntactic features of a word.

Conventional approach to doing such translation (dictionary lookup) is to perform a simple query on a suitably organized database (with the segmented word being used as the key). This content-based pattern matching and retrieval process can be efficiently and effectively realized by neural associative memories. Database query processing using neural associative memories is discussed in detail in Chapter 3 and is summarized briefly in what follows. Each word and its corresponding token are stored as an association pair in a neural associative memory. Each such association is implemented by a hidden neuron and its associated connections. A query is processed in two steps: *identification* and *recall*. During the identification step, a given word is compared to all stored words in parallel by the hidden neurons and their associated 1st-layer connections in the memory. Once a match is found, one of the hidden neurons is activated to recall the corresponding token using the 2nd-layer connections associated with the activated hidden neuron. The time required for processing such a query is of the order of 20 ns (at best) to 100 ns (at worst) given the current CMOS technology for implementation of artificial neural networks (see Section 3.3).

6.3 A Modular Neural Architecture for LR Parser (NNLR Parser)

LR(k) grammars generate the so-called deterministic context-free languages which can be accepted by deterministic push-down automata [74]. Such grammars find extensive applications in programming languages and compilers. LR parsing is a linear time table-driven algorithm which is widely used for syntax analysis of computer programs [2, 19, 170]. This algorithm involves extensive pattern matching which suggests the consideration of a neural network implementation using associative memories. This section proposes a modular neural network architecture for parsing LR(1) grammars. LR(k) parsers scan input from left to right and produce a rightmost derivation tree by using lookahead of k unscanned input symbols. Since any LR(k) grammar for k $\geq$ 1 can be transformed into an LR(1) grammar [170], LR(1) parsers are sufficient for practical applications [74].

An LR(1) grammar can be defined as $G_{LR(1)} = (V, T, \Upsilon, \Theta)$ [74], where V and T are finite sets of variables (nonterminals) and terminals respectively, Υ is a finite set of production rules, and $\Theta \in V$ is a special variable called the *start symbol*. V and T are disjoint. Each production rule is of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$. An LR(1) parser can be defined by a recursive symbolic function $\hat{f}_{LRParser} : \Delta^*\$ \rightarrow \Upsilon^*$, where Δ ($\Delta = T$ in the context), \$, and Δ^* are as in $\hat{f}_{LexAn}$, and Υ^* denotes the set of all sequences of production rules over the rule alphabet Υ. Although $\hat{f}_{LRPaser}$ corresponds in form to the recursive symbolic function $\hat{f}_{LexAn}$ in Section 6.2, it can not be realized simply by a Mealy machine which implements $\hat{f}_{LexAn}$. This is due to the fact that the one-to-one mapping relationship between every input symbol of the input string and the output symbol of the output string at corresponding position in a Mealy machine does not hold for $\hat{f}_{LRParser}$. A stack is required to store intermediate results of the parsing process in order to realize an LR(1) parser which is characterized by $\hat{f}_{LRParser}$.

6.3.1 Representation of parse table

Logically, an LR parser consists of two parts: a driver routine which is the same for all LR parsers and a parse table which is grammar-dependent [2]. LR parsing algorithm pre-compiles an LR grammar into a parse table which is referred by the driver routine for deterministically

parsing input string of lexical tokens by `shift/reduce` moves [2, 19]. Such a parsing mechanism can be simulated by a DPDA (deterministic pushdown automata) with ϵ-moves [74]. An ϵ-move does not consume the input symbol, and the input head is not advanced after the move. This enables a DPDA to manipulate a stack without reading input symbols. The neural network architecture for DPDA (NN DPDA) proposed in Section 5.3, augmented with an NN Stack (see Section 5.4), is able to parse DCFL. However, the proposed NN DPDA architecture cannot efficiently handle ϵ-moves because of the need to check for the possibility of an ϵ-move at every state. Therefore, a modified design for LR(1) parsing is discussed below.

Parse table and stack are two main components of an LR(1) parser. The access of parse table can be defined by the symbolic function $f_{ParseTable} : Q \times (\Delta \cup V \cup \{\$\}) \rightarrow A \times Q \times \Upsilon \times N \times V \times Z$ in terms of binary mapping. Here, Q is the finite set of states; Δ, V, \$, and Υ have the same meaning as in the definition of $G_{LR(1)}$ and $\hat{f}_{LRParser}$ given above; $A = \{$`shift, reduce`$\}$ is the set of parsing actions; N is the set of natural numbers; and $Z = \{$`error, in-progress, accept`$\}$ is the set of possible parsing status values.

A parse table can be realized using a BMP module as described in Section 2.2.5 in terms of binary mapping. The next move of the parsing automaton is determined by current input symbol a and the state q that is stored at the top of the stack. It is given by the parse table entry corresponding to $[q, a]$. Each such 2-dimensional parse table entry $action[q, a]$ is implemented as a 6-tuple binary code $< action, state, rule, length, lhs, status >$ in the BMP module for parse table where

- *action* is a 2-bit binary code denoting one of two possible actions, 01 (`shift`) or 10 (`reduce`);
- *state* is an S-bit binary number denoting "*the next state*";
- *rule* is an R-bit binary number denoting the grammar production rule r to be applied if the consulted *action* is a `reduce`;
- *length* is an L-bit binary number denoting the length of the right hand side of the grammar production rule r to be applied if the consulted *action* is a `reduce`;

- *lhs* is an H-bit binary code encoding the grammar nonterminal symbol at the left hand side of the grammar production rule r to be applied if the consulted *action* is a `reduce` and

- *status* is a 2-bit binary code denoting one of three possible *parsing status* codes, 00: `error`, 01: `in progress`, or 10: `accept` (used by higher-level control to acknowledge the success or failure of a parsing).

Note that the order of the tuple's elements arranged in Figure 6.3 is different from above. A canonical LR(1) parse table is relatively large and would typically have several thousand states for a programming language like C. SLR(1) and LALR(1) tables, which are far smaller than LR(1) table, typically have several hundreds of states for the same size of language, and they always have the same number of states for a given grammar [3] (The differences among LR, SLR, and LALR parsers are discussed in [19]). The number of states in the parse table of LALR(1) parsers for most programming languages is between about 200 and 450, and the number of symbols (lexical tokens) is around 50 [19], i.e., the number of table entries is between about 10000 and 22500.

Typically a parse table is realized as a 2-dimensional array in current computer systems. Memory is allocated for every entry of the parse table, and the access of an entry is via its offset in the memory, which is computed efficiently by the size of the fixed memory space for each entry and the indices of an entry in the array. However, it is much more natural to retrieve an entry in a table using content-based pattern matching on the indices of the entry. As described in Section 2.2.5, a BMP module can effectively and efficiently realize such content-based table lookup.

LR grammars used in practical applications typically produce parse tables with between 80% and 95% undefined error entries [19]. The size of the table is reduced by using lists which can result in a significant performance penalty. The use of a BMP for such table lookup help overcome this problem since undefined mappings are naturally realized by a BMP module without the need for extra space and without incurring any performance penalties. Thus, LALR(1) parsing (which is generally the technique of choice for parsing computer programs)

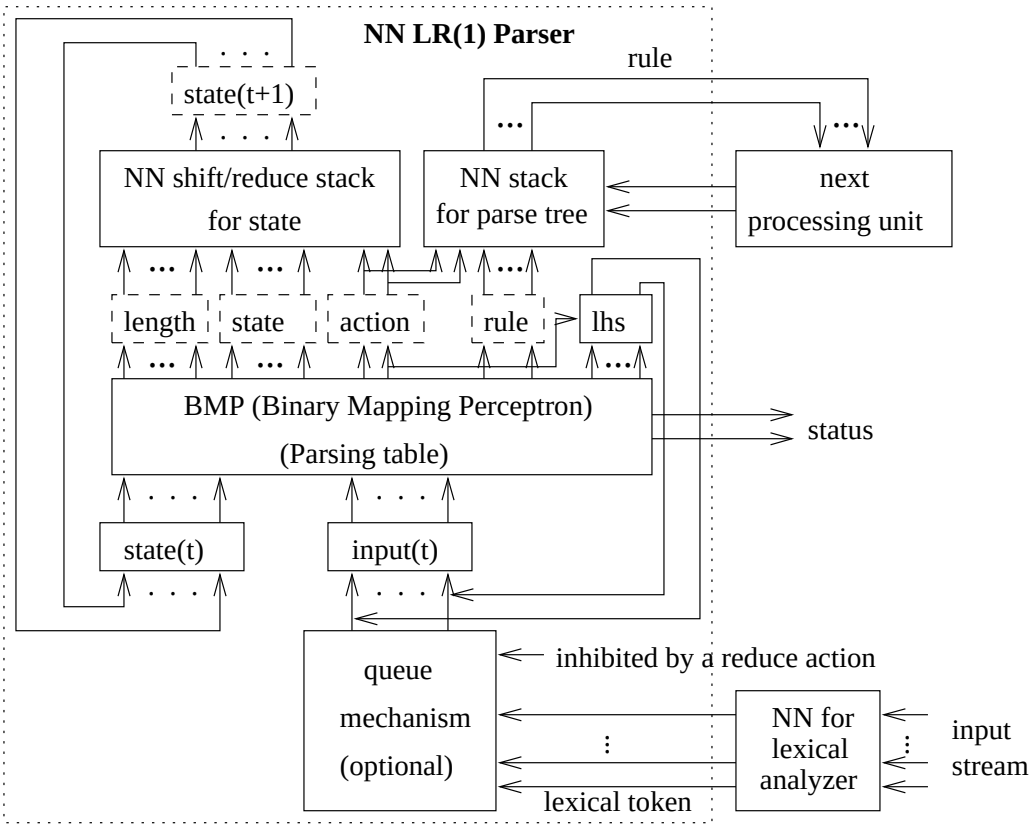

Figure 6.3 The proposed neural network architecture for LR(1) parser

table can be realized using at most about $22500 \times 20\% = 4500$ hidden neurons.

6.3.2 Representation of parsing moves and parse trees

A *configuration* of an LR parser is an ordered pair whose first component corresponds to the stack contents and whose second component is the part of the input that remains to be parsed. A configuration can be denoted by $(q_0 q_1 \cdots q_i,\ a_j a_{j+1} \cdots a_n\$)$, where q_i is the state on top of the stack, q_0 is the stack bottom symbol, a_j is current input symbol, and \$ is a special symbol denoting "*end of input*". The initial configuration is $(q_0,\ a_1 a_2 \cdots a_n\$)$. Let 0^k be a k-bit binary number (code) of all zeros denoting a value of `don't care` for $k \geq 1$. In the proposed NNLR Parser, the configurations resulting from one of four types of *moves* on parsing an input lexical token are as follows:

- If $action[q_i, a_j] =< 01, q, 0^R, 0^L, 0^H, 01 >$, the parser performs a `shift` move and enters the configuration $(q_0 q_1 \cdots q_i q,\ a_{j+1} \cdots a_n\$)$. Such a `shift` move is realized in one operation cycle in the proposed NNLR Parser.

- If $action[q_i, a_j] =< 10, 0^S, r, l, h, 01 >$, the parser performs a `reduce` by producing a binary number r (which denotes a grammar production rule $A \rightarrow \beta$ being applied, where the grammar nonterminal A is denoted by the binary code h, and l is the number of non-empty grammar symbols in β) as part of the parse tree, popping l symbols off the stack, consulting parse table entry $[q_{i-l}, h]$ and entering the configuration $(q_0 q_1 \cdots q_{i-l} q,\ a_{j+1} \cdots a_n\$)$ where $action[q_{i-l}, h] =< 01, q, 0^R, 0^L, 0^H, 01 >$. Such a `reduce` move is realized in two operation cycles in the proposed NNLR Parser since the parse table is consulted twice for simulating the move.

- If $action[q_i, a_j] =< 0^2, 0^S, 0^R, 0^L, 0^H, 10 >$, parsing is completed.

- If $action[q_i, a_j] =< 0^2, 0^S, 0^R, 0^L, 0^H, 00 >$, an error is discovered and the parser stops. Note that such an entry is a binary code of all zeros. (We do not discuss error handling any further in this chapter).

An LR parser scans input string from left to right and performs bottom-up parsing which results in a rightmost derivation tree in reverse. Thus, a stack can be used to store the *parse tree* (derivation tree) which is a sequence of grammar production rules (in reverse order) applied in the derivation of the scanned input string. The rule on top of the final stack which stores a successfully parsed derivation tree is a grammar production rule with the *start symbol* of an LR grammar at its left hand side. Note that each rule is represented by an R-bit binary number and the mapping from a binary-coded rule to the rule itself can be realized by a BMP module.

6.3.3 Architecture of NNLR parser

Figure 6.3 shows the architecture of a modular neural network design for an LR(1) parser which takes advantage of the efficient `shift/reduce` technique. The NNLR Parser uses an optional queue handler module and an NN stack which stores the parse tree (derivation tree). The queue handler stores lexical tokens extracted by the NN lexical analyzer described in Section 6.2 and facilitates the operation of lexical analyzer and parser in parallel. To extract the binary-coded grammar production rules in derivation order sequentially out of the NN stack which stores parse tree, the next processing unit connected to the NN stack sends binary-coded stack `pop` actions to the stack in an appropriate order.

6.3.3.1 Modules of NNLR Parser

The proposed NNLR Parser consists of a BMP module implementing the parse table, an NN `shift/reduce` stack storing states during `shift/reduce` simulation, a buffer (`state(t)`) storing the current state (from the top of the NN `shift/reduce` stack), and a buffer (`input(t)`) storing either current input lexical token or a grammar nonterminal symbol produced by last consulted parsing action which is a `reduce`. When the last consulted parsing action is a `reduce` encoded as 10; the grammar production rule to be reduced is pushed onto the stack for parse tree, the transmission of `input(t)` is from the latched buffer *lhs*, and the input from the queue mechanism is inhibited by the leftmost bit of the binary-coded `reduce` action. When the last consulted parsing action is a `shift` encoded as 01, the transmission of `input(t)` is from the

queue mechanism and the input from the latched buffer *lhs* is inhibited by the rightmost bit of the binary-coded `shift` action.

Parsing is initiated by reset signals to the NN `shift/reduce` stack and the NN stack storing parse tree. The signals reset the SPs of these two stacks to stack bottom and hence `state(t)` is reset to initial state. To avoid clutter, the reset signal lines are not shown in Figure 6.3. The current state buffer `state(t)` and the current input buffer `input(t)` need to be synchronized but the necessary synchronization circuit is omitted from Figure 6.3.

The operations of an LR parser can be viewed in terms of a sequence of transitions from an initial configuration to a final configuration. The transition from one configuration to another can be divided into two steps: the first involves consulting the parse table for next action using current input symbol and current state on top of the stack; the second step involves execution of the action — either a `shift` or a `reduce` — as specified by the parse table. In the NNLR Parser, the first step is realized by a BMP module which implements the parse table lookup; and the second step is executed by a combination of an NN `shift/reduce` stack which stores states, and an NN stack which stores the parse tree (and the BMP module when the next action is a `reduce`).

6.3.3.2 Complexity of the BMP module for parse table

Let M be the number of defined *action* entries in the parse table. All grammar symbols are encoded into H-bit binary codes. The BMP module for parse table uses $S + H$ input neurons, M hidden neurons, and $4 + S + R + L + H$ output neurons. Note that the BMP module produces a binary output of all zeros, denoting a parsing error (see previous description of *parsing status* code in an *action* entry of the parse table), for any undefined *action* entry in the parse table. The R-bit binary-coded grammar production rule is used as the stack symbol for the NN stack which stores the parse tree.

6.3.3.3 Complexity of the NN Stack for parse tree

Assume the pointer control module of the NN stack for parse tree use m_p bits to encode its SP values. Then the pointer control module of the NN stack for parse tree uses $m_p + 2$ input neurons, 3×2^{m_p} hidden neurons, and m_p output neurons. The stack memory module uses m_p input neurons, 2^{m_p} hidden neurons, and R output neurons. The write control module receives $m_p + 1$ binary inputs (the stack pointer + `push/pop` signal) and R binary inputs (the grammar production rule).

6.3.3.4 Complexity of the Shift/Reduce NN Stack

To efficiently implement the `reduce` action in LR parsing, the NN `shift/reduce` stack can be slightly modified from the NN stack described in Section 5.4 to allow multiple stack `pop`s in one operation cycle of the NNLR Parser. The number of `pop`s is coded as an L-bit binary number and equals the number of non-empty grammar symbols at the right hand side of the grammar production rule being reduced. It is used as input to the pointer control module and write control module in the NN `shift/reduce` stack. Thus, each of the modules use L additional input neurons in the NN `shift/reduce` stack as compared to the NN stack proposed in Section 5.4. The output from the NN parse table, namely, the S-bit binary code for state, is used as the stack symbol to the NN `shift/reduce` stack. Let the maximum number of non-empty grammar symbols that appear in the right hand side of a production rule in the LR grammar being parsed be L_m. Then k multiple `pop`s are implemented in the NN `shift/reduce` stack in a manner similar to a single `pop` in the NN stack proposed in Section 5.4 except that the SP value is decreased by k instead of 1, $1 \leq k \leq L_m$. Hence for each SP value, $L_m - 1$ additional hidden neurons are required to allow multiple `pop`s in the pointer control module.

6.3.4 NNLR Parser in action

This subsection illustrates the operation of the proposed NNLR Parser for a given LR(1) grammar. The example of LR(1) grammar (G_1) used here is taken from [3]. The BNF (Backus-Naur Form) description of the grammar G_1 is as follows:

$expression \rightarrow expression + term \mid term$

$term \rightarrow term \times factor \mid factor$

$factor \rightarrow (\ expression\ ) \mid identifier$

Using E, T, F, and I to denote expression, term, factor, and identifier (respectively), these rules can be rewritten in the form of production rules p_1 through p_6:

Production rule p_1 : E $\rightarrow$ E + T

Production rule p_2 : E $\rightarrow$ T

Production rule p_3 : T $\rightarrow$ T $\times$ F

Production rule p_4 : T $\rightarrow$ F

Production rule p_5 : F $\rightarrow$ (E)

Production rule p_6 : F $\rightarrow$ I

Then { I, +, $\times$, (,) } is the set of terminals (i.e. the set of possible lexical tokens from the lexical analyzer), { E, T, F } is the set of nonterminals, { p_1, p_2, p_3, p_4, p_5, p_6 } is the set of production rules, and E is the start symbol of the grammar G_1.

The operation of the parser is shown in terms of symbolic codes (instead of the binary codes used by the NN implementation) to make it easy to understand. Note however that the transformation of symbolic codes into binary form used by NNLR Parser is rather straightforward and has been explained in the preceding sections.

Let s and r denote the parsing actions `shift` and `reduce` and a, e, and i the parsing statuses `accept`, `error`, and `in-progress` respectively. The parse table of the LR(1) parser (more specifically, SLR(1) parser) for grammar G_1 is shown in Table 6.1.

The implementation and operations of the NN `shift/reduce` stack and the NN stack for parse tree follow the discussion and examples in Section 5.4 and they are not discussed here. The parse table can be represented by a binary mapping which in turn can be easily realized by a BMP module (see Section 2.2.5 for details). Following the notation introduced in Section 6.3 for the NN realization of the parse table, we have: $M = 45$ since there are 45 defined entries in the parse table; $S = \lceil log_2 12 \rceil = 4$ since there are 12 states; $H = \lceil log_2(8+2) \rceil = 4$ since there are 8 grammar symbols plus a *null symbol* ϵ and an additional *end-of-string* symbol \$;

Table 6.1 The parse table of the LR(1) parser for grammar G_1

State	I	+	×	(	)
q_0	(s,q_5,*,*,*,i)			(s,q_4,*,*,*,i)	
q_1		(s,q_6,*,*,*,i)			
q_2		(r,*,p_2,1,E,i)	(s,q_7,*,*,*,i)		(r,*,p_2,1,E,i)
q_3		(r,*,p_4,1,T,i)	(r,*,p_4,1,T,i)		(r,*,p_4,1,T,i)
q_4	(s,q_5,*,*,*,i)			(s,q_4,*,*,*,i)	
q_5		(r,*,p_6,1,F,i)	(r,*,p_6,1,F,i)		(r,*,p_6,1,F,i)
q_6	(s,q_5,*,*,*,i)			(s,q_4,*,*,*,i)	
q_7	(s,q_5,*,*,*,i)			(s,q_4,*,*,*,i)	
q_8		(s,q_6,*,*,*,i)			(s,q_{11},*,*,*,i)
q_9		(r,*,p_1,3,E,i)	(s,q_7,*,*,*,i)		(r,*,p_1,3,E,i)
q_{10}		(r,*,p_3,3,T,i)	(r,*,p_3,3,T,i)		(r,*,p_3,3,T,i)
q_{11}		(r,*,p_5,3,F,i)	(r,*,p_5,3,F,i)		(r,*,p_5,3,F,i)

State	$	E	T	F
q_0		(s,q_1,*,*,*,i)	(s,q_2,*,*,*,i)	(s,q_3,*,*,*,i)
q_1	(*,*,*,*,*,a)			
q_2	(r,*,p_2,1,E,i)			
q_3	(r,*,p_4,1,T,i)			
q_4		(s,q_8,*,*,*,i)	(s,q_2,*,*,*,i)	(s,q_3,*,*,*,i)
q_5	(r,*,p_6,1,F,i)			
q_6			(s,q_9,*,*,*,i)	(s,q_3,*,*,*,i)
q_7				(s,q_{10},*,*,*,i)
q_8				
q_9	(r,*,p_1,3,E,i)			
q_{10}	(r,*,p_3,3,T,i)			
q_{11}	(r,*,p_5,3,F,i)			

$R = \lceil log_2 6 \rceil = 3$ since there are 6 production rules; and $L = \lceil log_2 3 \rceil = 2$ since the maximum number of non-empty grammar symbols that appear in the right hand side of a production rule in the LR grammar G_1 is 3. Therefore, the BMP for parse table of G_1 has $S + H = 8$ input, 45 hidden, and $4 + S + R + L + H = 17$ output neurons.

Assume every identifier I is translated from a string of lower case English characters. The lexical analyzer L_1 which translates input strings of +, ×, (,), \$, `blank`, and lower case English characters into strings of lexical tokens can be realized by an NN DFA. Figure 6.4 shows the state diagram of the DFA M_{L_1} for L_1. Note that additional machinery needed for error handling is not included in the DFA M_{L_1}; and when the DFA sees a \$, it stops the processing of the input string and appends a \$ at the end of the output string.

The transition function δ_{L_1} of the DFA M_{L_1} is shown in Table 6.2. This function can be expressed as a binary mapping which in turn can be easily realized by a BMP module (see Section 2.2.5 for details). In the NN DFA, BMP module 1 realizes the transition function $\delta_{L_1} : Q \times \Gamma \rightarrow Q$ and BMP module 2 realizes a translation function $\lambda' : Q \rightarrow \Delta$ s.t. $\lambda'(q_2) =$ I, $\lambda'(q_4) = +$, $\lambda'(q_6) = \times$, $\lambda'(q_8) = ($, $\lambda'(q_{10}) =)$, and $\lambda'(q) = \epsilon$ (null symbol, which is discarded) for other $q \in Q$, where $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}\}$ is the set of states, $\Gamma = \{a, b, ..., z, +, \times, (,), \$, \text{blank}\}$ is the input alphabet, and $\Delta = \{\text{I}, +, \times, (,)\}$ is the output alphabet (i.e., the set of lexical tokens). The symbolic functions δ_{L_1} and λ' can be expressed as binary mappings which in turn can be realized by BMP modules (see Section 2.2.5 for details).

Let us now consider the operation of the LR(1) parser when it is presented with the input string $aa \times bb + cc$. This string is first translated by the lexical analyzer L_1 into a string of lexical tokens I×I+I which is then provided to the LR(1) parser. This translation is quite straightforward, given the state diagram and transition function δ_{L_1} (Table 6.2) of M_{L_1} and its translation function λ'. Note that there is a `space` between each pair of consecutive words in the input character string, and there is no `space` token between each pair of consecutive lexical tokens in the string of lexical tokens.

The string of lexical tokens is parsed by the LR(1) parser whose moves are shown in Table 6.3. At step 1, the parse table entry corresponding to (q_0, I) is consulted. Its value is

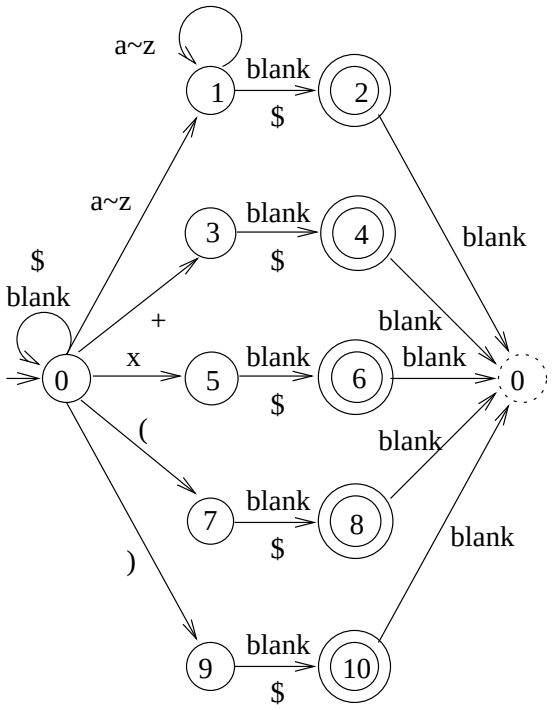

Figure 6.4 The state diagram of the DFA M_{L_1} for the lexical analyzer L_1

Table 6.2 The transition function δ_{L_1} of the DFA M_{L_1}

State	a, b, ..., z	+	×	(	)	blank	$
q_0	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_1	q_1					q_2	q_2
q_2	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_3						q_4	q_4
q_4	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_5						q_6	q_6
q_6	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_7						q_8	q_8
q_8	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_9						q_{10}	q_{10}
q_{10}	q_1	q_3	q_5	q_7	q_9	q_0	q_0

Table 6.3 Moves of the LR(1) parser for grammar G_1 on input string I×I+I

Step	*Content of shift/reduce stack*	*Remaining input*	*Referred entries of parse table*	*Content of parse tree stack*
(1)	q_0	I×I+I$	(q_0,I)	$\perp$
(2)	q_0q_5	×I+I$	(q_5,×), (q_0,F)	$\perp$
(3)	q_0q_3	×I+I$	(q_3,×), (q_0,T)	$\perp p_6$
(4)	q_0q_2	×I+I$	(q_2,×)	$\perp p_6p_4$
(5)	$q_0q_2q_7$	I+I$	(q_7,I)	$\perp p_6p_4$
(6)	$q_0q_2q_7q_5$	+I$	(q_5,+), (q_7,F)	$\perp p_6p_4$
(7)	$q_0q_2q_7q_{10}$	+I$	(q_{10},+), (q_0,T)	$\perp p_6p_4p_6$
(8)	q_0q_2	+I$	(q_2,+), (q_0,E)	$\perp p_6p_4p_6p_3$
(9)	q_0q_1	+I$	(q_1,+)	$\perp p_6p_4p_6p_3p_2$
(10)	$q_0q_1q_6$	I$	(q_6,I)	$\perp p_6p_4p_6p_3p_2$
(11)	$q_0q_1q_6q_5$	$	(q_5,$), ($q_6$,F)	$\perp p_6p_4p_6p_3p_2$
(12)	$q_0q_1q_6q_3$	$	(q_3,$), ($q_6$,T)	$\perp p_6p_4p_6p_3p_2p_6$
(13)	$q_0q_1q_6q_9$	$	(q_9,$), ($q_0$,E)	$\perp p_6p_4p_6p_3p_2p_6p_4$
(14)	q_0q_1	$	(q_1,$)	$\perp p_6p_4p_6p_3p_2p_6p_4p_1$

(s, q_5, $*$, $*$, $*$, i). This results in shifting I and pushing state q_5 onto the `shift/reduce` stack. At step 2, the table entry corresponding to (q_5, ×) is consulted first. Its value is (r, $*$, p_6, 1, F, i) which indicates a `reduce` on production rule p_6. Therefore, state q_5 is popped off the stack, and table entry corresponding to (q_0, F) is consulted next. The entry is (s, q_3, $*$, $*$, $*$, i) which means shifting F and pushing state q_3 onto the stack. The remaining steps are executed in a similar fashion. At the end of the moves (step 14), the sequence of production rules stored in the stack for parse tree can be applied in reverse order to derive the string I×I+I from grammar start symbol E.

6.4 Performance Analysis

This section explores potential performance advantages of the proposed neural network architecture for syntax analysis in comparison with that of current computer systems that employ inherently sequential index or matrix structure for information matching and retrieval. The performance estimates for the NNLR Parser assume hardware realization based on current CMOS VLSI technology. In the analysis that follows, it is assumed that the two systems have

comparable I/O performance and error handling capabilities.

To simplify the comparison, it is assumed that each instruction on a conventional computer takes τ ns (nanoseconds) on an average. For instance, on a relatively cost-effective 100 MIPS processor, a typical instruction would take 10 ns to complete. (The MIPS measure for speed combines clock speed, effect of caching, pipelining and superscalar design into a single figure for speed of a microprocessor). Similarly, we will assume that a single identification and recall operation by a neural associative memory takes α ns. Assuming hardware implementation based on current CMOS VLSI technology, $\alpha = 20$ ns (see Section 3.3).

Syntax analysis in a conventional computer typically involves: lexical analysis, grammar parsing, parse tree construction and error handling. These four processes are generally coded into two modules [2]. Error handling is usually embedded in grammar parsing and lexical analysis respectively, and parse tree construction is often embedded in grammar parsing. The procedure for grammar parsing is the main module. In single-CPU computer systems, even assuming negligible overhead for parameter passing, a procedure call entails, at the very minimum, (1) saving the context of the caller procedure and activation of the callee procedure which typically requires 6 instructions [105]; and (2) context restoration and resumption of caller procedure upon the return (exit) of the callee procedure, which typically requires at least 3 instructions [105]). Thus, a procedure call entails a penalty of 9 instructions or about 9τ ns.

6.4.1 Performance analysis of lexical analyzer

Lexical analysis can be performed by a DFA whose transition function can be represented as a 2-dimensional table with current state and current input symbol as indices. The continuous transition moves of such a DFA involve repetitive lookup of its next state from the table using current state and current input symbol at each move until an error state or an accepting state is reached. Such a repetitive table lookup involves content-based pattern matching and retrieval which can be performed potentially more efficiently by neural associative memories.

Each entry of the DFA transition table implemented on conventional computers usually

contains three parts: the next state; a code for whether the next state is an accepting state, an error state, or neither; and the lexical token to use if the next state is an accepting state. Implementing such a repetitive table lookup on conventional computers requires, at a minimum, six instructions: one (or two) multiplication and one addition to compute the offset in the transition table (to access the location where the next state is stored), one memory access to fetch the next state from the table, one addition to compute the offset of the second part in the transition table (based on the known offset of the first part), one memory access to fetch the second part from the table, and one branch-on-comparison instruction to jump back to the first instruction of the loop if the next state is neither an error state nor an accepting state. (Note that this analysis ignores I/O processing requirements). Thus, each state transition takes 6 instructions or 6τ ns.

In contrast, the proposed NN architecture for lexical analyzer computes the next state using associative (content-addressed) pattern *matching-and-retrieval* in a single *identification-and-recall* cycle of a BMP module. In the 2-dimensional table, the values of the two indices for an entry provide a unique pattern – the *index pattern*, for accessing the table entry. In the BMP module, each index pattern and the corresponding entry are stored as an association pair by a hidden neuron and its associated connections. The BMP performs a table lookup in two steps : *identification* and *recall*. In the identification step, a given index pattern is compared to all stored index patterns in parallel by the hidden neurons and their associated 1st-layer connections. Once a match is found, one of the hidden neurons is activated to recall the associated entry value using the 2nd-layer connections associated with the activated hidden neuron.

In program compilation, a segmented word is translated into a syntactically tagged token when the DFA for lexical analysis enters an accepting state. On conventional computers, this translation step costs, at the very minimum, three instructions (or 3τ ns): one addition to compute the offset of the third part in the transition table (based on the known offset of the first part), one memory access to fetch the lexical token from the table, and one branch instruction to jump back to the first instruction of the loop for carving next word.

In other syntax analysis applications that involve large vocabularies, a database lookup is typically used to translate a word into a syntactically tagged token. In this case, depending on the size of the vocabulary and the organization of the database, it would generally take more than 10 instructions to perform this translation. (See Chapter 3 for a comparison of database query processing using neural associative memories as opposed to conventional computers).

A BMP module is capable of translating a carved word into a token as described in Section 2.2.5 in a single cycle of *identification-and-recall* with a time delay of α ns. Note that this step can be pipelined (see the NNLR Parser in action in Section 6.3).

In summary, if we assume the average length of words in input string being W symbols and we ignore I/O, error handling and the overhead associated with procedure calls, it would take $(6W + 3)\tau$ ns on average to perform lexical analysis of a word on a conventional computer. In contrast, it would take $(W + 1)\alpha$ ns using the proposed NN lexical analyzer. This analysis ignores I/O and error handling. For example, assuming a 100 MIPS conventional computer ($\tau = 10$ ns), and current CMOS VLSI implementation of neural associative memories ($\alpha = 20$ ns), with $W = 5$, then the former takes 330 ns and the latter 120 ns.

6.4.2 Performance analysis of LR parser

LR parsing also involves repetitive table lookup which can be performed efficiently by neural associative memories. LR parser is driven by a 2-dimensional table (parse table) with current state and current input symbol as indices. Once a next state is retrieved, it is stored on a stack and is used as the current state for the next move. Parsing involves repetitive application of a sequence of `shift` and `reduce` moves. A `shift` move would take at least 6 instructions, or equivalently 6τ ns on a conventional computer. This includes 3 instructions to consult the parse table, 1 instruction to push the next state onto the stack, 1 instruction to increment the stack pointer, and 1 instruction to go back to the first instruction of the repetitive loop for next move. A typical `reduce` move involves a parse table lookup, a `pop` of the state stack, a `push` to store a rule into the stack for parse tree, and a `shift` move. Thus, a typical `reduce` would take at least $3 + 1 + 2 + 6 = 12$ instructions, or equivalently 12τ ns, on a conventional

computer.

In the proposed NNLR Parser, the computation delay consists of the delays contributed by the operation of the two NN Stacks and the BMP which stores the parse table. An NN Stack consists of two BMP modules, one of which is augmented with a write control module. Assuming that the computation delay of an NN stack is roughly equal to that of two sequentially linked BMPs (2α ns), a `shift` move (which takes one operation cycle of the NNLR Parser) and a typical `reduce` move (which takes two operation cycles of the NNLR Parser) would consume 3α ns and 6α ns respectively. (This analysis ignores the effect of queuing between the NNLR Parser and the NN lexical analyzer).

Assuming that the average length of words in input string be W symbols, and ignoring I/O, error handling and the overhead associated with procedure calls, parsing a word (a word has to be translated into a lexical token by lexical analysis first) by `shift` and `reduce` moves would take $(6W + 9)\tau$ ns and $(6W + 15)\tau$ ns respectively on a conventional computer.

In contrast, because the NNLR Parser and NN lexical analyzer can operate in parallel, `shift` and `reduce` moves take 3α ns or $(W + 1)\alpha$ ns (whichever is larger) and 6α ns or $(W + 1)\alpha$ ns (whichever is larger) respectively on the NNLR Parser.

Thus, as shown in Table 6.4, for typical values of α, τ and W, the proposed NNLR Parser offers a potentially attractive alternative to conventional computers for syntax analysis.

It should be noted that the preceding performance comparison has not considered alternative hardware realizations of syntax analyzers. These include hardware implementations of parsers using conventional building blocks used for building today's serial computers. We are not aware of any such implementations although clearly, they can be built. In this context it is worth noting that the neural architecture for syntax analysis proposed in this chapter makes extensive use of massively parallel processing capabilities of neural associative processors (memories). It is quite possible that other parallel (possibly non neural network) hardware realizations of syntax analyzers offer performance that compares favorably with that of the proposed neural network realization. We can only speculate as to why there appears to have been little research on parallel architectures for syntax analysis. Historically, research in

Table 6.4 A comparison of the estimated performance of the proposed NNLR Parser with that of conventional computer systems for syntax analysis

Type of overhead	*NNLR Parser*	*Conventional computers*
time for lexical analysis of a word	$(W+1)\alpha$	$(6W+3)\tau$
time for a `shift` move of parsing	$\max\,[3\alpha, (W+1)\alpha]$	$(6W+9)\tau$
time for a `reduce` move of parsing	$\max\,[6\alpha, (W+1)\alpha]$	$(6W+15)\tau$

high performance computing has focused primarily on speeding up the execution of numeric computations, typically performed by programs written in compiled languages such as C and FORTRAN. In such applications, syntax analysis is done during program compilation which is relatively infrequently compared to program execution. The situation is quite different in symbol processing (e.g., knowledge based systems of AI, analysis of mathematical expressions in software designed for symbolic integration, algebraic simplification, theorem proving) and interactive programming environments based on interpreted programming languages (e.g., LISP, JAVA). Massively parallel architectures for such tasks are only beginning to be explored.

6.5 Summary and Discussion

This chapter has explored the design of a neural architecture for syntax analysis of languages with known (a-priori specified) grammars. Syntax analysis is a prototypical symbol processing task with a diverse range of applications in artificial intelligence, cognitive modelling, and computer science. Examples of such applications include: language interpreters for interactive programming environments using interpreted languages (e.g., LISP, JAVA), parsing of symbolic expressions (e.g., in real-time knowledge based systems, database query processing, and mathematical problem solving environments), syntactic or structural analysis of large collections of data (e.g., molecular structures, engineering drawings, etc.), and high-performance compilers for program compilation and behavior-based robotics. Indeed, one would be hard-pressed to find a computing application that does not rely on syntax analysis at some level. The need for syntax analysis in real time calls for novel solutions that can deliver the desired performance at an affordable cost. Artificial neural networks, due to their potential advantages

for real-time applications on account of their inherent parallelism, offer an attractive approach to the design of high performance syntax analyzers.

The proposed neural architecture for syntax analysis is obtained through systematic and provably correct composition of a suitable set of component symbolic functions which are ultimately realized using neural associative processor (memory) modules. The neural associative processor (memory) is essentially a 2-layer perceptron which can store and retrieve arbitrary binary pattern associations [21]. It is a cost-effective SIMD (single instruction, multiple data) computer system for massively parallel pattern matching and retrieval. Since each component in the proposed neural architecture computes a well-defined symbolic function, it facilitates the systematic synthesis as well as analysis of the desired computation at a fairly abstract (symbolic) level. Realization of the component symbolic functions using neural associative processors (memories) allows one to exploit massive parallelism to support applications that require syntax analysis to be performed in real time.

The proposed neural network for syntax analysis is capable of handling sequentially presented character strings of variable length, and it is assembled from neural network modules for lexical analysis, stack processing, parsing, and parse tree construction. The neural network stack can realize stacks of arbitrary depths (limited only by the number of neurons available). The parser outputs the parse tree resulting from syntax analysis of strings from widely used subsets of deterministic context-free languages (i.e., those generated by LR grammars). Since logically an LR parser consists of two parts, a driver routine which is the same for all LR parsers, and a parse table which varies from one application to the next [3], the proposed NNLR Parser can be used as a general-purpose neural architecture for LR parsing.

It is relatively straightforward to estimate the cost and performance of the proposed neural architecture for syntax analysis based on the known computation delays associated with the component modules (using known facts or a suitable set of assumptions regarding current VLSI technology for implementing the component modules). Our estimates suggest that the proposed system offers a systematic and provably correct approach to designing cost-effective high-performance syntax analyzers for real-time syntax analysis using known (a-priori speci-

fied) grammars.

The choice of the neural associative processors (memories) as the primary building blocks for the synthesis of the proposed neural architecture for syntax analysis was influenced, among other things, by the fact that they find use in a wide range of systems in computer science, artificial intelligence, and cognitive modelling. This is because associative pattern matching and recall is central to pattern-directed processing which is at the heart of many problem solving paradigms in AI (e.g., knowledge based expert systems, case based reasoning) as well as computer science (e.g., database query processing, information retrieval). As a result, design, VLSI implementation, and applications of associative processors have been studied extensively in the literature [21, 23, 68, 78, 88, 97, 110, 124, 127, 151, 153]. The neural network architecture proposed in this chapter for syntax analysis demonstrates the versatility of neural associative processors (memories) as generic building blocks for systematic synthesis of modular massively parallel architectures for symbol processing applications.

It should be noted that the primary focus of this chapter was on taking advantage of massive parallelism and associative pattern storage, matching, and recall properties of a particular class of neural associative memories in designing high performance syntax analyzers for a-priori specified grammars. Consequently, it has not addressed several other potential advantages of neural network architectures for intelligent systems. Notable among these are inductive learning and fault tolerance.

Machine learning of grammars or grammar inference is a major research topic which has been, and continues to be, the subject of investigation by a large number of researchers in artificial intelligence, machine learning, syntactic pattern recognition, neural networks, computational learning theory, natural language processing, and related areas. The surveys of grammar inference in general can be found in [69, 96, 115, 137], and the recent results on grammar inference using neural networks can be found in [6, 13, 27, 38, 45, 44, 66, 77, 80, 116, 122, 123, 129, 159, 161, 166, 174, 192, 194, 197].

Fault tolerance capabilities of neural architectures under different fault models (neuron faults, connection faults, etc) has been the topic of considerable research [21, 165, 180] and is

beyond the scope of this chapter. However, it is worth noting that the proposed neural network design for syntax analysis inherits some of the fault tolerance capabilities of its primary building block, the neural associative processor (memory) (see Section 2.3.3 for details).

7 CONCLUSION

Traditional symbol processing models of AI and ANN have been viewed by many as radically (and perhaps even irreconcilably) different paradigms for the design of intelligent systems. But given the fact that they are essentially equivalent in terms of their computing capabilities, a more reasonable view is that they each represent different architectural commitments and hence different cost-performance tradeoffs within the space of possible designs for intelligent systems. This latter viewpoint argues for a somewhat systematic exploration of this design space in search of novel and efficient computational architectures for such systems. This dissertation takes a few small steps in this direction and adds to the growing body of literature [47, 72, 99, 179] that demonstrates the potential benefits of integrated neural-symbolic architectures that overcome some of the limitations of today's ANN and AI systems.

More specifically, this dissertation develops the theory and implementation of a neural architecture for associative memory which is capable of massively parallel best match, exact match, and partial match. It also demonstrates systematic, provably correct synthesis of efficient neural architectures respectively for information retrieval and database query processing, elementary logical inference, sequence processing, and syntax analysis using neural associative memories as the primary building blocks for massively parallel pattern-directed symbol processing. This facilitates the systematic analysis of the resulting computation performed by the resulting neural systems at a fairly abstract (symbolic) level.

BIBLIOGRAPHY

[1] Ackley, D. H., Hinton, G. D. and Sejnowski, T. J., A Learning Algorithm for Boltzmann Machines, *Cognitive Science*, vol. 9, pp. 147-169, 1985.

[2] Aho, A. V., Sethi, R. and Ullman, J. D., *Compilers: Principles, techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[3] Aho, A. V. and Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.

[4] Aitchison, J., Words in the Mind, in: *An Introduction to the Mental Lexicon*, Basil Blackwell, Oxford, 1987.

[5] Ajjanagadde, V. and Shastri, L., Efficient Inference with Multi-Place Predicates and Variables in a Connectionist System, *Proceedings of 11th Cognitive Science Society Conference*, pp. 396-403, Erlbaum, Hillsdale, NJ, 1989.

[6] Allen, R. B., Connectionist Language Users, *Connection Science*, vol. 2, no. 4, p. 279, 1990.

[7] Amari, S., Characteristics of Random Nets of Analog Neuron-like Elements, *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 2, pp. 643-657, 1972.

[8] Amari, S., Learning Patterns and Pattern Sequences by Self-Organizing Nets of Threshold Elements, *IEEE Transactions on Computers*, vol. 21, no. 11, pp. 1197-1206, 1972.

[9] Amari, S., Neural Theory of Association and Concept-Formation, *Biological Cybernetics*, vol. 26, pp. 175-185, 1977.

[10] Anderson, J. A., Silverstein, J. W., Ritz, S. A. and Jones, R. S., Distinctive Feature, Categorical Perception, and Probability Learning: Some Applications of a Neural Model, *Psychological Review*, vol. 84, pp. 413-451, 1977.

[11] Arbib, M., Schema Theory: Cooperative Computation for Brain Theory and Distributed AI, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 51-74, Academic Press, San Diego, CA, 1994.

[12] Bently, J. L., Multidimensional Binary Search Trees Used for Associative Searching, *Communications of the ACM*, vol. 18, no. 9, pp. 507-517, 1975.

[13] Berg, G., A Connectionist Parser with Recursive Sentence Structure and Lexical Disambiguation, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 32-37, MIT Press, Cambridge, MA, 1992.

[14] Bookman, L. A., A Framework for Integrating Relational and Associational Knowledge for Comprehension, in: *Computational Architectures Integrating Neural and Symbolic Processes : A Perspective on the State of the Art*, Sun, R. and Bookman, L. (Ed.), Chapter 9, pp. 283-318, Kluwer Academic Publishers, Norwell, MA, 1995.

[15] Butterworth, B., Lexical Representation, in: *Language Production Volume 2: Development, Writing and Other Language Processes*, Butterworth, B. (Ed.), pp. 257-294, Academic Press, London, 1983.

[16] Carpenter, G. and Grossberg, S., Adaptive Resonance Theory: Stable Self-Organization of Neural Recognition Codes in Response to Arbitrary Lists of Input Patterns, *8th Annual Conference of the Cognitive Science Society*, pp. 45-62, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[17] Carpenter, G. and Grossberg, S., A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine, *Computer Vision, Graphics, and Image Understanding*, vol. 37, pp. 54-116, 1987.

[18] Carpenter, G. and Grossberg, S., ART2: Self-Organization of Stable Category Recognition Codes for Analog Input Patterns, *Applied Optics*, vol. 26, pp. 4919-4930, 1987.

[19] Chapman, N. P., *LR Parsing : Theory and Practice*, Cambridge University Press, Cambridge, MA, 1987.

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

[21] 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, 1995a.

[22] Chen, C. and Honavar, V., A Neural Network Architecture for Syntax Analysis. Submitted. Preliminary version available as Iowa State University Dept. of Computer *Science Tech. Rep. ISU-CS-TR 95-18*, 1995b.

[23] 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.

[24] Chen, C. and Honavar, V., Neural Architectures for Information Retrieval and Query Processing, in: *Handbook of Natural Language Processing: Techniques and Applications for the Processing of Language as Text*, Moisl, H., Dale, R. and Somers, H. (Ed.), Marcel Dekker, New York, 1998.

[25] Chen, C. and Honavar, V., A Neural Architecture for Parallel Set Operations. Paper in preparation.

[26] Cohen, D., *Introduction to Computer Theory*, Wiley, New York, 1986.

[27] Das, S., Giles, C. L. and Sun, G. Z., Using Prior Knowledge in a NNDPA to Learn Context-Free Languages, in: *Advances in Neural Information Processing Systems 5*, Hanson, S. J., Cowan, J. D. and Giles, C. L. (Ed.), pp. 65-72, Morgan Kaufmann, San Mateo, CA, 1993.

[28] $D^{'}$Autrechy , C. L. and Reggia, J. A., An Overview of Sequence Processing by Connectionist Models, *Technical Report UMIACS-TR-89-82*, University of Maryland, College Park, MD, 1989.

[29] Dayhoff, J., *Neural Network Architectures : An Introduction*, Van Nostrand Reinhold, New York, 1990.

[30] Defiore, C. and Berra, P. B., A Data Management System Utilizing an Associative Memory, *AFIPS, Proceedings of the National Computer Conference*, vol. 42, pp. 181-185, 1973.

[31] Dolan, C. P. and Smolensky, P., Tensor Product Production System: A Modular Architecture and Representation, *Connection Science*, **1**, pp. 53-58, 1989.

[32] Dyer, M. G., Connectionist Natural Language Processing: A Status Report, in: *Computational Architectures Integrating Neural and Symbolic Processes : A Perspective on the State of the Art*, Sun, R. and Bookman, L. (Ed.), Chapter 12, pp. 389-429, Kluwer Academic Publishers, Norwell, MA, 1995.

[33] Elman, J. L., Finding Structure in Time, *Cognitive Science*, vol. 14., pp. 179-211, 1990.

[34] Fanty, M. A., Context-free Parsing with Connectionist Networks, *Proceedings of AIP Neural Networks for Computing, Conference No. 151*, pp. 140-145, Snowbird, UT, 1986.

[35] Fodor, J. and Pylyshyn, Z., Connectionism and Cognitive Architecture: A Critical Analysis, in: *Connections and Symbols*, Pinker, S. and Mehler, J. (Ed.), MIT Press, Cambridge, MA, 1988.

[36] Forster, K. I., Accessing the Mental Lexicon, in: *New Approaches to Language Mechanisms*, Walker, R. and Wales, R. J. (Ed.), pp. 257-287, North-Holland, Amsterdam, 1976.

[37] Frakes, W. B. and Baeza-Yates R. (Ed.), *Information Retrieval: Data Structures & Algorithms*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[38] Frasconi, P., Gori, M., Maggini, M. and Soda, G., Unified Integration of Explicit Rules and Learning by Example in Recurrent Networks, *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 2, pp. 340-346, 1995.

[39] Fukushima, K., Cognitron: A Self-Organizing Multilayered Neural Network, *Biological Cybernetics*, vol. 20, pp. 121-136, Nov. 1975.

[40] Fukushima, K., Miyake, S. and Ito, T., Neocognitron: A Neural Network Model for a Mechanism of Visual Pattern Recognition, *IEEE Transactions on System, Man, and Cybernetics*, SMC-13, no. 5, pp. 826-834, Sep./Oct., 1983.

[41] Fukushima, K., Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition, *Neural Networks*, vol. 1 , pp. 119-130, 1988.

[42] Gallant, S. I., Connectionist Expert Systems, *Communications of the ACM*, vol. 31, pp. 152-169, February 1988.

[43] Gallant, S. I., *Neural Network Learning and Expert Systems*, MIT Press, Cambridge, MA, 1993.

[44] Giles, C. L., Horne, B. W. and Lin, T., Learning a Class of Large Finite State Machines With a Recurrent Neural Network, *Neural Networks*, vol. 8, no. 9, pp. 1359-1365, 1995.

[45] Giles, C. L., Miller, C. B., Chen, D., Sun, G. Z. and Lee, Y. C., Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks, *Neural Computation*, vol. 4., no. 3., p. 380, 1992.

[46] Goldfarb, L. and Nigam, S., The Unified Learning Paradigm: A Foundation for AI, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 533-559, Academic Press, San Diego, CA, 1994.

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

[48] Gowda, S. M. et al., Design and Characterization of Analog VLSI Neural Network Modules, *IEEE Journal of Solid-State Circuits*, vol. 28, no. 3, pp. 301-313, 1993.

[49] Graf, H. P. and Henderson, D., A Reconfigurable CMOS Neural Network, *ISSCC Dig. Tech. Papers*, pp. 144-145, San Francisco, CA, 1990.

[50] Grant, D. et al., Design, Implementation and Evaluation of a High-Speed Integrated Hamming Neural Classifier, *IEEE Journal of Solid-State Circuits*, vol. 29, no. 9, pp. 1154-1157, Sep. 1994.

[51] Grossberg, S., Some Networks That Can Learn, Remember, and Reproduce Any Number of Space-Time Patterns II, *Studies in Applied Mathematics*, vol. 49, pp. 135-166, 1970.

[52] Grossberg, S., Contour Enhancement, Short-Term Memory, and Constancies in Reverberating Networks, *Studies in Applied Mathematics*, vol.52, pp. 217-257, 1973.

[53] Grossberg, S., Adaptive Pattern Classification and Universal Recording II: Feedback, Oscillation, Ilfaction, and Illusions, *Biological Cybernetics*, vol. 23, pp. 187-207, 1976.

[54] Grosspietsch, K. E., Intelligent Systems by Means of Associative Processing, in: *Fuzzy, Holographic, and Parallel Intelligence*, Souček, B. and the IRIS Group (Ed.), pp. 179-214, John Wiley & Sons, New York, 1992.

[55] Gupta, A., *Parallelism in Production Systems*, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Mar. 1986.

[56] Gupta, M. M. and Knopf, G. K., Neuro-Vision Systems: A Tutorial, in: *Neuro-Vision Systems: Principles and Applications*, Gupta, M. and Knopf, G. (Ed.), pp. 1-34, IEEE Press, New York, 1994.

[57] Hamilton, A. et al., Integrated Pulse Stream Neural Networks: Results, Issues, and Pointers, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 385-393, May 1992.

[58] Handke, I., *The Structure of Lexicon: Human versus Machine*, Mouton de Gruyter, Berlin, 1995.

[59] Hao, J. and Vandewalle, J., A New Model of Neural Associative Memories, *International Journal of Neural Systems*, vol. 5, no. 1, pp. 39-47, Mar. 1994.

[60] Hassoun, M., *Fundamentals of Artificial Neural Networks*, MIT Press, Cambridge, MA, 1995.

[61] Hayes-Roth, F., The Role of Partial and Best Matches in Knowledge Systems, in: *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (Ed.), pp. 557-574, Academic Press, New York, NY, 1978.

[62] Haykin, S., *Neural Networks*, MacMillan, New York, 1994.

[63] Hebb, D. O., *The Organization of Behavior*, John Wiley & Sons, New York, 1949.

[64] Hecht-Nielsen, R., Counterpropagation Networks, *Proceedings of IEEE First International Conference on Neural Networks*, vol. II, pp. 19-32, 1987.

[65] Hendler, J., Beyond the Fifth Generation: Parallel AI Research in Japan, *IEEE Expert*, pp. 2-7, Feb. 1994.

[66] Hester, K. A. et al., The Predictive RAAM: A RAAM That Can Learn to Distinguish Sequences from a Continuous Input Stream, *Proceedings of World Congress on Neural Networks*, vol. 4, pp. 97-103, San Diego, CA, June 1994.

[67] Hinton, G. D. and Sejnowski, T. J., Learning and Relearning in Boltzmann machines, in: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1: Foundations*, Rumelhart, D. E., McClelland, J. L. and the PDP Research Group, pp. 282-318, MIT Press, Cambridge, MA, 1986.

[68] Hinton, G. E., Implementing Semantic Networks in Parallel Hardware, in: *Parallel Models of Associative Memory*, Hinton, G. E. and Anderson, J. A. (updated Ed.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[69] Honavar, V., Toward Learning Systems That Integrate Different Strategies and Representations, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 615-644, Academic Press, San Diego, CA, 1994.

[70] 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.

[71] Honavar, V. and Uhr, L., Coordination and Control structures and Processes: Possibilities for Connectionist Networks, *Journal of Experimental and Theoretical Artificial Intelligence 2: 277-302*, 1990.

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

[73] 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.

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

[75] Hopfield, J. J., Neural Networks and Physical Systems with Emergent Collective Computational Abilities, *Proc. Natl. Acad. Sci. USA*, vol. 79, pp. 2554-2558, Apr. 1982.

[76] Hopfield, J. J. and Tank, D., Neural Computation of Decision in Optimization Problems, *Biological Cybernetics*, vol. 52, pp. 141-152, 1985.

[77] Horne, B., Hush, D. R. and Abdallah, C., A State Space Recurrent Neural Network with Application to Regular Grammatical Inference, *UNM Tech. Rep. No. EECE 92-002*, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM, 1992.

[78] Howe, D. B. and Asanović, K., SPACE: Symbolic Processing in Associative Computing Elements, in: *VLSI for Neural Networks and Artificial Intelligence*, Delgado-Frias, J. G. (Ed.), pp. 243-252, Plenum Press, New York, 1994

[79] Jackson, T. and Austin, J., The Representation of Knowledge and Rules in Hierarchical Neural Networks, in: *Neural Networks for Knowledge Representation and Inference*, Levine, D. S. and Aparicio IV, M.(Ed.), Chapter 8, pp. 206-238, Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

[80] Jain, A. N., Waibel, A. and Touretzky, D. S., PARSEC: A Structured Connectionist Parsing System for Spoken Language, *IEEE Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 205-208, San Francisco, CA, Mar. 1992.

[81] Jordan, M., Attractor Dynamics and Parallelism in a Connectionism Sequential Machine, *Program of the Eighth Annual Conference of the Cognitive Science Society*, pp. 531-546, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[82] Kitano, H., *Speech-to-Speech Translation: A Massively Parallel Memory Based Approach*, Kluwer Academic Publishers, Norwell, MA, 1994.

[83] Kleene, S. C., Representation of Events in Nerve Nets and Finite Automata, in: *Automata Studies*, Shannon, C. E. and McCarthy, J. (Ed.), pp. 3-42, Princeton University Press, Princeton, NJ, 1956.

[84] Kohonen, T., Correlation matrix memories, *IEEE Transactions on Computers*, vol. c-21, no. 4, pp. 353-359, Apr. 1972.

[85] Kohonen, T., *Associative Memory: A System-Theoretical Approach*, Springer, New York, 1977.

[86] Kohonen, T., *Self-Organization and Associative Memory*, Springer-Verlag, Berlin, 1984.

[87] Kohonen, T., *Content-Addressable Memories*, 2nd ed., Springer-Verlag, Berlin, 1987.

[88] Kogge, P., Oldfield, J., Brule, M. and Stormon, C., VLSI and Rule-based Systems, in: *VLSI for Artificial Intelligence*, Delgado-Frias, J. G. and Moore, W. R. (Ed.), pp. 95-108, Kluwer Academic Publishers, Norwell, MA, 1989.

[89] Kosko, B., Adaptive Bidirectional Associative Memories, *Applied Optics*, vol. 26, no. 23, pp. 4947-4960, Dec. 1987.

[90] Kosko, B., Bidirectional Associative Memories, *IEEE Transactions on System, Man, and Cybernetics*, vol. 18, no. 1, pp. 49-60, Jan./Feb. 1988.

[91] Kumagai, Y., Kamruzzaman, J. and Hikita, H., Further Cross Talk Reduction of Associative Memory and Exact Data Retrieval, *Proc. of IJCNN*, vol 3, pp. 1371-1378, San Francisco, 1993.

[92] Kumar, R., NCMOS: A High Performance CMOS Logic, *IEEE Journal of Solid-State Circuits*, vol. 29, no. 5, pp. 631-633, 1994.

[93] Kung, S. Y., *Digital Neural Networks*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[94] Lacher, R. C. and Nguyen, K. D., Hierarchical Architectures for Reasoning, in: *Computational Architectures Integrating Neural and Symbolic Processes : A Perspective on the State of the Art*, Sun, R. and Bookman, L. (Ed.), Chapter 4, pp. 117-150, Kluwer Academic Publishers, Norwell, MA, 1995.

[95] Lange, T. and Dyer, M., Frame Selection in a Connectionist Model, *Proceedings of the 11th Cognitive Science Conference*, pp. 706-713, Erlbaum, Hillsdale, NJ, 1989.

[96] Langley, P., *Elements of Machine Learning*, Morgan Kaufmann, Palo Alto, CA, 1995.

[97] Lavington, S. H., Wang, C. J., Kasabov, N. and Lin, S., Hardware Support for Data Parallelism in Production Systems, in: *VLSI for Neural Networks and Artificial Intelligence*, Delgado-Frias, J. G. (Ed.), pp. 231-242, Plenum Press, New York, 1994.

[98] LeCun, Y., Une Procedure D'apprentissage Pour Reseau a Seuil Assymetrique, *Proceedings of Cognitiva*, pp. 599-604, Paris, 1985.

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

[100] Lewis, H. R. and Papadimitriou, C. H., *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[101] Linde, R., Gates, R. and Peng, T., Associative Processor Application to Real-time Data Management, *AFIPS, Proceedings of the National Computer Conference*, vol. 42, pp. 187-195, 1973.

[102] Lippmann, R. P., An Introduction to Computing with Neural Nets, *IEEE ASSP Magazine*, pp. 4-22, Apr. 1987.

[103] Lont, J. B. and Guggenbühl W., Analog CMOS Implementation of a Multilayer Perceptron with Nonlinear Synapses, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 457-465, 1992.

[104] Lu, F. and Samueli, H., A 200-MHz CMOS Pipelined Multiplier-Accumulator Using a Quasi-Domino Dynamic Full-Adder Cell Design, *IEEE Journal of Solid-State Circuits*, vol. 28, no. 2, pp. 123-132, 1993.

[105] MacLennan, B. J., *Principles of Programming Languages : Design, Evaluation, and Implementation*, 2nd edition, CBS College Publishing, New York, NY, 1987.

[106] Masa, P., Hoen, K. and Wallinga, H., 70 Input, 20 Nanosecond Pattern Classifier, *IEEE International Joint Conference on Neural Networks*, vol. 3, Orlando, FL, 1994.

[107] Massengill, L. W. and Mundie, D. B., An Analog Neural Network Hardware Implementation Using Charge-Injection Multipliers and Neuron-Specific Gain Control, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 354-362, May 1992.

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

[109] McEliece, R. J., Posner, E. C., Rodemich, E. R. and Venkatesh, S. S., The Capacity of the Hopfield Associative Memory, *IEEE Trans. Inform. Theory*, vol. IT-33, no. 4, pp. 461-482, July 1987.

[110] McGregor, D., McInnes, S. and Henning, M., An Architecture for Associative Processing of Large Knowledge Bases (LKBs), *Computer Journal*, vol. 30, no. 5, pp. 404-412, Oct. 1987.

[111] McKenna, T. M., The Role of Interdisciplinary Research Involving Neuroscience in the Development of Intelligent Systems, in: *Artificial Intelligence and Neural Networks: Steps Towards Principled Integration*, Honavar, V. and Uhr, L. (Ed.), Academic Press, New York, NY, 1994.

[112] Medsker, L. R., *Hybrid Neural Network and Expert Systems*, Chapter 1, Kluwer Academic Publishers, Norwell, MA, 1994.

[113] Meyerowitz, A. L., Neural Networks: A Computer Science Perspective, *Naval Research Review*, vol. 43, No. 2. pp. 13-18.

[114] Micchelli, C. A., Interpolation of Scattered Data: Distance Matrices and Conditionally Positive Definite Functions, *Constructive Approximation*, pp. 11-22, 1986.

[115] Miclet, L., *Structural Methods in Pattern Recognition*, Springer-Verlag, New York, 1986.

[116] Miikkulainen, R, Subsymbolic Parsing of Embedded Structures, in: *Computational Architectures Integrating Neural and Symbolic Processes : A Perspective on the State of the Art*, Sun, R. and Bookman, L. (Ed.), Chapter 5, pp. 153-186, Kluwer Academic Publishers, Norwell, MA, 1995.

[117] Minsky, M., *Computation: Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, NJ, 1967.

[118] Minsky, M. and Papert, S., *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA, 1969.

[119] Mjolsness, E., Connectionist Grammars for High-Level Vision, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 423-451, Academic Press, San Diego, CA, 1994.

[120] Moon, G. et al., VLSI Implementation of Synaptic Weighting and Summing in Pulse Coded Neural-Type Cells, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 394-403, May 1992.

[121] Moulder, R., An Implementation of a Data Management system on associative Processor, *AFIPS, Proceedings of the National Computer Conference*, vol. 42, pp. 171-179, 1973.

[122] Mozer, M. C. and Bachrach, J., Discovering the Structure of a Reactive Environment by Exploration, *Neural Computation*, vol. 2., no. 4., p. 447, 1990.

[123] Mozer, M. C. and Das, S., A Connectionist Symbol Manipulator that Discovers the Structure of Context-Free Languages, *Advances in Neural Information Processing Systems* 5, p. 863, Morgan Kaufmann, San Mateo, CA, 1993.

[124] Naganuma, J., Ogura, T., Yamada, S. I. and Kimura, T., High-Speed CAM-Based Architecture for a Prolog Machine (ASCA), *IEEE transactions on Computers*, vol. 37, no. 11, pp. 1375-1383, Nov. 1988.

[125] Nakano, K., Associatron - A Model of Associative Memory, *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-2, no. 3, pp. 380-388, July 1972.

[126] Newell, A., Symbol Systems, *Cognitive Science* vol. 4, pp. 135-183, 1980.

[127] Ng, Y. H., Glover, R. J. and Chng, C. L., Unify with active Memory, in: *VLSI for Artificial Intelligence*, Delgado-Frias, J. G. and Moore, W. R. (Ed.), pp. 109-118, Kluwer Academic Publishers, Norwell, MA, 1989.

[128] Niranjan, M. and Fallside, F., Neural Networks and Radial Basis Functions in Classifying Static Speech Patterns, *Report CUED/FINFENG/TR 22*, University Engineering Department, Cambridge University, England, 1988.

[129] Noda, I. and Nagao, M., A Learning Method for Recurrent Neural Networks Based on Minimization of Finite Automata, *Proceedings of International Joint Conference on Neural Networks*, vol. 1, pp. 27-32, IEEE Press, Piscataway, NJ, 1992.

[130] Norman, D. A., Reflections on Cognition and Parallel Distributed Processing, in: *Parallel Distributed Processing*, McClelland, J., Rumelhard, D. and the PDP Research Group (Ed.), MIT Press, Cambridge, MA, 1986.

[131] Oh, H., *The Relaxation Method for Learning in Artificial Neural Networks*, Ph.D. Dissertation, Iowa State University, 1992.

[132] 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.

[133] Omlin, C. and Giles, C. L., Extraction and Insertion of Symbolic Information in Recurrent Neural Networks, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 271-299, Academic Press, New York, NY, 1994.

[134] Omlin, C. and Giles, C. L., Stable Encoding of Large Finite-State Automata in Recurrent Neural Networks with Sigmoid Discriminants, *Neural Computation*, vol. 8, no. 4, pp. 675-696, May 1996.

[135] Ozkarahan, E. A., Evolution and Implementation of the RAP Database Machine, *New Generation Computing*, **3**, pp. 237-271, 1985.

[136] Palm, G. et al., Knowledge Processing in Neural Architecture, in: *VLSI for Neural Networks and Artificial Intelligence*, Delgado-Frias, J. G. (Ed.), pp. 207-216, Plenum Press, New York, 1994

[137] Parekh, R. G. and Honavar, V., Automata Induction, Grammar Inference, and Language Acquisition, in: *Handbook of Natural Language Processing*, Moisl, H., Dale, R. and Somers, H. (Ed.), Marcel Dekker, New York, 1997.

[138] Parker, D. B., Learning Logic, *Invention Report*, S81-64, File 1, Office of Technology Licensing, Stanford University, 1982.

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

[140] Péter, R., *Recursive Functions in Computer Theory*, Halsted Press, New York, 1981.

[141] Pinkas, G., A Fault-Tolerant Connectionist Architecture for Construction of Logic Proofs, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 321-340, Academic Press, San Diego, CA, 1994.

[142] Pinkas, G., Propositional Logic, Nonmonotonic Reasoning, and Symmetric Networks – On Bridging the Gap Between Symbolic and Connectionist Knowledge Representation, in: *Neural Networks for Knowledge Representation and Inference*, Levine, D. S. and Aparicio IV, M.(Ed.), Chapter 7, pp. 175-203, Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

[143] Pollack, J. B., *On Connectionist Models of Language Processing*, Ph.D. Dissertation, Computer Science Department, University of Illinois, Urbana-Champaign, IL, 1987.

[144] Pollack, J. B., Recursive Distributed Representations, *Artificial Intelligence* **46**, pp. 77-105, 1990.

[145] Popescu, I., Hierarchical Neural Networks for Rules Control in Knowledge-Based Expert Systems, *Neural, Parallel & Scientific Computations* **3**, pp. 379-392, 1995.

[146] Powell, M. J. D., Radial Basis Function for Multi-variable Interpolation: A Review, *IMA Conference on Algorithms for the Approximation of Functions and Data*, RMCS, Shrivenham, England. Also *Report DAMTP/NA12*, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, 1985.

[147] Powell, M. J. D., Radial Basis Function for Multi-variable Interpolation: A Review, *Algorithms for Approximation*, Mason, J. C. and Cox, M. G. (Ed.), pp. 143-167, Oxford: Clarendon Press, 1987.

[148] Raghupathi, "RP" W. et al., Toward Connectionist Representation of Legal Knowledge, in: *Neural Networks for Knowledge Representation and Inference*, Levine, D. S. and Aparicio IV, M.(Ed.), Chapter 10, pp. 269-282, Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

[149] Renals, S. and Rohwer, R., Phoneme Classification Experiments Using Radial Basis Functions, *Proceedings of the IEEE/INNS International Joint Conference on Neural Networks*, vol. I, pp. 461-467, Washington, D. C., June 1989.

[150] Ripley, B. D., *Pattern Recognition and Neural Networks*, Cambridge University Press, New York, 1996.

[151] Robinson, I., The Pattern Addressable Memory: Hardware for Associative Processing, in: *VLSI for Artificial Intelligence*, Delgado-Frias, J. G. and Moore, W. R. (Ed.), pp. 119-129., Kluwer Academic Publishers, Norwell, MA, 1989.

[152] Robinson, M. E. et al., A Modular CMOS Design of a Hamming Network, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 444-456, 1992.

[153] Rodohan, D. and Glover, R., A Distributed Parallel Associative Processor (DPAP) for the Execution of Logic Programs, in: *VLSI for Neural Networks and Artificial Intelligence*, Delgado-Frias, J. G. (Ed.), pp. 265-273, Plenum Press, New York, 1994.

[154] Rogers, Jr., H., *Theory of Recursive Functions and Effective Computability*, MIT Press, Cambridge, MA, 1987.

[155] Rosenblatt, F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, Washington, D.C., 1962.

[156] Rumelhart, D. E., McClelland, J. L. and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1: Foundations*, MIT Press, Cambridge, MA, 1986.

[157] Salton, G. and McGill, M. J., *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

[158] Salton, G., *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesly, Reading, MA, 1989.

[159] Sanfeliu, A. and Alquezar, R., Understanding Neural Networks for Grammatical Inference and Recognition, in: *Advances in Structural and Syntactic Pattern Recognition*, Bunke, H. (Ed.), World Scientific, Singapore, 1992.

[160] Schneider, W., Connectionism: Is it a Paradigm Shift for Psychology?, *Behavior Research Methods, Instruments, and Computers*, **19**, pp. 73-83, 1987.

[161] Schulenburg, D., Sentence Processing with Realistic Feedback, *IEEE/INNS International Joint Conference on Neural Networks*, vol. IV, pp. 661-666, Baltimore, MD, 1992.

[162] Schuster, S. A., Nguyen, H. B., Ozkarahan, E. A. and Smith, K. C., RAP.2 - An Associative Processor for Databases and Its Applications, *IEEE Transactions on Computers*, vol. C-28, no. 6, pp. 446-458, 1979.

[163] Sedgewick, R., *Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1988.

[164] Selman, B. and Hirst, G., A Rule-based Connectionist Parsing System, *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA, 1985.

[165] Séquin, C. H. and Clay, R. D., Fault Tolerance in Artificial Neural Networks, *Proc. IJCNN*, vol. 1, pp. 703-708, San Diego, 1990.

[166] Servan-Schreiber, D., Cleeremans, A. and McClelland, J. L., Graded State Machines: The Representation of Temporal Contingencies in Simple Recurrent Neural Networks, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 241-269, Academic Press, New York, NY, 1994.

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

[168] Shavlik, J. W., A Framework for Combining Symbolic and Neural Learning, in: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), pp. 561-580, Academic Press, San Diego, CA, 1994.

[169] Siegelman, H. T. and Sontag, E. D., Turing-Computability with Neural Nets, *Applied Mathematics Letters*, vol. 4, no. 6, pp. 77-80, 1991.

[170] Sippu, S. and Soisalon-Soininen, E., *Parsing Theory, vol. II : LR(k) nad LL(k) Parsing*, Springer-Verlag, Berlin, 1990.

[171] Sloan, M. E., *Computer Hardware and Organization*, Science Research Associates, Chicago, 1976.

[172] Souček, B. and the IRIS Group (Ed.), *Neural and Intelligent Systems Integrations: Fifth and Sixth Generation Integrated Reasoning Information Systems*, John Wiley & Sons, New York, 1992.

[173] Stanfill, C. and Waltz, D., Toward Memory-Based Reasoning, *Communications of the ACM* **29**, pp. 1213-1228, 1986.

[174] Sun, G. Z., Giles, C. L., Chen, H. H. and Lee, Y. C., The Neural Network Pushdown Automation: Model, Stack and Learning Simulations, *Technical Report UMIA CS-TR-93-77*, Aug. 1993.

[175] Sun, R., On Variable Binding in Connectionist Networks, *Connection Science*, vol. 4, no. 2, pp. 93-124, 1992.

[176] 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, NY, 1994.

[177] Sun, R., Connectionist Models of Commonsense Reasoning, in: *Neural Networks for Knowledge Representation and Inference*, Levine, D. S. and Aparicio IV, M.(Ed.), Chapter 9, pp. 241-268, Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

[178] Sun, R., A Two-Level Hybrid Architecture for Structuring Knowledge for Commonsense Reasoning, in: *Computational Architectures Integrating Neural and Symbolic Processes : A Perspective on the State of the Art*, Sun, R. and Bookman, L. (Ed.), Chapter 8, pp. 247-281, Kluwer Academic Publishers, Norwell, MA, 1995.

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

[180] Swaminathan, G., Srinivasan, S., Mitra, S., Minnix, J., Johnson, B. and Iñigo, R., Fault Tolerance of Neural Networks, *Proc. of IJCNN*, vol 2, pp. 699-702, Washington DC, 1990.

[181] Thurber, K. J. and Wald, L. D., Associative and Parallel Processors, *Computing Surveys*, vol. 7, no. 4, pp. 215-255, 1975.

[182] Touretzky, D. S. and Hinton, G. E., Symbols among the Neurons: Details of a Connectionist Inference Architecture, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pp. 238-243, Morgan Kaufman, 1985.

[183] Turing, A. M., On Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Ser. 2, vol. 2, pp. 230-265, 1936.

[184] Uchimura, K. et al., An 8G Connection-per-second 54mW Digital Neural Network with Low-power Chain-Reaction Architecture, *ISSCC Dig. Tech. Papers*, pp. 134-135, San Francisco, CA, 1992.

[185] 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, NY, 1994.

[186] Ullman J. D., *Principles of Databases and Knowledge-base Systems*, vol. I, Chapter 6, Computer Science Press, Maryland, 1988.

[187] Van der Velde, F., Symbol Manipulation with Neural Networks: Production of a Context-free Language Using a Modifiable Working Memory, *Connection Science*, vol. 7, no. 3 & 4, pp. 247-280, 1995.

[188] Veezhinathan, J. and McCormick, B. H., Connectionist Plan Reminding in a Hybrid Planning Model, *Proceedings of IEEE International Conference on Neural Networks*, vol. II, pp. 515-523, San Diego, CA, 1988.

[189] Wasserman, P. D., *NeuralSource: The Bibliographic Guide to Artificial Neural Networks*, Van Nostrand Reinhold, New York, 1990.

[190] Watanabe, T. et al., A Single 1.5-V Digital Chip for a 10^6 Synapse Neural Network, *IEEE Transactions on Neural Networks*, vol. 4, no. 3, pp. 387-393, May 1993.

[191] Waterman, D. A. and Hayes-Roth, F. (Ed.), *Pattern-Directed Inference Systems*, Academic Press, New York, NY, 1978.

[192] Watrous, R. L. and Kuhn, G. M., Induction of Finite-State Languages Using Second-Order Recurrent Neural Networks, *Neural Computation*, vol. 4, No. 3, p. 406, 1992.

[193] Werbos, P. J., *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. Dissertation, Harvard University, 1974.

[194] Williams, R. J. and Zipser, D., A Learning Algorithm for Continually Running Fully Recurrent Neural Networks, *Neural Computation*, vol. 1, pp. 270-280, 1989.

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

[196] Yau, S. S. and Fung, H. S., Associative Processor Architecture – A Survey, *ACM Computing Surveys*, **9**, pp. 3-27, 1977.

[197] Zeng, Z., Goodman, R. M. and Smyth, P., Discrete Recurrent Neural Networks for Grammatical Inference, *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 320-330, Mar. 1994.