Investigations in neural network learning using geometrical approach

Wang, Di

2005

Wang, D. (2005). Investigations in neural network learning using geometrical approach.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2517>

<https://doi.org/10.32657/10356/2517>

Nanyang Technological University

Downloaded on 26 Apr 2023 19:56:32 SGT

INVESTIGATIONS IN NEURAL NETWORK LEARNING USING GEOMETRICAL APPROACH

WANG DI

**SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY**

2005

Investigations in Neural Network Learning Using Geometrical Approach

Wang Di

School of Computer Engineering

A thesis submitted to Nanyang Technological University
in fulfillment of the requirement for the degree of
Doctor of Philosophy

2005

Acknowledgements

I sincerely thank my supervisor Dr. Narendra S. Chaudhari, for his patient guidance and encouragement over the course of my research in Nanyang Technological University. His supervision has changed my view for life.

I also would like to acknowledge the financial support received from the School of Computer Engineering, Nanyang Technological University in the form of scholarship.

I also express my appreciation to Mr Tan Swee Huat and Mr Lim Hong Heng, the managers of the Center for Computational Intelligence for their kind assistance and support with all the convenience I needed. I am sincerity grateful to all of my colleagues in the Center for Computational Intelligence for their unhesitating help, earnest encouragement and long-standing friendship.

Most importantly, I would like to give special thanks to my parents and my husband for their invaluable encouragement and continuing spiritual support. They are always the source of motivation behind me.

Contents

Acknowledgements.....	i
Contents.....	ii
List of Figures.....	vi
List of Tables.....	viii
Glossary	ix
Abstract.....	x
Chapter 1 Introduction.....	1
1.1 Historical Background.....	1
1.2 Applications.....	3
1.3 Architecture and Components of Neural Networks.....	5
1.4 Neural Network Learning.....	9
1.4.1 Supervised Learning vs. Unsupervised Learning.....	11
1.4.2 Feed-Forward Learning vs. Feedback Learning.....	11
1.4.3 Loose Classification.....	12
1.5 Some Classical Learning Rules.....	13
1.5.1 Hebbian Learning Rule.....	14
1.5.2 Perceptron Learning Rule.....	15
1.5.3 Delta Learning Rule.....	16
1.5.4 Error Backpropagation Algorithm.....	18
1.5.5 Winner-Take-All Rule.....	19
1.5.6 Pseudo-Inversion Training.....	19
1.5.7 A Brief Summary.....	20
1.6 Constructive Learning.....	21
1.7 Our Research.....	23
1.7.1 Our Methods for Boolean Neural Networks.....	24
1.7.2 Our Methods for Real-Valued Problems.....	29
1.7.3 Contribution of Our Research.....	30
Chapter 2 Geometrical Learning.....	31
2.1 Geometrical Analysis for Boolean Neural Networks.....	31
2.1.1 Geometrical Properties of Boolean Vectors.....	31

2.1.2 Linear Separability of Boolean Vectors Based on Geometrical Concept.....	35
2.1.3 Linearly Separable Subsets of Boolean Vectors.....	39
2.2 Geometrical Analysis for Neural Networks in Real-Valued Inputs.....	40
2.3 Our Methods Based on Geometrical Concept for Neural Networks.....	42
 Chapter 3 Review of Learning Algorithms for Boolean Neural Networks.....	44
3.1 Overview of Boolean Neural Networks.....	44
3.2 Boolean-Like Training Algorithm (BLTA).....	47
3.3 Expand-and-Truncate Learning (ETL) Algorithms.....	53
3.4 Improved Expand-and-Truncate Learning (IETL) Algorithm.....	59
3.5 Newly Expand-and-Truncate Learning Algorithm (NETLA).....	61
3.6 Constructive Set Covering Learning Algorithm (CSCLA).....	62
3.7 Nearest-To-an-Exemplar (NTE) and Boolean K -Nearest Neighbor (BKNN).....	64
3.8 Summary.....	66
 Chapter 4 Linear Separability Analysis for Boolean Neural Networks.....	69
4.1 Introduction.....	69
4.2 Hypotheses about Linear Separability.....	70
4.3 Our Theorems for Linear Separability.....	79
4.4 Our Approaches for BNN Construction.....	84
 Chapter 5 Multi-Core Learning (MCL) and Multi-Core Expand-and-Truncate Learning (MCETL).....	86
5.1 Concept of Multi-Core Learning.....	86
5.2 Multi-Core Learning (MCL) Algorithm.....	88
5.2.1 Algorithm.....	88
5.2.2 Parameter Computation.....	92
5.3 Datasets for MCL.....	94
5.4 Multi-Core Expand-and-Truncate Learning (MCETL).....	103
5.5 An Example for MCETL.....	105
5.6 Concluding Remarks of MCL and MCETL.....	106
 Chapter 6 Fast Covering Learning Algorithm (FCLA) Based on Geometrical Expansion for Boolean Neural Networks.....	109
6.1 Introduction.....	109
6.2 Idea of Geometrical Method.....	113
6.3 Parameters for the Hidden Neurons.....	116
6.3.1 Primary Parameters: w_{ij} and t_1^j	116
6.3.2 Secondary Parameters: t_2^j and t_3^j	119
6.3.3 Parameters Discussion for a_1 , a_2 , and gap in the Third Method.....	122

6.4 Training the Hidden Layer.....	124
6.4.1 Geometrical Expansion.....	124
6.4.2 Geometrical Modification.....	130
6.5 Train the Output Layer.....	133
6.6 Hard-Limiter Activation Function.....	134
6.7 Datasets and Performance Analysis.....	135
6.7.1 Supervised Learning.....	135
6.7.2 Unsupervised Learning.....	151
6.8 Concluding Remarks for FCLA.....	157
 Chapter 7 Fast Covering Learning Algorithm for Real Valued Problems (FCLAR).....	159
7.1 Introduction.....	159
7.2 Geometrical Analysis.....	162
7.3 Neural Network Architectures.....	163
7.3.1 The Hidden Layer.....	164
7.3.2 The Output Layer.....	165
7.3.3 Activation functions.....	166
7.4 Supervised Learning and FCLAR.....	166
7.4.1 Primary Parameters: c and r_1	167
7.4.2 Secondary Parameters: r_2 and r_3	168
7.5 Our Proposed Method.....	169
7.5.1 Geometrical Expansion.....	169
7.5.2 Train the Hidden Layer.....	171
7.5.3 Sample Reordering.....	176
7.5.4 Train the Output Layer.....	177
7.6 Implement for Supervised Learning.....	177
7.6.1 Performance of FCLAR.....	178
7.6.2 Performance Comparison for Different Parameter Values: α and β	179
7.6.3 Performance Comparison with Traditional Training Algorithms.....	180
7.6.3.1 Computing iterations.....	181
7.6.3.2 Accuracy.....	181
7.6.3.3 Number of Hidden Neuron.....	181
7.6.4 Performance Comparison between FCLA and FCLAR.....	180
7.7 Unsupervised Learning and FCLAR.....	183
7.8 Conclusion Remarks.....	184
 Chapter 8 Concluding Remarks and Future Work.....	186
8.1 Concluding Remarks.....	186
8.2 Future Research.....	190
8.2.1 Neural Network Pruning.....	190
8.2.2 Insight to Cognitive Science.....	192

References.....193

List of Figures

Fig. 1.1 Single neuron illustration.....	7
Fig. 1.2 Sigmoidal function (1.1).....	7
Fig. 1.3 Sigmoidal function (1.2).....	7
Fig. 1.4 Hard-limiter function to $\{0,1\}$	7
Fig. 1.5 Hard-limiter function to $\{-1,1\}$	7
Fig. 1.6 A feed-forward neural network.....	9
Fig. 1.7 A feedback neural network.....	9
Fig. 1.8 A neural network taxonomy introduced by Bart Kosko.....	10
Fig. 2.1 Intersection of SHP and RHP.....	34
Fig. 2.2 Visualization of real-valued input space after data normalization.....	41
Fig. 3.1 A neural network structure constructed by BLTA.....	49
Fig. 3.2 Explicit representation (BLTA).....	49
Fig. 3.3 Generalization (for BLTA).....	49
Fig. 3.4 Modification (for BLTA).....	49
Fig. 3.5 A neural network structure constructed by ETL.....	54
Fig. 3.6 Training process of ETL.....	55
Fig. 3.7 Training process of IETL.....	60
Fig. 3.8 Training algorithm of CSCLA.....	64
Fig. 4.1 Visualization of separating a face for three-bit dimension.....	77
Fig. 5.1 Training process of MCL.....	88
Fig. 5.2 Training process of MCETL.....	88
Fig. 5.3 MCL algorithm.....	91
Fig. 5.4 Flowchart for MCL algorithm.....	92
Fig. 5.5 Circular region obtained by 6-bit quantization.....	96
Fig. 5.6 Neural network on MCL of 6-bit circle region approximation.....	97
Fig. 5.7 Neural network on ETL for 6-bit circle region approximation.....	97
Fig. 5.8 Neural network on IETL for 6-bit circle region approximation.....	97
Fig. 5.9 Neural network for 7-bit function problem on MCL.....	99
Fig. 5.10 Neural network for 7-bit function problem on ETL.....	99
Fig. 5.11 Neural network for 7-bit function problem on IETL.....	100
Fig. 5.12 Comparison between exact learning and approximate learning by MCL algorithm.....	102
Fig. 5.13 Neural network for 4-bit parity function on MCETL.....	106
Fig. 6.1 A three-layered neural network constructed by FCLA.....	113
Fig. 6.2 The process of geometrical expansion (in Boolean field).....	115
Fig. 6.3 A choice of r_2^j and r_3^j for FCLA.....	120
Fig. 6.4 Algorithm to compute the parameter <i>gap</i>	121
Fig. 6.5 Generalization process of FCLA.....	125
Fig. 6.6 Flowchart of generalization process for the hidden layer for FCLA	128
Fig. 6.7 Flow chart of one training circle for FCLA.....	129
Fig. 6.8 The modification process for FCLA.....	131
Fig. 6.9 Algorithm for modification process for FCLA.....	131

Fig. 6.10 Flowchart of modification process for FCLA.....	132
Fig. 6.11 Hidden neurons for sequence 1 of example 6.1.....	141
Fig. 6.12 Operations for sequence 1 of example 6.1.....	141
Fig. 6.13 Hidden neurons for sequence 2 of example 6.1.....	141
Fig. 6.14 Operations for sequence 2 of example 6.1.....	141
Fig. 6.15 Hidden neurons for sequence 1 of example 6.3.....	145
Fig. 6.16 Operations for sequence 1 of example 6.3.....	145
Fig. 6.17 Hidden neurons for sequence 2 of example 6.3.....	145
Fig. 6.18 Operations for sequence 2 of example 6.3.....	145
Fig. 7.1 Visualization of real-valued input space after data normalization.....	163
Fig. 7.2 Neural network structure for supervised learning.....	164
Fig. 7.3 Neural network structure for unsupervised learning.....	164
Fig. 7.4 The process of geometrical expansion for FCLAR.....	171
Fig. 7.5 The training algorithm of FCLAR.....	173
Fig. 7.6 Flowchart of FCLAR.....	174
Fig. 7.7 Flow chart of one training circle for FCLAR.....	175

List of Tables

Table 5.1 Weights and thresholds of MCL for 6-bit circle region approximation.....	98
Table 5.2 Weights and thresholds of IETL for 6-bit circle region approximation.....	98
Table 5.3 Weights and thresholds of MCL for 7-bit function problem.....	99
Table 5.4 Weights and thresholds of IETL for 7-bit function problem.....	100
Table 5.5 Weights and thresholds of MCL for 12-bit circle region approximation by exact learning.....	101
Table 5.6 Weights and thresholds of MCL for 12-bit circle region approximation by approximate learning.....	102
Table 5.7 Weights and thresholds of MCL for 4-bit parity function.....	103
Table 5.8 Weights and thresholds of MCETL for 4-bit parity function.....	105
Table 6.1 The fast covering learning process of 6-bit circle problem by $\sigma_1 = \sigma_2 = 1$	137
Table 6.2 The fast covering learning process of 6-bit circle problem by $a_1 = 1, a_2 = 1$	137
Table 6.3 The learning process of sequence 1 for example 6.2.....	143
Table 6.4 The learning process of sequence 2 for example 6.2.....	143
Table 6.5 FCLA & traditional algorithms: comparison for 5-bit parity problem.....	147
Table 6.6 Performance comparison for iris data classification.....	149
Table 6.7 Parameters analysis for a_1 and a_2 in FCLA.....	150
Table 6.8 Performance comparison for example 6.6.....	151
Table 6.9 Result for example 6.7 ($\xi_2=5, \xi_3=6$).....	153
Table 6.10 Result for example 6.7 ($\xi_2=6, \xi_3=7$).....	153
Table 6.11 Result for example 6.7 ($\sigma_1 = \sigma_2 = 1$).....	154
Table 6.12 Clustering of zoo dataset from Richard forsyth.....	156
Table 7.1 Iris data classification for the training set using FCLAR.....	178
Table 7.2 Iris data classification for the testing set using FCLAR.....	178
Table 7.3 Sensitive analysis for parameters of α and β	179
Table 7.4 Performance comparison for Iris data classification.....	180
Table 7.5 Performance comparison between FCLA and FCLAR (Iris dataset).....	183

Glossary

NN	Neural Network
BNN	Boolean Neural Network
BLTA	Boolean-Like Training Algorithm
ETL	Expand-and-Truncate Learning
IETL	Improved Expand-and-Truncate Learning
NETLA	Newly Expand-and-Truncate Learning Algorithm
CSCLA	Constructive Set Covering Learning Algorithm
NTE	Nearest-To-an- Exemplar
BKNN	Boolean K -Nearest Neighbor
MCL	Multi-Core Learning
MCETL	Multi-Core Expand-and-Truncate Learning
FCLA	Fast Covering Learning Algorithm
FCLAR	Fast Covering Learning Algorithm for Real-valued problems
SOM	Self-Organization Mapping
ART	Adaptive Resonance Theory
EBP	Error Propagation
WTA	Winner-Take-All
SHP	Separating Hypersphere
RHP	Reference Hypersphere
SITV	Set of Included True Vertices
ROA	Radius Of Attraction
EBPM	Error Propagation using Momentum
EBPT	Error Propagation using Tunneling
NoE	Number of Epochs
RBF	Radial Basis Function
SVM	Support Vector Machine

Abstract

Since the introduction of artificial neurons by McCulloch and Pitts in the early 1940's, many attempts aiming at fast learning algorithms with satisfactory performance have been made. Although neural networks have been successfully applied in many different areas, they still suffer from several problems, such as convergence problems and long training time. Almost all existing neural network learning algorithms are based on gradient descent technique, which needs iterative computations and long training time.

Determination of neural network structure to learn the training data set accurately without over-training is an important problem. In our framework, we express the network structure in terms of the number of hidden layers and the number of hidden neurons for each hidden layer. Constructive learning framework starts with a minimal structure and adds the hidden neurons until a satisfactory result is obtained. We develop frameworks for constructive learning. Our investigations start with Boolean neural networks; however we also adopt them for real valued inputs.

Our approaches are based on geometrical expansion and can avoid iterative computations. To do this, each hidden neuron is expressed in terms of its corresponding hyperplane or hypersphere with three central radii. The training data are input to the neural network sequentially. The training process of our methods can be visualized by the geometrical expansion process of these hyperspheres based on the geometrical location of the training data and their input sequence. If we define one scan of the training dataset as one training circle, each data item is learned immediately or backed up to be re-considered in the next training circle. The data item is removed immediately after it is learned. In this sense, each data item in the training set is only learned once. By doing this, the iterative

computations in traditional training algorithms are avoided. Given a training sequence in an arbitrary order, our method learns “easy” samples first and learns “confusing” samples after the “easy” ones by the backing up and re-consideration process, which makes our method more intelligent. In this thesis several methods to control the geometrical expansion process are proposed and analyzed. Our approaches turn out to gain faster training speed with satisfactory accuracy. We discuss problems both with Boolean inputs and with real-valued inputs respectively.

Our algorithms can be applied to both supervised learning (classification as an example) and unsupervised learning (clustering problems as examples). The advantages of our methods are illustrated by giving the results of suitable experiments.

Chapter 1

Introduction

1.1 Historical Background

The idea of neural networks was first proposed in the late 19th and early 20th centuries, motivated by the desire to understand the brain and to emulate some of its strengths by Hermann Von Helmholtz, Ernst Mach and Ivan Pavlov. It was primarily an interdisciplinary research in physics, psychology and neurophysiology.

The first formulation of the theory on neural computing is usually credited to McCulloch and Pitts [McCulloch 43], who proposed the arithmetic and logical functionality of neural networks. Other major contributors are Hebb [Hebb 49] who proposed unsupervised mechanism, and Lashley [Lashley 50] who formulated theories about neural functions of the brain, and insisted on the distributed processing machine of the brain. During the 1950's, several advances were made. They included the development of formulation of

perceptron neural networks and associate learning rule by Rosenblatt [Rosenblatt 58, 62]. Rosenblatt articulated the promise of neural inspired approaches for computation. Widrow and Hoff [Widrow 60, 62] introduced a learning algorithm to train adaptive linear neural networks. Nilsson [Nilsson, 65] gave a good summary of many developments in the 1950's in his book *Learning Machines*. In the 1970's Grossberg [Grossberg 76, 78] developed detailed mathematical analysis of the properties of neural networks, and developed theories of perceptron and memory. Minsky and Papert [Minsky, 69] showed that Rosenblatt's Perceptron could only solve certain classes of problems and had limitations. They mathematically analyzed the perceptron and demonstrated that perceptron and other one or two-layered neural networks could never solve problems that were not linearly separable. But a training procedure for a neural network with more layers could solve linearly inseparable problems. Other achievement included Kohonen [Kohonen 72] and Anderson [Anderson 72]. Later on, Hopfield [Hopfield 82] used statistical mechanics to explain the operation of a certain class of recurrent networks that could be used as an associative memory. Kohonen [Kohonen 82] described unsupervised training algorithms and self-organizing mappings. The well-known error back propagation algorithm was established by Werbos [Werbos 74] and later on formulated by Rumelhart [Rumelhart 86]. After that, although many attempts of improvements were made by Sejnowski [Sejnowski 87] and Fahlman [Fahlman 88], the improvements were limited. One of the latest important discoveries was made by Fukushima in 1986 [Fukushima, 86]. His neocognition neural networks represent a biocybernetic approach to visual pattern recognition.

Neural networks became hotter and hotter after the 1990's. The developments of powerful new architectures, learning algorithms, and their success in solving real world problems in many fields have established a presence for neural networks as a main part to build intelligent systems [Ian 99]. Neural networks have the capability to learn from examples, to generalize from new data, to work as universal computational devices, as well as the capability to predict, cluster, interpolate and approximate.

1.2 Applications

Nowadays neural networks attract attention from many fields. Computer scientists want to find out the properties of information processing and learning systems in general with neural networks. Many engineers want to exploit the capabilities of neural networks in their application areas. Cognitive scientists view neural networks as a possible apparatus to describe models of thinking and conscience. Neural physiologists use neural networks to describe and explore medium-level brain function. Physicists use neural networks to model phenomena in statistical mechanics and a lot of other tasks. Biologists use neural networks to interpret nucleotide sequence. Philosophers are also interested in neural networks for their good performance. This tremendous interest should be attributed to the development of learning theory, sophisticated training techniques of neural network architecture, and easy realization of specialized hardware for neural networks [Zaghloul 04a].

Generally speaking artificial neural networks have proven to be successful in the areas of

approximation, prediction, classification, expert systems, and system control.

From the viewpoint of applications: neural networks are widely utilized in many areas:

- (i) automotive guidance systems, damage assessment, machine vision [Bertrand 92, Neusser 93, Yu 97, Suganuma 98, Kontaratos 98];
- (ii) financial systems, banking, stock market analysis, predicting bank performance, credit card authorization [Swicegood 98, Wilkinson 93, Smith 02, Moutinho 02, Serrano 97, Curry 92];
- (iii) image processing including image matching, image compression, computer vision, etc. [Kulkarni 95, Kenyon 92, Zanela 00];
- (iv) medicine, including EEG and ECG signal analysis and understanding, diagnosis and assessment of various diseases, medical image processing, etc. [Paul 90, Eastaugh 01, Enderie 94, Chuang 02];
- (v) military system defense, weapon steering, target tracking, object discrimination, facial recognition;
- (vi) pattern recognition, including feature extraction [Saund 89], radar signal classification and analysis, signal procession, fingerprint identification, handwriting analysis, etc. [Nigrin 93, Atkinson 99, Clarkson 95, Kulkarni 95, Linggard 92, Masters 94];
- (vii) power systems, including systems state estimation [Kenyon 92], transient detection and classification, fault detection, diagnosis, and recovery [Maki 97, Li 97, Goel 00, Sreedhar 95, Iordache 91, Lehrasab 02, Dellomo 99, Wang 00, Toomarian 93, Moravej 01];

- (viii) speech recognition, speech comparison, text to speech synthesis [Chen 96, Olafsson 92, Lee 92];
- (ix) transportation, vehicle scheduling, routing systems, etc. [Potvin 93, Costa 97, Morshed 98];
- (x) applications to environmental systems for river stage, flood or rainfall forecasting [Tangang 98, Hsu 98, French 92, French 98, Coulibaly 00, Thirumalaiah 98, Tingsanchali 00, Walley 98, Yang 98, Zhang 97, Braddock 98], Rainfall-runoff modeling [Hsu 95, Mason 96, Shamseldin 97a, Shamseldin 97b], river flow/runoff prediction [Karinanithi 94, Elshorbagy 00], drought/tide estimation [Dowla 93, Crespo 93, Conrads 02].

1.3 Architecture and Components of Neural Networks

Neural networks can be defined as cognitive information processing structures based on the models of brain function. Also some other researchers look neural networks as massive parallel interconnected networks of simple elements and their hierarchical organizations, which are intended to interact with the objects of the real world in the same way as the biological nervous system does. Neural networks are also called neural nets, connectionist models, collective models, parallel distributed models, and artificial neural networks by different researchers.

A neural network consists of a large number of simple processing elements called neurons, units, cells or nodes (Fig. 1.1). Each neuron is connected to other neurons by means of

direct communication links, each with a connection weight. These weights represent information used by the network to solve special problems. Each neuron has an internal state, called its activation level. This activation level is a function of its inputs. Most activation functions are differentiable sigmoidal activation functions, for examples, equation (1.1) and equation (1.2). Neurons send their activation as a signal to several other neurons. A neuron only sends a signal at one time.

$$f(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{x})}. \quad (1.1)$$

$$f(\mathbf{x}) = \frac{2}{1 + \exp(-\mathbf{x})} - 1. \quad (1.2)$$

Using this kind of activation functions, a neuron maps the entire space into an open segment (0, 1) as shown in Fig. 1.2, or (-1, +1) as shown in Fig. 1.3.

In some other cases, hard-limiter function is applied to map the inputs to {0, 1} by using (1.3) as shown in Fig. 1.4, or {-1, 1} by using (1.4) as shown in Fig. 1.5.

$$f(\mathbf{x}) = \begin{cases} 1 & \phi(\mathbf{x}, \mathbf{w}, T) \geq 0 \\ 0 & \phi(\mathbf{x}, \mathbf{w}, T) < 0 \end{cases}. \quad (1.3)$$

$$f(\mathbf{x}) = \begin{cases} 1 & \phi(\mathbf{x}, \mathbf{w}, T) \geq 0 \\ -1 & \phi(\mathbf{x}, \mathbf{w}, T) < 0 \end{cases}. \quad (1.4)$$

Fig. 1.1 Single neuron illustration

Fig. 1.2 Sigmoidal function (1.1)

Fig. 1.3 Sigmoidal function (1.2)

Fig. 1.4 Hard-limiter function to $\{0,1\}$

Fig. 1.5 Hard-limiter function to $\{-1,1\}$

A neural network is characterized by its architecture (connection pattern between the neurons), its method to determine the connection weights (training algorithm), and its activation functions. We have discussed the activation functions above. Given an input, the response of a neural network depends on its connection weights. A neural network usually consists of several layers including one input layer, one output layer, and one or

more hidden layers. Within each layer, neurons usually have the same type of activation functions and the same pattern of connections to other neural layers. Normally, neurons within a layer are not interconnected at all, and neurons between two contiguous layers are fully interconnected. If there is no other connection, we call this kind of neural networks as feed forward neural networks as shown in Fig. 1.6. If part of the input features depend on the result of the output layer, or there are feedbacks from the upper neural layer to the lower neural layer, we call this kind of neural networks as recurrent neural networks as shown in Fig 1.7.

Generally speaking, there are several neural network architectures: Perceptron neuron networks, Winner-Take-All neural networks, Dynamic feedback neural networks (Hopfield networks, Bi-directional associative memories), Self-Organization Mapping neural networks, Adaptive Resonance Theory (ART) neural networks and Bayesian neural networks.

The process to gain connection weights and activation functions is called the “training” process. Usually neural networks with more hidden layers can solve more complicated problems, but the training process for such neural networks is more complex. For researchers who want to gain a neural network with better performance, how to train a neural network is the key problem.

Fig. 1.6 A feed-forward neural network

Fig. 1.7 A feedback neural network

1.4 Neural Network Learning

Learning is a basic task of neural networks. It refers to the rules, which govern the updating of parameters within neurons and between neurons.

There are various possible taxonomies for neural network learning. A popular one was proposed by Kosko [Kosko 90, 91, 92]. It classifies neural network learning in terms of how neural networks are encoded (how they store knowledge) and how they are decoded (how, once knowledgeable, they process new input data). In its encoding property, neural

network learning can either be supervised [Ryan 98] or unsupervised [Becker 96, Linsker 98]; in its decoding property, neural network learning can be either feed-forward (open-loop) or feedback (closed-loop). Fig 1.8 shows the taxonomies of neural network learning proposed by Bart Kosko.

Decoding		
Feed-forward	Feedback	
I	II	Supervised
III	IV	Unsupervised

Fig. 1.8 A neural network taxonomy introduced by Bart Kosko

Most applications are focused on training algorithms in quadrants I (supervised feed-forward learning) and III (unsupervised feed-forward learning). However quadrant II (supervised feedback learning) contains some important examples, and quadrant IV is considered as interesting fusions of contrasting approaches, called “unsupervised feedback” learning. In general, boundaries between the quadrants of these taxonomies are somewhat imprecise, and there are instances of network paradigms that do not always fit exactly into quadrants, but overlap the boundaries.

1.4.1 Supervised Learning vs. Unsupervised Learning

In terms of encoding property, neural network learning can be either supervised or unsupervised.

A supervised neural network is one in which, during training, the network is taught what response it should make to each input it receives. These input-output pairs are used for supervised learning of a neural network. The network outputs its response, which is a function of its initial connection weights and inputs. The network compares the actual response (output) with the taught response (output) and then adjusts its weights in such a way as to move it in the direction of what is being taught. Once the neural network has been trained, the training data (input-output pairs) is removed and new unknown patterns are presented. Then in its non-learning phase, the neural network responds to the new coming patterns according to the rules (activation functions and connection weights) it learned during the learning phase.

Unsupervised neural networks learn problems using built-in rules for self-modification. Such neural networks change their connection weights in response to the inputs they receive in the training phase, without any intervention by the teacher.

1.4.2 Feed-Forward Learning vs. Feedback Learning

During the non-learning phase a feed-forward neural network is a fixed structure to the

new inputs regardless of how it was encoded. After the learning phase, the connection weights and the activation functions are fixed and its operation is a deterministic mechanism.

On the other hand, a feedback neural network is a dynamic system during the non-learning phase. The output nodes response to the combination of the input nodes and the feedback from the output nodes. The state of the network reverberates until it reaches a global stable status. When it reaches its stability, the responses of the output nodes become fixed or oscillate locally around a set of fixed values. The dynamic behavior of feedback neural networks is interesting during decoding. They do not necessarily learn during encoding. Instead, the connection weights are constructed initially from a set of exemplars. Then during their non-learning phase, it reverberates to a solution in response to new inputs it receives.

1.4.3 Loose Classification

Back-propagation is a classical learning paradigm of supervised feed-forward neural network learning. Self-organizing mapping is a classical paradigm of unsupervised feed-forward neural networks. And Hopfield neural networks [Hopfield 82] form a classical paradigm of feedback neural networks. However this taxonomy is a loose method to classify neural networks. For example, the back-propagation learning belongs to supervised feed-forward neural network learning, but it includes feedback process in the training process.

In most applications, quadrant I is the simplest one to apply. These neural networks are used to solve problems where feedback is not present and external information is available to supervise the learning process. As the situation becomes more complex, because of the absence of supervisory information to teach the network, we need to move to other quadrants. If a problem is to establish boundaries between various classes of information, then it is operated in quadrant III. Clustering paradigms are operated in quadrant III. While applying paradigms in quadrant III, we should know how many classes we wish to establish, and we let the neural networks determine the boundaries. A greater generality degree is found in quadrant IV. In this quadrant the number of clusters and the boundaries between clusters are unknown.

1.5 Some Classical Learning Rules

We define the learning rate δ as a decreasing function of time that controls the size of weight and bias changes during learning, the learning constant rate α as a training constant that controls the size of weight and bias changes during learning, and the correctional constant λ as a parameter to compute the learning constant rate. And in the following expression, o is the actual neuron output and d as the desired neuron output. Then the connection weights are updated according to (1.5) and (1.6):

$$\mathbf{W}(n+1) = \Delta \mathbf{W}(n) + \mathbf{W}(n) . \quad (1.5)$$

$$\Delta \mathbf{W}(n) = \alpha \delta \mathbf{x} . \quad (1.6)$$

Several classical learning rules have been developed in early years based on the

parameters defined above.

Examples of learning rules are:

- Correlation rule, $\delta = d$. (1.7)

- Perceptron fix rule, $\delta = d - o$ and α is a constant. (1.8)

- Perceptron adjustable rule: $\delta = d - o$ and $\alpha' = \alpha \lambda \frac{\mathbf{x}\mathbf{W}^T}{\mathbf{x}\mathbf{x}^T} = \alpha \lambda \frac{\mathbf{x}\mathbf{W}^T}{\|\mathbf{x}\|^2}$. (1.9)

- LMS (Widrow-Hoff) rule, $\delta = d - \mathbf{x}\mathbf{W}^T$. (1.10)

- Delta Rule, $\delta = (d - o)f'$. (1.11)

- Pseudoinverse rule, $\delta = d - \mathbf{x}\mathbf{W}^T$ and $\mathbf{W} = (\mathbf{x}^T \mathbf{x})^{-1} \mathbf{x}^T d$. (1.12)

- Iterative pseudoinverse rule, $\delta = d - \mathbf{x}\mathbf{W}^T$ and $\mathbf{W} = (\mathbf{x}^T \mathbf{x})^{-1} \mathbf{x}^T d \frac{d - o}{f'}$. (1.13)

- Hebbian rule, $\delta = o$. (1.14)

Now we describe some learning rules in detail as follows.

1.5.1 Hebbian Learning Rule

For Hebbian learning rule, the learning rate δ is equal to the neuron's outputs o [Hebb 49]:

$$\delta = o. \quad (1.15)$$

The connection weights are updated as

$$\mathbf{W}(n+1) = \Delta \mathbf{W}(n) + \mathbf{W}(n), \quad (1.16)$$

$$\Delta \mathbf{W}(n) = \alpha \delta \mathbf{x}, \quad (1.17)$$

where α is the learning constant.

Then the increment of connection weights can be expressed by:

$$\Delta \mathbf{W}(n) = \alpha o \mathbf{x} . \quad (1.18)$$

The learning rule requires the weight initialization at small random values around zero prior to the learning phase. Hebbian learning rule represents a purely feedforward, supervised learning. Frequent input patterns will have more influence at the neuron's weights and will eventually produce the larger outputs.

1.5.2 Perceptron Learning Rule

For perceptron learning rule, the learning rate is the difference between the desired output and the actual output of the neural network [Rosenblatt 58, Zurada 92]. Perceptron learning is supervised learning and the learning rate is computed by:

$$\delta = d - o . \quad (1.19)$$

For bipolar neurons (in which a single axon and dendrite arise at opposite poles [Tom 97]), the perceptron learning rate is

$$\delta = d - \text{sign}(\text{net}(\mathbf{x})) . \quad (1.20)$$

where $\text{net}(\mathbf{x}) = \mathbf{x} \bullet \mathbf{w}^T$ (1.21)

Then the perceptron learning rule is:

$$\mathbf{W}(n+1) = \Delta \mathbf{W}(n) + \mathbf{W}(n) , \quad (1.22)$$

$$\text{where } \Delta \mathbf{W}(n) = \alpha \delta \mathbf{x} = \alpha (d - \text{sign}(\text{net}(\mathbf{x}))) \mathbf{x}, \quad (1.23)$$

where α is the learning constant.

As the function $\text{sign}(\text{net}(\mathbf{x}))$ returns either +1 or -1, the output value equals to either 1 or -1. The weight adjustment is zero when the desired output and the actual output agree.

And the connection weights are adjusted if and only if the output is incorrect. That is:

$$\Delta \mathbf{W}(n) = \begin{cases} \pm 2\alpha \mathbf{x} & o \neq d \\ 0 & o = d \end{cases}. \quad (1.24)$$

Even though these characteristics of the perceptron might give the impression of a fairly inefficient rule, perceptron learning is of central importance for supervised learning for neural networks. In case of discrete perceptron learning, this rule is called perceptron rule. In case of continuous perceptron learning, this rule is called Delta learning.

1.5.3 Delta Learning Rule

Delta learning rule [Zurada 92, Wilamowski 96] applies sigmoidal activation function, differentiable total error and gradient descent computing.

Delta learning rule is:

$$\mathbf{W}(n+1) = \Delta \mathbf{W}(n) + \mathbf{W}(n), \quad (1.25)$$

$$\text{where } \Delta \mathbf{W}(n) = \alpha \delta \mathbf{x}. \quad (1.26)$$

The learning rate δ is defined as:

$$\delta = (d - o)f' , \quad (1.27)$$

where α is the learning constant.

Substituting (1.27) into (1.26) the expression of $\Delta W(n)$ is given as follows:

$$\Delta W(n) = \alpha(d - o)f' \mathbf{x} . \quad (1.28)$$

Then Delta learning rule is written as:

$$\mathbf{W}(n+1) = \mathbf{W}(n) + \alpha(d - o)f' \mathbf{x} . \quad (1.29)$$

Opposed to perceptron learning rule, Delta learning rule is characterized by its continuous nature. Total error of a neural network is defined as:

$$TE = \sum_{p=1}^m (d_p - o_p)^2 , \quad (1.30)$$

where d_p and o_p are the desired output and the real output for pattern p , and m is the total number of patterns.

From

$$\frac{\partial(TE)}{\partial w_i} = - \sum_{p=1}^m 2(d_p - o_p) \frac{\partial o_p}{\partial w_i} , \quad (1.31)$$

$$\frac{\partial o_p}{\partial w_i} = \frac{\partial o_p}{\partial net_i} \frac{\partial net_i}{\partial w_i} f' x_i , \quad (1.32)$$

Δw_i is expressed by:

$$\Delta w_i = 2\alpha \sum_{p=1}^m (d_p - o_p) f' x_i^p . \quad (1.33)$$

1.5.4 Error Backpropagation Algorithm

Delta learning rule can be generalized for multi-layer neural networks. In case of multiple outputs, the weight increment is an accumulation of weight modifications from errors on all outputs [Wilamowski 96]:

$$\Delta w_p = \alpha \sum_{o=1}^{no} \sum_{p=1}^{np} [(d_{op} - o_{op}) F_o' \{z_p\} f'(net_p) x_p], \quad (1.34)$$

where np is the number of patterns, no is the number of outputs, $F_o' \{z_p\}$ is the slope of output o for pattern p , and $f'(net_p)$ is the slope of activation function. Term $(d_{op} - o_{op})$ means that weights are changed only when error exists.

Error Backpropagation (EBP) neural networks and gradient descent method generally provide good results. The objective of EBP is to reduce the total errors between the desired outputs and the actual outputs by gradient descent computing. However, in case of large or small learning rate, the process can either zigzag or be stuck in local minima. Different heuristic approaches can be found in the literature with variable learning rates. In addition, EBP has a tendency to oscillate. In order to smooth the oscillating process, Rumelhart had proposed the following weight adaptation [Rumelhart 86b]:

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1), \quad (1.35)$$

which was later formulated by Sejnowski and Rosenberg [Sejnowski 87] as:

$$w_{ij}(n+1) = w_{ij}(n) + (1 - \alpha) \Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1). \quad (1.36)$$

EBP algorithm can be sped up in different ways. An improved EBP learning algorithm was proposed by Fahlman [Fahlman 88], and is known as the quickprop.

Although EBP is a significant breakthrough in neural network research at that time, it has several disadvantages, such as oscillation, slow convergence, sensitivity to network parameters.

1.5.5 Winner-Take-All Rule

This learning rule differs substantially from any other rule previously discussed. This rule is used for unsupervised neural network learning. Winner-Take-All (WTA) is typically used for learning statistical properties of the inputs [Hecht-Nielsen 87a, 87b, 88, 90, Zurada 92]. WTA is also known as Kohonen's Network learning [Kohonen 82]. One of the neurons within a layer with the maximum response to the input $\mathbf{x}$ is declared as the winner. As a result, only the connection weights for the winner neuron are adjusted in a given unsupervised learning step. WTA is applied to unsupervised learning for clustering which strongly depends on: the initial set of randomly chosen weights, the number of hidden neurons and the order in which these training patterns are applied.

1.5.6 Pseudo-Inversion Training

Pseudo-Inversion training is derived from a simple problem of solving the following equation:

$$\mathbf{A}_{n \times m} \times \mathbf{X}_{m \times 1} = \mathbf{B}_{n \times 1} . \quad (1.37)$$

Solution we intend to look for is:

$$\mathbf{X}_{m \times 1} = \mathbf{A}_{n \times m}^{-1} \times \mathbf{B}_{n \times 1} . \quad (1.38)$$

A unique inverse can be found under the condition that matrix A is square with full rank.

However, when the rank is less than the dimension of the matrix, or when the matrix is not square, a number of possible non-unique cases may arise.

Generally speaking, the pseudo-inverse of an $m \times n$ matrix A is an $n \times m$ matrix:

$A^\perp$. We have:

$$AA^\perp A = A, \quad A^\perp AA^\perp = A^\perp, \quad \text{and} \quad (AA^\perp)^\perp = AA^\perp. \quad (1.39)$$

Apply these reasoning to neural networks, the Pseudo-Inversion learning rule is:

$$\mathbf{W} = (\mathbf{x}^T \mathbf{x})^{-1} \mathbf{x}^T d = \mathbf{x}^\perp d, \quad (1.40)$$

where

$$\mathbf{x}^\perp = (\mathbf{x}^T \mathbf{x})^{-1} \mathbf{x}^T. \quad (1.41)$$

This algorithm provides satisfactory results only with linear activation functions.

1.5.7 A Brief Summary

Neural networks are hierarchical structures of neurons arranged in layers. Inputs are applied at the input layer and targets are expected at the output layer. In supervised learning, a neural network responses to the input forward layer by layer, and computes its output. The output is then compared to the desired target. If there is an error (defined by the difference between the desired and actual output of the network), the parameters of the neural network are modified according to this error to the direction of less total errors.

This procedure is repeated until the error reaches some acceptable value. In unsupervised learning, the parameter modification process is based on the distance between the input and some exemplars. The objective is to decrease the total distance between all inputs and their corresponding exemplars.

It should be noticed that all the training algorithms discussed above are based on gradient descent technique. Gradient descent technique requires a great amount of iterative computation, which is time consuming. In addition, all of the above training algorithms fix the structures of neural networks before training. But it is difficult to determine a network structure (the number of hidden layers and the number of hidden neurons in each hidden layer), which can both guarantee convergence and avoid over fitting before training. Hence, many methods which do not prefix the network structure are proposed. These methods are called constructive learning. Constructive learning attracts many researchers because they adjust not only the parameters, but also the structure of a neural network by adding hidden neurons during training until a satisfactory result is obtained.

1.6 Constructive Learning

Non-constructive learning algorithms fix a neural network's structure before training. Hence they might not guarantee convergence. Constructive learning algorithms allow for incremental construction to avoid inappropriate choices of neural network topology. Non-constructive learning algorithms search for suitable connection weights in a priori fixed network structure. But there is no efficient method to determine the optimal

topology for a given problem. Too small a network is unable to learn the problem adequately, while too large a network tends to over fit the training data and result in a poor generalization performance. Usually non-constructive learning algorithms try different network topologies with different number of hidden neurons, and select a suitable one. This trial-and-error approach is computationally expensive.

On the contrary, constructive learning algorithms begin with a minimal neural network structure, or empty hidden layer, and dynamically increase the network by adding hidden neurons until a satisfactory solution is obtained. A remarkable characteristic of constructive learning algorithms is that they can change the topology of a neural network during training. Constructive learning algorithms search for the optimal topology as well as optimal parameters. Any hidden neuron added to a neural network aims to decrease training error. We can always decrease the training error by adding hidden neurons. Hence, based on finite inputs, all constructive learning algorithms guarantee convergence for the training data.

Many researchers have focused their research on designing, simple, fast, and effective training algorithms for neural networks based on constructive learning. Most of these algorithms are designed for Boolean neural networks, such as BLTA [Gray 92], ETL [Kim 95], IETL [Yamamoto 97], CSCLA [Ma 01], MCL [Wang 03a] and MCETL [Wang 03b]. These algorithms are explained in detail in chapter 3 and chapter 5. Constructive learning for real-valued inputs is also a hot topic [Meng 03].

Our research is focused on the investigations in constructive learning for both Boolean neural networks (in chapter 3, 4, 5 and 6) and real-valued neural networks (in chapter 7).

1.7 Our Research

To summarize, neural networks have been widely applied in many fields for their good performance. However, there are two fatal defects in traditional neural network learning algorithms: long training time and prefixed neural network structures. So many researchers focus their research on learning methods which do not prefix the neural network structure. The objective of our research is to design learning algorithms for neural networks based on geometrical expansion, which avoid the iterative computations in gradient descent-based algorithms. Our methods belong to constructive learning, which constructs neural network topologies during training. By doing this, our methods avoid blind selection of network structure before training, which is time consuming and cannot guarantee convergence.

Our work is divided into two parts, the learning methods for Boolean problems and the learning methods for real-valued problems.

1.7.1 Our Methods for Boolean Neural Networks

It is easy for us to begin with Boolean neural networks due to the special geometrical properties of Boolean vectors [Wang 03b, 04a]. In our work, we first introduce several

novel learning algorithms for Boolean Neural Networks. Our learning algorithms originate the ideas from Boolean-Like Training Algorithm (BLTA) [Gray 92] and Expand-and-Truncate Learning (ETL) [Kim 95]. BLTA and ETL algorithms are two excellent training algorithms for Boolean neural networks. In the late 1990's, other researchers also developed novel constructive learning algorithms for Boolean neural networks by improving BLTA and ETL, such as Improved Expand-and-Truncate Learning (IETL) algorithm [Yamamoto 97], Novel Expand-and-Truncate Learning Algorithm (NETLA) [Sung 02] and Constructive Set Covering Learning Algorithm (CSCLA) [Ma 01]. These methods are described in chapter 3.

In the investigation of learning algorithms for Boolean neural networks (BNNs), we identify several problems that have not been adequately tackled in existing learning algorithms for BNNs. Some of them are given below.

Balance between generalization and precision: All of the existing learning algorithms for Boolean neural networks (except for BLTA [Gray 92]) do not possess generalization capability. But in some cases, we need our learning algorithms to possess generalization capability. We want to predict possible data in the future and judge the property of the new coming data through the dataset we have known. The generalization advantage of our method is realized in the construction of the nodes and the weights. BLTA possesses generalization capability. It doubles the information represented by a hidden neuron based on Karnaugh map. When BLTA adds a vertex to a hidden neuron, this hidden neuron represents n samples before generalization and represents $2n$ samples after generalization.

The learning process of BLTA is explained in section 3.2 in detail. In the worst case, in each generalization circle, $(n-1)/2n$ errors will be introduced in the learning system. So after k circles of generalization, $1 - \frac{1+k}{2^k}$ errors are introduced. Hence, after k circles, the precision is only $\frac{1+k}{2^k}$ in the worst case. However, ETL and other algorithms do not possess generalization capability.

Low Training Speed: Although ETL has increased the training speed dramatically compared with gradient-descent based algorithms, ETL and IETL need to search all vertices in the training set, computing the value of $\sum_{i=1}^n w_i x_i$, to determine each neuron in the hidden layer, where x_i is the i th bit of the input vector and w_i is the connection weight for x_i . This operation takes long training time.

Identification of larger linearly separable subsets of vertices: Due to the second problem, we consider how to determine each neuron in the hidden layer by computing the value of $\sum_{i=1}^n w_i x_i$ for the current trained vertex, not all vertices in the training set. If a subset is linearly separable, we can separate it without computing the value of $\sum_{i=1}^n w_i x_i$ for all vertices in the training set. So the third problem we try to solve is identifying larger linearly separable subsets. Although some previous investigation about linear separability has been done [Park 91a], it may not be comprehensive. A further investigation is necessary for fast learning algorithms for Boolean neural networks. Hence, we investigate the properties of linearly separable subsets.

Topology: In addition, in previous training algorithms for Boolean neural networks (BNNs), a vertex (in the training set) must be selected as the core, based on which the remaining vertices are learned. The selected core and the training sequence affect the neural networks' structure dramatically. A little change of the input order of the training data may result in a quite different neural network structure (the number of hidden neurons in a neural network and the connection weights). So our research is also focused on the learning algorithms for Boolean neural networks, which are less affected by the selected core and the training sequence. It is necessary to do this if we want a more intelligent learning algorithm.

To tackle these problems, we take the following approaches.

We develop a Multi-Core Learning (MCL) algorithm and a Multi-Core Expand-and-Truncate Learning (MCETL) algorithm. MCL and MCETL begin with several core vertices, and need fewer hidden neurons than ETL and IETL in most cases. Also, MCL and MCETL algorithms result in simpler equations to compute the values of connection weights and thresholds than ETL. The number of operations (in terms of $\sum_{i=1}^n w_i x_i$) for MCL or MCETL is lower than those for ETL and IETL. Compared with CSCLA, MCL and MCTEL represent not only vertices with Hamming distance one, but also vertices at Hamming distance larger than one by a single hidden neuron. We illustrate our conclusion by comparing the learning results of MCL, MCETL, ETL, IETL and CSCLA by applying them to some classical examples.

In order to identify larger linearly separable subsets of vertices, we investigate some properties of linear separability for Boolean functions. First we argue some previous hypotheses on linear separability [Park 91a], and then propose our new theorems on linear separability.

In Kim and Park's study, they proposed several hypotheses [Kim 91a] and a theorem, Theorem 1.1 [Kim 95] about linearly separability as follows:

Hypotheses:

1. *A vertex,*
2. *Vertices with a vertex from which all the other vertices are separated by distance one,*
3. *All four vertices on a face,*
4. *All the vertices on any number of connected faces,*
5. *Each set of all the remaining vertices excluding each set of the vertices listed.*

Any subset of vertices which meets any conditions stated above is linear separable.

Theorem 1.1: *A function f is linearly separable if and only if there exists a hypersphere such that all true vertices lie inside or on the hypersphere, and all false vertices lie outside, or vice versa.*

Through further investigation, we proposed our new theorems as follows:

Theorem 1.2: *Given a core vertex, all vertices with Hamming distance one from the core, along with the core vertex can be linearly separated from the remaining vertices by a*

hyperplane.

Theorem 1.3: *Given a core vertex, all vertices with Hamming distance equal to or less than k from the core, along with the core vertex can be linearly separated from the remaining vertices by a hyperplane.*

All of the above hypotheses and theorems constitute the preliminary knowledge for our learning methods and will be discussed in chapter 4.

To balance generalization and precision, we propose a novel Fast Covering Learning Algorithm (FCLA) [Wang 04a] for Boolean neural networks based on multi-level geometrical expansion. Three regions are defined for each hidden neuron based on geometrical concept. The learning process is based on the judgment as to which region the new coming vertex belongs. These three regions have the same center, but different radii. These regions shift and expand in the training process until all vertices of the true subset are covered. To avoid over generalization, modification process is proposed to correct the introduced errors. FCLA possesses the generalization capability with affordable error and need no core vertex before training. Our algorithm is not sensitive to different input sequences (different input sequences result in similar network structures). We illustrate this advantage by examples in chapter 6. Finally FCLA needs less training operations as compared to previous algorithms for BNNs (BLTA, ETL, IETL and CSCLA). FCLA can be used to unsupervised learning as well as supervised learning for Boolean neural networks.

1.7.2 Our Methods for Real-Valued Problems

After covering Boolean problems, we propose learning algorithms dealing with problems with real-valued inputs based on constructive learning and geometrical expansion. When solving problems with Boolean inputs (only zero or one), it is easy to visualize the input space as a unit hypercube. Assume a hypersphere refers to an arbitrary sphere with arbitrary center and arbitrary radius in hyperspace. All input patterns are located on the surface of the exhypersphere (defined in section 2.1.1 page 32, also in [Wang 03b]) of that unit hypercube. Hence the geometrical expansion process of a hypersphere can be expressed by a linear function [Wang 04a].

For problems with real-valued inputs, the patterns (input vectors) are not located on the surface of a hypercube, but inside a hypercube. In this case, many attractive geometrical properties are lost. Hence, the expansion process of a hypersphere cannot be expressed by a linear function.

As an extension of our work for Boolean neural networks, we improve FCLA to solve problems with real-valued inputs based on the three-radii geometrical expansion by using hypersphere activation functions. We call the improved algorithm FCLAR. Each hypersphere can be represented by a hidden neuron. Hence the training process can be visualized by the expansion process of hidden neurons. The details of FCLAR and its applications for real valued problems are explained in chapter 7.

1.7.3 Contribution of Our Research

The contribution from this thesis is to design computationally efficient, fast, and dynamic covering algorithms based on the concept of geometrical expansion. Compared with traditional learning algorithms for neural networks, our novel methods are expected to have the following properties:

- Generalization capability with tolerant error;
- Fast learning speed;
- Constructive structure;
- Good accuracy.

Chapter 2

Geometrical Learning

2.1 Geometrical Analysis for Boolean Neural Networks

2.1.1 Geometrical Properties of Boolean Vectors

It is easy for us to begin our discussion with Boolean neural networks because of the special geometrical properties of Boolean vectors. We first introduce the definition and some preliminaries of Boolean neural networks.

We define a Boolean function

$$g(\mathbf{x}) = g(x_1, x_2, \dots, x_n) = (y_1, y_2, \dots, y_m), \quad (2.1)$$

as a mapping from $\{0,1\}^n$ to $\{0,1\}^m$. $(x_1, x_2, \dots, x_n)$ is the input variable space and $(y_1, y_2, \dots, y_m)$ is the output space. We can represent this Boolean function by a Boolean neural network. A Boolean neural network is defined as a system with Boolean input, Boolean

output and integer connection weights between different layers. We can gain conclusion (through the training process of our algorithm) that the connection weights by our algorithm are all integer values. This kind of neural systems with multiple inputs and multiple outputs is called *MIMO* neural systems. *MIMO* neural systems can be applied to solve multi-classification problems. If $m=1$, a mapping from $(0,1)^n$ to $(0,1)$ is a function:

$$g(\mathbf{x}) = g(x_1, x_2, \dots, x_n) = y. \quad (2.2)$$

This kind of neural systems with multiple inputs and single bit output are called *MISO* neural systems. The *MIMO* neural system can be represented by m *MISO* neural systems just by replication. So we focus our investigation on *MISO* neural systems as follows.

A Boolean function

$$f(\mathbf{x}) = f(x_1, x_2, \dots, x_n), \quad (2.3)$$

is a mapping from $\{0,1\}^n$ to $\{0,1\}$. A set of 2^n binary patterns, each with n bits, can be considered as an n -dimensional unit hypercube (in the input variable space). Each pattern is located on one vertex of the hypercube. Assume that a hypersphere refers to an arbitrary sphere with arbitrary center and arbitrary radius in hyperspace. We consider a minimal hypersphere enclosing all vertices of this hypercube, and define it as “exhyperspere”. All patterns also lie on the surface of this exhyperspere. Kim and Park defined this exhyperspere as the Reference Hypersphere (RHP) [Kim 95] as follows:

$$\sum_{i=1}^n (x_i - \frac{1}{2})^2 = \frac{n}{4}. \quad (2.4)$$

We want to separate a true subset of samples, $\{\mathbf{x} \mid f(\mathbf{x}) = 1\}$, from a false subset of

samples, $\{\mathbf{x} \mid f(\mathbf{x}) = 0\}$, by using one or more linearly separable Boolean functions.

Because all patterns lie on the surface of RHP, a true subset $\{\mathbf{x} \mid f(\mathbf{x}) = 1\}$ or a false subset $\{\mathbf{x} \mid f(\mathbf{x}) = 0\}$ has the following properties [Kim 95], which can also be used as a method to judge if a subset of vertices are linearly separable:

Theorem 2.1: *A function f is linearly separable if and only if there exists a hypersphere such that all true vertices lie inside or on a hypersphere, and all false vertices lie outside, or vice versa.*

Proof [Wang 04a]:

Suppose a set of true vertices within a hypersphere are defined as:

$$\sum_{i=1}^n (x_i - c_i)^2 \leq r^2, \quad (2.5)$$

where x_i is the i th bit of the input, c_i is the i th bit of the center of this separating hypersphere (SHP), and r is the radius of SHP.

SHP must intersect with RHP. If we represent an n -dimensional hypersphere as a circle and an $(n-1)$ -dimensional hyperplane as a line, we can show the intersection of SHP and RHP in Fig.2.1.

Subtracting (2.4) from (2.5): we obtain:

$$\sum_{i=1}^n (1 - 2c_i)x_i \leq r^2 - \sum_{i=1}^n c_i^2, \quad (2.6)$$

that is,
$$\sum_{i=1}^n (2c_i - 1)x_i \geq \sum_{i=1}^n c_i^2 - r^2. \quad (2.7)$$

Fig. 2.1 Intersection of SHP and RHP

Using (2.7) we define the connection weights w_i^h and threshold θ as follows:

$$w_i^h = 2c_i - 1. \quad (2.8)$$

$$\theta = \sum_{i=1}^n c_i^2 - r^2. \quad (2.9)$$

Thus all true vertices lie on one side of a hyperplane, and all false vertices lie on the other side.

Q.E.D.

Theorem 2.1 converts the criteria of linear separability from separating hyperplanes to separating hyperspheres.

Assume that we want to separate the true subset from the false subset by k n -dimensional hyperspheres. For achieving this separation, we need k $(n-1)$ -dimensional hyperplanes. For a neural network which makes use of this result, we observe that the number of separating hyperplanes (hyperspheres) corresponds exactly to the number of hidden neurons.

2.1.2 Linear Separability of Boolean Vectors Based on Geometrical Concept

One most important concept we apply in our investigation is linear separability. From Theorem 2.1, we conclude that a subset of vertices in a hypersphere can be represented by a hidden neuron with a linear activation function. This subset of vertices is linearly separable. Hence a key problem is to obtain an algorithm for finding the maximum linearly separable subsets. A Boolean function is defined as linearly separable, if and only if the true subset can be separated from the false subset by one hyperplane (hypersphere), which can be represented by a single hidden neuron. Otherwise this Boolean function is linearly inseparable. It has been shown that a linearly inseparable function can be decomposed into a series of linearly separable functions [Kim 95]. These linearly separable functions can be represented by a series of hidden neurons.

Now let us assume that we have a linearly separable set of vertices $X = \{\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^{C_0}\}$, where for each $\mathbf{x}^k$, $\mathbf{x}^k = (x_1^k, x_2^k, \dots, x_n^k)$ and $k=1 \dots C_0$. We now illustrate how to gain the linear representation (separating hyperplane) with an example.

Example 2.1: Consider a Boolean function, $f(x_1, x_2, x_3)$ with three variables, having (0,0,1) and (0,1,1) as true vertices. These two true vertices are linearly separable from the remaining ones. In the following, we give the method for obtaining a hyperplane to separate these two vertices {001, 011} from the remaining ones.

Let:

$$C_0 = \text{number of true vertices to be separated from the remaining ones.} \quad (2.10)$$

To obtain the separating hyperplane for each input bit (variable) x_i in f we define:

$$C_i = \sum_{k=1}^{C_0} x_i^k . \quad (2.11)$$

For our example with $\{001,011\}$ as the true subset, we have $C_0=2$, $C_1=0$, $C_2=1$, $C_3=2$.

After having obtained these C_i 's, we obtain w_i 's according to the following:

$$w_i = 2C_i - C_0 \quad \text{where } i=1 \dots n. \quad (2.12)$$

Thus, for our example with $\{001,011\}$, we have, $w_1=-2$, $w_2=0$, $w_3=2$.

Next, for the subset of true vertices, we define the threshold:

$$T = \min_{x_1, x_2} \left(\sum_{i=1}^n w_i x_i \right) . \quad (2.13)$$

Thus, for our example with $X=\{001,011\}$, we have: $T=\min\{2,2\}=2$.

Now in terms of the parameters discussed above, we generalize a theorem to obtain the separating hyperplane.

Theorem 2.2: Let $X = \{\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^{C_0}\}$, be a linearly separable set of vertices. Then the following hyperplane (2.14) is a separating hyperplane

$$\sum_{i=1}^n w_i x_i = T , \quad (2.14)$$

where the parameters w_i 's are obtained by (2.12) and T is obtained by (2.13).

Proof [Wang 04a]:

Our proof is based on Kim and Park's approach [Kim 95]. Let us consider the RHP:

$$(x_1 - \frac{1}{2})^2 + (x_2 - \frac{1}{2})^2 + \dots + (x_n - \frac{1}{2})^2 = \frac{n}{4}. \quad (2.15)$$

All 2^n vertices lie on this Reference Hypersphere (RHP). Let us now consider a subset of these 2^n vertices, which are linearly separable. They lie on some Separating Hypersphere (SHP) with $(c_1, c_2, \dots, c_n)$ as a center and r as a radius:

$$\sum_{i=1}^n (x_i - c_i)^2 = r^2. \quad (2.16)$$

In this hypersphere, we obtain the parameters c_i as given in (2.17):

$$c_i = \sum_{k=1}^{C_0} x_i^k / C_0. \quad (2.17)$$

We obtain the intersection of SHP and RHP from (2.15) and (2.16) as follows:

$$r^2 = \sum_{i=1}^n (1 - 2c_i)x_i + \sum_{i=1}^n c_i^2. \quad (2.18)$$

Then all vertices in (or on) SHP lie on one side of or on the hyperplane (2.18):

$$\sum_{i=1}^n (1 - 2c_i)x_i \leq r^2 - \sum_{i=1}^n c_i^2. \quad (2.19)$$

All the remaining vertices (vertices not in SHP, but on RHP) lie on the other side of the hyperplane (2.18):

$$\sum_{i=1}^n (1 - 2c_i)x_i > r^2 - \sum_{i=1}^n c_i^2. \quad (2.20)$$

So the intersection (2.18) of SHP and RHP is the separating hyperplane, which separates the true subset from the remaining ones.

Rewriting (2.18), we have:

$$\sum_{i=1}^n (2c_i - 1)x_i = \sum_{i=1}^n c_i^2 - r^2. \quad (2.21)$$

In (2.18), r is defined as the minimal value that meets the condition that all of these C_0 vertices are exactly in or on the hypersphere. So it is reasonable to define r as the largest Euclidean distance between the center of the hypersphere and all of these C_0 vertices in that linearly separable subset:

$$r^2 = \max_{k=1}^v \left(\sum_{i=1}^n (x_i^k - c_i)^2 \right). \quad (2.22)$$

Multiply (2.18) by C_0 , we obtain:

$$\sum_{i=1}^n C_0 (2c_i - 1)x_i = C_0 \left(\sum_{i=1}^n c_i^2 - r^2 \right). \quad (2.23)$$

In (2.23), we take $C_0(2c_i - 1)$ as the connection weight, and $C_0 \left(\sum_{i=1}^n c_i^2 - r^2 \right)$ as the threshold:

$$w_i = C_0(2c_i - 1). \quad (2.24)$$

$$T = C_0 \left(\sum_{i=1}^n c_i^2 - r^2 \right). \quad (2.25)$$

Substitute (2.10), (2.17) and (2.22) into (2.24) and (2.25), we obtain (2.12), and (2.13)

in the following step:

$$w_i = C_0(2c_i - 1) = 2C_0c_i - C_0 = 2\sum_{k=1}^{C_0} x_i^k - C_0 = 2C_i - C_0. \quad (2.26)$$

$$\begin{aligned} T &= C_0 \left(\sum_{i=1}^n c_i^2 - r^2 \right) = C_0 \sum_{i=1}^n c_i^2 - C_0 \max_{k=1}^{C_0} \left(\sum_{i=1}^n (x_i^k - c_i)^2 \right) \\ &= \min_{k=1}^{C_0} \left(\sum_{i=1}^n (2C_0c_i x_i^k - C_0(x_i^k)^2) \right). \end{aligned} \quad (2.27)$$

Since $x_i^k \in \{0, 1\}$, so $(x_i^k)^2 = x_i^k$. Hence:

$$T = \min_{k=1}^{C_0} \left(\sum_{i=1}^n C_0(2c_i - 1)x_i^k \right) = \min_{k=1}^{C_0} \left(\sum_{i=1}^n w_i x_i^k \right). \quad (2.28)$$

Q.E.D.

Equations (2.10) to (2.13) are key formulas to compute parameters of hidden neurons in Boolean neural networks and will be applied throughout our proposed learning algorithms for Boolean neural networks.

2.1.3 Linearly Separable Subsets of Boolean Vectors

Kim and Park also proposed some hypotheses about linear separability of n -dimensional Boolean vectors [Park 91a]. They are stated in chapter 1 (page 27). Since Kim and Park's work did not include the proof part, we formally prove the above hypothesis in chapter 4. Following these arguments, we propose two additional theorems to obtain linearly separable subset of vertices. They are stated in chapter 1 (page 27) as Theorem 1.2 and Theorem 1.3 .

All the augments of the above theorems and hypotheses form preliminary knowledge for our proposed algorithms for Boolean neural networks, and are discussed in detail in chapter 4.

2.2 Geometrical Analysis for Neural Networks in Real-Valued Inputs

We now briefly state the problems faced by constructive neural networks when the inputs are real values. Before investigating the geometrical properties of the input hyperspace, the real-valued input should be normalized. We normalize the input field $\mathbf{R}^n$ into a unit cube $[0, 1]^n$ by using equation (2.29):

$$x_i^{normalized} = \frac{x_i - x_i^{\min}}{x_i^{\max} - x_i^{\min}}, \quad (2.29)$$

where n is the dimension of the inputs (the number of the input features), x_i and $x_i^{normalized}$ are the i th feature of the input before and after normalization. $x_i^{\min}$ and $x_i^{\max}$ are the minimal and the maximal value of the i th feature of the input before normalization. $x_i^{\min} - x_i^{\max}$ is the diameter of normalization.

In our subsequent discussion of this thesis, all $\mathbf{x}$'s refer to the values after normalization. After normalization, any input vector $\mathbf{x}^k \in [0,1]^n$. Then, all inputs are located not only on the surface of a unit hypercube, but also within this unit hypercube. If we represent the n -dimensional hypercube as a unit square and the separating hypersphere as a circle, we can visualize the input space, the separating hypersphere (which includes a subset of samples within it), and its supposed separating hyperplane (the intersection of the separating hypersphere and the unit hypercube) in Fig. 2.2.

From Fig. 2.2, we observe, in real-valued cases, the input patterns are not located on the surface of a unit hypercube, but within a unit hypercube after normalization. Thus Theorem 2.1 does not hold any more. The intersection of the separating hypersphere and the unit hypercube cannot separate the subset within the separating hypersphere from the remaining samples. So a subset of samples within a hypersphere cannot be represented by a hidden neuron with a linear activation function.

Fig. 2.2 Visualization of real-valued input space after data normalization

In our research, we apply non-linear activation functions for the hidden neurons, which are hypersphere functions based on the Euclidean distance. And we use each hidden neuron to represent some local area of the input space in our study (refer to the detailed discussion in chapter 7). In each training step, our algorithm updates the parameters of one hidden neuron, which is the nearest to the currently trained sample in terms of geometrical concept, or has the highest relationship with the current trained sample.

2.3 Our Methods Based on Geometrical Concept for Neural Networks

The learning rule of our proposed methods (described in chapters 5, 6 and 7) is based on the geometrical expansion process. In the training process, the training data are input to the learning system sequentially. For a current trained sample, we first evaluate its geometrical location, and then decide the operation we should take. Our methods learn a sample immediately or back it up to be learned in the next training circle according to its geometrical location in the input space. To learn a sample, we either add a new hidden neuron to represent it or update the parameters of some hidden neuron by expanding its corresponding hypersphere according to the input's geometrical location. The details of parameter updating are discussed in chapters 6 and 7.

FCLA learns one object a time. Thus, the training time increases linearly corresponding to the actual number of training objects. FCLA always constructs a two-layered neural network for unsupervised learning and a three-layered neural network for supervised learning. The network does not increase in terms of neural layers when the work scales up. However, the number of hidden neurons increases depending on the datasets. In a large dataset in terms of dimensionality, many datamining methods are available to decrease the input dimensionality. A data preprocessing is necessary before training.

In the training process based on geometrical expansion, the training sample is removed after it is learned. Each sample causes parameter updating only once. By doing this, our method avoids iterative computations in traditional training algorithms for neural

networks. Hence, our proposed learning algorithms are much faster than traditional learning algorithms based on iterative computations.

Our methods apply localization functions. In each training step, our algorithms try to update only the parameters of one hidden neuron, the nearest one to the training sample. Parameters of other hidden neurons remain unchanged. These localization functions simplify the training process.

During the training process based on geometrical concept, the decision whether neurons fire or not is based on the similarity between data (the geometrical distance between the current trained sample and the existing hidden neurons). Actually, the learning process of human beings is also based on the similarities between objects in the real world. The similarities are usually defined as a function of distance.

Chapter 3

Review of Learning Algorithms for Boolean Neural Networks

3.1 Overview of Boolean Neural Networks

Boolean Neural Networks (BNNs) have been used in many fields such as data mining, classification and recognition. In recent years, the rapid developments of VLSI have made it easy to realize special BNNs on neural chips [Ghosh 94, Boahen 93, Zaghloul 04a, Zaghloul 04b]. Biology investigators prefer BNNs because there is a close analogy between a biological neuron and the processing element of a Boolean neuron. Many theoretical results concerning BNNs make it possible to develop simple and efficient learning algorithms for BNNs [Forcada 01, Mertens 97]. Innovative methods for BNN construction have been a hot topic of research in the last decade.

Deolalikar mapped Boolean functions to BNNs with zero threshold and binary $\{1, -1\}$

weights [Deolalikar 01]. He substituted a pair of input and output, say $\mathbf{x}$ and $\mathbf{y}$, as a new single normalized variable, say $\mathbf{z}_{\mathbf{x}\mathbf{y}}$, which converted multiple-class problems to two-class problems. Mertens and Engel [Mertens 97] investigated the Vapnik-Chervonenkis (VC) dimension of neural networks with binary weights and obtained the lower bounds for large systems through theoretical argument.

Kim and Roche [Kim 98] discussed and answered two mathematical questions for BNNs:

- (i) there exists a ρ ($0 < \rho < 1$), such that for all sufficiently large value n (n refers to the input dimension, as used in (2.2)) there is a BNN with n hidden neurons which can separate ρn random patterns with probability close to 1;
- (ii) it is impossible for a BNN of n hidden neurons to separate $(1 - o(1))n$ random patterns with probability greater than some positive constant.

These theoretical results have led to the development of simple and efficient BNN learning algorithms. Many excellent learning algorithms for BNNs have been proposed since the 1990's. Gray and Michel devised Boolean-Like Training Algorithm (BLTA) for construction of BNNs in 1992 [Gray 92]. BLTA does well in memorization and generalization, but many hidden neurons are needed. Kim and Park proposed Expand-and-Truncate Learning (ETL) algorithm in 1995 [Kim 95]. They defined the Set of Included True Vertices (SITV) as a set of true vertices, which can be separated from the remaining vertices by a hyperplane. The status of "true" and "false" vertices is converted, if SITV cannot be expanded further. Yamamoto and Saito improved ETL (called IETL) by modifying some vertices in SITV as "don't care" [Yamamoto 97]. Fewer neurons are

needed in IETL. ETL and IETL begin with selecting a true vertex as the core vertex for SITV. In both of these methods, the number of hidden neurons depends on the choice of the core vertex and the order to examine the status of vertices. Different choice of core vertex and different input order cause different structure of neural networks. In addition, ETL and IETL need to search all training samples which are not learned by the existing hidden neurons for determining each neuron in the hidden layer. If h hidden neurons are needed for n -dimensional inputs, the number of operations needed in terms of $\sum_{i=1}^n w_i x_i$ is: $O(h2^n)$.

Ma Xiaomin introduced the idea of weighted Hamming distance hypersphere [Ma 99], which improved the representation ability of each hidden neuron, hence improved the learning ability of BNNs. In his later research in 2001, based on the idea of weighted Hamming distance hypersphere, he proposed Constructive Set Covering Learning Algorithm (CSCLA) [Ma 01]. CSCLA needs an ancillary Neural Network. Hence it results in double work for the training space. Also in his work, he only considered including vertices with Hamming distance one from the core, not including vertices with Hamming distance more than one in a hidden neuron. So Xiaomin's neural networks need more hidden neurons. Sung proposed an optional synthesis method for BNNs using Newly Expand-and-Truncate Learning Algorithm (NETLA) to minimize not only the number of connections but also the number of neurons in the hidden layer [Sung 02].

Steinbach and Kohut discussed how to transfer linearly inseparable mapping to linearly separable mapping by expanding the input dimension [Steinbach 02]. Otherwise a linearly

inseparable function can only be represented by a nonlinear hidden neuron(s). A nonlinear hidden neuron may have greater representation ability than a linear hidden neuron, but the computation is more complex. Starzyk and Pang proposed evolvable BNNs for data classification [Starzyk 00]. They introduced an evolutionary idea to BNN learning algorithm by generating new features (combination of the input bits), and then selecting some features which make more contribution to linear separability. Chaudhari and Tiwari [Chaudhari 02] proposed combination of BLTA and ETL for adapting BNNs to handle multiple classes, as needed for many classification problems.

In section 3.2 to 3.7, we review briefly the main concepts of some of these earlier approaches. Specially we discuss:

- (i) Boolean-Like Training Algorithm (BLTA) [Gray 92],
- (ii) Expand-and-Truncate Learning (ETL) [Kim 95],
- (iii) Improved Expand-and-Truncate (IETL) [Yamamoto 97],
- (iv) Newly Expand-and-Truncate Learning Algorithm (NETLA) [Sung 02],
- (v) Constructive Set Covering Learning Algorithm (CSCLA) [Ma 01],
- (vi) Nearest-To-an-Exemplar (NTE) classifier and Boolean K -Nearest Neighbor (BKNN) classifier [Gazula 95].

3.2 Boolean-Like Training Algorithm (BLTA) [Gray 92]

Gray and Michel introduced Boolean-Like Training Algorithm (BLTA) in 1992. BLTA generates a four-layered feed-forward neural network architecture, which includes an

input layer (which receives input data and forward the data to the next layer), a hidden layer (which is the main part to learn logical operations between the input and inhibition layers), an inhibition layer (to correct the wrongly represented data by the hidden layer) and an output layer (which produces the output value of the network). The neural network structure for BLTA is shown in Fig. 3.1.

BLTA derives its original principles from Boolean algebra with extensions. Because BLTA does not depend on gradient descent techniques, the training speed of BLTA is increased. In addition, BLTA guarantees convergence for any Boolean function. BLTA is a dynamic technique which does not require a complete dataset prior to training.

A Boolean function can be represented by algebraic expression (for example, $F=ABC+AB\bar{C}=AB$), truth table or Karnaugh map [Su 73]. Karnaugh map is expressed by grouping or clustering the nearest neighbors with the same outputs, in groups consisting of powers of 2 (which will be 2-circles, 4-circles, etc.), which is the basis of reduction in Karnaugh domain [Su 73]. Karnaugh map helps to visualize BLTA as shown in Fig. 3.2, 3.3 and 3.4. Here an n -circle means a circle including n vertices.

When training the hidden layer, three modes of training operations are involved:

- (i) the explicit representation (Fig. 3.2)
- (ii) the generalization (Fig. 3.3)
- (iii) the modification (Fig. 3.4)

Fig. 3.1 A neural network structure constructed by BLTA

Explicit representation requires only the exact relations which are desired to be implemented; generalization allows the training of desired relation with the allowance that additional relations could be generated by the algorithm; while modification has the ability to forget or correct old undesired relations.

Fig. 3.2 Explicit representation

Fig. 3.3 Generalization

Fig. 3.4 Modification

BLTA obeys the following seven rules according to these three modes of training

operations to train neural networks.

R1: Grouping of two and only two elements is allowed (2-circles).

R2: No element can appear in more than one grouping.

R3: Any minterm that is not included in a 2-circle shall be represented as a 1-circle.

R4: A new minterm will be incorporated into a 2^M -circle ($M \geq 2$) if and only if an existing 2^{M-1} -circle which can be expanded to encompass the new minterm, otherwise use R5.

R5: Any new minterm that cannot be incorporated via R4 will be explicitly represented by R1, R2 and R3.

R6: During the initial representation of the datasets with generalization, the known minterm (a minterm is defined as a logical expression of n variables consisting of only the logical and operator and complements) of function F shall be represented first by generalization; this operation will be followed by explicit representation training of all known minterms of function $\overline{F}$, as required, to inhibit incorrectly generalized outputs of function F .

R6(a): During the initial representation of the datasets with generalization, the known minterm of function F_{imax} shall be represented first, followed by all minterms of function F_{imax-1} , and so forth, ultimately followed by all known minterms of F_0 .

R7: All known minterms of F and $\overline{F}$, to be later appended to the initially represented network, shall be added by explicit representation to the network if not correctly represented by the existing network.

R7(a): All new minterms of any F_i , to be later appended to the initially

represented network, whether for the purpose of adding new minterms or existing memories, shall be added via explicit representation to the network, if not already correctly represented by the existing network. In the case of modification operations the new explicitly represented minterm shall inhibit all existing incorrect memories.

R1, R2 and R3 are rules for explicit representation. R2 illustrates that any two 2-circles should not overlap. Through R1, R2 and R3 each hidden neuron represents only one or two vertices, and $2^{M/2}$ hidden neurons in the hidden layer are enough to explicit represent any M -input one-output Boolean function. This mode of operation is called explicit representation in BLTA. R4 and R5 are rules for generalization. Under R4, a 2-circle can be expanded into 4-circle, and then 8-circle and so on. By R4 and R5, a vertex is considered being generalized to a hidden neuron first, and if not successful, it will be explicitly represented by R1, R2 and R3. Rule R6 (R6(a)) talks about the order to generate minterms. If the representation of a new minterm causes a generalization which attempts to encompass the minterm F , and therefore creates a conflict between a previously known and currently represented minterm of $\overline{F}$ or F , then the use of generalization causes an “overgeneralization” of the dataset. In this case, R7 (R7a) gives the rule to modify existing memories using inhibition neurons.

The activation function for BLTA for Boolean neural networks is defined as follows.

Given the inputs $\mathbf{x}=(x_1, x_2, \dots, x_n)$:

$$Q_i = \begin{cases} -1 & \text{if } x_i = 0 \\ 1 & \text{if } x_i = 1 \end{cases}, \quad (3.1)$$

$$U = \sum_{i=1}^n w_i Q_i + \theta, \quad (3.2)$$

$$V = G(U) = \begin{cases} 0 & \text{if } U < 0 \\ 1 & \text{if } U \geq 0 \end{cases}, \quad (3.3)$$

where x_i is the i th bit of the input variable before prefiltering, Q_i is the i th bit of the input variable after prefiltering, w_i is the connection weight for x_i . θ is the threshold, and V is the hard-limiter output for that inhibition neuron.

The weights between the input layer and the hidden layer, those between the hidden layer and the inhibition layer, those between the inhibition layer and the output layer, and the thresholds of each neuron are defined based on the seven rules above.

The default connection weights between the input neuron and the hidden neuron are -1 's, which are changed to 1 's from all the input $x_i=1$ ($Q_i=1$) for the i th input neuron. That is $w_{ij}^1 = -1$ initially, $w_{ij}^1 = 1$ if $x_i=1$ ($Q_i=1$) when training the j th hidden neuron. From (3.2), we see for the j th hidden neuron, $w_{ij}^1 x_i = 1$, if $w_{ij}^1 = -1$ and $x_i=0$ ($Q_i=-1$), and $w_{ij}^1 x_i = 1$, if $w_{ij}^1 = 1$ and $x_i=1$ ($Q_i=1$), which is added to the value of U . when U is larger than $-\theta$, V fires (output 1).

BLTA can modify/forget incorrect, undesired, or outdated relations by new relations added to an existing network. The addition of a new relation to an existing network can be accomplished in two different ways via the rules of explicit representation. The process of forgetting begins as explicit representation. The next step is to connect the inhibition layer

neuron to the hidden layer neuron responsible for this new minterm to the output layer. Since a forgotten process is to be implemented, an inhibitory connection will be utilized to suppress the incorrect output. To train the inhibition layer, they defined the connection weights for the inhibition neuron as $w_{ij}^2=1$, and defined the connection weights for the inhibition neuron as $w_{ij}^2=-1$. Hence the over-generalizations caused by the beginning data are corrected by the subsequent data.

BLTA can be utilized in numerous modes of operation by selecting various features via inclusion or exclusion abilities and through the selection of parameters such as the maximal M -circle (defined on page 49) parameter. It is easy to show that BLTA can represent any Boolean function F . BLTA does well in generalization and memorization, but BLTA needs more hidden neurons as compared to the ETL method of Kim and Park, which we discuss in section 3.3.

3.3 Expand-and-Truncate Learning (ETL) Algorithms [Kim 95]

Kim and Park proposed Expand-and-Truncate Learning (ETL) algorithm based on geometrical concept in 1995. ETL finds a set of required separating hyperplanes and determines the connection weights and thresholds based on geometrical analysis of the given training set. Compared with EBP algorithm (page 28), ETL is faster. In ETL only integral connection weights and thresholds are used, which greatly facilitates its hardware implication. ETL guarantees convergence. And ETL automatically determines a required number of neurons in the hidden layer.

ETL constructs a three-layered neural network with one input layer, one hidden layer and one output layer, as shown in Fig. 3.5.

Fig. 3.5 A neural network structure constructed by ETL

A set of 2^n binary patterns each with n bits can be considered as an n -dimensional hypercube. Assume that those patterns can be separated into two sets (true and false) by k $(n-1)$ -dimensional hyperplanes. Kim and Park's ETL algorithm achieves this separation hyperplanes as follows. They first defined a set of true vertices, which can be separated from the remaining vertices by a hyperplane, as a Set of Included True Vertices (SITV). Using this concept, in their method, they expanded SITV to include more vertices with same output tag (true or false), and if SITV cannot be expanded further, the status of "true" vertices and "false" vertices were changed. Thus vertices between two consecutive hyperplanes have the same desired outputs. Two consecutive groups separated by a hyperplane have different desired outputs. So these k $(n-1)$ -dimensional hyperplanes partition all vertices into $k+1$ groups. Vertices in each group have the same desired output (either 0 or 1).

ETL algorithm begins by selecting one core vertex; the vertices which are not included in SITV are examined one by one; then SITV includes as many vertices as it can; if no more vertices can be added to SITV, the first separating hyperplane is found. However, if this hyperplane does not separate all true vertices from all false vertices, a second separating hyperplane is to be found. To obtain the second hyperplane, false vertices are converted to true vertices, and true vertices which are not in SITV are converted to false vertices, and the second separating hyperplane is obtained. This process goes on until all true vertices are separated from all false vertices.

If we visualize Reference Hypersphere (RHP: defined on page 32) [Kim 95] as a circle, we illustrate the training process of ETL using analogy shown in Fig. 3.6.

Fig. 3.6 Training process of ETL

White regions stand for true subsets and black regions stand for false subsets. The number in each region stands for the order of generating hyperplanes (hidden neurons). Based on the selected true core vertex (in region 1), ETL begins to extend SITV to cover as many

true vertices as possible. When SITV covers region 1 (reaches the boundary of region 1 and region 2), it meets a false vertex, which prevents SITV from further expansion. Then false vertices out of region 1 are converted to true vertices and true vertices out of region 1 are converted to false vertices. Hence the false vertex which blocks the hypersphere expansion will not block it any more. The SITV then expands to include false vertices in region 2 until it reaches region 3. This process goes on until it covers all true vertices or all false vertices.

If only one $(n-1)$ -dimensional hyperplane (one hidden neuron) is needed, this function is linearly separable. Otherwise, the linearly inseparable function should be decomposed into multiple linearly separable functions. It is easy to show that any binary-to-binary mapping can be decomposed into a series of linearly separable functions. ETL decomposes an arbitrary linearly inseparable function into multiple linearly separable functions as,

$$y(x_1, x_2, \dots, x_n) = x_1 \theta(x_2 \theta(\dots (x_{n-1} \theta x_n) \dots)) , \quad (3.4)$$

where x_i is the i th bit of the input and operator θ is either logical *AND* or logical *OR*.

Each linearly separable function is realized by a hidden neuron. The number of hidden neurons needed by ETL equals to the number of separating hyperplanes.

The j th hyperplane can be expressed by a hidden neuron:

$$\sum_{i=1}^n w_{ij}^h x_i - T_j = 0 , \quad (3.5)$$

where x_i is the i th bit of the input, w_{ij}^h is the connection weight of the j th hidden neuron

from x_i , and T_j is the threshold for the j th hidden neuron.

The hard-limiter activation function for the j th hidden neuron can be expressed by

$$y_j = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_{ij}^h x_i - T_j \geq 0 \\ 0 & \text{otherwise} \end{cases}, \quad (3.6)$$

where y_j is the output of the j th hidden neuron.

According to different cases, three training methods to compute w_{ij}^h and T_j are proposed by Kim and Park:

Method a:

$$w_{ij}^h = 1, \text{ if } f(x_i)=1 \text{ and } x_c^i = 1, \quad (3.7)$$

$$w_{ij}^h = -1, \text{ if } f(x_i)=1 \text{ and } x_c^i = 0, \quad (3.8)$$

$$w_{ij}^h = 2, \text{ if } f(x_i)=0 \text{ and } x_c^i = 1, \quad (3.9)$$

$$w_{ij}^h = -2, \text{ if } f(x_i)=0 \text{ and } x_c^i = 0, \quad (3.10)$$

$$T_j = \sum_{i=1}^n w_{ij}^h x_i - 1, \quad (3.11)$$

where x_c^i is the i th bit of the core, x_i is the i th bit of the input, w_{ij}^h is the connection weight of the j th hidden neuron from x_i , and T_j is the threshold for the j th hidden neuron.

Method b:

$$w_{ij}^h = 1, \text{ if } v_c^i = 1; \quad (3.12)$$

$$w_{ij}^h = -1, \text{ if } v_c^i = 0; \quad (3.13)$$

$$T_j = \sum_{i=1}^n w_{ij}^h v_c^i - (d-1) \quad (3.14)$$

where v_c^i is the i th bit of the core, x_i is the i th bit of the input, w_{ij}^h is the connection weight of the j th hidden neuron from x_i , and T_j is the threshold for the j th hidden neuron.

Method c:

$$w_{ij} = 2C_i - C_0, \quad (3.15)$$

$$C_i = \sum_{k=1}^{C_0} x_k^i, \quad (3.16)$$

where C_0 is the number of vertices represented by the hidden neuron, x_k^i is the i th bit of the k th sample, and w_{ij}^h is connection weight of the j th hidden neuron from x_i .

Then compute $f_{\max}$ and $t_{\min}$ by,

$$f_{\max} = \max_{f(\mathbf{x})=0} \left(\sum_{i=1}^n w_i x_i \right), \quad (3.17)$$

$$t_{\min} = \min_{f(\mathbf{x})=1} \left(\sum_{i=1}^n w_i x_i \right). \quad (3.18)$$

If $f_{\max} < t_{\min}$, then these two subsets can be linearly separated, and

$$T_j = \left\lceil \frac{t_{\min} + f_{\max}}{2} \right\rceil; \text{ otherwise if } f_{\max} > t_{\min}, \text{ these two subsets cannot be linearly}$$

separated, where T_j is the threshold for the hidden neuron.

The representation ability of a single hidden neuron is limited because the connection weights are restricted to 1, -1, 2, and -2 in *method a*, and 1 and -1 in *method b*. Comparatively, *method c* is more general. In Kim's work, he applies *method c* for

investigation. Also many investigations by other researchers are based on *method c*.

After all the required hyperplanes (hidden neurons) are found, one output neuron is needed to combine the outputs from hidden neurons. A hidden neuron is defined as a converted hidden neuron, if the neuron was determined based on converted true vertices which are originally given as false vertices and converted false vertices which are originally given as true vertices. Every even-number hidden neuron obtained by the above method is a converted hidden neuron. So formulas to determine the connection weights and thresholds for the output layer are defined as follows:

$$w_j^o \begin{cases} 1 & \text{if } j \text{ is odd} \\ -1 & \text{if } j \text{ is even} \end{cases}, \quad (3.19)$$

$$T^o = 1, \quad (3.20)$$

where w_j^o is the connection weight from the j th hidden neuron to the output neuron, and T^o is the threshold for the output neuron.

ETL algorithm has been used in many applications, such as handwritten digit recognition [Park 91b, 93] based on simulated light sensitive model, showing a remarkable result in both training speed and the number of hidden neurons. Also ETL has been used in pattern classification of breast cancer [Kim 92, 94] gaining satisfactory results.

3.4 Improved Expand-and-Truncate Learning (IETL) Algorithm [Yamamoto 97]

Yamamoto and Saito improved ETL by modifying some vertices in SITV as “don’t care”, and they called their method Improved ETL (IETL). The vertices in SITV are overlooked

when judging whether a vertex can be added to SITV. In ETL, after expanding the new hyperplane must include all vertices separated by previous hyperplanes. However, in IETL, the new hyperplane after its expansion can only include the part of vertices separated by previous hyperplanes. IETL improves ETL by using less hidden neurons to solve the same problems.

Here, we also visualize RHP as a circle. Fig. 3.7 shows the training process of IETL. IETL undergoes a training process similar to ETL (Fig. 3.6), however, the difference is: the vertices in SITV are considered as “don’t care”. Hence one region can overlap another region. In Fig. 3.7, region 2 overlaps region 1, 3, 4 and 5. The output decision is made by the region labeled by a smaller number.

Fig. 3.7 Training process of IETL

Similar to ETL, IETL begins with selecting a true vertex as the core vertex of SITV. SITV is expended by adding more true vertices until no vertex can be added any more. The reason why a hyperplane can not expand is due to the existence of false vertices around the SITV hypersphere. These false vertices block the expansion of the SITV hypersphere.

The false vertices are converted to true vertices, and true vertices, which are not in SITV, are converted to false vertices. IETL continues this process until all true vertices are separated from all false vertices. The number of hidden neurons needed by ETL/IETL depends on the selected core and the order to examine vertices to be included in SITV. Different choices of core vertex and different orders (permutations) of input values result in different structures of neural networks. IETL does not give guidelines about which part of vertices should be considered as “don’t care”. In addition, in IETL, vertices are considered in groups, not one by one. This makes programming of IETL difficult.

3.5 Newly Expand-and-Truncate Learning Algorithm (NETLA) [Sung 02]

Based on the ETL algorithm, Sung described an optimal synthesis method: Newly Expand-and-Truncate Learning Algorithm (NETLA) for BNNs. This synthesis method uses expanded sum of products of Boolean expressions. NETLA tries to reduce not only the number of hidden neurons but also the number of connections between the input neurons and the hidden neurons by involving both original terms and complement terms. NETLA also constructs a three-layered neural network similar to that constructed by ETL. The advantage of NETLA is that it can synthesize data samples regardless of the input order for BNNs. Given any Boolean expression, NETLA can represent it by its optimal synthesis method. However, if we only know the truth table, not the Boolean expression, NETLA does not work well because NETLA is based on expression reduction. In real world applications, we only know the truth table, not the Boolean expression in most cases. So NETLA needs to reduce the Boolean expression first. However, it is difficult to

let a machine reduce a Boolean expression to its minimal form automatically. Furthermore NETLA introduces complement terms. Each input bit is represented by both its original form and complement form. So a double number of input neurons are needed.

3.6 Constructive Set Covering Learning Algorithm (CSCLA) [Ma 01]

Ma Xiaomin introduced the idea of weighted Hamming distance hypersphere [Ma 99] which improved the representation ability of each hidden neuron, hence improved the learning ability of BNNs.

Ma Xiaomin defined Hamming distance and weighted sum Hamming distance between an arbitrary vertex $\mathbf{x}$ and a core vertex $\mathbf{x}^c$ as $d_H(\mathbf{x}^c, \mathbf{x})$ and $d_W(\mathbf{x}^c, \mathbf{x})$ by (3.21) and (3.22):

$$d_H(\mathbf{x}^c, \mathbf{x}) = \sum_{i=1}^n x_i^c \oplus x_i, \quad (3.21)$$

$$d_W(\mathbf{x}^c, \mathbf{x}) = \sum_{i=1}^n g_i (x_i^c \oplus x_i), \quad (3.22)$$

where x_i is the i th bit of the input variable, x_i^c is the i th bit of the core vertex, and g_i is the weight coefficient for the i th bit of the input.

Hence a weighted Hamming distance hypersphere can be defined as (3.23):

$$R(d_W) = \{ \mathbf{X} = (x_1, x_2, \dots, x_n) \in F_2^n \mid d_W(\mathbf{X}^c, \mathbf{X}) \leq d_W \} \quad (3.23)$$

where d_W is the radius of this weighted Hamming distance hypersphere.

Based on the definition above, given different values of g_i , some special hypersphere or

hyperplane is obtained:

- a) $g_1 = g_2 = g_3 = \dots = g_n = 1; d_w \neq 0$; a Hamming hypersphere;
- b) $g_1 = g_2 = g_3 = \dots = g_n = 1; d_w = 0$; only one point included;
- c) $g_1 = g_2 = g_3 = \dots = g_n = 1; g_{r+1} = g_{r+2} = g_{r+3} = \dots = g_n = 0; d_w = 0$; a hypercube;
- d) $g_1 = g_2 = g_3 = \dots = g_r = 1; g_{r+1} = g_{r+2} = g_{r+3} = \dots = g_n = 0; d_w = 1$; an $n-r$ dimensional hamming hypersphere.

$\bigcup_{i=1}^k A_i$ is the collection of the weighted Hamming hyperspheres, where A_i is a weighted Hamming hypersphere including true vertices, and i is its index. $\bigcup_{j=k+1}^L B_j$ is the collection of the weighted Hamming hyperspheres, where B_j is a weighted Hamming hypersphere including false vertices, and j is its index. The A_i or B_j represents a hidden neuron. Hence a Boolean function $\bigcup_{i=1}^k A_i \setminus \bigcup_{j=k+1}^L B_j$ can be represented correctly by a three-layer neural network with one input layer, one hidden layer, and one output layer. Each hidden neuron represents a subset of vertices with the same desired output (A_i or B_j).

Based on the idea of weighted Hamming distance hypersphere, Ma Xiaomin proposed Constructive Set Covering Learning Algorithm (CSCLA) as illustrated in Fig. 3.8. In the training process, CSCLA needs two neural networks: NN_1 which stands for the resultant network and NN_2 which is an ancillary interim network.

CSCLA has been used to generate the feedback function in nonlinear shift register synthesis. CSCLA is simple, reliable and has a small burden of computation. In CSCLA, only true or false subset is considered. Because CSCLA need not consider each

input-output pair when generating each hidden neuron, it is faster than ETL and IETL. CSCLA only examines every input-output pair once, and fits for on-line learning. Similar to both ETL and IETL algorithms, the convergence of CSCLA is completely guaranteed.

- 1) If $x \in A$, then set $w_{1i1}=2(2x_i-1)$; $T_{11}=2n-3$; $w_{2i1}=2x_i-1$; $T_{21}=n-3$; $m=1$; $q=1$.
- 2) An arbitrary $x \in A$, if at least one hidden neuron in NN_1 (the resultant network) fires, go to 5, else go to 3.
- 3) If at least one hidden neuron in NN_2 fires, $w_{1ij}=w_{1ij}$ if $w_{2ij}=2x_i-1$, $w_{1ij}=w_{2ij}$ if $-w_{2ij}=2x_i-1$. $T_{1j}=T_{1j}-1$, go to 5; else go to 4.
- 4) $m=m+1$, add a new hidden to NN_1 and NN_2 (an ancillary interim network) respectively. $w_{1im}=2(2x_i-1)$; $T_{1m}=2n-3$; $w_{2im}=2x_i-1$; $T_{2m}=n-3$.
- 5) $q=q+1$, if all vertices in A have been covered, go to 6, else go to 2.
- 6) Assign all connection weights between the hidden layer and the output layer as 1, and assign the thresholds of the output neuron as 1.

Fig. 3.8 Training algorithm of CSCLA

However, through the training process we observe that in CSCLA, the connection weights between the input layer and the hidden layer are constricted to 1, -1, 2 -2 which is the same as *method a* in ETL. The reason is that CSCLA only considers including vertices with Hamming distance one from the core, not vertices with Hamming distance more than one. So the representation ability of each hidden neuron is restricted.

3.7 Nearest-To-an-Exemplar (NTE) and Boolean K -Nearest Neighbor (BKNN)

[Gazula 95]

Two supervised learning algorithms (classifiers) based on Boolean neural networks were proposed by Gazula and Kabuka (1995): Nearest-To-an-Exemplar (NTE) classifier and

Boolean K -Nearest Neighbor (BKNN) classifier. These two classifiers use the idea of Radius Of Attraction (ROA) based on geometric concept. Their Boolean neural networks are feedforward neural networks with binary inputs (0,1), binary weights (-1, 1) and integer thresholds. These values simplify the networks' structure and computation process. Further they allow for easy hardware realization.

The connection weights and thresholds are defined as follows by (3.24) and (3.25). For the k th exemplar:

$$w_{ik} = 2a_{ik} - 1; \quad (3.24)$$

$$\theta_k = \sum_{i=1}^n a_{ik} w_{ik} - r_k \quad (3.25)$$

where w_{ik} is the connection weight between the i th input neuron and the hidden neuron standing for the k th exemplar, a_{ik} is the i th bit of the input for the k th exemplar, θ_k is the threshold of the hidden neuron standing for the k th exemplar, and r_k is the ROA of the k th exemplar. Here each exemplar is represented by a hidden neuron.

All patterns that differ from the stored pattern (the k th exemplar here) with a Hamming distance of r_k or less succeed in firing that hidden neuron (the hidden neuron standing for the k th exemplar). The training process of NTE and BKNN is implemented by memorization of the training vectors and generalization is implemented by the ROAs. Let C be the number of classes. The C exemplars stand for C classes. Each class converges in a hypersphere centered at its corresponding exemplar. The NTE classifier classifies a given test pattern $\mathbf{x}$ to class C_i ($i=1, \dots, C$) represented by the exemplar $\mathbf{a}_i$ (and encoded into the class node C_i) if and only if the Hamming distance between $\mathbf{x}$ and $\mathbf{a}_i$ is

less than or equal to ROA of the class node C_i .

NTE classifier can be considered as the 1-nearest neighbor algorithm. It can be extended to the k -nearest neighbor algorithm which is called BKNN. Using more exemplars, and duplicating the nodes to a single classifier, the classifier with the most nearest neighbors fires. The training process is similar to that of NTE algorithm (the 1-nearest neighbor algorithm).

3.8 Summary

In this chapter, we have reviewed some excellent learning algorithms designed for BNNs in the last decade. They are BLTA, ETL, IETL, CSCLA and NETLA, all of which belong to constructive learning. A remarkable characteristic of constructive learning algorithms is that they change the topology of a neural network during training. Compared to the algorithms which fix the topology of the neural network structure, and search for parameters in the parameter space, constructive learning algorithms search for the optimal topology as well as the optimal parameters. Kim and Park [Kim 95] have given the analysis of convergence for finite inputs of constructive learning algorithms. Based on their analysis, constructive learning algorithms guarantee convergence for given finite inputs. And constructive learning algorithms for BNNs guarantee convergence [Gray 92].

BLTA does well in memorization and generalization, but also too many hidden neurons may be needed. ETL finds a set of required separating hyperplanes and automatically

determines a required number of neurons in the hidden layer based on geometrical analysis of the training set. During training, the reason why a hyperplane cannot expand is due to the existence of false vertices around the SITV hypersphere. These false vertices block the expansion of the SITV hypersphere. IETL improved ETL by modifying some vertices in SITV as “don’t care”. As a result, IETL ameliorates ETL by using a less number of hidden neurons to solve the same problems. However a different choice of core vertex or different order of the input sequence causes a different neural network topology to be constructed by ETL and IETL. NETLA uses expanded sum of product of Boolean expression. NETLA can synthesize samples regardless of the input order for BNNs. But it is difficult to let a machine reduce a Boolean expression automatically. CSCLA was proposed based on the idea of weighted Hamming distance hypersphere. CSCLA is simpler and faster than ETL and IETL, because each input-output pair passes the learning system only once. While ETL and IETL consider all input-output pairs when training each hidden neuron, CSCLA only considers including vertices with Hamming distance one from the core, not more than one in a hidden neuron. So the representation ability of each hidden neuron is limited. NTE and BKNN are two excellent classifiers given proper exemplars. However the exemplars are unknown to us in most cases when solving real world problems.

Based on the investigation of previous algorithms, in chapter 5, we discuss our Multi-Core Learning (MCL) algorithm and Multi-Core Expand-and-Truncate Learning (MCETL) algorithm to solve the blocking problem in ETL and IETL. MCL and MCETL begin with multiple cores for multiple SITVs. All SITVs expand simultaneously.

MCETL combines MCL with the idea of expand-and-truncate mechanism. MCL and MCETL need fewer hidden neurons and use simpler equations to train BNNs than other learning algorithms. The number of computing operations for MCL and MCETL is lower than that for ETL and IETL.

Among all training algorithms of BLTA, ETL, IETL, CSCLA, and NETLA, only BLTA has generalization capability. In chapter 6, we give our novel Fast Covering Learning Algorithm (FCLA) for BNNs based on multi-level geometrical expansion. FCLA possesses generalization capability and needs no core vertex before training. In addition, FCLA is not sensitive to different input sequences. Finally FCLA needs little training operations. It determines whether to add a vertex to a hidden neuron by its geometrical location, not by computing the weighted sum of every vertex in the training set.

Chapter 4

Linear Separability Analysis for Boolean Neural Networks

4.1 Introduction

A key problem in the learning algorithms for Boolean Neural Networks (BNNs) is how to find proper linearly separable subsets in the training set. Kim investigated what kind of subsets in the Boolean vectors are linearly separable and then proposed hypotheses and theorems on linear separability. But he did not give the proof in his work. To complete Kim's work, we first give proofs for Kim's hypotheses and theorems. Then as an extension of Kim's work, we propose our own theorems about linear separability in Boolean vectors and show their validity.

All these hypotheses and theorems are needed for our proposed learning algorithms for BNNs: Multi-Core Learning (MCL) algorithm, Multi-Core Expand-and-Truncate

Learning (MCETL) algorithm and Fast Covering Learning Algorithm (FCLA), which are discussed in chapters 5 and 6 respectively.

4.2 Hypotheses about Linear Separability

Kim and Park proposed some hypotheses [Park 91a] about linear separability of n -dimensional Boolean vectors. They are stated in the following.

Hypotheses: The following set of vertices can be linearly separated from the rest:

1. *A vertex,*
2. *Vertices with a vertex from which all the other vertices are separated by distance one,*
3. *All four vertices on a face,*
4. *All the vertices on any number of connected faces,*
5. *Each set of all the remaining vertices excluding each set of the vertices listed.*

Since Kim and Park's work does not include the proof part, we proceed to prove the above Hypotheses [Wang 03b].

Proof (of hypothesis 1): Given one vertex x to be separated, according to (2.10)-(2.11)

we can construct the hidden neuron as:

$$w_i = \begin{cases} 1 & \text{if } x_i = 1 \\ -1 & \text{if } x_i = 0 \end{cases} . \quad (4.1)$$

Suppose there are n_1 true bits (bits having value one) in x . Then:

$$T = n_1. \quad (4.2)$$

For $\mathbf{x}$:

$$\sum_{i=1}^n w_i x_i \geq T \quad (\text{where } \sum_{i=1}^n w_i x_i = n_1 = T). \quad (4.3)$$

We obtain any other vertex with Hamming distance k from $\mathbf{x}$ by converting k bits from $\mathbf{x}$.

Whenever one bit is converted, one is subtracted from $\sum_{i=1}^n w_i x_i$.

So, for any vertex with Hamming distance k from $\mathbf{x}$:

$$\sum_{i=1}^n w_i x_i = T - k \leq T. \quad (4.4)$$

Q.E.D.

Proof (of hypothesis 2): Suppose we begin with one core vertex in SITV, say $\mathbf{x}^c = \{x_1^c, x_2^c, \dots, x_n^c\}$. Let $\mathbf{X} = \{\mathbf{x}^{j_1}, \mathbf{x}^{j_2}, \dots, \mathbf{x}^{j_r}\}$ be a set of vertices having Hamming distance equal to one from the core, where $\mathbf{x}^{j_i}$ is the vertex whose j_i th bit is different from the core, while other bits are equal to the core. We now show that $\{\mathbf{x}^{j_1}, \mathbf{x}^{j_2}, \dots, \mathbf{x}^{j_r}\}$ and $\mathbf{x}^c$ can be linearly separated from the remaining vertices. Totally $(r+1)$ vertices are included in SITV including the core vertex. These $(r+1)$ vertices can be linearly separated from the remaining vertices by hyperplane (2.14). From (2.10)-(2.13), we obtain:

$$C_0 = (r+1). \quad (4.5)$$

$$w_i = 2C_i - (r+1). \quad (4.6)$$

$$C_i = \sum_{l=1}^{r+1} x_i^l, \quad (4.7)$$

where x_i^l is the i th bit of $\mathbf{x}^{j_l}$.

If $r=1$, only one vertex $\mathbf{x}$ and the core vertex $\mathbf{x}^c$ are to be linearly separated from the remaining vertices. We assume $\mathbf{x}$ and $\mathbf{x}^c$ are different at the i th bit, and do not differ for other bits. From (2.11) and (2.12), we have: for the i th bit, $w_i = 0$.

Let there be n_1 one (true) bits in $\mathbf{x}$. Let j denote an index where the j th bit of $\mathbf{x}$ (which is the same as the j th bit of $\mathbf{x}^c$) is one. Then $w_j=2$. Let there be n_2 zero (false) bits in $\mathbf{x}$. Let the k th bit of $\mathbf{x}$ (which is the same as the k th bit of $\mathbf{x}^c$) be zero. Then $w_k=-2$.

From (2.10)-(2.13), the separating hyperplane is:

$$\sum_{i=1}^n w_i x_i = T = 2n_1. \quad (4.8)$$

We note that, for $\mathbf{x}$ (and $\mathbf{x}^c$),

$$\sum_{i=1}^n w_i x_i \geq 2n_1. \quad (4.9)$$

For other vertices $\mathbf{y}$, we can verify that

$$\sum_{i=1}^n w_i y_i < 2n_1. \quad (4.10)$$

Now suppose $1 < r \leq n$. Consider $(r+1)$ vertices $\mathcal{S} = \{\mathbf{x}^c, \mathbf{x}^{j_1}, \mathbf{x}^{j_2}, \dots, \mathbf{x}^{j_r}\}$. We show that hyperplane (2.14) separates $\mathcal{S}$.

From (2.12) we have:

$$w_i = \begin{cases} r-1, & \text{if } x_i^c = 1, \text{ and } \exists j_l \quad j_l = i, \\ r+1, & \text{if } x_i^c = 1, \text{ and } \forall j_l \quad j_l \neq i, \\ -(r-1), & \text{if } x_i^c = 0, \text{ and } \exists j_l \quad j_l = i, \\ -(r+1), & \text{if } x_i^c = 0, \text{ and } \forall j_l \quad j_l \neq i. \end{cases} \quad (4.11)$$

Let

$$N_1 = \{ \mathbf{x} \text{ in SITV} \mid x_i = 0, x_i^c = 1 \}, \quad |N_1| = n_1.$$

$$N_2 = \{ \mathbf{y} \text{ not in SITV} \mid y_i = 0, x_i^c = 1 \}, \quad |N_2| = n_2.$$

$$N_3 = \{ \mathbf{x} \text{ in SITV} \mid x_i = 1, x_i^c = 0 \}, \quad |N_3| = n_3.$$

$$N_4 = \{ \mathbf{y} \text{ not in SITV} \mid y_i = 1, x_i^c = 0 \}, \quad |N_4| = n_4.$$

We can verify that:

$$n_1 + n_3 = r. \quad (4.12)$$

$$n_2 + n_4 = n - r. \quad (4.13)$$

For core vertex $\mathbf{x}^c$, using the above equations, we have:

$$\sum_{i=1}^n w_i x_i^c = (r-1) \times 1 \times n_1 + (r+1) \times 1 \times n_2. \quad (4.14)$$

A vertex in N_1 (N_1 is included in SITV) is a result of converting one bit of the core vertex

from one to zero; then, $(r-1)$ is subtracted from $\sum_{i=1}^n w_i x_i^c$. So, we have:

$$\sum_{i=1}^n w_i x_i = (r-1) \times n_1 + (r+1) \times n_2 - (r-1). \quad (4.15)$$

Similarly, for a vertex in N_2 :

$$\sum_{i=1}^n w_i x_i = (r-1) \times n_1 + (r+1) \times n_2 - (r+1) . \quad (4.16)$$

For a vertex in N_3 :

$$\sum_{i=1}^n w_i x_i = (r-1) \times n_1 + (r+1) \times n_2 - (r-1) . \quad (4.17)$$

For a vertex in N_4 :

$$\sum_{i=1}^n w_i x_i = (r-1) \times n_1 + (r+1) \times n_2 - (r+1) . \quad (4.18)$$

We have covered all vertices whose Hamming distance is one from the core. The vertices whose Hamming distance is larger than one are obtained by converting the bits different from the core one by one. Each time we convert a bit, $\sum_{i=1}^n w_i x_i^c$ will be reduced by $(r+1)$ if $x_i^c = 1$, and reduced by $(r-1)$ if $x_i^c = 0$.

Thus we have:

$$\sum_{i=1}^n w_i x_i^c > \sum_{i=1}^n w_i x_i^{N_1} = \sum_{i=1}^n w_i x_i^{N_3} > \sum_{i=1}^n w_i x_i^{N_2} = \sum_{i=1}^n w_i x_i^{N_4} . \quad (4.19)$$

and,

$$\sum_{i=1}^n w_i x_i^{HD=1} > \sum_{i=1}^n w_i x_i^{HD>1} , \quad (4.20)$$

where HD stands for the Hamming distance of $\mathbf{x}$ ($\mathbf{x}=(x_1, x_2, \dots, x_n)$) from the core. So we can separate this subset from the remaining vertices by the hyperplane (2.14).

For all the vertices in the proposed subset:

$$\sum_{i=1}^n w_i x_i \geq T = \min_{X \in SITV} \left(\sum_{i=1}^n w_i x_i \right). \quad (4.21)$$

For all the vertices not in the proposed subset:

$$\sum_{i=1}^n w_i x_i < T. \quad (4.22)$$

Q.E.D.

Proof (of hypothesis 3): Suppose we want to separate four vertices: different at the i th bit and the j th bit with $\{00,01,10,11\}$, and with the same value at other bits.

According to (2.10)-(2.13) we can construct the hidden neuron as the following:

$$w_i = w_j = 0. \quad (4.23)$$

For other bits:

$$w_l = \begin{cases} 4 & \text{if } x_l = 1; \\ -4 & \text{if } x_l = 0. \end{cases} \quad (4.24)$$

Suppose that, except for the i th bit and the j th bit, there exists n_1 one (true) bits in these four vertices, then we have:

$$T = 4n_1. \quad (4.25)$$

For the four vertices to be separated:

$$\sum_{i=1}^n w_i x_i \geq T \quad (\text{in fact } \sum_{i=1}^n w_i x_i = 4n_1). \quad (4.26)$$

So we do not consider the i th bit and the j th bit into account. We obtain other vertex with

Hamming distance k by converting k bits. Whenever one bit is converted, 4 is reduced

from $\sum_{i=1}^n w_i x_i$.

So, for any vertex with Hamming distance k from $\mathbf{x}$ (obtained by converting k bits), $4k$ is

reduced from $\sum_{i=1}^n w_i x_i$:

$$\sum_{i=1}^n w_i x_i = T - 4k \leq T. \quad (4.27)$$

Q.E.D.

We can simplify the values of connection weights and threshold as follows.

$$w_i = w_j = 0; \quad (4.28)$$

For other bits:

$$w_l = \begin{cases} 1 & \text{if } x_l = 1; \\ -1 & \text{if } x_l = 0. \end{cases} \quad (4.29)$$

$$T = n_1. \quad (4.30)$$

After simplification all parameters are also integral values.

In three dimensions, we illustrate one case of linear separability using Fig. 4.1 as given below.

Fig. 4.1 Visualization of separating a face for three-bit dimension

In Fig. 4.1, each vertex of the cube can be represented by a three-bit binary variable (x_1, x_2, x_3) . A face means any one of the six surfaces constructing the cube. We can easily find a plane to separate one of these surfaces from the remaining vertices.

Proof (of hypothesis 4): From hypothesis 3, we conclude that we can separate a subset with the i th bit and the j th bit combinations having values $\{00,01,10,11\}$ by hyperplane:

$$\sum_{l=1}^n w_l x_l = T_1, \quad (4.31)$$

where,

$$w_i = w_j = 0. \quad (4.32)$$

For other bits :

$$w_l = \begin{cases} 1 & \text{if } x_l = 1; \\ -1 & \text{if } x_l = 0. \end{cases} \quad (4.33)$$

$$T_1 = n_1. \quad (4.34)$$

The k connected faces can be represented by a Boolean function with the following

nature:

$$f(x_1, x_2, \dots, x_n) = (x_1 \wedge x_2 \cdots \wedge x_j) \wedge (x_{i1} \vee x_{i2} \vee \cdots \vee x_{ik}), \quad (4.35)$$

where $\wedge$ is Boolean “AND” operation and $\vee$ is Boolean “OR” operation.

According to (2.10)-(2.13) we can construct the hidden neuron as:

$$w_i = 0. \quad (4.36)$$

$$w_{ij} = \begin{cases} n/k & \text{if } x_{ij} = 1, \\ -n/k & \text{if } x_{ij} = -1. \end{cases} \quad (4.37)$$

For other bits:

$$w_i = \begin{cases} n & \text{if } x_i = 1; \\ -n & \text{if } x_i = 0. \end{cases} \quad (4.38)$$

$$T = \min_{X \in f^{-1}(1)} \left(\sum_{i=1}^n w_i x_i \right). \quad (4.39)$$

For vertices on the connected surfaces:

$$\sum_{i=1}^n w_i x_i \geq T. \quad (4.40)$$

For other vertices:

$$\sum_{i=1}^n w_i x_i < T. \quad (4.41)$$

Q.E.D.

Part 5 of the hypotheses follows from the above four parts directly.

4.3 Our Theorems for Linear Separability

Following the arguments above, we propose two additional theorems to obtain linearly separable subset of vertices.

Theorem 4.1: Given a core vertex, all vertices whose Hamming distance is one from the core, along with the core can be linearly separated from the remaining vertices by a hyperplane.

Proof: Suppose $\mathbf{x}$ is an n -dimensional input. According to (2.10)-(2.13), we can construct the hidden neuron as follows.

Let there be n_1 one (true) bits in the core $\mathbf{x}^c$. Let j denote an index where the j th bit of $\mathbf{x}^c$ is one. Let there be n_2 zero (false) bits in the core $\mathbf{x}^c$. Let k denote an index where the k th bit of $\mathbf{x}^c$ is zero.

According to formula (2.10)-(2.13), we get:

$$C_0 = n + 1. \quad (4.42)$$

For the k th bit where $x_k^c = 0$,

$$C_k = 1, \quad (4.43)$$

$$w_k = -(n - 1). \quad (4.44)$$

For the j th bit where $x_j^c = 1$,

$$C_j = n, \quad (4.45)$$

$$w_j = (n - 1). \quad (4.46)$$

So for the core, we have:

$$\sum_{i=1}^n w_i x_i = n_1(n-1). \quad (4.47)$$

We can get vertices whose Hamming distance is one from the core by converting one bit of the core. Whenever we convert one bit of the core from one to zero or from zero to one,

$\sum_{i=1}^n w_i x_i$ will be reduced by $(n-1)$.

For any vertex $\mathbf{x}$ whose Hamming distance is one from the core, we have:

$$\sum_{i=1}^n w_i x_i = (n_1 - 1)(n-1). \quad (4.48)$$

In the same way we obtain $\sum_{i=1}^n w_i x_i$ for vertices $\mathbf{x}$ whose Hamming distance is m ($m > 1$)

from the core by converting m bits of the core, either from zero to one, or from one to

zero. Also whenever one bit is converted, $(n-1)$ is reduced from $\sum_{i=1}^n w_i x_i$. So we get for all

vertices whose Hamming distance is m from the core, we have:

$$\sum_{i=1}^n w_i x_i = (n_1 - m)(n-1). \quad (4.49)$$

So we obtain,

$$T = \min_{HD < 2} \sum_{i=1}^n w_i x_i = (n_1 - 1)(n - 1) . \quad (4.50)$$

For all vertices whose Hamming distance is equal to or less than one:

$$\sum_{i=1}^n w_i x_i \geq T . \quad (4.51)$$

While for all vertices whose Hamming distance is larger than one:

$$\sum_{i=1}^n w_i x_i < T . \quad (4.52)$$

Q.E.D.

Theorem 4.2: Given a core vertex, all vertices whose Hamming distance is equal to or less than k from the core, along with the core can be linearly separated from the remaining vertices by a hyperplane.

Proof: Using (2.10)-(2.13), we get:

$$C_0 = \sum_{j=0}^k C_n^j . \quad (4.53)$$

For the i th bit where $x_i^c = 0$, we have:

$$C_i = \sum_{j=1}^k (C_n^j - C_{n-1}^j) . \quad (4.54)$$

$$w_i = 2C_i - C_0 = -C_{n-1}^k . \quad (4.55)$$

For the i th bit where $x_i^c = 1$, we have:

$$C_i = 1 + \sum_{j=1}^k C_{n-1}^j . \quad (4.56)$$

$$w_i = 2C_i - C_0 = C_{n-1}^k . \quad (4.57)$$

For any vertex x we suppose:

m_1 bits are the same with the core and $x_i^c = 0$.

m_2 bits are different from the core and $x_i^c = 0$.

m_3 bits are different from the core and $x_i^c = 1$.

m_4 bits are the same with the core and $x_i^c = 1$.

So we have,

$$\sum_{i=1}^n w_i x_i = (m_4 - m_2) C_{n-1}^k . \quad (4.58)$$

Assume there are n_1 one (true) bits and n_2 zero (false) bits in the core x^c . Using (2.10)-(2.13), we obtain for the core, we have:

$$\sum_{i=1}^n w_i x_i = n_1 C_{n-1}^k . \quad (4.59)$$

We get vertices whose Hamming distance is one from the core by converting one bit of the core. Whenever we convert one bit of the core from one to zero or from zero to one,

$\sum_{i=1}^n w_i x_i$ will be reduced by C_{n-1}^k . So for any vertex x whose Hamming distance is one

from the core, we have:

$$\sum_{i=1}^n w_i x_i = (n_1 - 1) C_{n-1}^k . \quad (4.60)$$

In the same way we get $\sum_{i=1}^n w_i x_i$ for the vertices x whose Hamming distance is l ($1 < l < k$)

from the core by converting l bits of the core, either from zero to one, or from one to zero.

Also whenever one bit is converted, C_{n-1}^k is reduced from $\sum_{i=1}^n w_i x_i$. So we gain: for

vertices whose Hamming distance is l from the core, we have:

$$\sum_{i=1}^n w_i x_i = (n_1 - l) C_{n-1}^k. \quad (4.61)$$

So we get

$$T = \min_{HD \leq k} \sum_{i=1}^n w_i x_i = (n_1 - k) C_{n-1}^k. \quad (4.62)$$

For all vertices whose Hamming distance is equal to or less than k , we have:

$$\sum_{i=1}^n w_i x_i \geq T, \quad (4.63)$$

While for all vertices whose Hamming distance is larger than k , we have:

$$\sum_{i=1}^n w_i x_i < T. \quad (4.64)$$

Q.E.D.

Suppose that we wish to separate all vertices having Hamming distance less than or equal to k together with some vertices having Hamming distance $k+1$ from the core vertex. The above separating hyperplane construction does not work in this case. As an example, suppose we want to separate a subset $\{0101, 0100, 1101, 0001, 0111, 1001, 0110\}$ (the true subset) centered at 0101 from the remaining subset $\{0000, 0010, 0011, 1000, 1010,$

1011, 1100, 1110, 1111} (the false subset). Thus $k=1$, and for $k+1=2$, only vertices 1001 and 0110 are to be separated.

According to (2.10)-(2.13), we obtain $W=\{-3, 3, -3, 3\}$, $T=0$. The values of $\sum_{i=1}^4 w_i x_i$ for the true subset are $\{6, 3, 3, 3, 3, 0, 0\}$, and for the false subset are $\{0, -3, 0, -3, -6, -3, 0, -3, 0\}$. So we cannot separate these two subsets by a hyperplane obtained from (2.10)-(2.11).

All the hypotheses and theorems discussed in this chapter are the preliminaries for our learning algorithms for Boolean neural networks. Based on these hypotheses and theorems, we present MCL and MCETL in chapter 5.

4.4 Our Approaches for BNN Construction

An alternative method to train BNNs is to begin with several core vertices. Based on this idea, we present Multi-Core Learning (MCL) algorithm [Wang 03a] and Multi-Core Expand-and-Truncate Learning (MCETL) algorithm [Wang 03b] to construct BNNs in chapter 5. A performance comparison between the MCL, MCETL, ETL and IETL algorithms is also given in chapter 5. MCL and MCETL need fewer hidden neurons in most cases. Also MCL and MCETL give simpler equations to compute the values of connection weights and thresholds than ETL. The number of operations in terms of

$\sum_{i=1}^n w_i x_i$ needed for MCL and MCETL is lower than that for ETL and IETL. But we have difficulties in determining some parameters, such as the core vertices and the number of

the core vertices. In chapter 6, we describe a novel Fast Covering Learning Algorithm (FCLA) [Wang 04a] for BNNs based on multi-level geometrical expansion. The learning process of FCLA is based on the as to that which region the new coming vertex belongs. FCLA possesses the generalization capability and needs no core vertex before training. In addition, FCLA is not sensitive to different input sequences (different input sequences result in similar network structure). Finally FCLA also needs a less number of training operations and is faster, because it determines whether to add a vertex to a hidden neuron by judging which region it belongs to, not by computing the weighted sum of every vertex in the training set.

Chapter 5

Multi-Core Learning (MCL) and Multi-Core Expand-and-Truncate Learning (MCETL)

5.1 Concept of Multi-Core Learning

Based on previous learning algorithms discussed in chapter 3, and the hypotheses and theorems discussed in chapter 4, we propose the Multi-Core Learning (MCL) algorithm and the Multi-Core Expand-and-Truncate Learning (MCETL) algorithm.

Any binary-to-binary mapping can be decomposed into a series of linearly separable functions:

$$y(x_1, x_2, \dots, x_n) = x_1 \theta(x_2 \theta(\dots(x_{n-1} \theta x_n) \dots)) , \quad (5.1)$$

where operator θ is either logical AND or logical OR.

Also any binary-to-binary mapping (5.1) can be reduced to

$$y(x_1, x_2, \dots, x_n) = \bigcup_{j=1}^k A_j, \quad (5.2)$$

where A_j is a set of vertices that can be linearly separated from the remaining vertices. Each hidden neuron represents one linearly separated set of vertices, and hence the number of hidden neurons needed is the number of separating hyperplanes. MCL represents A_j ($j=1 \dots k$) by one hidden neuron. Thus, to represent the form (5.2) we need k hidden neurons altogether.

MCL and MCETL begin with several cores, and extend them simultaneously. Hence MCL and MCETL can avoid blocking problems in ETL and IETL. MCETL combines MCL with the idea of expand-and-truncate mechanism. In MCETL, when SITVs [Kim 95] cannot expand further more, instead of selecting new cores for new SITVs, we convert the status of all vertices not in SITVs, and then expand each SITV as ETL does. MCL and MCETL need fewer hidden neurons and use simpler equations to train BNNs than ETL does. In addition, the number of computing operations of MCL and MCETL is lower than that of ETL and IETL.

If we represent RHP (defined in (2.4)) by a circle, Fig. 5.1 shows the training process of MCL, and Fig. 5.2 shows the training process of MCETL. White regions stand for the true subsets and black regions stand for the false subsets. The figure in each region stands for the order of generating hyperplanes (hidden neurons). Kim and Park [Kim 95] have given the analysis of convergence for finite inputs of constructive learning algorithms. MCL and MCETL belong to constructive learning algorithms with finite inputs and

guarantee convergence.

Fig. 5.1 Training process of MCL

Fig. 5.2 Training process of MCETL

5.2 Multi-Core Learning (MCL) Algorithm [Wang 03a]

5.2.1 Algorithm

Multi-Core Learning constructs a three-layered neural network with one input layer, one hidden layer and one output layer.

The hard-limiter activation function for the j th hidden neuron is:

$$h_j = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_{ij} x_i \geq T_j \\ 0 & \text{otherwise} \end{cases}, \quad (5.3)$$

where w_{ij} is the connection weight between the i th bit of the input and the j th hidden neuron, x_i is the i th bit of the input, n is the dimension of the input, T_j is the threshold of the j th hidden neuron, and h_j is the hard limiter output of the j th hidden neuron.

The hard-limiter activation function for the output layer is:

$$y = \begin{cases} 1 & \text{if } \sum_{j=1}^m w_j^o h_j \geq T^o \\ 0 & \text{otherwise} \end{cases}, \quad (5.4)$$

where w_j^o is the connection weight between the j th hidden neuron and the output neuron, T^o is the threshold of the output neuron, and y is the final output.

We first select m true vertices as the core for each of these m SITVs. Every two cores have a Hamming distance larger than two. So $m \leq 2^{\lfloor \frac{n}{3} \rfloor}$, where n is the dimension of the input. In the training process of MCL, all hidden neurons are trained simultaneously. When each individual hidden neuron is trained, vertices represented by other hidden neurons are considered as “don’t care”.

We give the following steps for MCL algorithm as described in Fig. 5.3. To obtain an efficient neural network structure, we need to generate maximum linearly separable subsets. We need to combine two connected hidden neurons if possible. In step (d), a method is given to combine two continuous neurons. Sometimes, we do not expect a 100% precision; an approximate result is acceptable. A hidden neuron with weights

$\{w_1, w_2, \dots, w_l, \dots, w_k \pm \varepsilon \times C_0, \dots, w_n\}$ can be combined with a hidden neuron with weights $\{w_1, w_2, \dots, -w_l, \dots, w_k \mp \varepsilon \times C_0, \dots, w_n\}$; the connection weights for the result hidden neuron are $\{w_1, w_2, \dots, 0, \dots, w_k, \dots, w_n\}$. Here ε is a small constant value and C_0 is the number of vertices represented by this hidden neuron. The value of ε is up to the expected precision. Large value of ε causes more relax learning and more error. Small value of ε causes less relax learning and little error. The detailed training process is also shown in the flowchart in Fig. 5.4.

In *step a* of Fig. 5.4, the Hamming distance between any two cores must be more than two. After we select a true vertex randomly as a core, we convert 3 continuous bits (in groups) of this true vertex. If the result vertex is true, it is also considered as a SITV core. For example, we convert the 1st, 2nd 3rd bit to get X_1 , convert the 4th 5th, 6th bit to get X_2 , and convert all the 1st, 2nd 3rd, 4th 5th, 6th bit to get X_3 . Continuing with this example, suppose that we begin with an arbitrary true vertex $\{001001\}$, and $m=4$. Then if $\{001110, 110001, 110110\}$ are true vertices, they are taken as cores.

- a) Select a true vertex randomly as a core vertex. Convert 3 continuous bits (in groups) of this true vertex. If the result vertex is true, consider it as a SITV core too.
- b) Set $flag_j=0$ and $k_j=0$ for each SITV.
- c) Test vertices whose Hamming distance is one from SITV cores.
- d) When three vertices are included in $SITV_j$, test the vertex whose Hamming distance is two from $SITV_j$ core and one from each of the other two vertices.
If the tested vertex is false, go to step f to examine other vertices whose Hamming distance is one from the core; else if true, include this vertex in $SITV_j$, and combine this hidden neuron with other hidden neurons if possible. We scan the connection weights of each hidden neuron, and select two hidden neurons with all the corresponding bits of connection weights the same except for one contrary bit. We combine the two hidden neurons by setting the weight of the contrary bit as zero, and coping those of other bits.
- e) If more than three and less than $n+1$ vertices are enclosed in $SITV_j$, go to step l for $SITV_j$.
- f) If only two vertices are included and $flag_j=0$, the core is moved to the other vertex. Set $flag_j=1$, and go to step c . Else if only two vertices are included and $flag_j=1$, go to step l for $SITV_j$.
- g) If no other vertex except for the core is included, this corresponding hidden neuron can only separate one vertex from the rest vertices. Go to step l for $SITV_j$.
- h) If $n+1$ vertices are true (all vertices with Hamming distance one are true), set $k_j=2$ and, go to step k .
- i) Include all the vertices at Hamming distance k_j-1 . Test vertices at Hamming distance k_j from the $SITV_j$ core. If all C_n^k vertices at Hamming distance k_j are true, go to step k .
Else go to step l for $SITV_j$.
- j) Increment k_j (by one), and go to step j .
- k) If all the true vertices have been included in $\bigcup_{j=1}^k SITV_j$, stop the training process. Else
select a true vertex which is not included in $\bigcup_{j=1}^k SITV_j$ as the new core, and go to step c .

Fig. 5.3 MCL algorithm

Fig. 5.4 Flowchart for MCL algorithm

5.2.2 Parameter Computation

Based on the hypotheses and theorems in chapter 4, and the method to determine the separating hyperplanes discussed in section 2.1.2 of chapter 2, we give formulas for MCL to train the neural networks.

Formulas to compute the connection weights for the hidden neuron (hidden neuron j here) are as follows:

$$w_{ij} = 2C_{ij} - C_{0j}, \quad (5.5)$$

$$C_{ij} = \sum_{l=1}^{C_{0j}} x_i^l, \quad (5.6)$$

where C_{0j} is the number of vertices in $SITV_j$, x_i^l is the i th bit of the l th vertex in $SITV_j$.

Formula to compute the threshold for the hidden neuron (hidden neuron j here) is as follows:

$$T_j = \min_{X \in SITV_j} \left(\sum_{i=1}^n w_{ij} x_i \right), \quad (5.7)$$

where x_i is the i th bit of x . We compute all the results of $\sum_{i=1}^n w_{ij} x_i$ for each vertex x in $SITV_j$, and let T_j equal to the minimum of $\left\{ \sum_{i=1}^n w_{ij} x_i \right\}$. $SITV_j$ can be linearly separated from the remaining vertices.

For all vertices in $SITV_j$

$$\sum_{i=1}^n w_{ij} x_i \geq \min_{X \in SITV_j} \left(\sum_{i=1}^n w_{ij} x_i \right). \quad (5.8)$$

For the remaining vertices,

$$\sum_{i=1}^n w_{ij} x_i < \min_{X \in SITV_j} \left(\sum_{i=1}^n w_{ij} x_i \right). \quad (5.9)$$

Thus, for all vertices in $SITV_j$, $h_j=1$; for the remaining vertices (not included in $SITV_j$),

$h_j=0$. When we compute T_j , we need not compute $\sum_{i=1}^n w_{ij}x_i$ for any vertex outside $SITV_j$, which simplifies our training process. The number of operations needed by MCL is $O(2^n h)$, where h is the number of hidden neurons, and n is the input dimension.

The connection weights and threshold for the output layer can be obtained by the following formulas:

$$T^o=1, \quad (5.10)$$

$$w_j^o = 1, \quad (5.11)$$

where T^o is the threshold of the output neuron and w_j^o is the connection weight between the j th hidden neuron and the output neuron.

If $X \in \bigcup A_j$, then $\exists j$, st. $X \in A_j$, that is, the output of the j th hidden neuron is one, $h_j=1$. So $y(x_1, x_2, \dots, x_n)=1$.

If $y(x_1, x_2, \dots, x_n)=1$, then $\exists j$, st. $h_j=1$. That is, $\exists A_j$ st. $X \in A_j$, then $X \in \bigcup A_j$.

5.3 Datasets for MCL

We now give some examples to explain MCL algorithm. The performance comparison between MCL and previous learning algorithms for BNNs (discussed in chapter 3) is given to illustrate the advantages of MCL.

Example 5.1 Approximation of a circular region using 6-bit quantization [Gray 92, Kim 95, Yamamoto 97]

This example is to separate a circular region in 2-D space which is a square with sides of length 8 with the coordinate origin in the lower left corner. A circle of diameter 4 is placed within the square, and then the space is sampled with 64 grid points located at the center of 64 identical squares. These 64 identical squares cover the large square as shown in Fig. 5.5. The subset of true vertices for this Boolean function of 6 variables is: $T = \{010011, 011011, 011010, 101011, 100011, 100010, 011101, 011100, 010100, 101100, 100101, 100100\}$. This is a benchmark example used by many earlier researchers as well [Gray 92, Kim 95, Yamamoto 97].

We choose four SITVs with 011011, 100011, 011100, and 100100 as their cores respectively. We have chosen these core vertices whose Hamming distance between any two cores is larger than or equal to three. Then we test the vertices whose Hamming distance is one from the core for each SITV.

The four SITVs can be respectively expanded to $\{010011, 011011, 011010\}$, $\{101011, 100011, 100010\}$, $\{011101, 011100, 010100\}$, $\{101100, 100101, 100100\}$. Till now all true vertices have been included in $\bigcup_{j=1}^4 SITV_j$, and the training process stops. So four hidden neurons are needed.

According to (5.5)-(5.7) and (5.10)-(5.11), we obtain the results of connection weights

and thresholds which are shown in Table 5.1. Fig. 5.6 shows the neural network structure constructed by MCL with four hidden neurons.

Fig. 5.5 Circular region obtained by 6-bit quantization

Fig. 5.7 shows the neural network structure constructed by ETL. Five hidden neurons are needed by using the ETL algorithm. MCL is better than ETL because it uses fewer hidden neurons to solve the same problem. BLTA uses much more hidden neurons than ETL [Kim 95]. Using IETL of Yamamoto, also four hidden neurons are needed. Fig. 5.8 shows the neural network structure constructed by IETL. The results of connection weights and thresholds by IETL are shown in Table 5.2. Comparing Table 5.1 with Table 5.2, we observe that the values of connection weights and thresholds in MCL are much smaller than those in IETL. The smaller numerical values are preferred for hardware realization.

Fig. 5.6 Neural network on MCL of 6-bit circle region approximation

Fig. 5.7 Neural network on ETL for 6-bit circle region approximation

Fig. 5.8. Neural network on IETL for 6-bit circle region approximation

Table 5.1 Weights and thresholds of MCL for 6-bit circle region approximation

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	T_j
$j=1$	-3	3	1	-3	3	1	7
$j=2$	-3	3	1	3	-3	-1	6
$j=3$	3	-3	-1	-3	3	1	6
$j=4$	3	-3	-1	3	-3	-1	5

Table 5.2 Weights and thresholds of IETL for 6-bit circle region approximation

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	T_j
$j=1$	-6	6	2	-6	6	2	13
$j=2$	-58	-6	-2	-6	6	2	-11
$j=3$	-58	-6	-2	-6	6	2	-58
$j=4$	6	-6	-2	6	-6	-2	9

BLTA always uses much more hidden neurons than ETL, which has been analyzed in Kim and Park' work [Kim 95]. Hence, we do not consider BLTA for comparison in our work.

We now consider a few more examples.

Example 5.2 A given 7-bit function applied in [Kim 95, Yamamoto 97]

We now consider the following 7-bit Boolean function $f(x_1, x_2, \dots, x_7)$ having {0000000, 0000001, 0000010, 0000100, 0001000, 0010000, 0100000, 1000000, 0000011, 0000101, 0001001, 0010001, 0100001, 1000001, 0010101, 0100010, 0100100, 0101000, 0110000, 1000101, 1010001, 1100000, 1100101, 1000010, 1000100, 1010100, 1100100, 1100110, 1101010, 1110010, 1010000, 1001000, 1011101, 1110101, 1010101, 1010111} as the true vertices. It is a problem proposed by Kim [Kim 95]. We use this Boolean

function for comparison with ETL and IETL. The connection weights and thresholds for this 7-bit function by using MCL are shown in Fig. 5.9 and Table 5.3. Three hidden neurons are needed by MCL. While in ETL the number of needed hidden neurons is seven (shown in Fig. 5.10). IETL (shown in Fig. 5.11) also needs three hidden neurons. These values of connection weights and thresholds in IETL (Table 5.4) are also much larger than those in MCL.

Fig. 5.9 Neural network for 7-bit function problem on MCL

Table 5.3 Weights and thresholds of MCL for 7-bit function problem

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	w_{7j}	T_j
$j=1$	3	-1	-19	-29	-19	-29	-37	13
$j=2$	3	-3	1	-3	3	3	-1	9
$j=3$	2	2	-2	-2	3	2	0	8

Fig. 5.10 Neural network for 7-bit function problem on ETL

Fig. 5.11 Neural network for 7-bit function problem on IETL

Table 5.4 Weights and thresholds of IETL for 7-bit function problem

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	w_{7j}	T_j
$j=1$	6	-2	-38	-58	-38	-59	2	-75
$j=2$	-17	-15	13	15	15	15	-17	10
$j=3$	6	2	2	-2	6	-2	-2	9

Example 5.3 Approximation of a circular region using 12-bit quantization [Gray 92, Kim 95]

ETL and IETL learn exactly from the training set with 100% precision and can not generalize data not in the training set. However, often an approximate result is acceptable and a more general learning algorithm which can response correctly to new data is expected. So we relax the training process of the MCL algorithm by setting ε as 0.01.

Example 5.3 explains the relaxed version.

This example is similar to example 5.1, which separates a circular region in a 2-D square space with sides of length being 64 if the coordinate origin in the lower left corner. A

circle of diameter 32 is placed within the square, locating the center at (32, 32), and then the space is sampled with 4096 grad points located at the center of 4096 identical squares covering the larger square.

The results for approximation of a circular region using 12-bit quantization by MCL are shown in Table 5.5. We observe that twelve hidden neurons are needed by MCL to gain an exact result. In Kim and Park's work, seven hidden neurons are needed by using ETL to solve this problem. By twelve hidden neurons, MCL has a 100% precision. If we use four hidden neurons for MCL we get a 99.6% precision for approximation. The results of connection weights and thresholds are shown in Table 5.6. Fig. 5.12 shows the comparison between the exact learning results and the approximate learning results (99.6% precision) by the MCL algorithm. In the approximate results, 16 vertices out of 4096 vertices are mistakenly learned.

Table 5.5 Weights and thresholds of MCL for 12-bit circle region approximation by exact learning

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	w_{7j}	w_{8j}	w_{9j}	w_{10j}	w_{11j}	w_{12j}	T_j
$j=1$	-123	123	5	5	5	3	-123	123	123	5	1	1	373
$j=2$	-123	123	123	5	1	1	-123	123	5	5	5	3	373
$j=3$	-21	21	-21	15	7	3	-21	21	-21	15	7	3	74
$j=4$	-123	123	5	5	5	3	123	-123	-123	-5	-1	-1	243
$j=5$	-123	123	123	5	1	1	123	-123	-5	-5	-5	-3	355
$j=6$	-21	21	-21	15	7	3	21	-21	21	-15	-7	-3	70
$j=7$	123	-123	-5	-5	-5	-3	-123	123	123	5	1	1	335
$j=8$	123	-123	-123	-5	-1	-1	-123	123	5	5	5	3	243
$j=9$	21	-21	21	-15	-7	-3	-21	21	-21	15	7	3	70
$j=10$	123	-123	-5	-5	-5	-3	123	-123	-123	-5	-1	-1	225
$j=11$	123	-123	-123	-5	-1	-1	123	-123	-5	-5	-5	-3	225
$j=12$	21	-21	21	-15	-7	-3	21	-21	21	-15	-7	-3	66

Table 5.6 Weights and thresholds of MCL for 12-bit circle region approximation by approximate learning

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	w_{7j}	w_{8j}	w_{9j}	w_{10j}	w_{11j}	W_{12j}	T_j
$j=1$	-203	203	43	25	13	7	-203	203	43	25	13	7	462
$j=2$	-203	203	43	25	13	7	203	-203	-43	-25	-13	-7	374
$j=3$	203	-203	-43	-25	-13	-7	-203	203	43	25	13	7	374
$j=4$	203	-203	-43	-25	-13	-7	203	-203	-43	-25	-13	-7	286

Fig. 5.12 Comparison between exact learning and approximate learning by MCL algorithm

Example 5.4 4-bit parity function

MCL does well in nicely clustered problems. Unfortunately, MCL does not always result in good neural network structures. We illustrate this point by using an example of 4-bit parity function.

The results of connection weights and thresholds for 4-bit parity function by using MCL are shown in Table 5.7. Eight hidden neurons are needed by the algorithm, which are the same as BLTA. While ETL and IETL only need four hidden neurons.

In case of parity functions, any two neighboring vertices (any two vertices with Hamming distance one) belong to different sets. n -bit parity function is a case of bad clustered problem. Wholly 2^{n-1} hidden neurons are needed for an n -bit parity function by using MCL. However, in ETL and IETL, n hidden neurons are needed for n -bit parity problems.

Table 5.7 Weights and thresholds of MCL for 4-bit parity function

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	T_j
$j=1$	-1	-1	-1	-1	0
$j=2$	-1	-1	1	1	2
$j=3$	-1	1	-1	1	2
$j=4$	1	-1	-1	1	2
$j=5$	-1	1	1	-1	2
$j=6$	1	-1	1	-1	2
$j=7$	1	1	-1	-1	2
$j=8$	1	1	1	1	4

Each hidden neuron in MCL represents only one vertex when solving n -bit parity problems. If most hidden neurons represent only one vertex each, the whole neural network is inefficient. We then combine the idea of MCL with that of ETL. We call the resulting algorithm as Multi-Core Expand-and-Truncate Learning (MCETL).

5.4 Multi-Core Expand-and-Truncate Learning (MCETL) [Wang 03b]

Multi-Core Expand-and-Truncate Learning (MCETL) also constructs a three-layered neural network with one input layer, one hidden layer and one output layer. The hard-limiter activation function for the hidden layer and the output layer for MCETL are the same as those for MCL: (5.3) and (5.4).

The training process for the hidden layer of MCETL is similar to that of MCL except for the last step. Instead of selecting new cores for new SITVs, we convert the status of all vertices not in SITVs, and then expand each SITV as ETL does. We revise the last step (in Fig 5.3) as follows to obtain the MCETL algorithm:

l): If all the true vertices have been separated, end the training process. Else we convert the status of all vertices not in SITVs (i.e. true vertices are converted to false vertices, and vice versa). Then expand each SITV by adding as many true vertices as possible.

Also when each individual hidden neuron is trained, vertices covered by other hidden neurons are considered as “don’t care”.

Formulas to train hidden neurons (neuron j) in MCETL are the same as those in MCL, (5.5), (5.6) and (5.7).

Suppose we begin with m core vertices, and altogether we have h hidden neurons, we order these neurons as $\{1, 2, \dots, h\}$. The connection weights between the hidden layer and the output layer, and threshold for the output layer can be computed by the following formulas:

$$w_j^o = \begin{cases} \lfloor (h-j)/m \rfloor \times m + 1, & \text{if } \lfloor j/m \rfloor \text{ is even,} \\ \lfloor -(h-j)/m \rfloor \times m - 1, & \text{if } \lfloor j/m \rfloor \text{ is odd,} \end{cases} \quad (5.12)$$

$$T^o = \begin{cases} 0 & \text{if } h/m \text{ is even,} \\ 1 & \text{if } h/m \text{ is odd,} \end{cases} \quad (5.13)$$

where T^o is the threshold of the output neuron and w_j^o is the connection weight from the j th hidden neuron to the output neuron.

The validity of the MCETL algorithm follows the validity of ETL and MCL. We now illustrate the advantages of MCETL over MCL by giving the example of 4-bit parity function (Example 5.4).

5.5 An Example for MCETL

Example 5.6 4-bit parity function by MCETL

We begin constructing the neural network for 4-bit parity function with 2 core vertices: 0000 and 1111. 0000 and 1111 are selected as core vertices based on the rule that the Hamming distance between any two cores must be larger than two. Other vertices satisfying this restriction can also be considered as initial cores and the resultant network is the same. According to (5.5)-(5.7) and (5.12)-(5.13), we obtain the connection weights and thresholds as shown in Table 5.8, and the neural network structure shown in Fig. 5.13.

The neural network structure constructed by MCETL is the same as that by ETL and IETL. Four hidden neurons are needed by MCETL. The idea of expand-and-truncate is fit for bad clustered problems.

Table 5.8 Weights and thresholds of MCETL for 4-bit parity function

Neuron	w_{1j}	w_{2j}	w_{3j}	w_{4j}	T_j
$j=1$	-1	-1	-1	-1	0
$j=2$	1	1	1	1	4
$j=3$	-2	-2	-2	-2	-2
$j=4$	2	2	2	2	6

Fig. 5.13 Neural network for 4-bit parity function on MCETL

5.6 Concluding Remarks of MCL and MCETL

Multi-Core Learning (MCL) and Multi-Core Expand-and-Truncate Learning (MCETL) algorithms originate from Expand-and-Truncate Learning (ETL) and Improved Expand-and-Truncate Learning (IETL). ETL and IETL begin with a true vertex as the core of SITV. In ETL and IETL, SITV is expanded by adding more true vertices until no vertex can be added any more. Then false vertices are converted to true vertices, and true vertices, which are not in SITV, are converted to false vertices. This process continues until all true vertices are separated from all false vertices. The reason why a hyperplane cannot expand is due to the existence of false vertices around the SITV hypersphere. These false vertices block the expansion of the SITV hypersphere.

Our algorithms, MCL and MCETL, begin with several cores, and expand the corresponding SITVs simultaneously. These two algorithms avoid blocking problems in

ETL and IETL, and need fewer hidden neurons in most cases. Also MCL and MCETL use simpler equations to train the neural networks than ETL does. In some cases, IETL and MCL need a similar number of hidden neurons, but the values of connection weights and thresholds in MCL are much simpler than those in ETL and IETL. The smaller the value, the easier it can be realized by hardware. So we say MCL and MCETL are better than both ETL and IETL. Let h be the number of hidden neurons needed. The number of operations needed by ETL and IETL is $(O(2^nh))$. The reason for this is that, in ETL and IETL, we need to consider every input vectors in the training set for determining each hidden neuron. However, in MCL and MCETL, when we compute the parameters for one hidden neuron, vertices represented by other hidden neurons are not taken into account. Thus we observe that the number of operations needed by MCL and MCETL is $O(2^n+h)$.

However, ETL, IETL and MCETL do well in n -bit parity function. But MCL does not do well in this badly clustered problem, so we modify MCL to obtain MCETL.

In MCETL, if we begin with one core vertex, MCETL degenerates to the ETL algorithm. On the contrary, if we begin with h core vertices, and altogether h hidden neurons are needed, MCETL degenerates to MCL. So MCETL is more flexible and more efficient than the other two algorithms to solve Boolean function mapping problems.

For MCL and MCETL, we give a simple rule to decide the initial core vertices for SITVs. The distances between every two cores are larger than two. So in our method, given the dimension of the input n , we can determine m by using $m = 2^{\lfloor \frac{n}{3} \rfloor}$ ($m \leq 2^{\lfloor \frac{n}{3} \rfloor}$ is a bound).

This simple rule has computational advantage.

But MCL and MCETL have no generalization capability. We need to design an approach for fast training algorithms with generalization capability. We propose a novel Fast Covering Learning Algorithm (FCLA) for BNNs based on multi-level geometrical expansion in the next chapter.

Chapter 6

Fast Covering Learning Algorithm (FCLA) Based on Geometrical Expansion for Boolean Neural Networks

6.1 Introduction

In the last ten years, many novel learning algorithms for Binary Neural Networks (BNNs) have been proposed, including BLTA, ETL, IETL, NETLA and CSCLA, which are discussed in chapter 3. Further, we introduced our algorithms MCL and MCETL which are given in chapter 5. We have discussed the advantages and disadvantages of these algorithms. To avoid the disadvantages in these methods, we propose a novel Fast Covering Learning Algorithm (FCLA) for BNNs.

The FCLA is based on multi-level geometrical expansion. The main advantages of FCLA over previous learning algorithms for BNNs are:

- (i) The FCLA possesses the generalization capability, but all previous learning algorithms (except for BLTA) do not.
- (ii) The FCLA does not need a core vertex before training, which makes it more general to be applied.
- (iii) The FCLA needs less training time, because it determines whether to add a vertex to a hidden neuron by judging which region it belongs to, not by computing the weighted sum of every vertex in the training set.
- (iv) The FCLA is not sensitive to different input sequences of the same training set.

Before explaining FCLA, we first continue with the geometrical analysis in chapter 4. A set of 2^n binary patterns $\{0,1\}^n$, each with n bits, can be considered as an n -dimensional unit hypercube. Each pattern is located on one vertex of the hypercube. We can get the exhypersphere (RHP: Reference Hypersphere) of this unit hypercube with radius $\sqrt{n}/2$ as (6.1).

$$(x_1 - \frac{1}{2})^2 + (x_2 - \frac{1}{2})^2 + \cdots + (x_n - \frac{1}{2})^2 = \frac{n}{4}. \quad (6.1)$$

All patterns lie on the surface of RHP (shown in Fig. 2.1). Assume that those patterns can be separated into two (true and false) subsets by m $(n-1)$ -dimensional hyperplanes. These m $(n-1)$ -dimensional hyperplanes partition all vertices into $m+1$ groups. Vertices in each group have the same desired outputs: zero or one. The j th hyperplane can be represented by hidden neuron j . Suppose that, only one $(n-1)$ -dimensional hyperplane is needed to separate the true subset from the false subset, i.e., there exists such a hyperplane that all vertices in the true subset lie on one side of the hyperplane, and all vertices in the false

subset lie on the other side; then these two subsets are linearly separable. We can separate the true subset from the false subset by a hyperplane if the true (false) subset is linearly separable.

Our FCLA is based on Theorem 2.1. For the sake of clarity, we give the statement of the theorem below.

Theorem 6.1: *A Boolean function f is linearly separable if and only if there exists a separating hypersphere (SHP) such that all true vertices lie inside or on the surface of this hypersphere, and all false vertices lie outside. Let the SHP be expressed in the following form:*

$$\sum_{i=1}^n (x_i - c_i) = r^2, \quad (6.2)$$

where $(c_1, c_2, \dots, c_n)$ is the center of the hypersphere, r is the radius.

We get the intersection of SHP and RHP from (6.1) and (6.2) as (6.3):

$$\sum_{i=1}^n (1 - 2c_i)x_i = r^2 - \sum_{i=1}^n c_i^2. \quad (6.3)$$

From (2.5)-(2.7), we observe that, all vertices in SHP lie on one side or on the surface of hyperplane (6.3):

$$\sum_{i=1}^n (1 - 2c_i)x_i \leq r^2 - \sum_{i=1}^n c_i^2. \quad (6.4)$$

All the remaining vertices lie on the other side of hyperplane (6.3):

$$\sum_{i=1}^n (1 - 2c_i)x_i > r^2 - \sum_{i=1}^n c_i^2 . \quad (6.5)$$

So the intersection (6.3) of SHP and RHP is the separating hyperplane, which separates the true subset from the false subset. To restrict our discussion for integer-valued weights, we multiply both sides of (6.3) by the number of samples separated by that hypersphere:

C_0 . We get:

$$\sum_{i=1}^n (\{2 \sum_{k=1}^v x_i^k\} - C_0)x_i = C_0 [\{\sum_{i=1}^n c_i^2\} - r^2] . \quad (6.6)$$

The coefficients of x_i (on the left hand side of equation (6.6)) are the connection weights, and the term on the right hand side of equation (6.6) is the threshold of the corresponding hidden neuron. We represent (6.6) using a hard-limiter model for a hidden neuron, which has output zero for values less than the threshold, and has output one for values greater than or equal to the threshold. Each hidden neuron in a BNN represents one (of these m) hyperplane. Different approaches for construction of BNNs use different methods for determining these hyperplanes.

Each of these hyperplanes can be represented by a hidden neuron. Each hidden neuron in the hidden layer represents a linearly separable subset. So the number of hidden neurons needed is equal to the number of separating hyperplanes. Less number of hidden neurons and simple connection weights indicate a simple neural network structure. Many researchers try to simplify the neural network structure when solving problems. A neural network can be simplified by improving the representation capability of each hidden

neuron. In this chapter, we reduce the number of hidden neurons by expanding the scope of each hidden neuron in the training process based on multi-level geometrical analysis. We call our method Fast Covering Learning Algorithm (FCLA).

FCLA constructs a three-layered neural network with one input layer, one hidden layer and one output layer. A neural network constructed by FCLA is illustrated in Fig. 6.1.

FCLA begins with an empty hidden layer, and adds hidden neurons to the hidden layer when necessary. It belongs to constructive learning.

Fig. 6.1 A three-layered neural network constructed by FCLA

6.2 Idea of Geometrical Method [Wang 03c, 04a]

We first separate a linearly separable subset with C_0^j vertices by the j th hidden neuron. In our approach, we define the center of hypersphere j : c_j as the gravity center of these C_0^j vertices. Three regions are defined based on c_j : the match region defines a hypersphere that vertices (presented by this hidden neuron) are exactly in or on this hypersphere; the claim region defines a hypersphere that vertices within that region can

be represented by this hidden neuron by immediate expansion; the boundary region defines a region outside which vertices are not considered to be represented by this hidden neuron. Three central radii with center c_j for this hypersphere are defined as r_1^j , r_2^j and r_3^j , where $r_1^j < r_2^j < r_3^j$.

The r_1^j is called the match radius which defines the match region j . The r_1^j is the minimal value which meets the condition that all of the C_0^j vertices are exactly in or on hypersphere j . So it is reasonable to define r_1^j as the largest Euclidean distance between the center of hypersphere j and all these C_0^j vertices represented by hidden neuron j . Vertices having in the match region need not be learned again.

The r_2^j is called the claim radius which defines the claim region j . The r_2^j is a little larger than r_1^j . Vertices in the claim region j can be included in hypersphere j by its immediate expansion. Hence vertices in the claim region j cause the expansion of hypersphere j . This vertex is outside this hypersphere before expansion, and just on the surface of it after expansion.

The r_3^j is called the boundary radius which defines the boundary region j . Vertices in the boundary region j cannot be included in hypersphere j by its immediate expansion, but are expected to be included in hypersphere j in the next (or later) training circle. So vertices in the boundary region are backed up to be learned in the next training circle.

Vertices out of the boundary region j are too far from hypersphere j . We do not expect to

include these vertices in it. So a vertex out of the boundary region of any hidden neuron results in the creation of a new hypersphere (a new hidden neuron) to represent it.

Now we illustrate the training process by a visualized example in Fig 6.2. A new vertex (say x) within the “claim region” in Fig. 6.2 (a) (defined using r_2) causes an immediate expansion of the hypersphere, to get a hypersphere with the new vertex on the surface of it after expansion (Fig. 6.2 (b)). A new vertex within the “boundary region” (defined using r_3) cannot be included in the hypersphere by its immediate expansion, but is expected to be included in the next (or later) training circle. A new vertex (say y) (Fig. 6.2 (b)) beyond r_3 is too far from this hypersphere. So it causes to the creation of a new hypersphere (a new hidden neuron) to represent it (Fig. 6.2 (c)).

Fig. 6.2 The process of geometrical expansion (in Boolean field)

6.3 Parameters for the Hidden Neurons

6.3.1 Primary Parameters: w_{ij} and t_1^j

Each separating hyperplane is represented by a hidden neuron. The process of generating the separating hyperplane is equal to the process of generating the separating hypersphere. When the new coming vertex is in the claim region of some hidden neuron, or out of the boundary region of all hidden neurons, the parameters of the neural network need to be modified. We want to separate a linearly separable subset with C_0^j vertices by the j th hidden neuron. For separating hypersphere j we should first determine its center and radius.

We define the center of hypersphere j as the gravity center of these C_0^j vertices.

$$\mathbf{c}_j = (c_1^j, c_2^j, \dots, c_n^j), \quad (6.7)$$

$$c_i^j = \sum_{k=1}^{C_{0j}} x_i^k / C_0^j, \quad (6.8)$$

where $\mathbf{c}_j$ is the center of hypersphere j and x_i^k is the i th bit of the k th training vertex of these C_0^j vertices.

After defining the center, we continue to discuss the radius for the separating hypersphere.

The match radius is defined as the largest Euclidean distance between the center $\mathbf{c}_j$ and all of C_0^j vertices:

$$(r_1^j)^2 = \max_{k=1}^{C_0^j} \sum_{i=1}^n (x_i^k - c_{ij})^2, \quad (6.9)$$

where x_i^k is the i th bit of the k th training vertex of these C_0^j vertices.

The corresponding separating hyperplane (hyperplane j) is the intersection of RHS and hypersphere j :

$$\sum_{i=1}^n (2c_i^j - 1)x_i = \sum_{i=1}^n (c_i^j)^2 - (r_1^j)^2. \quad (6.10)$$

Equation (6.10) is the separating hyperplane which can be represented by a linear hidden neuron. In order to restrict our discussion for integer-valued weights, we multiply both sides of (6.10) by the number of vertices separated by that hypersphere: C_0^j . We get:

$$\sum_{i=1}^n (\{2 \sum_{k=1}^{C_0^j} x_i^k\} - C_0^j)x_i = C_0^j [\{\sum_{i=1}^n (c_i^j)^2\} - (r_1^j)^2]. \quad (6.11)$$

The coefficients of x_i (on the left hand side) are the connection weights, and the term on the right hand side is the threshold of the corresponding hidden neuron. We represent (6.11) using a hard-limiter model for a neuron, which has an output of zero for values less than the threshold, and one for values greater than or equal to the threshold.

From (6.11) we get the integer value of connection weights as follows:

$$w_{ij} = C_0^j (2c_i^j - 1) = 2 \sum_{k=1}^{C_0^j} x_i^k - C_0^j, \quad (6.12)$$

where w_{ij} is the connection weight between the i th bit of the input and hidden neuron j .

There are three thresholds for hidden neuron j according to three radii of hypersphere j :

t_1^j , t_2^j and t_3^j .

From (6.9) and (6.11), we get:

$$t_1^j = C_0^j \left(\sum_{i=1}^n (c_i^j)^2 - (r_1^j)^2 \right) = \min_{k=1}^{C_0^j} \sum_{i=1}^n C_0^j (2c_i^j x_i^k - (x_i^k)^2), \quad (6.13)$$

where x_i^k is 0 or 1, so $x_i^k = (x_i^k)^2$.

From (6.12) and (6.13) we get:

$$t_1^j = \min_{k=1}^{C_0^j} (C_0^j (2c_i^j - 1) x_i^k) = \min_{k=1}^{C_0^j} \sum_{i=1}^n w_{ij} x_i^k. \quad (6.14)$$

For all vertices included in hypersphere j :

$$\sum_{i=1}^n w_{ij} x_i^k \geq t_1^j. \quad (6.15)$$

For all vertices out of hypersphere j :

$$\sum_{i=1}^n w_{ij} x_i^k < t_1^j. \quad (6.16)$$

To avoid re-computing, we simplify (6.12) and (6.14) to obtain the equivalent form in (6.17)-(6.19) in order to compute the connection weights w_{ij} and the threshold t_1^j (for the j th hidden neuron).

For a new coming vertex $\mathbf{x}^k$, if $t_3^j > \mathbf{w}^T \mathbf{x}^k$ (where $\mathbf{w}^T$ is the transfer matrix of $\mathbf{w}$) for all

existing hidden neurons, a new hidden neuron (say, neuron j) is created to represent it.

The parameters, w_{ij} and t_1^j , for this neuron are:

$$w_{ij} = \begin{cases} 1 & \text{if } x_i = 1 \\ -1 & \text{if } x_i = 0 \end{cases}, \quad (6.17)$$

$$t_1^j = \sum_{i=1}^n w_i x_i^k, \quad (6.18)$$

where w_{ij} is the connection weight from i th input bit to hidden neuron j , and t_1^j is the match threshold for hidden neuron j .

For a new coming vertex $\mathbf{x}^k$, if $t_2^j \leq \mathbf{w}^T \mathbf{x}^k < t_1^j$, using (6.12) and (6.14), we have:

$$w_{ij} = \begin{cases} w_{ij} + 1 & \text{if } x_i = 1 \\ w_{ij} - 1 & \text{if } x_i = 0 \end{cases}. \quad (6.19)$$

$$t_1^j = \sum_{i=1}^n w_i x_i^k. \quad (6.20)$$

The parameters in (6.19) and (6.20) are the same as in (6.17) and (6.18).

6.3.2 Secondary Parameters: t_2^j and t_3^j

The parameters t_2^j and t_3^j are secondary parameters which are determined by parameters r_2^j and r_3^j to control the geometrical expansion process. We either add a new coming vertex to a hidden neuron or create a new hidden neuron to represent it, or put it aside to be reconsidered in the next training circle. To determine the values of thresholds, various approaches are possible. We introduce three approaches to determine the values of thresholds, which are given below. For any hidden neuron, $t_1^j > t_2^j > t_3^j$ due to $r_1^j < r_2^j < r_3^j$.

First Approach: Through the analysis above we have $t_1^j > t_2^j > t_3^j$. The simplest way to define t_1^j , t_2^j , and t_3^j is to use ξ_2 and ξ_3 as constant values and we have,

$$t_2^j = t_1^j - \xi_1, \quad (6.21)$$

$$t_3^j = t_2^j - \xi_2, \quad (6.22)$$

Second Approach: If we define the claim region by vertices with Hamming distance σ_1 from the match region, and the boundary region by vertices with Hamming distance σ_2 from the claim region. We assume every two bits of an input are orthogonal. For hidden neuron j , we define:

$$(r_2^j)^2 = (r_1^j)^2 + \sigma_1^2, \quad (6.23)$$

$$(r_3^j)^2 = (r_2^j)^2 + \sigma_2^2, \quad (6.24)$$

where σ_1 and σ_2 are the parameters to capture “geometric expansion”.

In terms of parameters σ_1 and σ_2 , we have Fig. 6.3 to visualize parameters r_2^j and r_3^j ,

Fig. 6.3 A choice of r_2^j and r_3^j for FCLA

According to (6.11), (6.23) and (6.24), we obtain equation to compute t_2^j and t_3^j as:

$$t_2^j = \min_{k=1}^{C_0^j} \sum_{i=1}^n w_{ij} x_i^k - C_0^j \sigma_1^2, \quad (6.25)$$

$$t_3^j = \min_{k=1}^{C_0^j} \sum_{i=1}^n w_{ij} x_i^k - C_0^j (\sigma_1^2 + \sigma_2^2). \quad (6.26)$$

Third Approach: Using a_1 , a_2 and gap , for the j th hidden neuron, we have,

$$t_2^j = t_1^j - a_1 \times gap, \quad (6.27)$$

$$t_3^j = t_2^j - a_2 \times gap, \quad (6.28)$$

where a_1 , a_2 are small integral values, and gap is a parameter to update t_2^j and t_3^j

Computing method of parameter gap is given in Fig. 6.4 in forms of pseudo-code.

```

int gap() // all parameters refer to hidden neuron j
{
//weight[i] is the connection weight  $w_i$ ;
    Temp1 = Min (Abs (weight[i]-weight[j]));
// which meets the condition Temp1>0; i, j = 1...InputDimension;
    Temp2 = Min (Abs (weight[i]));
// which meets the condition Temp2>0; i = 1...InputDimension;
    Temp = Min (Temp1, Temp2);
    Return (Temp);
}

```

Fig.6.4 Algorithm to compute the parameter gap .

The above three approaches to determine the thresholds t_2^j and t_3^j can be applied under different circumstances. Through our experiments, we observe the following. The first approach helps to balance the number of vertices represented by each hidden neuron, because the expansion process stops after a hidden neuron includes a certain number of

vertices. The second method is defined by Hamming distance, it performs well when solving “nicely clustered” problems (vertices with the same output have small Hamming distance while vertices with different output have large Hamming distance). The third approach aims at a slow expansion and high accuracy at the expense of comparably longer training time. In the third method, generally speaking, a large value of a_1 and a small value of a_2 causes fast learning but more errors, while a small value of a_1 and a large value of a_2 causes a precise learning with long training time. The parameters a_1 and a_2 are jointly considered for discussion. For a given problem, choice of these parameters gives our approach freedom to “adapt” it to suit the nature of the given problem.

6.3.3 Parameters Discussion for a_1 , a_2 , and gap in the Third Method

The third method discussed above has better performance and gives us more freedom in the training process. It is frequently used in our examples. Now we discuss the rationale for the link between parameters a_1 , a_2 , and gap . Firstly, the value of gap is computed through the algorithm in Fig. 6.4 on the fly. In other words, gap is the granularity decrease of converting one or two bits of the current hidden neuron. Our algorithm applies binary inputs and integer connection weights. We observe that the decrease of the weighted sum by converting one or two bits is not continuous, but in discrete steps (which we refer to as “granularity”). For example, if the current connection weight for a hidden neuron is $\{3, 3, -3, 1, -1, 3, 3\}$, and the activation threshold is 12, then the value of gap is the minimal nonzero value among the differences of any two weights, here $gap=2$. This

hidden neuron represents vertices $110 \begin{pmatrix} 00 \\ 10 \\ 11 \end{pmatrix} 11$. Next we seek the vertices nearest to the corresponding hypersphere. They are all vertices with weighed sum larger than or equal to $t_1^j - a_1 \times gap$. If $a_1=1$, then $t_1^j - a_1 \times gap=10$. The vertices meeting this condition are $\{1100111, 0101011, 1001011, 1111011, 1101001, 1101010\}$ except the three vertices having been represented by this hidden neuron. Any other vertices are farther than them to this hypersphere. An expansion caused by any of these vertices results in a subset which is linearly separable as well.

We set the initial values for the parameters a_1 and a_2 such that $a_1>0$ and $a_2>0$. When a_1 is larger than 1, a generation (expansion of a hypersphere) of a vertex may result in a representation involving some vertices not in the training set. During training, if some hypersphere does not expand in a whole training circle, we decrease a_2 (by value '1' in one iteration) for the hypersphere and the corresponding middle hypersphere expands by increasing a_1 by one on the fly. It is realized by $t_3^j = t_3^j + gap$ and $t_2^j = t_2^j - gap$. If some hypersphere does not expand in k continuous training circles, then $t_3^j = t_2^j - (a_2 - k) \times gap$ and $t_2^j = t_1^j - (a_1 + k) \times gap$. The parameters a_1 and a_2 therefore control the learning speeds and accuracy on the fly.

One conservative way is to set the initial value of a_1 as one and the initial value of a_2 as some large value. Anyway, a_1 increases and a_2 decreases on the fly. To do this, a little longer training time is needed than that with a large initial value of a_1 and a small initial value of a_2 .

Given a linearly separable subset, the values of w_{ij} , r_1^j , and t_1^j are fixed. The value of t_2^j depends on the required precision and the degree of generalization. The larger the required degree of precision, the larger t_2^j is, and vice versa. The value of t_3^j depends on the requirement of the neural network complexity and the learning speed. A fast and complex system determines a high value of t_3^j , and vice versa. The maximal value of the boundary radius r_3^j is $\sqrt{n}$, the diameter of RHS. And the minimal value of the match radius r_1^j is zero when hypersphere j only includes one vertex.

6.4 Training the Hidden Layer

6.4.1 Geometrical Expansion

Based on the analysis above, we obtain the geometrical expansion process of the hidden neuron.

The function for the j th hidden neuron is defined as:

$$f(\mathbf{w}^j, \mathbf{x}^k) = \sum_{i=1}^n w_{ij} x_i^k, \quad (6.29)$$

where $\mathbf{w}^j$ is $(w_{1j}, w_{2j}, \dots, w_{nj})$, the weight vector of hidden neuron j and $\mathbf{x}^k$ is $(x_1^k, x_2^k, \dots, x_n^k)^T$, the k th vertex.

Suppose there are K true samples in the true training set. The geometrical expansion process is shown in Fig. 6.5. As we have analyzed, in the geometrical expansion process, new data can be generalized to some hidden neuron through the hypersphere expansion.

In another words, the generalization function of FCLA is realized in the geometrical expansion process. This geometrical expansion process is also called the generalization process of FCLA by us. The hidden neurons created in this process are called generalization hidden neurons.

```

 $q=1$ ;
for ( $p=1, \dots$ , maximal interaction times) & if true sample set is not empty do
  for ( $k=1, \dots, K$ ) & if sample  $k$  is not removed
    for ( $j=1, \dots, q$ )
      test the  $j$  hidden neuron,
        if  $f(\mathbf{w}^j, \mathbf{x}^k) \geq t_1^j$ 
          ignore this sample, remove  $\mathbf{x}^k$  from the sample set, and go to
          test the next sample vector; jump out to check next input;
        if  $f(\mathbf{w}^j, \mathbf{x}^k) \geq t_2^j$ 
          expand hidden neuron  $j$  to include  $\mathbf{x}^k$ ; then remove  $\mathbf{x}^k$  from the
          sample and go to test the next sample vector; jump out to
          check next input;
      next  $j$ ;
    if there exists a  $j, f(\mathbf{w}^j, \mathbf{x}^k) \geq t_3^j$ 
      backup  $\mathbf{x}^k$  to be dealt with in the next training circle;
    if for all  $j$ 's,  $f(\mathbf{w}^j, \mathbf{x}^k) < t_3^j$ 
       $q=q+1$  create a new hidden neuron, neuron  $q$ , then remove from the
      sample  $\mathbf{x}^k$  from the training set;
    next  $k$ ;
    for ( $j=1, \dots, q$ )
      if in this training circle, hidden neuron  $j$  is not expanded
        shirk  $r_3^j$  and increase  $r_2^j$ ;
        revise  $t_2^j$  and  $t_3^j$ ;
      next  $j$ 
    next  $p$ 

```

Fig. 6.5 Generalization process of FCLA,

In Fig. 6.5, q stands for the current number of hidden neurons, $\mathbf{w}^j$ stands for the weight vector of the hidden neuron j , K is the total number of vertices in true subset and p stands for the number of iterations.

Select a sample $\mathbf{x}^k \in \text{true training set}$, and compute the value of $f(\mathbf{w}^j, \mathbf{x}^k)$.

If $f(\mathbf{w}^j, \mathbf{x}^k) \geq t_1^j$, it means that $\mathbf{x}^k$ has been covered by hypersphere j , and need not be learned again.

If $t_2^j \leq f(\mathbf{w}^j, \mathbf{x}^k) < t_1^j$, it means that $\mathbf{x}^k$ is not covered by hypersphere j , but can be included in it by an immediate expansion of hidden neuron j . Then we shift the center of hypersphere j to a proper position and expand it as big to exactly include $\mathbf{x}^k$.

If $f(\mathbf{w}^j, \mathbf{x}^k) < t_3^j$, $\mathbf{x}^k$ is too far from the cluster represented by hidden neuron j . If we expand hypersphere j to include $\mathbf{x}^k$, it may include much more other vertices between $\mathbf{x}^k$ and hypersphere j , which will cause over generalization. So if for all existing hidden neurons $f(\mathbf{w}^j, \mathbf{x}^k) < t_3^j$, a new hidden neuron is created to represent $\mathbf{x}^k$.

If $t_3^j \leq f(\mathbf{w}^j, \mathbf{x}^k) < t_2^j$, $\mathbf{x}^k$ is not so near as to be included in hidden neuron j immediately.

But it is possible to be included after some expansion to include some near vertices. If there exists a hypersphere (hidden neuron) which has $t_3^j \leq f(\mathbf{w}^j, \mathbf{x}^k) < t_2^j$, and is promising to include $\mathbf{x}^k$, then $\mathbf{x}^k$ will be left to be learned in the next training circle.

But if in one training circle, hidden neuron j does not expand, it is not expected to expand to include vertices within r_3^j in future training circles. In this case, r_3^j shrinks and r_2^j expands. It avoids some vertices trapped in boundary region j .

When $f(\mathbf{w}^j, \mathbf{x}^k) < t_3^j$ or $t_2^j \leq f(\mathbf{w}^j, \mathbf{x}^k) < t_1^j$, we need to modify the parameters of $\mathbf{w}^j$ and threshold as discussed below.

When $f(\mathbf{w}^j, \mathbf{x}^k) < t_3^j$, a new hidden neuron is created to represent the new coming vertex $\mathbf{x}^k$. The newly created hypersphere is centered at $\mathbf{x}^k = (x_1^k, x_2^k, \dots, x_n^k)$, and its radius is zero. This hypersphere (in fact a vertex $\mathbf{x}^k$) is large enough to include $\mathbf{x}^k$. The corresponding separating hyperplane passes $\mathbf{x}^k$. According to (6.12) the slope of this hyperplane $\mathbf{w}^j$ is $(2x_1^k - 1, 2x_2^k - 1, \dots, 2x_n^k - 1)$. And t_1^j is equal to $\sum_{i=1}^n x_i^k$.

If $t_2^j \leq f(\mathbf{w}^j, \mathbf{x}^k) < t_1^j$, we shift the center of hypersphere j to a proper position according to the position of $\mathbf{x}^k$ and expand it as much as just to include $\mathbf{x}^k$. The center of a hypersphere is always the gravity center of all the vertices included in that hypersphere. The radius is the Euclidean distance between the gravity center and the farthest vertex included in that hypersphere.

The flowchart of Generalization process for FCLA is shown in Fig. 6.6 and Fig. 6.7. In Fig. 6.6 and 6.7, q is the number of existing hidden neurons, j and g are index of hidden neurons, K is the number of vertices in the training set, and k is the index of vertices.

Fig. 6.6 Flowchart of generalization process for the hidden layer for FCLA

*¹Create a new hidden neuron: Set $q=q+1$, create hidden neuron q to represent this vertex using (6.17) and (6.18)

*²Expand the gth hidden: Train hidden neuron g using (6.19) and (6.20)

Fig. 6.7 Flow chart of one training circle for FCLA

6.4.2 Geometrical Modification

We call the training process described in Fig. 6.6 as the generalization process, and the hidden neuron created in the generalization process as generalization hidden neurons. In the generalization process, the generalization hidden neurons might generalize data which are not in the training set by geometrical expansion. However this generalization process might introduce some errors. Some generalization hidden neurons might represent some false vertices as well as true vertices. We should modify the hidden layer by adding some modification hidden neurons which represent some false vertices wrongly represented by the generalization hidden neurons.

The modification process follows the generalization process. First, in the generalization process, FCLA covers one class (say, the true class) by generalization hidden neurons. Then let all vertices belonging to the other class (say, the false class) go through the learning system. If the output is zero, then omit it. Else if the output is one, this false vertex has been wrongly generalized by some generalization hidden neuron in the generalization process. Then we will represent the wrongly represented vertices by additional hidden neurons. We connect these hidden neurons created in the modification process to the output neuron by negative connection weights to correct the vertices wrongly represent in the generalization process.

Fig. 6.8 visualizes the above modification process. A white vertex wrongly generalized to a generalization hidden neuron representing the black vertices is picked out by an

additional hidden neuron (a modification neuron) representing the white vertices. If this modification neuron fires, whether or not the generalization neuron fires, the output for the black class does not fire.

Fig.6.8 The modification process for FCLA

The training algorithm and the flow chart for the modification process of FCLA are shown in Fig.6.9 and Fig. 6.10. Suppose there are L false samples in the training set.

```

 $o=q+1$ ;
for ( $l=1, \dots, L$ )
  for ( $j=1, \dots, q$ )
    test the  $j$  hidden neuron,
    if  $f(\mathbf{w}^j, \mathbf{x}^k) \geq t_1^j$ 
      for ( $i=q+1, \dots, o$ )
        if for all  $i$ 's  $f(\mathbf{w}^j, \mathbf{x}^k) < t_1^j$ 
           $o=o+1$ , create a new hidden: neuron  $o$ ;
      next  $i$ ;
    next  $j$ ;
  next  $l$ .

```

Fig.6.9 Algorithm for modification process for FCLA

In Fig. 6.9, o stands for the current number of hidden neurons and w^j stands for the weight vector of hidden neuron j .

Fig. 6.10 Flowchart of modification process for FCLA

The equations to train the modification hidden neurons are the same as those in the generalization process:

$$w_{ij} = 2 \sum_{k=1}^{C_0^j} x_i^k - C_0^j. \quad (6.30)$$

$$t_1^j = \min_{k=1}^{C_0^j} \sum_{i=1}^n w_{ij} x_i^k. \quad (6.31)$$

t_2^j and t_3^j can be computed through different methods discussed in section 6.3.

In some cases, when the modification neurons are caused by some noisy data items located inside some generalization neuron, the modification stage might destroy the generalization advantage of FCLA. In this case, we can still avoid its negative affection by restricting the claim region of the modification neurons. However, in most cases, the modification neurons are located on the edge of some generalization neuron (including concave or hollow hyperspace), and the modification stage contributes to polishing the outskirts of some generalization neuron (created in the generalization process). This polishing increase the generalization capability of FCLA. Thus, generally speaking, the modification stage enhance the generalization advantage of FCLA.

6.5 Train the Output Layer

The output layer uses one output neuron to collect the outputs from the hidden neurons with connection weights w_j^o from the j th hidden neuron, and threshold t^o for the output neuron:

$$w_j^o = \begin{cases} 1 & \text{if } j = 1, \dots, q. \\ -q & \text{if } j = q + 1, \dots, m. \end{cases} \quad (6.32)$$

$$t^o = 1, \quad (6.33)$$

where q is the number of modification hidden neurons and m is the number of all hidden neurons.

6.6 Hard-Limiter Activation Function

The hard-limiter activation function for the j th hidden neuron can be expressed by:

$$h_j = \begin{cases} 1 & \text{if } \sum_{i=1}^n w_{ij} x_i - t_1^j \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (6.34)$$

And the hard-limiter activation function for the output neuron can be expressed by:

$$y = \begin{cases} 1 & \text{if } \sum_{j=1}^m w_j^o h_j - t^o \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (6.35)$$

If $y=1$, $\sum_{j=1}^m w_j^o h_j - t^o \geq 0$. That is at least one of the first q hidden neurons fires (output

1), and all the hidden neurons after q do not fire. $\mathbf{x}$ is represented by at least one of the generalization hidden neurons, and is not represented by any modification hidden neuron.

Then $\mathbf{x} \in \text{true subset}$.

If $\mathbf{x} \in \text{true subset}$, then $\mathbf{x}$ is represented by at least one of the generalization hidden neurons, and is not represented by any modification hidden neuron. Then at least one of the first q hidden neurons fires (output 1), and meanwhile all the hidden neurons after q

do not fire. Hence, $\sum_{j=1}^m w_j^o h_j - t^o \geq 0$, then $y=1$.

We can verify that, the values of connection weights and thresholds memorized by the neural networks are all integers.

6.7 Datasets and Performance Analysis

Before our discussion, it should be specified that all of the experimental work in this section is on P-II, 366 MHz machine, WinXP running VC++. The performance will increase much while our FCLA is running on an advanced computer system.

6.7.1 Supervised Learning

We first apply our algorithm FCLA to supervised learning. We illustrate the advantages of FCLA for supervised learning by approximation and classification problems. In order to make comparison, we also consider the example problems considered in previous study.

Example 6.1 Approximation of a circular region using 6-bit quantization (Example 5.2)

This problem has been referred as a benchmark example for MCL in chapter 5. We now solve it by using FCLA. This example is to separate a circular region in 2-D space which is a square with sides of length 8 and the coordinate origin in the lower left corner. A circle of diameter 4 is placed within the square, locating the center at (4, 4), and then the

space is sampled with 64 grad points located at the center of 64 identical squares covering the large square. 42 of these points fall outside of the circle (desired output 0), and 12 fall within the circle (desired output 1). These 12 points are {010011, 011011, 011010, 101011, 100011, 100010, 011101, 011100, 010100, 101100, 100101, 100100}. This problem has been visualized in Fig. 5.5 in chapter 5. We now cover the true vertices by the neural network using FCLA.

We use different input sequences to show that FCLA is not sensitive to the order of the input sequence. We first consider an arbitrary input sequence: {100100, 011100, 010011, 011010, 100010, 101011, 100011, 011101, 010100, 101100, 100101, 011011}.

We first apply FCLA to this problem by using the second method (using parameters σ_1 and σ_2) to control the geometrical expansion, where $\sigma_1 = \sigma_2 = 1$. The training process is shown in Table 6.1.

If we apply FCLA with the third method (using parameters a_1 and a_2) to control the geometrical expansion, where $a_1 = a_2 = 1$, the training process is shown in Table 6.2. The first column of Table 6.1 and 6.2 gives the sequentially input vertices. The second column represents the index of corresponding hidden neuron. The third column is the center of the corresponding separating hypersphere(s) obtained using (6.8). The connection weights w_i ($i=1 \dots 6$) in Table 6.1 and 6.2 are obtained using (6.17) and (6.19). The three thresholds in Table 6.1 are obtained using (6.18), (6.25) and (6.26). The three thresholds in Table 6.2 are obtained using (6.18), (6.27) and (6.28).

Table 6.1 The fast covering learning process of 6-bit circle problem by $\sigma_1 = \sigma_2 = 1$

Input	Neuron	Center	r_1	r_2	r_3	w_1	w_2	w_3	w_4	w_5	w_6	t_1	t_2	t_3
100100	1	(1,0,0,1,0,0)	0	1	$\sqrt{2}$	1	-1	-1	1	-1	-1	2	1	0
011100	2	(0,1,1,1,0,0)	0	1	$\sqrt{2}$	-1	1	1	1	-1	-1	3	2	1
010011	3	(0,1,0,0,1,1)	0	1	$\sqrt{2}$	-1	1	-1	-1	1	1	3	2	1
011010	Put aside.													
100010	Put aside.													
101011	4	(1,0,1,0,1,1)	0	1	$\sqrt{2}$	1	-1	1	-1	1	1	4	3	2
100011	4	(1,0,1/2,0,1,1)	1/2	$\sqrt{5}/2$	3/2	2	-2	0	-2	2	2	6	4	2
011101	2	(0,1,1,1,0,1/2)	1/2	$\sqrt{5}/2$	3/2	-2	2	2	2	-2	0	6	4	2
010100	2	(0,1,2/3,1,0,1/3)	$\sqrt{5}/3$	$\sqrt{14}/3$	$\sqrt{23}/3$	-3	3	1	3	-3	-1	6	3	0
101100	1	(1,0,1/2,1,0,0)	1/2	$\sqrt{5}/2$	3/2	2	-2	0	2	-2	-2	4	2	0
100101	1	(1,0,1/3,1,0,1/3)	$\sqrt{5}/3$	$\sqrt{14}/3$	$\sqrt{23}/3$	3	-3	-1	3	-3	-1	5	2	-1
011011	3	(0,1,1/2,0,1,1)	1/2	$\sqrt{5}/2$	3/2	-2	2	0	-2	2	2	6	4	2
011010	3	(0,1,2/3,0,1,2/3)	$\sqrt{5}/3$	$\sqrt{14}/3$	$\sqrt{23}/3$	-3	3	1	-3	3	1	7	4	1
100010	4	(1,0,1/3,0,1,2/3)	$\sqrt{5}/3$	$\sqrt{14}/3$	$\sqrt{23}/3$	3	-3	-1	-3	3	1	6	3	0

Table 6.2 The fast covering learning process of 6-bit circle problem by $a_1=1, a_2=1$

Input	Neuron	Center	w_1	w_2	w_3	w_4	w_5	w_6	t_1	t_2	t_3
100100	1	(1,0,0,1,0,0)	1	-1	-1	1	-1	-1	2	1	0
011100	2	(0,1,1,1,0,0)	-1	1	1	1	-1	-1	3	2	1
010011	3	(0,1,0,0,1,1)	-1	1	-1	-1	1	1	3	2	1
011010	Put aside.										
100010	Put aside.										
101011	4	(1,0,1,0,1,1)	1	-1	1	-1	1	1	4	3	2
100011	4	(1,0,1/2,0,1,1)	2	-2	0	-2	2	2	6	4	2
011101	2	(0,1,1,1,0,1/2)	-2	2	2	2	-2	0	6	4	2
010100	2	(0,1,2/3,1,0,1/3)	-3	3	1	3	-3	-1	6	5	4
101100	1	(1,0,1/2,1,0,0)	2	-2	0	2	-2	-2	4	2	0
100101	1	(1,0,1/3,1,0,1/3)	3	-3	-1	3	-3	-1	5	4	3
011011	3	(0,1,1/2,0,1,1)	-2	2	0	-2	2	2	6	4	2
011010	3	(0,1,2/3,0,1,2/3)	-3	3	1	-3	3	1	7	6	5
100010	4	(1,0,1/3,0,1,2/3)	3	-3	-1	-3	3	1	6	5	4

Table 6.1 and Table 6.2 are similar training processes. The difference is the criteria to expand hidden neurons. For this problem, although the expansion criterion is different, the resultant neural network is the same. In addition, the training process for Table 6.1 is the same as that for Table 6.2. Here we only explain the detailed training process through Table 6.2.

The first twelve rows in Table 6.2 correspond to 12 true vertices (say, $\mathbf{x}^1, \dots, \mathbf{x}^{12}$). The first column in these rows specifies these vertices. The vertex $\mathbf{x}^1 = 100100$ causes creation of hidden neuron 1, given in the first row, with weights $\mathbf{w}^1 = (1, -1, -1, 1, -1, -1)$ obtained using (6.17) and (6.19), and thresholds obtained using (6.18), (6.27) and (6.28). For the vertex $\mathbf{x}^2 = 011100$, $\mathbf{w}^1 \cdot \mathbf{x}^2 = -1$; so it is outside any region of interest for neuron 1, and causes neuron 2 to be created. Similarly, vertex $\mathbf{x}^3 = 010011$ causes neuron 3 to be created. Next, for vertex $\mathbf{x}^4 = 011010$, we have $\mathbf{w}^1 \cdot \mathbf{x}^4 = -3$, so it is outside any region of interest for neuron 1; however it is in the boundary region for both neurons 2 and 3, ($\mathbf{w}^2 \cdot \mathbf{x}^4 = 1$, $\mathbf{w}^3 \cdot \mathbf{x}^4 = 1$). Since none of the existing neurons can claim $\mathbf{x}^4$, we “put aside” this vertex for re-consideration in the next training circle. This is done by adding it in the shaded portion of Table 1, in row 13. Similarly, $\mathbf{x}^5 = 100010$ results in being put aside for re-consideration in the next training circle, thereby resulting in row 14. The vertices $\mathbf{x}^6, \dots, \mathbf{x}^{12}$ are in the claim region of one of the existing hidden neurons; so they cause parameter updating for the corresponding hidden neurons. After we finish these 12 vertices, we continue the training process of the two vertices put aside earlier. Neuron 3 and neuron 4 claim these vertices. Modified parameters (weights and threshold values) of these neurons are shown in rows 13 and 14.

After the generalization process, we let all false vertices go through the resultant neural network, and the outputs for these false vertices are all zeros. No hidden neuron is created in the modification process.

FCLA applies four hidden neurons by using both methods (as shown in Table 6.1 and Table 6.2).

Given any input sequence of these 12 true vertices, applying FCLA results in the same neural network structure with four hidden neurons. From Table 6.1 and Table 6.2 we observe that: given an arbitrary input sequence, FCLA results in a better neural network structure than ETL, which is the same as IETL does. The resultant neural network by ETL and IETL are not always satisfactory even after carefully consideration of the core vertex and the input sequence. Given a proper core vertex and input sequence, the simplest neural network obtained by ETL has five hidden neurons. In addition, in IETL which part of vertices is considered as “do not care” is not clear. Constructive Set Covering Learning Algorithm (CSCLA) only represented a limited number of vertices (maximally $n+1$, where n is the input dimension) by a hidden neuron. The best solution of MCL and MCETL [Wang 03a, 03b] to this example requires four hidden neurons. But how to determine the number of initial cores should also be considered. NETLA is based on the reduced form of Boolean expression, but it is not easy to reduce Boolean functions to their minimal expressions automatically. Hence it is not easy to program for IETL and NETLA. Thus, we do not consider IETL and NETLA for comparison in our research.

For different order of input sequences, FCLA constructs the similar neural network structure. For example, in solving this problem, FCLA always uses the same similar neural network structure with four hidden neurons given any input sequence of these twelve true vertices. To illustrate this advantage, we apply two arbitrary input sequences: one is {011011, 011010, 010011, 101011, 011101, 011100, 100011, 100010, 010100, 101100, 100101, 100100} used by Kim and Park [Kim 95], and the other is the sequence shown in Table 6.1 and Table 6.2. If we define the precision as the ratio of the number of vertices correctly represented by the neural network and the total number of vertices in the dataset, the comparison to CSCLA and ETL is shown in Fig. 6.11 to Fig. 6.14. 6.11 and 6.12 shows the result for the first input sequence. We observe that FCLA uses the simplest neural network structure and the least number of hidden neurons (four hidden neurons) to solve the problems of *Circular region representation using 6-bit space quantization*. FCLA and CSCLA are much faster (150 times faster) than ETL. FCLA requires a similar number of echoes with (exactly the same as or slightly more than) CSCLA, while constructs simpler neural network structure (with less hidden neurons) than CSCLA does. Referring to the input sequence in Table 6.1 and Table 6.2, we observe that FCLA also constructs simpler neural network (with four hidden neurons) than CSCLA (shown in Fig. 6.13) by using similar training time. FCLA applies fewer computing operations than ETL does in terms of $\sum_{i=1}^n w_i x_i$ (see in Fig. 6.14).

In summary, different input sequences result in quite different neural network structures by ETL and IETL. Given any input sequence, FCLA obtains the same neural network structure with four hidden neurons. In ETL and IETL, the core vertex and the input

sequence must be carefully determined before training, but there is no effective rule to determine the core vertex and input sequence yet. So FCLA has the best performance out of all learning algorithms discussed above.

Fig. 6.11 Hidden neurons for sequence 1 of example 6.1.

Fig. 6.13 Hidden neurons of sequence 2 of example 6.1.

Fig. 6.12 Operations for sequence 1 of example 6.1.

Fig. 6.14 Operations for sequence 2 of example 6.1.

Example 6.1 does not need the modification process during training. To illustrate the modification process of FCLA, we look into the following example.

Example 6.2 Considering a 3-bit Boolean function, we want to separate the true subset $\{000, 010, 011, 111\}$ from the false subset $\{001, 100, 101, 110\}$. We can begin with an arbitrary vertex in the true subset, and process an arbitrary sequence of true subsets or false subsets. We use the second method in FCLA by setting $\sigma_1 = \sigma_2 = 1$.

The learning process is shown in Table 6.3. The upper table shows the generalization process and the lower table shows the modification process.

In the generalization process, the first vertex, 111 causes to create hidden neuron 1; when 000 comes, it is located outside the boundary region 1, hence causes to create hidden neuron 2; when 010 comes, it is located in the claim region 2, hence causes the expansion of hypersphere 2; when 011 comes, it is located in the claim region 1, hence causes the expansion of hypersphere 1. All vertices in the true subset have been covered. In the modification process, the outputs of 001, 100, 101, 110 (false subset) are all zeros. So no modification hidden neuron is needed. Altogether 2 hidden neurons are needed.

If we consider a different sequence of the same training set {011, 000, 010, 111}. We can gain the result as shown in Table 6.4 (analysis is omitted). In this case, 3 hidden neurons are needed.

Table 6.3 The learning process of sequence 1 for example 6.2

Input	Hidden neuron	Center	$r1_j$	$r2_j$	$r3_j$	$w1_j$	$w2_j$	$w3_j$	$t1_j$	$t2_j$	$t3_j$
111	1	(1,1,1)	0	1	$\sqrt{2}$	1	1	1	3	2	1
000	2	(0,0,0)	0	1	$\sqrt{2}$	-1	-1	-1	0	-1	-2
010	2	(0,1/2,0)	1/2	$\sqrt{5}/2$	3/2	-2	0	-2	0	-4	$-4-2\sqrt{2}$
011	1	(1/2,1,1)	1/2	$\sqrt{5}/2$	3/2	0	2	2	4	0	$2\sqrt{2}$

Input	Model output	Hidden neuron	$w1_j$	$w2_j$	$w3_j$	$t1_j$
001	0 (ignore)					
100	0 (ignore)					
101	0 (ignore)					
110	0 (ignore)					

Table 6.4 The learning process of sequence 2 for example 6.2

Input	Hidden neuron	Center	$r1_j$	$R2_j$	$r3_j$	$w1_j$	$w2_j$	$w3_j$	$t1_j$	$t2_j$	$t3_j$
011	1	(0,1,1)	0	1	$\sqrt{2}$	-1	1	1	2	1	0
000	Backup										
010	1	(0,1,1/2)	1/2	$\sqrt{5}/2$	3/2	-2	2	0	2	-2	$-2-2\sqrt{2}$
111	1	(1/3,1,2/3)	$\sqrt{5}/3$	$\sqrt{14}/3$	$\sqrt{23}/3$	-1	3	1	3	$-2\sqrt{5}$	$-3-2\sqrt{10}$
000	1	(1/4,3/4,1/2)	$\sqrt{14}/4$	$\sqrt{30}/4$	$\sqrt{46}/4$	-2	2	0	0	$-4-2\sqrt{14}$	$-8-4\sqrt{7}$

Input	Model output	Hidden neuron	$w1_j$	$w2_j$	$w3_j$	$t1_j$
001	1	2	-1	-1	1	1
100	0 (ignore)					
101	0 (ignore)					
110	1	3	1	1	-1	2

From Table 6.3 and Table 6.4 we observe that: different input sequences result in similar neural network structures (in terms of similar number of hidden neurons). By using ETL and IETL different input sequences result in quite different neural network structures. The core vertex and the input sequence for ETL and IETL must be carefully determined before training. In ETL algorithm, three hidden neurons are needed for solving this example even after deep consideration, which is the same as the worst result of FCLA. The best solution of MCL and MCETL to this example requires two hidden neurons. But the number of initial cores have to be well determined. The Hamming distance between any two cores must be larger than two.

Example 6.3 A given 7-bit function [Kim 95]

Now we consider the given 7-bit function [Kim 95], which is also discussed in chapter 5. We apply ETL, CSCLA and FCLA to two different input sequences. We take the first sequence $S_1 = \{0000000, 0000001, 0000010, 0000100, 0001000, 0010000, 0100000, 1000000, 0000011, 0000101, 0001001, 0010001, 0100001, 1000001, 0010101, 0100010, 0100100, 0101000, 0110000, 1000101, 1010001, 1100000, 1100101, 1000010, 1000100, 1010100, 1100100, 1100110, 1101010, 1110010, 1010000, 1001000, 1011101, 1110101, 1010101, 1010111\}$, and second sequence $S_2 = \{0100001, 1010101, 0010000, 0100010, 1010000, 1011101, 0000001, 1000000, 1100100, 0001001, 1101010, 1001000, 0000011, 0010101, 1110010, 1010111, 0001000, 1000010, 0000101, 0101000, 1110101, 0100000, 1010001, 0100100, 0010001, 1010100, 0000010, 0000000, 0110000, 0000100, 1000100, 1100110, 1100000, 1000101, 1000001, 1100101\}$. We choose the parameters $\alpha_1=1$, and

$a_2=2$ in our FCLA formulation by using the third method to control the geometrical expansion process. For the first sequence, the methods FCLA, CSCLA and ETL need 5, 22, and 7 hidden neurons respectively (Fig. 6.15 and Fig 6.16). For the second sequence, FCLA, CSCLA and ETL use 7, 11 and 13 hidden neurons respectively (Fig. 6.17 and Fig 6.18). We gain the same conclusion from this problem that FCLA and CSCLA are much faster than ETL. FCLA uses less hidden neurons than CSCLA and ETL do.

Fig. 6.15 Hidden neurons for sequence 1 of example 6.3.

Fig. 6.17 Hidden neurons for sequence 2 of example 6.3.

Fig. 6.16 Operations for sequence 1 of example 6.3.

Fig. 6.18 Operations for sequence 2 of example 6.3.

Example 6.4 The n -bit parity function

The n -bit parity problem is a popular classification problem. The resulting network should generate a binary output one to an n -bit input with odd number of ones, and generate zero to an n -bit input with even number of ones. Many researchers tried to solve it based on back-propagation technique [Arulampalam 03, Lee 03]. Through their approaches, a large number of iterations and long training time are needed to solve this simple binary mapping problem.

Kim and Park applied ETL to solve n -bit parity problem by using n hidden neurons. Although FCLA and CSCLA are much faster than ETL, they need 2^{n-1} hidden neurons. For example, for 3-bit parity problem, ETL uses three hidden neurons, while both FCLA and CSCLA use four hidden neurons. Hence, for n -bit parity problems, ETL constructs a simpler neural network structure than FCLA and CSCLA do. In n -bit parity problem, two similar vertices (with Hamming distance one) always belong to two different classes. FCLA does not do so well in “poor clustered” problems such as n -bit parity problem.

We now compare FCLA with other approaches based on the gradient descent technique to solve the 5-bit parity problems [Lee, 2003] of Table 6.5 in order to illustrate the advantages of FCLA. These methods include: Error back-propagation using momentum (EBPM), Levenberg-Marquardt method, Lee's fast multi-layer perceptron training algorithm [Lee, 2003], and Error back-propagation using tunneling (EBPT). In FCLA we choose the parameters $a_1=1, a_2=1$ (for two vertices with Hamming distance one

always belong to two different classes). From Table 6.5, we observe that FCLA is much faster (several hundred times faster) than these gradient descent based algorithms, and meanwhile gains a better accuracy. In our experimental work, given different cores, we gain the same resultant neural network structure for this for 5-bit parity problem. Hence FCLA is preferred to gradient descent based algorithms to solve Boolean function mapping problem.

Table 6.5 FCLA & traditional algorithms: comparison for 5-bit parity problem.

	EBPM	LM	EBPT	Lee's (2003)	FCLA
Number of Epoch	9875	124	12657	443	2.25 (36 operations)
Time (s)	49.3	3.7	108.7	6.2	0.11
MSE	0.0114	0.0218	9.96E-6	4.37E-6	0
<p>NoE: number of Epochs EBPM: Error Back-Propagation using Momentum LM: Levenberg-Marquardt Method EBPT: Error Back-Propagation using Tunneling Lee's (training algorithm): Global Optimization for fast Multilayer Perceptron Training (2003)</p> <p>One operation is defined as one computation of $\sum_{i=1}^n w_{ij} x_i^k$ for $\mathbf{x}^k$</p>					

Example 6.5 Iris dataset classification using FCLA [Arulampalam 03, Lee 03]

We now apply our algorithm FCLA to a classical dataset, iris for classification. There are 150 items in the dataset with three target classes: Setosa, Versicolor and Verginica (50 items in each). Each of the items has four real-valued features. They are length of sepal, length of petal, width of sepal and width of petal.

FCLA is designed for BNNs. We first represent these four features by 55 bits, which is 55 binary inputs. These 55 bits can represent the four numerical features without information losing. In order to retain the ordinal property of continuous variables, we use thermometrical encoding. For example, we use 000, 001, 011, 111 represent 0, 1, 2, 3 respectively. We construct a neural network with one hidden layer and one output layer. There are two binary output neurons (y_1, y_2) in the output layer. The first output y_1 represents the class Setosa, and the second output y_2 represents the class Versicolor. Items neither belong to Setosa nor belong to Versicolor are considered as Verginica. Given an input, if the first output y_1 fires, then we do not consider the second output y_2 (in this case, $(y_1y_2) \in \{10, 11\}$), and this item belongs to Setosa. If the first output y_1 does not fire and the second output y_2 fires (in this case, $(y_1y_2) = (01)$), then this item belongs to Versicolor. Else if both outputs do not fire (in this case, $(y_1y_2) = (00)$), then this item belongs to Verginica.

To demonstrate the features of FCLA, we consider two cases. In the training process of both cases, we apply the third method (a_1 and a_2) to control the geometrical expansion process of the hidden neurons. We apply (6.18), (6.27) and (6.28) for obtaining three threshold values by setting $a_1=2$ and $a_2=6$.

Case 1. The training dataset contains 150 items (the whole Iris set), and we use the same dataset for testing. Table 6.6 compares the performance of FCLA with other training algorithms based on gradient descent technique. From the results of the second last column in Table 6.6, we conclude that FCLA uses a less number of operations. It needs

less training time, and results in zero error. FCLA has the best performance in terms of training speed (fifteen times to several hundred times faster than other algorithms).

Case 2. We randomly choose half of the iris dataset (75 items) as the training set, and let the remaining be the testing dataset. From the results of the last column in Table 6.6, we conclude that FCLA needs a less number of operations. It requires less training time, and results in 2.6% error, which is much less than EBPM and LM do (the second and the third column in Table 6.6). Furthermore, in this case, half of the dataset is used for training and the remaining is used for testing in our method. And the result (reported in the last column) refers to the training dataset. But other methods use the whole dataset for training. Hence FCLA has more general interest for real world applications.

Table 6.6 Performance comparison for iris data classification

	EBPM	LM	EBPT	Lee's (2003)	FCLA (150 training items)	FCLA (75 training items)
Number of Epochs (NoE)	10776	1278	36017	1584	2.17 (326 operations)	3.05 (229 operations)
Time (sec.)	90.8	16.2	287.4	23.6	1.0325	0.898
Mean Square Error (MSE)	0.2086	0.1428	9.84E-6	4.046E-6	0	0.026

In FCLA, the number of epochs (NoE) [Arulampalam 03, Lee 03], the training time, and the error depends on different values of parameters a_1 and a_2 . Using iris dataset and our formulation above, we give such a comparison in Table 6.7.

From Table 6.7, generally speaking, a large value of a_1 and a small value of a_2 cause fast learning but more errors, while a small value of a_1 and a large value of a_2 cause an accurate learning with a little longer training time. With the increase of the training set, the precision of both the training set and the testing set increases. We can always gain a 100 percent precision given proper parameters of $a_1=2$ and $a_2=6$, so the modification process is not applied. We need no modification hidden neurons in solving this problem.

Table 6.7 Parameters Analysis for a_1 and a_2 in FCLA

a_1	a_2	Training data/ testing data	NoE	Training operations	Errors for training set	Errors for testing set	Training time
2	6	50/100	4.58	229	0	0.04	0.898
2	4	50/100	4.08	204	0	0.05	0.873
3	4	50/100	3.66	183	0.02	0.08	0.867
2	6	70/150	3.47	243	0	0	0.913

Now we use FCLA to solve a real world Boolean problem and illustrate the function of modification process during training by Example 6.6.

Example 6.6 We now apply FCLA to a dataset including votes for each of the U.S. House of Representatives Congressmen [URL-Datasets] on the 16 key votes identified by the CQA (Congressional Quarterly Almanac). 16 attributes are used for each person, all of which are Boolean values (yes or no). Each person either belongs to democrat or republican. There are 435 (267 democrats, 168 republicans) items in the dataset. Many values in the dataset are missing. Using these 16 attributes, we classify the items in the dataset by party affiliation (democrat or republican). We use data without missing values

for training, and the whole dataset for testing. We apply generalization process by covering data whose party affiliation are republicans using fifteen hidden neurons. Five items are classified incorrectly because in the process of covering republicans by generalization, five items belonging to democrats are wrongly generalized. While we continue the training process by modifying the wrongly represented instances, four more hidden neurons (modification hidden neurons) are needed. FCLA can gain a 100% precision for the training set. The modification process also reduces errors for the testing set. From Table 6.8, we gain that FCLA is better than traditional training algorithms. FCLA increases the precision for both the training set and the testing set. The training time used by traditional algorithms do not available from the literatures. However, from previous examples, FCLA is faster (about one hundred times faster) than traditional algorithms. Hence, we do not discuss the training speed for this example.

Table 6.8 Performance comparison for example 6.6

	Previous result [16]	FCLA (without modification process)	FCLA (with modification process)
Training Precision	90-95%	97.8%	100%
Testing precision		95.1%	96.6%

6.7.2 Unsupervised Learning

FCLA can also be used to unsupervised learning for Boolean neural network construction.

FCLA constructs a two-layered neural network with one input layer and one output layer.

Neurons (output neurons for unsupervised learning or hidden neurons for supervised learning) collect samples with small value of distance in clusters. Each neuron (for unsupervised learning) itself in the output layer hence represents a cluster. However, for supervised learning, another layer (the output layer for supervised learning) is needed to collect the output from these neurons (hidden neurons for supervised learning) to give some meaningful result. The training process for the output layer for unsupervised learning is exactly the same as the training process for the hidden layer for supervised learning by using FCLA. We now illustrate the implementation of FCLA for unsupervised learning by clustering problems.

We begin with a random distribution having unique probability on 10-dimensional binary sample vectors [Chaudhari 03]. Then we go on with a real world clustering problem with Boolean features [Wang 04b].

Example 6.7 We generate 100 out of 1024 samples in the variable space with 10-bit variables. We apply the first method (ξ_2 and ξ_3) to control the geometrical expansion process using (6.21) and (6.22). We carry out the experimental work by using two pairs of parameters values and compare their results. We first set $\xi_1=5$ and $\xi_2=6$. Fourteen hidden neurons are needed. The report is shown in Table 6.9. When we set $\xi_1=6$, $\xi_2=7$, eight hidden neurons are needed. The report is shown in Table 6.10. We test the result by the remaining samples (924 out of 1024). We find that similar vertices always go to the same cluster. When ξ_1 and ξ_2 are small values, there are still some uncovered samples while when ξ_1 and ξ_2 are large values there is little overlap between different clusters.

Table 6.9 Result for example 6.7 ($\xi_1=5$, $\xi_2=6$).

Neuron j	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	w_{7j}	w_{8j}	w_{9j}	w_{10j}	t_{1j}
$j=1$	-4	-8	-10	-4	-4	-6	6	-4	-2	10	8
$j=2$	1	-7	7	-1	3	1	1	9	8	-9	27
$j=3$	-2	6	6	0	2	8	-8	-8	0	2	22
$j=4$	6	-8	2	-6	-2	6	2	-2	6	8	24
$j=5$	-5	7	-9	-7	1	-1	-1	7	-3	-9	7
$j=6$	1	5	-3	-5	-5	3	-3	-3	7	-5	11
$j=7$	0	-4	-4	4	4	6	6	2	-6	2	20
$j=8$	4	2	4	4	2	4	-2	4	-2	-2	22
$j=9$	3	5	3	5	-3	-5	7	-5	-3	5	20
$j=10$	7	-3	-3	7	-1	-7	-5	-1	-3	-1	11
$j=11$	-5	1	1	-5	5	1	-3	-3	3	1	9
$j=12$	-1	1	3	-3	-1	-1	1	5	-5	1	9
$j=13$	3	-3	-3	1	-3	5	-3	-3	-3	-5	5
$j=14$	-2	0	-2	2	-2	0	2	2	2	0	8

Table 6.10 Result for example 6.7 ($\xi_1=6$, $\xi_2=7$).

Neuron j	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	w_{7j}	w_{8j}	w_{9j}	w_{10j}	t_{1j}
$j=1$	-2	-12	-2	-6	-2	-6	6	-8	-6	16	6
$j=2$	2	-2	10	2	-2	2	0	12	0	-12	27
$j=3$	-3	9	5	-3	1	5	-3	-11	7	5	20
$j=4$	-2	6	-10	-8	-2	-2	0	8	-1	-8	6
$j=5$	0	-2	-6	2	-8	0	4	8	4	6	22
$j=6$	4	-2	-6	6	-2	2	2	-6	-6	-6	10
$j=7$	-1	1	1	3	3	3	1	3	-3	3	17
$j=8$	1	-3	-1	-1	1	1	-5	-1	3	-3	4

Now we apply (6.23) and (6.24) by setting $\sigma_1 = \sigma_2 = 1$ to solve the same problem. The result is reported in Table 6.11. Ten hidden neurons are needed here. From Table 6.11, we observe that the values of parameters in Table 6.11 are smaller than those in Table 6.9 and Table 6.10. But 6 hidden neurons (1, 2, 3, 5, 8, 9) only represent one vertex each. It is because the increase rate of the claim radius is too fast. In this case, the more vertices a

hidden neuron represents, the more it intends to expand. While if we fix the bias ξ_1 and ξ_2 , the result is reversed. The more vertices a hidden neuron represents, the less it intends to expand. The first method (using ξ_1 and ξ_2) for geometrical expansion is preferred if we want a more even distribution of different clusters.

Table 6.11 Result for example 6.7 ($\sigma_1 = \sigma_2 = 1$)

Neuron j	w_{1j}	w_{2j}	w_{3j}	w_{4j}	w_{5j}	w_{6j}	w_{7j}	w_{8j}	w_{9j}	w_{10j}	t_{1j}
$j=1$	-1	-1	-1	-1	1	-1	1	-1	-1	1	3
$j=2$	-1	-1	1	-1	1	1	1	1	1	-1	6
$j=3$	1	1	1	-1	-1	-1	-1	1	-1	-1	4
$j=4$	-3	3	-1	3	-1	-5	1	1	-1	1	3
$j=5$	-1	1	-1	1	1	-1	1	1	-1	-1	5
$j=6$	-1	-1	3	3	-1	3	-1	5	-5	-3	-2
$j=7$	-2	2	-2	0	-2	2	-2	-2	2	-2	6
$j=8$	1	-1	-1	1	-1	-1	1	-1	-1	1	4
$j=9$	-1	1	1	1	1	1	-1	-1	-1	1	6
$j=10$	3	-3	1	-5	3	3	3	-1	5	5	7

We now apply FCLA to a real world clustering problems to show its advantages.

Example 6.8 This is a dataset containing 101 instances [URL-Datasets]. Each instance has 15 Boolean-valued attributes (hair, feathers, egg, milk, airborne, aquatic, predator, toothed, backbone, breathes, venomous, fins, tail, domestic, catsize) and one numeric attribute ($\text{legs} \in \{0,2,4,5,6,8\}$). We represent this numeric attribute by 6 bits, each of which claims one value. That is, $\{000001, 000010, 000100, 001000, 010000, 100000\}$ represent $\{0,2,4,5,6,8\}$ respectively. We have no pre-knowledge about the data. We apply the third method by setting $a_1=1$, $a_2=20$ to control the geometrical expansion process.

The clustering process is very fast and takes about half a second on a P-II, 366 MHz machine. After training, all instances are clustered into 7 classes. It is observed that similar animals go to the same cluster. The comparison between the clustering by FCLA and a referred breakdown (7 clusters) by biology experts [URL-Datasets] is shown in Table 6.12.

The clustering by FCLA is not exactly the same as that given by the biology experts. However, it supposed we have no pre-knowledge about the dataset (the target value of the input patterns and the number of the target clusters), and the result is satisfactory. As shown in Table 6.12, several animals clustered by FCLA are different from those by the experts. For example, *porpoise* and *dolphin* are clustered with terrestrial animals by the biology experts because they belong to mammal. But they are clustered with other oceanic animals by FCLA because of their same value of the attributes with the oceanic animals, such as hair, feathers, aquatic, fins, tail, leg, etc. *Porpoise* and *dolphin* look like oceanic animals. *Pitviper*, *slowworm* and *tuatara* are mixed up with *Toad*, *newt*, and *frog* also because of their same values of attributes. These animals are all amphibians. But *Pitviper*, *slowworm*, and *tuatara* are considered different from *Toad*, *newt* and *frog* by the biology experts, due to their weight.

Although FCLA is not as smart as the biology experts, it demonstrates the ability to cluster data in a similar way with human beings. It supposed to need no pre-knowledge.

Table 6.12 Clustering of zoo dataset from Richard forsyth

Cluster index	Clustering by biology experts	Neuron index	Clustering by FCLA
1	Opossum, oryx, platypus, Polecat, pony, <i>porpoise</i> , puma, pussycat, raccoon, Reindeer, seal, Sealion, Squirrel, Vampire, vole, wallaby, wolf, mongoose, mole, mole, lynx, lion, leopard, hare, hamster, gorilla, goat, girl, giraffe, fruitbat, elephant, <i>dolphin</i> , deer, cheetah, cavy, calf, buffalo, boar, bear, antelope, aardvark	1&6	Opossum, oryx, platypus, Polecat, pony, puma, pussycat, raccoon, Reindeer, vole, Wolf, mongoose, mole, mole, lynx, lion, Leopard, hare, goat, giraffe, elephant, deer, cheetah, calf, buffalo, boar, bear, antelope, aardvark
		1	Cavy, hamster, wallaby,
		6	Seal, Sealion, Squirrel, Vampire, gorilla, girl, fruitbat,
2	Parakeet, penguin, pheasant, rhea, skimmer, Skua, sparrow, swan, vulture, wren, ostrich, Lark, kiwi, hawk, gull, flamingo, duck, dove, crow, chicken,	4	Parakeet, penguin, pheasant, rhea, skimmer, Skua, sparrow, swan, vulture, wren, ostrich, lark, kiwi, hawk, gull, flamingo, duck, dove, crow, chicken,
3	<i>Seasnake</i> , tortoise, <i>pitviper</i> , <i>slowworm</i> , <i>tuatara</i> ,	3	<i>Scorpion</i> , tortoise,
4	Pike, piranha, Seahorse, sole, stingray, Tuna, herring, haddock, dogfish, chub, catfish, carp, bass,	2	Pike, piranha, <i>porpoise</i> , Seahorse, <i>seasnake</i> , sole, stingray, Tuna, herring, haddock, <i>dolphin</i> , dogfish, chub, catfish, carp, bass,
5	Toad, newt, frog,	7	Toad, newt, <i>pitviper</i> , frog, <i>slowworm</i> , <i>tuatara</i> ,
6	Termite, wasp, moth, Ladybird, housefly, honeybee, gnat, flea,	3	Termite, wasp, moth, Ladybird, housefly, honeybee, gnat, flea,
7	<i>Scorpion</i> , Seawasp, slug, Starfish, lobster, worm, crayfish, crab, clam,	5	Seawasp, Starfish, crab,
		3&5	Clam, crayfish, worm, lobster, slug,

6.8 Concluding Remarks for FCLA

Fast Covering Learning Algorithm (FCLA) for Boolean neural networks is based on multi-level geometrical expansion. Two learning processes are involved: the generalization process and the modification process. Three regions are defined by geometrical concept. The learning process is based on the judgment of which region the new coming vertex belongs to. These three regions (match, claim, and boundary) have the same center, but a different radius. The match region and the claim region shift and expand in the training process until all vertices of the true (or false) subset are covered. Then to avoid over-generalization, modification process is designed to correct the errors introduced in the generalization process. Our construction process proceeds as follows. A hidden neuron first includes one vertex in it, and then “expands” using one of the three methods we have discussed in section 6.3.2. The expansion process is characterized by a suitable choice of parameters such as ξ_1 and ξ_2 , σ_1 and σ_2 , or a_1 , a_2 , and “gap” depending on the method to control the geometrical expansion process. If we set these parameters to be large values to include vertices with a larger distance, the hidden neuron can expand easily, but a vertex with a small distance will be wrongly included. We either introduce more errors (which is the cost we have to pay for having more generalization capability within the construction process of neural networks), or choose small values of these parameters such that not many errors are introduced, but long training time is needed.

FCLA can be applied to both supervised learning and unsupervised learning. FCLA constructs a three-layered neural network with one input layer, one hidden layer and one

output layer for supervised learning, while it always constructs a two-layered neural network with one input layer and one output layer for unsupervised learning. Each output neuron for unsupervised learning represents one cluster. The training process for the output layer for unsupervised learning is the same as the training process for the hidden layer for supervised learning.

The main advantages of FCLA over existing algorithms are: our algorithm possesses the capability of generalization, but all previous learning algorithms for BNNs (except for BLTA) do not. FCLA generalize near vertices by expanding hyperspheres to include farther vertices. In FCLA, we do not need assign the core vertex in advance, which makes our algorithm to be applicable more generally. FCLA is faster because we need not search a lot of training pairs for determining each hidden neuron. What FCLA does is to judge which region an input belongs to. FCLA is not based on gradient descent technique either. FCLA constructs simpler neural networks as well as gains satisfactory performance (training speed and accuracy). The last and the most important features of FCLA is that it is not sensitive to the change of input sequences of the same training set.

In summary, FCLA has a better performance than traditional algorithms based on gradient descent technique in terms of both training speed (about one hundred times faster) and accuracy through computer simulations. FCLA is preferred to previous training algorithms for BNNs (BLTA, ETL, IETL, CSCLA, MCL and MCETL) as well.

Chapter 7

Fast Covering Learning Algorithm for Real Valued Problems (FCLAR)

7.1 Introduction

Usually, input variables break down into two types: ordinal variables and non-ordinal variables. BNNs are preferred models to solve problems with non-ordinal variables, especially with Boolean (true or false) variables. In this chapter, we investigate the problems with ordinal variables.

Almost all current training algorithms for neural networks are based on the gradient descent technique, which causes iterative computing and long training time. For example, Error Back-Propagation (EBP) is a popular training algorithm, but it is based on gradient

descent and needs high number of iterations for training.

In the previous chapter, we have proposed Fast Covering Learning Algorithm (FCLA) for Boolean neural networks based on geometrical expansion. The FCLA avoids iterative computing and is much faster than gradient descent based algorithms. Although it gains good performance for Boolean neural networks, it does not cover problems with real-valued inputs.

This chapter presents an extended work of FCLA for Boolean neural network construction based on geometrical expansion [Wang 05]. We extend FCLA for real-valued applications. We called our improved algorithm FCLAR.

In our method, parameters are updated according to the geometrical location of the training samples in the input space, and each sample in the training set is learned only once. By doing this, for real-valued applications, FCLAR is able to avoid iterative computing and much faster than traditional training algorithms. Given an input sequence in an arbitrary order, FCLAR learns ‘easy’ samples (vertices near to the center of some hidden neuron) first and ‘confusing’ (vertices far from the center of some hidden neuron) samples are easily learned after these ‘easy’ samples. This sample reordering process is done on the fly based on geometrical concept.

In addition, traditional algorithms fix the neural network structure (the number of hidden layers and the number of hidden neurons in each hidden layer) before training. However, it is difficult to determine a proper structure in advance which can both guarantee convergence and avoid over fitting. Hence more and more researchers focus their study on constructive learning [Kim 95, Campbell 97, Er 03, Kwok 97]. Constructive learning

begins with a minimal (or empty) structure, and dramatically increases the network by adding hidden neurons until a satisfactory solution is found. The FCLAR begins with an empty hidden layer, and adds new hidden neurons when necessary. This constructive learning avoids blind selection of neural network structure.

The FCLAR can be used for both supervised learning and unsupervised learning to solve real valued problems. The experimental work for classification and clustering problems illustrates the advantages of FCLAR in both learning modes, especially in learning speed.

Some properties of FCLAR are:

- (i) *Fast*: Each sample in the training set is learned only once. Parameters are updated according to the location of the trained sample in the input space. By doing this, FCLAR is able to avoid iterative computing and much faster than traditional training algorithms without decreasing the learning performance (for example, classification accuracy).
- (ii) *Reordering*: Given an input sequence in an arbitrary order, FCLAR learns ‘easy’ samples first. The ‘confusing’ samples are learned after these ‘easy’ samples. This is a sample reordering process which is done on the fly based on geometrical analysis.
- (iii) *Constructive learning*: FCLAR begins with an empty hidden layer, and adds new hidden neurons or expands the existing hidden neurons according to the location of the sequentially coming data. This learning process avoids selecting the neural network structure blindly.
- (iv) *Geometrical expansion*: We visualize the hidden neurons in terms of hyperspheres. To expand a hypersphere, three different radii are introduced. The construction process makes use of three concentric hyperspheres based on these

three radii.

- (v) *Brain-like learning*: In a brain-like neural network, only part of the hidden neurons respond to some areas of the inputs. FCLAR has localization learning function. In each learning step of FCLAR, only the parameters of the selected neuron are updated, and other neurons remain unchanged. In addition, whether neurons should fire is based on the similarity between the input data and the center of the corresponding hidden neuron. Human beings learn based on the similarities between real objects. Similarities are usually defined as a distance function. FCLAR constructs neural networks based on geometrical distance of the training samples in the input space.

7.2 Geometrical Analysis

Before we explain our method, we analyze the geometrical properties of the variable space. As we have discussed in previous chapters, when solving problems with Boolean inputs (only zero or one), it is easy to visualize the input space as a unit hypercube. All input patterns are located on the vertices of this unit hypercube, hence on the surface of the exhypersphere of that unit hypercube. For Boolean (i.e., 0 or 1) values, the expansion process of a hypersphere can be expressed by a linear function. But when we come to problems with real-valued inputs, the patterns (input vectors) are not located on the surface of a unit hypercube, but in some hypercube. In this case, we cannot represent the expansion process by a linear function.

Consider a mapping: $y=f(\mathbf{x})=f(x_1, x_2, \dots, x_n)$. Assuming n is the dimension of the input vectors, the input space is considered as an n -dimensional hyperspace. Before training, we first normalize the inputs so that $0 \leq x_i \leq 1, i=1 \dots n$. For a training sample $\mathbf{x}^k=(x_1^k,$

$$x_2^k, \dots, x_n^k)$$

$$x_i^k = (x_i^k - \min(x_i)) / (\max(x_i) - \min(x_i)), \quad (7.1)$$

where $i=1 \dots n$, $\max(x_i)$ and $\min(x_i)$ are the maximal and minimal values of the i th feature in the input space. After normalization, all input patterns are located within a unit hypercube (Fig. 7.1).

Fig. 7.1 Visualization of real-valued input space after data normalization

Hence, using the normalized dataset, we construct neural networks based on their geometrical concept. We visualize hidden neurons in terms of localized functions: hypersphere activation functions.

7.3 Neural Network Architectures

FCLAR constructs a three-layered feed forward neural network with one input layer, one hidden layer and one output layer for supervised learning (as shown in Fig. 7.2), and a two-layered feed forward neural network with one input layer and one output layer for unsupervised learning (as shown in Fig. 7.3).

For unsupervised learning, the training process for the output layer is exactly the same

training process for the hidden layer for supervised learning. There are two layers, the input layer and the output layer in unsupervised learning in our method. The following discussion refers to supervised learning.

Fig. 7.2 Neural network structure for supervised learning

Fig. 7.3 Neural network structure for unsupervised learning

7.3.1 The Hidden Layer

A hidden neuron in Fig. 7.2 or an output neuron in Fig. 7.3 represents a corresponding hypersphere with center $\mathbf{c}$ and radius r_1 . While constructing a neuron, suppose that $\{\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^v\}$ are v samples included in one hypersphere (hidden neuron). In terms of these samples, the center is defined as the gravity center $\mathbf{c} = (c_1, c_2, \dots, c_n)$:

$$c_i = \sum_{k=1}^v x_i^k / v. \quad (7.2)$$

The radius r_1 is defined as the minimal Euclidean distance such that all these v vertices are exactly in or on the surface of the corresponding hypersphere.

$$r_1 = \min_{j=1}^v \|x^j - c\| = \min_{j=1}^v \left(\sum_{i=1}^n (x_i^j - c_i)^2 \right)^{1/2}, \quad (7.3)$$

where n is the dimension of the inputs, and $\|\cdot\|$ is the Euclidean distance.

Given c and r_1 we can separate these v true samples from the remaining samples. In another words, this hypersphere represents these v true samples. In order to get good representation ability of a hidden neuron, these v true samples should be a compact cluster. Two secondary central radii r_2 and r_3 are introduced to find compact clusters, where $r_1 < r_2 < r_3$.

7.3.2 The Output Layer

For supervised learning, the output layer has neurons to collect the outputs from the hidden layer. For example, for two-classification problems, one output neuron is enough to collect outputs from all hidden neurons with weight one and threshold one. Firing of one hidden neuron triggers firing of the output neuron. For multiple classification problems, the number of output neurons is equal to the number of classes minus one. Assuming we have m classes, $m-1$ output neurons are needed. Each output neuron collects the outputs from its corresponding hidden neurons representing its class. The value of connection weights is one and the value of threshold is one. There is no connection from hidden neurons representing other classes. The output neuron fires if one or more of its offspring hidden neurons fire. If all $m-1$ output neurons do not fire, then the testing pattern does not belong to the first $m-1$ classes, and is hence considered to be a member from the m th class. For unsupervised learning, the training process for the output layer is exactly the same as that for the hidden layer for supervised learning.

7.3.3 Activation Functions

Activation function for the hidden layer is defined as:

$$h_j = \begin{cases} 1 & \text{if } \|\mathbf{x} - \mathbf{c}^j\| \geq r \\ 0 & \text{otherwise} \end{cases}, \quad (7.4)$$

where h_j and $\mathbf{c}^j$ are the output and the center of the j th hidden neuron, and r^j is the activation radius of the j th hidden neuron, whose value equals to r_1^j of the corresponding hidden neuron.

Activation function for the output layer is defined as:

$$o_j = \begin{cases} 1 & \text{if } \sum_{i=1}^m h_i \geq 0 \\ 0 & \text{otherwise} \end{cases}, \quad (7.5)$$

where o_j is the output of the j th output neuron.

7.4 Parameter Discussion

FCLAR begins with an empty hidden layer (no hidden neuron). In the training process, the training samples are input to the learning system sequentially in an arbitrary order. FCLAR either learns a sample or back it up to be learned in the next training circle according to the sample's location. To learn a sample, FCLAR either creates a new hidden neuron to represent it or updates the parameters of some hidden neuron by expanding its corresponding hypersphere. We discuss the parameters for the hidden layer for supervised learning (the same for the output layer for unsupervised learning) in the following two subsections. We do not discuss parameters for the output layer for supervised learning because all the values of these parameters are zero or one according to subsection 7.3.2.

7.4.1 Primary Parameters: c and r_1

If a new hidden neuron is created, only one sample is represented by that hidden neuron. According to (7.2), the center $c = (c_1, c_2, \dots, c_n)$ is defined as the location of the corresponding sample,

$$c_i = x_i. \quad (i=1 \dots n) \quad (7.6)$$

According to (7.3), $r_1=0$. But, to start the training process, it is more reasonable to set the initial value of r_1 as a small value in terms of the average distance of the training data using (7.7).

$$r_1 = \lambda \times \left(\frac{V}{N}\right)^{1/n}, \quad (7.7)$$

where V is the volume of the input space, N is the number of samples and $0 < \lambda < 1$. We set $\lambda = 0.5$.

If a hypersphere (hidden neuron) is expanded, the center is adjusted using (7.2). The training samples are input to the learning system sequentially. The center of previous $v-1$ samples has been known. In order to avoid redundant computation, we adjust the center as:

$$c_i^{new} = \frac{c_i^{old} \times (v-1)}{v} + \frac{x_i^v}{v}, \quad (7.8)$$

where v is the number of samples represented by that hidden neuron after expansion, and x_i^v is the i th feature of the current trained sample. c_i^{old} is the i th feature of the center before hypersphere expansion, and c_i^{new} is the i th feature of the center after expansion.

To compute r_1 when expanding a hypersphere (hidden neuron), one constraint should be satisfied: the hypersphere after center adjustment and expansion must include all those

previously learned samples as well as the current trained sample: $\mathbf{x}^{current}$. According to the definition (7.3), the Euclidean distances from the new center to each sample represented by this hidden neuron should be computed. This iterative computing is time consuming. We can compute r_1 by using (7.9) and guarantee that the updated hidden neuron is able to represent the current trained sample without forgetting previous ones.

$$r_1^{new} = \max\{r_1^{old} + \|\mathbf{c}^{new} - \mathbf{c}^{old}\|, \|\mathbf{c}^{new} - \mathbf{x}^{current}\|\}. \quad (7.9)$$

7.4.2 Secondary Parameters: r_2 and r_3

The r_2 and r_3 are secondary parameters to control the hypersphere expansion process. The r_2 and r_3 are initially defined as (7.10) and (7.11), where r_2 and r_3 have the same center $\mathbf{c}$ with r_1 , and $r_1 < r_2 < r_3$:

$$r_2 = r_1 + \alpha \times \mu, \quad (7.10)$$

$$r_3 = r_1 + \beta \times \mu, \quad (7.11)$$

where α , β and μ are parameters to compute r_2 and r_3 .

The r_2 and r_3 are two important parameters to control the geometrical expansion process. The r_2 defines a trigger region to start a geometrical expansion, and r_3 is a restriction to stop a geometrical expansion. Given a set of samples, r_1 is fixed. Determining r_2 and r_3 equals to determining α , β and μ in (7.10) and (7.11).

The μ is a bias, which is considered as a unit to expand a hidden neuron. We to set μ as the average distance of the sample in the input space.

$$u = \left(\frac{V}{N}\right)^{1/n}, \quad (7.12)$$

The α and β are parameters to control how many (μ) units r_2 and r_3 should be away from r_1 . A large value of α and a small value of β jointly results in fast learning but more hidden neurons; while a small value of α and a large value of β results in less hidden neurons, but a little longer training time. Even when using a small value of α and a large value of β , FCLAR is much faster than traditional training algorithms based on gradient descent. This conclusion can be experimentally obtained from the dataset in the following part. A conservative way is to set α as smaller value and β as larger value. Experimentally, α is set to some value around 0.1 or less, and β is set to some value between 2 and 10. By doing this, the result neural network and its performance are satisfactory and stable.

7.5 Our Proposed Method

7.5.1 Geometrical Expansion

We first introduce the training process of geometrical expansion. In terms of these three radii: r_1 , r_2 and r_3 , three regions are defined for each hypersphere (hidden neuron). The region within r_1 is defined as the “match region”. A sample within the “match region” of some hidden neuron has been covered by its corresponding hypersphere and no action needs to be taken. The region between r_1 and r_2 is defined as the “claim region”. A sample within the “claim region” can be directly included by a slight expansion of the corresponding hypersphere. The region between r_2 and r_3 is defined as the “boundary region”. A sample within the “boundary region” is promising to be included by this hidden neuron in the near future and will be backed up to be learned later. A sample out of the “boundary region” is too far away and considered irrelevant to the existing hidden neurons. This sample requires a new hidden neuron to represent it.

Assume we express the input space by a two-dimensional plane, and express hyperspheres by circles. Fig. 7.4 explains how hyperspheres are created or expanded in the geometrical expansion process.

Fig. 7.4 (a) is the distribution of an arbitrary dataset. Black samples are expected to be separated from white samples. We cover the black samples using hidden neurons and the uncovered ones belong to white class. After several training steps, we gain the hidden layer with two hidden neurons as shown in Fig. 7.4 (b). Hidden neuron 1 and hidden neuron 2 represent x^1 and x^2 respectively. The new coming samples, x^3 and x^4 , within the “claim region” of hidden neuron 1 or hidden neuron 2 in Fig. 7.4 (b) cause an immediate expansion of these two hyperspheres. x^3 and x^4 are located on the surfaces of these two hyperspheres after expansion, shown in Fig. 7.4 (c). A new sample, say x^5 , in Fig. 7.4 (b) beyond r_3 of all hyperspheres causes to the creation of a new hypersphere (a new hidden neuron) to cover it (in Fig. 7.4 (c)). Fig. 7.4 (d) is the final training result based on geometrical expansion.

Fig. 7.4 The process of geometrical expansion for FCLAR

7.5.2 Train the Hidden Layer

FCLAR begins with an empty hidden layer. In the training process, samples are sequentially input to the learning system in an arbitrary order. To construct the neural network, we examine whether a coming “true” sample can be covered by one of the existing hidden neurons. When the first sample comes, the hidden layer is empty and no hidden neuron covers this sample. A new hidden neuron, the first hidden neuron, is created to represent it. Samples are removed immediately after they are learned (by parameter updating). The training process goes on. An incoming sample, x^k , causes one of

the following actions:

- (i) update the parameters of some hidden neuron, and remove $\mathbf{x}^k$;
- (ii) create a new hidden neuron to represent it, and remove $\mathbf{x}^k$;
- (iii) back up $\mathbf{x}^k$ to be learned in the next training circle.

Given a hidden neuron j with the center $\mathbf{c}^j$ and three radii r_1^j , r_2^j and r_3^j , we first compute the Euclidean distance $d(\mathbf{x}^k, \mathbf{c}^j)$ between the center $\mathbf{c}^j$ and sample $\mathbf{x}^k$: $d(\mathbf{x}^k, \mathbf{c}^j) = \|\mathbf{x}^k - \mathbf{c}^j\|$. If $d(\mathbf{x}^k, \mathbf{c}^j) \leq r_1^j$, this true sample is already covered by this hidden neuron, so nothing needs to be done. If $r_1^j < d(\mathbf{x}^k, \mathbf{c}^j) \leq r_2^j$, the sample $\mathbf{x}^k$ is within the “claim region” of this hidden neuron; so we update its parameters to include $\mathbf{x}^k$ in this neuron using (7.8) and (7.9). A new neuron with center $\mathbf{x}^k$ is created using (7.6) and (7.7) if $d(\mathbf{x}^k, \mathbf{c}^j) > r_3^j$ for all existing neurons ($j=1 \dots m$, where m is the number of existing hidden neurons). If $r_2^j < d(\mathbf{x}^k, \mathbf{c}^j) \leq r_3^j$, the sample $\mathbf{x}^k$ is within the “boundary region” of that neuron; in this case, we first examine whether other available hidden neurons can “match” or “claim” it. If it cannot be included in any other available neuron, we “put it aside” for re-consideration after other vertices are processed. Inclusion of other vertices to existing hidden neurons results in expansion of their “match” and “claim” regions; the samples, put aside earlier, may be claimed by such “expanded” neuron(s) later.

Fig. 7.5 and Fig. 7.6 give the algorithm and flowchart for FCLAR.

In Fig. 7.6 and 7.7, q is the number of existing hidden neurons, j and g are index of hidden neurons, K is the number of vertices in the training set, and k is the index of vertices.

```

q=1; //number of hidden neurons
p=1; //number of training circles
for (p=1,...,maximal training circles) & (the training set is not empty)
  for (k=1, ..., N) & if sample k is not removed
    // N is the number of the training samples
    for (j=1, ..., q)
      compute the distance  $d(\mathbf{x}^k, \mathbf{c}^j)$  from  $\mathbf{x}^k$  to  $\mathbf{c}^j$ 
      if  $d(\mathbf{x}^k, \mathbf{c}^j) \leq r_1^j$ 
        ignore this sample, remove  $\mathbf{x}^k$  from the training set, and jump
        out to check the next sample;
      else if  $r_1^j < d(\mathbf{x}^k, \mathbf{c}^j) \leq r_2^j$ 
        expand hidden neuron j to include  $\mathbf{x}^k$  using (7.8) and (7.9),
        then remove  $\mathbf{x}^k$  from the training set and jump out to check the
        next sample;
      next j;
      if for all j's,  $d(\mathbf{x}^k, \mathbf{c}^j) > r_3^j$ 
        q=q+1 create a new hidden neuron, neuron q, using (7.6) and
        (7.7), then remove the sample  $\mathbf{x}^k$  from the training set;
      else
        backup  $\mathbf{x}^k$  to be learned in the next training circle;
    next k;
    for (j=1, ..., q)
      if in this training circle, hidden neuron j is not expanded
        decrease  $r_3^j$  and increase  $r_2^j$  using (7.13) and (7.14);
    next j
  next p

```

Fig. 7.5 The training algorithm of FCLAR

Fig. 7.6 Flowchart of FCLAR

*¹Create a new hidden neuron: Set $q=q+1$, create hidden neuron q to represent this vertex using (7.6) and (7.7)

*²Expand the g th hidden: Train hidden neuron g using (7.13) and (7.14)

Fig. 7.7 Flow chart of one training circle for FCLAR

However, if the vertices “put aside” get trapped only within the boundary regions of some existing neurons, we update the parameters of the corresponding hidden neurons: expanding r_2 and decreasing r_3 using (7.13) and (7.14).

$$r_2 = r_2 + s \times \mu, \quad (7.13)$$

$$r_3 = r_3 - s \times \mu, \quad (7.14)$$

where s is considered as a step length for shrinking and expanding and $s < 1$. A large value of s results in fast but rough learning, while a small value of s results in refined but slow learning. In our research, $s=0.01$.

7.5.3 Sample Reordering

FCLAR learns the coming samples according to its geometrical location (near to or far from the existing hidden neurons). FCLAR learns “close” (“easy” to learn) samples first and then learns “far” (“hard” to learn) ones. In the training process, given a sample $\mathbf{x}^k$, if for some hidden neuron j , $r_2^j < d(\mathbf{x}^k, \mathbf{c}^j) \leq r_3^j$, and $d(\mathbf{x}^k, \mathbf{c}) > r_3$ for all other hidden neurons, $\mathbf{x}^k$ is backed up to be reconsidered in the next training circle. The back-up and re-consideration implements the sample reordering process. By doing this, FCLAR first learns “easy” samples which are more similar to the samples having been represented by some existing hidden neurons ($r_1^j < d(\mathbf{x}^k, \mathbf{c}^j) \leq r_2^j$). The ‘confusing’ samples ($r_2^j < d(\mathbf{x}^k, \mathbf{c}^j) \leq r_3^j$) are learned after learning these “easy” samples. This sample reordering process is done based on the geometrical analysis on the fly.

7.5.4 Train the Output Layer

If we apply our method to supervised learning, for example, classification problems, a three-layered neural network with one input layer, one hidden layer and one output layer is constructed. The output layer is required to collect the outputs from the hidden neurons with connection weights one and threshold one.

When we apply our method to unsupervised learning, for example, clustering problems, a two-layered neural network with one input layer and one output layer is constructed. The training process for the output layer of unsupervised learning is exactly the same training process for the hidden layer of supervised learning.

7.6 Supervised Learning and FCLAR

We illustrate the advantages of FCLAR in supervised learning for classification problems. Samples covered by one hidden neuron belong to the same class. Now we apply our algorithm to a widely used dataset, Iris dataset [URL-Datasets] for classification, which has been used in chapter 6. It consists of three target classes: Iris Setosa, Iris Verginica and Iris Versicolor. Each species contains 50 data samples. Each sample has four real-valued features: sepal length, sepal width, petal length and petal width. Before training, data are normalized by using (7.1). After normalization, all features of these samples are between zero and one. By doing this, all features have the same contribution to Iris classification. In our method, two output neurons are needed to represent Iris Setosa and Iris Verginica. Samples from Iris Setosa cause the first output neuron to fire. Samples from Iris Verginica cause the second output neuron to fire. The samples causing neither output neuron to fire belong to Iris Versicolor.

7.6.1 Performance of FCLAR

We use half of the data samples (75 items) for training and the remaining (75 items) for testing. The results of FCLAR classification for the training set and testing set are reported in Table 7.1 and Table 7.2 by setting $\alpha=0.1$ in (7.10), $\beta=5$ in (7.11), $\lambda=0.5$ in (7.7) and $s=0.01$ in (7.13) and (7.14).

For both the training set and the testing set, Setosa is 100% correctly classified because it does not overlap with the other two classes. For the other two classes, the number of misclassified samples is 2. The overall classification accuracy is 97.3% for both the training set and the testing set. FCLAR is very fast and only uses 0.22 second for training (Pentium 2-366 and Matlab 6.0) because each data sample is learned only once. The testing process is quite fast compared with the training process and have not considered in our discussion.

Table 7.1 Iris data classification for the training set using FCLAR

	No. of training data	Correctly classified	Misclassified	Accuracy	Training time
Setosa	25	25	0	100%	0.22 second
Verginica	25	25	0	100%	
Versicolor	25	23	2	92%	
Overall	75	73	2	97.3%	

Table 7.2 Iris data classification for the testing set using FCLAR

	No. of testing data	Correctly classified	Misclassified	Accuracy
Setosa	25	25	0	100%
Verginica	25	25	0	100%
Versicolor	25	23	2	92%
Overall	75	73	2	97.3%

7.6.2 Performance Comparison for Different Parameter Values: α and β

Now we analyze the performance of FCLAR for different values of α and β . Table 7.3 is the performance analysis of FCLAR for different values of parameters of α and β .

Generally speaking, a large value of α and a small value of β results in a fast learning but more hidden neurons are needed. The fifth and sixth rows use a little less training time by setting α larger than 0.1 and setting β as 5. A small value of α and a large value of β results in a less number of hidden neurons, but a little longer learning time. By setting α less than 0.1 (from the second row to the fourth row) or setting β larger than 5 (the last two rows), longer training time is needed.

Table 7.3 Sensitive analysis for parameters of α and β

Parameters	Training time	Training accuracy	Testing accuracy	No. of hidden neurons
$\alpha=0.1, \beta=5$	0.22	97.3%	97.3%	16
$\alpha=0.01, \beta=5$	0.23	97.3%	96%	17
$\alpha=0.001, \beta=5$	0.23	98.7%	96%	18
$\alpha=0.0001, \beta=5$	0.23	98.7%	96%	18
$\alpha=0.2, \beta=5$	0.22	97.3%	97.3%	16
$\alpha=0.3, \beta=5$	0.22	94.7%	94.7%	7
$\alpha=0.1, \beta=10$	0.22	97.3%	97.3%	16
$\alpha=0.01, \beta=10$	0.23	97.3%	96%	16
$\alpha=0.1, \beta=20$	0.23	97.3%	97.3%	16

α is set as some value around 0.1 and β is set as some value between 2 and 10. If α is less than 0.3, whether we set β as 5 (from the first row to the fifth row), 10 (the seventh and the eighth row), or 20 (the last row) the results are steady in terms of accuracy. In addition, the results are not sensitive to β . We can gain this conclusion from

the comparison between the first, seventh and last rows, as well as the comparison between the second and eighth rows. In the sixth row, we gain an accuracy of 94.7% using seven hidden neurons by setting α as 0.3. Finally, the training time does not change much. Given different values of α and β , FCLAR results in similar conclusion except for the sixth row, where $\alpha=0.3$, $\beta=5$. Hence, the parameter setting for α and β is more flexible and does not affect the whole performance dramatically.

7.6.3 Performance Comparison with Traditional Training Algorithms

We compare FCLAR with other neural network training algorithms: Error Backpropagation (EBP), Radial Basis Function (RBF) neural networks and Support Vector Machine (SVM), for Iris data classification. We use half of the data samples (75 items) for training and the remaining (75 items) for testing as well. The results are shown in Table 7.4.

Table 7.4 Performance comparison for Iris data classification

Method	Training time	No. of iterations	Misclassified for training set	Misclassified for testing set	Accuracy for training set	Accuracy for testing set	No. of hidden neurons
EBP	39.985	500	3	2	96%	97.3%	5
EBP	68.629	800	2	1	97.3%	98.6%	5
RBF	16.844	85	4	4	94.6%	94.6%	17
RBF	19.818	111	3	2	96%	97.3%	15
SVM	8.743	5000	4	5	94.6%	93.3%	NA
FCLAR	0.22	1	2	2	97.3%	97.3%	16

7.6.3.1 Computing Iterations

From Table 7.4, we observe that FCLAR is several hundred times faster than EBP and RBF. This is because FCLAR decides whether to learn the current trained sample (create a new hidden neuron or expand an existing hidden neuron) or back it up for the next training circle by its location. Each sample is removed immediately after learned and causes parameter updating once. In terms of computing iterations, FCLAR only uses one iteration for training. Hence, our algorithm avoids iterative computing in traditional learning algorithms for neural networks. The training results reported in Table 7.4 are obtained on Pentium 2-366 through Matlab 6.0. So the training speed may be greatly improved in an advanced computer system.

7.6.3.2 Accuracy

Iris Setosa does not overlap with the other two classes. However the other two classes overlap with each other greatly. In this case, FCLAR is better than SVM and RBF in both training accuracy and testing accuracy. EBP (in the second row) has a little better performance in accuracy at the expense of more than three hundred times of training time.

7.6.3.3 Number of Hidden Neurons

The number of hidden neurons needed by FCLAR is similar to that of RBF neural networks. FCLAR needs about two times more hidden neurons than EBP. However, FCLAR is several hundred times faster than EBP. With the current trend of availability of higher memory sizes for computers, we expect that triple storage would not limit the use of FCLAR for many practical problems.

In addition, the number of hidden neurons must be decided before training in both EBP and RBF neural networks. A different number of hidden neurons results in different training time and training accuracy. It is still a difficult task to determine the number of hidden neurons in advance. The only way is to try different numbers of hidden neurons and find an optimal structure, which is in general blind and time consuming. We do not have to know the number of hidden neurons needed by FCLAR; therefore FCLAR avoids this trying process and gains satisfactory results. We can say the overall performance of FCLAR is better than EBP, SVM and RBF neural networks, although it may need more hidden neurons than EBP.

7.6.4 Performance Comparison Between FCLA and FCLAR

We use the same dataset Iris dataset for comparison in chapter 6 and chapter 7. FCLA is designed for Boolean applications. FCLA uses Boolean vectors as the inputs, and the connection weights and thresholds of FCLA are all integral values. Data pre-processing must be done for Iris data in Chapter 6. There are many methods for data pre-processing (data quantization and encoding Boolean vectors).

Table 7.5 gives the performance comparison between FCLA and FCLAR given different parameters (to control the geometrical expansion). Generally speaking, FCLAR is faster than FCLA. FCLA gains a better performance in terms of training accuracy. But the testing accuracy of FCLA is affected by the parameters (a_1 and a_2). However FCLAR gains a more stable performance than FCLA (in terms of both accuracy and training time) given different parameters.

Table 7.5 Performance comparison between FCLA and FCLAR (Iris dataset)

Method	Parameters	Training/Testing data	Training accuracy	Testing accuracy	Training time
FCLA	$a_1=2, a_2=6$	50/100	100%	96%	0.898
FCLA	$a_1=2, a_2=4$	50/100	100%	95%	0.873
FCLA	$a_1=3, a_2=6$	50/100	98%	92%	0.867
FCLA	$a_1=2, a_2=6$	70/100	100%	100%	0.913
FCLA	$a_1=2, a_2=6$	75/100	100%	100%	0.928
FCLAR	$\alpha =0.1$ $\beta=5$	75/75	97.3%	97.3%	0.22
FCLAR	$\alpha =0.01$ $\beta=5$	75/75	97.3%	96%	0.23
FCLAR	$\alpha =0.001$ $\beta=5$	75/75	98.7%	96%	0.23
FCLAR	$\alpha =0.0001$ $\beta=5$	75/75	98.7%	96%	0.23
FCLAR	$\alpha =0.2$ $\beta=5$	75/75	97.3%	97.3%	0.22
FCLAR	$\alpha =0.3$ $\beta=5$	75/75	94.7%	94.7%	0.22
FCLAR	$\alpha =0.1$ $\beta=10$	75/75	97.3%	97.3%	0.22
FCLAR	$\alpha =0.01$ $\beta=10$	75/75	97.3%	96%	0.23
FCLAR	$\alpha =0.1$ $\beta=20$	75/75	97.3%	97.3%	0.23

7.7 Unsupervised Learning and FCLAR

Now we apply FCLAR to unsupervised learning by solving clustering problems to illustrate its advantages. FCLAR constructs a two-layered neural network with one input layer and one output layer.

We apply FCLAR to a clustering problem of statistic image segment dataset [URL-Datasets]. The dataset includes 2310 instances drawn randomly from a database of 7 outdoor images. They are brick face, sky, foliage, cement, window, path and grass. Each

instance is a 3x3 region with 19 continuous attributes: region-centroid-col, region-centroid-row, region-pixel-count, short-line-density-5, short-line-density-2, vedge-mean, vedge-sd, hedge-mean, hedge-sd, intensity-mean, rawred-mean, rawblue-mean, rawgreen-mean, exred-mean, exblue-mean, exgreen-mean, value-mean, saturatoin-mean and hue-mean.

We cluster these 2310 instances using FCLAR by setting $a_1=0.3$ and $a_2=0.5$. 68% instances of the dataset are correctly mapped to their corresponding cluster. However by using SOM, only an accuracy of 48% is obtained. This dataset is a disordered one. The training time is 0.884 second by using FCLAR and 2.47 seconds by using SOM and setting the number of iterations as 3000.

We gain similar conclusion that FCLAR is faster and have better clustering results. Its training process is exactly the same as the training process for the hidden layer in supervised learning.

7.8 Conclusion Remarks

We now give a brief summary of Fast Covering Learning Algorithm for Real valued problems (FCLAR) for neural networks. FCLAR is based on geometrical expansion in the input hyperspace. FCLAR begins with an empty hidden layer and constructs the neural network incrementally by either adding a new hidden neuron or expanding some existing hidden neuron according to the location of the sequentially coming data. To do this each hidden neuron is represented by its corresponding hypersphere. Three central radii are introduced for each hypersphere to control the geometrical expansion process. Three regions: a match region, a claim region, and a boundary region are defined in terms

of these three radii respectively. Match region and claim region shift and expand in the training process until all samples of the true subset are covered.

FCLAR can be applied to both supervised learning and unsupervised learning. FCLAR constructs a three-layered neural network with one input layer, one hidden layer and one output layer for supervised learning, while constructs a two-layered neural network with one input layer and one output layer for unsupervised learning. Each output neuron in unsupervised learning represents one cluster.

FCLAR is much (several hundred times) faster than other algorithms based on the gradient descent technique (such as EBP and RBF neural networks), and gains satisfactory accuracy. FCLAR decides whether to learn a trained sample or back it up for the next training circle by its location. A trained sample is removed immediately after learned and each training sample is only learned once. By doing this, FCLAR can avoid iterative computing.

FCLAR has some features of brain-like learning. It learns new knowledge without forgetting old ones. In addition, given a training sequence in an arbitrary order, FCLAR learns “easy” samples first and learns “confusing” samples after the “easy” ones, which makes FCLAR more intelligent.

From our experimental work, we conclude that FCLAR has a better performance than traditional algorithms based on gradient descent in terms of both training speed (about one hundred faster) and accuracy.

Chapter 8

Concluding Remarks and Future Work

8.1 Concluding Remarks

This thesis reviews historical work for neural networks and discusses the limitations of existing learning algorithms. Almost all learning algorithms for neural networks are based on the gradient descent technique [Tom 97], which causes iterative computing and long training time.

We begin our research with Boolean neural networks due to their simple geometrical properties. In construction of Boolean neural networks, each hidden neuron is represented by a hyperplane (the intersection of the separating hypersphere and the reference hypersphere). We propose three learning algorithms for Boolean neural network construction. These are Multi-Core Learning (MCL), Multi-Core Expand-and-Truncate

Learning (MCETL), and the Fast Covering Learning Algorithm (FCLA). All these three algorithms are based on theories about linear separability for Boolean vectors, which are proposed and verified in this thesis in chapter 4. Based on these theorems, the separating hyperspheres can be represented in forms of hyperplanes and hence can be represented by linear activation functions.

Our algorithms are based on geometrical expansion, which avoids the iterative computing in traditional algorithms. To do this, each hidden neuron is represented by its corresponding hyperplane or hypersphere with three central radii. The training process of our methods can be visualized by the geometrical expansion process of hyperspheres based on the geometrical location of the sequentially incoming data. Each data item is learned immediately or backed up to be learned in the next training circle. The data item is removed immediately after learned. In this sense, each data item in the training set is only learned once. By doing this, our method avoids the iterative computing in traditional algorithms. As we have known, each data item has to be learned interactively to adapt the neural network to it in traditional algorithms. Hence our methods are much more faster (several hundred times faster) than traditional learning algorithms. In addition, our methods gains satisfactory results in terms of accuracy along with their fast learning speed. We illustrated the advantages of our methods by using some benchmark datasets and we compared our methods with traditional learning algorithms.

MCL and MCETL begin with several cores, and extend the corresponding SITVs simultaneously. These two algorithms avoid blocking problems and need fewer hidden

neurons than previous algorithms such as ETL, IETL and CSCLA. Also, MCL and MCETL use simpler equations to train the Boolean neural networks. In addition, the values of connection weights and thresholds in MLC and MCETL are much simpler than those in ETL and IETL. Smaller values of parameters (connection weights and thresholds) can be easily realized by hardware. MCETL is the combination of MCL and ETL. In MCETL, if we begin with one core vertex, MCETL degenerates to the ETL algorithm. On the contrary, if we begin with h core vertices, and altogether h hidden neurons are needed, MCETL degenerates to the MCL algorithm. So MCETL is more flexible and efficient than the other three algorithms to solve Boolean function mapping problems.

FCLA for Boolean neural networks is based on multi-level geometrical expansion. Two learning processes are involved: the generalization process and the modification process. Three regions are defined in terms of geometrical concept. The learning process is based on the judgment as to which region the new coming vertex belongs. The expansion process is characterized by a suitable choice of parameters such as ξ_1 and ξ_2 , σ_1 and σ_2 , or a_1 , a_2 , and “gap” depending upon the method to control the geometrical expansion process. If we set these parameters as large values to include vertices with a larger distance, the hidden neuron can expand easily, but vertices with a small value of distance will be wrongly included. We either introduce more errors, or choose the values of these parameters such that not many errors are introduced.

FCLA possesses the generalization capability, but all previous learning algorithms for Boolean neural networks (except for BLTA) do not. FCLA generalize near vertices by

expanding hyperspheres to include farther vertices. In addition, FCLA need not assign the core vertex in advance, which makes FCLA to be applicable more generally. FCLA is faster because we need not to search a lot of training pairs for determining each hidden neuron. What we do is to judge which region an input belongs to. FCLA is not based on gradient descent either. Meanwhile FCLA constructs simpler neural networks as well as gains satisfactory performance (training speed and accuracy). The last and the most important features of FCLA is that it is not sensitive to the change of input sequences of the same training set. In summary, FCLA has a better performance experimentally than traditional algorithms based on gradient descent in terms of both training speed (about one hundred faster) and accuracy. FCLA is preferred to previous training algorithms for Boolean neural network as well.

We then extend FCLA for neural network construction to solve real-valued problems, which we call FCLAR. In the real valued applications, the input vectors are not included in a unit hypercube and some attracting properties of Boolean vectors are lost. In cases of real valued input, we represent the hidden neurons by the separating hyperspheres. And three central radii are introduced for each hypersphere to control the geometrical expansion process. FCLAR learns new knowledge without forgetting old ones. In addition, given a training sequence in an arbitrary order, FCLAR learns “easy” samples first and learns “confusing” samples after these “easy” ones, which makes FCLAR more intelligent.

FCLA and FCLAR can be applied to both supervised learning and unsupervised learning.

They construct a three-layered neural network with one input layer, one hidden layer and one output layer for supervised learning. For unsupervised learning, FCLA and FCLAR construct a two-layered neural network with one input layer and one output layer. Each output neuron in unsupervised learning represents one cluster.

FCLAR is much (several hundred times) faster than traditional learning algorithms based on gradient descent technique (such as EBP and RBF neural networks), and gains satisfactory accuracy as well.

8.2 Future Research

8.2.1 Neural Network Pruning

For the applications (for example, the Iris dataset in chapter 7), the neural network structure constructed by FCLAR requires more hidden neurons than other learning algorithms like EBP, RBF, SVM. Further, FCLAR may consist of redundant hidden neurons for the following reasons. Some hidden neurons may overlap with other hidden neurons; some hidden neurons may represent too few samples; some hidden neurons may represent samples with a low density; some hidden neurons may introduce many errors. The pruning process is discussed to trim the neural networks constructed by FCLAR. We now give some suggestions for neural network pruning.

- (i) Remove hidden neurons which are included by other hidden neurons. If a hidden neuron A fires whenever another hidden neuron B fires, we say B include A . This

side-effect may be caused by some improperly initialized parameters (for example, a_1 and a_2). These improper parameters may cause a fast expansion of some hidden neurons, which will include other hidden neurons. In this case, A is redundant and can be removed, for B can represent all samples represented by A .

- (ii) Merge hidden neurons which overlap with other hidden neurons to some degree.

The side-effect of overlapping of hidden neurons may be caused by the same reason discussed in the previous point. These overlapped hidden neurons should be merged into one large neuron.

- (iii) Remove hidden neurons representing less numbers of samples, which may be created by some noisy data. The judgment for 'less number' is different from case to case. It depends on the size of the training dataset and the input dimension. However, this judgment should be considered carefully, because a hidden neuron representing less numbers of samples does not always imply a less important cluster.

- (iv) Remove hidden neurons representing a loose density of samples. This judgment should be considered carefully too. A hidden neuron representing a loose density of samples does not always imply a less important cluster either.

- (v) Remove hidden neurons with more errors.

However, there are many parameters to be determined in the above. Different parameters could be applied to solve different problems. Such further investigations would benefit the compactness and generality of the resulting neural networks.

8.2.2 Insight to Cognitive Science

In each training step of FCLA and FCLAR, we update only parameters of one hidden neuron, the nearest one to the training sample. Parameters of other hidden neurons remain unchanged. A brain-like neural network should have the capacities of localized learning function. From visualization of the cortex [Valverde 71, Demeulemeester 88], it is known that only part of the neurons response to some areas of the input. During the training process of neural networks based on geometrical concept, whether neurons fire or not is based on the similarities between data items (the geometrical distance between the current trained data item and the existing hidden neurons). The learning process of human beings is based on the similarities between the real world objects. Thus this learning process based on geometrical concept is similar to that of a human being and is expected to be more intelligent. Further investigation of our research based on geometrical learning will lead us to a better understanding of the cognitive process of our brain. Also further investigation of our research may motivate more intelligent learning machines.

References

[Anderson 72] Anderson J.A., “A simple neural network generating an iterative memory”, in *Mathematical Biosciences*, Vol.14, No.3, pp.197-220, 1972.

[Arulampalam 03] Arulampalam G. and Bouzerdoun A., “A generalized feedforward neural network classifier”, in *Proceedings of International Joint Conference of Neural Networks (IJCNN03)*, Vol.1, pp.1429-1434, 2003.

[Atkinson 99] Atkinson P.M. and Nicholas J.T., *Advances in remote sensing and GIS analysis*, John Wiley & Sons, New York, NY, 1999.

[Becker 96] Becker S. and Plumbley M., “Unsupervised neural network learning procedures for feature extraction and classification” in *International Journal of Applied Intelligence*, Vol. 6, pp.185-203, 1996.

[Bertrand 92] Bertrand D.J., *Application of neural network to adaptive control theory for super-augmented aircraft*, Master thesis, Naval Postgraduate School, Monterey, CA, 1992.

[Boahen 93] Boahen K.A. and Andreou A.G., “Design of a bidirectional associative memory chip”, in *Associative Neural Memories: Theory and Implementation*, Ed. M.H. Hassoun, Ch.17, pp.288-305, Oxford Univ. Press, 1993.

[Braddock 98] Braddock R.D., Kremmer M.L. and Sanzogni L., “Feed-forward artificial neural network model for forecasting rainfall run-off”, in *Environmetrics*, Vol.9, No.4, pp.419-432, 1998.

[Campbell 97] Campbell C., “Constructive learning techniques for designing neural network systems”, in *Neural Network Systems, Techniques and Applications*, Ed. Leondes C.T., Academic Press, San Diego, CA, pp.1-54, 1997.

[Chaudhari 02] Chaudhari N.S. and Tiwari A., “Extending ETL for multi-class output”, in *Proceedings of International Conference on Neural Information Processing (ICONOP), Proceedings of Computational Intelligence for E-Age, Asia Pacific Neural Network Association (APNNA)*, pp.1777-1780, Singapore, Nov. 2002.

[Chaudhari 03] Chaudhari N.S. and Wang D., “Binary self-organization mapping based on fuzzy geometric expansion”, in *Proceedings of 4th International Conference on Information Communications & Signal Processing Fourth IEEE Pacific-Rim Conference on Multimedia*, Vol.2, pp.1211-1215, Singapore, Dec. 2003.

[Chen 96] Chen W.Y., Chen S.H. and Lin C.J., “A speech recognition method based on the sequential multi-layer perceptrons”, in *Neural Networks*, Vol.9, No.4, pp.655-669, 1996.

[Chuang 02] Chuang L. Hwang J.Y., Chang C.H., Yu C.H. and Chang F.M., “Ultrasound estimation of fetal weight with the use of computerized artificial neural network model”, in *Ultrasound in Medicine & Biology*, Vol.28, No.8, pp.991-996, 2002.

[Clarkson 95] Clarkson P.M. and Stark H., *Signal processing methods for audio, images and telecommunications*, Academic Press, San Diego, CA, 1995.

[Conrads 02] Conrads P.A., Roehl E.A. and Cook J.B., “Estimation of tidal marsh loading effects in a complex estuary”, in *Proceedings of 2002 AWRA Spring Speciatyl Conference on Coastal Water Resources*, Ed. Lesnik J.R., pp.307-312, 2002.

[Costa 97] Costa A. and Markellos R.N., “Evaluating public transport efficiency with neural network models”, in *Transportation Research Part C: Emerging Technology*, Vol.5, No.5, pp.301-312, 1997.

[Coulibaly 00] Coulibaly P., Anctil F., Rasmussen P. and Bobee B., “A recurrent neural networks approach using indices of low-frequency climatic variability to forecast regional annual runoff”, in *Hydrological Processes*, Vol.14, No.15, pp.2755-2777, 2000.

[Crespo 93] Crespo J.L. and Mora E., “Drought estimation with neural networks”, in *Advances in Engineering Software*, Vol.18, No.3, pp.167-170, 1993.

[Curry 93] Curry B. and Moutinho L., “Using advanced computing techniques in

banking”, in *The International Journal of Bank Marketing*, Vol.11, No.6, 1993.

[Dellomo 99] Dellomo M.R., “Helicopter gearbox fault detection: a neural network based approach”, in *Transactions of the ASME: Journal of Vibration and Acoustics*, Vol.121, No.3, pp.265-272, 1999.

[Demeulemeester 88] Demeulemeester H., Vandesande F., Orban G.A., Brandon C., Vanderhaeghen J.J. “Heterogeneity of GABAergic cells in cat visual cortex” in *Journal of Neuroscience* Vol.8, No.3, pp.988–1000, 1988.

[Deolalikar 01] Deolalikar V., “Mapping Boolean functions with neural networks having binary weights and zero thresholds”, in *IEEE Transactions on Neural Networks*, Vol.12, No.3, pp.639-642, July 2001.

[Dowla 93] Dowla F., Leach R., Glenn L., Moran B. and Heinle R., “Neural network methods in hydrodynamic yield estimation”, in *Pure and Applied Geophysics*, Vol.140, No.3, pp.427-454, 1993.

[Eastaugh 01] Eastaugh J.L., *Comparison of neural network and statistical techniques with clinical assessment in the Prediction of high-risk pregnancies*, Leeds, 2001.

[Elshorbagy 00] Elshorbagy A. and Simonovic S.P., “Performance evaluation of artificial neural networks for runoff prediction”, in *Journal of Hydrologic Engineering*, Vol.5, No.4,

pp.424-427, 2000.

[Enderie 94] Enderie J.D., “A physiological neural network for saccadic eye movement control”, in *Brooks Air Force Base*, Air Force Materiel Command, Armstrong Laboratory, TX, 1994.

[Er 03] Er M.J., Wu S., Gao Y., *Dynamic fuzzy neural networks-architectures, algorithms and applications*, McGraw-Hill Press, 2003.

[Fahlman 88] Fahlman S.E., “Faster-learning variations on back-propagation: An empirical study”, in *Proceedings of the Connectionist Models Summer School*, Eds. Sejnowski T.J., Hinton G.E. and Touretzky D.S., Morgan Kaufmann, San Mateo, CA, 1988.

[French 92] French M.N., Krajewski W.F. and Cuykendall R.R., “Rainfall forecasting in space and time using a neural network”, in *Journal of Hydrology*, Vol.137, No.1, pp.1-31, 1992.

[French 98] French M., Recknagel F., and Jarrett G.L., “Scaling issues in artificial neural network modeling and forecasting of algal bloom dynamics”, in *Proceedings of the International Water Resources Engineering Conference*, Vol.1, pp.891-896, 1998.

[Fukushima 86] Fukushima K., Miyake S. and Ito T., *Neocognition, a biocyberbetic*

approach to visual pattern recognition, NHK Science and Technical Research Laboratories, Tokyo, Japan, 1986.

[Gazula 95] Gazula S. and Kabuka M.R., “Design of supervised classifiers using Boolean neural networks”, in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.17, No.12, pp.1239-1246, Dec. 1995.

[Ghosh 94] Ghosh J., LaCour P. and Jackson S., “OTA based neural network architectures with on-chip tuning of synapses”, in *Proceedings of 7th International Conference on VLSI Design*, Calcutta, India, pp.71-76, Jan. 1994.

[Goel 00] Goel P., Dedeoglu G., Roumeliutis S.I., and Sukhatme G.S., “Fault detection and identification in a mobile robot using multiple model estimation and neural network”, in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA00)*, Vol.3, pp.2302-2309, 2000.

[Gray 92] Gray D.L. and Michel A.N., “A training algorithm for binary feed forward neural networks”, in *IEEE Transactions on Neural Networks*, Vol.3, No.2, pp.176-194, Mar. 1992.

[Grossberg 76] Grossberg S., “How does the brain build a cognitive code?”, in *Psychological Review*, Vol.88, pp.375-407, 1976.

[Grossberg 78] Grossberg S. “Behavioral contrast in short-term memory: Serial binary memory models or parallel continuous memory models?”, in *Journal of Mathematical Psychology*, Vol.3, pp.199-219, 1978.

[Hebb 49] Hebb D.O., *The organization of behavior: A neuropsychological theory*, John Wiley & Sons, New York, NY, 1949.

[Hecht-Nielsen 87a] Hecht-Nielsen R., “Nearest matched filter classification of spatio-temporal patterns”, in *Applied Optics*, Vol.26, No.10, pp.1892-1899, 1987.

[Hecht-Nielsen 87b] Hecht-Nielsen R., “Counter-propagation networks”, in *Proceedings of IEEE First International Conference on Neural Networks*, Vol.2, pp.19-32, 1987.

[Hecht-Nielsen 88] Hecht-Nielsen R., “Neurocomputing: picking the human brain”, *IEEE Spectrum*, Vol.25, No.3, pp.36-41, 1988.

[Hecht-Nielsen 90] Hecht-Nielsen R., *Neurocomputing*, Addison-Wesley, Boston, MA, 1990.

[Hopfield 82] Hopfield J.J., “Neural networks and physical systems with emergent collective computational abilities” in *Proceedings of the National Academy of Science*, Vol.79, pp. 2554-2558, 1982.

[Hsu 95] Hsu K., Gupta H.V., and Sorooshian S., “Artificial neural network modeling of the rainfall- runoff process”, in *Water Resources Research*, Vol.31, No.10, pp.2517-2530, 1995.

[Hsu 98] Hsu K., Gupta H.V. and Sorooshian S., “Stream flow forecasting using artificial neural networks”, in *Proceedings of the International Water Resources Engineering Conference*, Vol.2, pp.967-972, 1998.

[Ian 99] Ian C. and Jacek M.Z., *Knowledge-based neurocomputing*, MIT Press, Cambridge, MA, 1999.

[Iordache 91] Iordache O., Corriou J.P. and Tondeur D., “Neural network for system classification and process fault detection”, in *Hungarian Journal of Industrial Chemistry*, Vol.19, No.4, pp.245-326, 1991.

[Karunanithi 94] Karunanithi N., Grenney W.J., Whitley D., and Bovee K., “Neural networks for river flow predictions”, in *Journal of Computing in Civil Engineering*, Vol.8, No.2, pp.201-218, 1994.

[Kenyon 92] Kenyon N.D. and Nightingale C., *Audiovisual telecommunications*, Chapman & Hall, London, England, 1992.

[Kim 93] Kim J.H., Ham B., and Park Sung-Kwon, “The learning of multi-output binary

neural networks for handwriting digit reorganization”, in *Proceedings of International Joint Conference of Neural Networks (IJCNN93)*, Vol.1, pp.605-508. Oct. 1993.

[Kim 94] Kim J.H., Park S. and In Kim S., “The convergence learning of three-layer artificial neural networks for any binary-to-binary mapping”, in *Neural Networks*, Vol.7, pp.4755-4760, July 1994.

[Kim 95] Kim J.H. and Park Sung-Kwon, “The geometrical learning of binary neural networks”, in *IEEE Transactions on Neural Networks*, Vol.6, No.1, pp.237-247, June 1995.

[Kim 98] Kim J.H. and Roche J.R., “Covering cubes by random half cubes, with applications to binary neural networks”, in *Journal of Computer and System Science*, Vol.56, pp.223-252, 1998.

[Kohonen 72] Kohonen T., “Correlation matrix memories”, in *IEEE Transactions on Computers*, Vol.21, pp.353-359, 1972.

[Kohonen 82] Kohonen T., “Self-organized formation of topologically correct feature maps”, in *Biological Cybernetics*, Vol.43, pp.59-69, 1982.

[Kontaratos 98] Kontaratos M., Tillis T. and Kleanthous K., “Predicting parking characteristics: the use of neural networks to support parking management”, in *Neural*

Networks in Transport Applications, Eds. V. Himanen, P. Nijkamp and A. Reggiani, Ashgate, 1998.

[Kosko 92] B. Kosko and P. J. Pacini, "Adaptive fuzzy systems for target tracking", in *Intelligent Systems Engineering*, Vol.1, No.1, pp.3-21, 1992.

[Kosko 91] B. Kosko, "Stochastic competitive learning", in *IEEE Transactions on Neural Networks*, Vol.2, No.5, pp.522-529, 1991.

[Kosko 90] B. Kosko, "Unsupervised learning in noise", in *IEEE Transactions on Neural Networks*, Vol.1, No.1, pp.44-57, 1990.

[KulKarni 95] Kulkarni A.D., *Artificial neural networks for image understanding*, Van Nostrand Reinhold, New York, NY, 1995.

[Kwok, 97] Kwok T.Y. and Yeung D.Y., "Constructive algorithms for structure learning in feedforward neural networks for regression problems", in *IEEE Transaction on Neural Networks*, Vol.8, No.3, pp.630-645, 1997.

[Lashley 50] Lashley K. D., "In search of the engram", in *Symposia of the Society for Experimental Biology*, Vol.4, pp.454-482, 1950.

[Lee 92] Lee G.E. and Tattersall G.D., "Isolated word speech recognition using a neural

network based source model”, in *British Telecom Technology Journal*. Vol.10, No.3, pp.38-47, 1992.

[Lee 03] Lee J., “Global optimization for fast multilayer perceptron training”, in *Proceedings of International Joint Conference of Neural Networks (IJCNN03)*, Vol.1, pp.410-414, 2003.

[Lehrasab 02] Lehrasab N., Dassanayake H.P.B, Roberts C., Fararooy S., and Goodman C.J., “Industrial fault diagnosis: pneumatic train door case study”, in *Journal of Rail and Rapid Transit*, Vol.216, No.3, pp.175-183, 2002.

[Li 97] Li X. Visier J.C., and Vaezi-Nejad H., “A neural network prototype for fault detection and diagnosis of heating systems”, in *ASHRAE Transactions*, Vol.103, No.1, pp.634-644, 1997.

[Linggard 92] Linggard R., Myers D.J., and Nightingale C., *Neural networks for vision, speech, and natural language*, Chapman & Hall, London, England, 1992.

[Linsker 98] Linsker R., “Self-organization in a perceptual network: in *Computer*, Vol.21, pp.105-128, 1988.

[Ma 99] Ma X., Yang Y. and Zhang Z., “Research on the learning algorithm of binary neural networks,” in *Chinese Journal of Computers*, Vol.22, No.9, pp.931-935, Sept.

1999.

[Ma 01] Ma X., Yang Y. and Zhang Z., “Constructive learning of binary neural networks and its application to nonlinear register synthesis”, in *Proceedings of International Conference of Neural Information Processing*, Vol.1, pp.90-95, 2001.

[Maki 97] Maki Y. and Loparo K.A., “A neural network approach to fault detection and diagnosis in industrial processes”, in *IEEE Transactions on Control Systems Technology*, Vol.5, No.6, pp.529-541, 1997.

[Mason 96] Mason J.C., Price R.K., and Temme A., “A neural network model of rainfall runoff using radial basis functions”, in *Journal of Hydraulic Research*, Vol.34, No.4, pp.537-548, 1996.

[Masters 94] Masters T., *Signal and image processing with neural networks: a C++ sourcebook*, John Wiley & Sons, New York, NY, 1994.

[McCulloch 43] McCulloch W.S. and Pitts W.H., “A logical calculus of the ideas immanent in nervous activity”, in *Bulletin of Mathematical Biophysics*, Vol.5, pp.115-133, 1943.

[Mertens 97] Mertens S. and Engel A., “Vapnik-Chervonkis dimension networks with binary weights”, in *Physical Review E*, Vol.55, No.44, pp.4478-4491, Apr. 1997.

[Minsky 69] Minsky M.L. and Papert S.A., *Perceptrons*, MIT Press, Cambridge, MA, 1969.

[Moravej 01] Moravej Z., Vishwakarma D.N., and Singh S.P., “Radial basis function neural network model for protection of power transformer”, in *Electric Power Components and Systems*, Vol.29, No.4, pp.307-320, 2001.

[Morshed 98] Morshed J. and Kaluarachchi J.J., “Application of artificial neural network and genetic algorithm in flow and transport simulations”, in *Advances in Water Resources*, Vol. 22, No.2, pp.145-158, 1998.

[Moutinho 02] Moutinho L. and Phillips P.A., “The impact of strategic planning on the competitiveness, performance and effectiveness of bank branches: a neural network analysis”, in *The International Journal of Bank Marketing*, Vol.20, No.3, pp.102-110, 2002.

[Neusser 93] Neusser S., Nijhuis J., Spaanenburg L., Hoefflinger B., Franke U., Fritz H., “Neurocontrol for lateral vehicle guidance”, In *IEEE Micro*, Vol.13, No.1, pp. 57–66,1993.

[Nigrin 93] Nigrin A., *Neural networks for pattern recognition*, MIT Press, Cambridge, MA, 1993.

[Nilsson 65] Nilsson N.J., *Learning machines foundation of trainable pattern classifiers*, McGraw Hill, New York, NY, 1965.

[Olafsson 92] Olafsson S., “Dynamical neural networks for speech recognition”, in *British Telecom Technology Journal*, Vol.10, No.3, pp.48-58, 1992.

[Park 91a] Park Sung-kwon and Kim J.H., “A liberalization technique for linearly inseparable pattern,” in *Proceedings of 23rd Southeastern Symposium*, pp.207-211, Mar. 1991.

[Park 91b] Park Sung-kwon and Kim J.H., and Chung Ho-sun, “A training algorithm for discrete multiplayer perceptions”, in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS1991)*, June 1991.

[Park 92] Park Sung-Kwon, Marston A. and Kim J.H., “On the structural requirements of multiplayer perceptions in binary field”, in *System Theory, 1992. Proceedings. SSST/CSA 92. The 24th Southeastern Symposium on and The 3rd Annual Symposium on Communications, Signal Processing Expert Systems, and ASIC VLSI Design*, pp.203-207, Mar. 1992.

[Park 93] Park Sung-kwon and Kim J.H., “Geometrical learning algorithm for multiplayer neural network in a binary field”, in *IEEE Transactions on Computers*, Vol.42, No.8,

pp.988-992, Aug. 1993.

[Paul 90] Paul J. and Goidammer E., “Neural net applications in medicine”, in *Frontiers in Artificial Intelligence and Applications*, Eds. Stender J. and Addis T., IOS press, Washington, DC, pp.215-231, 1990.

[Potvin 93] Potvin J.Y., “The traveling salesman problem: a neural network perspective”, in *Journal of Computing, ORSA*, Vol.5, No.4, pp.328-348, 1993.

[Rosenblatt 58] Rosenblatt F., “The perceptron: a probabilistic model for information storage and organization in the brain”, in *Psychological Review*, No.65, pp.386-408, 1958.

[Rosenblatt 62] Rosenblatt F., *Principle of neurodynamics*, Spartan Books, New York, NY, 1962.

[Rumelhart 86] Rumelhart D.E., Hinton, G.E., and Williams, R.J., “Learning internal representation by error propagation”, in *Parallel Distributed Processing*, Eds. Rumelhart, D.E. and McClelland, J.L. Vol.1, pp.318-362, MIT Press, Cambridge, MA, 1986.

[Ryan 98] Ryan J., Lin Meng-Jang and Miikkulainen R.. “Intrusion detection with neural networks” In Jordan, M. I., Kearns, M. J., and Solla, S. A. (editors), *Advances in Neural Information Processing Systems, Vol.10*, pp.943-949, MIT Press, Cambridge, MA, 1998.

[Saund 89] Saund E., “Dimensionality-reduction using connectionist networks”, in *IEEE Transaction on Pattern Analysis and Machine Intelligence*, Vol.11, No.3, pp.304- 314, 1989.

[Sejnowski 87] Sejnowski T.J. and Rosenberg C.R., “Parallel neural networks that learn to pronounce English text”, in *Complex Systems*, Vol.1, pp.145-168, 1987.

[Serrano 97] Serrano C., “Feedforward neural networks in the classification of financial information”, in *European Journal of Finance*, Vol.3, No.3, pp.183-202, 1997.

[Shamseldin 97a] Shamseldin A.Y., “Application of a neural network technique to rainfall-runoff modeling”, in *Journal of Hydrology*, Vol.199, No.3, pp.272-294, 1997.

[Shamseldin 97b] Shamseldin A.Y., O’Connor K.M. and Liang G.C., “Methods for combining the outputs of different rainfall-runoff models”, in *Journal of Hydrology*, Vol.197, No.3, pp.203-229, 1997.

[Smith 02] Smith K.A. and Gupta J.N.D., *Neural networks in business: techniques and applications*, Idea Group Publishing, Hershey, PA, 2002.

[Sreedhar 95] Sreedhar R., Fernandez B. and Masada G.Y., “A neural network based adaptive fault detection scheme”, in *Proceedings of the American Control Conference*. Vol.5, pp.3259-3263, 1995.

[Starzyk 00] Starzyk J.A. and Pang J., "Evolvable binary artificial neural network for data classification", in *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, NV, June 2000.

[Steinbach 02] Steinbach B. and Kohut R., "Neural networks — a model of Boolean functions", in *Proceedings of 5th International Workshops on Boolean Problems*, pp. 223-240, Freiberg, Sept. 2002.

[Su 73] Su S. Y. H., "Trade your Karnaugh maps for topological procedures that easily reduce switching functions that have many variables", in *Electronic Design*, Vol.21, pp.88-92, 1973.

[Suganuma 98] Suganuma N., Bao Y. and Fujiwara N., "Detection license plate of moving vehicle using neural network" in *Proceedings of the IEEE International Conference on Intelligent Vehicles*, Vol.2, 1998.

[Sung 02] Sung Sang-Kyu, Jung J.W., Lee Joon-Tark and Choi Woo-Jin, "Optimal synthetical method for binary neural network Using NETLA", in *Lecture Notes in Artificial Intelligence (LNAI)*, Vol.2275, pp.236-244, 2002.

[Swicegood 98] Swicegood P.G., *Predicting poor bank performance, a comparison of neural networks, discriminant analysis, and professional human judgment*, PhD Thesis,

Florida State University, 1998.

[Tangang 98] Tangang F.T., Tang B., Monahan A.H., and Hsieh W.W., “Forecasting ENSO events: a neural network extended EOF approach”, in *Journal of Climate*, Vol.11, No.1, pp.29-41, 1998.

[Thirumalaiah 98] Thirumalaiah K., and Deo M.C., “River stage forecasting using artificial neural networks”, in *Journal of Hydrologic Engineering*, ASCE, Vol.3, No.1, pp.26-32, 1998.

[Tingsanchali 00] Tingsanchali T. and Gautam M.R., “Application of tank, NAM, ARMA and neural network models to flood forecasting”, in *Hydrological Processes*, Vol.14, No.14, pp.2473-2487, 2000.

[Tom 97] Tom M, *Machine Learning*, McGraw Hill, New York, NY, 1997.

[Toomarian 93] Toomarian N. and Kirkham H., *Communication and control for electric power systems: power system stability applications of artificial neural networks*, Technical Report DOE/ET/29372-T3, Jet Propulsion Lab, Pasadena, CA, 1993.

[URL-Datasets] <http://www.ics.uci.edu/~mlearn>

[Valverde 71] Valverde F., “Short axon neuronal subsystems in the visual cortex of the

monkey” in *International Journal of Neuroscience*, Vol.1 No.3, pp. 181–197, 1971.

[Walley 98] Walley W.J. and Fontana V.N., “Neural network predictors of average score per taxon and number of families at unpolluted river sites in Great Britain”, in *Water Research*, Vol.32, No.3, pp.613-622, 1998.

[Wang 00] Wang Z., *Artificial intelligence application in the diagnosis of power transformer incipient faults*, PhD Thesis, Virginia Polytechnic Institute and State University, 2000.

[Wang 03a] Wang D. and Chaudhari N.S., “A multi-core learning algorithm for binary neural networks”, in *Proceedings of International Joint Conference of Neural Networks*, Vol.1, pp.450-455, July 2003.

[Wang 03b] Wang D. and Chaudhari N.S., “Binary neural network training algorithms based on linear sequential learning,” in *International Journal of Neural Systems*, Vol.13, No.5, pp.333-351, Oct. 2003.

[Wang 03c] Wang D. and Chaudhari N.S., “A fast covering learning algorithm for boolean neural networks”, in *Proceedings of 2nd International Conference on Computational intelligence, Robotics and Autonomous Systems (CIRAS2003)*, Singapore, Dec. 2003.

[Wang 04a] Wang D. and Chaudhari N.S., "A novel training algorithm for boolean neural

networks based on multi-level geometrical expansion", in *Neurocomputing*, Vol.57C, pp.455-461, 2004.

[Wang 04b] Wang D., Chaudhari N.S., and Patra J.C., "A constructive unsupervised learning algorithm for clustering binary patterns", in *Proceedings of International Joint Conference on Neural Networks (IJCNN04)*, Vol.2, pp.1381-1386, Budapest, Hungary, July 2004.

[Wang 04c] Wang D. and Chaudhari N.S., "A fast constructive-covering algorithm for neural network construction", in *Proceedings of International Symposium on Computational Intelligence and Industrial Applications (ISCIIA2004)*, Haikou, China, Dec. 2004.

[Wang 05] Wang D., Chaudhari N.S., and Patra J.C., "Fast constructive-covering approach for neural networks", in *Proceedings of International Joint Conference on Neural Networks (IJCNN05)*, pp. 2167-2172, Montréal, Québec, Canada, July 2005.

[Werbos 74] Werbos P.J. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD thesis, Harvard University, Cambridge, MA, 1974.

[Widrow 60] Widrow B. and Hoff M.E., "Adaptive switching circuits", in *1960 IRE Western Electric Show and Convention Record*, Part 4, pp.96-104, Aug.1960.

[Widrow 62] Widrow B., “Generalization and information storage in neural networks of adaline neurons”, in *Self-Organization Systems*, Eds. Jovita M.C., Jacobi G.T., Goldstein G.D., Sparten Books, Washington, DC, pp.435-461, 1962.

[Wilamowski 96] Wilamowski B.M., “Neural networks and fuzzy systems”, Chapters 124.1 to 124.8 in the *Electronic Handbook*, CRC Press, pp.1893-1914, 1996.

[Wilkinson 93] Wilkinson J.T., *Explaining agricultural bank failures: a neural network approach*, PhD Thesis, Stonier Graduate School of Banking conducted by American Bankers Association at the University of Delaware, 1993.

[Yamamoto 97] Yamamoto A. and Saito T., “An improved expand-and-truncate learning”, in *Proceedings of IEEE International Conference on Neural Network (ICNN)*, Vol.2, pp.1111-1116, June 1997.

[Yang 98] Yang C.C., Chen C.S., and Chang L.C., “Modeling of watershed flood forecasting with time series artificial network algorithm”, in *Proceedings of the International Water Resources Engineering Conference*, Vol.1, pp.903-908, 1998.

[Yu 97] Yu T.C.B., “Intelligent automobile machine vision with smart pixels”, in *Proceedings of the SPIE*, The International Society for Optical Engineering, Vol.2902, pp.30-41, 1997.

[Zaghloul 04a] Zaghloul K.A. and Boahen K.A., “Optic nerve signals in a neuromorphic chip I: outer and inner retina models”, in *IEEE Transactions on Biomedical Engineering*, Vol.51, No.4, pp.657-666, 2004.

[Zaghloul 04b] Zaghloul K.A. and Boahen K.A., “Optic nerve signals in a neuromorphic chip II: testing and results”, in *IEEE Transactions on Biomedical Engineering*, Vol.51, No.4, pp.667-675, 2004.

[Zanella 00] Zanella A. and Taraglio S., “Neural-based system for obstacle detection and scene reconstruction”, in *Proceedings of SPIE, Enhanced and synthetic vision*, The International Society for Optical Engineering, Vol.4023, pp.191-200, 2000.

[Zhang 97] Zhang Q. and Stanley S.J., “Forecasting raw-water quality parameters for the North Saskatchewan River by neural network modeling”, in *Water Research*, Vol.31, No.9, pp.2340-2350, 1997.

[Zurada 92] Zurada J.M., *Artificial neural systems*, West Publishing, St. Paul, MN, 1992.