

İKİLİ YAPAY SİNİR AĞLARI İÇİN BİR ÖĞRENME ALGORİTMASI

YÜKSEK LİSANS TEZİ

Müh. Ersan ALFAN

Tezin Enstitüye Verildiği Tarih : 9 Haziran 1997

Tezin Savunulduğu Tarih : 18 Haziran 1997

Tez Danışmanı : Prof. Dr. Uğur ÇİLİNGİROĞLU

Diğer Jüri Üyeleri : Doç. Dr. Cüneyt GÜZELİŞ
: Doç. Dr. Acar SAVACI

HAZİRAN 1997

A LEARNING ALGORITHM FOR BINARY NEURAL NETWORKS

M. Sc. THESIS

Ersan ALFAN, B. Sc.

Date of Submission : 9 June 1997

Date of Defense : 18 June 1997

Supervisor : Prof. Dr. Uğur ÇİLİNGİROĞLU

Other Members : Assoc. Prof. Cüneyt GÜZELİŞ

: Assoc. Prof. Acar SAVACI

JUNE 1997

FOREWORD

Capacitive Threshold Logic (CTL) is an evolving technique for use in neural networks. For realizing boolean functions by CTL based binary neural networks, an efficient learning algorithm is needed. This study describes an efficient learning algorithm for binary neural networks.

I would like to show my gratitude to Prof. Dr. Uğur ÇİLİNGİROĞLU for his support. I also would like to thank to my all family for their helps and support, to Emanuel ABACI for his understanding and for his financial support, to all personnel in İMA Mühendislik A.Ş., and to my friend Önder ÖZTÜRK for their helps during the preparation of this study. I would like to be grateful to Providence for just in time support.

June 1997

Ersan ALFAN, B.Sc.

CONTENTS

FOREWORD	iii
CONTENTS	iv
LIST OF SYMBOLS	vi
LIST OF FIGURES	vii
LIST OF TABLES.....	viii
SUMMARY	ix
ÖZET	x
CHAPTER 1 INTRODUCTION	1
CHAPTER 2 MATHEMATICAL BACKGROUND OF THRESHOLD LOGIC	3
2.1 Introduction	3
2.2 Switching Functions	3
2.2.1 The n -Cube Q^n	3
2.2.2 Switching Functions.....	5
2.2.3 Cubical Complexes	6
2.3 Threshold Functions	12
2.3.1 Linear Separability	12
2.3.2 Characterizing Parameters.....	18
2.3.3 Unateness	22
CHAPTER 3 LEARNING ALGORITHMS FOR BINARY NEURAL NETWORKS	24
3.1 Preliminaries.....	24

3.2 Goemetrical Learning (ETL) Algorithm.....	25
3.2.1 Learning The Hidden Layer	27
3.2.2 Learning The Output Layer.....	39
3.3 Modified ETL Algorithm.....	47
3.3.1 Modification of Input Vectors.....	48
3.3.2 Prediction of Maximum Required Number of Neurons	55
3.3.3 Starting Vertex Type	62
3.3.4 Selection of Core Vertex and Determining the Initial Weights	64
3.3.5 Arranging The Weights When $t_{\min} < f_{\max}$ and $t_{\min} = f_{\max}$	67
3.3.6 Learning The Output Layer.....	70
3.3.7 Determination of Appropriate Starting Vertex Type.....	72
3.3.8 METL Algorithm Steps	72
3.4 Realization Performance of METL Algorithm.....	73
CONCLUSIONS	77
REFERENCES	78
APPENDICES.....	79
CURRICULUM VITAE	110

LIST OF SYMBOLS

\Rightarrow	implies
Q.E.D.	end of proof
iff	if and only iff
\in	is a member of
\cap	intersection
\cup	union
\subset	is contained in
*	don't care symbol
$\pi(S)$	power of the set
$S_i(K)$	sum of i^{th} coordinates of K
\mathcal{Q}	the set of the two integers 0 and 1
\mathcal{Q}^n	Cartesian product of n copies of \mathcal{Q}
R^n	Euclidean n -space
I^n	the real n -cube
f^n	n -input switching function
net	net function
x_i	i^{th} input
ω_i	i^{th} weight of net function
T	threshold of net function
ω_0	threshold of net function
SITV	set of true vertices
t_{\min}	minimum value of net function among all vertices in SITV
f_{\max}	maximum value of net function among vertices that are not in SITV
v_c	the core vertex
v_c^i	i^{th} bit of the core vertex
V_{minor}	desired output of minor vertices

LIST OF FIGURES

Figure 3.1	The structure of a three-layer BNN for the given example.	39
Figure 3.2	Input vectors are partitioned by ETL.....	44
Figure 3.3	Karnaugh-map of the given example.....	57
Figure 3.4	Karnaugh-map of the given example.....	59
Figure 3.5	Decomposition of the Karnaugh-map of the given example into 4 LS functions.....	61
Figure 3.6	Decomposition of the Karnaugh-map of a switching function into 2 LS functions.....	61
Figure 3.7	Karnaugh-map of the given example.....	63
Figure 3.8	Karnaugh-map of the given example.....	63
Figure 3.9	Karnaugh-map of the given example.....	69
Figure B.1	The positive canonic linearly separable map patterns of $n \leq 4$	84

LIST OF TABLES

Tablo 1.	Verilen Örneğin Komşular Tablosu.....	xiii
Tablo 2.	Verilen Örneğin İndirgenmiş Komşular Tablosu.....	xiv
Table 2.1	Truth Table of The Switching Function	22
Table 2.2	Chow Parameters of The Given Example	22
Table 3.1	The Analysis Of The Hidden Layer For The Given Example	38
Table 3.2	Truth Tables of Threshold Functions	49
Table 3.3	Original Switching Function and Reduced Switching Function.....	51
Table 3.4	Table of Neighbors of The Given Example	58
Table 3.5	Table of Neighbors After The Elimination Process	58
Table 3.6	Table of Neighbors of The Given Example	59
Table 3.7	Table of Neighbors After The Elimination Process	59
Table 3.8	Test Results of METL Algorithm.....	76
Table 3.9	Test Results of 5-Dimensional Switching Functions.....	76
Table A.1	Table of Chow Parameters	80

SUMMARY

This study describes a learning algorithm called *modified expand and truncate learning algorithm* (METL) to train multilayer binary neural networks with guaranteed convergence for any binary-to-binary mapping. The most significant contribution of this study is the development of learning algorithm for three-layer binary neural networks which guarantees the convergence, automatically determining a required number of neurons in the hidden layer. This algorithm can realize any n -dimensional binary-to-binary mapping by a binary neural network that has maximum n -neurons in the hidden layer. Furthermore, the learning speed of the METL algorithm is much faster than that of other learning algorithms in a binary field. Neurons in the binary neural network employ a hard-limiting activation function, with only integer weights and integer thresholds. Therefore, this will greatly facilitate actual hardware implementation of binary neural network using currently available digital VLSI technology.

Chapter 1 gives a short history of neural networks.

Chapter 2 includes mathematical background of threshold logic that is necessary for describing the learning algorithm.

Chapter 3 describes METL algorithm. The most important advantages of METL are explained with examples.

In the conclusion part, the advantages of METL algorithm are mentioned.

In the appendices, some necessary information and the source code of computer program that is used to test METL algorithm are listed.

ÖZET

İKİLİ YAPAY SİNİR AĞLARI İÇİN BİR ÖĞRENME ALGORİTMASI

Tümdevre teknolojisi son yıllarda, özellikle CMOS prosesinde, büyük gelişmeler göstermiştir [1]. Transistör boyutlarının giderek küçülmesi, tümdevre yoğunluğunu oldukça arttırmıştır. Ancak, küçük boyutlara inildikçe tasarım süresi uzamakta ve verim düşmektedir. Bu nedenle, tümdevre üretimindeki ekonomik kısıtlar kırılmakla birlikte, tümdevre yoğunluğu ve maliyet için bir optimum çözüm gerektirir. Kapasitif eşik lojiği (CTL) yapıları serim açısından oldukça sistematiktir ve bu yapıların serim alanı geleneksel lojik ile gerçekleştirilen devrelerden daha küçük olmaktadır [1], [2].

Boole fonksiyonlarının eşik lojiği kullanılarak gerçekleştirilmesi 1960'dan bu yana yoğun araştırmalar yapılan bir konudur. Eşik lojiğinin çok girişli sistemlerdeki kapasitesi ve karmaşık sayısal sistemlerin tasarımında yararlanılabilir bir alan olduğu ortaya konmuştur [1]. Herhangi bir boole fonksiyonu klasik AND - OR kapı dizisiyle gerçekleştirilebilmektedir. Benzer şekilde, AND ve OR kapısı görevi gören eşik lojiği yapıları ile de bu fonksiyonlar gerçekleştirilebilmektedir. Bu tür yapılar, yapay sinir ağlarının bir alt kümesidir ve bu tür ağlar üç tabakadan oluşmaktadır: Giriş tabakası, gizli tabaka ve çıkış tabakası. Böyle bir tasarım eşik lojiğinin avantajlarını kullanmadığı için önerilmemektedir.

Bir yapay sinir ağının CTL yapıları ile gerçekleştirilmesi için ağırlıkların ve eşiklerin pozitif tamsayı olması ve nöronun transfer fonksiyonunun kesin sınırlayıcı olması gerekmektedir. Ancak, dijital VLSI teknolojisiyle gerçeklemeye uygun olabilecek tamsayı eşik ve ağırlıklara sahip yapay sinir ağı modeli öneren etkili bir öğrenme algoritması bulunmamaktadır.

Bu çalışmada, üç tabakalı ikili yapay sinir ağları için nöron sayısını otomatik tespit eden bir öğrenme algoritması tanıtılmıştır. Bu algoritma genel olarak negatif ve pozitif ağırlıklarla çalışmaktadır. Bir nöral ağı CTL devrelerle gerçeklemek gerekirse, bu algoritmanın bulduğu negatif ağırlıklara karşı düşen girişin tümleyeni alınarak bu negatif ağırlıklar pozitif değerlerine dönüştürülmelidir. Bu algoritma, aşağıda Rosenblatt tarafından tanımlanan nöron modelini kullanmaktadır [3], [4]:

$$y = f(\xi)$$

$$\xi = \sum_{i=1}^n \omega_i x_i - \omega_0$$

Burada x_i , $i = 1, \dots, n$, girişleri, ω_i , $i = 1, \dots, n$, ağırlıkları, ω_0 , eşiği ve y , çıkışı göstermektedir. Nöronun transfer fonksiyonu $f(\xi)$ şu şekilde tanımlanmaktadır:

$$\begin{aligned} f(\xi) &= 0 & \text{if } \xi < 0 \\ f(\xi) &= 1 & \text{if } \xi > 0 \end{aligned}$$

$\xi = 0$ ile tanımlanan $(n-1)$ -boyutlu bir hiperdüzlem, lineer ayrılabilir bir boole fonksiyonunu sınıflayabilir. Ağ fonksiyonu olarak da adlandırılan bu hiperdüzlem şu şekilde tanımlanmaktadır:

$$net(x, \omega_0) = \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n - \omega_0 = 0$$

Lineer ayrılabilirlik özelliğine sahip bir boole fonksiyonu, yukardaki nöron modelindeki ağırlıklar ve eşik hesaplanarak tek bir nöronla gerçekleştirilebilmektedir.

Eğer verilen bir boole fonksiyonu bu özelliğe sahip değilse, bu fonksiyonun birkaç lineer ayrılabilir fonksiyonlara parçalanarak bir yapay sinir ağı ile gerçekleştirilmesi gerekmektedir. Bu amaçla, *geliştirilmiş genişlet-ve-kes öğrenme algoritması (modified expand-and-truncate learning algorithm, METL)* geliştirilmiştir.

Algoritma, fonksiyonun '1' olduğu ve '0' olduğu giriş vektörlerinin sayılarının belirlenmesi ile başlamaktadır. Eğer fonksiyonun '1' olduğu vektörlerin sayısı, '0' olduğu vektörlerin sayısından az ise $V_{\text{minor}}=1$, aksi takdirde $V_{\text{minor}}=0$ şeklinde tanımlanan bir azınlık gösterici tespit edilir. İstenen çıkışı V_{minor} 'a eşit olan giriş vektörleri, azınlık vektörleri; diğer vektörler de çoğunluk vektörleri olarak adlandırılmaktadır.

Giriş vektörleri gerektiği gibi sınıflandırıldıktan sonra, fonksiyonun hangi boyuta indirgenebileceği test edilir. Bunun için, azınlık vektörleri kümesi incelemeye alınır. Bu kümedeki her bir eleman incelenerek değeri değişmeyen bir giriş aranır. Örnek olarak şu fonksiyonu dikkate alalım. $\{x_3x_2x_1\} = \{000, 001, 010\}$ giriş vektörleri için f fonksiyonu '1' değerini alsın ve diğer giriş vektörleri için de f fonksiyonu '0' değerini alsın. Burada, azınlık vektörlerinin sayısı 3'tür ve bu vektörler incelendiğinde x_3 girişi hep '0' değerinde kaldığı görülmektedir. Böyle değişmeyen girişler tespit edildikten sonra V_{minor} değeri ve değişmeyen girişlerin değeri not edilir. Verilen örnekte $V_{\text{minor}} = 1$ ve $x_3 = 0$ olarak tespit edilir. Daha sonra, bu giriş azınlık vektörlerinden atılır ve çoğunluk vektörleri arasında, bu girişin değeri, not edilen değere eşit olmayan vektörler, çoğunluk vektörleri kümesinden atılır. Verilen örnekte, bu elemekten sonra ilk durumdaki azınlık vektörleri kümesi, $\{00, 01, 11\}$ şekline dönüşür ve çoğunluk vektörleri kümesi de sadece $\{11\}$ vektörünü içerir. Elemekten sonra elde edilen fonksiyon 2-boyutlu bir fonksiyondur ve benzer işlemler yapılarak bu fonksiyon da indirgenebilir. Bu indirgeme işlemi fonksiyon indirgenemez bir duruma gelene kadar devam edilir.

Fonksiyon indirgendikten sonra fonksiyonun gerçekleştirilebileceği maksimum nöron sayısı belirlenir. Klasik lojik ailelerden bildiğimiz gibi herhangi bir n -boyutlu boole fonksiyonu 2^{n-1} nöron ile gerçekleştirilebilir. Bu limit, Karnaugh diyagramlarında açıkça

görülebilmektedir. Fonksiyonun yapısını bildiğimiz takdirde, bu sınır azınlık vektörlerinin sayısı kadar olmaktadır ve şu şekilde ifade edilir:

$$N_{v_{\max}} = \min \left[2^n - \sum_{i=1}^n f^n(v_i), \sum_{i=1}^n f^n(v_i) \right]$$

Lineer Ayrıştırma (*LS Decomposition, LSD*) adı verilen bir metod, gerçekleştirilebilecek maksimum nöron sayısını bulmak için kullanılabilir. Bunun için azınlık vektörlerinden oluşan bir küme alınır. Her bir küme elemanı, alt alta bir tablo oluşturacak şekilde yazılır ve bunlar *grup liderleri* olarak adlandırılırlar. Sonra, her elemanın komşu vektörleri tespit edilir. Bunun için Hamming uzaklığı ‘1’ olan vektörler tespit edilir ve grup liderinin yanına yazılır. Bu şekilde tablo oluşturulduktan sonra sıfırdan farklı en az komşulara sahip vektörler tespit edilir. Bu vektörlerin komşularının grup lideri olduğu gruplar listelenir ve en çok komşuya sahip olan grup ilk Lineer Ayrılabilir (LA) fonksiyon olarak seçilir. Bu seçilen grubun elemanları tablodan tamamen silinir ve geri kalan vektörler için bu işlemler yenilenir. Örnek olarak, 4-girişli bir fonksiyon düşünelim. Bu fonksiyon {4, 5, 6, 7, 15} giriş vektörleri için ‘1’, diğerleri için ‘0’ değerini alsın. Bu vektörlerin komşularının oluşturduğu tablo, Tablo 1’deki gibidir.

Tablo 1. Verilen Örneğin Komşular Tablosu

Grup Lideri	Komşular	Komşu Sayısı
4	5, 6	2
5	4, 7	2
6	4, 7	2
7	5, 6, 15	3
15	7	1

Tablo 1’de görüldüğü gibi vektör 15, en az komşuya sahiptir. Bunun komşusunu incelersek, vektör 7’nin en fazla komşuya sahip olduğunu görürüz. Sonuçta, bu grubu LA fonksiyonu olarak seçip bu elemanları tablodan kaldırırsak Tablo 2 oluşur:

Tablo 2. Verilen Örneğin İndirgenmiş Komşular Tablosu

Grup Lideri	Komşular	Komşu Sayısı
4		0

Buradaki vektör 4 de, ikinci LA fonksiyon olur ve sonuçta ana fonksiyon, 2 alt LA fonksiyona ayrıştırılmış bulunmaktadır. Bu şekilde elde edilen nöron sayısını N_{LSDmax} ile gösterelim.

Son olarak, herhangi bir n -boyutlu boole fonksiyonu maksimum n nöron ile gerçekleştirilebilmektedir. Bunun için şu 4-girişli fonksiyonu dikkate alalım. $\{0, 3, 5, 6, 9, 10, 12, 15\}$ için fonksiyon ‘1’, diğer vektörler için de ‘0’ değerini alsın. Bu, giriş vektörlerinin en kötü dağılımıdır. Eğer $\{1\}$ ve $\{14\}$ vektörleri, çıkışı ‘1’ olacak şekilde dönüştürülürse, elde edilen fonksiyon, 2 nöron ile gerçekleştirilebilmektedir. Esas fonksiyona dönmek için bu değişikliği geri almamız gerektiğinden bu işlem için de 2 nöron gerekmektedir. Sonuçta, $h_1h_2+h_3h_4$ şeklinde gerçekleştirilecek bu fonksiyon için toplam 4 nöron gerekmektedir. Bu sayı da giriş vektörünün boyutuna eşittir. Genelleştirirsek, eğer n -girişli bir boole fonksiyonu, m -girişli bir fonksiyona indirgenebiliyorsa, gereken maksimum nöron sayısı şu şekilde bulunur:

$$N_{dimmax} = m$$

Bütün bu sınırlamalar birleştirilirse bir boole fonksiyonunu gerçeklemek için gereken maksimum nöron sayısı şu şekilde bulunur:

$$N_{\max} = \min[N_{v_{\max}}, N_{LSD_{\max}}, N_{\dim_{\max}}]$$

Temel vektörün tipini (doğru veya yanlış) belirlemek için şu yöntem kullanılabilir: İlk önce tüm vektörlerin (doğru ve yanlış) komşularının sayısı belirlenir. Komşu sayısı en fazla olan vektörün tipi, başlangıç tipi olarak seçilir. Eğer tipleri farklı iki vektör en fazla komşu sayısına sahipse, azınlık vektörlerinin tipi başlangıç tipi olarak seçilir. Belirlenen başlangıç tipi 0 (yanlış) ise doğru vektörler yanlış vektöre ve yanlış vektörler de doğru vektöre dönüştürülür. Başlangıç tipini belirleyen vektör temel vektör olarak seçilerek şu formüle göre ilk ağırlıklar bulunur:

$$\omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n - \omega_0 = 0,$$

$$\omega = 2, \quad \text{Eğer } f(v_i) = 1 \text{ ve } v_c^i = 1,$$

$$\omega = -2, \quad \text{Eğer } f(v_i) = 1 \text{ ve } v_c^i = 0,$$

$$\omega = 4, \quad \text{Eğer } f(v_i) = 0 \text{ ve } v_c^i = 1,$$

$$\omega = -4, \quad \text{Eğer } f(v_i) = 0 \text{ ve } v_c^i = 0,$$

$$\omega_0 = \sum_{k=1}^n \omega_k v_c^k - 3$$

Burada, v_c , temel vektörü; v_i , temel vektöre Hamming uzaklığı '1' olan vektörleri; ω , eşiği; v_c^i ise temel vektörün i -nci bitini göstermektedir. Temel vektör ile aynı tipteki v_i vektörleri bir kümede toplanır.

İlk ağırlıklar hesaplandıktan sonra temel vektöre en yakın aynı tipteki bir vektör seçilerek kümeye yerleştirilir ve aşağıdaki bağıntılar gereğince ağırlıklar hesaplanır:

$$C_i = \sum_{k=1}^{C_0} v_k^i$$

$$\omega_i = (4C_i - 2C_0) \quad i = 1 \dots n$$

$$t_{\min} = \min \left[\sum_{i=1}^n \omega_i v_t^i \right], \quad f_{\max} = \max \left[\sum_{i=1}^n \omega_i v_r^i \right]$$

$$\omega_0 = \frac{t_{\min} + f_{\max}}{2}$$

Eğer $t_{\min} > f_{\max}$ koşulu gerçekleşiyorsa, bu denenen vektör kümeye dahil edilebilir ve bulunan ağırlıklar bu kümedeki vektörler ile diğer vektörleri birbirinden kesin olarak ayırır. Bu şart gerçekleşmediği takdirde ağırlıkları değiştirerek bu şart sağlanmaya çalışılır. Bunun için t_{\min} 'i sağlayan küme içindeki en az '1' sayısına sahip vektör ($v_{t_{\min}}$) ve f_{\max} 'ı sağlayan en az '1' sayısına sahip, küme içinde olmayan vektör ($v_{f_{\max}}$) belirlenir ve aşağıdaki bağıntı gereğince ağırlıklar değiştirilir.

$$\omega_i' = \omega_i + 2v_{t_{\min}}^i - 2v_{f_{\max}}^i$$

$t_{\min} < f_{\max}$ durumunda $f_{\max} - t_{\min} < Dif$ şartı aranır. Dif değişkeni, ilk değer olarak $+\infty$ olarak seçilir ve bu iki şart sağlandığında ağırlıklar değiştirilir ve $Dif = f_{\max} - t_{\min}$ olarak seçilir. Ağırlıklar değiştirildikten sonra t_{\min} ve f_{\max} tekrar hesaplanır ve bu işlemler $t_{\min} > f_{\max}$ oluncaya veya hiçbir şart sağlanmayıncaya kadar devam ettirilir. Eğer yukarıdaki hiçbir şart sağlanmıyorsa, o zaman bu denenen vektör kümeye dahil edilemez ve hesaplanmış ağırlıklar iptal edilerek bir önceki ağırlık değerlerine döndülür.

Bu koşullar uyarınca her doğru vektör kümeye yerleştirilmeye çalışılır. Eğer geriye kalan hiçbir doğru vektör kümeye yerleştirilemiyorsa, en son hesaplanan ağırlıklar, birinci nöronun ağırlıkları olarak kabul edilir ve tüm geriye kalan vektörler dönüştürülür (doğrular yanlışlara ve yanlışlar da doğrulara). Bu dönüştürmeden sonra

doğru vektörler tekrar kümeye yerleştirilmeye çalışılır. Bu şekilde devam edilerek tüm doğru vektörler kümeye dahil edilir. İşlem sonunda otomatik olarak gerekli nöron sayısı da belirlenmiş olmaktadır.

Tüm nöronlar belirlendikten sonra indirgenmiş fonksiyon, şu bağıntılar gereğince ana fonksiyona dönüştürülür:

$$\begin{aligned}
 net_{f^n}(X, \omega'_0) &= \omega_i x_i + net_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0) + \omega_0 - \omega'_0 \\
 \omega_i &= -\min[net_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0)] + 1, \quad \text{Eğer } V_{\text{minor}} = 0 \text{ ve } v_m^i = 0, \\
 \omega'_0 &= \omega_0 \\
 \omega_i &= \min[net_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0)] - 1, \quad \text{Eğer } V_{\text{minor}} = 0 \text{ ve } v_m^i = 1, \\
 \omega'_0 &= \omega_0 + \omega_i \\
 \omega_i &= -\max[net_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0)] - 1, \quad \text{Eğer } V_{\text{minor}} = 1 \text{ ve } v_m^i = 0, \\
 \omega'_0 &= \omega_0 \\
 \omega_i &= \max[net_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0)] + 1, \quad \text{Eğer } V_{\text{minor}} = 1 \text{ ve } v_m^i = 1, \\
 \omega'_0 &= \omega_0 + \omega_i
 \end{aligned}$$

Burada f^n , n -girişli fonksiyonu; v_m^i , f^n 'nin azınlık vektörlerinin i -nci bitini; $V_{\text{minor}} = f^n(v_m)$; x_i , f^n 'nin azınlık vektörlerinin değeri değişmeyen girişini göstermektedir. Fonksiyon kaç kere indirgenmişse o kadar bu bağıntılar uygulanarak ana fonksiyona dönülür.

Tüm ağırlıklar bulunduktan sonra çıkış nöronunun ağırlıklarını hesaplamak için bir T vektörü tanımlanır. Bu vektörün elemanları şu şekilde belirlenir: Eğer i -nci nöron gerçek doğru vektörlere dayanılarak oluşturulduysa $T^i = 1$, dönüştürülmüş doğru vektörlere (gerçekte yanlış vektörler) dayanılarak oluşturulduysa $T^i = 0$ olarak seçilir.

Bu vektör bulunduktan sonra tüm dönüştürülmüş nöronların ağırlıkları -1 ile çarpılarak esas formlarına döndürülür ve n , gizli tabaka içindeki toplam nöron sayısını göstermek şartıyla çıkış nöronunun ağırlıkları şu sistematik metod ile bulunur.

Metod, en içteki fonksiyon olan net_n 'den başlamaktadır. $net_n = 2h_n - 1$ olarak seçilerek şu bağıntılar gereğince net_{n-1} bulunur:

$$net_{n-1} = (-\min[net_n] + 1)h_{n-1} + net_n, \quad \text{Eğer } T^{n-1} = 1 \text{ ise,}$$

$$net_{n-1} = (\max[net_n] + 1)h_{n-1} + net_n - (\max[net_n] + 1), \quad \text{Eğer } T^{n-1} = 0 \text{ ise.}$$

Bu metoda, net_1 bulunana kadar devam edilir. Bulunan bu ağ fonksiyonunda h_i 'ler i -nci nöronun çıkışını; h_i 'lerin katsayıları, çıkış nöronunun bu gizli nöronlara ait ağırlıkları ve sondaki negatif sayı da çıkış nöronunun eşikini göstermektedir.

Çıkış nöronunun ağırlıklarının bulunması ile METL algoritması, verilen bir boole fonksiyonunu 3 tabakalı bir yapay sinir ağı ile gerçekleştirmiş olmaktadır.

METL algoritmasının en önemli özelliği, elde edilen yapay sinir ağının, negatif ağırlıklar dönüştürülmek suretiyle CTL devreleri ile VLSI teknolojisi kullanılarak gerçekleştirmenin mümkün olması ve önerilen bu ağın gizli tabakadaki nöron sayısının, en fazla giriş vektörünün boyutuna eşit olmasıdır. Bu son özellik, bize n -girişli genel amaçlı CTL tabanlı bir PLA devresinin, n tane gizli nörona sahip olabileceğini göstermektedir. Örneğin; 16-girişli genel amaçlı NOR-NOR PLA devresinin ilk NOR dizisinde 32768 tane hat bulunurken, aynı giriş sayısına sahip CTL-CTL PLA devresinde bu sayı sadece 16'dır. Sonuçta, tümleştirme bakımından çok büyük bir yer kazancı sağlanmaktadır. METL algoritması, bazen bu sınırı aşan sonuçlar üretebilmektedir. Farklı başlangıç noktaları seçilerek her zaman n -girişli bir boole fonksiyonu, gizli tabakasında maksimum n nöron bulunduran 3 tabakalı, tüm ağırlıkların ve eşiklerin tamsayı olduğu bir yapay sinir ağı ile gerçekleştirilebilmektedir.

CHAPTER 1

INTRODUCTION

Capacitive Threshold Logic (CTL, [2]) is an evolving technique for use in neural networks. The increasing importance of CTL requires the realization of boolean functions (switching functions) by CTL circuits since threshold logic gates have some advantages on traditional logic gates [2]. One of the important advantages of threshold logic gates is that the realization of a switching function by threshold logic gates requires smaller layout area than that of the traditional one.

Linear separability of a switching function is a very important property for the realization of switching functions by threshold logic. A switching function that has the property of linear separability can be realized by using one threshold logic gate. If a switching function does not have this property, then more than one threshold logic gates are required to realize the switching function. To realize these linearly inseparable functions, first, the function must be decomposed into multiple linearly separable (LS) functions and each LS function must be realized by a threshold logic gate. Then, these gates must be combined by another threshold logic gate to produce the desired output. We call this kind of topology as *binary neural network*.

The threshold logic gates can be called as *neurons* since the neuron model uses the principles of threshold logic. To realize a LS switching function by threshold logic gate, we must determine the weights and threshold of corresponding neuron. The weights and threshold of a neuron can be determined by using learning algorithms for neural networks.

Any switching function can be realized by three-layer binary neural network. To realize binary neural networks by CTL, the weights and thresholds of the neurons

must be positive integer numbers and the activation function of a neuron must be hard-limiting [2]. Since there has not been an efficient learning algorithm for three-layer neural networks which employs a hard-limiting activation function, realization of switching functions causes some optimization problems. In recent years, the back propagation learning (BPL) algorithm has been applied to many binary-to-binary mapping problems [5], [3]. However, since the BPL algorithm searches the solution in continuous space, the BPL algorithm applied to switching functions results in long training time and inefficient performance. Also in the BPL algorithm, the number of neurons in the hidden layer required to solve a given problem is not known *a priori*. It has been widely recognized that Stone-Weierstrass's theorem does not give a practical guideline in determining the required number of neurons [6], [7]. Boolean-like training algorithm (BLTA, [5]) solves binary-to-binary mapping problems by using four-layer binary feedforward neural network (BFNN) which is not an optimal solution. The expand-and-truncate learning algorithm (ETL, [6]) solves any binary-to-binary mapping problems by using three-layer binary neural network which is sufficient for VLSI implementations. The ETL algorithm has some weak concepts that cause unoptimal solutions. The ETL algorithm can not realize some linearly separable functions by only one neuron. This weakness causes high number of neurons in the hidden layer.

In this study, a geometrical learning algorithm called *modified expand-and-truncate learning algorithm (METL)* is described to train a three-layer binary neural network for the generation of binary-to-binary mapping. The neurons in this binary neural network employ hard-limiting activation functions and the weights and thresholds of the neurons are integer numbers. This will greatly facilitate hardware implementation of the binary neural network using CTL circuits. METL algorithm solves the weakness problem of ETL algorithm. In the next chapter, the fundamentals of threshold logic are described. Then, in Chapter 3, the learning algorithms (ETL and METL) for the binary neural networks are described.

CHAPTER 2

MATHEMATICAL BACKGROUND OF THRESHOLD LOGIC

2.1 Introduction

In this chapter, the fundamentals of threshold logic are described. In the first part, the switching functions that are used in boolean logic, are described and necessary definitions and theorems are given. In the second part, special switching functions that are called threshold functions, are described and definitions and theorems related to threshold functions are given.

2.2 Switching Functions

A switching function of n -variables is a two-valued function f defined on the vertices of the unit cube in the n -dimensional space [8, p.1-55]. The two values of f are always denoted by the integers 1 and 0 with $0 < 1$ as usual. The theory of switching functions is given next.

2.2.1 The n -Cube Q^n

Let Q denote the set which consists of the two values of the classical two-valued logic. The two elements in Q are denoted by various notations such as *TRUE* (T) and *FALSE* (F), 1 and -1, or 1 and 0. For definiteness, we will denote the two elements in Q by the two integers 1 and 0 throughout the thesis.

For any given positive integer n , consider the Cartesian power

$$Q^n = Q \times \dots \times Q,$$

which is the Cartesian product of n copies of Q . Thus, the elements of Q^n are the 2^n ordered n -tuples.

$$(x_1, x_2, \dots, x_n),$$

where the k^{th} coordinate x_k is an element in Q for every $k = 1, 2, \dots, n$.

Hereafter, Q^n will be called the n -cube and its 2^n elements its *points or vertices*.

Since the coordinates x_k of the points of Q^n are either 1 or 0, there is no danger of ambiguity if we delete the commas between the coordinates as well as the parentheses at both ends. Then each point of Q^n may be represented by a sequence

$$x_1x_2\dots x_n$$

of n juxtaposed binary digits, and hence corresponds to an integer x satisfying

$$0 \leq x \leq 2^n - 1.$$

On the other hand, since Q is a subset of the R of all real numbers, the n -cube Q^n can be considered as a subset of the Euclidean n -space R^n . In fact, Q^n consists of the 2^n vertices of the *real n -cube*

$$I^n = I \times \dots \times I$$

which is the Cartesian product of n copies of the closed unit interval $I = [0, 1]$ of real numbers r satisfying $0 \leq r \leq 1$.

2.2.2 Switching Functions

By a *switching function*, we mean a function

$$f: Q^n \rightarrow Q$$

from the n -cube Q^n into Q . In other words, a switching function $f(x_1, \dots, x_n)$ of n -variables x_1, \dots, x_n is defined by assigning one of the two integers in Q to each of the 2^n points (x_1, \dots, x_n) of Q^n . Thus, there are

$$\phi(n) = 2^{2^n}$$

switching functions of n -variables.

The table of correspondence of a switching function $f: Q^n \rightarrow Q$ is called the *truth table* of f .

A switching function $f: Q^n \rightarrow Q$ is completely determined by either of its two inverse images

$$f^{-1}(1) = \{x \in Q^n \mid f(x) = 1\},$$

$$f^{-1}(0) = \{x \in Q^n \mid f(x) = 0\},$$

which can be called the *on-set* and the *off-set* of the switching function f respectively. When the coordinates of the points in $f^{-1}(1)$ are fully displayed in the form of an array, $f^{-1}(1)$ is usually called the *on-array* or the *on-matrix* of the switching function f . In a similar situation, $f^{-1}(0)$ is usually called the *off-array* or the *off-matrix* of f .

Since Q consists of two elements 1 and 0, there are exactly two *constant functions* of n variables: the switching function $f: Q^n \rightarrow Q$ with $f(x) = 1$ for all $x \in Q^n$ is called the *unit function* and that with $f(x) = 0$ for all $x \in Q^n$ is called the *zero function*. These constant functions of n variables are denoted by 1 and 0 respectively when there is no danger of ambiguity.

Let i be any integer satisfying $0 \leq i \leq n$. By the i^{th} *elementary function* of n -variables, we mean the switching function

$$e_i: Q^n \rightarrow Q$$

defined by $e_i(x) = x_i$ for every point $x = x_1x_2\dots x_n$ of the n -cube Q^n . Thus the value of the function e_i at an arbitrary point x of Q^n is exactly the i^{th} coordinate x_i of the point x . Because of this, we will denote the i^{th} elementary function of n -variables by the symbol x_i .

2.2.3 Cubical Complexes

Let n be a given positive integer and consider the n -cube Q^n . We will define the notion of a cube in Q^n . For this purpose, let us consider the set

$$Q^* = \{1, 0, *\}$$

which consists of three elements, namely, the integers 1 and 0 together with the “don’t care” symbol *. Thus, we have

$$Q \subset Q^*, \quad Q^* = Q \cup \{*\}.$$

On the other hand, let

$$L = \{1, 2, \dots, n\}$$

denote the set of the first n positive integers.

To define the notion of a cube in Q^n , let

$$\phi: L \rightarrow Q^*$$

be an arbitrarily given function defined on the set l with values in Q^* . Then ϕ determines a subset C of the n -cube Q^n as follows: A point $x = x_1 \dots x_n$ of Q^n is in the subset C if and only if (iff)

$$x_i = \phi(i)$$

for every integer $i \in L$ such that $\phi(i) \neq *$. This subset C of Q^n will be called the *cube* in Q^n defined by the function ϕ , which will be referred to as the coordinate function of the cube C . For each integer $i \in L$, the element $\phi(i) \in Q^*$ will be called the i^{th} coordinate of the cube C . Since the cube C is completely determined by its coordinates $\phi(1)\phi(2)\dots\phi(n)$, we can denote the cube C in Q^n as

$$C = \phi(1)\phi(2)\dots\phi(n).$$

Let C be an arbitrarily given cube in Q^n defined by its coordinate function $\phi : L \rightarrow Q^*$. By the dimension of the cube C , we mean the cardinality of the subset $\phi^{-1}(*)$ of C ; in other words, the dimension of C is the number of $*$'s in its coordinates

$$\phi(1)\phi(2)\dots\phi(n).$$

The dimension of C is denoted by the symbol $\dim(C)$ and obviously satisfies the inequalities

$$0 \leq \dim(C) \leq n.$$

If $\dim(C) = r$, then C will be called an r -cube in Q^n . In this case, C consists of the 2^r points of Q^n which can be obtained by replacing the r $*$'s in the coordinates of C by the integers 1 or 0.

Since the correspondence between the functions $\phi: L \rightarrow Q^*$ and the cubes

$$C = \phi(1)\phi(2)\dots\phi(n)$$

is obviously one to one, there are 3^n cubes in the n -cube Q^n .

The cubes C and D in Q^n of the same dimension are said to be *consecutive* iff they differ in one and only one coordinate. Since C and D are of the same dimension, it is quite clear that the lone disagreeing coordinate of C and D can never be a *. For example, the cubes $0^{**}1$ and $0^{**}0$ in Q^4 are consecutive.

A 0-cube C in Q^n consists of a single point $x \in Q^n$ and may be identified with the point x . Hereafter, the 0-cubes in Q^n will be called the *vertices* of Q^n .

The 1-cubes in Q^n will be called the *edges* of Q^n . An arbitrary edge e of Q^n consists of two consecutive vertices of Q^n called the *end points* of e . Conversely, any two consecutive vertices u and v of Q^n form an edge of Q^n obtained by replacing the disagreeing coordinate in u and v by the “don’t care” symbol *. For example, the edge of Q^4 formed by the vertices 0111 and 0101 is $01*1$.

The 2-cubes in Q^n will be called the *squares* of Q^n . An arbitrary square s of Q^n consists of four points of Q^n called the *vertices* of s . If we replace one of the two *’s in the coordinates of the square s by the integers 1 or 0 respectively, we obtain two consecutive edges d and e of Q^n , called a pair of *opposite sides* of the square s . Besides, we have

$$s = d \cup e;$$

in words, the square s is the union of any pair of opposite sides. Conversely, any two consecutive edges d and e of Q^n form a square of Q^n obtained by replacing the disagreeing coordinate (which clearly can never be a $*$) by $*$. For example, the square of Q^4 formed by the edges $01*1$ and $00*1$ is $0**1$.

Now let $r \geq 3$ and consider any given r -cube C in Q^n . If we replace one of the r $*$'s in the coordinates of C by the integers 1 or 0 respectively, we obtain two consecutive $(r-1)$ -cubes D and E , called a pair of *opposite faces* of the r -cube C . Besides, C is the union of any pair of opposite faces; in symbols,

$$C = D \cup E..$$

Conversely, any two consecutive $(r-1)$ -cubes D and E in Q^n form a pair of opposite faces of an r -cube C whose coordinates are obtained by replacing the disagreeing coordinates are obtained by replacing the disagreeing coordinate in D or E by the “don't care” symbol $*$. For example, the 3-cube in Q^4 formed by the consecutive squares $0**1$ and $0**0$ is $0***$.

Let F be an arbitrarily given subset of the n -cube Q^n . By the *cubical complex* of the set F in Q^n , we mean the collection $K(F)$ of all cubes in Q^n contained in the set F . For each non-negative integer $r \leq n$, let $K_r(F)$ denote the collection of all r -cubes in Q^n contained in the set F . Then $K(F)$ is the union of these collections; in symbols,

$$K(F) = K_0(F) \cup K_1(F) \cup \dots \cup K_n(F).$$

To construct the cubical complex $K(F)$ of a set F in Q^n , it suffices to construct the collections,

$$K_0(F), K_1(F), \dots, K_n(F)$$

by the following algorithm.

By definition, $K_0(F)$ is the set of all points in F , that is

$$K_0(F) = F.$$

Let $r > 0$ and assume that $K_{r-1}(F)$ has already been constructed. Consider all possible pairs of consecutive $(r-1)$ -cubes in $K_{r-1}(F)$. If D and E is such a pair, then D and E form a pair of opposite faces of an r -cube $C(D, E)$ whose coordinates are obtained by replacing the lone disagreeing coordinate in D or E by $*$. Since $D \subset F$ and $E \subset F$, we have

$$C(D, E) = D \cup E \subset F.$$

Then $K_r(F)$ is the collection of all r -cubes $C(D, E)$ for all possible pairs of consecutive $(r-1)$ -cubes D and E in $K_{r-1}(F)$. This completes the inductive description of the algorithm for constructing $K(F)$.

By a *cubical cover* of the set F in Q^n , we mean a sub-collection γ of the cubical complex $K(F)$ such that F is the union of all cubes in γ . Thus, a sub-collection γ of $K(F)$ is a cubical cover of F iff every point of F is contained in at least one cube of γ . In particular, $K_0(F)$ is a cubical cover of F .

Now let us consider an arbitrarily given switching function

$$f: Q^n \rightarrow Q$$

of n variables. By the *cubical complex* $K(f)$, we mean $K(F)$ for the on-set $F = f^{-1}(1)$ of f in Q^n ; and every cubical cover γ of F will be called a *cubical cover* of the switching function f .

For example, suppose that $n = 4$ and $f^{-1}(1) = \{0000, 1001, 1100, 1110\}$. By definition, $K_0(f) = F = f^{-1}(1)$ consists of these 4 vertices. Among these vertices, there is only one pair of consecutive vertices 1100 and 1110. This pair forms an edge $11*0$. Thus, $K_1(f)$ consists of a single edge $11*0$ and, consequently, $K_r(f)$ is empty for every $r = 2, 3$ and 4 . Hence $K(f)$ consists of four vertices and one edge. Besides, one can see that 0000, 1001, $11*0$ constitute a cubical cover of the switching cover.

2.3 Threshold Functions

Threshold functions are special switching functions that have the property of linear separability. In this section, the properties of threshold functions are described.

2.3.1 Linear Separability

A switching function $f: Q^n \rightarrow Q$ of n variables is said to be *linearly separable* provided there exist a hyperplane (i.e. an $(n-1)$ -dimensional linear variety) π in the Euclidean n -space R^n which strictly separates $f^{-1}(1)$ from $f^{-1}(0)$; that is to say, the on-set $f^{-1}(1)$ of f lies on one side of π , the off-set $f^{-1}(0)$ of f lies on the other side of π , and the intersection $\pi \cap Q^n$ is empty. The hyperplane π will be called a *strict separating hyperplane* of the given linearly separable switching function f .

Let π be any strict separating hyperplane of a given linearly separable switching function $f: Q^n \rightarrow Q$ and let

$$\omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n = T$$

be an equation of the hyperplane π in the n variables x_1, \dots, x_n . Multiplying both sides of the equation by -1 if necessary, we may always assume that, for an arbitrary point

$$x = x_1 \dots x_n$$

of the n -cube Q^n , we have

$$\begin{aligned} \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n &> T, & \text{if } f(x) = 1, \\ \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n &< T, & \text{if } f(x) = 0. \end{aligned}$$

In this case, the system

$$(\omega_1, \dots, \omega_n; T)$$

of $n + 1$ real numbers is called a *strict separating system* for the linearly separable function f . The first n real numbers $\omega_1, \dots, \omega_n$ in this system are called the *weights*, and the last real number T is referred to as the *threshold*.

The definition of linear separability given above implies the following theorem.

Theorem 2.1: A switching function $f: Q^n \rightarrow Q$ of n variables is linearly separable iff, for each subset K of $n + 2$ points of Q^n , there exists a hyperplane π_K in the Euclidean n -space R^n which strictly separates $K \cap f^{-1}(1)$ from $K \cap f^{-1}(0)$.

Proposition 2.2: For an arbitrarily given switching function $f: Q^n \rightarrow Q$ of n variables, the following five conditions are equivalent:

(i) f is linearly separable.

(ii) There exist real numbers a_1, \dots, a_n and A such that, for an arbitrary point $x = x_1 \dots x_n$ of Q^n , we have

$$\begin{aligned} a_1x_1 + a_2x_2 + \dots + a_nx_n &\geq A, & \text{if } f(x) = 1, \\ a_1x_1 + a_2x_2 + \dots + a_nx_n &< A, & \text{if } f(x) = 0, \end{aligned}$$

(iii) There exist real numbers b_1, \dots, b_n and B such that, for an arbitrary point $x = x_1 \dots x_n$ of Q^n , we have

$$\begin{aligned} b_1x_1 + b_2x_2 + \dots + b_nx_n &\leq B, & \text{if } f(x) = 1, \\ b_1x_1 + b_2x_2 + \dots + b_nx_n &> B, & \text{if } f(x) = 0, \end{aligned}$$

(iv) There exist real numbers c_1, \dots, c_n and C such that, for an arbitrary point $x = x_1 \dots x_n$ of Q^n , we have

$$\begin{aligned} c_1x_1 + c_2x_2 + \dots + c_nx_n &< C, & \text{if } f(x) = 1, \\ c_1x_1 + c_2x_2 + \dots + c_nx_n &\geq C, & \text{if } f(x) = 0, \end{aligned}$$

(v) There exist real numbers d_1, \dots, d_n and D such that, for an arbitrary point $x = x_1 \dots x_n$ of Q^n , we have

$$\begin{aligned} d_1x_1 + d_2x_2 + \dots + d_nx_n &> D, & \text{if } f(x) = 1, \\ d_1x_1 + d_2x_2 + \dots + d_nx_n &\leq D, & \text{if } f(x) = 0, \end{aligned}$$

Proof: (i) \Rightarrow (ii). Assume that f is linearly separable. Then, by definition, f admits a strict separating system $(\omega_1, \dots, \omega_n; T)$. Take $A = T$ and $a_i = \omega_i$ for every $i = 1, \dots, n$. Then the condition (ii) holds.

(ii) \Rightarrow (iii). Assume that f satisfies the condition (ii). Take $B = -A$ and $b_i = -a_i$ for every $i = 1, \dots, n$. Then we obtain (iii) by multiplying the inequalities in (ii) by -1 .

(iii) \Rightarrow (iv). Assume that f satisfies the condition (iii). Consider the off-set $F' = f^{-1}(0)$ of f . Then for each point $x = x_1 \dots x_n$ in F' , we have

$$p(x) = b_1x_1 + \dots + b_nx_n - B > 0.$$

Define a positive real number r as follows: If F' is empty, set $r = 1$; otherwise set r to be the smallest of the real numbers $p(x)$ for all $x \in F'$. Then we obtain (iv) by taking $C = B + r$ and $c_i = b_i$ for every $i = 1, \dots, n$.

(iv) \Rightarrow (v). Assume that f satisfies the condition (iv). Take $D = -C$ and $d_i = -c_i$ for every $i = 1, \dots, n$. Then we obtain (v) by multiplying the inequalities in (iv) by -1 .

(iii) \Rightarrow (iv). Assume that f satisfies the condition (iii). Consider the on-set $F = f^{-1}(1)$ of f . Then for each point $x = x_1 \dots x_n$ in F , we have

$$q(x) = d_1x_1 + \dots + d_nx_n - D > 0.$$

Define a positive real number s as follows: If F is empty, set $s = 1$; otherwise set s to be the smallest of the real numbers $q(x)$ for all $x \in F$. Let $T = D + \frac{1}{2}s$ and $\omega_i = d_i$ for every $i = 1, \dots, n$. Then

$$(\omega_1, \dots, \omega_n; T)$$

is clearly a strict separating system for f . Hence (i) holds.

Q.E.D.

For an arbitrary point $x = x_1 \dots x_n$ of the n -cube Q^n , if we have

$$\begin{aligned} \omega_1x_1 + \omega_2x_2 + \dots + \omega_nx_n &\geq T, & \text{if } f(x) = 1, \\ \omega_1x_1 + \omega_2x_2 + \dots + \omega_nx_n &< T, & \text{if } f(x) = 0, \end{aligned}$$

then, the system $(\omega_1, \dots, \omega_n; T)$ of $n + 1$ real numbers is called a *separating system* for the linearly separable function f . Hence, every strict separating system for f is always a separating system for f but the converse is not always true.

Theorem 2.3: If weights and threshold are integer numbers, every separating system $(\omega_1, \dots, \omega_n; T)$ can be transformed into a strict separating system $(\omega'_1, \dots, \omega'_n; T')$ by setting

$$\begin{aligned} \omega'_i &= 2\omega_i, \\ T' &= 2T - 1. \end{aligned}$$

Proof: As $(\omega_1, \dots, \omega_n; T)$ is a separating system, the following inequalities are satisfied

$$\begin{aligned} \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n &\geq T > T - 1, & \text{if } f(x) = 1, \\ \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n &\leq T - 1 < T, & \text{if } f(x) = 0, \end{aligned}$$

By multiplying these two equations by 2, we obtain

$$\begin{aligned} 2\omega_1 x_1 + 2\omega_2 x_2 + \dots + 2\omega_n x_n &\geq 2T > 2T - 2, & \text{if } f(x) = 1, \\ 2\omega_1 x_1 + 2\omega_2 x_2 + \dots + 2\omega_n x_n &\leq 2T - 2 < 2T, & \text{if } f(x) = 0, \end{aligned}$$

As $2T - 2 < 2T - 1 < 2T$, we obtain

$$\begin{aligned} 2\omega_1 x_1 + 2\omega_2 x_2 + \dots + 2\omega_n x_n &> 2T - 1, & \text{if } f(x) = 1, \\ 2\omega_1 x_1 + 2\omega_2 x_2 + \dots + 2\omega_n x_n &< 2T - 1, & \text{if } f(x) = 0, \end{aligned}$$

If we set $\omega'_i = 2\omega_i$ and $T' = 2T - 1$ for every $i = 1, \dots, n$, we have the strict separating system $(\omega'_1, \dots, \omega'_n; T')$. Q.E.D.

Corollary 2.4: If a switching function $f : Q^n \rightarrow Q$ is linearly separable, so is its complement $f' : Q^n \rightarrow Q$.

Proof: Since f is linearly separable, it admits a separating system $(\omega_1, \dots, \omega_n; T)$. Then, the condition (iv) in Proposition 2.2 holds for its complement f' with $C = T$ and $c_i = \omega_i$ for every $i = 1, \dots, n$. Hence f' is linearly separable. Q.E.D.

Definition: Linearly separable switching functions are called *threshold functions*.

2.3.2 Characterizing Parameters

Let K be any subset of the n -cube Q^n . By the *power* $\pi(K)$, we mean the number of points in K . The *sum of the i^{th} coordinates of K* is defined as

$$S_i(K) = \sum_{x \in K} x_i .$$

The $n + 1$ non-negative integers

$$\pi(K), S_1(K), \dots, S_n(K)$$

will be called the *parameters* of the set $K \subset Q^n$. These can be easily computed as follows: For each point $x \in K$, consider the $(n + 1)$ -vector

$$V_x = (1, x_1, \dots, x_n)$$

of the Euclidean $(n + 1)$ -space R^{n+1} . Then the parameters of K are the coordinates of the vector sum

$$V_k = \sum_{x \in K} V_x .$$

Two sets are K and L in Q^n are said to be *equipollent* iff they have the same parameters; that is,

$$\pi(K) = \pi(L), \quad S_i(K) = S_i(L)$$

for every $i = 1, \dots, n$.

Let $f : Q^n \rightarrow Q$ be an arbitrarily given switching function of n variables. Then the parameters of the on-set $F = f^{-1}(1)$ will be called the *parameters* of the function f ; in symbols,

$$\pi(f) = \pi(F), \quad S_i(f) = S_i(F)$$

for every $i = 1, \dots, n$. Two switching functions $f, g : Q^n \rightarrow Q$ are said to be *equipollent* iff their on-sets $F = f^{-1}(1)$ and $G = g^{-1}(1)$ are equipollent. In other words, f and g are *equipollent* iff they have the same parameters.

Theorem 2.5: If a switching function $f : Q^n \rightarrow Q$ is equipollent to a threshold function $g : Q^n \rightarrow Q$, then $f = g$.

Proof: To prove the theorem by contradiction, let us assume $f \neq g$ and consider the following sets in Q^n :

$$F = f^{-1}(1), \quad G = g^{-1}(1), \quad F' = f^{-1}(0), \quad G' = g^{-1}(0).$$

Since $f \neq g$, we have $F \neq G$ and $F' \neq G'$. Since $\pi(F) = \pi(G)$, it is clear that

$$\pi(G \cap F') = \pi(G' \cap F) = k > 0.$$

Let y_1, \dots, y_k denote the k points of $G \cap F'$ and let z_1, \dots, z_k denote the k points of $G' \cap F$. Consider the intersection $G \cap F$. If $G \cap F$ is not empty, denote the points of $G \cap F$ by x_1, \dots, x_j . Then we have

$$\begin{aligned} G &= \{x_1, \dots, x_j \mid y_1, \dots, y_k\}, \\ F &= \{x_1, \dots, x_j \mid z_1, \dots, z_k\}. \end{aligned}$$

Since F and G are equipollent, we have

$$\sum_{i=1}^j x_i + \sum_{i=1}^k y_i = \sum_{i=1}^j x_i + \sum_{i=1}^k z_i.$$

Hence we obtain the equality

$$\sum_{i=1}^k y_i = \sum_{i=1}^k z_i.$$

On the other hand, if $G \cap F$ is empty, then we have

$$G = \{y_1, \dots, y_k\}, \quad F = \{z_1, \dots, z_k\}.$$

In this case, the equipollence of G and F directly implies the equality

$$\sum_{i=1}^k y_i = \sum_{i=1}^k z_i \quad \text{Q.E.D.}$$

Corollary 2.6: If two distinct switching functions $f, g : Q^n \rightarrow Q$ are equipollent, then neither of them is linearly separable.

Corollary 2.7: A threshold function $f : Q^n \rightarrow Q$ is completely determined by its parameters.

The *Chow label* of f is a set of parameters of f and it is defined by

$$b_0 = 2\pi(f) - 2^n$$

$$b_i = 2 \cdot (2S_i(f) - \pi(f)).$$

Chow label of f can also be found by following equations,

$$y_0 = 1,$$

$$y_i = 2x_i - 1,$$

$$f^n(y) = 2f^n(x) - 1, \quad y = y_1 \dots y_n, \quad x = x_1 \dots x_n,$$

$$b_i = \sum_{x \in (F \cup F')} f^n(y) y_i, \quad i = 0, \dots, n.$$

These labels provide simple means to ascertain whether or not an arbitrarily given switching function is one of the known threshold functions. The ordered absolute value of Chow labels (parameters) of threshold functions ($n \leq 6$) are listed in Appendix A.

Example: Consider a switching function as shown in Table 2.1. The Chow label of this function is found as shown in Table 2.2.

Table 2.1 Truth Table of The Switching Function

$x_1x_2x_3$	000	001	010	011	100	101	110	111
f^3	0	1	1	1	0	1	0	1

Table 2.2 Chow Parameters of The Given Example

	y_0	y_1	y_2	y_3	$f(y)$	$f(y) y_0$	$f(y) y_1$	$f(y) y_2$	$f(y) y_3$
0	+1	-1	-1	-1	-1	-1	+1	+1	+1
1	+1	-1	-1	+1	+1	+1	-1	-1	+1
2	+1	-1	+1	-1	+1	+1	-1	+1	-1
3	+1	-1	+1	+1	+1	+1	-1	+1	+1
4	+1	+1	-1	-1	-1	-1	-1	+1	+1
5	+1	+1	-1	+1	+1	+1	+1	-1	+1
6	+1	+1	+1	-1	-1	-1	-1	-1	+1
7	+1	+1	+1	+1	+1	+1	+1	+1	+1
$b_i = \sum_{x \in (F \cup F')} f(y) y_i$						+2	-2	+2	+6

2.3.3 Unateness

A switching function $f: Q^n \rightarrow Q$ of n variables is said to be *positive in its i^{th} variable* iff f has a polynomial expression in which no term contains x_i as a factor [8, p.68-70]. Similarly, f is said to be *negative in its i^{th} variable* iff it has a polynomial expression in

which no term contains x_i as a factor. The switching function f is said to be *unate in its i^{th} variable* provided that it is positive or negative in its i^{th} variable.

A switching function $f: Q^n \rightarrow Q$ is said to be *unate* iff it is unate in each of its variables. *Positive* and *negative* switching functions are defined similarly.

A unate switching function $f: Q^n \rightarrow Q$ is positive iff

$$2S_i(f) \geq \pi(f)$$

for every $i = 1, \dots, n$.

CHAPTER 3

LEARNING ALGORITHMS FOR BINARY NEURAL NETWORKS

3.1 Preliminaries

Rosenblatt introduced a learning algorithm to accompany a mathematical model which had previously been presented to account for phenomena observed in a biological neural setting [3], [4]. This perceptron model of neuron has inputs x_i , and connection weights ω_i , $i = 1, \dots, n$, a threshold ω_0 , a neural transfer characteristic $f(\xi)$, and an output y . This model of neuron is described by

$$y = f(\xi) \quad (3.1a)$$

$$\xi = \sum_{i=1}^n \omega_i x_i - \omega_0 \quad (3.1b)$$

where x_i , $i = 1, \dots, n$, represent inputs, ω_i , $i = 1, \dots, n$, represent weights, ω_0 denotes a threshold and y denotes the output of a neuron. For the binary outputs, the most useful activation function $f(\xi)$ is described by

$$\begin{aligned} f(\xi) &= 0 & \text{if } \xi < 0 \\ f(\xi) &= 1 & \text{if } \xi \geq 0 \end{aligned} \quad (3.2)$$

This function is called hard-limiting activation function. $\xi = 0$ represents a hyperplane which separates the input space into two classes.

The perceptrons whose activation function is hard-limiting, are called *hard-limiting perceptrons*. The neuron in the Binary Neural Networks (BNN) employs a hard-limiting activation function that only integer weights and integer threshold are accepted. This property give chance to facilitate hardware implementation by using Capacitive Threshold Logic (CTL) circuits and other VLSI technologies.

The perceptrons whose activation function is soft-limiting, are called *soft-limiting perceptrons*. Sigmoid is an example of soft-limiting activation function and it is described by

$$y = \frac{1}{1 + e^{-\beta\xi}} \quad (3.3)$$

This activation function turns a perceptron into a classifier with graded decision boundaries [3]. This kind of functions mostly use in back propagation learning algorithms.

Each neuron in BNN represents a hyperplane which separates the input vectors whose desired output is 1 from the other input vectors whose desired output is 0. Hence training inputs located between two neighboring hyperplanes have the same desired output.

3.2 Geometrical Learning (ETL) Algorithm

The geometrical learning algorithm called expand-and-truncate learning (ETL) is used to train a three-layer binary neural network (BNN) for the generation of binary to binary mapping. The ETL algorithm finds a set of required separating hyperplanes and determines the integer weights and integer thresholds of neurons, based on a geometrical analysis of given training inputs. ETL algorithm always guarantees convergence for any binary-to-binary mapping, automatically determining the required

number of neurons in the hidden layer, while back propagation learning algorithm [3] can not guarantee convergence and can not determine the required number of hidden neurons. Before the description of the algorithm, some preliminary concepts will be explained.

Assume that a set of n -bit input training vectors is given and desired binary output of each training vector is known. An n -bit input vector can be also considered as a vertex of an n -dimensional hypercube [6]. As mentioned before, the two classes of training input vectors (desired output of 1 and 0) can be separated by an $(n-1)$ -dimensional hyperplane which is expressed as a net function,

$$net(x, \omega_0) = \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n - \omega_0 = 0 \quad (3.4)$$

where the ω 's are constant. In this case, the set of training inputs is said to be linearly separable, and the $(n-1)$ -dimensional hyperplane is the *separating hyperplane*. This kind of hyperplane can be realized by n -input neuron with a hard-limiting activation function.

If a given binary-to-binary mapping function has the property of linear separability, then, this function can be realized by only one neuron; otherwise, more than one neurons are required to realize the function. To realize linearly inseparable switching functions, they must be decomposed into two or more linearly separable (LS) switching functions and all these LS switching functions must be combined to form the desired output.

This algorithm shows how to decompose linearly inseparable switching functions into multiple LS switching functions based on geometrical approach and combine these LS switching functions to form desired output. With this algorithm, any binary-to-binary

mapping functions can be realized by three-layer Binary Neural Network (BNN) with one hidden layer.

3.2.1 Learning The Hidden Layer

Geometrical learning algorithm is called as *expand-and-truncate learning (ETL)*. ETL is used to separate an arbitrarily linearly inseparable switching function into multiple LS switching functions. ETL will determine the required number of LS switching functions, each of which is realized by a neuron in the hidden layer.

Based on a geometrical analysis of the training inputs, ETL finds a set of hyperplanes so that inputs located between two neighboring hyperplanes have the same desired outputs. The number of neurons in the hidden layer is equal to the number of hyperplanes since separating hyperplane can be realized by a neuron with hard-limiting activation function. Fundamental ideas of the algorithm is explained by using a simple example. For this example, let us consider a switching function of three input variables $f(x_3, x_2, x_1)$. If the inputs are {001, 010, 100, 101}, then the desired output of the function is 1. If the inputs are {000, 111}, then the desired output of the function is 0. For the inputs {011, 110}, the desired output values are not cared. So there are only six training input vectors to realize the function.

An n -bit input can be considered as a vertex in an n -dimensional hypercube [6]. In our example these input vectors are the vertex of a unit cube. The vertex whose desired output is 1 is called *true vertex*. If the desired output of the vertex is 0, then this vertex is called *false vertex*.

Definition: A set of included true vertices (SITV) is a set of true vertices which can be separated from the rest vertices by a hyperplane [6].

ETL algorithm starts by selecting a true vertex. The first selected true vertex is called as a *core vertex*. The first vertex can be selected based on the clustering center found

by the modified k -nearest neighboring algorithm [6]. For this example, the first selected vertex is $\{001\}$.

Theorem 3.1: Let a set of n -bit vertices consist of a core vertex v_c and the vertices v_i 's for $i = 1 \dots n$, whose i^{th} bit is different from that of v_c (i.e., whose Hamming distance from the core vertex is 1). The following hyperplane always separates the true vertices in this set from other training vertices (i.e., false vertices in this set as well as false and true vertices whose Hamming distance from the core vertex is more than 1):

$$\omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n - \omega_0 = 0,$$

where

$$\omega_i = 1, \quad \text{if} \quad f(v_i) = 1 \text{ and } v_c^i = 1,$$

$$\omega_i = -1, \quad \text{if} \quad f(v_i) = 1 \text{ and } v_c^i = 0,$$

$$\omega_i = 2, \quad \text{if} \quad f(v_i) = 0 \text{ and } v_c^i = 1,$$

$$\omega_i = -2, \quad \text{if} \quad f(v_i) = 0 \text{ and } v_c^i = 0,$$

$$\omega_0 = \sum_{k=1}^n \omega_k v_c^k - 1$$

v_c^i indicates the i^{th} bit of the vertex v_c . The weights are assigned such that if $v_c^i = 1$ then $\omega_i > 0$, else $\omega_i < 0$.

Proof: The proof can be done by using the given weights (ω 's) and threshold (ω_0),

$$\sum_{k=1}^n \omega_k v_t^k - \omega_0 \geq 0 \quad \text{for any true vertex } v_t \text{ in the set,}$$

and

$$\sum_{k=1}^n \omega_k v_r^k - \omega_0 < 0 \quad \text{for any other training vertex } v_r.$$

Case 1: The core true vertex v_c :

$$\sum_{k=1}^n \omega_k v_c^k - \omega_0 = \sum_{k=1}^n \omega_k v_c^k - \left(\sum_{k=1}^n \omega_k v_c^k - 1 \right) > 0$$

Case 2: $f(v_i) = 1$ and $v_c^i = 1$ ($v_i^i = 0$):

$$\begin{aligned} \sum_{k=1}^n \omega_k v_i^k - \omega_0 &= \sum_{k=1}^n \omega_k v_c^k - \omega_i v_c^i - \omega_0 \\ &= \sum_{k=1}^n \omega_k v_c^k - 1 - \left(\sum_{k=1}^n \omega_k v_c^k - 1 \right) \geq 0 \end{aligned}$$

Case 3: $f(v_i) = 1$ and $v_c^i = 0$ ($v_i^i = 1$):

$$\begin{aligned} \sum_{k=1}^n \omega_k v_i^k - \omega_0 &= \sum_{k=1}^n \omega_k v_c^k + \omega_i v_i^i - \omega_0 \\ &= \sum_{k=1}^n \omega_k v_c^k - 1 - \left(\sum_{k=1}^n \omega_k v_c^k - 1 \right) \geq 0 \end{aligned}$$

Case 4: $f(v_i) = 0$ and $v_c^i = 1$ ($v_i^i = 0$):

$$\begin{aligned}
\sum_{k=1}^n \omega_k v_i^k - \omega_0 &= \sum_{k=1}^n \omega_k v_c^k - \omega_i v_c^i - \omega_0 \\
&= \sum_{k=1}^n \omega_k v_c^k - 2 - \left(\sum_{k=1}^n \omega_k v_c^k - 1 \right) < 0
\end{aligned}$$

Case 5: $f(v_i) = 0$ and $v_c^i = 0$ ($v_i^i = 1$):

$$\begin{aligned}
\sum_{k=1}^n \omega_k v_i^k - \omega_0 &= \sum_{k=1}^n \omega_k v_c^k + \omega_i v_i^i - \omega_0 \\
&= \sum_{k=1}^n \omega_k v_c^k - 2 - \left(\sum_{k=1}^n \omega_k v_c^k - 1 \right) < 0
\end{aligned}$$

Case 6: Let v_d be a vertex whose Hamming distance from the core vertex is more than 1. As the weights are assigned such that if $v_c^i = 1$ then $\omega_i > 0$, else $\omega_i < 0$,

$$\sum_{k=1}^n \omega_k v_d^k - \omega_0 \leq \sum_{k=1}^n \omega_k v_c^k - 2 - \left(\sum_{k=1}^n \omega_k v_c^k - 1 \right) < 0 \quad \text{Q.E.D.}$$

By using Theorem 3.1, the hyperplane - $x_3 - 2x_2 + 2x_1 - 1 = 0$ will separate SITV $\{001, 101\}$ from the other training vertices $\{000, 010, 100, 111\}$. The “don’t care” vertex $\{011\}$ is assumed as a false vertex while calculating the weights according to the core vertex. This hyperplane will be expanded to add to SITV possible more input vertices which produce the same output, while keeping linear separability. To choose an input vertex to be included in SITV, it is more useful to select the nearest true vertex to the vertices in SITV in the Euclidean distance sense. By selecting the nearest vertex, the probability of existing hyperplane that separates the vertices in SITV from the rest vertices, becomes higher. The nearest vertex can be easily found by

considering Hamming distance from the vertices in SITV. In the given example, the nearest vertex is selected as $\{100\}$. This vertex is called *trial vertex*. This trial vertex is added to SITV such that the hyperplane can separate the true vertices $\{001, 101, 100\}$ from the other training vertices $\{000, 010, 111\}$. To determine whether such a hyperplane exists and find the hyperplane, a geometrical approach is used.

Theorem 3.2: Consider a switching function $f : \{0,1\}^n \rightarrow \{0,1\}$. The value of f divides the 2^n points of n -tuples (i.e. 2^n vertices of n -cube) into two classes: those for which the function is 0 and those for which it is 1. A function f is linearly separable if and only if there exist a hypersphere such that all true vertices lie inside or on the hypersphere, and all false vertices lie outside, vice versa [6].

Proof: Consider a reference hypersphere (RHS).

$$\left(x_1 - \frac{1}{2}\right)^2 + \left(x_2 - \frac{1}{2}\right)^2 + \dots + \left(x_n - \frac{1}{2}\right)^2 = \frac{n}{4} \quad (3.5)$$

Notice that the center of the RHS is the center of the n -dimensional hypercube, and all the 2^n vertices are on the RHS.

Sufficiency: Suppose that only k vertices lie inside or on the hyperspace, $\sum_{i=1}^n (x_i - c_i)^2 = r^2$ and the other vertices lie outside the hypersphere. This implies that for the k vertices,

$$\sum_{i=1}^n (x_i - c_i)^2 \leq r^2 \quad (3.6)$$

and for the other vertices lying outside,

$$\sum_{i=1}^n (x_i - c_i)^2 > r^2 \quad (3.7)$$

Unless $k = 2^n$ or 0, the hypersphere must intersect with the RHS. If $k = 2^n$ or 0, all or none are true vertices. In these cases, f becomes trivial. For the nontrivial f , the intersection of the two hyperspheres must be found. When Equation (3.5) is subtracted from the Equation (3.6), the following expression will be obtained,

$$\sum_{i=1}^n (1 - 2c_i)x_i \leq r^2 - \sum_{i=1}^n c_i^2 \quad (3.8)$$

Equation (3.8) indicates that the k vertices lie on a side of or on the hyperplane, $\sum_{i=1}^n (1 - 2c_i)x_i = r^2 - \sum_{i=1}^n c_i^2$. Also, by subtracting Equation (3.5) from Equation (3.7), we can show that the other vertices lie in the other side of the same hyperplane. Therefore, the sufficiency of the theorem has been proved.

Necessity: Suppose that k true vertices lie in one side of or on the hyperplane,

$$\sum_{i=1}^n \omega_i x_i = \omega_0 \quad (3.9)$$

where the ω 's ($i = 0, \dots, n$) are arbitrary constant, and the false vertices lie in the other side.

First suppose that

$$\sum_{i=1}^n \omega_i x_i \leq \omega_0 \quad (3.10)$$

for the k true vertices and $\sum_{i=1}^n \omega_i x_i > \omega_0$ for the false vertices. As Equation (3.5) is true for any vertex, adding Equation (3.5) to Equation (3.10), we obtain

$$\sum_{i=1}^n (\omega_i x_i + x_i^2 - x_i) \leq \omega_0. \quad (3.11)$$

Equation (3.11) is true only for the k true vertices. This equation is modified to obtain,

$$\sum_{i=1}^n \left(x_i - \frac{1}{2}(1 - \omega_i) \right)^2 \leq \omega_0 + \sum_{i=1}^n \frac{1}{4}(1 - \omega_i)^2. \quad (3.12)$$

This equation indicates that these k true vertices are lie inside or on the hypersphere.

Secondly, consider that

$$\sum_{i=1}^n \omega_i x_i > \omega_0 \quad (3.13)$$

for the k false vertices. Adding Equation (3.5) to Equation (3.13) we obtain

$$\sum_{i=1}^n \left(x_i - \frac{1}{2}(1 - \omega_i) \right)^2 > \omega_0 + \sum_{i=1}^n \frac{1}{4}(1 - \omega_i)^2 . \quad (3.14)$$

This indicates that the k true vertices lie inside or on the hypersphere, and the false vertices lie outside the hypersphere.

Q.E.D.

Consider RHS and an n -dimensional hypersphere which has its radius r and its center at $\left(\frac{C_1}{C_0}, \frac{C_2}{C_0}, \dots, \frac{C_n}{C_0} \right)$. C_0 is the number of elements in SITV including trial vertex and C_i is calculated as follows:

$$C_i = \sum_{k=1}^{C_0} v_k^i \quad (3.15)$$

where v_k is an element in SITV, and v_k^i is the i^{th} bit of v_k . The point $\left(\frac{C_1}{C_0}, \frac{C_2}{C_0}, \dots, \frac{C_n}{C_0} \right)$ in the n -dimensional space represents the center of gravity of all elements in SITV.

If SITV is linearly separated from the other training vertices, there must exist a hyperspace as shown in Theorem 3.2, to include SITV and exclude the other training vertices. To find such a hypersphere, consider the hypersphere whose center is located at the center of gravity of all elements in SITV. If this hypersphere separates, this one

can do with the minimum radius. On the other hand, a hypersphere with its center away from it must have a longer radius in order to include all the elements in SITV. This will obviously increase the chance of including non-SITV elements. Hence the hypersphere with its center at the center of gravity is selected and is called as a *separating hypersphere* which is expressed as

$$\left(x_1 - \frac{C_1}{C_0}\right)^2 + \left(x_1 - \frac{C_2}{C_0}\right)^2 + \cdots + \left(x_1 - \frac{C_n}{C_0}\right)^2 = r^2. \quad (3.16)$$

When this separating hyperspace intersects RHS, an $(n - 1)$ -dimensional hyperplane is found as shown in Theorem 3.2. By subtracting Equation (3.16) from Equation (3.5) and multiplying by C_0 , the following hyperplane is obtained (ω_0 is a constant number):

$$(2C_1 - C_0)x_1 + (2C_2 - C_0)x_2 + \cdots + (2C_n - C_0)x_n - \omega_0 = 0 \quad (3.17)$$

If there exists a separating hyperplane,

$$\sum_{i=1}^n (2C_i - C_0)v_i' - \omega_0 \geq 0 \quad \text{for each vertex } v_i \text{ in SITV,}$$

and,

$$\sum_{i=1}^n (2C_i - C_0)v_i' - \omega_0 < 0 \quad \text{for each vertex } v_r \text{ from the rest vertices.}$$

Therefore, each vertex v_i in SITV and each vertex v_r satisfy the following equation:

$$\sum_{i=1}^n (2C_i - C_0)v_i' > \sum_{i=1}^n (2C_i - C_0)v_r' \quad (3.18)$$

Let t_{min} be the minimum value of $\sum_{i=1}^n (2C_i - C_0)v_i'$ among all vertices in SITV and f_{max} be the maximum of $\sum_{i=1}^n (2C_i - C_0)v_r'$ among the rest vertices.

If $t_{min} > f_{max}$, then there exist a separating hyperplane which is

$$(2C_1 - C_0)x_1 + (2C_2 - C_0)x_2 + \dots + (2C_n - C_0)x_n - \omega_0 = 0$$

where $\omega_0 = \left\lceil \frac{t_{min} + f_{max}}{2} \right\rceil$ and $\lceil x \rceil$ is the smallest integer greater than or equal to x .

If $t_{min} \leq f_{max}$, then there does not exist a separating hyperplane, thus the trial vertex is removed from SITV. For the given example, $t_{min} = \text{Minimum}(x_3 - 3x_2 + x_1)$ for SITV $\{001, 101, 100\}$, thus $t_{min} = 1$; $f_{max} = \text{Maximum}(x_3 - 3x_2 + x_1)$ for the vertices $\{000, 010, 111\}$, thus $f_{max} = 0$. Since $t_{min} > f_{max}$ and $\omega_0 = 1$, the hyperplane, $x_3 - 3x_2 + x_1 - 1 = 0$ separates the vertices in SITV $\{001, 101, 100\}$ from the rest vertices.

To include more true vertices, another true vertex is selected using the same criteria as earlier, and tested whether the new trial vertex can be added to SITV or not. This procedure continues until no more true vertices can be added to SITV. For the given example, the elements of SITV are only $\{001, 101, 100\}$. If all the true vertices are included in SITV, the given switching function is an LS switching function and only

one neuron is required for realizing the function. However, if all true vertices can not be included in SITV, more than one neurons are required for the given function.

The reason why the first hyperplane could not expand to add more true vertices to SITV, is due to the existence of false vertices around the hypersphere. That is, these false vertices prevent the expansion of the first hypersphere. In order to train more vertices, the expanded hypersphere must include the false vertices in addition to the true vertices in SITV of the first hypersphere. For this reason, false vertices are converted into true vertices, and true vertices which are not in SITV are converted into false vertices. Here the desired output for each vertex is only temporarily converted. The conversion is needed only to obtain the separating hyperplane. Now, expand the first hypersphere to add more true vertices to SITV, until no more true vertices can be added to SITV. When the expanded hypersphere meets with RHS, the second hyperplane (i.e. neuron) is found.

If SITV includes all true vertices, then the geometrical learning is converged. Otherwise, the training vertices which are not in SITV are converted again, and the same procedure repeats again. The above procedure can get stuck even when there are more true vertices still left to be included. Consider the case that when ETL tries to add any true vertex to SITV, no true vertex can be included. At this point, ETL converts the not-included true vertices and false vertices into the false vertices and true vertices, respectively. When ETL tries to include any true vertex, no true vertex can be included even after conversion. Hence the procedure is trapped and it can not proceed any more. This situation is due to the limited degree of freedom in separating hyperplanes using only integer coefficients (i.e. weights). If this situation does not occur until SITV includes all true vertices, the ETL algorithm is converged with finding all required neurons in the hidden layer.

If the above situation occurs, ETL declares these vertices in SITV as “don’t care” vertices in order to consider these vertices no more in the finding of other required neurons. Then ETL continues by selecting a new core vertex based on the clustering

center among the remaining true vertices. Until all true vertices are included, ETL proceeds in the same way as explained before. Therefore the convergence of the ETL algorithm is always guaranteed. The selection of core vertex is not unique in the process of finding separating hyperplanes. Accordingly, the number of separating hyperplanes for a given problem can vary depending upon the selection of the core vertex and the orderings of adding trial vertices. By trying all possible selections, the minimal number of separating hyperplanes can always be found.

Let us discuss on the 3-bit switching function example given before. As SITV of the first neuron includes only {001, 101, 100}, the remaining vertices are converted to expand the first hypersphere. That is, the false vertices {000, 111} are converted into true vertices, and the remaining true vertex {010} converted into a false vertex. Choose one true vertex, (for example {000}) and test if this trial vertex can be added to SITV. It turns out that SITV includes all currently declared true vertices {001, 101, 100, 000, 111}. Therefore, the algorithm is converged finding two separating hyperplanes, that is two required neurons in the hidden layer. The second required hyperplane is $x_3 - 3x_2 + x_1 + 2 = 0$. Figure 3.1 shows the structure of BNN for the given example. Table 3.1 shows the outputs of the neurons in the hidden layer for input vertices. In Table 3.1, notice that linearly inseparable input vertices are transformed into linearly separable switching function at the output of the hidden layer.

Table 3.1 The Analysis Of The Hidden Layer For The Given Example

Input Vectors	Desired Output	Hidden Layer		Output Neuron
		1 st Neuron	2 nd Neuron	
{001, 100, 101}	1	1	1	1
{000, 111}	0	0	1	0
{010}	1	0	0	1

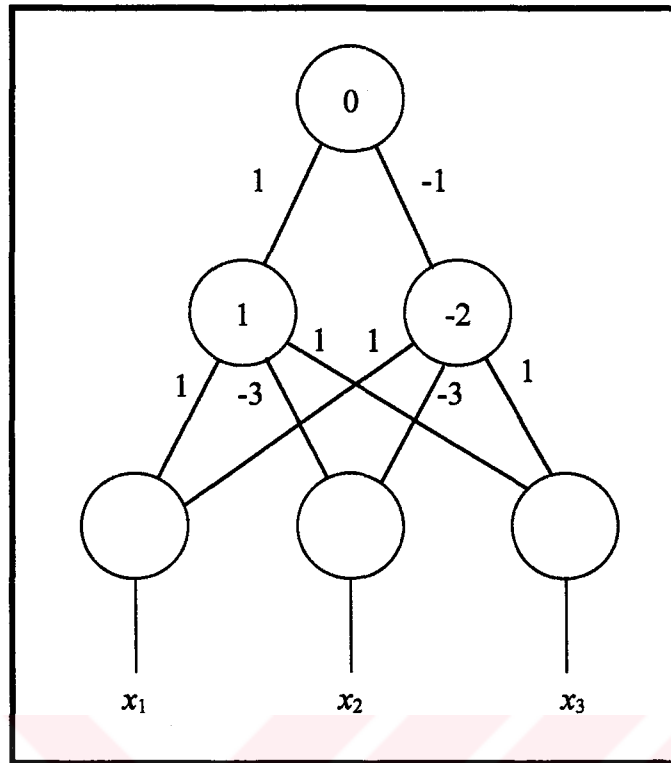


Figure 3.1 The structure of a three-layer BNN for the given example. The numbers inside circles indicate thresholds.

3.2.2 Learning The Output Layer

After all required hyperplanes (i.e., all required neurons on the hidden layer) are found, one output neuron is required in the output layer to combine the outputs of the neurons in the hidden layer. To find the weights and threshold of output neuron, the following definition will be used:

Definition: 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. If all necessary hidden neurons are found using only one core vertex, then every even-numbered hidden neuron is a converted hidden neuron, such as the second neuron in Figure 3.1.

If ETL finds all necessary separating hyperplanes with only true core vertex, the weights and threshold of the output neuron are set as follows. The weight of the link from the odd-numbered hidden neuron to the output neuron is set to 1. The weight of the link from the even-numbered neuron to the output neuron is set to -1, as each even-numbered neuron is a converted hidden neuron. By setting the threshold of the output neuron to 0 (1) if the hidden layer has an even (odd) number of neurons, the three-layer BNN always produce the desired output to each training input. Figure 3.1 shows the weights and the threshold of the output neuron for the given example.

If ETL uses more than one core vertex to find all necessary hyperplanes, the weights and threshold of the output neuron can not be determined directly as mentioned above. In this case, to find the weights and threshold of the output neuron, the following definition must be used:

Definition: A positive successive product (PSP) function is defined as a boolean function can be expressed as

$$B(h_1, h_2, \dots, h_n) = h_1 O(h_2 O(\dots O(h_{n-1} O h_n) \dots)) \quad (3.19)$$

where the operator O is either logical AND or logical OR. A PSP function can also be expressed as

$$B(h_1, h_2, \dots, h_n) = h_1 O(B(h_2, h_3, \dots, h_n)) \quad (3.20)$$

and

$$B(h_{n-1}, h_n) = h_{n-1} O h_n \quad (3.21)$$

An example of a PSP function is

$$B(h_1, h_2, \dots, h_7) = h_1 + h_2(h_3 + h_4(h_5 + h_6 h_7))$$

From the definition of a PSP function, it can be easily shown that a PSP function is always a positive unate function. Note that an LS switching function is always a unate function, but a unate function is not always an LS function.

Theorem 3.3: A PSP function is an LS switching function [6].

Proof: Express a PSP function as

$$B(h_1, h_2, \dots, h_n) = h_1 O(B(h_2, h_3, \dots, h_n))$$

then the function in the inner most nest is

$$B(h_{n-1}, h_n) = h_{n-1} O h_n$$

First, consider the case that the operator O is logical OR, i.e., $B(h_{n-1}, h_n) = h_{n-1} + h_n$.

$B(h_{n-1}, h_n)$ is obviously an LS function. Second, consider the case that the operator O

is logical AND, i.e., $B(h_{n-1}, h_n) = h_{n-1}h_n$. $B(h_{n-1}, h_n)$ is also an LS function. Therefore, the function in the inner most nest, $B(h_{n-1}, h_n)$ is always an LS function. Since the function in the inner most nest can be consider as a binary variable to the function in the in the next nest, the function in next nest is also an LS function. Continuing this process, a PSP function can be expressed as $B(h_1, h_2, \dots, h_n) = h_1 O_z$, where z is a binary variable corresponding to $B(h_1, h_2, \dots, h_n)$. Q.E.D.

Theorem 3.3 means that a neuron with a hard-limiting activation function can map any PSP function since a PSP function is a LS function. Using a PSP function, an output neuron function can be expressed as the function of the outputs of the hidden neurons.

A neuron is supposed to assign one to the side of a hyperplane having true vertices, and zero to the other side. However, in ETL, a converted hidden neuron assigns one to the side of a hyperplane having original false vertices and zero to the other side having original true vertices. Therefore, without transforming the outputs of converted hidden neurons, an output neuron function can not be a PSP function of the outputs of hidden neurons. In order to make a PSP function, the output of each converted hidden neuron is complemented and fed into the output neuron. Complementing the output of a converted hidden neuron is identical to multiplying by (-1) the weight from this neuron to the output neuron and subtracting this weight from the threshold of the output neuron. That is, if the output neuron is realized by the weight-threshold $\{\omega_1, \omega_2, \dots, \omega_j, \dots, \omega_n; \omega_0\}$ whose inputs are $h_1, h_2, \dots, h_j, \dots, h_n$, than the output neuron is also realized by weight-threshold $\{\omega_1, \omega_2, \dots, -\omega_j, \dots, \omega_n; \omega_0 - \omega_j\}$ whose inputs are $h_1, h_2, \dots, h_j, \dots, h_n$.

Theorem 3.4: After the hidden neurons are determined by ETL, an output neuron function can always be expressed as PSP function of the outputs of hidden neurons if the output of each converted hidden neuron is complemented.

Proof: Without loss of generality, let us assume that ETL finds i_1 hidden neurons $\{n_{11}, n_{12}, \dots, n_{1i_1}\}$ from the first core vertex, i_2 hidden neurons $\{n_{21}, n_{22}, \dots, n_{2i_2}\}$ from the second core vertex, and i_k hidden neurons $\{n_{k1}, n_{k2}, \dots, n_{ki_k}\}$ from the k^{th} core vertex. Let h_{ij} be either the output of the n_{ij} neuron if j is an odd number, or the complemented output of the n_{ij} neuron if j is an even number (i.e. n_{ij} is a converted hidden neuron). The first neuron n_{11} separates only true vertices. Hence if $h_{11} = 1$, then the output of the output neuron should be one regardless of the outputs of other hidden neurons. Therefore, the output neuron function can be expressed as

$$B(h_{11}, h_{12}, \dots, h_{ki_k}) = h_{11} + \left(B(h_{12}, \dots, h_{ki_k}) \right),$$

representing a logical OR operation.

The second neuron n_{12} separates only false vertices. Thus the side of a hyperplane for $h_{12} = 1$ includes true vertices as well as false vertices, and true vertices will be separated by the following hidden neurons. Note that the true vertices which are not separated by n_{11} are located only in the side of a hyperplane for $h_{12} = 1$. Therefore, the output neuron function can be expressed as

$$\begin{aligned} B(h_{11}, h_{12}, \dots, h_{ki_k}) &= h_{11} + \left(B(h_{12}, \dots, h_{ki_k}) \right) \\ &= h_{11} + h_{12} \left(B(h_{13}, \dots, h_{ki_k}) \right), \end{aligned}$$

representing a logical AND operation.

Now this expressions can be generalized for a neuron n_{ij} as follows.

If j is an odd number, then $B(h_j, h_{j+1}, \dots, h_{k_k}) = h_j + B(h_{j+1}, \dots, h_{k_k})$, representing a logical OR operation, and if j is an even number, then $B(h_j, h_{j+1}, \dots, h_{k_k}) = h_j \left(B(h_{j+1}, \dots, h_{k_k}) \right)$, representing a logical AND operation.

Therefore, the output neuron function can always be expressed as a PSP function

$$B(h_{11}, h_{12}, \dots, h_{k_k}) = h_{11} O \left(h_{12} O \left(\dots O \left(h_{k_{k-1}} O h_{k_k} \right) \right) \dots \right),$$

where the operator O following h_{ij} indicates logical OR if j is an odd number, or indicates logical AND if j is an even number. Q.E.D.

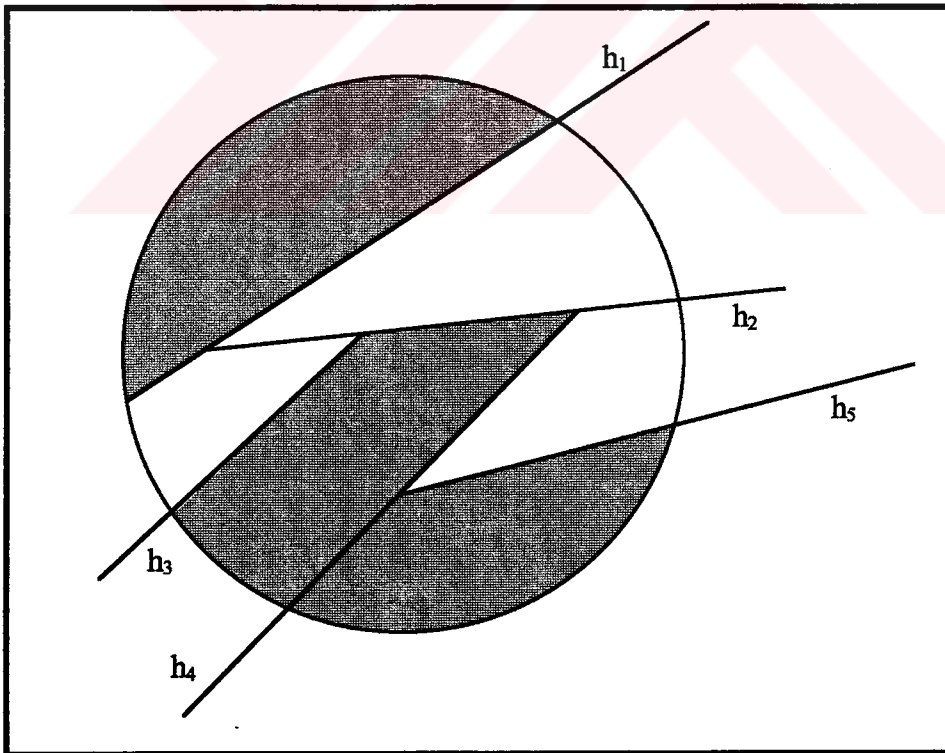


Figure 3.2 Input vectors are partitioned by ETL.

As an example, consider Figure 3.2 where only the dashed region requires the desired output as one. In Figure 3.2, h_1 separates ones, thus the OR operation follows. The same thing is true for h_4 . As h_2 separates zero in Figure 3.2, the AND operation follows. The same things are true for h_3 . Therefore, the output can be expressed by PSP function as

$$B(h_1, h_2, h_3, h_4, h_5) = h_1 + h_2(h_3(h_4 + h_5)). \quad (3.22)$$

Theorem 3.4 shows that an output neuron function is an LS function of the outputs of hidden neurons. The way to determine the weights of the output neuron is to find a PSP function, and then transform the PSP function into the net function. For an n variable PSP function $f(h_1, h_2, \dots, h_n)$, there exist a systematic method to generate a net function, $net(H, \omega)$. The systematic method is given next.

First, the method starts from the innermost net function net_n . The net_n is set to $h_n - 1$ since $net_n \geq 0$ if $h_n = 1$ and $net_n < 0$ if $h_n = 0$. Let us find the next net function net_{n-1} . If the operation between h_n and h_{n-1} is a logical OR, then

$$net_{n-1} = (-\min[net_n])h_{n-1} + net_n, \quad (3.23)$$

where $\min[net_n]$ is the minimum value of net_n . Since $\min[net_n] = \min[h_n - 1] = -1$, $net_{n-1} = h_{n-1} + h_n - 1$.

If the operation between h_n and h_{n-1} is a logical AND, then

$$net_{n-1} = (\max[net_n] + 1)h_{n-1} + net_n - (\max[net_n] + 1), \quad (3.24)$$

where $\max[net_n]$ is the maximum value of net_n . Since $\max[net_n] = \max[h_n - 1] = 0$, $net_{n-1} = h_{n-1} + h_n - 2$.

Continuing this process until n becomes one, the net function $net(H, \omega_0)$ is determined. The connection weight between the output neuron and the i^{th} hidden neuron is the coefficient of h_i in the net function, and the threshold of the output neuron is the constant in the net function.

Let us consider Equation (3.22) to generate a net function from a PSP function,

$$\begin{aligned} net_5 &= h_5 - 1 \\ net_4 &= (-\min[net_5])h_4 + net_5 = h_4 + h_5 - 1 \\ net_3 &= (\max[net_4] + 1)h_3 + net_4 - (\max[net_4] + 1) = 2h_3 + h_4 + h_5 - 3 \\ net_2 &= (\max[net_3] + 1)h_2 + net_3 - (\max[net_3] + 1) = 2h_2 + 2h_3 + h_4 + h_5 - 5 \\ net_1 &= (-\min[net_2])h_1 + net_2 = 5h_1 + 2h_2 + 2h_3 + h_4 + h_5 - 5. \end{aligned}$$

Therefore, the net function for Equation (3.22) is expressed as

$$net(H, \omega_0) = 5h_1 + 2h_2 + 2h_3 + h_4 + h_5 - 5$$

Notice that if $B(x_1, x_2, \dots, x_n) = 1$, then $net(X, \omega_0) \geq 0$, else $net(X, \omega_0) < 0$.

From the above discussion the following theorem can be stated:

Theorem 3.5: For any generation of binary-to-binary mapping, the ETL algorithm always converges and finds the three-layer BNN whose hidden layer has as many neurons as separating hyperplanes.

3.3 Modified ETL Algorithm

The ETL algorithm has some algorithmic restrictions. In some cases, the ETL algorithm could not find the desired number of neurons even if a very efficient core selection algorithm is used. For example, consider the 5-input switching function 69631 (i.e. if the input vertices are $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 16\}$, then the desired output is 1; otherwise, the desired output is 0). The Chow parameters of this function are

$$b_5 = -22, b_4 = -10, b_3 = -10, b_2 = -2, b_1 = -2, b_0 = -6.$$

The ordered absolute value of these parameters are (22,10,10,6,2,2) which indicates that this switching function has the property of linear separability (see Appendix A), thus, it must be realized by only one neuron. ETL can realize this switching function by two neurons, even if it is linearly separable. Core vertex selection does not effect the determining of the weights of the neuron when realizing the linear separable switching functions. To overcome this algorithmic restriction and to improve the ability of finding possible minimum number of neurons of ETL, we must modify some fundamentals of the algorithm. This new algorithm is called as *modified expand-and-truncate learning algorithm (METL)*.

METL contains the following modifications:

1. Arranging and reducing the switching function to obtain its equivalent reduced switching function that is called as *minfunction*.

2. Prediction of maximum number of neurons that the minfunction can be realized by.
3. Determining the starting vertex type.
4. Determining the core vertex and calculating initial weights.
5. Modification of weights in the cases $t_{min} < f_{max}$ and $t_{min} = f_{max}$.
6. Determining the weights of output neuron by using a *neuron-type vector*.

METL can realize n -dimensional binary-to-binary mapping by maximum n -hidden neurons. The possibility of realizing n -dimensional switching functions by more than n -hidden neurons is smaller than 1% and the reason of this case depends on the selection of the core vertex and the starting point. In the next section, all these modifications are explained in detail.

3.3.1 Modification of Input Vectors

Assume that an n -dimensional net function $net_{f^n}(X, \omega_0) = \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n - \omega_0 = 0$ is given. Consider the case of increasing input dimension so that if $x_{n+1} = 1$, the $f^{n+1}(x_i) = 0$ and if $x_{n+1} = 0$, the $f^{n+1}(x_i) = f^n(x_i)$. This $(n+1)$ -dimensional switching function can be realized by setting $\omega_{n+1} = -\max(net_{f^n}(X, \omega_0)) - 1$ and the ω_i 's $i = 1, \dots, n$ equal to the weights of $net_{f^n}(X, \omega_0)$. So, if some of the inputs of the input vectors that have the same desired outputs, do not change (all 0 or 1), these inputs do not effect the linear separability and the weights of the neurons can be determined without considering into these inputs. To determine such an $(n+1)$ -dimensional reducible functions the following definitions will be used.

Definition: V_{minor} is a indicator of minority of vertices of a switching function. If the number of true vertices is less than or equal to the number of false vertices, than $V_{minor} = 1$, otherwise $V_{minor} = 0$. In symbols,

$$\begin{aligned}
 V_{\text{minor}} &= 1 && \text{if } \pi(f^{-1}(1)) \leq \pi(f^{-1}(0)), \\
 V_{\text{minor}} &= 0 && \text{if } \pi(f^{-1}(1)) > \pi(f^{-1}(0)),
 \end{aligned}$$

Definition: *Minor vertex* is a kind of vertex whose desired output is equal to V_{minor} .

Definition: *Major vertex* is a kind of vertex whose desired output is equal to $1 - V_{\text{minor}}$.

Consider an n -dimensional switching function. If the order of inputs and their corresponding desired outputs are changed, a new function will be obtained. This new function has the same characteristics with the first function and can be realized by the same weights and thresholds after the corresponding changes are made. These two functions are called *equivalent functions*.

Example: Consider two threshold functions as shown in Table 3.2.

Table 3.2 Truth Tables of Threshold Functions

$x_3x_2x_1$	f_1^3	$x_3x_2x_1$	f_2^3
000	1	000	1
001	1	001	1
010	1	010	0
011	0	011	0
100	0	100	1
101	0	101	0
110	0	110	0
111	0	111	0

If the order of inputs of the function f_2^3 are changed as $x_2x_3x_1$, the desired outputs of the functions become identical, so these functions are equivalent functions. The net function of f_1^3 is calculated as

$$net_{f_1^3}(X, \omega_0) = -4x_3 - 2x_2 - 2x_1 + 3 = 0.$$

As desired output of the function f_2^3 is identical with the desired output of the function f_1^3 when the order of inputs changed, the net function of the function f_2^3 can be obtained by making the same order changes to the weights. The net function of f_2^3 is obtained as

$$net_{f_2^3}(X, \omega_0) = -2x_3 - 4x_2 - 2x_1 + 3 = 0.$$

For determining whether or not a given n -dimensional switching function can be reduced to $(n-1)$ -dimensional reduced switching function, first, find cubical complex of major vertices. If there is $(n-1)$ -cube in the cubical complex of major vertices, then, this function can be reduced, otherwise it can not be reduced. In the given example, cubical complex of major vertices of f_1^3 is $\{011, 1^{**}\}$. Since this cubical complex contains 2-cube, this function can be reduced to 2-dimensional switching function.

We can also determine this property by taken minor vertices into consideration. First, form a set of n -bit minor vertices. If some of the bits of all these vertices in the set do not change, then this function can be arranged and reduced; otherwise, the function can not be reduced. In the given above example, this set is formed as $\{000, 001, 010\}$. In this set, only the input x_3 remains the same ($x_3 = 0$) so this function can be reduced.

For reducing process, first, determine the unchanged bits (i.e. inputs) of minor vertices and note their values. Then, eliminate these inputs from the input vectors. In the given example, this input is x_3 and its value is 0. After the elimination, the set has the elements of $\{00, 01, 10\}$ which belong to 2-dimensional switching function, thus, the dimension of the function is reduced. Table 3.3 shows the original switching function and the reduced switching function.

Table 3.3 Original Switching Function and Reduced Switching Function

$x_3x_2x_1$	f_1^3	x_2x_1	f_1^2
000	1	00	1
001	1	01	1
010	1	10	1
011	0	11	0
100	0		
101	0		
110	0		
111	0		

After all these reducing processes, determine the net function of the reduced function. In the given example, the net function is $net(X, \omega) = -2x_2 - 2x_1 + 3 = 0$. Notice that this net function realizes only the reduced function. To realize the original function, we must calculate the weights of the inputs that are eliminated. The following theorem describes the way of calculation.

Theorem 3.6: Consider an n -dimensional linearly separable switching function that can be reduced to $(n-1)$ -dimensional linearly separable switching function. If the eliminated input is x_i and the desired output of the minor vertices is V_{minor} , then the following net function always realizes the n -dimensional switching function.

$$net_{f^n}(X, \omega'_0) = \omega_i x_i + net_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0) + \omega_0 - \omega'_0 \quad (3.25)$$

where

$$\omega_i = -\min \left[net_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0) \right] + 1, \quad \text{if } V_{\text{minor}} = 0 \text{ and } v_m^i = 0,$$

$$\omega'_0 = \omega_0$$

$$\omega_i = \min \left[\text{net}_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0) \right] - 1, \quad \text{if } V_{\text{minor}} = 0 \text{ and } v_m^i = 1,$$

$$\omega'_0 = \omega_0 + \omega_i$$

$$\omega_i = -\max \left[\text{net}_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0) \right] - 1, \quad \text{if } V_{\text{minor}} = 1 \text{ and } v_m^i = 0,$$

$$\omega'_0 = \omega_0$$

$$\omega_i = \max \left[\text{net}_{f^{n-1}}(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n, \omega_0) \right] + 1, \quad \text{if } V_{\text{minor}} = 1 \text{ and } v_m^i = 1,$$

$$\omega'_0 = \omega_0 + \omega_i$$

v_m^i indicates the i^{th} bit of the minor vertex v_m .

Proof: The proof can be done by showing that with the given weights (ω 's). As the function f^n can be reduced to the function f^{n-1} the desired output of minor vertices of f^n is equal to the desired output of the corresponding vertices of f^{n-1} . Note that v_m indicates n -bit minor vertex, v_n indicates any n -bit input vector.

$$\text{net}_{f^n}(X, \omega_0) = \sum_{j=1}^n \omega_j v_n^j - \omega_0$$

$$\text{net}_{f^{n-1}}(X, \omega_0) = \sum_{j=1}^{i-1} \omega_j v_n^j + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0$$

Case 1: $V_{\text{minor}} = 0$ and $v_m^i = 0$, (i.e. if $v_n^i = 0$, then $f^n(v_n) = f^{n-1}(v_{n-1})$, else $f^n(v_n) = 1$).

a. $v_n^i = 0$.

$$\begin{aligned} \text{net}_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\ &= \sum_{j=1}^{i-1} \omega_j v_n^j + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 = \text{net}_{f^{n-1}}(X, \omega_0). \end{aligned}$$

b. $v_n^i = 1$.

$$\begin{aligned}
 net_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\
 &= \sum_{j=1}^{i-1} \omega_j v_n^j - \min[net_{f^{n-1}}] + 1 + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega_0 \\
 &= net_{f^{n-1}} - \min[net_{f^{n-1}}] + 1 \geq \min[net_{f^{n-1}}] - \min[net_{f^{n-1}}] + 1 > 0.
 \end{aligned}$$

Case 2: $V_{\text{minor}} = 0$ and $v_n^i = 1$, (i.e. if $v_n^i = 1$, then $f^n(v_n) = f^{n-1}(v_{n-1})$, else $f^n(v_n) = 1$).

a. $v_n^i = 0$.

$$\begin{aligned}
 net_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\
 &= \sum_{j=1}^{i-1} \omega_j v_n^j + \sum_{j=i+1}^n \omega_j v_n^j - \min[net_{f^{n-1}}] + 1 - \omega_0 \\
 &= net_{f^{n-1}} - \min[net_{f^{n-1}}] + 1 \geq \min[net_{f^{n-1}}] - \min[net_{f^{n-1}}] + 1 > 0.
 \end{aligned}$$

b. $v_n^i = 1$.

$$\begin{aligned}
 net_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\
 &= \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - (\omega_0 + \omega_i) = net_{f^{n-1}}(X, \omega_0).
 \end{aligned}$$

Case 3: $V_{\text{minor}} = 1$ and $v_n^i = 0$, (i.e. if $v_n^i = 0$, then $f^n(v_n) = f^{n-1}(v_{n-1})$, else $f^n(v_n) = 0$).

a. $v_n^i = 0$.

$$\begin{aligned}
net_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\
&= \sum_{j=1}^{i-1} \omega_j v_n^j + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 = net_{f^{n-1}}(X, \omega_0).
\end{aligned}$$

b. $v_n^i = 1$.

$$\begin{aligned}
net_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\
&= \sum_{j=1}^{i-1} \omega_j v_n^j - \max[net_{f^{n-1}}] - 1 + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega_0 \\
&= net_{f^{n-1}} - \max[net_{f^{n-1}}] - 1 \leq \max[net_{f^{n-1}}] - \max[net_{f^{n-1}}] - 1 < 0.
\end{aligned}$$

Case 4: $V_{\text{minor}} = 1$ and $v_n^i = 1$, (i.e. if $v_n^i = 1$, then $f^n(v_n) = f^{n-1}(v_{n-1})$, else $f^n(v_n) = 0$).

a. $v_n^i = 0$.

$$\begin{aligned}
net_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\
&= \sum_{j=1}^{i-1} \omega_j v_n^j + \sum_{j=i+1}^n \omega_j v_n^j - \max[net_{f^{n-1}}] - 1 - \omega_0 \\
&= net_{f^{n-1}} - \max[net_{f^{n-1}}] - 1 \leq \max[net_{f^{n-1}}] - \max[net_{f^{n-1}}] - 1 < 0.
\end{aligned}$$

b. $v_n^i = 1$.

$$\begin{aligned}
net_{f^n}(X, \omega'_0) &= \sum_{j=1}^n \omega_j v_n^j - \omega'_0 = \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i v_n^i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - \omega'_0 \\
&= \sum_{j=1}^{i-1} \omega_j v_n^j + \omega_i + \sum_{j=i+1}^n \omega_j v_n^j - \omega_0 + \omega_0 - (\omega_0 + \omega_i) = net_{f^{n-1}}(X, \omega_0).
\end{aligned}$$

Q.E.D.

Theorem 3.6 can be also applied to f^{n-1} if it can be reduced to f^{n-2} . This reduction process stops when the last net function can not be reduced. Since the linearly inseparable switching functions can be decomposed into multiple LS switching functions, Theorem 3.6 can be applied to all these LS functions in the same way.

In the given above example, f_1^2 can also be reduced to f_1^1 . To do this, first, form the set of minor vertices. Since this set has only the element of $\{11\}$, we can not determine the unchanged bit. In this case, select the first input on the left as unchanged input. Here, this unchanged input is x_2 and eliminate this input after setting $V_{\text{minor}} = 0$ and $v_m^2 = 1$. Since f_1^1 can not be reduced, calculate the weights and threshold of neuron that realizes f_1^1 . Then, in accordance with the Theorem 3.6, find the net function of f_1^3 .

This process help us to calculate the weights and thresholds of the neurons that realize n -dimensional binary-to-binary mapping quickly and to find possible minimum number of neurons. In some cases, ETL algorithm finds the required number of neurons that realize f^n , more than the required number of neurons that realize f^{n-1} if f^n can be reduced to f^{n-1} . METL algorithm always guarantees the same required number of neurons that realize f^n and f^{n-1} .

3.3.2 Prediction of Maximum Required Number of Neurons

METL uses three starting vertex type to find optimal required number of neurons to realize any n -dimensional binary-to-binary mapping. For the determination of starting vertex type, the maximum number of neurons must be known. This section describes the way of finding the maximum required number of neurons that can realize n -dimensional binary-to-binary mapping.

By using Karnaugh-maps any n -dimensional binary-to-binary mapping can be realized by 2^{n-1} neurons. This is the general limit of number of neurons. According to the

structure of an n -dimensional switching function, the maximum required number of neurons can be expressed by the following equation.

$$N_{v_{\max}} = \min \left[2^n - \sum_{i=1}^n f^n(v_i), \sum_{i=1}^n f^n(v_i) \right] \quad (3.26)$$

The Equation (3.26) indicates that the maximum number of neurons is equal to the total number of minor vertices. This is true as anybody can form the neurons each for the minor vertices and combine all these neurons with an output neuron behaves as AND gate if $V_{\text{minor}} = 0$ or as OR gate if $V_{\text{minor}} = 1$.

The following method can be used to find the maximum number of neurons. This method can also be used to decompose the linearly inseparable switching functions into multiple LS switching functions. The method is formed based on Karnaugh-maps, and is called as *LS Decomposition Method (LSD)*. LSD method decomposes any switching function into one or multiple LS switching functions. After the decomposition process, the net functions of these LS functions can be calculated by using one of learning algorithms for binary-to-binary mappings.

The LSD method starts by forming a set of minor vertices. After this set is formed, find consecutive pairs of each vertex in this set. To do this, first, select the first minor vertex as *group leader* and find other minor vertices whose Hamming distance from the group leader is 1 and group all these vertices. These vertices can be called as *neighbors of group leader*. Then, select the next minor vertex to form a new group and find its neighbors. Continue this process until the last minor vertex is selected and grouped. List these groups in an order and determine the number of elements in each group without considering the group leader (i.e. the number of neighbors of each group leader). After all the number of neighbors are determined, find the group whose number of neighbors is the smallest and positive (i.e. not equal to zero). Analyze all the vertex in that group to find a group that has the maximum number of neighbors

and whose group leader is the element of analyzed group. Then, note the group leader and the number of neighbors of this group and analyze the next group if there are more than 1 groups that have the minimum number of neighbors. Repeat this process until all the groups that have the minimum positive number of neighbors, are analyzed. Select a group leader that has the maximum number of neighbors among the noted vertices. The selected group is the first LS function and can be realized by one neuron. To find the next LS function, eliminate all the elements of the found group from the other groups and repeat these processes until no minor vertices left. If the number of elements of some of the groups is 0 (i.e. the group has the element of only the group leader), these vertices must be realized by individual neurons. In some cases, selecting the group that has the maximum number of neighbors causes the required number of neurons to increase, thus, LSD method can not find the minimum number of neurons but limits the required number of neurons.

Example 1: Consider a 4-dimensional switching function as shown in Figure 3.3.

	x_2'	x_2'	x_2	x_2		
x_4'					x_3'	Minor Vertices
x_4'	1	1	1	1	x_3	{4,5,6,7,15}
x_4			1		x_3	Major Vertices
x_4					x_3'	{0,1,2,3,8,9,10,11,12,13,14}
	x_1'	x_1	x_1	x_1'		

Figure 3.3 Karnaugh-map of the given example.

According to the structure of this function, we can see that $V_{\text{minor}} = 1$ and minor vertices are {4, 5, 6, 7, 15}. The minor vertices whose Hamming distance from the group leader is 1, are grouped. $HD(4,5) = HD(5,7) = HD(7,15) = HD(6,7) = HD(4,6) = 1$. If all these groups are written on a table, Table 3.4 is formed.

In Table 3.4, select the vertex 15 whose number of neighbors is 1. Note the number of neighbors of the vertex 7 that is the neighbor of the vertex 15. Since there are no other vertices that has the minimum number of neighbors, the group, whose group leader is the vertex 7, is selected as the first LS function. Then, remove the vertices {7,5,6,15} from the table. After that process, we obtain Table 3.5.

Table 3.4 Table of Neighbors of The Given Example

Group Leader	Neighbors	Number of Neighbors
4	5, 6	2
5	4, 7	2
6	4, 7	2
7	5, 6, 15	3
15	7	1

Table 3.5 Table of Neighbors After The Elimination Process

Group Leader	Neighbors	Number of Neighbors
4		0

Since there is only one group leader, select this group to form the second LS function. Thus, the function is decomposed into two LS functions {7,5,6,15} and {4}. As mentioned before, LSD method can not find the minimum number of neurons. If these two LS functions are analyzed, it can be seen that only one neuron is sufficient to realize this function, thus, this function has the property of linear separability.

Example 2: Consider a 4-dimensional function as shown in Figure 3.4. According to the structure of this function, we can see that $V_{\text{minor}} = 1$ and minor vertices are {0,2,3,5,7,9,10}. The minor vertices whose Hamming distance from the group leader is 1, are grouped. If all these groups are written on a table, Table 3.6 is formed.

In Table 3.6, select the vertex 0 whose number of neighbors is 1. Note the number of neighbors of the vertex 2 which is the neighbor of the vertex 0. Then, select the group leader 5 and note the number of neighbors of the vertex 7 which is the neighbor the vertex 5. Selection of the vertex 10 is not necessary as the neighbor of the vertex 10 is equal to the neighbor of the vertex 0. Since the number of neighbors of the vertex 2 is greater than the number of neighbors of the vertex 7, the group, whose group leader is the vertex 2, is selected as first LS function. Then, remove the vertices $\{2,0,3,10\}$ from the table. After that process, we obtain Table 3.7.

	x_2'	x_2'	x_2	x_2		
x_4'	1		1	1	x_3'	Minor Vertices
x_4'		1	1		x_3	$\{0,2,3,5,7,9,10\}$
x_4		1		1	x_3	Major Vertices
x_4					x_3'	$\{1,4,6,8,11,12,13,14,15\}$
	x_1'	x_1	x_1	x_1'		

Figure 3.4 Karnaugh-map of the given example

Table 3.6 Table of Neighbors of The Given Example

Group Leader	Neighbors	Number of Neighbors
0	2	1
2	0,3,10	3
3	2,7	2
5	7	1
7	3,5	2
9		0
10	2	1

Table 3.7 Table of Neighbors After The Elimination Process

Group Leader	Neighbors	Number of Neighbors
5	7	1
7	5	1
9		0

After the similar processes, the second LS function is found as {5,7} and the third LS function is found as {9} since the vertex 9 has no neighbors. Thus, the function can be realized by 3 neurons.

By using LSD method, the maximum number of neurons that can realize the known function is expressed as N_{LSDmax} .

The last limitation of required number of neurons depends on the tests on METL algorithm. According to the results of tests on METL, any n -dimensional binary-to-binary mapping can be realized by maximum n -neurons. To prove this limitation, consider the following example whose Karnaugh-map is given in Figure 3.5.

This is the worst case of distribution of true vertices. If the vertices {0001} and {1110} are converted into true vertices, the new function can be realized by 2 neurons. We must remove the changes we made and this can be done by another 2 neurons. Thus, the function can be realized by 4 neurons which is equal to the dimension of input vectors. Linearly separable blocks on Karnaugh-maps are listed in Appendix B.

From the above discussion this limitation can be expressed as

$$N_{\text{dimmax}} = n \quad (3.27)$$

If the switching function can be reduced to m -dimensional switching function, then, Equation (3.27) becomes

$$N_{\text{dimmax}} = m \quad (3.28)$$

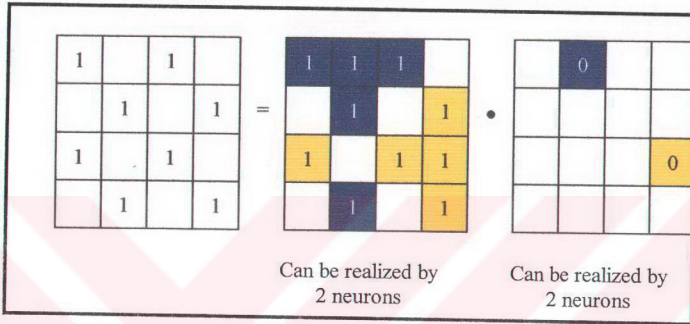


Figure 3.5 Decomposition of the Karnaugh-map of the given example into 4 LS functions.

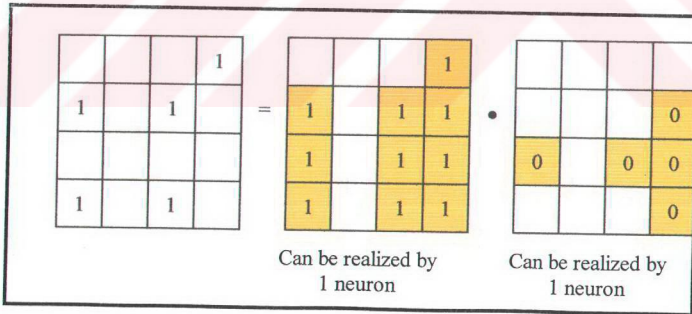


Figure 3.6 Decomposition of the Karnaugh-map of a switching function into 2 LS functions.

When all these limitations are combined, the following maximum number of neurons that can realize the function will be obtained:

$$N_{\max} = \min[N_{v\max}, N_{LSD\max}, N_{\dim\max}] \quad (3.29)$$

3.3.3 Starting Vertex Type

ETL algorithm selects the core vertex among the true vertices. In some cases, this selection increases the number of neurons and also increases the calculation time. Consider the following switching function whose Karnaugh-map is given in Figure 3.7.

If we select the core vertex among true vertices, we can select {0010} as core vertex (red vertex in yellow part in Figure 3.7). (Core selection method will be given in the next section). Since Hamming distance of this vertex from the other true vertices is greater than 1, only the core vertex is included in SITV. After the weights of the first neuron is calculated, the false vertices are converted into true vertices and true vertices into false vertices. Now the vertices, whose Hamming distance from the core vertex is 1, can be included in SITV. At the end, SITV has the vertices {0010, 0011, 0110, 0000, 0001, 1010} (yellow and blue parts in Figure 3.7) and again ETL converts the vertices into original values after the weights of second neuron are calculated. Now all the true vertices can be included in SITV (yellow, blue and red parts in Figure 3.7). Thus, this function is realized by 3 neurons if we select the core vertex among true vertices.

If we select the core vertex among false vertices, we can select {1110} as core vertex (red vertex in yellow parts in Figure 3.8). As there are many vertices, whose Hamming distance from the core vertex is 1, the vertices {1110, 0110, 1010, 1111, 1100, 1101} can be included in SITV (yellow parts in Figure 3.8). After the weights

of the first neuron are calculated, all the remaining vertices are converted into opposite type. In this case, all the false vertices that are originally true vertices can be included in SITV (yellow and blue parts in Figure 3.8). Thus, the function is realized by 2 neurons.

As shown in the above example, selection of vertex type effects the number of neurons. To realize a switching function by possible minimum number of neurons, the following method can be used.

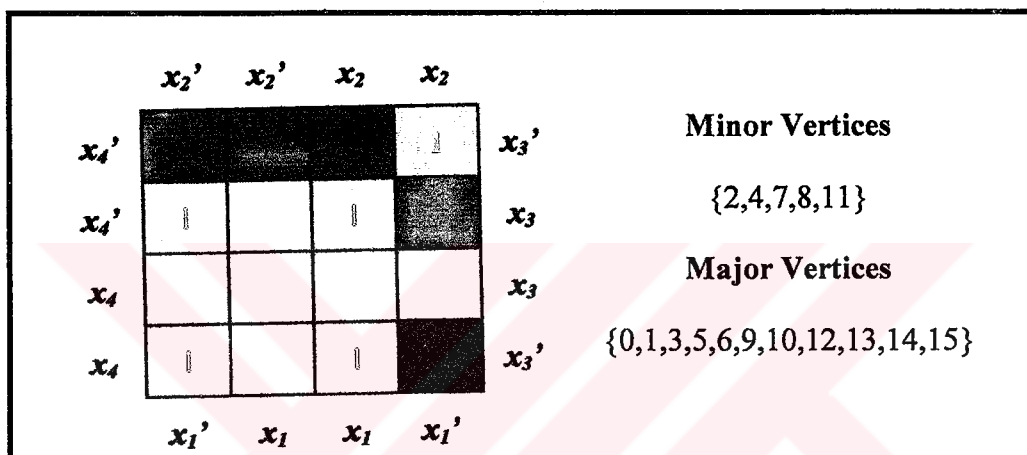


Figure 3.7 Karnaugh-map of the given example.

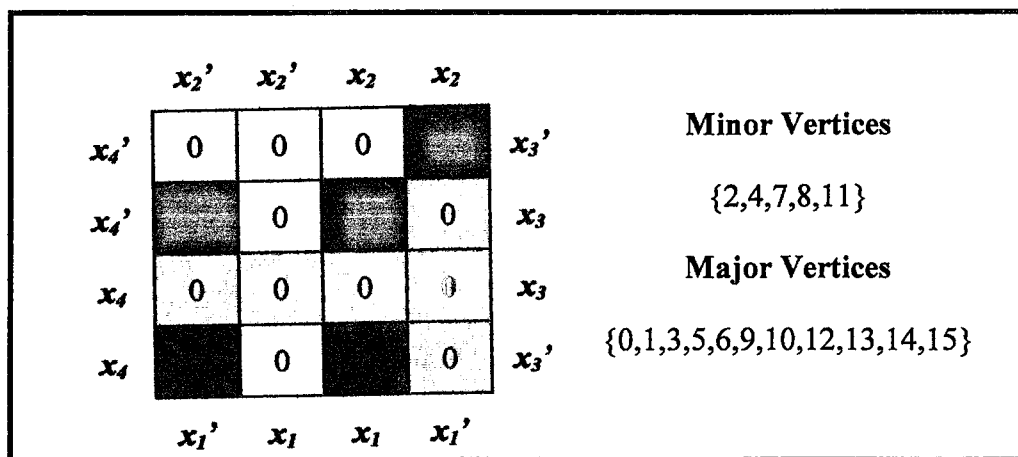


Figure 3.8 Karnaugh-map of the given example.

First of all, find the number of neighbors of all vertices. Then, find the vertex that the number of neighbors is the highest. The starting vertex type is the type of the found vertex. If the found vertex is a false vertex, then the starting vertex type is false, otherwise, the starting vertex type is true. If there are more than one vertices that the number of neighbors is the highest, and these vertices have the different desired outputs, then, select the starting vertex type as the type of minor vertices.

After the starting vertex type is determined, the core vertex must be selected and several core vertex selection methods are mentioned in the next section.

3.3.4 Selection of Core Vertex and Determining the Initial Weights

There are several methods to select the core vertex. Each of them has some benefits to realize the switching function by minimum number of neurons. According to the structure of the switching function, an appropriate method can be used to reach the minimum number of neurons.

Method 1. Popular Vertex

The method that is used for finding starting vertex type can be used to select the core vertex. In this method, we select the vertex that has the highest number of neighbors as a core vertex. First of all, find the number of neighbors of all vertices, then find the vertex that the number of neighbors is the highest. If there are more than 1 vertices that the number of neighbors is the highest and these vertices have different desired outputs, then, select the smallest minor vertex as the core vertex. This method help us to realize some of the functions by minimum number of neurons but not all of them.

Method 2. Quick LSD

This method is formed based on LSD method (Section 4.3.2) but the method stops when the first LS function is found. The group leader of this LS function is selected as the core vertex.

Method 3. Center of Gravity

After the starting vertex type is determined, find the center of gravity of these type of vertices in accordance with the following equation.

$$C_i = \sum_{j=1}^k v_j^i \quad (3.30)$$

where k is the total number of the vertices and assign the bits of the core vertex as

$$\begin{aligned} v_c^i &= 1 & \text{if} & \quad 2C_i - k \geq 0 \\ v_c^i &= 0 & \text{if} & \quad 2C_i - k < 0 \end{aligned}$$

If the formed core vertex is not in the set, then find the nearest vertex whose Hamming distance from the formed core vertex and select this vertex as the core vertex.

After the selection of the core vertex by using one of these methods or another one, the initial weights can be found in accordance with the following equations. In VLSI applications, to prevent the effects of device tolerances, the weights are set so that none of the vertices lie on the hyperplane.

The modified net function is formed as

$$\omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n - \omega_0 = 0, \quad (3.31)$$

where

$$\omega_i = 2, \quad \text{if} \quad f(v_i) = 1 \text{ and } v_c^i = 1,$$

$$\omega_i = -2, \quad \text{if} \quad f(v_i) = 1 \text{ and } v_c^i = 0,$$

$$\omega_i = 4, \quad \text{if} \quad f(v_i) = 0 \text{ and } v_c^i = 1,$$

$$\omega_i = -4, \quad \text{if} \quad f(v_i) = 0 \text{ and } v_c^i = 0,$$

$$\omega_0 = \sum_{k=1}^n \omega_k v_c^k - 3$$

v_c^i indicates the i^{th} bit of the vertex v_c . The weights are assigned such that if $v_c^i = 1$ then $\omega_i > 0$, else $\omega_i < 0$. Note that $\text{HD}(v_i, v_c) = 1$ and v_c is the core vertex.

The parameters (i.e. weights) of the hyperplane that separates expanded SITV from the rest of the vertices, are also modified. ETL finds the center of gravity of expanded SITV and forms the hyperplane in accordance with the Equation (3.17). To separate SITV from the rest vertices strictly, this Equation (3.17) is modified as

$$(4C_1 - 2C_0)x_1 + (4C_2 - 2C_0)x_2 + \dots + (4C_n - 2C_0)x_n - \omega_0 = 0 \quad (3.32)$$

If there exists a separating hyperplane, the weights and threshold are assigned as

$$\omega_i = (4C_i - 2C_0) \quad i = 1 \dots n \quad (3.33a)$$

$$t_{\min} = \min \left[\sum_{i=1}^n \omega_i v_i^i \right] \quad (3.33b)$$

$$f_{\max} = \max \left[\sum_{i=1}^n \omega_i v_r^i \right] \quad (3.33c)$$

$$\omega_0 = \frac{t_{\min} + f_{\max}}{2} \quad (3.33d)$$

3.3.5 Arranging The Weights When $t_{\min} < f_{\max}$ and $t_{\min} = f_{\max}$

ETL algorithm excludes the trial vertex from the SITV in the cases $t_{\min} < f_{\max}$ and $t_{\min} = f_{\max}$. This process causes some linearly separable functions to be realized by two neurons. While realizing linearly separable functions, the selection of the core vertex and determining the starting vertex type do not effect the results since all the true or false vertices must be included in SITV, and calculations for previous trial vertices do not effect the calculations of next trial vertices. The reason that ETL can not realize some linearly separable functions by one neuron is due to the restriction of ETL algorithm. To overcome this restriction, the following method must be used in the cases $t_{\min} < f_{\max}$ and $t_{\min} = f_{\max}$.

First, find minimum value of $\sum_{i=1}^n \omega_i v_i^i$ among all vertices in SITV, note the vertices that realize this minimum value (t_{\min}) and select the vertex which realizes t_{\min} and which has the minimum number of ones (i.e. Hamming distance from the vertex 0 is minimum) among all the vertices that realize t_{\min} . Then find maximum value of $\sum_{i=1}^n \omega_i v_r^i$ among the rest vertices, note the vertices that realize this maximum value (f_{\max}) and select the vertex which realizes f_{\max} and which has the minimum number of ones among all the vertices that realize f_{\max} . We call the selected vertex that realizes t_{\min} as $v_{t_{\min}}$ and the selected vertex that realizes f_{\max} as $v_{f_{\max}}$. The modification of the weights in the cases $t_{\min} < f_{\max}$ and $t_{\min} = f_{\max}$ is described next.

Case 1: $t_{\min} = f_{\max}$

In this case, modify all the weights in accordance with the following equation.

$$\omega_i' = \omega_i + 2v_{t_{\min}}^i - 2v_{f_{\max}}^i \quad (3.34)$$

Equation (3.34) forces t_{\min} to increase and f_{\max} to decrease so that we can place the hyperplane between the vertices in SITV and the rest vertices.

Case 2: $t_{\min} < f_{\max}$ and $|f_{\max} - t_{\min}| < Dif$

In this case, we define a new variable to hold the difference between f_{\max} and t_{\min} . When a new trial vertex is included in SITV, Dif is set to $+\infty$. If $|f_{\max} - t_{\min}| < Dif$ then set $Dif = f_{\max} - t_{\min}$ and modify the weights in accordance with the Equation (3.34), otherwise this trial vertex can not be included in SITV.

If the weights are modified, find the minimum value of $\sum_{i=1}^n \omega_i v_t^i$ and the maximum value of $\sum_{i=1}^n \omega_i v_r^i$ and repeat the modification process until $t_{\min} > f_{\max}$ or the trial vertex can not be included in SITV.

To realize a function by possible minimum number of neurons, this modification must be applied if no true vertices can be included in SITV. If this modification method manages to include a trial vertex, then try all true vertices to include in SITV without using this method and repeat this process until the next neuron is required.

Example: Consider a 5-dimensional switching function as shown in Figure 3.9.

Suppose that SITV has included all the true vertices except the vertex 7 and the algorithm starts to include the trial vertex 7. C_i 's are calculated as

$$C_0 = 13, C_1 = -1, C_2 = -1, C_3 = -5, C_4 = -5, C_5 = -11$$

and the net function of SITV is

$$-2x_1 - 2x_2 - 10x_3 - 10x_4 - 22x_5 - \omega_0 = 0.$$

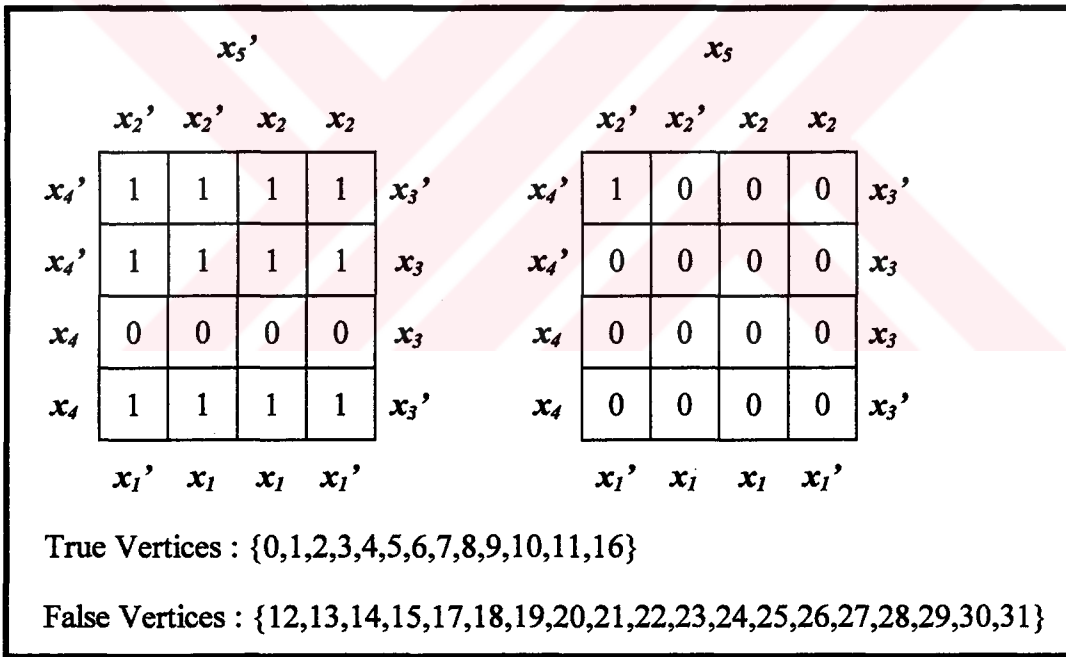


Figure 3.9 Karnaugh-map of the given example.

Since $t_{\min} = -22 < f_{\max} = -20$, ETL algorithm excludes the vertex 7 from SITV and realizes the function by 2 neurons. In this case, modify the weights in accordance with the Equation (3.34). Note that $v_{\min} = 16$ and $v_{\max} = 12$. The new net function is obtained as

$$-2x_1 - 2x_2 - 12x_3 - 12x_4 - 20x_5 - \omega_0 = 0.$$

If we calculate t_{\min} and f_{\max} , we can see that $t_{\min} = -20 > f_{\max} = -22$. Thus, all true vertices can be included in SITV and the function can be realized by only one neuron whose weights are equal to the modified weights.

3.3.6 Learning The Output Layer

After all required hyperplanes (i.e. neurons) are found, the weights and threshold of the output neuron must be determined. ETL uses a PSP function and a systematic method to calculate the weights. However, METL uses a neuron-type vector T instead of PSP function and a modified systematic method. Neuron-type vector is formed based on the type of hidden neurons. To form the vector $T = (t_1 t_2 \dots t_n)$, the following definition is used.

Definition: If a neuron is determined based on true vertices, this neuron is called as *true-type neuron*. If a neuron is determined based on false vertices then this neuron is called as *false-type neuron*.

If i^{th} neuron is a true-type neuron then set $T^i = 1$. If i^{th} neuron is a false-type neuron then set $T^i = 0$. After forming the vector T , apply the following systematic method.

First, the method starts from the innermost net function net_n . The net_n is set to $2h_n - 1$ since $net_n \geq 0$ if $h_n = 1$ and $net_n < 0$ if $h_n = 0$. Let us find the next net function net_{n-1} . If $T^{n-1} = 1$, then

$$net_{n-1} = (-\min[net_n] + 1)h_{n-1} + net_n, \quad (3.35)$$

where $\min[net_n]$ is the minimum value of net_n . Since $\min[net_n] = \min[2h_n - 1] = -1$,
 $net_{n-1} = 2h_{n-1} + 2h_n - 1$.

If $T^{n-1} = 0$, then

$$net_{n-1} = (\max[net_n] + 1)h_{n-1} + net_n - (\max[net_n] + 1), \quad (3.36)$$

where $\max[net_n]$ is the maximum value of net_n . Since $\max[net_n] = \max[2h_n - 1] = 1$,
 $net_{n-1} = 2h_{n-1} + 2h_n - 3$.

Continuing this process until n becomes one, the net function $net(H, \omega_0)$ is determined. The connection weight between the output neuron and the i^{th} hidden neuron is the coefficient of h_i in the net function, and the threshold of the output neuron is the constant in the net function. This systematic method is formed based on Theorem 3.6.

Example: Suppose that a 6-dimensional switching function is realized by 6 neurons and its neuron-type vector is $(t_1 t_2 t_3 t_4 t_5 t_6) = (010110)$. This vector indicates that the core vertex is selected among the false vertices. The net function of output neuron of this function is calculated as

$$net_6 = 2h_6 - 1$$

$$net_5 = (\max[net_6] + 1)h_5 + net_6 - (\max[net_6] + 1) = 2h_5 + 2h_6 - 3$$

$$net_4 = (\max[net_5] + 1)h_4 + net_5 - (\max[net_5] + 1) = 2h_4 + 2h_5 + 2h_6 - 5$$

$$net_3 = (-\min[net_4] + 1)h_3 + net_4 = 6h_3 + 2h_4 + 2h_5 + 2h_6 - 5$$

$$net_2 = (\max[net_3] + 1)h_2 + net_3 - (\max[net_3] + 1) = 8h_2 + 6h_3 + 2h_4 + 2h_5 + 2h_6 - 13$$

$$net_1 = (-\min[net_2] + 1)h_1 + net_2 = 14h_1 + 8h_2 + 6h_3 + 2h_4 + 2h_5 + 2h_6 - 13$$

3.3.7 Determination of Appropriate Starting Vertex Type

Starting vertex type can be changed if the selected type can not manage to realize an n -bit binary-to-binary mapping by maximum n neurons. If the core vertex is selected among the minor vertices (Minor Start METL) and this function can not be realized by maximum number of neurons by which this function must be realized, then, select a core vertex among the major vertices (Major Start METL) and start the algorithm at the beginning. If again the function can not be realized by maximum number of neurons, then select the core vertex among the minor vertices by using Quick LSD method, and apply the algorithm until SITV can not be expanded to include trial vertices. Find the weights of the first neuron and switch to second neuron. For the second neuron, reset SITV and select another core vertex by using Quick LSD among the remaining minor vertices and expand SITV to include trial vertices. Continue this process until no more minor vertices left.

If a switching function can be realized by more than N_{\max} by using Quick LSD method, this switching function must be realized by using Major Start METL or Minor Start METL.

3.3.8 METL Algorithm Steps

- Step 1) Find the number of true vertices and false vertices.
- Step 2) Determine the type and unchanged bits of minor vertices.
- Step 3) Reduce the switching function.
- Step 4) Repeat Step 1 to Step 3 until the function can not be reduced.
- Step 5) Find N_{\max} that must realize the switching function.
- Step 6) Find the number of neighbors of minor vertices and the rest vertices.
- Step 7) Determine the appropriate starting vertex type.

- Step 8) Select a core vertex by using one of selection methods and find initial weights in accordance with the Equation (3.31).
- Step 9) Select a nearest true vertex as a trial vertex and calculate the weights. Since the false vertices are converted into true vertices to include in SITV, the trial vertex is always a true vertex.
- Step 10) Find t_{\min} and f_{\max} in accordance with the Equations (3.33b) and (3.33c).
- Step 11) Modify the weights in accordance with the Equation (3.34) if $t_{\min} = f_{\max}$ or $t_{\min} < f_{\max}$.
- Step 12) Repeat Step 10 and Step 11 until $t_{\min} > f_{\max}$ or modification can not be done.
- Step 13) If $t_{\min} > f_{\max}$, include the trial vertex in SITV and calculate the weights in accordance with the Equations (3.33a) and (3.33d). If $t_{\min} < f_{\max}$, then, exclude the trial vertex from SITV.
- Step 14) Repeat the steps 9 to 13 until no more trial vertices can be included in SITV or all the true vertices are included in SITV.
- Step 15) If all the true vertices (original or converted) are not included in SITV, then, convert all the vertices that are not in SITV and switch to next neuron.
- Step 16) Repeat the steps 9 to 15 until all the original or converted true vertices are included in SITV.
- Step 17) Return to original function from the reduced one by arranging and calculating the corresponding weights by using Theorem 3.6.
- Step 18) Determine the weights of output neuron.

3.4 Realization Performance of METL Algorithm

A computer program is used to find the realization performance of METL Algorithm. The whole program code, which is written in Turbo Pascal, is listed in Appendix C.

Program uses the following neuron model during the calculations of weights and thresholds:

$$y = f(\xi)$$

$$\xi = \sum_{i=1}^n \omega_i x_i + \omega_0$$

where x_i , $i = 1, \dots, n$, represent inputs, ω_i , $i = 1, \dots, n$, represent weights, ω_0 denotes a threshold and y denotes the output of a neuron. The activation function $f(\xi)$ is described by

$$f(\xi) = 0 \quad \text{if } \xi < 0$$

$$f(\xi) = 1 \quad \text{if } \xi > 0$$

Since computers have memory limits and Turbo Pascal denotes the numbers by maximum 32 bits, program can realize maximum 11-dimensional switching functions by required number of neurons.

Program has three starting type algorithm. These are Minor Start METL, Major Start METL and Eliminating METL. Minor Start METL algorithm selects the core vertex among minor vertices. Major Start METL algorithm selects the core vertex among major vertices. Eliminating METL selects the core vertex by using Quick LSD method and expand SITV to include more minor vertices. this algorithm do not convert any of the vertices. If no minor vertices can be included in SITV, the elements of SITV are assigned as “don’t care” and a new SITV is formed among the rest vertices by using the same criteria.

Table 3.8 shows the total test results of METL Algorithm and Table 3.9 shows the test results of 5-dimensional switching functions. According to the test results, METL managed to realize over 1,000,000 switching functions by maximum 5 neurons that is equal to the dimension of input vectors. The worst cases of distribution of true vertices of 5-dimensional, 6-dimensional, 7-dimensional and 8-dimensional switching functions are also tested and METL can manage to realize these functions by 5, 6, 7, and 8 hidden neurons, respectively.

According to the programming structure, program may realize a switching function by number of neurons that is greater than the dimension of its input vectors. This case may be occur as the program uses strict rules for selecting core vertex and this selection may not be an optimal one.



Table 3.8 Test Results of METL Algorithm

Input Dim.	Solution OK	Solution None	Linearly Separable Functions	2 Neurons	3 Neurons	4 Neurons	5 Neurons	Total Number of Functions Tested	Total Number of Functions	Maximum Number of Neurons	Function Number of MaxNeuron
1	4	-	4	-	-	-	-	4	4	1	1
2	16	-	14	2	-	-	-	16	16	2	6
3	256	-	104	150	2	-	-	256	256	3	105
4	65,536	-	1,882	35,733	27,764	157	-	65,536	65,536	4	9,939
5	1,623,841	-	3,703	246,702	1,056,626	311,147	5,663	1,623,841	4,294,967,296	5	103,348

Table 3.9 Test Results of 5-Dimensional Switching Functions

Start	End	Total Funcs	OK	None	1 Nrm	2 Nrms	3 Nrms	4 Nrms	5 Nrms	Total	MaxNeuron	NeuronFuncNo
0	65,535	65,536	65,536	-	1,882	35,733	27,764	157	-	65,536	4	9,939
65,536	166,260	100,725	100,725	-	280	22,967	67,728	9,727	23	100,725	5	103,348
166,261	267,710	101,450	101,450	-	213	18,785	66,527	15,844	81	101,450	5	169,525
267,711	465,592	197,882	197,882	-	237	27,156	131,358	38,854	277	197,882	5	275,811
465,593	581,400	115,808	115,808	-	150	18,086	77,329	19,873	370	115,808	5	465,618
581,401	772,900	191,500	191,500	-	173	23,188	127,658	39,758	723	191,500	5	596,657
772,901	981,240	208,340	208,340	-	218	26,160	135,984	45,269	709	208,340	5	772,920
981,241	1,087,575	106,335	106,335	-	261	20,081	70,414	15,469	110	106,335	5	981,349
1,087,576	1,320,000	232,425	232,425	-	188	30,811	156,603	44,073	750	232,425	5	1,095,890
1,320,001	1,368,500	48,500	48,500	-	0	3,205	32,887	12,208	200	48,500	5	1,320,250

CONCLUSIONS

In this study, a learning algorithm called *Modified Expand-and-Truncate Learning (METL) Algorithm* is developed to train three-layer BNN for any binary-to-binary mapping problem (switching function). This algorithm always converges and finds a solution to realize any n -dimensional switching function by maximum n neurons. The learning speed of METL is much faster than the other learning algorithms for the generation of switching functions since METL calculates the weights and thresholds directly. The other algorithms require high number of iterations for each training vector while METL requires usually 1 or 2 iterations for each training vector. The neuron in the BNN employs a hard-limiting activation function, only integer weights and integer thresholds. Therefore, this will greatly facilitate actual hardware implementation of the BNN using currently available VLSI technology.

Since METL finds a solution to realize any n -dimensional switching function by maximum n neurons, CTL based programmable logic arrays (CTL-CTL PLA) can be implemented on a very small layout area. Traditional NOR-NOR PLA requires 2^{n-1} lines in the first NOR array to realize any n -dimensional switching function, thus the layout area exponentially increases with the dimension of the inputs. For example, 32-bit general purpose NOR-NOR PLA requires 2,147,483,648 lines in the first NOR array, but 32-bit general purpose CTL-CTL PLA (three-layer binary neural network) requires only 32 neurons in the hidden layer.

The learning speed of METL decreases as the dimension of input vectors increases since the training vectors increases exponentially. For example, a 32-input switching function has 4,294,967,296 training vectors which require long time to calculate the weights and thresholds. Several dedicated machines that work in parallel can be used to calculate these weights and thresholds. This algorithm can be modified to use for higher-valued logic (for example 3-valued logic).

REFERENCES

- [1] GÖKDUMAN, İ.; *CTL Kapı Tasarımı*, İTÜ Elektrik - Elektronik Fakültesi Elektronik ve Haberleşme Bölümü, Bitirme Ödevi, 1994
- [2] ÖZDEMİR, H.; KEPKEP, A.; PAMİR, B.; LEBLEBİCİ Y.; ÇİLİNGİROĞLU, U.; *A Capacitive Threshold-Logic Gate*, Journal of Solid State Circuits, pp. 1141-1150, Vol. 31, No. 8, August 1996.
- [3] ÇİLİNGİROĞLU, U.; *Neural Networks and Fuzzy Systems*, İstanbul Teknik Üniversitesi İleri Elektronik Teknolojileri Araştırma Geliştirme Vakfı, 15 Aralık 1993.
- [4] ROSENBLATT, R.; *Principles of Neurodynamics*, New York, Spartan Books, 1959.
- [5] GRAY, D. L.; MICHEL, A. N.; *A Training Algorithm for Binary Feedforward Neural Networks*, IEEE Transactions on Neural Networks, pp. 176-194, Vol. 3, No. 2, (March 1992).
- [6] KIM, J. H.; PARK, S.; *The Geometrical Learning of Binary Neural Networks*, IEEE Transactions on Neural Networks, pp. 237-247, Vol. 6, No. 1, (January 1995).
- [7] COTTER, N. E.; *The Stone-Weierstrass Theorem and its Application to Neural Networks*, IEEE Transactions on Neural Networks, Dec. 1990.
- [8] HU, S.; *Threshold Logic*, University of California Press, Berkeley and Los Angeles 1965.
- [9] HURST, S.L.; *The Logical Processing of Digital Signals*, Crane, Russak & Company, Inc., New York, 1978.

APPENDIX A

CHOW PARAMETER CLASSIFICATIONS FOR ALL LINEARLY SEPARABLE BINARY FUNCTIONS OF $n \leq 6$

Notes

- (1) For any n -binary function $f(y)$ with binary inputs y_i , $i = 1, \dots, n$, $f(y)$, $y_i \in \{-1, 1\}$, the Chow parameters are defined as

$$b_i = \sum_{x \in (F \cup F^n)} f(y) y_i, \quad i = 0, \dots, n$$

where $y_0 = 1$ ¹.

- (2) The canonic tables list the $|b_i|$ values for all the linearly separable functions in descending magnitude order. Each entry uniquely defines one and only one standard (or “representative”) function. If the Chow parameters for any non-linearly-separable function are computed, the resultant parameter values will not be found in these standard tabulations.
- (3) The minimum integer realizing weight/threshold values $|a_i|$, $i = 1, \dots, n$, are tabulated against each $|b_i|$ classification entry. Although the maximum, co-equal, and minimum values of the $|a_i|$ ’s reflect the maximum, co-equal, and minimum values of the $|b_i|$ ’s, there is no simple arithmetic relationship between them.
- (4) For any chosen set of gate input weights a_1 to a_n , the resultant gate-threshold value is given by taking the remaining tabulated a_i , value ($=a_0$) and evaluating

¹ HURST, S.L., *The Logical Processing of Digital Signals*, Crane, Russak & Company, Inc., New York, 1978. p530 - 536.

$$t = \frac{1}{2} \left\{ \left(\sum_{i=1}^n a_i \right) - a_0 + 1 \right\}.$$

- (5) Notice that all the entries for any n appear in the subsequent tabulation for $n + 1$, but with all values multiplied by 2 in the latter and with a further zero-valued component. The multiplication by 2 is because there are twice the number of minterms present in the $n + 1$ case compared with the n -valued-case.

Table A.1 Table of Chow Parameters

n	$ b_i $					$ a_i $				
$n \leq 3$										
1	8	0	0	0		1	0	0	0	
2	6	2	2	2		2	1	1	1	
3	4	4	4	0		1	1	1	0	
$n \leq 4$										
1	16	0	0	0	0	1	0	0	0	0
2	14	2	2	2	2	3	1	1	1	1
3	12	4	4	4	0	2	1	1	1	0
4	10	6	6	2	2	3	2	2	1	1
5	8	8	8	0	0	1	1	1	0	0
6	8	8	4	4	4	2	2	1	1	1
7	6	6	6	6	6	1	1	1	1	1
$n \leq 5$										
1	32	0	0	0	0	0	1	0	0	0
2	30	2	2	2	2	2	4	1	1	1
3	28	4	4	4	4	0	3	1	1	1
4	26	6	6	6	2	2	5	2	2	2
5	24	8	8	4	4	4	4	2	2	1
6	24	8	8	8	0	0	2	1	1	1
7	22	10	10	6	2	2	5	3	3	2
8	22	10	6	6	6	6	3	2	1	1
9	20	12	12	4	4	0	3	2	2	1
10	20	12	8	8	4	4	4	3	2	2
11	20	8	8	8	8	8	2	1	1	1
12	18	14	14	2	2	2	4	3	3	1
13	18	14	10	6	6	2	5	4	3	2
14	18	10	10	10	6	6	3	2	2	2
15	16	16	16	0	0	0	1	1	1	0
16	16	16	12	4	4	4	3	3	2	1
17	16	16	8	8	8	0	2	2	1	1
18	16	12	12	8	8	4	4	3	3	2
19	14	14	14	6	6	6	2	2	2	1
20	14	14	10	10	10	2	3	3	2	2
21	12	12	12	12	12	0	1	1	1	1

Table A.1 Continued

n	$ b_i $							$ a_i $						
$n \leq 6$														
1	64	0	0	0	0	0	0	1	0	0	0	0	0	0
2	62	2	2	2	2	2	2	5	1	1	1	1	1	1
3	60	4	4	4	4	4	0	4	1	1	1	1	1	0
4	58	6	6	6	6	2	2	7	2	2	2	2	1	1
5	56	8	8	8	8	0	0	3	1	1	1	1	0	0
6	56	8	8	8	4	4	4	6	2	2	2	1	1	1
7	54	10	10	10	6	2	2	8	3	3	3	2	1	1
8	54	10	10	6	6	6	6	5	2	2	1	1	1	1
9	52	12	12	12	4	4	0	5	2	2	2	1	1	0
10	52	12	12	8	8	4	4	7	3	3	2	2	1	1
11	52	12	8	8	8	8	8	4	2	1	1	1	1	1
12	50	14	14	14	2	2	2	7	3	3	3	1	1	1
13	50	14	14	10	6	6	2	9	4	4	3	2	2	1
14	50	14	10	10	10	6	6	6	3	2	2	2	1	1
15	50	10	10	10	10	10	10	3	1	1	1	1	1	1
16	48	16	16	16	0	0	0	2	1	1	1	0	0	0
17	48	16	16	12	4	4	4	6	3	3	2	1	1	1
18	48	16	16	8	8	8	0	4	2	2	1	1	1	0
19	48	16	12	12	8	8	4	8	4	3	3	2	2	1
20	48	12	12	12	12	8	8	5	2	2	2	2	1	1
21	46	18	18	14	2	2	2	7	4	4	3	1	1	1
22	46	18	18	10	6	6	2	9	5	5	3	2	2	1
23	46	18	14	14	6	6	6	5	3	2	2	1	1	1
24	46	18	14	10	10	10	2	7	4	3	2	2	2	1
25	46	14	14	14	10	10	6	7	3	3	3	2	2	1
26	44	20	20	12	4	4	0	5	3	3	2	1	1	0
27	44	20	20	8	8	4	4	7	4	4	2	2	1	1
28	44	20	16	16	4	4	4	6	4	3	3	1	1	1
29	44	20	16	12	8	8	4	8	5	4	3	2	2	1
30	44	20	12	12	12	12	0	3	2	1	1	1	1	0
31	44	16	16	16	8	8	8	4	2	2	2	1	1	1
32	44	16	16	12	12	12	4	6	3	3	2	2	2	1
33	42	22	22	10	6	2	2	8	5	5	3	2	1	1
34	42	22	22	6	6	6	6	5	3	3	1	1	1	1
35	42	22	18	14	6	6	2	9	6	5	4	2	2	1
36	42	22	18	10	10	6	6	6	4	3	2	2	1	1
37	42	22	14	14	10	10	2	7	5	3	3	2	2	1
38	42	18	18	18	6	6	6	5	3	3	3	1	1	1
39	42	18	18	14	10	10	6	7	4	4	3	2	2	1
40	42	18	14	14	14	14	2	5	3	2	2	2	2	1
41	40	24	24	8	8	0	0	3	2	2	1	1	0	0
42	40	24	24	8	4	4	4	6	4	4	2	1	1	1
43	40	24	20	12	8	4	4	7	5	4	3	2	1	1
44	40	24	20	8	8	8	8	4	3	2	1	1	1	1
45	40	24	16	16	8	8	0	4	3	2	2	1	1	0
46	40	24	16	12	12	8	4	8	6	4	3	3	2	1
47	40	20	20	16	8	8	4	8	5	5	4	2	2	1
48	40	20	20	12	12	8	8	5	3	3	2	2	1	1
49	40	20	16	16	12	12	4	6	4	3	3	2	2	1
50	40	16	16	16	16	16	0	2	1	1	1	1	1	0
51	38	26	26	6	6	2	2	7	5	5	2	2	1	1

Table A.1 Continued

n	$ b_i $							$ a_i $						
52	38	26	22	10	10	2	2	8	6	5	3	3	1	1
53	38	26	22	10	6	6	6	5	4	3	2	1	1	1
54	38	26	18	14	10	6	2	9	7	5	4	3	2	1
55	38	26	18	10	10	10	6	6	5	3	2	2	2	1
56	38	26	14	14	14	6	6	5	4	2	2	2	1	1
57	38	22	22	14	10	6	6	6	4	4	3	2	1	1
58	38	22	22	10	10	10	10	3	2	2	1	1	1	1
59	38	22	18	18	10	10	2	7	5	4	4	2	2	1
60	38	22	18	14	14	10	6	7	5	4	3	3	2	1
61	38	18	18	18	14	14	2	5	3	3	3	2	2	1
62	36	28	28	4	4	4	0	4	3	3	1	1	1	0
63	36	28	24	8	8	4	4	6	5	4	2	2	1	1
64	36	28	20	12	12	4	0	5	4	3	2	2	1	0
65	36	28	20	12	8	8	4	7	6	4	3	2	2	1
66	36	28	16	16	12	4	4	6	5	3	3	2	1	1
67	36	28	16	12	12	8	8	8	7	4	3	3	2	2
68	36	24	24	12	12	4	4	7	5	5	3	3	1	1
69	36	24	24	12	8	8	8	4	3	3	2	1	1	1
70	36	24	20	16	12	8	4	8	6	5	4	3	2	1
71	36	24	20	12	12	12	8	5	4	3	2	2	2	1
72	36	24	16	16	16	8	8	4	3	2	2	2	1	1
73	36	20	20	20	12	12	0	3	2	2	2	1	1	0
74	36	20	20	16	16	12	4	6	4	4	3	3	2	1
75	34	30	30	2	2	2	2	5	4	4	1	1	1	1
76	34	30	26	6	6	6	2	7	6	5	2	2	2	1
77	34	30	22	10	10	6	2	8	7	5	3	3	2	1
78	34	30	18	14	14	2	2	7	6	4	3	3	1	1
79	34	30	18	14	10	6	6	9	8	5	4	3	2	2
80	34	30	14	14	10	10	10	7	6	3	3	2	2	2
81	34	26	26	10	10	6	6	5	4	4	2	2	1	1
82	34	26	22	14	14	6	2	9	7	6	4	4	2	1
83	34	26	22	14	10	10	6	6	5	4	3	2	2	1
84	34	26	18	18	14	6	6	5	4	3	3	2	1	1
85	34	26	18	14	14	10	10	6	5	4	3	3	2	2
86	34	22	22	18	14	10	2	7	5	5	4	3	2	1
87	34	22	22	14	14	14	6	4	3	3	2	2	2	1
88	34	22	18	18	18	10	6	5	4	3	3	3	2	1
89	32	32	32	0	0	0	0	1	1	1	0	0	0	0
90	32	32	28	4	4	4	4	4	4	3	1	1	1	1
91	32	32	24	8	8	8	0	3	3	2	1	1	1	0
92	32	32	20	12	12	4	4	5	5	3	2	2	1	1
93	32	32	16	16	16	0	0	2	2	1	1	1	0	0
94	32	32	16	16	8	8	8	4	4	2	2	1	1	1
95	32	32	12	12	12	12	12	3	3	1	1	1	1	1
96	32	28	28	8	8	8	4	6	5	5	2	2	2	1
97	32	28	24	12	12	8	4	7	6	5	3	3	2	1
98	32	28	20	16	16	4	4	6	5	4	3	3	1	1
99	32	28	20	16	12	8	8	7	6	5	4	3	2	2
100	32	28	16	16	12	12	12	5	4	3	3	2	2	2
101	32	24	24	16	16	8	0	4	3	3	2	2	1	0
102	32	24	24	16	12	12	4	5	4	4	3	2	2	1
103	32	24	20	20	16	8	4	6	5	4	4	3	2	1

Table A.1 Continued

n	$ b_i $							$ a_i $						
104	32	24	20	16	16	12	8	7	6	5	4	4	3	2
105	32	20	20	20	20	8	8	3	2	2	2	2	1	1
106	30	30	30	6	6	6	6	3	3	3	1	1	1	1
107	30	30	26	10	10	10	2	5	5	4	2	2	2	1
108	30	30	22	14	14	6	6	4	4	3	2	2	1	1
109	30	30	18	18	18	2	2	5	5	3	3	3	1	1
110	30	30	18	18	10	10	10	3	3	2	2	1	1	1
111	30	30	14	14	14	14	14	2	2	1	1	1	1	1
112	30	26	26	14	14	10	2	6	5	5	3	3	2	1
113	30	26	22	18	18	6	2	7	6	5	4	4	2	1
114	30	26	22	18	14	10	6	8	7	6	5	4	3	2
115	30	26	18	18	14	14	10	6	5	4	4	3	3	2
116	30	22	22	22	18	6	6	4	3	3	3	2	1	1
117	30	22	22	18	18	10	10	5	4	4	3	3	2	2
118	28	28	28	12	12	12	0	2	2	2	1	1	1	0
119	28	28	24	16	16	8	4	5	5	4	3	3	2	1
120	28	28	20	20	20	4	0	3	3	2	2	2	1	0
121	28	28	20	20	12	12	8	4	4	3	3	2	2	1
122	28	28	16	16	16	16	12	3	3	2	2	2	2	1
123	28	24	24	20	20	4	4	5	4	4	3	3	1	1
124	28	24	24	20	16	8	8	6	5	5	4	3	2	2
125	28	24	20	20	16	12	12	7	6	5	5	4	3	3
126	26	26	26	18	18	6	6	3	3	3	2	2	1	1
127	26	26	22	22	22	2	2	4	4	3	3	3	1	1
128	26	26	22	22	14	10	10	5	5	4	4	3	2	2
129	26	26	18	18	18	14	14	4	4	3	3	3	2	2
130	26	22	22	22	14	14	14	4	3	3	3	2	2	2
131	24	24	24	24	24	0	0	1	1	1	1	1	0	0
132	24	24	24	24	12	12	12	2	2	2	2	1	1	1
133	24	24	20	20	16	16	16	5	5	4	4	3	3	3
134	22	22	22	18	18	18	18	3	3	3	2	2	2	2
135	20	20	20	20	20	20	20	1	1	1	1	1	1	1

APPENDIX B

THE CHARACTERISTIC KARNAUGH MAP PATTERNS FOR ALL LINEARLY SEPARABLE FUNCTIONS OF $n \leq 4$

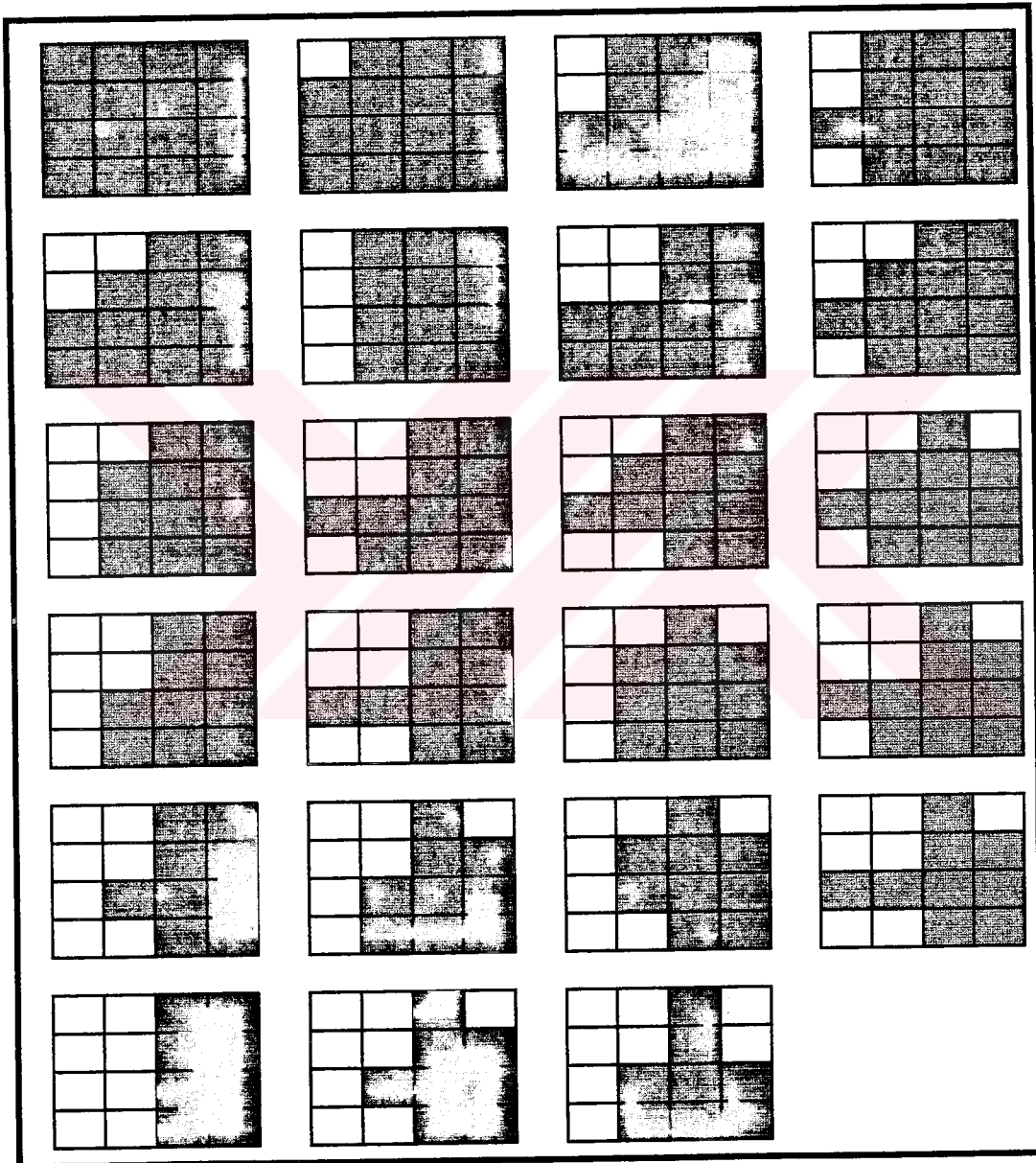


Figure B.1 The positive canonic linearly separable map patterns of $n \leq 4$. The position may be anywhere and with any orientation within the map layout.

APPENDIX C

SOURCE CODE OF THE COMPUTER PROGRAM

```
{G+} {286 instructions are on}
{IFDEF CPU87}
{N+} {80x87 support is on}
{ELSE}
{N-} {80x87 support is off}
{ENDIF}
{E-} {80x87 emulation is off}
{R-} {Range checking is off}

program METL;
uses crt, dos;
const
    max_dimension      = 11;
    power_of_max_dimension = 2048;
    BackUpFrequency : word = 500;
    FileChangeFrequency = 1000000;
type
    one_dimension      = array[0..power_of_max_dimension-1] of byte;
    dimension_string = string[max_dimension+1];
    input_array        = array[0..power_of_max_dimension-1] of dimension_string;
var
    dimension           : byte;
    OriginalDim         : byte;
    zin                : byte;
    n                  : byte;
    number_width       : byte;
    control_value      : byte;
    N_Of_Zeros         : byte;
    Output             : byte;
    MaxNeurons         : byte;
    neuron            : byte;
    MinNeuron          : byte;
    NeuronLimit        : byte;
    errcode            : integer;
    WTCOUNT            : word;
    one_counter        : word;
    zero_counter       : word;
    MaxNrn             : longint;
    MaxFuncNo          : longint;
    SolutionOK         : longint;
    SolutionNone       : longint;
    FuncNo             : longint;
    StartFuncNo        : longint;
    EndFuncNo          : longint;
    error              : Boolean;
    Overflow           : Boolean;
    LinearSeparable    : Boolean;
    NewTry             : Boolean;
    FirstOptimize      : Boolean;
    NeuronOpti         : Boolean;
    Algorithm          : char;
    MinAlgo            : char;
    OriginalAlgo       : char;
    question           : char;
    MinRealize         : char;
    auto              : char;
    FastAnalyze        : char;
    BackupProcess      : char;
    TmpChar            : char;
    NeuronFileName     : Text;
    ResultFileName     : Text;
    inputs             : input_array;
    dimenstr           : string[2];
    Number             : string[3];
    desired            : one_dimension;
    NeuronWeight       : array[0..2*max_dimension] of integer;
    weight            : array[1..2*max_dimension,0..max_dimension] of longint;
    SITV              : array[0..2,0..1,1..power_of_max_dimension] of word;
    MaxNrnDim         : array[0..max_dimension] of longint;

function ipower(base, pow : byte) : longint;
```



```

begin
    ipower := trunc( exp( pow * ln(base) ) );
end;

function sgn(inp : integer) : byte;
begin
    if inp >= 0 then
        sgn := 1
    else
        sgn := 0;
    end;
end;

function bit(number:word;ix:byte):byte;
var
    i      : byte;
    code : integer;
begin
    val(inputs[number,Originaldim-ix+1],i,code);
    bit:=i;
end;

procedure ask_yes_no(var answer : char);
begin
    repeat
        answer := upcase(readkey);
    until (answer = 'N') or (answer = 'Y');
    if answer = 'Y' then
        writeln('Yes')
    else
        writeln('No');
    end;
end;

procedure ToFile(process:char);
var
    i      : byte;
    j      : word;
    Present : PathStr;
begin
    case process of
        'I': begin
            i:=0;
            repeat
                str(i,Number);
                if i<10 then
                    insert('0',Number,1)
                else
                    if i<100 then
                        insert('0',Number,1);
                    Present:=fsearch('FAR-'+Number+'.txt','.');
                    inc(i);
                until (Present='') or (i=0);
                Assign(ResultFileName,'FAR-'+Number+'.txt');
                Rewrite(ResultFileName);
                Close(ResultFileName);
                Assign(NeuronFileName,'Nrn-'+Number+'.txt');
                Present:=fsearch('Nrn-'+Number+'.txt','.');
                if Present='' then
                    begin
                        ReWrite(NeuronFileName);
                        writeln(NeuronFileName,['Synonyms']);
                        writeln(NeuronFileName);
                        writeln(NeuronFileName,'<FuncNo> = <Neuron> of <NeuronLimit>');
                        writeln(NeuronFileName,'<FuncNo> has Wrong Trial Process !! ');
                        writeln(NeuronFileName,'<FuncNo> exceeds the Neuron Limit !!!! ');
                        writeln(NeuronFileName,'<FuncNo> has No Neural NetWork !! It must have
                            maximum <NeuronLimit> neurons..');
                        writeln(NeuronFileName);
                        writeln(NeuronFileName,['Results']);
                        writeln(NeuronFileName);
                        Close(NeuronFileName);
                    end;
                end;
            until (Present='') or (i=0);
        'W': begin
            if Auto = 'Y' then
                begin
                    Rewrite(ResultFileName);
                    writeln(ResultFileName,['Auto Analyze Result']);
                    writeln(ResultFileName);
                    writeln(ResultFileName,'Dimension = ',dimension:10);
                    writeln(ResultFileName,'Heuristic Analyze = ',MinRealize:10);
                    write(ResultFileName,'Algorithm Used = ');
                    case Algorithm of
                        'J' : writeln(ResultFileName,'Major Start':10);
                        'N' : writeln(ResultFileName,'Minor Start':10);
                        'E' : writeln(ResultFileName,'Eliminate':10);
                        'A' : writeln(ResultFileName,'Auto Select':10);
                    end;
                end;
            end;
        end;
    end;
end;

```



```

        writeln(ResultFileName);
        writeln(ResultFileName,'Start Function No      = ',StartFuncNo:10);
        writeln(ResultFileName,'End Function No      = ',EndFuncNo:10);
        writeln(ResultFileName);
        writeln(ResultFileName,'Last Function No     = ',FuncNo:10);
        writeln(ResultFileName);
        writeln(ResultFileName,'Total Functions Test = ',FuncNo-StartFuncNo+1:10);
        writeln(ResultFileName);
        writeln(ResultFileName,'Solution OK      = ',SolutionOK:10);
        writeln(ResultFileName,'Solution None   = ',SolutionNone:10);
        writeln(ResultFileName);
        writeln(ResultFileName,'Linear Separable Funcs = ',MaxNrnDim[1]:10);
        writeln(ResultFileName,'# of MaxNeuron Error = ',MaxNrn:10);
        writeln(ResultFileName,'Wrong Trial Numbers = ',WTCCount:10);
        writeln(ResultFileName);
        writeln(ResultFileName,'Max Number of Neurons = ',MaxNeurons:10);
        writeln(ResultFileName,'Max Neuron Func No = ',MaxFuncNo:10);
        writeln(ResultFileName);
        if dimension < MaxNeurons then
            for i:=1 to MaxNeurons do
                writeln(ResultFileName,i:2,' Neurons = ',MaxNrnDim[i]:10)
            else
                for i:=1 to dimension do
                    writeln(ResultFileName,i:2,' Neurons = ',MaxNrnDim[i]:10);
                Close(ResultFileName);
            end
        else
            begin
                Rewrite(ResultFileName);
                writeln(ResultFileName,['Function Analyze Result']);
                writeln(ResultFileName);
                writeln(ResultFileName,'Dimension      = ',dimension:10);
                writeln(ResultFileName,'Heuristic Analyze= ',MinRealize:10);
                write(ResultFileName,'Algorithm Used = ');
                case Algorithm of
                    'J' : writeln(ResultFileName,'Major Start':10);
                    'N' : writeln(ResultFileName,'Minor Start':10);
                    'E' : writeln(ResultFileName,'Eliminate':10);
                    'A' : begin
                        if Not FirstOptimize then
                            writeln(ResultFileName,'Major Start':10)
                        else
                            if Not NeuronOpti then
                                writeln(ResultFileName,'Minor Start':10)
                            else
                                writeln(ResultFileName,'Eliminate':10);
                            end;
                        end;
                    end;
                writeln(ResultFileName);
                write(ResultFileName,'Function Output = []');
                for i := ipower(2,dimension)-1 downto 0 do
                    write(ResultFileName,desired[i]);
                    writeln(ResultFileName,',');
                    writeln(ResultFileName);
                    writeln(ResultFileName,'Linear Separable = ',LinearSeparable:10);
                    writeln(ResultFileName);
                    writeln(ResultFileName,'Number of Neurons= ',neuron:10);
                    writeln(ResultFileName);
                    if Not error then
                        begin
                            writeln(ResultFileName);
                            for j := 1 to neuron do
                                begin
                                    for i := 0 to dimension do
                                        writeln(ResultFileName,'Weight['',j:2,',',',i:2,']= ',weight[j,i]:10);
                                        writeln(ResultFileName);
                                    end;
                                for j:=0 to neuron do
                                    writeln(ResultFileName,'NeuronWeight['',j:2,']= ',NeuronWeight[j]:10);
                                end
                            end
                        else
                            begin
                                writeln(ResultFileName);
                                writeln(ResultFileName,'This function can not be expressed');
                                writeln(ResultFileName,'Please Control Your Source Code !!!!!');
                                end;
                            Close(ResultFileName);
                        end;
                    end;
                end;
            end;
        end;
    end;

procedure Initialization;
var
    i : word;
begin
    BackupProcess := 'N';

```

```

SolutionOK      := 0;
SolutionNone    := 0;
MaxNeurons     := 0;
MaxNrn         := 0;
NewTry         := False;
EndFuncNo      := 2147483647;
for i:=1 to dimension do
    MaxNrnDim[i] := 0;
write('Please enter the dimension of the inputs ( 0 < x <= ',max_dimension:2,' ) : ');
readln(dimenstr);
val(dimenstr,dimension,errcode);
str(dimension,dimenstr);
number_width := length(dimenstr);
if (dimension>max_dimension) or (dimension<0) then
    dimension:=max_dimension;
gotoxy(60,1);
writeln(dimension:number_width);
if dimension = 0 then
    begin
        clrscr;
        writeln('Thanks for using this program. ');
        halt;
    end
else
    OriginalDim := dimension;
    writeln;
    write('Do you want to activate the fast analyzer ? ');
    ask_yes_no(FastAnalyze);
    if FastAnalyze='N' then
        begin
            writeln;
            write('Do you want to activate the automatic input generator ? ');
            ask_yes_no(auto);
        end
    else
        auto:='Y';
        writeln;
        write('Please select the algorithm (maJor - miNor - Eliminate - Auto) ? ');
        repeat
            Algorithm := Ucase(Readkey);
            if Algorithm = Chr(13) then
                Algorithm := 'A';
        until (Algorithm = 'J') or (Algorithm = 'N') or (Algorithm = 'E') or (Algorithm = 'A');
        case Algorithm of
            'J' : writeln('Major Start');
            'N' : writeln('Minor Start');
            'E' : writeln('Eliminate');
            'A' : writeln('Auto Select');
        end;
        writeln;
        if auto='Y' then
            begin
                write('Please enter the starting function number : ');
                readln(StartFuncNo);
                if (dimension<5) and (StartFuncNo > ipower(2,ipower(2,dimension))) then
                    StartFuncNo:= StartFuncNo and (ipower(2,ipower(2,dimension))-1);
                if StartFuncNo = -2 then
                    if dimension < 5 then
                        StartFuncNo:=ipower(2,ipower(2,dimension)-1)
                    else
                        StartFuncNo:=ipower(2,30)-1+ipower(2,30)
                    end
                else
                    if StartFuncNo<0 then
                        StartFuncNo:=0;
                end;
                writeln;
                write('Please enter the ending function number : ');
                readln(EndFuncNo);
                if (dimension < 5) and (EndFuncNo > ipower(2,ipower(2,dimension))) then
                    EndFuncNo:= EndFuncNo and (ipower(2,ipower(2,dimension))-1);
                if EndFuncNo=-1 then
                    if dimension < 5 then
                        EndFuncNo:=ipower(2,ipower(2,dimension))-1
                    else
                        EndFuncNo:=ipower(2,30)-1+ipower(2,30)
                    end
                else
                    if EndfuncNo=-2 then
                        if dimension < 6 then
                            EndFuncNo:=ipower(2,ipower(2,dimension)-2)-1+ipower(2,ipower(2,dimension)-2)
                        else
                            EndFuncNo:=ipower(2,30)-1+ipower(2,30)
                        end
                    else
                        if (EndFuncNo<StartFuncNo) and (StartFuncNo <> ipower(2,30)-1+ipower(2,30)) then
                            EndFuncNo := StartFuncNo;
                end;
                writeln;
                if EndFuncNo - StartFuncNo > FileChangeFrequency then
                    begin
                        write('Do you want to activate the backup process ? ');

```

```

                ask_yes_no(BackupProcess);
                writeln;
            end
        else
            BackupProcess := 'N';
        end;
        write('Do you want a Heuristic analyze ? ');
        ask_yes_no(MinRealize);
        ToFile('I');
    end;

procedure find_output;
var
    i          : byte;
    j          : word;
    k          : word;
    history    : word;
    ksi        : integer;
    OutputKsi  : integer;
begin
    history := ipower(2,dimension)-1;
    error := false;
    for j := 0 to history do
        begin
            OutputKsi := NeuronWeight[0];
            for k := 1 to neuron do
                begin
                    ksi := weight[k,0];
                    for i := 1 to dimension do
                        ksi := ksi + bit(j,i)*weight[k,i];
                    OutputKsi := OutputKsi + sgn(ksi)*NeuronWeight[k];
                end;
                if (desired[j] = 0) or (desired[j] = 1) then
                    error := error or (sgn(OutputKsi) <> desired[j]);
                end;
            end;
        end;
    end;

function binary(number1:longint; dim:byte):string;
var
    TmpString : string;
    counter   : byte;
    holder    : string;
    negative   : boolean;
begin
    holder:='';
    negative := number1 < 0;
    if number1 < 0 then
        number1 := number1 and (not (1 shl 31));
    if dim = 32 then
        counter := dim - 1
    else
        counter := dim;
    while counter > 0 do
        begin
            str(number1 mod 2,TmpString);
            insert(TmpString,holder,1);
            number1 := number1 div 2;
            dec(counter);
        end;
    if dim = 32 then
        if negative then
            insert('1',holder,1)
        else
            insert('0',holder,1);
        binary:=holder;
    end;

function HD(a,b:word):byte;
var
    Temp      : word;
    NumberOfOnes : byte;
    i         : byte;
    TmpStr    : dimension_string;
begin
    NumberOfOnes := 0;
    Temp := a or b;
    TmpStr := Binary(Temp,OriginalDim);
    For i := 1 to length(TmpStr) do
        if copy(TmpStr,i,1)='1' then
            inc(NumberOfOnes);
    Temp := a and b;
    TmpStr := Binary(Temp,OriginalDim);
    For i := 1 to length(TmpStr) do
        if copy(TmpStr,i,1)='1' then
            dec(NumberOfOnes);
    HD := NumberOfOnes;

```

```

end;

Function Nearest(Base,bit:word):word;
begin
    Nearest := Base xor ipower(2,bit-1);
end;

procedure TrainGeometrically;
var
    c                : array[0..max_dimension] of longint;
    Element          : array[0..2] of word;
    NeuronMatrix     : array[0..1] of word;
    SmallVertex      : array[1..max_dimension] of byte;
    Dim              : array[1..max_dimension] of byte;
    OriginalMask      : array[1..max_dimension] of word;
    ModifiedMask     : array[1..max_dimension] of word;
    MaskMask         : array[1..max_dimension] of word;
    NewDesired       : one_dimension;
    core             : word;
    i                : word;
    j                : word;
    k                : word;
    l                : word;
    m                : word;
    n                : word;
    klimit           : word;
    tminNo           : word;
    fmaxNo           : word;
    Minl             : word;
    Mink             : word;
    Minj             : word;
    Maxk             : word;
    ZeroK            : word;
    ZeroHD           : word;
    NeuronOptiLimit  : word;
    NeuronDimLimit   : word;
    NeuronVertexLimit : word;
    Dif              : word;
    Tempo            : longint;
    Max              : longint;
    Min              : longint;
    tmin             : longint;
    fmax             : longint;
    hold             : longint;
    MaxNet           : longint;
    Tmp2             : longint;
    Center           : longint;
    Finish           : Boolean;
    Expand           : Boolean;
    ClassControl     : Boolean;
    WrongTrial       : Boolean;
    HDControl        : Boolean;
    TopStart         : Boolean;
    FastFind         : Boolean;
    NewCore          : Boolean;
    Equal            : Boolean;
    Small            : Boolean;
    CanBeOptimized   : Boolean;
    Reduce           : Boolean;
    Reduced          : Boolean;
    WeightOpti       : Boolean;
    SequentialSearch : Boolean;
    MaxHD            : Byte;
    MinHD            : Byte;
    Tmp              : Byte;
    Vertex           : Byte;
    MinkVertexType   : Byte;
    VertexType       : Byte;
    VertexStart      : Byte;
    VerySmallVertex  : Byte;
    OriginalVertexStart : Byte;
    TempA            : Byte;
    MaskBit          : Byte;
    ReduceProcess    : Byte;
    RP               : Byte;
begin
    Element[0]       := zero_counter;
    Element[1]       := one_counter;
    Element[2]       := 0;
    TmpChar          := ' ';
    MaxHD            := 1;
    NewCore          := False;
    NeuronOpti       := False;
    FirstOptimize    := False;
    WeightOpti       := False;
    SequentialSearch := False;
    Dif              := 65535;

```

[illegible]

```

SITV[1,1,m]:=SITV[1,1,m] or (1 shl tmp);
end
else
begin
tmp := Dim[ReduceProcess]-j-1;
SITV[1,1,m]:=SITV[1,1,m] and (not (1 shl tmp));
end;
end;
if TempA = 1 then
begin
tmp := Dim[ReduceProcess]-k;
SITV[1,1,m]:=SITV[1,1,m] or (1 shl tmp);
end
else
begin
tmp := Dim[ReduceProcess]-k;
SITV[1,1,m]:=SITV[1,1,m] and (not (1 shl tmp));
end;
end;
tmp := Dim[ReduceProcess]-k;
ModifiedMask[ReduceProcess]:=ModifiedMask[ReduceProcess] and (not (1 shl tmp));
tmp := Dim[ReduceProcess]-i;
ModifiedMask[ReduceProcess]:=ModifiedMask[ReduceProcess] or (1 shl tmp);
inc(k);
inc(i);
end
else
begin
inc(k);
inc(i);
end;
end
end
else
inc(i);
until i > Dim[ReduceProcess];
if HD(0,OriginalMask[ReduceProcess]) > 0 then
dimension := HD(0,OriginalMask[ReduceProcess]);
MaskMask[ReduceProcess] := 0;
for i:=1 to Dim[ReduceProcess]-dimension do
begin
if bit(SITV[VerySmallVertex,1,1],Dim[ReduceProcess]-i+1) = 1 then
begin
tmp := Dim[ReduceProcess]-i;
MaskMask[ReduceProcess] := MaskMask[ReduceProcess] or (1 shl tmp);
end
else
begin
tmp := Dim[ReduceProcess]-i;
MaskMask[ReduceProcess] := MaskMask[ReduceProcess] and (not (1 shl tmp));
end;
end;
for j := 1 to Element[VerySmallVertex] do
for k:=1 to Dim[ReduceProcess]-dimension do
begin
tmp := Dim[ReduceProcess]-k;
SITV[VerySmallVertex,1,j]:=SITV[VerySmallVertex,1,j] and (not(1 shl tmp));
end;
j:=1;
repeat
Reduce := True;
for i:=1 to Dim[ReduceProcess]-dimension do
Reduce:=Reduce and (bit(SITV[1-VerySmallVertex,1,j],Dim[ReduceProcess]-i+1)
=bit(MaskMask[ReduceProcess],Dim[ReduceProcess]-i+1));
if Reduce then
begin
for k:=1 to Dim[ReduceProcess]-dimension do
begin
tmp := Dim[ReduceProcess]-k;
SITV[1-VerySmallVertex,1,j]:=SITV[1-VerySmallVertex,1,j] and (not(1 shl tmp));
Reduced := True;
end;
inc(j);
end
else
begin
SITV[1-VerySmallVertex,1,j] := SITV[1-VerySmallVertex,1,Element[1-VerySmallVertex]];
dec(Element[1-VerySmallVertex]);
Reduced := True;
end;
until j > Element[1-VerySmallVertex];
end;
until Dim[ReduceProcess] = HD(0,OriginalMask[ReduceProcess]);
for i:=0 to 1 do
for j := 1 to Element[i] do
NewDesired[SITV[i,1,j]] := i;
NeuronLimit := NeuronDimLimit;
if NeuronVertexLimit < NeuronLimit then

```

```

NeuronLimit := NeuronVertexLimit;
NeuronOptiLimit := 0;
for i:=1 to Element[VerySmallVertex]-1 do
begin
  if SITV[VerySmallVertex,0,i] = 0 then
  begin
    inc(NeuronOptiLimit);
    for j:=i+1 to Element[VerySmallVertex] do
    begin
      if HD(SITV[VerySmallVertex,1,i],SITV[VerySmallVertex,1,j]) = 1 then
      begin
        inc(SITV[VerySmallVertex,0,i]);
        inc(SITV[VerySmallVertex,0,j]);
      end;
    end;
  end;
end;
if SITV[VerySmallVertex,0,Element[VerySmallVertex]] = 0 then
inc(NeuronOptiLimit);
for i:= 1 to Element[VerySmallVertex] do
  SITV[VerySmallVertex,0,i] := 0;
CanBeOptimized := True;
if (Algorithm = 'A') or (Algorithm = 'E') then
if (NeuronOptiLimit <= NeuronLimit) then
begin
  NeuronLimit := NeuronOptiLimit;
  CanBeOptimized := True;
end;
ZeroHD := 0;
VertexType := 0;
Maxk := 0;
Minj := 1;
Mink := Element[Vertex];
for l := 0 to 1 do
begin
  Vertex := l*VerySmallVertex + (1-l)*(1-VerySmallVertex);
  ZeroK := 0;
  if Element[Vertex] > 0 then
  begin
    SITV[Vertex,0,Element[Vertex]] := 0;
    for i:=1 to Element[Vertex]-1 do
    begin
      SITV[Vertex,0,i] := 0;
      for j:=i+1 to Element[Vertex] do
      begin
        gotoxy(OriginalDim+8,20);
        write('i = ',i:5,'      j = ',j:5);
        if keypressed then
        begin
          TmpChar := readkey;
          if TmpChar = '5' then
            halt;
        end;
        if HD(SITV[Vertex,1,i],SITV[Vertex,1,j]) = 1 then
        begin
          inc(SITV[Vertex,0,i]);
          inc(SITV[Vertex,0,j]);
        end;
      end;
    end;
    if (SITV[Vertex,0,i] > Maxk) and ((Algorithm = 'A') or ((Algorithm='J')
and (Vertex<>VerySmallVertex)) or ((Algorithm='N') and (Vertex=VerySmallVertex))) then
    begin
      Maxk := SITV[Vertex,0,i];
      VertexType := Vertex;
      Minj := i;
    end;
    if (SITV[Vertex,0,i] < Mink) and (SITV[Vertex,0,i]>0) then
    begin
      Mink := SITV[Vertex,0,i];
      MinkVertexType := Vertex;
    end
  else
    if SITV[Vertex,0,i]=0 then
      inc(ZeroK);
    end;
  if (SITV[Vertex,0,Element[Vertex]] > Maxk) and ((Algorithm = 'A') or ((Algorithm='J')
and (Vertex<>VerySmallVertex)) or ((Algorithm='N') and (Vertex=VerySmallVertex))) then
  begin
    Maxk := SITV[Vertex,0,Element[Vertex]];
    VertexType := Vertex;
    Minj := Element[Vertex];
  end;
  if (SITV[Vertex,0,Element[Vertex]] < Mink) and (SITV[Vertex,0,Element[Vertex]]>0) then
  begin
    Mink := SITV[Vertex,0,Element[Vertex]];
    MinkVertexType := Vertex;
  end
end

```

```

else
    if SITV[Vertex,0,Element[Vertex]]=0 then
        inc(ZeroK);
    end;
    if ZeroK > ZeroHD then
        ZeroHD := ZeroK;
    end;
if Algorithm = 'A' then
    begin
        VertexStart := VertexType;
        FirstOptimize := False;
    end;
Vertex := VertexStart;
Minl := Minj;
if (Algorithm = 'E') or (Element[0] = Element[1]) then
    begin
        Vertex := VerySmallVertex;
        VertexStart := VerySmallVertex;
        if NeuronLimit = NeuronOptiLimit then
            core := SITV[Vertex,1,1]
        else
            if (Mink = 1) then
                begin
                    Minj := Minl;
                    Maxk := Element[Vertex];
                    core := SITV[Vertex,1,1];
                    for i := 1 to Element[Vertex] do
                        begin
                            if SITV[Vertex,0,i] = 1 then
                                begin
                                    if i=1 then
                                        j := 2
                                    else
                                        j := 1;
                                    FastFind := False;
                                    repeat
                                        begin
                                            if (HD(SITV[Vertex,1,i],SITV[Vertex,1,j])=1) then
                                                begin
                                                    FastFind := True;
                                                    if (SITV[Vertex,0,j] < Maxk) then
                                                        begin
                                                            Minj := j;
                                                            Maxk := SITV[Vertex,0,j];
                                                        end
                                                    else
                                                        if (SITV[Vertex,0,j] = Maxk)
                                                            and ((HD(0,SITV[Vertex,1,j]) = HD(0,SITV[Vertex,1,Minj]))
                                                            and (SITV[Vertex,1,j] < SITV[Vertex,1,Minj])) or
                                                            (HD(0,SITV[Vertex,1,j]) > HD(0,SITV[Vertex,1,Minj])) then
                                                                Minj := j;
                                                        end;
                                                    if j + 1 = i then
                                                        inc(j,2)
                                                    else
                                                        inc(j);
                                                    end;
                                                until FastFind;
                                                core := SITV[Vertex,1,Minj];
                                            end;
                                        end;
                                    end;
                                end;
                            end;
                        end;
                    end;
                end;
            else
                begin
                    core := SITV[Vertex,1,Minj];
                end;
            end;
        else
            core := SITV[Vertex,1,Minj];
            CanBeOptimized := CanBeOptimized and (ZeroHD<=NeuronLimit) and ((Mink<>Element[Vertex])
            or (Element[VerySmallVertex] <= NeuronLimit));
        if vertex = 1 then
            NeuronMatrix[(neuron-1) div 16]:=1 shl 15;
        j:=1;
        repeat
            if SITV[Vertex,1,j] = core then
                begin
                    inc(Element[2]);
                    dec(Element[Vertex]);
                    SITV[2,1,Element[2]] := core;
                    SITV[2,0,Element[2]] := SITV[Vertex,0,j];
                    SITV[Vertex,1,j] := SITV[Vertex,1,Element[Vertex]+1];
                    SITV[Vertex,0,j] := SITV[Vertex,0,Element[Vertex]+1];
                end
            else
                inc(j);
            until (Element[2] = 1) or (j>Element[Vertex]);

```



```

if Element[2] = 0 then
begin
  MinHD := dimension+1;
  FastFind := False;
  k := 1;
  Mink := 1;
  repeat
    if HD(core,SITV[Vertex,1,k]) <= MinHD then
    begin
      if (HD(core,SITV[Vertex,1,k])=MinHD) then
      begin
        if ((HD(0,SITV[Vertex,1,k])=HD(0,SITV[Vertex,1,Mink]))
          and (SITV[Vertex,1,k]<SITV[Vertex,1,Mink]))
        or (HD(0,SITV[Vertex,1,k])<HD(0,SITV[Vertex,1,Mink])) then
          Mink := k;
        end
      else
      begin
        Mink := k;
        MinHD := HD(core,SITV[Vertex,1,k]);
      end;
    end;
    inc(k);
  until (k > Element[Vertex]) or FastFind;
  k := Mink;
  core:=SITV[Vertex,1,k];
  inc(Element[2]);
  dec(Element[Vertex]);
  SITV[2,1,Element[2]] := core;
  SITV[2,0,Element[2]] := SITV[Vertex,0,k];
  SITV[Vertex,1,k] := SITV[Vertex,1,Element[Vertex]+1];
  SITV[Vertex,0,k] := SITV[Vertex,0,Element[Vertex]+1];
  k:=1;
  Mink := 1;
end;
for i := 1 to dimension do
  if NewDesired(Nearest(core,i)) = Vertex then
  begin
    inc(Element[2]);
    dec(Element[Vertex]);
    SITV[2,1,Element[2]] := Nearest(core,i);
    j := 1;
    repeat
      if SITV[Vertex,1,j] = SITV[2,1,Element[2]] then
      begin
        SITV[Vertex,1,j] := SITV[Vertex,1,Element[Vertex]+1];
        SITV[2,0,Element[2]] := SITV[Vertex,0,j];
        SITV[Vertex,0,j] := SITV[Vertex,0,Element[Vertex]+1];
        break;
      end;
      inc(j);
    until (j > Element[Vertex]+1);
    if bit(core,i) = Vertex then
      weight[neuron,i] := 2
    else
      weight[neuron,i] := -2;
    end
  else
    if bit(core,i) = Vertex then
      weight[neuron,i] := 4
    else
      weight[neuron,i] := -4;
  end;
weight[neuron,0] := 3*(2*Vertex-1);
for i := 1 to dimension do
  weight[neuron,0] := weight[neuron,0] - weight[neuron,i]*bit(core,i);
if Element[1-Vertex] = 0 then
begin
  NeuronLimit := 1;
  if Vertex=0 then
    weight[1,0] := -1
  else
    weight[1,0] := 1;
  for i:= 1 to dimension do
    weight[1,i] := 0;
  Finish := True;
end
else
  Finish := False;
WrongTrial := False;
repeat
  k := 1;
  kLimit := 0;
  Expand := False;
  TopStart := True;
  while (Element[Vertex] <> 0) and (k <= Element[Vertex]) and (Not Finish) do
  begin
    if keypressed then

```

```

begin
  TmpChar := readkey;
  case TmpChar of
    '0' : Question := 'N';
    '1' : Finish := True;
  end;
end;
gotoxy(OriginalDim+8,7);
write('k      = ',k:5);
gotoxy(OriginalDim+8,8);
write('SITV   = ',Element[2]:5);
gotoxy(OriginalDim+8,10);
write('Neuron = ',Neuron:5);
gotoxy(OriginalDim+8,14);
write('WrongT = ',WTCOUNT:5);
if Element[2] = 0 then
  begin
    WrongTrial := False;
    for i:=1 to Element[Vertex] do
      SITV[Vertex,0,i] := 0;
    Mink := Element[Vertex];
    Maxk := 0;
    for i:=1 to Element[Vertex]-1 do
      begin
        for j:=i+1 to Element[Vertex] do
          begin
            gotoxy(OriginalDim+8,20);
            write('i = ',i:5,'      j = ',j:5);
            if keypressed then
              begin
                TmpChar := readkey;
                if TmpChar = '5' then
                  halt;
                end;
            if HD(SITV[Vertex,1,i],SITV[Vertex,1,j]) = 1 then
              begin
                inc(SITV[Vertex,0,i]);
                inc(SITV[Vertex,0,j]);
              end;
            end;
            if (SITV[Vertex,0,i] > Maxk) and ((Algorithm = 'A') or ((Algorithm='J')
            and (Vertex<>VerySmallVertex)) or ((Algorithm='N')
            and (Vertex=VerySmallVertex))) then
              begin
                Maxk := SITV[Vertex,0,i];
                VertexType := Vertex;
                if ((SITV[Vertex,1,i] < SITV[Vertex,1,Minj])
                and (SITV[Vertex,0,i]=Maxk))
                or (SITV[Vertex,0,Element[Vertex]]>Maxk) then
                  Minj := i;
                end;
                if (SITV[Vertex,0,i] < Mink) and (SITV[Vertex,0,i]>0) then
                  Mink := SITV[Vertex,0,i];
                end;
                if (SITV[Vertex,0,Element[Vertex]] >= Maxk) and ((Algorithm = 'A') or ((Algorithm='J')
                and (Vertex<>VerySmallVertex)) or ((Algorithm='N') and (Vertex=VerySmallVertex))) then
                  begin
                    Maxk := SITV[Vertex,0,Element[Vertex]];
                    VertexType := Vertex;
                    if ((SITV[Vertex,1,Element[Vertex]] < SITV[Vertex,1,Minj])
                    and (SITV[Vertex,0,Element[Vertex]]=Maxk))
                    or (SITV[Vertex,0,Element[Vertex]]>Maxk) then
                      Minj := Element[Vertex];
                    end;
                    if (SITV[Vertex,0,Element[Vertex]] < Mink) and (SITV[Vertex,0,Element[Vertex]]>0) then
                      Mink := SITV[Vertex,0,Element[Vertex]];
                    Minl := Minj;
                    if (Algorithm='E') or NeuronOpti then
                      begin
                        if NeuronLimit = NeuronOptiLimit then
                          core := SITV[Vertex,1,1]
                        else
                          if (Mink = 1) then
                            begin
                              Minj := Minl;
                              Maxk := Element[Vertex];
                              core := SITV[Vertex,1,1];
                              for i := 1 to Element[Vertex] do
                                begin
                                  if SITV[Vertex,0,i] = 1 then
                                    begin
                                      if i=1 then
                                        j := 2
                                      else
                                        j := 1;
                                      FastFind := False;
                                      repeat

```

```

begin
if (HD(SITV[Vertex,1,1],SITV[Vertex,1,j])=1) then
begin
FastFind := True;
if (SITV[Vertex,0,j] < Maxk) then
begin
Minj := j;
Maxk := SITV[Vertex,0,j];
end
else
if (SITV[Vertex,0,j] = Maxk) and
((HD(0,SITV[Vertex,1,j]) =
HD(0,SITV[Vertex,1,Minj]))
and (SITV[Vertex,1,j] <
SITV[Vertex,1,Minj])) or
(HD(0,SITV[Vertex,1,j]) >
HD(0,SITV[Vertex,1,Minj])) then
Minj := j;
end;
if j+1=1 then
inc(j,2)
else
inc(j);
end;
until FastFind;
core := SITV[Vertex,1,Minj];
end;
end;
else
begin
core := SITV[Vertex,1,Minj];
end;
end;
else
core := SITV[Vertex,1,Minj];
j:=1;
repeat
if SITV[Vertex,1,j] = core then
begin
inc(Element[2]);
dec(Element[Vertex]);
SITV[2,1,Element[2]] := core;
SITV[2,0,Element[2]] := SITV[Vertex,0,j];
SITV[Vertex,1,j] := SITV[Vertex,1,Element[Vertex]+1];
SITV[Vertex,0,j] := SITV[Vertex,0,Element[Vertex]+1];
end
else
inc(j);
until (Element[2] = 1) or (j>Element[Vertex]);
if Element[2] = 0. then
begin
MinHD := dimension+1;
FastFind := False;
j := 1;
Mink := 1;
repeat
if HD(core,SITV[Vertex,1,j]) <= MinHD then
begin
if HD(core,SITV[Vertex,1,j]) = MinHD then
begin
if ((HD(0,SITV[Vertex,1,j])=
HD(0,SITV[Vertex,1,Mink])) and
(SITV[Vertex,1,j] < SITV[Vertex,1,Mink]))
or (HD(0,SITV[Vertex,1,j]) >
HD(0,SITV[Vertex,1,Mink])) then
Mink := j;
end
else
begin
MinHD := HD(core,SITV[Vertex,1,j]);
Mink := j;
end;
end;
inc(j);
until (j > Element[Vertex]) or FastFind;
j := Mink;
core:=SITV[Vertex,1,j];
inc(Element[2]);
dec(Element[Vertex]);
SITV[2,1,Element[2]] := core;
SITV[2,0,Element[2]] := SITV[Vertex,0,j];
SITV[Vertex,1,j] := SITV[Vertex,1,Element[Vertex]+1];
SITV[Vertex,0,j] := SITV[Vertex,0,Element[Vertex]+1];
j:=1;
Mink := 1;
end;
end;

```

```

for i := 1 to dimension do
  if NewDesired[Nearest(core,i)] = Vertex then
    begin
      j := 1;
      repeat
        if SITV[Vertex,1,j] = Nearest(core,i) then
          begin
            inc(Element[2]);
            SITV[2,1,Element[2]] := Nearest(core,i);
            SITV[2,0,Element[2]] := SITV[Vertex,0,j];
            dec(Element[Vertex]);
            SITV[Vertex,1,j] := SITV[Vertex,1,Element[Vertex]+1];
            SITV[Vertex,0,j] := SITV[Vertex,0,Element[Vertex]+1];
            break;
          end;
          inc(j);
        until (j > Element[Vertex]+1);
        if bit(core,i) = Vertex then
          weight[neuron,i] := 2
        else
          weight[neuron,i] := -2;
        end
      end
    else
      if bit(core,i) = Vertex then
        weight[neuron,i] := 4
      else
        weight[neuron,i] := -4;
        weight[neuron,0] := 3*(2*Vertex-1);
        for i := 1 to dimension do
          weight[neuron,0] := weight[neuron,0] - weight[neuron,i]*bit(core,i);
        end;
        c[0] := Element[2];
        if Element[Vertex] <> 0 then
          begin
            inc(Element[2]);
            dec(Element[Vertex]);
            c[0] := Element[2];
            center := core;
            if not SequentialSearch then
              begin
                MinHD := dimension+1;
                FastFind := False;
                repeat
                  if (HD(center,SITV[Vertex,1,k]) <= MaxHD) then
                    begin
                      Mink := k;
                      FastFind := True;
                      MinHD := HD(center,SITV[Vertex,1,k]);
                      MaxHD := MinHD;
                    end
                  end
                until (k > Element[Vertex]+1) and (Not FastFind) and (Not TopStart) and HDControl then
                  begin
                    k:=1;
                    klimit := 0;
                    TopStart := True;
                    HDControl := False;
                    inc(MaxHD);
                  end;
                until (k > Element[Vertex]+1) or FastFind;
                if Not FastFind then
                  begin
                    MaxHD := MinHD;
                    klimit := 0;
                  end;
                k:=Mink;
              end;
            SITV[2,1,Element[2]] := SITV[Vertex,1,k];
            SITV[2,0,Element[2]] := SITV[Vertex,0,k];
            SITV[Vertex,1,k] := SITV[Vertex,1,Element[Vertex]+1];
            SITV[Vertex,0,k] := SITV[Vertex,0,Element[Vertex]+1];
          end;
        end;
        for i := 1 to dimension do
          begin
            c[i] := 0;
            for j := 1 to Element[2] do
              inc(c[i],bit(SITV[2,1,j],i));
            end;
            c[i] := 2*(2*c[i] - c[0]);
          end;
        end;
      end;
    end;
  end;
end;

```

```

end;
Equal := False;
Small := False;
repeat
  tmin := 0;
  for i := 1 to dimension do
    tmin := tmin + c[i]*bit(SITV[2,1,Element[2]],i);
  tminno := SITV[2,1,Element[2]];
  for i := 1 to Element[2]-1 do
    begin
      hold := 0;
      for j := 1 to dimension do
        hold := hold + c[j]*bit(SITV[2,1,i],j);
      if hold < tmin then
        begin
          tmin := hold;
          tminno := SITV[2,1,i];
        end
      end
    end
  if (hold = tmin) and (HD(0,SITV[2,1,i]) < HD(0,tminno)) then
    tminno := SITV[2,1,i];
  end;
  fmax := 0;
  if Element[Vertex] <> 0 Then
    begin
      for i := 1 to dimension do
        fmax := fmax + c[i]*bit(SITV[Vertex,1,Element[Vertex]],i);
        fmaxNo := SITV[Vertex,1,Element[Vertex]];
      end
    end
  else
    begin
      for i := 1 to dimension do
        fmax := fmax + c[i]*bit(SITV[1-Vertex,1,Element[1-Vertex]],i);
        fmaxNo := SITV[1-Vertex,1,Element[1-Vertex]];
      end;
    end;
  for i := 1 to Element[Vertex] do
    begin
      hold := 0;
      for j := 1 to dimension do
        hold := hold + c[j]*bit(SITV[Vertex,1,i],j);
      if hold > fmax then
        begin
          fmax := hold;
          fmaxNo := SITV[Vertex,1,i];
        end
      end
    end
  if (hold = fmax) and (HD(0,SITV[Vertex,1,i]) < HD(0,fmaxNo)) then
    fmaxno := SITV[Vertex,1,i];
  end;
  for i := 1 to Element[1-Vertex] do
    begin
      hold := 0;
      for j := 1 to dimension do
        hold := hold + c[j]*bit(SITV[1-Vertex,1,i],j);
      if hold > fmax then
        begin
          fmax := hold;
          fmaxNo := SITV[1-Vertex,1,i];
        end
      end
    end
  if (hold = fmax) and (HD(0,SITV[1-Vertex,1,i]) < HD(0,fmaxNo)) then
    fmaxno := SITV[1-Vertex,1,i];
  end;
  ClassControl := True;
  if WeightOpti and ((tmin=fmax)) or ((tmin<fmax)
    and (fmax-tmin<Dif)) then
    begin
      ClassControl := False;
      Equal := Equal or (tmin = fmax);
      Small := Small or (tmin < fmax);
      if Small then
        Dif := fmax - tmin;
        for i:=1 to dimension do
          if (bit(fmaxNo,i)=1) then
            dec(c[i],2);
          for i := 1 to dimension do
            if bit(tminNo,i)=1 then
              inc(c[i],2);
            end;
          end;
        until ClassControl;
        Dif := 65535;
      if (tmin > fmax) or (Element[1-Vertex]=0) then
        begin
          for i:=1 to dimension do
            weight[neuron,i] := (2*Vertex-1)*c[i];
          if Element[1-Vertex] <> 0 then
            weight[neuron,0] := (1-2*Vertex)*((tmin + fmax) div 2)

```

```

else
    weight[neuron,0] := Vertex;
Expand := True;
HDControl := True;
WrongTrial := False;
weightOpti := False;
if k = klimit then
    klimit := 0
else
    if klimit <> 0 then
        k := klimit;
if (k > Element[Vertex]) and (Not TopStart) then
    begin
        k:=1;
        Expand := False;
        TopStart := True;
        klimit := 0;
        HDControl := False;
    end
else
    if TopStart then
        begin
            MaxHD := MinHD;
            TopStart := False;
        end;
end
else
    begin
        dec(Element[2]);
        inc(Element[Vertex]);
        SITV[Vertex,1,Element[Vertex]] := SITV[Vertex,1,k];
        SITV[Vertex,0,Element[Vertex]] := SITV[Vertex,0,k];
        SITV[Vertex,1,k] := SITV[2,1,Element[2]+1];
        SITV[Vertex,0,k] := SITV[2,0,Element[2]+1];
        if klimit = 0 then
            klimit := k;
if (MinHD > MaxHD) and (Not HDControl) and (Not TopStart) then
    k := Element[Vertex]+1
else
    inc(k);
if TopStart then
    begin
        MaxHD := MinHD;
        TopStart := False;
    end;
end;
if (Expand or (Not WeightOpti) or (Not SequentialSearch)) and (k > Element[Vertex]) then
    begin
        k := 1;
        klimit := 0;
        Expand := False;
        HDControl := False;
        TopStart := True;
        if Not WeightOpti then
            WeightOpti := (Not Expand)
        else
            begin
                SequentialSearch := True;
                weightOpti := False;
            end;
    end;
end;
if (Element[Vertex] = 0) or (Finish) Then
    Finish := True
else
    if WrongTrial and (((Algorithm='A') and FirstOptimize and
    Not CanBeOptimized) or (Algorithm<>'A')) then
        begin
            inc(WTCount);
            gotoxy(OriginalDim+8,14);
            write('WrongT = ',WTCount:5);
            Append(NeuronFileName);
            writeln(NeuronFileName,FuncNo,' has Wrong Trial Process !! ');
            Close(NeuronFileName);
            if Not NeuronOpti then
                if (VertexStart <> Vertex) then
                    begin
                        Vertex := 1 - Vertex;
                        Element[2] := 0;
                        NewCore := True;
                        WeightOpti:= False;
                        SequentialSearch := False;
                        MaxHD := 1;
                        if vertex = 1 then
                            begin
                                tmp := 15 - ((neuron-1) mod 16);
                                NeuronMatrix[(neuron-1) div 16]:=(1 shl tmp)

```

```

or NeuronMatrix[(neuron-1) div 16];
    end
  else
    begin
      tmp := 15 - ((neuron-1) mod 16);
      NeuronMatrix[(neuron-1) div 16] := (not (1 shl tmp))
        and NeuronMatrix[(neuron-1) div 16];
    end;
  end
end
else
  begin
    repeat
      SITV[1-Vertex,1,Element[1-Vertex]+1] := SITV[2,1,Element[2]];
      SITV[1-Vertex,0,Element[1-Vertex]+1] := SITV[2,0,Element[2]];
      inc(Element[1-Vertex]);
      dec(Element[2]);
      until NewDesired[SITV[2,1,Element[2]]] = Vertex;
      Element[2] := 0;
      NewCore := True;
      WeightOpti := False;
      SequentialSearch := False;
      dec(neuron);
      MaxHD := 1;
      if vertex = 1 then
        begin
          tmp := 15 - ((neuron-1) mod 16);
          NeuronMatrix[(neuron-1) div 16] := (1 shl tmp)
            or NeuronMatrix[(neuron-1) div 16];
        end
      end
    end
    begin
      tmp := 15 - ((neuron-1) mod 16);
      NeuronMatrix[(neuron-1) div 16] := (not (1 shl tmp))
        and NeuronMatrix[(neuron-1) div 16];
    end;
  end
end
else
  begin
    Element[2] := 0;
    SequentialSearch := False;
    WeightOpti := False;
  end;
end
end
else
  begin
    if neuron = 22 then
      begin
        repeat
          TextBackGround(red);
          TextColor(White);
          clrscr;
          gotoxy(25,10);
          write('Program Limit OverFlow - Funcno : ',FuncNo);
          gotoxy(17,12);
          write('Number of Neurons Exceed 22 Which Is Not Optimal');
          for zin := 7 to 9 do
            begin
              sound(zin*450);
              delay(200);
              nosound;
              delay(10);
            end;
          until keypressed;
          halt;
        end;
      inc(neuron);
      MaxHD := 1;
      WrongTrial := True;
      WeightOpti := False;
      SequentialSearch := False;
      Finish := False;
    if ((Element[VertexStart]>1) and (Not NeuronOpti) and (Neuron<=NeuronLimit))
      or (Algorithm='J') or (Algorithm='N') then
      Vertex := 1 - Vertex
    else
      begin
        if (neuron=Neuronlimit) and (FirstOptimize )
          and (Element[VertexStart]>1) and (neuron<NeuronVertexLimit) then
          Vertex := 1-Vertex
        else
          if Not (NeuronOpti or NewCore) then
            if (Neuron>=NeuronLimit) then
              begin
                if FirstOptimize then
                  begin
                    if CanBeOptimized then
                      begin

```

```

repeat
SITV[NewDesired[SITV[2,1,Element[2]]],1,Element[
NewDesired[SITV[2,1,Element[2]]]+1]
:=SITV[2,1,Element[2]];
SITV[NewDesired[SITV[2,1,Element[2]]],0,Element[
NewDesired[SITV[2,1,Element[2]]]+1]
:=SITV[2,0,Element[2]];
inc(Element[NewDesired[SITV[2,1,Element[2]]]]);
dec(Element[2]);
until Element[2] = 0;
Neuron := 1;
VertexStart := VerySmallVertex;
Vertex := VertexStart;
NeuronOpti := True;
end
else
if Element[VertexStart] <> 1 then
Vertex := 1 - Vertex
else
begin
Element[2] := 0;
Vertex := VertexStart;
end;
end
else
begin
repeat
SITV[NewDesired[SITV[2,1,Element[2]]],1,Element[
NewDesired[SITV[2,1,Element[2]]]+1]
:=SITV[2,1,Element[2]];
SITV[NewDesired[SITV[2,1,Element[2]]],0,Element[
NewDesired[SITV[2,1,Element[2]]]+1]
:=SITV[2,0,Element[2]];
inc(Element[NewDesired[SITV[2,1,Element[2]]]]);
dec(Element[2]);
until Element[2] = 0;
Neuron := 1;
VertexStart := 1-VertexStart;
Vertex := VertexStart;
FirstOptimize := True;
end
end
else
if Element[VertexStart] <> 1 then
Vertex := 1 - Vertex
else
begin
Element[2] := 0;
Vertex := VertexStart;
end
end
else
begin
if (VertexStart = Vertex) then
begin
Element[2] := 0;
NewCore := True;
MaxHD := 1;
if vertex = 1 then
begin
tmp := 15 - ((neuron-1) mod 16);
NeuronMatrix[(neuron-1) div 16] := (1 shl tmp)
or NeuronMatrix[(neuron-1) div 16];
end
else
begin
tmp := 15 - ((neuron-1) mod 16);
NeuronMatrix[(neuron-1) div 16] := (not (1 shl tmp))
and NeuronMatrix[(neuron-1) div 16];
end;
end
end
else
begin
repeat
SITV[Vertex,1,Element[Vertex]+1] := SITV[2,1,Element[2]];
SITV[Vertex,0,Element[Vertex]+1] := SITV[2,0,Element[2]];
inc(Element[Vertex]);
dec(Element[2]);
until NewDesired[SITV[2,1,Element[2]]] = 1-Vertex;
Element[2] := 0;
NewCore := True;
dec(neuron);
MaxHD := 1;
Vertex := VertexStart;
if vertex = 1 then
begin
tmp := 15 - ((neuron-1) mod

```

16);


```

NeuronMatrix[(neuron-1) div 16] := (1 shl tmp)
or NeuronMatrix[(neuron-1) div 16];
end
else
begin
tmp := 15 - ((neuron-1) mod 16);
NeuronMatrix[(neuron-1) div 16] := (not (1 shl tmp))
and NeuronMatrix[(neuron-1) div 16];
end;
end;
end;
if vertex = 1 then
begin
tmp := 15 - ((neuron-1) mod 16);
NeuronMatrix[(neuron-1) div 16] := (1 shl tmp) or NeuronMatrix[(neuron-1) div 16];
end
else
begin
tmp := 15 - ((neuron-1) mod 16);
NeuronMatrix[(neuron-1) div 16] := (not (1 shl tmp))
and NeuronMatrix[(neuron-1) div 16];
end;
end;
if (neuron >= MinNeuron) and (Algorithm <> 'A') then
Finish := True;
until Finish;
for RP := ReduceProcess-1 downto 1 do
begin
m := Dim[RP] - HD(0, OriginalMask[RP]) + 1;
k := HD(0, OriginalMask[RP]);
if (HD(0, OriginalMask[RP]) < Dim[RP]) and (HD(0, OriginalMask[RP]) > 0) then
for j := 1 to neuron do
begin
max := weight[j, 0];
min := weight[j, 0];
for i := 1 to HD(0, OriginalMask[RP]) do
begin
if weight[j, i] >= 0 then
inc(max, weight[j, i])
else
dec(min, -weight[j, i]);
end;
i := 1;
k := m;
while i <> k do
begin
if bit(OriginalMask[RP], Dim[RP] - i + 1) = 1 then
begin
if bit(ModifiedMask[RP], Dim[RP] - k + 1) = 1 then
begin
Tempo := weight[j, Dim[RP] - k + 1];
for l := k - 1 downto i do
weight[j, Dim[RP] - l] := weight[j, Dim[RP] - l + 1];
weight[j, Dim[RP] - i + 1] := Tempo;
inc(i);
inc(k);
end
else
inc(k);
end
else
inc(i);
end;
end;
k := 1;
for l := 1 to Dim[RP] do
begin
if bit(OriginalMask[RP], Dim[RP] - l + 1) = 0 then
begin
MaskBit := bit(MaskMask[RP], Dim[RP] - k + 1);
weight[j, Dim[RP] - l + 1] := (2 * MaskBit - 1) * ((1 - SmallVertex[RP]) * min + SmallVertex[RP] * max) +
(1 - 2 * MaskBit) * (1 - 2 * SmallVertex[RP]);
weight[j, 0] := weight[j, 0] - (MaskBit) * (weight[j, Dim[RP] - l + 1]);
if Maskbit = 1 then
if weight[j, Dim[RP] - l + 1] >= 0 then
inc(max, weight[j, Dim[RP] - l + 1])
else
dec(min, -weight[j, Dim[RP] - l + 1]);
inc(k);
end;
end;
end;
end;
dimension := OriginalDim;
NeuronOpti := False;
NeuronWeight[0] := -1;
NeuronWeight[neuron] := 2;

```

```

MaxNet := 1;
for i := neuron-1 downto 1 do
begin
    tmp := 15 - ((i-1) mod 16);
    if NeuronMatrix[(i-1) div 16] and (1 shl tmp) <> 0 then
    begin
        NeuronWeight[i] := -NeuronWeight[0]+1;
        inc(MaxNet,NeuronWeight[i]);
    end
    else
    begin
        NeuronWeight[i] := MaxNet+1;
        dec(NeuronWeight[0],NeuronWeight[i]);
    end;
end;
end;

end;

procedure input_generator;
var
i      : Word;
k      : Byte;
l      : Byte;
AutoFill : Byte;
q      : Char;
ManualEdit : Char;
code   : Integer;
j      : Longint;
begin
    AutoFill := 3;
    ManualEdit := ' ';
    if auto='N' then
    begin
        writeln;
        write('Do you want to edit the outputs ? ');
        ask_yes_no(q);
        if q='Y' then
        begin
            writeln;
            write('Do you want to edit the outputs manually ? ');
            ask_yes_no(ManualEdit);
            if ManualEdit='N' then
            begin
                writeln;
                write('Please enter the number of function : ');
                readln(StartFuncNo);
                for j:= 0 to ipower(2,dimension)-1 do
                    desired[j] := 0;
                FuncNo := StartFuncNo;
                While (FuncNo <> 0) do
                begin
                    j:=Trunc( ln (FuncNo+0.5) / ln (2) );
                    desired[j] := 1;
                    FuncNo := FuncNo mod ipower(2,j);
                end;
            end;
        end;
        end;
        ClrScr;
    end
    else
        q:='N';
        one_counter := 0;
        zero_counter := 0;
        for i := 0 to ipower(2,dimension) - 1 do
        begin
            inputs[i] := binary(i,dimension);
            if (dimension < 5) or (auto<>'Y') then
            begin
                gotoxy(1,(i mod 23)+1);
                write(' '+inputs[i]+' ');
            end;
            inputs[i] := inputs[i]+'1';
            if ManualEdit='Y' then
            begin
                repeat
                    if AutoFill = 3 then
                        readln(desired[i])
                    else
                        desired[i] := AutoFill;
                    if (desired[i] > 2) and (desired[i]<6) then
                    begin
                        AutoFill := desired[i] - 3;
                        desired[i] := AutoFill;
                    end;
                until (desired[i]>=0) and (desired[i]<6);
            end
        end
    end
end;

```

```

else
  if (auto='N') and (q='N') then
    begin
      desired[i] := bit(i,dimension) and (bit(i,dimension-1) or bit(i,dimension-2));
    end;
  if (dimension < 5) or (auto <> 'Y') then
    begin
      gotoxy(dimension+3,(i mod 23)+1);
      write(desired[i]);
    end;
  if desired[i] = 1 then
    begin
      inc(one_counter);
      SITV[1,1,one_counter] := i;
      SITV[1,0,one_counter] := 0;
    end
  else
    if desired[i] = 0 then
      begin
        inc(zero_counter);
        SITV[0,1,zero_counter] := i;
        SITV[0,0,zero_counter] := 0;
      end;
  if auto<>'Y' then
    if ( (i+1) mod 23 = 0 ) then
      begin
        if FastAnalyze='N' then
          TmpChar := readkey;
          for k:=1 to 24 do
            begin
              gotoxy(1,k);
              write(' ':dimension+4);
            end;
          end;
        end;
      end;
end;
end;

procedure analyzer;
var
  i      : byte;
  j      : word;
begin
  if auto='N' then NewTry := True;
  input_generator;
  gotoxy(dimension+8,1);
  write('Number of ones      = ',one_counter:5);
  gotoxy(dimension+8,2);
  write('Number of zeros    = ',zero_counter:5);
  MinAlgo := 'J';
  MinNeuron := 2*Max_dimension+1;
  OriginalAlgo := Algorithm;
  if (Algorithm = 'A') and (MinRealize='Y') then
    begin
      i:=1;
      repeat
        case i of
          1 : Algorithm := 'A';
          2 : Algorithm := 'J';
          3 : Algorithm := 'N';
          4 : Algorithm := 'E';
        end;
        TrainGeometrically;
        one_counter := 0;
        zero_counter := 0;
        for j := 0 to ipower(2,dimension)-1 do
          begin
            if desired[j] = 1 then
              begin
                inc(one_counter);
                SITV[1,1,one_counter] := j;
                SITV[1,0,one_counter] := 0;
              end;
            if desired[j] = 0 then
              begin
                inc(zero_counter);
                SITV[0,1,zero_counter] := j;
                SITV[0,0,zero_counter] := 0;
              end;
            end;
          if MinNeuron > Neuron then
            begin
              MinAlgo := Algorithm;
              MinNeuron := Neuron;
            end;
          inc(i);
        until (i>4) or (MinNeuron < 3);
        Algorithm := MinAlgo;

```

```

        MinNeuron:=2*max dimension+1;
        TrainGeometrically;
    end
else
    TrainGeometrically;
if auto='Y' then
    Algorithm := OriginalAlgo;
find_output;
if Not Error then
    begin
        inc(MaxNrnDim[neuron]);
        if (neuron > NeuronLimit) and (Auto='Y') then
            begin
                inc(MaxNrn);
                Append(NeuronFileName);
                writeln(NeuronFileName,FuncNo,' exceeds the Neuron Limit !!!! ');
                Close(NeuronFileName);
            end;
        if neuron = 1 then
            begin
                LinearSeparable := True;
                gotoxy(dimension+40,7);
                write('LinSeparable  = ',MaxNrnDim[1]:10);
            end;
        if auto='Y' Then
            begin
                inc(SolutionOK);
                Append(NeuronFileName);
                writeln(NeuronFileName,FuncNo,' = ',Neuron,' of ',NeuronLimit);
                Close(NeuronFileName);
                gotoxy(dimension+40,4);
                write('Sol OK      = ',SolutionOK:10);
                gotoxy(dimension+40,9);
                write('MxNeuron Error = ',MaxNrn:10);
                if neuron = 1 then
                    begin
                        gotoxy(dimension+40,11+1);
                        write(' 1 Neuron      = ',MaxNrnDim[1]:10);
                    end
                else
                    begin
                        if neuron > dimension then
                            TextColor(yellow);
                        gotoxy(dimension+40,11+neuron);
                        write(Neuron:2,' Neurons      = ',MaxNrnDim[neuron]:10);
                        TextColor(white);
                    end;
                if neuron > MaxNeurons then
                    begin
                        MaxNeurons := neuron;
                        MaxFuncNo := FuncNo;
                        gotoxy(dimension+8,4);
                        write('Max No of Neurons = ',MaxNeurons:10);
                        gotoxy(dimension+8,5);
                        write('Func No of Neuron = ',MaxFuncNo:10);
                    end;
                end
            end
        else
            begin
                Append(NeuronFileName);
                writeln(NeuronFileName,'This Function = ',Neuron,' of ',NeuronLimit);
                Close(NeuronFileName);
            end;
        if FastAnalyze='N' then
            begin
                gotoxy(dimension+11,22);
                write('Press any key to see the weights...');
                TmpChar := ReadKey;
                clrscr;
                gotoxy(1,1);
                write('Neuron':6);
                for i:= 0 to dimension do
                    write('W[':4,i,']');
                for j := 1 to neuron do
                    begin
                        if ((2*j+1) mod 21) = 0 then
                            begin
                                readln;
                                clrscr;
                            end;
                        gotoxy(1,(2*j+1) mod 21);
                        write(j:6);
                        for i := 0 to dimension do
                            write(weight[j,i]:6);
                        end;
                    end
                if neuron > 8 then
                    begin

```

```

        readln;
        clrscr;
        i:=1;
    end
    else
        i:= wherey+2;
        gotoxy(1,i);
        write('Output Weights are');
        gotoxy(1,i+2);
        write('OW[10y+x]':9,'0':6);
        inc(i,4);
        for j:=1 to 9 do
            write(j:7);
        for j := 0 to neuron do
            begin
                if (j mod 10 = 0) then
                    begin
                        if j<>0 then
                            inc(i,1);
                        if i>23 then
                            begin
                                readln;
                                i:=1;
                                clrscr;
                            end;
                        gotoxy(1,i);
                        write(j div 10 : 6);
                    end;
                gotoxy(10+7*(j mod 10),i);
                write(NeuronWeight[j]:6);
            end;
        for zin := 0 to 5 do
            begin
                sound(700);
                delay(120);
                nosound;
                delay(60);
            end;
        end;
    end
else
    begin
        gotoxy(dimension+40,1);
        write('No neural network      ');
        Append(NeuronFileName);
        writeln(NeuronFileName,FuncNo,' has No Neural NetWork !!
            It must have maximum ',NeuronLimit,' neurons..');
        Close(NeuronFileName);
        if auto='Y' Then
            begin
                inc(SolutionNone);
                gotoxy(dimension+40,5);
                write('Sol None      = ',SolutionNone:10);
            end;
        end;
    if FastAnalyze='N' then
        begin
            gotoxy(dimension+8,25);
            write('FAR~'+Number);
            gotoxy(dimension+8,23);
            write(' ':32);
            gotoxy(dimension+8,23);
            write('Do you want to try again ? ');
            ask_yes_no(question);
            if question='Y' then
                clrscr;
            end
        else
            if TmpChar <> '0' then
                begin
                    question:='Y';
                    if TmpChar = '2' then
                        begin
                            ToFile('W');
                            ToFile('I');
                            TmpChar := ' ';
                        end;
                    end;
                else
                    question:='N';
            end;
    end;

procedure auto_input_generator;
var
    carrier : byte;
    j       : longint;
    z       : char;

```

```

i      : word;
begin
  FuncNo := 0;
  OverFlow := False;
  for j:= 0 to ipower(2,dimension)-1 do
    desired[j] := 0;
  FuncNo := StartFuncNo;
  While (FuncNo <> 0) do
    begin
      j:=Trunc( ln (FuncNo+0.5) / ln (2) );
      desired[j] := 1;
      FuncNo := FuncNo mod ipower(2,j);
    end;
  if StartFuncNo = ipower(2,30)-1+ipower(2,30) then
    begin
      FuncNo := 0;
      carrier := 1;
    end
  else
    begin
      FuncNo := StartFuncNo;
      carrier := 0;
    end;
  while (question<>'N') and (not OverFlow) and (not NewTry) do
    begin
      for j:=0 to ipower(2,dimension)-1 do
        begin
          desired[j] := (1-desired[j]) * carrier + desired[j] * (1-carrier);
          if carrier = 1 then
            if desired[j] = 0 then
              carrier := 1
            else
              carrier := 0;
          end;
        end;
      OverFlow := Carrier=1;
      carrier:=1;
      if Not OverFlow then
        begin
          gotoxy(dimension+40,2);
          write('Function No = ',FuncNo:10);
          gotoxy(dimension+40,4);
          write('Sol OK = ',SolutionOK:10);
          gotoxy(dimension+40,5);
          write('Sol None = ',SolutionNone:10);
          gotoxy(dimension+40,7);
          write('LinSeparable = ',MaxNrnDim[1]:10);
          gotoxy(dimension+40,9);
          write('MxNeuron Error = ',MaxNrn:10);
          gotoxy(dimension+40,11+1);
          write(' 1 Neuron = ',MaxNrnDim[1]:10);
          for i := 2 to dimension do
            begin
              gotoxy(dimension+40,11+i);
              write(i:2,' Neurons = ',MaxNrnDim[i]:10);
            end;
          gotoxy(dimension+8,4);
          write('Max No of Neurons = ',MaxNeurons:10);
          gotoxy(dimension+8,5);
          write('Func No of Neuron = ',MaxFuncNo:10);
          TextColor(yellow);
          TextBackground(blue);
          gotoxy(15,24);
          write('Program Durdurmak ~#in 0 TuYuna BasInz... FAR-'+Number);
          TextColor(white);
          TextBackground(blue);
          gotoxy(1,1);
          analyzer;
          if keypressed then
            begin
              TmpChar := readkey;
              if TmpChar <> '0' then
                question := 'Y'
              else
                question := 'N';
            end;
          if (FuncNo mod (BackUpFrequency-dimension*((BackupFrequency div 10)-1)) = 0)
            or (FuncNo=EndFuncNo) or (question='N') or OverFlow or NewTry then
            ToFile('W');
          if ((FuncNo mod FileChangeFrequency=0) and (FuncNo<>0) and
            (BackupProcess='Y')) or (TmpChar='2') then
            begin
              ToFile('W');
              ToFile('I');
              TmpChar:=' ';
            end;
          end;
        end;
      end
    end
  end
end


```

```

        else
            NewTry := auto='N';
            inc(FuncNo);
            if FuncNo > EndFuncNo then
                break;
        end;
    end;
end;

(Main Program)
begin
    clrscr;
    question := ' ';
    repeat
        TextColor(white);
        TextBackGround(blue);
        clrscr;
        initialization;
        clrscr;
        if auto='Y' then
            auto_input_generator
        else
            begin
                analyzer;
                Tofile('W');
            end;
        until (question = 'N') or (not NewTry);
        gotoxy(1,25);
        for zin := 7 to 9 do
            begin
                sound(zin*450);
                delay(200);
                nosound;
                delay(10);
            end;
    end;
end.

```



CURRICULUM VITAE

Ersan ALFAN was born in Istanbul on June 22, 1971. Having completed primary school education in Istanbul, 1982, he was graduated from Dođuş College in 1989. In 1989, he became the championship of Marmara region in TÜBİTAK Mathematics Competition.

In 1993, he has been graduated from Istanbul Technical University, Electronics and Communication Department with a good degree. In 1994, he began to pursue his master of science education in Istanbul Technical University Electronics and Communication Department.

