NEURONALE NETZE ALS MODELL BOOLESCHER FUNKTIONEN

Von der Fakultät für Mathematik und Informatik
der Technischen Universität Bergakademie Freiberg

genehmigte

DISSERTATION

zur Erlangung des akademischen Grades

Doktor-Ingenieur
(Dr.-Ing.)

vorgelegt

von M.Sc. Roman Kohut

geboren am 19. August 1979 in Werchnja

Gutachter: Prof. Dr.-Ing. habil. Bernd Steinbach, Freiberg
Prof. Dr.-Ing. habil. Wolfgang Rehm, Chemnitz
Prof. Dr. rer. nat. habil. Günther Palm, Ulm

Tag der Verleihung: 30. Mai 2007

ABSTRACT

This dissertation presents neural networks as a model of Boolean functions. A new kind of Boolean neural networks (BNN) was developed. The basic element of Boolean neural networks is a new Boolean neuron (BN) that operates in contrast to classical neuron by Boolean signals directly and uses Boolean operations for processing only. A sequential algorithm was developed in order to train the BNN. This algorithm guarantees a quick convergence and needs therefore a short training time. A new created procedure to synthesize the architecture of the BNN based on this training algorithm. Furthermore, the developed training forms a new special decomposition method for Boolean functions. Neural networks can be realized in both software and hardware. The very high complexity of the hardware realization of usual neural networks was simplified substantially by the use of BN and BNN. The number of necessary CLBs (configurable logic blocks) for the implementation of a single neuron was reduced by about two orders of magnitudes. A one Boolean neuron is mapped onto one single LUT (lookup table) directly. The training algorithm of the BNN was adapted for a very compact mapping of the BNN into a FPGA structure. The synthesis effort for hardware implementation of BNN could be reduced significantly by both the specification of the BNN using UML models and the application of the MDA technique for hardware/software-Co-design.

* * *

In der vorliegenden Arbeit werden die Darstellungsmöglichkeiten Boolescher Funktionen durch Neuronale Netze untersucht und eine neue Art von Booleschen Neuronalen Netzen (BNN) entwickelt. Das Basiselement Boolescher Neuronaler Netze ist ein neuartiges Boolesches Neuron (BN), das im Gegensatz zum klassischen Neuron direkt mit Booleschen Signalen operiert und dafür ausschließlich Boolesche Operationen benutzt. Für das Training der BNN wurde ein sequentieller Algorithmus erarbeitet, der eine schnelle Konvergenz garantiert und somit eine kurze Trainingzeit benötigt. Dieser Trainingsalgorithmus bildet die Grundlage eines neuen geschaffenen Verfahrens zur Architektursynthese der BNN. Das entwickelte Training stellt darüber hinaus ein spezielles Dekompositionsverfahren Boolescher Funktionen dar.
Neuronale Netze können sowohl in Software als auch in Hardware realisiert werden. Der sehr hohe Aufwand der Hardware-Realisierung üblicher Neuronaler Netzen wurde durch die Verwendung von BN und BNN wesentlich vereinfacht. Die Anzahl erforderlicher CLBs (configurable logic blocks) zur Realisierung eines Neurons wurde um 2 Größenordnungen verringert. Ein Boolesche Neuron wird direkt auf eine einzige LUT (lookup table) abgebildet. Für diese sehr kompakte Abbildung der BNN in eine FPGA-Struktur wurde der Trainingsalgorithmus des BNN angepasst. Durch die Spezifikation der BNN mit UML-Modellen und die Anwendung der MDA-Technologie zum Hardware/Software-Codesign konnte der Syntheseaufwand für Hardware-Realisierung von BNN signifikant verringert werden.

VORWORT

Die vorliegende Arbeit entstand in den Jahren 2002-2006 einschließlich meines Studiums im Graduiertenkolleg "Räumliche Statistik" am Institut für Informatik der Technischen Universität Bergakademie Freiberg. Dem Freistaat Sachsen und der Deutschen Forschungsgemeinschaft, die die finanzielle Förderung der Arbeit ermöglichen, seien an dieser Stelle gedankt.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. habil. Bernd Steinbach, der mir das interessante Thema überließ und diese Arbeit ermöglichte. Durch seine jederzeit gewährte Unterstützung und fachliche Hinweise hat er beträchtlich zum Gelingen der Arbeit beigetragen.
Bei Herrn Prof. Dr.-Ing. habil. Wolfgang Rehm, Leiter der Professur Rechnerarchitektur an der Technischen Universität Chemnitz, Fakultät für Informatik und Herrn Prof. Dr. rer. nat. habil. Günther Palm, Leiter der Abteilung Neuroinformatik an der Universität Ulm, Fakultät für Informatik, die die weiteren Gutachten übernahmen, möchte ich mich herzlich bedanken.

Für die anhaltende Unterstützung meines Promotionsvorhabens während meines Studium im Graduiertenkolleg bin ich dankbar der Koordinatorin des Graduiertenkollegs Frau Gudrun Seifert, dem Sprecher des Graduiertenkollegs Herrn Prof. Dr.-Ing. habil. Dietrich Stoyan, den Kollegiatinnen und Kollegiaten.

Schließlich gilt der Dank meinen Freunden und Kollegen. Hervorheben möchte ich Frau Irmgard Gugel, die sich der Mühe unterzog, sehr ausführlich das Script durchzusehen und mich auf Mängel aufmerksam zu machen.
Ganz besonderer Dank gilt meiner Frau Olha und meinen Eltern, Mykola und Jaroslawa Kohut für ihre stete Liebe und Geduld. Sie unterstützten mich während meiner Ausbildung und Doktorandenzeit geduldig und nach Kräften. Щиро дякую за Вашу турботу, любов, підтримку і розуміння, за те, що Ви завжди були поруч. This thesis is dedicated to all of them.

Roman Kohut Freiberg, November 2006

INHALTSVERZEICHNIS

SYMBOLVERZEICHNIS

Operatoren und Mengenoperatoren

$A=\{a_1, a_2, \ldots, a_n\}$	Menge von Elementen $a_1, a_2, \ldots, a_n$
$a \in A$	a Element von A
$\forall$	für alle
U_{in}	Menge der Eingabeneuronen,
U_{out}	Menge der Ausgabeneuronen
U_{hidden}	Menge der versteckten/verborgenen Neuronen
$A \subseteq B$	Teilmenge oder gleiche Menge
$A \cup B$	Vereinigung
$A \cap B$	Durchschnitt
$\varnothing$	leere Menge
$\lvert A \rvert$	Elementesanzahl der Menge A
$A \rightarrow B$	Abbildung der Menge A in die Menge B
x_i	Boolesche Variable
$\boldsymbol{x}=(x_1, x_2, \ldots, x_n)$	Vektor aus n Booleschen Variablen
$\lvert \boldsymbol{x} \rvert$	Länge des Vektors $\boldsymbol{x}$
a^T	Transponierter Vektor
$a\ b$	Skalarprodukt zweier Vektoren
$\boldsymbol{B}=\{0,1\}$	Boolescher Raum
$\boldsymbol{B}^n$	n-dimensionaler Boolescher Raum

Funktionen

$f(\boldsymbol{x})$	Boolesche Funktion
f_0	Nullfunktion, $f(\boldsymbol{x})=0$
f_1	Einsfunktion, $f(\boldsymbol{x})=1$
$\mathbf{F}_2^{\,n}$	Menge Boolescher Funktionen von 2 Variablen
$\frac{\partial f(\boldsymbol{x})}{\partial x_i}$	partielle Ableitung der Booleschen Funktion $f(\boldsymbol{x})$ nach x_i

Boolesche Operatoren

$not(a)$, $\bar{a}$	Negation
$a \wedge b$	Konjunktion
$a \vee b$	Disjunktion
$a \oplus b$	Antivalenz
$a \odot b$	Äquivalenz
D_E ($\boldsymbol{a}$, $\boldsymbol{b}$)	Euklidischer Abstand zwischen den Vektoren $\boldsymbol{a}$ und $\boldsymbol{b}$
D_{HM} ($\boldsymbol{a}$, $\boldsymbol{b}$)	Hemming-Abstand zwischen den Vektoren $\boldsymbol{a}$ und $\boldsymbol{b}$
s_{ab}	Ähnlichkeit der Vektoren $\boldsymbol{a}$ und $\boldsymbol{b}$

Neuronen

act_t	Aktivierungszustand zum Zeitpunkt t
t	Zeitpunkt
$act(t)$, f_{act}	Aktivierungsfunktion
net	Netzeingabe
f_{net}	Netzeingabefunktion
f_{out}	Ausgabefunktion
f_T	Transferfunktion
$\lambda_1, ..., \lambda_l$; $\theta_1, ..., \theta_k$	Parameter
$w_1, w_2, \ldots, w_{Nx}$	Verbindungsgewichte
$x_1, x_2, ..., x_{Nx}$	Eingabesignale
y, y_i	Ausgabesignal
ω_i	Gewichtsfunktion für die Eingabe i
N_z	Neuronenanzahl in der verborgenen Schicht
w_{ij} , v_{ij} , u_{ij}	zugeordnetes Gewicht
k, $k()$, $k^{(i)}$	k-Funktion
$[w_{ij}]$, $[v_{jk}]$	Gewichtsmatrix
$\boldsymbol{w}$	Gewichtsvektor
θ	Schwellwert
F, ψ	Schwellwertfunktion

Andere

'*', '-', '~'	„don't care"
$\pi(\boldsymbol{a}^k)$	innere Abbildung von $\boldsymbol{a}^k$
$[\boldsymbol{a}^k]$	Klasse von inneren Abbildungen $\pi(\boldsymbol{a}^k)$
ε^k	Kostenfunktion für „tiling" Trainingsalgorithmus
$E(X_i)$	Eingangskodierungsfunktion
$\boldsymbol{\Omega}$	Kumulationsoperator
$\boldsymbol{\Omega}\,[\boldsymbol{x}, \boldsymbol{w}]$	Kumulationsoperator mit den Vektoreingaben $\boldsymbol{x}$ und $\boldsymbol{w}$
Ω_0^{Ni}	Kumulationsoperator mit den Grenzen 0 und N_i
F_j	Polynom-Operator
G	Graph
sgn()	Vorzeichenfunktion (mit Wertebereich{-1,0,1})

ABKÜRZUNGSVERZEICHNIS

ABNN	Adaptierbares Boolesches Neuronales Netz
AF	Antivalenzform
ANN	Artificial Neural Network
BDD	Binary Decision Diagram (Binäre Lösungsdiagramm)
BF	Boolesche Funktion
BLTA	Boolean-Like Training Algorithm
BN	Boolesches Neuron
BNN	Boolesches Neuronales Netz
BP	Back Propagation
BV	Binärvektor
BVL	Binärvektorliste
CLB	Configurable Logic Block
CPU	Central Processing Unit
CSCLA	Constructive Set Covering Learning Algorithm
DF	Disjunktive Form
DFF	D-Flip-Flop
DRAM	Dynamic Random Access Memory
EEPROM	Electrical Erasable and Programable ROM
EF	Äquivalenzform
ES	Embedded Systems
ETL	Expand-and-Truncate Learning
FBNN	Feed Back Neural Network
FF	Flip-Flop
FFNN	Feed Forward Neural Network
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSMD	Finite State Machine with Datapath
FTF	Functional on the Tabular Functions
HC	Hamming-Clustering
HwNN	Hardware Neuronales Netz
IETL	Improved Expand-and-Truncate Learning
IOB	Input Output Block
ITA	Iterativer Trainingsalgorithmus
KF	Konjunktive Form
KI	Künstliche Intelligenz
KN	Künstliches Neuron
KNN	Künstliches Neuronales Netz
LCA	Logic Cell Array
LUT	Lookup Table

MAL	MOCCA Action Language
MCETL	Multi-Core Expand-and-Truncate Learning
MCL	Multi-Core Learning
MDA	Model-Driven-Architecture
MLBNN	Multi-Layer Boolean Neural Network
MLNN	Multi-Layer Neural Network
MOCCA	MOdel Compiler for reConfigurable Architecture
MPGA	Masked-Programmable Gate Array
MUX	Multiplexer
NETLA	Newly Expanded and Truncated Learning Algorithm
NN	Neuronales Netz
NRE	Non-Recurring Engineering
PDA	Platform Dependent Application
PE	Processing Element
PIM	Platform Independent Model
PLA	Programmable Logic Array
PSM	Platform Specific Model
RAM	Random Access Memory
ROM	Read-Only Memory
RTR	Run-Time Reconfigurable
RUP	Rational Unified Process
SITV	Set of Included True Vertices
SRAM	Static Random Access Memory
STA	Sequentieller Trainingsalgorithmus
SWL	Sequential Window Learning
TPM	Target Platform Model
TV	Ternärvektor
TVL	Ternary Vector List (Ternärvektorliste)
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Kapitel 1

Einleitung

Die ständig wachsenden Anforderungen verschiedener Zweige der Computerwissenschaft und Informationstechnologie haben in den letzten 20 Jahren eine besonders starke Entwicklung der Künstlichen Intelligenz (KI) und Künstlichen Neuronalen Netze (KNN) hervorgerufen. KNN werden häufig in den Anwendungsbereichen Data Mining, Quantum Computing und Robotertechnik [43], [66], [83], [125], aber auch darüber hinaus angewendet. Eine der wichtigsten Rollen spielen KNN bei der Modellierung von verschiedenen Prozessen und Objekten. Obwohl es sehr viele Anwendungsgebiete von KNN gibt, wurden KNN anfangs zur Modellierung Boolescher Funktionen entwickelt [99]. Künstliche Neuronale Netze, die für die Modellierung Boolescher Daten Verwendung finden, werden Boolesche Neuronale Netze (BNN) genannt. Aus der Literatur sind viele Anwendungsbeispiele von BNN für Data Mining [66], [173], [175], Klassifizierung [142], Mustererkennung [74] etc. bekannt. Bei der Anwendung von KNN und noch stärker bei BNN gibt es noch viele offene Fragen. Zum Beispiel, die Entwicklung innovativer Methoden zum effektiven Training Boolescher Neuronaler Netze ist zur Zeit immer noch ein aktuelles Thema der laufenden Forschungen [41], [76], [91], [165]. Die Flexibilität und in den letzten Jahren stark gestiegene Leistungsfähigkeit von „Run-Time Reconfigurable“ (RTR)-Systemen haben die Vorteile von Hardware-Realisierungen von KNN und insbesondere Boolescher Neuronaler Netze verstärkt [11], [48], [59], [180].

In den letzten 20 bis 30 Jahren sind die wissenschaftlichen Forschungen an KNN als Modell Boolescher Funktionen in den Hintergrund getreten und KNN wurden meistens für andere Aufgabengebiete eingesetzt. In dieser Arbeit werden Künstliche Neuronale Netze in ihrer ursprünglichen Intention betrachtet und zur Modellierung Boolescher Funktionen verwendet. Boolesche Funktionen werden insbesondere für Design- und Entwurfsprozesse digitaler elektronischer Schaltungen angewendet. Seit der Entwicklung der ersten integrierten Schaltung in den 60er Jahren sind die Entwurfseinheit und die Packungsdichte auf den Chips millionenfach gestiegen. Die ständig zunehmende Entwicklungsgeschwindigkeit der Mikroelektronik erfordert den Entwurf immer komplexerer Boolescher Schaltnetzwerke. Dabei müssen ständig neue leistungsfähigere Designmetho-

den entwickelt werden. Für diese werden kompaktere Datenstrukturen für Boolesche Funktionen und deren schnelle Verarbeitung benötigt.
Aus diesen aktuellen Anforderungen motivieren die Forschungsziele dieser Arbeit. Es sollen die Fragen beantwortet werden, ob sich Neuronale Netze als alternative kompakte Datenstruktur für Boolesche Funktionen eignen und ob eine effiziente Verarbeitung Boolesche Funktionen unter Verwendung von BNN möglich ist.
Die Forschungen auf dem Gebiet Boolescher Funktionen haben in den letzten Jahren bestätigt, dass dekompositorische Syntheseverfahren den überdeckenden Syntheseverfahren wesentlich überlegen sind. Ein ungelöstes Problem besteht aber immer noch darin, dass aus lokalen Entwurfsentscheidungen nicht auf das globale Entwurfsergebnis geschlossen werden kann. In Anbetracht der Fähigkeiten Neuronale Netze zur Optimierung soll in dieser Dissertation die Frage beantwortet werden, welchen Beitrag Neuronale Netze zum Finden geeigneter Dekompositionsfunktionen für Boolescher Funktionen leisten können. Dafür ist eine neue Art der Booleschen Neuronalen Netze für die kompakte Darstellung und schnelle Berechnung Boolescher Funktionen zu entwickeln.
Die Anwendung Neuronaler Netze setzt das Training des Netzes auf die konkrete Aufgabe voraus. Basierend auf dem Verfahren der Back Propagation [173], [134] zum Training von Neuronalen Netze wurden mehrere iterativen Trainingsalgorithmen (ITA) entwickelt. Obwohl die iterativen Trainingsalgorithmen in der Praxis das Training Neuronaler Netze dominieren, sind mehre bisher nicht gelöste Probleme von ITA bekannt. Dazu gehören zum Beispiel lokale Minima, flache Plateaus, die Wahl der Schrittweite und die Wahl des Dynamikbereiches [179]. KNN mit ITA garantieren keine schnelle Konvergenz und benötigen oft eine lange Trainingszeit, die überproportional von der Netzgröße und von der Größe der Lerndatenmenge abhängt. Kritisch ist weiterhin, dass unbekannt ist, ob für eine vorgegebene Netzstruktur die gewünschte Fehlergrenze als Abbruchkriterium überhaupt erreicht werden kann.
Um diese Probleme zu überwinden wurden als Alternative die sequentiellen Trainingsalgorithmen (STA) vorgeschlagen [19], [42], [93]. Im Vergleich zu ITA haben STA viele Vorteile aber haben auch eigene Probleme. Zu den Nachteilen von STA gehört das rechenaufwendige Training eines einzelnen Schwellwertneurons. Das Netzdesign ergibt sich im Verlauf des sequentiellen Trainings, so dass mit Sicherheit ein nutzbares Neuronales Netz entsteht. Allerdings garantieren die bekannte STA nicht, dass eine minimale Netzstruktur erzeugt wird [8].
Die bisher betrachteten Probleme von Trainingsalgorithmen wirken sich verstärkt bei Booleschen Neuronalen Netzen aus. Ein weiteres Problem von BNN mit sequentiellen Trainingsalgorithmen besteht darin, dass man beim Training ein extrem großes Speichervolumen braucht [82], [154].

Ein Ziel dieser Arbeit besteht darin die geschilderten Probleme des Trainings von BNN zu überwinden oder wenigstens stark abzuschwächen. Durch die Konzentration auf das Paradigma des sequentiellen Trainings treten die Nachteile der iterativen Trainingsmethoden nicht auf und die Netzstruktur kann implizit ermittelt werden. Mit den zu erarbeitenden sequentiellen Trainingsmethoden soll eine schnelle Konvergenz und eine optimale Netzstruktur der BNN erreicht werden. Diese Trainingsmethoden sollen mit möglichst kleinem Speichervolumen auskommen und die Charakteristika für jedes einzelne Neuron durch einfache Berechnungen finden.

Das Problemfeld von BNN ist sehr groß. Bekannte Verfahren für Boolesche Funktionen könnten zur Weiterentwicklung von Trainingsmethoden von Neuronalen Netzen dienen. Mit BNN könnten Boolesche Funktionen mit gewünschten Eigenschaften wie zum Beispiel geringer Leistungsverbrauch oder kurze Laufzeit aus Funktionsmengen ausgewählt werden. Sehr interessant wäre es auch zu analysieren, welche Abhängigkeit zwischen der Komplexität von Booleschen Funktionen und der Komplexität von BNN bestehen. Dieses Aufgabenspektrum geht weit über die Möglichkeiten hinaus, die in einer Dissertation fundiert bearbeitet werden kann. Deshalb bleiben die in diesem Abschnitt genannten Problemen zukünftigen Arbeiten überlassen. Beachtliche Teilergebnisse zur Komplexität Neuronaler Netze findet man in [142], [120]. Ihre Kombination mit dem bekannten Wissen zur Komplexität Boolescher Funktionen [168], [49] könnte zu neuen umfassenderen Erkenntnissen führen.

Da es in dieser Arbeit immer um Künstliche Neuronen (KN) und Künstliche Neuronale Netze geht, werden sie in dieser Arbeit einfach Neuronen und Neuronale Netze (NN) genannt. Neuronale Netze zur Modellierung Boolescher Daten werden als Boolesche Neuronale Netze (BNN) bezeichnet.

Die vorliegende Arbeit hat folgenden Aufbau. Die zum Verständnis der Arbeit erforderlichen Grundlagen über Booleschen Funktionen, Neuronalen Netzen und RTR-Systeme werden im **Kapitel 2** vorgestellt. Das **Kapitel 3** befasst sich mit klassischen Neuronen und Booleschen Neuronalen Netzen. Dabei werden Lösungsmöglichkeiten des Problems der linearen Separierbarkeit betrachtet. Auch werden wesentliche bekannte Ergebnisse Boolescher Neuronaler Netze dargestellt, ihre Vor- und Nachteile bewertet und kritisch analysiert. Im **Kapitel 4** werden das Boolesche Neuron und das Boolesche Neuronale Netz beschrieben und ein geeigneter Trainingsalgorithmus entwickelt. Die Möglichkeit des Einsatzes der RTR-Architektur bei der Hardware-Realisierung von Booleschen Neuronalen Netzen wird im **Kapitel 5** analysiert. Zur Optimierung wird der Trainingsalgorithmus des BNN an die FPGA-Eigenschaften angepasst. An das zusammenfassende **Kapitel 6** schließen sich die Bibliographie und einige Anhänge mit weiterführenden Informationen an.

Kapitel 2

Grundlagen

2.1 Boolesche Funktionen

2.1.1 Boolesche Funktionen und ihre Darstellung

Eine **Boolesche Variable** ist eine Variable, die nur die Werte 0 oder 1 (falsch oder wahr, negativ oder positiv) annehmen kann. Einen Vektor aus n Booleschen Variablen $\boldsymbol{x}=(x_1, x_2, x_3, \ldots, x_n)$: $x_i \in \boldsymbol{B}$ nennt man **Binärvektor** (**BV**) der Länge n, wobei $\boldsymbol{B}=\{0,1\}$ der Boolesche Raum ist. Ein n-dimensionaler **Boolescher Raum** $\boldsymbol{B}^n$ wird durch die Menge aller möglichen Binärvektoren der Länge n definiert [130]:

$$\boldsymbol{B}^n = \{\boldsymbol{x} \mid \boldsymbol{x} = (x_1, x_2, \ldots, x_n) \text{ mit } x_i \in \boldsymbol{B}\} \tag{2.1}$$

Die Zahl n bestimmt die Raumdimension, n ist gleich der Länge der Binärvektoren und bestimmt die Elementanzahl des Raumes zu 2^n.

Eine Boolesche **Operation** nennt man eine mathematische Operation, die auf einem bzw. mehreren Booleschen Elementen (Variable, Konstante) definiert ist und ein Boolesches Resultat ergibt. Die auf mehreren Booleschen Variablen definierten Booleschen Operationen werden auch **Verknüpfungsoperationen** genannt [126]. Die Operationszeichen werden **Operatoren** genannt. In der Tabelle 2.1 sind die Booleschen Grundoperationen, die auch als Verknüpfungsoperationen verwendet werden können, gesammelt.

Tabelle 2.1 Boolesche Grundoperationen

Operation	Schreibweise	Verknüpfung
Negation	$\overline{a}$	-
Konjunktion	$a \wedge b$	AND-Verknüpfung
Disjunktion	$a \vee b$	OR-Verknüpfung
Antivalenz	$a \oplus b$	EXOR-Verknüpfung
Äquivalenz	$a \odot b$	Äquivalenz -Verknüpfung

In dieser Arbeit werden die in der Tabelle 2.1 angeführten Operationszeichen verwendet, wobei eine Vereinbarung gilt, dass AND-Zeichen weggelassen werden können.
Für die angegebenen Booleschen Grundoperationen gelten die Gesetze der Idempotenz, Verknüpfung mit Null und Eins, Verknüpfung mit dem Komplement, die Gesetze der Kommutativität, Assoziativität, Distributivität und Absorption und die Sätze von de Morgan und von Stone, die in der Literatur ausführlich beschrieben sind [12], [25], [126] and [130].

Definition 2.1. Eine **Boolesche Funktion** (**BF**) $y = f(x_1, x_2, \ldots, x_n)$ von n Booleschen Variablen definiert man als eindeutige Abbildung des n-dimensionalen Booleschen Raumes auf den Booleschen Raum ersten Grades $\boldsymbol{B}^n \rightarrow \boldsymbol{B}$.

Jede Boolesche Funktion hängt mindestens von einer Booleschen Variable ab und produziert nur einen Ausgang y, der zum Raum $\boldsymbol{B}$ gehört. Die Verknüpfungen aus der Tabelle 2.1 können auch als Boolesche Funktionen betrachtet werden.
Wenn ein BV aus n Booleschen Variablen besteht, dann gibt es

$$\left|\mathbf{F}_2^{\,n}\right| = 2^{\left|\boldsymbol{B}^n\right|} = 2^{2^n} \tag{2.2}$$

verschiedene Boolesche Funktionen, die auf diesen BV definiert werden können. Dabei bezeichnet $\mathbf{F}_2^{\,n}$ eine Menge aller Booleschen Funktionen, die von n Booleschen Variablen abhängen.
Zum Beispiel gibt es $2^{2^1} = 4$ Boolesche Funktionen, die auf eine einzige Boolesche Variable a definiert sind. Das sind 0, 1, a und $\overline{a}$.
In der Tabelle 2.2 werden elementare Boolesche Funktionen: Negation (NOT), Disjunktion (OR), Konjunktion (AND), Antivalenz (EXOR) und Äquivalenz eingeführt. Dabei werden zwei Darstellungsweisen Boolescher Funktionen beschrieben durch Boolesche Ausdrücke und Wertetabellen.

Tabelle 2.2 Elementare Boolesche Funktionen

Operanden		Negation	Disjunktion	Konjunktion	Antivalenz	Äquivalenz
x_1	x_2	$f(x_1) = \overline{x_1}$	$f(x_1, x_2) = x_1 \vee x_2$	$f(x_1, x_2) = x_1 x_2$	$f(x_1, x_2) = x_1 \oplus x_2$	$f(x_1, x_2) = x_1 \odot x_2$
0	0	1	0	0	0	1
0	1	1	1	0	1	0
1	0	0	1	0	1	0
1	1	0	1	1	0	1

Die **Wertetabelle** einer auf n Variablen definierten Booleschen Funktion besteht aus n+1 Spalten (n Operandenspalten und 1 Funktionsspalte) und 2^n Zeilen. Die Funktionswerte werden in der Funktionsspalte vollständig erklärt, d.h. die Boolesche Funktion wird durch die Wertetabelle komplett beschrieben.

Boolesche Ausdrücke werden definiert durch:

1. Die Konstanten 0 und 1, die Variablen x_1, x_2,…, x_n sind Boolesche Ausdrücke.
2. Sind A und B Boolesche Ausdrücke, dann sind $\overline{A}$, $\overline{B}$, $A \vee B$, AB, $A \oplus B$ und $A \odot B$ auch Boolesche Ausdrücke.
3. Durch die Verknüpfung zweier Booleschen Ausdrücke mit einer Booleschen Operation entsteht ein neuer Boolescher Ausdruck.

In der Menge aller Booleschen Ausdrücke gibt es vier Ausdrucksformen Boolescher Funktionen, die besondere Bedeutung haben [12]:

Disjunktive Form (**DF**) ist eine Disjunktion einzelner Konjunktionen,

Antivalenzform (**AF**) ist eine Antivalenz einzelner Konjunktionen,

Konjunktive Form (**KF**) ist eine Konjunktion einzelner Disjunktionen,

Äquivalenzform (**EF**) ist eine Äquivalenz einzelner Disjunktionen.

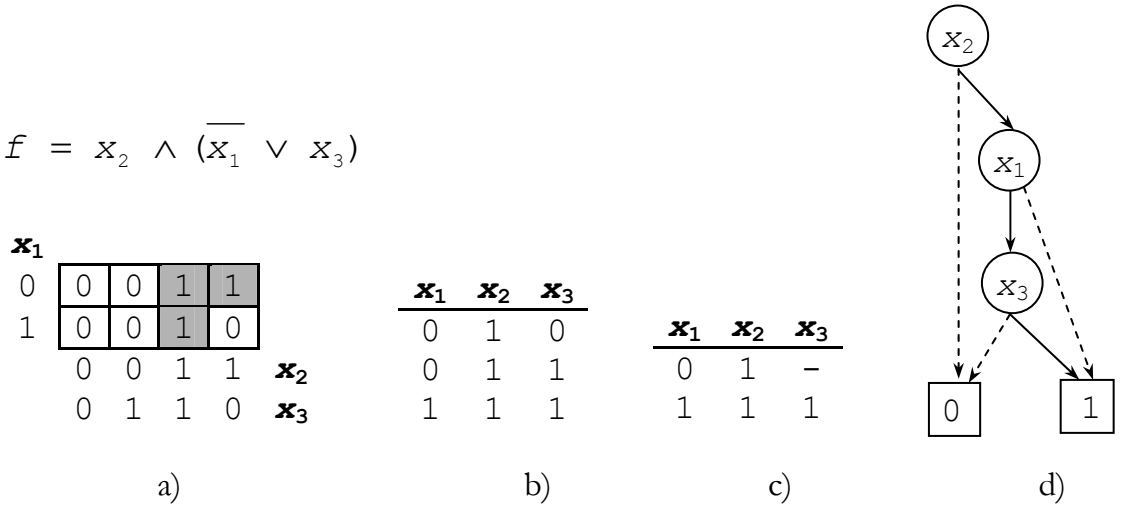
Ein Beispiel der Formen DF, AF, KF und EF für Boolesche Funktionen ist in (2.3) dargestellt.

$$\begin{aligned} f_1(\boldsymbol{x}) &= \overline{x_1}x_2 \vee x_1x_3 \\ f_2(\boldsymbol{x}) &= \overline{x_1}x_2 \oplus x_1x_3 \\ f_3(\boldsymbol{x}) &= (\overline{x_1} \vee x_2)\,(x_1 \vee x_3) \\ f_4(\boldsymbol{x}) &= (\overline{x_1} \vee x_2) \odot (x_1 \vee x_3) \end{aligned} \qquad (2.3)$$

Zur Beschreibung, Speicherung und Bearbeitung Boolescher Funktionen gibt es viele Möglichkeiten. In überwiegendem Maß basieren die Beschreibungsweisen Boolescher Funktionen entweder auf Tabellen oder Entscheidungsdiagrammen. Zu den einfachsten und bekanntesten Darstellungsweisen Boolescher Funktionen gehören **Karnaugh-Pläne – KP**, **Binärvektorlisten – BVL**, **Ternärvektorlisten – TVL**, **Binäre Entscheidungsdiagramme** (Binary Decision Diagram – **BDD**) etc. Ein Beispiel zu verschiedenen Beschreibungsweisen einer Booleschen Funktion wird in der Abbildung 2.1 gezeigt.

Ein **Karnaugh-Plan** ist eine zweidimensionale rechteckige Tabelle, deren Koordinaten in Gray-Code [25], [130] angegeben sind. Eine **Binärvektorliste** (BVL) ist eine Tabelle, in der die Variablen des Raumes und die Werte des einzelnen BV spaltengerecht untereinander geschrieben werden. Für die Bearbeitung umfangreicher Probleme haben sich **Ternärvektorlisten** als besonders effektiv erwiesen [84], [128], [129]. Als TVL kann ein Boolescher Ausdruck in jeder der vier Formen (2.3) dargestellt werden. Eine TVL besteht aus **Ternärvektoren – TV** (dreiwertige Vektoren), die die Konjunktionen von Variablen einer Funktion in disjunktiver Form oder Antivalenzform bzw. Disjunktionen von Variab-

len einer Funktion in konjunktiver Form oder Äquivalenzform abbilden. Den Spalten der TVL sind die Boolesche Variablen x_i zugeordnet. Ein TV beschreibt durch „0“, dass eine Variable negiert auftritt, durch „1“, dass eine Variable nicht negiert auftritt, oder durch „-“, dass eine Variable nicht vorhanden ist. Jeder Ternärvektor mit k Strichen „-“ repräsentiert 2^k Binärvektoren. Die ausführliche Beschreibung der Kodierung ist in [25], [47] zu finden.



a) b) c) d)

Abbildung 2.1 Darstellung einer Booleschen Funktion:
a) Karnaugh-Plan; b) BVL; c) TVL; d) BDD

Neben den TVL treten BDD als fundamentale und besonders wichtige Beschreibungsweisen hervor. BDD wurden aus Binären Entscheidungsbäumen entwickelt und stellen Boolesche Funktionen durch Graphen dar (siehe Abb. 2.1.d). Dabei soll in jedem Knoten des Graphen eine Boolesche Variable abgefragt werden. Gemäß dem Wert der entsprechenden Variablen werden Pfade durch den Graphen weiterverfolgt. Es gibt viele Erweiterungen der BDD: geordnete BDD - OBDD [31], [45] und [100], Funktionale BDD – FDD [45] und [73], Kronecker Funktionale BDD – KFDD [45], [46], [44] und [85] etc. Eine ausführliche Beschreibung von BDD ist in [85] und [137] zu finden.

Alle diese Datenstrukturen haben spezifische Vorteile in Bezug auf den Speicherbedarf für spezielle Funktionsklassen oder in Bezug auf die Kompliziertheit für bestimmte Operationen. Deshalb hängt die optimale Darstellung Boolescher Daten von ihrer Struktur und den algorithmischen Anforderungen an die zu lösende Aufgaben ab. Es gibt keine Darstellung, die für alle Fälle optimal wäre.

Boolesche Funktionen mit einem gemeinsamen Merkmal können als Klassen Boolescher Funktionen betrachtet werden. Zu den wichtigsten gehören duale und selbstduale, monotone, lineare, symmetrische, Schwellwertfunktionen und Funktionen mit einer festen Zahl von Einsen in der Wertetabelle. Die besondere Bedeutung haben auch partiell definierte Funktionen, die die Funktionsverbände bzw. Funktionsintervalle charakterisieren. Eine

ausführliche Definition und Beschreibung von Klassen Boolescher Funktionen ist in [23], [25] und [130] zu finden.
In dieser Arbeit werden lineare Boolesche Funktionen verwendet. Deswegen ist der Bergriff der Linearität Boolescher Funktionen im Folgenden gegeben.

Definition 2.2. Eine Boolesche Funktion $f(\boldsymbol{x})$ ist in der Variable x_i **linear**, wenn gilt:

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = f(x_i, \boldsymbol{x}_0) \oplus f(\overline{x_i}, \boldsymbol{x}_0) = 1 \tag{2.4}$$

wobei $\frac{\partial f(\boldsymbol{x})}{\partial x_i}$ die partielle Ableitung einer Boolesche Funktion $f(\boldsymbol{x})$ nach der Variable x_i ist [24], [25], [129], [130] und [157].

Definition 2.3. Eine Boolesche Funktion $f(\boldsymbol{x})$ ist **linear**, wenn (2.4) für jede Variable x_i des Vektors $\boldsymbol{x}$ gilt.
Die zwei weitere wichtige Ableitungsoperationen werden in dem Booleschen Differentialkalkül definiert.

Definition 2.4. Das **partielle Minimum** und das **partielle Maximum** einer Booleschen Funktion $f(\boldsymbol{x})$ nach der Variable $x_i \in \boldsymbol{x}$ werden in (2.5) und (2.6) definiert:

$$\min_{x_i} f(\boldsymbol{x}) = f(x_i, \boldsymbol{x}_0) \wedge f(\overline{x_i}, \boldsymbol{x}_0) \tag{2.5}$$

$$\max_{x_i} f(\boldsymbol{x}) = f(x_i, \boldsymbol{x}_0) \vee f(\overline{x_i}, \boldsymbol{x}_0) \tag{2.6}$$

Das **k-fache Minimum** (2.7) und das **k-fache Maximum** (2.8) einer Booleschen Funktion $f(\boldsymbol{x})$ entstehen durch die mehrfache Ausführung der partiellen Minima und Maxima nach Variablen $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$.

$$\min_{\boldsymbol{x}_k}{}^{k} f(\boldsymbol{x}) = \min_{x_{i_1}} \left(\min_{x_{i_2}} \left(\ldots \min_{x_{i_k}} f(\boldsymbol{x}) \ldots \right) \right) \tag{2.7}$$

$$\max_{\boldsymbol{x}_k}{}^{k} f(\boldsymbol{x}) = \max_{x_{i_1}} \left(\max_{x_{i_2}} \left(\ldots \max_{x_{i_k}} f(\boldsymbol{x}) \ldots \right) \right) \tag{2.8}$$

2.1.2 Euklidischer und Hamming-Abstand

Zwischen zwei Vektoren mit gleicher Länge kann eine Ähnlichkeit gemessen werden. Zur Berechnung des Ähnlichkeitsmaßes zwischen zwei beliebigen Vektoren wird oft der quadratische **euklidische Abstand** benutzt. Der euklidische Abstand wird durch (2.9) definiert.

$$D_E(\boldsymbol{a}_i, \boldsymbol{a}_j) = \sum_{k=1}^{n} (a_{ik} - a_{ik})^2, \tag{2.9}$$

wobei D_E - euklidischer Abstand zwischen den Binärvektoren $\boldsymbol{a}_i$ und $\boldsymbol{a}_j$,

a_{ik} und a_{jk} - k-te Elemente der Vektoren $\boldsymbol{a}_i$ und $\boldsymbol{a}_j$ und

n – Dimension der Eingangsvektoren.

Für die Binärvektoren $\boldsymbol{a}_i$ und $\boldsymbol{a}_j$, deren Elemente nur die Werte 0 oder 1 annehmen können, ist der quadratische euklidische Abstand (2.9) gleich dem Hamming-Abstand D_{HM} (2.10).

$$D_{HM}(\boldsymbol{a}_i, \boldsymbol{a}_j) = \sum_{k=1}^{n} a_{ik} \oplus a_{ik} \tag{2.10}$$

Deshalb benutzt man oft als Ähnlichkeitskriterium zwischen zwei Binärvektoren den euklidischen Abstand, der eine geometrische Visualisierung in bezug auf Hypersphären erlaubt.

Unter Verwendung des euklidischen Abstandes D_E und folglich des Hamming-Abstandes D_{HM} wird die Ähnlichkeit zwischen zwei beliebigen Binärvektoren $\boldsymbol{a}_i$ und $\boldsymbol{a}_j$ durch (2.11) definiert:

$$s_{ij} = n - D_E(\boldsymbol{a}_i, \boldsymbol{a}_j) \tag{2.11}$$

wobei s_{ij} - Ähnlichkeit zwischen den beliebigen zwei Binärvektoren.

2.1.3 Dekomposition

Jede Boolesche Funktion kann in andere Boolesche Funktionen zerlegt werden. Die allgemein bekannten und in der Literatur ausreichend beschriebenen Dekompositionsverfahren sind die Dekompositionen in zwei Boolesche Funktionen. Besonders verbreitet sind die Shannon- und Davio-Dekompositionen.

Shannon-Dekomposition: Ist $f(\boldsymbol{x})=f(x_1, x_2,\ldots, x_n)$ eine Boolesche Funktion $\boldsymbol{B}^n \rightarrow \boldsymbol{B}$, so gilt für alle $x_i \in (x_1, x_2,\ldots, x_n)$:

$$\begin{aligned} f(x_i, \boldsymbol{x}_0) &= \overline{x_i} f_{S0}(\boldsymbol{x}_0) \vee x_i f_{S1}(\boldsymbol{x}_0) \\ &= \overline{x_i} f(x_i = 0, \boldsymbol{x}_0) \vee x_i f(x_i = 1, \boldsymbol{x}_0) \end{aligned} \tag{2.12}$$

Die Funktionen $f_{S0}(\boldsymbol{x}_0)$ und $f_{S1}(\boldsymbol{x}_0)$ werden als Cofaktoren bezeichnet und beschreiben die Funktion $f(x_i, \boldsymbol{x}_0)$ an den Stellen x_i=0 und x_i=1.

Davio-Dekomposition: Ist $f(\boldsymbol{x})=f(x_1, x_2,\ldots, x_n)$ eine Boolesche Funktion $\boldsymbol{B}^n \rightarrow \boldsymbol{B}$, so gilt für alle $x_i \in (x_1, x_2,\ldots, x_n)$:

$$\begin{aligned} f(x_i, \boldsymbol{x}_0) &= f_{S0}(\boldsymbol{x}_0) \oplus x_i f_D(\boldsymbol{x}_0) \\ &= f(x_i = 0, \boldsymbol{x}_0) \oplus x_i (f(x_i = 0, \boldsymbol{x}_0) \oplus f(x_i = 1, \boldsymbol{x}_0)) \end{aligned} \tag{2.13}$$

$$\begin{aligned} f(x_i, \boldsymbol{x}_0) &= f_{S1}(\boldsymbol{x}_0) \oplus \overline{x_i} f_D(\boldsymbol{x}_0) \\ &= f(x_i = 1, \boldsymbol{x}_0) \oplus \overline{x_i} (f(x_i = 0, \boldsymbol{x}_0) \oplus f(x_i = 1, \boldsymbol{x}_0)) \end{aligned} \tag{2.14}$$

Es gibt verschiedene Methoden zur Zerlegung Boolescher Funktionen. Bezüglich des Ansatzes der Dekomposition unterscheiden sich die Curtis- und Bi-Dekompositionen signifikant. Eine ausführliche Beschreibung von Dekompositionsmethoden ist in [26], [85]-[87], [89], [108], [126], [130], [137], [150]-[152] zu finden.

2.2 Neuronale Netzwerke

2.2.1 Künstliches Neuron

Ein (**künstliches**) **Neuron** (**KN**) ist ein mathematisches Modell einer Nervenzelle des menschlichen zentralen Nervensystems. Das Neuron kann als ein einfacher Prozessor gesehen werden. Ein verallgemeinertes künstliches Neuron mit vier Eingängen, entsprechenden Gewichten und einem Ausgang wird in Abbildung 2.2 gezeigt.

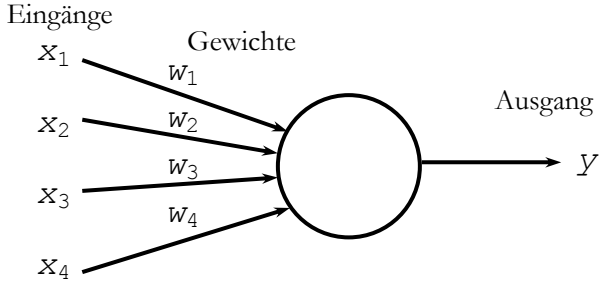


Abbildung 2.2 Ein einfaches Neuron

Für das bessere Verständnis des Aufbaus und der Arbeitsweise von Neuronen wird eine detailliertere Darstellung des Neurons in der Abbildung 2.3 gegeben.

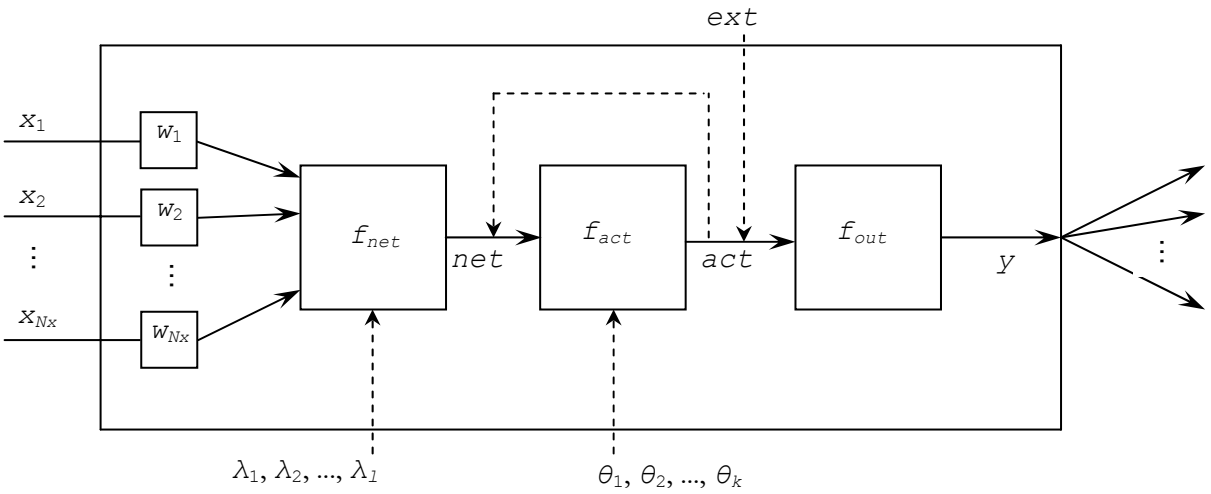


Abbildung 2.3 Aufbau eines Neurons [28]

Ein Neuron besitzt die folgenden Bestandteile.

- **Aktivierungszustand** (activation) $act(t)$. Er gibt den Grad der Aktivierung des Neurons im Moment t an.
- **Aktivierungsfunktion** f_{act}. Sie gibt an, wie sich ein neuer Aktivierungszustand act_t des Neurons aus der alten Aktivierung act_{t-1} und der Netzeingabe net ergibt.
- **Ausgabefunktion** f_{out}. Die Ausgabe des Neurons wird durch eine so genannte Ausgabefunktion aus der Aktivierung des Neurons definiert.

Die Berechnungen eines einzelnen Neurons bestehen aus drei Teilfunktionen: eine Netzeingabefunktion (2.15), eine Aktivierungsfunktion (2.16) und eine Ausgabefunktion (2.17).

$$net = f_{net}(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{\lambda}) = f_{net}(x_1, x_2, ..., x_{Nx}, w_1, w_2, \ldots, w_{Nx}, \lambda_1, \lambda_2, ..., \lambda_l), \quad (2.15)$$

$$act_t = f_{act}(net, \theta_1, \theta_2, ..., \theta_k, act_{t-1}) \quad (2.16)$$

$$y = f_{out}(act_t) \quad (2.17)$$

Die Netzeingabefunktion f_{net} berechnet aus den Eingaben $x_1, x_2, ..., x_{Nx}$ und den Verbindungsgewichten $w_1, w_2, \ldots, w_{Nx}$ die Netzeingabe net. In diese Berechnung können eventuell zusätzliche Parameter $\lambda_1, \lambda_2, ..., \lambda_l$ eingehen. Aus der Netzeingabe net, einer bestimmten Anzahl von Parametern $\theta_1, \theta_2, ..., \theta_k$ und eventuell einer Rückführung der aktuellen Aktivierung des Neurons berechnet die Aktivierungsfunktion f_{act} die neue Aktivierung act_t des Neurons. Schließlich wird aus der Aktivierung act durch die Ausgabefunktion f_{out} die Ausgabe y des Neurons berechnet. Durch die externe Eingabe ext

kann die (Anfangs-) Aktivierung des Neurons festgelegt werden. Die Zahl l der zusätzlichen Argumente der Netzeingabefunktion und die Zahl k der Argumente der Aktivierungsfunktion hängen von der Art dieser Funktionen und dem Aufbau des Neurons ab. Meist hat die Netzeingabefunktion nur 2 Argumente (die Ausgaben der Vorgängerneuronen und die zugehörigen Gewichte). Die Aktivierungsfunktion hat meist auch zwei Argumente: die Netzeingabe und einen Parameter (z.B. für Schwellwertelemente ist dieser Parameter ein Schwellwert). Die Ausgabefunktion hat dagegen nur die Aktivierung als Argument und dient meistens nur dazu, die Ausgabe des Neurons in einen gewünschten Wertebereich zu transformieren (meist durch eine lineare Abbildung) [28].
Normalerweise besteht die Aktivierungsfunktion aus einer nichtlinearen Transformation. Es kann aber auch eine lineare Abhängigkeit als Aktivierungsfunktion verwendet werden. Dann spricht man von einem **linearen** Neuron [112].

2.2.2 Neuronales Netz

Um die Berechnungsmöglichkeiten von einzelnen Neuronen zu erhöhen, schaltet man mehrere Neuronen zu Netzen von Neuronen (Neuronale Netze) zusammen. **Neuronale Netze** (**NN**) werden oft auch als **künstliche Neuronale Netze** (**KNN**) oder **artificial neural networks** (**ANN**) bezeichnet und sind Systeme zur Informationsverarbeitung, die aus einer großen Anzahl einfacher parallel arbeitender Neuronen (Zellen, Einheiten) bestehen. Neuronale Netze sind modular aufgebaute Berechnungsmodelle, deren Funktionsprinzipien von biologischen Nerven-Systemen abgeleitet wurden, und deren herausragende Eigenschaft die Lernfähigkeit ist. In vielen Anwendungen dienen sie der Beschreibung und Berechnung von stetigen oder (partiell) differenzierbaren Funktionen. Man kann ein gegebenes Neuronales Netz auch als eine Datenstruktur auffassen, die mit Hilfe geeigneter (Auswertungs-) Methoden eine Funktion definiert [62], [83] und [166].
Eine wichtige Eigenschaft eines Neuronalen Netzes ist seine Struktur. Einer der besten Wege um die Struktur eines Neuronalen Netzes zu beschreiben, ist die Verwendung von gerichteten Graphen.
Definition 2.5. Ein gerichteter **Graph** ist ein Paar $G = (V, E)$ bestehend aus einer endlichen Menge V von **Knoten** (vertices, nodes) und einer Menge $E \subseteq V \times V$ von **Kanten** (edges), wobei eine Kante $e = (u, v) \in E$ vom Knoten u auf den Knoten v **gerichtet** sei.
Definition 2.6. Ein künstliches **Neuronales Netz** ist ein gerichteter Graph $G = (U, C)$, dessen Knoten $u \in U$ **Neuronen** (units) und dessen Kanten $c \in C$ **Verbindungen** (connections) heißen. Die Menge U der Knoten ist unterteilt in die Menge U_{in} der

Eingabeneuronen, die Menge U_{out} der **Ausgabeneuronen** und die Menge U_{hidden} der **versteckten (verborgenen) Neuronen**. Es gilt

$$U = U_{in} \cup U_{out} \cup U_{hidden},$$

$$U_{in} \neq \emptyset, \quad U_{out} \neq \emptyset, \quad U_{hidden} \cap (U_{in} \cup U_{out}) \neq \emptyset \tag{2.18}$$

und jeder Verbindung $(u, v) \in C$ ist ein Gewicht w_{uv} zugeordnet.
Eine Visualisierung der allgemeinen Struktur des Neuronalen Netzes ist in der Abbildung 2.4 dargestellt.

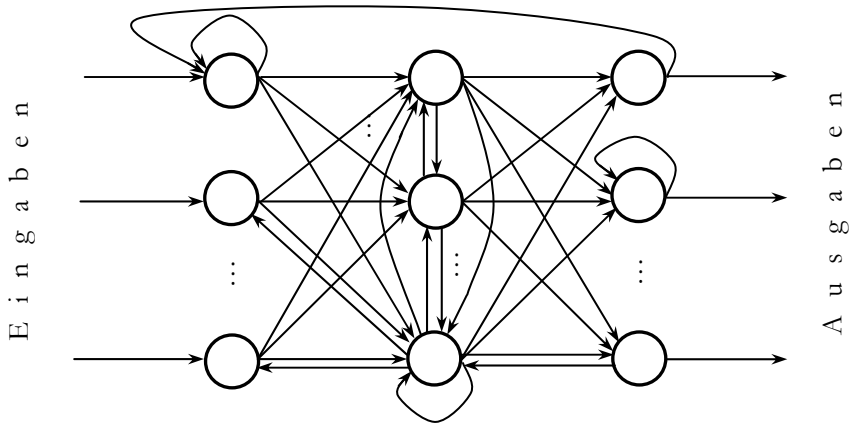


Abbildung 2.4 Allgemeine Struktur des neuronalen Netzes

Außer normalen **vorwärts** und **rückwärts gerichteten** Verbindungen zwischen Neuronen verschiedener Nachbarschichten sind in der Abbildung 2.4 auch die **lateralen** Verbindungen zwischen Neuronen einer Schicht und **rückgekoppelte** Verbindungen eines Neurons enthalten.

2.2.3 Klassen von Neuronalen Netzen

Seit den ersten Versuchen von Warren McCullock und Walter Pitts ein künstliches Neuronales Netz zu schaffen und es bei Bearbeitung der Information zu verwenden, wurde eine große Menge von Neuroarchitekturen und Paradigmen für vielfältige Ziele entwickelt.
Neuronale Netze unterscheiden sich nach verschiedenen Kriterien [112] und [125], z.B.:

nach der Lernmethode: überwacht, bestärkend, unüberwacht,

nach dem Lerntyp: Fehlerkorrektur, Hebbsches Lernen, konkurrierend etc.

nach der Art der Bestimmung der Musterklassen:

fest vorgegebene Anzahl von Mustern mit typischen Merkmalen,

Musterbestimmung mittels Selbstorganisation,

nach der Art der Berechnung: deterministisch, wahrscheinlichkeitstheoretisch,

nach der Art der Eingangs- und Ausgangssignale: digital, analog,

nach der Eingangs-/Ausgangsrelation:

mit Musterordnung, als Assoziativspeicher etc.

nach dem Architekturtyp:

Feed Forward (FFNN, Neuronale Netzwerke ohne Rückkopplungen),

Feed Back (FBNN, rückgekoppelte Neuronale Netzwerke),

nach der Zahl von Schichten: ein– und mehrschichtige Netze,

nach der Anwendung:

Klassifizierung, Mustererkennung, Allgemeine Abbildung, Vorhersage, Optimierung etc.

nach der Realisierungsart: Software-, Hardware- und Hybride-Netze.

Eine bedeutende Rolle spielt die Klassifizierung von Neuronalen Netzen nach ihrem Trainingsalgorithmus. Dabei unterscheidet man:

Netze mit **iterativen** Trainingsalgorithmen (**ITA**),

Netze mit **sequentiellen** (oder konstruktiven) Trainingsalgorithmen (**STA**).

Ein typisches und meist verwendetes Beispiel des ITA ist das Backpropagation Verfahren (Fehlerrückführungs-Methode). STA benutzen nichtiterative Berechnungen und basieren oft auf einer graphischen Darstellung des zu modellierenden Objektes. Dabei sind STA aussichtsvoller als ITA, weil sie eine schnellere Konvergenz garantieren und kürzere Trainingszeit brauchen. In den letzten zwei Jahrzehnten wurden STA sehr populär, weil sie viele Vorteile im Vergleich zu ITA haben. Im nächsten Abschnitt wird eine allgemeine Strategie von STA beschrieben.

Die angegebene Klassifizierung ist natürlich nicht vollständig, da noch viele andere Kriterien definiert werden können. Die folgende Definition erklärt eine spezielle Art von Neuronalen Netzen, die als Boolesche (manchmal als binäre) Neuronale Netze bezeichnet werden.

Definition 2.7. Als ein **Boolesches Neuronales Netz** definiert man ein Neuronales Netz, das für die Bearbeitung Boolescher Daten vorgesehen ist. Die Ein- sowie Ausgabesignale des Netzes sind Boolesche Werte. Dabei gibt es keine Beschränkung der Art der Gewichte, der Transferfunktionen der Neuronen, der Arbeitsweise des Netzes oder überhaupt der Zugehörigkeit des Netzes zu einer Netzwerkklasse nach anderen Kriterien.

2.2.4 Sequentielle Trainingsalgorithmen

Eine ausführliche Beschreibung von sequentiellen Algorithmen (STA) ist in vielen Quellen zu finden [33], [92], [97], [113] und [165]. Einer der ersten Sätze für einen sequentiellen Trainingsalgorithmus für Boolesche Neuronale Netze wurde von Marchand, Golea

und Rujan vorgeschlagen [93]. Seine Implementierung hatte spezifische praktische Probleme, wie eine exponentielle Erhöhung der Gewichte in der Ausgangsschicht und eine Verlängerung der für die Synthese des ganzen Netzes erforderlichen Zeit. Auch entstanden Schwierigkeiten, wenn die Ausgangsdimension größer als Eins war, da keine Erweiterung für das Standardverfahren angegeben wurde. Auf der anderen Seite garantierte ihr Algorithmus eine kleine Struktur und eine annehmbare Trainingszeit des Netzes.
Später entstanden mehrere Arbeiten über sequentielle Trainingsalgorithmen, die die vielen Vorteile der konstruktiven Trainingsmethoden von Neuronalen Netzen im Vergleich zu iterativen Trainingsmethoden bewiesen haben.
Abbildung 2.5 zeigt ein Struktogramm eines verallgemeinerten STA von BNN für die Abbildung einer Booleschen Funktion.

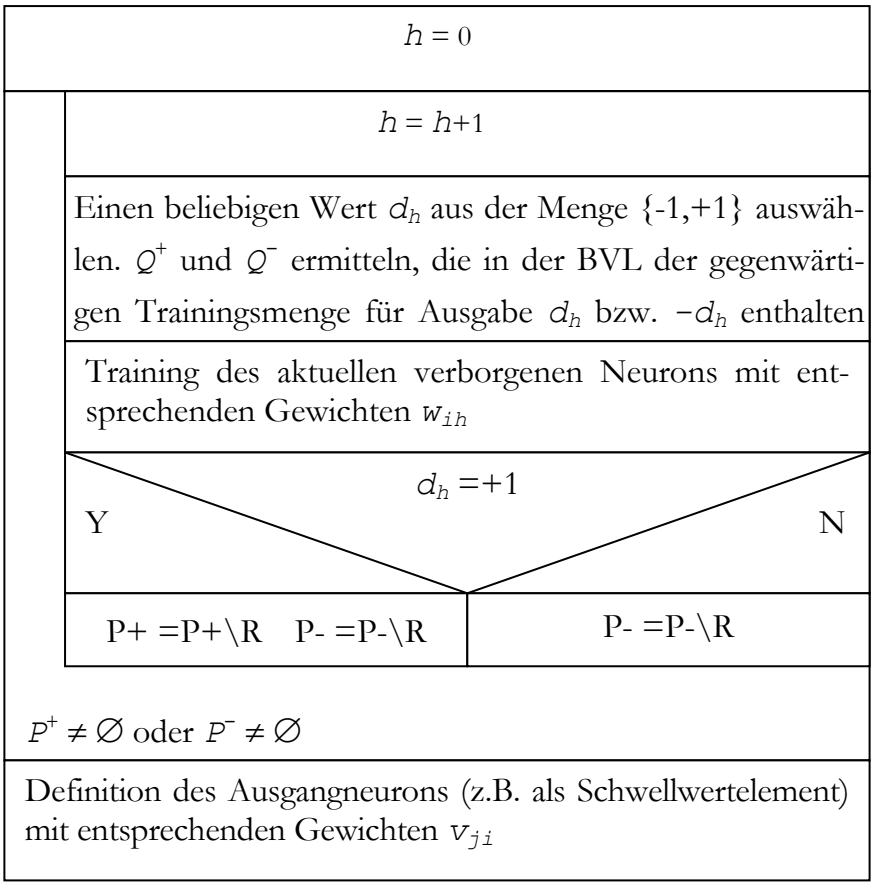


Abbildung 2.5 Struktogramm des verallgemeinerten STA von BNN

Dabei werden folgende Bezeichnungen verwendet:

h – Anzahl von verborgenen Neuronen,

$[w_{ij}]$ – *Gewichtsmatrix* für Verbindungen zwischen Eingangs- und verborgener Schicht,

$[v_{jk}]$ – *Gewichtsmatrix* für Verbindungen zwischen verborgener und Ausgangsschicht,

P^+, P^- - BVL aus der Lernmenge mit positiven "+" beziehungsweise negativen "-" Funktionswerten,

$Q^+ \subseteq P^+$, $Q^- \subseteq P^-$ - BVL zum Training des aktuellen verborgenen Neurons mit positiven "+" beziehungsweise negativen "-" Funktionswerten,

$R \subseteq Q^+$ - *Untermenge* aus BV, für die das verborgene Neuron mit Nummer h eine Ausgabe +1 zur Verfügung stellt.

Definition 2.8. Das Training eines einzelnen verborgenen Neurons umfasst die Erzeugung seiner Transferfunktion und der Gewichten w_{ih} zwischen Eingangsneuronen und aktuellem Neuron. Dabei soll das trainierte Neuron zumindest für ein BV aus Q^+ eine Ausgabe +1 und für alle BV aus Q^- eine Ausgabe -1 (bzw. 0) liefern.

Eine ausführliche Beschreibung von STA zur Entwicklung von BNN ist in [114] zu finden.

Die Mehrheit von existierenden konstruktiven Trainingsalgorithmen für Neuronale Netzen kann man gemäß der Vorgehensweise beim Aufbau von Netzen in Vorwärts- und Rückwärtsmethoden klassifizieren. Die Begriffe „vorwärts" und „rückwärts" sind nicht identisch den oben benutzten Begriffen für die Verbindungen in Netzen und für die Klasse von FFNN und FBNN. Die Vorwärtsmethoden von STA fügen neue Neuronen an die vorhandenen Ausgänge des Netzes an. Umgekehrt fügen die rückwärts gerichteten Techniken neue Neuronen zwischen Eingang- und verbogener Schichte ein. Der „Tiling" Algorithmus [103] mit seiner einfachsten Variante dem „Tower" Algorithmus [57] und [115] oder die Entscheidungsbaum-Algorithmen [60] und [141] sind typische Beispiele von konstruktiven Vorwärtsalgorithmen. Der Aufbau des Netzes ist dagegen in der „Upstart"-Methode [53] rückwärts gerichtet.

2.2.5 Hardwarerealisierungen von Neuronalen Netzen

Abhängig von der Art der Signale werden mit Hardware realisierte Neuronale Netze (HwNN) in 3 Kategorien eingeteilt: digital, analog, hybrid [158]. Die in dieser Arbeit im Mittelpunkt stehenden Booleschen Neuronalen Netze sind besonders für die digitalen Hardware-Realisierungen geeignet, weil diese Netze mit Booleschen Daten arbeiten. Es gibt viele Technologien zur Realisierung von Neuronalen Netzen in digitale Hardware. Bekannt sind die Beispiele von HwNN in FPGA, VLSI, WSI [38], [59], [63], [64], [136] und [180]. Hardware-Realisierungen von Neuronalen Netzen in FPGA sind besonders verbreitet, weil diese viele Vorteile im Vergleich zu den anderen Basistechnologien besitzen [111]. Die erste erfolgreiche FPGA-Realisierung von künstlichen Neuronalen Netzen wurde 1992 veröffentlicht [34].

Man unterscheidet HwNN auch danach, ob das Training *On-Chip* oder *Without-Chip* ausgeführt wird. In weiteren Kapiteln werden Hardware-Realisierungen von Booleschen Neuronalen Netzen mit dem Training außerhalb des Chips betrachtet.

2.3 RTR-Systeme

2.3.1 Rekonfigurierbare Systeme

Die rekonfigurierbaren Rechensysteme basieren auf einer adaptierbaren, rekonfigurierbaren Hardware. Abhängig von der rekonfigurierbaren Hardware-Plattform kann die rekonfigurierbare Logik nacheinander für verschiedene Aufgaben verwendet werden. Die Hauptbesonderheit der rekonfigurierbaren Hardware ist die Fähigkeit, das benötigte Verhalten der Hardware im richtigen Zeitintervall zur Verfügung zu stellen. Außerdem sind rekonfigurierbare Systeme wegen ihrer vielen gleichzeitig arbeitenden Elemente in der Lage, modernste Computer für viele Probleme zu überbieten.
Nicht jede rekonfigurierbare Hardware ist während der Laufzeit rekonfigurierbar. Zur Konfigurationsspeicherung und Neuladung der Konfiguration beim Rücksetzen des rekonfigurierbaren Geräts, bekannt als statische Rekonfiguration, wird ein nichtflüchtiger RAM verwendet. Ein während der Laufzeit rekonfigurierbares System (Run-Time Reconfigurable System RTR-System) kann sein Verhalten durch das dynamische Überschreiben einer Konfiguration während der Laufzeit ändern. Dabei braucht man das System nicht zurückzusetzen.
Diese Eigenschaften besitzen Field Programmable Gate Arrays (FPGA) [30], die Xilinx Mitte der 1980er Jahre eingeführt hat. RTR-Computersysteme schließen einen klassischen Mikroprozessor und ein programmierbares Logikteil, wie FPGA zusammen [15] und [17].

2.3.2 Field Programmable Gate Arrays - FPGA

Es gibt viele Arten rekonfigurierbarer Logik, aber FPGA dominieren auf diesem Gebiet. FPGA wurden als eine Kombination von zwei Technologien entwickelt: Programmable Logic Array (PLA) und Mask-Programmable Gate Arrays (MPGA). Wie PLA sind FPGA elektrisch vollprogrammierbar. Die höheren Kosten zu einmal programmierbaren Technik (Non-Recurring Engineering (NRE) amortisieren sich schnell. Wie MPGA können sie sehr komplizierte Berechnungen auf einem einzelnen Chip mit den Millionen von CLB durchführen [39]. FPGA ermöglichen nicht nur die Programmierung der Logikzellen sondern auch die Programmierung der Verbindungen zwischen ihnen.

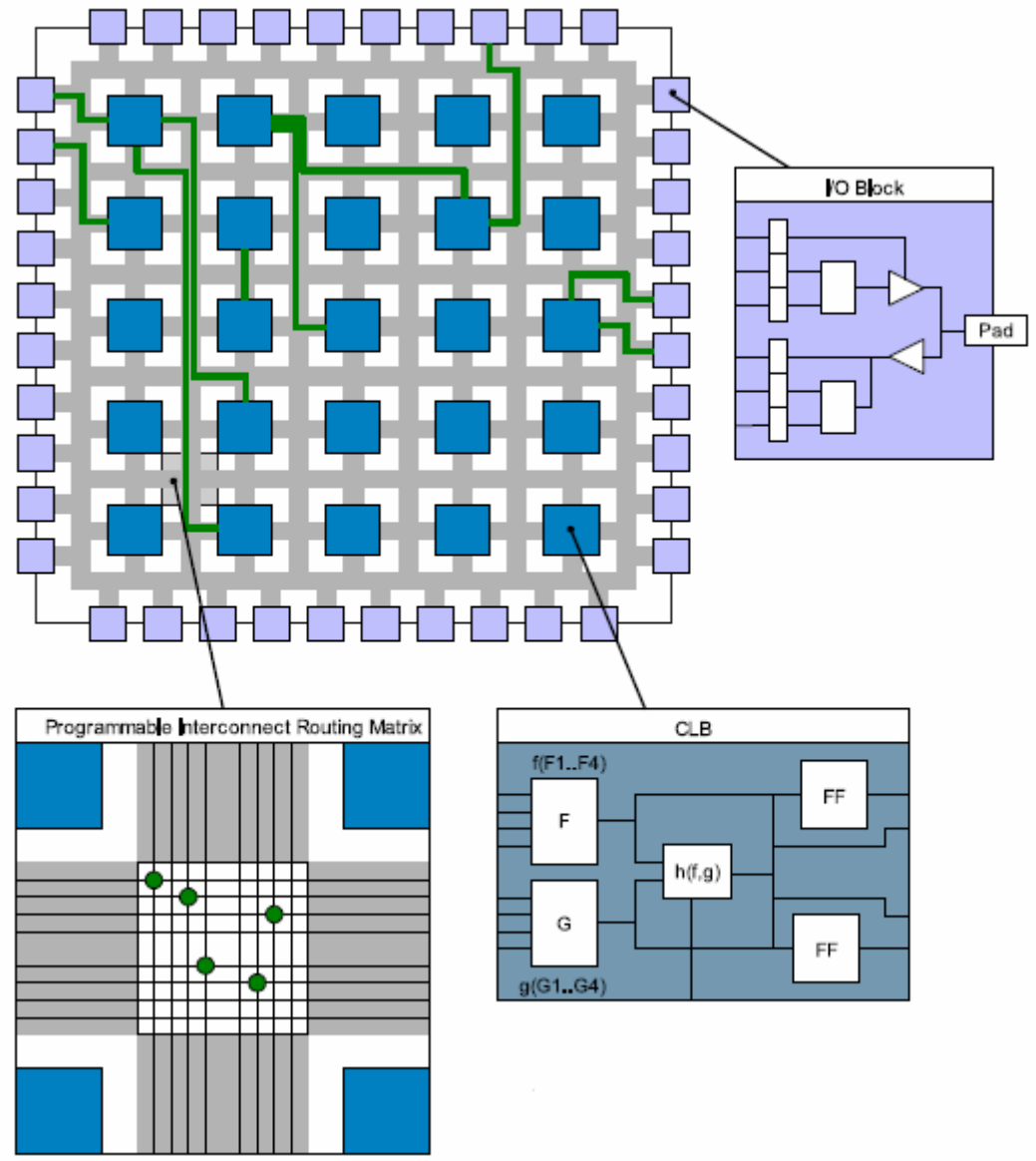


Abbildung 2.6 Allgemeine FPGA-Struktur [110] und [131]

Eine typische innere Struktur eines FPGA [22] und [69] wird in Abbildung 2.6 gezeigt. FPGA bestehen aus drei Schlüsselteilen:

- konfigurierbare Logikblöcke (configurable logic blocks - CLB),
- Eingabe/Ausgabe-Blöcke (I/O-blocks, IOB),
- Verbindungsnetz (interconnect network).

Xilinx bezeichnet die Logikblöcke als konfigurierbare Logikblöcke (CLB) und die FPGA selbst als Logic Cell Arrays (LCA). Ein typischer CLB enthält zwei identische Logikblöcke (Slice) mit eigenen Look-up Tabellen (LUT), D-Flip-Flops (DFF) und einer schnellen Übertragungslogik (siehe Abbildung 2.7).

Ein CLB enthält meist zwei oder vier LUT. Die Look-up table ist das grundlegende Rechenelement im FPGA. Eine LUT enthält 4-Eingänge. Flip Flops (FF) dienen der Speicherung von Ausgabewerten der LUT oder Außenquellsignale. D-Flip-Flops erlauben

dem Benutzer, effiziente synchrone Designs zu erzeugen und können für Parallelverarbeitung, Register, Zustandsmaschinen oder jede andere Situation verwendet werden, für die eine Taktgebung erforderlich ist. Multiplexer (MUX) werden in CLB verwendet, um den LUT-Ausgang oder ein anderes CLB-Eingangssignal mit dem FF-Eingang oder einem CLB-Ausgang zu verbinden.

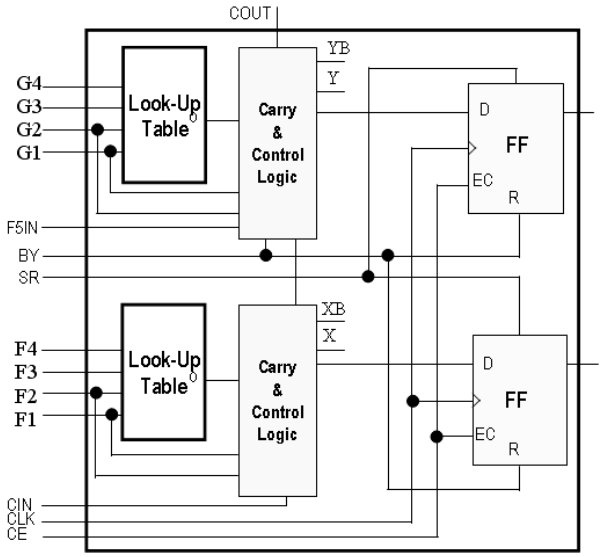


Abbildung 2.7 Allgemeine CLB-Struktur [80]

Eine LUT mit n Eingängen ermöglicht die Speicherung der 2^n Werte einer Booleschen Funktion mit n Eingängen. Durch eine LUT kann jede für n Variablen definierte Boolesche Funktion realisiert werden. Die Werte der Funktion für jede Kombination der n Variablen werden berechnet und in der LUT gespeichert. Die Eingangsvariablen werden verwendet, um in der LUT die Position auszuwählen, an der der richtige Wert gespeichert ist. Das Ergebnis erscheint am LUT-Ausgang. Die Anzahl vom Design benötigten CLBs bestimmt die Größe des erforderlichen FPGA-Bausteins [20] und [39].

Die I/O-Blöcke am Rand des FPGAs verbinden das FPGA mit der Außenwelt. Die Anzahl von IOB hängt von der Größe des FPGA-Baustein ab und stimmt mit der Anzahl der I/O-Pins überein. Das programmierbare Verbindungsnetz für FPGA besteht aus horizontalen und vertikalen Verbindungen, die die Ein- und Ausgänge von CLB mit den IOB in verschiedenen Zeilen und Spalten verbinden. Es kann sehr flexible genutzt werden. Schaltungsmatrizen erlauben eine Programmierung der Verbindungen und unterstützen programmierbare Multiplexers.

Die Konfiguration für die LUT, die Multiplexers und das programmierbare Verbindungsnetz werden in Anti-Fuse, EEPROM/Flash oder statische RAM (SRAM) gespeichert. Für RTR-Systeme wird SRAM verwendet.

2.3.3 Hardware/Software-CoDesign

Hardware/Software-CoDesign vereinigt die Phasen des Modellierens, der Analyse, der Synthese und der Simulation von Systemen, die aus zusammenwirkenden Hard- und Software-Modellen bestehen. Damit versucht man, die bei der Entwicklung von Hardware/Software-Systemen bestehenden Probleme zu lösen. Verallgemeinert besteht CoDesign aus drei Teilen: CoSpezifikation, CoSynthese und CoSimulation. Ausführliches zum Hardware/Software-CoDesign ist in [65], [68] und [104] zu finden.
Mit der Entwicklung von Objekt-Orientierter Programmierung und Modellierung wurden Technologien und Methoden zur Analyse und zum Design entwickelt. Dabei entstand eine spezielle Sprache, die Unified Modeling Language (UML) [27], [117] und [135], zur Modellierung von Systemen. Verschiedenen Vorgehensmethoden wie z.B. Unified Process oder Rational Unified Process (RUP) [72] setzen die UML als Modellierungssprache voraus. Die UML ist eine graphische Sprache für die Visualisierung, die Spezifikation, die Entwicklung und Dokumentation von Artefakten eines Softwaresystems. Die aktuelle Version 2.0 der UML-Spezifikation enthält viele Verbesserungen der Semantik und Diagramm-Typen. Obwohl die UML insbesondere für den Entwurf objektorientierter Software entwickelt wurde, kann man die UML auch für den Hardware/Software-CoDesign verwenden. Trotz der verschiedenen Fachgebiete und der verschiedene Begriffe sind die Bezeichnungen aus der UML universell nutzbar.
Durch die UML werden 13 Diagramme zur Modellierung verschiedener Aspekte eines Systems unterstützt. Diese umfassen zur Modellierung der Systemsstruktur:

- Package Diagram,
- Class Diagram,
- Object Diagram,
- Composite Structure Diagram,
- Component Diagram,
- Deployment Diagram,

und zur Modellierung des Systemsverhaltens:

- Use Case Diagram,
- Communication Diagram,
- Sequence Diagram,
- Timing Diagram,
- Interaction Overview Diagram,
- State chart Diagram,
- Activity Diagram.

Einige diese Diagramme werden in dieser Arbeit verwendet.

2.3.4 MOdel Compiler for reConfigurable Architecture

Aus der Vielzahl von Entwicklungstools für das Hardware/Software-CoDesign wird ein Modell-Compiler – MOCCA (MOdel Compiler for reConfigurable Architecture - MOCCA) in dieser Arbeit benutzt. MOCCA wird laufend verbessert und erweitert. Hauptsächlich ist dieser Compiler für Anwendungen mit rekonfigurierbaren Architekturen vorgesehen, aber kann auch bei anderen Aufgaben eingesetzt werden. MOCCA erschließt eine neue Entwicklungsmethode des UML-basierenden Hardware/Software-CoDesigns, die auf dem Konzept der Modell-gesteuerten Architektur (Model Driven Architecture Approach - MDA) aufbaut. Es wird die objektorientierte Methodik für alle Phasen des Entwicklungszyklus verwendet. Sie reicht von der Spezifikation bis zur Realisierung, zur Installierung und zum Tests. Durch die Verwendung der UML wird die Visualisierung jedes Aspekts während des Designprozesses unterstützt [15]-[17].
Für die Beschreibung der Modelle benutzt MOCCA ein speziell entwickelte Aktionssprache (MOCCA Action Language (MAL)), die als eine Erweiterung zur UML - Aktionssemantik in die UML-Spezifikation 1.5 eingeführt wurde. Die Syntax und Semantik der MAL orientieren sich an Java mit der zusätzlichen Unterstützung der UML-Konzepte bezüglich aktiver Klassen, Vereinigungen, des Zustands und der Zeit.
Der gesamte Prozess der Entwicklung von Anwendungen für FPGA, der durch MOCCA durchgeführt und gesteuert wird, basiert auf Plattformmodellen (siehe Abbildung 2.8). Dieser Prozess umfasst die automatische Partitionierung, die Bewertung und Realisierung des Systems in Hardware/Software-Modulen. Für Design, Entwicklung und Realisierung werden dabei verschiedene Modelle verwendet.

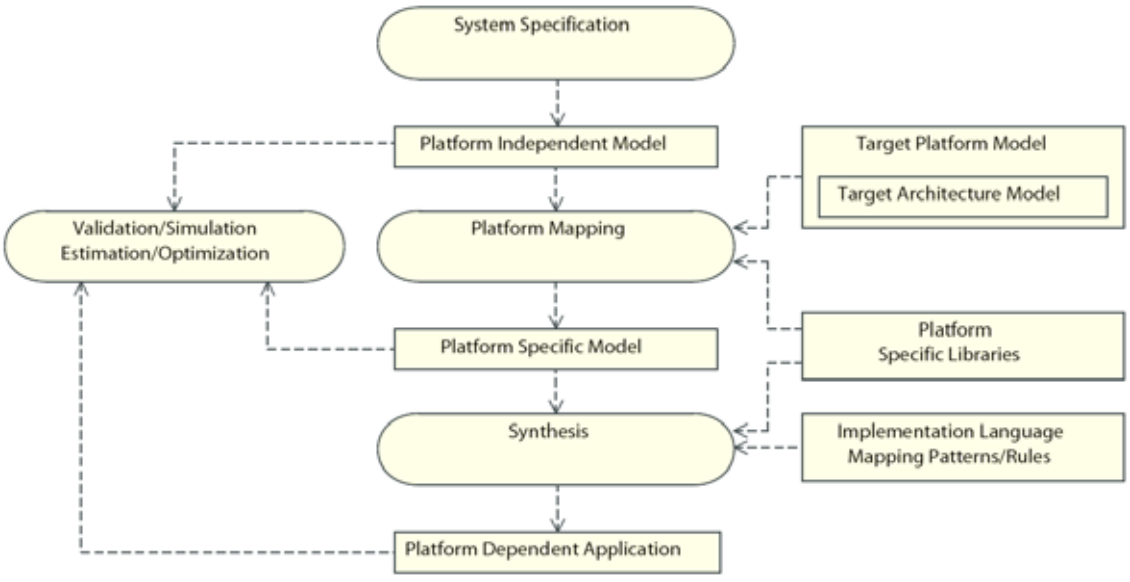


Abbildung 2.8 UML- basierendes CoDesign – Aktivitäten und Artefakten [146]

Die Abbildung 2.8 zeigt die grundlegenden Aktivitäten und Artefakte der Entwicklungsmethode. Die dargestellte Methode vereinigt die grundsätzliche Vorgehensweise des Hardware/Software-CoDesigns mit dem MDA-Konzept. Das System wird durch ein Plattform-unabhängiges Modell (PIM - platform independent model) vorgegeben. Das Ziel-Plattformmodell (TPM - target platform model) definiert die Dienstleistungen, die durch das Zielsystems bereitgestellt werden. Von MOCCA wird ein PIM in ein Plattform-spezifisches Modell (PSM - platform specific model) umgewandelt. Während der Synthese wird aus dem PSM eine Plattform-abhängige Anwendung (PDA - platform dependent application) erzeugt, die auf der Ziel-Plattform ausgeführt werden kann. Solch eine Methodik erleichtert die Prüfung, die Übertragbarkeit, die Anpassungsfähigkeit, und die Widerverwendung des Systems.

2.4 Darstellung von Algorithmen

In dieser Arbeit werden Algorithmen durch einen Pseudo-Code beschrieben. In der angegebenen Algorithmusvorlage wird ein Beispiel gezeigt.

Algorithmus 2.1 Vektor-Gewicht – Berechnung des Gewichts eines Vektors

Eingabe:
$\boldsymbol{V}$ - Vektor
Ausgabe:
g – Gewicht des Vektors $\boldsymbol{V}$

```
VEKTOR-GEWICHT(V)
 1  g ← 0
 2  for (i ← 0,…, LENGHT(V)-1)
 3      g ← g + V[i]
 4  return g
```

Die Vorlage beginnt mit dem Name des Algorithmus und einer kurzen Beschreibung. Danach werden Ein- und Ausgabeparameter definiert. Der Pseudo-Code hält sich an die folgende Regel:
Die Anweisung $a \leftarrow b$ weist den Wert von b der Variablen a zu.
Vektoren bzw. Matrizen werden durch eckige Klammern $\boldsymbol{V}[i]$ bzw. $\boldsymbol{A}[i,j]$ indiziert. Der Anfangsindex von Vektoren ist 0. Die Länge eines Vektors liefert die Funktion LENGHT($\boldsymbol{V}$).

Strukturen von verschiedenen Elementen werden in geschweiften Klammern deklariert, z.B. `net={k_set, w_set, L }`. Ein Zugriff auf einzelne Elemente der Struktur wird durch den Strukturnamen und die Elementsbezeichnung realisiert, z.B. `net.k_set`.
Der Bereich einer Steueranweisung wird durch eingerückte Zeilen sichtbar. Im obengenannten Beispiel besteht der Körper der **for**-Schleife nur aus der Zeile 3.
Die Anweisung **if** erfordert eine Bedingung nach der ein **then**-Block folgt. Ein **else**-Block ist optional.
Die Schleifen-Anweisungen **for** und **while** überprüfen ihre Bedingungen vor der ersten Ausführung des Schleifen-Körpers, d.h. der Körper der Schleife kann auch gar nicht ausgeführt werden.
Die **do…while**-Anweisung führt ihren Körper mindestens einmal aus. Die Bedingung wird am Ende jedes Durchlaufs geprüft.
Die Schleifen-Anweisungen **while…** und **do…while** werden wiederholt, solange die Bedingung nach **while** wahr ist.
Ergebniswerte werden durch die **return**-Anweisung zurückgegeben und der Algorithmus wird sofort angehalten.
Bei der Verwendung weiterer spezieller Anweisungen wird eine entsprechende Erklärung angegeben.

Kapitel 3

Klassische Booleschen Neuronalen Netze

3.1 Einzelnes Neuron

3.1.1 Problem der linearen Separierbarkeit (EXOR-Funktion)

Es gibt viele Verfahrensweisen, wie man die Technik Neuronaler Netze verwendet, um eine einzelne Boolesche Funktion oder eine ganze Funktionsmenge durch ein Neuronales Netz darzustellen [76], [83], [106], [109], [132] und [154]. Da das einfachste Boolesche Neuronale Netz aus einem einzigen Neuron besteht, wird in diesem Abschnitt die Fähigkeit eines einzelnen Booleschen Neurons zur Darstellung von Booleschen Funktionen analysiert. Weiter befasst sich dieses Kapitel mit klassischen Booleschen Neuronalen Netzen. Zum Verständnis der bei der Darstellung Boolescher Funktionen durch Neuronale Netze auftretenden Probleme werden wesentliche bekannte Ergebnisse Boolescher Neuronaler Netze analysiert.

Die Untersuchungen haben ergeben, dass nicht jede, sogar elementare, Boolesche Funktion (z.B. EXOR) durch ein Boolesches Neuron problemlos abgebildet werden kann. Eines der bahnbrechenden Ergebnisse in der Geschichte der Neuronalen Netze war der Beweis von Minsky und Papert [105] über die Unmöglichkeit einer Darstellung aller beliebigen funktionellen Abhängigkeiten durch das Perzeptron von Rosenblatt (1962) [132], das als erstes Neuronales Netz entwickelt wurde. Sie bewiesen diese Eigenschaft von Neuronalen Netzen für die Booleschen Funktion exklusiv OR (EXOR) [70], [106], [159] und [166].

Man betrachtet die in der Tabelle 2.2 beschriebene Wertetabelle von elementaren Booleschen Funktionen OR, AND und EXOR. Die Neuronen im Perzeptron von Rosenblatt [132] werden Schwellwertelemente genannt, weil eine einfache Schwellwertfunktion mit einem Schwellwert θ als Aktivierungsfunktion verwendet wird. Die Ausgabe eines Neurons wird durch (3.1) berechnet.

$$y = \begin{cases} 1 & falls \quad \sum_i w_i x_i \geq \theta \\ 0 & sonst \end{cases}, \tag{3.1}$$

Ein entsprechendes Perzeptron für die Darstellung dieser drei Booleschen Funktionen wird in Abbildung 3.1 gezeigt.

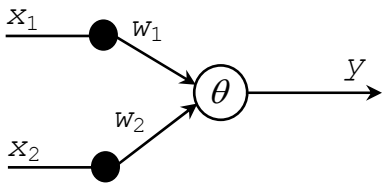


Abbildung 3.1 Perzeptron für eine Boolesche Funktion mit 2 Eingänge

Die Funktionen OR und AND können durch ein Neuron dargestellt werden. Eine einfache Erklärung dafür wird in Abbildung 3.2 gezeigt. Im Fall eines Neurons mit zwei Eingängen wird die Aktivierungsfunktion (3.1) zur Geradengleichung in der Ebene $x_1 - x_2$ umgewandelt:

$$x_1 w_1 + x_2 w_2 = \theta. \tag{3.2}$$

Diese Gerade trennt alle schwarzen Punkte von allen weißen Punkten in Abbildung 3.2, wobei schwarze Punkte die Einswerte der Funktion abbilden und die weißen Punkte entsprechend die Nullwerte.

Definition 3.1. Funktionen, die mittels des einfachen Perzeptrons von Rosenblatt (Schwellwertelement) berechnet werden können, nennt man **linear separierbar**.

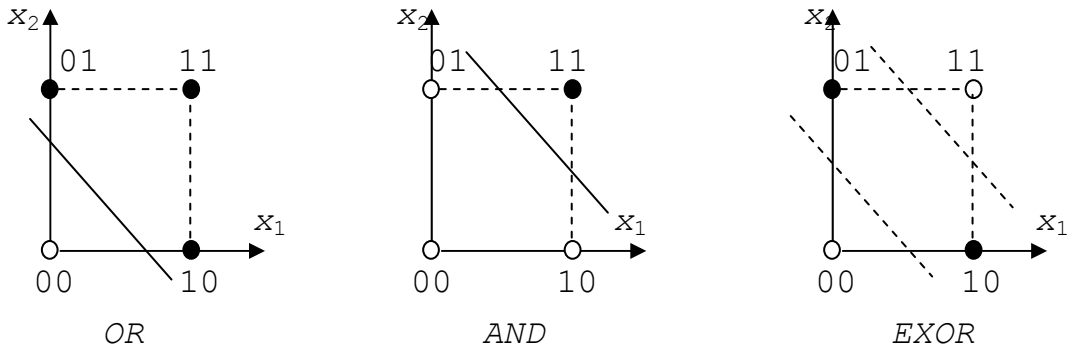


Abbildung 3.2 Lineare Separierbarkeit am Beispiel der OR-, AND- und EXOR-Funktionen

Wenn es um mehr als zwei Boolesche Eingangsvariablen geht, trennt eine Ebene bzw. Hyperebene die beiden Gebiete der jeweils gleichartigen Punkte im Würfel bzw. im Hyperwürfel untereinander.

Für die EXOR-Funktion

$$f(x_1, x_2) = x_1 \oplus x_2 = x_1\overline{x_2} \vee \overline{x_1}x_2 \tag{3.3}$$

bestehen keine Trenngerade, Trennebene oder Hyperebene (siehe Abbildung 3.2). Gerade an diesem Beispiel bewiesen Minsky und Papert, dass die Darstellung einiger funktioneller Abhängigkeiten durch ein Neuron unmöglich ist [106]. Dieses Problem wurde als **Problem der linearen Separierbarkeit** bezeichnet.

Tabelle 3.1 Linear separierbare Boolesche Funktionen [166], [169] und [179]

Variablenanzahl	Boolesche Funktionen	linear separierbare Funktionen
1	4	4
2	16	14
3	256	104
4	65536	1882
5	$4{,}3 \times 10^9$	94572
6	$1{,}8 \times 10^{19}$	15 028 134

In Tabelle 3.1. ist zu sehen, dass der Anteil von linear separierbaren Booleschen Funktionen mit steigender Variablenanzahl deutlich sinkt. Deshalb ist ein einschichtiges Perzeptron bei der Darstellung Boolescher Funktionen sehr beschränkt.

3.1.2 Lösungen des EXOR-Problems

Netze anstatt Neuronen.

Es ist bekannt, dass der Beweis von Minsky und Papert [106] nur für das einfachste Perzeptron von Rosenblatt gültig ist, das aus einem einzelnen Neuron besteht. Wenn man mehrere Schwellwertelemente zusammenschaltet, d.h. von Schwellwertelementen zu Netzen übergeht, kann man die Ausdrucksmächtigkeit von Schwellwertelementen deutlich erhöhen [28], [71], [124], [134], [167], [168]-[172]. Darin besteht die einfachste Lösung des EXOR-Problems, also des Problems linearer Separierbarkeit. Der Kern dieser Methode besteht in einer Zerlegung von nichtmonotonen Booleschen Funktionen in Superposition von monotonen Booleschen Funktionen. Ein Beispiel einer solchen Zerlegung können die disjunktiven oder konjunktiven Formen von Booleschen Funktionen sein, die nur monotone Boolesche Operationen einschließen. Jede monotone Operation bzw. Funktion wird durch ein Neuron dargestellt.

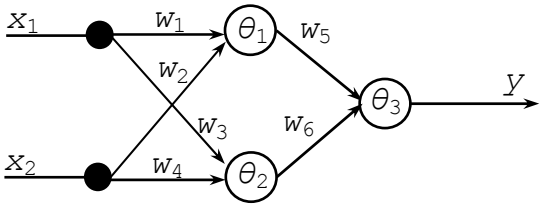


Abbildung 3.3 Netz für die Berechnung der EXOR-Funktion

Diese einfache Methode kann für beliebige nichtmonotone Boolesche Funktionen verwendet werden. Jedoch erfordert sie die Nutzung von Neuronen mit einer großen Anzahl von Eingängen oder führt auch zu komplizierten mehrschichtigen Strukturen von Netzen. Um eine EXOR-Funktion zu realisieren, braucht man drei Neuronen (siehe Abbildung 3.3).

Neuronen mit **funktionell verbundenen Eingängen.** Bei einer anderen bekannten Lösungsmethode des EXOR-Problems verwendet man in einem Neuron ein zusätzliches Eingangssignal, das aus den ursprünglichen Eingangssignalen gebildet wird. Diese Methode wurde von Mkrttschjan [109] vorgeschlagen und benötigt ein zusätzliches OR-Gatter, das das zusätzliche Eingangssignal a_3 liefert.

$$a_3 = x_1 \vee x_2 \tag{3.4}$$

Die Hauptidee ist die Transformation der auf zwei Variablen definierten Booleschen Funktion in den dreidimensionalen Booleschen Raum. Dabei wird die Gleichung (3.2) in (3.5) umgewandelt.

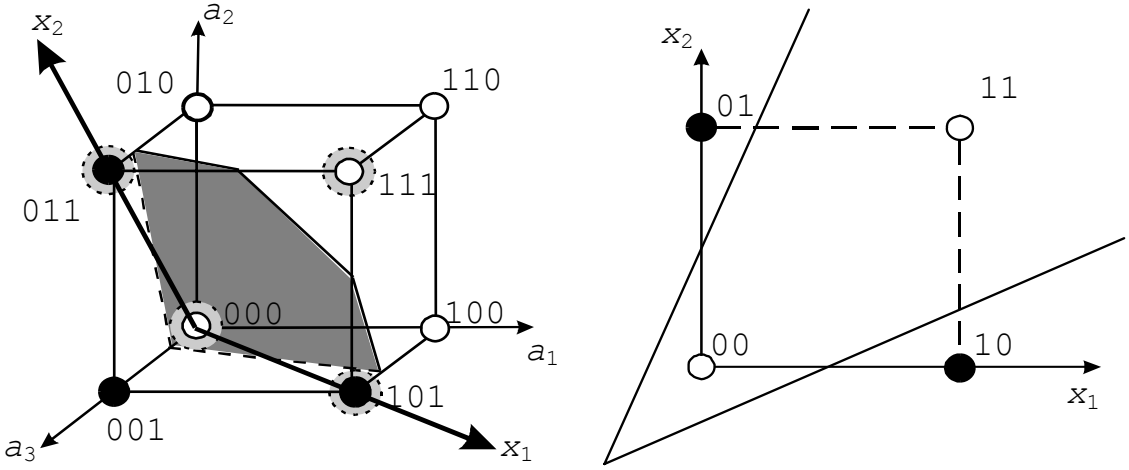
$$w_1\, a_1 + w_2\, a_2 + w_3\, a_3 = \theta \tag{3.5}$$

wobei $a_1 = x_1$, $a_2 = x_2$, $a_3 = x_1 \vee x_2$ und die Werte w_1 =-1, w_2 =-1, w_3 = 2, θ = 0,5 existieren, bei denen die EXOR-Funktion erfolgreich modelliert werden kann.
Unter Verwendung (3.4) wird (3.3) in die Boolesche Funktion (3.6) umgewandelt.

$$f(a_1, a_2, a_3) = a_1\overline{a_2}a_3 \vee \overline{a_1}a_2a_3 \vee \overline{a_1}\,\overline{a_2}a_3 \tag{3.6}$$

In der Abbildung 3.4 sieht man, dass die schattierte Ebene einen Schnitt des Würfels mit der Trennebene (3.5) bildet. Die mit einem punktierten Kreis gezeigten Punkte stellen vier Punkte aus dem zweidimensionalen Raum dar. Die Achse x_1 ist in der Ebene a_1a_3 modelliert worden und die Achse x_2 entsprechend in der Ebene a_2a_3. Die Schnittlinie der Trennebene mit den Ebenen a_1a_3 und a_2a_3 ist eine gebrochene Linie. Eine Projektion dieser Linie in den zweidimensionalen Eingangsraum ist auch eine gebrochene Linie.

Diese gebrochene Linie trennt die Punkte, die den Nullwerten entsprechen, von den Punkten, die den Einswerten entsprechen.



a) b)

Abbildung 3.4 Darstellung der EXOR-Funktion:
a) Trennebene, b) Schnittprojektion in 2-dimensionalem Raum

Diese Methode basiert auf der Einführung zusätzlicher Eingänge, die auch **funktionell verbundene** Eingänge genannt werden. Signale dieser zusätzlichen Eingänge lassen sich als Funktionswerte von existierenden Eingangssignalen bilden. Neuronale Netze mit funktionell verbundenen Eingängen wurden von Jok-Han Pao entwickelt und sind als **Neuronale Netze höherer Ordnung** bekannt [118].

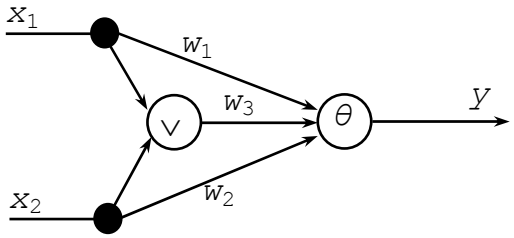


Abbildung 3.5 Darstellung der EXOR-Funktion mit zwei Neuronen

Die Methode von Mkrttschjan kann auch für beliebige Boolesche Funktionen verwendet werden. Dazu benötigt man aber weitere Neuronen für eine Berechnung von zusätzlichen Eingangssignalen.

Bei der Realisierung der EXOR-Funktion (3.3) wird ein Netz mit zwei Neuronen verwendet, wobei das erste Neuron eine OR-Funktion bildet. Ein entsprechendes Netz zur Darstellung der EXOR-Funktion zeigt Abbildung 3.5.

Neuron ist kein Schwellwertelement.

Eine Alternative zu den zwei oben beschriebenen Methoden ist die Verwendung einer komplizierten Funktion anstatt einer Schwellwertfunktion als Aktivierungsfunktion des Neurons. In unseren früheren Forschungen [78] und [154] wurde vorgeschlagen, ein Polynom oder eine trigonometrische Funktion zu benutzen:

$$d\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)^k - \theta \geq 0, \tag{3.7}$$

$$\tanh\left(w_0 + \sum_{i=1}^{n} w_i x_i\right) - \theta \geq 0, \tag{3.8}$$

wobei tanh – der hyperbolische Tangens ist, d und k - Koeffizienten.
Wie man in Abbildung 3.6 sieht, trennt eine graphische Darstellung der Aktivierungsfunktion (3.7) alle Punkte mit Nullwerten von allen anderen Punkten mit Einswerten.

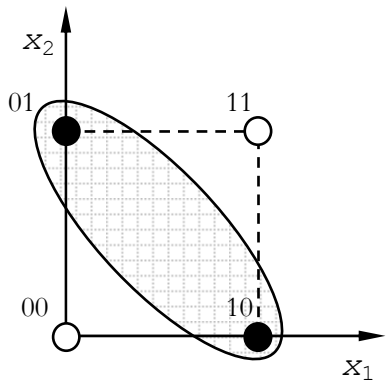


Abbildung 3.6 Geometrie des Neurons mit einem Polynom als Aktivierungsfunktion

Im Fall der EXOR-Funktion (3.3) existieren die Koeffizientenwerte der Aktivierungsfunktion (3.7), bei denen nur ein einziges Neuron die EXOR-Funktion abbilden kann.

Beispiel: Für $k = 2$, $w_0 = d = -1$, $w_1 = w_2 = 1$ and $\theta = -0.5$ folgt aus der (3.7)

$$-0.5 - (x_1 + x_2 - 1)^2 \geq 0. \tag{3.9}$$

Ein Vorteil dieser Methode ist eine Eins-zu-Eins Abbildung einer Booleschen Funktion in das Neuron. Für die Darstellung (3.3) reicht ein in Abbildung 3.1 dargestelltes Neuron, das aber kein Schwellwertelement ist, weil seine Aktivierungsfunktion keine lineare Schwellwertfunktion ist.

Neuron mit komplexen Gewichten.

Eine weitere Methode, die das Perzeptron zur Darstellung der nicht separierbaren Booleschen Funktionen befähigt, wurde von Aizenberg vorgeschlagen [1]-[4]. Seine Idee ist trivial. Die Gewichtskoeffizienten im Neuron können komplex sein, und die Aktivierungsfunktion wird durch (3.10) bestimmt:

$$f(z) = \begin{cases} 0, & 0 \le \arg(z) < \frac{\pi}{2} \quad oder \quad \pi \le \arg(z) < \frac{3\pi}{2} \\ 1, & \frac{\pi}{2} \le \arg(z) < \pi \quad oder \quad \frac{3\pi}{2} \le \arg(z) < 2\pi \end{cases} \tag{3.10}$$

Es ist leicht zu sehen, dass die nicht linear separierbare Funktion (3.3) mit dieser Erweiterung des Perzeptrons realisiert werden kann.

Diese Methode kann auch für komplizierte Boolesche Funktionen angewendet werden. Der Autor beschrieb einem modifizierten Trainingsalgorithmus des normalen Perzeptrons für sein Neuron mit komplexen Gewichten, so dass nichtmonotone Boolesche Funktionen mit mehreren Variablen durch ein Neuron realisiert werden können.

3.2 Neuronale Netzwerke

3.2.1 Backpropagation Boolesche Neuronale Netze

Im vorangehenden Abschnitt wurden Möglichkeiten zur Darstellung elementarer Boolescher Funktionen durch ein einzelnes Boolesches Neuron behandelt. Für die typischen Einsatzfelder von Booleschen Neuronalen Netzen, wie Data mining [66], [173] und [175], Klassifikation [142], Mustererkennung [74], ist aber die Modellierung Boolescher Funktionen mit großen Anzahl von Variablen erforderlich. Mit der Erhöhung der Zahl von Variablen erhöht sich auch die Kompliziertheit dieser Funktionen. Mit wachsender Kompliziertheit der zu modellierenden Funktionen (außer linearer Separierbarkeit) entstehen auch viele anderen Fragen und Probleme. Dabei bleibt aber auch das oben diskutierte Problem der linearen Separierbarkeit bestehen. Alle Schwierigkeiten und Probleme, die bei der Modellierung von BF entstehen, gehen von den Besonderheiten des zu modellierenden Objektes und von dem Trainingsalgorithmus des Neuronalen Netzes aus.

Am meistens verbreitet zur Lösung von Abbildungsaufgaben der Eingangs- in die Ausgangsdatenmengen sind die vorwärts gerichteten Netze (FFNN). Da eine Boolesche Funktion ein statisches Objekt ist, verwendet man auch für die Modellierung Boolescher Funktionen die FFNN–Netze. Ein typisches Beispiel von feedforward–Netzen sind Neuronale Netze mit dem Backpropagation-Lernverfahren (Fehlerrückführungsmethode), das

in einer Vielzahl von Anwendungsbereichen untersucht und erfolgreich eingesetzt wurde. Es besitzt daher eine höhere praktische Relevanz. Viele existierende Variationen des Backpropagations werden zur Modellierung Boolescher Funktionen verwendet. Dabei hat das Lernverfahren Backpropagation viele Nachteile, die sich bei der Abbildung Boolescher Funktionen mit vielen Variablen auf FFNN besonders stark auswirken. Zu den kritischen Aspekten des Backpropogation-Verfahrens als Optimierungsverfahren gehören lokale Optima, Verhalten von BP bei Plateaus und Schluchten, Wahl der Lernrate, Wahl der Topologie des Netzes und Wahl der Fehlerrate [138].

Adaptierbare Boolesche Neuronale Netze. Lauria u.a. präsentieren in ihrem Buch „Adaptable Boolean neural networks" Adaptierbare Boolesche Neuronale Netze (ABNN), die zur Parameteranpassung fähig sind. Im ersten Teil ihres Buches erörtern die Autoren die Beiträge von McCulloch, Pitts [99], Hebb [67] und Caianiello [32]. Für die Beschreibung und Steuerung von Booleschen Netzen wurde eine höhere Programmiersprache „CONNET" und ein „Assembler" als eine formale Sprache vorgeschlagen. Der „Assembler" wird mit einem Booleschen Neuronalen Netz assoziiert und ist sehr praktisch bei der Beschreibung von Feingranulararchitekturen (FPGA, VLSI), die während der Laufzeit erlauben, Ressourcen zu verteilen und Steuerungs- und Datenpfaden zu komprimieren. Auch für die Simulation von Eingangssteuerungen und ganzen Netzarchitekturen wird der „Assembler" benutzt. CONNET bietet einen effizienten Knotenverteilungsalgorithmus an, der für die Netzsimulation und eine hocheffektive Erfüllung der Hebbschen Regel unentbehrlich ist. Im zweiten Teil des Buches werden Probleme analysiert, die durch die Implementierung der Hebbschen Regel entstehen. Als eine Alternative zur Hebbschen Regel wurde eine Unterteilung der Struktur des Booleschen Neuronalen Netzes in Baugruppen von Neuronen vorgeschlagen, wobei die Neuronenanzahl in jeder Gruppe begrenzt ist. Für die Beschreibung von Booleschen Neuronalen Netzen mit adaptierbaren Gewichten wurde eine neue Art des endlichen Zustandsautomaten, ein endlicher adaptierbarer Zustandsautomat, vorgeschlagen. Dabei hängen die Zustands- und die Ausgabefunktion dieses Zustandsautomaten eindeutig von Eingangssignalen des Netzes ab. Ein großer Vorteil von ABNN ist ihre Fähigkeit auch bei einer begrenzten Anzahl von Trainingspaaren zu lernen, und die richtige Ausgabe zu produzieren.

Funktionell erweiterte Boolesche Neuronale Netze. Eine effektive Verwendung der funktionellen Erweiterung von Eingängen nach Pao [119] wurde für Boolesche Neuronale Netze von Chu in der Arbeit [36] präsentiert. Ein einfacher Algorithmus für die Erzeugung des funktionell erweiterten Eingangs im Rahmen des ganzen Trainingsalgorithmus des Netzes wird von ihm angegeben. Außerdem wurde eine Methode zur Verminderung

der Anzahl von erweiterten Neuronen vorgeschlagen, die für ein erfolgreiches Training des Netzes notwendig sind.

Die Gleichung eines Neuronalen Netzes kann man in Matrixform beschreiben [119]:

$$F(\boldsymbol{X}\,\boldsymbol{W})=\boldsymbol{D} \tag{3.11}$$

wobei $\boldsymbol{X}$ – Matrix der Eingaben,

$\boldsymbol{W}$ – Matrix der Verbindungsgewichte,

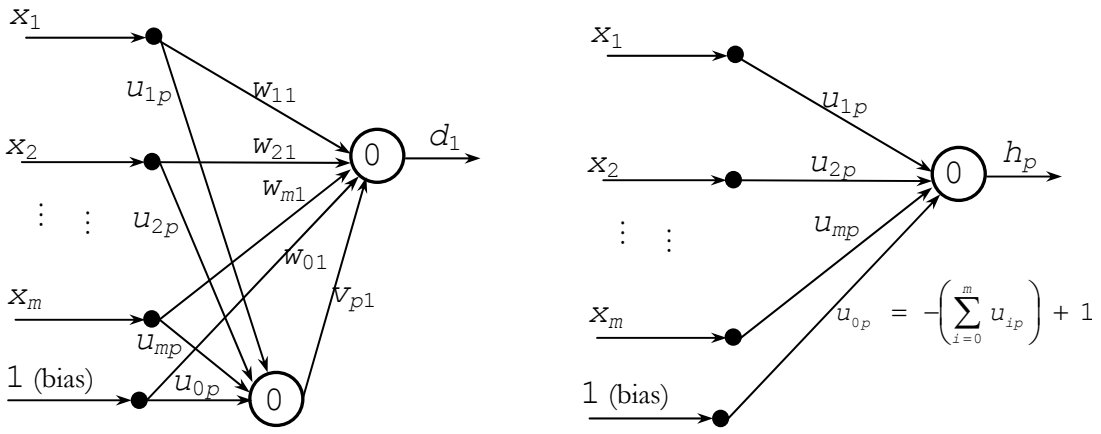
$\boldsymbol{D}$ – Matrix der Ausgaben,

F – Schwellwertfunktion.

Für die N Eingänge und M Trainingsbeispiele (der größte Wert von M ist 2^N) besteht eine Matrix $\boldsymbol{X}$ aus M Zeilen und N Spalten. Man nimmt an, dass die Vektoren $\boldsymbol{x}_i=(x_1, x_2, \ldots, x_N)$ für $i=1, \ldots, 2^N$ linear unabhängig sind. Dann gibt es zwei Fälle:

- falls $N+1 \geq M$, dann existiert die Lösung für (3.11),
- falls $N+1<M$, dann kann die Lösung für (3.11) nicht existieren, d.h. der Trainingsalgorithmus kann keine korrekten Werte der Matrix $\boldsymbol{W}$ finden.

Im zweiten Fall wird die Matrix $\boldsymbol{X}$ erweitert, d.h. funktionell erweiterte Eingänge werden zugegeben. Die Anzahl der zur Matrix $\boldsymbol{X}$ zu addierenden Spalten $H=M-N-1$. Dabei sollen alle Zeilenvektoren der Matrix $\boldsymbol{X}$ linear unabhängig bleiben.



a) b)

Abbildung 3.7 a) Allgemeine Struktur des Perzeptrons mit einem funktionell erweiterten Neuron;
b) Allgemeine Struktur des funktionell erweiterten Neurons

Die Abbildung 3.7 zeigt eine allgemeine Struktur des funktionell erweiterten Booleschen Neuronalen Netzes nach Chu und ein funktionell erweitertes Neuron.

Von Chu wurde eine Methode zur Bestimmung von Gewichten jedes erweiterten Neurons entwickelt, wobei die Bedingung der linearen Unabhängigkeit der Zeilenvektoren der Matrix $\boldsymbol{X}$ erfüllt wird.

Für i=0, …, N; j=1, …, T; k=1, …, M; p=0, …, H:

$$v_{pj} = w_{ij} = 0 \tag{3.12}$$

$$u_{ip} = x_i^k \tag{3.13}$$

$$u_{0p} = -\left(\sum_{i=0}^{N} x_i^k\right) + 1 \tag{3.14}$$

wobei j – Nummer des Ausgangsneurons,
k – Zeilennummer in der Matrix $\boldsymbol{X}$,
p – Nummer des funktionell erweiterten Neurons,
v_{pj} – Gewicht zwischen dem p-ten funktionell erweiterten Neuron und dem j-ten Ausgangsneuron,
u_{ip} – Gewicht zwischen dem i-ten Eingangsneuron und dem p-ten funktionell erweiterten Neuron,
w_{ij} – Gewicht zwischen dem i-ten Eingangsneuron und dem j-ten Ausgangsneuron.

Für weitere Anpassungen von Verbindungsgewichten wird ein typischer Backpropagation Trainingsalgorithmus verwendet.

Boolesche Neuronale Netze mit Booleschen Gewichten und Nullschwellwerten.
Verschiedenen Wissenschaftler haben sich in ihren Forschungen mit Booleschen Neuronalen Netzen mit Booleschen Gewichten auseinander gesetzt. In der Arbeit von Deolalikar [41] wird die Fähigkeit von BNN mit Booleschen Gewichten und Nullschwellwerten zur Darstellung Boolescher Funktionen betrachtet. Ein mathematisches Modell eines Netzes mit diesen Beschränkungen wurde entwickelt. Anhand dieses Modells werden algebraische Manipulationen gezeigt. Im mathematischen Modell wird ein Paar von Eingang und Ausgang (X und Y) durch eine einzelne „normalisierte“ Variable Z_{XY} ersetzt. Vorausgesetzt wird dafür, dass nur ein- oder zweischichtige Netze zur Darstellung einer gegebenen Booleschen Funktion verwendet werden. Eine besondere Eigenschaft dieses Modells ist die Möglichkeit, ein- und zweischichtige Netze zu vergleichen.
Deolalikar definiert Variablen $\boldsymbol{Z}_{ki}^{l}$, l=1, $1 \le k \le K$, $2 \le i \le m_L$:

$$\boldsymbol{Z}_{ki}^{1} = Y_{ki}\boldsymbol{X}_k \tag{3.15}$$

und für $2 \le l \le L$+1:

$$\boldsymbol{Z}_{ki}^{l} = \left(\operatorname{sgn}\left(\boldsymbol{w}_1^{l-1} \cdot \boldsymbol{Z}_{ki}^{l-1}\right) \cdot \cdot \cdot \operatorname{sgn}\left(\boldsymbol{w}_{m_{l-1}}^{l-1} \cdot \boldsymbol{Z}_{ki}^{l-1}\right)\right) \tag{3.16}$$

wobei sgn() – die Vorzeichenfunktion, die +1 nur dann zurückgibt, wenn ihr Argument positiv ist,

l – Schichtnummer, L – Anzahl von Schichten im Netz,

K – Anzahl von binären Eingangsvektoren Boolescher Funktion,

k – Nummer des Vektors, i – Neuronsnummer in der Schicht L,

Y_{ki} – Funktionswert, $\boldsymbol{X}_k$ – Eingangsvektor und $\boldsymbol{w}_i^l$ – Gewichtsvektor.

Es wurde bewiesen, dass folgende drei Aussagen für ein Netz mit fixierten Gewichten gleichwertig sind:

Das Netz realisiert die Abbildungen $\boldsymbol{X}_k \rightarrow Y_k$, $1 \le k \le K$.

Das Netz realisiert jede Abbildung $\boldsymbol{Z}_{ki}^1 \rightarrow (* \cdots 1 \cdots *)$, $1 \le k \le K$, $1 \le i \le m_L$, wobei ‚*' „don't care" bezeichnet und 1 in der Position i steht. Das bedeutet, falls $\boldsymbol{Z}_{ki}^1$ als Netzeingang betrachtet wird, dann ist die Ausgabe des Neurons mit der Nummer i gleich 1.

$\boldsymbol{Z}_{ki}^{L+1} \rightarrow (* \cdots 1 \cdots *)$, $1 \le k \le K$, $1 \le i \le m_L$, wobei ‚1' wieder in der Position i steht.

Auf diese Weise können die Eingangsvariablen $\boldsymbol{X}$ und die Ausgangsvariable Y im Netz mit Nullschwellwert durch eine einzelne „normalisierte" Variable $\boldsymbol{Z}$ ersetzt werden. Das Netz stellt $\boldsymbol{X}_k$ in Y_k dar, wenn $\boldsymbol{Z}_{ki}^1$ gleich $(* \cdots 1 \cdots *)$ ist. Genauer gesagt, wenn man ein Netz mit dem Eingang $\boldsymbol{Z}_{ki}^1$ hat, produziert ein i–tes Neuron der Schicht L eine Ausgabe 1, und diese Feststellung gilt für $1 \le k \le K$ und $1 \le i \le m_L$. Entsprechend darf man ein Ein-Ausgangspattern zu +1 normalisieren und weiterhin nur den normalisierten Ausgang $\boldsymbol{Z}$ benutzen. Darüber hinaus gilt, wenn die Eingabe der ersten Neuronenschicht $\boldsymbol{Z}_{ki}^1$ ist, dann wird die Ausgabe der (l-1)-ten Schicht als $\boldsymbol{Z}_{ki}^l$, $2 \le l \le L+1$ festgelegt.

Auf diesem Weg wurde eine Transformation der Mehrfachklassifikationsaufgabe [97] in die Zweifachklassifikationsaufgabe durchgeführt.

Eine Beschränkung zu den Nullschwellwerten macht die Netze noch einfacher. Außerdem wird gezeigt, dass die Hinzufügung einer dritten Neuronenschicht innerhalb des entwickelten Modells keinen Effekt bringt, und die durch die zweite Schicht gelegten Einschränkungen nicht gelöst werden können.

3.2.2 Boolesche Neuronale Netze mit sequentiellen Trainingsalgorithmen

Im vorangehenden Kapitel wurde eine Klassifizierung von Neuronalen Netzen nach verschiedenen Kriterien aufgeführt. Basierend auf den Trainingsalgorithmen von Neuronalen Netzen unterscheidet man zwei Gruppen von Verfahrensmethoden. Zur ersten Gruppe gehören Netze mit fixierter Struktur wie in der Neuronalen Netze mit Backpropagation-

Verfahren (BP). Zunächst wird die Anzahl von verborgenen Schichten und die Neuronenanzahl in jeder verborgenen Schicht festgelegt. Dann werden die Verbindungsgewichte und Schwellwerte im Parameter-Raum durch Abweichungsverminderung zwischen den Ergebniswerten der Berechnung und gewünschten Ergebniswerten angepasst. Diese Algorithmen können keine schnelle Konvergenz garantieren und brauchen eine lange Trainingszeit, die von der Netzgröße und von der Größe der Lerndatenmenge überproportional abhängt. Das zu trainierende Netz kann sehr oft die gewünschte Fehlergröße nicht erreichen.

Zur anderen Gruppe Neuronaler Netze gehören Netze, die durch sequentielle Trainingsalgorithmen spezifiziert werden. Im Verlauf des Trainings dieser Netze werden verborgene Schichten und verborgene Neuronen hinzugefügt. Beispiele solcher Methoden sind das ETL-Training (Expand-and-Truncate Learning) [75], das CSCLA-Training (Constructive Set Covering Learning Algorithm) [90] oder das FTF-Training (Functional on the tabular functions) [161], aber auch [19] und [33]. Sequentielle Trainingsalgorithmen sind aussichtsvoller, weil sie schneller konvergieren und somit eine kürzere Trainingszeit brauchen. Der Unterschied in der Trainingszeit zwischen sequentiellen Neuronalen Netzen und Neuronalen Netzen mit fixierter Struktur steigt besonders mit einer Vergrößerung der Variablenanzahl Boolescher Funktionen.

Seit Anfang der 90er Jahre sind die innovativen sequentiellen Trainingsalgorithmen und die konstruktiven Methoden zur Entwicklung von BNN Hauptrichtung der Forschungen vieler Wissenschaftler auf dem Gebiet der Neuronalen Netze [13], [19], [42], [88], [91] und [178].

Erste sequentielle Boolesche Perzeptrons. Marchand und Golea betrachten in ihrer Arbeit [93] ein Perzeptron mit einer festen Anzahl von Eingängen, mit einem Ausgang und mit einer nicht angegebenen Anzahl von verborgenen Neuronen. Sie schlugen einen der ersten sequentiellen Trainingsalgorithmen für feedforward Neuronale Netze vor. Ihr Lösungsvorschlag zur Strukturerrichtung des Netzes garantiert eine minimale Netzstruktur, wobei die Lernzeit annehmbar ist. Ihre Methode und zwei andere sequentielle Trainingsalgorithmen für feedforward Neuronale Netze, die fast gleichzeitig in [103] und [133] vorgeschlagen wurden, wurden mit üblichen iterativen Algorithmen verglichen und vorteilhaft bewertet. Der beschriebene Algorithmus ist eine wesentliche Verbesserung des Verfahrens von Rujan und Marchand [133]. Die neue Methode erlaubt einen größeren Satz von Netzstrukturen. Die durch Neuronen zu modellierenden Hyperebenen können sich innerhalb eines Hyperwürfels überschneiden. Gewichte dürfen ganze Zahlen sein und statt eines erschöpfenden Suchverfahrens in einer beschränkten Lösungsmenge wird ein Perzeptron-Algorithmus [105] verwendet. Im Vergleich zu dem Verfahren von

Mezard und Nadal, bei dem man mehrere verborgene Schichten erhält und zwei Arten von Neuronen in jeder Schicht vorkommen [103], ist in der Methode aus [93] nur eine Art von Neuronen notwendig und es wird ein minimaler Fehler erreicht.
Etwas später wurde ein Perzeptron-Trainingsalgorithmus mit Booleschen Gewichten und beliebigem Schwellwert von Golea und Marchand vorgeschlagen [60]. Netzparameter, die mit allen Trainingsbeispielen übereinstimmen, werden im Verlauf des Algorithmus nicht gesucht. Statt dessen wird ein binäres Perzeptron konstruiert, das auf verschiedenen in den Lehrbeispielen erhaltenen Probabilistic-Schätzungen basiert. Ein Vorteil dieses Verfahrens besteht darin, dass die Trainingszeit des Perzeptrons bei einer größeren Anzahl von Trainingsbeispielen nur linear vergrößert wird.

Kostenfunktion von Booleschen Neuronalen Netzen. Mayoraz und Aviolat stellen in ihrer Arbeit [97] neue Ideen für Trainingsalgorithmen vor. Neue Optimierungskriterien für das Training eines jeden hinzugefügten Neurons wurden angegeben, wobei sowohl die Neuronen- als auch die Schichtenanzahl in der Netzstruktur reduziert wurden. Der Funktionstest ihrer Vorschläge wurde bei der Entwicklung von Feed Forward Neuronalen Netzen mit Booleschen Schwellwerten und diskreten Gewichten erbracht. Die Konvergenz dieser Algorithmen wurde bewiesen. Einige experimentelle Ergebnisse in Bezug auf die Größe und die Generalisierungsfähigkeit der erzeugten Netze wurden angegeben.
Gemäß der Aufbauweise des Netzes unterscheiden Autoren zwei Kategorien von Trainingsalgorithmen für Boolesche Neuronalen Netze: Vorwärts- und Rückwärtsalgorithmen. Die Vorwärtsmethoden fügen neue Neuronen nach dem vorhandenen Teil des Netzes hinzu. Umgekehrt fügen die rückwärts gerichteten Trainingsalgorithmen neue Neuronen zwischen der Eingangschicht und bereits gebautem Teil des Netzes, in der Tat vor der vorhandenen verbogenen Schicht oder Teil der Schicht. Als Beispiele von vorwärtsgerichteten Trainingsalgorithmen sind „Tiling“ Algorithmus [103] und seine vereinfachte Variante „Tower“ Algorithmus [57], [115] und die Entscheidungsbaum-Algorithmen [60], [141]. Ein typisches Beispiel von konstruktiven Vorwärtsalgorithmen ist „Upstart” Methode [53].
Für die Darstellung einer nicht separierbaren Booleschen Funktion durch Vorwärts- sowie Rückwärtsalgorithmen wurden sequentielle Transformationen dieser Funktion durchgeführt, um sie zu vereinfachen und durch ein Neuron abbilden zu können. Dabei wurden folgende Definitionen zugrunde gelegt.

Definition 3.1. Eine Boolesche Funktion f: $\boldsymbol{B}^n \rightarrow \boldsymbol{B}$ ist eine **lineare Boolesche Schwellwertfunktion**, falls $w \in \boldsymbol{R}^n$ und $w_0 \in \boldsymbol{R}$ existieren, so dass gilt:

$$\forall \boldsymbol{b} \in \boldsymbol{B}^n, f(\boldsymbol{b}) = \mathrm{sgn}(w_0 + \boldsymbol{w}^{\mathrm{T}}\boldsymbol{b}), \tag{3.17}$$

Wobei sgn() – Vorzeichenfunktion, die +1 nur zurückgibt, wenn Argument positiv ist,

w_0 – Schwellwert

$\boldsymbol{w}$ – Gewichtsvektor der Funktion f,

$\boldsymbol{b}$ – Eingang.

Definition 3.2. Eine mit $\{(\boldsymbol{a}^k, b^k)\}_{k=1}^{p} \subset \boldsymbol{B}^n \times \boldsymbol{B}$ gegebene Darstellungsaufgabe T ist **kohärent**, falls $b^k \neq b^l$ für $\boldsymbol{a}^k \neq \boldsymbol{a}^l$ und jedes $k \neq l$. Diese Aufgabe ist **linear separierbar**, falls sie mit einem einzelnen Booleschen Neuron berechnet werden kann.

Algorithmus 3.1 MayAvi-Train - Skelett des Trainingsalgorithmus für BNN

Eingabe:
$T=\{(\boldsymbol{a}^k, b^k)\}$
Ausgabe:
net - Netz, das T ausführt

MAYAVI-TRAIN(T)
1 Füge eine Eingangsschicht ein
2 **do**
3 Start einer neuen Schicht
4 **do**
5 Lege die Parameter der Kostenfunktion $c(\boldsymbol{w}, w_0)$ fest
6 Füge ein neues Perzeptron ein
7 $(\boldsymbol{w}, w_0) \leftarrow (\boldsymbol{0}, ½)$, wobei $(\boldsymbol{w}, w_0)$ die Gewichte des neuen Perzeptrons sind
8 **do**
9 $(\boldsymbol{w}, w_0) \leftarrow \arg\min \{c(\boldsymbol{w}', w_0') \mid (\boldsymbol{w}', w_0') = m(\boldsymbol{w}, w_0), m \in M\}$
10 **while** ein Abbruchkriterium wahr ist
11 **while** alle Klassen sind wahr
12 $T=\{(\boldsymbol{a}^k \leftarrow \pi(\boldsymbol{a}^k), b^k)\}$, wobei π eine durch die neue gebildete Schicht realisierte Abbildung ist
13 **while** die neue gebildete Schicht hat ein Neuron
14 **return** net

Bei der Konstruktion einer neuen Schicht mit m Neuronen wird die Aufgabe $\{(\boldsymbol{a}^k, b^k)\} \subset \boldsymbol{B}^n \times \boldsymbol{B}$ durch eine Abbildung $\pi : \boldsymbol{B}^n \to \boldsymbol{B}^m$ in eine neue Aufgabe $\{(\pi(\boldsymbol{a}^k), b^k)\} \subset \boldsymbol{B}^m \times \boldsymbol{B}$ umgewandelt. $\pi(\boldsymbol{a}^k)$ bezeichnet eine innere Abbildung von $\boldsymbol{a}^k$. Eine Menge aller $\boldsymbol{a}^l$ mit gleichen Abbildungen, wie $\boldsymbol{a}^k$ durch π, ist eine Klasse von inneren Abbildungen $\pi(\boldsymbol{a}^k)$ und wird mit $[\boldsymbol{a}^k]$ bezeichnet. Die Klasse von inneren Abbildungen $[\boldsymbol{a}^k]$ ist **falsch**, falls sie aus Paaren $\{\boldsymbol{a}^k, \boldsymbol{a}^l\}$ mit $b^k \neq b^l$ besteht.

Von Mayoraz und Aviolat wurde ein globales Skelett präsentiert (siehe Algorithmus 3.1), das mehrere Varianten von Trainingsalgorithmen für BNN erlaubt.
Der Kern des Algorithmus liegt in der Kostenfunktion $c(\boldsymbol{w}, w_0)$. Diese führt die lokale Suche zur besten Gewicht-Konfiguration des neuen Neurons aus. Die Trennungsqualität wird durch das Minimum der Kostenfunktion gegeben. $c=0$ bedeutet, dass alle Klassen mit den gegenwärtigen Neuronen getrennt sind, und die Kostenfunktion der gegenwärtige Schicht abgeschlossen ist. Alle weiteren Algorithmen verwenden dieses Skelett und unterscheiden sich nur in der Definition der Kostenfunktion $c(\boldsymbol{w}, w_0)$.
Es wurde bewiesen, dass die in ihren früheren Arbeiten [96] und [98] beschriebenen Varianten der Kostenfunktion nicht ideal waren und die Konvergenz nicht garantiert wurde. Deshalb wurden neue Arten der Kostenfunktion angegeben, die die Konvergenz garantieren. Formel (3.18) beschreibt eine allgemeine Form der Kostenfunktion.

$$c(\boldsymbol{w}, w_0) = \sum_{[\boldsymbol{a}^k] \notin \boldsymbol{F}} \min\left(\varepsilon^k, \left|[\boldsymbol{a}^k]\right| - \varepsilon^k\right) \tag{3.18}$$

wobei $[\boldsymbol{a}^k]$ – eine falsche Klasse,

$\boldsymbol{F}$ - Menge von falschen Klassen und

ε^k - die klassische Kostenfunktion für den „Tiling“ Trainingsalgorithmus [103]:

$$\varepsilon^k = \left|\left\{\boldsymbol{a}^l \in [\boldsymbol{a}^k] \mid b^l \neq \operatorname{sgn}\left(w_0 + \boldsymbol{w}^T \boldsymbol{a}^l\right)\right\}\right| \tag{3.19}$$

Ein anderer Weg zur Verbesserung des Trainingsalgorithmus besteht in der Analyse der Potentiale des neuen Neurons und eine weitere Minimierung der Kostenfunktion. Statt der Gewichte sollen die Potentialwerte als Argumente der Kostenfunktion verwendet werden. Wie die in [97] angeführten Beispiele zeigten, benötigen ihre Netze im Vergleich zu anderen Trainingsmethoden eine kleinere Neuronenanzahl.

3.2.3 „Boolean-like“ Trainingsalgorithmus

Gray und Michel führten im Jahre 1992 einen Trainingsalgorithmus (BLTA - Boolean-like Training algorithm) für Boolesche Neuronale Netze ein [61]. BLTA generiert eine vierschichtige Architektur des FF-Netzes, das eine Eingangsschicht, eine verborgene Schicht, eine Hemmungsschicht und eine Ausgangsschicht umfasst. Grundsätze von BLTA wurden aus der Booleschen Algebra mit einer Erweiterung abgeleitet. Dabei wurde die Trainingsgeschwindigkeit von BLTA im Vergleich zu Lernverfahren von Netzen, die die Gradient-Abstiegstechnik benutzen, vergrößert. BLTA benutzt eine dynamische Technik, die eine vollständige Lerndatenmenge verlangt und dabei die Konvergenz für jede Boolesche Funktion garantiert. Besonders, bei vollständig definierten Funktionen mit vielen Variablen ist eine große Anzahl von verborgenen Neuronen erforderlich. Das ist ein

Nachteil des BLTA. Grundsätzlich basiert BLTA auf der Technik der Funktionsminimierung durch Karnaugh-Pläne.

Das Neuron wird durch (3.20) und (3.21) beschrieben:

$$U = \sum_{i=1}^{m} W_i X_i + \theta \tag{3.20}$$

$$V = G(U) = \begin{cases} 0, & U < 0; \\ 1, & U \geq 0, \end{cases} \tag{3.21}$$

wobei

X_i - Eingabesignal, i=1, …, m,

W_i - Gewichtswert, i=1, …, m,

θ - Schwellwert und

V – Boolesches Ausgabesignal der Aktivierungsfunktion des Neurons G mit reellem Argument U.

Die Gewichte zwischen den Schichten können drei Werte annehmen {-1; 0; 1}, wobei der Nullwert das Fehlen einer entsprechenden Verbindung bedeutet. Die Schwellwerte sind ganze Zahlen.

Vor dem Training werden die Eingabesignale durch die Eingangsschicht zu {-1; 1} kodiert:

$$Q_i = E(X_i) = \begin{cases} 1, & X_i = 0 \\ -1, & X_i = 1 \end{cases} \tag{3.22}$$

wobei

$E(X_i)$ - die Eingangskodierungsfunktion und

Q_i – das Ausgabesignal der Eingangsschicht ist.

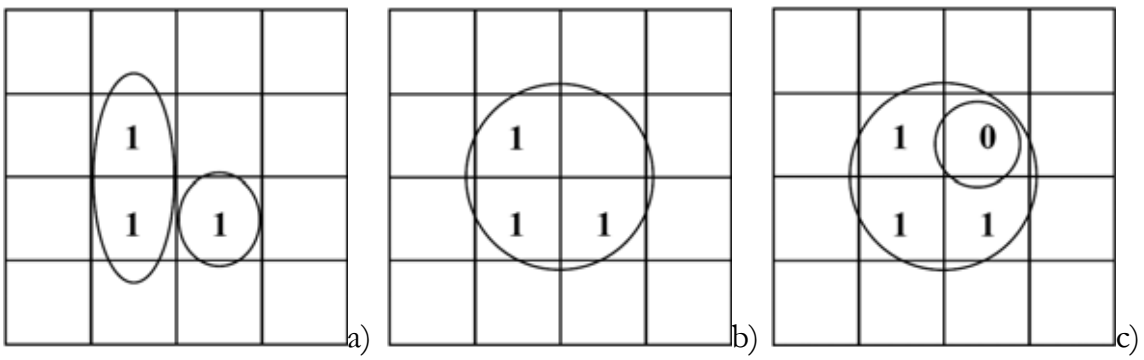


Abbildung 3.8 Operationen nach Gray-Michel
für den Aufbau einer verborgenen Schicht von BNN:
a) „ausführliche Darstellung“, b) „Generalisierung“, c) „Modifizierung“ [61].

Für die Erzeugung einer verborgenen Schicht werden drei Arten von Operationen definiert:

1. „ausführliche Darstellung“,
2. „Generalisierung“,
3. „Modifizierung“ (Abb. 3.8).

Die ausführliche Darstellung verlangt nur die genauen Abhängigkeiten der Signale, die implementiert werden sollen. Die Generalisierung erlaubt eine Abbildung dieser Abhängigkeiten über zusätzliche Abhängigkeiten, die durch den Algorithmus erzeugt werden können. Die Modifizierung ist in der Lage, alle unerwünschten Beziehungen zu korrigieren.

Gemäß diesen drei Operationen werden folgende sieben Regeln befolgt, um Neuronale Netze zu trainieren:

R1: Erlaubt ist eine Gruppierung von genau zwei Elementen (2-Kreis).

R2: Jedes Element darf nur in einer Gruppe auftreten.

R3: Jeder Minterm, der in einem 2-Kreis nicht eingeschlossen wird, soll als ein 1-Kreis vertreten werden.

R4: Ein neuer Minterm wird in einem 2M-Kreis vereinigt ($M \geq 2$) wenn und nur wenn ein $2M^{-1}$-Kreis existiert, der den neuen Minterm umfassen kann, sonst wird R5 verwendet.

R5: Jeder neue Minterm, der über R4 nicht vereinigt werden kann, wird durch R1, R2 und R3 ausführlich abgebildet.

R6: Der bekannte Minterm der Funktion F soll zuerst bei der Generalisierung abgebildet werden; diese Operation wird durch die ausführliche Darstellung von allen bekannten Mintermen der Funktion F, wie erforderlich, befolgt, um die falsch generalisierte Ausgabe der Funktion F zu hemmen.

R7: Alle bekannten Minterme von F und $\overline{F}$, die durch das vorhandene Netz nicht korrekt abgebildet sind, sollen durch die „ausführliche Darstellung“ an das Netz hinzugefügt werden.

R1, R2 und R3 sind Regeln für die ausführliche Darstellung. R2 illustriert, dass sich zwei 2-Kreise nicht überschneiden dürfen. Jedes verborgene Neuron bildet durch R1, R2 und R3 nur 1 oder 2 Punkte ab, und jede Boolesche Funktion mit M Eingängen kann durch 2M/2 verborgene Neuronen in einer verborgenen Schicht ausführlich dargestellt werden. Diese Verfahrensweise wird „ausführliche Darstellung“ in BLTA genannt. R4 und R5 sind Regeln für die „Generalisierung“. R4 erlaubt eine Erweiterung eines 2-Kreises in einen 4-Kreis, dann 8-Kreis und so weiter. Es wird zuerst versucht einen Funktionswert durch die Regeln R4 und R5 in einem verborgenen Neuron abzubilden. Wenn das nicht erfolgreich ist, werden R1, R2 und R3 für eine ausführliche Darstellung benutzt. Regel R6

definiert die Generalisierungsordnung eines Minterms. Wenn ein Konflikt zwischen einem vorher bekannten und einem neuen, durch die Generalisierung abgebildeten, inneren Minterm der Funktion F oder $\overline{F}$ aufgetreten ist, bedeutet das eine „Übergeneralisierung". Dabei sollte die Regel R7 für die Korrektur vorhandener Speicher durch hemmende Neuronen verwendet werden.
BLTA erweitert das Minimierungsverfahren für Boolesche Funktionen durch Karnaugh-Pläne, aber es benutzt für die Darstellung einer Booleschen Funktion durch ein Neuronales Netz eine kaum modifizierte disjunktive Form. Jede beliebige Boolesche Funktion kann durch das BLTA abgebildet werden. Dazu werden viele verborgene Neuronen benötigt, besonders bei der Modellierung von komplizierten Booleschen Funktionen.

3.2.4 „Expand-and-Truncate"-Trainingsalgorithmen

Kim und Park schlugen den Trainingsalgorithmus „Expand-and-Truncate Learning" (ETL) [75] vor. Sie definierten in der geometrischen Darstellung Boolescher Funktion eine Menge von Punkten (Set of Included True Vertices (SITV), die den Einswerten einer zu modellierenden Booleschen Funktion entsprechen und von den restlichen Punkten durch eine Hyperebene getrennt werden können. Im Verlauf des ETL-Algorithmus können die Mengen von Eins-Punkten und Null-Punkten umgetauscht werden, falls das SITV nicht erweitert werden kann.
ETL basiert auf der geometrischen Analyse der Lerndatenmenge. Durch das ETL wird ein Satz von trennenden Hyperebenen erzeugt, die für das Trennen der Eins-Punkte von Null-Punkten erforderlich sind. Dabei werden die Verbindungsgewichte und Schwellwerte bestimmt. Die Autoren vergleichen ihre Methode mit BP. Als Vorteile von ETL werden folgende Eigenschaften genannt: In ETL werden nur ganze Zahlen für die Verbindungsgewichte und Schwellwerte verwendet, was eine Hardware-Implementierung erleichtert. ETL garantiert die Konvergenz und bestimmt automatisch eine erforderliche Neuronenanzahl in der verborgenen Schicht. ETL baut ein dreischichtiges Neuronales Netz mit einer Eingangsschicht, einer verborgenen Schicht und einer Ausgangsschicht auf.
Eine Boolesche Funktion mit n Eingängen wird als ein n-dimensionaler Hyperwürfel betrachtet, wobei alle positiven Punkte dieses Hyperwürfels von negativen Punkten durch k (n-1)-dimensionale Hyperebenen getrennt werden können. Um alle Hyperebenen zu finden, wird zuerst eine Menge von positiven Punkten (SITV) definiert, die von den restlichen Punkten durch eine Hyperebene getrennt werden können. Das gefundene SITV wird erweitert, um möglichst viele Punkte mit demselben Wert einzuschließen. Kann die SITV nicht weiter vergrößert werden, so wird ein nächstes SITV von negativen Punkten

bestimmt und so weiter. Am Ende sind alle SITV gefunden, wobei die Punkte zwischen zwei konsekutiven Hyperebenen denselben gewünschten Ausgang haben. Zwei durch eine Hyperebene getrennte Konsekutivgruppen der Punkte haben verschiedene Ausgänge. Demnach trennen diese k (n-1)-dimensionalen Hyperebenen alle Punkte x_1, x_2, …, x_n in k+1 Gruppen von Punkten mit denselben Werten (entweder 0 oder 1).
Eine graphische Darstellung des ETL-Algorithmus wird in der Abbildung 3.9 gezeigt.

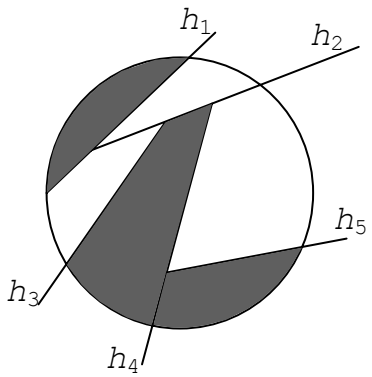


Abbildung 3.9 Visualisierung des ETL-Trainings

Schwarze Gebiete bezeichnen positive SITV und weiße Gebiete bezeichnen SITV von negativen Punkten. Der ETL-Algorithmus beginnt mit der Auswahl von einem Kernpunkt. Punkte, die in kein SITV eingeschlossen sind, werden nacheinanderen untersucht, um in einer neuen SITV mit diesem Kernpunkt eingeschlossen zu werden. Zum Schluss schließt SITV soviel wie möglich Punkte ein. Wenn keine Punkte mehr dem SITV hinzugefügt werden können, ist der erste Trennhyperebene gefunden. wenn jedoch diese Hyperebene nicht alle positiven Punkte von allen negativen Punkten trennt, so muss eine zweite Trennhyperebene gefunden werden. Dieser Prozess wird fortgesetzt, bis alle positiven Punkte von allen negativen Punkten getrennt wurden. Auf diese Weise wird eine linear untrennbare Funktion in eine Reihe von linear trennbaren Funktionen zerlegt, wobei jede linear trennbare Funktion durch ein verborgenes Neuron abgebildet wird. Ausgaben von Neuronen der verborgenen Schicht werden durch in der Ausgangsschicht mit logischen Operationen AND oder OR verknüpft.
Trennt h_1 SITV mit Einsen wie in Abbildungen 3.9 dann wird OR benutzt. Trennt h_2 SITV mit Nullen, dann wird AND benutzt. Das Ausgangssignal des Neurons für ein in Abbildung 3.9 abgebildetes ETL wird durch (3.23) beschrieben:

$$B(h_1, h_2, h_3, h_4, h_5) = h_1 + h_2(h_3(h_4 + h_5)). \tag{3.23}$$

Die Anzahl der durch ETL erzeugten verborgenen Neuronen ist gleich der Anzahl von Hyperebenen. Jede Hyperebene wird durch ein verborgenes Neuron realisiert.
Die Aktivierungsfunktion des Schwellwertelementes wird durch (3.24) beschrieben.

$$y = \begin{cases} 1, & \sum_{i=1}^{n} w_i x_i - T \geq 0 \\ 0, & sonst \end{cases} \tag{3.24}$$

wobei y – Ausgabe des Neurons,

w_i – Gewicht zwischen dem Eingang mit der Nummer i und Neuron,

T – Schwellwert.

Für die Bestimmung von Gewichten und Schwellwerten von verborgenen Neuronen wurden drei Methoden vorgeschlagen. Die ersten zwei Methoden sind stark beschränkt, weil die Verbindungsgewichte auf 1, -1, 2 und -2 in der ersten Methode, und auf 1 und -1 in der zweiten Methode eingeschränkt wurden. Im Vergleich zu den ersten beiden Methoden wurde dritte Methode mehr verallgemeinert und wird meistens in der Praxis verwendet. Die Verbindung zwischen dem Neuron auf der verborgenen Schicht und dem Neuron auf der Ausgangsschicht wird 1 gesetzt, wenn das verborgene Neuron ein SITV von positiven Punkten abbildet. Für ein SITV von negativen Punkten wird als Verbindungsgewicht -1 gewählt. Der ETL-Algorithmus wurde in Anwendungen, wie die Ziffer-Erkennung in Handschriften [120][122] oder die Muster-Klassifikation des Brustkrebses [123] erfolgreich verwendet. Dabei hat sich ein gutes Ergebnis sowohl in Bezug auf die Geschwindigkeit des Trainingsalgorithmus als auch in Bezug auf die Anzahl von verborgenen Neuronen gezeigt.

Weitere Modifikationen von ETL wurden im Jahre 1997 von Yamamoto und 2002 von Shimada und Saito vorgeschlagen. Yamamoto und Saito verbesserten ETL bei der Modifizierung einiger Punkte des SITV in „unbestimmte“ (“don’t care”) und nannten ihren Algorithmus „Improved Expand-and-Truncate Learning“ (IETL). Erwartungsgemäß ist eine kleinere Neuronenanzahl erforderlich.

In ETL sowie in IETL hängt die Anzahl von verborgenen Neuronen von der Wahl des Kernpunktes und der Ordnung der SITV-Ausbreitung ab. Folglich werden verschiedene Strukturen von Neuronalen Netzen generiert. Verschiedene Kernpunkte und verschiedene Ordnungen (Permutationen) von Eingangswerten führen zu verschiedene Strukturen von Neuronalen Netzen. Außerdem müssen viele Trainingsbeispiele im Verlauf von ETL und IETL durchsucht werden, um jedes Neuron in der verborgenen Schicht zu bestimmen.

Im ETL-Algorithmus müssen alle durch eine vorherige Hyperebene getrennten Punkte in einem neuen SITV eingeschlossen werden. Der IETL-Algorithmus erlaubt dagegen, einen Teil von Punkten, die durch eine vorherige Hyperebene getrennt wurden, in einem neuen SITV auszuschließen. Bei der Visualisierung des Prozesses sieht es wie eine Überdeckung von Gebieten aus. Es wurde aber nicht angegeben, ob diese Punkte als „unbestimmte“

("don't care") sein müssen. Außerdem werden die Punkte nicht nacheinanderen betrachtet, was eine Programmierung des IETL schwierig macht.
Schimada und Saito betrachteten zwei Probleme des ETL: das Steuern einer Anzahl von verborgenen Neuronen und die Verminderung der Streuung von Parametern. Sie schlugen einen flexiblen Trainingsalgorithmus für 3-schichtige BNN vor. Alle Netzparameter sind ganze Zahlen. Der Algorithmus basiert auf dem ETL, wobei ein genetischer Algorithmus [40] und [139] für die Bestimmung der Verbindungsgewichte verwendet wird. Ihr Trainingsalgorithmus wird GAETL genannt. Im Vergleich zu ETL und IETL kann GAETL die Anzahl von verborgenen Neuronen reduzieren und die Parameterstreuung vermindern oder sogar völlig beseitigen. Ein generiertes BNN hat eine einfache Struktur und ist gut für die Hardware-Implementierung geeignet [140].
Noch ein weiterer Trainingsalgorithmus "Newly Expanded and Truncated Learning Algorithm" (NETLA) wurde von Sung vorgeschlagen. Eine optionale Methode zur Synthese von BNN wurde auf der Basis des ETL-Trainings erarbeitet und führt zur Minimierung der Neuronenanzahl in der verborgenen Schicht sowie der Anzahl von Verbindungen zwischen den Neuronen [155]. Dabei wird eine erweiterte Summe der Produkte von Booleschen Ausdrücken benutzt. Ein Vorteil von NETLA besteht darin, dass die Datenbeispiele unabhängig von der Eingangsordnung zusammengefügt werden können. Für die korrekte Arbeit des Algorithmus muss die Boolesche Funktion durch einen Ausdruck beschrieben werden, weil NETLA auf der Ausdrucksminimierung basiert. Jedes Eingangsbit soll sowohl durch seine ursprüngliche Form als auch durch seine Komplement zur Verfügung stehen. Das führt zur Verdoppelung der Anzahl von Eingangsneuronen.
Eine Kombination von BLTA und ETL wurde von Chaudhari und Tiwari [35] verwendet, um ein Netz zu trainieren, das Klassifikationsprobleme behandeln kann. Ihre Ideen wurden von Wang und Chaudhari weitergeführt und ein neuer Lernalgorithmus „Multi-Core Learning" (MCL) wurde vorgeschlagen. MCL ist ein Algorithmus, der auf ETL und BLTA basiert. Weitere Verbesserungen des MCL liefen auf einen flexiblen Trainingsalgorithmus „Multi-Core Expand-and-Truncate Learning" (MCETL) hinaus. Der Hauptunterschied zwischen ETL (IETL) und MCETL liegt darin, dass MCETL mit mehreren Kernpunkten beginnen und sie gleichzeitig erweitern kann. Wird MCETL mit einem Kernpunkt begonnen, entartet MCETL zum ETL Algorithmus. Für MCETL wurde auch eine einfache Wahlregel von Kernpunkten für SITV angegeben.
In den meisten Fällen reduziert MCETL die Anzahl von verborgenen Neuronen und braucht eine kürzere Trainingszeit im Vergleich zu ETL und IETL. In einigen Fällen benötigen IETL und MCETL die gleiche Anzahl von verborgenen Neuronen, aber die Gewichte und Schwellwerte in MCETL sind viel kleiner als diejenigen in ETL und IETL, was für die Hardware-Implementierung von BNN wichtig ist. Außerdem werden für die

Bestimmung von Verbindungsgewichten und Schwellwerten der verborgenen Schicht weniger Operationen benötigt. MCETL verbessert die Generalisierungs- und Darstellungsfähigkeit von BLTA, ETL und IETL.

3.2.5 Kaskaden- und Oil-Spot-Training

Eine sehr interessante Methode zur Lösung des Problems der Wahl der Neuronenanzahl im Netz, die für die Abbildung gegebener Boolescher Funktion notwendig ist, wurde von Martinelli, Mascioli und Bei [94] vorgeschlagen. In ihrer Arbeit beschäftigen sie sich mit einer vorgegebenen Kaskadenstruktur des Netzes und dem entsprechenden Trainingsalgorithmus, sowie der Kompliziertheit und die Abbildungsfähigkeit des Netzes. Ihr Algorithmus basiert auch auf geometrischen Annäherungen. Die Wissenschaftler betrachten den Fall einer Abbildung von unvollständig definierten Booleschen Funktionen. Dabei wird die Generalisierungsfähigkeit des Netzes untersucht. Der Algorithmus erzeugt automatisch ein Neuronales Netz, ist immer konvergent und kann sowohl binäre aber auch reelle Eingänge akzeptieren.

Grundsätzlich wird eine vollständige Abbildung einer Booleschen Funktion durch 2^N Beispiele beschrieben, wobei jedes Beispiel durch ein Eingabemuster und die zugehörige Ausgabe vorgegeben wird. Die Ausgabemuster werden durch Punkte eines Hyperwürfels im N-dimensionalen Eingangsraum durch 1 oder 0 dargestellt, abhängig von der gewünschten Ausgabe.

Die Diagonale des Hyperwürfels nennt man die Gerade, die einen Punkt aus der Klasse $0(V_0)$ mit dem Punkt aus der Klasse N (V_N) verbindet. Dann gilt folgende geometrische Eigenschaft: Die Punkte der Klasse K ($1 \le K \le N$-1) liegen in einer Hyperebene orthogonal zur Diagonale und schneiden die Hyperebene in einer Entfernung $K/\sqrt{N}$ von V_0. Die Gleichung dieser Hyperebene ist:

$$\sum_{i=1}^{N} x_i = K . \tag{3.25}$$

Folglich werden die Punkte der Klasse K durch zwei zu (3.25) parallele Hyperebenen abgegrenzt. Diese Hyperebenen sind vom Punkt V_0 jeweils $(K + b_1)/\sqrt{N}$ und $(K - b_2)/\sqrt{N}$ mit $0 < b_1, b_2 < 1$ entfernt. Deshalb kann das Paritätsproblem durch die Verwendung einer Reihe aus N zu (3.25) parallelen Hyperebenen gelöst werden:

$$\sum_{i=1}^{N} x_i = 0.5 + K \qquad K = 0, 1, \ldots, N - 1 \tag{3.26}$$

Die erwähnten Hyperebenen teilen den Hyperwürfel in N+1 Hyperbereiche, wobei jeder Hyperbereich die Punkte nur einer Klasse enthält. In Abbildung 3.10 werden die Paritätsentscheidungsbereiche im Fall N=3 gezeigt.

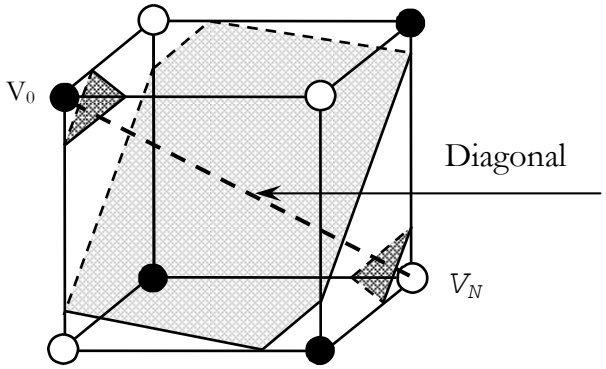


Abbildung 3.10 Visualisierung der Paritätsentscheidungsbereiche für N=3 [94]

Die durch den Trainingsalgorithmus erzeugte Netzstruktur (Abbildung 3.11) enthält M Neuronen, wobei $M \geq (N+1)/2$

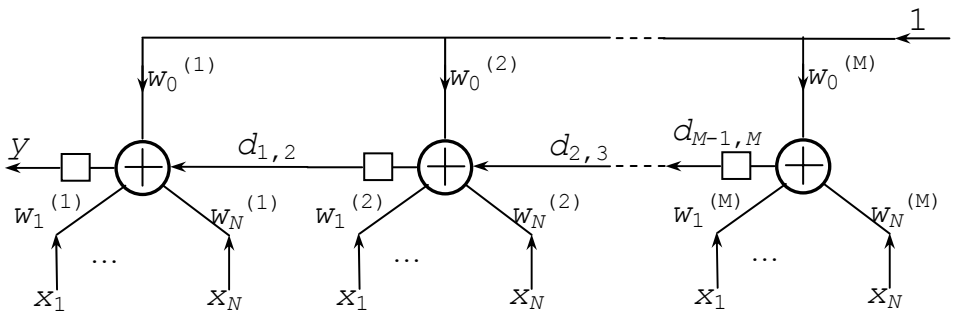


Abbildung 3.11 Kaskaden-Perzeptron [94]

Ein anderes auf dem Booleschen Hyperwürfel basierendes Trainingsverfahren für Boolesche Neuronale Netze wurde Oil-Spot-Algorithmus genannt. Es baut zweischichtige Neuronale Netze für die angegebenen Trainingsbeispiele auf. Dabei kann dieser Algorithmus auch auf reelle Werte erweitert werden. Im Vergleich zu einem Kaskadennetz optimiert dieser Trainingsalgorithmus die Netzstruktur in Hinsicht auf Neuronenanzahl.
Das Training führt im binären Hyperwürfel des Eingaberaums die Vereinigungen von Unterräumen ein. Jeder Unterraum bildet ein angegebenes Trainingsbeispiel ab. Die Suche der kleinsten Vereinigung von Unterräumen, die mit einem gegebenen Satz von nichtlinear trennbareren Trainingsbeispielen übereinstimmt, ist ein NP vollständiges Problem. Um dieses Problem zu lösen, wurde ein Approximationsalgorithmus angegeben, der eine erfolgreiche Abbildung jeder Booleschen Funktion garantiert. Der Algorithmus zeichnet sich durch eine direkte Kontrolle von trennenden Hyperebenen eines Entscheidungsbe-

reiches aus. Um die Unterräume, die den Entscheidungsbereich für das Problem bilden, zu bestimmen, werden die Punkte des Booleschen Hyperwürfels durch ein Verfahren untersucht [95]. Dieses Verfahren baut eine Schicht von verborgenen Neuronen schrittweise auf und, falls nötig, erzeugt er bei in jedem Schritt ein neues Neuron. Das Ziel ist, ein größt mögliches linear trennbares Gebiet aus dem verbliebenen Teil des Hyperwürfels zu finden.

3.2.6 Hamming-Abstand-basierende Trainingsalgorithmen

Xiaomin führte eine Hypersphäre des gewichteten Hamming-Abstands [91] ein. Durch die Anwendung dieser Idee erreicht man eine bessere Darstellungsfähigkeit jedes verborgenen Neurons und folglich eine verbesserte Lernfähigkeit von BNN. Im Jahre 2001 wurde von Xiaomin der „Constructive Set Covering" Trainingsalgorithmus - Constructive Set Covering Learning Algorithm (CSCLA) - vorgeschlagen [90].
Gemäß dem Hamming-Abstand (2.10) zwischen einem beliebigen Punkt x und einem Kernpunkt x^c gilt:

$$d_H\left(x^c, x\right) = \sum_{i=1}^{n} x_i^c \oplus x_i \tag{3.27}$$

Für einen gewichteten Hamming-Abstand zwischen x und x^c gilt:

$$d_W\left(x^c, x\right) = \sum_{i=1}^{n} g_i\left(x_i^c \oplus x_i\right) \tag{3.28}$$

wobei i – Nummer des Bits in der Eingangsvariable x_i und im Kernpunkt x_i^c.
Folglich kann eine Hypersphäre des gewichteten Hamming-Abstands durch (3.29) definiert werden:

$$R\left(d_w\right) = \left\{X = \left(x_1, x_2, \ldots, x_n\right) \in F_2^n \middle| d_W\left(X^c, X\right) \le d_W\right\} \tag{3.29}$$

wobei d_W – der Radius der Hypersphäre des gewichteten Hamming-Abstands ist.
Verschiedene Werte g_i und d_W erzeugen spezielle Hypersphären oder Hyperebenen:
$g_1 = g_2 = ... = g_n = 1$; $d_w \neq 0$ - eine Hamming-Hypersphäre,
$g_1 = g_2 = ... = g_n = 1$; $d_w = 0$ - nur ein Punkt eingeschlossen,
$g_1 = g_2 = ... = g_r = 1$; $g_{r+1} = g_{r+2} = ... = g_n = 0$; $d_w = 0$ - ein Hyperwürfel,
$g_1 = g_2 = ... = g_r = 1$; $g_{r+1} = g_{r+2} = ... = g_n = 0$; $d_w = 1$ - eine n-r-dimensionale Hamming-Hypersphäre.

$\bigcup_{i=1}^{k} A_i$ und $\bigcup_{j=k+1}^{L} B_j$ sind die Sätze von gewichteten Hamming-Hypersphären, wobei A_i eine gewichtete Hamming-Hypersphäre ist, die nur positive Punkte und B_j eine gewichte-

te Hamming-Hypersphäre, die nur negative Punkte einschließt. A_i oder B_j vertreten ein verborgenes Neuron. Folglich kann eine Boolesche Funktion $\bigcup_{i=1}^{k} A_i \Big/ \bigcup_{j=k+1}^{L} B_j$ durch ein 3-schichtiges Neuronales Netz mit einer Eingangsschicht, einer verborgener Schicht und einer Ausgangsschicht korrekt abgebildet werden. Jedes verborgene Neuron bildet eine Teilmenge von Punkten ab, die denselben gewünschten Ausgangswert (A_i oder B_j) haben.

CSCLA ist einfach, zuverlässig und braucht wenig Berechnungen. In CSCLA werden nur positive oder negative Punkte betrachtet. CSCLA ist schneller als ETL und IETL, weil nicht jedes Eingangs-Ausgangspaar bei der Erzeugung eines verborgenen Neurons betrachtet wird. Die Konvergenz von CSCLA wird garantiert. Ein großer Nachteil dieses Algorithmus ist die Notwendigkeit eines Hilfsnetzes. Deshalb verdoppelt sich der rechnerische Aufwand der Trainingsphase. In der Arbeitsphase bilden Punkte mit einem Hamming-Abstand 1 eine Teilmenge, die durch ein verborgenes Neuron abgebildet wird. Alle anderen Punkte mit dem Hamming-Abstand größer 1 dürfen diese Teilmenge nicht angehören. Das führt zur Vergrößerung der Anzahl von verborgenen Neuronen in Netzen, die von Xiaomin entwickelt wurden.

Ein sequentielles „Fenstertraining“ (sequential window learning - SWL) wird von Muselli für das Konstruieren von 2-schichtigen Booleschen Neuronalen Netzen präsentiert. Das Verfahren des sequentiellen Trainings [114] wurde zur Entwicklung von Neuronalen Netzen mit einer beliebigen Anzahl von Ausgängen erweitert. Das Trainingsverfahren kann mit einem beliebigen Trainingspaar aus der Lerndatenmenge begonnen werden.

Eine neue Art des Neurons mit einer fensterartigen Aktivierungsfunktion wurde eingeführt, das die Konvergenzgeschwindigkeit und die Kompaktheit der resultierenden Netze beträchtlich vergrößert. Dieses fensterartige Neuron erlaubt eine Entwicklung des schnellen SWL-Trainings, das auf der Lösung von algebraischen linearen Gleichungen basiert.

Ein fensterartiges Neuron wird durch (3.30) beschrieben.

$$y = \varphi\left(\sum_{i=0}^{n} w_i x_i\right) = \begin{cases} +1 & falls \left|\sum_{i=0}^{n} w_i x_i\right| \le \delta \\ -1 & sonst. \end{cases} \tag{3.30}$$

wobei der reelle Wert δ als Amplitude bezeichnet wird und φ die Aktivierungsfunktion des Fensterneurons ist.

Vereinfacht wird $\delta = 0$ angenommen, aber in der Praxis benutzt man meistens für die Amplitude Werte, die nicht gleich 0 sind.

Ein Fensterneuron kann immer durch drei Schwellwertneuronen ersetzt werden. Tatsächlich wird die Ausgabe eines allgemeinen Fensterneurons durch (3.31) gegeben.

$$y = \varphi\left(\sum_{i=0}^{n} w_i x_i\right) = \psi\left(\sum_{i=0}^{n} w_i x_i + \delta\right) - \psi\left(\sum_{i=0}^{n} w_i x_i - \delta\right) - 1 \qquad (3.31)$$

wobei ψ - Schwellwertfunktion mit dem Schwellwert 0.

Obwohl SWL eine effiziente Entwicklung des 2-schichtigen Perzeptrons erlaubt, hängt die Generalisierungsfähigkeit des Algorithmus von einer Vielfalt von Faktoren ab, die nicht direkt kontrollierbar sind. Deshalb wurde eine Technik für die Vorverarbeitung von gegebenen Trainingsdaten erarbeitet, die Hamming-Clustering (HC) genannt wird.

Eine übliche Methode des sequentiellen Trainings von BNN gruppiert die Eingabemuster, die zu derselben Klasse gehören, d.h. einander gemäß dem Hamming-Abstand nah sind. Diese Prozedur erzeugt einige Cluster im Eingangsraum, die die Klassenerweiterung bestimmen. Eine grundsätzliche Rolle in HC spielt eine Schablone. Die Schablone ist eine Zeichenkette aus binären Bestandteilen ‚+', ‚-' und „don't care"-Symbolen. Diese Schablone ähnelt einem Ternärvektor, wobei die Zeichen ‚1', '0, und ‚-' durch ‚+', ‚-' und ‚0' ersetzt wurden. Das HC-Verfahren ist eine Art der Funktionsminimierung mit der Ternärvektorliste [107], [148] und [153]. Das HC-Verfahren untersucht die lokalen Eigenschaften des Lernsatzes und veranlasst jede globale Überprüfung der Trainingsdatenmenge im Verlauf von konstruktiven Trainingsmethoden. Die Generalisierungsfähigkeit des gesamten Booleschen Netzes wird vergrößert. Hamming-Clustering kann nicht nur direkt im Trainingsalgorithmus SWL implementiert, sondern auch bei allen anderen Trainingsverfahren von vorwärtsgerichteten Booleschen Neuronalen Netzen verwendet werden. Ein SWL mit HC ist ein gutes Mittel zur Minimierung der Netzkompliziertheit und zur Verbesserung der Erkennungsgenauigkeit [21] und [113]. Auf diese Weise kann man ein optimales Verhältnis zwischen der Lokalisierungsfähigkeit und der Kapazität des Netzes finden [29] und [113]. HC ist auch im Stande, irrelevante Eingänge innerhalb des gegenwärtigen Lernsatzes zu erkennen und nutzlose Verbindungen zu entfernen. Die Kompliziertheit des resultierenden Netzes wird dadurch reduziert, was eine Strukturvereinfachung nach sich zieht. Diese Tatsache ist mit der Vapnik-Chervonenki-Dimension [14], [102] und [162] des Systems streng verbunden, die von der Anzahl der Gewichte im Neuronalen Netz abhängt [113].

Der Hamming-Abstand wird auch von Amaldi benutzt, um die Prinzipien der beiden von ihm entwickelten Trainingsalgorithmen zu erklären. Die erste Methode heißt SHIFT-Training und erzeugt Netze mit einer einzelnen verborgenen Schicht. Die PTI-Methode wurde für mehrschichtige Netze entwickelt. Die resultierenden Netze garantieren eine erfolgreiche Abbildung für jede gegebene Aufgabe mit Booleschen oder reellwertigen Eingängen. Das SHIFT-Trainingsverfahren ist nichts anderes als UPSTART-Training [53]. Das TILING-Training [103] mit einigen Änderungen im Anpassungsprozess der Gewich-

te eines gerade hinzufügten verborgenen Neurons baut ähnlich zu dem PTI-Training eine Netzstruktur Schicht für Schicht auf.

In Arbeit von Frean [51] wird eine gesteuerte Abkühlung (simulated annealing - SA) für die Suche eines optimalen (oder nahezu optimalen) Gewichtsvektors für jedes einzelne Neuron verwendet.

Dazu werden die Gewichte eines laufenden Neurons durch (3.32) verändert:

$$\boldsymbol{w}^{i+1} = \boldsymbol{w}^{i} + \eta_i \, (b^k - y^k)\boldsymbol{a}^k \tag{3.32}$$

wobei η_i - die Schrittweite, $\boldsymbol{w}^i$ - der Gewichtsvektor, $\boldsymbol{a}^k$ - der Eingabevektor, b^k - das richtige Ausgabesignal und y^k - das erhaltene Ausgabesignal ist.

Eine Besonderheit des SA-Verfahrens liegt in der speziellen Berechnungsweise der Schrittweite:

$$\eta_i = \frac{t}{t_0} \exp\left(\frac{-\left|v_i^k\right|}{t}\right) \tag{3.33}$$

wobei v_i^k - die Eingabe für $\boldsymbol{w}^i$ und $\boldsymbol{a}^k$, t und t_0 – Steuerparameter sind, die als „Temperatur“ bezeichnet werden [6] und [9].

Für die Bestimmung eines Gewichtsvektors des aktuellen Neurons können andere Methoden verwendet werden. Aber, wie es in den früheren Arbeiten von Amaldi gezeigt wurde, ist das SA - Verfahren vorteilhaft im Vergleich mit einigen anderen Methoden [6], [51], [52] und [56], die auf der minimalen mittleren quadratischen Abweichung oder auf der Minimierung der Quer-Entropie basieren.

3.3 Problemanalyse und Bewertung

In diesem Kapitel wurden die wichtigsten vorhandenen Erkenntnisse in der Entwicklung von BNN diskutiert. Außer der Effektivität und verschieden Vorteilen jeder einzelnen Methode wurden auch zahlreiche, bei der Modellierung Boolescher Funktionen durch Neuronale Netze entstehende Probleme, erörtert. Zu Problemen von iterativen Trainingsmethoden gehören alle aktuellen Probleme der iterativen Optimierungsverfahren, wie z.B. lokale Minima, flache Plateaus, Wahl der Schrittweite und des Dynamikbereiches [179]. BNN mit ITA- Algorithmen können keine schnelle Konvergenz garantieren und benötigen lange Trainingszeit, die überproportional von der Netzgröße und von der Größe der Lerndatenmenge abhängt. Das zu trainierende Netz kann den gewünschten Fehler sehr oft nicht erreichen.

Wie man sehen konnte, spielen die Booleschen Neuronalen Netze mit sequentiellen Trainingsstrategien eine bedeutendere Rolle. STA lösen die meisten Probleme, die bei der Anwendung von ITA vorhanden sind. Eine bemerkenswerte Eigenschaft von konstruktiven Lernalgorithmen ist, dass sie die Netzwerkarchitektur während des Trainings ändern. Im Vergleich mit den Algorithmen, die die Netzwerkarchitektur fixieren und die Parameter im Parameterraum herauszufinden versuchen, erhält man durch konstruktive Lernalgorithmen die optimale Netzwerkarchitektur sowie die optimalen Parameter. Konstruktive Lernalgorithmen garantieren die Konvergenz für BNN mit gegebenen endlichen Eingängen. Die Anzahl von verborgenen Schichten und verborgenen Neuronen wird automatisch ermittelt. Sequentielle Trainingsalgorithmen sind viel aussichtsvoller als ITA, weil sie eine schnellere Konvergenz garantieren und somit eine geringere Trainingszeit benötigen. Der Unterschied in der Trainingszeit zwischen sequentiellen Neuronalen Netzen und Neuronalen Netzen mit fixierter Struktur vergrößert sich besonders mit Vergrößerung der Anzahl der Variablen Boolescher Funktionen.

Die STA-Methoden werden durch Vorschläge von Marchand, Golea, Mayoraz, Aviolat, Gray, Kim, Yamamoto, Schimada, Sung, Chaudhari, Wang, Martinelle, Mascioli, Xiaomin, Muselli, Amaldi und vielen anderen präsentiert. Zu diesen STA-Methoden zählen u.a. SHIFT- Training, PTI-Methode, UPSTART-Training, TILING-Training, MCL, MCETL, FCLA, BLTA, ETL, IETL GAETL, NETLA, Oil-Spot Training, CSCLA und SWL, [37], [142], [155], [156] und [163]-[165].

Konstruktive (sequentielle) Strategien haben aber auch Nachteile. Da die STA relativ jung sind, existieren gegenwärtig noch relativ wenig wissenschaftliche Erkenntnisse. Jedoch wurden sie von einigen Wissenschaftlern erforscht und versucht zu klassifizieren. Amaldi hat in [8] alle Beschränkungen von STA in zwei Hauptarten unterteilt. Zu den ersten Problemen gehört das rechenaufwendige Training eines einzelnen Schwellwertneurons. Wenn die Anzahl von richtigen Klassifikationen maximiert oder die Anzahl von Fehlern minimiert werden soll, sind die Probleme, eine optimale Lösung zu finden oder zu approximieren NP-hard [5], [7], [8] und [10]. Diese Probleme verschärfen sich für Boolesche Neuronale Netze [8].

Das zweite Problem liegt in der sequentiellen Methode. Das Netzdesign wird im Verlauf des sequentiellen Trainings von einzelnen Neuronen gelöst. Jedoch, selbst wenn optimale Gewichtsvektoren für jedes einzelne Neuron verfügbar wären, können sequentiellen Strategien nicht garantieren, dass eine minimale Netzstruktur hervorgebracht wird.

Ein anderes großes Problem der Booleschen Neuronalen Netze mit sequentiellem Trainingsalgorithmus besteht darin, dass man beim Lernen ein riesiges Speichervolumen braucht [82], [154]. Besonders gilt das für mehrdimensionale Aufgaben mit großen Lerndatenmengen, die durch höhere Kompliziertheit charakterisiert werden. Zu den Aufgaben

mit gesteigerter Kompliziertheit gehören zweifellos die Darstellungs- und Verarbeitungsaufgaben Boolescher Funktionen, die von einer großen Anzahl der Variablen abhängen.
Aus der Analyse dieser Probleme folgt, dass ein Nachteil der Verwendung Neuronaler Netze (einschließlich sequentieller BNN) für die kompakte Darstellung und schnelle Bearbeitung Boolescher Daten in der nicht binären Zahlendarstellung liegt. Sehr oft versucht man durch die Erhöhung der Byteanzahl in der binären Zahlendarstellung eine möglichst kleinste Abweichung zu erreichen.
Für die Abbildungsaufgaben Boolescher Funktionen ist sogar die kleinste Abweichung unzulässig, deshalb kann die Genauigkeit nicht durch weitere Erhöhung der Byteanzahl für die Darstellung einen einzelnen Booleschen Werts erreicht werden. Daraus folgt, dass die Ideen, für die Verbindungsgewichte in BNN nur die Boolesche Werte zu verwenden [41], weiter beibehalten und erweitert werden sollten.
Prinzipiell können in Neuronalen Netzen die Eingangssignale und die Gewichte der Neuronen als reelle, dezimale oder binäre Zahlen dargestellt werden, aber in der Regel verwendet man reelle, seltener – dezimale Zahlen. Das Ausgangssignal des Neurons wird durch die Aktivierungsfunktion determiniert und kann ebenfalls reell, dezimal oder Boolesch sein. In jedem Fall operiert die Transferfunktion des Neurons mit Daten, die nicht Boolesch sind. Da die reellen und dezimalen Zahlen im Rechner durch mehr als ein Bit dargestellt werden, ist die Benutzung solcher Variablen für die Booleschen Daten, die nur ein Bit für jede Variable brauchen, unrationell. Außerdem benötigt man für die Realisierung mathematischer Operationen mit reellen oder dezimalen Zahlen viel mehr Rechenressourcen als für die Ausführung der Booleschen Operationen mit Booleschen Operanden.
Um diese Mängel zu beheben, müsste man Neuronale Netze verwenden, die die oben erwähnten Probleme nicht besitzen. Im folgenden Kapitel wird als Basiselement solcher Netze eine neue Art eines Neurons vorgeschlagen, das direkt mit Booleschen Signalen operiert und ausschließlich Boolesche Operationen benutzt. Diese Art des Neurons wird Boolesches Neuron (oder Boolesches Neuronales Element) genannt. Bereits hier lässt sich erkennen, dass dieses Boolesche Neuron ein viel versprechendes Mittel für die Entwicklung und den Aufbau neuen Typen von Netze sein wird.

Kapitel 4

Netze aus Booleschen Neuronen

4.1 Boolesches Neuron

4.1.1 Struktur eines Booleschen Neurons

Das mathematische Modell eines neuen Neurons, wurde erstmals in den Arbeiten [78]-[79] und [82] vorgestellt, die als Vorarbeiten vorliegender Dissertationsschrift entstanden sind. Diese neue Art eines Neurons wurde Boolesches Neuron (BN) genannt. Dabei wurden auch seine Struktur sowie der Aufbau eines Netzes aus Booleschen Neuronen beschrieben.

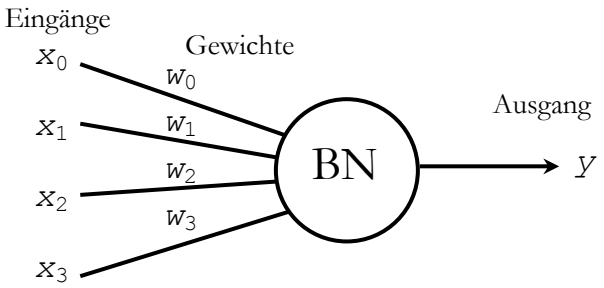


Abbildung 4.1 Allgemeine Struktur des Booleschen Neurons

Eine allgemeine Struktur eines einzelnen Booleschen Neurons mit vier Eingängen wird in der Abbildung 4.1 gezeigt. Man sieht, dass die Struktur eines Booleschen Neurons im Vergleich zur Struktur eines normalen Neurons nicht verändert wurde. Hieraus ergibt sich die Möglichkeit, das Boolesche Neuron entweder für die Synthese bekannter Modelle von Neuronalen Netzen zu verwenden oder neue Neuronale Architekturen mit Booleschen Neuronen zu entwickeln. In dieser Arbeit liegt der Schwerpunkt auf der Entwicklung eines neuen Neuronalen Netzes.

Das einfachste, aus einem einzelnen Neuron bestehende Boolesche Neuronale Netz zeigt Abbildung 4.1. Ein Boolesches Neuron kann also als ein einfaches Boolesches Neuronales Netz betrachtet werden.

4.1.2 Mathematische Beschreibung

Abbildungsfunktion. Ein Boolesches Neuron verhält sich grundsätzlich genau so wie ein normales Neuron. Die Abbildungsfunktion legt fest, wie aus den Eingabesignalen das Ausgabesignal gebildet wird. Für den allgemeinen Fall unterscheidet man drei Teilfunktionen, die zu der Abbildungsfunktion gehören (2.15) - (2.17). Da es sich hier um ein Boolesches Neuron handelt, müssen alle drei Teilfunktionen durch Boolesche Funktionen beschrieben werden. Als Ausgabefunktion können nur vier mögliche Boolesche Funktionen verwendet werden. Das folgt daraus, dass die Ausgabefunktion eines Neurons nur einen Eingabeparameter hat und laut (2.2) gibt es nur vier verschiedene Boolesche Funktionen, die für eine Booleschen Variable definiert sind. Alle möglichen für eine Variable definierten Booleschen Funktionen sind in der Tabelle 4.1 angegeben.

Tabelle 4.1 Wertetabelle der für eine Variable definierten Booleschen Funktionen

x	f_0=0	f_1=x	f_2=$not(x)$	f_3=1
0	0	0	1	1
1	0	1	0	1

Die Funktionen f_0 und f_3 sind uninteressant, weil diese konstant sind und nicht von der Eingangsvariable x abhängen. Daraus folgt, dass nur die Funktionen f_1 (Identität) und f_2 (Negation) als Ausgabefunktion des Booleschen Neurons benutzt werden können. Die Ausgabefunktion aller Booleschen Neuronen kann somit nur die Identität (4.1) oder die Negation (4.2) sein.

$$y(act_B) = act_B \tag{4.1}$$

$$y(act_B) = not(act_B), \tag{4.2}$$

wobei y - Ausgabesignal Boolesches Neuron,
act_B - Ausgangssignal der Aktivierungsfunktion,
not – Boolesche Operation „Negation“,
$y, act_B \in \{0,1\}$.

Daraus folgt, dass die Ausgabefunktion in die Aktivierungsfunktion integriert werden kann und somit nicht explizit auftreten muss. Da der Abhängigkeitscharakter der Abbildungsfunktion dabei unverändert bleibt, braucht man die Aktivierungs- und Ausgabefunktion nicht zu unterscheiden. Im Folgenden nehmen wir an, dass ein Boolesches Neuron aus zwei Teilen besteht: einem kumulativen Teil und einer Aktivierungsfunktion. Die Aktivierungsfunktion kann auch als Transferfunktion bezeichnet werden.

Aktivierung. Die schematische Darstellung eines Booleschen Neurons ist in der Abbildung 4.2 angegeben.

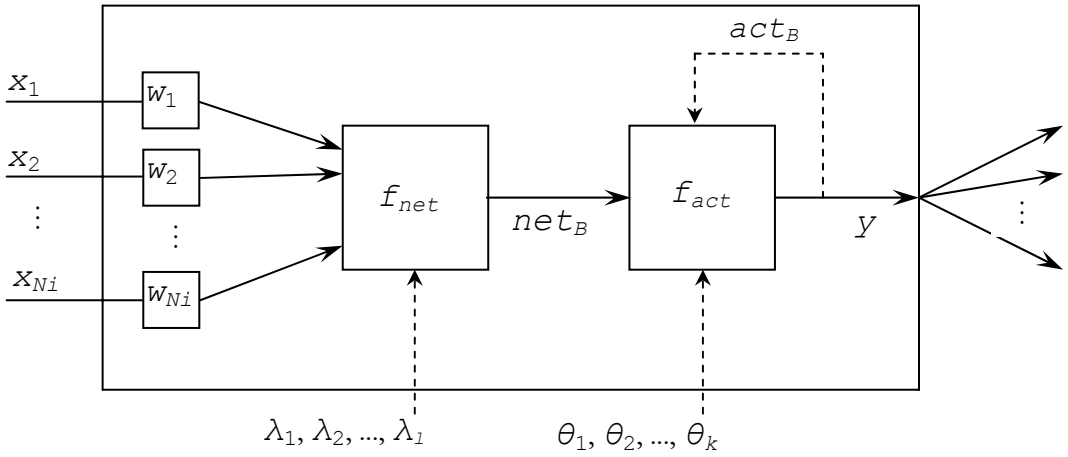


Abbildung 4.2 Schematische Darstellung eines einzelnen Booleschen Neurons

Die in der Abbildung 4.2 gezeigte Aktivierungsfunktion f_{act} ist eine Boolesche Transferfunktion, die von einem Argument net_B und eventuell einer Rückführung der aktuellen Aktivierung des Neurons $act_B^{(-1)}$ und einer bestimmten Anzahl von Parametern $\theta_1, \theta_2, ..., \theta_k$ abhängt.

$$act_B = f_{act}(net_B\ [, act_B^{(-1)},\ \theta_1, \theta_2, ..., \theta_k]\ ), \tag{4.3}$$

wobei gilt: $act_B, net_B, act_B^{(-1)}, \theta_1, \theta_2, ..., \theta_k \in \{0,1\}$.

Eine Aktivierungsfunktion mit der Rückführung der aktuellen Aktivierung des Neurons $act_B^{(-1)}$ wird in dieser Arbeit nicht benutzt.

Im einfachsten Fall hängt die Aktivierungsfunktion nur von einem Argument net_B und, einige Beispiele solcher Transferfunktionen sind in der Tabelle 4.1 zu sehen. Dabei können nur die Funktionen f_1 (Identität) und f_2 (Negation) als Transferfunktion des Booleschen Neurons benutzt werden. Die Funktionen f_0 und f_3 werden in der Regel als Transferfunktionen nicht verwendet. Die Verwendung von einer konstanten Funktionen als Transferfunktion ist aber nicht grundsätzlich ausgeschlossen.

Ähnlich, wie unter den normalen Neuronen, unterscheidet man ein lineares und ein nichtlineares Boolesches Neuron. Bei den Transferfunktionen $f_T = f_1$ oder $f_T = f_2$ (Tab. 4.1) wird ein Boolesches Neuron ein **lineares** Boolesches Neuron genannt.

Definition 4.1. Ein **lineares** Boolesches Neuron sei ein Boolesches Neuron mit der linearen Booleschen Aktivierungsfunktion.

Definition 4.2. Ein **nichtlineares** Boolesches Neuron sei ein Boolesches Neuron mit einer nichtlinearen Booleschen Aktivierungsfunktion.

Netzeingabefunktion. Das Argument net_B der Transferfunktion f_{act} bezeichnet man als **Netzeingabe**, die durch die Netzeingabefunktion f_{net} aus den Eingabesignalen $x_1, x_2, ..., x_{Ni}$, den Verbindungsgewichten $w_1, w_2, \ldots, w_{Ni}$ und eventuell einer bestimmten Anzahl von zusätzlichen Parametern $\lambda_1, \lambda_2, ..., \lambda_l$ berechnet wird:

$$net_B = f_{net}(\boldsymbol{x}_B, \boldsymbol{w}_B\ [, \boldsymbol{\lambda}_B]\ ), \tag{4.4}$$

wobei gilt: $\boldsymbol{x}_B = \{ x_1, x_2, ..., x_{Ni} \}$ - Vektor der Eingabesignale,

$\boldsymbol{w}_B = \{ w_1, w_2, \ldots, w_{Ni} \}$ - Vektor der Verbindungsgewichte,

$\boldsymbol{\lambda}_B = \{ \lambda_1, \lambda_2, ..., \lambda_l \}$ - Vektor zusätzlicher Parameter,

f_{net} –Boolesche Netzeingabefunktion,

N_i – Eingangsanzahl des Neurons,

$x_i, w_i, \lambda_j, f_{net}, net_B \in \{0, 1\}$ und

Index B bezeichnet den Booleschen Typ der Variablen.

Da im Folgenden stets Boolesche Neuronen behandelt werden, kann der Index B weggelassen werden. In die Berechnungen des Booleschen Neurons kann eine bestimmte Anzahl zusätzlicher Parameter $\lambda_1, \lambda_2, ..., \lambda_l$ und $\theta_1, \theta_2, ..., \theta_k$ eingehen. In dieser Arbeit werden solche Varianten von Booleschen Neuronen nicht benutzt.

Kumulationsoperator. Die Netzeingabe net wird durch einen kumulierten Wert der gewichteten Eingabesignale des Booleschen Neurons dargestellt.

$$net = \boldsymbol{\Omega}[\boldsymbol{x}, \boldsymbol{w}] = \Omega_1^{Ni}[\omega_i(x_i, w_i)], \tag{4.5}$$

wobei gilt: $\boldsymbol{\Omega}$ - der Kumulationsoperator mit den Vektoreingaben $\boldsymbol{x}$ und $\boldsymbol{w}$,

Ω_1^{Ni} - der Kumulationsoperator mit Zahleneingaben; die Grenzen 0 und N_i zeigen, dass der Operator für alle Argumente von i=1 bis $i=N_i$ gilt,

ω_i – Gewichtsfunktion für die Eingabe i.

Der Kumulationsoperator kann eine beliebige Boolesche Operation sein. Für die normalen Neuronen benutzt man meist als Kumulationsoperator eine Summe von gewichteten Eingabesignalen. Als Kumulationsoperator Boolescher Neuronen kann auch eine beliebige Boolesche Funktion sein. Wie weiter in diesem Kapitel beschrieben wird, eine Verwendung beliebiger Booleschen Funktion (auch nichtlinearer BF) als Kumulationsoperator von Booleschen Neuronen erweitert die Entwicklungs- und Konstruktionsmöglichkeiten Boolescher Neuronalen Netze.

Gewichtsfunktionen. Die Gewichtsfunktionen von Booleschen Neuronen ω_i und ω_j für $i \neq j$ sind voneinander unabhängig und können verschieden sein. Es ist aber nicht ausgeschlossen, dass innerhalb eines Booleschen Neurons, innerhalb einer Schicht des

Netzes oder überhaupt im ganzen Netz gleiche Boolesche Gewichtsfunktionen benutzt werden.

Wie in dem normalen Neuron ist ein Gewicht w_i jedem Eingang x_i eines Booleschen Neurons zugeordnet. Die synaptischen Gewichte eines Neurons bewirken die Verstärkung oder Hemmung der Eingangssignale. Für ein Boolesches Neuron können die synaptischen Gewichte nur die Werte 0 oder 1 annehmen. Das Eingangssignal wird durch das Gewicht $w_i = 1$ verstärkt und durch das Gewicht $w_i = 0$ gehemmt. Das Verbindungsgewicht des Booleschen Neurons ist bestimmend für die Existenz des entsprechenden Eingangs. Ist $w_i = 0$, fehlt das Eingangssignal x_i, bei $w_i = 1$ dagegen hat das Neuron einen Eingang mit dem Signal x_i. Dieses wird durch (4.6) beschrieben.

$$\begin{aligned} w_i{=}0 &\rightarrow \text{Ausgangssignal hängt nicht von } x_i \text{ ab,} \\ w_i{=}1 &\rightarrow \text{Ausgangssignal hängt von } x_i \text{ ab.} \end{aligned} \tag{4.6}$$

Entscheidend für die Anwendung von (4.6) ist die Wahl eines Kumulationsoperators der Netzeingabefunktion, weil (4.6) für jeden beliebigen Operator gelten soll. Man kann das Gewichten als eine Art Filter mit einer Filterfunktion ω_i beschreiben, die auf einem Paar (x_i, w_i) Eingabesignal-Gewicht definiert ist. Die Filterfunktion ω_i wurde oben Gewichtsfunktion genannt. Nachfolgend wird ein Fall beschrieben, in dem die Gewichtsfunktionen aller synaptischen Verbindungen eines Boolesches Neurons gleich sind.

Die einfachste und meist angewandte Art der Gewichtsfunktion ist eine elementare Boolesche Operation. Einige Beispiele, je nach Kumulationsoperator **Ω**, werden in Tabelle 4.2 gezeigt.

Tabelle 4.2 Beispiele von Gewichtsfunktionen für ausgewählte Kumulationsoperatoren

x_i	w_i	$\omega(x_i, w_i)$			
		Ω = '$\vee$'	**Ω** = '$\wedge$'	**Ω** = '$\odot$'	**Ω** = '$\oplus$'
x	0	0	1	1	0
x	1	x	x	x	x

Die angeführten Beispielfunktionen beschränken die möglichen Gewichtsfunktionen nicht. Es gilt: als Gewichtsfunktion eines Booleschen Neurons kann jede der 16 möglichen für 2 Argumente definierten Booleschen Funktionen gewählt werden.

Boolesches Neuron mit einer verallgemeinerten Transferfunktion. Da im Folgenden nur Boolesche Neurone ohne zusätzliche Parameter (**λ** und **θ**) und ohne Rückführung der aktuellen Aktivierung des Neurons $act_B^{(-1)}$ benutzt werden, hängt die Aktivierungsfunktion solcher Neuronen nur von einem Parameter ab und somit ist linear. Die

Verwendung nur linearer Boolescher Neuronen beschränkt die Entwicklungs- und Konstruktionsmöglichkeiten Boolescher Neuronalen Netze. Deswegen wird eine Modifizierung in die Berechnungen des Booleschen Neurons eingeführt.

Für die Berechnung der Netzeingabe wird eine Boolesche Netzeingabefunktion ohne Kumulationsoperator benutzt. Als Netzeingabefunktion wird eine beliebige Boolesche Funktion verwendet, die auf dem Vektor gewichteter Eingangssignale des Neurons definiert ist. Ähnlich, wie die Ausgabefunktion eines Booleschen Neurons oben ausgelassen und die Aktivierungsfunktion mit der Ausgabefunktion zusammengesetzt wurde, kann die Aktivierungsfunktion auch mit der Netzeingabefunktion verbunden werden. Die Netzeingabefunktion übernimmt dann auch die Rolle der Aktivierungsfunktion.

Die mit der Aktivierungsfunktion verbundene Netzeingabefunktion wird auch als Transferfunktion des Booleschen Neurons bezeichnet. Auf diese Weise wird eine Transferfunktion verallgemeinert und kann nichtlinear sein. Zu unterscheiden ist ein lineares Boolesches Neuron mit einer linearen Aktivierungsfunktion und ein verallgemeinertes Boolesches Neuron ohne Kumulationsoperation, das eine nichtlineare Transferfunktion hat.

Das Ausgangssignal des Booleschen Neurons y wird durch eine Relation zwischen den mit Booleschen Werten gewichteten Eingaben und der Booleschen Ausgabe definiert. Analog zur mathematischen Beschreibung eines normalen Neurons wird das Ausgangssignal eines Booleschen Neurons durch (4.7) beschrieben.

$$y = f_T(\boldsymbol{x}, \boldsymbol{w}), \tag{4.7}$$

wobei gilt: f_T –Transferfunktion des Booleschen Neurons.

Bei der Verwendung eines Booleschen Neurons mit einer Booleschen Transferfunktion wird die Zeit für die Konvertierung eines Eingangsvektors in das Ausgangssignal des Booleschen Neurons wesentlich reduziert. Ein weiterer Vorteil des Booleschen Neurons besteht in der Reduzierung des erforderlichen Speicherbedarfs, da für die Speicherung der Booleschen Daten viel weniger Speicherplatz als für reelle Daten benötigt wird. Ein zusätzlicher Vorteil ist die Eignung des Booleschen Neurons für die Hardware-Implementierung des Neuronalen Netzes, das nur aus Booleschen Neuronen besteht. Die oben angeführte verallgemeinerte Struktur und mathematische Beschreibung des Booleschen Neurons erlaubt eine Synthese von verschiedenen Strukturen Boolescher Neuronaler Netze, die zur Lösung unterschiedlicher Boolescher Aufgaben verwendet werden können.

Anderseits ist ein Boolesches Neuron kein „Allheilmittel“. Das Spektrum der Aufgaben, die mit Booleschen Neuronalen Netzen gelöst werden können, beschränkt sich auf Aufgaben, die mit der Booleschen Logik beschrieben werden können.

4.2 Boolesche Neuronale Netze mit einer verborgenen Schicht

4.2.1 Training

Im vorherigen Abschnitt wurde ein Boolesches Neuron als ein elementares Bauelement Boolescher Neuronaler Netze eingeführt. Boolesche Neuronale Netze können verschiedene Strukturen und Zielsetzungen haben. Mögliche Anwendungen Boolescher Neuronaler Netze werden später diskutiert. Zunächst stehen die Entwicklung und Beschreibung der Struktur des Booleschen Neuronalen Netzes im Vordergrund der Betrachtungen. Es existieren verschiedene Methoden zur Entwicklung von BNN, 2 Hauptmethoden werden ausführlicher dargestellt.

Die erste Methode besteht darin, dass übliche Neuronale Elemente in Strukturen von bekannten Neuronalen Netzen durch Boolesche Neuronen ersetzt werden und die entsprechenden Trainingsparadigmen für Boolesche Neuronen angepasst werden.

Die zweite Methode besteht in der Entwicklung neuer bisher unbekannten Strukturen und Paradigmen Boolescher Neuronaler Netze. Zuerst wird ein Neuronales Netz ausgewählt und auf dessen Basis versucht, ein Boolesches Neuronales Netz zu entwickeln.

Wie es im Kapitel 2 beschrieben wurde, kann man alle Neuronalen Netze nach dem Trainingsalgorithmus in zwei Grundtypen klassifizieren. Zur ersten Gruppe gehören iterative Trainingsalgorithmen, wie z.B. die bekannte Backpropagation-Methode. Eine andere Gruppe von Neuronalen Netzen benutzt nichtiterative Trainingsalgorithmen, die auch sequentielle oder konstruktive Trainingsalgorithmen genannt werden. Zweifellos besitzen die nichtiterativen Trainingsalgorithmen einen Vorzug im Vergleich zu den iterativen Algorithmen. Besonders gilt dies bei der Modellierung mehrdimensionaler Objekte, zu denen auch die Darstellungs- und Berechnungsaufgaben komplizierter Boolescher Funktionen gehören.

Als Basis für die Entwicklung von BNN wurde sequentielle Strategie des Trainings gewählt. Netze, die einen nichtiterativen Trainingsalgorithmus benutzen, benötigen im Vergleich zu anderen Neuronalen Netzen eine kürzere Trainingszeit und erzielen eine höhere Genauigkeit [159]. Durch den Einsatz des Booleschen Neurons anstelle des normalen Neuronalen Elements wurde eine neue Art eines Neuronalen Netzes erarbeitet, das Boolesches Neuronales Netz (BNN) genannt wird. Ehe ein Boolesches Neuronales Netz verwendet werden kann, muss es trainiert werden. Für das Training und die Anwendung des Booleschen Neuronalen Netzes werden speziell entwickelte Algorithmen benutzt, die im Weiteren beschrieben werden.

Der Trainingsprozess eines Booleschen Neuronalen Netzes, wie aller künstlichen Neuronalen Netze, liegt in der Bestimmung von Parametern. Eine ausführliche Beschreibung eines trainierten Booleschen Neuronalen Netzes mit einer bestimmten Architektur ist eine Darstellung jedes Booleschen Neurons und die Bestimmung aller Parameter für jedes Neuron. Ein Boolesches Neuronales Netz wird durch die Mengen synaptischer Funktionen und der Transferfunktionen der Neuronen beschrieben. Bei der Anwendung des trainierten Booleschen Neuronalen Netzes bekommt das Netz einen Vektor von Eingabesignalen (x_1, x_2, ..., x_{Nx}) und liefert einen Vektor von Ausgabesignalen (y_1, y_2, ..., y_{Ny}).
Die Grundidee der vorgeschlagenen Lernmethode des Booleschen Neuronalen Netzes ist eine Darstellung jeder Booleschen Funktion durch ein endliches Polynom (4.8) der vorher unbekannten Booleschen Funktionen $g_1(\boldsymbol{x})$, $g_2(\boldsymbol{x})$, ..., $g_{Nz}(\boldsymbol{x})$:

$$y_j(\boldsymbol{x}) = \overset{Nz}{\underset{i=1}{F_j}} \left[\omega_i\left(g_i(\boldsymbol{x}), w_i\right)\right], \tag{4.8}$$

wobei gilt: F_j – Polynom-Operator, eine Boolesche Operation,
N_z – Neuronenanzahl in der verborgenen Schicht.

Als Basis für das Lernverfahren, muss man zunächst eine Lernmenge definieren, die ein zu modellierendes Objekt beschreibt. Im Fall eines BNN ist dieses Objekt eine Boolesche Funktion oder Funktionsmenge. Eine Besonderheit der Darstellungsaufgabe Boolescher Daten besteht darin, dass man Neuronale Netze nur im Wiederherstellungsregime benutzen kann. Zur Lernmenge müssen alle Eingangsvektoren gehören, für die in der Arbeitsphase des Booleschen Neuronalen Netzes die Funktionswerte reproduziert werden sollen. Im Folgenden wird der allgemeine Fall einer Funktionsmenge betrachtet, da eine Funktionsmenge auch aus einer einzelnen Booleschen Funktion bestehen kann. Im Idealfall gehören zu der Lernmenge alle Binärvektoren der Booleschen Funktionsmenge, d.h. die gesamte Wertetabelle einer Menge Boolescher Funktionen. Die Eingabesignale des Netzes sind Funktionsargumente, die Ausgabesignale Funktionswerte.

Tabelle 4.3 Lernmenge eines BNN

$x_{1,1}$	$x_{1,2}$	$x_{1,Nx}$	$y_{1,1}$	$y_{1,2}$	$y_{1,Ny}$
$x_{2,1}$	$x_{2,2}$	$x_{2,Nx}$	$y_{2,1}$	$y_{2,2}$	$y_{2,Ny}$
$x_{i,1}$	$x_{i,2}$	$x_{i,Nx}$	$y_{i,1}$	$y_{i,2}$	$y_{i,Ny}$
$x_{2^{Nx},1}$	$x_{2^{Nx},2}$	$x_{2^{Nx},Nx}$	$y_{2^{Nx},1}$	$y_{2^{Nx},2}$	$y_{2^{Nx},Ny}$

Da die Funktionsmenge aus N_y Booleschen Funktionen y_1, y_2, ..., y_{Ny} besteht und jede Funktion y_i von N_x Eingangsvariablen x_1, x_2, ..., x_{Nx} abhängt, hat die Wertetabelle 2^{Nx} Zeilen und N_x+N_y Spalten, wie es in Tabelle 4.3 gezeigt ist. Alle Zeilen der Wertetabelle

sind in steigender Reihenfolge der Dezimaläquivalente der Eingangsvektoren geordnet, d.h. vom Eingabevektor (0, 0, …, 0) bis (1, 1, …, 1). Es besteht eine eindeutige Abhängigkeit zwischen der Werte jeder Booleschen Funktion und der Zeilennummern. Wegen dieser Zuordnung der Eingangsvektoren zu den Zeilen in der Wertetabelle, liefern die Werte der Eingabesignale x_1, x_2, ..., x_{Nx} keine zum Zeilenindex zusätzlichen Informationen für das Training des Netzes und können aus der Lernmenge ausgeschlossen werden. Die Lernmenge des Booleschen Neuronalen Netzes wird durch die Matrix $\boldsymbol{A}$ (4.9) dargestellt [82].

$$\boldsymbol{A} = \begin{Vmatrix} y_{1,1} & y_{1,2} & y_{1,Ny} \\ y_{2,1} & y_{2,2} & y_{2,Ny} \\ y_{i,1} & y_{i,2} & y_{i,Ny} \\ y_{2^{Nx},1} & y_{2^{Nx},2} & y_{2^{Nx},Ny} \end{Vmatrix} = \begin{Vmatrix} a_{1,1} & a_{1,2} & a_{1,Ny} \\ a_{2,1} & a_{2,2} & a_{2,Ny} \\ a_{i,1} & a_{i,2} & a_{i,Ny} \\ a_{2^{Nx},1} & a_{2^{Nx},2} & a_{2^{Nx},Ny} \end{Vmatrix} \tag{4.9}$$

Weiter wird das Trainingsverfahren des Booleschen Neuronalen Netzes beschrieben, wobei die Antivalenz als Basisoperation und somit als Polynom-Operator ausgewählt wurde. Die Folge der Transformationen in dem Trainingsalgorithmus umfasst folgende Schritte.
Man bestimmt einen Basiszeilenvektor $\boldsymbol{v}=(v_1, v_2, ..., v_{Nx})$, der für die weitere Parameterbestimmung des zu trainierenden Booleschen Neuronalen Netzes benutzt wird. Die Bestimmungsmethode des Vektors $\boldsymbol{v}$ kann verschieden sein. Als Grundkonzept des Trainingsalgorithmus wird hier zunächst die einfachste Variante betrachtet. Der Basisvektor $\boldsymbol{v}$ wird aus den Matrixzeilen frei gewählt, für die der durch (4.10) berechnete Koeffizient $D_i=1$ ist.

$$D_i = \bigvee_{j=1}^{N_y} a_{i,j}. \tag{4.10}$$

Wenn keine Zeile existiert, für die $D_i=1$, wird dieses Prozess beendet, d.h., das Neuronale Netz ist trainiert. Bei der Existenz von $D_i=1$ werden weitere Schritte durchgeführt.
Für jede Zeile berechnet man den Wert k_i:

$$k_i = \bigvee_{j=1}^{N_y} \left(a_{i,j} \wedge v_j\right). \tag{4.11}$$

Man fasst die Werte k_i für alle Zeilen zu einem Spaltenvektor $\boldsymbol{k}=(k_1, k_2, ..., k_{2^{Nx}})$ zusammen und erhält eine vollständige Wertetabelle der Booleschen Funktion $k^{(1)}$, die als eine Transferfunktion des verborgenen Neurons dargestellt wird. Jeder Wert $k_i^{(1)}$ dieser Funktion entspricht einer Zeile mit der Nummer i, jede Zeile ist eindeutig einem Vektor der Eingabesignale (x_1, x_2, ..., x_{Nx}) zugeordnet. Folglich ist jeder Wert $k_i^{(1)}$ einem Vektor der Eingabesignale (x_1, x_2, ..., x_{Nx}) eindeutig zugeordnet.

Man transformiert die Werte der Trainingsmatrix $\boldsymbol{A}$ durch (4.12).

$$a_{i,j}^{(2)} = a_{i,j}^{(1)} \oplus (v_j^{(1)} \wedge k_i^{(1)}), \tag{4.12}$$

wobei die in Klammern hochgestellten Indizes den Trainingszyklus angeben.
Das Lernverfahren wird iterativ mit dem ersten Schritt fortgesetzt. Die Iteration bricht ab, wenn das angegebene Kriterium im Schritt 2 erfüllt wird.
Im allgemein gilt es für den z-ten Trainingszyklus:

$$a_{i,j}^{(z+1)} = a_{i,j}^{(z)} \oplus \left(v_j^{(z)} \wedge k_i^{(z)}\right), \tag{4.13}$$

$$k_i^{(z)} = \bigvee_{j=1}^{N_y} \left(a_{i,j}^{(z)} \wedge v_j^{(z)}\right), \tag{4.14}$$

$$D_i = \bigvee_{j=1}^{N_y} a_{i,j}^{(z)}, \tag{4.15}$$

wobei $\boldsymbol{v}^{(z)}$- ein aus dem z-ten Trainingszyklus aus der Matrix $\boldsymbol{A}$ ausgewählter Zeilenvektor ist, für den $D_i=1$ gilt.
Die Gesamtheit aller Zeilenvektoren $\boldsymbol{v}^{(z)}$, $z=(1, 2, ..., N_z)$ wird als eine Matrix $\boldsymbol{V}$ bezeichnet. Diese Matrix $\boldsymbol{V}$ kann auch als eine Gesamtheit der Spaltenvektoren $\boldsymbol{v}_j$, $j=(1, 2, ..., N_y)$ betrachtet werden. Jeder Spaltenvektor stellt dann den Vektor der Gewichte für das j-ten Booleschen Neuron in der Ausgabeschicht des Netzes.
Zu beachten ist, dass die Werte der als Basiszeilenvektor $\boldsymbol{v}^{(z)}$ gewählten Zeile im Verlauf der Berechnung (4.13) in Nullwerte gesetzt werden. Da Matrix $\boldsymbol{A}$ aus 2^{Nx} Zeilen besteht und wenigstens eine Zeile dieser Matrix mit jedem Trainingszyklus in Null umgewandelt wird, terminiert der Trainingsalgorithmus nach maximal 2^{Nx} Iterationen. Die Matrix $\boldsymbol{A}$ ist am Ende der Trainingsprozedur eine Nullmatrix.

$$a_{i,j}^{(2^{N_x})} = 0, \qquad \forall\, i, j. \tag{4.16}$$

Daraus folgt, dass die Anfangsmatrix $\boldsymbol{A}$ durch endlich viele Antivalenzoperationen (4.17) wiederhergestellt werden kann.

$$a_{i,j}^{(0)} = \bigoplus_{z=1}^{Nz} \left(k_i^{(z)} \wedge v_j^{(z)}\right) \tag{4.17}$$

N_z ist die Neuronenanzahl in der verborgenen Schicht des BNN und auch die Anzahl der Trainingszyklen.
Die Formeln (4.13) - (4.15) definieren zusammen den Trainingsalgorithmus eines Booleschen Neuronalen Netzes. Die Spaltenvektoren $\boldsymbol{v}_j$ und die Booleschen Basisfunktionen $k^{(z)}$ sind das Resultat des Trainings. Aus (4.17) sieht man, dass die Elemente der Spal-

tenvektoren $\boldsymbol{v}_j$ die Gewichte der Neuronen in der Ausgabeschicht des BNN sind, die auch als $\boldsymbol{w}_j = (w_{1,j}, w_{2,j}, ..., w_{Nz,j})$ für $j = (1, 2, ..., N_y)$ bezeichnet werden. Die Booleschen Basisfunktionen $k^{(z)}$ hängen vom Vektor der Eingabesignale $\boldsymbol{x}$ ab und werden durch die Booleschen Neuronen der verborgenen Schicht realisiert.

4.2.2 Struktur

Für alle in dieser Arbeit betrachteten Netze werden nur die Booleschen Neuronen verwendet, deren Struktur und Arbeitsweise auf die Verwendung Boolescher Variablen für die Eingabe-, Ausgabe- und Zwischendaten beschränkt ist. Die Informationsverarbeitung in 3-schichtigen BNN wird hauptsächlich von den Booleschen Neuronen der verborgenen Schicht vorgenommen. Deswegen wird in diesem Abschnitt die Synthese von Booleschen Neuronen vorrangig für die verborgene Schicht behandelt.
Wie im Kapitel 2 gezeigt wurde, können Neuronale Netze mit Hilfe von Graphen (im Sinne der Graphentheorie) beschrieben werden. Um die Struktur eines Booleschen Neuronalen Netzes zu bilden, sind die Formeln des Trainingsprozesses (4.13) - (4.15) zu beachten. Aus den mathematischen Ausdrücken (4.13) und (4.17) folgt, dass die Elemente der Matrix $\boldsymbol{A}$ nach z Umwandlungszyklen des Trainingsalgorithmus folgendermaßen dargestellt werden können:

$$a_{i,j}^{(z)} = a_{i,j}^{(1)} \oplus \bigoplus_{l=1}^{z-1} \left(k_i^{(l)} \wedge v_j^{(l)} \right)$$

$$= a_{i,j}^{(1)} \oplus \bigoplus_{l=1}^{z-1} \left(k_i^{(l)} \wedge a_{v,j}^{(l)} \right) \tag{4.18}$$

wobei gilt: $a_{v,j}^{(l)}$ - Element der Matrix $\boldsymbol{A}$ aus der v-ten Zeile, die als Basiszeilenvektor $\boldsymbol{v}$ vor dem l-ten Umwandlungsschritt des Trainingsalgorithmus gewählt war.

Unter Verwendung von (4.18) und (4.15), wird der Ausdruck (4.14) in (4.19) transformiert:

$$k_i^{(z)} = \bigvee_{j=1}^{N_y} \left(\left(a_{i,j}^{(1)} \oplus \bigoplus_{l=1}^{z-1} \left(k_i^{(l)} \wedge a_{v,j}^{(l)} \right) \right) \wedge a_{v,j}^{(z)} \right)$$

$$= \bigvee_{j=1}^{N_y} \left(\left(a_{i,j}^{(1)} \wedge a_{v,j}^{(z)} \right) \oplus \left(\bigoplus_{l=1}^{z-1} \left(k_i^{(l)} \wedge a_{v,j}^{(l)} \wedge a_{v,j}^{(z)} \right) \right) \right) \tag{4.19}$$

$$= f\left(\boldsymbol{x}, k_i^{(z-1)}, k_i^{(z-2)}, \ldots, k_i^{(1)} \right) \tag{4.20}$$

Zu beachten ist, dass die Boolesche Transferfunktion $k^{(z)}$ des z–ten verborgenen Booleschen Neurons z nicht nur vom Vektor der Eingabesignale $\boldsymbol{x}$ abhängt, wie es vorher ge-

zeigt wurde, sondern auch von den Booleschen Transferfunktionen aller vorher festgelegten verborgenen Neuronen $\{k^{(z-1)}, k^{(z-2)}, ..., k^{(1)}\}$. Die Struktur der verborgenen Schicht wird durch (4.19) definiert. Die Transferfunktionen von Neuronen in der verborgenen Schicht werden durch (4.14), die Ausgabesignale werden durch (4.17) angegeben. Deswegen ist es ausreichend, in der Ausgabeschicht lineare Boolesche Neuronen zu benutzen, die eine Antivalenz der mit den ermittelten Gewichtskoeffizienten gewichteten Ausgabesignale der verborgenen Schicht bilden.
Unter Berücksichtigung von (4.13) - (4.15) und (4.20) ergibt sich ein Graph, der die Struktur des Booleschen Neuronalen Netzes angibt.

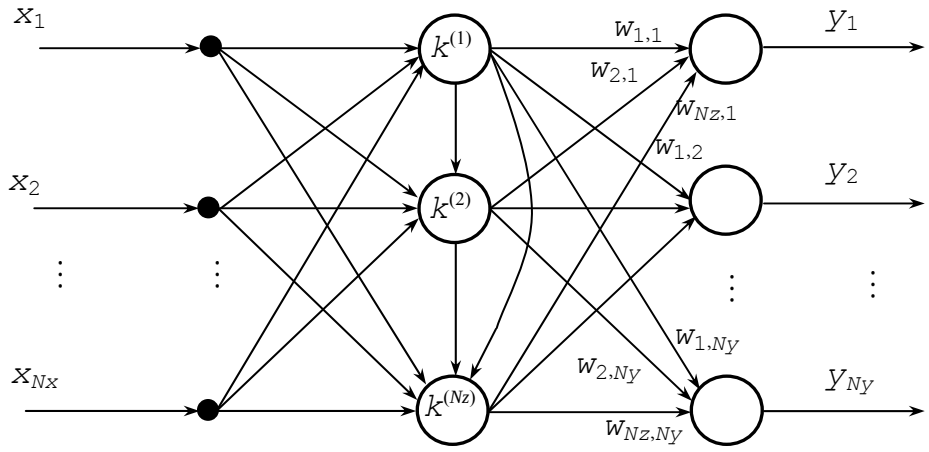


Abbildung 4.3 Struktur eines Booleschen Neuronalen Netzes
mit vorwärts gerichteten und lateralen Verbindungen

Das Boolesche Neuronale Netz hat eine verborgene Schicht. Die Ausgabesignale der verborgenen Booleschen Neuronen werden durch die Transferfunktionen dieser Neuronen gebildet. Die Neuronenanzahl in der verborgenen Schicht ist gleich der Anzahl der Trainingszyklen.
Wie in Abbildung 4.3 gezeigt, hat die allgemeine Struktur eines Booleschen Neuronalen Netzes außer den üblichen vorwärts gerichteten Verbindungen zwischen den Neuronen von Nachbarschichten, auch laterale Verbindungen zwischen den Neuronen in der verborgenen Schicht. Die Notwendigkeit zur Einführung solcher Verbindungen folgt aus (4.19). Der linke Teil von (4.19) gibt die vorwärts gerichteten Verbindungen an und der rechte Teil mit Koeffizienten beschreibt die lateralen Verbindungen in der verborgenen Schicht.
Das Vorhandensein lateraler Verbindungen kann als Nachteil der entwickelten BNN angesehen werden, weil die Berechnungsgeschwindigkeit in der Arbeitsphase des BNN reduziert wird. Der Effekt der Parallelverarbeitung der Information in Neuronalen Netzen kann deshalb völlig oder teilweise verloren gehen. Grundlegende Eigenschaften der Boo-

leschen Logik ermöglichen es aber, dass Boolesche Neuronale Netze in der Arbeitsphase von lateralen Verbindungen befreit werden können. So wird dieser Mangel behoben. Zusätzlich kann die Gewichtsbestimmung der eliminierten lateralen Verbindungen entfallen. Die in Sinne der Berechnungseffektivität verbesserte Struktur ist in Abbildung 4.4 angegeben.

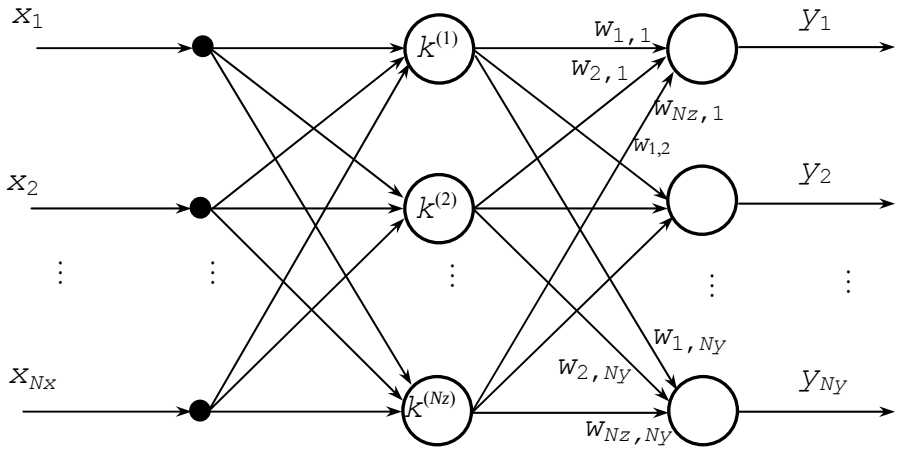


Abbildung 4.4 Struktur Boolesches Neuronalen Netzes ohne lateralen Verbindungen

Um diese Befreiung der BNN - Struktur von lateralen Verbindungen zu erklären, wird im folgenden Abschnitt ein Algorithmus zur Verwendung eines vorher trainierten Booleschen Neuronalen Netzes beschrieben.

4.2.3 Arbeitsweise

Bei der Verwendung von Booleschen Neuronalen Netzen für die Darstellung der Funktionsmenge werden die Eingabesignale auf die Netzeingänge gegeben. Die Ausgabesignale, die ein BNN an seinen Ausgängen erzeugt, sollen den Werten Boolescher Funktionen aus der zu modellierenden Funktionsmenge entsprechen. Der Nutzungsalgorithmus des BNN besteht aus folgenden Schritten.

1. Für einen gegebenen Vektor der Eingabesignale werden die Funktionswerte aller N_z Transferfunktionen der verborgenen Neuronen berechnet.

Da alle Funktionen $\{k^{(1)}, k^{(2)}, ..., k^{(Nz)}\}$ während des Trainingsalgorithmus durch ihre Wertetabelle definiert wurden, und zwar nur in Abhängigkeiten von Eingabesignalen $(x_1, x_2, ..., x_{Nx})$, können sie gleichzeitig und unabhängig von einander berechnet werden. Die Formel (4.20) beschreibt also Restriktionen zwischen den *K*-Funktionen. Da die *K*-Funktionen nur von den Eingangsvariablen abhängen, kann Struktur des BNN auf laterale Verbindungen zwischen den Neuronen in einer verborgenen Schicht verzichten.

2. Formel (4.17) berechnet die Ausgabesignale.

Bei Bedarf können nur die aus der zu modellierenden Funktionsmenge ausgewählten Funktionen (und nicht alle N_Y Funktionen) berechnet werden.
Da die Neuronen der verborgenen Schicht eine bedeutende Rolle bei der Informationsbearbeitung spielen, erfüllen die Booleschen Elemente der Eingans- und Ausgangsschicht nur eine Nebenaufgabe. Durch die Booleschen Neuronen der Ausgabeschicht wird (4.17) realisiert. Die Neuronenanzahl in der Ausgabeschicht ist gleich der Anzahl der zu modellierenden Funktionen. Die Anzahl von Booleschen Neuronen in der Eingangsschicht entspricht der Anzahl von Eingangsvariablen, von denen die zu modellierenden Booleschen Funktionen abhängen. Die Eingangsschicht des BNN kann als „Pseudoschicht" von Booleschen Neuronen bezeichnet werden, da nur die Eingabesignale auf die Eingänge der Booleschen Neuronen der verborgenen Schicht verteilt werden.

4.2.4 Eigenschaften

Boolesche Neuronale Netze gehören zur Klasse der vorwärts gerichteten Netze (feed forward neural networks). Der Trainingsalgorithmus des BNN ist ein typischer sequentieller Trainingsalgorithmus, in dessen Verlauf verborgene Neuronen gebildet werden. In unserem Fall wird ein 3-schichtiges Boolesches Neuronales Netz verwendet, d.h. BNN hat eine Eingangs-, eine Ausgangs- und eine verborgene Schicht. Dabei ist die Neuronenanzahl in der Eingangsschicht gleich der Anzahl der Variablen der zu modellierenden Booleschen Funktionen. Die Neuronenanzahl in der Ausgangsschicht des BNN ist gleich der Anzahl von Booleschen Funktionen.
Zu beachten ist, dass es zwischen den Booleschen Neuronen der verborgenen Schicht und den Booleschen Neuronen der Ausgangsschicht des Netzes einen Unterschied gibt. Jedes Boolesche Neuron in der verborgenen Schicht hat eine eigene Transferfunktion, die sich von der Transferfunktion aller anderen Booleschen Neuronen in dieser Schicht unterscheidet. Diese Transferfunktion eines Booleschen Neurons wird durch alle Eingangsvariablen oder eine Teilmenge von Eingangsvariablen definiert. Die konkrete Transferfunktion wird während des Trainingsprozesses des Netzes bestimmt. Ein z–tes Boolesches Neuron in der verborgenen Schicht eines Booleschen Neuronalen Netzes wird durch (4.21) beschrieben.

$$y^{(z)} = f^{(z)}(\boldsymbol{x}, \boldsymbol{w}_h{}^{(z)}) \tag{4.21}$$

Dabei gilt folgende Bedingung:

$$f^{(m)} \neq f^{(n)} : \quad \forall\, m \neq n; \quad m,n \in [1, N_z] \tag{4.22}$$

wobei gilt:

z - Nummer des verborgenen Booleschen Neurons, $z = 1, \ldots, N_z$,

N_z - Anzahl der Booleschen Neuronen in der verborgenen Schicht,

$y^{(z)}$ - Ausgangssignal des z–ten verborgenen Booleschen Neurons,

$f^{(z)}$ - Transferfunktion des z–ten Booleschen Neurons,

$\boldsymbol{x} = \{x_1, x_2, \ldots, x_{Nx}\}$ - Vektor der Eingangssignale des BNN,

$\boldsymbol{w}_h{}^{(z)} = \{w_{h1}{}^{(z)}, w_{h2}{}^{(z)}, \ldots, w_{hNx}{}^{(z)}\}$ – Gewichtsvektor des z–ten verborgenen Booleschen Neurons.

Die mathematische Beschreibung eines Booleschen Neurons der Ausgangsschicht bleibt im Vergleich zu (4.21) unverändert. Nur die Bedingung (4.22) wird in (4.23) umgewandelt.

$$y^{(j)} = f^{(j)}(\boldsymbol{k}^{(j)}, \boldsymbol{w}_o{}^{(j)})$$

$$f^{(m)} = f^{(n)}: \ \forall\, m \neq n; \quad m,n \in [1, N_y] \tag{4.23}$$

wobei gilt:

j - Nummer des Booleschen Neurons in der Ausgabeschicht, $j = 1, \ldots, N_y$, N_Y - Anzahl der zu modellierenden Booleschen Funktionen, die gleich mit der Neuronenanzahl in der Ausgabeschicht des BNN ist,

$y^{(j)}$ - Ausgangssignal des j–ten Booleschen Neurons in der Ausgabeschicht,

$f^{(j)}$ - Transferfunktion des j–ten Booleschen Neurons in der Ausgabeschicht,

$\boldsymbol{k} = \{k^{(1)}, k^{(2)}, \ldots, k^{(Nz)}\}$ -Vektor der Eingangssignale der Ausgabeschicht des BNN,

$\boldsymbol{w}_o{}^{(j)} = \{w_{o1}{}^{(j)}, w_{o2}{}^{(j)}, \ldots, w_{oNz}{}^{(j)}\}$ – Gewichtsvektor des j–ten Neurons in der Ausgabeschicht.

Alle Neuronen in der Ausgabeschicht haben eine festgelegte Boolesche Transferfunktion. Diese Transferfunktion verknüpft die gewichteten Eingangssignale des Neurons mit einer für alle Ausgangsneuronen gleichen Booleschen Operation, wie z.B. "Konjunktion" (AND), "Disjunktion" (OR), "Äquivalenz" (EXAND) oder "Antivalenz" (EXOR).
Der Trainingsprozess eines Booleschen Neuronalen Netzes übernimmt die angegebene Boolesche Operation für Transferfunktionen aller Neuronen der Ausgangsschicht. Durch diese Boolesche Operation, die auch **Basisoperation** genannt wird, wird ein Kumulationsoperator **Ω** der Ausgangsneuronen definiert. Wie noch gezeigt wird, bestimmt diese Basisoperation auch den Typ einer Dekomposition der Menge Boolescher Funktionen.

4.2.5 Boolesche Neuronale Netze für AND-, OR-, EXOR- und Äquivalenz-Dekomposition Boolescher Funktionsmengen

Im vorherigen Abschnitt wurde ein Beispiel des Booleschen Neuronalen Netzes beschrieben, wobei die Antivalenz von Eingangssignalen als Basisoperation für die Neuronen in der Ausgangsschicht definiert war. Jetzt werden BNN mit den Basisoperationen „AND", „OR" und „EXAND" betrachtet. Dabei werden die Transferfunktionen von

Booleschen Neuronen der verborgenen Schicht und die Gewichte Boolescher Neuronen in der Ausgangsschicht in anderer Weise berechnet. Die Hauptidee der Verwendung der BNN-Technologie für die Zerlegung einer Menge Boolescher Funktionen liegt im Aufbau des BNN, dessen Neuronen in der Ausgangsschicht für die Basisoperation eine vorher bestimmte Boolesche Operation benutzen. Für die AND-Zerlegung einer Booleschen Funktionsmenge wird die Operation „AND" verwendet [79].

BNN mit AND-Dekomposition. Eine Lernmenge für das zu konstruierende BNN ist wie im vorherigen Beispiel eine Matrix $\boldsymbol{A}$ (4.9), die durch die volle Wertetabelle der zu modellierenden Menge Boolescher Funktionen oder durch eine Untermenge von Zeilen dieser Wertetabelle bestimmt wird.
Hier wird der Trainingsalgorithmus für die volle Wertetabelle betrachtet. Jede Zeile der Matrix $\boldsymbol{A}$ hat einen Koeffizienten m, und jede Spalte hat einen Koeffizienten n. Die Koeffizienten m und n werden auch Gewichte genannt und durch (4.24) und (4.25) berechnet.

$$m_i = \sum_{j=1}^{N_y} a_{i,j}. \tag{4.24}$$

$$n_j = \sum_{i=1}^{2^{N_x}} a_{i,j} \tag{4.25}$$

Gilt $\forall\ i,\ m_i = N_y$, so wird der Algorithmus beendet.
Andernfalls werden die weiteren Schritte des Algorithmus ausgeführt. Im folgenden Schritt wird ein Grundzeilenvektor $\boldsymbol{v}$ aus der Matrix $\boldsymbol{A}$ des Trainingsprozesses gewählt. Dafür sind zwei zusätzliche Funktionen zu benutzen, die zuerst beschrieben werden.
Die Funktion

$$\mathrm{min}(\boldsymbol{N}) \tag{4.26}$$

sucht in der Menge $\boldsymbol{N}$ alle minimalen Elemente $\{min_1, min_2, \ldots\}$ und gibt eine Menge $\boldsymbol{N}_I=\{I_{min1}, I_{min2}, \ldots\}$ der Indizes von den gefundenen minimalen Elementen zurück.
Die zweite Funktion

$$\mathrm{Imax}(\boldsymbol{S}) \tag{4.27}$$

bekommt als Argument eine Menge $\boldsymbol{S} = \{(s_1, Is_1), (s_2, Is_2), \ldots,(s_i, Is_i), \ldots\}$ der zugeordneten Paare (s_i,Is_i). Die Funktion sucht ein maximales Element s_{max} und gibt den Wert Is_{max} zurück, der dem gefundenen s_{max} zugeordnet ist.
Um die Grundzeile zu finden, wird (4.26) für $\boldsymbol{N}=\{n_1, n_2, \ldots, n_{2^{Nx}}\}$ benutzt. Gibt es nur ein minimales Element in der Menge $\boldsymbol{N}$, besteht die Menge $\boldsymbol{N}_I=\{I_{min1}\}$ aus einem Element I_{min1}, und die Nummer der Grundzeile

$$I_v = I_{min1} \tag{4.28}$$

Sonst ist $\boldsymbol{N}_I=\{I_{min1}, I_{min2}, \ldots\}$, und die Nummer der Grundzeile I_v wird durch (4.29) bestimmt

$$I_v = \mathrm{Imax}(\boldsymbol{S}) \tag{4.29}$$

wobei $Is_1=I_{min1}$, $Is_2=I_{min2}$, ..., $Is_i=I_{mini}$, ...
und die Werte s_i durch (4.30) berechnet werden.

$$s_i = \sum_{j=1}^{N_y} \left(n_j \; a_{Is_i,j}\right) \tag{4.30}$$

Die Elemente des Grundzeilenvektors $\boldsymbol{v}$ der Matrix $\boldsymbol{A}$ werden als negierte Werte der Zeile mit der Nummer I_v durch (4.31) definiert.

$$v_j = \overline{a_{I_v,j}} \tag{4.31}$$

Die Werte der Teilfunktion k werden durch (4.32) berechnet.

$$k_i = \bigvee_{j \in N_j} a_{i,j} \tag{4.32}$$

wobei gilt: $\boldsymbol{N}_j=\{I_{max1}, I_{max2}, \ldots\}$ - eine Menge von Spaltennummern der Matrix $\boldsymbol{A}$, die den Nummern von maximalen Elementen des Gewichtsvektors $\boldsymbol{n}$ gleich sind. Dabei sind die entsprechenden Elemente des Grundzeilenvektors $v_i = 1$, $\forall\ i \in \boldsymbol{N}_j$.

Wurden die Elemente der Spalte mit der Nummer $i \in \boldsymbol{N}_j$ im Verlauf vorheriger Zyklen des Lernverfahrens geändert, darf diese Spalte in der gegenwärtigen Matrix durch die Spalte der primären Matrix $\boldsymbol{A}$ mit der Nummer i ersetzt werden. Dieses Ersetzen hat zum Ziel, das Gewicht des Vektors $\boldsymbol{k}$ zu minimieren, wobei das Gewicht des Vektors $\boldsymbol{k}$ die Anzahl seiner Einswerte ist.
Anschließend wird durch (4.33) der Gewichtsvektor für die Neuronen der Ausgangsschicht des BNN berechnet.

$$w_j = v_j \wedge \left(\bigwedge_{i \in N_{k0}} \overline{a_{i,j}} \right) \tag{4.33}$$

wobei gilt: $\boldsymbol{N}_{k0}$ - Menge von Zeilennummern $i \in \boldsymbol{N}_{k0}$, für die $k_i = 0$.
Als nächstes werden neue Werte der Matrix $\boldsymbol{A}$ durch (4.34) berechnet.

$$a_{i,j}^{(z+1)} = \begin{cases} a_{i,j}^{(z)}, & w_j = 0 \\ a_{i,j}^{(z)} \odot k_i, & w_j = 1 \end{cases} \tag{4.34}$$

wobei gilt: $a_{i,j}^{(z+1)}$ und $a_{i,j}^{(z)}$ - Elemente der Matrix $\boldsymbol{A}$ nach dem $z+1$–ten und dem z–ten Trainingszyklus.

Die folgenden Trainingszyklen beginnen jeweils mit der Bestimmung der Zeilengewichte durch (4.24).

BNN mit OR-Dekomposition. Die Lernmenge ist wieder eine volle Wertetabelle der Menge der zu modellierenden Booleschen Funktionen. Die Gewichte für jede Zeile und jede Spalte der Matrix $\boldsymbol{A}$ werden durch die Formel (4.24) und (4.25) berechnet. Man prüft die Bedingung, ob $\forall\ i: m_i = 0$. Wenn ja, dann wird der Algorithmus beendet. Ansonsten wird der Algorithmus mit der Berechnung eines Grundzeilenvektor ähnlich wie bei der AND-Dekomposition fortgesetzt.

Dafür werden wieder zwei zusätzliche Funktionen benötigt, die zuerst beschrieben werden.

Die Funktion

$$\mathrm{max}(\boldsymbol{N}) \tag{4.35}$$

sucht in der Menge $\boldsymbol{N}$ alle maximalen Elemente $\{max_1, max_2, \ldots\}$ und gibt eine Menge $\boldsymbol{N}_I=\{I_{max1}, I_{max2}, \ldots\}$ der Indizes von den gefundenen maximalen Elementen zurück.

Die zweite Funktion

$$\mathrm{Imin}(\boldsymbol{S}) \tag{4.36}$$

bekommt als Argument eine Menge $\boldsymbol{S} = \{(s_1,Is_1), (s_2,Is_2), \ldots,(s_i,Is_i), \ldots\}$ von Paaren (s_i,Is_i). Die Funktion sucht ein minimales Element s_{min} und gibt den Wert Is_{min} zurück, der dem gefundenen s_{min} zugeordnet ist. Die Nummer des Grundzeilenvektors

$$I_v = I_{max1} \tag{4.37}$$

wenn die durch (4.35) ermittelte Menge $\boldsymbol{N}_I =\{I_{max1}\}$ nur aus einem Element I_{max1} besteht. Sonst ist $\boldsymbol{N}_I=\{I_{max1}, I_{max2}, \ldots\}$ und die Nummer des Grundzeilenvektors wird durch (4.36) bestimmt.

$$I_v = \mathrm{Imin}(\boldsymbol{S}), \tag{4.38}$$

wobei gilt: $Is_1=I_{max1}$, $Is_2=I_{max2}$, …, $Is_i=I_{maxi}$, …

und die Werte s_i werden durch (4.39) berechnet.

$$s_i = \sum_{j=1}^{N_y} \left(n_j\ a_{Is_i,j}\right) \tag{4.39}$$

Die Elemente des Grundzeilenvektors $\boldsymbol{v}$ sind gleich den Elementen der Zeile a_{Iv}.

$$v_j = a_{Iv,j} \tag{4.40}$$

Die Elementswerte des Spaltenvektors $\boldsymbol{k}$ werden durch (4.41) berechnet.

$$k_i = \bigwedge_{j \in N_j} a_{i,j} \tag{4.41}$$

wobei gilt: $\boldsymbol{N}_j = \{I_{min1}, I_{min2}, \ldots\}$ - eine Menge von Spaltennummern der Matrix $\boldsymbol{A}$, die den Nummern von minimalen Elementen des Gewichtsvektors $\boldsymbol{n}$ gleich sind. Dabei sind die entsprechenden Elemente des Grundzeilenvektors $v_i = 1, \ \forall \ i \in \boldsymbol{N}_j$.

Wenn die Elemente der Spalte mit Nummer $i \in \boldsymbol{N}_j$ im Verlauf vorheriger Zyklen des Lernverfahrens geändert wurden, darf diese Spalte in der gegenwärtigen Matrix $\boldsymbol{A}$ durch Spalte der primären Matrix $\boldsymbol{A}$ mit derselben Nummer i ersetzt werden. Dabei wird das Gewicht des Vektors $\boldsymbol{k}$ maximiert.

Danach berechnet man durch (4.42) die Gewichte für Neuronen der Ausgangsschicht des BNN.

$$w_j = v_j \wedge \left(\bigwedge_{i \in N_{k1}} a_{i,j} \right) \tag{4.42}$$

wobei gilt: $\boldsymbol{N}_{k1}$ - Menge von Zeilennummern $i \in \boldsymbol{N}_{k1}$, für die $k_i = 1$.

Als nächstes werden neue Werte der Matrix $\boldsymbol{A}$ durch (4.43) berechnet.

$$a_{i,j}^{(z+1)} = \begin{cases} a_{i,j}^{(z)}, & w_j = 0 \\ a_{i,j}^{(z)} \oplus k_i, & w_j = 1 \end{cases} \tag{4.43}$$

Die folgenden Trainingszyklen beginnen jeweils mit der Bestimmung der Zeilengewichte durch (4.24).

BNN mit EXOR-Dekomposition. Das Training eines BNN mit EXOR als Grundoperation ist dem Trainingsalgorithmus des BNN mit OR-Grundoperation sehr ähnlich. Die Wertetabelle einer Booleschen Funktionsmenge bildet eine Lernmenge $\boldsymbol{A}$ für das Netz. Durch die Formeln (4.24) und (4.25) werden die Gewichte für jede Zeile und jede Spalte der Matrix $\boldsymbol{A}$ berechnet. Man prüft die Bedingung, ob $\forall \ i, m_i = 0$. Wenn ja, dann wird der Algorithmus beendet. Sonst werden die Berechnungen im Algorithmus weiter fortgesetzt und ein Grundzeilenvektor $\boldsymbol{v}$ wird genau so wie bei der OR-Dekomposition bestimmt. Elementswerte des Spaltenvektors $\boldsymbol{k}$ werden durch (4.41) berechnet. Wenn aber eine Spalte der Matrix $\boldsymbol{A}$ mit Nummer $i \in \boldsymbol{N}_j$ in vorherigen Zyklen des Lernverfahrens geändert wurde, darf diese Spalte nicht in der aktuellen Matrix durch die Spalte der primären Matrix $\boldsymbol{A}$ mit derselben Nummer i ersetzt werden. Für das OR-Netz war es zulässig,

weil Disjunktion eines Eins mit einem beliebigen Signal laut Gesetz der „Verknüpfung mit Eins“ ist immer 1. Für die EXOR-Operation ist eine Antivalenz eines Eins mit einem beliebigen Signal a ein Komplement dieses Signals - $\bar{a}$ [130].
Die Gewichtsbestimmung für die Neuronen der Ausgangsschicht des BNN wird etwa komplizierter als bei den vorherigen BNN durchgeführt. Dafür werden die Anzahl der Einsen $S_j^{(1)}$ und die Anzahl der Nullen $S_j^{(0)}$ für jede Matrixspalte, für die $i \in \boldsymbol{N}_{k1}$, berechnet.

$$S_j^{(1)} = \sum_{i \in N_{k1}} a_{i,j}, \tag{4.44}$$

$$S_j^{(0)} = \left|N_{k1}\right| - S_1. \tag{4.45}$$

wobei gilt: $|\boldsymbol{N}_{k1}|$ - die Anzahl der Element der Menge $\boldsymbol{N}_{k1}$.

Wenn $S_j^{(1)} \neq S_j^{(0)}$, dann wird w_j durch (4.46) festgelegt.

$$w_j = \begin{cases} 1, & S_j^{(1)} > S_j^{(0)} \\ 0, & S_j^{(1)} < S_j^{(0)} \end{cases}, \tag{4.46}$$

In Formeln (4.44) und (4.45) wird die Anzahl von Einselementen und von Nullelementen in der Wertetabelle jeder Funktion berechnet, die bei dem entsprechenden Gewicht w_j=1 negiert werden. Es folgt aus dem Gesetz der „Verknüpfung mit Eins“, weil eine Antivalenz eines Eins mit einem beliebigen Signal a ein Komplement dieses Signals - $\bar{a}$ ist [130]. Da der Trainingsalgorithmus beendet wird, wenn $\forall\, i\colon\ m_i = 0$, wird das Gewicht durch (4.46) definiert, so dass möglichst mehr Elementen in der Wertetabelle der Funktionsmenge in 0 geändert werden und somit der Algorithmus möglichst schnell beendet wird.
Wenn aber $S_j^{(1)} = S_j^{(0)}$, ist (4.46) wenig effektiv und deshalb werden die zusätzlichen Summen $S_j^{(11)}$ und $S_j^{(00)}$ von Produkten der Zeilengewichte der Matrix $\boldsymbol{A}$ und der Elemente in den Spalten berechnet:

$$S_j^{(11)} = \sum_{i \in N_{k1}} \left(m_i * a_{i,j}\right) \tag{4.47}$$

$$S_j^{(00)} = \sum_{i \in N_{k1}} \left(m_i * \overline{a_{i,j}}\right). \tag{4.48}$$

Die Gewichtswerte für die Neuronen der Ausgangsschicht des EXOR-BNN werden durch (4.49) definiert.

$$w_j = \begin{cases} 1, & S_j^{(11)} > S_j^{(00)} \\ 0, & S_j^{(11)} \le S_j^{(00)} \end{cases} \tag{4.49}$$

Die Idee besteht in der Verminderung der Anzahl von Einswerten in der Matrixzeilen mit Nummern $i \in \boldsymbol{N}_{k1}$, wenn das Gesamtgewicht der Zeilen mit $a_{i,j}=1$, $i \in \boldsymbol{N}_{k1}$ größer als das Gesamtgewicht der Zeilen mit $a_{i,j}=0$, $i \in \boldsymbol{N}_{k1}$.
Als nächstes werden neue Werte der Matrix $\boldsymbol{A}$ durch (4.43) berechnet und die folgenden Trainingszyklen mit der Bestimmung der Zeilengewichte durch (4.24) fortgesetzt.

BNN mit Äquivalenz-Dekomposition. Eine Äquivalenz-Dekomposition kann ähnlich zu einer EXOR-Dekomposition durchgeführt werden. Mit Anwendung der Sätze von de Morgan wird jeder EXOR-Ausdruck in einen Äquivalenz-Ausdruck transformiert. Demnach können alle Booleschen Funktionen aus der Lernmenge $\boldsymbol{A}$ zunächst negiert werden, dann wird der beschriebene Trainingsalgorithmus für ein EXOR-Netz durchgeführt. Die so erhaltenen Transferfunktionen k für die Neuronen in der verborgenen Schicht werden wieder negiert [78]-[79].

Rekonstruktion der zerlegten Booleschen Funktionen. In der Arbeitsphase des Booleschen Neuronalen Netzes werden die ursprünglichen Funktionswerte aus den Funktionswerten der Neuronen der verborgen Schicht zurück gewonnen.. Für die Arbeitsphase des trainierten BNN kann der Nutzungsalgorithmus verwendet werden, der im Abschnitt 4.2.3 allgemein beschrieben wurde. Hier wird im zweiten Schritt des Algorithmus statt (4.17) folgende Formel (4.50) verwendet.

$$a_{i,j}^{(1)} = \underset{z=1}{\overset{Nz}{\Omega}} \left(k_i^{(z)} \wedge w_{z,j} \right), \tag{4.50}$$

wobei gilt: $\Omega \in \{\wedge, \vee, \oplus, \odot\}$.
Der Kumulationsoperator $\boldsymbol{\Omega}$ der Ausgangsneuronen wird in Abhängigkeit von der Art der Zerlegung gewählt.

4.2.6 Beispiel zur Dekomposition einer Funktionsmenge

In diesem Abschnitt wird ein Beispiel zur EXOR-Dekomposition einer Funktionsmenge beschrieben. Ein Beispiel zur OR-Dekomposition einer Funktionsmenge ist im Anhang A.1 enthalten.
Zuerst wird der Trainingsalgorithmus eines BNN durchgeführt, wofür ein BNN mit EXOR-Dekomposition ausgewählt wurde. Im Verlauf des Trainingsalgorithmus wird eine

Struktur des BNN synthetisiert, dessen Korrektheit durch den Nutzungsalgorithmus des BNN in der Arbeitsphase überprüft wird.

Zu zerlegen ist eine Menge von N_y=10 Booleschen Funktionen $y_1, y_2, ..., y_{10}$, $y_i = f(x_1, x_2, x_3)$. Die Wertetabelle dieser Booleschen Funktionen wird in Tabelle 4.4 angeführt.

Tabelle 4.4 Wertetabelle von 10 Booleschen Funktionen

x_1	x_2	x_3	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
0	0	0	0	0	1	1	0	0	1	1	1	0
0	0	1	1	1	1	0	0	1	0	1	1	0
0	1	0	0	0	1	0	1	1	0	0	1	1
0	1	1	1	1	0	1	0	1	1	0	0	0
1	0	0	0	1	0	0	1	0	1	1	1	0
1	0	1	1	1	1	1	1	0	1	0	1	1
1	1	0	1	1	0	1	0	1	1	0	0	0
1	1	1	0	0	1	0	1	1	0	0	1	1

Das Netz hat 10 Ausgänge für 10 Boolesche Funktionen und folglich besteht die Ausgabeschicht des Netzes aus 10 Booleschen Neuronen mit EXOR-Transferfunktion. Da die gegebenen Funktionen von 3 Booleschen Variablen abhängen N_x=3, besitzt das Netz 3 Eingänge, d.h. die Eingabeschicht besteht aus 3 Neuronen. Für die weitere Bestimmung der Netzstruktur und der anderen Parameter des Netzes wird ein Training durchgeführt.

Training. Der rechte Teil der Wertetabelle 4.4 dient als Lernmenge des Booleschen Neuronalen Netzes und wird durch die Matrix $\boldsymbol{A}$ (4.9) dargestellt. Der Algorithmus beginnt mit der Berechnung von Gewichtskoeffizienten des Vektors $\boldsymbol{m}$ für jede Zeile der Matrix $\boldsymbol{A}$ durch (4.24).

Tabelle 4.5 Anfangsmatrix $\boldsymbol{A}$

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	$\boldsymbol{m}$	$\boldsymbol{k}^{(1)}$
0	0	0	1	1	0	0	1	1	1	0	5	0
1	1	1	1	0	0	1	0	1	1	0	6	0
2	0	0	1	0	1	1	0	0	1	1	5	1
3	1	1	0	1	0	1	1	0	0	0	5	0
4	0	1	0	0	1	0	1	1	1	0	5	0
5	1	1	1	1	1	0	1	0	1	1	8	1
6	1	1	0	1	0	1	1	0	0	0	5	0
7	0	0	1	0	1	1	0	0	1	1	5	1
$\boldsymbol{n}$	4	5	5	4	4	5	5	3	6	3		
$\boldsymbol{v}$	1	1	1	1	1	0	1	0	1	1		
$\boldsymbol{w}_1$	0	0	1	0	1	1	0	0	1	1		
$S_j^{(1)}$	1	1	3	1	3	2	1	0	3	3		
$S_j^{(0)}$	2	2	0	2	0	1	2	3	0	0		

Solange Werte $m_i \neq 0$ existieren, wird der Algorithmus weitergeführt. Das Training wird mit der Berechnung von Gewichtskoeffizienten des Vektors $\boldsymbol{n}$ für jede Spalte der Matrix $\boldsymbol{A}$ durch (4.25) fortgesetzt. Zur Bestimmung des Grundzeilenvektors $\boldsymbol{v}$ wird eine Menge aller maximalen Elemente des Vektors $\boldsymbol{m}$ gesucht. Dieser Vektor $\boldsymbol{m}$ hat nur einen maximalen Elementswert 8, und $I_v = I_{max1} = 5$. Entsprechend (4.40) sind die Elementeswerte des Grundzeilenvektors $v_j = a_{5,j}$. Um die Werte des Vektors $\boldsymbol{k}^{(1)}$ zu berechen, sucht man die Menge $\boldsymbol{N}_j$. Dafür werden die Elemente des Gewichtsvektors $\boldsymbol{n}$ gewählt, für denen die die entsprechenden Elemente des Grundzeilenvektors $v_i = 1$ (unterstrichene Werte des Zeilenvektors $\boldsymbol{n}$). Unter dieser Werte (4, 5, 5, 4, 4, 5, 6, 3) findet man alle minimale Elemente. In diesem Fall besteht nur eins Wert, der dem minimalen Element - 3 gleich ist und somit Menge $\boldsymbol{N}_j = \{10\}$, wobei 10 – die Nummer des Elementes mit dem Wert 3 im Vektor $\boldsymbol{n}$ ist. Da die Menge $\boldsymbol{N}_j$ aus nur einem Element besteht, wird die letzte Spalte der Matrix $\boldsymbol{A}$ als Vektor $\boldsymbol{k}^{(1)}$ gewählt (4.41), der gleichzeitig eine Boolesche Transferfunktion des aktuellen verborgenen Neurons ist. Der Vektor $\boldsymbol{k}^{(1)}$ hat Einsen für die Elemente $\boldsymbol{N}_{k1} = \{2, 5, 7\}$. Zu beachten ist, dass die Nummerierung der Zeilen der Matrix $\boldsymbol{A}$ mit 0 beginnt. Demzufolge werden die Verbindungsgewichte für die Neuronen der Ausgangsschicht des BNN durch (4.44) - (4.49) bestimmt. Dafür werden die Anzahl der Einsen $S_j^{(1)}$ und die Anzahl der Nullen $S_j^{(0)}$ für jede Matrixspalte, für die $i \in \boldsymbol{N}_{k1}$, berechnet. Zum Beispiel für die erste Spalte $S_1^{(1)} = 1$ und $S_1^{(0)} = 2$, weil $a_{2,1} = 0$, $a_{5,1} = 1$, $a_{7,1} = 0$ und $|\boldsymbol{N}_{k1}| = 3$. Da $S_j^{(1)} \neq S_j^{(0)}$ $(1 \neq 2)$, dann $\boldsymbol{w}_{1,1} = 0$ nach (4.46). In Tabelle 4.5 sind diese Gewichte in einen Vektor $\boldsymbol{w}_1$ zusammengefasst.

Tabelle 4.6 Matrix $\boldsymbol{A}$ nach dem ersten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	$\boldsymbol{m}$	$\boldsymbol{k}^{(2)}$
0	0	0	1	1	0	0	1	1	1	0	5	1
1	1	1	1	0	0	1	0	1	1	0	6	1
2	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	1	0	1	1	0	0	0	5	0
4	0	1	0	0	1	0	1	1	1	0	5	0
5	1	1	0	1	0	1	1	0	0	0	5	0
6	1	1	0	1	0	1	1	0	0	0	5	0
7	0	0	0	0	0	0	0	0	0	0	0	0
$\boldsymbol{n}$	4	5	2	4	1	4	5	3	3	0		
$\boldsymbol{v}$	1	1	1	0	0	1	0	1	1	0		
$\boldsymbol{w}_2$	0	0	1	1	0	0	1	1	1	0		

Als nächstes werden die neuen Werte der Matrix $\boldsymbol{A}$ durch (4.43) berechnet und die folgenden Trainingszyklen mit der Bestimmung der Zeilengewichte durch (4.24) fortgesetzt (siehe Tabelle 4.6). Am Ende des zweiten Trainingszyklussees sind die weiteren Vektoren

$\boldsymbol{k}^{(2)}$ und $\boldsymbol{w}_2$ bestimmt, die in den Matrizen $\boldsymbol{K}$ und $\boldsymbol{W}$ in den Tabellen 4.10 und 4.11 aufgenommen werden. Die Resultate der Trainingsschritte 2, 3 und 4 werden in den Tabellen 4.7 – 4.9 dargestellt.

Tabelle 4.7 Matrix $\boldsymbol{A}$ nach dem zweiten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	$\boldsymbol{m}$	$\boldsymbol{k}^{(3)}$
0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	1	1	0	0	0	5	0
2	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	1	0	1	1	0	0	0	5	0
4	0	1	0	0	1	0	1	1	1	0	5	1
5	1	1	0	1	0	1	1	0	0	0	5	0
6	1	1	0	1	0	1	1	0	0	0	5	0
7	0	0	0	0	0	0	0	0	0	0	0	0
$\boldsymbol{n}$	4	5	0	4	1	4	5	1	1	0		
$\boldsymbol{v}$	0	1	0	0	1	0	1	1	1	0		
$\boldsymbol{w}_3$	0	1	0	0	1	0	1	1	1	0		

Tabelle 4.8 Matrix $\boldsymbol{A}$ nach dem dritten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	$\boldsymbol{m}$	$\boldsymbol{k}^{(4)}$
0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	1	1	0	0	0	5	1
2	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	1	0	1	1	0	0	0	5	1
4	0	0	0	0	0	0	0	0	0	0	0	0
5	1	1	0	1	0	1	1	0	0	0	5	1
6	1	1	0	1	0	1	1	0	0	0	5	1
7	0	0	0	0	0	0	0	0	0	0	0	0
$\boldsymbol{n}$	4	4	0	4	0	4	4	0	0	0		
$\boldsymbol{v}$	1	1	0	1	0	1	1	0	0	0		
$\boldsymbol{w}_4$	1	1	0	1	0	1	1	0	0	0		

Nach dem vierten Trainingszyklus sind alle Elemente des Vektors $\boldsymbol{m}$ sowie der Matrix $\boldsymbol{A}$ gleich Null geworden (siehe Tabelle 4.9). Somit ist das Training beendet.

Tabelle 4.9 Matrix $\boldsymbol{A}$ nach dem vierten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	$\boldsymbol{m}$
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0

Das Boolesche Neuronale Netz wurde trainiert und kann jetzt in der Arbeitsphase verwendet werden. Die erzeugte Netzstruktur wird in Abbildung 4.5 gezeigt. Die Anzahl der verborgenen Neuronen ist gleich der Anzahl der durchgeführten Trainingszyklen N_z=4. Jedes verborgene Neuron bildet seine Transferfunktion ab und liefert sein Ausgabesignal an die Eingänge der Ausgabeneuronen. Jedes der 10 Neuronen in der Ausgabeschicht gibt die Werte der modellierten Booleschen Funktionen aus.

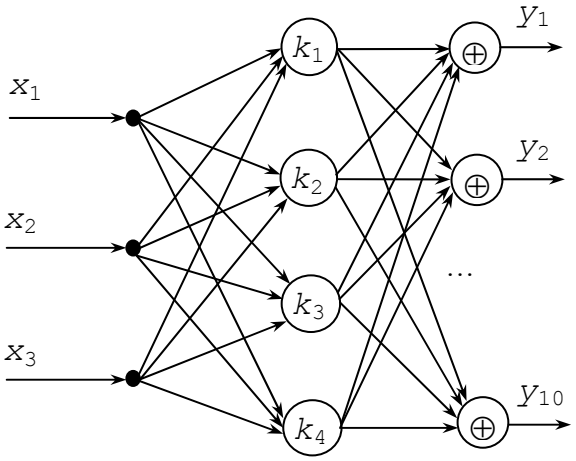


Abbildung 4.5 Struktur des EXOR-BNN

Zu den Ergebnissen des Trainingsalgorithmus gehören auch andere Parameter des Booleschen Netzes: die Transferfunktionen der vier verborgenen Neuronen (Tabelle 4.10) und die Verbindungsgewichte der zehn Neuronen in der Ausgangsschicht (Tabelle 4.11).

Tabelle 4.10 Transferfunktionen der verborgenen Booleschen Neuronen

x_1	x_2	x_3	$\boldsymbol{k}^{(1)}$	$\boldsymbol{k}^{(2)}$	$\boldsymbol{k}^{(3)}$	$\boldsymbol{k}^{(4)}$
0	0	0	0	1	0	0
0	0	1	0	1	0	1
0	1	0	1	0	0	0
0	1	1	0	0	0	1
1	0	0	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	0	0	1
1	1	1	1	0	0	0

Tabelle 4.11 Verbindungsgewichte der Ausgangsschicht

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
$\boldsymbol{w}_1$	0	0	1	0	1	1	0	0	1	1
$\boldsymbol{w}_2$	0	0	1	1	0	0	1	1	1	0
$\boldsymbol{w}_3$	0	1	0	0	1	0	1	1	1	0
$\boldsymbol{w}_4$	1	1	0	1	0	1	1	0	0	0

Ist das Gewicht gleich 1, existiert die Verbindung zwischen den entsprechenden verborgenen und Ausgabeneuronen. Ist das Gewicht gleich 0 existiert keine Verbindung, d.h. eine Eingabe des Ausgangsneurons fehlt. In diesem Fall kann jedes Ausgangsneuron maximal 4 Eingänge haben.
So wurden die 10 Funktionen $y_1, y_2, ..., y_{10}$ in vier Boolesche Teilfunktionen $k_1, k_2,\ldots, k_4$ durch EXOR-Dekomposition zerlegt.

Nutzung des trainierten BNN. Um die Korrektheit des Trainings nachzuweisen, werden die Ausgaben des Netzes für konkrete Eingangsbeispiele überprüft. Dazu wird der Nutzungsalgorithmus angewendet. Dieser besteht aus 2 Schritten. Zuerst werden die vier Transferfunktionen der verborgenen Neuronen für die gegebenen Eingangssignale x_1, x_2, x_3 parallel berechnet.

Tabelle 4.12 Wiederherstellung der Menge Boolescher Funktionen

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
$\boldsymbol{w}_1$	0	0	1	0	1	1	0	0	1	1
$\boldsymbol{w}_2$	0	0	1	1	0	0	1	1	1	0
$\boldsymbol{w}_3$	0	1	0	0	1	0	1	1	1	0
$\boldsymbol{w}_4$	1	1	0	1	0	1	1	0	0	0

	x_1	x_2	x_3	$\boldsymbol{k}^{(1)}$	$\boldsymbol{k}^{(2)}$	$\boldsymbol{k}^{(3)}$	$\boldsymbol{k}^{(4)}$
0	0	0	0	0	1	0	0
1	0	0	1	0	1	0	1
2	0	1	0	1	0	0	0
3	0	1	1	0	0	0	1
4	1	0	0	0	0	1	0
5	1	0	1	1	0	0	1
6	1	1	0	0	0	0	1
7	1	1	1	1	0	0	0

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
0	0	1	1	0	0	1	1	1	0
1	1	1	0	0	1	0	1	1	0
0	0	1	0	1	1	0	0	1	1
1	1	0	1	0	1	1	0	0	0
0	1	0	0	1	0	1	1	1	0
1	1	1	1	1	0	1	0	1	1
1	1	0	1	0	1	1	0	0	0
0	0	1	0	1	1	0	0	1	1

Danach werden die von den Neuronen auf der Ausgabeschicht erzeugten Booleschen Funktionen durch (4.50) berechnet. Als eine Testmenge werden alle möglichen Kombinationen aus Eingangssignalen x_1, x_2, x_3 verwendet. Die vollständige Beschreibung Arbeitsphase des Netzes wird in der Tabelle 4.12 aufgeführt.
Der Vergleich der Tabellen 4.4 und 4.12 zeigt, dass die EXOR-Dekomposition der zehn Booleschen Funktionen korrekt durchgeführt wurde. Das Boolesche Neuronale Netz kann zur Darstellung und EXOR-Zerlegung verwendet werden.

4.3 Mehrschichtige Boolesche Neuronale Netze

Die im vorangehenden Abschnitt betrachteten Varianten von Booleschen Neuronalen Netzen hatten in ihren Strukturen nur eine Schicht aus verborgenen Neuronen. Im Gebiet der üblichen Neuronalen Netze sind die Netze mit einer verborgenen Schicht (3-schichtige Neuronale Netze) in der Praxisanwendung stark eingeschränkt. Infolgedessen wurden die mehrschichtigen Neuronalen Netze (MLNN) entwickelt. Eine der bekanntesten Arten von mehrschichtigen Neuronalen Netzen ist ein mehrschichtiges Perzeptron, dessen zahlreiche Modifikationen sehr weit verbreitet und für verschiedene Zielsetzungen anwendbar sind. Diese Erfahrung nutzend liegt es nahe, mehrschichtige Boolesche Neuronale Netze (Multilayer Boolean neural networks - MLBNN) zu entwickeln. Die Methoden dafür können verschieden sein. Es wird vorgeschlagen, mit einem Ansatz zu beginnen, der sich an den Prinzipien der normaler sequentieller Neuronaler Netze orientiert.

4.3.1 Erweiterung der Ausgangsschicht

Die erste Methode zum Erzeugen eines mehrschichtigen BNN liegt in einer schrittweisen Transformation verborgener Neuronenschichten. Das Verfahren beginnt mit dem Training eines Booleschen Netzes mit einer verborgenen Schicht. Als Resultat bekommt man die Transferfunktionen Boolescher Neuronen in der verborgenen Schicht, die tatsächlich von den Eingabesignalen des Netzes abhängen. Dann gibt es zwei verschiedene Wege. Wie das folgende Beispiel zeigt ist die erste mögliche Variante wenig sinnvoll.

Beispiel. Es sei eine aus der Booleschen Funktionen y_1, y_2,..., y_{Ny} bestehende Funktionsmenge gegeben. Beim Training eines BNN mit beliebig gewählten Grundoperationen (z.B. wird OR gewählt) werden die Transferfunktionen der Neuronen einer verborgenen Schicht k_1, k_2,..., k_{Nk} und die Gewichte von Neuronen in der Ausgabeschicht bestimmt. Die Grundidee des nächsten Schritts liegt darin, dass die Ausgabe der Neuronen der ersten verborgenen Schicht als modifizierte Eingaben des nächsten Netzes angenommen werden können. Das zweite Netz soll eine Abbildung dieser Eingaben in die Ausgabe y_1, y_2,..., y_{Ny} realisieren. Dafür wiederholt sich das Lernverfahren genauso, wie für das erste BNN mit einer verborgenen Schicht. Dabei darf noch eine zusätzliche Eingabe mit einem konstanten Signal zu den modifizierten Eingaben hinzugefügt werden.
Die Anzahl verborgener Schichten ist theoretisch unbegrenzt. Nur werden die Ausgabesignale von Neuronen der verborgenen Schicht eines 3-schichtigen Netzes als die Eingaben des nächsten Netzes betrachtet. Die Eingangsanzahl vergrößert sich um Eins bei jedem Schritt, wenn ein zusätzlicher Eingang mit einem konstanten Signal verwendet wur-

de. Zurück zum Beispiel: Im 2. Schritt wird ein Netz mit der gleichen Grundoperation (hier OR) zum Training verwendet. Im allgemeinen Fall ist die Anzahl von k–Funktionen N_k größer als die Anzahl von Booleschen Variablen N_x, $\boldsymbol{x}=(x_1, x_2,..., x_{Nx})$, weil die Abhängigkeiten y_i von $\boldsymbol{x}$ beliebige Boolesche Funktionen und die Abhängigkeiten y_i von $\boldsymbol{k}$ - OR-Verknüpfungen sind. Infolgedessen besteht die Trainingsmatrix (Wertetabelle der Funktionsmenge) für das 2. Training aus mehr Zeilen als für das 1. Training. Im allgemeinen Fall, je größer die Trainingsmatrix desto mehr verborgenen Neuronen benötigt, um diese Matrix abzubilden. Daraus folgt, dass die Anzahl von h–Funktionen (4.51) größer als die Anzahl von k–Funktionen ist, $N_h > N_k$.

$$h_i = h^{(i)}(k_1, k_2,..., k_{Nk}), \quad i=(1, 2,..., N_h), \tag{4.51}$$

wobei $h^{(i)}$ – Transferfunktionen der verborgenen Neuronen des 2. Netzes ist.

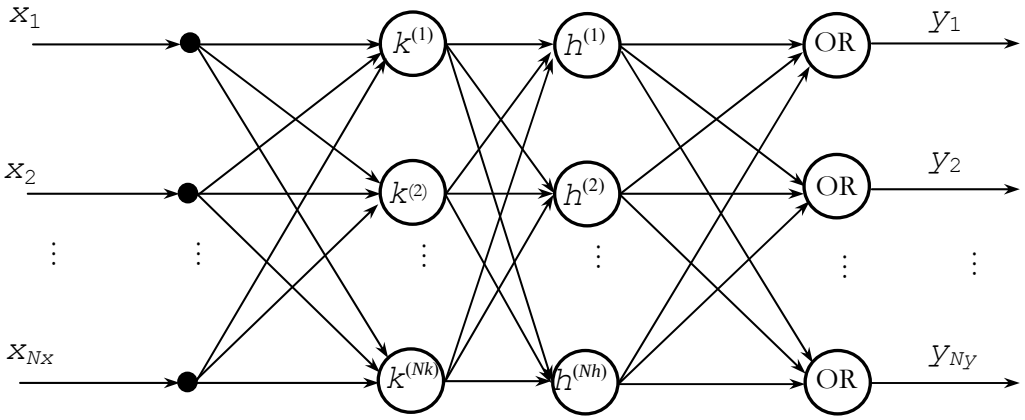


Abbildung 4.6 Struktur eines BNN nach dem OR-OR-Training

Eine Auswahl der selben Grundoperation für das 1. und 2. Training führt nur zur Vergrößerung der Anzahl von Neuronen im Netz und somit zur komplizierteren Struktur des Netzes. Eine entsprechende Struktur des BNN ist in Abbildung 4.6 dargestellt. Dabei gibt es keine Voraussetzung dafür, dass die Anzahl von Verbindungen und somit Anzahl von Eingängen in Neuronen vermindert wird. Ein solcher Ansatz bringt folglich für den Aufbau mehrschichtiger BNN keinen Nutzen.

Verwendet man im 2. Schritt ein Netz mit einer anderen Grundoperation (dies können im beschriebenen Fall AND, EXOR, oder Äquivalenz sein) zum Training, so unterscheiden sich in Resultat die Transferfunktionen der verborgenen Neuronen des 2. Netzes von denen des 1. Netzes. Die zugehörige Struktur des BNN ist in Abbildung 4.7 dargestellt.

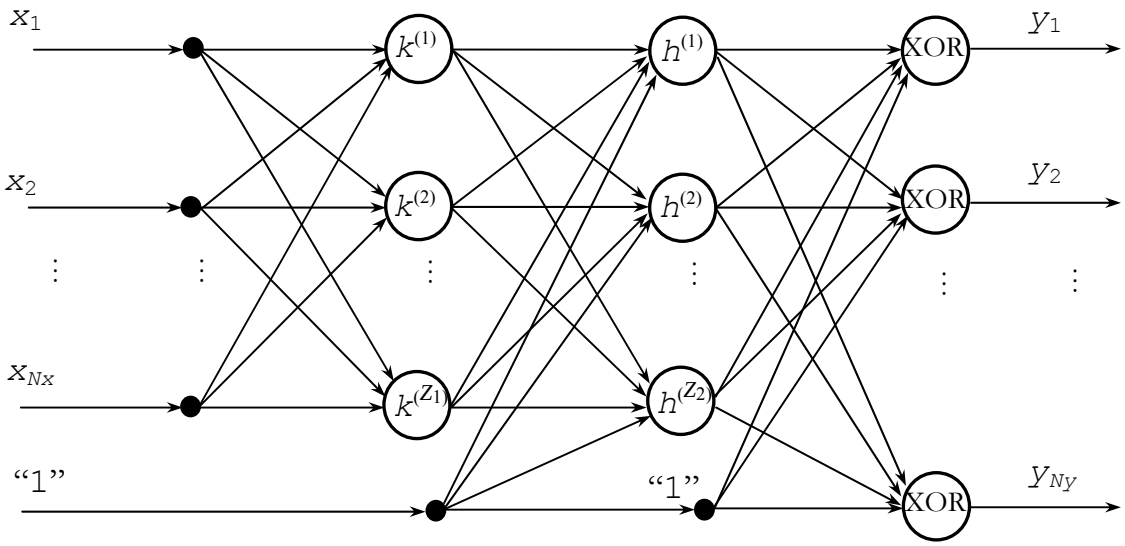


Abbildung 4.7 Struktur eines BNN nach dem OR-XOR-Training

Die Wahl des OR-XOR-BNN kann sinnvoll sein, wenn man eine mit einer Verknüpfung (OR) dargestellte Funktionsmenge durch eine andere Boolesche Verknüpfung (XOR) transformieren will. Im Sinne der Kompaktheit der Darstellung der gegebenen Menge Boolescher Funktionen bleibt aber eine solche Bildungsmethode von mehrschichtigen BNN wenig effektiv. Weil die Anzahl von Neuronen im Netz vergrößert und keine Verminderung der Anzahl von Eingängen in Neuronen garantiert wird, führt diese Aufbaumethode von mehrschichtigen BNN auch in diesem Fall zur komplizierteren gesamten Struktur des Netzes.

4.3.2 Erweiterung der verborgenen Schicht

In diesem Abschnitt wird eine andere Methode zur Entwicklung eines mehrschichtigen Booleschen Neuronalen Netzes vorgestellt, die auf dem schrittweisen Aufbau von Schichten verborgener Neuronen basiert. Wie in der vorangehenden Methode wird zunächst das Training eines BNN mit einer verborgenen Schicht durchgeführt. Dabei bilden die im Verlauf des Trainings erzeugten Transferfunktionen der verborgenen Neuronen k_1, k_2, ..., k_z eine Lernmenge für ein nachfolgendes Boolesches Netz, das im zweiten Schritt trainiert werden soll. Wie in Abbildung 4.7 gezeigt wird, dient der zweite Schritt zur Zerlegung der in erstem Schritt entstandenen Schicht von verborgenen Neuronen in zwei Schichten: Eine verborgene Schicht und eine Ausgabeschicht des zweiten BNN. Diese Methodik kann sowohl für die Vereinfachung der Transferfunktionen von verborgenen Neuronen als auch bei einer Beschränkung der Eingangsanzahl für die Neuronen verwendet werden. Obwohl die gesamte Zahl von Neuronen im Netz vergrößert werden kann, es können mehr Gewichten 0 sein.

Die Ausgabeschichten des ersten und zweiten Netzes können auch zu einer Schicht zusammengefügt werden. Dabei erhielt man wieder ein 3-schichtiges BNN. Es entsteht die Frage, welche Boolesche Operation als Grundoperation des nächsten BNN verwendet werden soll. Bei gleicher Grundoperation für beide Lernverfahren bekommt man nach Verbinden der Ausgabeschichten der beiden Netze ein 3-schichtiges BNN mit einer Eingangsschicht, einer verborgenen Schicht und linearen Neuronen mit der entsprechenden Grundoperation in der Ausgabeschicht.

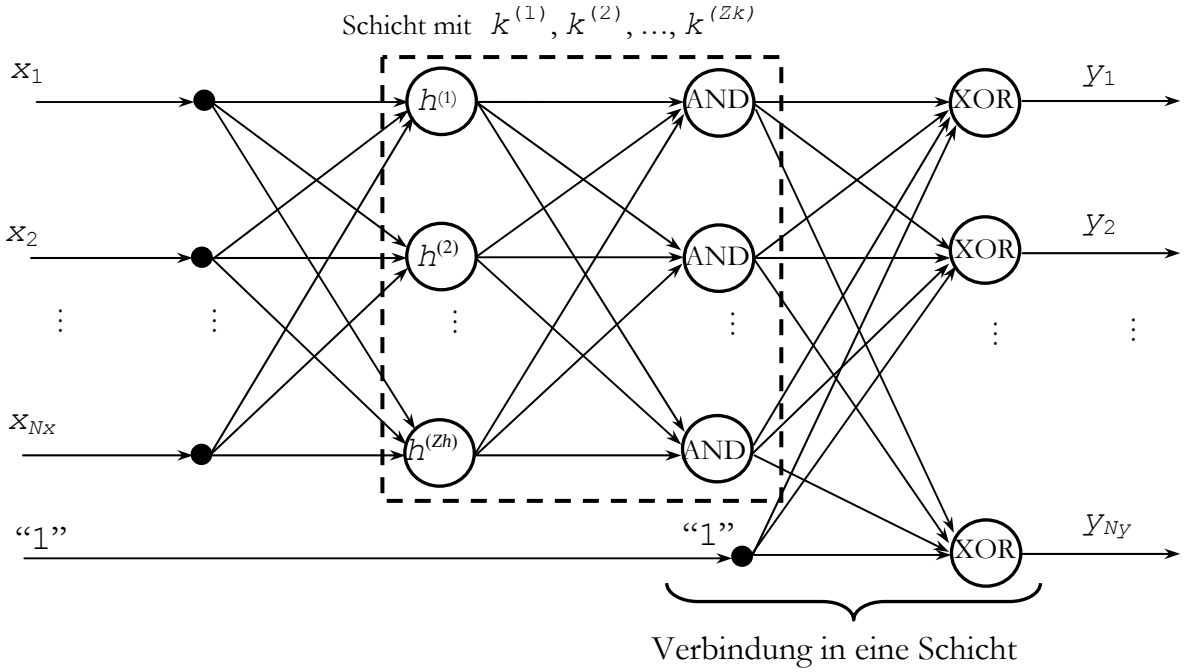


Abbildung 4.8 Struktur eines AND-XOR-BNN

Interessanter ist die Auswahl von verschiedenen Grundoperationen bei dem ersten und zweiten Training. In diesem Fall ist eine Ausgabeschicht aus unterschiedlichen Booleschen Neuronen das Resultat einer Substitution von Ausgabeschichten des ersten und zweiten Netzes. Die Transferfunktionen von Neuronen in dieser entstehenden Ausgangsschicht können beliebige Boolesche Funktionen sein. Es können die Transferfunktionen der Neuronen entweder aus einer verborgenen oder aus der Ausgabeschicht als neue Lernmenge betrachtet werden. Entsprechende Schichten werden immer wieder in eine zweischichtige Struktur zerlegt. Diese Transformationen können prinzipiell mehrfach wiederholt werden, da theoretisch keine Begrenzung existiert.
Als Empfehlung für die Auswahl der Grundoperationen werden folgende Operationsmengen vorgeschlagen.

Tabelle 4.13 Beispiel von Grundoperationen für ein BNN mit 2 verborgenen Schichten

Ausdrucksform	1. Operation	2. Operation
Disjunktive Form	OR	AND
Konjunktive Form	AND	OR
Antivalenzform	EXOR	AND
Äquivalenzform	Äquivalenz	OR

Diese Empfehlung basiert auf der Theorie Boolescher Normalformen. Es ist bekannt, dass jede Boolesche in einer der vier Grundformen aus der Tabelle 4.13 dargestellt werden kann. Diese Grundformen verwenden die 1. Operation als äußere Operation und die 2. Operation als inner Operation. In diesen vier Booleschen Grundformen kann auch die Negation benutzt werden. Eine Negation kann man mit der Antivalenz und Äquivalenz realisieren. Deshalb ist es sinnvoll, für die Schritte des EXOR- oder Äquivalenz-Trainings eine zusätzliche Eingabe mit einem entsprechenden konstanten Signal zu verwenden. In Abbildung 4.8 wird eine Netzstruktur des AND-XOR-BNN mit einem konstanten "1" - Signal dargestellt.

Kapitel 5

Hardware-Realisierung von Booleschen Neuronalen Netzen mit FPGA

5.1 Boolesche Neuronale Netze im FPGA

5.1.1 Abbildung eines Booleschen Neurons im FPGA

Im vorangehenden Kapitel wurden Boolesche Neuronale Netze als Modell Boolescher Funktionen betrachtet. Die vorgeschlagene BNNs sind in der Lage, die Booleschen Funktionen kompakt darzustellen und schnell zu berechnen. Die entwickelten Trainingsalgorithmen beschreiben eine neu Dekompositionsmethode für eine gegebene Menge Boolescher Funktionen. Bei der Dekomposition Boolescher Funktionen wird die künstliche Intelligenz von Booleschen Neuronalen Netzen verwendet, um sowohl eine hohe Kompaktheit der Darstellung als auch eine kurze Berechnungszeit zu erreichen. Im Kapitel 5 wird die Intelligenz von BNN auch zur kompakten Darstellung Boolescher Funktionen im FPGA (field-programmable gate arrays) ausgenutzt. Dabei wird eine Hardware-Realisierung von Booleschen Neuronalen Netzen in FPGA entwickelt.

Die überwiegende Mehrheit von künstlichen Neuronalen Netzen wird als ein ausführbares Programm auf einem Rechner realisiert, weil Software-NN im Vergleich zu Hardware-NN flexibler sind. Spezialisierte Hardware-Realisierungen von NN besitzen unter mehreren Aspekten viele Vorteile [158]. Diese basieren insbesondere aus dem Parallelismus, der Modularität und der dynamischen Anpassungsfähigkeit von NN sowie BNN in Hardware-Realisierungen [180]. Außerdem hat sich bei der weltweiten Nutzung von NN gezeigt, dass die höhere Geschwindigkeit und die niedrigeren Kosten der Hardware-NN wesentlich zu ihrem erfolgreichen Einsatz beitragen.

Wie bereits erläutert, können alle Hardware-NN in drei Kategorien eingeteilt werden: digital, analog und hybrid [158]. Die den Hauptgegenstand dieser Arbeit bildenden Boo-

leschen Neuronalen Netze sind besonders geeignet für digitale Hardware-Realisierungen, weil diese Netze nur mit Booleschen Daten arbeiten. Es gibt viele Technologien zur Realisierung von Neuronalen Netze in digitaler Hardware. Hardware-Realisierungen von Neuronalen Netzen in FPGA sind besonders verbreitet, weil sie viele Vorteile im Vergleich zu anderen Hardware-Realisierungen besitzen [111]. Die allgemein bevorzugte Hardware-Realisierung von NN als FPGA bietet eine Möglichkeit, die entwickelten Hardware-Realisierungen von Booleschen Neuronalen Netzen mit anderen bekannten FPGA-Realisierungen von Neuronalen Netzen zu vergleichen.

Bei der Hardware-Realisierung Neuronaler Netze in FPGA wird eine große Anzahl von CLB (configurable logic block) für die Abbildung eines Neurons benötigt. Ein Neuron wird folglich durch eine mehrstufige Struktur aus CLB gebildet, was zu einer Verlangsamung der Berechung führt. Es besteht also ein Abbildungsproblem für NN auf FPGA. Dutzende oder auch Hunderte von CLB können erforderlichen, um ein übliches Neuron eines Netzes im FPGA zu realisieren. Eine der bekannten FPGA-Realisierungen eines 3-schichtigen Feed Forward-Netzes GANGLION [34] braucht 640 – 784 CLB pro ein Neuron. Das in [63][177] [176] beschriebene Hopfield-Netz besteht aus 64 Neuronen, von denen jedes 26 CLB einer XC4000-Xilinx-FPGA-Karte benötigt. Ein gutes Ergebnis wurde von M. Gschwind in [64] erreicht. Dort sind nur 22 CLB zur Abbildung eines Neurons im FPGA erforderlich.

In der Abbildung 5.1 wird dargestellt, dass es 3 Ursachen für das Abbildungsproblem von Neuronen auf CLBs gibt. Dabei handelt es sich um:

- den Typ der Ein- und Ausgangsdaten des Neurons,
- die Kompliziertheit der Transferfunktion des Neurons,
- die Anzahl von Eingängen des Neurons.

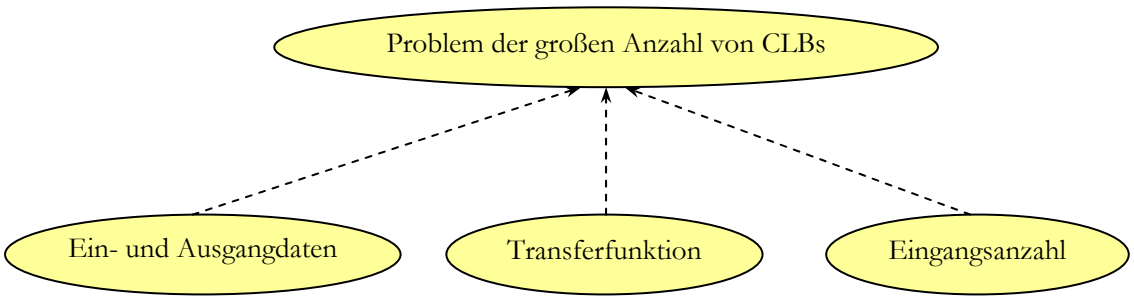


Abbildung 5.1 Problem einer großen Anzahl von CLB zur Abbildung eines Neurons

Die Verwendung dezimaler oder reeller Ein- und Ausgangsdaten sowie nicht Boolescher Transferfunktion von Neuronen führt zur Vergrößerung der Anzahl von CLB, die für die Realisierung eines Neurons erforderlich sind. Für ein Boolesches Neuron treten 2 dieser Nachteile üblicher Neuronen nicht auf, weil es nur Boolesche Daten bearbeitet und weil

es nur eine Boolesche Transferfunktion besitzt. Durch die beschränkte Anzahl von Eingängen eines CLB können nicht alle Booleschen Neuronen auf einen einzelnen CLB direkt abgebildet werden.

Die Trainingsalgorithmen von Booleschen Neuronalen Netzen können aber so modifiziert werden, so dass eine Einschränkung der Eingangsanzahl von Neuronen in einer verborgenen Schicht sowie einer Ausgangsschicht des Netzes berücksichtigt wird. Für die verborgene Schicht bedeutet eine solche Einschränkung praktisch eine Dekomposition der Booleschen Transferfunktion des BN. Eine Ausgangsschicht mit Booleschen Neuronen, die eine einheitliche Boolesche Operation als Transferfunktion haben, kann durch eine Superposition der Booleschen Funktionen dargestellt werden. Folglich kann ein Boolesches Neuron in der Ausgangsschicht mit beliebiger Anzahl von Eingängen durch eine Kaskadierung Boolescher Neuronen dargestellt werden (siehe Abbildung 5.2).

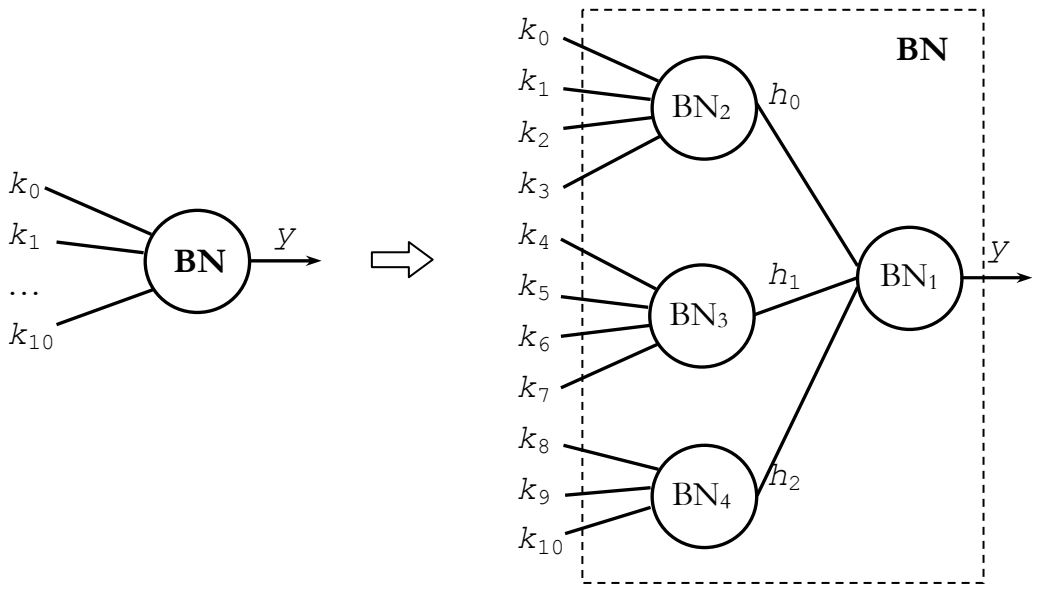


Abbildung 5.2 Darstellung eines Booleschen Neurons der Ausgangsschicht durch eine Kaskade

Jedes Boolesche Neuron in dieser Kaskade kann direkt in einem Logikblock eines CLB abgebildet werden, wobei die Struktur und Logik des Booleschen Neurons eine direkte Abbildung der Architektur des Booleschen Neurons im FPGA erlauben.

Die Formel (5.1) zeigt ein Beispiel der Superposition einer Booleschen Funktion y, die von 10 Teilfunktionen abhängt. Es wird angenommen, dass die zulässige Anzahl der Eingänge kleiner oder gleich 4 ist.

$$y = k_0 \oplus k_1 \oplus k_2 \oplus k_3 \oplus k_4 \oplus k_5 \oplus k_6 \oplus k_7 \oplus k_8 \oplus k_9 \oplus k_{10} \quad (5.1)$$

$$= h_0 \oplus h_1 \oplus h_2 \quad (5.2)$$

wobei

$$h_0 = k_0 \oplus k_1 \oplus k_2 \oplus k_3,$$
$$h_1 = k_4 \oplus k_5 \oplus k_6 \oplus k_7,$$
$$h_2 = k_8 \oplus k_9 \oplus k_{10}.$$

Da die Anzahl von k-Funktionen größer 4 ist, werden die k-Funktionen zu drei Gruppen aufgespaltet. Dabei hat jede Gruppe höchstens 4 k-Funktionen. Eine Verbindung der k-Funktionen durch die Boolesche Operation EXOR in einer einzelnen Gruppe bildet eine neue Funktion h_i, die von höchstens 4 Argumenten abhängt. Eine Verbindung der 3 h-Funktionen durch die Boolesche Operation EXOR ergibt die ursprüngliche Funktion y (5.2), die jetzt direkt nur von 3 Argumenten abhängt und folglich die Einschränkung von maximal 4 Eingängen erfüllt. Nach (5.1) und (5.2) wird das Boolesche Neuron in eine 2-schichtigen Struktur erweitert. Diese Erweiterung der Ausgangsneuronen ist unbegrenzt, durch die Anwendung der Superposition kann eine beliebig große mehrschichtige Struktur von BNN erzeugt werden. Die Darstellung der Booleschen Funktion y durch ein Boolesches Neuron in der Ausgangsschicht eines allgemeinen BNN und durch eine Struktur von Neuronen wird in Abbildung 5.2 gezeigt. Die Vergrößerung der Anzahl von Schichten kann zur Erhöhung der Berechnungszeit der Booleschen Funktion führen [80].

Wie im Kapitel 2 gezeigt wurde, besteht ein CLB aus 2 identischen Logikblöcken (Slices) mit jeweils 2 LUT (lookup table). Jede LUT hat 4 Eingänge und einen Ausgang und kann eine beliebige Boolesche Funktion, die von bis zu 4 Booleschen Variablen abhängt, abbilden. Entsprechend kann ein Logikblock (slice) zwei solche Boolesche Funktionen oder eine Boolesche Funktion mit 5 Argumenten realisieren. Eine beliebige Boolesche Funktion, die durch 6 Argumente definiert ist, kann durch einen CLB dargestellt werden.
Eine Einschränkung der Eingangsanzahl jedes Neurons im Booleschen Neuronalen Netz auf die Eingangsanzahl von LUT im verwendeten FPGA bietet die Möglichkeit, dieses Boolesche Neuron in einer LUT darzustellen. In Abbildung 5.3 wird eine Darstellung eines Booleschen Neurons mit 4 Eingängen durch eine LUT gezeigt.

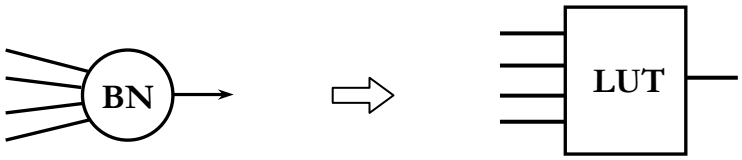


Abbildung 5.3 Abbildung eines Booleschen Neurons auf eine LUT

Eine ganze Netzstruktur wird dabei eins-zu-eins in der Struktur aus LUT abgebildet. Die Abbildung 5.4 zeigt die Realisierung eines Booleschen Neuronalen Netzes in einer Struk-

tur aus LUTs, wobei eine Einschränkung der Eingangsanzahl auf 4 Eingänge je Neuron im dargestellten Netz verwendet wird [81].

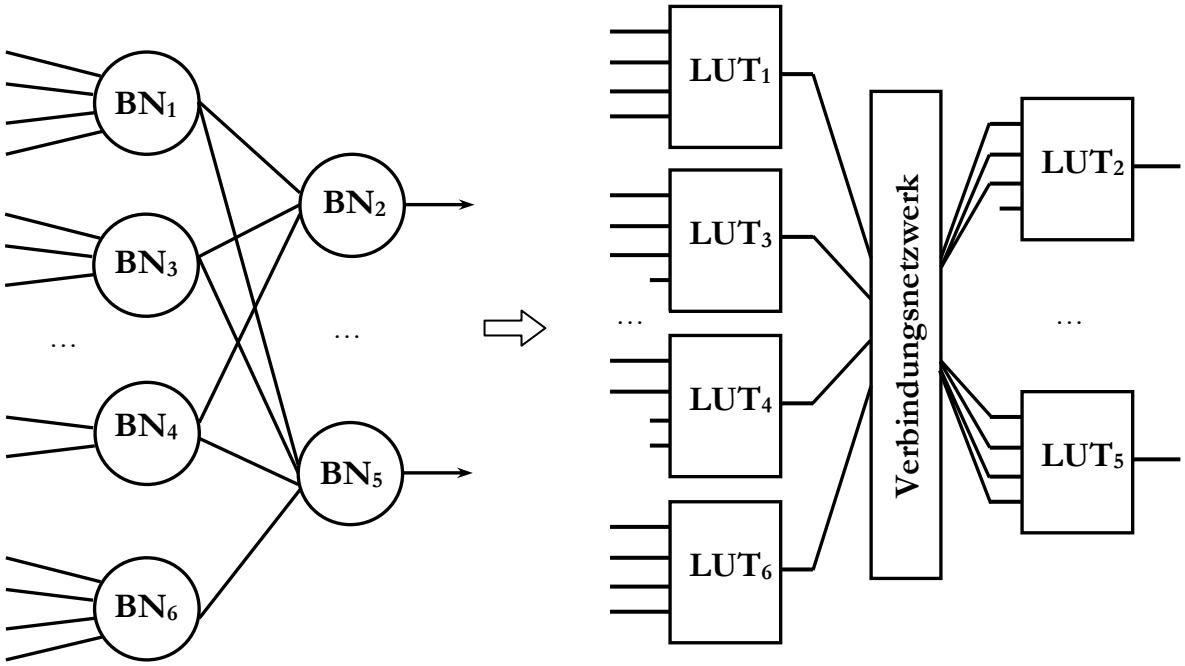


Abbildung 5.4 Abbildung eines Booleschen Neuronalen Netzes in eine Struktur aus LUTs

Bei der Einschränkung der Eingangsanzahl von Booleschen Neuronen auf die Eingangsanzahl eines Logikblocks (slice) bzw. eines CLB erhält man eine Abbildung eines Booleschen Neurons auf ein slice bzw. ein CLB (siehe Abbildung 5.5).

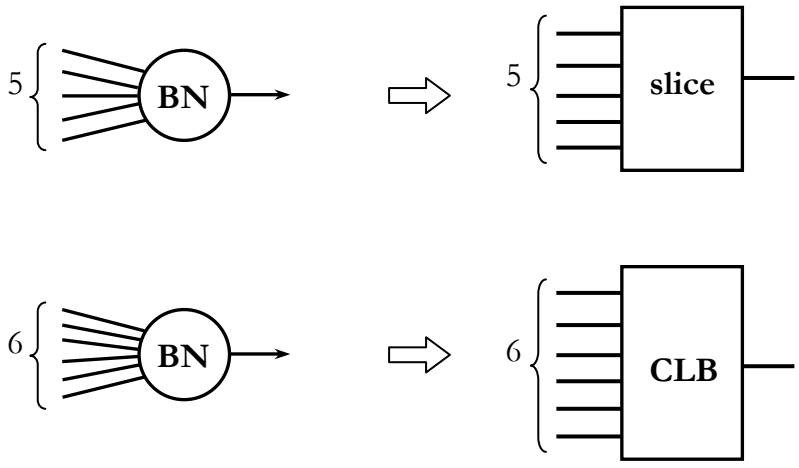


Abbildung 5.5 Abbildung eines BNN in Teile eines FPGA

Eine entsprechende Netzstruktur, die aus Booleschen Neuronen mit 5 bzw. 6 Eingängen besteht, wird in eine aus slices bzw. aus CLB bestehende Struktur abgebildet [80], [81].
Die Verwendung solcher Einschränkungen der Eingangsanzahl von Booleschen Neuronen kann zu einer Vergrößerung der erforderlichen Anzahl von Neuronen im Netz füh-

ren. Der Vorteil dieser Einschränkungen besteht darin, dass nur eine LUT (slice, CLB) für die FPGA-Realisierung jedes Booleschen Neurons benötigt wird. Es können also 4 Boolesche Neuronen durch ein CLB abgebildet werden. Im Vergleich zu schon bekannten NN-Realisierungen in FPGA wird ein sehr großer Gewinn erreicht.

Tabelle 5.1 Bekannte FPGA-Realisierungen von Neuronalen Netzen

Netz	Anzahl von CLB pro ein Neuron
GANGLION [34]	640 – 784
Xilinx-Netz [177]	51
Hopfield-Netz [63][176]	26
Netz von Gschwind [64]	22

Durch Verwendung des Booleschen Neurons für die Modellierung Boolescher Funktionen in FPGA wurde ein großes Problem der Hardware-Realisierung von Neuronalen Netzen in FPGA gelöst. Die Anzahl der CLB, die zur Abbildung eines Neurons erforderliche sind, wurde auf 1 reduziert. Da in einem CLB sogar 4 Boolesche Neuronen realisiert werden können ergibt sich als neuer Hardware-Bedarf für ein Boolesches Neuron sogar nur ¼ CLB.

5.1.2 Adaptierter Trainingsalgorithmus

Für eine erfolgreiche Abbildung eines Booleschen Neuronalen Netzes in FPGA-Struktur soll die Einschränkung der Eingangsanzahl von Booleschen Neuronen im Trainingsalgorithmus des BNN berücksichtigt werden. Ein angepasster Trainingsalgorithmus wird im Folgenden beschrieben.

Die Hauptidee dieses Algorithmus liegt in der sequentiellen Dekomposition aller Booleschen Funktionen der Funktionsmenge. Zunächst wird die Eingangsschicht des Netzes eingefügt, Alg. 5.1 - Zeile 1. Die Anzahl von Neuronen in der Eingangsschicht ist gleich der Anzahl aller Argumente der Menge Boolescher Funktionen. Dann beginnt die Entwicklung der verborgenen Schicht und die Anzahl von verborgenen Neuronen wird gleich 0 gesetzt. Danach wird eine Funktion nach der anderen aus der Funktionsmenge gewählt, Alg. 5.1 - Zeile 4. In Zeile 5 wird die Bedingung geprüft, ob die gewählte Boolesche Funktion aufgespaltet werden kann. Diese Bedingung unterscheidet sich für die verschiedenen Grundoperationen des Trainingsalgorithmus (siehe Beschreibung von Trainingsmethoden in 4.2.1 und 4.2.5). Zum Beispiel, für OR und EXOR-Dekomposition gilt:

Solange die gewählte Boolesche Funktion die Einswerte hat, wird ein neues verborgenes Neuron eingefügt (Alg. 5.1 - Zeile 6), eine entsprechende Transferfunktion $k[i]$ durch die

Funktion FINDK (*bf*, *limit*) gesucht (Alg. 5.1 - Zeile 7) und, wenn möglich, von allen Funktionen in der Funktionsmenge abgespaltet (Alg. 5.1 - Zeile 8). Dabei erhält man einen Gewichtsvektor *w*[*i*]. Die Funktionen FINDK() und REDUCTIONOFYSET() (Alg. 5.1 – Zeilen 7 und 8) werden weiter in Algorithmen 5.2 und 5.3 ausführlich beschrieben. Jede neu ermittelte Transferfunktion und die neuen Gewichte werden zu der entsprechenden Menge und Matrix hinzugefügt (Alg. 5.1 - Zeilen 9 und 10). Sind alle Boolesche Funktionen zerlegt, d.h. die Menge von *k*-Funktionen und Gewichten erzeugt, wird die Superposition verwendet (Alg. 5.1 - Zeile 12), um die Abbildung von Neuronen der Ausgangsschicht in die FPGA-Struktur zu ermöglichen.

Algorithmus 5.1 Train – Training des BNN für Abbildung in FPGA-Struktur

Eingabe:

Yset - Menge Boolescher Funktionen

limit- Maximale Anzahl von Eingängen für Neuronen

Ausgabe:

net={*Kset*, ***W***} - Netz

```
TRAIN(Yset, limit )
 1   Füge eine Eingangsschicht ein
     //Start einer verborgenen Schicht
 2   i←0
 3   for (func ← 0,…,Ny-1)   //Ny - Anzahl der Funktionen in der Menge
 4         bf←SELECTFROMYSET(func)
 5         while ISNOTDECOMPOSED(bf) do
 6               i++                    // Füge ein neues Neuron ein
 7               k[i] ← FINDK (bf, limit, flag)
 8               w[i] ← REDUCTIONOFYSET(k[i])
 9               Kset ← ADDTOSET(k[i])
10               W ← ADDTOSET (w[i])
     //Anpassung der Ausgangsschicht
12   W ←SUPERPOS(W, limit)
13   return net
```

Im Algorithmus 5.1 (Zeile 7) wird die Funktion FINDK() aufgerufen, die eine Transferfunktion für das eingefügte verborgene Neuron sucht. Der Algorithmus 5.2 beschreibt die Arbeitsweise dieser Funktion. Als Eingangsparameter bekommt die FINDK() die Boolesche Funktion *bf* und die gewünschte Anzahl von Eingängen des verborgenen Neurons *limit*. Der Wert *limit* ist gleichzeitig die gewünschte Anzahl von Variablen für zu suchende Transferfunktion dieses Neurons. Als Ausgabe liefert der Algorithmus die Boolesche Funktion *k* zurück. Die zu suchende Funktion soll eine Teilfunktion der angegebenen Funktion *bf* sein und wenn möglich, von höchstens *limit* Booleschen

Variablen abhängen. Hängt die gefundene Transferfunktion von *limit* Booleschen Variablen ab, ist der Parameter *flag* gleich 1, und falls die gefundene Transferfunktion von mehr als *limit* Booleschen Variablen abhängt, *flag* =0.

Algorithmus 5.2 FindK – Suche der Transferfunktion eines verborgenen Neurons

Eingabe:

bf – Boolesche Funktion

limit- Anzahl von Eingängen des Neurons

Ausgabe:

k – Transferfunktion des verborgenen Neurons

flag = {1 falls *k* -Funktion gefunden; 0 sonst}

```
FINDK (bf, limit, flag)
 1   diff ←N_x -limit;  flag ←0; i ←0  //N_x - Variablenanzahl in bf
 2   while (flag=0 && i<N_x) do
 3         if (min_{x_i} bf !=1)then
 4                     k← min_{x_i} bf
 5                     flag ←1
 6         i++
 7   if (flag=1 && diff >1) then
 8         k2←FINDK(k_/2,limit,flag2)//k_/2 - Hälfte der Wertetabelle der k
 9         if (flag2=1) then k←2*k2 //* -Wiederholung der Wertetabelle der k2
10   if (flag=1) then return res
11   else    return bf
```

Der Algorithmus funktioniert folgenderweise. Zunächst wird eine Differenz *diff* zwischen der Variablenanzahl in der Funktion *bf* und der gewünschte Anzahl von Variablen in der zu suchenden Funktion *k* berechnet, eine Hilfsvariable *i* initialisiert und *flag* =0, Alg.5.2 – Zeile 1. Dann in der **while**-Schleife (Alg. 5.2 –Zeilen 2-7) wird eine Teilfunktion der angegebenen Funktion *bf* gesucht. Diese Teilfunktion soll von *Nx*-1 Booleschen Variablen abhängen. Solange solche Teilfunktion nicht gefunden (*flag*=0) und der Hilfsparameter $i<N_x$, wird das partielle Minimum der Funktion *bf* nach der Variable x_i berechnet. Falls das Minimum keine konstante Funktion 1 sind, Alg.5.2 – Zeile 3, wird die Teilfunktion *k* der Funktion *bf* gefunden, Alg.5.2 – Zeile 4, und *flag* auf 1 gesetzt, Alg.5.2 – Zeile 5. Falls in der **while**-Schleife eine Teilfunktion der angegebenen Funktion *bf* gefunden wurde und die Differenz *diff*>1, Alg. 5.2 – Zeile 7, wird ein rekursiver Aufruf der Funktion FINDK() durchgeführt, Alg. 5.2 – Zeile 8. Dafür wird eine neue Boolesche Funktion durch eine Hälfte der Wertetabelle der gefundenen Teilfunktion *k* definiert ($k_{/2}$) und als Eingabeparameter der Funktion FINDK() betrachtet, Alg. 5.2 – Zeile 8. Die Wertetabelle der Funktion $k_{/2}$ besteht aus der Hälfte der Wertetabelle der

gefundenen Funktion *k* und somit hängt von *Nx* - 1 Booleschen Variablen ab. Die Funktion FINDK() in der Zeile 8 liefert die Funktion *k*2 zurück, und falls *flag*2=1, wird die Funktion *k* durch die Wiederholung der Wertetabelle der Funktion *k*2 definiert (*k*←2**k*2), Alg. 5.2 – Zeile 9. Abhängig vom Wert des *flag* wird entweder die gefundene Teilfunktion der Funktion *bf* oder die selbst Quellfunktion *bf* als Rückgabeparameter zurückgeliefert, Alg. 5.2 – Zeilen 10-11.

Im Algorithmus 5.2 wurde die Funktion FINDK() für OR- bzw. EXOR-Dekomposition beschrieben. Dies erkennt man aus den Zeilen 3-4 des Algorithmus, wo das partielle Minimum der Funktion *bf* berechnet wurden. Für die AND- bzw. EXAND-Dekomposition bleibt der Algorithmus unverändert, nur anstatt des Minimums wird das partielle Maximum der Funktion *bf* berechnet.

Algorithmus 5.3 Reduction of *Yset* – Abspalten der *k*-Funktion von allen Funktionen in der Funktionsmenge *Yset*

Eingabe:
k – Transferfunktion des verborgenen Neurons
Ausgabe:
w – Vektor von Gewichten für Neuronen in der Ausgabeschicht des Netzes

```
REDUCTIONOFYSET(k)
 1   for (func ← 0,…,Ny−1)   //Ny - Anzahl der Funktionen in der Menge
 2         w[i] ← 1
 3         for (j ← 0,…,Np−1)
 4               if (k[j] =1 && func[j] =0) then
 5                     w[i] ← 0
 6         if (w[i] =1) then
 7               for (j ← 0,…,Np−1)
 8                     if (k[j] =1) then
 9                           func[j]← 0
10   return w
```

Da der Algorithmus 5.2 der Funktion FINDK() oben für OR- bzw. EXOR-Dekomposition erklärt wurde, wird Algorithmus 5.3 die Funktion REDUCTIONOFYSET() auch für OR-Dekomposition angeführt.

Der Algorithmus **Reduction** bekommt die Boolesche Funktion *k* als Eingabeparameter und prüft, ob diese Funktion von jeder Funktion der zu modellierenden Funktionsmenge abgespaltet werden kann. Dabei wird der Vektor von Gewichten für die Neuronen der Ausgabeschicht des Booleschen Neuronalen Netzes bestimmt.

Für jedes Neuron der Ausgabeschicht des Netzes bzw. jede Funktion der Funktionsmenge, die durch dieses BNN dargestellt werden soll, Alg. 5.3 – Zeile 1, wird das Gewicht zunächst auf 1 gesetzt, Alg. 5.3 – Zeile 2. Falls es für mindestens einen Einswert der *k*-

Funktion einen Nullwert der aus der Menge gewählten Funktion gibt, Alg. 5.3 – Zeile 4, wird das entsprechende Gewicht in 0 gesetzt, Alg. 5.3 – Zeile 5. Wenn das Gewicht gleich 1 geblieben ist, Alg. 5.3 - Zeile 6, werden die Werte, die aus der Menge der gewählten Funktionen für jeden Einswert der *k*-Funktion ausgewählt wurden, Alg. 5.3 – Zeile 8, auf 0 gesetzt, Alg. 5.3 – Zeile 9.

Für die AND-Dekomposition prüft man in der Zeile 4, ob es für mindestens einen Nullwert der *k*-Funktion einen Einwert der aus der Menge gewählten Funktion gibt. Das Abspalten jeder Funktion in der Zeilen 8-9 wird wie folgt auch geändert: Wenn das Gewicht gleich 1 geblieben ist, wird der Wert der aus der Menge gewählten Funktion für jeden Nullwert der *k*-Funktion, in 1 gesetzt. Für die EXOR bzw. EXAND-Dekomposition wird die Funktion REDUCTIONOFYSET() hier nicht betrachtet, weil es kein Problem ist, auch für diese Operationen ein entsprechender Algorithmus zu entwickeln. Dafür wird vorgeschlagen, die Formeln 4.44-4.49 zu verwenden.

Zur Erhöhung der Wirksamkeit des adaptierten Algorithmus kann man eine Mischung aus der FINDK–Funktionen für verschiedene Operationen (z.B. OR und AND) verwenden. Eine Erklärung dafür wird am Beispiel 5.1 gegeben, wo die Funktionen FINDK für OR- und AND-Dekomposition im Algorithmus eine nach anderen aufgerufen werden. Wenn eine *k*-Funktion durch die FINDK() geliefert wird, folgt danach den Aufruf einer entsprechenden REDUCTIONOFYSET-Funktion.

Beispiel 5.1. Es werden die Boolesche Funktionen y_1, y_7, und y_9 aus der Funktionsmenge (Anhang A.1) gewählt. Im Resultat des OR-Trainings können diese Funktionen durch die OR-Verknüpfung der Transferfunktionen der verborgenen Neuronen k_1, k_2, k_3, k_4 realisiert werden (A.1-A.3). $y_1=k_4$ und y_7, y_9 hängen jeweils von 3 k-Funktionen ab, wobei gilt: k_2 ist auf x_1 und x_2 definiert, und k_1, k_3, k_4 hängen von 3 Boolesche Variablen x_1, x_2, x_3 ab. Nehmen wir an, dass eine LUT 2 Eingänge hat. Für die direkte Realisierung der Transferfunktionen der Booleschen Neuronen wird der oben beschriebene adaptierte Trainingsalgorithmus verwendet.

$$k_1 = \overline{x_2}x_3 \qquad k_2 = x_2 \vee x_3 \qquad k_3 = x_1 \oplus x_3 \qquad k_4 = \overline{x_2} \qquad k_5 = x_1 \odot x_3 \tag{5.3}$$

$$y_1 = k_1 \vee k_2 \wedge k_3 \qquad y_7 = k_3 \vee k_4 \qquad y_9 = k_4 \vee k_5 \tag{5.4}$$

Die erzeugte Netzstruktur wird in Abbildung 5.6 gezeigt. Jedes Neuron in der verborgenen Schicht hat eine eigene Transferfunktion, die sich von den Transferfunktionen aller anderen verborgenen Neuronen unterscheidet (5.3).

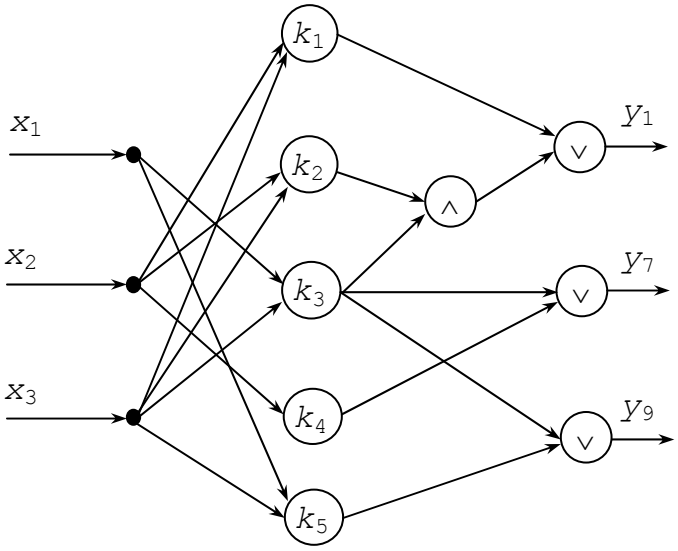


Abbildung 5.6 BNN mit 2-Eingängigen Neuronen

Alle Transferfunktionen hängen von 2 Eingangssignalen ab. Das Ausgangsneuron, das die Funktion y_1 realisiert, wurde in 2 Neuronen aufgespaltet, weil jedes Neuron laut Aufgabenstellung höchstens 2 Eingänge haben darf. Die Spaltung des Ausgangsneurons wurde durch die Verwendung der Superposition für eine Ausgangsschicht erreicht. Der Algorithmus der Superposition wird in diesem Abschnitt noch beschrieben.

Tabelle 5.2 Wertetabelle der Transferfunktionen k_1, k_2,…, k_5 und der Ausgangsfunktionen y_1, y_7, und y_9

	x_1	x_2	x_3	k_1	k_2	k_3	k_4	k_5	y_1	y_7	y_9
0	0	0	0	0	0	0	1	1	0	1	1
1	0	0	1	1	1	1	1	0	1	1	1
2	0	1	0	0	1	0	0	1	0	0	1
3	0	1	1	0	1	1	0	0	1	1	0
4	1	0	0	0	0	1	1	0	0	1	1
5	1	0	1	1	1	0	1	1	1	1	1
6	1	1	0	0	1	1	0	0	1	1	0
7	1	1	1	0	1	0	0	1	0	0	1

Im Vergleich zu dem Beispiel aus dem Anhang A.1, wo nur 4 *k*-Funktionen zur Abbildung der 10 *y*-Funktionen benötigt wurden, braucht man in diesem Fall zur Abbildung der 3 *y*-Funktionen schon 5 *k*-Funktionen, die in der Tabelle 5.2 angegeben sind. Der adaptierte Algorithmus liefert aber Boolesche Funktionen, die nur von 2 Boolesche Variable abhängen. Dies ermöglicht eine direkte Abbildung der Transferfunktionen von Booleschen Neuronen in eine FPGA-Struktur, deren LUTs nur 2 Eingänge besitzen.

Der Algorithmus der Superposition verwendet Prinzipien, die in den Formeln (5.1) und (5.2) gezeigt wurden. Weiter wird eine mögliche Realisierung dieses Algorithmus beschrieben.

Algorithmus 5.4 Superpos – Verwendung der Superposition für eine Ausgangsschicht

Eingabe:

W – Gewichtsmatrix des Netzes

limit- Maximale Anzahl von Eingängen für Neuronen

Ausgabe:

W - Gewichtsmatrix, die zur Anzahl von Eingängen angepasst ist

```
SUPERPOS(W, limit)
 1   for (i ← 0,…, Ny)
 2       nl=0          //Anzahl der Kaskaden in der Ausgangsschicht
 3       nnl=1         //Neuronenanzahl in der Kaskade nl
 4       iOut←1        //Neuronenanzahl in der Ausgangsschicht für Yset[i]
 5       n_inp←limit   //Eingangsanzahl der Ausgangsschicht
 6       g←VEKTOR-GEWICHT(W[i])
 7       while g>n_inp do
 8           iOut++    //Neuronenanzahlerhöhung
 9           if nnl != POW(limit, nl)
10               then  nnl++      //Neuronenanzahlerhöhung
11               else  nl++       //Kaskadenanzahlerhöhung
12                     nnl←1
13           n_inp←nnl*(limit -1)+ POW(limit, nl)
14       CHANGE(W) //Anpassung der Gewichte
15   return W
```

Eine Alternative zum Algorithmus der Superposition ist die Verwendung der Methode zur Entwicklung des mehrschichtigen Netzes entsprechend Abschnitt 4.3.1.b. Nachdem eine verborgene Neuronenschicht aufgebaut ist und Gewichtskoeffizienten für alle Booleschen Neuronen gefunden sind, wird das Training eines weiteren BNN durchgeführt. Als Eingangssignale des 2. Netzes werden die Ausgabewerte der *k*-Funktionen des 1. Netzes verwendet. Dabei wird eine andere Basisoperation für das zweite Training benutzt. Als Ergebnis erhält man eine weitere Kaskade (weitere Schicht) von Booleschen Neuronen und eine Ausgangsschicht mit einer anderen Basisoperation. In einigen Fällen führt das zur Verkleinerung der Anzahl von Eingängen für die Ausgangsneuronen.

In der Tabelle 5.3 werden die Ergebnisse des Trainings für die normalen und adaptierten OR- und AND-Algorithmen dargestellt. Für das Training wurden einige Benchmarks benutzt. Aus diesen Ergebnissen erkennbar, dass die Anzahl der verborgenen Neuronen für die adaptierten Algorithmen für LUTs mit 4 Eingängen fast überall größer als die Anzahl der verborgenen Neuronen ohne Beschränkungen der Anzahl von Eingängen.

Tabelle 5.3 Ergebnisse des adaptierten Algorithmus

#inputs	#outputs	bench	$\#k_{OR}$	$\#k_{OR,4}$	$\#k_{AND}$	$\#k_{AND,4}$
7	10	5xp1	29	46	26	51
7	5	5xp_5	21	37	18	39
9	1	9sym1	4	20	8	36
14	8	alu4	17	731	17	596
5	5	bw_5	11	15	13	13
5	9	bw	17	25	20	25
15	8	b12	10	49	10	27
7	2	con1	4	12	4	10
15	38	alcom	38	40	46	74

Die Bezeichnungen in der Tabelle haben folgende Bedeutung:

#inputs – Anzahl der Booleschen Variablen,

#outputs - Anzahl der Booleschen Funktionen,

bench - verwendeter Benchmark als Quelle der Funktionen,

$\#k_{OR}$, $\#k_{AND}$ – Anzahl der verborgenen Neuronen für OR- und AND-Training ohne Beschränkungen der Anzahl von Eingängen,

$\#k_{OR,4}$, $\#k_{AND,4}$ – Anzahl der verborgenen Neuronen für adaptierten OR- und AND-Algorithmen für LUTs mit 4 Eingängen.

Nur in einem Fall (AND-Dekomposition bw_5) ist die Anzahl der verborgenen Neuronen für beide AND-Trainings gleich. Dieses Ergebnis wird dadurch erklärt, dass die 5 Booleschen Funktionen in diesem Benchmark nur von 5 Booleschen Variablen abhängen. Die Beschränkung der Anzahl von Eingängen in LUTs auf 4 Eingänge ist sehr nah zu der Anzahl von Booleschen Variablen und somit beide Trainingsalgorithmen liefern gleiche Transferfunktionen für Booleschen Neuronen in der verborgenen Schicht des Netzes. Das gleiche gilt auch für bw und OR-Dekomposition von bw_5. Der kleine Unterschied zwischen der Anzahl von verborgenen Neuronen für den normalen und adaptierten Algorithmen offenbart die guten Eigenschaften des normalen Algorithmus, einfache Teilfunktionen zu erzeugen, oder auch das Vorhandensein von Funktionen in der Funktionsmenge, die durch den Algorithmus mit Beschränkung der Anzahl von Eingängen in LUTs und der gewählten Basisoperation schlecht dekomponierbar sind. Für die Funktionen, die von weit mehr als 4 Variablen (Beschränkung von Eingängen im Algorithmus) abhängen aber gut dekomponierbar sind, erzeugen beide Trainingsalgorithmen fast die gleiche Anzahl der verborgenen Neuronen, z.B. Benchmark alcom.

Die Strukturen von Booleschen Neuronalen Netzen nach den AND-Trainingsalgorithmen (mit und ohne Beschränkung der Anzahl von Eingänge in Neuronen) für Benchmark alcom werden im Anhang A.3, Abb. A.2-A.3, dargestellt.

5.2 Hardware-Realisierung von Booleschen Neuronalen Netzen am Beispiel von Virtex II-FPGA

5.2.1 Methodik

Zur Implementierung eingebetteter Systeme bzw. Anwendungen in FPGAs gibt es viele Methoden. Das zu realisierende Objekt bzw. System muss dazu auf jedem Fall zunächst spezifiziert werden. Dafür benutzt man eine Hardware-Beschreibungssprache, eine grafische Eingabe eines Schaltplans oder endlichen Automaten. Am weitesten verbreitet ist die Verwendung von Hardware-Beschreibungssprachen, zum Beispiel VHDL oder Verilog. Nach der Beschreibung innerhalb des Entwurfsflusses folgen weitere Schritte wie die funktionale Simulation, die Synthese und die Implementierung. Erst danach kann man den realen FPGA mit dem erzeugte Bit-Stream konfigurieren.
Trotz aller Fortschritte in der Synthesetechnologie auf hoher Ebene ist der Abstand zwischen dem Problem-Niveau und dem Implementierungsniveau noch sehr groß. Eine direkte Spezifikation und Beschreibung des zu realisierenden in FPGA Systems durch z.B. VHDL ist sehr aufwändig. Im Anhang B wird die VHDL-Beschreibung eines kleinen Designs angeführt. Dafür sind mehr als ein Tausend Zeilen des VHDL-Codes erforderlich. Eine mögliche Lösung dieses Problems wäre die Spezifikation auf der Systemebene mit einer Sprache, die die automatisierte Transformation dieser Spezifikation in eine ausführbare Anwendung ermöglichen. Eine sehr verbreitete und oft angewendete Sprache, die für die objektorientierte Spezifikation und für das Design auf der Systemebene benutzt wird, ist die Unified Modeling Language (UML). Eine effiziente Methodik zur Realisierung von Systemen in FPGA wurde von Fröhlich, Steinbach und Beierlein entwickelt [145]-[146]. Diese Methodik verbindet das Paradigma des Hardware/Software-CoDesigns mit dem Konzept Systementwicklung in einer Modell-gesteuerten Architektur (MDA-Model Driven Architecture) [54], [55], [101] und [116]. Für die Beschreibung des zu realisierenden Objektes wird die UML 2.0 verwendet. Die exakte UML-Systemspezifikation von Software- und Hardware-Teilen wird auf der Basis von Plattformmodellen durch ein spezielles Werkzeug, den MOCCA-Compiler (MOdel Compiler for reConfigurable Architectures) [110], in eine ausführbare Anwendung transformiert.
Im Folgenden wird das gesamte Verfahren zur Abbildung einer Booleschen Funktion im FPGA durch ein Boolesches Neuronales Netz mit der Verwendung der UML und MOCCA-Compiler beschrieben. Ein allgemeines Schema dieses Verfahrens wird in der Abbildung 5.7 gezeigt. Das gesamte Verfahren besteht aus 2 Prozessen:

- Erzeugung der Struktur eines Booleschen Neuronalen Netzes,
- Abbildung dieses Netzes im FPGA.

Die Eingangsdaten für den ersten Prozess bildet eine Boolesche Funktion bzw. eine Menge Boolescher Funktionen. Die Erzeugung der Struktur und Bestimmung aller Parameter des Booleschen Neuronalen Netzes wird in 3 Prozeduren des 1. Prozesses durchgeführt:

- Vorbearbeitung der Eingangsdaten,
- Training des BNN,
- Darstellung des trainierten BNN durch UML-Modelle.

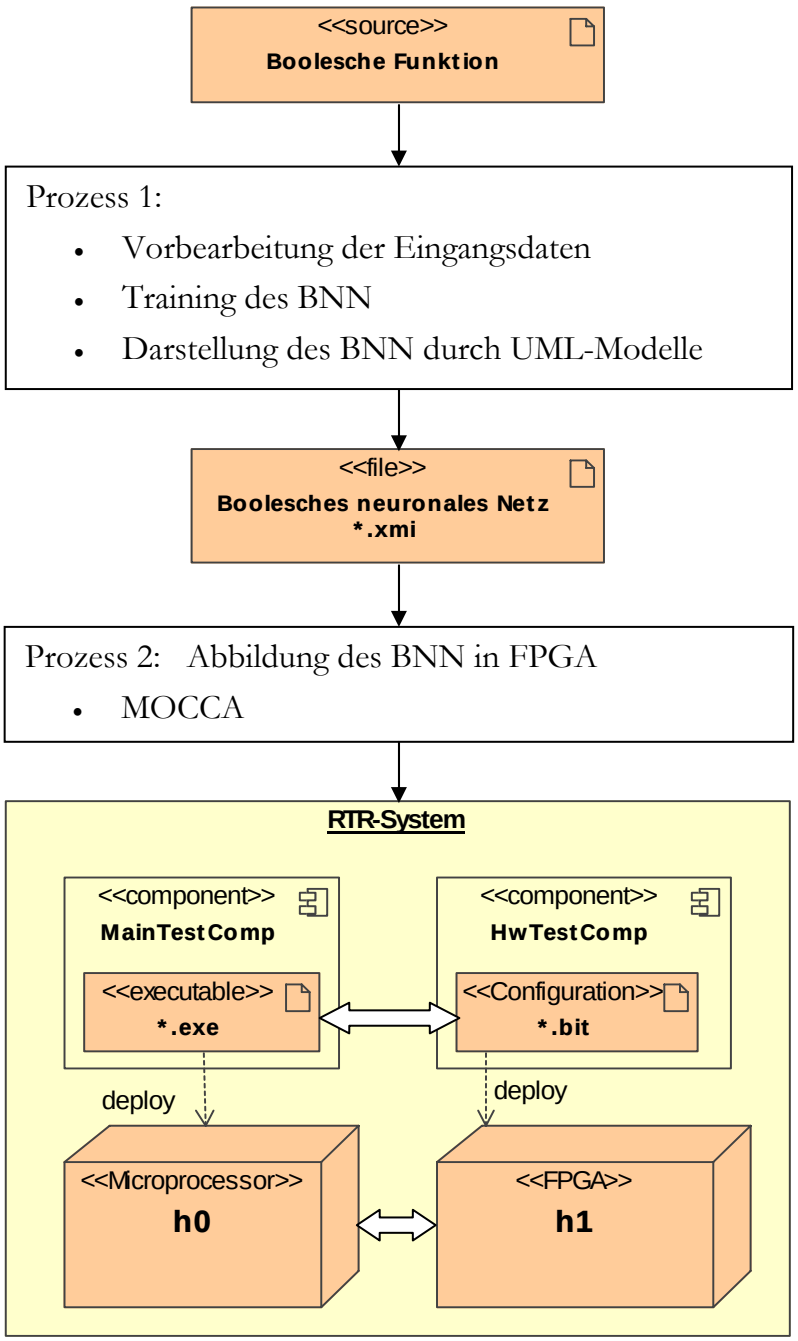


Abbildung 5.7 Abbildungsverfahren einer Booleschen Funktion im FPGA

Zunächst wird die angegebene Boolesche Funktion in eine Trainingsmatrix umgewandelt, die für das Training des BNN verwendet werden kann. Eine Menge Boolescher Funktionen wird in einer Projekt-Datei spezifiziert. In diesem Projekt wird der Name des Projektes, die Anzahl der Booleschen Funktionen in der zu modellierenden Funktionsmenge und die genauen Dateinamen mit den Pfaden jeder Booleschen Funktion angegeben. Ein Beispiel eines Projektes zeigt die Abbildung 5.8.

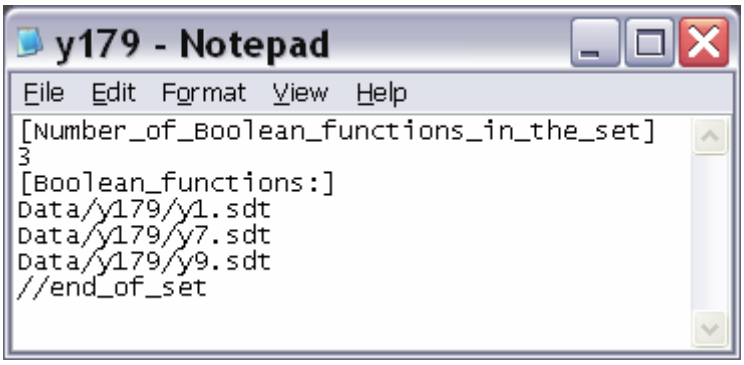


Abbildung 5.8 Projekt -Datei

Beispiel 5.2. Der Projekt-Name ist der Dateiname ohne Erweiterung „`y179`“. Jede einzelne Boolesche Funktion wird in einer eigenen Datei gespeichert, zum Beispiel „`Data/y179/y1.sdt`“. Das `*.sdt`-Format der gespeicherten Funktionen wird durch das spezielle Werkzeug „XBOOLE-Monitor“ [25][130][144] unterstützt. Die Abbildung 5.9 zeigt das Hauptfenster des XBOOLE-Monitors mit der geöffneten TVL der Booleschen Funktion `y1`.

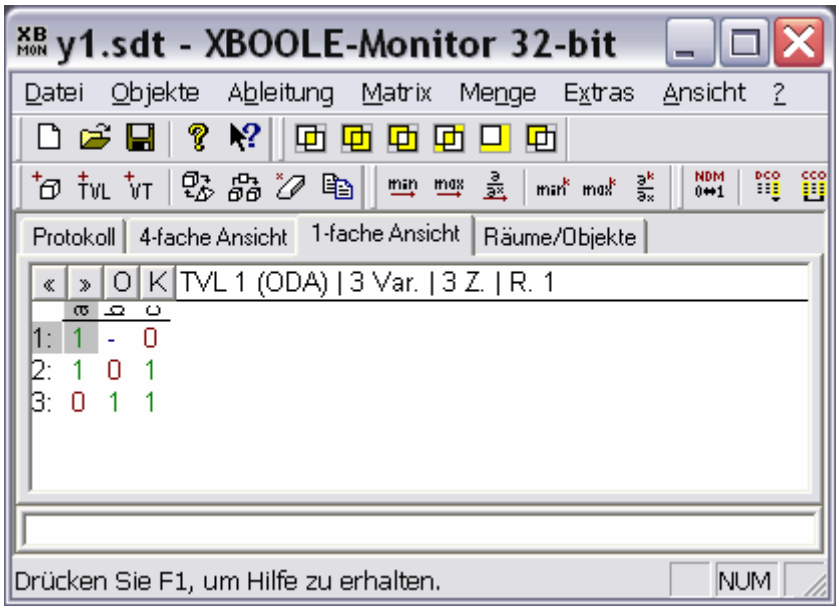


Abbildung 5.9 TVL der Booleschen Funktion y1
im Programmfenster des XBOOLE-Monitors

Zum Training des Booleschen Neuronalen Netzes wird der in vorangehendem Abschnitt beschriebene Trainingsalgorithmus benutzt. Als Resultat des Trainings werden die Mengen synaptischer Gewichte und Transferfunktionen jedes Neurons sowie eine Struktur des Netzes erzeugt. Die Eingangsdaten für den zweiten Prozess bildet ein trainiertes Boolesches Neuronales Netz, das durch ein UML-Modell vollständig beschrieben wird. Ein Beispiel des durch UML beschriebenen trainierten Booleschen Neuronalen Netzes wird weiter unten angegeben. Eine Darstellung des Netzes durch ein UML-Modell ist für den 2. Prozess erforderlich. Der 2. Prozess transformiert das trainierte Netz in eine FPGA-Struktur. Für diese Abbildung wird eine Methode des Hardware/Software-CoDesigns von in der Laufzeit rekonfigurierbaren (Run-Time Reconfigurable - RTR) Architekturen benutzt. Diese Methodik wird durch den MOCCA-Compiler [110] unterstützt. Durch den Compiler wird die Systemprüfung, die Plattformtransformation und Anwendungssynthese automatisch durchgeführt. Die synthetisierte Anwendung nutzt das Leistungspotenzial der RTR-Architektur effizient aus und kann direkt ausgeführt werden. Die MOCCA-Umgebung wurde nicht für eine spezielle Klasse von Anwendungen entwickelt. Jedoch wurde MOCCA in [146] für die Beschleunigung von berechnungsaufwendigen Algorithmen im Bereich Boolescher Problemen und Neuronaler Netze als besonders gut geeignet vorgestellt.

In der Abbildung 5.10 wird das Schema der Arbeitsweise von MOCCA dargestellt. Für die Entwicklung des MOCCA-Systems wurden folgende Konzepte benutzt:

- Validierung von Modellen
- Optimierung (etwa 20 Optimierungsalgorithmen)
- Eliminierung des Pseudo-Codes
- Übertragung von globalen Konstanten
- Unterstützung von Substitutionen
- Generierung von C++ und VHDL-Code
- Optimierung von gemeinsamen Ausdrücken etc.

Hier werden nur einige Aspekte der Anwendung von MOCCA gezeigt, da MOCCA nicht der Bestandteil dieser Dissertation ist. Eine gründliche Diskussion zu MOCCA sowie eine ausführliche Beschreibung der gesamten Entwicklungsmethode sowie Beispiele sind in vielen Publikationen, z.B. [15], [81], [110] und [145]-[146] zu finden.

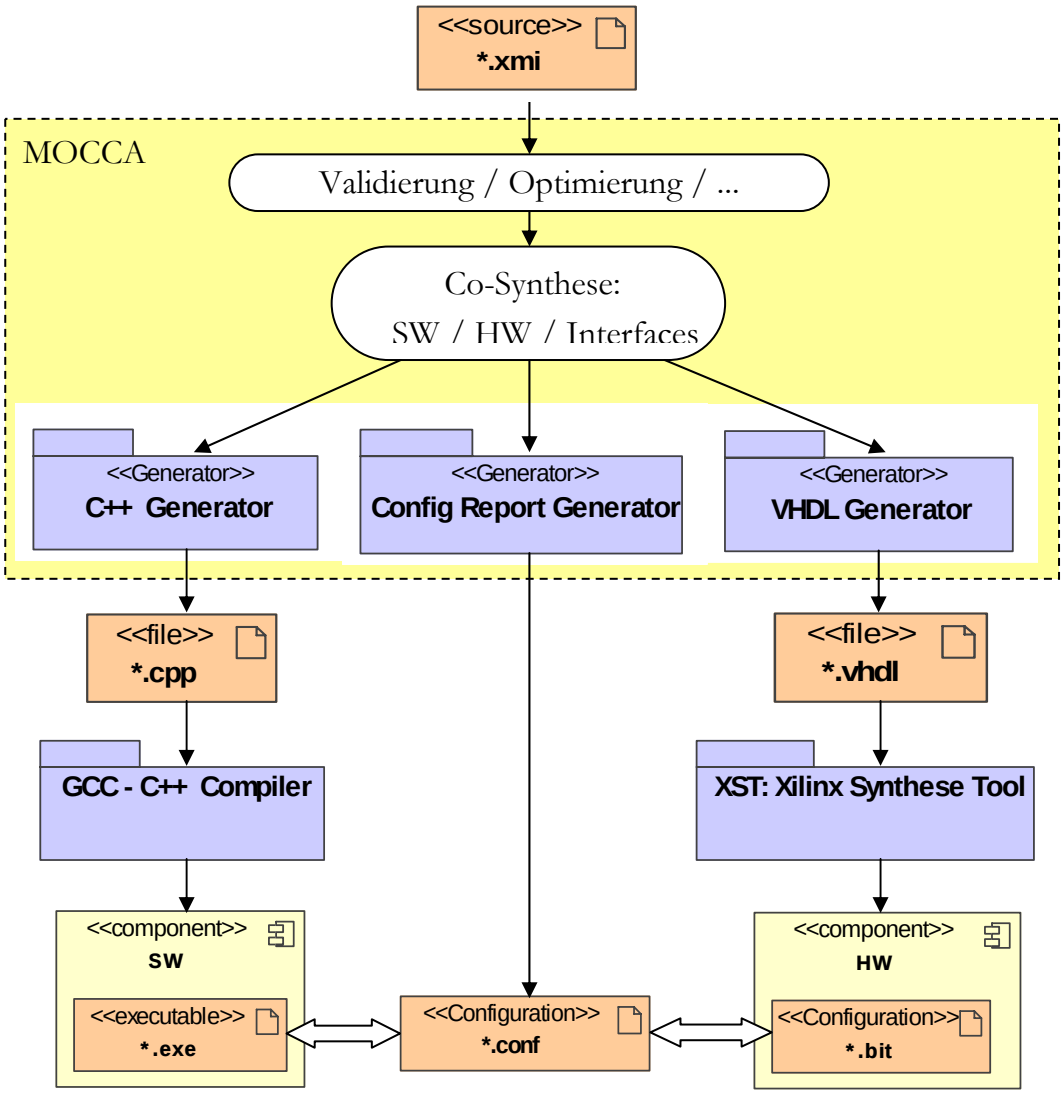


Abbildung 5.10 Allgemeines Schema von MOCCA

Für die Synthese/Kompilierung der Konfiguration des Bit-Streams/exe-Datei werden sowohl kommerzielle Tools als auch GNU-Lizenz-Tools verwendet. Für die Synthese des Bit-Streams, Konfigurierung des FPGAs wird das „Xilinx Synthese Tool“ benutzt. Die Kompilierung von C++-Quell-Code wird mit dem GCC-Compiler durchgeführt [58]. Nach der Synthese und Kompilierung werden Bit-Stream und exe-Datei in die Anwendung übertragen, die auf der Ziel-Architektur direkt ausgeführt werden kann.

5.2.2 UML-Modelle

Die Plattformmodelle legen Eigenschaften fest, die das Fundament des Entwicklungsprozesses bilden. Jede Plattform wird durch ein spezielles Plattformmodell angegeben. Plattformmodelle abstrahieren von Details der Plattform, aber tragen genug Information, um

Wiederholungen im Designfluss zu vermeiden. Sie sind Basis für die Definition der Deployment-Modelle, die die ausführbare Systeme beschreiben. Die Beziehung zwischen Plattformmodellen und den konkreten Modellen von Anwendungen wird in Abbildung 5.11 gezeigt.

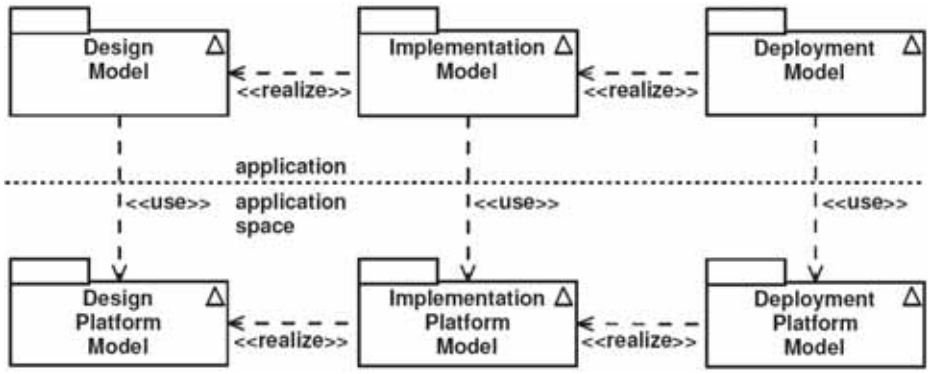


Abbildung 5.11 Beziehungen zwischen den Modellen [145]

Alle Modelle werden mit der Unified Modeling Language (UML) [117], [135] und einer speziell entwickelten Aktionssprache - MAL (MOCCA Action Language) - beschrieben. Die UML 2.0 und die MAL werden zur Spezifikation von Objekt-Orientierten Systemen auf der Systemebene verwendet. Durch die MAL wird das detaillierte Verhalten von UML-Modellen spezifiziert. Diese Sprache ist konform zur UML-Action-Semantik.

Design-Modell.

Ein Design-Modell definiert eine von der Realisierung unabhängige Beschreibung von Use Cases des Systems. Dieses Modell definiert auch die Struktur und das Verhalten des Systems. Die Systemstruktur wird durch UML-Elemente wie Klassen, Interfaces und Beziehungen zwischen ihnen beschrieben. Das Verhalten des Systems kann mit Operationen und Zustandsautomaten beschrieben werden. Das detaillierte Verhalten des Systems wird durch UML-Aktionen definiert, wobei MAL als eine Aktionssprache verwendet wird.

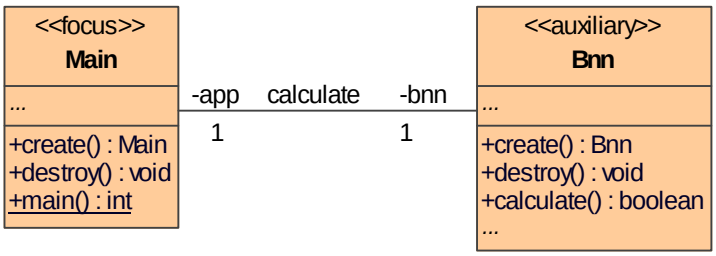


Abbildung 5.12 Design-Modell

Beispiel 5.3. Als Beispiel des Hardware/Software-CoDesigns wurde das trainierte Boolesche Netz aus dem Anhang A.1 benutzt. Abbildung A.1 zeigt die Struktur des Netzes. Die Transferfunktionen der verborgenen Booleschen Neuronen und die Verbindungsgewichte der Booleschen Neuronen der Ausgangsschicht sind in den Tabellen A.7 und A.8 dargestellt. Die 10 Boolesche Funktionen y_1, y_2, ..., y_{10} werden durch 4 Boolesche Transferfunktionen k_1, k_2,…, k_4 abgebildet (A.1)-(A.3). Das diesem BNN entsprechende UML-Design-Modell wird in der Abbildung 5.12 angegeben. Das System besteht aus den beiden Klassen `Main` und `Bnn`. Durch die Methode `calculate()` der Klasse `Bnn` wird das trainierte Boolesche Neuronale Netz definiert. Das Objekt der Klasse `Main` erzeugt das Objekt der Klasse `Bnn`, definiert die Eingangssignale, ruft `calculate()` auf und erhält die entsprechenden Ergebnissignale zurück.

Dieses UML-Modell des Boolesche Neuronales Netzes dient als ein Muster für den Entwurf von BNN in FPGA. In Einzelfällen können zusätzliche Attribute, Parameter und Methode definiert werden.
Jedes Design-Modell basiert auf einer Design-Plattform, die durch ein Design-Plattformmodell beschrieben ist. Der Inhalt des Design-Plattformmodells hängt vom gewählten Anwendungsgebiet ab. Das Design-Plattformmodell definiert Typen, ihre Wertebereiche und Beziehungen, die für das System-Design verwendet werden. Für jeden Typ werden die Beziehungen zu anderen Typen, die unterstützten Operationen und Wertebereiche definiert.

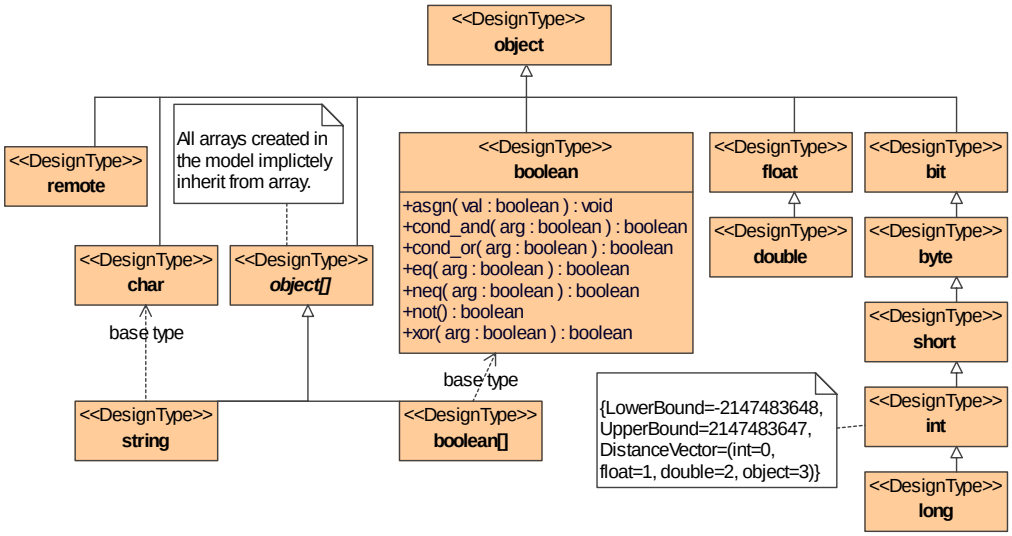


Abbildung 5.13 Design-Plattformmodell: Designtypen

Beispiel 5.4. In der Abbildung 5.13 wird ein Teil des Design-Plattformmodells gezeigt. Dieses Beispiel stellt einige Designtypen dar, die in Design-Modellen verwendet werden können, die auf diesem Modell basieren. Für den Booleschen Typ werden die definierten Operationen angegeben. Man erkennt, dass es sich um einfache logische Operationen handelt. In einem Kommentar wird als Beispiel der Wertebereich für den `int` Typ erläutert. Design-Plattformmodelle enthalten normalerweise auch zusätzliche Typen, z.B. für die Eingabe und Ausgabe.

Das Design-Plattformmodell wurde nicht speziell für eine konkrete Aktionssprache entwickelt. Der Modell-Compiler verwendet dieses Modell zur Überprüfung und Optimierung des Design-Modells. Nutzer können neue Typen und Operationen zur Design-Plattform hinzufügen, die durch den Compiler als primitive Typen behandelt werden. Für diese Elemente kann der Nutzer eine geeignete Implementierung in der Implementation-Plattform angeben.

Implementation-Modell.

Das Implementation-Modell beschreibt eine Realisierung des Design-Modells in Bezug auf Klassen, Komponenten, Artefakte und Beziehungen. Dieses Modell hat die gleiche Funktionsweise wie das Design-Modell, aber eine andere Realisierung. Das Implementation-Modell beschreibt eine mögliche Realisierung der Struktur und des Verhaltens mit den Diensten, die durch das Implementation-Modell bereitgestellt sind. Für ein gegebenes Design-Modell können viele Implementation-Plattformmodelle gegeben werden.

Das Implementation-Modell wird aus einem Design-Modell durch eine Reihe von Transformationen und Abbildungen erzeugt. Jedes Implementation-Modell basiert auf einer spezifischen Implementation-Plattform (specific implementation platform). Durch die Implementation-Plattformen wird die Realisierung der Design-Plattformen beschrieben, wobei jede Implementation-Plattform eine Design-Plattform realisiert. Jede Implementation-Plattform wird durch ein Implementation-Plattformmodell beschrieben. Für jedes Bearbeitungselement in der Hardware-Plattform soll ein Implementation-Plattformmodell definiert werden.

Eine Implementation-Plattform besteht aus einer Menge von Typen, Bedingungen, Transformationen und Tools, die zur Realisierung des Design-Modells verwendet werden können. Wie Design-Plattformen werden Implementation-Plattformen durch eine Menge von Klassen und ihre Eigenschaften beschrieben. Dieses Modell wird durch Modell-Compiler bei der Plattformabbildung, Bewertung und Synthese verwendet.

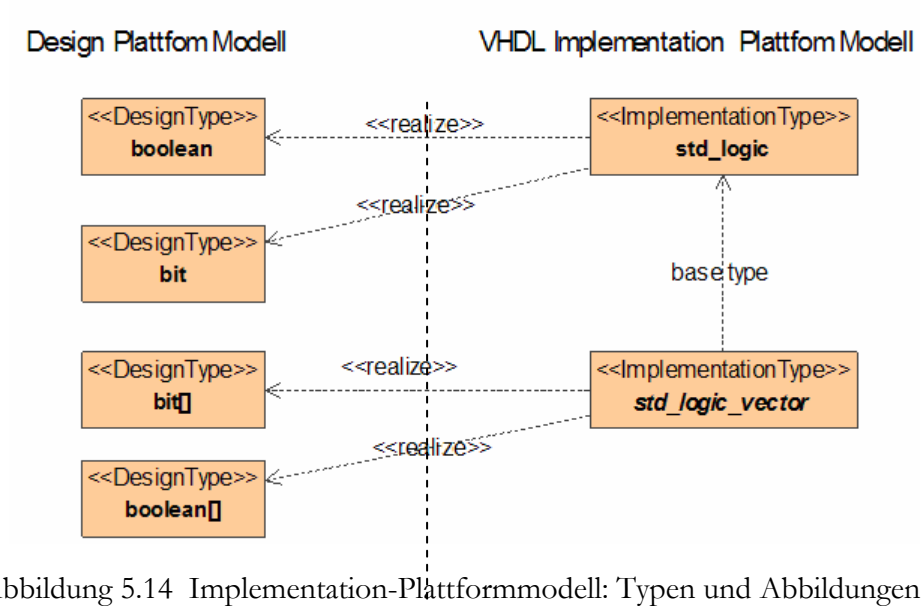


Abbildung 5.14 Implementation-Plattformmodell: Typen und Abbildungen

Beispiel 5.5. Ein Design-Modell kann in Software, Hardware oder eine Mischung von Software und Hardware implementiert werden. MOCCA führt die Transformation vom Design-Modell des BNN aus dem Beispiel 5.3 in das Implementation-Modell automatisch durch. Dabei werden C++ und VHDL-RTL Implementation-Plattformen verwendet. Ein Teil des entsprechenden Implementation-Plattformmodells wird in Abbildung 5.14 gezeigt. Das Diagramm beschreibt Design- und Implementation-Typen, die für die Realisierung von Designtypen verwendet werden.

Deployment-Modell

Ein Deployment-Modell beschreibt die Realisierung des Implementation-Modells in einer bestimmten Hardware-Architektur. Die Knoten sind Mikroprozessoren, rekonfigurierbare logische Geräte oder abstrakte Deployment-Plattformen. Gemäß der UML-Spezifikation kann ein Knoten aus einem Bearbeitungselement (PE-processing element), einem reservierten Speicher und einer Peripherie bestehen. Jedes Anwendungs-Modell basiert auf einer Hardware-Plattform. Eine Hardware-Plattform legt fest, welche Implementation-Plattformen realisiert werden können. Eine Implementation-Plattform kann durch verschiedene Hardware-Plattformen realisiert werden. Eine Hardware-Plattform wird durch das Hardware-Plattformmodell beschrieben.

Durch die Hardware-Plattform werden die Knoten, Datenkommunikationspfade und Einschränkungen einer Hardware-Architektur definiert. Hardware-Plattformen geben keine Mikroarchitektur von Hardware-Knoten an; sie definieren Dienste, die durch die Hardware-Mittel bereitgestellt werden. Zum Beispiel können die Anzahl der Logik- und Speicherelemente, die Taktfrequenz und das Kommunikationsprotokolle beschrieben werden. Das Hardware-Plattformmodell muss genug Information enthalten, um die ge-

wünschte Designqualität zu ermöglichen. Die Informationen des Hardware-Plattformmodells werden zur Parametrisierung von Implementation-Plattformen verwendet.

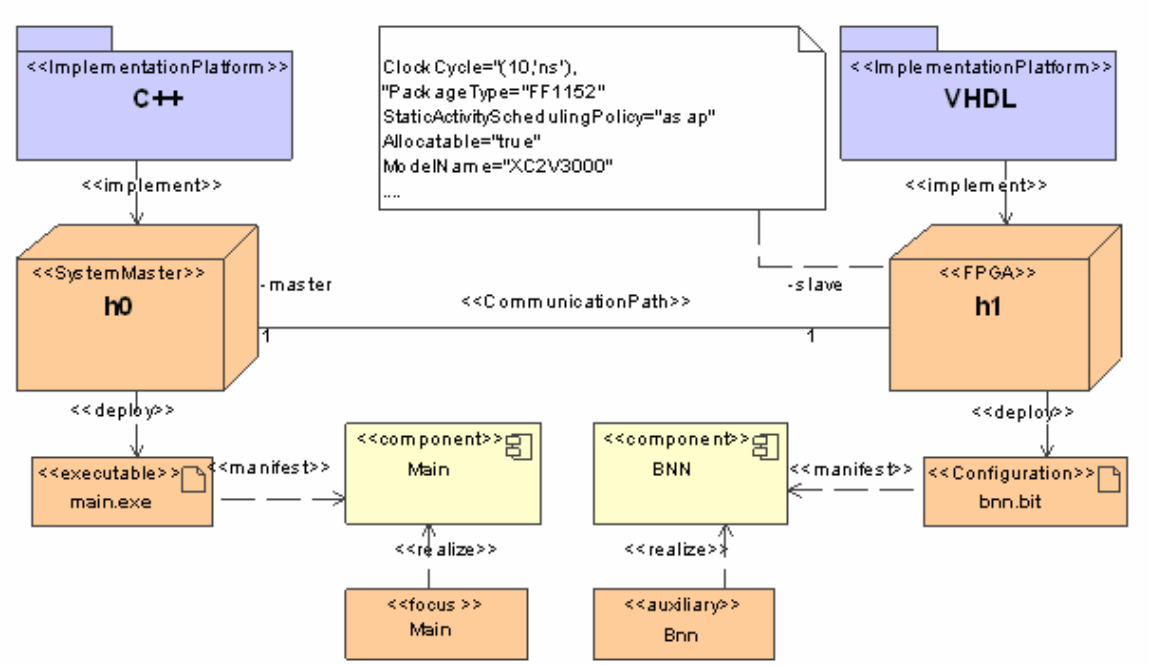


Abbildung 5.15 Deployment-Modell

Beispiel 5.6. Die Abbildung 5.15 zeigt ein Teil eines Hardware-Plattformmodells und ein auf dieser Hardware-Plattform basierendes Deployment-Modell. Die Hardware-Plattform besteht aus zwei Knoten `h0` und `h1`, die durch einen Kommunikationspfad verbunden sind. Artefakte, die auf den Knoten abgearbeitet werden können, werden durch die Implementation-Plattform beschrieben. Das Artefakt `main.exe` ist ein ausführbares Programm für `h0`. Es manifestiert die Komponente, die die `Main`-Klasse enthält. Die Struktur und Logik des BNN werden durch eine Komponente beschrieben, die den Knoten `h1` mit einem Bit-Stream `bnn.bit` konfiguriert.

Die verbundenen Deployment- und Implementation-Modelle ergänzen einander. Das Deployment-Plattformmodell und das Implementation-Plattformmodell werden Ziel-Plattformmodelle genannt (TPM - target platform model). Beide sind Plattform-spezifische Modelle (platform specific model) [15]. Ein Design-Modell kann auf einer konkreten Plattform gemäß dem Deployment-Plattformmodell realisiert werden, wobei das Design-Modell und Design-Plattformmodell nicht geändert werden müssen.

5.3 Hardware/Software-CoDesign

5.3.1 RTR-Manager

Die in der rekonfigurierbaren Hardware realisierten Objekte werden durch einen speziellen Dienst gesteuert, der RTR-Manager [131] genannt wird. Dieser Manager verkapselt die Details der rekonfigurierbaren Hardware. Die wichtigste Aufgabe dieses Dienstes ist das Erzeugen und Löschen von Hardware-Objekten. Auf Anforderung werden Hardware-Objekte erzeugt und gelöscht. Bei der Erzeugung eines Hardware-Objekts ruft der RTR-Manager dieses Objekt durch seinen Typ auf. Der RTR-Manager wählt nach einem entsprechenden Objekt aus dem konfigurierten FPGA und liefert zugeordnetes Proxy-Objekt an die Anwendung.
Die lokalen Proxy-Objekte dienen der Kommunikation zwischen Software- und Hardware-Objekten. Für jedes Hardware-Objekt, auf das durch ein Software-Objekt zugegriffen wird, wird ein lokales Proxy-Objekt realisiert. Das Proxy-Objekt kapselt den Kommunikationsmechanismus und wird in dem Implementation-Plattformmodell modelliert (siehe Abbildung 5.16).

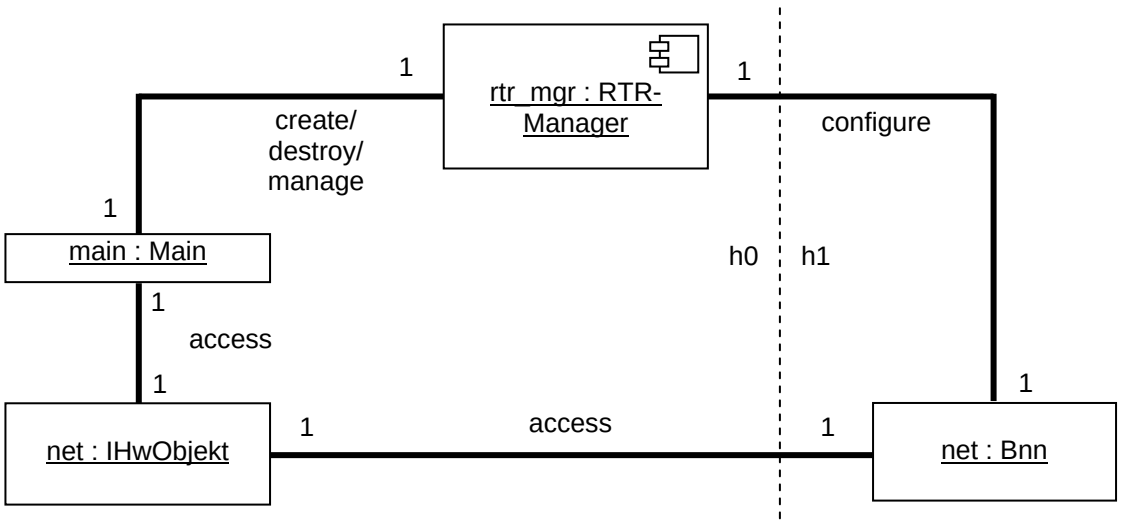


Abbildung 5.16 Software-Architektur von BNN

Beispiel 5.7. Eine grundlegende Architektur des Systems für BNN-Design wird in Abbildung 5.16 gezeigt. Die Instanz `main` der Klasse `Main` und das Proxy-Objekt `net:IHwObjekt` für das Hardware-Objekt werden in der Software realisiert. Das Objekt `net` der Klasse `Bnn` wird durch die rekonfigurierbare Ressourcen eines FPGA realisiert. Jedes Hardware-Objekt wird von der Software durch das Proxy-Objekt aufgerufen. Die Proxy-Objekte werden durch den RTR-Manager bei der Erfüllung ihrer Aufgaben

unterstützt. Die Proxy-Objekte werden für einen einfachen und schnellen Zugriffmechanismus von Software-Objekten zu Hardware-Objekte verwendet.
Der Lebenszyklus und die Zugriffsmechanismen von Objekten und Komponenten, die in der rekonfigurierbaren Hardware zu Realisieren sind, werden durch ein Hardware / Software-Interface definiert. Der Lebenszyklus der Hardware-Objekte unterscheidet sich vom Lebenszyklus der Software-Objekte. Lebenszyklen von Hardware- und Software-Objekten beeinflussen einander.
Da eine dynamische Erzeugung bzw. Zerstörung von Objekten in der Hardware uneffizient ist, werden diese Objekte in der Kompilierungszeit erzeugt und in die Bit-Stream-Konfiguration synthetisiert. Auf Anforderung werden die Hardware-Objekte belegt. Dabei dient der RTR-Manager als Objektvermittler.

5.3.2 Softwaremodul

Entsprechend den Implementation- und Deployment-Modellen transformiert der MOCCA-Compiler UML-Design automatisch in die Hardware- und Software-Modulen. Die auf Mikroprozessor-Knoten zu realisierenden Klassen des Implementation-Modells werden direkt in C++ umgewandelt. Ein Beispiel dazu wird in der Abbildung 5.17 gezeigt.

```
/*** Definition of the operations of Class Data.Main ***/

   int Main::main( void )
   {
      ...
      smartptr<IHwObject> net;
      net = RTRManager::getInstance()->createObject( 0 );
      do {
         ...   /* Initialisation of input variables */
         net->execute<char>( 5, 128 ); /* execute Bnn::init_x(boolean[]) */
         net->execute<char>( 5, 2 ); /* execute Bnn::calculate() */
         …
         if( net->read<bool>( 166 ) ) {
                              net->execute<char>( 4, 128 ); /* execute Bnn::build_y() */ }
         …
      }while( ... );
      RTRManager::getInstance()->destroyObject( &net );
   }
```

Abbildung 5.17 Software-Realisierung von `Main::main`

Beispiel 5.8. Die Abbildung 5.17 zeigt einen Teil der Operation `Main::main` des Softwareteils von eines BNN. Der vollständige Softwareteil der Realisierung von BNN ist im Anhang B.1 zu finden. Zuerst wird ein Hardware-Objekt des Netzes `net` durch den RTR-Manager bereitgestellt. In der Schleife werden die Eingangssignale initialisiert und

dem `net`-Objekt übergegeben. Dann wird die `calculate`-Methode ausgeführt. Wenn die `calculate()` beendet wurde, werden die Ausgangssignalen aus dem `net`-Objekt zurückgelesen. Am Ende der Operation wird das Hardware-Objekt gelöscht.

5.3.3 Hardwaremodul

Das Verhalten der Operationen von Klassen wird gemäß dem FSMD-Modell (Finite State Machine with Datapath) als Moore Automat [55] modelliert. Jedes Verhalten wird als Controller mit einem zugeordneten Datenfluss realisiert. Im Datenflussteil sind Elemente zum Ausführen von Grundoperationen, zur Auswertung von Bedingungen, zur Eingaben, zur Ausgaben, und zur Speicherung von Zwischenergebnissen enthalten. Die Ergebnisse der Bedingungen steuern den Controller. Der Controller wird als Automat realisiert. Jeder Operation des Datenflusses werden bestimmte Zustände des Automaten zugeordnet. Operationen, die in einem Takt ausgeführt werden, sind mit einem Automatenzustand zugeordnet. Multizyklus-Operationen werden mit mehreren aufeinander folgenden Zuständen assoziiert. Mehrere unabhängige Operationen werden gemeinsam in einem Takt ausgeführt. Abhängige Operationen werden in aufeinander folgenden Takten ausgeführt. Zustandsübergänge werden synchron durchgeführt [145].
Im Gegensatz zu Softwarerealisierungen wird ein Verhalten für jedes Hardware-Objekt separat durchgeführt, d.h. für die Objekte mit dem gleichen Verhalten wird eine eigene Realisierung des Verhaltens verwendet. Demzufolge ist keine Synchronisation von gleichzeitigen Ausführungen desselben Verhaltens in verschiedenen Objekten erforderlich. Um Konfliktsituationen zu vermeiden, werden lokale Kopien von Attributen, Eingabe- und Ausgabeparametern gespeichert. Alle Modifizierungen des Datenflusses werden auf den Kopien durchgeführt. Wenn keine ausführliche Ausgabe im Modell angegeben wird, werden diese Kopien am Ende der Berechnung zurück synchronisiert.

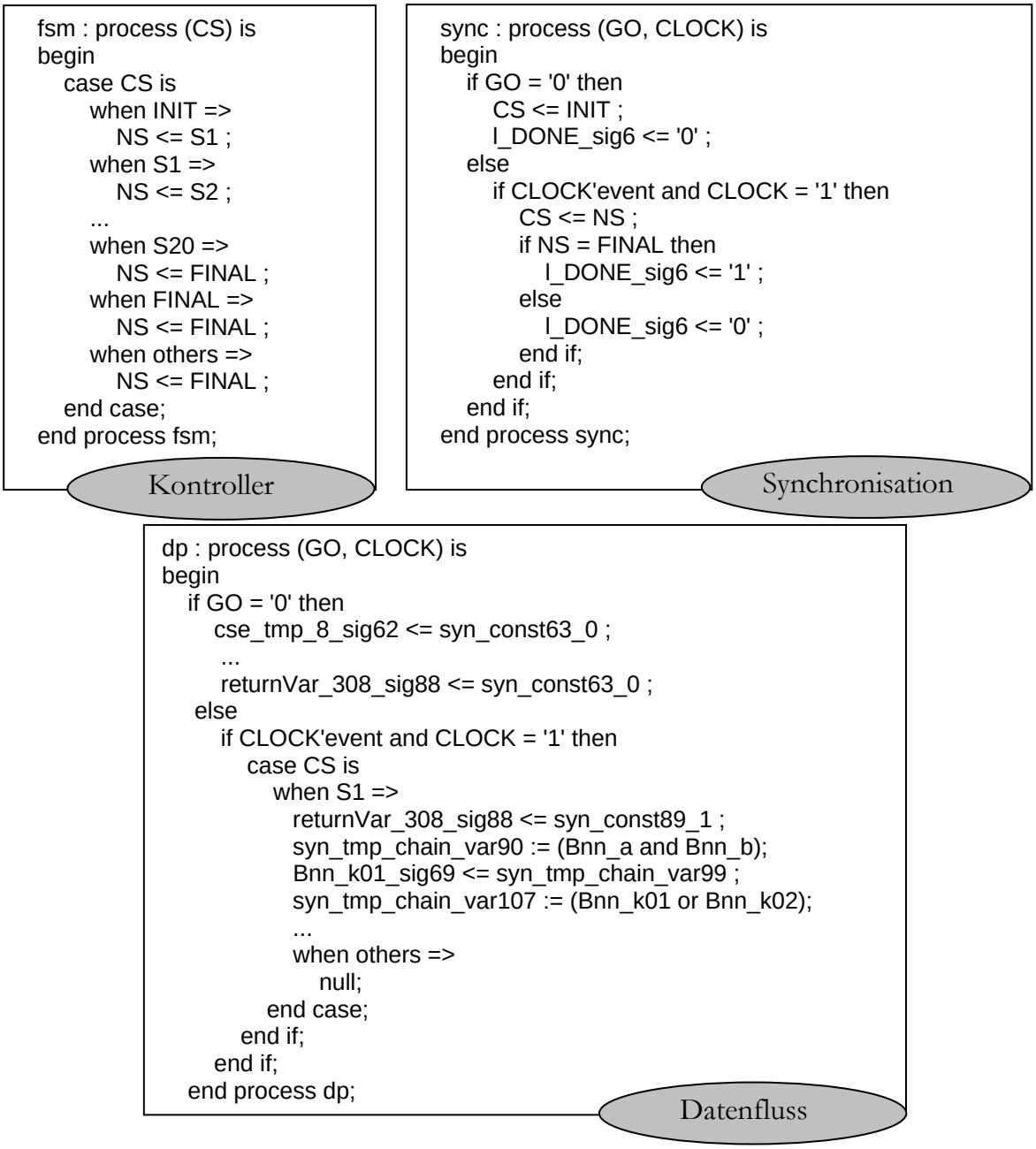


Abbildung 5.18 VHDL-Realisierung der Funktion Bnn::calculate

Beispiel 5.9. In der Abbildung 5.18 wird ein Teil der VHDL-Realisierung der Funktion `Bnn::calculate` gezeigt, der die grundsätzliche Vorgehensweise erkennen lässt. Der vollständige VHDL-Code ist im Anhang B.2 zu finden. Der Automat wird in einen Controller, einen Datenfluss, und einen Synchronisationsprozess zerlegt. Wenn der Automat im Endzustand ist, wird der gegenwärtige Automatenzustand und das DONE Signal synchron gesetzt.

5.4 Bewertung experimentaler Ergebnisse

5.4.1 Experimente

Zur Bewertung der vorgeschlagenen Methode von FPGA-Realisierung Boolescher Neuronaler Netze wurden mehrere Experimente durchgeführt. Um möglichst anschauliche Ergebnisse zu erzielen wurde das schon oben beschriebene einfache BNN aus den Beispielen 5.3-5.9 verwendet. Basierend auf dem in der Abbildung 5.12 dargestellten UML-Modell wurden 5 verschieden Design-Modelle BNN1-BNN5 des gegebenen Netzes entworfen. Die genaue Beschreibung dieser Designs ist im Anhang C.1 zu finden. Die 5 gewählten BNN-Modelle dienen der Bewertung von verschiedenen Synthesekonzepten. Auf diese Weise werden die Vor- und Nachteile der vorgeschlagenen Methode erkennbar. Für jedes Design-Modell wurden zwei Deployment-Modelle entworfen, welche die Realisierung von BNN in `h0` (Mikroprozessor) und `h1` (FPGA) beschreiben. Die Ziel-Plattform bestand aus der C/C++ Implementation-Plattform, der VHDL Implementation-Plattform und der PC-basierenden Deployment-Plattform.

Beispiel 5.10. Im ersten Experiment wurden BNN1-BNN5 zur rekonfigurierbaren Logik manuell zugeteilt, während die `Main`-Klasse zum Mikroprozessor `h0` zugeordnet ist. Als rekonfigurierbare Logik `h1` wurde eine Xilinx Virtex-II FPGA-Karte (Slave) mit etwa 3 Millionen Gatter verwendet. Mikroprozessor `h0` war einen Pentium IV Prozessor (Master). Die Taktfrequenzen sind entsprechend 100 MHz und 2.4 GHz. Der Slave realisiert das `Bnn`-Objekt, und der Master ist für die Eingabe und Übermittlung der Eingangsinformation zum `Bnn`-Objekt und für den Empfang der Ergebnisse von dem `Bnn`-Objekt verantwortlich. Master und Slave sind durch einem PCI 33-MHz-Bus verbunden.
Für das zweite Experiment wurden BNN1-BNN5 in der C/C++ Softwareplattform implementiert. Es ist wichtig zu betonen, dass dafür keine Änderung im Design-Modell des BNN vorgenommen wurde.

Die experimentellen Ergebnisse sind im Anhang C dargestellt. Die Tabellen C.1 und C.2 zeigen die mittlere Kompilationszeiten für die FPGA- und die Software-Realisierungen Boolescher Neuronaler Netze. Die zeitlichen Messungen der Datenkommunikation und der Ausführung von BNNs in dem FPGA werden in Tabellen C.3 und C.4 angegeben. Entsprechende Kommunikations- und Ausführungslatenzen von Software-BNNs sind in Tabellen C.5 und C.6 zu finden. Die Nullwerte in diesen Tabellen zeigen an, dass die jeweilige Operation durch das Design nicht realisiert wurde. Die Tabelle C.7 enthält die durch MOCCA vorab geschätzten Ausführungszeiten für die Operation

`Bnn::calculate()` im FPGA. Quantitative Ergebnisse der FPGA-Realisierungen von allen Designs werden im Anhang C.3 angegeben. Die Tabellen C.8 und C.9 zeigen die Ressourceausnutzung für die Realisierung der Klasse `Bnn` und der Methode `Bnn::calculate()`.

5.4.2 Qualitätsbewertungen

MOCCA bittet umfassende Möglichkeiten zur Optimierung des CoDesigns. Abhängig von dem gewählten Optimierungsgrad variiert die gesamte Zeit der Kompilation/Synthese des Design-Modells in die ausführbaren Hardware/Software-Module. Bei dem in durchgeführten Experimenten eingestellten höchsten Optimierungsgrad wurden akzeptable Kompilation/Synthesezeiten erreicht. Für das Netz aus unserem Beispiel beträgt die Synthesezeit abhängig von dem Optimierungsgrad 2-27 Sekunden. Ein Vergleich der durchschnittlichen Kompilationszeiten von MOCCA für FPGA- und Software-Implementierungen wird in Abbildung 5.19 gezeigt. Der Parameter t_{sum} gilt als die gesamte Kompilationszeit des Designs und ist die Summe von folgenden Werten:

t_{opt} – Optimierungszeit,

t_{map} – Zeit der Plattformabbildung,

t_{syn} – Synthese-Zeit.

Die Softwarekompilation ist ca. zweimal schneller als die Hardware-Synthese. Ausführliche Angaben zu Kompilationszeiten für FPGA- und Software-Realisierungen von BNNs sind im Anhang C.1 (Tab. C.1, C.2 und Abb. C.1, C.2) zu finden.

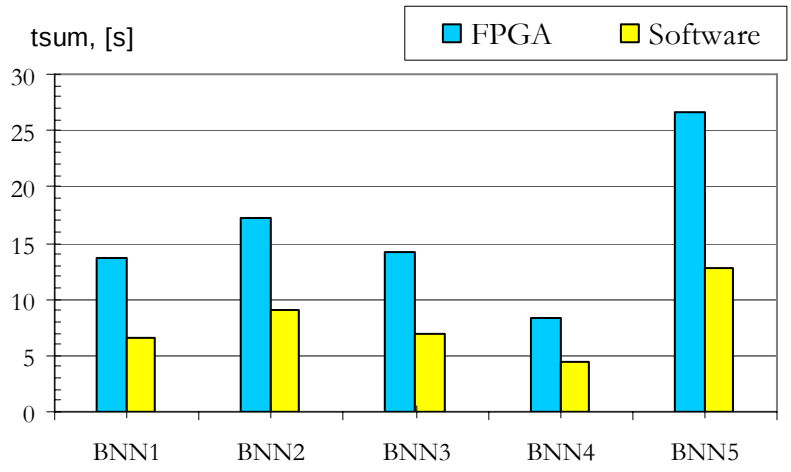


Abbildung 5.19 Mittlere Kompilationszeiten von Design-Modellen zur FPGA- und Software-Implementierungen.

Die gesamte Kompilationszeit des Designs hängt natürlich von der Kompliziertheit des UML-Modells ab. In zahlreichen Experimenten wurde festgestellt, dass die Kompilation bis zu 5 Minuten benötigt.

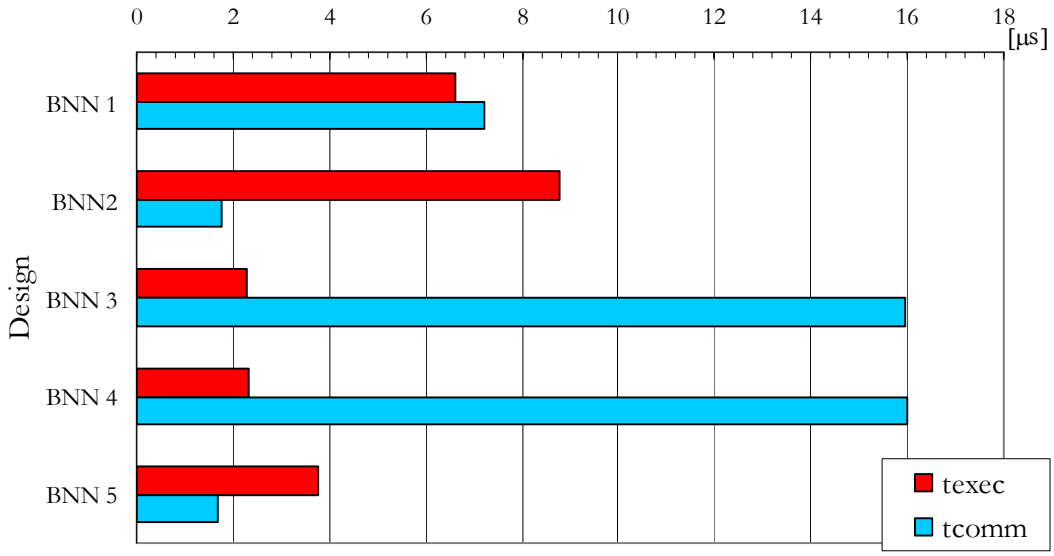


Abbildung 5.20 Kommunikations- und Ausführungszeiten von BNNs in FPGA.

Nach der Kompilation/Synthese aller BNN-Designs wurden die Kommunikations- und Ausführungszeiten der implementierten BNNs gemessen. In Abbildung 5.20 werden die gemessenen Zeitwerte für FPGA-Realisierungen von BNNs dargestellt.
Alle Zeitmessungen wurden aus der Software heraus vorgenommen. Die Ausführungszeit t_{exec} ist die Summe von Ausführungszeiten aller angerufenen Operationen, einschließlich den Kommunikationsaufwand für das Auslösen und den Test von GO/DONE Signalen:

$$t_{exec} = t_{exec,init\_x} + t_{exec,calculate} + t_{exec,get\_y} \tag{5.3}$$

Kommunikationszeit t_{comm} besteht aus dem Zeitaufwand zur Datenübertragung zu und von dem Netz (5.4):

$$t_{comm} = t_{write,x} + t_{read,y} \tag{5.4}$$

Da die Kommunikationslatenz durch den PCI-Bus im Vergleich zu der Leistung von FPGA größer ist, sind Kommunikationszeiten von BNN1, BNN3 und BNN4 größer als ihre Ausführungszeiten. In den Designs BNN2 und BNN5 werden Eingangs- und Ausgangssignale in eine 32-Bits Integer-Variable kodiert und dementsprechend ist der Zeitaufwand für die Ausführung größer als für die Kommunikation. Zur Kodierung und Dekodierung von der Signalvektoren hat das BNN2 zwei Operationen `init_x()` und `get_y()`. Im BNN5 wird die Kodierung und Dekodierung von Signalvektoren direkt in

der Methode `calculate()` durchgeführt. Da der Aufruf jeder Operation zu einem zusätzlichen Kommunikationsaufwand führt, ist BNN5 mehr als doppelt so schnell wie BNN2.
Die Kommunikations- und Ausführungslatenzen von den BNNs, die als Software implementierten wurden, sind in Abbildung 5.21 dargestellt. Detaillierte Messwerte werden im Anhang C.2 (Tab. C.5 und Tab. C.6) angeführt.

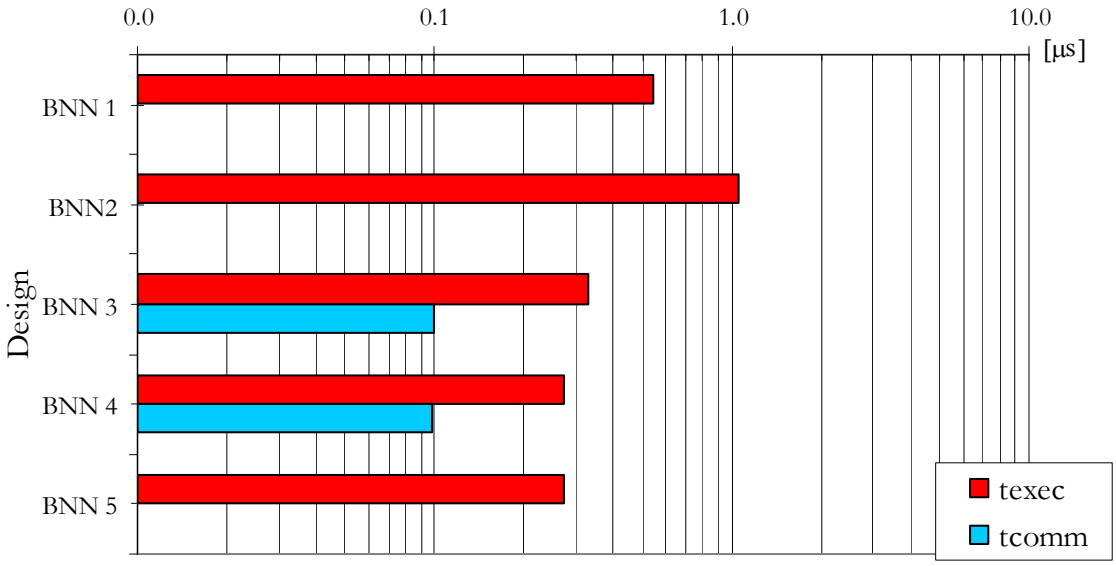


Abbildung 5.21 Kommunikations- und Ausführungszeiten von BNNs in Software.

Wie auch bei der FPGA-Realisierung enthält t_{exec} den Teil des Kommunikationsaufwandes. Zeit t_{comm} zeigt den Zeitaufwand der gesamten Kommunikation, die durch alle modellierten Variabelenzuweisungen verursacht werden. Für alle Designs bei der Software-Implementierung ist die Kommunikationszeit kleiner als die Ausführungszeit. In BNN1, BNN2 und BNN5 ist diese Latenz unbedeutend, weil nur Zeiger oder Integer-Werte übertragen werden. Die Abbildung 5.21 enthält deshalb für diese BNN keine Zeitangaben für die Kommunikation.

Der größte Teil des Zeitaufwandes t_{exec} für Hardwareteil, die in den Softwareteil gemessen wurde, wird bei der Kommunikation benutzt. Die tatsächliche Zeit, die für die Berechnung des gesamten Netzes in FPGA benutzt wird, ist im Vergleich zu t_{exec} viel kleiner. Wenn der ganze Kommunikationsaufwand berücksichtigt wird, ist die Ausführungszeit von BNNs für Software-Plattform etwa 5-8 Mal kürzer als für die FPGA-Plattform. Andererseits, wenn die reinen Ausführungszeiten des Mikroprozessors und der rekonfigurierbaren Logik verglichen werden, ergeben sich umgekehrte Verhältnisse.

In der Tabelle C.7 (Anhang C.2 Zeitmessungen) werden die tatsächlichen Ausführungszeiten für die Methode `calculate()` angeführt, die durch den direkt in der rekonfigurierbaren Logik integrierten Logik-Analysator gemessen wurden. Die minimale Ausführungszeit beträgt 50 ns. Da die Logiktiefe des Netzes 7 ist und die Latenz jeder LUT auf dem verwendeten FPGA ≈ 7.14 ns beträgt, ist die Ausführungszeit in 50 ns optimal ($7 \cdot 7.14\text{ns} \approx 50\text{ns}$).

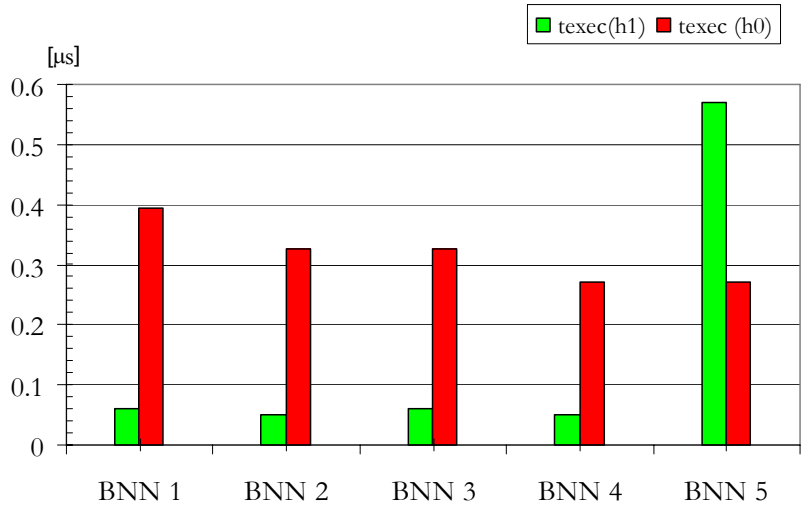


Abbildung 5.22 Ausführungszeiten von `Bnn::calculate()` in FPGA und Software.

Die gleiche Ausführungszeit der Methode `calculate()` für Designs BNN1-BNN4 zeigt, dass die FPGA-Realisierungen von `calculate()` gleichwertig ist. Die längere Ausführungszeit für im BNN5 bestätigt die kompliziertere Realisierung `calculate()` Methode in diesem Design. Dagegen sind die Ausführungszeiten der Methode `calculate()` in Software-Implementierung fast gleichmäßig, was erklärt die sehr ähnliche Software-Implementierung von `calculate()`. Für die BNN1-BNN4 ist die Ausführung von `Bnn::calculate()` in FPGA ca. fünfmal schneller als in Software und für das BNN5 benötigt der FPGA zweimal mehr Zeit als der Mikroprozessor.

5.4.3 Quantitative Bewertungen

Zur effektiven Bewertung von FPGA-Realisierungen Boolescher Neuronaler Netze werden die quantitativen Parameter von generierten Schaltungen ermittelt. Aus dem vorangehenden Abschnitt ist bekannt, dass die Designs BNN1-BNN4 eine Logiktiefe von 7 haben. Im Anhang C.3 werden die weiteren geschätzten quantitativen Parameter angegeben. Der benötigte FPGA-Bereiche für die Realisierung der Klasse `Bnn` wird in der Abbildung 5.23 dargestellt, wobei gilt:

#LUT – Anzahl der belegten LUTs,

#FSM - Anzahl der Zustände aller Automaten,
#FF - Anzahl der verwendeten Flip-Flops.

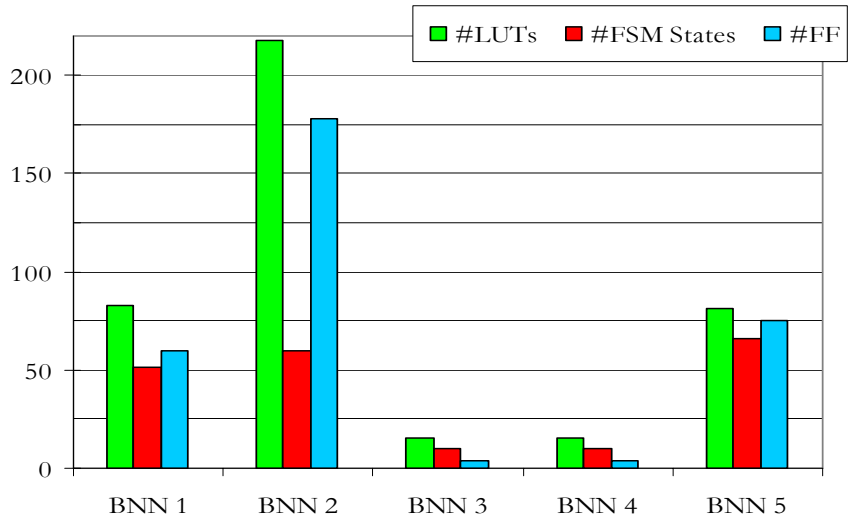


Abbildung 5.23 Verwendung des FPGA-Bereiches durch `Bnn`.

Die Anzahl von LUTs, Flip-Flops sowie die Anzahl der Automatenzustände sind für alle Realisierungen verschieden. Der Grund dafür besteht in verschieden Realisierungen der Datenübertragung, der Kodierung und der Dekodierung von Signalvektoren und der Beschreibung Klasse `Bnn` in den BNN-Designs. Die Flip-Flops werden zur Speicherung der Eingangs-, Ausgangsdaten und der Zwischenwerte benutzt. Die Verwendung von Vektoren zur Kodierung und Dekodierung der Eingangs- und Ausgangssignale in Integer-Variablen in BNN1, BNN2 und BNN5 führt zur Vergrößerung der Anzahl erforderlicher FF.
Da die Realisierung von Designs mit weniger Automatenzuständen weniger Ressourcen benötigt, kann die Anzahl von Zuständen als Kompliziertheitsmaß des entsprechenden Designs verwendet werden. Aus der Sicht der Kompliziertheit sind BNN3 und BNN4 die besten Designs aus den fünf verglichen UML-Modellen. Designs mit einer größerer Anzahl von Zuständen enthalten Schleifen oder Bedingungen, durch die eine kompliziertere Zustandlogik entsteht (BNN1, BNN2 und BNN5).

Die meisten Ressourcen werden durch die Methode `calculate()` verwendet, die nach dem Inlining aller *k*- und *y*-Funktionen die volle Struktur des BNN abbildet. Durch den höchsten Optimierungsgrad, der für die Experimente in MOCCA eingestellt wurde, haben die meistens Designs eine gleichwertige FPGA-Realisierungen von `calculate()`. Nur die Realisierung des Designs BNN5 ist ca. fünfmal aufwendiger als die Anderen (BNN1-BNN4) und dementsprechend werden mehr LUTs und Flip-Flops benötigt. Das

wurde dadurch verursacht, dass die Methode `calculate()` im BNN5 nicht nur die Struktur des Netzes, sondern auch die Kodierung und Dekodierung von Signalvektoren enthält. Der durch `calculate()` verwendete FPGA-Bereiche wird in der Abbildung 5.24 dargestellt.

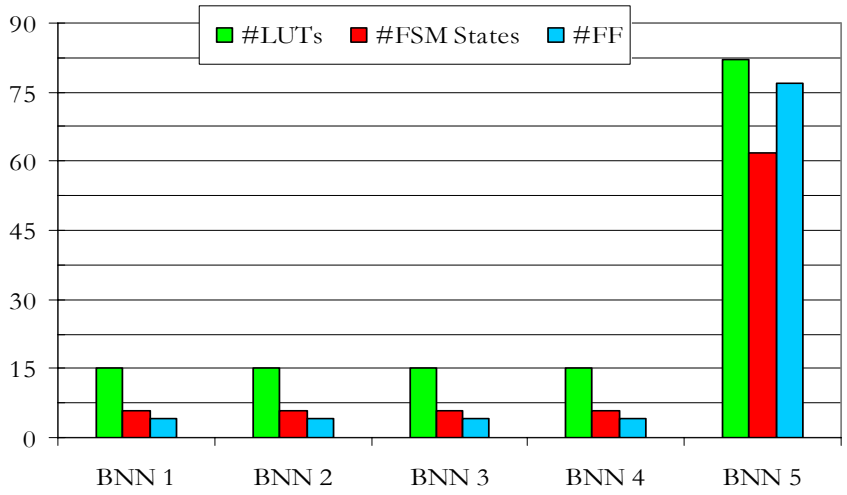


Abbildung 5.24 Verwendung des FPGA-Bereiches durch `Bnn::calculate()`

Da nur fünfzehn LUTs mit jeweils 4 Eingängen zur Realisierung der Netzstruktur verwendet werden, ist solches BNN-Design und Realisierung in FPGA optimal. 14 Booleschen Neuronen werden durch 12 LUTs realisiert. Die zwei Neuronen in der Ausgangschicht (y10 und y1) wiederholen die Ausgangsignale von verborgenen Neuronen k1 und k4. Deswegen wird keine zusätzliche LUT zur Abbildung dieser Ausgangneuronen benutzt. Während die 12 LUTs die Logik von Booleschen Neuronen des Netzes abbilden, werden drei LUTs zur Realisierung der Logik des Automaten benutzt. Diese Realisierung des Automaten ist noch nicht optimal und kann weiter verbessert werden. In mehreren Experimenten wurde die erforderliche Anzahl von LUTs für Realisierung der Automaten durch Optimierungseinstellungen des Modellcompilers auf 1 minimiert. Doch in einigen Fällen genügt eine LUT nicht, um den sequentiellen Ablauf in FPGA zu realisieren. Dieser Fakt ist aber kein Nachteil von BNN, weil die Realisierung der Logik jedes Booleschen Neuron in FPGA höchstens eine LUT erfordert. Die Verbesserung der Realisierung der Steuerautomaten kann ein weiterer Entwicklungsschritt von MOCCA sein.

Die Analyse der Logik von einzelnen LUTs sowie der ganzen Schaltung, die aus dem VHDL-Code generiert wird, ermöglicht ein spezielles Synthese-Tool. In dieser Arbeit wurde dafür Xilinx Synthese Tool (XST) benutzt. XST erlaubt eine Visualisierung von logischen Elementen der Schaltungen, die für jede einzelne Methode sowie das ganze Ob-

jekt generiert werden. Im Schaltplan werden alle logische Elemente des FPGA und die Verbindungen zwischen den Elementen dargestellt, die für die Realisierung der ausgewählten Methode oder des `Bnn`-Objektes verwenden werden.

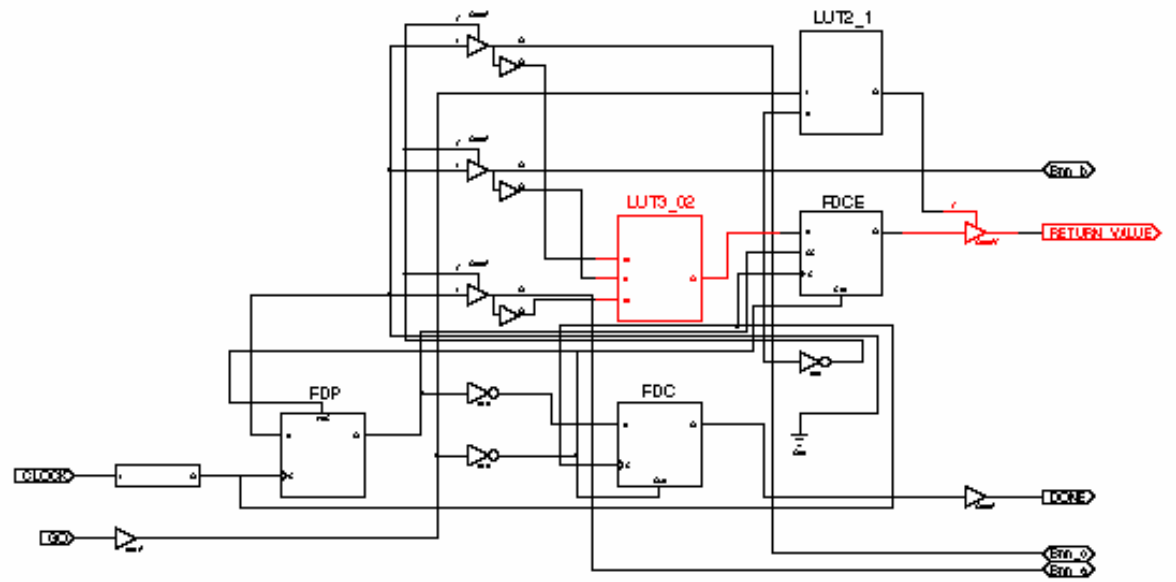
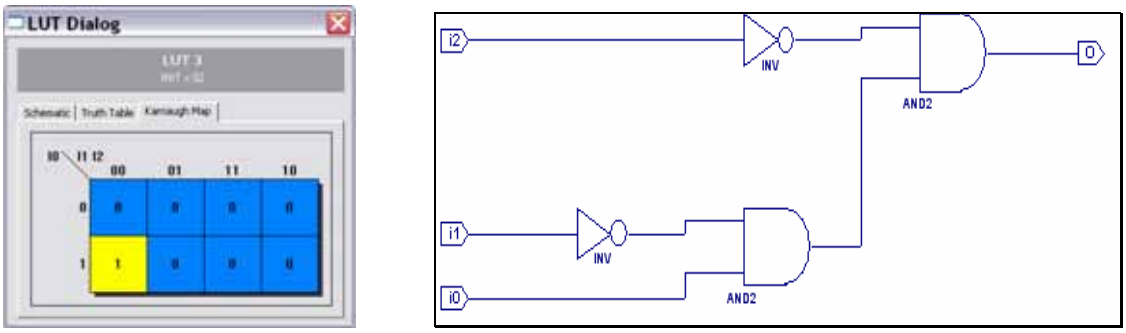


Abbildung 5.25 Schaltplan für `k3()`

Beispiel 5.11. Als Beispiel wird ein Schaltplan für die Methode `k3()` des `Bnn`-Objektes generiert und in der Abbildung 5.25 wiedergegeben. Dieses Schema stellt eine optimale Realisierung sowohl der Logik des Neurons, als auch der Logik des Automaten dar. Während die LUT3_02 (in rot markiert) ein Boolesches Neuron mit der Transferfunktion k_2 in der verborgenen Schicht des BNN abbildet, dient die LUT2_1 der Bestimmung des Nachfolgezustandes des Automaten. Der in der Abbildung 5.26 dargestellte Karnaugh-Plan (links) und der Schaltplan (rechts) der LUT3_02 beweisen die Korrektheit der Realisierung des BN `k3()`.



a) b)

Abbildung 5.26 Karnaugh-Plan und Schaltplan von LUT3_02 für das BN `k3()`

Die Schaltpläne für die Funktionen `y0-y9` werden im Anhang C.4 Abbildungen C.7-C.17 dargestellt.

In diesem Kapitel wurde eine neu Methode zur Realisierung Boolescher Neuronaler Netze in rekonfigurierbaren Rechenarchitekturen vorgestellt. Die Beschreibung von BNN durch UML-Modelle gewährleistet eine einfache, schnelle und flexible Spezifikation von verschiedenen BNN-Designs. Ein plattformunabhängiges Design-Modell kann durch den MOCCA-Compiler in ausführbare Hardware/Software Module sehr schnell und effizient automatisch transformiert werden. Dabei ist es von der großer Bedeutung, dass verschiedene BNN-Realisierungen schnell erzeugt werden können, ohne das Systemdesign ändern zu müssen. Dafür wurde eine nützliche Kombination des MDA-Konzepts und des Hardware/Software CoDesigns angewendet.
Die Ergebnisse der durchgeführten Experimente zeigen, dass die Boolesche Neuronale Netze besonders für die Hardware-Realisierung auf FPGA-basierten rekonfigurierbaren Rechenarchitekturen geeignet sind. FPGA-Strukturen erlauben eine hohe Parallelität bei den Berechnungen und eine schnelle Rekonfigurierbarkeit bei der Realisierung eines anderen BNN-Designs. Durch die Verwendung von Booleschen Neuronen in Neuronalen Netzen für die Abbildung Boolescher Funktionen in FPGA wird die Anzahl erforderlicher CLB für die Realisierung jedes Neurons deutlich vermindert. Jedes Boolesche Neuron mit seiner Logik kann direkt in eine einzige LUT abgebildet werden, was einen großen Vorteil im Vergleich zu bekannten FPGA-Realisierungen von Neuronen ist. Die Struktur des trainierten Netzes wird eins-zu-eins in das FPGA, als ein logisches Netzwerk von LUTs, abgebildet.
Um eine hohe Anschaulichkeit zu erreichen, wurden alle Beispiele in diesem Kapitel mit einem kleinen und sehr einfachen Netz erläutert. Weitere Experimente komplizierteren Logikdesigns für BNNs zeigen ähnliche Ergebnisse. Obwohl die FPGA-Realisierung von Booleschen Neuronalen Netzen im Vergleich zu Software-BNN viel schneller sind, hat die Kommunikation zwischen Objekten einen großen Einfluss auf die Berechnungsgeschwindigkeit. Der uneffektive Datenaustausch zwischen dem Mikroprozessor und der rekonfigurierbaren Logik und die hohe Kommunikationslatenz ist ein Problem. Zur Vergrößerung der Systemleistung sollten die Objekte auf die verschiedenen Architekturelemente so aufgeteilt werden, so dass die Kommunikation über den PCI-Bus minimiert wird. Das ist besonders wichtig, wenn die Kommunikationslatenzen die Ausführungslatenzen der Operation überschreiten. Im Systemdesign sollten bevorzugt alle oft angefragte Daten lokal gespeichert werden. Die Anzahl von Methoden, auf die von entfernten Objekten zugegriffen werden, aber die keinen angemessenen Betrag zur Arbeit leisten, sollte minimal sein.

Kapitel 6

Zusammenfassungen

In der vorliegenden Arbeit wurden die Darstellungsmöglichkeiten Boolescher Funktionen durch Neuronale Netze untersucht und eine neue Art von Neuronalen Netzen – Boolesche Neuronale Netze - entwickelt. Das Basiselement Boolescher Neuronaler Netze ist ein neuartiges Boolesches Neuron, das direkt mit Booleschen Signalen operiert und dafür ausschließlich Boolesche Operationen benutzt. Für das Training und die Anwendung der BNN wurden geeignete Algorithmen und Verfahren erarbeitet. Eine neue Methode zum Aufbau von BNN mit einem sequentiellen Training wurde vorgestellt. Es wurde eine Variante zur Architektursynthese der BNN entwickelt.

Da das BNN einen sequentiellen Trainingsalgorithmus benutzt, treten Probleme iterativer Trainingsmethoden wie z.B. lokale Minima, lange Trainingszeit etc. nicht auf. Es wird eine schnelle Konvergenz des Trainingsalgorithmus für BNN garantiert. Das Netz findet die optimale Netzwerkstruktur und die optimalen Netzparameter selbst. Die entwickelten BNN besitzen bedeutende Vorteile im Vergleich zu bekannten Booleschen Neuronalen Netzen, die sowohl iterative als auch sequentielle Trainingsalgorithmen (STA) verwenden. Der beim Training notwendige Speicherbedarf wurde deutlich vermindert. Die Trainingszeit bleibt angemessen. Durch die Verwendung Boolescher Neuronen mit Booleschen Transferfunktionen wird auch die Berechnungszeit für die Konvertierung des Eingangsvektors in das Ausgangssignal wesentlich reduziert.

Mit dem entwickelten Trainingsalgorithmus des BNN steht ein spezielles Dekompositionsverfahren Boolescher Funktionen bereit. Eine Menge von Booleschen Funktionen $A=\{y_1(\boldsymbol{x}), y_2(\boldsymbol{x}), ..., y_{Ny}(\boldsymbol{x})\}$, $\boldsymbol{x}=(x_1, x_2, x_3,\ldots, x_n)$ wird beim Training in gemeinsame einheitliche Boolesche Basisfunktionen $k_1(\boldsymbol{x}), k_2(\boldsymbol{x}), ..., k_{Nz}(\boldsymbol{x})$ dekomponiert, die als Transferfunktionen der verborgenen Neuronen betrachtet werden. Die Booleschen Basisfunktionen hängen vom Vektor der Eingabensignale $\boldsymbol{x}$ ab. Eine Verknüpfung der ermittelten k-Funktionen durch Basisoperation $\Omega\in\{\wedge, \vee, \oplus, \odot\}$ bildet die gegebene Boolesche Funktion oder Funktionsmenge. Aus den k-Funktionen werden alle gegebenen Booleschen Funktionen gebildet.

Da das Boolesches Neuron nur die Booleschen Signale bearbeitet und nur die Boolesche Operationen dafür benutzt, sind die Boolesche Neuronale Netze besonders geeignet für die Hardware-Realisierung in FPGA-basierten RTR-Systemen. Durch die Verwendung des Booleschen Neurons für die Modellierung Boolescher Funktionen in FPGA wird das Problem der ansonsten großen Anzahl von erforderlichen CLB für die Abbildung jedes Neurons gelöst. Die Anzahl der für die Realisierung eines Neurons notwendigen LUTs wurde auf 1 reduziert. Folglich können in einem CLB eines Virtex2-FPGAs sogar 4 Boolesche Neuronen realisiert werden. Jedes Boolesche Neuron mit seiner eigenen Logik kann dabei direkt in eine LUT abgebildet werden, was ein großer Vorteil im Vergleich zu bekannten FPGA-Realisierungen von BNN ist [34], [63], [64], [80], [176] und [177].
Die entwickelten Booleschen Neuronalen Netze haben sich als effiziente Modelle Boolescher Funktionen bewährt. Sie gewährleisten die Lösung der meisten klassischen Probleme der Modellierung Boolescher Funktionen durch Neuronale Netze. Anderseits ist ein Boolesches Neuronales Netz mit seinen Booleschen Neuronen kein „Allheilmittel“, das alle Probleme löst. Das Spektrum der Aufgaben, die durch die aus Booleschen Neuronen aufgebauten Neuronalen Netze gelöst werden können, beschränkt sich auf das Gebiet der Booleschen Logik. Das heißt, die Ein- und Ausgabedaten des zu modellierenden Objektes müssen in Boolesche Werte umgewandelt werden, um eine Anwendung von BNN zu ermöglichen.
Alle in der Arbeit vorgestellten Algorithmen und Verfahren wurden programmiert und getestet. Basis dafür bilden die Algorithmen zum Training der BNN. Mit den klassischen Neuronen und Booleschen Neuronalen Netzen wurden im Kapitel 3 einige neue Lösungen für das Problem der linearer Separierbarkeit formuliert. Dieses Kapitel trägt sehr zum Verständnis der Probleme bei, die bei der Darstellung Boolescher Funktionen durch Neuronale Netze auftreten. Den Schwerpunkt dieser Arbeit bilden das Boolesche Neuron und die Booleschen Neuronalen Netze. Dazu wurde in Kapitel 4 die Struktur des Booleschen Neurons angegeben und eine exakte mathematische Beschreibung spezifiziert. Danach wurde gezeigt, wie die Booleschen Neuronalen Netze aus Booleschen Neuronen aufgebaut werden. Hierzu wurden Trainings- und Anwendungsalgorithmen für BNN, ihre Strukturen und Eigenschaften beschrieben. Ein neues Dekompositionsverfahren für eine Menge Boolescher Funktionen wurde am Beispiel einer AND-Dekomposition gezeigt. Eine Anwendung der Hardware-Realisierungen der entwickelten Booleschen Neuronalen Netze in RTR-Systemen wurde im Kapitel 5 vorgestellt. Für eine äußerst kompakte Abbildung der BNN in eine FPGA-Struktur wurde der Trainingsalgorithmus des BNN angepasst und eine Darstellung des BNN durch die UML-Modelle anhand MDA-Technologie für das Hardware/Software-Codesign verwendet.

Zusammenfassend lässt sich feststellen, dass die entwickelten Booleschen Neuronalen Netze einen Beitrag zum Entwurf komplexerer Boolescher Schaltungsnetzwerke leisten. Durch die Verwendung Boolescher Neuronen für den Aufbau Boolescher Neuronaler Netze wurde eine kompakte Darstellung und eine schnelle Berechnungsweise Boolescher Funktionen geschaffen. Dadurch ist eine effiziente Hardware-Realisierung der BNN in FPGA-Strukturen möglich. Die Fähigkeit der Künstlichen Intelligenz bzw. der Neuronalen Netze, neue Erkenntnisse zu erzeugen, wird erfolgreich bei der Modellierung Boolescher Funktionen angewendet. Durch die Verwendung Boolescher Neuronen und Neuronale Netze ergibt sich eine neue Abbildungsmöglichkeit Boolescher Funktionen in FPGA.

In dieser Arbeit wurden Boolesche Neuronale Netze als effizientes Modell Boolescher Funktionen vorgestellt. Für die zukünftige Entwicklung Boolescher Neuronaler Netze und deren Anwendung zur Modellierung Boolescher Funktionen kann es darüber hinaus von Bedeutung sein,

- neue Architekturen und Paradigmen, wie z.B. FB-Netze, Kohonen-Karte [77], Hopfield-Netze [71] etc., basierend auf dem Booleschen Neuron zu erarbeiten. Dies kann die Anwendung von BNN zur Lösung von Aufgaben, wie z.B. der Klassifizierung, Mustererkennung, Vorhersage oder Optimierung ermöglichen.
- die Verwendung von TVL für die Darstellung innerer Daten in Trainingsalgorithmen Boolescher Neuronaler Netze. Dadurch könnte die Berechnungsleistung von BNN noch erhöht werden.
- die Trainingsalgorithmen für die Anwendung der BNN zur Dekomposition Boolescher Funktionsmenge weiter zu verbessern.

Basierend auf Ergebnisse dieser Arbeit sollten weitere Forschungen in folgenden Richtungen fortgesetzt werden:

- Abbildungsmethoden beliebiger BNN-Strukturen in FPGA mit und ohne On-Chip-Training.
- Erhöhung der Abbildungseffizienz Boolescher Neuronaler Netze in FPGA-Strukturen basierend auf dem Hardware/Software-CoDesign für RTR-Architekturen.
- Optimierung der Beschreibung von Booleschen Neuronalen Netzen durch UML-Modelle.

LITERATURVERZEICHNIS

[1] Aizenberg, I. Solving the Parity n Problem and other Nonlinearly Separable Problems Using a single Universal Binary Neuron. Advances in Soft Computing. Springer Series In book "Computational Intelligence, Theory and Application" (B. Reusch – Ed.) (Proceedings of the 9th International Conference on Computational Intelligence), Springer, Berlin, Heidelberg, New York, pp. 457-471, 2006.

[2] Aizenberg, I., Aizenberg, N., Krivosheev, G. Multilayered and Universal Binary Neurons: Learning algorithms, applications to image processing and recognition. In Lectures Notes in Artificial Intelligence, Berlin: Springer-Verlag, Vol.1715: Machine Learning an Data Mining in Pattern Recognition – Proc of the First International Workshop MLDM'99, Leipzig, Germany, September 1999.

[3] Aizenberg, I., Aizenberg, N., Vandewalle, J. Multy-Valued and Universal Binary Neurons: Theory, Learning and Applications. Boston, MA, Kluwer, 275 pp., ISBN 0-7923-7824-5, 2000.

[4] Aizenberg, I., Moraga, C. and Paliy, D. A Feedfoeward Neural Network based on Multi-Valued Neurons. In Computational Intelligence, Theory and Applications. Advances in Soft Computing , XIV, (B. Reusch - Ed.), Springer, Berlin, Heidelberg, New York, pp. 599-612, 2005.

[5] Amaldi, E. and Kann, V. The complexity and approximability of finding maximum feasible subsystems of linear relations. Theoret. Comput. Sci. 147, pp. 181-210, 1995.

[6] Amaldi, E. From finding maximum feasible subsystems of linear systems to feed forward neural network design, PhD thesis No. 1282, Department of Mathematics, Swiss Federal Institute of Technology, Lausanne, 1994.

[7] Amaldi, E. On the complexity of training perceptrons. Artificial Neural Networks, Elsevier Science Publishers B.V., Amsterdam, pp. 55-60, 1991.

[8] Amaldi, E., Bertrand, G. Two Constructive Methods For Designing Compact Feedforward Networks Of Threshold Units.Int. J. of Neural Systems, Vol.8, No.5-6, pp. 629-645, 1998.

[9] Amaldi, E., Diderich, C. and Hauser, R. On the probabilistic and thermal perceptron algorithms," manuscript, 1998.

[10] Arora, S., Babai, L., Stern, J. and Sweedyk, Z. The hardness of approximate optima in lattices, codes, and systems of linear equations. J. Comput. Syst. Sci. 54, pp. 317-331, 1997.

[11] Bade, S., Hutchings, R. FPGA-based stochastic neural networks-implementation. In Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp. 189-198, 1994.

[12] Barthel, R. Grundlagen einer Booleschen Signaltheorie. Wiss. Schriftenreihe der Technischen Hochschule Karl-Marx-Stadt 13/1984, Sektion Informationstechnik 112p., 1984.

[13] Bartlett, P., Shai, B-D. Hardness results for neural network approximation problems.Theoretical Computer Science Volume 284 , Issue 1, pp. 53-66, 2002.

[14] Bartlett,P., Maass, W. Vapnik-Chervonenkis dimension of neural nets. In The Handbook of Brain Theory and Neural Networks. pp. 1188-1192. MIT Press, 2003.

[15] Beierlein, T., Fröhlich, D. and Steinbach, B. UML-Based Co-Design of Reconfigurable Architectures. In Proceedings of the Forum on Specification and Design Languages (FDL'03), Frankfurt a.M., Germany, 2003.

[16] Beierlein, T., Fröhlich, D., Steinbach, B. Object-Oriented Co-Design for Run-Time Reconfigurable Architectures with UML. in: Computer - Aided Design of Discrete Devices - CAD DD 2004, Proceedings of the Fifth International Conference , Minsk, Belarus, Volume 1, pp. 23 – 30, 2004.

[17] Beierlein, T., Fröhlich, D., Steinbach, B. UML-Based Co-Design for Run-Time Reconfigurable Architectures. In Proceedings of the FORUM on Specification and Design Languages, pp. 285 - 296, Frankfurt, Germany, September 23 - 26, 2003.

[18] Bishop, C. Neural Networks for Pattern Recognition. New York: Oxford University Press, 1995.

[19] Biswas, N. and Kumar, R. A new algorithm for learning representations in Boolean neural networks.Current Science Vol.59, No. 12, 25 Jun, 1990.

[20] Blake, J., Maguire, L., McGinnity, T., Roche, B., and McDaid, L. The Implementation of Fuzzy Systems, Neural Networks and Fuzzy Neural Networks using FPGAs, Information Sciences: An International Journal, Elsevier, pp. 151-168, No. 112, 1998.

[21] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, K., Learnability and the Vapnik-Chervonenkis Dimension", J. ACM, Vol. 36, pp. 929-965, 1989.

[22] Bobda, Ch. Synthesis of Dataflow Graphs for Reconfigurable Systems Using Temporal Partitioning and Temporal Placement. PhD thesis, Paderborn University, Germany, April 2003.

[23] Bochman, D. Einführung in die strukturelle Automatentheorie. Berlin: Verlag Technik, München: Hanser-Verlag, 1975.

[24] Bochman, D., Posthoff, Ch. Binäre dynamische Systeme. Berlin: Akademie-Verlag, München; Oldenburg-Verlag, Moskau: Energoatomisdat, 1981.

[25] Bochman, D., Steinbach, B., Logikentwurf mit XBOOLE. Verlag Technik, Berlin, 1991.

[26] Bochmann, D., Dresig, F., Steinbach, B. A new decomposition method for multilevel circuit design. The Proceeding of "The European Conference on Design Automation", Amsterdam, The Netherlands, pp. 374 - 377, 1991.

[27] Booch, G., Rumbaugh, J., Jacobson, I. Das UML - Benutzerhandbuch, Aus dem Amerikan. Von Kahlbrandt, B. und Reder, D. - Bonn Addison-Wesley-Longman, 561p., 1999.

[28] Borgelt, C., Klawonn, F., Kruse, R., Nauck, D. Neuro-Fuzzy-Systeme. 3.Auflage Vieweg Verlag, 434 p. ISBN 3-528-25265-0, Wiesbaden, 2003.

[29] Bottou, L., Vapnik, V. Local learning algorithms. Neural Computation, Vol. 4, pp. 888-900, 1992.

[30] Brown, S., Francis, R., Rose, J. and Vranesic, Z. Field-Programmable Gate Arrays. Kluwer Academic Publishers, 228 p., 1992.

[31] Bryant, R. Graph-Based Algorithm for Boolean Function Manipulation, IEEE Trans. Comp., vol C-35, pp. 677-69, 1986.

[32] Caianiello, E. Outline of the theory of thought processes and thinking machines. J. of Theor. Biol., 2, pp. 204-235, 1961.

[33] Campbell, C. Constructive learning techniques for designing neural network systems. In Neural Network Systems, Techniques and Applications ed. C. T. Leondes, Academic Press, San Diego, 1997.

[34] Charles, E.C., Blanz, W.E. GANGLION – a fast field-programmable gate array implementation of a connectionist classifier. IEEE Journal of Solid-State Circuits, 27(3), pp. 288-299, March, 1992.

[35] Chaudhari, N., Tiwari, A. Extending ETL for multi-class output. International Conference on Neural Information Processing, 2002 (ICONIP'02). In, Proc. Computational Intelligence for E-Age, Asia Pacific Neural Network Asscoication (APNNA), Singapore, pp.1777-1780, 2002.

[36] Chu, A. A neural-based boolean function generator. Int. J. Electronics, vom. 74, no. 1, pp. 21-34, 1993.

[37] Chuanyj, J., Demetri, P. Capacity of two-layer feedforward neural networks with binary weights. IEEE Trans. Info. Theory, vol. IT-44, no. 1, pp. 256-268, Jan. 1998.

[38] Cloutier J., Cosatto E., Pigeon S., Boyer F. and Simard P. VIP: an FPGA-based processor for image processing and neural networks. In Proc. MicroNeuro, pp. 330-336, 1996.

[39] Compton, K., Hauck, S. An Introduction to Reconfigurable Computing, Invited Paper, IEEE Computer, April, 2000.

[40] Davis, L. Handbook of genetic algorithms, Von Nostrand Reinhold, 1991.

[41] Deolalikar, V. Mapping Boolean functions with neural networks having binary weights and zero thresholds. IEEE Tran. on Neural Networks 12(4), pp.1-8, 2001.

[42] Deolalikar, V. New approaches to learning and classification in feedforward neural networks, M. Tech. Dissertation, Indian Institute of Technology, Bombay, July 1994.

[43] Diuk, B., Samoylenko, A. Data Mining: Lehrkurs. Piter Press, 368p., 2001.

[44] Drechsler, R. Ordered Kronecker Functional Decision Diagrams und ihre Anwendung. Reihe Informatik. Modell Verlag Karben, 1996.

[45] Drechsler, R., Becker, B. Graphenbasierte Funktionsdarstellungen, B. G. Teubner Stuttgart, 1998.

[46] Drechsler, R., Sarabi, A., Theobald, M., Becker, B., Perkowski M. Efficient representation and manipulation of switching functions based on Ordered Kronecker Functional Decision Diagrams, in Design Automation Conf., pp.415-419, 1994.

[47] Dresig, F., Kümmling N., Steinbach, B., Wazel, J. Programmieren mit XBOOLE, Wissenschaftliche Schriftenreihe Technische Universität Chemnitz, Chemnitz, 1992.

[48] Eldredge, J., Hutchings, B. RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs. .In Proceedings of the IEEE World Conference on Computational Intelligence, 1994.

[49] Franco, L. A measure for the complexity of Boolean functions related to their implementation in neural networks. http://www.citebase.org/, 25 p., submitted on Nov. 9. 2001.

[50] Forcada, M., Carrasco, R. Finite-state computation in analog neural networks: steps towards biologically plausible models? Emergent Neural Computational Architectures Based on Neuroscience, Lecture Notes in Computer Science, Springer 2036, pp.487-501, 2001.

[51] Frean, M. A "thermal" perceptron learning rule. Neural Computation 4(6), pp. 946-957, 1992.

[52] Frean, M. Small nets and short paths. PhD thesis, Department of Physics, University of Edinburgh, Scotland, 1990.

[53] Frean, M. The upstart algorithm: A method for construction and training feedforward neural networks. Neural Computation, 2(2), pp.198-209, 1990.

[54] Gajski, D. and Vahid, F. Specification and Design of Embedded Hardware-Software Systems. IEEE Design and Test of Computers, pages 53–66, 1995.

[55] Gajski, D.D. Principles of Digital Design. Prentice Hall Inc., 1997.

[56] Gallant, S. Perceptron-based learning algorithms," IEEE Trans. Neural Networks 1, pp. 179-191, 1990.

[57] Gallant, S. Three constructive algorithms for network learning. In Proc. 8th Ann Conf of Cognitive Science Soc. Amherst, pp. 652-660, August 1986.

[58] GCC, the GNU Compiler Collection. http://gcc.gnu.org/

[59] Girau, B., LORIA. Dependencies of composite connections in Field Programmable Neural Arrays, NeuroCOLT2 Technical Report Series NC-TR-99-001, June, 1999.

[60] Golea, M., Marchand, M. A growth algorithm for neural network decision trees. Europhysics Letters, 12(3) pp. 205-210, 1990.

[61] Gray, D., Michel, A. A training algorithm for binary feed forward neural networks. IEEE Tran. on Neural Networks 3(2), pp.176-194, 1992.

[62] Gschwendtner A. B. DARPA Neural Network Study. AFCEA International Press, 60 p., 1988.

[63] Gschwind, M., Salapura, V., Maischberger, O. A Generic Building Block for Hopfield Neural Networks with On-Chip Learning. IEEE International Symposium on Circuits and Systems, Atlanta, GA, May 1996.

[64] Gschwind, M., Salapura, V., Maischberger, O. Space efficien neural net implementation. In Proc. of the Second International ACM/SIGDA workshop on Field-Programmable Gate Arrays, Berkeley, CA, February 1994.

[65] Gupta, R., Micheli, G. Readings in hardware/software co-design. Giovanni de Micheli, Rolf Ernst, Wayne Wolf, chapter Hardware-Software Cosynthesis for Digital Systems, pages 5–17. Volume of Micheli [40], 2002.

[66] Hand, D., Mannila H. and Smyth P. Principles of Data Mining (MIT Press), 2001.

[67] Hebb, D. The Organization of Behavior. New York City, Wiley Publications, 1949.

[68] Henkel, J., Ernst, R., Benner, T. Readings in hardware/software co-design / Giovannide Micheli, Rolf Ernst, Wayne Wolf, chapter Hardware-Software Cosynthesis for Microcontrollers, pages 18–29. Volume of Micheli [40], 2002.

[69] Hertwig, A., Brück, R. Entwurf digitaler Systeme, chapter 5.3 Field Programmable Gate Arrays, pages 112–134. Carl Hanser Verlag München Wien, 2000.

[70] Hertz, J., Krogh, A., Palmer, R. Introduction to the Theory of Neural Computation, Addison-Wesley, Reading, Mass., 1991.

[71] Hopfield, J. Neural networks and physical systems with emergent collective computational abilities. In Proc. Natl. Acad. Sci. USA, №79, pp. 2554-2558, 1982.

[72] IBM/Rational. The Rational Unified Process, http://www-306.ibm.com/software/awdtools/rup/, January 2004.

[73] Kebschull, U., Schubert, E., Rosenstiel, W. Multilevel Logic Synthesis Based on Functional Decision Diagrams, Proc. EDAC, pp. 43-47, 1992.

[74] Kim, J., Ham, B. and Park, S.-K. The learning of multi-output binary neural networks for handwriting digit recognition. Proc. of 1993 International Joint Conference on Neural Networks (IJCNN), 1, pp.605-508, 1993.

[75] Kim, J., Park, S.-K. The geometrical learning of binary neural networks. IEEE Tran. Neural Networks 6(1), pp.237-247, 1995.

[76] Kim, J., Roche, J. Covering cubes by random half cubes, with applications to binary neural networks. Journal of Computer and System Science 56(2), pp.223-252, 1998.

[77] Kohonen, T. Self-organized formation of topologically correct feature maps. Biological Cybernethics. v.43. pp. 56-69, 1982.

[78] Kohut, R. Modeling of Boolean Functions using Neural Networks. Proceeding of Graduiertenkolleg "Space Statistic". Reports for DFG, Freiberg, pp. 107-120, 2003.

[79] Kohut, R., Steinbach, B. Decomposition of Boolean Function Sets for Boolean Neural Networks. in: Steinbach, B. (Hrsg.): Boolean Problems, Proceedings of the 5th International Workshops on Boolean Problems, Freiberg, Germany, September, 2004.

[80] Kohut, R., Steinbach, B. The Structure of Boolean Neuron for the Optimal Mapping to FPGAs. The Experience of Designing and Application of CAD Systems in Microelectronics - Proceedings of the VIII-th International Conference CADSM 2005. Lviv – Polyana, Ukraine, 2005.

[81] Kohut, R., Steinbach, B., Fröhlich D. FPGA Implementation of Boolean Neural Networks using UML Boolean Problems, Proceedings of the 7h International Workshops on Boolean Problems, September 2006 Freiberg University of Mining and Technology, Freiberg, 2006.

[82] Kohut, R., Steinbach, B. Boolean Neural Networks. In WSEAS Transactions on Systems, Issue 2, Volume 3, pp. 420 - 425, April 2004.

[83] Kröse, B., v.d. Smagt, P. An introduction to Neural Networks. University of Amsterdam, 1996.

[84] Kühnrich, M. Ternärvektorlisten und ihre Anwendung auf binäre Schaltnetzwerke. Dissertation A. Karl-Marx-Stadt: Tchnische Hochschule, 1979.

[85] Lang, Ch. Bi-Decomposition of Function Sets using Multi-Valued Logic. Dissertation thesis, Freiberg University of Mining and Technology, Germany, 2003.

[86] Lang, Ch. Mehrfachnutzung von Schaltungsteilen bei der dekompositorischen Synthese von kombinatorischen Schaltungen. Diploma thesis, Technische Universität Chemnitz, Germany, 1995.

[87] Lang, Ch., Steinbach, B. Decomposition of Multi-Valued Functions into Min- and Max-Gates. in: Proceedings of the 31st IEEE International Symposium on Multiple-Valued Logic, Warsaw, Poland, pp. 173 - 178, May 22-24, 2001.

[88] Lauria, F., Sette, M., Visco, S. Adaptable Boolean neural networks - Consorzio Editoriale Fridericiana, Liguori ed. Napoli, 212 p. ISBN 88-207-2676-9, 1997.

[89] Le, T. A-Dekomposition und ihre Anwendung bei der Synthese mehrstufiger Schaltungen. In Workshop Boolesche Probleme, Freiberg, Germany, pp. 81-89, Okt. 1994.

[90] Ma, X., Yang, Y. and Zhang, Z. Constructive learning of binary neural networks and its application to nonlinear register synthesis. Proc. of International Conference on Neural Information Processing (ICONIP)'01 1, pp.90-95, 2001.

[91] Ma, X., Yang, Y. and Zhang, Z. Research on the learning algorithm of binary neural networks. Chinese Journal of Computers 22(9), pp.931-935, 1999.

[92] Marchand, M., Golea, M. On learning simple neural concepts: From halfspace intersections to neural decision lists. Network, vol. 4, pp. 67–85, 1993.

[93] Marchand, M., Golea, M., Rujan, P. A Convergence Theorem for Sequential Learning in Two-Layer Perceptrons. Europhysics Letters, 11 1990, pp. 487–492, 1990.

[94] Martinelli, G., Mascioli, F., Bei, G. Cascade Neural Network for Binary Mapping. IEEE Trans. On neural networks, Vol. 4, No. 1, Jan. 1993.

[95] Mascioli, F., Martinelli, G. A constructive algorithm for binary neural networks. the oil_spot algorithm. IEEE Trans. On neural networks, Vol. 6, No. 3, May. 1995.

[96] Mayoraz, E. Feedforward Boolean Networks with Discrete Weights: Computational Power and Training. PhD thesis, Swiss Federal Institute of Technology, Departament of Mathematics, 1993.

[97] Mayoraz, E., Aviolat, F. Constructive training methods for feedforward neural networks with binary weights. Int. J. of Neural Systems, vol.7, no.2, pp. 149-66, May 1996.

[98] Mayoraz, E.On the power of networks of majority functions. In A. Prieto, editor. Lecture Notes in Computer Science 540, pp. 78-85. IWANN'91, Springer-Verlag, 1991.

[99] McCulloch, W.S., Pitts, W. A logical calculus of ideas immanent in nervous activity, Bull. Mathematical Biophysics, vol. 5. pp. 115-133, 1943.

[100] Meinel, C., Theobald, T. Algorithmen und Datenstrukturen im VLSI-Design.OBDD - Grundlagen und Anwendungen. Springer-Verlag, Berlin Heidelberg NewYork, 283p., 1998.

[101] Mellor, S. and Balcer, M. Executable UML - A Foundation for Model Driven Architecture. Addison Wesley Longman Publishers, 2002.

[102] Mertens, S., Engel, A. Vapnic-Chervonenkis dimension of neural networks with binary weights. Physical Review E 55(4), pp.4478-4491, 1997.

[103] Mezard, M., Nadal, J.-P. Learning in Feedforward Layered Networks: the Tiling Algorithm. Journal of Physics A, 22 (1989), pp. 2191–2204, 1989.

[104] Micheli, G., Gupta, R. Readings in hardware/software co-design. Giovanni de Micheli, Rolf Ernst, Wayne Wolf, chapter Hardware-Software Co-Design, pages 30–44. Volume of Micheli [40], 2002.

[105] Minsky, M., Papert, S. Perceptrons. 2nd edition, MIT Press, Cambridge, MA, 1988.

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

[107] Mishchenko, A., Files, C. Perkowski, M., Steinbach, B., Dorotska, Ch. Implicit Algorithms for Multi-Valued Input Support Minimization. in: Steinbach, B. (Hrsg.): Boolean Problems, Proceedings of the 4th International Workshops on Boolean Problems, 2000.

[108] Mishchenko, A., Steinbach, B., Perkowski, M. Bi-Decomposition of Multi-Valued Relations. The 10th International Workshop on Logic & Synthesis (IWLS'01), Granlibakken (California) USA, pp. 35 – 40, June 12-15, 2001.

[109] Mkrttschjan, S. Neurons and neural networks - Introduction in the theory of formal neurons. Energy, Moscow, 1971.

[110] MOCCA Project. The MOCCA-compiler for run-time reconfigurable architectures. http://www.htwm.de/lec/mocca, Web-Pages, Aug. 2006.

[111] Morgan, P., Ferguson, A. and Bolouri H. Cost-performance analysis of FPGA, VLSI and WSI implementations of a RAM-based neural network. Proc. 4th IEEE Int. Conf. on Microelectronics for Neural Networks and Fuzzy Systems (MicroNeuro'94), pp. 235-243, 1994.

[112] Müller, B., Reinhardt, J., Strickland, M. Neural Networks An Introduction. Sprigeg-Verlag Berlin Heidelberg New York, 330 p.,1995.

[113] Muselli, M. On Sequential Construction of Binary Neural Networks. IEEE Transactions on neural networks, Vol. 6, No. 3, May 1995.

[114] Muselli, M. Sequential constructive techniques. In Optimization Techniques, vol. 2 of Neural Network Systems, Techniques and Applications. Ed. C. Leondes, New York: Academic Press, pp. 81–144, 1998.

[115] Nadal, J.-P. Study of a growth algorithm for a feedforward network. International J. of Neural Systems, 1(1) pp. 55-59, 1989.

[116] Object Management Group - Architecture Board ORMSC (2001). Model driven architecture - a technical perspective (MDA). http://www.omg.org, Web-Pages, 2004.

[117] Object Management Group. OMG Unified Modelling Language Specification. Version 2.0, July 2006. http://www.omg.org, Web-Pages, 2005.

[118] Osowski, S. Sieci neuronowe w vięciu algorytmicznym. Wydawnictwa Naukowo-Techniczne, Wydanie drugie, 349 p., Warszawa, 1996.

[119] Pao, Y.-H. Adaptive Pattern Recognition and Neural Networks. Massachusetts: Western Reserve University, 1989.

[120] Parberry, I. Circuit Complexity and Neural Networks. MIT Press, Cambridge, Massachusetts, 304 p., 1994.

[121] Park, S.-K., Kim, J. and Chung, H.-S. A training algorithm for discrete multilayer perceptions. Proc. of IEEE International Symposium on Circuits and Systems, (ISCAS)'91, Singapore, pp.2140-2143, 1991.

[122] Park, S.-K., Kim, J. Geometrical learning algorithm for multiplayer neural network in a binary field. IEEE Trans. Computers 42(8), pp.988-992, 1993.

[123] Park, S.-K., Marston, A. and Kim, J. On the structural requirements of multiplayer perceptions in binary field. 24th Southeastern Symp. on System Theory & 3rd Annual Symp. on, Comm., Signal Proc. and ASIC VLSI Design, Proc. SSST/CSA 92, pp.203-207, 1992.

[124] Parker, D. Learning Logic, Technical Report TR-87, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA, 1985.

[125] Patterson, D. Künstliche Neuronale Netze: Das Lehrbuch Prentice Hall, 506p., 1996.

[126] Perkowski, M. A new representation of strongly unspecified switching functions and its application to multi-level AND/OR/EXOR synthesis. In Second Workshop on Applications of Reed-Muller Expansion in Circuit Design, Chiba City, Japan, pp.143-151, 1995.

[127] Posthoff, Ch., Bochmann, D., Haubold, K. Diskrete Mathematik, BSB B. G. Teubner Verlagsgesellschaft, Leipzig, 1986.

[128] Posthoff, Ch., Steinbach, B. Binäre dynamische Systeme - Algorithmen und Programme. Wissenschaftliche Schriftenreihe der Technischen Hochschule, Karl-Marx-Stadt, Heft 8, 1979.

[129] Posthoff, Ch., Steinbach, B. Binäre Gleichungen - Algorithmen und Programme. Wissenschaftliche Schriftenreihe der Technischen Hochschule, Karl-Marx-Stadt, Heft 1, 1979.

[130] Posthoff, Ch., Steinbach, B. Logic Functions and Equations. Binary Models for Computer Science. Springer Dordrecht, The Netherlands, 392p., 2004.

[131] Riedel, H. Design and Implementation of a Runtime Enviroment for RTR-Systems. Diploma Thesis, January 2004. University of Applied Sciences Mittweida, Germany, 2004.

[132] Rosenblatt, R. Principles of Neurodynamics. Spartan Books, New York, 1962.

[133] Rujan, P., Marchand, M. Complex Systems 3, pp. 229-242 1989. Proceedings of IJCNN 1989, Washington D.C, Vol II. pp. 105-110, 1989.

[134] Rumelhart D., Hinton G., Williams R. Learning representations by back-propagating errors. Nature, 1986, vol. 323., pp. 533-536, 1986.

[135] Rupp, C., Hahn, J., Queins, S., Jeckle, M., Zengler, B. UML 2 glasklar. Praxiswissen für die UML-Modellierung und –Zertifizierung, 2. Auflage, Hanser Verlag München, 559p., 2005.

[136] Salapura, V., Gschwind, M. and Maischberger, O. A Fast FPGA Implementation of a General Purpose Neuron. In Field-Programmable Logic: Architectures, Synthesis and Applications 4th International Workshop on Field Programmable Logic and Applications, Lecture Notes in Computer Science 849, Springer Verlag, Berlin, 1994.

[137] Sasao, B., Butler, J. On Bi-decompositions of logic functions. International Workshop on Logic Synthesis, Lake Tahoe, USA, pp. 18-21, May 1997.

[138] Scherer, A. Neuronale Netze: Grundlage und Anwendungen. Vieweg Verlag, 249 S. ISBN 3-528-05465-4, Braunschweig-Wiesbaden, 1997.

[139] Shimada, M., Saito, T. A GA-Based Learning Algorithm for Binary Neural Networks. IEICE Trans. Fundamentals, Vol. E85-A, No. 11, Nov., 2002.

[140] Shimada, M., Saito, T. A simple learning of Binary Neural Networks with virtual teacher signals. Proc. IEEE/INNS IJCNN, Vol. 3, pp. 2024-2047, 2001.

[141] Sirat, J., Nadal, J.-P. Neural trees: a new tool for classification. Network, 1, pp. 423-438, 1990.

[142] Siu, K., Roychowdhury, V., Kailath, T. Discrete Neural Computation: A Theoretical Foundation, Prentice Hall Information and System Sciences, Englewood Cliffs, New Jersey, 407 p., 1995.

[143] Starzyk, J.A., Pang, J. Evolvable binary artificial neural network for data classification. In Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, Nevada, USA, 2000.

[144] Steinbach, B. XBOOLE - A Toolbox for Modeling, Simulation, and Analysis of Large Digital Systems. System Analysis and Modeling Simulation, Gordon & Breach Science Publishers, 9(1992), Number 4, pp. 297-312, 1992.

[145] Steinbach, B., Beierlein, T. and Fröhlich, D. Hardware/Software Codesign of Reconfigurable Architectures Using UML. in: Grant, M.; Müller, W.: UML for SOC Design, Springer, Dordrecht, Printed in The Netherlands, pp. 89 – 117, 2005.

[146] Steinbach, B., Beierlein, T., Fröhlich, D. UML-Based Co-Design for Run-Time Reconfigurable Architectures. in: Grimm, Ch.: Languages for System Specification, Kluwer Academic Publishers, Boston, Dordrecht, London, Printed in The Netherlands, pp. 5 – 19, 2004.

[147] Steinbach, B., Fröhlich, D., Beierlein, T. Hardware/Software Codesign of Reconfigurable Architectures Using UML. in: Grant, M., Müller, W. UML for SOC Design, Springer, Dordrecht, Printed in The Netherlands, pp. 89 – 117, 2005.

[148] Steinbach, B., Kempe, G. Minimization of AND-ExOR expressions. Proceedings of "IFIP WG 10.5 - Workshop on Applications of the Reed- Muller Expansion in Circuit Design", Hamburg, pp. 20–26, 1993.

[149] Steinbach, B., Lang, Ch. A General Data Structure for EXOR - Decomposition of Sets of Switching Function. in Proceedings of the 3 rd International Workshops on Boolean Problems, Freiberg, pp. 59 - 66, Sep. 1998.

[150] Steinbach, B., Lang, Ch. Exploiting Functional Properties of Boolean Functions for Optimal Multi-Level Design by Bi-Decomposition. in Yanushkevich, S. N.: Artificial Intelligence in Logic Design, Kluwer Academic Publisher, Dordrecht, The Netherlands, pp. 159–200, 2004.

[151] Steinbach, B., Lang, Ch., Perkowski, M. Bi-Decomposition of Discrete Function Sets. in 4th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design 1999 (Reed-Muller 99), Victoria, Canada, pp. 233 - 252, 1999.

[152] Steinbach, B., Perkowski, M., Lang, Ch., Bi-Decomposition of Multi-Valued Functions for Circuit Design and Data Mining Applications. in Proceedings of the 29th IEEE International Symposium on Multiple-Valued Logic, Freiburg, pp. 50 - 58, May, 1999.

[153] Steinbach, B., Zakrevskij, A. New Approaches for Minimization of ESOPs. Proceedings of the IEEE Design and Diagnostics of Electronic Circuits and Systems (5th International Workshop IEEE DDECS 2002), Brno, Czech Republic, pp. 336 - 339, 2002.

[154] Steinbach, B., Kohut, R. Neural Networks – A Model of Boolean Functions. In Proceedings of 5th International Workshop on Boolean Problems, Freiberg, Germany, pp. 191-206, September 19-20, 2002.

[155] Sung, S.-K., Jung, J., Lee, J.-T. and Choi, W.-J., Opitonal Synthedid Method for Binary Neural Network Using NETLA. Lecture Notes in Artificial Intelligence (LNAI), Vol.2275, pp.236-244, 2002.

[156] Suyari, H., Matsuba, I. New information theoretical approach to the storage capacity of neural networks with binary weights Proc. 9th Int. Conf. On Artificial Neural Networks: ICANN '99, vol. 1, pp. 431-436, Sept. 1999.

[157] Thayse, A. Boolean differential calculus. Heidelberg: Springer-Verlag, 1981.

[158] The Italian Managing Node of NeuroNet, http://www.dibe.unige.it/neuronet/, Web-Pages, 2004.

[159] Tkachenko, R., Kohut, R. Feed forward neural networks: the problems of synthesis and using. Bulletin of Lviv Polytechnic National University, Computer Engineering and Information Technologies, № 433, Lviv, pp. 166-171, 2001.

[160] Tkachenko, R., Kohut, R. Functional extension of inputs in feed forward neural network with non-iteration learning. Technical news 1(12), 2(13), Lviv, pp. 91-94, 2001.

[161] Tkachenko, R. Feed forward neural networks with non-iteration learning procedure. Habilitation thesis, Lviv, 2000.

[162] Vapnik, V., Chervonenkis, A.. On the uniform convergence of relative frequencies of events to their probabilities. Theory of Probability and its Applications, 16(2), pp.264--280, 1971.

[163] Wang, D., Chaudhari, N. A multi-core learning algorithm for binary neural networks. In Proceedings of the International Joint Conference on Neural Networks (IJCNN '03) (Portland, USA) 1, pp.450-455, 2003.

[164] Wang, D., Chaudhari, N. An Approach for Construction of Boolean Neural Networks Based on Geometrical Expansion, Neurocomputing, vol. 57, no. 4, pp. 455-461, 2004.

[165] Wang, D., Chaudhari, N. Binary Neural Network Training Algorithms Based On Linear Sequential Learning, International Journal of Neural Systems, 13(5) Oct. 2003.

[166] Wasserman, P. Neural Computing Theory and Practice. Van Nostrand Reinhold, New York, 1989.

[167] Werbos, P. The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting, Wiley, 1994.

[168] Wegener, I. The Complexity of Boolean Functions. Johann Wolfgang Goethe-Universität, Stuttgart: Teubner, Chichester, New York, Brisbane, Toronto, Singapore: Wiley, 470 p., 1989.

[169] Widner, R. Single-stage logic. AIEE Fall General Meeting, 1960, in Wasserman, P. Neural Computing, Theory and Practice, Van Nostrand Reinhold, 1989.

[170] Widrow, B. The speed of adaptation in adaptive control system, American Rocket Society Guidance Control and Navigation Conference, pp.1933-1961, 1961.

[171] Widrow, B. A statistical theory of adaptation. Adaptive control systems. New York: Pergamon Press, 1963.

[172] Widrow, B., Hoff, M.E. Adaptive switching circuits. IRE WESCON Convention Record, part 4, pp. 96-104. New York: Institute of Radio Engineers, 1960.

[173] Widrow B., Lehr M. A 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. Proceedings of IEEE. Vol. 78., № 9., pp. 1415-1442, 1990.

[174] Windeatt, T., Ghaderi, R. Binary labeling and decision-level fusion. Information Fusion 2, pp.103-112, 2001.

[175] Windeatt, T., Tebbs, R. Spectral technique for hidden layer neural network training. Pattern Recognition Letters 18(8), pp.723-731, 1997.

[176] Xilinx. The Programmable Logic Data Book. Xilinx, Inc., San Jose, CA, 1993.

[177] Xilinx. XACT Reference Guide, Xilinx, San Jose, CA, 1992.

[178] Yamamoto, A., Saito, T. An improved Expand-and-Truncate Learning. Proc. of IEEE International Conference on Neural Networks (ICNN) 2, pp.1111-1116, 1997.

[179] Zell, A. Simulation neuronaler Netze. Oldenburg Verlag, 624 p., 1997.

[180] Zhu, J., Sutton, P. FPGA Implementations of Neural Networks – a Survey of a Decade of Progress. In Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL 2003), Lisbon, Sep., 2003.

Anhang A

Boolesche Neuronale Netze

A.1 Beispiel des Netzes für die OR-Dekomposition

A.1.1 Vorbemerkungen

Für die OR-Dekomposition wurden 10 Booleschen Funktionen $y_1, y_2, ..., y_{10}$ gewählt. Jede Boolesche Funktion hängt von 3 Booleschen Variablen x_1, x_2, x_3 ab. Die Wertetabelle der angegebenen Booleschen Funktionen wird in der Tabelle A.1 dargestellt.

Tabelle A.1 Wertetabelle der angegebenen Booleschen Funktionen

x_1	x_2	x_3	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
0	0	0	0	0	1	1	0	0	1	1	1	0
0	0	1	1	1	1	1	0	1	1	1	1	0
0	1	0	0	0	1	0	1	1	0	0	1	1
0	1	1	1	1	0	1	0	1	1	0	0	0
1	0	0	0	1	0	0	1	0	1	1	1	0
1	0	1	1	1	1	1	1	1	1	0	1	1
1	1	0	1	1	0	1	0	1	1	0	0	0
1	1	1	0	0	1	0	1	1	0	0	1	1

Das Netz hat 10 Ausgänge für 10 Boolesche Funktionen und folglich besteht die Ausgabeschicht des Netzes aus 10 Booleschen Neuronen mit OR-Transferfunktion. Da die gegebenen Funktionen von 3 Booleschen Variablen abhängen N_x=3, besitzt das Netz 3 Eingänge, d.h. die Eingabeschicht besteht aus 3 Neuronen. Für die weitere Bestimmung der Netzstruktur und der anderen Parameter des Netzes wird das Training durchgeführt.

A.1.2 Training

Der rechte Teil der Wertetabelle A.1 dient als Lernmenge des Booleschen Neuronalen Netzes und wird durch Anfangsmatrix $\boldsymbol{A}$ dargestellt. Der Algorithmus beginnt mit der Berechnung von Gewichtskoeffizienten des Vektors $\boldsymbol{m}$ für jede Zeile der Matrix $\boldsymbol{A}$. Solange

$m_i \neq 0$ existieren, wird der Algorithmus weitergeführt. Das Training wird mit der Berechnung von Gewichtskoeffizienten des Vektors ***n*** für jede Spalte der Matrix ***A*** durch fortgesetzt. Weiter in Tabellen sind die Umwandlungen im Verlauf des Trainings dargestellt.

Tabelle A.2 Anfangsmatrix ***A***

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	***m***	k_1
0	0	0	1	1	0	0	1	1	1	0	5	0
1	1	1	1	1	0	1	1	1	1	0	8	0
2	0	0	1	0	1	1	0	0	1	1	5	1
3	1	1	0	1	0	1	1	0	0	0	5	0
4	0	1	0	0	1	0	1	1	1	0	5	0
5	1	1	1	1	1	1	1	0	1	1	9	1
6	1	1	0	1	0	1	1	0	0	0	5	0
7	0	0	1	0	1	1	0	0	1	1	5	1
n	4	5	5	5	4	6	6	3	6	3		
v	1	1	1	1	1	1	1	0	1	1		
w_1	0	0	1	0	1	1	0	0	1	1		

Tabelle A.3 Matrix ***A*** nach dem ersten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	***m***	k_2
0	0	0	1	1	0	0	1	1	1	0	5	1
1	1	1	1	1	0	1	1	1	1	0	8	1
2	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	1	0	1	1	0	0	0	5	0
4	0	1	0	0	1	0	1	1	1	0	5	0
5	1	1	0	1	0	0	1	0	0	0	4	0
6	1	1	0	1	0	1	1	0	0	0	5	0
7	0	0	0	0	0	0	0	0	0	0	0	0
n	4	5	2	5	1	3	6	3	3	0		
v	1	1	1	1	0	1	1	1	1	0		
w_2	0	0	1	1	0	0	1	1	1	0		

Tabelle A.4 Matrix ***A*** nach dem zweiten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	***m***	k_3
0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	1	0	0	0	0	3	0
2	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	1	0	1	1	0	0	0	5	0
4	0	1	0	0	1	0	1	1	1	0	5	1
5	1	1	0	1	0	0	1	0	0	0	4	0
6	1	1	0	1	0	1	1	0	0	0	5	0
7	0	0	0	0	0	0	0	0	0	0	0	0
n	4	5	0	3	1	3	4	1	1	0		
v	0	1	0	0	1	0	1	1	1	0		
w_3	0	1	0	0	1	0	1	1	1	0		

Tabelle A.5 Matrix **A** nach dem dritten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	**m**	k_4
0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	1	1	0	0	0	5	1
2	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	1	0	1	1	0	0	0	5	1
4	0	0	0	0	0	0	0	0	0	0	0	0
5	1	1	0	1	0	1	1	0	0	0	5	1
6	1	1	0	1	0	1	1	0	0	0	5	1
7	0	0	0	0	0	0	0	0	0	0	0	0
n	4	4	0	4	0	4	4	0	0	0		
v	1	1	0	1	0	1	1	0	0	0		
w_4	1	1	0	1	0	1	1	0	0	0		

Tabelle A.6 Matrix **A** nach dem vierten Trainingszyklus.

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	**m**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0

Nach dem vierten Trainingszyklus sind alle Elemente des Vektors **m** sowie der Matrix **A** gleich Null geworden (Tabelle A.6). Demzufolge ist das Training beendet.
Das Boolesche Neuronale Netz wurde trainiert. Die erzeugte Netzstruktur wird in Abbildung A.1 gezeigt.

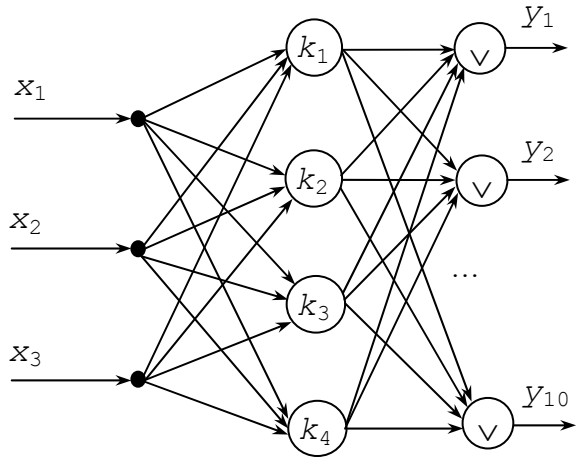


Abbildung A.1 Struktur des OR-BNN

A.1.3 Ergebnisse

Im Resultat des Trainings wurden die Transferfunktionen der vier verborgenen Neuronen (Tabelle A.7) und die Verbindungsgewichte der zehn Neuronen in der Ausgangsschicht (Tabelle A.8) erhalten.

Tabelle A.7 Transferfunktionen der verborgenen Booleschen Neuronen

x_1	x_2	x_3	k_1	k_2	k_3	k_4
0	0	0	0	1	0	0
0	0	1	0	1	0	1
0	1	0	1	0	0	0
0	1	1	0	0	0	1
1	0	0	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	0	0	1
1	1	1	1	0	0	0

Tabelle A.8 Verbindungsgewichte der Ausgangsschicht

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
$\boldsymbol{w}_1$	0	0	1	0	1	1	0	0	1	1
$\boldsymbol{w}_2$	0	0	1	1	0	0	1	1	1	0
$\boldsymbol{w}_3$	0	1	0	0	1	0	1	1	1	0
$\boldsymbol{w}_4$	1	1	0	1	0	1	1	0	0	0

Ist das Gewicht gleich 1, existiert die Verbindung zwischen den entsprechenden verborgenen und Ausgabeneuronen. Ist das Gewicht gleich 0 existiert keine Verbindung, d.h. eine Eingabe des Ausgangsneurons fehlt.

Solcherweise wurden die 10 Funktionen $y_1, y_2, ..., y_{10}$ in vier Boolesche Teilfunktionen $k_1, k_2, \ldots, k_4$ durch OR-Dekomposition zerlegt.

$$k_1 = \overline{x_1} x_2 \overline{x_3} \vee x_1 x_3 \qquad k_2 = \overline{x_1}\,\overline{x_2} \qquad k_3 = x_1 \overline{x_2}\,\overline{x_3} \qquad k_4 = \overline{x_1} x_3 \vee x_1 (x_2 \oplus x_3) \tag{A.1}$$

$$y_1 = k_4 \qquad y_2 = k_3 \vee k_4 \qquad y_3 = k_1 \vee k_2 \qquad y_4 = k_2 \vee k_4 \qquad y_5 = k_1 \vee k_3 \tag{A.2}$$

$$y_6 = k_1 \vee k_4 \qquad y_7 = k_2 \vee k_3 \vee k_4 \qquad y_8 = k_2 \vee k_3 \qquad y_9 = k_1 \vee k_2 \vee k_3 \qquad y_{10} = k_1 \tag{A.3}$$

A.2 BNN mit dem adaptierten Training

A.1.1 Vorbemerkungen

Es werden die Boolesche Funktionen y_1, y_7, und y_9 aus der Funktionsmenge (Anhang A.2 – Tabelle A.1) gewählt.

Tabelle A.9 Wertetabelle der Ausgangsfunktionen y_1, y_7, und y_9

	x_1	x_2	x_3	y_1	y_7	y_9
0	0	0	0	0	1	1
1	0	0	1	1	1	1
2	0	1	0	0	0	1
3	0	1	1	1	1	0
4	1	0	0	0	1	1
5	1	0	1	1	1	1
6	1	1	0	1	1	0
7	1	1	1	0	0	1

In der Ausgabeschicht des Booleschen Neuronalen Netzes gibt es 3 Boolesche Neuronen, die Funktionswerte der 3 gegebenen Booleschen Funktionen liefern. Die gegebenen Funktionen hängen von 3 Booleschen Variablen N_x=3 ab, deshalb besitzt das Netz 3 Eingänge. Nehmen wir an, dass eine LUT 2 Eingänge hat. Für die optimale Realisierung der Transferfunktionen der Booleschen Neuronen wird der adaptierte Trainingsalgorithmus verwendet.

A.1.2 Training

Der rechte Teil der Wertetabelle A.9 dient als Lernmenge des Booleschen Neuronalen Netzes. Laut dem Algorithmus 5.1 wird zunächst die Eingangsschicht des Netzes eingefügt. Die Anzahl von Neuronen in der Eingangsschicht ist gleich der Anzahl aller Argumente der Menge Boolescher Funktionen. Die Anzahl von verborgenen Neuronen wird auf 0 gesetzt. Für die Entwicklung der verborgenen Schicht des Netzes wird die Funktion y_9 aus der Funktionsmenge gewählt. Da es in der Wertetabelle dieser Funktion noch Einswerte gibt, Tab. A.10 – Spalte 7, wird der Algorithmus weitergeführt. In weiteren Tabellen sind die Umwandlungen im Verlauf des Trainings dargestellt. Die Wertetabelle der Funktion y_9 wird in zwei Hälften aufgeteilt und miteinander umgetauscht, Tab. A.10 – Spalten 7-8. Jeder Wert der Funktion y_9, Tab. A.10 – Spalte 7, wird dem entsprechenden Wert der Spalte 8 zugeordnet. Danach wird eine Konjunktion der beiden einander zuge-

ordneten Werte und somit das partielle Minimum der Funktion y_9 nach der Variable x_1 berechnet. Da nicht alle Werte in der Spalte 9 - Tab.A.10 gleich 0 sind, wird das berechnete Minimum der Funktion y_9 als die Transferfunktion des ersten verborgenen Neurons k_{OR} betrachtet.

Tabelle A.10 Suche der k-Funktion für OR- und AND-Operation

0	1	2		3	4	5	6	7	8		9		10	11		13
y_1	y_7	y_9	⇨		x_1	x_2	x_3	y_9		OR	k_{OR}	⇨	y_9		AND	
0	1	1		0	0	0	0	1	1	→	1		~	~	→	1
1	1	1		1	0	0	1	1	1	→	1		~	~	→	1
0	0	1		2	0	1	0	1	0	→	0		1	0	→	1
1	1	0		3	0	1	1	0	1	→	0		0	1	→	1
0	1	1		4	1	0	0	1	1	→	1		~	~	→	1
1	1	1		5	1	0	1	1	1	→	1		~	~	→	1
1	1	0		6	1	1	0	0	1	→	0		0	1	→	1
0	0	1		7	1	1	1	1	0	→	0		1	0	→	1

Die in der Spalte 10 der Tabelle A.10 gezeigte Wertetabelle der Funktion ist das Resultat des Abspaltens der k_{OR} von der in der Spalte 7 der Tabelle A.10 dargestellten Funktion y_9. Es gilt: jeder Einswert der Funktion y_9 wird auf ‚~' gesetzt, wenn der entsprechende Wert der Funktion k_{OR} gleich 1 ist. Solcherweise erhält man einen Funktionsverband, Tab. A.10 – Spalte 10. Die ‚don't care'- Werte des Funktionsverbandes können in weiteren schritten des Algorithmus sowohl 0 als auch 1 annehmen. Da die Wertetabelle der Funktion y_9, Spalte 10, gleichzeitig die Eins- und Nullwerte enthält und somit weiter dekomponiert werden kann, wird ihre Wertetabelle wieder in zwei Hälften aufgeteilt, Tab. A.10 – Spalte 11, und jeder Wert der Funktion y_9 der Spalte 10 wird dem entsprechenden Wert der Spalte 11 zugeordnet. In diesem Fall wird aber eine k-Funktion für die AND-Dekomposition gesucht, deshalb wird eine Disjunktion der beiden einander zugeordneten Werte und somit das partielle Maximum der Funktion y_9 nach der Variable x_1 berechnet. Da man kein Nullwert dabei erhalten hat, Tab. A.10 – Spalte 13, wurde keine Teilfunktion der Funktion y_9 für die AND-Dekomposition gefunden und folglich werden das partielle Minimum und Maximum der Funktion y_9 nach einer anderen Variable in weiteren Schritten des Trainings berechnet. Die Umwandlungen im Verlauf des Trainings sind in weiteren Tabellen dargestellt.

Hier wird zunächst das Abspalten der Funktionen betrachtet. Der Vektor von Gewichten für die Neuronen in der Ausgabeschicht des Booleschen Neuronalen Netzes wird durch die REDUCTIONOFYSET(), Alg.5.3, bestimmt und die k_{OR}-Funktion von allen Funktionen der Funktionsmenge entsprechend abgespaltet.

Tabelle A.11 Abspalten einer k_{OR}–Funktion von den y_1, y_7 und y_9

	y_1	y_7	y_9	k_{OR}		y_1	y_7	y_9
	0	1	1	1	→	0	~	~
	1	1	1	1	→	1	~	~
	0	0	1	0	→	0	0	1
	1	1	0	0	→	1	1	0
	0	1	1	1	→	0	~	~
	1	1	1	1	→	1	~	~
	1	1	0	0	→	1	1	0
	0	0	1	0	→	0	0	1
w	0	1	1					

Da das Gewicht w für y_1 gleich 0 ist, bleibt y_1 unverändert. Für die y_7 und y_9 gilt dabei die gleiche Regel wie schon für die in der Tabelle A.10 – Spalte 10 beschriebene Operation: Der Einswert der aus der Menge gewählten Funktion wird auf ‚~' gesetzt, falls der entsprechende Wert der k-Funktion gleich 1 ist.

Nach diesem Schritt hat die y_9 in ihrer Wertetabelle nur entweder Nullwerte oder ‚~' und somit ist dekomponiert. Deshalb wählt man weiter aus der gegebenen Funktionsmenge eine andere Funktion - y_7. Im diesem Beispiel ist die Funktion y_7 schon teilweise dekomponiert, weil es ‚~'-Werte in ihrer Wertetabelle gibt (Tab. A.12).

Tabelle A.12 Funktionsmenge nach dem Abspalten der zweiten k-Funktion

	y_1	y_7	y_9	k_{OR}		y_1	y_7	y_9
	0	~	~	1	→	0	~	~
	1	~	~	0	→	1	~	~
	0	0	1	1	→	0	0	~
	1	1	0	0	→	1	1	0
	0	~	~	0	→	0	~	~
	1	~	~	1	→	1	~	~
	1	1	0	0	→	1	1	0
	0	0	1	1	→	0	0	~
w	0	0	1					

In den Tabellen A.13-A.15 werden die weitere gefundene Funktione und die Funktionsmenge nach dem Abspalten dieser k-Funktionen dargestellt.

Tabelle A.13 Funktionsmenge nach dem Abspalten der dritten k-Funktion

y_7
~
~
0
1
~
~
1
0

⇨

k_{OR}
0
1
0
1
1
0
1
0

	y_1	y_7	y_9		y_1	y_7	y_9
	0	~	~	→	0	~	~
	1	~	~	→	1	~	~
	0	0	~	→	0	0	~
	1	1	0	→	1	~	0
	0	~	~	→	0	~	~
	1	~	~	→	1	~	~
	1	1	0	→	1	~	0
	0	0	~	→	0	0	~
w	0	1	0	↑			

Tabelle A.14 Funktionsmenge nach dem Abspalten der vierten k-Funktion

y_1
0
1
0
1
0
1
1
0

⇨

k_{OR}
0
1
0
0
0
1
0
0

	y_1	y_7	y_9		y_1	y_7	y_9
	0	~	~	→	0	~	~
	1	~	~	→	~	~	~
	0	0	~	→	0	0	~
	1	~	0	→	1	~	0
	0	~	~	→	0	~	~
	1	~	~	→	~	~	~
	1	~	0	→	1	~	0
	0	0	~	→	0	0	~
w	1	0	0	↑			

Tabelle A.15 Funktionsmenge nach dem Abspalten der fünften k-Funktion

	y_1
0	0
1	~
2	0
3	1
4	0
5	~
6	1
7	0

		k_{AND}
0	→	0
~	→	1
1	→	1
0	→	1
0	→	0
~	→	1
0	→	1
1	→	1

	y_1	y_7	y_9		y_1	y_7	y_9
	0	~	~	→	~	~	~
	~	~	~	→	~	~	~
	0	0	~	→	0	0	~
	1	~	0	→	1	~	0
	0	~	~	→	~	~	~
	~	~	~	→	~	~	~
	1	~	0	→	1	~	0
	0	0	~	→	0	0	~
w	1	0	0	↑			

Tabelle A.15 zeigt, wie eine k_{AND}-Funktion für die AND-Operation gefunden wurde. Das Abspalten einer k_{AND} –Funktion unterscheidet sich von dem Abspalten einer k_{OR}– Funktion. Bei dem Abspalten einer k_{AND}-Funktion gilt: jeder Nullwert der aus der Menge gewählten Funktion wird auf ‚~' gesetzt, falls der entsprechende Wert der k-Funktion gleich 0 ist. Zu beachten ist, dass die Anzahl von Nullwerten in der Wertetabelle der y_1 wurde nicht verändert. Aber, bei der AND-Dekomposition endet der Trainingsalgorithmus, wenn es kein Nullwert in der Wertetabelle einer Funktion gibt. Der positive Effekt einer

Mischung von OR- und AND-Operationen im Trainingsalgorithmus des BNN wird in der Tabelle A.16 dargestellt.

Tabelle A.16 Funktionsmenge nach dem Training

y_1	k_{OR}
~	0
~	1
0	0
1	1
~	1
~	0
1	1
0	0

	y_1	y_7	y_9		y_1	y_7	y_9
	~	~	~	→	~	~	~
	~	~	~	→	~	~	~
	0	0	~	→	0	0	~
	1	~	0	→	~	~	0
	~	~	~	→	~	~	~
	~	~	~	→	~	~	~
	1	~	0	→	~	~	0
	0	0	~	→	0	0	~
w	1	0	0	↗			

Wie die Tabelle A.16 zeigt, wurde die letzte gefundene *k*–Funktion wieder für die OR-Operation gefunden und somit die Bedingung des Anhaltens des Trainings wird nochmals geändert. Jetzt wieder gilt: wenn es kein Einswert in der Wertetabelle einer Funktion gibt, hält der Algorithmus an. In diesen Schritt gefundene *k*–Funktion wurde schon früher für die andere Funktion aus der Funktionsmenge erhalten und schon existiert im Booleschen Neuronalen Netz als Transferfunktion eines verborgenen Booleschen Neurons. Deshalb wird die y_1 durch das Ausgabesignal des schon vorhandenen Neurons berechet und somit kein neues Neuron wird in die verborgene Schicht eingefügt.

Tabelle A.17 Transferfunktionen von verborgenen Neuronen

	x_1	x_2	x_3	k_1	k_2	k_3	k_4	k_5
0	0	0	0	1	1	0	0	0
1	0	0	1	1	0	1	1	1
2	0	1	0	0	1	0	0	1
3	0	1	1	0	0	1	0	1
4	1	0	0	1	0	1	0	0
5	1	0	1	1	1	0	1	1
6	1	1	0	0	0	1	0	1
7	1	1	1	0	1	0	0	1

Nach dem, als alle Funktionen in der Funktionsmenge dekomponiert wurden, ist die verborgene Schicht des BNN gebaut. Die Transferfunktionen von verborgenen Neuronen (Tabelle A.17) und die Verbindungsgewichte von Booleschen Neuronen in der Ausgabeschicht des Netzes (Tabelle A.18) wurden erhalten. Ein Neuron in der Ausgabeschicht des Netzes hat 3 Eingänge, weil unter den Verbindungsgewichten dieses Neurons drei Gewichte gleich 1 sind (siehe Gewichte für die Funktion y_1 in der Tabelle A.18).

Tabelle A.18 Verbindungsgewichte von Neuronen der Ausgabeschicht

	k_1	k_2	k_3	k_4	k_5
y_1	0	0	1	1	1
y_7	1	0	1	0	0
y_9	1	1	0	0	0

Da jedes Neuron laut Aufgabenstellung höchstens 2 Eingänge haben darf, wird dieses Neuron durch die Superposition in 2 Neuronen aufgespaltet. Jetzt ist das Training beendet und das Boolesche Neuronale Netz wurde trainiert. Die erzeugte Netzstruktur wird in Abbildung 5.6 gezeigt.

A.3 AND-Netze für den Benchmark alcom

Die Abbildung A.2 zeigt eine Struktur des BNN mit Neuronen ohne Beschränkung der Anzahl von Eingängen. Dieses Netz besteht aus 15 Neuronen in der Eingangsschicht, 46 Neuronen in der verborgenen Schicht und 38 Neuronen in der Ausgangsschicht. Die Transferfunktionen verborgener Neuronen hängen von der Eingangsvariablen x_1, x_2, ..., x_{15} ab. Die Verbindungen zwischen Neuronen der Eingangsschicht und Neuronen der verborgenen Schicht, die in den Abbildungen A.2 - A.3 dargestellt sind, entsprechen tatsächlichen Transferfunktionen verborgener Neuronen nicht. Die Verbindungen zwischen verborgenen Neuronen und Neuronen in der Ausgabeschicht wurden aber entsprechend den Gewichtsvektoren, die während des Trainings erhalten wurden, gebaut. Da die in der Abbildung A.3 dargestellte Netzstruktur zu groß für Darstellung auf einer Seite ist, wurde dieses Netz in zwei Teile aufgeteilt (Seiten 143-144). Im Vergleich zu dem Netz mit Neuronen ohne Beschränkungen (Abb. A.2), hat das Netz mit den auf 4 Eingängen beschränkten Neuronen (Abb. A.3) in der verborgenen Schicht 74 verborgene Neuronen. Es ist fast doppelt soviel wie im Netz ohne Beschränkungen. Die Transferfunktionen dieser 74 verborgenen Neuronen hängen aber von höchstens 4 Booleschen Variablen ab. Auch Neuronen in der Ausgabeschicht besitzen höchstens 4 Eingänge. Dafür wurde die Ausgabeschicht durch den **Superpos** –Algorithmus (Kapitel 5 - Alg. 5.4) in 2 Kaskaden aufgeteilt und somit die gesamte Anzahl von Neuronen in der Ausgangschicht wurde um 18 Neuronen vergrößert.
Die Beschränkung der Anzahl von Eingängen für Neuronen ermöglicht eine direkte Abbildung der Transferfunktionen von Booleschen Neuronen und somit der gesamten Struktur des BNN in eine FPGA-Struktur mit 4-Eingingen LUTs.

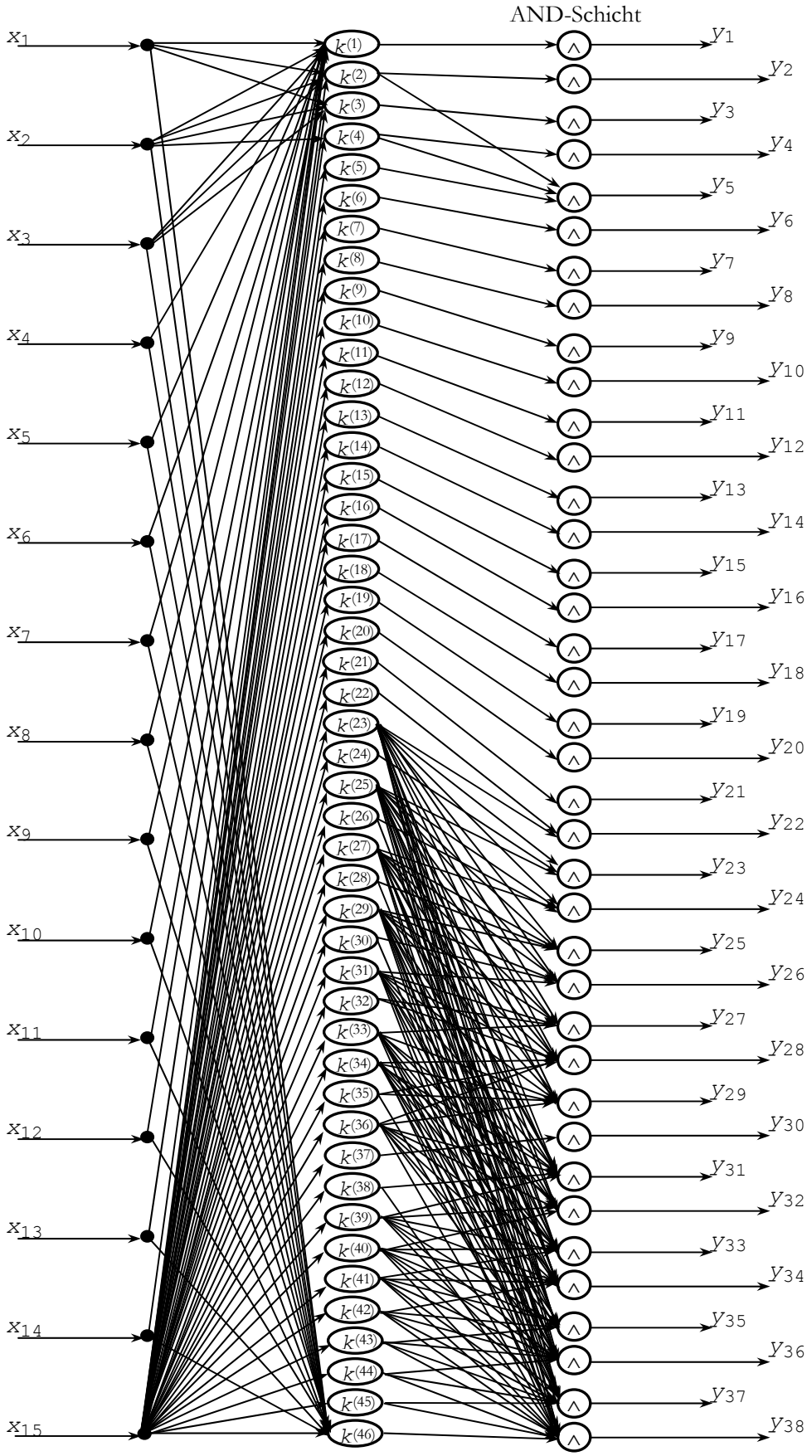
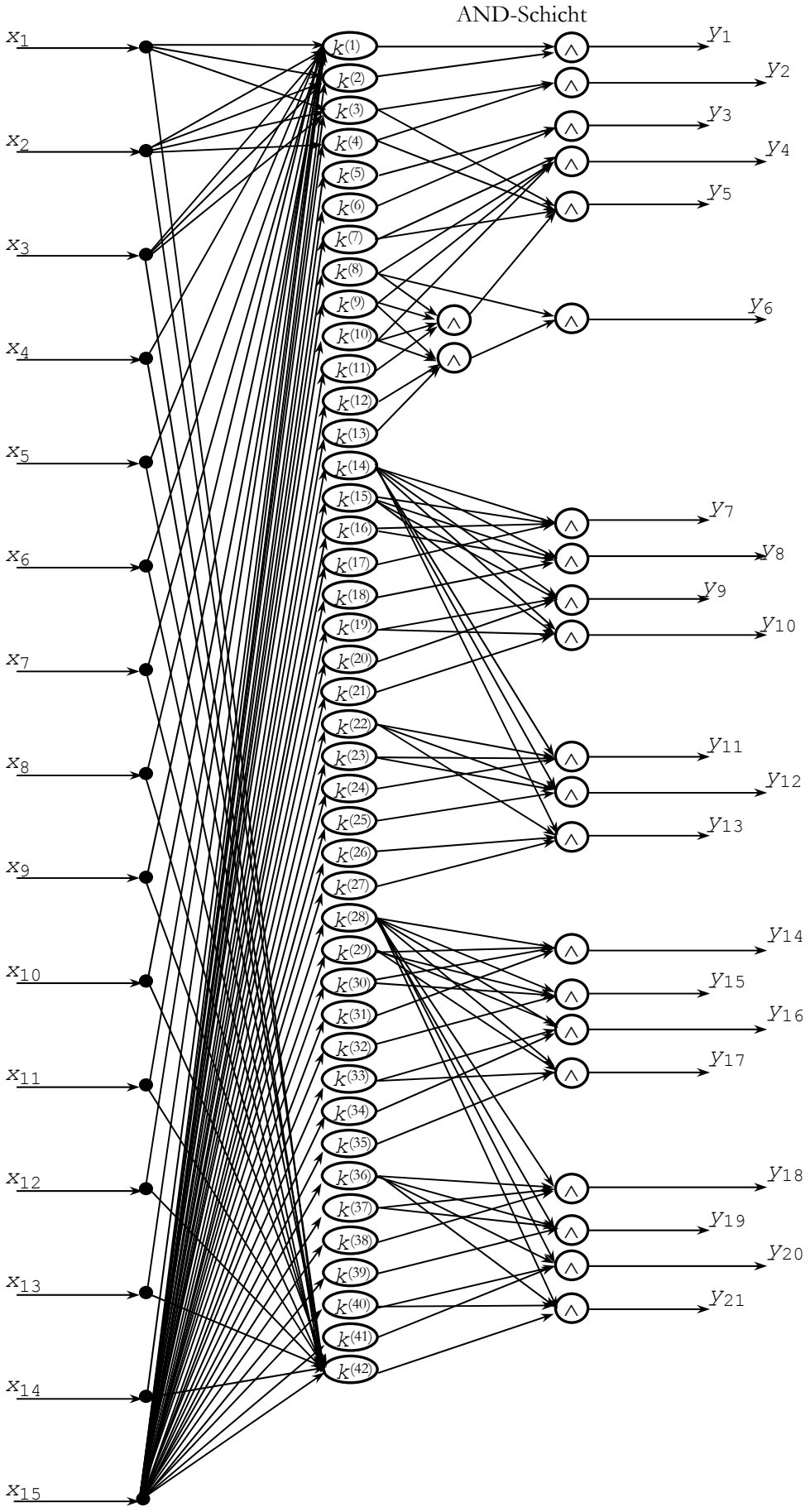


Abbildung A.2 BNN mit Neuronen ohne Beschränkung der Anzahl von Eingängen


AND-Schicht
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
k(1)
k(2)
k(3)
k(4)
k(5)
k(6)
k(7)
k(8)
k(9)
k(10)
k(11)
k(12)
k(13)
k(14)
k(15)
k(16)
k(17)
k(18)
k(19)
k(20)
k(21)
k(22)
k(23)
k(24)
k(25)
k(26)
k(27)
k(28)
k(29)
k(30)
k(31)
k(32)
k(33)
k(34)
k(35)
k(36)
k(37)
k(38)
k(39)
k(40)
k(41)
k(42)
y1
y2
y3
y4
y5
y6
y7
y8
y9
y10
y11
y12
y13
y14
y15
y16
y17
y18
y19
y20
y21

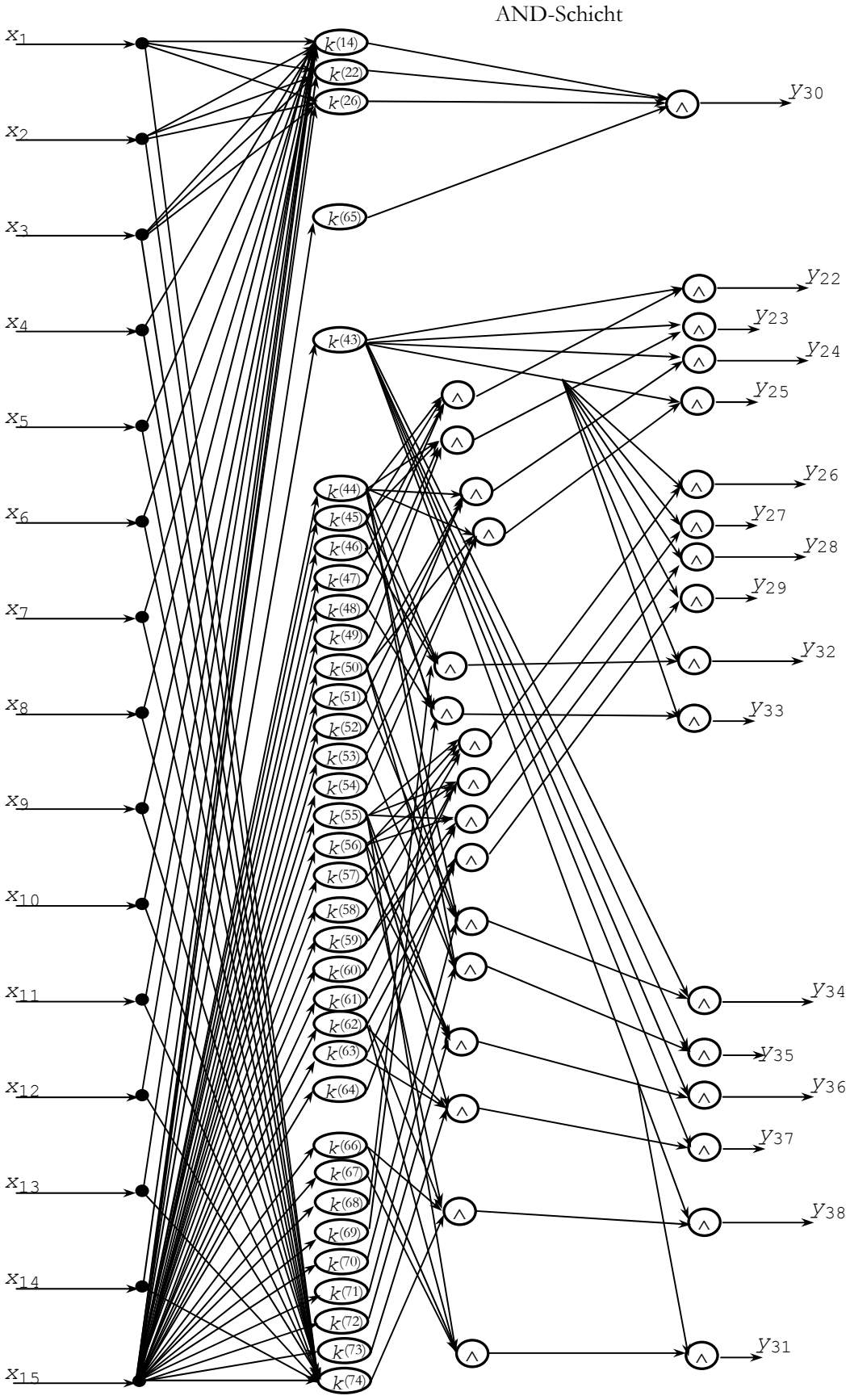


Abbildung A.3 BNN mit auf 4 Eingänge beschränkten Neuronen

Anhang B

FPGA-Realisierung von BNN

B.1 C++ Implementation

B.1.1 main.cpp

PROGRAMMLISTING B.1 MAIN.CPP

```
1.  #include "Main.h"
2.  // dependencies
3.  #include "runtime\cpp\RTRManager.h"
4.  #include <stddef.h>
5.  #include <string.h>
6.  #include <iostream>
7.  #include "datamodel\include\utility\smartptr.h"
8.  #include "datamodel\include\IHwObject.h"
9.
10. /**
11.  * Definition of the operations of Class Data.Main
12.  */
13.     Main::Main( )    { }
14.
15.    int Main::main( void )
16.    {
17.       int tmp;
18.       smartptr<IHwObject> net;
19.       int j;
20.       int outp;
21.       int inp;
22.
23.       inp = 0;
24.       std::cout << "Test of Boolean neural network...\n" << std::endl;
25.       net = RTRManager::getInstance()->createObject( 0 );
26.       for(inp = 0; (inp < 8); inp = inp + 1)
27.       {
28.          tmp = inp;
29.          j = 0;
30.          std::cout << "x = ";
31.          for(j = 0; (j < 3); j = j + 1)          {
32.             if( (( tmp & 1 ) == 1) )              {
33.                std::cout << "1 ";              }
34.             else              {
35.                std::cout << "0 ";              }
36.             tmp = tmp >> 1;
37.             /* assignment blocked: j = j + 1; */
38.          }
39.          std::cout << std::endl;
40.          net->write<int>(24, inp );
41.          net->execute<char>( 4, 1 ); /* execute Operation Bnn::init_x(Data.Mocca Design Platform.int) :
    Data.Mocca Design Platform.void */
42.          inp = net->read<int>( 24 );
43.
44.          net->execute<char>( 4, 128 ); /* execute Operation Bnn::calculate() : Data.Mocca Design
    Platform.boolean */
45.          if( net->read<bool>( 34 ) )
```

```
46.             {
47.                /* assignment blocked: j = 0; */
48.              net->execute<char>( 5, 128 ); /* execute Operation Bnn::get_y() : Data.Mocca Design Platform.int */
49.                outp = net->read<int>( 44 );
50.                tmp = outp;
51.                std::cout << "y = ";
52.                for(j = 0; (j < 10); j = j + 1)
53.                {
54.                   if( (( tmp & 1 ) == 1) ) {
55.                      std::cout << "1 ";
56.                   }
57.                   else {
58.                      std::cout << "0 ";
59.                   }
60.                   /* assignment blocked: j = j + 1; */
61.                   tmp = tmp >> 1;
62.                }
63.                std::cout << std::endl;
64.             }          /* assignment blocked: inp = inp + 1; */
65.          }
66.          std::cout << "done" << std::endl;
67.          RTRManager::getInstance()->destroyObject( &net );
68.          return 0;
69.       }
70.
71.        Main::~Main( )    {    }
```

B.1.2 Main.h

PROGRAMMLISTING B.2 MAIN.H

```
1.  #ifndef H_DATA_MAIN_H
2.  #define H_DATA_MAIN_H
3.
4.  // forward type declarations
5.
6.  #include "runtime\cpp\RTRManager.h"
7.  #include "datamodel\include\utility\smartptr.h"
8.  #include <stddef.h>
9.  #include <string.h>
10. #include <iostream>
11. #include "datamodel\include\IHwObject.h"
12.
13. class Main{
14. /**
15.  * Declaration of the operations of Class Data.Main
16.  */
17.    public: Main( );
18.    public: static int main( void );
19.    public: ~Main( );
20. };
21. #endif /* H_DATA_MAIN_H */
```

B.1.3 Main.cpp

PROGRAMMLISTING B.3 MAIN.CPP

```
1.  #include "Data\Main.h"
2.
3.  #include "datamodel\include\Common.h"
4.  #include "runtime\cpp\RTRManager.h"
5.  #include "oslf\OSLF.h"
6.
7.  #include <stddef.h>
8.  #include <stdio.h>
9.  #include <string.h>
10.
11. int main( int argc, char * argv[] )
12. {
13.       int success = Fail;
```

```
14.  int RESULT;
15.  Main * main;
16.
17.
18.
19.      OSLF_Init oslf;
20.
21.      printf( "Executing %s...\n", argv[0] );
22.
23.      /**
24.       * Initialize RTR Manager from --rtr-configuration
25.       */
26.      if(argc > 1)
27.      {
28.              for(int i = 1; i < argc; i++ )
29.              {
30.                      if( argv != NULL && strcmp(argv[i], "--rtr-configuration") == 0)
31.                      {
32.                              if( argc > i + 1 ){
33.                                      success = RTRManager::getInstance()-
     >readConfigurationFromFile( argv[i+1] );
34.                              }
35.                              else
36.                              {
37.                                      printf( "Missing parameter for command line option '--rtr-
     configuration'. The option is ignored.\n" );
38.                              }
39.                      }
40.              }
41.              // everything was fine?
42.              if( success == Fail ){
43.                      printf( "RTR Manager not initialized. Please start application with command line option
     --rtr-configuration <rtr-config-file>.\n \
44.                              Example: %s --rtr-configuration %s.conf", argv[0], argv[0] );
45.                      fflush(stdout);
46.                      return -1;
47.              }
48.      }
49.      else    {
50.                      char str[255];
51.                      printf( "RTR Manager not initialized. Please start application with command line option
     --rtr-configuration <rtr-config-file>.\n \
52.                              Example: %s --rtr-configuration %s.conf", argv[0], argv[0] );
53.                      // try default configuration name
54.                      sprintf(str, "%s.conf", argv[0] );
55.                      printf( "Trying configuration file %s ...\n", str );
56.                      success = RTRManager::getInstance()->readConfigurationFromFile( str );
57.      }
58.      // everything was fine?
59.      if( success == Fail )
60.      {
61.              printf( "Initialization of application failed. See log of RTR-Manager and system log for more
     information." );
62.              fflush(stdout);
63.              return -1;
64.      }
65.     /**
66.      * Call main-method here
67.      */
68.  main = new Main( );
69.
70.  RESULT = main->main( );
71.
72.  delete main;
73.      printf( "Execution of %s done.", argv[0] );
74.  return RESULT;
75.  }
```

B.2 VHDL Implementation

B.2.1 VHDL_Bnn_init_x

VHDL-LISTING B.4 INIT_X()

```
1.  library IEEE;
2.  use IEEE.numeric_std.all;
3.  use IEEE.std_logic_1164.all;
4.  use IEEE.std_logic_arith.all;
5.  use IEEE.std_logic_1164.std_logic;
6.  use IEEE.std_logic_1164.std_logic_vector;
7.
8.  library work;
9.  use work.mocca_pkg.all;
10. use work.mocca_pkg.mINT;
11. -- end of imports from Package work
12.
13. -- Entity declaration
14.
15. entity
      ImplementationPlatforms_VHDL_Bnn_init_x_void_int
      is
16.     port (
17.         CLOCK : in std_logic; -- CLOCK
18.         GO : in std_logic; -- GO
19.         DONE : out std_logic; -- DONE
20.         Bnn_a : out std_logic; -- Data.Bnn.a
21.         x : inout std_logic_vector(31 downto 0); --
      Data.Bnn.init_x.318.x
22.         Bnn_c : out std_logic; -- Data.Bnn.c
23.         Bnn_b : out std_logic-- Data.Bnn.b
24.          );
25. end
      ImplementationPlatforms_VHDL_Bnn_init_x_void_int;
26.
27. -- Architecture definition(s) for VHDL Entity
      ImplementationPlatforms_VHDL_Bnn_init_x_void_int
28.
29. architecture mixed of
      ImplementationPlatforms_VHDL_Bnn_init_x_void_int
      is
30.     type FsmState is (INIT, S1, S2, S3, S4, S5, S6, S7,
      S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18,
      S19, S20, FINAL);
31.
32.     signal CS : FsmState;
33.     signal NS : FsmState;
34.     signal I_DONE_sig0 : std_logic;
35.     signal OUTPUT_ENABLE_1 : std_logic;
36.     signal i_ren_sig0 : mINT;
37.
38.     constant syn_const1_0 : mINT :=
      "00000000000000000000000000000000";
39.
40.     signal Bnn_a_sig2 : std_logic;
41.     constant syn_const3_0 : std_logic := '0';
42.     signal x_sig4 : mINT;
43.     signal syn_conv_sig5 : std_logic_vector(31 downto
      0);
44.     signal x_sig6 : std_logic_vector(31 downto 0);
45.     signal x_sig8 : mINT;
46.     signal Bnn_c_sig9 : std_logic;
47.     signal Bnn_b_sig10 : std_logic;
48.     signal syn_tmp_sig11 : std_logic;
49.     constant syn_const12_3 : mINT :=
      "00000000000000000000000000000011";
50.
51.     constant syn_const14_1 : mINT :=
      "00000000000000000000000000000001";
52.
53.     signal syn_tmp_sig15 : std_logic;
54.     constant syn_const16_1 : std_logic := '1';
55.     signal syn_tmp_sig18 : std_logic;
56.     signal syn_tmp_sig20 : std_logic;
57.
58. begin
59.     DONE <= I_DONE_sig0 ;
60.     OUTPUT_ENABLE_1 <= GO or I_DONE_sig0 ;
61.     Bnn_a <= Bnn_a_sig2  when ( OUTPUT_ENABLE_1
      = '1' ) else
62.     'Z' ;
63.
64.     x <= syn_conv_sig5  when ( I_DONE_sig0 = '1' )
      else     (others => 'Z') ;
65.
66.     syn_conv_sig5 <= conv_std_logic_vector(x_sig4) ;
67.
68.     x_sig6 <= x ;
69.     x_sig8 <= mINT(x_sig6) ;
70.
71.     Bnn_c <= Bnn_c_sig9  when ( OUTPUT_ENABLE_1
      = '1' ) else
72.     'Z' ;
73.
74.     Bnn_b <= Bnn_b_sig10  when ( OUTPUT_ENABLE_1
      = '1' ) else
75.     'Z' ;
76.
77.     dp : process (GO, CLOCK) is
78.         variable syn_chain_var13 : mINT;
79.         variable syn_chain_var17 : mINT;
80.         variable syn_chain_var19 : mINT;
81.         variable syn_chain_var21 : mINT;
82.         variable syn_chain_var22 : mINT;
83.     begin
84.         if GO = '0' then
85.             i_ren_sig0 <= syn_const1_0 ;
86.             Bnn_a_sig2 <= syn_const3_0 ;
87.             x_sig4 <= syn_const1_0 ;
88.             Bnn_c_sig9 <= syn_const3_0 ;
89.             Bnn_b_sig10 <= syn_const3_0 ;
90.             syn_tmp_sig11 <= syn_const3_0 ;
91.             syn_tmp_sig15 <= syn_const3_0 ;
92.             syn_tmp_sig18 <= syn_const3_0 ;
93.             syn_tmp_sig20 <= syn_const3_0 ;
94.         else
95.             if CLOCK'event and CLOCK = '1' then
96.                 case CS is
97.                     when INIT =>
98.                         i_ren_sig0 <= syn_const1_0 ;
99.                     when S1 =>
100.                        syn_tmp_sig11 <=
      conv_std_logic(i_ren_sig0 < syn_const12_3) ;
101.                    when S4 =>
102.                        syn_chain_var13 :=
      mINT(std_logic_vector(x_sig8) and
      std_logic_vector(syn_const14_1));
```

```
                    syn_tmp_sig15 <=
conv_std_logic(syn_chain_var13 = syn_const14_1) ;
                when S7 =>
                    Bnn_a_sig2 <= syn_const16_1 ;
                when S8 =>
                    Bnn_a_sig2 <= syn_const3_0 ;
                when S9 =>
                    syn_chain_var17 :=
mINT(std_logic_vector(x_sig8) and
std_logic_vector(syn_const14_1));
                    syn_tmp_sig18 <=
conv_std_logic(syn_chain_var17 = syn_const14_1) ;
                when S12 =>
                    Bnn_b_sig10 <= syn_const16_1 ;
                when S13 =>
                    Bnn_b_sig10 <= syn_const3_0 ;
                when S14 =>
                    syn_chain_var19 :=
mINT(std_logic_vector(x_sig8) and
std_logic_vector(syn_const14_1));
                    syn_tmp_sig20 <=
conv_std_logic(syn_chain_var19 = syn_const14_1) ;
                when S17 =>
                    Bnn_c_sig9 <= syn_const16_1 ;
                when S18 =>
                    Bnn_c_sig9 <= syn_const3_0 ;
                when S19 =>
                    syn_chain_var21 :=
mINT(shift_right(signed(x_sig8),
conv_integer(syn_const14_1)));
                    x_sig4 <= syn_chain_var21 ;
                    syn_chain_var22 := (i_ren_sig0 +
syn_const14_1);
                    i_ren_sig0 <= syn_chain_var22 ;
                when others =>
                    null;
            end case;
        end if;
    end if;
  end process dp;

  fsm : process (syn_tmp_sig11, i_ren_sig0,
syn_tmp_sig15, syn_tmp_sig18, syn_tmp_sig20, CS)
is
  begin
    case CS is
      when INIT =>
        NS <= S1 ;
      when S1 =>
        NS <= S2 ;
      when S2 =>
        if (syn_tmp_sig11 = '1') then
          NS <= S3 ;
        else
          NS <= FINAL ;
        end if;
      when S3 =>
        case ( i_ren_sig0 ) is
          when
"00000000000000000000000000000000" =>
            NS <= S4 ;
          when
"00000000000000000000000000000001" =>
            NS <= S9 ;
          when
"00000000000000000000000000000010" =>
            NS <= S14 ;
          when others =>
            NS <= FINAL ;
        end case;
      when S4 =>
        NS <= S5 ;
      when S5 =>
        NS <= S6 ;
      when S6 =>
        if (syn_tmp_sig15 = '1') then
          NS <= S7 ;
        else      NS <= S8 ;
        end if;
      when S7 =>
        NS <= S19 ;
      when S8 =>
        NS <= S19 ;
      when S9 =>
        NS <= S10 ;
      when S10 =>
        NS <= S11 ;
      when S11 =>
        if (syn_tmp_sig18 = '1') then
          NS <= S12 ;
        else            NS <= S13 ;
        end if;
      when S12 =>
        NS <= S19 ;
      when S13 =>
        NS <= S19 ;
      when S14 =>
        NS <= S15 ;
      when S15 =>
        NS <= S16 ;
      when S16 =>
        if (syn_tmp_sig20 = '1') then
          NS <= S17 ;
        else        NS <= S18 ;
        end if;
      when S17 =>
        NS <= S19 ;
      when S18 =>
        NS <= S19 ;
      when S19 =>
        NS <= S20 ;
      when S20 =>
        NS <= S1 ;
      when FINAL =>
        NS <= FINAL ;
      when others =>
        NS <= FINAL ;
    end case;
  end process fsm;

  sync : process (GO, CLOCK) is
  begin
    if GO = '0' then
      CS <= INIT ;
      I_DONE_sig0 <= '0' ;
    else
      if CLOCK'event and CLOCK = '1' then
        CS <= NS ;
        if NS = FINAL then
          I_DONE_sig0 <= '1' ;
        else
          I_DONE_sig0 <= '0' ;
        end if;
      end if;
    end if;
  end process sync;

end mixed;
 -- End of entity declaration
```

B.2.2 VHDL_Bnn_create

VHDL-LISTING B.5 CREATE ()

```
226. library IEEE;
227. use IEEE.numeric_std.all;
228. use IEEE.std_logic_1164.all;
229. use IEEE.std_logic_arith.all;
230. use IEEE.std_logic_1164.std_logic;
231. use IEEE.std_logic_1164.std_logic_vector;
232. -- end of imports from Package IEEE
233.
234. library work;
235. use work.mocca_pkg.all;
236. use work.mocca_pkg.mINT;
237. -- end of imports from Package work
238.
239. -- Entity declaration
240.
241. entity
       ImplementationPlatforms_VHDL_Bnn_create_Bnn
       is
242.     port (
243.         CLOCK : in std_logic; -- CLOCK
244.         GO : in std_logic; -- GO
245.         DONE : out std_logic-- DONE
246.          );
247. end
       ImplementationPlatforms_VHDL_Bnn_create_Bnn;
248.
249. -- Architecture definition(s) for VHDL Entity
       ImplementationPlatforms_VHDL_Bnn_create_Bnn
250.
251. architecture mixed of
       ImplementationPlatforms_VHDL_Bnn_create_Bnn
       is
252.     type FsmState is (INIT, FINAL);
253.
254.     signal CS : FsmState;
255.
256.     signal NS : FsmState;
257.
258.     signal I_DONE_sig2 : std_logic;
259.
260.     signal OUTPUT_ENABLE_3 : std_logic;
261.
262. begin
263.     DONE <= I_DONE_sig2 ;
264.
265.     OUTPUT_ENABLE_3 <= GO or I_DONE_sig2 ;
266.
267.   dp : process (GO, CLOCK) is
268.   begin
269.     if GO = '0' then
270.     else
271.       if CLOCK'event and CLOCK = '1' then
272.         case CS is
273.           when INIT =>
274.           when others =>
275.             null;
276.         end case;
277.       end if;
278.     end if;
279.   end process dp;
280.
281.   fsm : process (CS) is
282.   begin
283.     case CS is
284.       when INIT =>
285.         NS <= FINAL ;
286.       when FINAL =>
287.         NS <= FINAL ;
288.       when others =>
289.         NS <= FINAL ;
290.     end case;
291.   end process fsm;
292.
293.   sync : process (GO, CLOCK) is
294.   begin
295.     if GO = '0' then
296.       CS <= INIT ;
297.       I_DONE_sig2 <= '0' ;
298.     else
299.       if CLOCK'event and CLOCK = '1' then
300.         CS <= NS ;
301.         if NS = FINAL then
302.           I_DONE_sig2 <= '1' ;
303.         else
304.           I_DONE_sig2 <= '0' ;
305.         end if;
306.       end if;
307.     end if;
308.   end process sync;
309.
310. end mixed;
```

B.2.3 VHDL_Bnn_ calculate

VHDL-Listing B.6 calculate()

```
311. library IEEE;
312. use IEEE.numeric_std.all;
313. use IEEE.std_logic_1164.all;
314. use IEEE.std_logic_arith.all;
315. use IEEE.std_logic_1164.std_logic;
316. use IEEE.std_logic_1164.std_logic_vector;
317. -- end of imports from Package IEEE
318. library work;
319. use work.mocca_pkg.all;
320. use work.mocca_pkg.mINT;
321. -- end of imports from Package work
322. -- Entity declaration
323. entity
       ImplementationPlatforms_VHDL_Bnn_calculate_boole
       an is
324.     port (
325.         CLOCK : in std_logic; -- CLOCK
326.         GO : in std_logic; -- GO
327.         DONE : out std_logic; -- DONE
328.         Bnn_y04 : out std_logic; -- Data.Bnn.y04
329.         Bnn_c : in std_logic; -- Data.Bnn.c
330.         Bnn_y03 : out std_logic; -- Data.Bnn.y03
331.         Bnn_k02 : inout std_logic; -- Data.Bnn.k02
332.         Bnn_y09 : out std_logic; -- Data.Bnn.y09
333.         Bnn_k01 : inout std_logic; -- Data.Bnn.k01
334.         Bnn_y05 : out std_logic; -- Data.Bnn.y05
335.         Bnn_y01 : inout std_logic; -- Data.Bnn.y01
336.         Bnn_y06 : out std_logic; -- Data.Bnn.y06
337.         Bnn_a : in std_logic; -- Data.Bnn.a
338.         Bnn_y07 : out std_logic; -- Data.Bnn.y07
339.         Bnn_k03 : inout std_logic; -- Data.Bnn.k03
340.         Bnn_y08 : out std_logic; -- Data.Bnn.y08
341.         Bnn_b : in std_logic; -- Data.Bnn.b
342.         Bnn_y02 : out std_logic; -- Data.Bnn.y02
343.         Bnn_y00 : out std_logic; -- Data.Bnn.y00
344.         Bnn_k04 : inout std_logic; -- Data.Bnn.k04
345.         RETURN_VALUE : out std_logic--
       Data.Bnn.calculate.returnVar_308
346.          );
347. end
       ImplementationPlatforms_VHDL_Bnn_calculate_boole
       an;
348. ImplementationPlatforms_VHDL_Bnn_calculate_boolea
       n
349. architecture mixed of
       ImplementationPlatforms_VHDL_Bnn_calculate_boole
       an is
350.     type FsmState is (INIT, S1, S2, S3, S4, FINAL);
351.     signal CS : FsmState;
352.     signal NS : FsmState;
353.     signal I_DONE_sig4 : std_logic;
354.     signal OUTPUT_ENABLE_5 : std_logic;
355.     signal Bnn_y04_sig25 : std_logic;
356.     constant syn_const26_0 : std_logic := '0';
357.     signal Bnn_c_sig27 : std_logic;
358.     signal Bnn_y03_sig28 : std_logic;
359.     signal Bnn_k02_sig29 : std_logic;
360.     signal Bnn_k02_sig30 : std_logic;
361.     signal Bnn_y09_sig31 : std_logic;
362.     signal Bnn_k01_sig32 : std_logic;
363.     signal Bnn_k01_sig33 : std_logic;
364.     signal Bnn_y05_sig34 : std_logic;
365.     signal Bnn_y01_sig35 : std_logic;
366.     signal Bnn_y01_sig36 : std_logic;
367.     signal Bnn_y06_sig37 : std_logic;
368.     signal Bnn_a_sig38 : std_logic;
369.     signal Bnn_y07_sig39 : std_logic;
370.     signal Bnn_k03_sig40 : std_logic;
371.     signal Bnn_k03_sig41 : std_logic;
372.     signal Bnn_y08_sig42 : std_logic;
373.     signal Bnn_b_sig43 : std_logic;
374.     signal Bnn_y02_sig44 : std_logic;
375.     signal Bnn_y00_sig45 : std_logic;
376.     signal Bnn_k04_sig46 : std_logic;
377.     signal Bnn_k04_sig47 : std_logic;
378.     signal returnVar_308_sig48 : std_logic;
379.     constant syn_const49_1 : std_logic := '1';
380. begin
381.     DONE <= I_DONE_sig4 ;
382.     OUTPUT_ENABLE_5 <= GO or I_DONE_sig4 ;
383.     Bnn_y04 <= Bnn_y04_sig25  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
384.     Bnn_c_sig27 <= Bnn_c ;
385.     Bnn_y03 <= Bnn_y03_sig28  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
386.     Bnn_k02 <= Bnn_k02_sig29  when ( I_DONE_sig4 = '1'
       ) else     'Z' ;
387.     Bnn_k02_sig30 <= Bnn_k02 ;
388.
389.     Bnn_y09 <= Bnn_y09_sig31  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
390.     Bnn_k01 <= Bnn_k01_sig32  when ( I_DONE_sig4 = '1'
       ) else     'Z' ;
391.     Bnn_k01_sig33 <= Bnn_k01 ;
392.     Bnn_y05 <= Bnn_y05_sig34  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
393.     Bnn_y01 <= Bnn_y01_sig35  when ( I_DONE_sig4 = '1'
       ) else     'Z' ;
394.     Bnn_y01_sig36 <= Bnn_y01 ;
395.     Bnn_y06 <= Bnn_y06_sig37  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
396.     Bnn_a_sig38 <= Bnn_a ;
397.     Bnn_y07 <= Bnn_y07_sig39  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
398.     Bnn_k03 <= Bnn_k03_sig40  when ( I_DONE_sig4 = '1'
       ) else     'Z' ;
399.     Bnn_k03_sig41 <= Bnn_k03 ;
400.     Bnn_y08 <= Bnn_y08_sig42  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
401.     Bnn_b_sig43 <= Bnn_b ;
402.     Bnn_y02 <= Bnn_y02_sig44  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
403.   Bnn_y00 <= Bnn_y00_sig45  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
404.     Bnn_k04 <= Bnn_k04_sig46  when ( I_DONE_sig4 = '1'
       ) else     'Z' ;
405.     Bnn_k04_sig47 <= Bnn_k04 ;
406.     RETURN_VALUE <= returnVar_308_sig48  when (
       OUTPUT_ENABLE_5 = '1' ) else     'Z' ;
407.     dp : process (GO, CLOCK) is
408.         variable syn_chain_var50 : std_logic;
409.         variable syn_chain_var51 : std_logic;
410.         variable syn_chain_var52 : std_logic;
411.         variable syn_chain_var53 : std_logic;
412.         variable syn_chain_var54 : std_logic;
413.         variable syn_chain_var55 : std_logic;
414.         variable syn_chain_var56 : std_logic;
415.         variable syn_chain_var57 : std_logic;
416.         variable syn_chain_var58 : std_logic;
417.         variable syn_chain_var59 : std_logic;
418.         variable syn_chain_var60 : std_logic;
419.         variable syn_chain_var61 : std_logic;
420.         variable syn_chain_var62 : std_logic;
421.         variable syn_chain_var63 : std_logic;
422.         variable syn_chain_var64 : std_logic;
423.         variable syn_chain_var65 : std_logic;
424.         variable syn_chain_var66 : std_logic;
425.         variable syn_chain_var67 : std_logic;
```

```
426.        variable syn_chain_var68 : std_logic;
427.        variable syn_chain_var69 : std_logic;
428.        variable syn_chain_var70 : std_logic;
429.        variable syn_chain_var71 : std_logic;
430.        variable syn_chain_var72 : std_logic;
431.        variable syn_chain_var73 : std_logic;
432.      begin
433.        if GO = '0' then
434.          Bnn_y04_sig25 <= syn_const26_0 ;
435.          Bnn_y03_sig28 <= syn_const26_0 ;
436.          Bnn_k02_sig29 <= syn_const26_0 ;
437.          Bnn_y09_sig31 <= syn_const26_0 ;
438.          Bnn_k01_sig32 <= syn_const26_0 ;
439.          Bnn_y05_sig34 <= syn_const26_0 ;
440.          Bnn_y01_sig35 <= syn_const26_0 ;
441.          Bnn_y06_sig37 <= syn_const26_0 ;
442.          Bnn_y07_sig39 <= syn_const26_0 ;
443.          Bnn_k03_sig40 <= syn_const26_0 ;
444.          Bnn_y08_sig42 <= syn_const26_0 ;
445.          Bnn_y02_sig44 <= syn_const26_0 ;
446.          Bnn_y00_sig45 <= syn_const26_0 ;
447.          Bnn_k04_sig46 <= syn_const26_0 ;
448.          returnVar_308_sig48 <= syn_const26_0 ;
449.        else
450.          if CLOCK'event and CLOCK = '1' then
451.            case CS is
452.              when INIT =>
453.                returnVar_308_sig48 <=
      syn_const49_1 ;
454.    syn_chain_var50 := (Bnn_a_sig38 and Bnn_b_sig43);
455.  syn_chain_var51 := (Bnn_a_sig38 and Bnn_c_sig27);
456.  syn_chain_var52 := (not Bnn_b_sig43);
457.  syn_chain_var53 := (not Bnn_c_sig27);
458.  syn_chain_var54 := (not Bnn_a_sig38);
459.  syn_chain_var55 := (syn_chain_var54 and
      Bnn_b_sig43);
460.  syn_chain_var56 := (syn_chain_var50 and
      syn_chain_var53);
461.  syn_chain_var57 := (syn_chain_var54 and
      syn_chain_var53);
462.  syn_chain_var58 := (Bnn_a_sig38 and
      syn_chain_var52);
463.  syn_chain_var59 := (syn_chain_var54 and
      syn_chain_var52);
464.  Bnn_k03_sig40 <= syn_chain_var59 ;
465.  syn_chain_var60 := (syn_chain_var55 and
      syn_chain_var53);
466.  syn_chain_var61 := (syn_chain_var58 and
      syn_chain_var53);
467.  Bnn_k02_sig29 <= syn_chain_var61 ;
468.  syn_chain_var62 := (syn_chain_var60 or
      syn_chain_var51);
469.  Bnn_k04_sig46 <= syn_chain_var62 ;
470.  Bnn_y09_sig31 <= syn_chain_var62 ;
471.                    syn_chain_var63 := (syn_chain_var58
      and Bnn_c_sig27);
472.  syn_chain_var64 := (syn_chain_var57 or
      syn_chain_var63);
473.  syn_chain_var65 := (syn_chain_var61 or
      syn_chain_var59);
474.  Bnn_y07_sig39 <= syn_chain_var65 ;
475.  syn_chain_var66 := (syn_chain_var61 or
      syn_chain_var62);
476.  Bnn_y04_sig25 <= syn_chain_var66 ;
477.  syn_chain_var67 := (syn_chain_var59 or
      syn_chain_var62);
478.  Bnn_y02_sig44 <= syn_chain_var67 ;
479.  syn_chain_var68 := (syn_chain_var65 or
      syn_chain_var62);
480.  Bnn_y08_sig42 <= syn_chain_var68 ;
481.  syn_chain_var69 := (syn_chain_var64 or
      syn_chain_var56);
482.  Bnn_k01_sig32 <= syn_chain_var69 ;
483.  syn_chain_var70 := (syn_chain_var69 or
      syn_chain_var59);
484.  Bnn_y03_sig28 <= syn_chain_var70 ;
485.  Bnn_y00_sig45 <= syn_chain_var69 ;
486.  syn_chain_var71 := (syn_chain_var69 or
      syn_chain_var61);
487.  Bnn_y01_sig35 <= syn_chain_var71 ;
488.  syn_chain_var72 := (syn_chain_var69 or
      syn_chain_var62);
489.  Bnn_y05_sig34 <= syn_chain_var72 ;
490.  syn_chain_var73 := (syn_chain_var71 or
      syn_chain_var59);
491.  Bnn_y06_sig37 <= syn_chain_var73 ;
492.                    when others =>
493.                      null;
494.              end case;
495.          end if;
496.        end if;
497.      end process dp;
498.      fsm : process (CS) is
499.      begin
500.        case CS is
501.          when INIT =>
502.            NS <= S1 ;
503.          when S1 =>
504.            NS <= S2 ;
505.          when S2 =>
506.            NS <= S3 ;
507.          when S3 =>
508.            NS <= S4 ;
509.          when S4 =>
510.            NS <= FINAL ;
511.          when FINAL =>
512.            NS <= FINAL ;
513.          when others =>
514.            NS <= FINAL ;
515.        end case;
516.      end process fsm;
517.      sync : process (GO, CLOCK) is
518.      begin
519.        if GO = '0' then
520.          CS <= INIT ;
521.          I_DONE_sig4 <= '0' ;
522.        else
523.          if CLOCK'event and CLOCK = '1' then
524.            CS <= NS ;
525.            if NS = FINAL then
526.              I_DONE_sig4 <= '1' ;
527.            else
528.              I_DONE_sig4 <= '0' ;
529.            end if;
530.          end if;
531.        end if;
532.      end process sync;
533.  end mixed;
```

B.2.4 VHDL_Bnn_get_y

VHDL-Listing B.7 get_y()

```
534. library IEEE;
535. use IEEE.numeric_std.all;
536. use IEEE.std_logic_1164.all;
537. use IEEE.std_logic_arith.all;
538. use IEEE.std_logic_1164.std_logic;
539. use IEEE.std_logic_1164.std_logic_vector;
540. -- end of imports from Package IEEE
541.
542. library work;
543. use work.mocca_pkg.all;
544. use work.mocca_pkg.mINT;
545. -- end of imports from Package work
546.
547. -- Entity declaration
548.
549. entity
     ImplementationPlatforms_VHDL_Bnn_get_y_int is
550.   port (
551.       CLOCK : in std_logic; -- CLOCK
552.       GO : in std_logic; -- GO
553.       DONE : out std_logic; -- DONE
554.       Bnn_y07 : in std_logic; -- Data.Bnn.y07
555.       Bnn_y04 : in std_logic; -- Data.Bnn.y04
556.       Bnn_y03 : in std_logic; -- Data.Bnn.y03
557.       Bnn_y08 : in std_logic; -- Data.Bnn.y08
558.       Bnn_y09 : in std_logic; -- Data.Bnn.y09
559.       Bnn_y02 : in std_logic; -- Data.Bnn.y02
560.       Bnn_y05 : in std_logic; -- Data.Bnn.y05
561.       Bnn_y01 : in std_logic; -- Data.Bnn.y01
562.       Bnn_y06 : in std_logic; -- Data.Bnn.y06
563.       Bnn_y00 : in std_logic; -- Data.Bnn.y00
564.       RETURN_VALUE : out std_logic_vector(31
     downto 0)-- Data.Bnn.get_y.returnVar_310
565.       );
566. end
     ImplementationPlatforms_VHDL_Bnn_get_y_int;
567.
568. -- Architecture definition(s) for VHDL Entity
     ImplementationPlatforms_VHDL_Bnn_get_y_int
569.
570. architecture mixed of
     ImplementationPlatforms_VHDL_Bnn_get_y_int is
571.   type FsmState is (INIT, S1, S2, S3, S4, S5, S6,
     S7, S8, S9, S10, S11, S12, S13, S14, S15, S16,
     S17, S18, S19, S20, S21, S22, S23, S24, S25,
     S26, FINAL);
572.   signal CS : FsmState;
573.   signal NS : FsmState;
574.   signal I_DONE_sig6 : std_logic;
575.   signal OUTPUT_ENABLE_7 : std_logic;
576.   signal y_ren_sig75 : mINT;
577.   constant syn_const76_0 : mINT :=
     "00000000000000000000000000000000";
578.   signal i_ren_sig77 : mINT;
579.   signal Bnn_y07_sig78 : std_logic;
580.   signal Bnn_y04_sig79 : std_logic;
581.   signal Bnn_y03_sig80 : std_logic;
582.   signal Bnn_y08_sig81 : std_logic;
583.   signal Bnn_y09_sig82 : std_logic;
584.   signal Bnn_y02_sig83 : std_logic;
585.   signal Bnn_y05_sig84 : std_logic;
586.   signal Bnn_y01_sig85 : std_logic;
587.   signal Bnn_y06_sig86 : std_logic;
588.   signal Bnn_y00_sig87 : std_logic;
589.   signal returnVar_310_sig88 : mINT;
590.   signal syn_conv_sig89:std_logic_vector(31
     downto 0);
591.   signal syn_tmp_sig90 : std_logic;
592.   constant syn_const91_0 : std_logic := '0';
593.   constant syn_const92_10 : mINT :=
     "00000000000000000000000000001010";
594.   constant syn_const94_1 : mINT :=
     "00000000000000000000000000000001";
595. begin
596.   DONE <= I_DONE_sig6 ;
597.   OUTPUT_ENABLE_7 <= GO or I_DONE_sig6 ;
598.   Bnn_y07_sig78 <= Bnn_y07 ;
599.   Bnn_y04_sig79 <= Bnn_y04 ;
600.   Bnn_y03_sig80 <= Bnn_y03 ;
601.   Bnn_y08_sig81 <= Bnn_y08 ;
602.   Bnn_y09_sig82 <= Bnn_y09 ;
603.   Bnn_y02_sig83 <= Bnn_y02 ;
604.   Bnn_y05_sig84 <= Bnn_y05 ;
605.   Bnn_y01_sig85 <= Bnn_y01 ;
606.   Bnn_y06_sig86 <= Bnn_y06 ;
607.   Bnn_y00_sig87 <= Bnn_y00 ;
608.   RETURN_VALUE <= syn_conv_sig89  when (
     OUTPUT_ENABLE_7 = '1' ) else
609.   (others => 'Z') ;
610.
611.   syn_conv_sig89 <=
     conv_std_logic_vector(returnVar_310_sig88) ;
612.   dp : process (GO, CLOCK) is
613.     variable syn_chain_var93 : mINT;
614.     variable syn_chain_var95 : mINT;
615.     variable syn_chain_var96 : mINT;
616.     variable syn_chain_var97 : mINT;
617.     variable syn_chain_var98 : mINT;
618.     variable syn_chain_var99 : mINT;
619.     variable syn_chain_var100 : mINT;
620.     variable syn_chain_var101 : mINT;
621.     variable syn_chain_var102 : mINT;
622.     variable syn_chain_var103 : mINT;
623.     variable syn_chain_var104 : mINT;
624.     variable syn_chain_var105 : mINT;
625.   begin
626.     if GO = '0' then
627.       y_ren_sig75 <= syn_const76_0 ;
628.       i_ren_sig77 <= syn_const76_0 ;
629.       returnVar_310_sig88 <= syn_const76_0 ;
630.       syn_tmp_sig90 <= syn_const91_0 ;
631.     else
632.       if CLOCK'event and CLOCK = '1' then
633.         case CS is
634.           when INIT =>
635.             i_ren_sig77 <= syn_const76_0 ;
636.             y_ren_sig75 <= syn_const76_0 ;
637.           when S1 =>
638.             syn_tmp_sig90 <=
     conv_std_logic(i_ren_sig77 < syn_const92_10) ;
639.           when S5 =>
640.             syn_chain_var93 :=
     mINT(std_logic_vector(y_ren_sig75) or
     std_logic_vector(syn_const94_1));
641.             y_ren_sig75 <= syn_chain_var93 ;
642.           when S7 =>
643.             syn_chain_var95 :=
     mINT(std_logic_vector(y_ren_sig75) or
     std_logic_vector(syn_const94_1));
644.             y_ren_sig75 <= syn_chain_var95 ;
645.           when S9 =>
646.             syn_chain_var96 :=
     mINT(std_logic_vector(y_ren_sig75) or
     std_logic_vector(syn_const94_1));
647.             y_ren_sig75 <= syn_chain_var96 ;
648.           when S11 =>
```

```
                    syn_chain_var97 :=
mINT(std_logic_vector(y_ren_sig75) or
std_logic_vector(syn_const94_1));
                    y_ren_sig75 <= syn_chain_var97
;
                when S13 =>
                    syn_chain_var98 :=
mINT(std_logic_vector(y_ren_sig75) or
std_logic_vector(syn_const94_1));
                    y_ren_sig75 <= syn_chain_var98
;
                when S15 =>
                    syn_chain_var99 :=
mINT(std_logic_vector(y_ren_sig75) or
std_logic_vector(syn_const94_1));
                    y_ren_sig75 <= syn_chain_var99
;
                when S17 =>
                    syn_chain_var100 :=
mINT(std_logic_vector(y_ren_sig75) or
std_logic_vector(syn_const94_1));
                    y_ren_sig75 <= syn_chain_var100
;
                when S19 =>
                    syn_chain_var101 :=
mINT(std_logic_vector(y_ren_sig75) or
std_logic_vector(syn_const94_1));
                    y_ren_sig75 <= syn_chain_var101
;
                when S21 =>
                    syn_chain_var102 :=
mINT(std_logic_vector(y_ren_sig75) or
std_logic_vector(syn_const94_1));
                    y_ren_sig75 <= syn_chain_var102
;
                when S23 =>
                    syn_chain_var103 :=
mINT(std_logic_vector(y_ren_sig75) or
std_logic_vector(syn_const94_1));
                    y_ren_sig75 <= syn_chain_var103
;
                when S24 =>
                    syn_chain_var104 :=
mINT(shift_left(signed(y_ren_sig75),
conv_integer(syn_const94_1)));
                    y_ren_sig75 <= syn_chain_var104
;
                    syn_chain_var105 := (i_ren_sig77
+ syn_const94_1);
                    i_ren_sig77 <= syn_chain_var105
;
                when S26 =>
                  returnVar_310_sig88 <=
y_ren_sig75 ;
                when others =>
                    null;
            end case;
        end if;
    end if;
  end process dp;

  fsm : process (syn_tmp_sig90, i_ren_sig77,
Bnn_y00_sig87, Bnn_y01_sig85, Bnn_y02_sig83,
Bnn_y03_sig80, Bnn_y04_sig79, Bnn_y05_sig84,
Bnn_y06_sig86, Bnn_y07_sig78, Bnn_y08_sig81,
Bnn_y09_sig82, CS) is
  begin
    case CS is
      when INIT =>
        NS <= S1 ;
      when S1 =>
        NS <= S2 ;
      when S2 =>
        if (syn_tmp_sig90 = '1') then
          NS <= S3 ;
        else
          NS <= S26 ;
        end if;
      when S3 =>
        case ( i_ren_sig77 ) is
          when
"00000000000000000000000000000000" =>
            NS <= S4 ;
          when
"00000000000000000000000000000001" =>
            NS <= S6 ;
          when
"00000000000000000000000000000010" =>
            NS <= S8 ;
          when
"00000000000000000000000000000011" =>
            NS <= S10 ;
          when
"00000000000000000000000000000100" =>
            NS <= S12 ;
          when
"00000000000000000000000000000101" =>
            NS <= S14 ;
          when
"00000000000000000000000000000110" =>
            NS <= S16 ;
          when
"00000000000000000000000000000111" =>
            NS <= S18 ;
          when
"00000000000000000000000000001000" =>
            NS <= S20 ;
          when
"00000000000000000000000000001001" =>
            NS <= S22 ;
          when others =>
            NS <= FINAL ;
        end case;
      when S4 =>
        if (Bnn_y00_sig87 = '1') then
          NS <= S5 ;
        else
          NS <= S24 ;
        end if;
      when S5 =>
        NS <= S24 ;
      when S6 =>
        if (Bnn_y01_sig85 = '1') then
          NS <= S7 ;
        else
          NS <= S24 ;
        end if;
      when S7 =>
        NS <= S24 ;
      when S8 =>
        if (Bnn_y02_sig83 = '1') then
          NS <= S9 ;
        else
          NS <= S24 ;
        end if;
      when S9 =>
        NS <= S24 ;
      when S10 =>
        if (Bnn_y03_sig80 = '1') then
          NS <= S11 ;
        else
          NS <= S24 ;
        end if;
      when S11 =>
        NS <= S24 ;
      when S12 =>
```

```
754.              if (Bnn_y04_sig79 = '1') then
755.                 NS <= S13 ;
756.              else
757.                 NS <= S24 ;
758.              end if;
759.           when S13 =>
760.              NS <= S24 ;
761.           when S14 =>
762.              if (Bnn_y05_sig84 = '1') then
763.                 NS <= S15 ;
764.              else
765.                 NS <= S24 ;
766.              end if;
767.           when S15 =>
768.              NS <= S24 ;
769.           when S16 =>
770.              if (Bnn_y06_sig86 = '1') then
771.                 NS <= S17 ;
772.              else
773.                 NS <= S24 ;
774.              end if;
775.           when S17 =>
776.              NS <= S24 ;
777.           when S18 =>
778.              if (Bnn_y07_sig78 = '1') then
779.                 NS <= S19 ;
780.              else
781.                 NS <= S24 ;
782.              end if;
783.           when S19 =>
784.              NS <= S24 ;
785.           when S20 =>
786.              if (Bnn_y08_sig81 = '1') then
787.                 NS <= S21 ;
788.              else
789.                 NS <= S24 ;
790.              end if;
791.           when S21 =>
792.              NS <= S24 ;
793.           when S22 =>
794.              if (Bnn_y09_sig82 = '1') then
795.                 NS <= S23 ;
796.              else
797.                 NS <= S24 ;
798.              end if;
799.           when S23 =>
800.              NS <= S24 ;
801.           when S24 =>
802.              NS <= S25 ;
803.           when S25 =>
804.              NS <= S1 ;
805.           when S26 =>
806.              NS <= FINAL ;
807.           when FINAL =>
808.              NS <= FINAL ;
809.           when others =>
810.              NS <= FINAL ;
811.        end case;
812.     end process fsm;
813.
814.     sync : process (GO, CLOCK) is
815.     begin
816.        if GO = '0' then
817.           CS <= INIT ;
818.           I_DONE_sig6 <= '0' ;
819.        else
820.           if CLOCK'event and CLOCK = '1' then
821.              CS <= NS ;
822.              if NS = FINAL then
823.                 I_DONE_sig6 <= '1' ;
824.              else
825.                 I_DONE_sig6 <= '0' ;
826.              end if;
827.           end if;
828.        end if;
829.     end process sync;
830.
831. end mixed;
```

B.2.5 VHDL_Bnn_destroy

VHDL-LISTING B.8 DESTROY()

```
832. library IEEE;
833. use IEEE.numeric_std.all;
834. use IEEE.std_logic_1164.all;
835. use IEEE.std_logic_arith.all;
836. use IEEE.std_logic_1164.std_logic;
837. use IEEE.std_logic_1164.std_logic_vector;
838. -- end of imports from Package IEEE
839.
840. library work;
841. use work.mocca_pkg.all;
842. use work.mocca_pkg.mINT;
843. -- end of imports from Package work
844.
845. -- Entity declaration
846.
847. entity
       ImplementationPlatforms_VHDL_Bnn_destroy_voi
       d is
848.     port (
849.         CLOCK : in std_logic; -- CLOCK
850.         GO : in std_logic; -- GO
851.         DONE : out std_logic-- DONE
852.          );
853. end
       ImplementationPlatforms_VHDL_Bnn_destroy_voi
       d;
854.
855. -- Architecture definition(s) for VHDL Entity
       ImplementationPlatforms_VHDL_Bnn_destroy_voi
       d
856.
857. architecture mixed of
       ImplementationPlatforms_VHDL_Bnn_destroy_voi
       d is
858.     type FsmState is (INIT, FINAL);
859.
860.     signal CS : FsmState;
861.
862.     signal NS : FsmState;
863.
864.     signal I_DONE_sig8 : std_logic;
865.
866.     signal OUTPUT_ENABLE_9 : std_logic;
867.
868. begin
869.     DONE <= I_DONE_sig8 ;
870.
871.     OUTPUT_ENABLE_9 <= GO or I_DONE_sig8 ;
872.
873.   dp : process (GO, CLOCK) is
874.   begin
875.     if GO = '0' then
876.     else
877.       if CLOCK'event and CLOCK = '1' then
878.         case CS is
879.           when INIT =>
880.           when others =>
881.             null;
882.         end case;
883.       end if;
884.     end if;
885.   end process dp;
886.
887.   fsm : process (CS) is
888.   begin
889.     case CS is
890.       when INIT =>
891.         NS <= FINAL ;
892.       when FINAL =>
893.         NS <= FINAL ;
894.       when others =>
895.         NS <= FINAL ;
896.     end case;
897.   end process fsm;
898.
899.   sync : process (GO, CLOCK) is
900.   begin
901.     if GO = '0' then
902.       CS <= INIT ;
903.       I_DONE_sig8 <= '0' ;
904.     else
905.       if CLOCK'event and CLOCK = '1' then
906.         CS <= NS ;
907.         if NS = FINAL then
908.           I_DONE_sig8 <= '1' ;
909.         else
910.           I_DONE_sig8 <= '0' ;
911.         end if;
912.       end if;
913.     end if;
914.   end process sync;
915.
916. end mixed;
```

B.2.6 VHDL_Architecture_Bnn

VHDL-LISTING B.9 ARCHITECTURE_BNN

```
917. library IEEE;
918. use IEEE.numeric_std.all;
919. use IEEE.std_logic_1164.all;
920. use IEEE.std_logic_arith.all;
921. use IEEE.std_logic_1164.std_logic;
922. use IEEE.std_logic_1164.std_logic_vector;
923. library work;
924. use work.mocca_pkg.all;
925. use work.mocca_pkg.mINT;
926. entity Bnn is
927.     port (
928.         CLOCK : in std_logic; -- CLOCK
929.         RESET : in std_logic; -- Data.Bnn
930. GO_ImplementationPlatforms_VHDL_Bnn_init_x_void_int : in std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.init_x
931. DONE_ImplementationPlatforms_VHDL_Bnn_init_x_void_int : out std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.init_x
932.         Bnn_c : inout std_logic; -- Data.Bnn.c
933.         Bnn_a : inout std_logic; -- Data.Bnn.a
934.         Bnn_b : inout std_logic; -- Data.Bnn.b
935.         Bnn_init_x_318_x : inout std_logic_vector(31 downto 0); -- Data.Bnn.init_x.318.x
936. GO_ImplementationPlatforms_VHDL_Bnn_create_Bnn : in std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.create
937. DONE_ImplementationPlatforms_VHDL_Bnn_create_Bnn : out std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.create
938. GO_ImplementationPlatforms_VHDL_Bnn_calculate_boolean : in std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.calculate
939. DONE_ImplementationPlatforms_VHDL_Bnn_calculate_boolean : out std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.calculate
940.         Bnn_y04 : inout std_logic; -- Data.Bnn.y04
941.         Bnn_y03 : inout std_logic; -- Data.Bnn.y03
942.         Bnn_k02 : inout std_logic; -- Data.Bnn.k02
943.         Bnn_y09 : inout std_logic; -- Data.Bnn.y09
944.         Bnn_k01 : inout std_logic; -- Data.Bnn.k01
945.         Bnn_y05 : inout std_logic; -- Data.Bnn.y05
946.         Bnn_y01 : inout std_logic; -- Data.Bnn.y01
947.         Bnn_y06 : inout std_logic; -- Data.Bnn.y06
948.         Bnn_y07 : inout std_logic; -- Data.Bnn.y07
949.         Bnn_k03 : inout std_logic; -- Data.Bnn.k03
950.         Bnn_y08 : inout std_logic; -- Data.Bnn.y08
951.         Bnn_y02 : inout std_logic; -- Data.Bnn.y02
952.         Bnn_y00 : inout std_logic; -- Data.Bnn.y00
953.         Bnn_k04 : inout std_logic; -- Data.Bnn.k04
954. RETURN_VALUE_ImplementationPlatforms_VHDL_Bnn_calculate_boolean : out std_logic; -- Data.Bnn.calculate.returnVar_308
955. GO_ImplementationPlatforms_VHDL_Bnn_get_y_int : in std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.get_y
956. DONE_ImplementationPlatforms_VHDL_Bnn_get_y_int : out std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.get_y
957. RETURN_VALUE_ImplementationPlatforms_VHDL_Bnn_get_y_int : out std_logic_vector(31 downto 0); -- Data.Bnn.get_y.returnVar_310
958. GO_ImplementationPlatforms_VHDL_Bnn_destroy_void : in std_logic; -- Data.ImplementationPlatforms.VHDL.Bnn.destroy
959. DONE_ImplementationPlatforms_VHDL_Bnn_destroy_void : out std_logic--
     Data.ImplementationPlatforms.VHDL.Bnn.destroy);
960. end Bnn;
961. -- Architecture definition(s) for VHDL Entity Bnn
962. architecture rtl of Bnn is
963.   component ImplementationPlatforms_VHDL_Bnn_init_x_void_int
964.     port (
965.       CLOCK : in std_logic;
966.       GO : in std_logic;
967.       DONE : out std_logic;
968.       Bnn_a : out std_logic;
969.       x : inout std_logic_vector(31 downto 0);
970.       Bnn_c : out std_logic;
971.       Bnn_b : out std_logic );
972.   end component ImplementationPlatforms_VHDL_Bnn_init_x_void_int;
973.   for I_ImplementationPlatforms_VHDL_Bnn_init_x_void_int_I : ImplementationPlatforms_VHDL_Bnn_init_x_void_int use entity ImplementationPlatforms_VHDL_Bnn_init_x_void_int(mixed);
974.   signal I_done_DONE_sig23 : std_logic;
975.   component ImplementationPlatforms_VHDL_Bnn_create_Bnn
976.     port (
977.       CLOCK : in std_logic;
978.       GO : in std_logic;
979.       DONE : out std_logic );
980.   end component ImplementationPlatforms_VHDL_Bnn_create_Bnn;
981.   for I_ImplementationPlatforms_VHDL_Bnn_create_Bnn_I : ImplementationPlatforms_VHDL_Bnn_create_Bnn use entity ImplementationPlatforms_VHDL_Bnn_create_Bnn(mixed);
982.   signal I_done_DONE_sig24 : std_logic;
983.   component ImplementationPlatforms_VHDL_Bnn_calculate_boolean
984.     port (
985.       CLOCK : in std_logic;
986.       GO : in std_logic;
987.       DONE : out std_logic;
988.       Bnn_y04 : out std_logic;
989.       Bnn_c : in std_logic;
990.       Bnn_y03 : out std_logic;
991.       Bnn_k02 : inout std_logic;
992.       Bnn_y09 : out std_logic;
993.       Bnn_k01 : inout std_logic;
994.       Bnn_y05 : out std_logic;
995.       Bnn_y01 : inout std_logic;
996.       Bnn_y06 : out std_logic;
997.       Bnn_a : in std_logic;
998.       Bnn_y07 : out std_logic;
999.       Bnn_k03 : inout std_logic;
1000.      Bnn_y08 : out std_logic;
1001.      Bnn_b : in std_logic;
1002.      Bnn_y02 : out std_logic;
1003.      Bnn_y00 : out std_logic;
1004.      Bnn_k04 : inout std_logic;
1005.      RETURN_VALUE : out std_logic );
1006.  end component ImplementationPlatforms_VHDL_Bnn_calculate_boolean;
1007.  for I_ImplementationPlatforms_VHDL_Bnn_calculate_boolean_I : ImplementationPlatforms_VHDL_Bnn_calculate_boolean use entity
```

```
    ImplementationPlatforms_VHDL_Bnn_calculate_b
    oolean(mixed);
1008.  signal I_done_DONE_sig74 : std_logic;
1009.  component
    ImplementationPlatforms_VHDL_Bnn_get_y_int
1010.      port (
1011.          CLOCK : in std_logic;
1012.          GO : in std_logic;
1013.          DONE : out std_logic;
1014.          Bnn_y07 : in std_logic;
1015.          Bnn_y04 : in std_logic;
1016.          Bnn_y03 : in std_logic;
1017.          Bnn_y08 : in std_logic;
1018.          Bnn_y09 : in std_logic;
1019.          Bnn_y02 : in std_logic;
1020.          Bnn_y05 : in std_logic;
1021.          Bnn_y01 : in std_logic;
1022.          Bnn_y06 : in std_logic;
1023.          Bnn_y00 : in std_logic;
1024.          RETURN_VALUE : out std_logic_vector(31
    downto 0) );
1025.  end component
    ImplementationPlatforms_VHDL_Bnn_get_y_int;
1026.  for
    I_ImplementationPlatforms_VHDL_Bnn_get_y_int
    _I :
    ImplementationPlatforms_VHDL_Bnn_get_y_int
    use entity
    ImplementationPlatforms_VHDL_Bnn_get_y_int(m
    ixed);
1027.  signal I_done_DONE_sig106 : std_logic;
1028.  component
    ImplementationPlatforms_VHDL_Bnn_destroy_voi
    d
1029.      port (
1030.          CLOCK : in std_logic;
1031.          GO : in std_logic;
1032.          DONE : out std_logic );
1033.  end component
    ImplementationPlatforms_VHDL_Bnn_destroy_voi
    d;
1034.  for
    I_ImplementationPlatforms_VHDL_Bnn_destroy_v
    oid_I :
    ImplementationPlatforms_VHDL_Bnn_destroy_voi
    d use entity
    ImplementationPlatforms_VHDL_Bnn_destroy_voi
    d(mixed);
1035.  signal I_done_DONE_sig107 : std_logic;
1036. begin
1037. I_ImplementationPlatforms_VHDL_Bnn_init_x_void
    _int_I :
    ImplementationPlatforms_VHDL_Bnn_init_x_void_
    int
1038.      port map ( GO =>
    GO_ImplementationPlatforms_VHDL_Bnn_init_x_v
    oid_int,
1039.          DONE => I_done_DONE_sig23,
1040.          Bnn_c => Bnn_c,
1041.          Bnn_a => Bnn_a,
1042.          Bnn_b => Bnn_b,
1043.          x => Bnn_init_x_318_x,
1044.          CLOCK => CLOCK );
1045. DONE_ImplementationPlatforms_VHDL_Bnn_init_x
    _void_int <= I_done_DONE_sig23 ;
1046. I_ImplementationPlatforms_VHDL_Bnn_create_Bnn
    _I :
    ImplementationPlatforms_VHDL_Bnn_create_Bnn
1047.      port map ( GO =>
    GO_ImplementationPlatforms_VHDL_Bnn_create_Bnn
1048.          DONE => I_done_DONE_sig24,
1049.          CLOCK => CLOCK );
1050. DONE_ImplementationPlatforms_VHDL_Bnn_create_Bnn
    <= I_done_DONE_sig24 ;
1051. I_ImplementationPlatforms_VHDL_Bnn_calculate_boolean_
    I :
    ImplementationPlatforms_VHDL_Bnn_calculate_boolean
1052.      port map ( GO =>
    GO_ImplementationPlatforms_VHDL_Bnn_calculate_boole
    an,
1053.          DONE => I_done_DONE_sig74,
1054.          Bnn_y04 => Bnn_y04,
1055.          Bnn_c => Bnn_c,
1056.          Bnn_y03 => Bnn_y03,
1057.          Bnn_k02 => Bnn_k02,
1058.          Bnn_y09 => Bnn_y09,
1059.          Bnn_k01 => Bnn_k01,
1060.          Bnn_y05 => Bnn_y05,
1061.          Bnn_y01 => Bnn_y01,
1062.          Bnn_y06 => Bnn_y06,
1063.          Bnn_a => Bnn_a,
1064.          Bnn_y07 => Bnn_y07,
1065.          Bnn_k03 => Bnn_k03,
1066.          Bnn_y08 => Bnn_y08,
1067.          Bnn_b => Bnn_b,
1068.          Bnn_y02 => Bnn_y02,
1069.          Bnn_y00 => Bnn_y00,
1070.          Bnn_k04 => Bnn_k04,
1071.          RETURN_VALUE =>
    RETURN_VALUE_ImplementationPlatforms_VHDL_Bnn_ca
    lculate_boolean,
1072.          CLOCK => CLOCK );
1073. DONE_ImplementationPlatforms_VHDL_Bnn_calculate_bool
    ean <= I_done_DONE_sig74 ;
1074.  I_ImplementationPlatforms_VHDL_Bnn_get_y_int_I :
    ImplementationPlatforms_VHDL_Bnn_get_y_int
1075.      port map ( GO =>
    GO_ImplementationPlatforms_VHDL_Bnn_get_y_int,
1076.          DONE => I_done_DONE_sig106,
1077.          Bnn_y04 => Bnn_y04,
1078.          Bnn_y03 => Bnn_y03,
1079.          Bnn_y09 => Bnn_y09,
1080.          Bnn_y05 => Bnn_y05,
1081.          Bnn_y06 => Bnn_y06,
1082.          Bnn_y01 => Bnn_y01,
1083.          Bnn_y07 => Bnn_y07,
1084.          Bnn_y08 => Bnn_y08,
1085.          Bnn_y02 => Bnn_y02,
1086.          Bnn_y00 => Bnn_y00,
1087.          RETURN_VALUE =>
    RETURN_VALUE_ImplementationPlatforms_VHDL_Bnn_ge
    t_y_int,
1088.          CLOCK => CLOCK );
1089. DONE_ImplementationPlatforms_VHDL_Bnn_get_y_int <=
    I_done_DONE_sig106 ;
1090. I_ImplementationPlatforms_VHDL_Bnn_destroy_void_I :
    ImplementationPlatforms_VHDL_Bnn_destroy_void
1091.  port map ( GO =>
    GO_ImplementationPlatform_VHDL_Bnn_destroy_void
1092.          DONE => I_done_DONE_sig107,
1093.          CLOCK => CLOCK );
1094. DONE_ImplementationPlatforms_VHDL_Bnn_destroy_void
    <= I_done_DONE_sig107 ;
1095. end rtl;
```

Anhang C

Experimentale Ergebnisse

C.1 Vorbemerkungen

C.1.1 BNN1

Im diesen Design werden die Eingangssignale `x0`, `x1`, `x2`, die Ausgangssignale der Transferfunktionen `k01`,...,`k04` und der Ausgangsschicht `y00`,...,`y09` durch Skalarattributen dargestellt. Der Vektor von Eingangssignalen wird durch den Array dem Netz übertragen. Die Methode `init_x()` ermittelt aus diesem Array die einzelne Skalarattribute (Programmlisting C.2). Nach Berechnungen werden die erhaltene Skalarwerte im einen Array in `get_y()` zusammengesammelt (Programmlisting C.3) und aus dem Netz bekommen. Bei der Modellierung der Kommunikation der Ausgangs- und der verborgenen Schicht wird eine Methode pro Neuron `k1()`,..., `k4()`, `y0()`,..., `y9()` verwendet. Diese Methoden werden synchron durch `calculate()` aufgerufen. Die Methode `calculate()` bildet das gesamte Netz ab.

PROGRAMMLISTING C.1 CALCULATE()

```
1096. k01=k1(); k02=k2(); k03=k3(); k04=k4(); y00=y0();
1097. y01=y1(); y02=y2(); y03=y3(); y04=y4(); y05=y5();
1098. y06=y6(); y07=y7(); y08=y8(); y09=y9();
1099. return true ;
```

PROGRAMMLISTING C.2 INIT_X()

```
1. x0= inputs[0]; x1=inputs[1]; x2=inputs[2];
```

PROGRAMMLISTING C.3: GET_Y()

```
1. y[0]=y00; y[1]=y01; y[2]=y02; y[3]=y03; y[4]=y04;
2. y[5]=y05; y[6]=y06; y[7]=y07; y[8]=y08; y[9]=y09;
```

C.1.2 BNN2

Die Implementierung von BNN2 ist gleich wie BNN1. Nur, die Eingangs- und Ausgangssignale werden in eine 32-Bits Integer-Variable kodiert, die zu und aus dem Netz übertragen wird. Dabei stellen die einzelnen Bits Werte von Eingangs- und Ausgangssignalen dar. Ko- und Dekodierung werden durch die Methode `get_y()` und `init_x()` entsprechend erfühlt (Programmlisting C.5, C.4).

PROGRAMMLISTING C.4 INIT_X() VON BNN2

```
1.  for( int i=0; i<3; i++) {
2.    switch (i) {
3.          case 0 :
4.                  if ((x&1)==1) x0=true;
5.                  else x0= false; break;
6.          case 1 :
7.                  if ((x&1)==1) x1=true;
8.                  else x1= false; break;
9.          case 2 :
10.                 if ( ( x&1) == 1) x2=true ;
11.                 else x2= false; break;
12.         }
13.   x = x>>1;
14. }
```

PROGRAMMLISTING C.5 GET_Y() VON BNN2

```
1.  int y=0;
2.  for (int i=0; i <10; i++) {
3.    switch (i) {
4.          case 0 : if (y00)  y|=1; break;
5.          case 1 : if (y01)  y|=1; break;
6.          case 2 : if (y02)  y|=1; break;
7.          case 3 : if (y03)  y|=1; break;
8.          case 4 : if (y04)  y|=1; break;
9.          case 5 : if (y05)  y|=1; break;
10.         case 6 : if (y06)  y|=1; break;
11.         case 7 : if (y07)  y|=1; break;
12.         case 8 : if (y08)  y|=1; break;
13.         case 9 : if (y09)  y|=1; break;
14.         }
15.   y = y<<1;
16. }
17. return y;
```

C.1.3 BNN3

Das BNN3 stimmt mit dem BNN1 überein, nur, beim dem BNN3 wurden die Kodierung und Dekodierung von Werten in und aus dem Array weggelassen. Jeder Eingangs- und Ausgangswert wird einzeln in und aus dem Netz übertragen.

C.1.4 BNN4

Das BNN4 stellt eine Variante von BNN3 dar, wobei die Realisierung von Transferfunktionen der Neuronen durch die einzelnen Methoden nicht vorgenommen wurde. Stattdessen wurde die Berechnung des gesamten Netzes in der Methode `calculate()` durchgaführt (Programmlisting C.6). Dieses Design hilft einige mögliche bei der Dekompositionsmodellierung entstandene negative Effekte zu entdecken. Auch, die quantitativen Werte der FPGA-Realisierungen können analysiert werden, so dass die beim Training erzeugte Netzstruktur mit der durch MOCCA erzeugten Schaltung aus LUTs verglichen werden kann.

PROGRAMMLISTING C.6 CALCULATE() VON BNN4

```
1. k01=!x0&&!x2 || x0&&!x1&&x2 || x0&&x1&&!x2;
2. k02=x0&&!x1&&!x2;          k03=!x0&&!x1;
3. k04=!x0&&x1&&!x2 || x0&&x2;
4. y00=k01;            y01=k01 || k02;    y02=k03 || k04;
5. y03=k01 || k03;   y04=k02 || k04;    y05=k01 || k04;
6. y06=k01 || k02 || k03;   y07=k02 || k03;
7. y08=k02 || k03 || k04;   y09=k04;
8. return true ;
```

C.1.5 BNN5

Im BNN5 wurden anstelle von Attributen Parameter verwendet. Ansonsten ist das BNN5 gleich BNN2. Im diesem Design wird der Aufwand von Methoden `init_x()` und `get_y()` durch `calculate()` übernommen. Dieses Design macht Netz einfacher in Anwendung, weil die Datenkommunikation in `calculate()` realisiert wird.

C.2 Zeitmessungen

Tabelle C.1 Mittlere Kompilationszeiten für FPGA-Implementationen von BNNs

Design	t_{opt}, [ms]	t_{map}, [ms]	t_{syn}, [ms]	t_{sum}, [ms]
BNN1	7756	3178	2731	13666
BNN2	10186	3672	3364	17222
BNN3	8680	3092	2508	14280
BNN4	4028	2447	1789	8264
BNN5	17124	5098	4458	26680

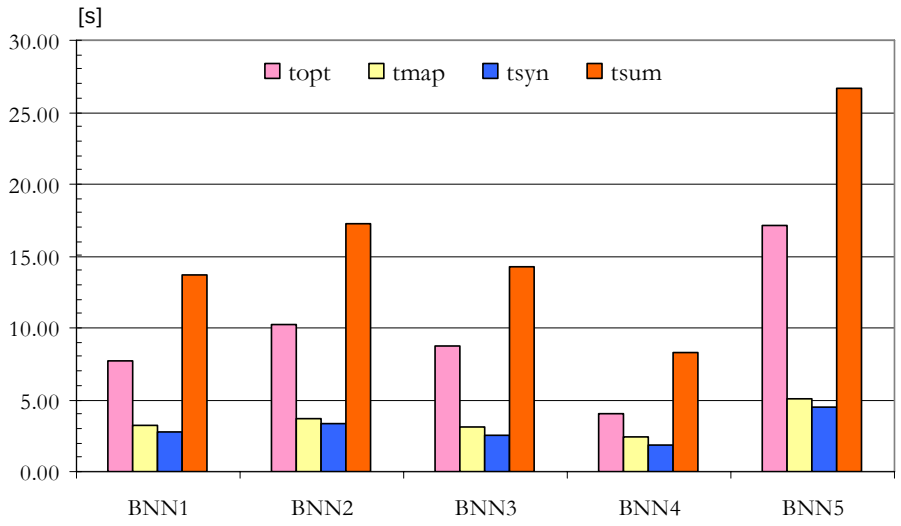


Abbildung C.1 Mittlere Kompilationszeiten für FPGA-Implementationen von BNNs

Tabelle C.2 Mittlere Kompilationszeiten für Software-Implementationen von BNNs

Design	t_{opt}, [ms]	t_{map}, [ms]	t_{syn}, [ms]	t_{sum}, [ms]
BNN1	3931	2281	366	6578
BNN2	4797	3835	396	9027
BNN3	4066	2420	356	6842
BNN4	2772	1305	309	4386
BNN5	7616	4675	519	12809

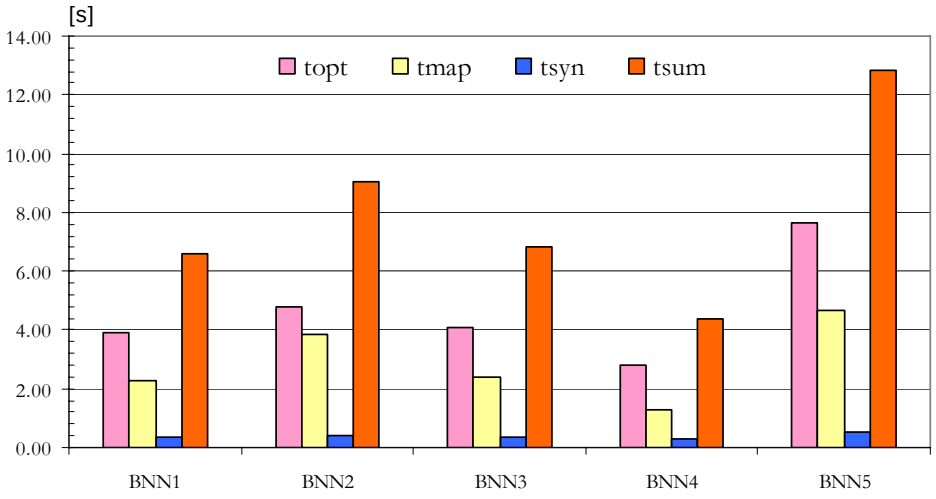


Abbildung C.2 Mittlere Kompilationszeiten für Software-Implementationen von BNNs

Tabelle C.3 Ausführungszeiten von FPGA-BNNs

Design	$t_{exec,init\_x}$,[ns]	$t_{exec,calculate}$,[ns]	t_{exec,get_y},[ns]	t_{exec},[ns]
BNN1	2233,75	2261,71	2131,25	6626,71
BNN2	2810,50	2244,40	3718,71	8773,61
BNN3	0,00	2278,35	0,00	2278,35
BNN4	0,00	2318,28	0,00	2318,28
BNN5	0,00	3758,98	0,00	3758,98

Tabelle C.4 Kommunikationszeiten von FPGA-BNNs

Design	$t_{write,x}$,[ns]	$t_{read,y}$,[ns]	t_{comm},[ns]
BNN1	200,01	7019,42	7219,43
BNN2	146,43	1600,44	1746,87
BNN3	293,53	15663,90	15957,43
BNN4	315,16	15691,85	16007,01
BNN5	96,84	1597,77	1694,62

Tabelle C.5 Ausführungszeiten von Software-BNNs

Design	$t_{exec,init\_x}$,[ns]	$t_{exec,calculate}$,[ns]	t_{exec,get_y},[ns]	t_{exec},[ns]
BNN1	82,20	393,04	70,22	545,46
BNN2	216,99	325,81	511,85	1054,64
BNN3	0,00	327,14	0,00	327,14
BNN4	0,00	271,90	0,00	271,9
BNN5	0,00	270,23	0,00	270,23

Tabelle C.6 Kommunikationszeiten von Software-BNNs

Design	$t_{write,x}$,[ns]	$t_{read,y}$,[ns]	t_{comm},[ns]
BNN1	0,00	0,00	0,00
BNN2	0,00	0,00	0,00
BNN3	47,26	52,25	99,51
BNN4	46,92	51,58	98,51
BNN5	0,00	0,00	0,00

Tabelle C.7 Ausführungszeiten von `Bnn::calculate()` in FPGA-BNNs

Design	$t^{h1}_{exec,calculate}$,[ns]	$t\hat{}^{h1}_{exec,calculate}$,[ns]	t_{err}, [ns]	t_{err}, [%]
BNN1	60	50	-10	16,67
BNN2	50	50	0	0
BNN3	60	50	-10	16,67
BNN4	50	50	0	0
BNN5	570	510	-60	10,53

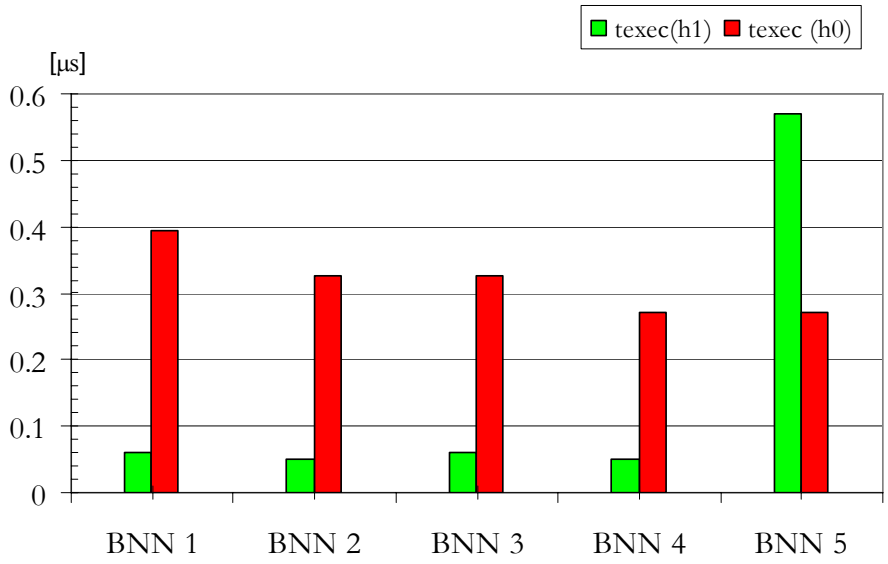


Abbildung C.3 Ausführungszeiten von `Bnn::calculate()` in FPGA und Software

C.3 Quantitative Ergebnisse

Tabelle C.8 Ressourceausnutzung für die Realisierung der Klasse `Bnn`

Design	#FSM States	#FF	#LUT	fmax [MHz]
BNN1	51	60	83	238
BNN2	60	178	218	171
BNN3	10	4	15	354
BNN4	10	4	15	354
BNN5	66	75	81	253

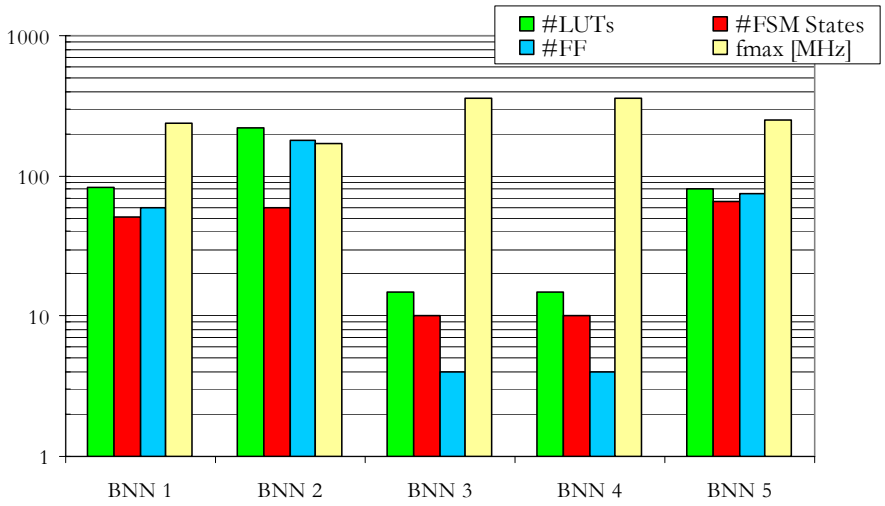


Abbildung C.4 Ressourceausnutzung für die Realisierung der Klasse `Bnn`

Tabelle C.9 Ressourceausnutzung für die Realisierung der Methode `Bnn::calculate()`

Design	#FSM States	#FF	#LUT	fmax [MHz]
BNN1	6	4	15	354
BNN2	6	4	15	354
BNN3	6	4	15	354
BNN4	6	4	15	354
BNN5	62	77	82	256

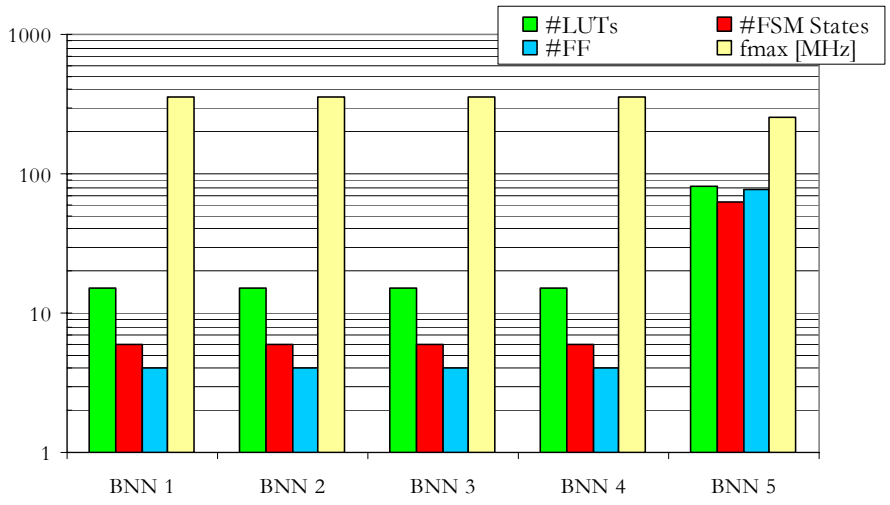


Abbildung C.5 Ressourceausnutzung für die Realisierung der Methode `Bnn::calculate()`

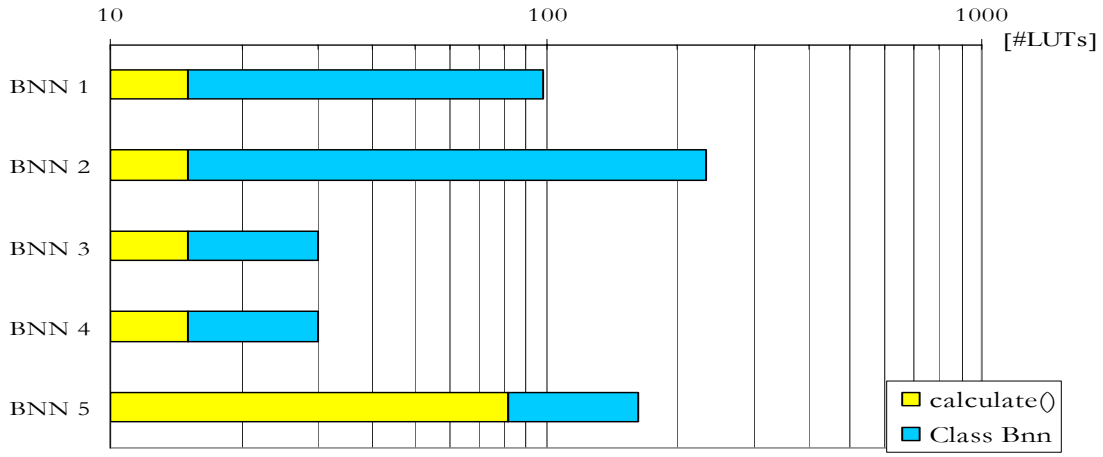


Abbildung C.6 LUTs-Ausnutzung für die Realisierung `Bnn` und `calculate()`

C.4 Technologische Schaltpläne

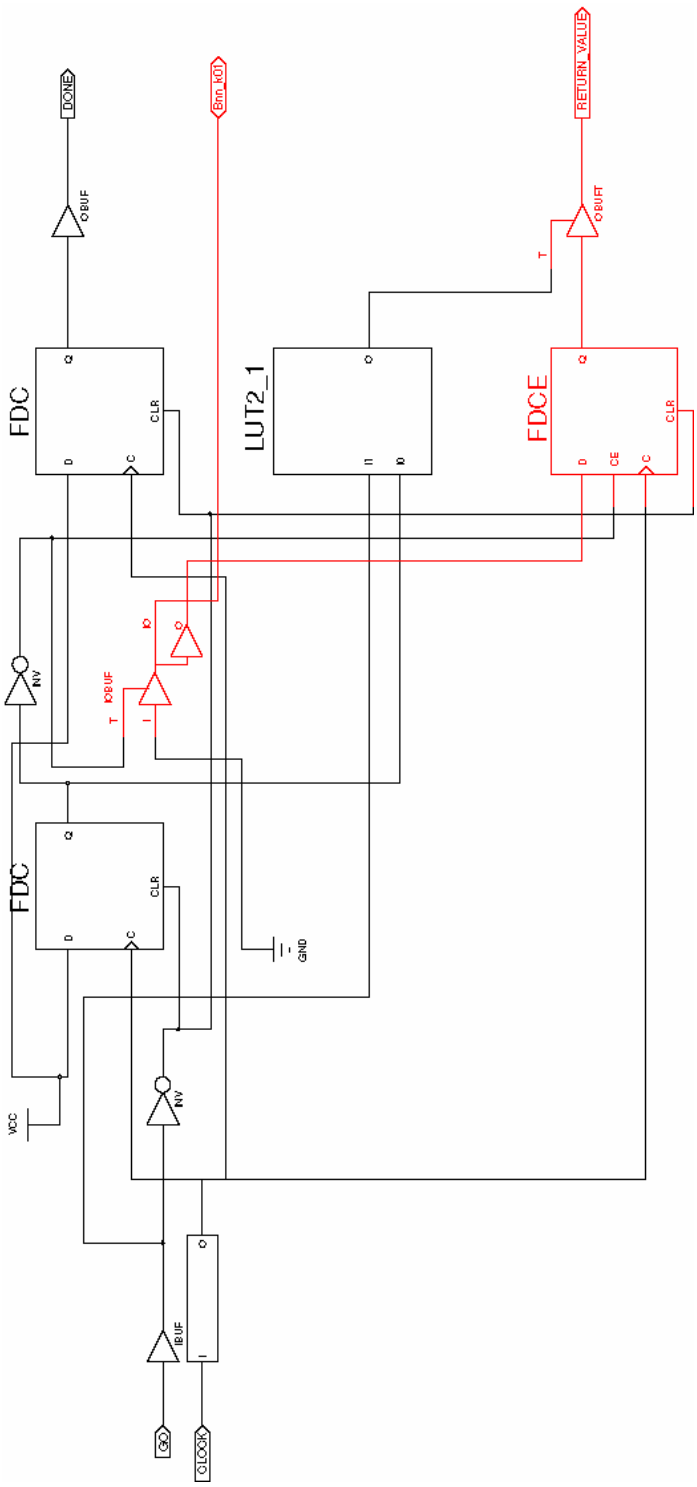


Abbildung C.7 Schaltplan für y0()

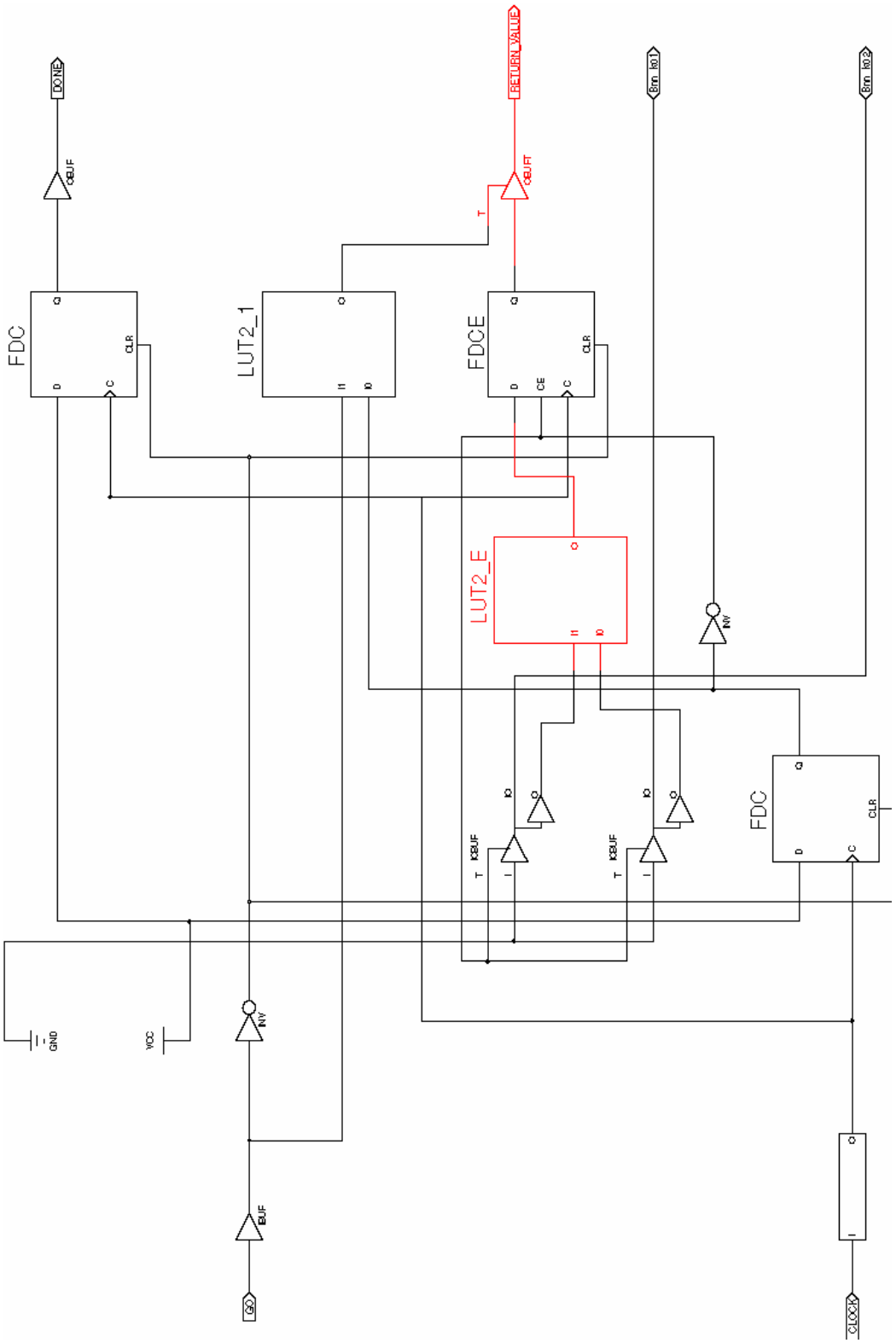


Abbildung C.8 Schaltplan für y1()

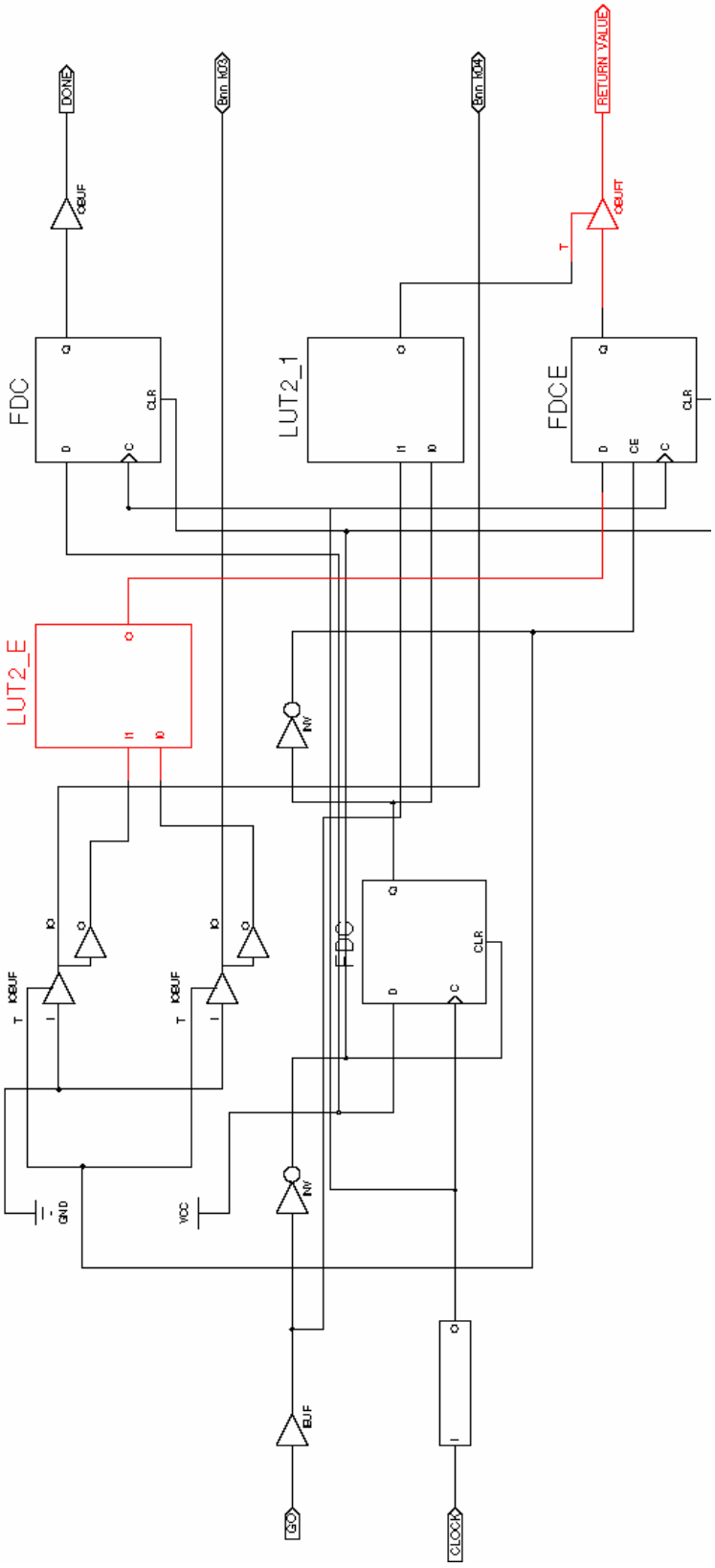


Abbildung C.9 Schaltplan für y2()

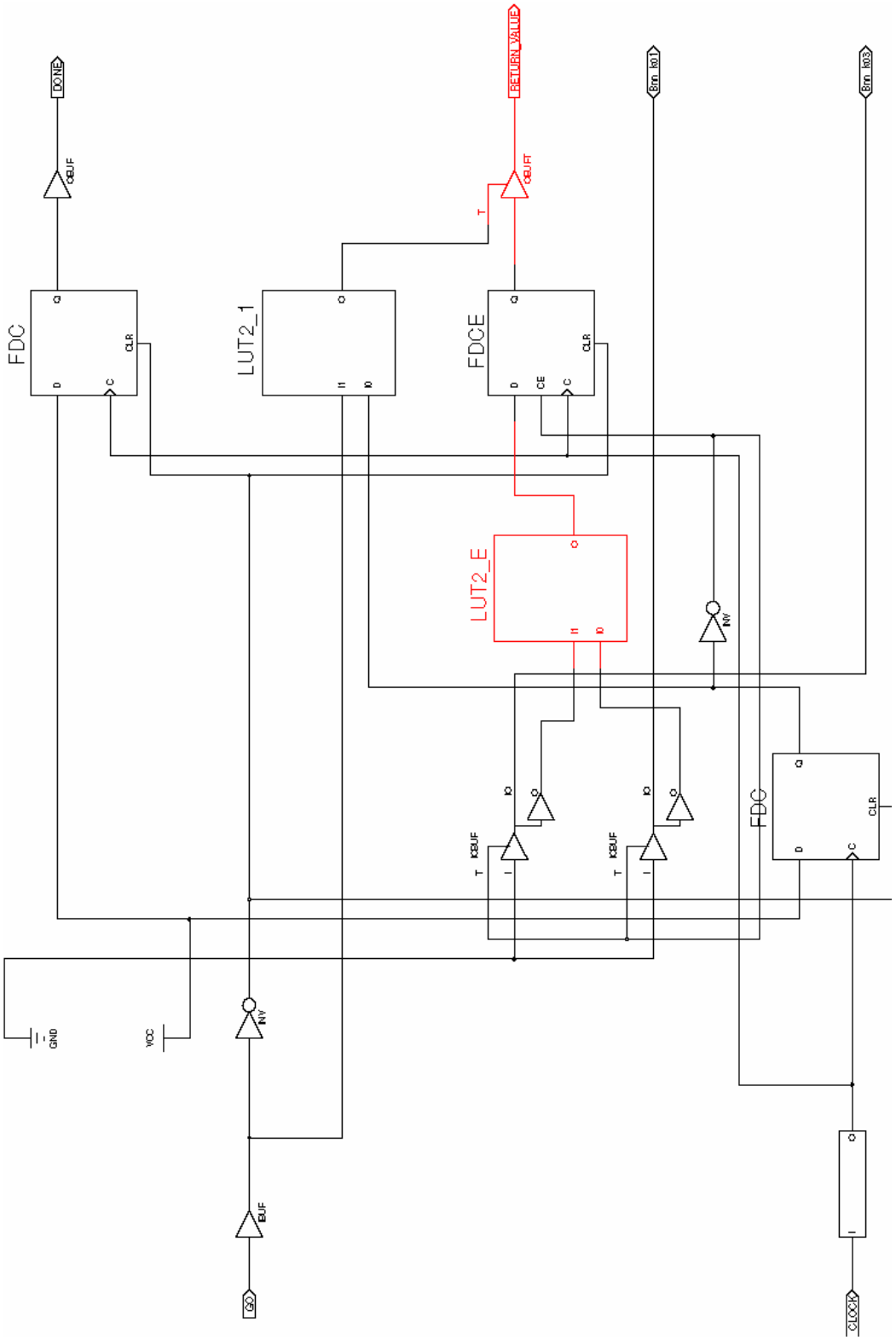


Abbildung C.10 Schaltplan für y3()

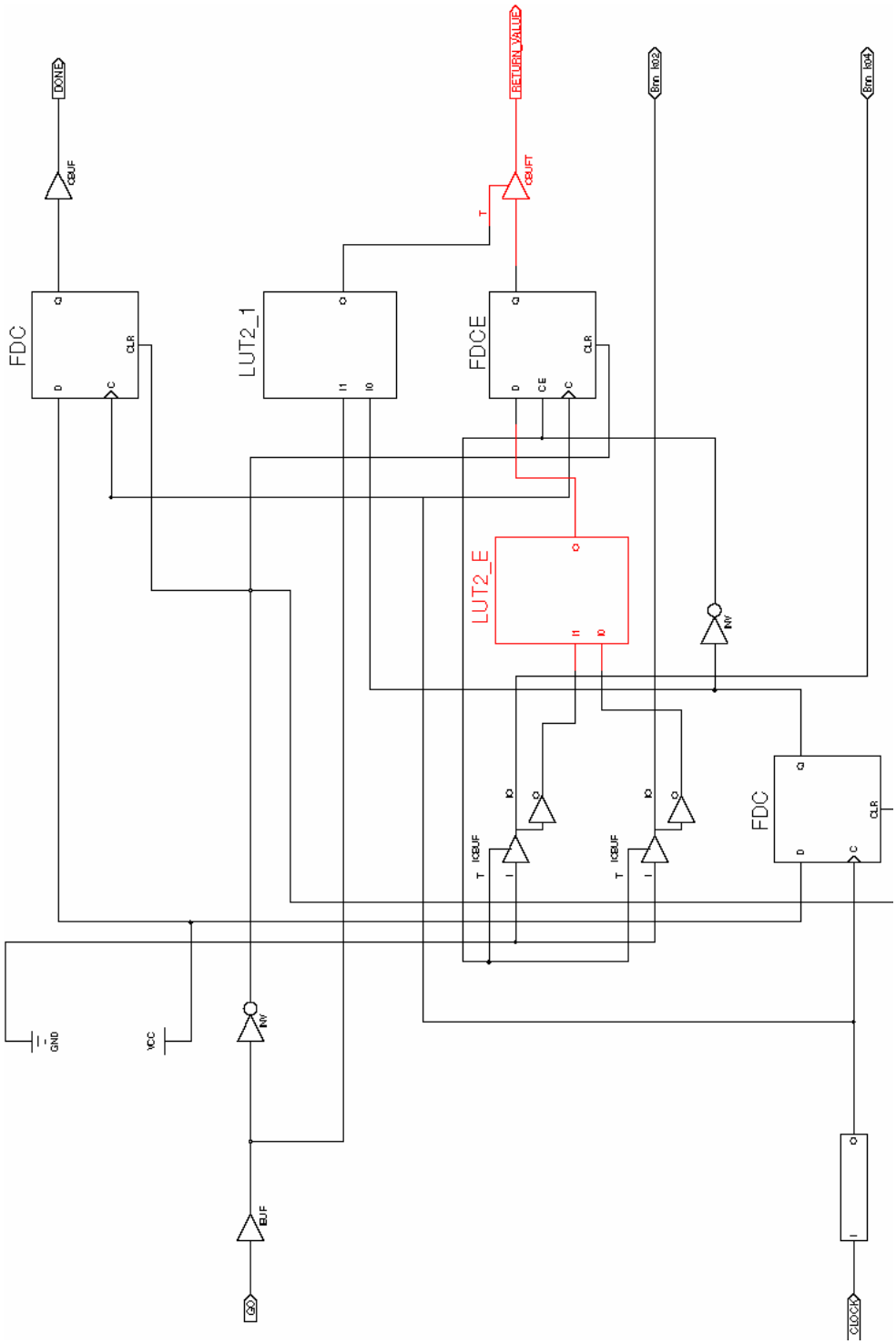


Abbildung C.11 Schaltplan für y4()

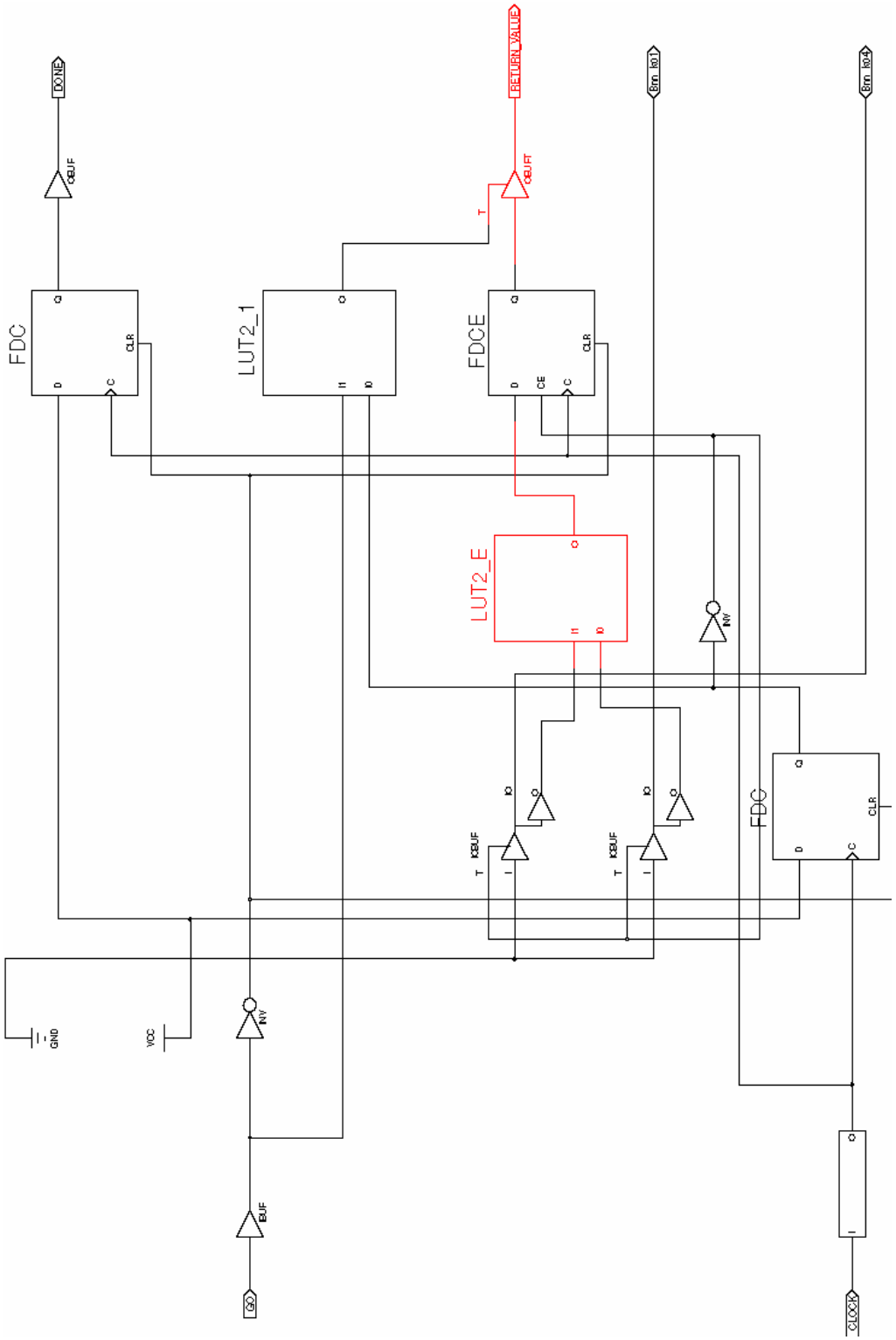


Abbildung C.12 Schaltplan für y5()

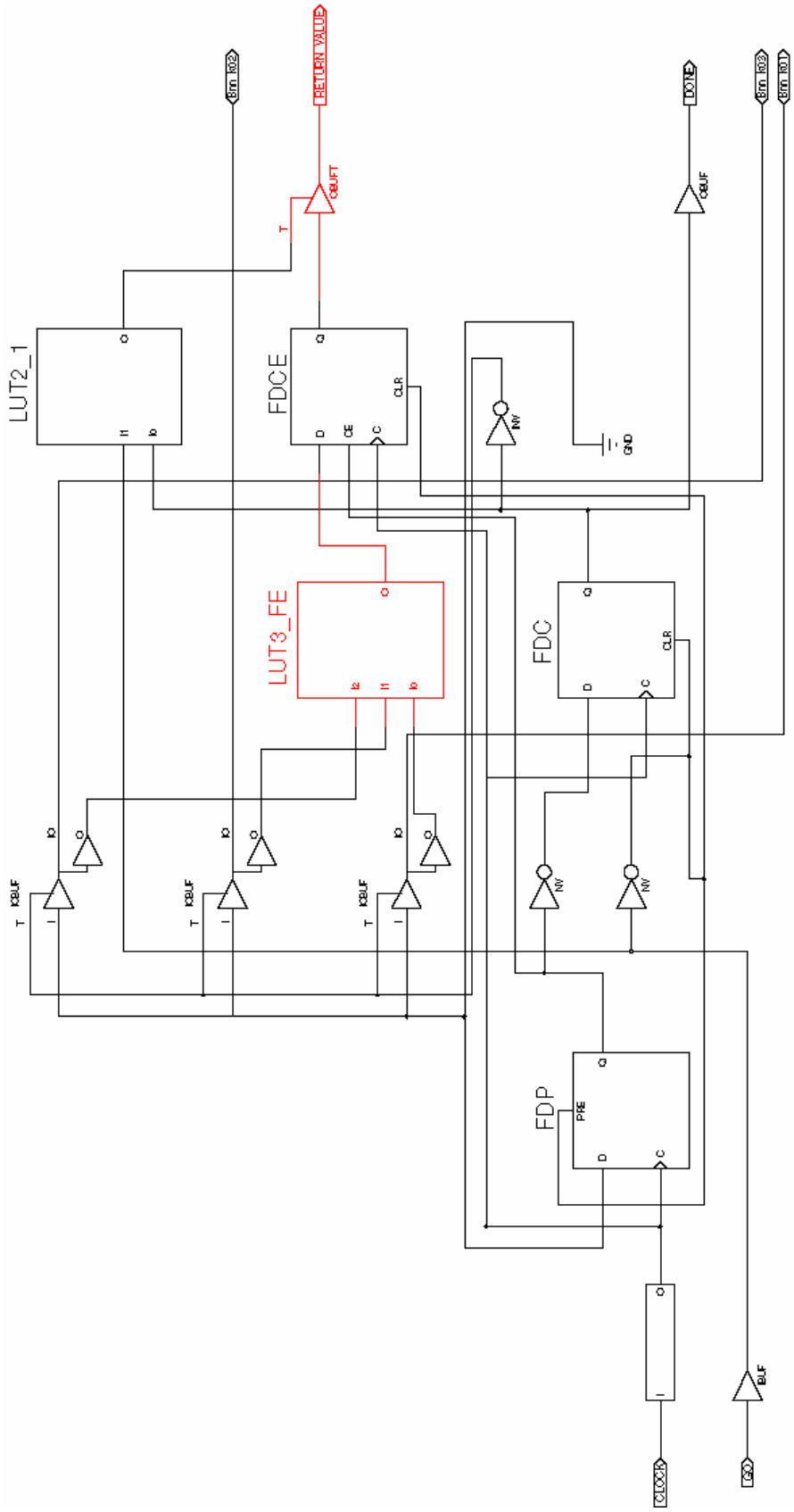


Abbildung C.13 Schaltplan für y6()

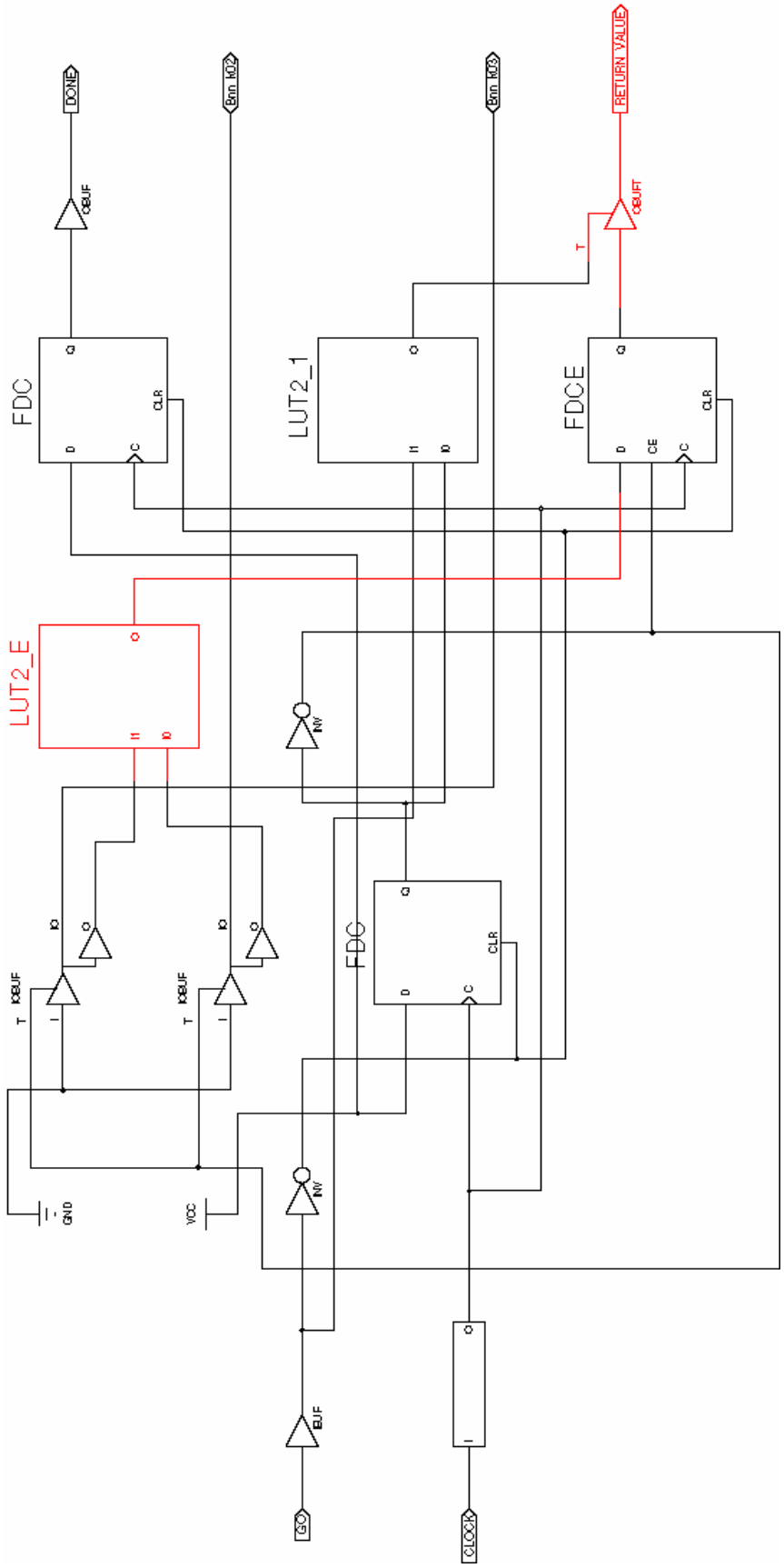


Abbildung C.14 Schaltplan für y7()

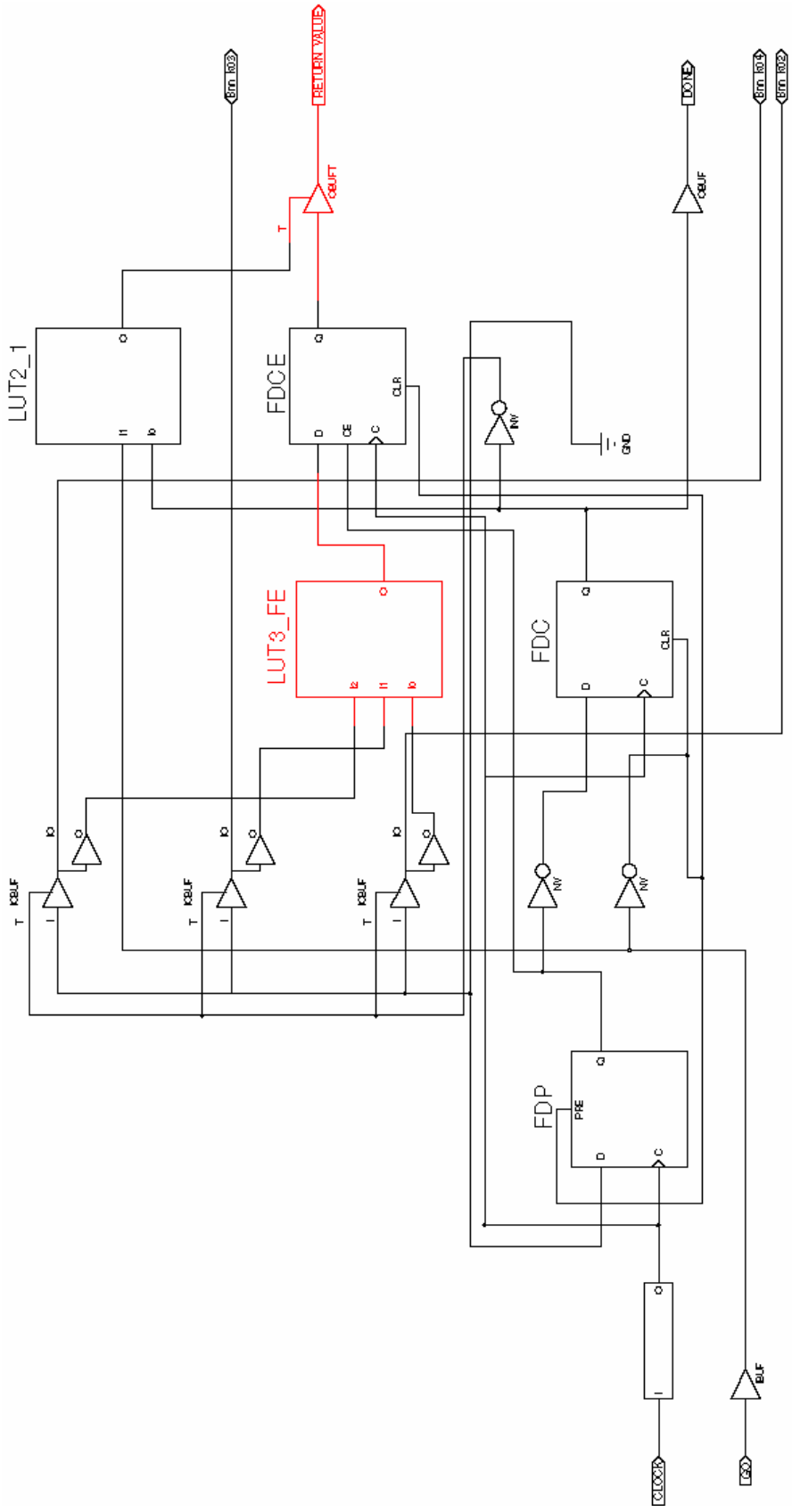


Abbildung C.15 Schaltplan für y8()

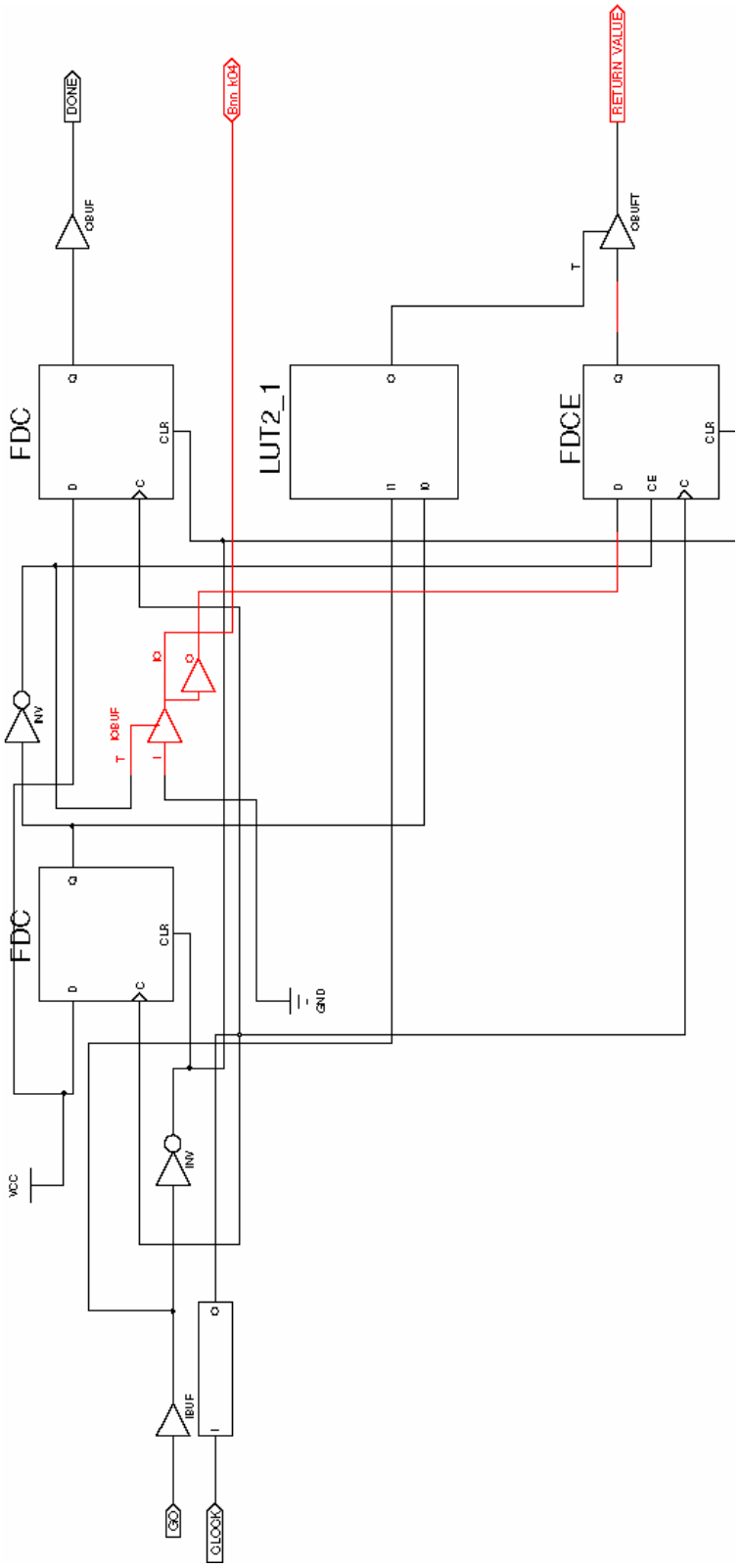


Abbildung C.16 Schaltplan für y9()

INDEX

U

V

W

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

GLOSSAR

A

Aktivierungsfunktion Aktivierungsfunktion gibt an, wie sich ein neuer Aktivierungszustand des Neurons aus dem alten Aktivierungszustand und der Netzeingabe des Neurons ergibt.

B

Boolesche Variable Element einer Booleschen Algebra, das immer einen von zwei Werten annimmt. Dieses Wertepaar wird je nach Anwendung u. a. als „wahr/falsch", „true/false" oder „1/0" bezeichnet.

Binary Decision Diagram (**BDD**) Datenstruktur zur Repräsentation Boolescher Funktionen. Binary Decision Diagrams werden vor allem im Bereich der Hardwaresynthese und -verifikation eingesetzt.

Binärvektor (**BV**) Binärvektor der Länge n ist ein n–Tupel aus Booleschen Variablen $\boldsymbol{x}=(x_1, x_2,\ldots, x_n)$: $x_i \in \{0,1\}$

Binärvektorliste (**BVL**) Tabelle, in der die Variablen des Raumes und die Werte des einzelnen BV spaltengerecht untereinander geschrieben werden.

Boolesches Neuronales Netz Neuronales Netz, das für die Bearbeitung Boolescher Daten vorgesehen ist. Die Ein- sowie Ausgabesignale des Netzes sind Boolesche Werte.

Boolesche Funktion (**logische Funktion**) Mathematische Funktion der Form $f: \boldsymbol{B}^n \rightarrow \boldsymbol{B}$ (auch allgemeiner $f: \boldsymbol{B}^n \rightarrow \boldsymbol{B}^m$). $\boldsymbol{B}$ ist dabei ein Boolescher Raum.

Boolescher Operator (**Operation**) Boolesche Funktion, der ein Operationssymbol zugeordnet ist.

Boolescher Raum Menge $\boldsymbol{B}^n=\{0,1\}^n$ aller möglichen Binärvektoren der Länge n ist der n-dimensionale Boolesche Raum.

C

Codesign Prozess des Modellierens, der Analyse, der Synthese und der Simulation von Systemen, die aus zusammenwirkenden Hardware- und Softwaremodulen bestehen.

D

Datenstruktur Bestimmte Art, Daten zu verwalten und miteinander zu verknüpfen, um in geeigneter Weise auf diese zugreifen und diese manipulieren zu können. Datenstrukturen sind immer mit bestimmten Operationen verknüpft, um eben diesen Zugriff und diese Manipulation zu ermöglichen.

don't care Minterm, der undefinierte Werte bezeichnet (z.B. '-', '*').

E

Euklidischer Abstand Mathematische Distanzfunktion für zwei Punkte oder Vektoren, die sich als euklidische Norm des Differenzvektors zwischen den beiden Punkten berechnet.

F

Funktionsdekomposition Zerlegung einer Funktion in Teilfunktionen.

G

Graph Mathematische Struktur, ein Paar $G = (V, E)$ aus Knoten (auch Ecken oder Punkte), die durch Kanten verbunden sein können.

H

Hamming-Abstand Anzahl der Stellen, in dem sich zwei Binärvektoren gleicher Länge unterscheiden.

K

Künstliches Neuron (**KN**) Mathematisches Modell einer Nervenzelle des menschlichen zentralen Nervensystems. Das Neuron kann als ein einfacher Prozessor gesehen werden.

Karnaugh-Plan Karnaugh-Veitch-Diagramm (KV-Diagramm) dient der übersichtlichen Darstellung und Vereinfachung Boolescher Funktionen.

L

Laterale Verbindung Verbindung zwischen Neuronen einer Schicht.

Lerndatenmenge (Trainingsdatenmenge) Datenmenge, die zum Training des neuronalen Netzes verwendet wird.

M

Minterm (**Elementarkonjunktion**) Konjunktionsterm, d.h. eine Anzahl von Literalen, die alle durch ein logisches AND verknüpft sind. Dabei müssen alle n Variablen der betrachteten n-stelligen Booleschen Funktion im Konjunktionsterm vorkommen.

N

Neuronales Netz (**NN**) Gerichteter Graph $G = (U, C)$, dessen Knoten $u \in U$ Neuronen (units) und dessen Kanten $c \in C$ Verbindungen (connections) heißen. NN ist ein System zur Informationsverarbeitung, das aus einer großen Anzahl einfacher parallel arbeitender Neuronen (Zellen, Einheiten) besteht.

P

Perzeptron Vereinfachtes Neuronenmodell, das zuerst von Frank Rosenblatt 1958 vorgestellt wurde.

Prozess Definierter Ablauf von Zuständen eines Systems.

S

Schwellwert Wert, der als Grenze für die Verarbeitung eines Signales verwendet wird.

Schwellwertfunktion Funktion mit einem Schwellwertparameter, die in Abhängigkeit vom Eingabewert und einer Schwellwert einen Booleschen Ausgabewert erzeugt.

T

Ternärvektor (**TV**) Dreiwertiger Vektor, in dem eine Variable durch „0“ beschrieben wird, wenn sie negiert auftritt; durch „1“, wenn sie nicht negiert auftritt; oder durch „-“, wenn sie nicht vorhanden ist.

Ternärvektorliste Datenstruktur zur Darstellung Boolescher Funktionen im Computer durch Tabellen. TVL besteht aus Ternärvektoren, die die Konjunktionen von Variablen einer Funktion in der disjunktiven Form oder Antivalenzform bzw. Disjunktionen von Variablen einer Funktion in der konjunktiven Form oder Äquivalenzform abbilden.

Training eines Neurons Ermitteln der Transferfunktion und der Gewichten w_{ih} für ein Neuron.

Trainingsmatrix Lerndatenmenge, die im Form eine Matrix dargestellt ist.

Transferfunktion Mathematische Beschreibung des Verhaltens eines Systems, das genau einen Eingang und einen Ausgang besitzt.

W

Wertetabelle Tabelle mit Spalten oder Zeilen, in die Argumente und die zugehörigen Funktionswerte einer Funktion eingetragen sind.

THESEN

ZUR DISSERTATION
„Neuronale Netze als Modell Boolescher Funktionen“

1. Die gewaltigen technologischen Fortschritte in der Mikroelektronik erfordern den Entwurf immer komplexerer Boolescher Schaltnetzwerke. Dabei müssen ständig neue leistungsfähigere Designmethoden entwickelt werden. Für diese werden kompaktere Datenstrukturen für Boolesche Funktionen und deren schnelle Verarbeitung benötigt.
2. In der vorliegenden Arbeit wurden Künstliche Neuronale Netze in ihrer ursprünglichen Intention betrachtet und zur Modellierung Boolescher Funktionen verwendet.
3. Die neu entwickelte Art von Booleschen Neuronalen Netzen eignet sich für die Darstellung und schnelle Verarbeitung Boolescher Funktionen.
4. Das Basiselement Boolescher Neuronaler Netze ist ein neuartiges Boolesches Neuron (BN), das im Gegensatz zum klassischen Neuron direkt mit Booleschen Signalen operiert und dafür ausschließlich Boolesche Operationen benutzt.
5. Der erarbeitete sequentielle Algorithmus für das Training der BNN garantiert eine schnelle Konvergenz und benötigt somit nur eine kurze Trainingzeit.
6. Dieser Trainingsalgorithmus bildet die Grundlage eines neuen geschaffen Verfahrens zur Architektursynthese der BNN.
7. Da das BNN einen sequentiellen Trainingsalgorithmus benutzt, treten die bekannten Probleme iterativer Trainingsmethoden wie z.B. lokale Minima oder lange Trainingszeit nicht auf. Das Verfahren zur Architektursynthese findet sowohl die optimale Netzwerkstruktur als auch die optimalen Netzparameter.
8. Die entwickelten BNN besitzen bedeutende Vorteile im Vergleich zu bekannten Booleschen Neuronalen Netzen. Das bisher für das Training notwendige Speichervolumen wurde deutlich vermindert. Durch die Verwendung Boolescher Neuronen mit Booleschen Transferfunktionen wird auch die Trainingsgeschwindigkeit sowohl des einzelnes Neurons auch des gesamtes Netzes erhöht.
9. Die entwickelte Trainingsmethode stellt darüber hinaus ein spezielles Dekompositionsverfahren Boolescher Funktionen dar. Eine Menge von Booleschen Funktionen $A=\{y_1(\boldsymbol{x}), y_2(\boldsymbol{x}), ..., y_{Ny}(\boldsymbol{x})\}$, $\boldsymbol{x}=(x_1, x_2, x_3,\ldots, x_n)$ wird beim Training in gemeinsame einheitliche Boolesche Basisfunktionen $k_1(\boldsymbol{x}), k_2(\boldsymbol{x}), ..., k_{Nz}(\boldsymbol{x})$ dekomponiert, die als Transferfunktionen der verborgenen Neuronen betrachtet werden. Die Booleschen Basisfunktionen hängen vom Vektor der Eingabensignale $\boldsymbol{x}$ ab. Eine Verknüpfung der bestimmten k-

Funktionen durch Basisoperation $\Omega \in \{\wedge, \vee, \oplus, \odot\}$ bildet die gegebene Boolesche Funktion oder Funktionsmenge. Aus den *k*-Funktionen werden alle gegebenen Booleschen Funktionen gebildet.

10. Die Flexibilität und die in den letzten Jahren stark gestiegene Leistungsfähigkeit von „Run-Time Reconfigurable" (RTR)-Systemen haben die Vorteile von Hardware-Realisierungen Künstlicher Neuronaler Netze und insbesondere Boolescher Neuronaler Netze verstärkt.
11. Da das Boolesches Neuron nur Booleschen Signale bearbeitet und nur Boolesche Operationen dafür benutzt, sind Boolesche Neuronale Netze besonders für die Hardware-Realisierung in FPGA-basierenden RTR-Systemen geeignet.
12. Der sehr hohe Aufwand der Hardware-Realisierung üblicher Neuronaler Netzen wurde durch die Verwendung von BN und BNN wesentlich vereinfacht.
13. Durch die Verwendung des Booleschen Neurons für die Modellierung Boolescher Funktionen in FPGA wird das Problem der ansonsten großen Anzahl von erforderlichen CLB für die Abbildung jedes Neurons gelöst.
14. Die Anzahl erforderlicher CLBs (configurable logic blocks) zur Realisierung eines Neurons wurde um 2 Größenordnungen verringert.
15. Es wurde sogar erreicht, dass 4 Boolesche Neuronen in einem CLB eines Virtex II-FPGAs Boolesche Neuronen realisiert werden können.
16. Jedes Boolesche Neuron mit seiner eigenen Logik kann dabei direkt in eine einzige LUT (lookup table) abgebildet werden, was ein großer Vorteil im Vergleich zu bekannten FPGA-Realisierungen von BNN ist.
17. Um diese äußerst kompakte Abbildung der BNN in eine FPGA-Struktur zu erreichen, wurde der Trainingsalgorithmus des BNN an die technologischen Randbedingungen der FPGA angepasst.
18. Um automatisierten Hardware/Software-Codesign der BNN unter Verwendung der MDA-Technologie zu ermöglichen wurden zur Darstellung der BNN UML-Modelle verwendet.
19. Die entwickelten Booleschen Neuronalen Netze leisten einen großen Beitrag zum Entwurf komplexerer Boolescher Schaltungsnetzwerke.
20. Die entwickelten Booleschen Neuronalen Netze haben sich als effiziente Modelle Boolescher Funktionen bewährt. Sie gewährleisten die effiziente Modellierung Boolescher Funktionen durch Neuronale Netze.
21. Die Fähigkeit der Künstlichen Intelligenz bzw. der Neuronalen Netze, neue Erkenntnisse zu erzeugen, wurde erfolgreich bei der Modellierung Boolescher Funktionen angewendet.