Technische Universität Dresden
Fakultät Informatik

Großer Beleg

zum Thema

Neuronale Netze zum induktiven Lernen über Aussagenlogischen Programmen

von
Jörg Wunderlich
geboren am 30.03.1973 in Dresden

eingereicht am 15. September 1997

Betreuer: Dipl.–Inf. Yvonne Kalinke
Verantwortlicher Hochschullehrer: Prof. Dr. Steffen Hölldobler

Inhaltsverzeichnis

1. Einleitung

Die Grundlage für den Erfolg der Wissensverarbeitung und deren Anwendung in der Wirtschaft ist die Erstellung von Expertensystemen. Expertensysteme sollen die Fähigkeiten und das Wissen menschlicher Experten über ein bestimmtes Gebiet speichern, verändern, anwenden, und somit selbst als Experten tätig sein. Dazu ist es notwendig, das vorhandene Wissen formal in Regeln zu fassen, die die Maschine repräsentieren und interpretieren kann. Die gebräuchlichste Form von Regeln bilden dabei logische Programme. Doch wie gewinnt man aus dem Wissen eines menschlichen Experten formale Regeln? Wie kann man Zusammenhänge finden, die nicht einmal menschlichen Experten bewußt sind? Und wie nutzt man bereits vorhandenes Wissen, um neues Wissen zu aquirieren? Mit diesen Fragen befaßt sich die vorliegende Arbeit unter dem Aspekt, dabei konnektionistische Methoden zum maschinellen Lernen zu nutzen. Neuronale Netze sind komplexe und starke Werkzeuge zum induktiven Lernen von Beispielen. Dementsprechend schwierig ist es jedoch, das konnektionistische Modell eines Künstlichen Neuronalen Netzes in Form von Regeln wiederzugeben, zumal Regeln der realen Welt sehr vielfältig sein können. Die Untersuchungen beschränken sich deshalb vorerst auf Wissen in aussagenlogischer Form. Ausgehend von bereits vorhandenem Wissen in Form eines aussagenlogischen Programmes wird ein Neuronales Netz erstellt, das die Semantik des Wissens darstellt. Danach wird das erstellte Netz mit neuem Wissen in Form von Beispielen trainiert. Der letzte Schritt besteht in der Extraktion von Regeln und ihrer Darstellung als logisches Programm, welches das neue Wissen repräsentiert.

Das erste Kapitel beschäftigt sich mit den Grundlagen der Wissensverarbeitung. Es beschreibt die Theorie logischer Programme und gibt eine Einführung in Neuronale Netze. Danach werden die wichtigsten Algorithmen für den Prozess der Wissensaquisition mit Neuronalen Netzen erläutert. Abschnitt 3.1 beschreibt dabei die Erstellung eines semantikgleichen Neuronalen Netzes aus einem aussagenlogischen Programm, Abschnitt 3.2 das induktive Lernen im Neuronalen Netz, und Abschnitt 3.3 das Extrahieren der Regeln aus dem Netz. Die Realisierung, auf denen die Tests basieren und die Ergebnisse der Untersuchungen werden in 4 und 5 gezeigt. Die Arbeit berichtet über bisher bekannte Möglichkeiten und Grenzen des induktiven Lernens über aussagenlogischen Programmen. Im Anhang A und B werden Hinweise zur Nachnutzung der Implementierungen gegeben. Ausgewählte graphische Darstellungen von Lernvorgängen werden im Anhang C gezeigt.

2. Grundlagen

2.1 Wissen und Wissensbasierte Systeme

Bevor man sich mit den speziellen Algorithmen und Methoden dieser Arbeit beschäftigt, ist zunächst zu klären, welche Bedeutung hinter grundlegenden Begriffen steht. Hennings [Hen91] gibt einen komplexen Überblick über die Probleme der Wissensverarbeitung. Er betrachtet dabei vor allem symbolische Ansätze. Eine Betrachtung konnektionistischer[1] Modelle zur Wissensverarbeitung liefern Nauck, Klawonn und Kruse ([NKK96]).

Definition 2.1.1 (Wissensbasis [Hen91]) *Eine Wissensbasis ist die Menge allen Wissens, welches zum Aufbau organisierter Wissensstrukturen mit Hilfe von Wissensrepräsentationsmechanismen oder –sprachen benutzbar gemacht werden kann, und in jedem* Wissensbasierten System *notwendig vorhanden sein muß.*

Der Begriff Wissensbasiertes System steht demzufolge als Oberbegriff f"ur alle Systeme, die wissensbasiert arbeiten.

Definition 2.1.2 (Expertensystem [Hen91]) *Expertensysteme sind Systeme, die nicht nur auf algorithmischen Prozessen unter Verwendung von Daten und Programmen operieren, sondern auch Wissen und Erfahrung nutzen, welches von (menschlichen) Experten u.U. computer–unterstützt gewonnen (akquiriert), artikuliert (repräsentiert), verarbeitet (manipuliert) und bewertet (evaluiert) werden kann. Deshalb gehören sie zur Gruppe der Wissensbasierten Systeme.*

Ein wichtiger Begriff in der Wissensverarbeitung ist die Inferenz bzw. das Schließen. Dabei gibt es unterschiedliche Formen. Die beiden wichtigsten Formen der Inferenz sind die Induktion und die Deduktion. Induktives Schließen versucht aus gegebenen Sachverhalten allgemeine Konzepte zu ermitteln. Deduktive Verfahren beruhen darauf, aus vorhandenem Wissen spezielle Zusammenhänge zu ermitteln, die durch das bereits vorhandene Wissen gestützt werden.

[1] basierend auf Neuronalen Netzen

2.2 Semantik logischer Programme

2.2.1 Prädikatenlogik erster Stufe

Die Prädikatenlogik erster Sufe ist eine einfache, ausdrucksstarke und gut formalisierte Sprache zur Abbildung von Ereignissen und Fakten der realen Welt. Sie eignet sich zur Darstellung von Wissen und darauf aufsetzenden Deduktionsmechanismen. Ein wichtiges Mittel zur Repräsentation von realem–Welt–Wissen sind deshalb logische Programme, die die Zusammenhänge zwischen verschieden Fakten beschreiben und somit erst das Erschließen neuer Zusammenhänge, neuen Wissens, ermöglichen. Die Theorie der logischen Programme bildet die Grundlage für die logische Programmiersprache PROLOG. Sie vereinigt den Aspekt der Theorie mit der Programmierung und der Erstellung von Applikationen und ermöglicht damit, daß Wissen unterschiedlichster Domänen in Form von Fakten und Regeln als PROLOG–Programm geschrieben werden kann. Logische Programmierung kann deshalb nicht nur zum Beweisen, sondern auch zum Rechnen verwendet werden.[2]

Die Theorie logischer Programme soll nun formalisiert werden. Die nachfolgenden Definitionen sind auf Lloyd [Llo93] und Reichel [Rei94] zurückzuführen.

Definition 2.2.1 (Alphabet) *Das Alphabet der Prädikatenlogik erster Stufe besteht aus sieben Symbolklassen:*

- *Variablen*
- *Konstanten*
- *Funktionssymbolen*
- *Prädikatssymbolen*
- *Junktoren*
- *Quantoren*
- *Begrenzungssymbolen*

Junktoren sind $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, Quantoren sind $\exists$, $\forall$ und Begrenzer sind (,) und ,.

Unter Junktoren, Quantoren und Begrenzern gibt es unterschiedliche Prioritäten in der Abarbeitung. Die Hierarchie ist dabei wie folgt: [3]

$$\begin{array}{c} \neg,\ \forall,\ \exists \\ \wedge \\ \vee \\ \rightarrow,\ \leftrightarrow \end{array}$$

[2] siehe auch [Apt97]

[3] höchste Priorität oben

Die Junktoren $\rightarrow$ und $\leftrightarrow$ können vereinfacht beschrieben werden durch:

$$\begin{array}{lll} A \rightarrow B & \text{gdw.} & \neg A \vee B \\ A \leftrightarrow B & \text{gdw.} & A \wedge B \vee \neg A \wedge \neg B \end{array}$$

Definition 2.2.2 (Term) *Ein Term ist induktiv definiert wie folgt:*

- *Eine Variable ist ein Term.*
- *Eine Konstante ist ein Term.*
- *Ist f ein n–stelliges Funktionssymbol und sind $t_1, \ldots, t_n$ Terme, dann ist $f(t_1, \ldots, t_n)$ ein Term.*

Definition 2.2.3 (Formel) *Eine Formel ist induktiv definiert wie folgt:*

- *Ist p ein n–stelliges Prädikatssymbol und sind $t_1, \ldots, t_n$ Terme, dann ist $p(t_1, \ldots, t_n)$ eine Formel (genannt* atomare Formel *oder kurz* Atom*).*
- *Sind F und G Formeln, dann sind auch $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$ und $F \leftrightarrow G$ Formeln.*
- *Ist F eine Formel und X eine Variable, dann sind auch $\forall X.F$ und $\exists X.F$ Formeln.*

Definition 2.2.4 (Sprache) *Die Sprache der Prädikatenlogik erster Stufe besteht aus der Menge aller Formeln, die aus den Symbolen des Alphabets konstruiert werden können.*

Die Bezeichnung von Elementen in einer Formel kann hinsichtlich der Semantik sehr hilfreich, aber auch irreführend sein. So folgert man aus der Formel $Stadt(Dresden)$ sehr schnell, daß Dresden eine Stadt ist. Genausogut könnte das Prädikat $Stadt$ aber auch bedeuten, daß Dresden ein Dorf ist, oder die Konstante $Dresden$ könnte für Leipzig stehen und die Formel sagt, daß Leipzig eine Stadt ist. Für die Formel sind aber auch noch ganz andere Bedeutungen möglich, die mit den Bezeichnern überhaupt keine Verbindung haben. Deshalb wollen wir eine Konvention einführen, in der wir alle Variablen mit Großbuchstaben X, Y, Z, Konstanten mit Kleinbuchstaben $a, b, c, \ldots$, Funktionszeichen mit $f, g, h, \ldots$ und Prädikatszeichen mit $p, q, r, \ldots$ bezeichnen.

Definition 2.2.5 (Auftreten einer Variable) *Der Gültigkeitsbereich eines Quantors $\forall X$ ($\exists X$) in $\forall X.F$ ($\exists X.F$) ist F. Ein* gebundenes Auftreten *einer Variable in einer Formel ist ein Auftreten direkt nach einem Quantor, oder innerhalb des Gültigkeitsbereiches des Quantors. Jedes andere Auftreten einer Variable ist* frei.

Eine streng formalisierte Definition für die Menge der freien Variablen liefert Reichel [Rei94].

Definition 2.2.6 (freie Variablen) *Für jede Formel F definieren wir die Menge $var(F)$ der* freien Variablen *in F induktiv durch:*

$$\begin{aligned}
&var(p(t_1, \ldots, t_n)) = \textstyle\bigcup_{i=1}^{n} var(t_i),\\
&var(\neg F) = var(F),\\
&var(F \wedge G) = var(F) \cup var(G),\\
&var(F \vee G) = var(F) \cup var(G),\\
&var(\exists X.F) = var(F) - \{X\},\\
&var(\forall X.F) = var(F) - \{X\}
\end{aligned}$$

mit $var(X) = \{X\}$, $var(c) = \emptyset$ und $var(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} var(t_i)$.

Definition 2.2.7 (Grundterm) *Ein Term t mit $var(t) = \emptyset$ wird als Grundterm bezeichnet.*

Definition 2.2.8 (Grundatom) *Eine atomare Formel A mit $var(A) = \emptyset$ wird als Grundatom bezeichnet.*

Definition 2.2.9 (abgeschlossene Formel) *Eine Formel F mit $var(F) = \emptyset$ wird als abgeschlossene Formel bezeichnet .*

Definition 2.2.10 (Literal) *Ein Literal ist ein Atom oder ein negiertes Atom. Ein* positives Literal *ist ein Atom. Ein* negatives Literal *ist die Negation eines Atomes.*

Definition 2.2.11 (Programmklausel) *Eine Programmklausel ist eine Klausel der Form*

$$A \leftarrow L_1, \ldots, L_m$$

wobei A ein Atom ist und $L_1, \ldots, L_m$ Literale. Wir bezeichnen A als Kopf der Klausel *und $L_1, \ldots, L_m$ als* Körper der Klausel.

Definition 2.2.12 (definite Programmklausel) *Eine definite Programmklausel ist eine Klausel der Form*

$$A \leftarrow A_1, \ldots, A_m$$

wobei $A, A_1, \ldots, A_m$ Atome sind.

Eine Programmklausel, die im Klauselkörper kein Literal enthält, nennt man einen Fakt. Ein Fakt ist eine definite Programmklausel und wird in der Form $A \leftarrow$ oder A geschrieben. Ein Fakt legt fest, daß das Atom unabhängig von anderen Literalen immer gilt.

Definition 2.2.13 (normales Programm) *Ein normales Programm ist eine endliche Menge von Programmklauseln.*

Definition 2.2.14 (definites Programm) *Ein definites Programm ist eine endliche Menge von definiten Programmklauseln.*

Zur Vereinfachung werden normale logische Programme im folgenden kurz als Programme bezeichnet.

Definition 2.2.15 (Herbrand–Universum) *Das Herbrand–Universum $U_{\mathcal{P}}$ für ein Programm $\mathcal{P}$ ist die Menge aller Grundterme, die mit den Konstanten und Funktionssymbolen, die in $\mathcal{P}$ vorkommen, gebildet werden können. Falls in $\mathcal{P}$ keine Konstanten vorkommen, definieren wir eine beliebige Konstante, um Grundterme formen zu können.*

Definition 2.2.16 (Herbrand–Basis) *Die Herbrand–Basis $B_{\mathcal{P}}$ für ein Programm $\mathcal{P}$ ist die Menge aller Formeln $p(t_1, \ldots, t_n)$, die mit allen in $\mathcal{P}$ vorkommenden n–stelligen Prädikatszeichen p und den Termen $t_1, \ldots, t_n$ des Herbrand–Universums $U_{\mathcal{P}}$ gebildet werden können.*

Beispiel 2.2.1 *Gegeben sei folgendes logische Programm P:*

$$\begin{array}{rcl} p(f(a)) & \leftarrow & \\ q(X,a) & \leftarrow & p(X) \\ q(X,Y) & \leftarrow & q(Y,X) \end{array}$$

Die erste Klausel von P ist ein Fakt und die anderen Klauseln sind ebenfalls definit. Folglich ist P ein definites logisches Programm. Wir bestimmen nun für P das Herbrand–Universum U_P und die Herbrand–Basis B_P mit

$$\begin{array}{rcl} U_P & = & \{a, f(a), f(f(a)), f(f(f(a))), \ldots\} \\ B_P & = & \{p(a), q(a,a), p(f(a)), q(a, f(a)), q(f(a), a), q(f(a), f(a)), p(f(f(a))), \ldots\} \end{array}$$

Wie man leicht sieht, sind beide Mengen abzählbar unendlich. Verallgemeinert kann man sagen, daß alle prädikatenlogischen Programme mit mindestens einem Funktionssymbol und einem n–stelligen Prädikatszeichen ($n \geq 1$) ein abzählbar unendliches Herbrand–Universum und eine abzählbar unendliche Herbrand–Basis haben.

Definition 2.2.17 (Wahrheitswerte, Herbrand–Interpretation) *Die Elemente der Menge $\{\ \mathbf{0}\ ,\ \mathbf{1}\ \}$ werden Wahrheitswerte genannt. Eine Herbrand–Interpretation I für ein Programm $\mathcal{P}$ ist eine Abbildung $I : B_P \rightarrow \{\ \mathbf{0}\ ,\ \mathbf{1}\ \}$ der Elemente der Herbrand–Basis von $\mathcal{P}$ auf die Wahrheitswerte 0 für* **falsch** *bzw. 1 für* **wahr** *.*

Die Schreibweise $A \in I$ bzw. $A \notin I$ ($A \in B_P$) sei äquivalent zu $I(A) = \mathbf{1}$ bzw. $I(A) = \mathbf{0}$. Wir vereinbaren außerdem $I(\neg A) = \mathbf{1}$ gdw. $I(A) = \mathbf{0}$.

Nach der Interpretation von Atomen wollen wir die Interpretation I^* von Formeln betrachten. Sie ist eine Erweiterung von I. Seien F und G Formeln. Es gilt:

$$\begin{aligned}
I^*(F) &= I(F) \text{ falls } F \in B_{\mathcal{P}} \\
I^*((F \wedge G)) &= \begin{cases} 1 & \text{falls} \quad I^*(F) = 1 \text{ und } I^*(G) = 1 \\ 0 & \text{sonst} \end{cases} \\
I^*((F \vee G)) &= \begin{cases} 1 & \text{falls} \quad I^*(F) = 1 \text{ oder } I^*(G) = 1 \\ 0 & \text{sonst} \end{cases} \\
I^*(\neg F) &= \begin{cases} 1 & \text{falls} \quad I^*(F) = 0 \\ 0 & \text{sonst} \end{cases}
\end{aligned}$$

Definition 2.2.18 (Modell) *Ist I eine Herbrand–Interpretation und F eine abgeschlossene Formel, dann ist I ein Modell für F (notiert mit $I \models F$), falls $I^*(F) =$* **1** *.*

Definition 2.2.19 (Modell) *Ist I eine Herbrand–Interpretation und S eine Menge von Formeln, dann ist I ein Modell für S, falls I ein Modell für jede Formel in S ist.*

Definition 2.2.20 (logische Konsequenz) *Sei S eine Menge abgeschlossener Formeln und F eine abgeschlossene Formel. Wir bezeichnen F als eine logische Konsequenz aus S, falls für jede Interpretation I gilt:*

$$I \models S \quad \Rightarrow \quad I \models F$$

Definition 2.2.21 (Herbrand–Modell) *Sei $\mathcal{P}$ ein normales logisches Programm. Ein Herbrand–Modell für $\mathcal{P}$ ist eine Herbrand–Interpretation, die Modell für $\mathcal{P}$ ist.*

Die *deklarative Semantik* eines Programes $\mathcal{P}$ kann durch ein Herbrand–Modell beschrieben werden. Wählt man aus der Menge aller Herbrand–Interpretationen das bezüglich der Inklusionsrelation $\subseteq$ kleinste Modell M_P aus, so ist dies für definite Programme eindeutig. Das gefundene Modell M_P entspricht dem Durchschnitt aller Herbrand–Modelle für P und der Menge der logischen Konsequenzen aus P. Genau diese Menge beschreibt die Bedeutung des Programmes, die deklarative Semantik. Für die Berechnung des kleinstes Modelles nutzen wir einen Operator T_P.

Definition 2.2.22 (Operator $\mathbf{T_P}$) *Sei P ein normales logisches Programm. Die Abbildung $T_P : 2^{B_P} \rightarrow 2^{B_P}$ ist wie folgt definiert. Sei I eine Herbrand–Interpretation für P. Dann gilt $T_P(I) = \{A \in B_P \mid A \leftarrow L_1, L_2, \ldots, L_n$ ist Grundinstanz einer Klausel in P und $I(L_1) = I(L_2) = \cdots = I(L_n) =$* **wahr** *$\}$.*

Für die Klasse der definiten Programme kann gezeigt werden, daß die Menge der Herbrand–Interpretationen für P einen vollständigen Verband bildet und die Funktion T_P monoton ist. Damit ist der kleinste Fixpunkt von T_P eindeutig bestimmt und entspricht genau der Menge M_P.

Definition 2.2.23 (Fixpunkt) *Sei $T : L \rightarrow L$ eine Abbildung der Menge L auf sich selbst. Wir bezeichnen a ($a \in L$) als Fixpunkt von T, falls $T(a) = a$.*

Definition 2.2.24 ($\mathbf{T} \uparrow \alpha$) *Sei $T : L \rightarrow L$ eine Abbildung der Menge L auf sich selbst und $\perp$ das kleinste Element von L. Wir definieren*

$$\begin{aligned} T \uparrow 0 &= \perp \\ T \uparrow \alpha &= T(T \uparrow (\alpha - 1)) \end{aligned}$$

Proposition 2.2.1 ([Tar55]) *Sei L ein vollständiger Verband und $T : L \rightarrow L$ sei monoton. Dann hat T einen kleinsten Fixpunkt[4] $\mathrm{lfp}(T)$ und einen größten Fixpunkt[5] $\mathrm{gfp}(T)$.*

Proposition 2.2.2 ([Llo93]) *Sei L ein vollständiger Verband und $T : L \rightarrow L$ sei stetig. Dann gilt $\mathrm{lfp}(T) = T \uparrow \omega$ mit $\omega \rightarrow \infty$.*

Die Monotonie einer Abbildung kann man aus der Stetigkeit folgern, weil die Stetigkeit eine strengere Eigenschaft ist. Lloyd zeigt, daß für ein definites Programm P der Operator T_P stetig ist.

Beispiel 2.2.2 *Betrachten wir unser Programm aus Beispiel 2.2.1:*
Die Interpretation $I = \{p(f(a)), q(f(a), a), q(a, f(a))\}$ ordnet den 3 in ihr enthaltenen Grundatomen den Wahrheitswert **wahr** *und allen anderen Formeln aus der Herbrand–Basis B_P den Wahrheitswert* **falsch** *zu. I ist ein Element aus der Potenzmenge*

$$2^{B_P} = \{\emptyset, \{p(a)\}, \{p(f(a))\}, \ldots, \{p(a), q(a,a)\}, \{p(a), q(f(a), a)\}, \ldots, \{p(a), p(f(a)), q(a,a)\}, \ldots\}$$

und gleichzeitig ein Herbrand–Modell für P, weil I ein Modell für jede Formel in P ist.
Die Iteration von T_P ergibt:

$$\begin{aligned} T_P \uparrow 0 &= \perp = \emptyset \\ T_P \uparrow 1 &= T_P(\emptyset) = \{p(f(a))\} \\ T_P \uparrow 2 &= T_P(\{p(f(a))\}) = \{p(f(a)), q(f(a), a)\} \\ T_P \uparrow 3 &= T_P(\{p(f(a)), q(f(a), a)\}) = \{p(f(a)), q(f(a), a), q(a, f(a))\} \\ T_P \uparrow 4 &= T_P(\{p(f(a)), q(f(a), a), q(a, f(a))\}) = \{p(f(a)), q(f(a), a), q(a, f(a))\} = T_P \uparrow 3 = I \end{aligned}$$

I entspricht dem kleinsten Fixpunkt $lfp(T_P)$ und beschreibt die Semantik von P.

[4] engl. least fixed point
[5] engl. greatest fixed point

2.2.2 Aussagenlogik

Im Gegensatz zur Prädikatenlogik erster Stufe besteht das Alphabet der Aussagenlogik lediglich aus Variablen, Junktoren und Begrenzungssymbolen. Damit ist die Aussagenlogik in ihren Repräsentationsmöglichkeiten weit weniger mächtig als die Prädikatenlogik. Das sieht man auch daran, daß für die Darstellung $Stadt(Dresden)$ in der Aussagenlogik eine aussagenlogische Variable $DresdenistStadt$ benötigt wird. Will man einen solchen Zusammenhang für andere Städte darstellen, so kann dies in Prädikatenlogik mit demselben Prädikat getan werden, aber in der Aussagenlogik ist für jede Stadt eine neue Variable notwendig.

Die Aussagenlogik als Vereinfachung der Prädikatenlogik eignet sich hervorragend, um komplizierte Methoden der Prädikatenlogik zu untersuchen. Eine aussagenlogische Variable entspricht im prädikatenlogischen Sinn einem 0–stelligen Prädikatszeichen. Damit können auch die Definitionen der Prädikatenlogik auf die Aussagenlogik reduziert werden. Ein Vorteil, der daraus sofort ersichtlich wird ist, daß die Herbrand–Basis B_P eines aussagenlogischen Programmes stets eine endliche Menge ist.

Wir stellen aussagenlogische Variablen durch Großbuchstaben $A, B, C, \ldots$ dar.

Beispiel 2.2.3 *Gegeben sei eine Menge aussagenlogischer Formeln:*

$$\begin{aligned} A &= B \wedge C \vee \neg B \wedge \neg C \\ B &= C \\ C &= \textbf{wahr} \end{aligned}$$

Betrachtet man das äquivalente aussagenlogische Programm P mit

$$\begin{aligned} A &\leftarrow B, C \\ A &\leftarrow \neg B, \neg C \\ B &\leftarrow C \\ C &\leftarrow \end{aligned}$$

so ergibt sich

$$\begin{aligned} U_P &= \emptyset \\ B_P &= \{A, B, C\} \\ 2^{B_P} &= \{\emptyset, \{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\} \\ T_P \uparrow 0 &= \bot = \emptyset \\ T_P \uparrow 1 &= T_P(\emptyset) = \{A, C\} \\ T_P \uparrow 2 &= T_P(\{C\}) = \{B, C\} \\ T_P \uparrow 3 &= T_P(\{B, C\}) = \{A, B, C\} \\ T_P \uparrow 4 &= T_P(\{A, B, C\}) = \{A, B, C\} = T_P \uparrow 3 \end{aligned}$$

Daraus folgt: $lfp(T_P) = \{A, B, C\}$

2.3 Künstliche Neuronale Netze

Künstliche Neuronale Netze bestehen aus einer endlichen Anzahl von Neuronen[6] und reell gewichteten Verbindungen zwischen ihnen.

Das Verhalten jeder Einheit wird durch ihren Schwellwert[7] θ, eine Aktivierungsfunktion $\psi : R^n \to R$ und eine Ausgabefunktion $\phi : R \to R$ bestimmt. Die Netzeingabe (das Potential) einer Einheit V_i ist die Summe der gewichteten Ausgaben von allen Einheiten V_j, mit denen eine Verbindung w_{ij} existiert, die von V_j nach V_i verläuft.

Jedes Neuron berechnet die Funktionskomposition $\psi\phi : R^n \to R$, aus der Aktivierungsfunktion und der Ausgabefunktion. Das Argument dieser Funktionskomposition ist die Netzeingabe der Einheit.

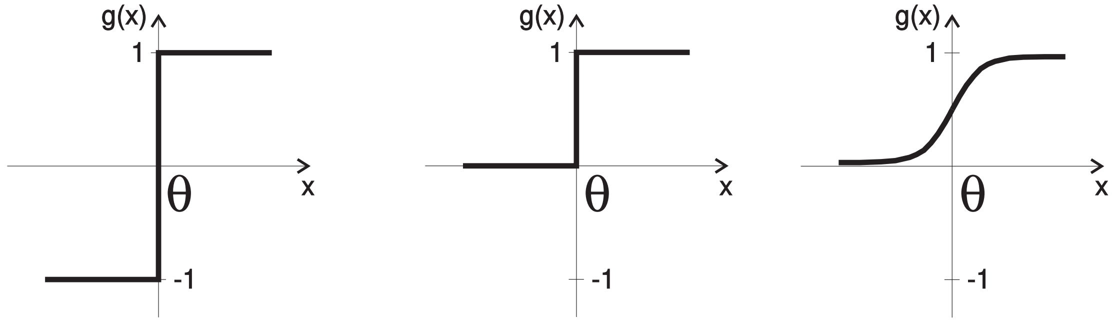
Die am häufigsten verwendeten Aktivierungsfunktionen sind die Vorzeichenfunktion (SIGN–Funktion), die Schwellwertfunktion (STEP–Funktion) und sigmoide Funktionen. Tabelle 2.1 zeigt den Verlauf dieser Funktionen. Die STEP–Funktion und die SIGN–Funktion sind lineare, nichtstetige Funktionen und können die Aktivierung einer Einheit lediglich auf 2 verschiedene Werte abbilden. Neuronen mit einer STEP–Aktivierungsfunktion heißen deshalb auch binäre Schwellwerteinheiten, weil sie die Aktivierung auf die binären Werte **0** und **1** abbilden und damit mit Wahrheitswerten angeben, ob der interne Schwellwert überschritten wurde oder nicht. Sigmoide Funktionen sind nichtlineare, stetige Funktion mit unendlich vielen Funktionswerten, die sich mit dem Parameter $s \to \infty$ beliebig gegen die Schwellwertfunktion annähern lassen. Als Standard wird für die Sigmoide Funktion ein Wert $s = 1$ verwendet. In den folgenden Ausführungen wird die Aktivierungsfunktion einer Einheit mit g bezeichnet.

Die Ausgabefunktion bei Künstlichen Neuronalen Netzen wird meist nur angewendet, wenn die Aktivierung einer Einheit zur Berechnung ihres Folgezustandes herangezogen wird. Da dies nur in einigen Modellen Künstlicher Neuronaler Netze auftritt, wird als Ausgabefunktion häufig die Identitätsfunktion $f(x) = x$ verwendet. Damit ist die Aktivierung einer Einheit gleich der Ausgabe einer Einheit. In den weiteren Betrachtungen werden wir uns deshalb auf die Aktivierungsfunktion beschränken und die Ausgabefunktion außer acht lassen.

Die existierenden Modelle Künstlicher Neuronaler Netze finden in unterschiedlichen Gebieten Anwendung. Die folgenden Ausführungen beschränken sich auf vorwärtsgerichtete Netze, da sie für unseren Problembereich am geeignetsten sind. Eine breitgefächerte Einführung in die Theorie Neuronaler Netze liefern Hertz, Krogh und Palmer [HKP91], deren Notation für vorwärtsgerichtete Netze in den folgenden Abschnitten verwendet wird.

[6] der Begriff entstammt dem biologischen Vorbild; im Künstlichen Neuronalen Netz spricht man auch kurz von Einheiten

[7] häufig wird auch der Begriff Bias verwendet



$g(x) = \begin{cases} 1 & : \text{ falls } \quad x \geq \theta \\ -1 & : \text{ sonst} \end{cases}$	$g(x) = \begin{cases} 1 & : \text{ falls } \quad x \geq \theta \\ 0 & : \text{ sonst} \end{cases}$	$g(x) = \frac{1}{1+e^{-s(x-\theta)}}$

Tabelle 2.1: Skizzen der Aktivierungsfunktionen SIGN, STEP und Sigmoid für einen Schwellwert $\theta = 0$. Die Standard–Sigmoid–Funktion verwendet den Parameter $s = 1$. Mit $s \to \infty$ nähert sich die Sigmoid–Funktion beliebig genau an die STEP–Funktion an.

2.3.1 Vorwärtsgerichtete Neuronale Netze

In einem vorwärtsgerichteten Netz sind die Neuronen schichtartig angeordnet. Es gibt eine Eingabeschicht, eine Ausgabeschicht und eine endliche Zahl von Schichten, die dazwischen liegen. Diese Schichten werden interne oder verdeckte Schichten genannt. Jedes Neuron im Netz ist einer der Schichten fest zugeordnet. Die Verbindungen zwischen den Neuronen sind gerichtet und gehen immer in der Richtung von der Eingabeschicht zur Ausgabeschicht. Es gibt keine Verbindungen zwischen Einheiten ein und derselben Schicht und es gibt keine Verbindungen zu Neuronen vorhergehender Schichten.

Eine Kante w_{ij}^m im vorwärtsgerichteten Netz verbindet die Einheit V_j^{m-1} in der Schicht $m-1$ mit der Einheit V_i^m in der Schicht m. Die Schicht 0 bezeichnet die Eingabeschicht und die Schicht M $(0 \leq m \leq M)$ die Ausgabeschicht des Netzes.

Bemerkung:
Die Bezeichnung V_i^m steht gleichermaßen für die Einheit und deren Aktivierung bzw. Ausgabe.

Eingabeneuronen haben nur Verbindungen zu verdeckten Einheiten oder zu Ausgabeeinheiten. Ihre Eingaben werden extern angelegt und ohne interne Berechnung gewichtet an die Nachfolgereinheiten weitergegeben. Ausgabeneuronen haben nur Verbindungen von Eingabeeinheiten oder von verdeckten Einheiten. Ihre Aktivierungen werden nach außen als Ausgabe des Netzes abgegeben. Verdeckte Einheiten haben sowohl Verbindungen von verdeckten Einheiten oder Eingabeeinheiten, als auch zu verdeckten Einheiten oder Ausgabeeinheiten.

Die Netzeingabe h_i^m eines Neurons V_i^m berechnet sich wie folgt:

$$h_i^m = \sum_j w_{ij}^m V_j^{m-1}$$

Die Aktivierung[8] V_i^m einer Einheit berechnet sich aus

$$V_i^m = g(h_i^m) = g(\sum_j w_{ij}^m V_j^{m-1})$$

wobei g die Aktivierungsfunktion darstellt.

An die Neuronen der Eingabeschicht wird ein Eingabevektor angelegt. Die Neuronen berechnen schichtweise synchron ihre Aktivierung und geben sie über die Verbindungen gewichtet an die nachfolgenden Einheiten weiter. Die Neuronen der Ausgabeschicht bilden die Ausgabe des Netzes. Sei k_e die Anzahl der Einheiten in der Eingabeschicht und k_a die Anzahl der Einheiten in der Ausgabeschicht, so berechnet das Netz eine deterministische Funktion

$$f : R^{k_e} \to R^{k_a}$$

Für einen Betrachter von außen haben nur die Eingabe– und die Ausgabeneuronen eine ersichtliche Bedeutung. Das ist der Grund, warum die internen Schichten verdeckte oder auch versteckte Schichten genannt werden.

Definition 2.3.1 (einfaches Perzeptron) *Ein einfaches Perzeptron ist ein vorwärtsgerichtetes Netz ohne verdeckte Schichten und mit genau einer binären Schwellwerteinheit in der Ausgabeschicht.*

Abbildung 2.1 zeigt ein einfaches Perzeptron.

Definition 2.3.2 (Perzeptron) *Ein Perzeptron ist ein vorwärtsgerichtetes Netz ohne verdeckte Schichten und einer endlichen Zahl $n \geq 1$ von binären Schwellwerteinheiten in der Ausgabeschicht. Jede Einheit in der Ausgabeschicht arbeitet wie ein einfaches Perzeptron.*

Definition 2.3.3 (mehrschichtiges Netz) *Ein mehrschichtiges vorwärtsgerichtetes Netz ist ein vorwärtsgerichtetes Netz mit mindestens einer verdeckten Schicht.*

Definition 2.3.4 (schichtweise vollvernetzt) *Ein vorwärtsgerichtetes Netz heißt schichtweise vollvernetzt, wenn für jedes Paar (V_i^{m+1}, V_j^m) von Einheiten benachbarter Schichten eine Verbindung w_{ij}^{m+1} (für alle i, j, m) im Netz existiert. Bei Betrachtungen über die Leistungsfähigkeit von vorwärtsgerichteten Netzen wird immer von einer schichtweise vollvernetzten Topologie ausgegangen.*

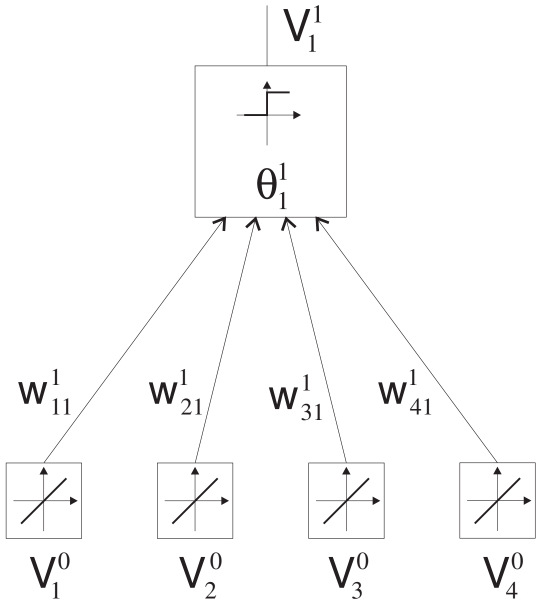


Abbildung 2.1: Ein einfaches Perzeptron, bestehend aus 4 Eingabeeinheiten und einer Ausgabeeinheit. Die Aktivierung der Ausgabeeinheit wird mit einer Schwellwertfunktion berechnet. Die Eingabeeinheiten geben die extern angelegte Eingabe gewichtet an die Ausgabeeinheit weiter. Dies wird dargestellt durch eine Identitätsfunktion als Aktivierungsfunktion.

In der Literatur werden für mehrschichtige Netze unterschiedliche Zählweisen angegeben. Viele Autoren definieren ein einfaches Perzeptron als einschichtiges Neuronales Netz. Sie zählen die Eingabeschicht nicht als Schicht mit, weil sie keine Berechnung durchführt und die Eingabevektoren lediglich an die folgenden Schichten weitergibt. Wir wollen aus Gründen der Übersichtlichkeit die Eingabeschicht als vollwertige Schicht aktzeptieren und mitzählen. Ein n–schichtiges vorwärtsgerichtetes Netz besteht demnach aus einer Eingabeschicht, einer Ausgabeschicht und $n-2$ verdeckten Schichten. Ein Perzeptron ist demzufolge ein 2–schichtiges vorwärtsgerichtetes Netz.

Proposition 2.3.1 *Ein Perzeptron ist in der Lage linear separable Funktionen zu berechnen.*

Definition 2.3.5 (linear separabel [Kal94]) *Zwei Mengen A und B von Punkten im n–dimensionalen Raum sind* linear separabel*, falls reelle Zahlen $w_1, \ldots, w_n$ und γ*

[8] hier: Aktivierung gleich Ausgabe

existieren, so daß für jeden Punkt $(x_1, \ldots, x_n) \in A$

$$\sum_{i=1}^{n} w_i x_i \geq \gamma$$

und für jeden Punkt $(x_1, \ldots, x_n) \in B$

$$\sum_{i=1}^{n} w_i x_i < \gamma$$

gilt.

Für den 2–dimensionalen Fall bedeutet diese Definition anschaulich, daß eine Gerade gefunden werden kann, die alle Punkte der Ebene entsprechend der Zugehörigkeit zu den Mengen A und B voneinander trennt.

Proposition 2.3.2 *Ein n–schichtiges vorwärtsgerichtetes Netz ($n > 2$) ist in der Lage jede Borel–meßbare Funktion zu approximieren.*

Die Klasse der Borel–meßbaren Funktionen ist die kleinste Klasse, die die stetigen Funktionen enthält. Sie ist bezüglich der Summe, der Differenz und des Produktes endlich vieler dieser Funktionen und bezüglich des Grenzüberganges einer überall konvergenten Folge dieser Funktionen abgeschlossen. Einfach gesagt, besteht diese Klasse aus den stetigen Funktionen und den Funktionen, die man aus ihnen durch wiederholte Grenzprozesse gewinnen kann.

Da binäre Schwellwerteinheiten eine lineare Funktion berechnen, können Netze aus binären Schwellwerteinheiten auch nur lineare Funktionen darstellen. Dies ist ein Grund, warum man häufig sigmoide Funktionen als Aktivierungsfunktionen benutzt. Dagegen geht es in Wissensbasierten Neuronalen Netzen meist darum, Ja– oder Nein–Entscheidungen zu treffen. Deshalb kommen dort binäre Schwellwerteinheiten zum Einsatz. Oft werden diese jedoch durch sigmoide Funktionen approximiert, weil für verschiedene Lernverfahren stetige Funktionen benötigt werden(siehe Abschnitt 3.2). Dann liegen die Werte der Aktivierung einer Einheit im Intervall $[0, 1]$ und das Ergebnis entspricht nicht explizit einem Wahrheitswert **wahr** oder **falsch** . Wir wollen deshalb vereinbaren, bei Verwendung von sigmoiden Funktionen Aktivierungen in einem Intervall $(A_{min}, 1]$ als nahe 1 und in einem Intervall $[0, 1 - A_{min}]$ als nahe 0 zu bezeichnen ($0,5 \leq A_{min} \leq 1$). Der Wert A_{min} beschreibt demnach, ab welcher Aktivierung einer Einheit der Wahrheitswert **wahr** zugeordnet wird. Er wird nach den Erfordernissen der Anwendung gewählt. Aktivierungen nahe 0 wird der Wahrheitswert **falsch** zugeordnet.

Rekurrente Netze sind Netze, in denen rekurrente Verbindungen erlaubt sind. Dies sind Verbindungen zu vorhergehenden Schichten oder Verbindungen zwischen Neuronen ein und derselben Schicht.

Definition 2.3.6 (n–schichtiges rekurrentes Netz [Kal94]) *Ein n–schichtiges rekurrentes Netz N besteht aus einem n–schichtigen vorwärtsgerichteten Netz mit derselben Anzahl von Einheiten in der Ein– und Ausgabeschicht. Jede Einheit* V_k^{n-1} *in der Ausgabeschicht wird mit der korrespondierenden Einheit* V_k^0 *in der Eingabeschicht durch eine mit dem Gewicht* 1 *bewertete Kante verbunden.*

Die Abbildung 2.2 zeigt den Aufbau eines 3–schichtigen vorwärtsgerichteten Netzes.

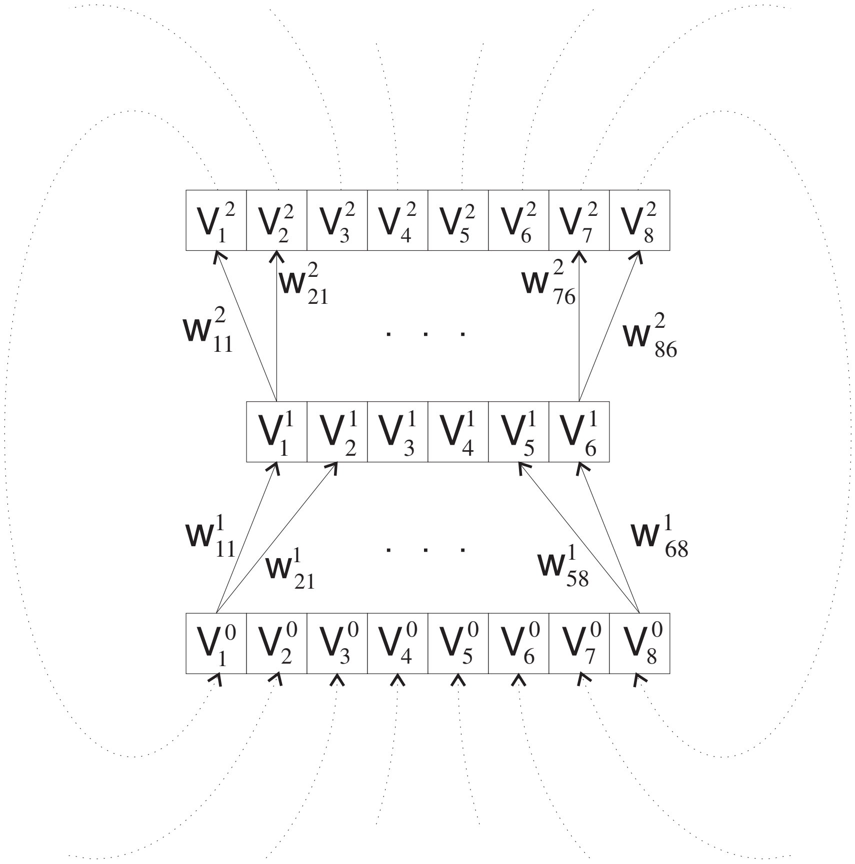


Abbildung 2.2: Beispiel eines 3–schichtigen vorwärtsgerichteten Netzes mit gleicher Anzahl von Ein– und Ausgabeeinheiten und vollvernetzten Schichten. Verbindet man jede Einheit V_k^2 in der Ausgabeschicht und jede Einheit V_k^0 in der Eingabeschicht mit einer mit 1 gewichteten Kante erhält man ein 3–schichtiges rekurrentes Netz.

3. Algorithmen

3.1 Berechnung von Tp mit Neuronalen Netzen

Für alle definiten Programme ist der Operator T_P eine stetige Funktion. Damit gehört sie zur Klasse der Borel–meßbaren Funktionen und kann durch ein 3–schichtiges Neuronales Netz hinreichend genau approximiert werden.

Für die Berechnung der Iterationen des Operators T_P mit einem mehrschichtigen rekurrenten Netz ist es notwendig, den Definitionsbereich des Operators T_P auf die Eingabeschicht und den Wertebereich auf die Ausgabeschicht eines vorwärtsgerichteteten Netzes zu kodieren. In beiden Fällen handelt es sich um Interpretationen I einer Herbrand–Basis B_P. Wie wir bereits gezeigt haben, ist die Herbrand–Basis B_P eines prädikatenlogischen Programmes meist eine abzählbar unendliche Menge. Eine äquivalente Interpretation I könnte demnach auch abzählbar unendlich viele Formeln enthalten, denen der Wahrheitswert wahr zugeordnet werden soll. Das Problem der bijektiven Abbildung einer unendlichen Menge von Formeln auf eine endliche Anzahl von Einheiten in einem Neuronalen Netz konnte bisher jedoch nicht zufriedenstellend gelöst werden. Für die Aussagenlogik gilt, daß alle Programme eine endliche Herbrand–Basis B_P und damit endliche Interpretationen I besitzen. Das oben beschriebene Problem tritt also hier nicht auf.

3.1.1 Berechnung mit binären Schwellwerteinheiten und W=1

Yvonne Kalinke [Kal94] hat eine Kodierung von Interpretationen I einer Herbrand–Basis B_P vorgestellt, zusammen mit einem Algorithmus, der für ein beliebiges aussagenlogisches Programm ein 3–schichtiges vorwärtsgerichtetes Netz konstruiert, das den Operator T_P für dieses Programm berechnet.

Das Netz wird aus binären Schwellwerteinheiten aufgebaut. Die Einheiten in der Ein– und Ausgabeschicht des Netzes repräsentieren jeweils ein Element der Herbrand–Basis B_P. Die Eingabeschicht steht für die Eingangsinterpretation I und die Ausgabeschicht für die Ausgangsinterpretation $T_P \uparrow 1(I)$, wobei jede Einheit mit einem Wahrheitswert **wahr** Element der Interpretation ist und alle anderen Einheiten nicht.

Die Idee des Algorithmus besteht darin, die logischen Verknüpfungen UND und ODER jeweils mit einer Schicht des Neuronalen Netzes zu berechnen. Die Konjunktion (UND) entscheidet, ob alle Literale L_i einer Programmklausel erfüllt sind, und die Disjunktion (ODER) berechnet, ob eine der Programmklauseln mit dem Atom A_i im Klauselkopf aktiv ist. Der Algorithmus erstellt demnach ein 3–schichtiges vorwärtsgerichtetes Netz aus binären Schwellwerteinheiten zur Berechnung von T_P. Die Verbindungen zwischen den Einheiten werden mit einem konstanten Betrag $W = 1$ gewichtet. Die Schwellwerte der Einheiten werden auf die Anforderungen des Programmes angepaßt.

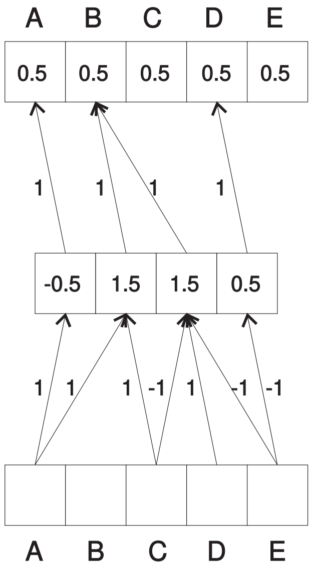


Abbildung 3.1: Konstruktion eines 3–schichtigen Netzes aus binären Schwellwerteinheiten zur Berechnung von T_P für das Programm $P = \{A; B \leftarrow A, C; B \leftarrow \neg C, D, \neg E; D \leftarrow \neg E\}$.

vereinfachter Algorithmus aus [Kal94]:
Gegeben sei ein aussagenlogisches Programm P, bestehend aus J Programmklauseln P_j mit $0 \leq j \leq J$ der Form $A_j \leftarrow L_{j1}, L_{j2}, \ldots, L_{jm_j}$ mit $m_j \geq 0$. Das Programm bildet eine endliche Herbrand–Basis $B_P = \{F_1, F_2, \ldots, F_N\}$.

1. Erstelle für jedes Element F_n $(0 \leq n \leq N)$ der Herbrand–Basis B_P jeweils eine Einheit V_n^0 in der Eingabe– und V_n^2 in der Ausgabeschicht des Netzes
2. Erstelle für jede Programmklausel P_j des Programmes P eine Einheit V_j^1 in der verdeckten Schicht des Netzes
3. Setze den Schwellwert aller Einheiten in der Ausgabeschicht auf $\theta_n^2 = \frac{W}{2} = 0.5$
4. Bearbeite alle Programmklauseln P_j, wie folgt

a) Ist P_j ein Fakt, dann verbinde die Einheit des Atoms A_j in der Eingabeschicht mit der Einheit der Programmklausel P_j in der verdeckten Schicht und wichte sie mit $W = 1$

b) Ist P_j kein Fakt, dann verbinde die Einheit jedes Literals L_{jl} $(1 \leq l \leq m_j)$ in der Eingabeschicht mit der Einheit der Programmklausel P_j in der internen Schicht und wichte sie mit $W = 1$, wenn das Literal positiv, sonst mit $-W = -1$

c) Setze den Schwellwert θ_j^1 in der internen Schicht auf einen Wert $(k-1)W < \theta_j^1 < kW$, wobei k die Anzahl der in V_j^1 endenden positiv gewichteten Verbindungen ist

d) Verbinde die Einheit V_j^1 in der internen Schicht mit der Einheit des Atoms A_j in der Ausgabeschicht und wichte diese Verbindung mit $W = 1$

Damit erhält man ein 3–schichtiges vorwärtsgerichtetes Netz, das einmal den Operator T_P auf die gegebene Interpretation anwendet. Verbindet man jede Einheit k in der Ausgabeschicht mit einer mit 1 gewichteten Kante zur Einheit k in der Eingabeschicht, so berechnet das entstandene rekurrente Netz den Fixpunkt des Operators T_P sofern dieser existiert.

3.1.2 Berechnung mit Sigmoider Aktivierungsfunktion und $\mathbf{W(A_{min})}$

Aktivierungsfunktion	h_1^m–h_4^m	θ_1^m–θ_4^m	V_1^m–V_4^m	h_1^{m+1}	θ_1^{m+1}	V_1^{m+1}
STEP–Funktion $g(h) = \begin{cases} 1 & : \text{falls} \quad h \geq \theta \\ 0 & : \text{sonst} \end{cases}$	$0,0$	$0,5$	$0,0$	$4 * 0,0 = 0,0$	$0,5$	$0,0$
Sigmoide Funktion $g(h) = \frac{1}{1+e^{-(h-\theta)}}$	$0,0$	$0,5$	$0,37$	$4 * 0,37 = 1,48$	$0,5$	$0,73$

Tabelle 3.1: Die Tabelle zeigt die Unterschiede bei Verwendung von verschiedenen Aktivierungsfunktionen. Eine Einheit, die mit binären Schwellwerteinheiten nicht aktiv ist, kann mit sigmoider Aktivierungsfunktion aktiviert sein.

Wie man im Abschnitt 3.2 noch sehen wird, ist es für das Lernen enorm wichtig, ein vorwärtsgerichtetes Netz mit sigmoider Aktivierungsfunktion in den Einheiten zu nutzen. Erstellt man jedoch auf Grundlage des Algorithmus aus [Kal94] ein Netz und verwendet sigmoide Aktivierungsfunktionen anstatt Schwellwertfunktionen, so kann dieses Netz eine völlig andere Funktion berechnen, als das topologisch gleiche Netz mit binären Schwellwerteinheiten. Die Aktivierungen der Einheiten liegen dann nämlich nicht mehr bei **0** für nicht aktivierte Neuronen und **1** für aktivierte Neuronen, sondern in den Intervallen $[0, 0.5]$ und $(0.5, 1]$. Angenommen, es gäbe ein Neuron in der Ausgabeschicht mit einem Schwellwert $\theta = 0.5$ und 4 mit 1 gewichteten Verbindungen, dann ist das Neuron im Netz mit binären Schwellwerteinheiten nicht aktiv, wenn alle 4 Vorgänger nicht aktiv sind. Im sigmoiden Netz jedoch könnte jeder der 4 nicht aktiven Vorgänger eine Aktivierung von 0.3 liefern. Die Summe der Aktivierungen würde den Schwellwert überschreiten und unser Neuron wäre plötzlich zu Unrecht aktiviert (siehe Tabelle 3.1). Genau dieser Effekt tritt bei einem mit dem Betrag $W = 1$ gewichteten Netz mit sigmoiden Aktivierungsfunktionen häufig ein, so daß unser Netz nicht garantiert, den Operator T_P zu berechnen.

Basierend auf [Kal94] geben Garcez, Zaverucha und Carvalho [GZC96] einen Algorithmus zur Erstellung eines 3–schichtigen vorwärtsgerichteten Netzes mit sigmoider Aktivierungsfunktion an. Die Autoren nennen ihn "Übersetzungsalgorithmus". Der Übersetzungsalgorithmus wichtet alle Verbindungen und Schwellwerte im Netz entsprechend einem speziellen Betrag $W = W(A_{min})$. Die Schwellwerte in der Ausgabeschicht erhalten den Wert $\frac{W(A_{min})}{2}$. Das Gewicht $W(A_{min})$ wird für jedes Programm P spezifisch berechnet, in Abhängigkeit von der Aktivierung A_{min}, die angibt, ab welcher Aktivierung einem Neuron der Wahrheitswert **wahr** zugeordnet werden soll.

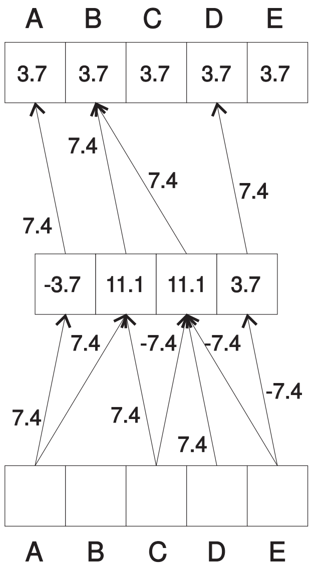


Abbildung 3.2: Konstruktion eines 3–schichtigen Netzes mit sigmoiden Aktivierungsfunktionen zur Berechnung von T_P für das Programm $P = \{A; B \leftarrow A, C; B \leftarrow \neg C, D, \neg E; D \leftarrow \neg E\}$. Das Gewicht $W(A_{min}) = 7.4$ mit $A_{min} = 0,9$ wurde nach dem Algorithmus von Garcez, Zaverucha und Carvalho bestimmt.

Der vereinfachte Algorithmus[1] zur Berechnung von $\mathbf{W(A_{min})}$:

1. Berechne die maximale Aktivierung $A_{max} = 1 - A_{min}$ mit der einem Neuron der

[1] Die Realisierung des Algorithmus, die Garcez, Zaverucha und Carvalho vorschlagen, ist wesentlich effizienter, aber zum besseren Verständis ist diese Darstellung günstiger.

Wahrheitswert **falsch** zugeordnet wird

2. Für jede Klausel P_j der Form $A_j \leftarrow L_1, \ldots, L_k$ im Programm P:
   a) Ermittle die Anzahl p der positiven Literale
   b) Ermittle die Differenz $d = 2p - k$ der Anzahl der positiven und negativen Literale
   c) Wenn $dA_{max} < 0.5$ dann berechne $W_j = \frac{\ln(A_{max}) - \ln(A_{min})}{dA_{max} - 0.5}$, sonst berechne $W_j = \frac{\ln(A_{min}) - \ln(A_{max})}{A_{min} - 0.5}$
3. Berechne $W_{min} = \max(W_j)$ für alle j
4. Für jedes Atom A_i welches im Kopf von mindestens einer Klausel steht, berechne $W_i = \frac{\ln(A_{max}) - \ln(A_{min})}{n_i A_{max} - 0.5}$, wobei n_i die Anzahl der Klauseln ist, in denen A_i im Kopf steht
5. Berechne $W_{min} = \max(W_{min}, W_i)$ für alle i
6. Liefere Ergebnis $W = W_{min}$

Die Autoren geben in ihrer Arbeit einen Beweis an, daß für jedes normale aussagenlogische Programm ein 3–schichtiges vorwärtsgerichtetes Netz mit halb–linearen[2] Aktivierungsfunktionen existiert, das den Operator T_P für das Programm berechnet, nämlich genau dann, wenn das Netz mit einem Gewicht $W \geq W(A_{min})$ nach dem beschriebenen Übersetzungsalgorithmus erstellt wird. Im nächsten Abschnitt werden wir zeigen, daß für ein konstantes A_{min} sehr wohl Programme existieren, für die dies nicht zutrifft.

3.1.3 Ein Gegenbeweis

Beispiel 3.1.1 *Sei das folgende aussagenlogische Programm P gegeben:*

$$\begin{array}{lcl} A & \leftarrow & B, \neg C \\ A & \leftarrow & C, \neg D \\ A & \leftarrow & D, \neg E \\ A & \leftarrow & E, \neg F \\ A & \leftarrow & F, \neg G \\ & \ldots & \\ A & \leftarrow & Y, \neg Z \end{array}$$

Die Herbrand–Basis B_P besteht aus den Buchstaben des Alphabets. Jede Programmklausel P_j hat die Form $A \leftarrow L_{j1}, L_{j2}$, wobei jedes Literal L_{j1} ein Atom ist und als Bezeichner

[2] die Sigmoide Funktion ist halb–linear

den $(j+1)$–ten Buchstaben aus dem Alphabet besitzt. Das Literal L_{j2} jeder Programmklausel ist ein negiertes Atom und trägt den Namen des $(j+2)$–ten Buchstaben aus dem Alphabet.

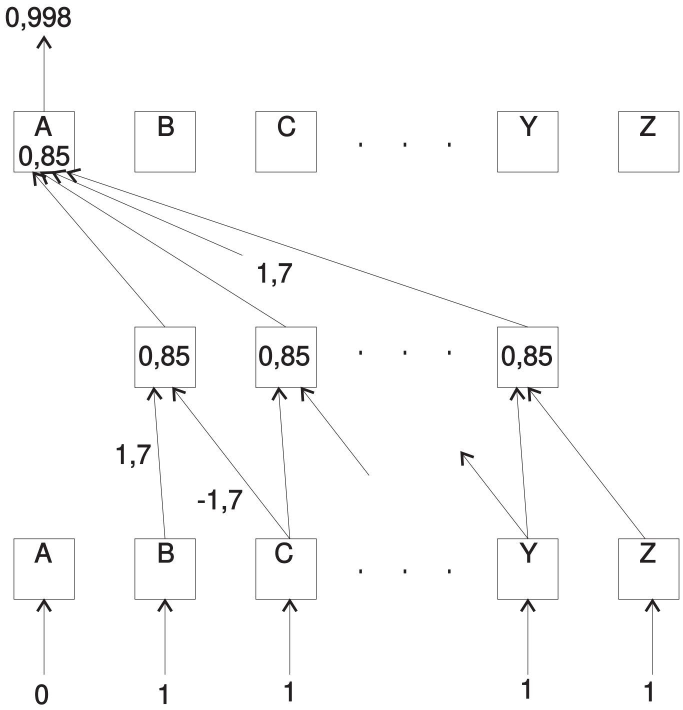


Abbildung 3.3: Abbildung eines Netzes mit Sigmoid–Funktion, das den Operator T_P nicht korrekt berechnet. Alle Verbindungen sind mit dem Betrag $W(A_{min})$ gewichtet. Die Schwellwerte in den Einheiten der verdeckten und Ausgabeschicht haben den Wert $\frac{W(A_{min})}{2} = 0,85$. Bei Anlegen der Interpretation $I = \{B, C, D, \ldots, Z\}$, haben alle verdeckten Einheiten die Aktivierung $0,3$. Ihnen wird der Wahrheitswert **falsch** zugeordnet. Die Ausgabeeinheit realisiert die Disjunktion der verdeckten Einheiten. Sie erhält eine Netzeingabe $h = 24 * 0,3 = 7,2$. Der Schwellwert von A ist überschritten und der Aktivierung $0,998$ wird der Wahrheitswert **wahr** zugeordnet.

Zur Berechnung von T_P mit dem Übersetzungsalgorithmus wird ein 3–schichtiges vorwärtsgerichtetes Netz aus Einheiten mit sigmoider Aktivierungsfunktion erstellt. Es besteht aus 26 Eingabe– und Ausgabeeinheiten und 24 Einheiten in der verdeckten Schicht. Die minimale Aktivierung sei $A_{min} = 0,7$. Dann beträgt das Gewicht für den Übersetzungsalgorithmus $W(A_{min}) = 1,7$.
Die Herbrand–Interpretation $I = \{B, C, D, E, F, G, \ldots, Y, Z\}$, ordnet allen Elementen

der Herbrand–Basis, außer A, den Wahrheitswert **wahr** zu. Die Berechnung von T_P ergibt $T_P \uparrow 1(I) = \emptyset$. Die Berechnung von T_P mit dem vom Übersetzungsalgorithmus erstellten Netz ergibt $T_P^{Netz} \uparrow 1(I) = \{A\}$. Abbildung 3.3 zeigt das erstellte Netz und die falsche Berechnung für einen Eingabevektor.

Theorem:
Sei der Übersetzungsalgorithmus zur Erstellung eines 3–schichtigen vorwärtsgerichteten Neuronalen Netzes mit sigmoider Aktivierungsfunktion gegeben. Für jeden Wert A_{min} mit $0,5 < A_{min} \leq 1$ existiert ein normales logisches Programm P, für das es kein Gewicht $W = W(A_{min})$ gibt, mit dem das nach dem Übersetzungsalgorithmus erstellte Netz den Operator T_P berechnet.

Beweis:
Die Einheiten in der Ausgabeschicht des Netzes sollen die ODER–Funktion realisieren. Sei n_i die Anzahl der Programmklauseln, in denen das Atom A_i im Kopf vorkommt. Dann hat nach dem Übersetzungsalgorithmus die zu A_i korrespondierende Einheit in der Ausgabeschicht genau n_i mit W gewichtete Verbindungen von Einheiten der verdeckten Schicht. Diese Ausgabeeinheit soll aktiv sein, wenn mindestens eine der n_i verdeckten Einheiten aktiv ist. Daraus folgt für die Aktivierung:

$$\frac{1}{1 + e^{-(WA_{min} - \frac{W}{2})}} \geq A_{min} \tag{3.1}$$

Durch Umstellen der Gleichung nach W und unter Beachtung von $A_{max} = 1 - A_{min}$ ergibt sich

$$W \geq \frac{\ln(A_{min}) - \ln(A_{max})}{A_{min} - \frac{1}{2}} \tag{3.2}$$

Für eine gegebene minimale Aktivierung A_{min} ist dieser Wert für alle Programme P konstant und bildet eine untere Grenze für W. Für die Berechnung der Disjunktion muß außerdem gelten, daß die Ausgabe den Wert **falsch** hat, wenn alle n_i verdeckten Einheiten inaktiv sind. Einer Aktivierung wird der Wahrheitswert **falsch** zugeordnet, wenn ihr Wert kleiner als A_{max}. Damit folgt für diesen Fall

$$\frac{1}{1 + e^{-(n_i W A_{max} - \frac{W}{2})}} \leq A_{max} \tag{3.3}$$

Durch schrittweises Umstellen und Vereinfachen ergeben sich die äquivalenten Gleichungen 3.4 - 3.7:

$$\frac{1 - A_{max}}{A_{max}} \leq e^{-W(n_i A_{max} - \frac{1}{2})} \tag{3.4}$$

$$\ln(\frac{A_{min}}{A_{max}}) \leq -W(n_i A_{max} - \frac{1}{2}) \tag{3.5}$$

$$\ln(A_{min}) - \ln(A_{max}) \leq -W(n_i A_{max} - \frac{1}{2}) \tag{3.6}$$

$$\ln(A_{max}) - \ln(A_{min}) \geq W(n_i A_{max} - \frac{1}{2}) \tag{3.7}$$

In Abhängigkeit vom Vorzeichen der Differenz $(n_i A_{max} - \frac{1}{2})$ gibt es zwei völlig verschiedene Lösungen für W:

$$W \leq \frac{\ln(A_{max}) - \ln(A_{min})}{n_i A_{max} - \frac{1}{2}} \text{ für } (n_i A_{max} - \frac{1}{2}) > 0 \tag{3.8}$$

$$W \geq \frac{\ln(A_{max}) - \ln(A_{min})}{n_i A_{max} - \frac{1}{2}} \text{ für } (n_i A_{max} - \frac{1}{2}) < 0 \tag{3.9}$$

Für ein logisches Programm, in dem ein solches Atom A_i enthalten ist, daß für die Zahl n_i gilt

$$(n_i A_{max} - \frac{1}{2}) > 0 \tag{3.10}$$

muß das Gewicht W zur Berechnung von T_P den Ungleichungen 3.8 und 3.2 genügen. Unter Beachtung von $A_{max} = 1 - A_{min}$ und $0,5 < A_{min} \leq 1$ folgt aus 3.8, daß $W < 0$ und aus 3.2, daß $W > 0$. Damit existiert für jedes Programm P, in dem ein beliebiges Atom A_i enthalten ist, das 3.10 erfüllt, kein W, mit dem das aus dem Übersetzungsalgorithmus entstandene Netz den den Operator T_P berechnet. Da n_i die Anzahl der Programmklauseln bezeichnet, in denen A_i im Klauselkopf enthalten ist, kann man für jedes konstante A_{min} bzw. A_{max} ein Programm finden, das die Ungleichung 3.10 erfüllt, indem man eine beliebige Programmklausel der Form $A_i \leftarrow L_1, \ldots, L_k$ $(k \geq 1)$ mindestens m mal in ein Programm aufnimmt mit $m > \frac{1}{2A_{max}}$. □

Damit ist bewiesen, daß der Übersetzungsalgorithmus nicht garantiert, für jedes normale logische Programm den Operator T_P zu berechnen. Bei vielen logischen Programmen treten die Elemente der Herbrand–Basis lediglich im Klauselkopf weniger Programmklauseln auf. Deshalb sollte man den Wert A_{min} groß wählen, damit das Netz aus dem Übersetzungalgorithmus T_P für eine große Klasse von Programmen korrekt berechnet. Bei der Implementierung des Algorithmus zur Berechnung von $W(A_{min})$ sollte man darauf achten, daß $A_{min} \geq 0$ und $n_i A_{max} \neq 0,5$, da sonst eine Division durch 0 auftritt.

3.2 Induktives Lernen in vorwärtsgerichteten Neuronalen Netzen

Das Wissen eines Neuronalen Netzes wird in erster Linie bestimmt durch die Gewichte zwischen den Neuronen. Beim Lernen ist es deshalb notwendig, diese Gewichte in der gewünschten Form zu verändern. Dies geschieht durch das Lernen von Beispielen. Dem Netz werden Eingabevektoren ξ^μ und die dazugehörigen Ausgabevektoren ζ^μ $(\mu \geq 1)$ präsentiert und das Netz versucht im Laufe des Lernvorganges, diese Muster zu approximieren. Meist kann man dem Netz nicht für jede Belegung der Eingabeschicht die zu lernende Ausgabe übergeben. Entweder es existieren ganz einfach zu viele Muster, oder die Ergebnisse für einige Belegungen sind unwichtig, oder überhaupt noch nicht bekannt. In diesen Fällen wird eine wichtige Eigenschaft Neuronaler Netze ausgenutzt. Neuronale Netze, die mit einer Menge von Beispielen trainiert wurden, liefern meist auch für Eingabemuster, die nicht in der Trainingsmenge enthalten waren, ein brauchbares oder sogar richtiges Ergebnis. Diese Eigenschaft nennt man Generalisierungsfähigkeit und bildet neben der parallelen Verarbeitung einen Hauptvorteil im Lernen mit Neuronalen Netzen. Die Generalisierungsfähigkeit eines Netzes ist vorwiegend von seiner Größe abhängig. Hat ein Netz zu wenig Neuronen, dann kann die gewünschte Funktion möglicherweise nicht hinreichend genau approximiert werden. Ist es aus zu vielen Schichten und Einheiten aufgebaut, besteht die Gefahr, daß es die Trainingsmenge auswendig lernt und die Generalisierungsfähigkeit verloren geht.

Definition 3.2.1 (Beispiel) *Das Lernen einer Funktion aus Beispielen dieser Funktion nennt man induktives Lernen. Ein Beispiel μ repräsentiert dabei immer ein Paar von Eingabevektor $\xi^\mu = (\xi_1^\mu, \ldots, \xi_{k_e}^\mu)$ und Ausgabevektor $\zeta^\mu = (\zeta_1^\mu, \ldots, \zeta_{k_a}^\mu)$.*

Im Neuronalen Netz werden Beispiele auch als Muster bezeichnet. Der Index μ bezeichnet das μ–te Beispiel in der Trainingsmenge und k_e und k_a entsprechen der Anzahl der Eingabe– bzw Ausgabeneuronen des Netzes.

Betrachtet man in einem vorwärtsgerichteten Netz jede Ausgabeeinheit k seperat, so berechnet das Netz für jede dieser k_a Ausgabeeinheiten die Funktion $f_k : R^{k_e} \to R$. Wird der Aktivierung einer Ausgabeeinheit bei der Präsentation eines Eingabevektors ξ^μ ein Wahrheitswert zugeordnet, so wird dieser Eingabevektor entsprechend dem Wahrheitswert **wahr** , positiv oder **falsch** , negativ von dieser Ausgabeeinheit klassifiziert. Einen Lernvorgang, bei dem die Menge aller als positiv klassifizierten Eingabevektoren zunimmt, nennt man Generalisierung. Einen Lernvorgang, bei dem die Menge aller als negativ klassifizierten Eingabevektoren zunimmt, nennt man Spezialisierung.

Definition 3.2.2 (generalisierendes Beispiel) *Sei ein Beispiel μ mit einem binären Ausgabevektor $\zeta^\mu : \{\ \mathbf{0}\ ,\ \mathbf{1}\ \}^{k_a})$ gegeben. Sei weiterhin die Ausgabe des Netzes $O^\mu : R^{k_a}$ gegeben. Das Beispiel μ wird als generalisierendes Beispiel bezeichnet, wenn für alle O_i^μ, denen der Wahrheitswert* **wahr** *zugeordnet wird, auch $\zeta_i^\mu =$* **wahr** *und mindestens ein O_j^μ existiert, dem der Wahrheitswert* **falsch** *zugeordnet wird, und bei dem $\zeta_j^\mu =$* **wahr** *.*

Definition 3.2.3 (spezialisierendes Beispiel) *Sei ein Beispiel μ mit einem binären Ausgabevektor $\zeta^\mu : \{$ **0** , **1** $\}^{k_a})$ gegeben. Sei weiterhin die Ausgabe des Netzes $O^\mu : R^{k_a}$ gegeben. Das Beispiel μ wird als spezialisierendes Beispiel bezeichnet, wenn für alle O_i^μ, denen der Wahrheitswert* **falsch** *zugeordnet wird, auch $\zeta_i^\mu =$* **falsch** *und mindestens ein O_j^μ existiert, dem der Wahrheitswert* **wahr** *zugeordnet wird, und bei dem $\zeta_j^\mu =$* **falsch** *.*

Die Unterscheidung von spezialisierenden und generalisierenden Beispielen spielt in den Untersuchungen zur Konsistenz von aussagenlogischen Programmklauseln während des Lernens eine Rolle (Abschnitt 5).

3.2.1 Der bekannteste Lernalgorithmus – Backpropagation

Backpropagation ist ein Gradientenabstiegsverfahren. Dies hat seine Ursache darin, wie die Gewichte beim Lernen verändert werden. Neben den Gewichten zwischen den Einheiten werden auch die Schwellwerte der Neuronen verändert. Sie werden als Verbindungen betrachtet, die mit dem Betrag des Schwellwertes gewichtet sind und an denen immer eine Aktivierung -1 anliegt[3].

Für die Formalisierung des Verfahrens wurde erneut die Terminologie und Notation von Hertz, Krogh und Palmer [HKP91] verwendet.

Das Lernverfahren besteht aus 3 wesentlichen Schritten:

1. Anlegen des Eingabevektors ξ^μ an die Eingabeschicht und Vorwärtspropagieren der Aktivierungen durch alle Schichten
2. Berechnung der Fehler δ_i^M in der Ausgabeschicht des Netzes und Rückwärtspropagieren der Fehler bis zur ersten verdeckten Schicht
3. Berechnung der notwendigen Gewichtsänderungen Δw_{ij}^m und Verändern der Gewichte im Netz

Diese Schritte werden nacheinander für jedes zu lernende Beispiel durchgeführt. Die beiden Propagierungsschritte werden in Abbildung 3.2 dargestellt. Einen Lernvorgang, in dem alle Muster genau einmal trainiert wurden, nennt man eine Epoche.

Die Güte einer gelernten Trainingsmenge wird bestimmt durch die *globale Fehlerfunktion*

$$E[\mathbf{w}] = \frac{1}{2} \sum_{\mu i} [\zeta_i^\mu - O_i^\mu]^2$$

Anschaulich gesprochen gibt dieser Fehler für die aktuelle Wichtung $\mathbf{w}$ im Netz die Summe aller quadratischen Unterschiede zwischen realer Ausgabe $O^\mu = V^{M\mu}$ und geforderter Ausgabe ζ^μ an. Damit gilt für alle möglichen Zustände des Netzes $E[\mathbf{w}] \geq 0$.

[3] alternativ: Gewicht $= -\theta$, Aktivierung $= +1$

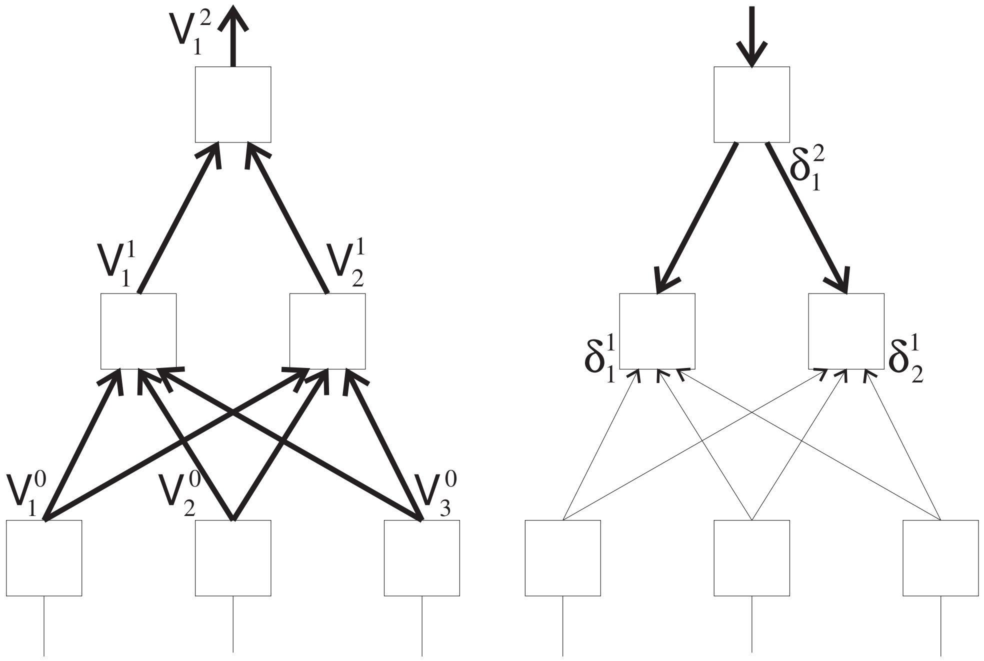


Tabelle 3.2: Die 2 Propagierungsphasen. Die Aktivierungen V_i^m werden von der Eingabeschicht bis zur Ausgabeschicht vorwärtspropagiert. Die lokalen Fehler δ_i^m werden von der Ausgabeschicht bis zur ersten verdeckten Schicht zurückpropagiert.

Das Ziel des Lernens ist es demzufolge, den globalen Fehler zu minimieren. Die Fehlerfunktion bildet im Raum der Gewichte eine mehrdimensionale, gebirgsähnliche Kurvenlandschaft mit Tälern und Bergen. Bei Netzen, die aufgrund ihrer Topologie in der Lage sind, die gewünschte Funktion hinreichend genau zu approximieren, ist das globale Minimum der Fehlerfunktion im Raum der Gewichtsvektoren $E[\mathbf{w}] = 0$. Deshalb müssen die Gewichte in der Richtung zum globalen Minimum hin geändert werden. Um dies zu erreichen, verändert man die Gewichte proportional zum Gradienten[4] der Gewichte über der Fehlerfunktion. Für die Ausgabeschicht M des Netzes ergibt dies

$$\begin{aligned} \Delta w_{ij}^M = -\eta \frac{\partial E}{\partial w_{ij}^M} &= \eta \sum_\mu [\zeta_i^\mu - O_i^\mu] g'(h_i^{M\mu}) V_j^{(M-1)\mu} \\ &= \eta \sum_\mu \delta_i^{M\mu} V_j^{(M-1)\mu} \end{aligned}$$

mit dem lokalen Fehler

$$\delta_i^{M\mu} = g'(h_i^{M\mu})[\zeta_i^\mu - O_i^\mu]$$

[4] die Ableitung

des Neurons i bei Präsentation des Beispiels μ. Durch die Berechnung dieses Gradienten wird garantiert, daß sich die Gewichtsänderung immer in der Richtung zu einem Minimum in der Fehlerfläche hin bewegt.

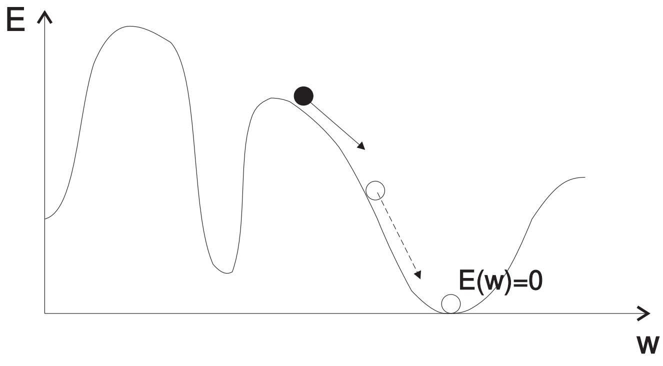


Abbildung 3.4: Prinzip des Gradientenabstiegs. Der schwarze Kreis kennzeichnet den aktuellen Zustand des Netzes, der Pfeil den Gradienten $\frac{\partial E}{\partial \mathbf{w}}$. Die Gewichte werden proportional zum Gradienten geändert und gelangen in das nächstliegende Minimum der Fehlerkurve. Zur Vereinfachung wurde eine zweidimensionale Darstellung in Abhängigkeit von der Menge aller Gewichte $\mathbf{w}$ gewählt. In Wirklichkeit ist die Fehleroberfläche meist mehrdimensional.

Wir wollen nun den kompletten Ablauf von Backpropagation für ein Muster nachvollziehen.

1. Aus der Menge der Trainingsmuster wird ein Beispiel μ ausgewählt und die Eingabeschicht mit dem Eingabevektor ξ^μ belegt, so daß
$$V_k^0 = \xi_k^\mu \quad \text{für alle} \quad k$$
Danach wird das angelegte Muster vorwärtspropagiert durch alle Schichten m des Netzes mit
$$V_i^m = g(h_i^m) = g(\sum_j w_{ij}^m V_j^{m-1})$$
für alle i und damit alle Ausgaben V_i^M des Netzes berechnet.

2. Der Ausgabevektor ζ^μ des Beispiels μ wird mit der Ausgabe des Netzes verglichen und die Fehler
$$\delta_i^M = g'(h_i^M)[\zeta_i^\mu - V_i^M]$$
in der Ausgabeschicht berechnet.
Diese Fehler werden durch alle versteckten Schichten zurückpropagiert mit
$$\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ji}^m \delta_j^m$$

3. Nun werden für alle Einheiten aus ihren persönlichen Fehlern die notwendigen Gewichtsänderungen
$$\Delta w_{ij}^{m} = \eta \delta_i^m V_j^{m-1}$$
ermittelt und anschließend durch
$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij}$$
neu gesetzt.

Der Parameter η beim Backpropagation–Algorithmus bezeichnet die Lernrate. Sie gibt an, wie stark die Gewichtsänderung dem Gradienten folgt. Die Lernrate sollte nicht zu groß gewählt werden. Häufig werden Werte zwischen 0.1 und 1 verwendet.

Bei der Anwendung von Backpropagation ist zu beachten, daß die verwendete Aktivierungsfunktion g stetig und damit differenzierbar ist, da sonst nicht gewährleistet ist, daß eine Ableitung g' existiert.

Bei mehrschichtigen Netzen ist die Fehlerfläche wesentlich komplexer als zum Beispiel bei Perzeptrons. Häufig tritt beim Lernen einer Funktion mit Backpropagation deshalb der Fall ein, daß das Netz in ein lokales Minimum gerät oder bei einer großen Lernrate zwischen den Wänden eines Minimums oszilliert. Es existieren verschiedene Verbesserungen, die das Lernen mit Backpropagation verbessern sollen.

- Backpropagation mit Momentum–Term
  Beim Trainieren wird eine Art Durchschnitt zwischen letzter und aktueller Gewichtsänderung vorgenommen. Damit wird ein wildes Oszillieren in der Fehleroberfläche verhindert und der Algorithmus kommt schneller und robuster zum Ziel. Der Momentum–Term bestimmt, in welchem Verhältnis der Durchschnitt gebildet wird.

- Backpropagation mit flat–spot–elimination[5]
  In Bereichen, in denen der Anstieg der Fehlerfunktion fast 0 ist, ist auch die Gewichtsänderung sehr klein. Damit kommt das Netz dem Minimum nur sehr langsam näher und zum Lernen werden viele Epochen benötigt. Das wird verhindert, indem man zum Gradienten eine Konstante hinzuaddiert.

- Backpropagation mit Weight-Decay[6]
  Mit jedem Trainingsdurchgang wird vom Betrag der Gewichte ein bestimmter Prozentsatz subtrahiert. Damit verfällt die Stärke der Gewichte mit fortschreitender Dauer, es sei denn, die Gewichte werden durch das Trainieren geändert. Gewichte, deren Betrag kaum geändert wird, verfallen mehr und mehr, und können gelöscht werden, wenn ihr Wert fast 0 ist. Dadurch werden unnötige Verbindungen im Netz beseitigt.

[5] Überspringen flacher Gebiete
[6] Gewichtszerfall

3.3 Regelextraktion in Neuronalen Netzen

Wie bereits gezeigt wurde, sind bereits kleine Neuronale Netze in der Lage, relativ komplexe Funktionen zu approximieren. Die Bedeutung einzelner Neuronen und ihre Beziehungen untereinander sind dabei relativ schwer nachzuvollziehen. Deshalb versucht man bei der Extraktion von Regeln aus einem Neuronalen Netz, die Funktion, die das Netz realisiert, durch einfache, überschaubare Regeln darzustellen. Dafür existieren zwei verschiedene Prinzipien. Zerlegende Methoden betrachten das Neuronale Netz als White–Box (durchsichtiger Kasten) und bestimmen die Regeln aus der inneren Struktur des Netzes. Pädagogische Ansätze dagegen betrachten lediglich die Ausgabe des Netzes in Abhängigkeit von bestimmten Eingabemustern. Sie kümmern sich nicht um den internen Aufbau des Netzes. Sie behandeln das Netz wie einen undurchsichtigen Kasten, eine Black–Box. Der einfachste und sicherste Weg, Regeln zu ermitteln, ist, dem Netz alle möglichen binären Eingabevektoren zu präsentieren. Damit berechnet man die Ausgabevektoren und für jedes Neuron in der Ausgabeschicht, das mit dem angelegten Eingangsmuster eine Aktivierung nahe 1 erhält, erstellt man eine Regel der Form $A_j \leftarrow L_1, L_2, \ldots, L_n$. Dabei bezeichnet A_j das aktivierte Neuron in der Ausgabeschicht des neuronalen Netzes. Die L_i sind Literale, die die Neuronen der Eingabeschicht repräsentieren und positiv sind, wenn die Eingabe 1 ist, andernfalls negativ. Dies Methode garantiert, daß das entstehende logische Programm dieselbe Funktion berechnet, wie das Netz. Aber bei großen Netzen ist sie nicht realisierbar. Bei einem Netz mit n Eingabeeinheiten gibt es 2^n verschiedene Eingabebelegungen. Die Anzahl der zu testenden Muster (und damit der Rechenaufwand) wächst also exponentiell mit der Anzahl der Eingabeeinheiten des Neuronalen Netzes. Dies ist der entscheidende Nachteil. Für die Regelextraktion gibt es deshalb verschiedene andere Lösungsansätze. Wir wollen uns mit drei dieser Ansätze näher beschäftigen. Die beiden ersten Ansätze beruhen auf dem zerlegenden Prinzip. Sie erstellen Regeln für jede Einheit in der Ausgabeschicht und in den verdeckten Schichten und gehen dabei von zwei Grundannahmen aus:

- In trainierten Neuronalen Netzen sind die Einheiten entweder maximal aktiv (sehr nahe 1) oder maximal inaktiv (sehr nahe 0)
- Das Training verändert die Bedeutung von Einheiten nicht signifikant

Die 3. Methode benutzt eine Kombination aus Beschneidung und pädagogischer Regelextraktion. Einen Vergleich verschiedener Methoden und ihrer Möglichkeiten hinsichtlich der Erstellung logischer Regeln ist in [GZC96] und [TS92] zu finden.

3.3.1 SUBSET – Algorithmen

SUBSET–Algorithmen spielen in der Literatur eine große Rolle. Sie basieren darauf, für alle verdeckten Einheiten und Ausgabeeinheiten eines neuronalen Netzes Teilmengen[7]

[7] Subset = Teilmenge

von Verbindungen zu ermitteln, die garantieren, daß der Schwellwert des Neurons überschritten wird, wenn sie aktiviert sind. Um alle Teilmengen, die diese Bedingung erfüllen, zu finden, ist es notwendig, alle möglichen Teilmengen zu erzeugen. Dies geschieht mit einfachen Suchalgorithmen (Breitensuche, Tiefensuche), die meist noch durch Heuristiken ergänzt werden, um den Suchraum einzuschränken. Aus den gefundenen Teilmengen können logische Regeln in Form von Programmklauseln aufgestellt werden. Danach werden alle subsumierten Regeln entfernt, so daß nur ein minimales Modell für die Aktivierung des Neurons übrigbleibt. Tabelle 3.3 stellt den Algorithmus für das Beispiel aus Abbildung 3.5 dar.

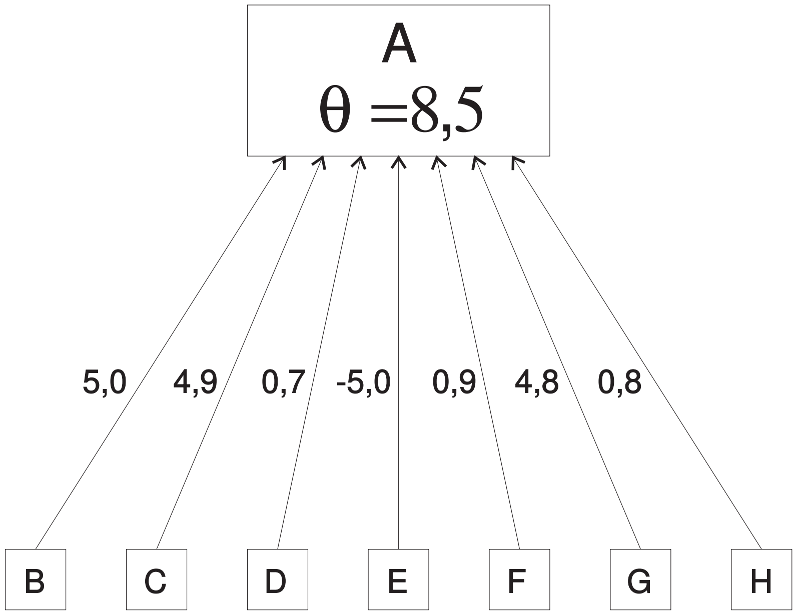


Abbildung 3.5: Beispiel eines Neurons A, für das Regeln extrahiert werden sollen. Die Aktivierungen der Vorgängereinheiten B, C, D, E, F und H sind entweder maximal aktiv oder maximal inaktiv. Deshalb kann angenommen werden, daß die Aktivierung, die das Neuron A über eine Verbindung erhält, gleich dem Gewicht ist.

Der Begriff der Subsumtionsrelation kommt aus der Theorie Terminologischer Systeme und soll hier kurz beschrieben werden.

Definition 3.3.1 (Subsumtion [BHS93]) *Mit Hilfe des logischen Folgerungsbegriffs läßt sich die von einer* Terminologie T *bestimmte Subsumtionsrelation* $\preceq_T$ *als*

$$KB_1 \preceq_T KB_2 \quad gdw. \quad T \models KB_1 \subseteq KB_2$$

definieren.

Definition 3.3.2 (Terminologie [BHS93]) *Eine Terminologie ist eine Konjunktion von* terminologischen Axiomen*, wobei jedes Konzept höchstens einmal auf der linken Seite eines der terminologische Axiome vorkommt.*

Definition 3.3.3 (terminologisches Axiom [BHS93]) *Formal definiert ist jedes terminologische Axiom entweder von der Form $K \subseteq KB$ oder von der Form $K = KB$, wobei K ein Konzept und KB eine Konzeptbeschreibung ist.*

Angewandt auf unser Problem bedeutet dies, daß eine Regel A eine andere Regel B subsumiert, wenn alle Modelle für die Regel B gleichzeitig Modelle der Regel A sind. Dies ist wiederum genau dann der Fall, wenn die ermittelte Teilmenge an Verbindungen Subset$_A$ eine Teilmenge der Menge Subset$_B$ an Verbindungen für die Regel B ist.

Ermitteln der Teilmengen	*Entfernen subsumierter Regeln*
$A \leftarrow B, C, G$ $A \leftarrow B, C, G, D$ $A \leftarrow B, C, G, D, E$ $A \leftarrow B, C, G, D, E, F$ $A \leftarrow B, C, G, D, E, F, H$ $A \leftarrow B, C, G, E$ $\ldots$	$A \leftarrow B, C, G$
$A \leftarrow B, C, \neg E$ $\ldots$	$A \leftarrow B, C, \neg E$
$A \leftarrow B, G, \neg E$ $\ldots$	$A \leftarrow B, G, \neg E$
$A \leftarrow C, G, \neg E$ $\ldots$	$A \leftarrow C, G, \neg E$

Tabelle 3.3: Extrahieren von Regeln für das Neuron aus Abbildung 3.5 mit einem SUBSET–Algorithmus. Zuerst werden alle Teilmengen ermittelt, deren Aktivierung den Schwellwert übersteigt. Diese Teilmengen werden als logische Klauseln aufgeschrieben. Anschließend werden subsumierte Regeln entfernt.

Die Kosten zum Finden aller Teilmengen steigen proportional zur Potenzmenge der Verbindungen jeder Einheit. Damit haben SUBSET–Algorithmen eine exponentielle Zeit– und Platzkomplexität. Dies ist ihr entscheidender Nachteil. Heuristiken können den Suchraum zwar beträchtlich einschränken, haben aber keinen Einfluß auf die Komplexität und vernachlässigen Teilmengen, die den Schwellwert einer Einheit übersteigen. SUBSET–Algorithmen sind deshalb für große neuronale Netze mit vielen Verbindungen zwischen den Einheiten nur begrenzt einsetzbar.

Die Originalarbeiten über SUBSET–Algorithmen stammen von FU[Fu91].

Towell und Shavlik [TS92] beschreiben einen vereinfachten SUBSET–Algorithmus. Dabei ermittelt man aus den positiv gewichteten Verbindungen die Teilmengen, die den Schwellwert des Neurons übersteigen. Danach werden für jede dieser Teilmengen aus dem Übertrag zwischen der Summe der positiven Gewichte und dem Schwellwert die minimale Teilmenge der negativ gewichteten Verbindungen bestimmt, die den Betrag dieser Differenz überschreiten. Schließlich werden die ermittelten Teilmengen in logische Klauseln gefaßt. Um die Komplexität des Algorithmus zu beschränken, legt man obere Grenzen für die Anzahl der Teilmengen fest.

Für jede versteckte und jede Ausgabeeinheit:

1. Extrahiere bis zu β_p Teilmengen einlaufender, positiv gewichteter Verbindungen, deren Summe größer ist, als der Schwellwert

2. Für jede Teilmenge $\mathcal{P}$ von den β_p gefundenen Teilmengen:

   a) Extrahiere bis zu β_n minimale Teilmengen von negativ gewichteten Verbindungen, deren Summe größer ist, als die Summe von $\mathcal{P}$ weniger dem Schwellwert der Einheit

   b) Sei $\mathcal{Z}$ ein neues Prädikat

   c) Für jede gefundene Teilmenge $\mathcal{N}$ erstelle eine Regel:
   "wenn $\mathcal{N}$ dann $\mathcal{Z}$ "

   d) Erstelle eine Regel:
   "wenn $\mathcal{P}$ und nicht $\mathcal{Z}$ dann <Name der Einheit>"

3.3.2 Der MOFN – Algorithmus

Wegen der Nachteile der SUBSET–Algorithmen entwickelten Towell und Shavlik den MOFN–Algorithmus. Dabei nutzen sie aus, daß Neuronale Netze besonders gut geeignet sind, Konzepte vom Typ "sind M Einheiten aus einer Gruppe mit N Einheiten aktiv, dann ist deren Nachfolger aktiv" zu lernen. Diese Konzepte nennen wir M von N bzw. M of N. Darauf basiert die Idee, einfache Regeln der Form M von N aus einem Neuronalen Netz zu extrahieren. Diese Form der Regeln wird für jede verdeckte Einheit und jede Ausgabeeinheit des Neuronalen Netzes aufgestellt.

Der MOFN–Algorithmus besteht aus 6 Schritten:

1. Clustern
   Die reell gewichteten Verbindungen zu einem Neuron werden geclustert, indem ähnliche Gewichte zusammengefaßt werden. Dabei wird ein hierarchischer Clusteralgorithmus benutzt, der auf Sammlungen[8] basiert. Am Beginn des Algorithmus besitzt jede Verbindung ihr eigenes Cluster. Danach werden die Cluster mit dem geringsten Abstand zusammengefügt. Dieser Prozess wird fortgesetzt, bis kein Paar

[8] engl.: JOINS

von Clustern mehr existiert, deren Abstand kleiner als eine fest definierte Grenze ist. Towell und Shavlik empfehlen in ihrer Arbeit einen maximalen Abstand von 0.25.

2. Durchschnittswerte bilden
   Die Gewichte der Verbindungen jedes Clusters werden addiert und deren Durchschnitt gebildet. Anschließend wird jede Verbindung dieses Clusters mit diesem Durchschnittswert gewichtet. Damit haben alle Gewichte eines Clusters denselben Wert.

3. Eliminieren
   Beim Eliminieren sollen alle Cluster und deren Verbindungen gelöscht werden, die keinen Einfluß auf die Aktivierung des Neurons haben. Dies kann auf zwei Wegen geschehen:

   - Mittels algorithmischer Methoden wird bestimmt, ob das Potential (Summe aller Gewichte) eines Clusters in den verschiedenen möglichen Potentialstufen Einfluß darauf hat, ob der Schwellwert überschritten wird oder nicht. Cluster, die in keiner der möglichen Potentialstufen die Aktivierung beeinflussen, werden gelöscht. Dieses Verfahren ist den SUBSET–Algorithmen sehr ähnlich. Die Aufwand wird jedoch durch das Clustern verringert.
   - Auf heuristischem Weg werden die Trainingsmuster an das Netz angelegt und sequentiell die Aktivierungen der Cluster auf Null gesetzt. Cluster, bei denen sich eine qualitative Veränderung in der Aktivierung der Einheit ergibt, werden als notwendig markiert. Nach dem Präsentieren aller Muster werden die Cluster gelöscht, die nicht als notwendig markiert wurden.

   Durch das Eliminieren sollen vor allem kleine Cluster mit niedrigen Gewichten gelöscht werden.

4. Optimieren
   Durch die ersten 3 Schritte kann sich die Häufigkeit ändern, mit der eine Einheit aktiviert wird. Deshalb ist es notwendig eine Optimierung durchzuführen. Dazu werden die Gewichte der Cluster gespeichert und die Schwellwerte des Netzes noch einmal mit den Mustern der Trainingsmenge angelernt. Um der regelähnlichen Form des Netzes gerecht zu werden wird dabei eine sigmoide Funktion verwendet, die sich sehr stark an die Schwellwertfunktion annähert. Die Autoren empfehlen einen Parameter $s \geq 5$ für sigmoide Funktion.

5. Extrahieren
   Dieser Schritt faßt die Schwellwerte der Einheiten und die Gewichte der Cluster direkt in Regeln in der Form, daß die Einheit aktiv ist, wenn die Summe der gewichteten Vorgänger den Schwellwert überschreitet.

6. Vereinfachen
   Die Regeln sollen zur Form M von N umgeformt werden. Dazu werden die Schwellwerte und Gewichte ersetzt. MOfN testet die möglichen Kombinationen von Vorgängern,

die den Schwellwert überschreiten. Aus einer extrahierten Regel können mehrere vereinfachte Regeln entstehen. Ergibt eine Regel mehr als 5 neue, so verbleibt sie in ihrem ursprünglichen Zustand.

Zwischenschritt	*Ergebnis*
Clustern	$\{B, C, G\}, \{E\}, \{D, F, H\}$
Durchschnitt bilden	$\{B, C, G\} \rightarrow 4,9$ $\{D, F, H\} \rightarrow 0,8$ $\{E\} \rightarrow -5,0$
Eliminieren	$\{B, C, G\}$ $\{E\}$
Optimieren	$\theta_A = 8,0$
Extrahieren	$A \leftarrow 8,0 < 4,9 \times \text{anzahl\_wahr}\{B, C, G\} + (-5,0) \times \text{anzahl\_wahr}\{E\}$
Vereinfachen	$A \leftarrow 3 \text{ von}\{B, C, G\}$ $A \leftarrow 2 \text{ von}\{B, C, G\}\text{und } 1 \text{ von}\{\neg E\}$

Tabelle 3.4: Extrahieren von Regeln für das Neuron aus Abbildung 3.5 mit dem MOFN–Algorithmus. Das Ergebnis des Algorithmus sind einfache, aber ausdrucksstarke Regeln, die zeigen, daß das Neuron A aktiv ist, wenn mindestens M Einheiten aus einer Gruppe von N Einheiten aktiv sind.

Die Komplexität des MOFN–Algorithmus kann nicht exakt dargelegt werden, weil er in der Optimierungsphase Backpropagation verwendet. Da das Netz aber bereits mit den Mustern trainiert wurde und weniger Verbindungen im Netz sind, als vor Ausführung von MOFN wird dieser Schritt relativ wenig Zeit in Anspruch nehmen. Das Clustern der Verbindungen hat eine Zeit– und Platzkomplexität von $O(u \times l^2)$, wobei u die Anzahl der zu bearbeitenden Einheiten im Netz und l die durchschnittliche Zahl von Verbindungen zu einer Einheit. Mit der Anzahl der Trainingsmuster n benötigt die Eliminierung $O(n \times u \times l)$ an Zeit. Alle anderen Schritte haben einen Aufwand von $O(u \times l)$. Für Schritt 6 ist noch anzumerken, daß ein Vereinfachen ohne Begrenzung auf 5 neue Regeln exponentiellen Aufwand hätte.

Towell und Shavlik zeigen in ihrer Arbeit, daß MOFN weniger und das Netz genauer beschreibende Regeln erzeugt, als SUBSET–Algorithmen. Tabelle 3.4 verdeutlicht dies am Beispiel aus Abbildung 3.5.

3.3.3 Methode von Garcez, Zaverucha und Carvalho

Garcez, Zaverucha und Carvalho [GZC96] empfehlen eine weitere Methode, die auf einer Mischung aus zerlegendem und pädagogischem Ansatz beruht. Sie ist direkt darauf ausgerichtet, logische Programmklauseln zu erstellen, die die Aktivierung der Eingabeneuronen auf die Aktivierung der Ausgabeneuronen abbilden.

Beginnend mit einer heuristischen Vereinfachung sollen die Einheiten in der Eingabeschicht und in der Ausgabeschicht reduziert werden. In der ersten verdeckten Schicht werden dazu, wie bei MOFN, die Verbindungen zu einer verdeckten Einheit geclustert, deren Durchschnitt gebildet, und die Cluster entfernt, die keinen Einfluß auf die Aktivierung der verdeckten Einheit haben. Beim Eliminieren der unnötigen Cluster verwenden die Autoren einen speziellen Algorithmus, der abhängig ist von der Aktivierung A_{min}, ab der ein Neuron als aktiv gilt. Entstehen durch das Eliminieren der Cluster Eingabeeinheiten, die keine Verbindungen zur verdeckten Schicht mehr haben, so werden diese ebenfalls entfernt.

B	C	G	E	A	*erstellte Regel*
0	0	0	0	0	
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	0	
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	1	$A \leftarrow \neg B, C, G, \neg E$
0	1	1	1	0	
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	1	$A \leftarrow B, \neg C, G, \neg E$
1	0	1	1	0	
1	1	0	0	1	$A \leftarrow B, C, \neg G, \neg E$
1	1	0	1	0	
1	1	1	0	1	$A \leftarrow B, C, G, \neg E$
1	1	1	1	1	$A \leftarrow B, C, G, E$

Tabelle 3.5: Extrahieren von Regeln für das Neuron aus Abbildung 3.5 mit der Methode von Garcez, Zaverucha und Carvalho. Wie bei MOFN sind ähnliche Gewichte zusammengefaßt und das Cluster $\{D, F, H\}$ gelöscht worden. Es werden logische Programmklauseln erzeugt, wenn die Einheit A aktiv ist. Die Regel bildet den kompletten Eingabevektor auf die Aktivierung von A ab.

Im nächsten Schritt wird für jeden Fakt A_j in der Ausgabeschicht (Ausgabeeinheit, die immer wahr ist) eine Regel A_j. bzw. $A_j \leftarrow$ in das zu erstellende Programm P eingefügt und das Neuron selbst gelöscht. Die Autoren gehen dabei von einer Topologie des untrainierten Netzes aus, bei der jeder Fakt des ursprünglichen aussagenlogischen Programmes an ein spezielles Eingabeneuron T gekoppelt wird, das immer den Wahrheitswert wahr hat. Um zu bestimmen, ob eine Ausgabeeinheit nach dem Lernen noch einen Fakt darstellt, werden die Bezeichner der Neuronen in den vorhergehenden Schichten beobachtet, mit denen das Netz verbunden ist.

Im pädagogischen Teil des Algorithmus werden dem verkleinerten Netz Eingabemuster präsentiert. Das Netz berechnet die Ausgabe. Für jede Einheit in der Ausgabeschicht, die aktiv ist, wird eine Klausel $A_j \leftarrow L_1, L_2, \ldots L_K$ in das Programm P aufgenommen, wobei A_j das Element der Herbrand–Basis, das durch die Ausgabeeinheit repräsentiert wird. K ist die Anzahl der verbliebenen Eingabeeinheiten, und das Literal L_k $(0 \leq k \leq K)$ ist positiv, wenn die k. Eingabeeinheit den Wahrheitswert wahr hat, sonst negativ.

Das Verfahren garantiert, daß Programmklauseln erstellt werden, die die direkte Abhängigkeit der Ausgabeneuronen von den Eingabeneuronen beschreibt, wie sie das Netz berechnet. Für eine konstante Menge n von Beispielen und eine Anzahl o verbliebener Ausgabeeinheiten beträgt die Zeitkomplexität zur Ermittlung der Regeln aus dem Netz $O(n \times o)$. Will man die Funktion vollständig darstellen, ist es notwendig, alle möglichen Eingabevektoren zu testen. Deren Zahl beträgt jedoch 2^i, mit der Anzahl i der Eingabeneuronen. Bei großen Neuronalen Netze erzwingt der exponentielle Aufwand auch hier die Nutzung einer endlichen Menge von Beispielen. Diese Menge sollte außerdem größer sein, als die Trainingsmenge, da das Lernen sonst nahezu unnötig wäre. Ein weiterer Nachteil ist, daß die Generalisierungsfähigkeit des Neuronalen Netzes beim Lernen nur wenig ausgenutzt wird. Die Aktivierung einer Ausgabeeinheit wird auf alle verbliebenen Einheiten in der Eingabeschicht zurückgeführt. Im ungünstigsten Fall könnte dies nahezu die gesamte Herbrand–Basis umfassen. In [GZC96] wird auf gute Ergebnisse in verschiedenen Lernsystemen verwiesen. Eine Verringerung der Komplexität ist jedoch nur möglich, wenn das Netz stark beschnitten werden kann. Das ist abhängig von der verwendeten Domäne bzw. der zu lernenden Funktion.

4. Realisierung des Gesamtprozesses

Der Gesamtprozess des induktiven Lernens in wissensbasierten Neuronalen Netzen mit den Phasen Transformation des aussagenlogischen Programmes in ein vorwärtsgerichtetes Netz, induktives Lernen und Regelextraktion im trainierten Netz, wurde auf der Basis des Stuttgarter Neuronale Netze Simulators (SNNS) realisiert. Damit wurden verschiedene Tests durchgeführt, deren Ergebnisse im Kapitel 5 dargelegt werden. Um die Untersuchungsergebnisse reell einschätzen zu können, soll jetzt noch einmal auf die genaue Implementierung eingegangen werden.

4.1 Realisierung der Algorithmen

Das gegebene Wissen in Form des aussagenlogischen Programmes wurde nach dem Übersetzungsalgorithmus in ein 3–schichtiges vorwärtsgerichtetes Neuronales Netz transformiert. Die Neuronen erhielten sigmoide Aktivierungsfunktionen mit dem Parameter $s = 1$ (Standard). Die Verbindungen zwischen den Einheiten wurde mit dem Wert $W(A_{min})$ gewichtet (siehe Abschnitt 3.1). Gleichzeitig mit der Erstellung des Netzes wurden zwei Beispielmengen erstellt. Die eine Menge repräsentiert dabei gültige Muster (Muster, die der Berechnung von T_P entsprechen) und die zweite Menge enthält Beispiele, die der Operator T_P nicht berechnet (ungültige Muster). Da die Anzahl möglicher Eingabevektoren exponentiell mit der Zahl der Eingabeneuronen wächst, kann nur eine begrenzte Trainingsmenge erstellt werden. Die Muster der gültigen Menge wurden so erstellt, daß für jede Klausel der Form $A_j \leftarrow L_1, L_2, \ldots, L_K$ aus dem Programmm P ein Beispiel erzeugt wird, in dem die Konjunktion der Literale erfüllt ist, und ein Beispiel, in dem jedes der Literale den Wahrheitswert falsch besitzt. Die ungültigen Beispiele wurden per Zufall erstellt. Beide Mengen wurden beim Training sowohl zum Lernen, als auch zum Testen verwendet.

Für das Trainieren des vorwärtsgerichteten Netzes wurde der Simulator SNNS genutzt. Der verwendete Lernalgorithmus war Standard–Backpropagation. Verbesserungen von Backpropagation könnten zwar das Lernverhalten verbessern, würden aber möglicherweise die Testergebnisse zu stark verändern. Die gewählte Lernrate η wurde variiert. Meist betrug ihr Wert jedoch $\eta = 0.2$. Die maximale Toleranz d_{max} beim Lernen wurde auf 0.0 gesetzt. Eine Trainingsmenge wurde solange angelernt, bis der globale Fehler kleiner als 1 war. Wurden dazu mehr als 4000 Epochen benötigt, oder änderte das Netz seinen Fehler während 1000 Epochen nicht, so wurde das Training abgebrochen.

Die Grundlage der Regelextraktion bildete der MOFN–Algorithmus von Towell und Shavlik. Das Clustern der Verbindungen jeder Einheit in der verdeckten Schicht oder Ausgabeschicht erfolgte mit einem hierarchischen Clusteralgorithmus[1], der auf Sammlungen (JOINs) beruht. Er fügt solange Cluster zusammen, bis kein Paar von Clustern mehr existiert, dessen Abstand kleiner ist, als 0.125. Dieser Wert beschreibt den Radius innerhalb eines Clusters. Als Abstandsmaß zwischen zwei Gewichten wurde der Euklidsche Abstand und zwischen Mengen bzw. Clustern der Hausdorffabstand[2] verwendet. Wegen der Vorteile in der Zeitkomplexität wurde beim Eliminieren die heuristische Methode unter Nutzung der Trainingsmenge eingesetzt. Dadurch ist das Ergebnis der Regelextraktion stark von dieser Menge abhängig. Cluster erschienen als notwendig, sobald sie in mindestens einem Muster einen Unterschied von 0.4 in der Aktivierung des Neurons nach sich zogen. Die Optimierung trainierte die Schwellwerte mindestens 10 Epochen und höchstens 50 Epochen, da davon ausgegangen werden kann, daß das Netz bereits auf die Traningsmenge trainiert war. War der globale Fehler des Netzes kleiner als 0.5 wurde hier bereits die Optimierung beendet. Das Vereinfachen wurde mit einem SUBSET– ähnlichen Algorithmus, der als Tiefensuche implementiert war, durchgeführt. Die Menge der aus einer komplexen Regel zu erstellenden einfachen Regeln wurde auf 20 begrenzt.

4.2 Nachbearbeitung der MOFN–Regeln

Da das Ergebnis der Regelextraktion wieder als aussagenlogisches Programm vorliegen sollte, war es notwendig, einige weitere Schritte auszuführen:

- Umwandlung der Regeln der Form M von N in Regeln der Form M von M
- Substitution der Einheiten der verdeckten Schicht in den Regeln der Ausgabeeinheiten
- Sammeln der positiven und negativen Literale in jeweils einem Cluster
- Enfernen unerfüllbarer Regeln und subsumierter Regeln
- Niederschreiben der Regeln als aussagenlogisches Programm

Die Abarbeitung dieses Vorganges soll an einem Beispiel demonstriert werden.

[1] Die Grundlagen von Clusteralgorithmen sind in [JD88] und [Miy90] zu finden.

[2] Der Hausdorffabstand ist ein Begriff aus der Theorie metrischer Räume. Er ist ein Maß für die Distanz zwischen Mengen und ist abhängig vom Abstandsmaß zwischen den Elementen dieser Mengen. Ausführungen zu diesem Thema sind in [Bar93] zu finden.

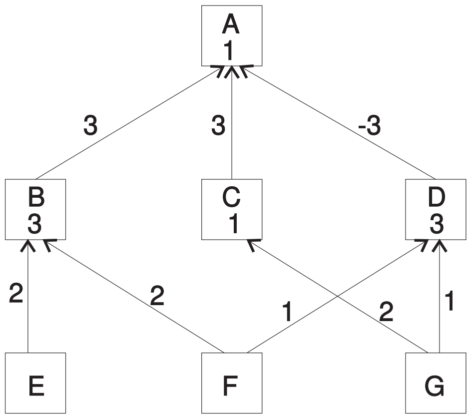


Abbildung 4.1: Ein 3–schichtiges vorwärtsgerichtetes Netz nach der Bearbeitung mit MOFN. Unnötige Verbindungen sind entfernt worden und Gewichte eines Clusters haben denselben Wert. Das Netz hat eine Ausgabeeinheit A und jeweils 3 Einheiten in der verdeckten und in der Eingabeschicht. Die Zahlen unter den Namen der Einheiten geben den internen Schwellwert an.

Beispiel 4.2.1 *Nach der Bearbeitung durch MOFN sei ein Neuronales Netz entstanden, aus dem die Regeln*

$$\begin{aligned}
A \leftarrow 1 < \quad & 3 \times anzahl_wahr\{B, C\} \quad +(-3) \times anzahl_wahr\{D\} \\
B \leftarrow 3 < \quad & 2 \times anzahl_wahr\{E, F\} \\
C \leftarrow 1 < \quad & 2 \times anzahl_wahr\{G\} \\
D \leftarrow 3 < \quad & 1 \times anzahl_wahr\{F, G\}
\end{aligned}$$

extrahiert wurden. Abbildung 4.1 zeigt ein solches Netz. Das Vereinfachen der Regeln im letzten Schritt des Algorithmus ergibt:

$$\begin{aligned}
A &\leftarrow 2\ von\{B, C\} \\
A &\leftarrow 1\ von\{B, C\} und\ 1\ von\{\neg D\} \\
B &\leftarrow 2\ von\{E, F\} \\
C &\leftarrow 1\ von\{G\}
\end{aligned}$$

Damit die Regeln direkt in Programmklauseln übersetzt werden können, müssen sie in eine Form M von M transformiert werden. Das hätte zur Folge, daß wirklich jedes Ele-

ment einer Gruppe[3] notwendig wäre, und eine Gruppe damit faktisch eine Konjunktion von Elementen darstellt. Allerdings können aus einer MOFN–Regel mehrere neue Regeln entstehen. Diese Regeln werden kombinatorisch erzeugt, indem die möglichen Teilmengen einer Gruppe mit denen der anderen Gruppen kombiniert werden. Die Ausdrucksstärke einer MOFN–Regel geht damit verloren und wird auf viele speziellere Regeln verteilt. Die Umwandlung hat eine Zeit– und Platzkomplexität von $O(r \times g^c)$. r bezeichnet die Anahl der Regeln, g die durchschnittliche Anzahl der Gruppen in jeder Regel und c die durchschnittliche Anzahl von Kombinationen, die aus einer Gruppe hervorgehen. Für ein Cluster der Form M von N ergeben sich $c = \frac{N!}{M!(N-M)!}$ Kombinationen. Für das dargestellte Beispiel sind $r = 4$, $g \approx 1$ und $c = 1$.
Die Nachteile des Algorithmus sind offensichtlich sein exponentieller Aufwand und der Verlust der Ausdrucksstärke der Regeln. Für die Erstellung eines aussagenlogischen Programmes ist er jedoch notwendig. Die Komplexität wird in der Implementierung durch eine Grenze beschränkt, die festlegt, wieviele Regeln M Of M aus einer MOFN–Regel entstehen dürfen. Beim Experimentieren wurde ein Wert von 100 genutzt. Für das angeführte Beispiel ergeben sich folgende neue Regeln:

$$\begin{aligned}
A &\leftarrow 2 \text{ von}\{B, C\} \\
A &\leftarrow 1 \text{ von}\{B\}\text{und } 1 \text{ von}\{\neg D\} \\
A &\leftarrow 1 \text{ von}\{C\}\text{und } 1 \text{ von}\{\neg D\} \\
B &\leftarrow 2 \text{ von}\{E, F\} \\
C &\leftarrow 1 \text{ von}\{G\}
\end{aligned}$$

Wir haben nun eine Abbildung jeder Einheit von den Einheiten der Vorgängerschicht. Die Einheiten der verdeckten Schicht sind keine Elemente der Herbrand–Basis unseres Ausgangsprogrammes. Für sie müßten demzufolge neue Variablen eingeführt werden. Die Herbrand–Basis unseres Programmes soll jedoch konstant bleiben. Also ist es notwendig, die versteckten Einheiten in den Regeln zu ersetzen, durch die Regeln, durch die sie selbst beschrieben werden. Ein weiteres Problem tritt auf, weil die zu ersetzenden Einheiten als negative Literale auftreten können, jedoch nur Regeln für einen Wahrheitswert wahr existieren. Also müssen Regeln aufgestellt werden, die beschreiben, wann eine Einheit der verdeckten Schicht nicht aktiv ist:

$$\begin{aligned}
\neg B &\leftarrow 1 \text{ von}\{\neg E, \neg F\} \\
\neg C &\leftarrow 1 \text{ von}\{\neg G\} \\
\neg D &\leftarrow
\end{aligned}$$

Dazu ist auf die ursprünglich extrahierten Regeln des Netzes zurückzugreifen und ein dem Vereinfachen ähnlicher Algorithmus anzuwenden. In der Implementierung werden

[3] als Gruppe soll allgemein jede Menge von Einheiten bezeichnet werden, die durch Transformation aus einem Cluster hervorgegangen und in der Regel enthalten ist.

für alle verdeckten Einheiten solche Regeln erstellt, um auch bei Netzen mit mehr als 3 Schichten eine Substitution zu ermöglichen. Die Regeln

$$\begin{array}{rcl} \neg B & \leftarrow & 1 \text{ von}\{\neg E\} \\ \neg B & \leftarrow & 1 \text{ von}\{\neg F\} \\ \neg C & \leftarrow & 1 \text{ von}\{\neg G\} \\ \neg D & \leftarrow & \end{array}$$

ergeben sich nach der Transformation in M von M. Nun kann das Ersetzen der verdeckten Einheiten durchgeführt werden.

$$\begin{array}{rcl} A & \leftarrow & 2 \text{ von}\{B, C\} \\ & \leftarrow & 1 \text{ von}\{C\}\text{und } 2 \text{ von}\{E, F\} \\ & \leftarrow & 2 \text{ von}\{E, F\}\text{und } 1 \text{ von}\{G\} \\ & \leftarrow & 3 \text{ von}\{E, F, G\} \\ A & \leftarrow & 2 \text{ von}\{E, F\} \\ A & \leftarrow & 1 \text{ von}\{G\} \end{array}$$

Auch dabei können aus einer Regel mehrere neue entstehen. Die Platz– und Zeitkomplexität beträgt $O(r_o \times (r_h)^h)$, wobei r_o die absolute Anzahl der Regeln für Ausgabeeinheiten und r_h die durchschnittliche Anzahl der Regeln für jede verdeckte Einheit. h bezeichnet die durchschnittliche Anzahl der Einheiten in einer der r_o Regeln. Angewandt auf das Beispiel bedeutet dies, $r_o = 3$, $h = 2$ und $r_h = 1$.
Das Sammeln der Einheiten in jeweils einer Gruppe für positive und negative Literale wird in unserem Beispiel gleich mit durchgeführt. Nach dem Entfernen subsumierter Regeln ergibt sich

$$\begin{array}{rcl} A & \leftarrow & 2 \text{ von}\{E, F\} \\ A & \leftarrow & 1 \text{ von}\{G\} \end{array}$$

bzw. als aussagenlogisches Programm

$$\begin{array}{rcl} A & \leftarrow & E, F \\ A & \leftarrow & G \end{array}$$

Die Transformation nach Regeln der Form M von M und das Substituieren haben exponentiellen Aufwand. Die anderen Schritte lediglich polynomiale Komplexität. Die gesamte Nachbearbeitung der MOFN–Regeln hat deshalb exponentielle Platz– und Zeitkomplexität und transformiert die wenigen, MOFN–Regeln in viele aussagenlogische

Klauseln. Damit geht die Übersichtlichkeit und Ausdrucksstärke von MOFN verloren. Für die Experimente war dies jedoch notwendig. Die Nachbearbeitung kann problemlos durchgeführt, wenn der MOFN–Algorithmus viele unnötige Verbindungen entfernt und kleine Cluster bildet. Ist dies nicht der Fall, kann die Anzahl der zu erstellenden Regeln förmlich "explodieren".

4.3 Verwendete Domänen

In der Literatur werden für Untersuchungen der Regelextraktion in Neuronalen Netzen häufig Domänen aus dem Gebiet der Molekularbiologie oder der Medizinischen Diagnostik genutzt. Im Rahmen dieser Arbeit wurden Wissensbanken aus dem Bereich der Hardwareverifikation und der Botanik verwendet.

4.3.1 Hardwareverifikation

Das Wissen, das zur Erstellung aussagenlogischer Programme diente, lag in Form von aussagenlogischen Formeln in konjunktiver Normalform vor. Der Wahrheitswert einer solchen Formel entsprach der Aussage über die Korrektheit der Schaltung. Jede aussagenlogische Variable gab Auskunft über eine Teilschaltung oder über Messungen in ihnen. Die zugehörigen Programme waren normale logische Programm und definite Programme.

$$\begin{aligned} F \quad = \quad & V5 \wedge (\neg V6 \vee V3) \wedge (\neg V6 \vee \neg V2) \wedge (\neg V7 \vee V1 \vee V6) \wedge (\neg V8 \vee V3) \wedge \\ & (\neg V8 \vee \neg V4) \wedge (\neg V9 \vee V1 \vee V8) \wedge (\neg V10 \vee V7 \vee V9) \wedge (\neg V11 \vee V2 \vee \neg V3) \wedge \\ & (\neg V12 \vee V11) \wedge (\neg V12 \vee V1) \wedge (\neg V13 \vee V4 \vee \neg V3) \wedge (\neg V14 \vee V13) \wedge \\ & (\neg V14 \vee \neg V1) \wedge (\neg V15 \vee V12 \vee V14) \wedge (\neg V16 \vee V15) \wedge (\neg V16 \vee V10) \wedge \\ & (\neg V17 \vee V2 \vee V1 \vee \neg V4) \wedge (\neg V18 \vee \neg V2 \vee V1 \vee V4 \vee \neg V3) \wedge (\neg V19 \vee V18) \wedge \\ & (\neg V19 \vee V17) \wedge (\neg V5 \vee V19) \wedge (\neg V5 \vee V16) \end{aligned}$$

```
%==== Symbols ====
% V1 := B
% V2 := A
% V3 := D
% V4 := C
% V5 := _1749
% V6 := _1735
% V7 := _1736
% V8 := _1737
% V9 := _1738
```

% V10 := __1739
% V11 := __1740
% V12 := __1741
% V13 := __1742
% V14 := __1743
% V15 := __1744
% V16 := __1745
% V17 := __1746
% V18 := __1747
% V19 := __1748

Beispiel eines Problems aus der Hardwareverifikation und Bedeutung der aussagenlogischen Variablen. Leider konnte die genaue Bedeutung der Variablen nicht einer direkten Teilschaltung zugeordnet werden, weil keine zugehörigen Schaltpläne vorlagen. Die Probleme sind jedoch der realen Welt entnommen und nicht künstlich erzeugt.

4.3.2 Botanik

Um die Algorithmen an allgemein verständlichen Konzepten darzustellen, wurden aus Wissenssammlungen der Botanik Teile entnommen und als aussagenlogische Programme dargestellt. Die Auszüge sind deshalb nur begrenzt, und die Gültigkeit der Programmklauseln beschränkt sich auf die ausgewählten botanischen Begriffe. Im wesentlichen geht es in den Programmen um die Bestimmung von Baumarten, anhand von Blattmerkmalen. Die entstehenden Programme waren sowohl normal, als auch definit. Die Daten entstammen [Coo94] und [Bru95].

Übersicht über verwendete Nadelbäume		
Familie	*Gattung*	*Name*
pinaceae	abies	weisstanne
	larix	europlaerche
	picea	gemeinefichte
	pinus	zirbelkiefer
		schwarzkiefer
		gemeinekiefer
taxaceae	taxus	gemeineeibe
taxodiaceae	sequoiadendron	mammutbaum
cupressaceae	thuja	lebensbaum
	juniperus	wacholder

Übersicht über verwendete Merkmale	
Merkmal	*Bedeutung*
sommergruen	Baum trägt nur im Sommer Blätter
immergruen	Baum traegt immer Blätter
frzapfen	Früchte sind Zapfen
blnadeln	Blätter sind Nadeln
blschuppenf	Blätter sind schuppenfoermig
blbueschwirt	Blätter in Büscheln oder Wirteln
bleinzeln	Blätter einzeln
blverdeckt	Austriebe von Blättern oder Blattgrund verdeckt
blletztgruen	letztjährige Triebe grün
blfaecher	Blattwerk fächerförmig
blwedeln	Blattwerk in unregelmässigen Wedeln

5. Ergebnisse

5.1 Ein Beispiel aus der Botanik

Gegeben sei die Aufgabe, Wissen in Form eines logischen Programmes darüber zu aquirieren, wie man aus den Blattmerkmalen eines Baumes seine Familie, Art oder Gattung bestimmt. Das gewählte Beispiel ist natürlich nur ein kleiner Auszug aus diesem Komplex und beschränkt sich auf die Gattungen Larix(Lärchen), Pinus(Kiefern), Picea(Fichten) und einige Blattmerkmale. Das gegebene Wissen in Form eines aussagenlogischen Programmes $P_{Initial}$ besteht aus den folgenden Programmklauseln:

$\mathbf{P_{Initial}}$:
nadelbaum $\leftarrow$ blnadeln.
nadelbaum $\leftarrow$ larix.
nadelbaum $\leftarrow$ picea.
nadelbaum $\leftarrow$ pinus.
pinus $\leftarrow$ immergruen, blnadeln, $\neg$blschuppenf, $\neg$bleinzeln, frzapfen.

Das aus diesem Programm erstellte semantikgleiche neuronale Netz besteht aus 9 Eingabe– und Ausgabeeinheiten (Anzahl aussagenlogischer Variablen bzw. Größe der Herbrand–Basis) und 5 Einheiten in der verdeckten Schicht (Anzahl der Regeln) und berechnet den Operator T_P für das Programm. Abbildung 5.1 zeigt das entsprechende SNNS–Netzwerk.

Das Netz wird nun mit neuem Wissen in Form von Beispielen trainiert. Die Menge aller möglichen Eingabevektoren für das Netz beträgt $2^9 = 512$. Aus dieser Menge wurden 60 Eingabemuster zufällig ausgewählt und der erwünschte Ausgabevektor für das Netz nach dem Programm

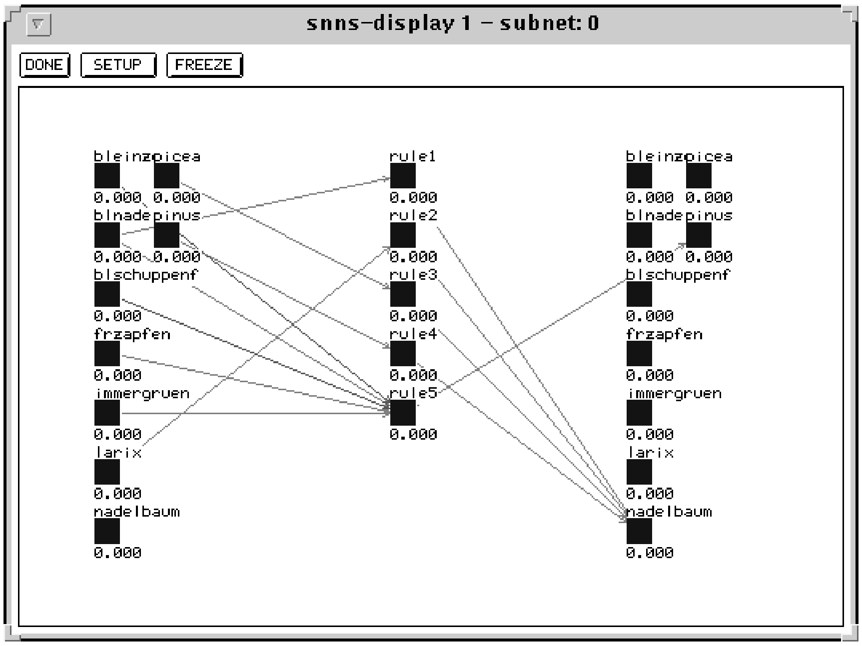


Abbildung 5.1: Das SNNS–Netzwerk zum gegebenen aussagenlogischen Programm $P_{Initial}$. Das Netz ist schichtweise vollvernetzt. Alle mit 0 gewichteten Verbindungen werden nicht angezeigt.

$\mathbf{P_{Training}}$:
nadelbaum ← blnadeln.
nadelbaum ← frzapfen.
nadelbaum ← larix.
nadelbaum ← picea.
nadelbaum ← pinus.
larix ← blnadeln, ¬immergruen.
larix ← frzapfen, ¬immergruen.
pinus ← blnadeln, immergruen, ¬bleinzeln, ¬blschuppenf.
pinus ← frzapfen, immergruen, ¬bleinzeln, ¬blschuppenf.
picea ← blnadeln, immergruen, bleinzeln, ¬blschuppenf.
picea ← frzapfen, immergruen, bleinzeln, ¬blschuppenf.

berechnet. Aus den Paaren von Eingabe– und erwünschtem Ausgabevektor wurden die Trainingsbeispiele erstellt. Das Netz soll folglich dieses Wissen trainieren. Es enthält dabei sowohl alte, als auch neue Zusammenhänge. Abbildung 5.2 zeigt den Verlauf des globalen Fehlers während des Lernvorganges.

Aus dem angelernten Netz wird das neue Wissen extrahiert. Ergebnis der Wissensaquisition ist das Programm

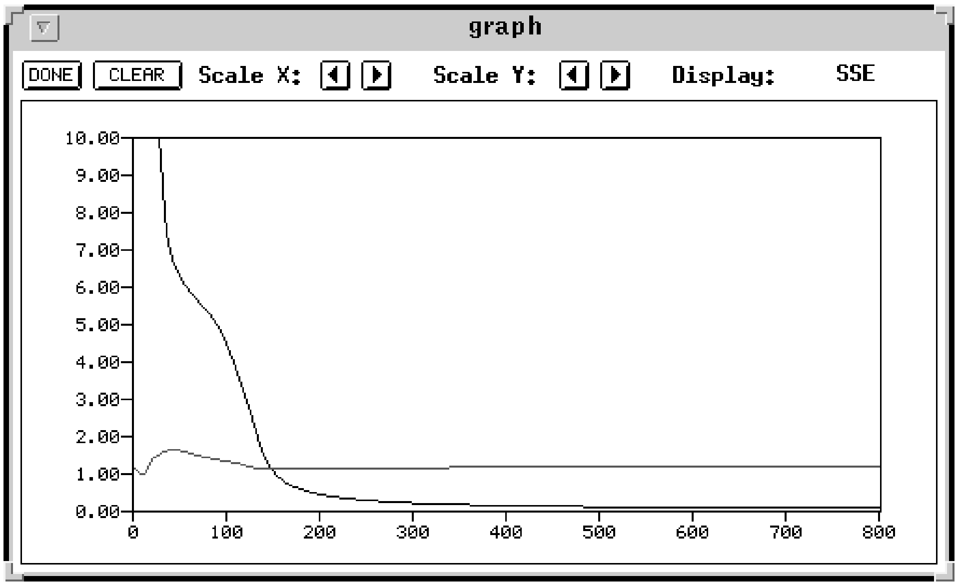


Abbildung 5.2: Der Verlauf des globalen Fehlers $E(\mathbf{w})$ während des Lernvorganges. Die Traningsmenge verursacht zu Beginn einen sehr großen Fehler. Am Ende des Trainings ist ihr Fehler fast 0, die Trainingsbeispiele sind gelernt. Die Testmenge(Muster des Initialprogrammes) hat am Beginn einen Fehler nahe 0 und nimmt nach dem Training einen konstanten Wert ein.

$\mathbf{P}_{\mathbf{Extrahiert}}$:
nadelbaum ← bleinzeln, blnadeln, ¬blschuppenf, ¬immergruen.
nadelbaum ← frzapfen, picea.
nadelbaum ← blnadeln, picea.
nadelbaum ← pinus.
nadelbaum ← larix.
larix ← frzapfen, ¬immergruen.
larix ← blnadeln, ¬immergruen.
pinus ← blnadeln, immergruen, ¬bleinzeln, ¬blschuppenf.
pinus ← frzapfen, immergruen, ¬bleinzeln, ¬blschuppenf.
picea ← blnadeln, immergruen, pinus, ¬blschuppenf.
picea ← blnadeln, immergruen, bleinzeln, ¬blschuppenf.

Die Güte der extrahierten Regeln wurde anhand der Trainingsmenge überprüft. Daraus ergab sich, daß von den gegebenen 60 Beispielen 53 Beispiele korrekt klassifiziert wurden. In den anderen 7 Beispielen trat jeweils nur ein Fehler bei der Klassifizierung einer der aussagenlogischen Variablen *nadelbaum* oder *picea* auf. Die Regelextraktion hat also mit einer Fehlerwahrscheinlichkeit von etwa 1,5% die Funktion des Netzes korrekt wiedergegeben.

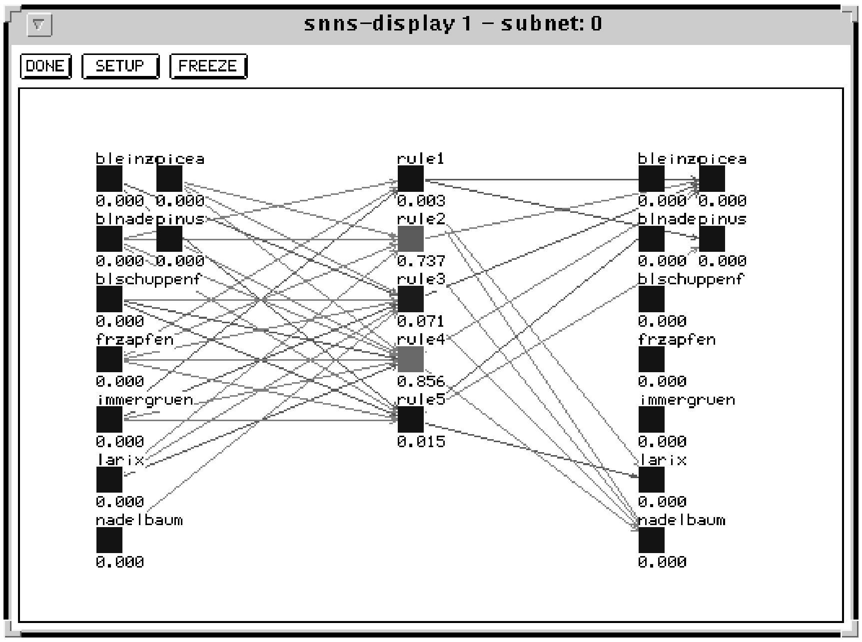


Abbildung 5.3: Das SNNS–Netzwerk nach dem Training bei Anlegen desselben Eingabevektors wie in Abbildung 5.1. Die Anzahl der stark gewichteten Verbindungen hat zugenommen. Die Bedeutung der Einheiten in der verdeckten Schicht hat sich verändert. Ihre Aktivierungen sind nicht mehr nur maximal.

Aus den dargestellten Ergebnissen läßt sich folgendes erkennen:

- Die Regelextraktion beschreibt mit einer relativ geringen Fehlerwahrscheinlichkeit die Funktion des Netzwerkes, obwohl sich die Bedeutung der Einheiten in der verdeckten Schicht geändert hat und die Aktivierungen nicht mehr nur maximal sind.
- Die Übersichtlichkeit der Funktion des Netzes geht verloren.
- Die Generalisierungsfähigkeit des Netzes wird beim Lernen ausgenutzt und bleibt bei der Regelextraktion erhalten.
- Das angelernte Wissen im Netz ist sehr stark von den gewählten Trainingsbeispielen abhängig; ein direkter Zusammenhang zwischen einer zu lernenden Programmklausel und einem Beispiel kann nicht gefunden werden.

- Es können Zusammenhänge vom Netz erkannt werden, die in der Realität falsch sind.

5.2 Weitere Untersuchungen

Notwendigkeit vollvernetzter Schichten:
Zur Berechnung der Semantik des Ursprungsprogrammes sind nur einige Verbindungen im Netz notwendig. Werden zum Lernen lediglich diese Verbindungen benutzt, schränkt das die Möglichkeiten des Netzes sehr stark ein. Es können lediglich Zusammenhänge zwischen Elementen der Herbrand–Basis trainiert werden, die schon im Ausgangsprogramm enthalten waren. Für das Lernen jeder beliebigen Funktion ist es notwendig, die Schichten voll zu vernetzen. Dies kann getan werden, indem zwischen jedem Paar von Einheiten benachbarter Schichten eine Verbindung mit dem Betrag 0 eingeführt wird. Das Lernverhalten kann man verbessern, wenn man die Gewichte zufällig mit Werten nahe 0 wichtet.

Einfluß von $\mathbf{A_{min}}$ auf das Lernverhalten:
Die Wahl des Parameters A_{min} bei der Erstellung des Netzes hat großen Einfluß auf die Fähigkeit des Netzes, zu lernen. Wählt man A_{min} sehr groß, bedeutet dies, daß jede Einheit bei Anlegen eines Eingabevektors entweder eine sehr hohe Aktivierung (≈ 1) oder eine sehr kleine Aktivierung (≈ 0) hat. In diesen Bereichen ist der Gradient der Sigmoid–Funktion fast 0. Das Ergebnis ist, daß das Netz zum Anlernen der Trainingsmenge sehr viele Epochen benötigt. Meist tritt überhaupt kein sichtbarer Lernerfolg ein. Ist A_{min} klein, dann liegen die Aktivierungen der Neuronen in einem Bereich, in dem das Netz besser lernfähig ist. Mit sinkendem Wert von A_{min} besteht allerdings die Gefahr, daß die erstellten Netze den Operator T_P nicht mehr korrekt berechnen. Man müßte also vor dem eigentlichen Lernvorgang die Beispiele von T_P trainieren. Außerdem sind dann die Vorraussetzungen für die Regelextraktion nicht gegeben, weil die Einheiten nicht mehr nur maximal aktiv oder inaktiv sind. In der Botanik–Domäne berechneten die Netze auch bei Werten um $0,6$ T_P gut. In der Hardwareverifikation führten sie dagegen zu großen Fehlerwerten. Einen Lernen mit $A_{min} \geq 0,9$ war in beiden Domänen fast völlig erfolglos. Die besten Resultate wurden mit Werten im Bereich $0,7 \leq A_{min} \leq 0,8$ erreicht.

Qualität der extrahierten Regeln:
Die Vorraussetzung für den Extraktionsalgorithmus war, daß alle Einheiten des Netzes entweder maximal aktiv oder maximal inaktiv sein sollten. Die Bedeutung der Einheiten sollte sich außerdem nicht ändern. Für die Einheiten in der Ausgabeschicht konnten diese Vorraussetzungen nach dem Training auch erfüllt werden. In der verdeckten Schicht änderte sich während des Lernens die Bedeutung der Einheiten. War die zu lernende Funktion sehr komplex und die Anzahl der verdeckten Einheiten klein, wurde in der verdeckten Schicht eine Art Kodierung für die Eingabevektoren erzielt. Die Bedeutung der Einheiten war nicht mehr ersichtlich und die Aktivierungen nicht mehr nur maximal. In den Schichten wurden kaum noch die ursprünglichen UND bzw. ODER–Funktionen realisiert. Bei Netzen, bei denen die Bedingungen gegeben waren, lag die Menge falsch

klassifizierter Beispiele mit den extrahierten Programmklauseln bei etwa 15%. Bei komplexen Funktionen war die Qualität der MOFN–Regeln wesentlich besser, als die der logischen Programmklauseln. In der Hardwareverifikation ist die Qualität der Regeln schwer zu beurteilen, weil die Ausgabevektoren für den größten Teil der Beispiele gleich sind. Genaue Fehlerquoten zu MOFN sind in [TS92] zu finden.

Ausnutzung der Generalisierungsfähigkeit:
Während der Experimente wurde oft der Versuch unternommen, ein Zielprogramm zu erstellen, welches vom Netz gelernt werden sollte, und als Trainingsmenge wurden Beispiele genutzt, die diesem Zielprogramm entsprachen. Meist enthielt das extrahierte Programm jedoch völlig andere Programmklauseln als die erwünschten. Um dem Netz ein gewünschtes Zielprogramm anzutrainieren, mußte die Trainingsmenge meist alle möglichen Eingabevektoren enthalten. Ein direkter Zusammenhang zwischen einem Trainingsmuster und einer daraufhin gelernten Programmklausel konnte nicht gefunden werden. Dies ist als Vorteil zu werten, da es zeigt, daß die Generalisierungsfähigkeit des Netzes ausgenutzt wird. Das Netz erkennt Zusammenhänge aus der Trainingsmenge. Wissen, das in Form von Programmklauseln vorliegt, braucht auch nicht erst zu aquiriert werden, sondern unbekannte Zusammenhänge sollen ermittelt werden. Genau dies geschieht. Im Bereich der Hardwareverifikation war es schwerer, Muster zu finden, in denen die Hardware als korrekt galt. Dies wären genau die Muster gewesen, die die Formel erfüllen. Meist enthielt die automatisch erzeugte Trainingsmenge deshalb Beispiele mit gleichen Ausgabevektoren. Die Generalisierung ging dann soweit, daß die Formel immer falsch war. In der Domäne Botanik wurden Zusammenhänge zwischen Elementen der Herbrand–Basis gelernt, die in der Realität völlig falsch sind (Bsp.: $gemeinefichte \leftarrow europlaerche$). Hier wäre es vielleicht sinnvoll, eine hierarchische Netzstruktur zu nutzen, wie sie in [TS92] verwendet wird.

Konsistenz der ursprünglichen Regeln: Die Netze haben aufgrund ihrer Topologie das Bestreben, die ursprünglichen Regeln beizubehalten. Trainingsmengen, die nur aus spezialisierenden Beispielen bestanden, benötigten wesentlich mehr Epochen zum Lernen, als generalisierende Beispiele. Spezialisierende Beispiele "widersprechen" dem initialen Programm. Sie verursachen, daß bereits vorhandenes Wissen revidiert werden muß. Generalisierende Beispiele fügen praktisch neues Wissen hinzu. Die Netztopologie unterstützt die Naturtreue der Regeln.

6. Zusammenfassung und zukünftige Arbeiten

Die vorliegende Arbeit untersuchte die Möglichkeiten, vorhandenes Wissen in Form eines aussagenlogischen Programmes mit Hilfe neuronaler Netze zu erweitern oder zu verändern. Als entscheidende Problemstellen erwiesen sich dabei die Transformationen zwischen der symbolischen Darstellung des Wissens und der konnektionistischen Repräsentation.
Die Operation T_P kann für jedes normale logische Programm mit einem 3–schichtigen vorwärtsgerichteten Netz aus binären Schwellwerteinheiten berechnet werden. In der Arbeit wurde gezeigt, das mit dem Übersetzungsalgorithmus nicht für alle normalen logischen Programme ein solches Netz mit sigmoider Aktivierungsfunktion erstellt werden kann.
Die Fähigkeiten Neuronaler Netze im induktiven Lernen sind sehr gut, werden jedoch beeinflußt durch die vorgegebene Topologie und die Wichtung der Verbindungen. Es hat sich gezeigt, daß bei der Erstellung des Netzes mit zu hohen Gewichten ein erfolgreiches Training fast gänzlich verhindert wird. Wissen, das bereits im ursprünglichen Programm enthalten war, und Konzepte, die generalisiert werden müssen, werden durch die verwendete Topologie beim Lernen unterstützt.
Um die Vorbedingungen für zerlegende Regelextraktionsverfahren zu erfüllen, ist es notwendig, das Netz sehr lange zu trainieren, so daß alle Einheiten maximal aktiv oder maximal inaktiv sind. Das gelang solange recht gut, wie die Vielfalt der zu trainierenden Muster nicht zu groß wurde. Waren die zu lernenden Funktionen zu komplex, dann änderte sich die Bedeutung der Einheiten in der verdeckten Schicht und ihre Aktivierungen waren nicht mehr nur maximal.
Die zerlegenden Algorithmen zur Regelextraktion sind für die Darstellung als logisches Programm nur für kleinere und mittlere Netze sinnvoll anwendbar. Ist es jedoch nicht dringend notwendig, logische Programmklauseln zu erstellen, zeigt MOFN mit den wenigen, komplexen Regeln gute Ergebnisse, die für die Verarbeitung Expertensysteme gut geeignet sind. Die Ausdrucksstärke und Übersichtlichkeit der MOFN–Regeln sollte nicht durch Umformungen in Programmklauseln zerstört werden. Außerdem ist die Zeit– und Platzkomplexität dafür exponentiell.
Eine gute und sichere Methode zur Extraktion von Regeln ist der pädagogische Ansatz mit vorbereitendem Vereinfachen des Netzes. Die Generalisierungsfähigkeit des Netzes

wird dabei nur gering ausgenutzt. Es sind weitere Deduktionsmechanismen notwendig, um allgemeine Konzepte zu finden, da in ungünstigen Fällen die Aktivierung einer Einheit durch die ganze Herbrand–Basis beschrieben wird.

Die Arbeit zeigt, daß die Probleme der Transformation von formalem Wissen in Neuronale Netze und umgekehrt noch nicht gelöst sind. Weiterführende Arbeiten könnten sich damit beschäftigen, zu untersuchen, ob und wie ein Algorithmus arbeiten könnte, der für alle logischen Programme ein semantikgleiches Netz erstellt. Für die Verbesserung der Lernfähigkeit wäre es interessant, ob für bestimmte Domänen die Verwendung einer hierarchischen Netzstruktur besser geeignet ist. Überlegungen zur Regelextraktion sollten dahin gehen, weniger streng deterministische Algorithmen zu entwickeln, da die Komplexität der Neuronalen Netze meist zu groß dafür ist. Interessant könnte dabei sein, ob völlig andere Algorithmen, die heuristisch oder stochastisch arbeiten, wesentliche Fortschritte erzielen (z.Bsp.: generische Algorithmen). Ein besonders wichtiger Punkt ist der Übergang von der Aussagenlogik zur Darstellung prädikatenlogischen Wissens. Erst wenn es gelingt, prädikatenlogische Regeln auf Neuronalen Netzen umzusetzen, kann man der Aquisition von realem–Welt–Wissen näher kommen.

Für die Betreuung dieser Belegarbeit und die Hinweise zur Literatur, sowie die Lektüre des Manuskripts möchte ich Yvonne Kalinke danken.

Für das mehrfache Korrekturlesen, die wertvollen Hinweise zu Form und Inhalt der Arbeit, sowie die anspornenden Ratschläge danke ich Frank Wehner.

Gerd Neugebauer unterstützte mich bei der Suche nach einer geeigneten Wissensbasis für meine Tests und stellte mir die Daten der Hardwareverifikation zur Verfügung. Dafür Vielen Dank !

A. Dateiformate

In den Abschnitten der Programmdokumentation B wird häufig auf verschiedene Dateitypen verwiesen. Um die Beschreibungen leichter verständlich zu machen, folgt eine kurze Übersicht dieser Dateitypen.

Übersicht über verwendete Dateien		
Dateityp	*Beschreibung*	*verwendet in*
"<name>.net"	SNNS–Datei zur Speicherung von Netzwerken	plp2snns, mofn
"<name>.pat"	SNNS–Datei zur Speicherung von Trainingsbeispielen	plp2snns, mofn
"<name>.cfg"	SNNS–Konfigurationsdatei	plp2snns
"<name>.sat"	Textdatei, welche eine aussagenlogische Formel in konjunktiver Normalform repräsentiert	sat2plp
"<name>.plp"	Textdatei, die ein aussagenlogisches Programm darstellt	sat2plp, plp2snns, mofn
"mofn.protocol"	Protokolldatei, die erstellt wird	mofn

Ausführliches über das Dateikonzept von SNNS wird in [SNN95] erläutert. Auf die Dateien der Form ".sat" und ".plp" wird nachfolgend eingegangen.

A.1 ".sat"–Datei

- Textdatei, die eine aussagenlogische Formel in konjunktiver Normalform repräsentiert
- besteht aus einer endlichen Menge von Klauseln, wobei jede Klausel in einer Textzeile steht
- jede Klausel hat die Form ([–] <AL–Variable 1>,[–] <AL–Variable 2>, ...)
- das Ende der Formel wird gekennzeichnet durch "."

```
(1)
(-2,-3,-4)
(-5,-6,-7)
(-8,5)
(-8,2)
.
```

$$F = 1 \wedge (\neg 2 \vee \neg 3 \vee \neg 4) \wedge (\neg 5 \vee \neg 6 \vee \neg 7) \wedge (\neg 8 \vee 5) \wedge (\neg 8 \vee 2)$$

Beispiel einer ".sat"–Datei und die sie repräsentierende aussagenlogische Formel. Die Bezeichner der aussagenlogischen Variablen können auch Ziffernfolgen sein. Sie sollten nicht länger sein als 20 Textzeichen, und eine Klausel nicht länger als 100 Zeichen.

A.2 ".plp"–Datei

- Textdatei, die ein aussagenlogisches Programm darstellt
- besteht aus einer endlichen Menge von Programmklauseln, wobei jede Klausel in einer Textzeile steht
- ein Fakt hat die Form <AL–Variable>.
- eine Klausel hat die Form <AL–Variable> :– [–] <AL–Variable>,[–] <AL–Variable>,...
- Klauseln müssen nicht dringend mit "." abgeschlossen werden
- die Bezeichner der aussagenlogischen Variablen sollten mit einem Buchstaben beginnen, ein Bezeichner darf nicht länger als 20 Textzeichen, eine Klausel nicht länger als 150 Zeichen sein

```
nadelbaum :- zapfen.
nadelbaum :- nadeln.
larix :- nadelbaum,-immergruen.
europlaerche :- larix.
```

$$\begin{aligned} \text{nadelbaum} &\leftarrow \text{zapfen} \vee \text{nadeln} \\ \text{larix} &\leftarrow \text{nadelbaum} \wedge \neg\text{immergruen} \\ \text{europlaerche} &\leftarrow \text{larix} \end{aligned}$$

Beispiel einer ".plp"–Datei und die sie repräsentierende Formelmenge.

B. Programmdokumentation

Das induktive Lernen im Neuronalen Netz wurde mit Hilfe des Stuttgarter Neuronale Netze Simulators (SNNS) durchgeführt. Auf dessen Basis und unter Nutzung seines Kernels wurden ein Programm zur Erstellung eines Neuronalen Netzes aus einem aussagenlogischen Programm und ein Programm zur Regelextraktion erstellt. Außerdem wurde ein Werkzeug entwickelt, welches eine Formel in konjunktiver Normalform in ein aussagenlogisches Programm transformiert. Es wurde benötigt, weil die Daten einer Domäne in dieser Form vorlagen. Die Programme sind nicht an die im Beleg beschriebenen Domänen gebunden und sind deshalb vielseitig nachnutzbar. Die in den Programmen verwendeten Dateiformate sind in Abschnitt A beschrieben. Ausführungen über den Simulator SNNS gibt [SNN95].

Programmübersicht	
Programm	*Beschreibung*
sat2plp	Konvertierung einer aussagenlogischen Formel in konjunktiver Normalform in ein äquivalentes aussagenlogisches Programm
plp2snns	Erstellen von semantikgleichen Neuronalen Netzen für SNNS aus einem aussagenlogischen Programm
mofn	Extrahieren von Regeln aus einem SNNS–Netzwerk auf der Basis des MOFN–Algorithmus und Umwandlung der Regeln in ein aussagenlogisches Programm

B.1 sat2plp

Aufgabe:
Das Programm sat2plp erstellt aus einer Textdatei mit einer aussagenlogischen Formel in konjunktiver Normalform ein äquivalentes aussagenlogisches Programm.

Benutzung:
Mit dem Start des Programmes ist der Name der Quelldatei anzugeben, die die Formel enthält. Der Name der Zieldatei für die Ausgabe des Programmes ist optional. Das Programm wird gestartet mit:

> sat2plp <sat-Datei> [<plp-Datei>]

Wird keine Ausgabe–Datei angegeben, so wird eine Datei mit dem Namen der Eingabe–Datei und dem Kürzel ".plp" genutzt.

```
> sat2plp counter.sat

line 1 is no clause
start of formula found in line 2
transformation: counter.sat -> counter.plp

end of formula found in line 302
transformation finished !

>
```

Abbildung B.1: Beispiel für die Ausführung des Programmes sat2plp

Eingabedatei:
Eine Datei vom Typ ".sat" oder gleicher Syntax (siehe A). Zeichenketten vor oder nach der Formel werden ignoriert. Als Formel gilt alles, was mit einer Klauselform beginnt und durch einen Punkt beendet wird.

Ausgabedatei:
Eine Datei vom Typ ".plp", die für die Weiterbearbeitung mit dem Programm plp2snns gedacht ist. Ist eine Datei mit dem angegebenen Namen schon vorhanden, wird sie überschrieben.

Besonderheiten:
Die aussagenlogische Formel liegt in konjunktiver Normalform vor. Aussagenlogische Programme lassen sich jedoch nur aus Formeln in disjunktiver Normalform direkt ableiten. Deshalb wird ein aussagenlogisches Programm erstellt, welches die Negation der gegebenen Formel repräsentiert. Während der Wahrheitswert der Formel angibt, ob die Formel wahr ist, beschreibt das Programm, ob die Formel falsch ist. Die Äquivalenz der beiden Formen bzw. der Wissensgehalt bleibt jedoch derselbe. Für die Darstellung des „Falschheitswertes" der Formel wird im Programm eine aussagenlogische Variable "not_F" eingeführt. Bestehen die aussagenlogischen Variablen nur aus Zahlen, wird ihnen ein Zeichen "v" vorangesetzt.

gegebene Formel	*erstelltes Programm*
(7)	not_F :- -v7.
(-8,-1,2)	not_F :- v8, v1, -v2.
(-9,-1,-3)	not_F :- v9, v1, v3.
(-10,9)	not_F :- v10, -v9.
(-10,8)	not_F :- v10, -v8.
(-11,-1,-2,-3).	not_F :- v11, v1, v2, v3.

Abbildung B.2: Beispiel für die Transformation einer aussagenlogischen Formel in ein Programm mit sat2plp.

B.2 plp2snns

Aufgabe:
Das Programm plp2snns erstellt aus einem aussagenlogischen Programm semantikgleiche Neuronale Netze für SNNS.

Benutzung:
Für die Ausführung des Programmes wird der Name der Datei benötigt, die das aussagenlogische Programm darstellt und optional der Name der zu erstellenden SNNS–Netzwerkdatei. Das Programm wird gestartet mit:

> plp2snns <plp-Datei> [<net-Datei>] [-p]

Wird kein Name für die Netzwerkdatei angegeben, so wird für die Ausgaben der Name der Programmdatei mit den entsprechenden Endungen verwendet. Die Option "-p" kann gewählt werden, um den Ablauf der Transformation auf der Standardausgabe zu protokollieren.

```
> plp2snns botanik1.plp
reading input-file
building network
writing output-files
writing pattern-files
>
```

Abbildung B.3: Beispiel für die Ausführung des Programmes plp2snns

Eingabedatei:
Eine Datei vom Typ ".plp" oder gleicher Syntax.

Ausgabedateien:
plp2snns erstellt verschiedene Dateien, die es ermöglichen, unkompliziert und schnell mit ihnen unter SNNS zu experimentieren. Folgende Dateien werden erzeugt:

- eine Datei "∼.bin.net" mit einem SNNS–Netzwerk aus binären Schwellwerteinheiten. Das Netz wurde erstellt nach dem Modell von [Kal94], bestehend aus gewichteten Verbindungen mit dem Betrag 1 und minimaler Vernetzung zwischen den Schichten.
- eine Datei "∼.min.net" mit einem Netz aus Einheiten mit sigmoider Aktivierungsfunktion. Das Netz ist genauso aufgebaut, wie "∼.bin.net", aber mit dem Wert $W(A_{min})$ nach [GZC96] gewichtet.
- eine Datei "∼.net". Das Netzwerk entspricht dem der Datei "∼.min.net". Zusätzlich wurde es schichtweise vollvernetzt, indem zwischen allen Einheiten zweier benachbarter Schichten mit 0 gewichtete Kanten erstellt wurden, wenn noch keine Verbindung existierte.
- eine SNNS–Konfigurationsdatei "∼.cfg" , welche auf die optimale Darstellung des Netzes für das Training eingestellt ist.
- eine SNNS–Musterdatei "∼.valid.pat". Sie enthält nur gültige Muster (Muster, die dem Operator T_P entsprechen). Für jede Regel im gegebenen Programm wird ein Beispiel erstellt, in dem die Konjunktion der Literale erfüllt ist und ein Beispiel, in dem alle Literale nicht erfüllt sind.
- eine SNNS–Musterdatei "∼.invalid.pat" mit bis zu 64 ungültigen Mustern (Muster, die T_P nicht berechnet). Die Beispiele werden durch Zufall erstellt.

Alle Dateien, die während der Abarbeitung erstellt werden, überschreiben bereits vorhandene namensgleiche Dateien.

Besonderheiten:
Für die Netzwerke werden die Lernfunktion "Std_Backpropagation" und die Update–Funktion "Topological_Order" eingestellt.
Die erstellten Einheiten in der Ein– und Ausgabeschicht erhalten als Namen die ihnen zugeordnete aussagenlogische Variable der Herbrand–Basis.
Der Schwellwert einer Einheit geht unter SNNS unterschiedlich in die Berechnung der Aktivierung ein. Bei Einheiten mit Schwellwertfunktion "Act_Perceptron" wird er von der Netzeingabe des Neurons subtrahiert, bei Einheiten mit sigmoider Aktivierungsfunktion "Act_Logistic" wird er addiert. Um diesen Unterschied auszugleichen, haben die binären Schwellwerteinheiten positive Schwellwerte, die sigmoiden Einheiten dagegen negative Schwellwerte.
Die SNNS–Konfigurationsdatei, die erstellt wird, ist die Kopie einer Datei "std.cfg", die sich im selben Verzeichnis befinden sollte. Wird die Datei mit neuen Einstellungen überschrieben, so erzeugt das Programm dementsprechend andere Konfigurationsdateien.

B.3 mofn

Aufgabe:
Das Programm mofn extrahiert aus einem SNNS–Netzwerk Regeln und erstellt daraus ein aussagenlogisches Programm. Grundlage der Regelextraktion ist der Algorithmus MOFN aus [TS92].

Benutzung:
Zur Durchführung des Algorithmus wird das Netzwerk und die Trainingsmenge benötigt. Der Nutzer muß deshalb beim Start den Namen der Netzwerkdatei und der Musterdatei angeben. Der Name der resultierenden Programmdatei ist optional. Wird er nicht angegeben, wird der Name des Netzes mit der Endung "∼.mofn.plp" verwendet. Das Programm kann gestartet werden mit:

> mofn <net-Datei> [<pat-Datei>] [<plp-Datei>]

oder nur durch

> mofn

und anschließende Eingabe aller Dateinamen.

Eingabedateien: Eine SNNS–Netzwerkdatei ".net" und eine SNNS–Patterndatei bzw. Dateien gleicher Syntax.

Ausgabedateien:

- die resultierende Programmdatei "∼.plp". Sie stellt das extrahierte aussagenlogische Programm dar.
- eine Datei "mofn.protocol", die die einzelnen Schritte des Algorithmus dokumentiert.

Die Dateien, die während der Abarbeitung erstellt werden, überschreiben bereits vorhandene namensgleiche Dateien.

Besonderheiten:
Das Programm extrahiert für jede Einheit in der verdeckten Schicht und in der Ausgabeschicht Regeln nach dem Algorithmus MOFN. Sind diese Regeln extrahiert, werden sie in zusätzlichen Schritten (siehe Abschnitt 4) in ein aussagenlogisches Programm transformiert. Dabei werden die Ausgabeeinheiten im direkten Zusammenhang mit den Eingabeeinheiten dargestellt. In dem resultierenden Programm befinden sich keine Bezeichner von verdeckten Einheiten.
Während der Bearbeitung des Netzes arbeitet das Programm mit den Nummern der Einheiten. Deshalb werden in der Protokollierungsdatei auch diese Identifikationsnummern

verwendet. Das abschließende aussagenlogische Programm stellt die Zusammenhänge jedoch durch die Namen der Einheiten dar, sofern sie existieren.
Der unterschiedliche Einfluß des Schwellwertes in Abhängigkeit von der Aktivierungsfunktion wird wie bei plp2snns durch Vorzeichenwechsel ausgeglichen, wenn die Aktivierungsfunktion nicht "Act_Perceptron" ist.
Das Programm ist in der Lage, aus jedem mehrschichtigen Neuronalen Netz Regeln zu extrahieren. Damit verwertbare Ergebnisse zustande kommen, sollten jedoch gewisse Vorbedingungen erfüllt sein:

- Das Netz sollte so trainiert sein, daß alle Einheiten entweder maximal aktiv oder maximal inaktiv sind.
- Alle Einheiten im Netz sollten eine Aktivierungsfunktion haben, die die Aktivierung auf ein Intervall [0,1] abbildet. Dies sind in erster Linie die Funktionen "Act_Perceptron" und "Act_Logistic".
- Die angegebene Pattern–Datei sollte die Trainingsmenge enthalten, mit der das Netz angelernt wurde.

Unter Umständen kann es passieren, daß wegen der exponentiellen Entwicklung der Anzahl der Regeln ein Auslasten des Speichers eintritt. Der Algorithmus wird dann abgebrochen und eine Fehlermeldung der Art

error occured during transformation –> algorithm is cancelled

erscheint. Der vom Programm belegte Speicher wird daraufhin wieder freigegeben und das Programm beendet. Tritt ein Fehler im Zusammenhang mit einer SNNS–Meldung auf, liegt die Ursache dafür in der Bearbeitung durch den Simulator.

```
> mofn

SNNS 3D-Kernel V4.10
–Extracting rules from Neural Networks–

usage: mofn <NET-file> [<PAT-file>] [<Rule-file>]

Filename of the network file: botanik1.net

Filename of the pattern file: botanik.valid.pat

Filename of the output file: botanik.mofn.plp

Loading the network ...

Network name: botanik1
No. of units : 90
No. of input units : 35
No. of output units: 35
No. of sites: 0
No. of links: 1400
Learning function: Std_Backpropagation
Update function : Topological_Order

Loading the patterns ...

No. of patterns: 40

STEP 1 OF MOFN: CLUSTERING
STEP 2 OF MOFN: AVERAGING
STEP 3 OF MOFN: ELIMINATING
STEP 4 OF MOFN: OPTIMIZING
STEP 5 OF MOFN: EXTRACTING
STEP 6 OF MOFN: SIMPLIFYING

creating rules for negative literals
transforming rules from MOfN to MOfM
substitution of hidden units
transforming the rules to a logic programme

Saving protocol file mofn.protocol
Writing output file botanik1.mofn.plp...
>
```

Abbildung B.4: Beispiel für die Ausführung des Programmes mofn

C. Trainingsverläufe

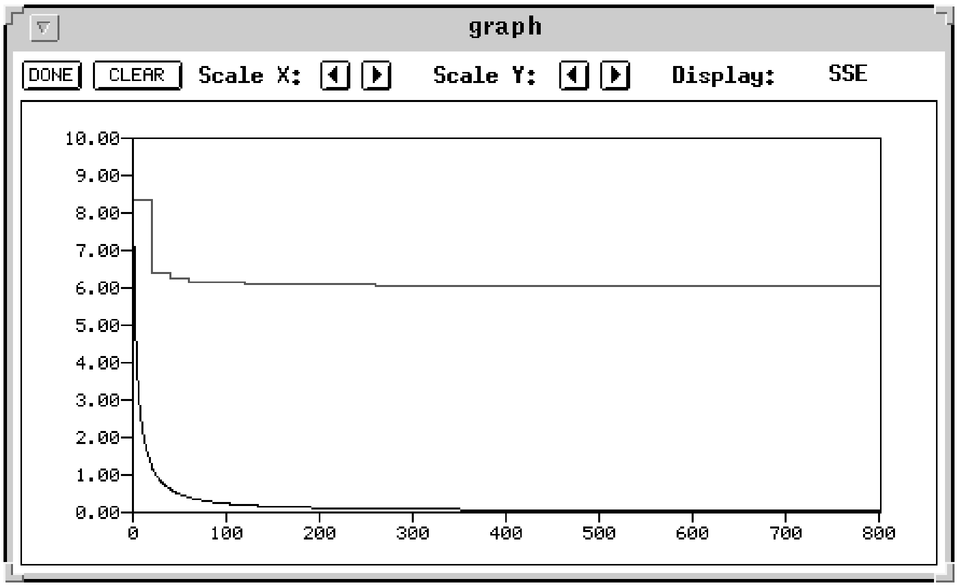
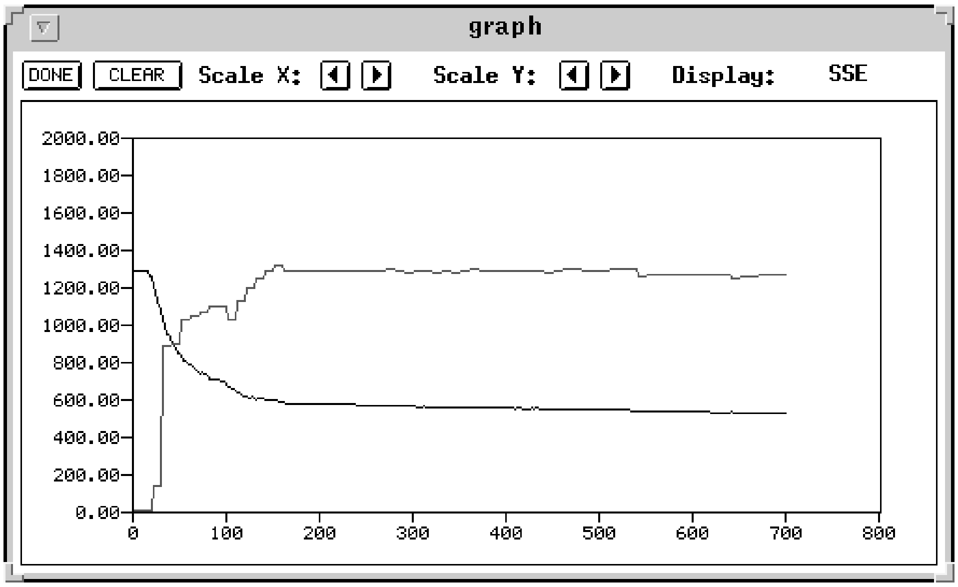


Abbildung C.1: Der Graph zeigt den Verlauf der globalen Fehlerfunktion beim Training in der Domäne der Hardwareverifikation. Das Netz wurde nach dem Übersetzungsalgorithmus mit $W(A_{min} = 0,7)$ erstellt. Der große Fehler der Testmenge zeigt, daß das Netz den Operator T_P nicht korrekt berechnet.



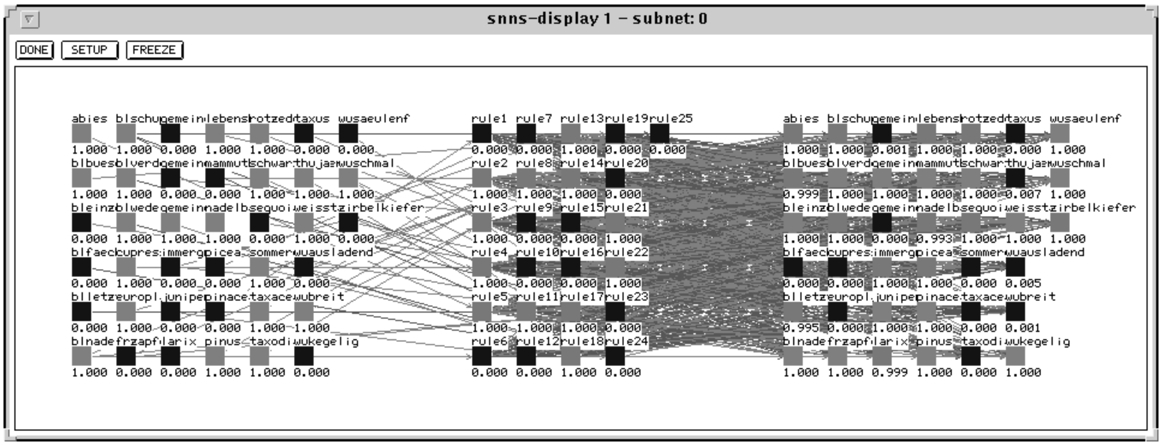
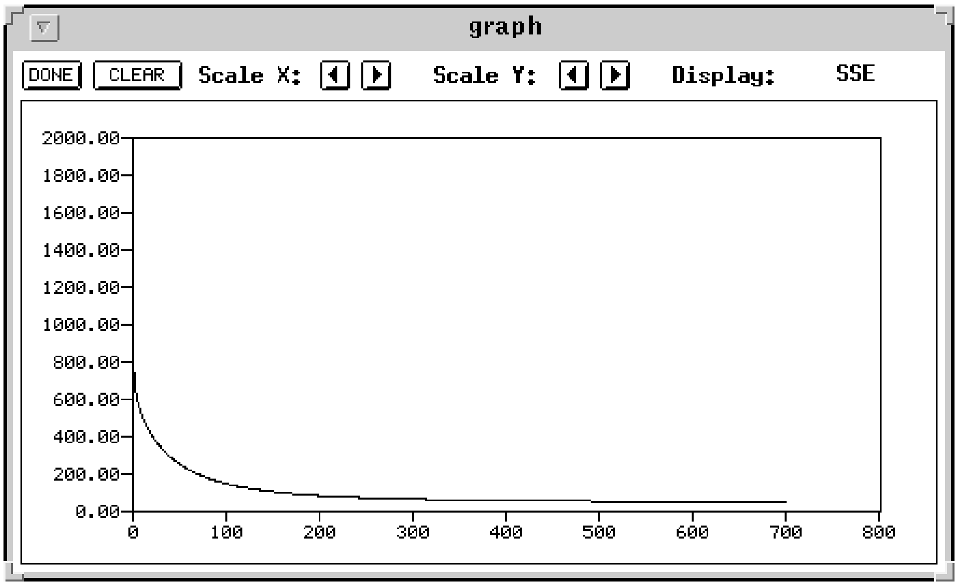


Abbildung C.2: Der Graph zeigt den Verlauf der globalen Fehlerfunktion beim Training einer Menge von 64 zufälligen Beispielen. Das Netz wurde nach dem Übersetzungsalgorithmus erstellt und mit dem Wert $W(A_{min} = 0,9)$ gewichtet. Beim Training wurde die Lernrate $\eta = 10$ verwendet. Bei kleineren Lernraten wurden kaum Lernfortschritte festgestellt. Die Darstellung des Netzes unten verdeutlicht, daß die Gradienten der verdeckten Einheiten fast 0 sind, und nur die Verbindungen zur Ausgabeschicht trainiert werden.



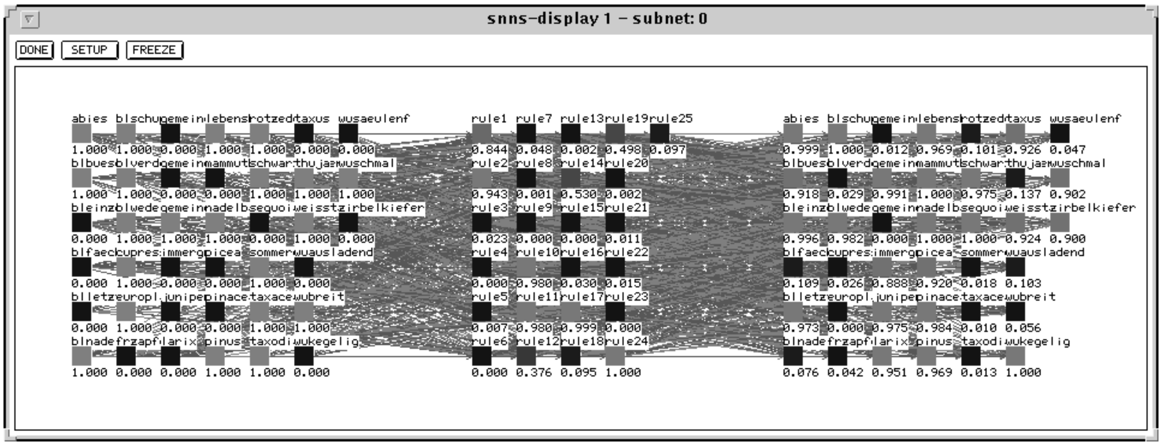
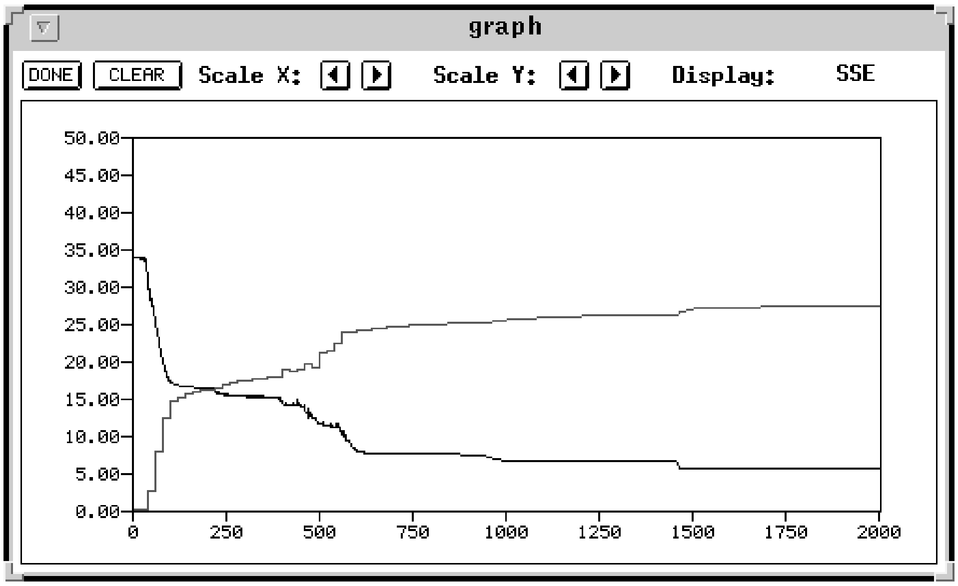


Abbildung C.3: Der Graph zeigt den Verlauf der globalen Fehlerfunktion beim Training derselben Menge von zufälligen Beispielen, wie in Abbildung C.2. Die Gewichte wurden im Intervall $[-1, 1]$ initialisiert und die Lernrate beträgt $\eta = 0,2$. Im Vergleich ist ein wesentlich besseres Lernverhalten zu erkennen. Die Darstellung des Netzes zeigt, daß alle Verbindungen vom Training betroffen sind.



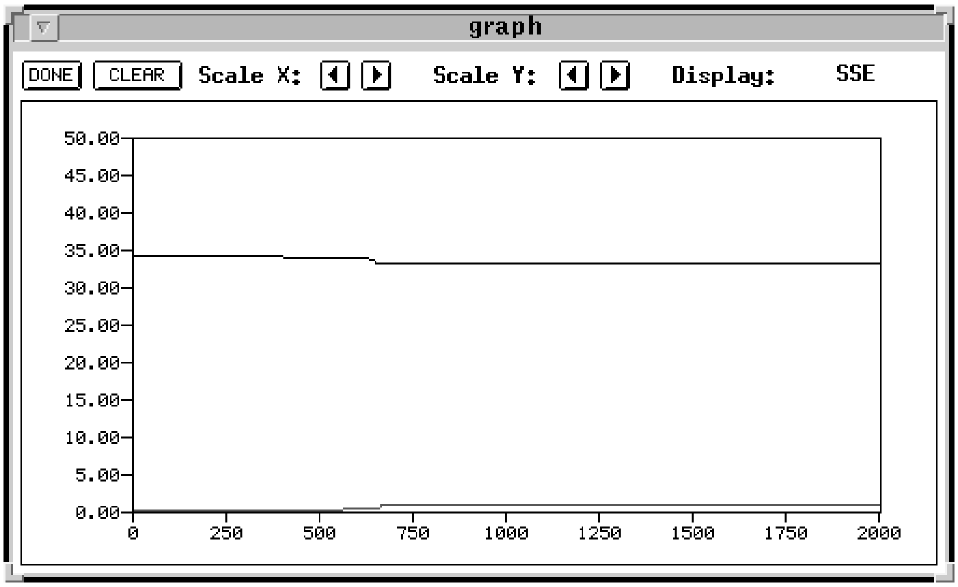


Abbildung C.4: Die Graphen zeigen den Verlauf der globalen Fehlerfunktion beim Training in der Domäne Hardwareverifikation. Das Netzwerk wurde nach einem initialen Programm erstellt. In der oberen Abbildung bestand die Trainingsmenge nur aus generalisierenden Beispielen. Im unteren Teil mußte das Netz nur spezialisierende Beispiele lernen. Die Anzahl der Muster und die Lernrate waren jeweils gleich. Es wird deutlich, daß das Netz die generalisierenden Muster wesentlich schneller lernen kann. Das Netz hat das Bestreben, das ursprüngliche Wissen beizubehalten.

D. Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet, sowie Zitate kenntlich gemacht habe.

Dresden, den 15. September 1997

Jörg Wunderlich

Literaturverzeichnis

[Apt97] Apt, Krzysztof: From logic programming to Prolog / Krzysztof R. Apt, Premtice Hall, 1997

[Bar93] Barnsley, M. F.: Fractals Everywhere / Michael Barnsley, with the assistance of Hawley Rising. Academic Press, 1993

[BHS93] Bibel, Wolfgang: Wissensrepräsentation und Inferenz: eine grundlegende Einführung / Wolfgang Bibel. Zusammen mit Steffen Hölldobler und Torsten Schaub. - Braunschweig; Wiesbaden: Vieweg, 1993

[Bru95] BRUNS: Pflanzen / Sortimentskatalog 95/96

[Coo94] Coombes, Allen J.: Laub– und Nadelbäume / Allen J. Coombes. - deutsche Ausgabe; Ravensburger Buchverlag, 1994

[GZC96] d'Avila Garcez, A. S.; Zaverucha, G.; de Carvalho, L. A. V.: Logic Programming and Inductive Learning in Artificial Neural Networks. / Knowledge Representation in Neural Networks. (pp. 33 - 46); Berlin; Logos Verlag, 1997

[Ede87] Edelsbrunner, Herbert: Algorithms in combinatorial geometry. EATCS monographs on theoretical computer science; v. 10, Springer-Verlag Berlin Heidelberg, 1987

[Fu91] Fu, L.M.: Rule learning by searching on adapted nets. Proceedings of the Ninth National Conference on Artificial Intelligence (pp. 590-595). Anaheim, CA: AAAI Press, 1991

[Hen91] Hennings, Ralf-Dirk: Informations- und Wissensverarbeitung : Theoretische Grundlagen Wissensbasierter Systeme / Ralf-Dirk Hennings. - Berlin ; New York : de Gruyter, 1991

[HKP91] Hertz, John A.: Introduction to the theory of neural computation / John A. Hertz, Richard G. Palmer, Anders S. Krogh. Santa Fe Institute studies in the sciences of complexity. Lecture notes : v. 1, Addison-Wesley, 1991

[JD88] Jain, Anil K., Algorithms for clustering data / Anil K. Jain, Richard C. Dubes. Prentice Hall advanced reference series, 1988

[Kal94] Kalinke, Yvonne: Ein massiv paralleles Berechnungsmodell für normale logische Programme. Diplomarbeit, Dresden, 1994

[Kop96] Kopka, Helmut: LaTeX: Einführung / Helmut Kopka. -2., überarb. Aufl.- Bonn; Reading, Mass. [u.a.] : Addison-Wesley, 1996

[Llo93] Lloyd, J. W.: Foundations of logic programming. Springer-Verlag Berlin Heidelberg, 1993

[LP90] Lowes, Martin: Programmieren mit C : ANSI Standard / von Martin Lowes und Augustin Paulik. -Stuttgart : Teubner, 1990

[Miy90] Miyamoto, Sadaaki: Fuzzy sets in information retrieval and cluster analysis / Sadaaki Miyamoto. Theory and decision library. Kluwer Academic Publishers, 1990

[NKK96] Nauck, Detlef; Klawonn, Frank; Kruse, Rudolf: Neuronale Netze und Fuzzy-Systeme / Braunschweig ; Wiesbaden : Vieweg, 1996

[Rei94] Reichel, H.: Logik für Informatiker. Vorlesungsskript, 1994

[RN95] Russell, Stuart J.; Norvig, Peter: Artificial intelligence: a modern approach / Stuart Russell, Peter Norvig. Prentice-Hall, Inc., 1995

[Sch89] Schildt, Herbert: Turbo-C-Befehlsbibliothek / Herbert Schildt. -Hamburg; New York: McGraw-Hill, 1989

[SNN95] SNNS - Stuttgart Neural Network Simulator, User Manual, Version 4.1, 1995

[Tar55] Tarski, A.: A lattice–theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, 5:285–309, 1955

[TS92] Towell, Geoffrey G.; Shavlik, Jude W.: Extracting Refined Rules from Knowledge-Based Neural Networks. Machine Learning, Vol 131, (pp 71-101); 1993

[vLe90] van Leeuwen, Jan: Handbook of theoretical computer science / editor, Jan van Leeuwen. v. A. Algorithms and complexity, Elsevier Science Publishers B.V., 1990