Friedrich–Alexander–Universität Erlangen–Nürnberg
Lehrstuhl für Künstliche Intelligenz
Prof. Dr. Dr. H. Stoyan
Prof. Dr. G. Görz

Integration von neuronalen Netzen in mathematische Prozeßmodelle mit objekt-orientierten Methoden

Diplomarbeit im Studienfach Informatik
vorgelegt von

Markus Gälli

Betreuer:
Prof. Dr. Günther Görz
Lars Kindermann

Beginn der Arbeit:	1. Februar 1996
Abgabe der Arbeit:	1. August 1996

Erklärung

Ich versichere, daß ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, Juli 1996

Zusammenfassung

Ein erfolgreicher Einsatzschwerpunkt neuronaler Netze im industriellen Bereich liegt in der Simulation und Optimierung komplexer Fertigungsprozesse. Es wird in diesem Fall die Möglichkeit untersucht, neuronale Netze in vorhandene hochentwickelte mathematisch-physikalische Modelle eines Walzvorgangs zu integrieren. Deren Lernfähigkeit erlaubt es, eine Vielzahl schwer erfaßbarer Einflüsse der Anlage auf die Qualität des Produkts zu berücksichtigen.

Als Werkzeug für die Integration neuronaler Netze und mathematischer Modelle wurde DOLPHIN (Distributed Object-oriented Local Programable HybrId Network) entwickelt.

In diesem modularen, Smalltalk-basierten System liegen neuronale Komponenten sowie mathematische Standardfunktionen als verknüpfbare Objekte vor, welche als Gesamtsystem wie ein klassisches Backpropagationnetz trainierbar sind. Das Wissen um die Gleichheit von Subsystemen der zu modellierenden Anlage kann in das neuronale Netz als weiteres Vorwissen eingebracht werden.

Inhaltsverzeichnis

1 Einleitung

Künstliche neuronale Netze werden immer häufiger dazu verwendet, komplexe Abläufe, deren Parameter nicht alle bekannt sind oder für die sich noch kein exaktes physikalisch-mathematisches Modell gefunden hat, zu simulieren. Aufgabe dieser Arbeit war es, ein Simulationswerkzeug zu programmieren, das es ermöglicht, unzureichendes aber dennoch vorhandenes Wissen über physikalisch-mathematische Eigenschaften eines zu simulierenden Systems mit der Adaptionsfähigkeit künstlicher neuronaler Netze zu verbinden. Die Netze, die mit Hilfe dieses Simulators erzeugt werden, sollten in der Lage sein, Vorwissen aufzunehmen, sich aber insgesamt wie ein klassisches Backpropagationnetz trainieren zu lassen.

1.1 Die Anforderungen an DOLPHIN

Folgende Aspekte wurden bei der Architektur des Simulators DOLPHIN (=**D**istributed **O**bject-oriented **L**ocal **P**rogramable **H**ybr**I**d **N**etworks) berücksichtigt.

1.1.1 Beliebige Aktivierungsfunktionen

Es besteht der Verdacht, daß sich bisher bestehende Systeme häufig an der Simulation biologischer neuronaler Netze und weniger an der Simulation mit künstlichen neuronalen Netzen orientieren. So wurde weder ein kommerzielles noch ein frei zugängliches Softwaresystem gefun-

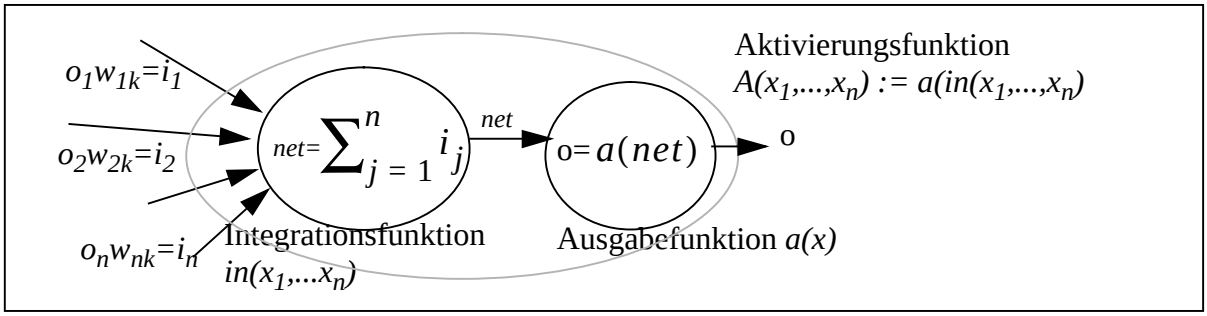


Abb. 1.1 Das klassische Modell eines Neurons N_k

den, das eine gültige Verallgemeinerung des Backpropagation-Algorithmus benutzt, um mit beliebigen differenzierbaren Funktionen einen Gradientenabstieg durchzuführen. In der

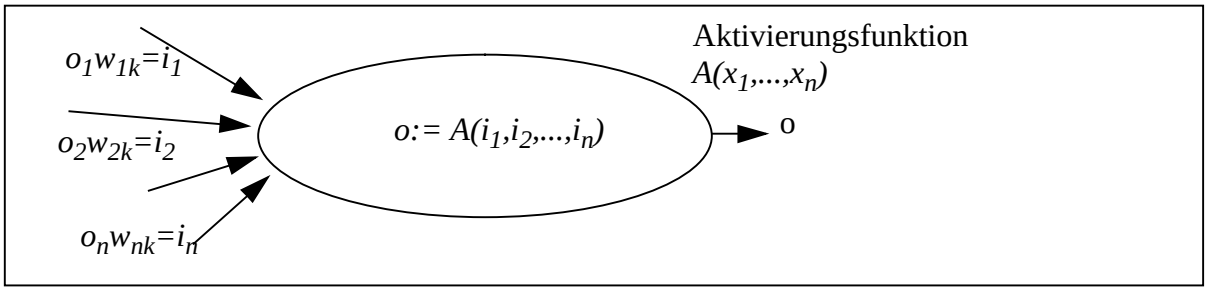


Abb. 1.2 Das Modell eines Neurons N_k in DOLPHIN

Standardliteratur zu neuronalen Netzen [RuCl] [Roj] [RMS] wird Backpropagation nur für eine eingeschränkte Menge an differenzierbaren Ausgabefunktionen $a(net)$ erklärt, so daß in dieser Arbeit eine Herleitung eines allgemeineren Verfahrens für beliebige Aktivierungsfunktionen mehrerer Veränderlicher $A(i_1,...,i_n)$ gegeben wird.

1.1.2 Kompatibilität und Erweiterbarkeit durch Objektorientierung

Ein konfigurierbares Werkzeug wie DOLPHIN kann man auch als eine Art Laborumgebung bezeichnen, in der das Experimentiergerät von hoher Kompatibilität und Erweiterbarkeit und damit Wiederverwendbarkeit sein sollte. Wiederverwendbarkeit ist aber eines der Hauptargumente für objektorientierte Programmierung, so daß man sich für dieses Programmierparadigma entschied.

Hohe Kompatibilität erleichtert sowohl dem Programmierer die Arbeit, da dieser zwar allgemeiner, aber dafür weniger programmieren muß, als auch dem Benutzer den Umgang mit dem System, da er nicht für jedes Modul eine neue Bedienung zu lernen hat und für ihn die Funktionalität des Simulators somit durchschaubar bleibt.

Eine datenfluß-orientierte Sichtweise [Lin] eines datenverarbeitenden Systems erzeugt hohe Kompatibilität unter den einzelnen Komponenten. Dazu ist es nötig, daß die Ausgangsdaten einer Komponente die Eingangsdaten der nächsten Komponente darstellen. Beim klassischen Backpropagationalgorithmus [RuCl] wird aber ein Unterschied zwischen Eingangs-, Ausgangs- und versteckten Schichten gemacht. Die Daten, die an einen Eingang zurückpropagiert werden, sind nicht die gleichen, die von einem Ausgang gelesen werden können. Es wurde in Kapitel 3.5 der Versuch unternommen, einen einzigen Baustein zu finden, aus dem sich Backpropagationnetze zusammensetzen lassen, mit Hilfe dieses Elementes könnten dann leicht verschiedene Backprop-Netze hintereinandergeschaltet werden.

Erweiterbarkeit spielt in einem laborartigen Umfeld eine große Rolle, so entstehen auch erst durch die Arbeit mit dem Experimentierbaukasten neue Beschreibungsversuche des zu simulierenden Modells. Beispielsweise konnte eine erst im Laufe der Arbeit entstandene Idee zur besseren Modellierung eines technischen Prozesses durch „Shared Weights“ schnell in DOLPHIN integriert werden.

1.1.3 Interaktivität und Visualisierung

In einem Labor soll man experimentieren können. Dazu ist es notwendig, in das Versuchsgeschehen interaktiv eingreifen zu dürfen und jegliche Versuchsparameter einsehen, visualisieren und auch verändern zu können. Idealerweise sollte man sich noch während eines Versuchs neue Werkzeuge basteln können. Der interaktive Charakter DOLPHINs wurde mit Hilfe der rein objektorientierten Entwicklungsumgebung Smalltalk erreicht, da dort der compile/run/debug Zyklus objektorientierter Compiler-Sprachen und die damit verbundene Trennung von Programmierung und Benutzung, und somit auch von Programmierer und Benutzer, entfällt und man ein System erhält, in welchem auf alle Objekte zur Laufzeit zugegriffen werden kann. Da in Smalltalk selbst Klassen Objekte sind, können auch diese während des Experimentes geändert wer-

den. Der selbstreflexive Charakter des Smalltalk-Systems erlaubt so beispielsweise die Änderung einer Klasse einer Aktivierungsfunktion durch den Benutzer, die sich dann sofort auf alle Objekte, die diese Funktion momentan benutzen, auswirkt.

Zusätzlich bietet der Interpreter-Charaker Smalltalks eine „kostenlose“ Skriptsprache, die bei geschickter Bezeichnung der Klassen und Methoden durchaus Ähnlichkeit zu natürlicher Sprache hat und somit leicht zu erlernen ist.

DOLPHIN sollte seinen momentanen Zustand auch auf vielerlei Arten visualisieren. So sollte das Netz 1:1 mit einem graphischen Editor erzeugt werden können, in welchem auch zur Laufzeit eines Trainings Netzeinheiten ein- oder ausgehängt werden können sollten. Ausgewählte Variablen dieser Einheiten sollten sofort im Editorfenster sichtbar gemacht werden können. Dies wurde beispielsweise durch Einfärbung der Kanäle entsprechend ihrer Gewichte und auch durch direkte Darstellung ihrer Gewichtsgröße erreicht. Prinzipiell sollten alle Variablen des Netzes in DOLPHIN mittels der `Inspector`-Klasse des Smalltalk-Systems jederzeit anzeigbar und änderbar sein. Die Trainingswerte und die Ausgabewerte aller Neuronen sollten in einer Tabelle ausgegeben werden, die Werte dieser Tabelle sollten untereinander in Abhängigkeit gebracht und in einer 2-dimensionalen Kurve dargestellt werden können. Ebenso sollte die Fehlerkurve des Lernvorgangs angezeigt werden. Der Zustand eines Ausgangsneurons sollte mittels eines rezeptiven Feldes in Abhängigkeit des Netzes beschrieben werden können.

1.2 Problemgebiet Walzstraße

Diese Anforderungen ergaben sich aus einem am FORWISS bearbeiteten Projekt AENEAS in Zusammenarbeit mit der Siemens AG[MPGS]. Im Folgenden soll das Modell eines Walzgerüstes skizziert, das genaue Simulationsziel erläutert und Lösungswege, die sich durch die Hybridität DOLPHINs realisieren lassen, aufgezeigt werden.

1.2.1 Modell eines Walzgerüstes

Eine Walzstraße besteht aus mehreren, hintereinandergeschalteten Gerüsten, die sukzessiv ein einlaufendes Metallband pressen. Die Kraft, mit welcher jedes einzelne Gerüst das Band bear-

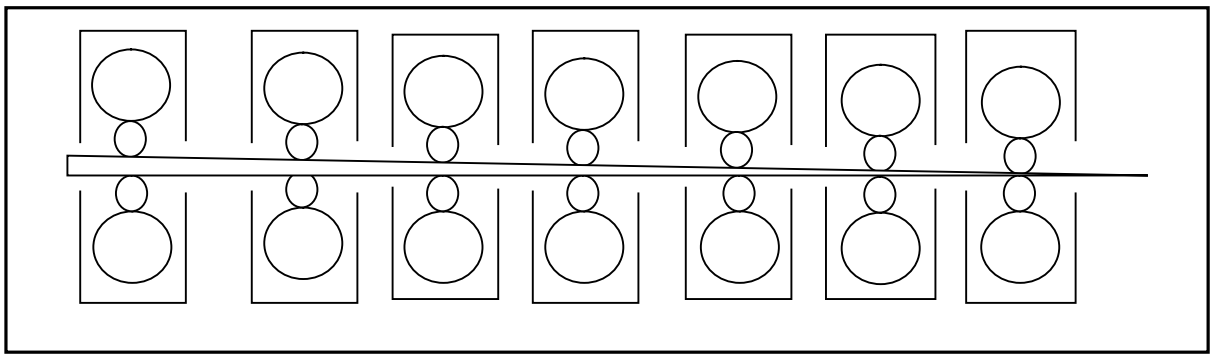


Abb. 1.3 Schema einer Walzstraße mit sieben Gerüsten

beitet, kann eingestellt werden, so daß eine gewünschte Ausgangsdicke des Bandes durch viele

verschiedene Konfigurationen der Walzstraße erreicht werden kann. Zusätzlich zur Ausgangsdicke interessiert neben anderen hier vernachlässigten Größen vor allem auch das sogenannte Profil des Bandes, das ebenfalls abhängig von der Einstellung der Walzkraft jedes Gerüstes ist. Entscheidend für das Ausgangsprofil des Bandes sind auch die sogenannten Walzspaltprofile der einzelnen Gerüste, die die Geometrie der Walzen beschreiben.

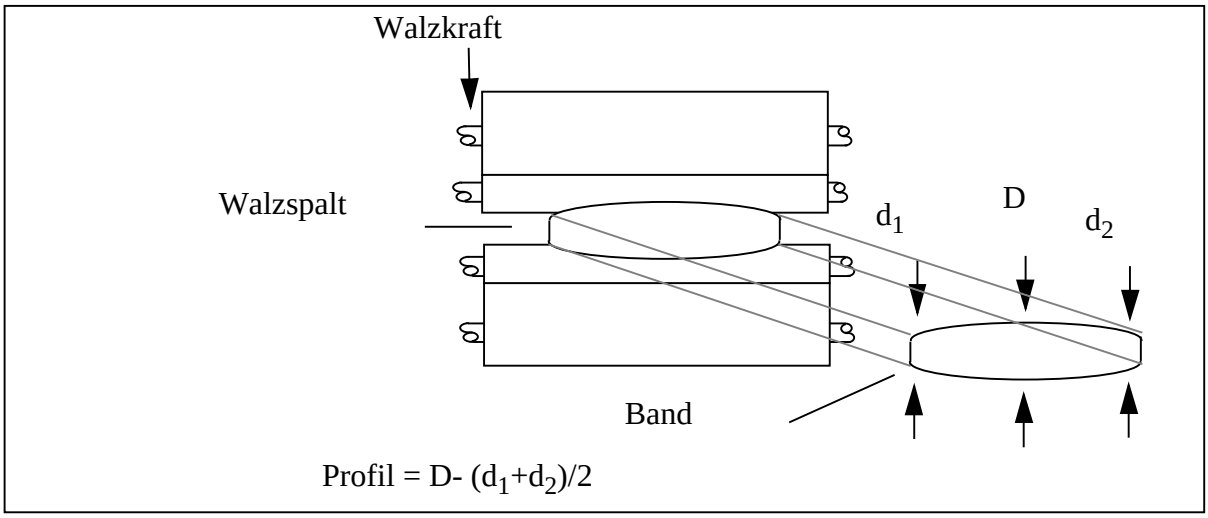


Abb. 1.4 Schema eines Walzgerüstes

1.2.2 Simulationsziel

Gesucht ist eine Einstellung der Walzkräfte, die die Walzstraße Bänder einer gewünschten Ausgangsdicke und eines bestimmten Profils produzieren läßt. Als Zwischenschritt dazu wäre es wichtig, das Verhalten eines einzelnen Walzgerüstes zu kennen. Dies ist aber nicht direkt meßbar, da auf Grund der hohen Temperatur des Bandes keine Meßgeräte zwischen den einzelnen Gerüsten angebracht werden können. Man hat jedoch ein mathematisches Modell in Form einer Funktion mit einem noch unbekannten Parameter k, wie das Ausgangsprofil vom Eingangsprofil und dem Walzspaltprofil eines Gerüstes abhängt. Der Parameter k selber hängt von physikalischen Kenngrößen wie der Höhe des Walzspaltes und der Breite des Bandes ab.

1.2.3 Annäherung des Simulationsziels durch die Hybridität DOLPHINs

Man hat somit bereits Vorwissen über das zu simulierende Modell, das in einer differenzierbaren „Walzfunktion“ festgehalten wird. Man kennt das Eingangsprofil des Bandes am Anfang und das Ausgangsprofil des Bandes am Ende der Walzstraße, die Breiten und Dicken des Bandes an den einzelnen Gerüsten und hat eine Größe für das Walzspaltprofil (Abbildung 1.5). Die kleinen Netze sollen den Parameter k möglichst gut einstellen, so daß man eine Beschreibung des Ein- Ausgabeverhaltens der einzelnen Gerüste erhält.

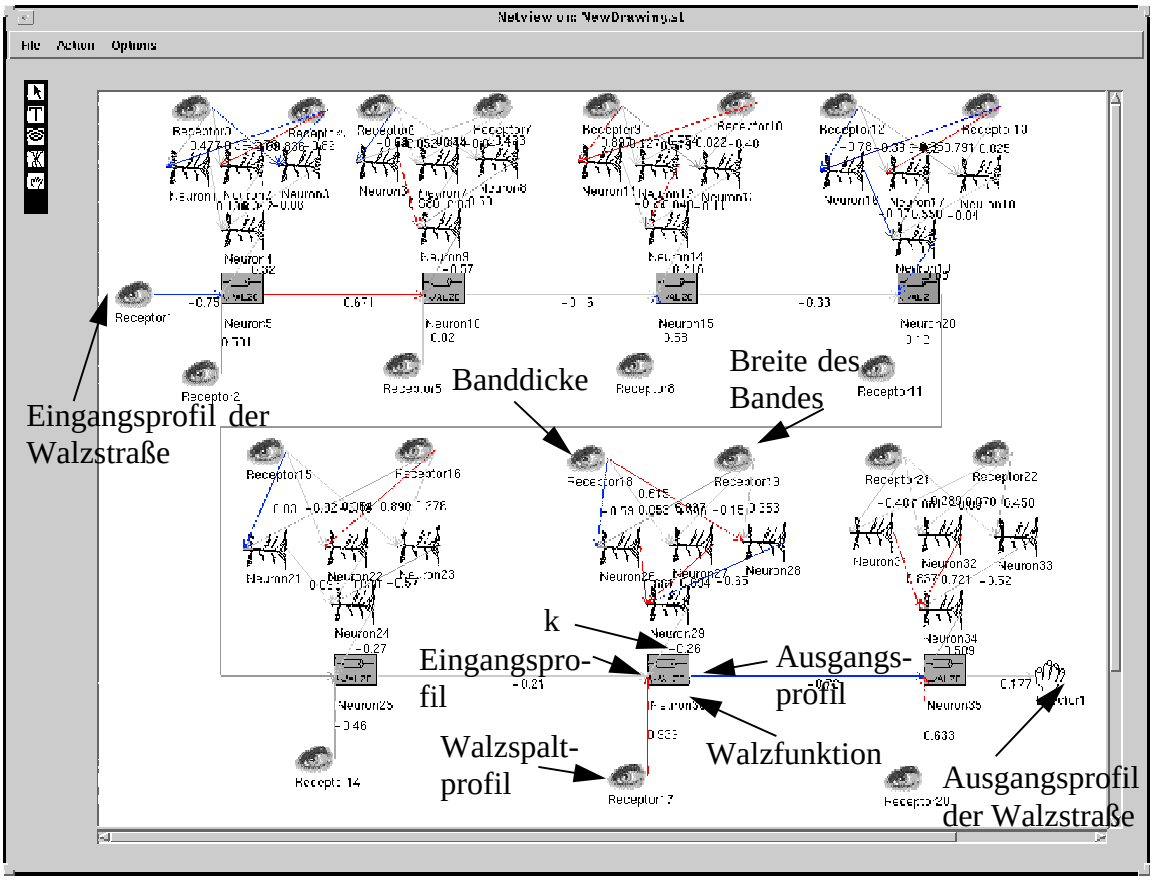


Abb. 1.5 Das DOLPHIN-Netz zur Simulation eines Walzwerkes

2 Grundlagen

Es soll hier kurz auf die Grundlagen von neuronalen Netzen und objektorientierter Programmierung, dort insbesondere auf das Entwicklungssystem Smalltalk eingegangen werden.

2.1 Neuronale Netze

2.1.1 McCulloch Pitts-Neuron

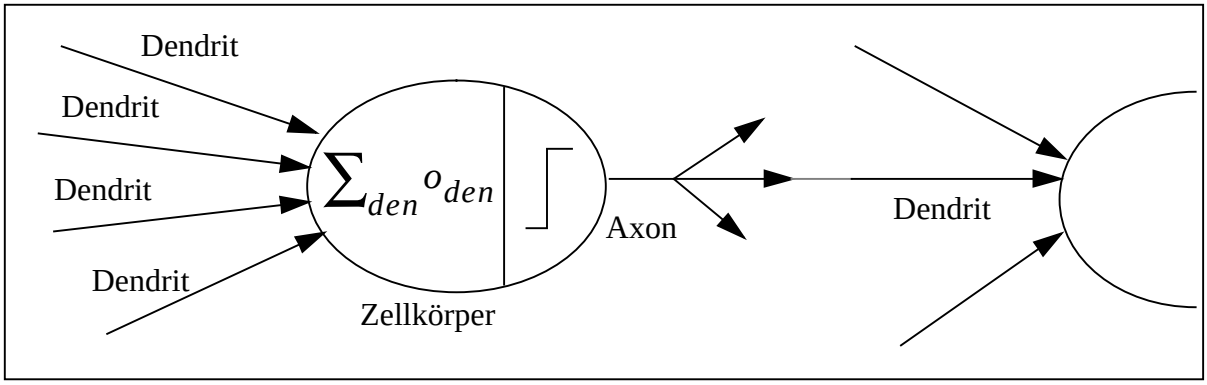


Abb. 2.1 Modell eines Neurons nach McCulloch Pitts

Nach der Vorstellung von McCulloch und Pitts [McP] besteht ein (biologisches) Neuron im wesentlichen aus Dendriten, einem Zellkörper und einem Axon. Die Dendriten stellen dem Neuron die Eingabewerte in Form von Spannungsänderungen zur Verfügung, das Neuron feuert, je nachdem, ob es einen gewissen Grenzwert seines Eingangspotentials überschreitet, ein Signal an sein Axon, welches sich wiederum verzweigen kann und damit als Dendrit für seine angrenzenden Neuronen wirkt.[1] Sie bauten aus dieser Vorstellung ein erstes mathematisches Modell eines Neurons, das allerdings nur Binärwerte als Ausgabe hatte. Die Eins als Ausgabe stand für „Feuer“ und die Null für „Kein Feuer“. Es wurde bewiesen, daß mit einem Netz, das aus solchen McCulloch Pitts-Neuronen besteht, jeder endliche Automat nachgebaut werden konnte, was als Indiz für die Richtigkeit dieser Vorstellung von biologischen Neuronen gewertet wurde.

2.1.2 Multilayer-Perzeptron

Das Modell, das mit am bedeutungsvollsten für die weitere Entwicklung künstlicher neuronaler Netze war, ist das sogenannte Perzeptron [Ros]. Es hat wie das künstliche „Urneuron“ von McCulloch und Pitts eine Treppenfunktion als Ausgabefunktion, erweitert es aber um gewichtete Kanäle. Durch Änderung der Gewichte in diesen Kanälen kann sich das Perzeptron adaptieren, was gegenüber den statisch verdrahteten McCulloch Pitts-Neuronen einen Vorteil darstellte.

1. Die Verbindung zwischen zwei Neuronen, die ansonsten je nach Standpunkt als Axon oder als Dendrit bezeichnet würde, wird im weiteren Kanal genannt werden.

Minsky und Papert [Min] zeigten die prinzipielle Beschränktheit des einschichtigen Perzeptrons. Insbesondere fand man, daß das einschichtige Perzeptron nur sogenannte linear separierbare Funktionen lernen kann. Linear separierbar sind Funktionen $f \in F((\Re^n) \rightarrow \{0,1\})$, deren Ausgaberaum durch eine (*n-1*)-dimensionale Ebene in zwei Räume geteilt werden kann, in welchen jeweils nur die Null oder nur die Eins als Ausgabewerte vorkommen. Für *n=2* wurde die XOR-Funktion als Beispiel für eine nicht linear separierbare Funktion angegeben, die somit nicht von einem einschichtigen Perzeptron gelernt werden kann. Erst das Anhängen einer verborgenen und einer Ausgangsschicht macht das Perzeptron zu einem universellen Funktionsapproximator [Gal]. Ein Perzeptron mit beliebig vielen Schichten nennt man Multilayer-Perceptron oder kurz MLP.

2.1.3 Backpropagation

Der momentan in technischen Anwendungen verbreitetste Algorithmus zur Adaption der Gewichte bei MLP ist Backpropagation (kurz: BP). Mit seiner Hilfe wird ein Gradientenabstiegsverfahren in einem mehrschichtigen Funktionennetz unter Ausnutzung der Kettenregel implementiert. Bei Gradientenabstiegsverfahren werden Funktionen mit freien Parametern minimiert oder maximiert, indem sie nach diesen freien Parametern abgeleitet werden, und dann der jeweilige Parameter ein Stück in die so errechnete Richtung verschoben wird.
BP sucht das Minimum einer Fehlerfunktion durch Abstieg in der Gradientenrichtung [Roj]. Die Fehlerfunktion bildet im allgemeinen die quadrierte Differenz zwischen den Ausgaben des Netzes an einer Stelle und den gewünschten Ausgaben des Netzes an dieser Stelle. Bei den freien Parameter von BP-Netzen handelt es sich im Allgemeinen um die Gewichte der Kanäle, die die künstlichen Neuronen verbinden. BP wurde in den siebziger Jahren von mehreren Forschern gleichzeitig entwickelt, und wurde dann durch [RuCl] populär.

Die künstlichen Neuronen, die in einem BP-Netz verwendet werden, gingen aus dem Modell des Perzeptrons hervor und benutzten deshalb meistens die Summation als Integrations- und eine *squashing-function* als Ausgabefunktion. Eine Darstellung, wie und warum BP auch auf anderen Funktionen angewendet werden kann, findet sich in Kapitel 3.1 auf Seite 16.

Die Neuronen in einem BP-Netz geben kontinuierliche Werte aus, deswegen hat ihre *squashing-function* im Gegensatz zu den Ausgabefunktionen des McCulloch Pitts-Neurons und des Perzeptrons einen kontinuierlichen Wertebereich. Eine typische konitinuierliche *squashing-function*, die man auch Sigmoide nennt, ist in Abbildung 2.2 zu sehen.

Die Gewichte können in zwei Momenten mit den vom BP-Algorithmus ausgerechneten Gewichtsänderungen aufgefrischt werden, zum einen dann, wenn ein Input-Output-Paar trainiert wurde, was man als Online-Training oder Pattern-Update bezeichnet, zum anderen dann, wenn alle Input- Outputpaare einer vorgegebenen Trainingsdatenmenge eingespeist wurden, hier spricht man von Batch-Update. Beim Batch-Update werden die Gewichtsänderungen eines Kanals aufsummiert und erst nach Durchlauf aller Daten wird diese Änderung auf das Gewicht des entsprechenden Kanals addiert.

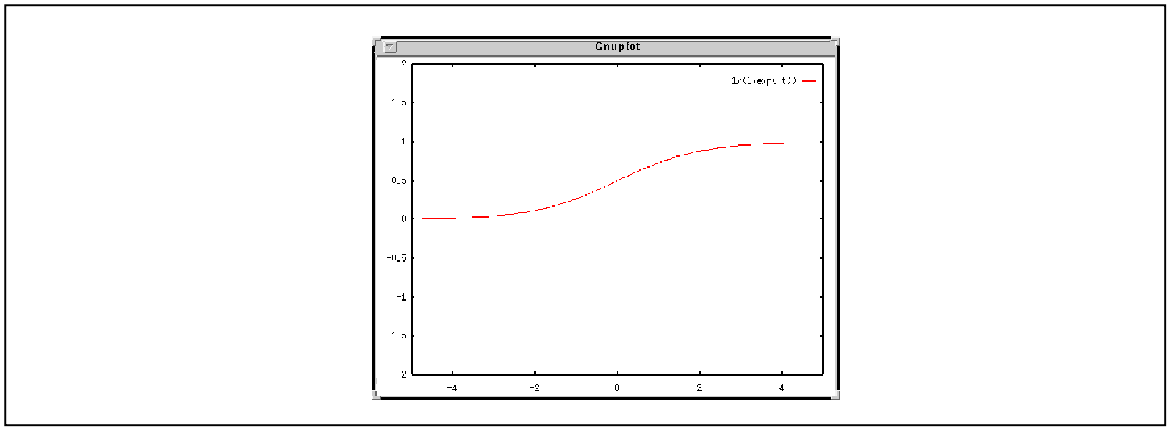


Abb. 2.2 Eine sigmoide Squashingfunction $\frac{1}{1+e^{-t}}$

2.2 Objektorientierte Programmierung

2.2.1 Definition

> „Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.“[Boo]

2.2.2 Ziele objektorientierter Programmierung

Objektorientierte Programmierung hilft, Kosten zu senken, indem sie auf lange Sicht Entwicklungszeiten spart. Die Entwickler müssen zwar in objektorientierter Programmierung ausgebildet werden, dieser Lernaufwand scheint sich aber in deutlich erhöhter Produktivität im Vergleich mit prozeduralen Sprachen niederzuschlagen.

2.2.3 Wiederverwendung

Der wichtigste, weil zeitsparendste Faktor, objektorientierter Programmierung ist die Möglichkeit der Wiederverwendung. Wiederverwendet werden Klassen, Komponenten, Entwurfsmuster und Frameworks.

2.2.3.1 Klassen

Eine Klasse ist eine Menge von Objekten, die die gleiche Struktur und das gleiche Verhalten haben [Boo]. Ein Element dieser Menge nennt man Instanz. Alle Instanzen einer Klasse verstehen die gleichen Nachrichten, durch welche sie ihre Zustände ändern und angeregt werden können, selber Nachrichten an andere Instanzen zu senden. Klassen werden bei sogenannter einfa-

cher Vererbung hierarchisch in einem Klassenbaum angeordnet. Dann hat bis auf die Wurzeln dieses Klassenbaumes dabei jede Klasse genau eine Vaterklasse. Dabei können die Instanzen einer Klasse alle Methoden verstehen, die in einem ihrer Vorfahren implementiert ist, insofern keine speziellen Schutzmechanismen dieses verhindern.

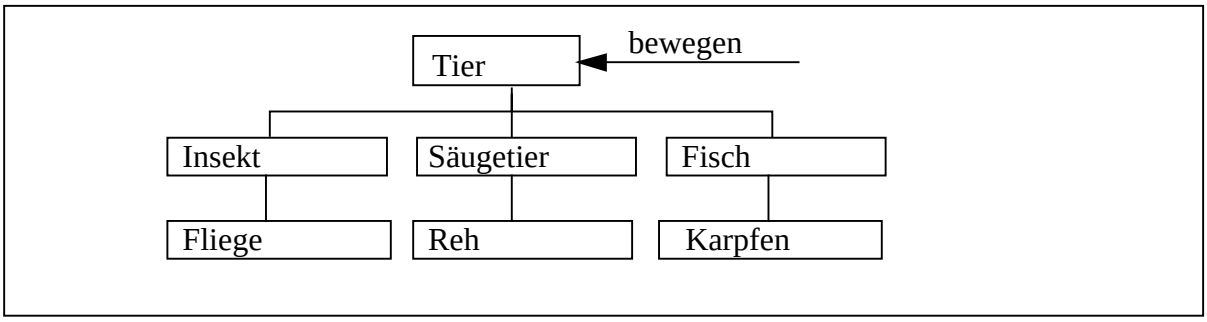


Abb. 2.3 Einfache Vererbung

Die Klassenhierarchie muß aber nicht streng baumartig aufgebaut sein. Dann kann eine Klasse mehrere Väterklassen besitzen und somit von verschiedenen Ahnenlinien erben. So können von

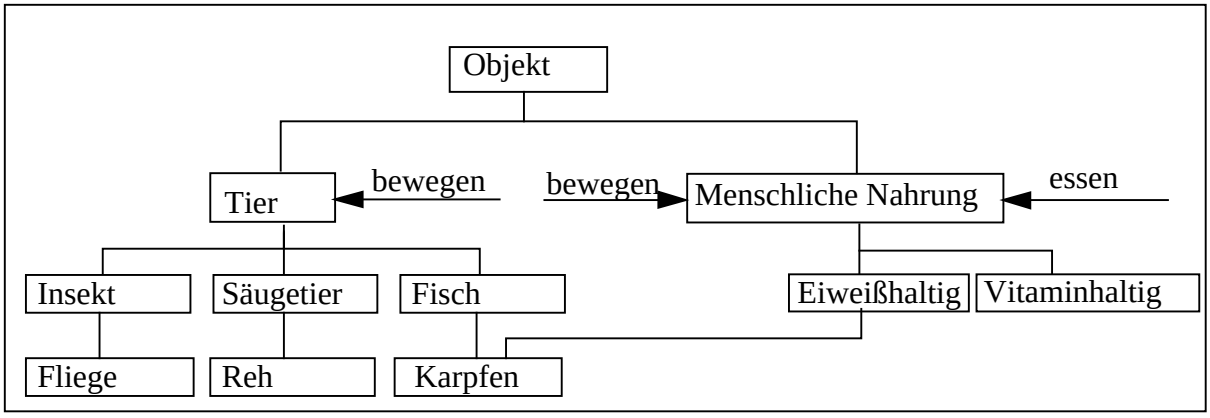


Abb. 2.4 Mehrfachvererbung

einer Klasse prinzipiell sämtliche Methoden anderer Klassen wiederverwendet werden. Hierbei treten allerdings zusätzlich zu einer größeren Unüberschaubarkeit des Klassennetzes zwei schwerwiegende Probleme auf. Zum einen kann es zu einem Namenskonflikt kommen. Wenn beispielsweise in Abbildung 2.4 jedem Objekt eine Instanzvariable ’Zweck’ zugeordnet werden soll, die beim Tier mit ’Zweckfrei’ und bei der menschlichen Nahrung mit ’Selbsterhaltung des Menschen’ gefüllt werden soll, stellt sich die Frage nach dem Zweck eines Karpfens.

Zweitens kann sowohl einem Tier als auch der menschlichen Nahrung die Nachricht ’bewegen’ geschickt werden, so daß hier unklar ist, ob der Karpfen seine Position schwimmend oder auf einem Teller verändern soll. Terry Montlick beschreibt in [Mon], wie sich ohne Mehrfachvererbung in Smalltalk sogenannte MixIn-Klassen [GHJV] realisieren lassen. Diese Information kam für mich zu spät, als daß ich MixIn-Klassen in DOLPHIN benutzen konnte.

2.2.3.2 Komposition

„Favor object composition over class inheritance“ [GHJV]

> „A system should be built wih a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework“ (Ingalls booch S. 123)

Ein System kann durch Komposition bereits vorhandener Teile erzeugt werden. So ist eine Bibliothek eine Ansammlung von Büchern und nicht eine spezielle Art von Buch. Ein Telephon ist nicht ein spezielles Mikrophon sondern besitzt ein Mikrophon. Der Vorteil von Komposition gegenüber von Vererbung liegt darin, daß das Verhalten eines Objektes auch zur Laufzeit verändert werden kann, indem es aus neuen Komponenten zusammengesetzt wird. Somit ist der Benutzer des Programms derjenige, der eine größere Flexibilität erhält, sofern es ihm gestattet wird, die Objekte selber zu komponieren.

2.2.3.3 Entwurfsmuster

Die Definition von Entwurfsmuster (engl. „design pattern“) stammt von einem Architekten, wird jedoch in der Standardlektüre zu „Design Patterns“ [RHJV] als unbedingt auf objektorientierte Methodik übertragbar angesehen:

> „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way, that you can use this solution a million times over, without ever doing it the same way twice.“ (Christopher Alexander)

Ein Entwurfsmuster besteht aus vier wesentlichen Elementen,

- dem Name des Musters, um mit anderen Entwicklern eine gemeinsame Sprache zu sprechen.
- dem Problembereich, der die Fälle beschreibt, in welchen das Muster angewendet werden soll.
- der Lösung, die auf abstraktem Niveau beschreibt, wie mit Hilfe dieses Musters das Problem in den Griff zu bekommen ist.
- den Konsequenzen, die sich aus der Anwendung des Entwurfsmusters ergeben.

Im Folgenden sollen die zwei für die Entwicklung von DOLPHIN zentralen Muster kurz beleuchtet werden. (Im weiteren Verlauf der Arbeit soll zur Erläuterung von Objekt- und Klassen-

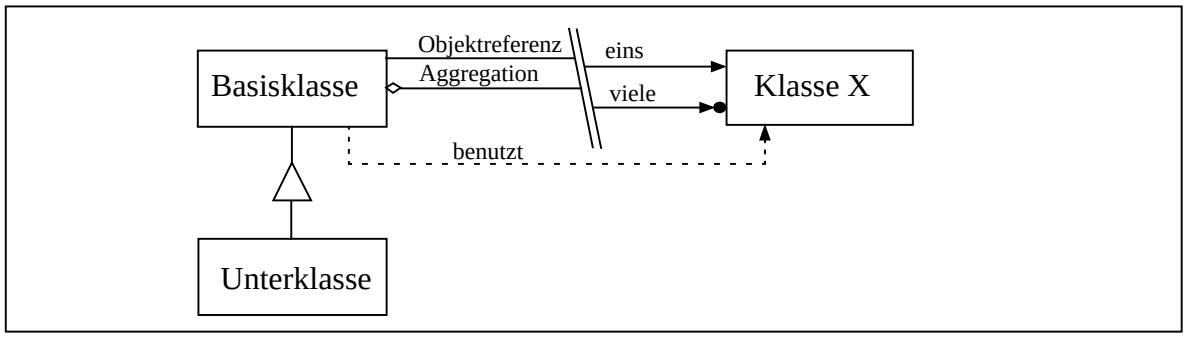


Abb. 2.5 Die graphische Notation

beziehungen die Notation aus [RHJV] wie in Abbildung 2.5 übernommen werden. Alle Objektbezeichnungen und Methodennamen werden außerhalb der graphischen Notation ab sofort als `Codetext` gesetzt.)

2.2.3.3.1 Observer

In Abbildung 2.6 ist eine vereinfachte Beschreibung eines Observers mit Smalltalk-Syntax angegeben. Dieses Muster wird dann benutzt, wenn man eine möglichst lose Verbindung zwischen einem Beobachter (=Interessent) und einem zu beobachtenden Objekt (=Subjekt) erzeugen möchte. Das Subjekt führt dazu eine Liste von Interessenten, die sich bei ihm angemeldet haben. Wird eine Instanzvariable des Subjekts geändert, steht es dem Subjekt frei, alle Interessenten zu benachrichtigen.Ein Subjekt kann verschiedene Instanzvariablen (in Smalltalk Aspekte) haben, die sich ändern können.

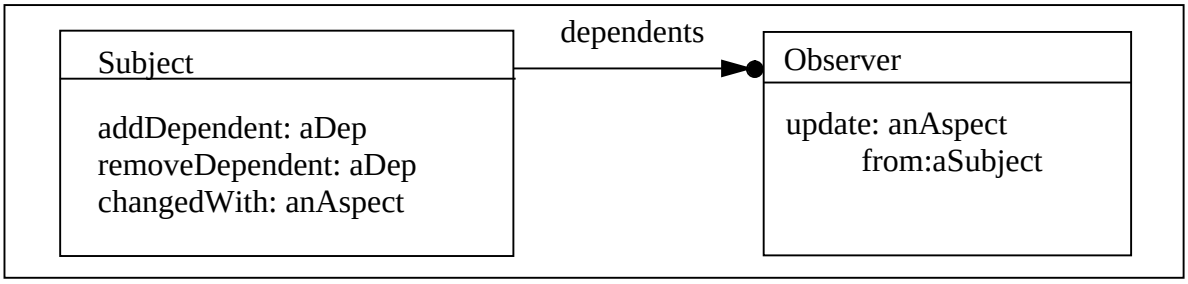


Abb. 2.6 Observer

Wenn sich ein Aspekt des Subjekts ändert, kann dieses alle seine Interessenten von der Änderung dieses Aspekts in Kenntnis setzen, ohne sich darum kümmern zu müssen, ob der jeweilige Interessent an diesem Aspekt interessiert ist.

Man verwendet hier einen vorgegebenen Kommunikationsweg zwischen Objekten wieder, der über spezielle Schnittstellen standardisiert ist. Es liegt keine feste Verdrahtung durch Klassenbildung zwischen Objekten vor, vielmehr können sich Interessenten jeglicher Art während der Laufzeit an- oder abmelden. Dies entpricht der goldenen Regel aus 2.2.3.2. Diese lose Kopplung trägt aber auch zu einer größeren Unübersichtlichkeit des Systems bei.

2.2.3.3.2 Composite

Das Composite-Entwurfsmuster komponiert Objekte in Baumstrukturen, um Teil-Ganzes-Hierarchien darzustellen. Es erlaubt Benutzern einzelne Objekte und Objektkompositionen dieser Strukturen gleich zu behandeln. [GHJV]. Das Interface, das sowohl Leaf als auch Composite erfüllen sollen, wird dazu in der abstrakten Klasse Component spezifiziert.

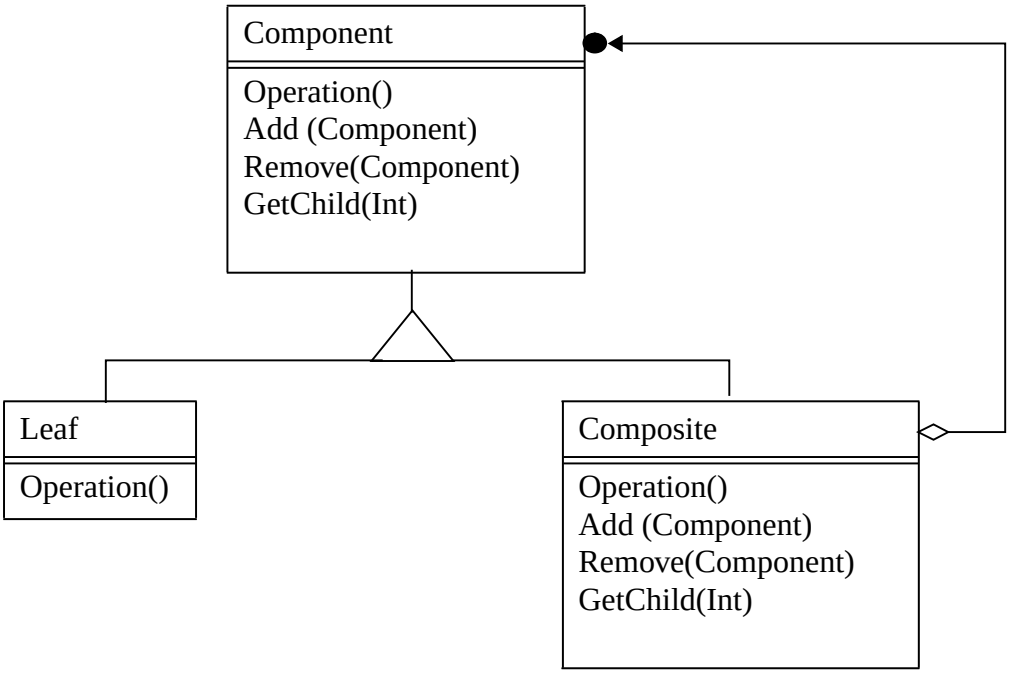


Abb. 2.7 Die Struktur des Composite-Entwurfsmusters

2.2.3.4 Frameworks

„A framework is a set of cooperating classes that make up a reusable design for a specific class of software. (...) The framework dictates the architecture of your application.“ [GHJV]

Frameworks bringen zwei große Vorteile. Zum einen kann durch ein dem Problem angemessenes Framework das gewünschte Programm schneller implementiert werden, da man eine Menge an Funktionalität zur Lösung des Problems bereits mitgeliefert bekommt, zum anderen lassen sich Probleme, die innerhalb eines Frameworks programmiert werden können, beim nächsten Mal schneller lösen, da man dann auch schon die Handhabung des Frameworks kennt.

Ein Framework ist kein Entwurfsmuster, da es spezieller auf einen Typ Anwendung hin konstruiert ist, bereits Code enthalten und mehrere Entwurfsmuster umschließen kann. Ein Entwurfsmuster enthält keine Frameworks.

Idealerweise sollten Frameworks mit Hilfe von Entwurfsmustern beschrieben werden, da sich mit deren Hilfe abstrakte Sachverhalte schnell erläutern lassen. So wurde auch HotDraw, einem Framework für graphische Editoren, welches für DOLPHIN verwendet wurde, in [Joh] vollkommen durch Entwurfsmuster beschrieben.

2.2.3.4.1 Model View Controller

Das Model-View-Controller (MVC) Konzept wird unter anderem in Smalltalk dazu benutzt, graphische Benutzeroberflächen zu erstellen. Die Idee der „vielfenstrigen“ GUI von Smalltalk wurde zwar weithin kopiert und fand über Apple Eingang in die meisten modernen Betriebssysteme. Das Grundkonzept dieser GUI in Form des MVC-Paradigmas setzt sich allerdings erst allmählich durch. Zu seinem Durchbruch könnte [GHJV] verhelfen, da in diesem sehr populären Buch nicht nur eines der zentralen Entwurfsmuster dieses Frameworks, der Observer, erklärt wird, sondern auch MVC als motivierendes Beispiel in der Einleitung vorgestellt wird.

„Model“ steht für das Datenmodell und beschreibt das Verhalten der abstrakten, problembezogenen Datentypen. „View“ läßt eine Sicht auf diese Daten zu und „Controller“ koordiniert die Benutzereingaben von Maus und Tastatur mit dem „View“ und dem „Model“.

Diese drei Einheiten sind lose miteinander gekoppelt, so daß jede Ausprägung einer solchen Einheit in Form einer speziellen Klasse mit verschiedenen Ausprägungen der jeweils anderen Einheiten leicht verbunden werden kann. So ist es einer Ansicht einer zweidimensionalen Kurve gleichgültig, ob diese einen Temperaturverlauf oder einen Aktienkurs angibt. Die lose Kopplung zwischen dem Model und dem View wird durch die Anwendung des Observer-Entwurfsmusters realisiert. Als weiteres zentrales Entwurfsmuster dient MVC noch Composite, mit dessen Hilfe beispielsweise ein Subview genauso verschoben werden kann, wie ein Fenster, das aus mehreren Subviews besteht.

2.3 Smalltalk

Warum Smalltalk für einen Simulator für neuronale Netze?

> Smalltalk is built on the model of communicating objects. Large applications are viewed in the same way as the the fundamental units from which the system is built.[GoRo]

2.3.1 Allgemeine Eigenschaften

Smalltalk ist eine objektorientierte Entwicklungsumgebung, die in den 70er Jahren am Palo Alto Research Center (PARC) entstand. Die Wurzeln liegen bei der Programmiersprache Simula und Alan Kay, der die Mitarbeiter dort dazu anstiftete, sich an die Herstellung eines rein objektorientierten Softwaresystems zu wagen.

In der Smalltalk-Umgebung sorgt eine virtuelle Maschine für die Verwaltung und den Nachrichtenaustausch der Objekte. Sie interpretiert dazu den maschinennahen Bytecode, eine Art P-Code, der beim Kompilieren einer Methode erzeugt wird. Man kann deswegen nicht von einer Interpreter- oder einer Compilersprache reden. Die virtuelle Maschine beschreibt zusammen mit

ihren Objekten den Zustand des Systems. Die Gesamtheit der Objekte und deren Schachtelung nennt man auch Image des Systems. Dieses Image kann gespeichert und wieder geladen werden, so daß jederzeit der Gesamtzustand des Systems persistent gemacht werden kann.

In Smalltalk gibt es keine Typdeklaration, erst wenn an ein Objekt eine Nachricht geschickt wird, kann festgestellt werden, ob es diese versteht. Dies hat den Vorteil, daß der Programmierer keine Deklarationen a la *Integer aCounter* angeben muß, aber den Nachteil, daß Typverletzungen erst zur Laufzeit entdeckt werden können. Jedoch könnte ein Objekt gefragt werden, ob es eine Nachricht versteht, bevor diese abgeschickt wird. Smalltalk ist ungebunden polymorph, das heißt, daß die gleiche Methode nicht nur innerhalb einer Klasse mehrfach sondern in allen Klassen implementiert sein kann.

Standardmäßig ist in Smalltalk keine Mehrfachvererbung vorgesehen, es gibt aber Klassenbibliotheken, die diese erlauben, aber die Performanz des Systems herabsetzen und die in 2.2.3.1. erwähnten Nachteile mit sich bringen. In Dolphin wurde auf sie verzichtet.

Smalltalk ist rein objektorientiert, so daß man nicht in Gefahr kommt, in prozedurale Denkmuster zurückzufallen. So basiert auch die gesamte Arithmetik in Smalltalk auf dem Senden von Nachrichten von Objekt zu Objekt. Dies hat den großen Vorteil, daß die Metapher „Objektorientierung“ auch dort nicht gesprengt wird, aber den Nachteil, daß die Berechnung arithmetischer Ausdrücke eine Art Flaschenhals im System bildet. Trotz vieler Verbesserungen im Laufe der letzten Jahre gibt es bei der Arithmetik zusammen mit dem „objektorientierten“ Zugriff auf Filesysteme immer noch die größten Performanzdefizite im Vergleich zu anderen Programmiersprachen.

Die Klassenbibliothek ist sehr reichhaltig, so gibt es auch Klassen für *Process* und dessen Verwaltung in Form von *Semaphore* und speziell sogar zur Verwaltung von Threads in Form von *RecursionLock*, der es einem einen Semaphor setzenden Thread erlaubt, selber diesen Semaphor umfahren zu können. Die Speicherfreigabe erfolgt in Smalltalk automatisch durch einen Garbagecollector, der Programmierer muß sich nicht um die Entsorgung nicht mehr benötigter Objekte kümmern. Eine Schnittstellenkapselung ist nur durch die stille Vereinbarung der Smalltalkprogrammierer gegeben, nicht öffentliche Methoden in einer Methodensammlung namens ’*private*’ abzulegen. Theoretisch kann auf diese aber trotzdem von jeder anderen Klasse zugegriffen werden. Das Standardwerk zum Thema Smalltalk ist das sogenannte Purple Book von Adele Goldberg und David Robson. [GoRo].

2.3.2 VisualWorks 2.5

Die verwendete Smalltalk-Version ist VisualWorks 2.5. Ein großer Vorteil dieser Entwicklungsumgebung ist die Plattformunabhängigkeit der Images. So gibt es virtuelle Maschinen für Alpha-Dec, Sun-Solaris, Windows 3.1,Windows 95, Windows NT, OS/2 und Apple PowerMac, die alle auf dem gleichen Image arbeiten können.

Es gibt in VisualWorks 2.5 einen „Builder“ zur Erstellung graphischer Benutzeroberflächen. Leider waren meine Kenntnisse dieses Builders noch zu beschränkt, als daß ich ihn auf die Erstellung von Oberflächen neuronaler Netze einschränken bzw. erweitern hätte können. Jedoch

lassen sich mit Hilfe dieses Builders sehr leicht Applikationsoberflächen erzeugen, die damit auch ohne viel Aufhebens wieder verworfen werden können, was dem sogenannten Rapid Prototyping sehr entgegen kam.

Bei der Entscheidung für eine Entwicklungsumgebung spielte auch die Möglichkeit eine Rolle, wie gut diese an moderne Datenbanken anschließbar wäre. Für VisualWorks steht eine Anschlußmöglichkeit an die relationalen Datenbanksysteme DB2, Oracle und Sybase und an diverse objektorientierte Datenbanken zur Verfügung.

Zum Gedanken Hybridität auch dadurch zu erreichen, bereits vorhandene Trainingsdaten aus einer Datenbank einzuholen, kam bei der Entscheidung für ein System auch die Idee, die Netze auf mehrere Rechner zu verteilen, dafür war von Vorteil zu wissen, daß es mit HP-Distributed Smalltalk einen anschlußfähigen, den Corba 2.0 Standard benutzenden „Verteiler“ gab.

3 Das hybride Modell

Der Hauptvorteil von neuronalen Netzen bei der Systemidentifikation liegt eigentlich in der Modellfreiheit. Hat man ein physikalisch-mathematisches Modell, dem die vorhandenen Daten zu Grunde liegen, ist es in der Regel schneller und exakter, mit statistischen Methoden die freien Parameter der Modellgleichungen zu bestimmen. Hat man jedoch mathematisch-physikalische Modelle nur für Teile eines Systems, ist es außerordentlich schwierig, die Daten statistisch zu modellieren. Daher werden in letzter Zeit vermehrt Versuche gemacht, mathematische Modelle und neuronale Netze in hybriden Systemen zu kombinieren. Wenn physikalisch-mathematische Zusammenhänge bekannt sind, sollen sie mit in das Netz eingebracht werden können oder unbekannte Teile eines Modells sollen durch eingebettete Netze ersetzt werden. Dieses ist mit dem klassischen BP-Algorithmus, zumindest wie er in den Standardwerken zu neuronalen Netzen [RuCl] [Roj] beschrieben wird, zunächst nicht möglich, wenn das neuronale Netze nicht am Ausgang des Systems ist. MLPs brauchen einen Sollwert an ihrem Ausgang zur Generierung des Fehlersignals. Werden hinter dem Netz die Ausgangsdaten noch weiter manipuliert, wäre eine Möglichkeit, die entsprechenden Gleichungen zu invertierten, um dem Netz doch einen Sollwert zur Verfügung zu stellen. Eine Alternative ist, der Simulation der hinter dem Netz liegenden Transformation die Fähigkeit der Fehlerbackpropagation beizubringen. Dann kann wie gewöhnlich am Ausgang ein Fehler berechnet werden und ein Fehlersignal rückwärts zum Netz transportiert werden. Der Gradientenabstieg erfolgt quer durch das gesamte Modell aus Neuronen und mathematischen Funktionen.

So kann sich das Gesamtnetz immer wie ein klassisches BP-Netz trainieren lassen. Es wurde dazu zunächst der BP-Algorithmus in DOLPHIN auf objektorientierte Weise implementiert. Um eine möglichst hohe Wiederverwendbarkeit und Flexibilität von DOLPHIN zu erreichen, wurde hier bereits darauf geachtet, eine allgemein gültige Netzfunktionalität zu schaffen.

Im Folgenden wird gezeigt, welche Wege gefunden wurden, den BP-Algorithmus innerhalb DOLPHINs zu erweitern, um das Gesamtnetz mit seiner Hilfe flexibler trainieren zu können. Experimente und deren Ergebnisse zur Hybridität von DOLPHIN finden sich in Kapitel 6.

3.1 Hybridität durch beliebige Aktivierungsfunktionen

Es soll zunächst kurz auf drei Punkte eingegangen werden, warum biologische neuronale Netze und künstliche BP-Netze nur sehr wenig miteinander gemein haben. Weshalb der klassische BP-Algorithmus technisch aber dennoch Sinn macht, wie und warum man ihn mit anderen differenzierbaren Funktionen als der klassischen Summations- und Sigmoidkombination, erweitern kann, wird anschließend erklärt werden.

Die Summation als Integrations- und eine Sigmoide als Ausgabefunktionen beim BP-Algorithmus in künstlichen neuronalen Netzen zu verwenden, kam, wie in Kapitel 2.1 gezeigt wurde, ursprünglich von der Idee, ein Modell zur Nachbildung biologischer neuronaler Netze zu finden, deren hohe Leisungsfähigkeit als durchaus anerkannt gilt.

Der BP-Algorithmus ahmt das Vorbild der Biologie aber kaum nach. Zum einen liegt die Rechenkapazität moderner Computer noch weit unter der Anzahl der Neuronen eines menschlichen Gehirns. Außerdem kann der zueinander asynchrone Charakter des Verhaltens biologischer Neuronen, die ihr Potential, auch ohne zu feuern, mit der Zeit wieder abbauen, kaum mit der „event"-gesteuerten Methode des klassischen BP-Algorithmus nachgeahmt werden. Bei diesem wird jeder Eingangswert solange gespeichert, bis an allen Dendriten Werte anliegen, erst dann kann das Neuron feuern und seine Eingängswerte wieder vergessen. Des weiteren ist kaum anzunehmen, daß es in biologischen Netzen eine Art Vorwärts- und Rückwärtsschritt geben kann, der wie in 4.1.1 erklärt wird, zeitlich nicht lokal abläuft.

Es kann gesagt werden,

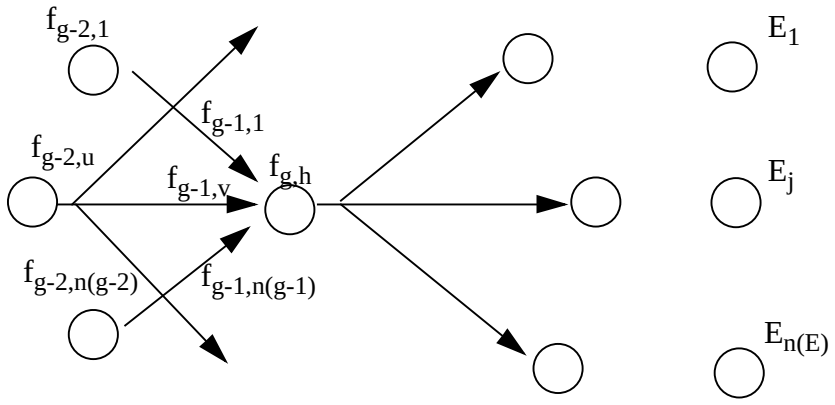
> „daß die Aktivität biologischer Neuronen keine Schwellwertoperation ist, sondern eine hochgradig nichtlineare, komplexe Antwort auf die Gesamtheit der momentanen Eingaben sowie den eigenen Momentanzustand" [Lit]

Andersherum stellt sich die Frage, ob sich BP-Netze mit sigmoiden Aktivierungsfunktionen zur Lösung technischer Probleme besonders gut eignen, und wie sich der BP-Algorithmus mit anderen Aktivierungsfunktionen erweitern läßt. Der BP-Algorithmus mit Sigmoiden als Ausgabefunktion hat sich in vielen technischen Anwendungen unter Beweis gestellt. Die Sigmoide ist besonders gut zur Annäherung von Funktionen geeignet, weil sie einen linearen und einen nichtlinearen Anteil besitzt und im Limes beschränkt ist. Durch eine ausreichend komplexe Linearkombination von Sigmoiden kann jede stetig differenzierbare Funktion beliebig genau angenähert werden. Tatsächlich konnte man beweisen, daß dazu eine einzige versteckte Schicht sigmoider Neuronen in einem BP-Netz ausreicht, vorausgesetzt man verwendet genügend Neuronen [Gal].

Wenn aber jegliche differenzierbare Funktion als Aktivierungsfunktion benutzt werden kann, könnte man beispielsweise bereits mit $A_{(x_1 \ldots x_m)} = \sum_{i=1}^{m} x^i$ alle differenzierbaren Funktionen durch ein Polynom approximieren, die einzustellenden Koeffizienten wären die Gewichte der Eingangskanäle. In der Literatur [RuCl] [Roj] wird Backpropagation meistens nur mit der Summation als Integrations- und einer Sigmoiden als Ausgabefunktion hergeleitet.

Ich habe deswegen im Folgenden Backpropagation für beliebige Aktivierungsfunktionen selber hergeleitet. In dem Modell aus 3.1 steht der Index g für eine Funktionsschicht des Netzes. Eine Funktionsschicht besteht in diesem Fall entweder ganz aus Kanälen oder ganz aus Neuronen. Vorausgesetzt wird nur, daß alle Einheiten, die in Richtung Ausgabe von einer ausgewählten Einheit erreichbar sind, einen höheren Schichtindex haben, als die ausgewählte Einheit. Dies läßt sich immer realisieren, da das Netz in unserem Fall nicht zirkulär aufgebaut sein darf. Es lassen sich mit dieser Definition von Schicht aber prinzipiell auch „Cross-Connections" realisieren.

Ziel des Backpropagationschrittes ist es, die Gewichte so zu verändern, daß der Gesamtfehler des Netzes kleiner wird. Der Gesamtfehler wird häufig über die Summe der quadrierten Differenzen zwischen den gewünschten und den tatsächlichen Ausgabewerten definiert. Dieser muß dann nach dem Gewicht abgeleitet werden, dessen Änderung gesucht ist. (2).



(1) $$f_{g-1,v} = f_{g-2,u} w_{u,v}$$

(2) $$\frac{dE}{dw_{u,v}} = \frac{d}{dw_{u,v}} \left(\sum_{j=1}^{n(E)} \frac{1}{2} (d_e - f_{E,j})^2 \right)$$

(3) $$\frac{dE}{dw_{u,v}} = \sum_{j=1}^{n(E)} \left((d_e - f_{E,j})(-1) \frac{df_{E,j}}{dw_{u,v}} \right)$$

(4) $$\frac{df_{E,j}}{dw_{u,v}} = \frac{df_{E,j}}{df_{g-1,v}} \frac{df_{g-1,v}}{dw_{u,v}} = \frac{df_{E,j}}{df_{g-1,v}} f_{g-2,u}$$

(5) $$\frac{df_{E,j}}{df_{g-1,v}} = \frac{df_{E,j} df_{g,h}}{df_{g,h} df_{g-1,v}}$$

(6) $$\frac{df_{E,j}}{df_{g-2,u}} = \sum_{e=1}^{n(g-1)} \frac{df_{g-1,e}}{df_{g-2,u}} \left(\frac{df_{E,j}}{df_{g-1,e}} \right) = \sum_{e=1}^{n(g-1)} \left(w_{u,(g-1,e)} \frac{df_{E,j}}{df_{g-1,e}} \right)$$

(7) =(5)in (4) $$\frac{df_{E,j}}{dw_{u,v}} = \frac{df_{E,j} df_{g,h}}{df_{g,h} df_{g-1,v}} f_{g-2,u}$$

(8) =(7)in (3) $$\frac{dE}{dw_{u,v}} = \sum_{j=1}^{n(E)} \left((d_e - f_{E,j})(-1) \frac{df_{E,j} df_{g,h}}{df_{g,h} df_{g-1,v}} f_{g-2,u} \right)$$

Abb. 3.1 Eine Herleitung des Backprop-Algorithmus für beliebige Aktivierungsfunktionen

In (3) tritt zum ersten Mal die Kettenregel in Aktion. Der Term aus (2) muß nachdifferenziert werden, die „Nachdifferenz“ $\frac{df_{E,j}}{dw_{u,v}}$ wird in (4) weiter aufgelöst. Hier wird die Tatsache benützt, daß die Ausgabe eines Effektors abgeleitet nach einem Gewicht w_{uv} gleich dem Produkt von der Ableitung der Ausgabe des Effektors nach der Ausgabe an diesem Kanal $f_{g-1,v}$ und der Ableitung der Kanalfunktion nach dem Gewicht ist. Die Ableitung der Kanalfunktion (1) nach dem Gewicht ist gleich der Eingabe des Kanals $f_{g-2,u}$ (4). In (5) findet sich eine der beiden für BP entscheidenden Gleichungen, in diesem Fall für den Schritt von einer Schicht von Neuronenfunktionen zu einer Schicht von Kanalfunktionen. Die Ableitung des Fehlers nach einem Kanalfunktionswert $f_{g-1,v}$ist gleich mit der Ableitung des Fehlers nach einem vorausgegangenen Funktionswert $f_{g,h}$ eines Neurons multipliziert mit der lokal berechenbaren Ableitung von $f_{g,h}$ nach $f_{g-1,v}$. Für $f_{g,h}$ kann dabei jede beliebige partiell differenzierbare Funktion benützt werden. Die andere Gleichung findet sich in (6), hier ist der BP-Schritt analog zu (5) von der Schicht der Kanalfunktionen zur Schicht der Neuronenfunktionen dargestellt. Die Summe in (6) kommt daher, daß sich die Fehler der verschiedenen Ausgänge eines Neurons addieren. Damit ist gezeigt worden, daß sich die Fehler der Schichten g-2 und g-1 immer aus den bereits ermittelten Fehlern der Schichten *g-1* bzw. *g* berechnen lassen und dies auch für beliebige differenzierbare Aktivierungsfunktionen eines Neurons gilt.

3.2 Nicht trainierbare Kanäle

Um gewisse physikalische Eigenschaften festzuhalten, kann es sinnvoll sein, manchen Kanälen die Lernfähigkeit abzusprechen. Deren Gewicht bleibt dann auf einem einstellbaren, konstanten Wert. Realisiert wurde dies, indem nur trainierbare Kanäle mit den gelernten Gewichtsdeltas auf den neusten Stand gebracht werden. Siehe dazu auch die Beispiele in 6.1.3. Sehr häufig werden Kanäle mit dem Gewichtsfaktor eins als passive Datenpfade benutzt.

3.3 Lokale Einstellbarkeit der Lernrate

Ein weiterer Vorteil von DOLPHIN gegenüber anderen Simulatoren ist, daß man die Lernrate nicht nur global, sondern auch für jeden Kanal gesondert einstellen kann. Dadurch können manche Verbindungen schwächer trainiert werden als andere. Dies ist sinnvoll, wenn man einen durch spezielle Funktionen ins Netz codierten Zusammenhang nur „schwer“ änderbar machen möchte.

3.4 Shared Weights

Der Begriff der „Shared Weights“ geht ursprünglich auf [NoHi] zurück. Diese hatten damit als Ziel eine Performanzsteigerung der Netze vor Augen.

Jedem Gewicht w_i kann ein Set anderer Gewichte $[w_k]$ zugeordnet werden, die von diesem Gewicht beeinflußbar sind, definiert durch Faktoren s_{ik} , die die Stärke dieser Beeinflussung angeben. Zusätzlich zum Δw_k, das durch das Lernen mit Backprop ermittelt wird, erhält man dann einen zusätzlichen Δ-Term, $\Delta_{share} w_k = \sum_i s_{ik}(w_i - w_k)$,

der in jedem Lernschritt zum Gewicht addiert wird:

- $w_k = w_k + \Delta w_k + \Delta_{share} w_k$
- $s_{ik} = 0$ bedeutet normales Backprop
- Ist $s_{ik}=1$ wird Gewicht w_k total auf den Wert von w_i gesetzt
- Um symmetrisches Annähern der Gewichte zu erreichen, sind s_{ik} und s_{ki} beide zu setzen.

3.4.1 Vereinfachung von Netzen

Um an zwei Stellen im Netz Werte mit einem identischen, aber adaptierbaren Faktor zu multi-

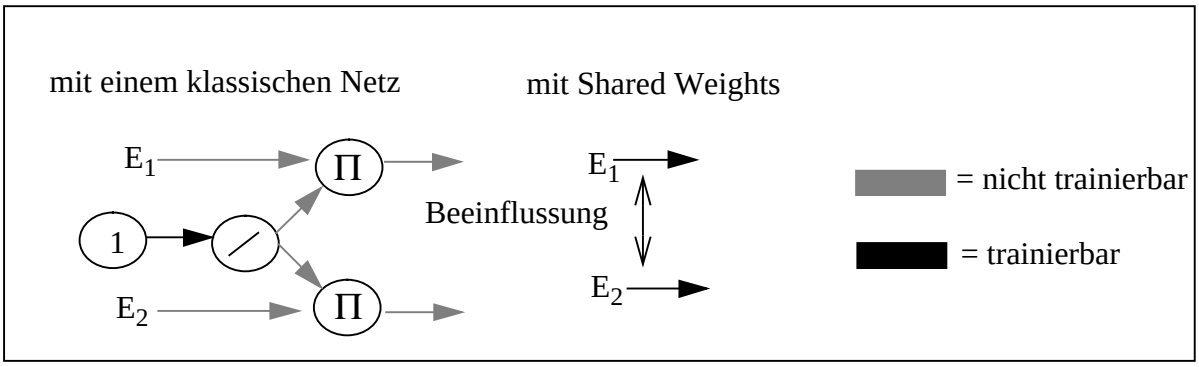


Abb. 3.1 Implementierung eines an mehreren Stellen benötigten zu adaptierenden Parameters

plizieren, bräuchte man ein Konstrukt aus vier Neuronen und sieben Kanälen, während Shared Weights ohne zusätzliche Elemente auskommen.

3.4.2 Simulation eines Mischvorgangs

Stellen wir uns eine Bäckerei vor, die zwei identische Maschinen benutzt, um Bestandteile eines späteren Teigs zu mischen. Es gebe eine Mischmaschine für Bestandteil 1 und eine für Bestandteil 2. Man vermutet einen Zusammenhang zwischen der Außentemperatur *T* und dem Ergebnis des Mischvorgangs.

Man vermutet auch, daß die Außentemperatur bei beiden Mischmaschinen eine ähnliche Rolle spielen wird, wobei leichte Abweichungen in Kauf genommen würden. Man erstellt ein Netz gemäß Abbildung 3.2. Die beiden Gewichte w1 und w2 lassen sich miteinander teilen.

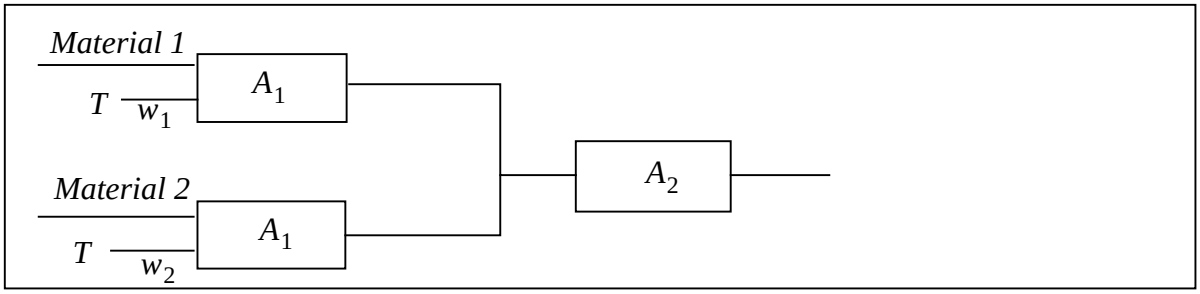


Abb. 3.2 Eine hypothetische Bäckerei

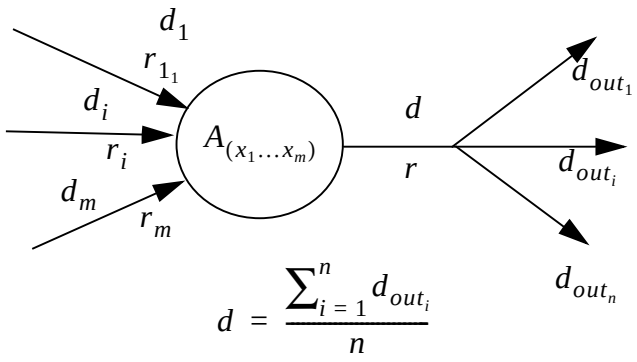
3.5 Komposition eines BP-Netzes aus anderen BP-Netzen

> „...daß mit dem Error-Backpropagation-Algorithmus nur ein Lernsignal abgeleitet werden kann, gemäß dem die inneren Gewichte adaptiert werden können. Es steht jedoch nach wie vor kein direkter Zielwert zur Verfügung, den die inneren Neuronen approximieren sollen.“ [Lit]

Im Folgenden wird erklärt und motiviert werden, welcher Ansatz verfolgt wurde, klassische BP-Netze zu erweitern, indem diese selbst wieder als Funktionsbausteine für weitere BP-Netze angesehen werden.

Ein solches Netz hätte den großen Vorteil, daß man kleinere Subnetze, die in einem größeren Netz mehrmals vorkommen, bei Vorhandensein geeigneter Daten vortrainieren und aus ihnen „zwanglos“ das neue Netz zusammenstellen könnte. Sie wären als Module persistent zu machen und könnten leicht wiederverwendet werden.

Eine der wesentlichen Ideen bei SEAME, einem objektorientierten Simulator für neuronale Netze [Lin], ist die sogenannte datenflußorientierte Sichtweise. Kompatibilität unter den Einheiten wird bei datenflußorientierter Herangehensweise dadurch erzeugt, daß der Eingang einer Einheit das Datenformat kennt, was der Ausgang der zu ihr kompatiblen Einheit zur Verfügung stellt. Datenflußorientiert ist bei SESAME so auch die Implementation des BP-Algorithmus. Es können dabei BP-fähige Bausteine beliebig kombiniert werden. Eine Komposition solcher BP-Bausteine, die sich bei [Lin] auch Z(unsammengesetzter)-Baustein nennt, kann aber selbst nicht wieder als Baustein in einem größeren BP-Netz verwendet werden. Die Frage lautete, wie eine Einheit beschaffen sein müßte, aus der sich ein klassisches BP-Netzwerk bauen ließ, das selbst als Baustein für weitere BP-Netzwerke dienen könnte. Die Grundidee bestand darin, nicht den Fehler zurückzupropagieren, sondern den gewünschten Eingabewert d_{in_i} aus den Eingabewerten r_{in_i}, dem Ausgabewert r und den gewünschten Ausgabewerten d_{out_i} zu ermitteln und zurückzureichen. Dies hätte den Vorteil, daß man zwischen der Ausgabeschicht und den Berechnungsschichten keine künstliche Trennung einführen müsste, da beim klassischen Backpropagation die Einheiten der Ausgabeschichten Soll- und Istwerte, die Einheiten der versteckten Schichten aber Fehler beim Rückwärtsschritt erhalten. Man betrachte den Moment, in dem das



Neuron alle Eingangswerte an seinen Dendriten, alle Ausgangswerte an seinen Axonen und alle gewünschten Ausgangswerte an seinen Axonen zur Verfügung stehen. Dies ist bei einem Neuron der hinteren Schicht (ehemals Effektor) immer möglich. Es muß dann den einen gewünschten Ausgabewert d berechnen, und aus diesem die gewünschten Eingabewerte d_{in_i} ermitteln. Der gewünschte Ausgabewert d wird einfach durch Mittelwertbildung der gewünschten Ausgabewerte $d_{out\text{-}i}$ erzeugt.

Das Neuron habe die allgemeine Aktivierungsfunktion

$$A(x_1 \ldots x_m)$$

Eine Linearisierung von *A* an der Stelle des alten Eingangsvektors $\bar{r}$ liefert in Matrixschreibweise

$$A(\bar{x}) \approx (r + \bar{A}) \cdot (\bar{x} - \bar{r}) = \bar{r} + \begin{bmatrix} A_1 & \ldots & A_m \end{bmatrix} \cdot \left(\begin{bmatrix} x_1 \\ \ldots \\ x_m \end{bmatrix} - \begin{bmatrix} r_1 \\ \ldots \\ r_m \end{bmatrix} \right) = r + \sum_{i=1}^{m} A_i \cdot (x_i - r_i)$$

dabei ist A_i die partielle Ableitung von $A(x)$ an der Stelle $\bar{r}$

$$A_i = \frac{\partial}{\partial x_i} A(r_1 \ldots r_m)$$

dann ist $d = A(d_1 \ldots d_m)$ angenähert

$$d = r + \sum_{i=1}^{m} A_i \cdot (d_i - r_i)$$

Die Bedingung, daß der neue Eingabevektor ($d_1 \ldots d_m$) möglichst ähnlich dem alten ($r_1 \ldots r_m$)

sein soll, also

$$\sum_{i=1}^{m}(d_i - r_i)^2 \rightarrow min$$

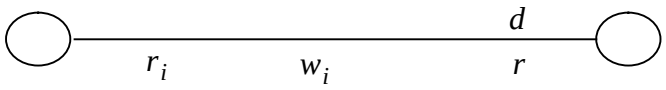
ist genau dann erfüllt, wenn die Verschiebungen (d_i-r_i) alle gleich groß sind, daher kann man sie aus der Summe herausziehen

$$d - r = (d_i - r_i)\sum_{i=1}^{m} A_i$$

also sind die optimalen „desired inputs“ d_i

$$d_i = r_i + \frac{d - r}{\sum_{i=1}^{m}\frac{\partial}{\partial x_i}A(r_1 \ldots r_m)}$$

Die Ableitung des Fehlers nach dem Gewicht in einem Kanal kann leicht ausgerechnet werden, da das *d* eines Kanals dem d_i seiner Ausgangseinheit entspricht. Die Gewichtsänderung ist wieder gleich dieser Ableitung multipliziert mit einer Lernrate.



$$E = \frac{1}{2}(r - d)^2$$

$$E = \frac{1}{2}(r_i w_i - d)^2$$

$$\frac{dE}{dw_i} = (r - d)r_i$$

$$\Delta w_i = \frac{dE}{dw_i}\gamma = (r - d)r_i\gamma$$

Wenn die Ableitung der Aktivierungsfunktion nach dem Fehler sehr klein wird, ändert sich d_i sehr stark, wodurch auch das nächste w_i einen großen Gewichtszusatz oder eine große Gewichtsminderung erfährt. Dies ist das genau gegenteilige Verhalten zu klassischem Backpropagation, bei diesem steht die Ableitung $\frac{\partial}{\partial x_i}A(r_1 \ldots r_m)$ im Zähler, so daß sich dort nur sehr geringe Gewichtsdifferenz ergibt.

4 Entwicklung und Architektur von Dolphin

Zunächst möchte in diesem Kapitel darauf eingehen, welche Strategien verwendet wurden, DOLPHIN zu programmieren, um so seine Entwicklungsgeschichte zu dokumentieren und auch ein Beispiel zu geben, nach welchem Schema DOLPHIN ausgebaut werden kann.

Anschließend werden die Klassen, die im Zusammenspiel das Modell eines neuronalen Netzes darstellen, beleuchtet. Danach wird die Einbindung des Modells in das Framework namens HotDraw erklärt, das in diesem Fall erlaubt, neuronale Netze graphisch zu editieren. Schließlich erkläre ich die Verschaltung der Benutzerschnittstellen untereinander und mit dem Modell.

4.1 Strategien

Eine Strategie ist ein Aspekt eines Prozeßmodells, der beschreibt, auf welche Weise Aktivitäten wie Analyse, Design oder auch Dokumentation eines Projektes durchgeführt werden. Die Strategie der zeitlichen Abfolge der Aktivitäten beim Wasserfall-Modell ist beispielsweise rein sequentiell. Andere Strategien für objektorientierte Programmierung sind Rapid Prototyping und Wiederverwendbarkeit.

4.1.1 Inkrementell iterativer Ansatz

Häufig wird die zeitliche Abfolge beim objekt-orientierten Programmieren mit Hilfe des Schlagwortes „Inkrementell iterativer Ansatz“ beschrieben. Da auch bei der Entwicklung von DOLPHIN diese Strategie Verwendung gefunden hat, möchte ich sie im Folgenden kurz erklären und dann zeigen, wie sie auf DOLPHIN Einfluß genommen hat.

Inkrementell meint, daß jedes Modul für sich entwickelt wird, um dann in das Gesamtsystem eingebettet zu werden. Zunächst wurde das Modell eines neuronalen Netzes entwickelt, gefolgt von der Einbettung in das zurechtgestutzte Framework von Hotdraw. Um dieses Modul herum wurden nach und nach die anderen Bausteine wie etwa das rezeptive Feld hinzugefügt. In Abbildung 4.1 sieht man die wichtigsten Bausteine und wie sie in das System integriert wurden.

Die iterative Strategie wird auch als opportunistische Strategie bezeichnet, da dem Programmierer nicht vorgeschrieben wird, in welcher Reihenfolge er die Komponenten zu bearbeiten hat. Er kann immer gerade die Teile bearbeiten, die ihm im Moment am geeignetsten erscheinen. Das Erstellen eines Softwaresystems mit dieser Strategie wird bei Goldberg in [GoRu] auch mit dem Lösen eines „Jigsaw-puzzles“, zu deutsch eines (Laubsäge)-Puzzles verglichen, bei dem man auch ganz opportunistisch zunächst die Puzzleteile zusammenhängt, die einem gerade recht kommen. Besonders negativ scheint sie und „ihre“ Firma ParcPlace-Digitalk dieser Art von Opportunismus nicht gegenüber zustehen, da das Produkt der Vereinigung aus VisualWorks und Visual Smalltalk Enterprise, und damit wohl einer der wichtigsten Smalltalk-Dialekte der Zukunft, „Jigsaw“ heißen soll.

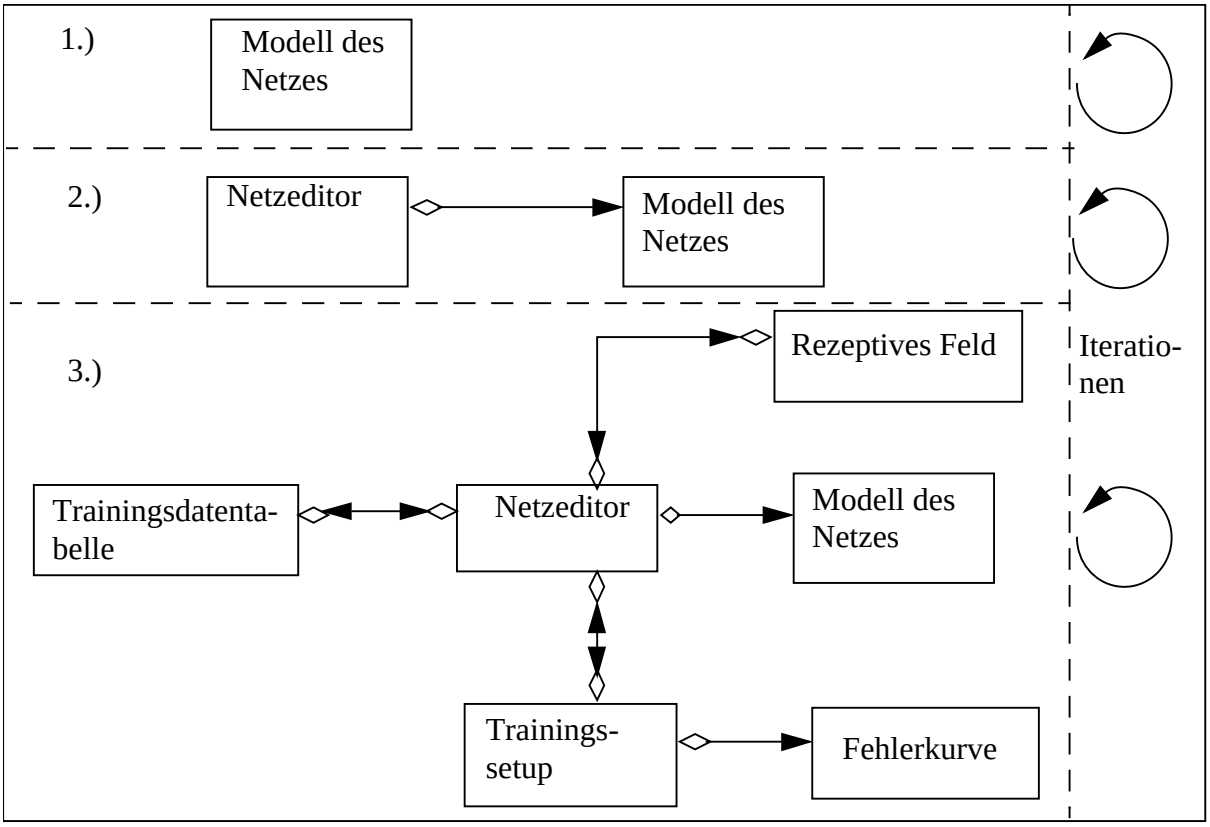


Abb. 4.1 Einheiten des inkrementell, iterativenEntwicklungszyklus

Die Entwicklung von Dolphin war iterativ, das heißt, daß das jetzige Ergebnis ein Produkt von vielen Verbesserungszyklen ist. Das lag zum einen daran, daß sich meine Smalltalk-Kenntnisse im Laufe der Arbeit verbesserten und ich so später manche Protokolle eleganter programmieren konnte, zum anderen aber auch, daß sich meine Sicht des Problemgebietes und seiner möglichen Lösungen änderte und sich so auch hier viele Verbesserungen ergaben.

Ich hatte sehr schnell das erste Netzwerkmodell programmiert, dieses hat aber noch insgesamt fünf große Änderungen erfahren, bis es zum heutigen Zustand gelangte. So kam erst die Signalverwaltung durch Dictionaries hinzu, bevor auf sie wieder aus Effizienzgründen verzichtet wurde. Anschließend wurde noch dreimal die komplette Simulation des Signalflusses im Netz verändert. Diese Änderungen ließen sich alle auf Grund der Unabhängigkeit des Modells von seinen Beobachtern leicht verwirklichen. Es gab kaum Seiteneffekte, die aufgefangen hätten werden müssen. Die einzelnen Bausteine paßten durch ihre festgelegten Schnittstellen weiterhin zusammen.

4.1.2 Wiederverwendung

Wiederverwendung findet sich bei Dolphin auf zwei verschiedene Arten. Man kann etwas wiederbenutzen oder aber auch versuchen, ein wiederverwendbares Modul zu schreiben. So hatte ich zunächst nach einem graphischen Editor Ausschau gehalten, der graphenartige Gebilde erzeugen kann. Das gefundene Werkzeug erwies sich nach zweitägigem Test als nur bedingt geeignet. Die Ausgabegraphik dieses Werkzeugs ließ sich nur indirekt verändern, eine Benutzer-

oberfläche, die einen direkten Zugriff auf die Elemente des Netzes erlaubte, konnte damit nicht realisiert werden. Schließlich fand ich HotDraw, das mir die größten Freiheiten einzuräumen schien und womit ich das Modell unabhängig von der Oberfläche gestalten konnte. Das hatte zum einen den Vorteil, daß jederzeit auch noch andere Sichtweisen auf das Modell programmiert werden können, die auf die Sicht durch HotDraw keinen Einfluß nehmen und daß sich auch das Modell ändern kann, solange nur die Schnittstellen die gleichen bleiben.

Als wiederverwertbare Komponente ist so das Modell des Netzes selber entstanden, das auch mittels einer Skriptsprache bedient werden kann.

Ebenso wiederverwertbar ist auch die Anzeige des rezeptiven Feldes. Diese Anzeige konnte zum Beispiel sofort in einer Anwendung namens `ThreeDUserDefinedFunctionPlotter` benutzt werden, die die Funktionswerte einer vom Benutzer in Smalltalk-Syntax eingegebenen Funktionen als Farbverlauf darstellt. `ThreeDUserDefinedFunctionPlotter` kann dazu verwendet werden, Ähnlichkeiten zwischen dem Ein- / Ausgabeverhalten eines neuronalen Netzes und einer vorgegebenen Funktion festzustellen.

Als weitere wiederverwendbare Klasse wurde `TwoDFunctionPlot` programmiert, die sowohl in der Anzeige der Fehlerkurve als auch in der Anzeige der „Netzfunktionen“ Verwendung findet. Die Wiederverwendung dieser Anzeigen geschieht nicht durch Ableitung von Klassen, sondern durch Komposition von sogenannten `ApplicationModel`s. Ein `ApplicationModel` ist eine Anwendung zusammen mit einem zugehörigen Fenster. Mit Hilfe des `GUI-Builder` kann in VisualWorks ein `ApplicationModel` in ein anderes `ApplicationModel` eingebaut werden. Dazu wird das Fenster des wiederverwerteten Bausteins einfach in das Fenster des Wiederverwenders injiziert.

An dieser Stelle bleibt noch zu erwähnen, daß auch reger Gebrauch der Klassenbibliotheken von VisualWorks gemacht wurde. Da diese teilweise einen so hohen Abstraktionsgrad erreicht haben, erscheint mir auch hier schon der Begriff Wiederverwendung angebracht.

4.1.3 Rapid Prototyping

Viele objektorientierte Systeme bieten Frameworks zur Erzeugung grafischer Benutzeroberflächen (= GUI, Graphical User Interface). Die Oberflächen werden dabei nicht programmiert sondern interaktiv generiert, wodurch sehr schnell erste Fenster produziert werden können. Die Funktionen, die den jeweiligen Buttons oder Menu-Einträgen zu Grunde liegen, müssen hier noch nicht implementiert sein, es geht vielmehr darum, dem Benutzer und auch dem Programmierer einen ersten Eindruck des Gesamtsystems zu vermitteln. So können in einem sehr frühen Stadium Analysefehler erkannt und behoben werden.

Wichtig für den Programmierer ist dabei, daß der durch das GUI-Framework erzeugte Quellcode möglichst transparent in seine Klassenbibliothek einbaut wird, bei Bedarf leicht geändert werden kann und die Interface-Struktur zwischen dem Modell und den erzeugten Fenstern übersichtlich bleibt.

4.1.3.1 Rapid Prototyping in VisualWorks

Im Folgenden möchte ich kurz auf die Erstellung von Oberflächen unter VisualWorks2.x eingehen. Zunächst kann mit Hilfe des `UI-Painter,` der selber ein `ApplicationModel` ist, eine Fensteroberfläche entworfen werden. Das GUI-Framework von VisualWorks 2.x legt die Spezifikation der Fenster und der verwendeten sogenannten Widgets (`Button`, `SelectionList`, `ComboBox` etc.) in einer Unterklasse von `ApplicationModel` als Klassenmethode (...`Spec`) ab. Dabei werden Informationen über die Position, Farbe und Größe ebenso festgehalten, wie beispielsweise die Namen der Methoden, welche bei Druck eines bestimmten Knopfes aufgerufen werden sollen. Diese Unterklasse von `ApplicationModel` wird vom Benutzer durch in `ApplicationModel` definierte Methoden (zum Beispiel ‚`open`') instanziiert, der `UIBuilder` erzeugt dann aus der Spezifikation die Fenster der laufenden Anwendung.

4.2 Architektur von DOLPHIN

4.2.1 Das Modell eines künstlichen neuronalen Netzes von DOLPHIN

Ein Netz in DOLPHIN besteht aus zusamenhängenden Subnetzen und sogenannten Netzblättern. Der Zusammenhang wird durch Verbindungen der Netzblätter durch sogenannte Kanäle hergestellt. Die Kanäle und die Netzblätter bestimmen im Wesentlichen die Funktionalität des Netzes, das Netz selber stellt ein Interface zu seiner Benutzung zur Verfügung.

Die Eigenschaften des Netzes zerfallen also in eine topologische und eine funktionale Komponente. Die topologische Komponente sollte möglichst allgemeingültig für alle vorstellbaren Netze im Sinne von Knoten mit beliebigen Verknüpfungen durch gerichtete Kanten programmiert werden. Die Verfeinerung dieser topologischen Komponente durch eine funktionale Komponente erfolgt durch Unterklassenbildung der topologischen Klassen `NetLeaf` und `NetComposite` (siehe Abbildung 4.2), die zusammen mit `NetComponent` ein Composite-Entwurfsmuster bilden.

4.2.1.1 Der topologische Aspekt des Netzwerkmodells

Grundsätzlich kennt eine Netzkomponente (NetLeaf oder NetComposite) vom Gesamtnetz nur ihre eingehenden und ausgehenden Kanäle (Dendriten und Axone), was in der abstrakten Klasse `NetComponent` dieser Anwendung des Composite-Entwurfsmusters festgelegt wird.

Es sollen nicht nur Netzblätter mit Netzblättern und Netze mit Netzen, sondern auch Netze mit Netzblättern verbunden werden können. Um die Behandlungsgleichheit dieser beiden Unterklassen bei Verbindungsoperationen zu wahren, werden die Axone einer Komponente dabei in einer weiteren Aggreation partitioniert, da ein Netz im Gegensatz zu einem Netzblatt verschiedenartige Ausgänge haben kann. Die Axone eines Blattes finden sich immer in der ersten Partition ihrer Axonenmenge. Wird ein Netz an einer Eingangsposition mit einer Komponente an

einer Ausgangsposition verknüpft, wird nur das Netzblatt, das im tiefsten Inneren des Netzes an dieser Eingangsposition hängt, um einen Eingabekanal erweitert und dieser mit der Komponente verbunden.

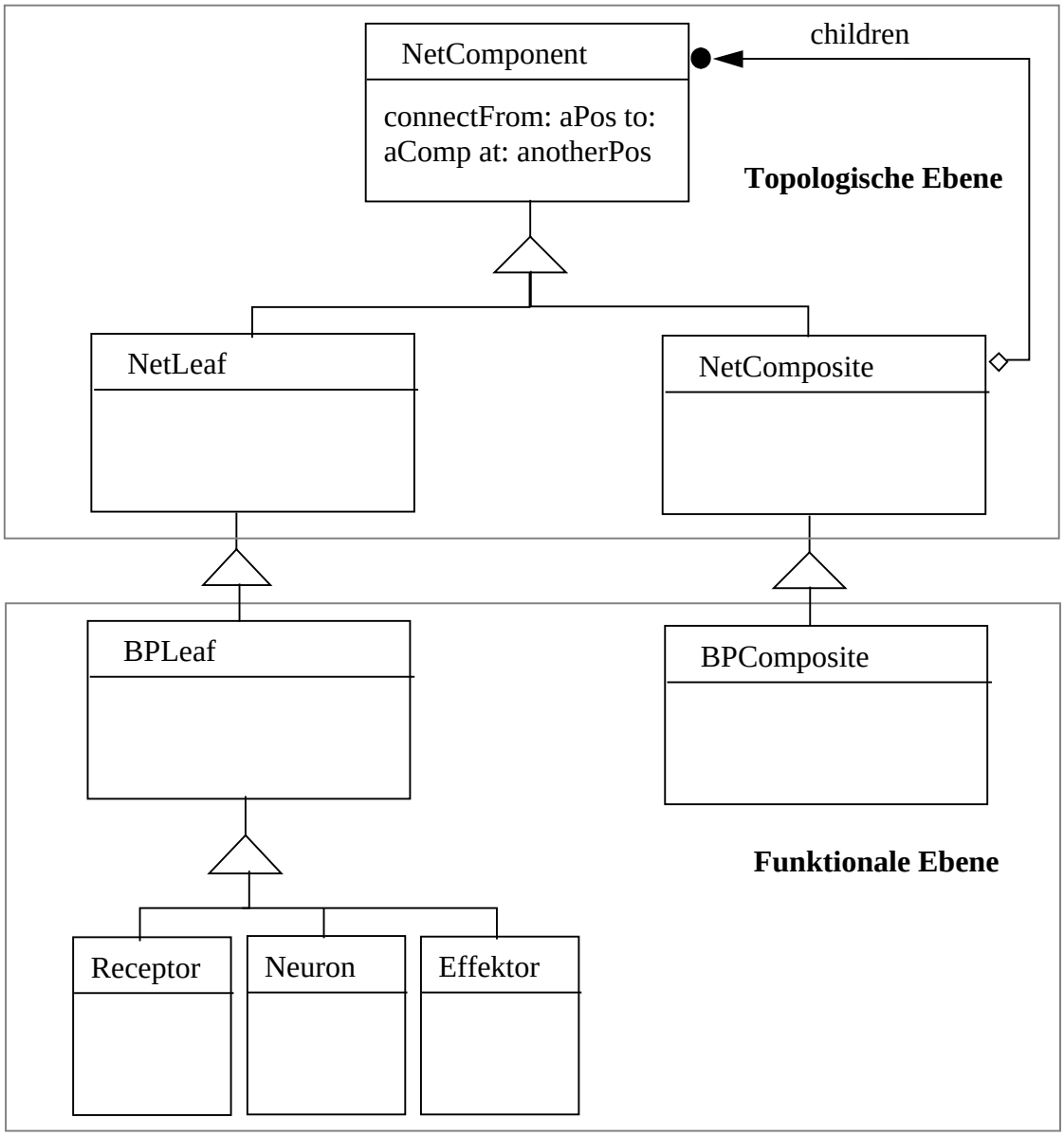


Abb. 4.2 Das Composite-Entwurfsmuster als Grundlage des Dolphin-Modells

So besitzt ein Netz auch keine Dendriten oder Axone, sondern ermittelt diese von seinen Kindern und Kindeskindern, die letztendlich aus Blättern bestehen müssen.

Die Kanäle wissen um ihre ein- und ausgehenden Netzblätter und es wird über eine Art doppelt verkettete Liste (siehe Abbildung 4.3) die gesamte Netzstruktur aufgebaut. Jedes Netzblatt hat eine geordnete Menge an Dendriten und eine geordnete Menge an geordneten Axonenmengen.

Es gibt keine „globale“ Verbindungsmatrix, in der die Netzstruktur abgebildet ist, wobei diese leicht zu erzeugen oder einzulesen wäre, um ein Interface zu externen NN-Simulatoren zu schaffen.

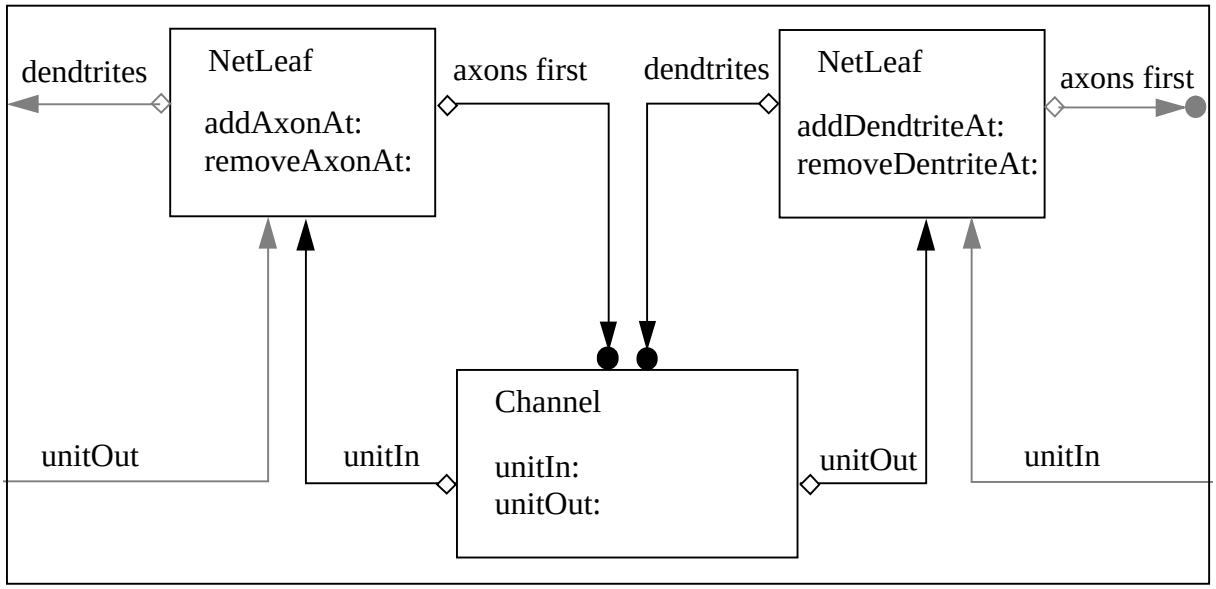


Abb. 4.3 Die Verschaltung der Netzblätter und ihrer Kanäle als doppelt verkettete Liste

Jedem Kind eines Netzes ist ein Index *n* zugeordnet, indem es das *n*. Element der geordneten Menge von Kindern dieses Netzes ist. Zusammen mit dem Namen seiner Klasse ist so jedes Element innerhalb des Kontextes seines Vaternetzes eindeutig bestimmt. Jeder Kanal ist dann eindeutig durch den Namen seiner Eingangs- und seiner Ausgangseinheit festgelegt.

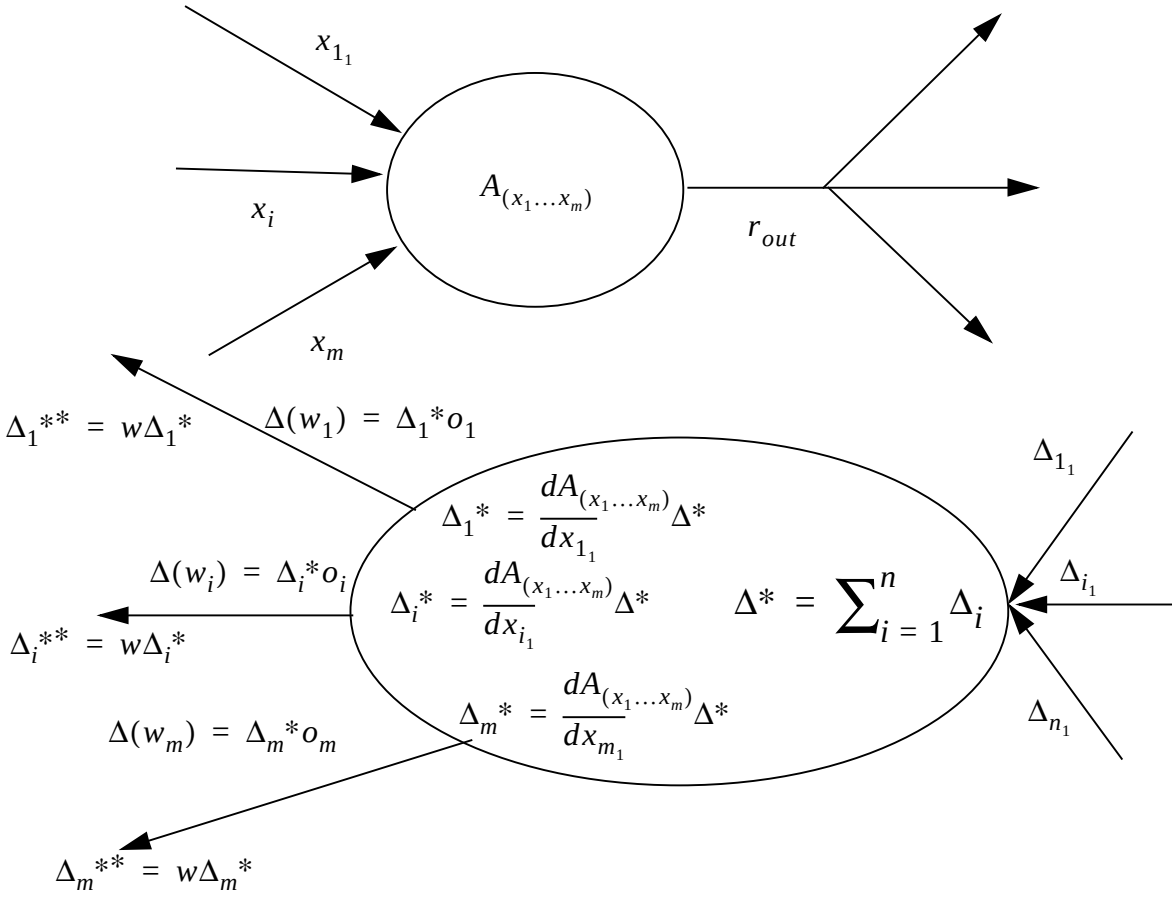
4.2.1.2 Der funktionale Aspekt des Netzwerkmodells

Um die funktionale Komponente eines speziellen Netzwerktyps zu implementieren, muß das topologisch allgemein gehaltene Framework von DOLPHIN mit Fleisch gefüllt werden. Dazu werden spezielle Unterklassen von `NetLeaf`, `NetComposite` und `Channel` gebildet, die mit den geforderten Methoden und Instanzvariablen ausgestattet werden.

4.2.1.3 Umsetzung des Backpropagation-Verfahrens

Die drei Unterklassen, die gebildet wurden, heißen `BPLeaf`, `BPComposite` und `BPChannel`. Da es mir nicht rechzeitig gelungen war, die einheitliche Betrachtung aller Blätter des BP-Verfahrens zu implementieren (siehe auch 3.5), mußte es spezielle Eingabe-, Verarbeitungs- und Ausgabeeinheiten geben. Von `BPLeaf` wurden deshalb noch die Klassen `Receptor`, `Neuron` und `Effector` abgeleitet. Es soll im Folgenden der Ablauf eines Trainingslaufes durch das BP-Verfahren mit einem Trainingssample skizziert werden, so daß klar wird, welche Instanzvariablen und Methoden von Nöten waren.

4.2.1.3.1 Der zeitliche Ablauf eines Backpropagationschrittes



Der BP-Vorwärtsschritt

(Vorbedingung: An allen Rezeptoren liegen Eingabewerte an)

1.) Rezeptor: Aktivierungsfunktion des Rezeptors auf die Eingabe anwenden

2.) Rezeptor: Ergebnis an alle Axone schicken

3.) Kanal: Eingabewert mit Gewicht multiplizieren

4.) Kanal: Ergebnis an Ausgang weitergeben

5.) Netzblatt: Wenn alle Eingänge gesendet haben, Aktivierungsfunktion auf die Eingabe an wenden

6.) Netzblatt: Wenn das Blatt ein Effektor ist, den Rückwärtsschritt einleiten

7.) Netzblatt sonst: Ergebnis aus 5.) an alle Axone schicken (weiter mit 3.)

Der BP-Rückwärtsschritt

8.) Effektor: An die Dendriten die Differenz zwischen dem gewünschten Zielwert und dem eigenen Ergebnis als Delta zurückpropagieren

9.) Kanal: Um die Gewichtsänderung des Kanals auszurechnen, muß das empfangene Delta mit der Lernrate des Kanals, dem Wert der Ableitung der Aktivierungsfunktion nach dieser Stelle und dem Eingangswert des Kanals multipliziert werden.
Das Delta, das der Kanal jetzt an seine Eingangseinheiten schickt, ist das alte Delta multipliziert mit dem Gewicht des Kanals.

10.) Netzblatt: Wenn das Netzblatt ein Rezeptor ist, nichts mehr tun.

11.) Netzblatt sonst: Wenn alle Ausgänge ihre Deltas gesendet haben, die Summe der Deltas berechnen und als neues Delta an die Eingänge schicken. (weiter mit 9.)

4.2.1.3.2 Benötigte Instanzvariablen

Zunächst sollen die Variablen des Kanals und damit von `BPChannel` betrachtet werden. Von zentraler Bedeutung ist das Gewicht eines Kanals `weight,` das als einziger freier Parameter im klassischen Backpropalgorithmus das Lernen erst ermöglicht und in 3.) und 9.) gebraucht wird. Wichtig ist auch, den Ausgabewert eines Kanals zwischenzuspeichern. Dieser wird benötigt, da die Aktivierungsfunktion in 5.) erst dann auf alle Eingaben eines Netzblattes angewendet werden kann, wenn diese vorhanden sind. Bis dahin müssen sie in `result` abgelegt werden.

Der Wert der Ableitung einer Aktivierungsfunktion `derivedResult` an einer Stelle, also an einem Kanal, kann ebenfalls direkt in diesem gespeichert werden, da er von ihm in 9.) lokal benötigt wird. Dies kann auch bereits beim Hinweg im Punkt 5.) geschehen. Wichtig ist auch die Lernrate `rate`, die in 9.) zur Berechnung der Gewichtsänderung dient und prinzipiell für jeden Kanal verschieden sein kann.

Ein Netzblatt `BPLeaf` braucht als Instanzvariablen einen Merker `result` für die Ausgabe, da dieser gleichbedeutend mit dem Eingangswert eines Kanals in 9.) ist. Auch muß sich das BPLeaf die Summe der Deltas `deltaSum` in 11.) merken, wenn sich nicht die Kanäle ihre einzelnen Deltas merken sollen.

Jedem BPLeaf wird auch eine Aktivierungsfunktion in einem ValueHolder namens `activationFunction` als Instanzvariable gegeben. Die Realisation beliebig selektierbarer, verschiedener Aktivierungsfunktionen für ein Neuron war durch den objektorientierten Ansatz einfach. Jedem Neuron kann eine Strategie [GHJV] in Form einer Aktivierungsfunktion zugewiesen werden, welche das Ausgabeverhalten bei einer bestimmten Eingabe und die Ableitung für jede Position an dieser Eingabe beschreibt.

Eine Ausnahme vom Lokalitätsprinzip, das besagt, daß jede Netzkomponente nur ihre Ein- und Ausgangskanäle kennt, wird beim BP-Algorithmus gemacht. So hat ein `BPComposite` jeweils einen `ValueHolder` (vergleiche Subject im Observer Pattern) für eine geordnete Menge von Rezeptoren und einen für eine geordnete Menge von Effektoren. Dies war notwendig, da ansonsten nie ein gesamtes Netz hätte trainiert werden können, sondern immer nur einzelne Rezeptoren und Effektoren mit den Trainingsdaten gefüttert hätten werden können. Jedem Rezeptor und jedem Effektor sind jeweils ein Index zugeordnet, der aus der Position des jeweiligen Elements in den geordneten Mengen `BPComposite receptors`, oder `BPComposite effectors` abgeleitet werden kann.

Dafür, ob weitere Instanzvariablen in `BPLeaf` oder `BPChannel` eingeführt werden müssen, ist von Interesse, wie in 5.) bzw. 11.) entschieden wird, ob alle Eingangswerte oder alle Deltas an einem BPLeaf angekommen sind.

Zunächst wurde versucht, eine Methode zu finden, die das BP-Verfahren nicht nur örtlich, sondern auch zeitlich lokal macht. BP ist deshalb zunächst nicht zeitlich lokal, da ein `BPLeaf` bei der Präsentation eines Trainingssamples für das Netz nach einem Vorwärtsschritt auch immer erst den Rückwärtsschritt abwarten muß, bevor es wieder aktiv werden kann. Erst dann können Werte wie `derivedResult`, `result` oder `deltaSum` neu gesetzt werden. Gesucht war eine Möglichkeit, das Netz gleichzeitig trainieren und als Vorhersagewerkzeug nutzen zu können. Dazu wurden alle Trainingsdaten in den Kanälen in Dictionaries (Abbildung 4.4) verwaltet, jedem Datum eines Trainingssamples wurde eine ID zugeordnet, so daß die einem Sample entsprechenden Daten immer wieder zusammengesucht werden konnten. Jedoch stellte sich heraus, daß eine solche Implementierung in der ohnehin vergleichsweise langsamen Smalltalk-Entwicklungsumgebung zu unerträglichen Antwortzeiten führte, da die Verwaltung der Dictionaries enormen Aufwand bedeutete.

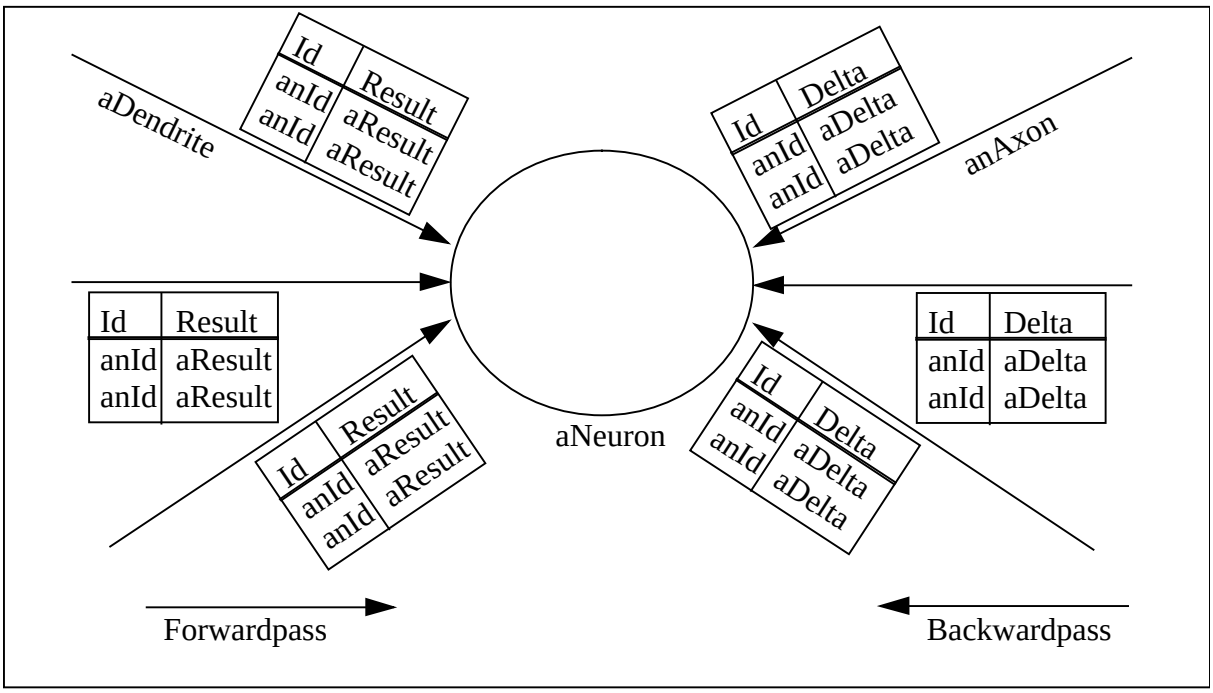


Abb. 4.4 Auflösung der zeitlichen Nichtlokalität durch Dictionaries

Bei der „Dictionary-Methode“ wurde in 5.) und 11.) nachgeschaut, ob schon alle Daten einer ID zur Berechnung des entsprechenden Schrittes vorhanden sind.

In der momentanen Implementierung wird dies über zwei Zähler erledigt, die sich als Instanzvariablen `forwardCounter` und `backpropCounter` in BPLeaf niederschlagen. Man verläßt sich hier darauf, daß nur Daten eines Samples zu einem Zeitpunkt im Netz unterwegs sind.

4.2.2 Weitere Aspekte der funktionalen Architektur

Da in Smalltalk auch Klassen Objekte sind, sind alle „im System fest verdrahteten“ Aktivierungsfunktionen als Unterklasse von `Function` implementiert worden. Dies hat den Vorteil, daß bei der Änderung der Definition einer speziellen Aktivierungsfunktion sofort alle Neuronen mit dieser Funktion weiterarbeiten. Für die Veränderung steht auch ein Funktionseditor zur Verfügung, der allerdings bisher nur Funktionen in Smalltalk-Syntax versteht. Dazu wird die Smalltalk-Klasse Compiler benutzt, mit deren Hilfe man Strings zur Laufzeit in ausführbaren Programmcode (einen `BlockClosure`) übersetzen kann. Hierfür ist es allerdings notwendig, das gesamte Entwicklungssystem von VisualWorks zur Verfügung zu haben, da in der kostenlosen Runtime-Version der `Compiler` entfernt wird. Man könnte aber auch einen der Smalltalk Compiler-Compiler wie [SLS] als Präprozessor benutzen, um die gängige mathematische Funktionsnotation in Smalltalk-Syntax zu übersetzen. Ein anderes zukünftiges Projekt könnte es dann sein, ein Mathematikprogramm zur Generierung der Ableitungen anzubinden, oder selbst einen universelles Ableitungsprogramm hinzuzufügen.

4.2.2.1 Shared Weights

Hier wurde einfach jedem Kanal (=teilender Kanal) noch eine Liste von anhängigen (=geteilten) Kanälen mitgegeben. Wird das Gewicht des Kanals geändert, nähert der teilende Kanal alle Gewichte seiner geteilten Kanäle in Abhängigkeit einer Rate an sein Gewicht an. Wird ein Kanal gelöscht, müssen alle Verweise auf ihn als geteilten Kanal mitgelöscht werden. Dazu muß der geteilte Kanal auch seinen „teilenden“ Kanal kennen. Es kann nur einen solchen teilenden Kanal für einen geteilten Kanal geben, da es widersprüchlich wäre, wenn zwei oder mehr teilende Kanäle versuchen würden, das Gewicht eines geteilten Kanals auf ihre Gewichte zu bringen. Es liegt somit eine 1:N-Beziehung zwischen teilenden und geteilten Kanälen vor. Jeder Kanal führt dazu zwei Instanzvariablen, eine (möglicherweise leere) Aggreation `SharedValueHolders`, in welcher die geteilten Kanäle verwaltet werden und eine Instanz `shareTo` in welcher, bei Bedarf, der teilende Kanal gespeichert ist.

4.2.3 Die Einbindung in HotDraw

HotDraw ist ein Framework für graphische Editoren [Bra][Joh], mit deren Hilfe zweidimensionale Zeichnungen editiert werden können. Es wurde ursprünglich von Kent Beck und Ward Cunningham entwickelt. Es bietet die Möglichkeit, eine Werkzeugleiste zur Editierung von Graphiken mit eigenen Werkzeugen zu bestücken, mit welchen sich wiederum ebenfalls selbst bestimmbare Aktionen in einer Darstellung der editierbaren Graphik auslösen lassen. Sobald ein neues graphisches Objekt in der Graphik erzeugt wird, wird einer Sammlung von Gleichungen, die die Abhängigkeiten der Objekte untereinander beschreiben, eine neue Gleichung hinzugefügt. Wird ein Objekt in der Graphik verändert, muß die Lösung des Gleichungssystems neu berechnet werden. Die aufgefrischte Graphik wird dann aus diesem gelösten Gleichungssystem bestimmt. Das Prinzip, nach dem Programme mit Hilfe solcher Gleichungen oder Constraints geschrieben werden können, wird auch in [AbSu] näher erläutert.

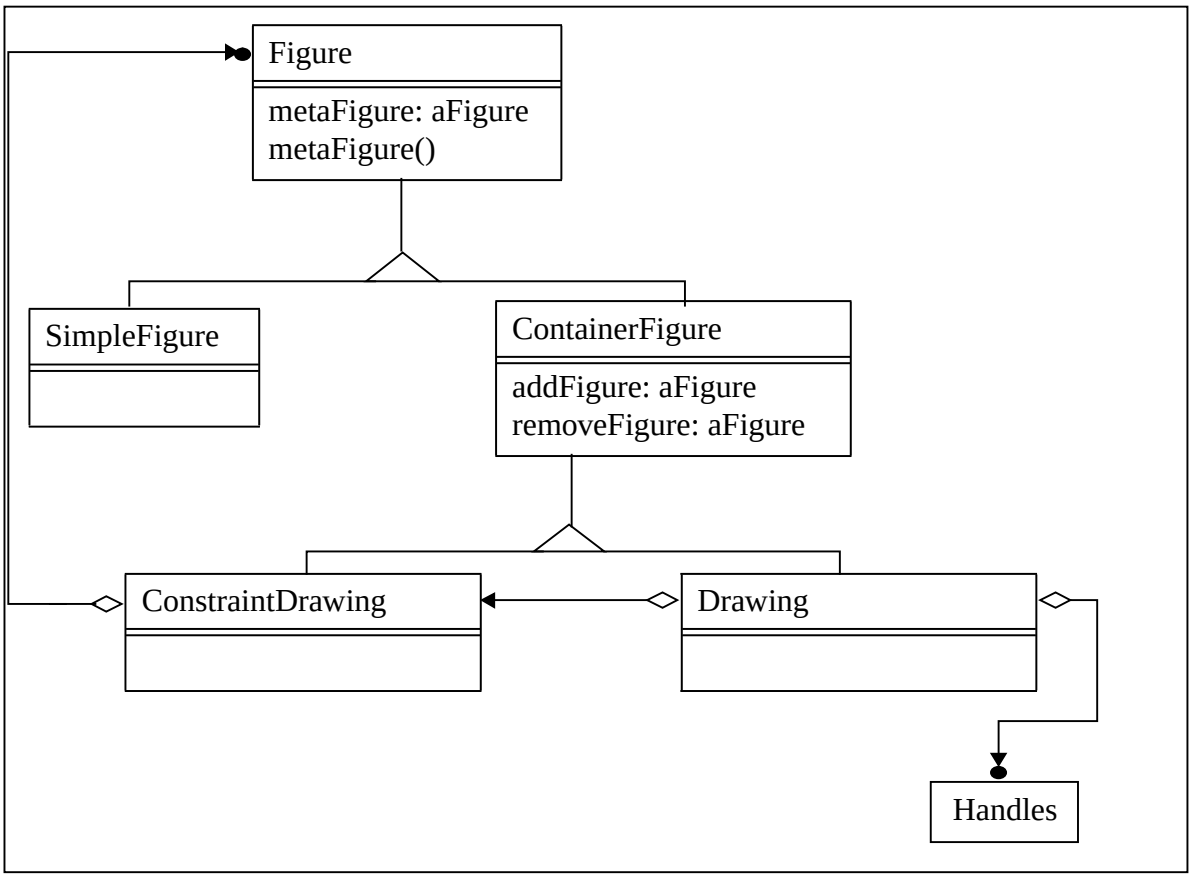


Abb. 4.5 Die Anwendung des Composite-Entwurfsmusters in HotDraw

Die Gleichungen oder auf Englisch „Constraints“ bilden eine eigene Klasse in HotDraw und könnten auch für nicht graphische Belange eingesetzt werden. Jedoch waren die verschiedenen Namen für die Klassen der Constraints eher kryptisch (`SkyBlueConstraint`, `CobaltBlueConstraint`) und die Klassen selbst leider unkommentiert. Als Anfänger wollte ich nicht zu tief in dieses Framework einsteigen, so daß ich mich gegen eine Einbettung des Netzwerkmodells in diese Constraint-Klassen entschied, obwohl es auch ein spannender (aber wahrscheinlich auch sehr langsamer) Ansatz gewesen wäre, BP über sich lösende Gleichungssysteme auszudrücken. Die Einbettung des Modells geschah anstattdessen über eine sogenannte `metaFigure`, einer Instanzvariable von `Figure`, über die die Kommunikation von einer Zeichnung zu einem Modell möglich war. Die `metaFigure` kennt die sie beinhaltende Figur nicht, so daß sie nicht in Versuchung kommen kann, einen Aspekt ihrer Darstellung zu verwenden, um ihren Zustand zu ändern.

4.2.4 Die einzelnen Ansichten von DOLPHIN

DOLPHIN sollte datenorientiert sein, so daß alle „Macht“ innerhalb einer Sitzung von der Datentabelle `DataListView` ausgeht. Wird diese geschlossen, schließen sich auch alle zu dieser Sitzung gehörenden Fenster.

Auf die Klasse NetComponent und ihre Unterklassen kann über zwei Wege zugegriffen werden, so über die `metaFigure` einer `NetFigure`, die ein Kind von `Drawing` ist, als auch über die Kinder der `metaFigure` einer `Drawing`, also über die Kinder eines Netzes `NetComposite`. So konnte sowohl schnell das Interface des Gesamtnetzes von allen anderen Komponenten erreicht werden, allerdings mußte beim Ein- und Aushängen einer `NetFigure` samt ihrer `metaFigure` darauf geachtet werden, diese `metaFigure` dem Netzmodell `NetComposite` auch als Kind, Effektor oder Rezeptor bekannt zu machen.

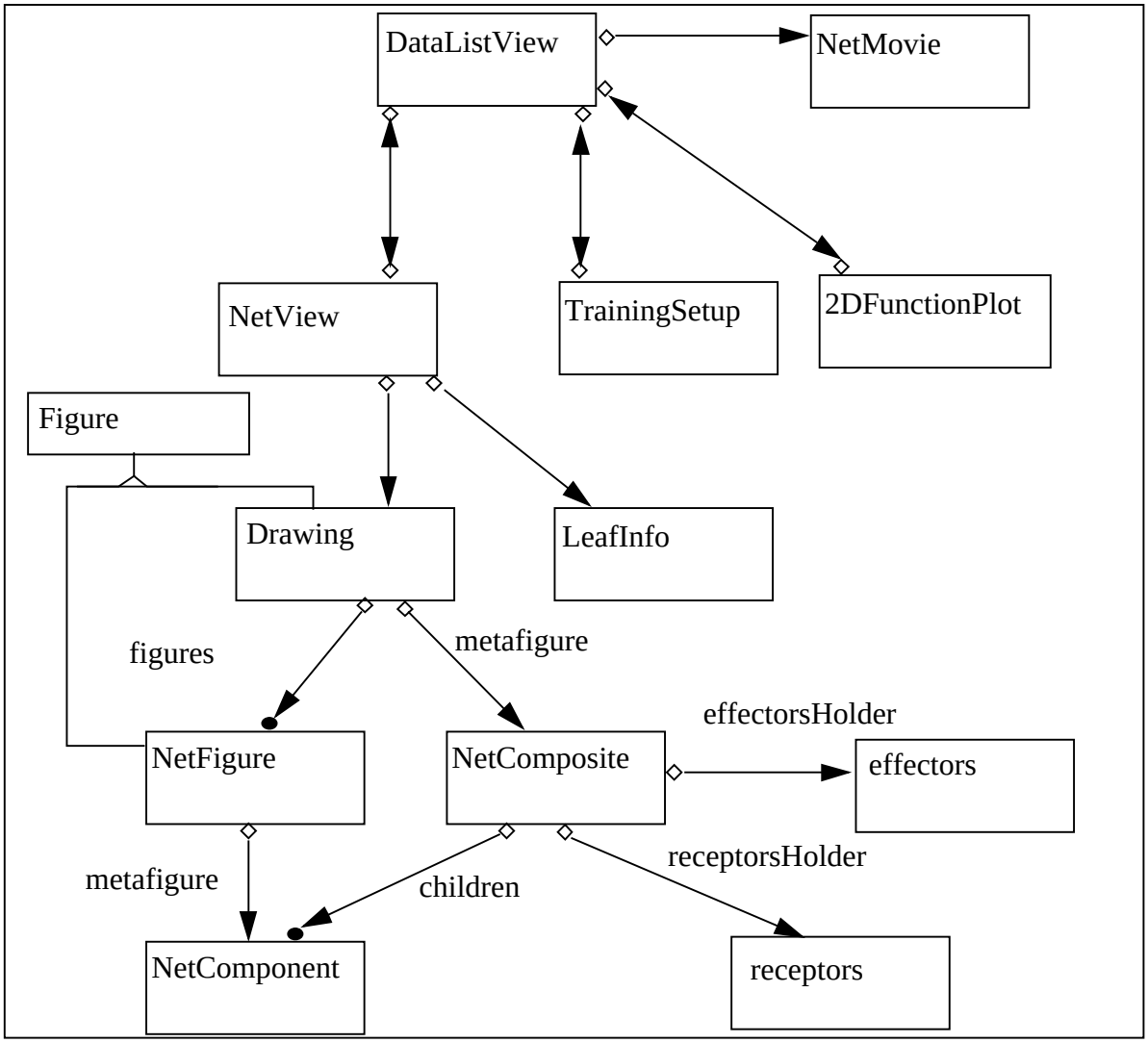


Abb. 4.6 Die Kenntnisbeziehungen zwischen Modell und Ansichten DOLPHINs

4.2.4.1 Die Tabelle

Von allen hier nicht näher aufgeschlüsselten Komponenten DOLPHINs möchte ich kurz auf DataListView eingehen, da mir der entstandene endliche Automat von gewisser Wiederverwendbarkeit zu sein scheint.

Die Datentabelle aus `DataListView` sollte in der Lage sein, verschiedene Textformate zu lesen.

Das Textformat wurde folgendermaßen spezifiziert:

Eine Datenzeile besteht aus Zahlen, die durch Kommata, Strichpunkte, oder eine beliebig lange aber nicht leere Kette von Leerzeichen oder Tabulatoren getrennt werden. Folgen mehrere Kommata oder Strichpunkte aufeinander, wird in einem gedachten Zwischenraum eine 0 angenommen. Folgt nach dem Komma oder dem Strichpunkt das Ende der Zeile oder das Ende des Files, wird ebenfalls eine 0 zwischen dem Komma und dem Ende angenommen. Steht das Komma oder der Strichpunkt am Anfang der Zeile, wird auch eine 0 zwischen dem Anfang und dem Komma angenommen. Folgt eine Kette von Leerzeichen oder Tabulatoren etwas anderem, als einer Zahl, wird diese Kette ignoriert.

Da ich zum Zeitpunkt der Programmierung dieses Teils noch nichts von Compiler-Compilern für Smalltalk wußte und mir eine Einbindung von yacc-artigen Werkzeugen in das Smalltalk-System kompliziert erschien, schrieb ich den Automaten, der Ausdrücke der obenstehenden Spezifikation erkennt, kurzerhand selber. Die Zustände des Automaten werden durch entsprechend numerierte Methoden abgebildet. Wird ein Zustand von einem anderen aufgerufen, an deren Verbindung ein Seiteneffekt ausgelöst werden soll, wird dieser Effekt vor Aufruf der entsprechenden Methode gestartet.

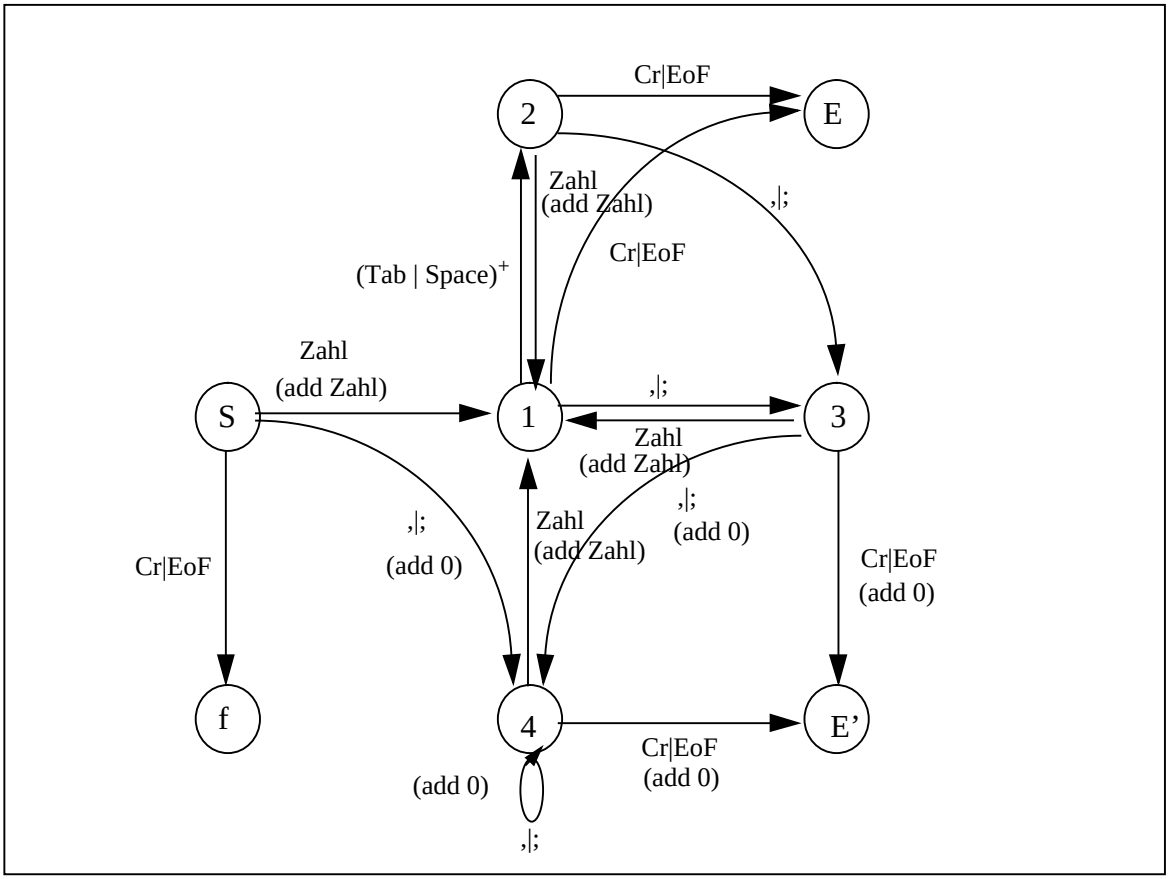


Abb. 4.7 Der endliche, minimale Automat zur Erkennung einer Zeile des gewünschten Eingabeformats

5 Bedienung

Bedienen läßt sich DOLPHIN, indem mit seiner inherenten Skriptsprache ein neues Netz erzeugt wird oder indem die graphische Oberfläche ausgenutzt wird. In beiden Fällen muß zunächst das aktuelle Image mittels `visualworks dolphin.im` gestartet werden. Neben `dolphin.im` muß dazu in dem selben Verzeichnis auch noch `dolphin.cha` und `visual.sou` vorhanden sein. Man sieht folgende Oberfläche von VisualWorks 2.5.

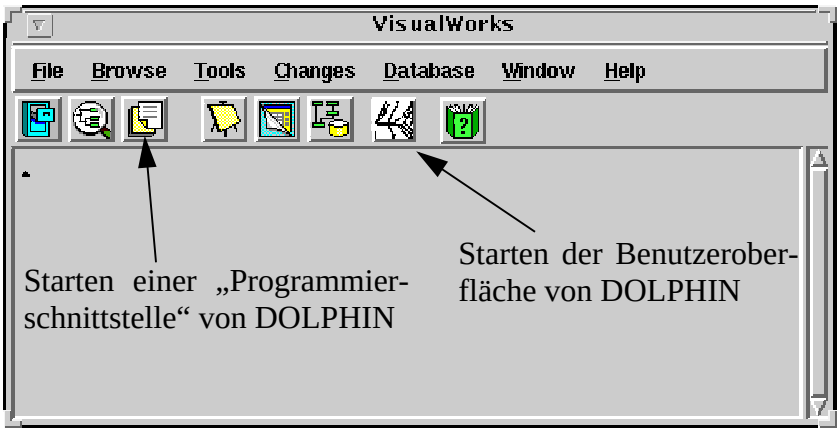


Abb. 5.1 Startfenster von DOLPHIN

5.1 Programmierschnittstellen

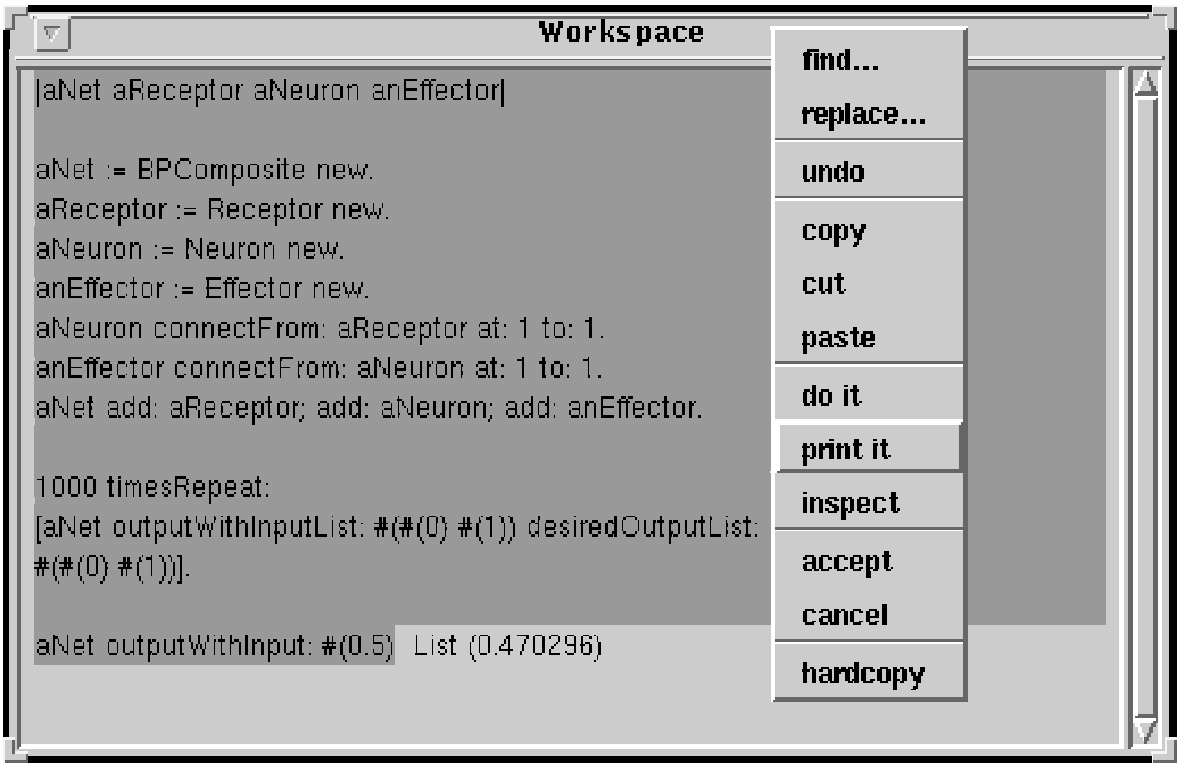


Abb. 5.2 Beispielskript einer textbasierten DOLPHIN-Sitzung

Hat man die Programmierschnittstelle von DOLPHIN geöffnet, erscheint ein einfacher Editor. wie in Abbildung 5.2. Ausdrücke in diesem Editor werden ausgewertet, indem sie erst ausgewählt werden und dann mittels der mittleren (Unix) oder rechten (Windows) Maustaste der Menupunkt `doIt` oder `printIt` aufgerufen wird.

Innerhalb des Editor stehen einem die gesamten Klassen dieses Images zur Verfügung. Instanzen dieser Klassen werden im Allgemeinen mit `new` angelegt.

Um der Instanz einen Namen zu geben, durch den man später wieder auf sie zugreifen kann, ist es wesentlich, temporäre Variablen anlegen zu können. Dies wird erreicht, indem man am Code-Anfang die später gebrauchten Variablen zwischen zwei Senkrecht-Strichen einschließt

5.1.1 Die wichtigsten Befehle des Modells von Dolphin

NetComponent ist eine abstrakte Klasse und zugleich Vaterklasse von NetLeaf und von NetComposite (vergleiche Abbildung 4.2). Abstrakte Klassen geben nur das Interface vor, das von ihren Unterklassen implementiert werden muß. Sie selbst sollen nicht instanziiert werden.Wenn eine Methode von allen Unterklassen von NetComponent implementiert ist, wird nur NetComponent als Empfänger angegeben.

`Receptor new`

> erzeugt und initialisiert einen Rezeptor mit einem freien Axon und `SumUpIdentity` als Aktivierungsfunktion.

`Neuron new`

> erzeugt und initialisiert ein Neuron mit einem freien Axon, einem freien Dendrit und `SumUpIdentity` als Aktivierungsfunktion.

`Effector new`

> erzeugt und initialisiert einen Effektor mit einem unfreien Axon, einem freien Dendrit und `SumUpIdentity` als Aktivierungsfunktion.

`BPComposite new`

> erzeugt und initialisiert ein leeres Backprop-Netz mit einer Lernrate von 0,1 und einem Biasrezeptor, an dem der Wert 1 anliegt.

`BPLeaf function: aFunction`

> ordnet einem NetLeaf eine Aktivierungsfunktion zu. Da die Funktionen als Klassen abgelegt sind, kann hier direkt der Name der Klasse angegeben werden.

`BPLeaf function`

Liefert die aktuelle Aktivierungsfunktion.

`BPLeaf result`

Liefert das letzte Ergebnis einer Berechnung, sonst nil.

`NetComponent dendrites`

Liefert alle Dendriten eines `NetComponent.`

`NetComponent unfreeDendrites`

Liefert alle unfreien Dendriten eines `NetComponente`.

`NetComponent freeDendrites`

Liefert alle freien Dendriten eines `NetComponent.`

`NetComponent axons`

Liefert alle Axone eines `NetComponent.`

`NetComponent unfreeAxons`

Liefert alle unfreien Axone eines `NetComponent.`

`NetComponent freeAxons`

Liefert alle freien Axone eines `NetComponent.`

`NetComponent channels`

Liefert eine Menge aller Kanäle des `NetComponent`, also alle Dendriten zusammen mit allen Axonen.

`NetComponent addDendrite`

Addiert einen Dendriten an ein `NetComponent.`

`NetComponent addAxon`

Addiert ein freies Axon an ein `NetComponent.`

`BPComposite updateMethod: aMethod`

Gibt an, wann die Gewichte aktualisiert werden sollen.

`aMethod` ist Element von {#updateNow,#updateLater). Mit #updateNow werden die Gewichte nach der Präsentation eines Trainingssamples aufgefrischt, was dem Online-Update entspricht. Bei #updateLater werden die Gewichtsänderungen aller Samples der Tabelle aufsummiert und erst diese Summe mit dem alten Gewicht verrechnet. Dies entspricht dem Batch-Update.

`BPLeaf initializeWeights`

Initialisiert die Gewichte der Kanäle von BPLeaf neu.

`BPComposite initializeWeights`

Initialisiert die Gewichte der Kanäle von BPComposite neu.

`BPComposite add: aComponent`

Addiert eine Komponente (Receptor; Neuron; Effector).

`BPComposite remove: aComponent`

Löscht eine Komponente.

`BPComposite receptors`

Liefert alle Rezeptoren des Netzes.

`BPComposite effectors`

Liefert alle Effektoren des Netzes.

`BPComposite outputWithInput: someInput desiredOutput: someDesiredOutput`

Trainiert das Netz mit einem korrespondierenden Ein- und Vorgabevektor. Es wird vorausgesetzt, daß die Kardinalität des Eingabevektors mit der Anzahl der Rezeptoren des Netzes übereinstimmt und die ebenso die Kardinalität des Vorgabevektors gleich der Anzahl der Effektoren des Netzes ist. Es wird die Liste der Resultate der Effektoren zurückgegeben.

`BPComposite errorWithInput: someInput desiredOutput: aDesiredOutput`

Trainiert das Netz mit einem korrespondierenden Ein- und Vorgabevektor. Es wird vorausgesetzt, daß die Kardinalität des Eingabevektors mit der Anzahl der Rezeptoren des Netzes übereinstimmt und die ebenso die Kardinalität des Vorgabevektors gleich der Anzahl der Effektoren des Netzes ist. Es wird der Fehler zurückgegeben.

`BPComposite outputWithInput: someInput`

Liefert das Ergebnis des Netzes angewandt auf den Eingabevektor `someInput.` Es wird vorausgesetzt, daß die Kardinalität des Eingabevektors mit der Anzahl der Rezeptoren des Netzes übereinstimmt.

`BPComposite outputWithInputList: anInputList desiredOutputList: aDesiredOutputList`

Trainiert das Netz mit einer Liste von korrespondierenden Ein- und Vorgabevektoren. (Siehe auch Abbildung 5.2). Es wird vorausgesetzt, daß die Kardinalität eines Eingabevektors mit der Anzahl der Rezeptoren des Netzes übereinstimmt und die ebenso die Kardinalität eines Vorgabevektors gleich der Anzahl der Effektoren des Netzes ist.

`BPComnposite rate: aRate`

Setzt die Lernrate aller Kanäle des Netzes auf eine reelle Zahl `aRate` und merkt sich diese.

`BPComposite rate`

Liefert die zuletzt eingestellte Lernrate des Netzes.

`BPComposite reinitialize`

Setzt alle Zähler des Netzes auf 0.

`NetComponent connectFrom: aNetComponent at: anAxonPosition to: aDentritePosition`

Erzeugt eine Verbindung von einer `NetComponent` an einer Ausgangsposition `anAxonPosition` zu einer anderen Komponente `aNetComponent` an einer Eingangsposition `aDentritePosition`. Die Positionen beziehen sich beide auf freie Axone bzw. Dendriten, diese beiden freien Kanäle werden bei diesem Schritt zu einem verschmolzen.

`BPComponent index`

Liefert einen Index, der zusammen mit dem Namen der Klasse (`NetComponent class`) eine eindeutige Identifizierung einer Einheit in einem Netz erlaubt.

`BPChannel weight: aWeight`

Setzt das Gewicht eines Kanals auf eine reelle Zahl `aWeight.`

`BPChannel weight`

Liefert das Gewicht eines Kanals.

`BPChannel trainable: aBool`

Macht einen Kanal trainier- oder untrainierbar.

`BPChannel trainable`

Gibt an, ob ein Kanal trainiert werden darf.

`BPChannel rate: aRate`

Setzt die Lernrate eines Kanals.

`BPChannel rate`

Liefert die Lernrate eines Kanals.

`Channel unitIn: aUnit`

Setzt die Einheit `NetLeaf`, für die der Kanal ein Axon darstellt, auf `aUnit.`

`Channel unitIn`

Liefert die Einheit `NetLeaf`, von der der Kanal die Eingabe empfängt.

`Channel unitOut: aUnit`

Setzt die Einheit `NetLeaf`, für die der Kanal einen Dendriten darstellt, auf `aUnit.`

`Channel unitOut`

Liefert die Einheit `NetLeaf`, an die der Kanal die Ausgabe weiterleitet.

`BPChannel randomWeightRange: aPositiveNumber`

Setzt das Gewicht des Kanals auf einen zufälligen Wert zwischen (`-aPositive-Number`/2) und (`aPositiveNumber`/2).

`BPChannel addSharedValueHolder: aChannel`

Erzeugt eine Abhängigkeit von `aChannel` zum Empfänger.

Das Gewicht von `aChannel` wird bei Änderung des Gewichts vom Empfänger um einen bestimmten Prozentsatz an dieses angenähert.

`BPChannel sharedValueHolders`

Liefert alle Abhängigen des Empfängers.

`BPChannel result`

Liefert das Ergebnis einer Berechnung im Kanal.

5.1.2 Bereits implementierte Aktivierungsfunktionen

Name der Funktion	Funktion
SumUpIdentity	$f(x_1,...,x_n) := \sum_{i=1}^{n} x_i$
SumUpSin	$f(x_1,...,x_n) := \sin \sum_{i=1}^{n} x_i$
SumUpCos	$f(x_1,...,x_n) := \cos \sum_{i=1}^{n} x_i$
SumUpXPower2	$f(x_1,...,x_n) := \left(\sum_{i=1}^{n} x_i \right)^2$
SumUpXPower3	$f(x_1,...,x_n) := \left(\sum_{i=1}^{n} x_i \right)^3$
SumUpXPower4	$f(x_1,...,x_n) := \left(\sum_{i=1}^{n} x_i \right)^4$
SumUpSquashWithE	$f(x_1,...,x_n) := \frac{1}{\left(1 + e^{-\sum_{i=1}^{n} x_i} \right)}$
MultiplyIdentity	$f(x_1,...,x_n) := \prod_{i=1}^{n} x_i$
WalzFunction	$f(w,i,k) := (1-k)*i+(k*w)$
Polynom-Funktion	$f(x_1,...,x_n) := \sum_{i=1}^{n} x_i^i$

Läuft der Definitionsbereich von x_1 bis x_n, bedeutet dies, daß die Funktion beliebigstellig ist.

5.2 Benutzung der Oberfläche

Die Benutzung der Oberfläche möchte ich anhand des Exklusiv-Oder Beispiels ([[Roj]] und auch Kapitel 6.1.1) erklären.

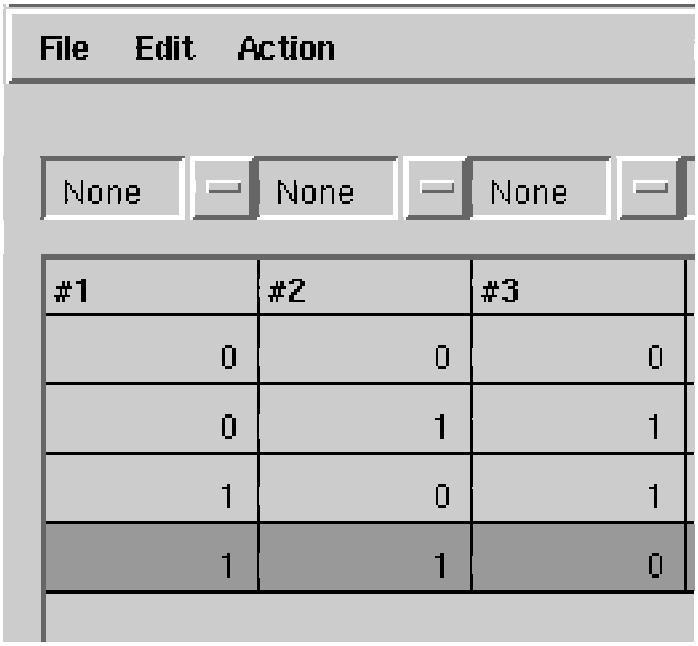


Abb. 5.3 Die Trainingsdaten in der editierbaren Tabelle

5.2.1 Öffnen Der Tabelle

Wurde auf das „Neuron“ aus Abbildung 5.1 geklickt, erscheint eine leere Tabelle. Diese kann mittels eines PopUp-Menus ausgelöst von der mittleren Maustaste um editierbare Zeilen erweitert werden.

In Abbildung 5.3 wurden bereits die Trainingsdaten eingegeben. Dabei ist zu beachten, daß in einer Zelle jeglicher Smalltalk-Ausdruck eingegeben und ausgeführt werden kann, welches innerhalb von unsicheren Betriebssystemen wie Windows 3.1 ein echter Sicherheitsmangel ist. Der Vorteil ist aber, daß so auch mathematische Berechnungen in einer Zelle durchgeführt werden können.

Eine Zeile dieser Tabelle steht für einen Trainingssample. Ein Trainingssample der Länge k setzt sich zusammen aus einer Inputliste $i:=(i_1...i_m)$ und einer Vorgabenliste $d:=(d_1...d_n)$ mit $k:=m+n$. Gesucht ist das Netz N, das für alle Samples dieser Tabelle möglichst gut die Funktion $N(i_1...i_m)=(d_1...d_n)$ beschreibt.

Die Daten einer Tabelle können auch mittels File>>Load Data eingelesen und mit File >> Save Data abgespeichert werden. Momentan ist die Zeilenlänge eines Datenfiles auf maximal 20 Daten beschränkt. Zu den Dateiformaten, die gelesen werden können, siehe auch 4.2.4.1.

Die Tabelle dient als Kontrollzentrum von DOLPHIN, wenn sie mittel File>>Exit geschlossen wird, werden auch alle zu dieser Dolphin-Sitzung gehörenden Fenster geschlossen.

Der nächste Schritt besteht darin, den jetzt vorhandenen Daten ein Netz zuzuordnen. Dazu wird ein Editorfenster für das Netz geöffnet. Der entsprechende Befehl wird ausgelöst im Menupunkt Action>>Connect To Net.

5.2.2 Der Netzwerkeditor

Mit Hilfe dieses Editors ist es möglich, Effektoren, Neuronen und Rezeptoren zu erzeugen und diese miteinander zu verbinden. Ebenso können die Netzblätter sowie ihre Verbindungen wieder gelöscht werden. Alle Netzkomponenten sind im Editor beliebig verschiebbar. Die Eigenschaften aller Netzkomponenten können eingesehen und verändert werden. Der Editor wirkt gleichzeitig als Sicht auf das Netz, indem die Gewichte sowohl durch Farbgebung als auch durch einen Kommentar mit dem aktuellen Wert visualisiert werden.

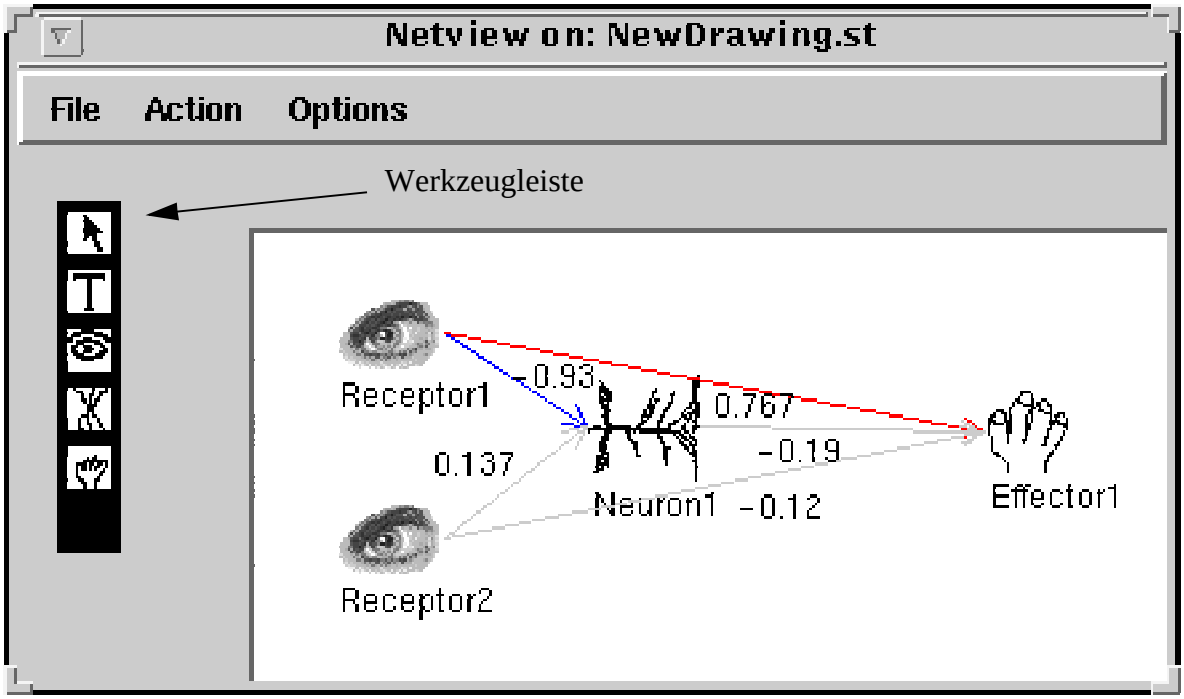


Abb. 5.4 Der Netzwerkeditor mit Netz zum Lösen des XOR-Problems

Die Werte der Komponenten sind auf zwei Arten einsicht- und veränderbar. So kann mittels des Popupmenupunktes `inspect` auf alle Instanzvariablen einer Komponente zugegriffen werden.

Über den Popupmenupunkt `properties` können bestimmte Eigenschaften bequemer eingestellt werden, so ist es möglich die Aktivierungsfunktion eines Netzblattes aus einer Liste auszuwählen.

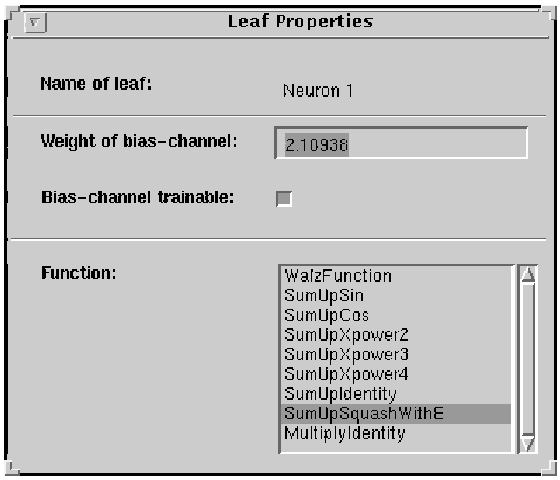


Abb. 5.5 Eigenschaftseditor eines Netzblattes

5.2.2.1 Erzeugen eines Netzblattes

Rezeptoren, Neuronen und Effektoren werden mittels den entsprechenden Icons der Werkzeugleiste erzeugt. Nach der Selektion des gewünschten Blattes auf der Werkzeugleiste können beliebig viele Instanzen dieses Blattes durch Klicken in den Editorraum erschaffen werden, so ist es sinnvoll, erst alle gewünschten Blätter zu erzeugen, bevor mit der Verbindung der Blätter begonnen wird.

Der editierbare Raum kann durch Vergrößern des Fensters ebenfalls vergrößert werden. Hat das Fenster seine Maximalgröße erreicht, stehen zusätzlich horizontale und vertikale Scrollbars zur Verfügung, die den Bereich noch erweitern können.

Beim Erzeugen eines Blattes ändert sich automatisch der Inhalt der Menus über jeder Spalte der Datentabelle, indem das entsprechende Blatt in die jeweilige Liste hinzugefügt wird.

5.2.2.2 Verbindung der Blätter

Werden Rezeptoren oder Neuronen im editierbaren Raum mittels des Selektionspfeils, dem obersten Werkzeug aus der Werkzeugleist, ausgewählt, erscheint ein kleines Quadrat an ihrem rechten Rand. Von diesem aus lassen sich durch Drücken der linken Maustaste Pfeile zu Neuronen und Effektoren ziehen.

Gelangt die Pfeilspitze in die Nähe eines potentiellen Partners, zieht dessen Magnet die Spitze an seinen Verbindungspunkt, falls noch keine so gerichtete Verbindung zwischen diesen beiden Blättern vorhanden ist. Wird dann die Maustaste losgelassen, entsteht eine Verbindung zu diesem Blatt. Da das Netz nicht in Schichten aufgebaut wird, können auch, wie in Abbildung 5.4 zu sehen ist, alle Rezeptoren und Neuronen mit allen Neuronen und Effektoren verbunden werden. Um gleichzeitig mehrere Ausgangseinheiten mit mehreren Eingangseinheiten zu verbin-

den, können die Eingangseinheiten mittels eines Lassos ausgewählt werden, die Ausgabeeinheiten werden mit Hilfe der Shift-Taste und der Maus selektiert. Über den Popupmenupunkt `Connect to selected figures` werden dann die entsprechenden Verbindungen gezogen. Zirkuläre Verbindungen werden noch nicht vom System abgefangen, obwohl sie bei dem einzigen bisher implementierten Lernalgorithmus durch Backpropagation keinen Sinn machen.

Das Gewicht des entstandenen Kanals wird mit einem Zufallswert in einem vorgegebenen Bereich initialisiert.

5.2.2.3 Löschen der Verbindungen und der Blätter

Ein Blatt oder ein Kanal muß zum Löschen zunächst wieder mittels des Selektionspfeils ausgewählt werden. Gelöscht wird er dann durch die „DEL“-Taste oder durch den Befehl `delete`, der, nach erfolgreicher Selektion der Komponente, über ein Popup-Menu, ausgelöst durch die mittlere Maustaste, aufgerufen werden kann.

5.2.2.4 Shared Weights

Zwischen den Gewichten der Kanäle können Abhängigkeiten in Form von Shared Weights gebildet werden. Dazu wird nach der Selektion eines Kanals über das Popup-Menu der Befehl `share to:...` aufgerufen. Das Gewicht des nächsten Kanal, der selektiert wird, ist dann abhängig vom Gewicht des Ursprungskanals.

5.2.2.5 Laden und Speichern eines Netzes

Ein erzeugtes Netz kann auch abgespeichert und wieder neu geladen werden. Dabei wird sowohl die graphische Repräsentation als auch das Modell des Netzes persistent gemacht bzw. wieder eingelesen. Voraussetzung dafür ist allerdings, daß sich nach dem Speichern eines Netzes nicht die ihm zu Grunde liegenden Klassen geändert haben.

5.2.2.6 Ändern des Gewichtsinitialisierungsbereichs

Der Bereich, in dem Gewichte eines neuen Kanals initialisiert werden, läßt sich über `Options>>Change initial weight range` einstellen. Der Bereich ist immer symetrisch um den Nullpunkt gelegen.

5.2.2.7 Ändern des Bias

Alle Neuronen und Effektoren des Netzes werden beim Erschaffen automatisch an ein Biasneuron angehängt. Dieses ist für das ganze Netz dasselbe und liefert als Defaultwert eine 1 an alle seine angehängten Blätter, wobei dieser Ausgabewert noch mit dem Gewicht des Biaskanals mutlipliziert wird. Den Ausgabewert des Biasneurons kann man unter Options>>Change bias ändern.

5.2.2.8 Initialisierung der Gewichte

Ruft man `Action>>Forget weights` auf, werden alle trainierbaren Kanäle des Netzes mit neuen Zufallszahlen innerhalb des in 5.2.2.6 beschriebenen Bereichs geändert.

5.2.3 Daten mit den Konnektoren verbinden

Um einen Konnektor mit einer Spalte zu verbinden, muß sein Name in dem Popup-Menu über der entsprechenden Spalte selektiert werden (Abbildung 5.6). Dabei wird momentan noch nicht berücksichtigt, daß immer nur eine Spalte mit einem Rezeptor oder Effektor verbunden werden kann. Auch läßt sich eine Spalte in der jetzigen Version von DOLPHIN noch nicht mit mehreren Konnektoren verbinden.

Die Verbindung einer Spalte mit dem Effektor `Eff-j` besagt, daß diese Spalte die Elemente d_j aller Trainingssamples $((i_1...i_m),(d_1...d_n))$ enthält, wird sie mit einem Rezeptor `Rec-j` verknüpft, beinhaltet sie entsprechend die Elemente i_j. Es lassen sich auch die Ergebnisse von Neuronen oder Effektoren mit einer leeren Spalte verbinden. Dazu selektiert man über einer leeren Spalte `Neu-j` respektive `Out-j`. Wird die Tabelle dann beim Training aufgefrischt, werden die aktuellen Ergebnisse dieser Blätter in die entsprechenden Spalten geschrieben.

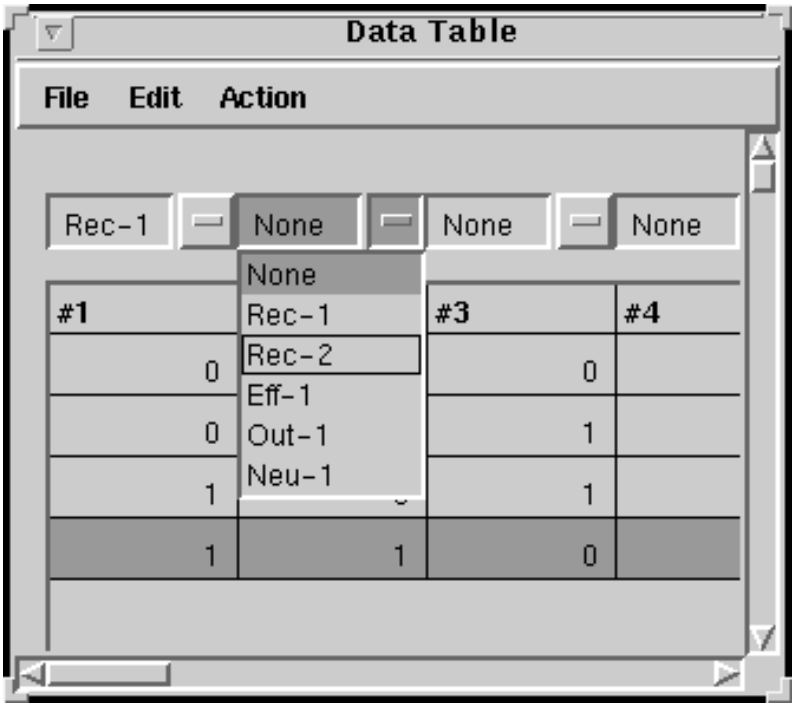


Abb. 5.6 Die Verbindung der Daten mit dem Netz

Alle Konnektoren (Rezeptoren und Effektoren) müssen mit einer Spalte der Tabelle verbunden werden, bevor mit dem Training des Netzes begonnen werden kann. Das Training des Netzes kann dann mittels `Action>>Train net` gestartet werden.

5.2.4 Das Trainingssetup

Beim Trainingssetup (Abbildung 5.7) ist generell nur die linke Spalte der Input-Zeilen vom Benutzer bedienbar. In der rechten Spalte werden die aktuellen Werte des Netzes angezeigt. In der linken Spalte werden die Abbruchkriterien spezifiziert, nach welchen der Lernvorgang gestoppt werden soll. Der Lernvorgang wird abgebrochen, wenn eines der Kriterien erfüllt ist.

Es kann als Abbruchkriterium in `Iterations` eine maximale Epochenzahl angegeben werden, in `Error` der minimale Fehler und in `Error change` die minimale Fehleränderung. Um einen endlosen Lernvorgang zu erreichen, kann man bei Iterations auch 0 angeben.

Um den Trainingslauf zu starten, muß der Train-Button gedrückt werden, um ihn anzuhalten, der Stop-Button. Der Lauf kann dann durch erneutes Drücken des Train-Buttons fortgesetzt werden. Im untersten Input-Fenster der linken Spalte `Update every...` kann angegeben werden, nach wieviel Iterationen die verschiedenen Sichten auf das Netz aufgefrischt werden sollen. Mittels des Actual-Buttons werden alle Sichten auf das Netz aktualisiert. Wenn die Effektoren des Netzes mit leeren Spalten verknüpft wurden, kann das Netz nicht trainiert werden, jedoch sind mittels `Train` oder `Actual` die Ausgaben des Netzes ermittelbar und erscheinen in den eventuell vorhandenen `Out-x` Spalten der Tabelle.

Im Menupunkt `Options` sind die Lernrate des Netzes und die Update-Methode der Gewichte bestimmbar. Von dort aus läßt sich auch die Anzeige des Fehlerverlaufs starten.

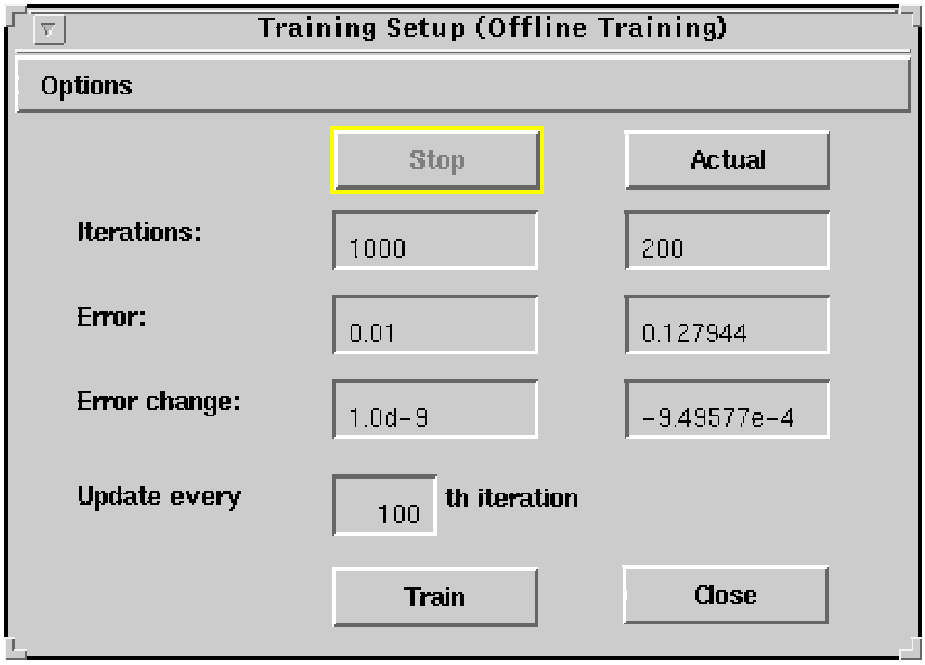


Abb. 5.7 Trainingssetup

5.2.4.1 Die Fehlerkurve

Der Fehler wird nur alle dann ausgerechnet, wenn die Trainingsansichten aufgefrischt werden.

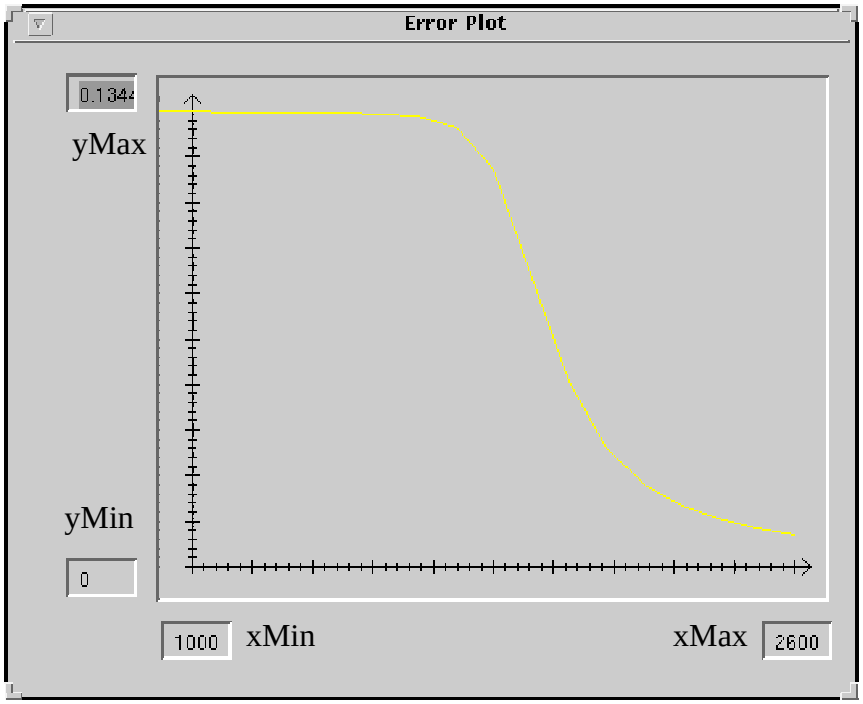


Abb. 5.8 Ansicht der Fehlerkurve des XOR-Problems

Dies geschieht entweder mit dem Ablauf eines `update every...` Zyklus oder mittels des Actual-Buttons.

Das Trainingssetup merkt sich die Fehlerpunkte, so daß die Ansicht der Fehlerkurve jederzeit geöffnet oder geschlossen werden kann. Die Ansicht der Fehlerkurve stellt sich automatisch auf den Bereich des Fehlers ein, jedoch sind xMin, xMax und yMin ,yMax frei wählbar. Durch Verkleinern des Bereichs kann somit auch noch das Verhalten der Kurve bei stark gesunkenem Fehler beobachtet werden. Der Fehler ist zur Zeit noch nicht normierbar.

5.2.5 Das rezeptive Feld

Die Darstellung des rezeptiven Feldes eines Netzes wird aus der Datentabelle über den Menupunkt `Action>>Evaluate net` aufgerufen.

Mit dem rezeptiven Feld läßt sich eine dreidimensionale Funktion (x,y,f(x,y)) darstellen. f(x,y) wird dazu in einen Farbwert übersetzt. Dabei steht die Farbe Rot für den maximal und die Farbe Blau für den minimal möglichen Wert, der dargestellt können werden soll. Schwarz und Weiß werden die Punkte gezeichnet, die oberhalb bzw. unterhalb dieser Grenzen liegen. Die gewünschten Ausgabewerte können, falls sie innerhalb des Darstellungsbereiches liegen, mit in das Feld eingezeichnet werden. Der maximale Wert der Vorgabe wird dabei durch ein rotes Rechteck und der minimale Wert der Vorgabe durch einen blauen Kreis symbolisiert. In der linken Tabelle werden zwei Rezeptoren bestimmt, in welche die X- und die Y-Werte des ausge-

wählten Bereichs als Eingabe in das Netz gesteckt werden. Der Effektor, dessen Ausgabe gezeichnet werden soll, wird über das Popup-Menu oberhalb der linken Tabelle bestimmt. Die Eingangswerte der nicht mit x oder y verknüpften Rezeptoren werden ebenfalls in der linken Liste gesetzt.

Die Ansicht des rezeptiven Feldes hat zwei verschiedene Betriebsmodi, die unter `Options>>Turn on/off automatic zoom` eingestellt werden können.In einem Fall ermittelt sie die Grenzen des darzustellenden Bereiches bei jeder Auffrischung selbständig, im anderen Fall können die Grenzen beliebig eingestellt werden. Dies geschieht analog zu 5.2.4.1 in xMin,xMax,yMin,yMax,zMin und zMax, wobei die z-Werte den Farbverlauf bestimmen.

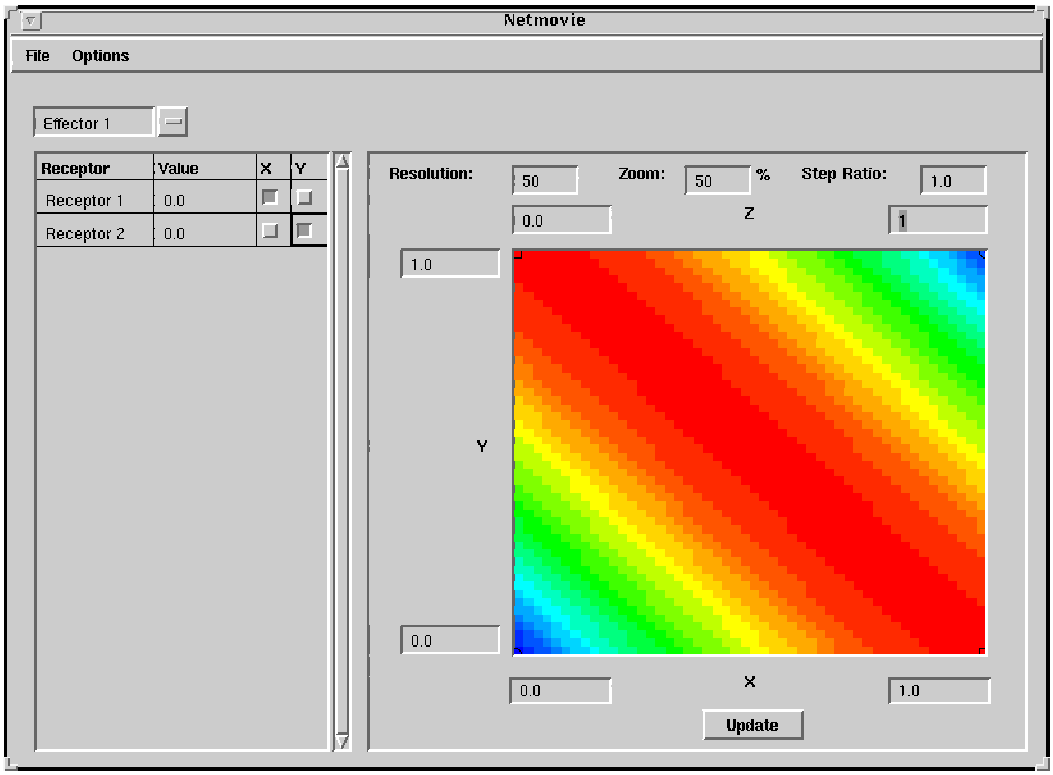


Abb. 5.9 Das rezptive Feld nach erfolgreichem Training des XOR-Problems

Zusätzlich kann noch die Auflösung des Feldes in `Resolution` und ein Zoomfaktor in `Zoom` eingestellt werden. Es kann auf dem Feld mittels der Cursortasten umhergewandert werden, dabei wird der sichtbare Ausschnitt um die Größe des Bereichs multipliziert mit den Inhalt von `Step Ratio` verschoben. Gezoomt kann auch über die Tastatur werden, PageUp zoomt um den Faktor aus `Zoom` heraus, PageDown entsprechend hinein.

5.2.6 Die Netzfunktionen

Um diese Ansicht aufzurufen, löst man in der Datentabelle den Menupunkt `Action>>Display Functions` aus. Mit ihr besteht die Möglichkeit, die Abhängigkeit von zwei Spalten

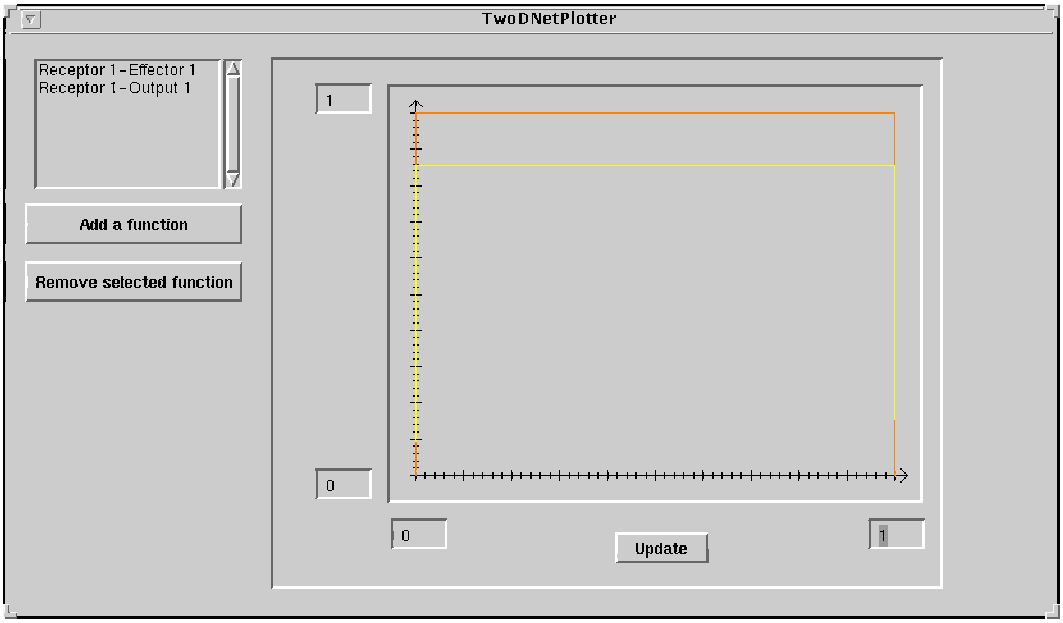


Abb. 5.10 Zwei „Netzfunktionen“ nach dem Training von XOR

aus der Datentabelle zu visualisieren. Die Spalten werden durch einen Dialog ausgewählt, der durch Drücken von `Add a function` ausgelöst wird.

6 Ergebnisse

Zunächst wurde DOLPHIN an klassischen Beispielen getestet, auch zur Kontrolle ob der Backpropagation-Algorithmus korrekt implementiert wurde. Desweiteren kann man nur an überschaubaren Problemen, die mit beiden Methoden angegangen werden können, beurteilen, ob der hybride Ansatz eine Verbesserung mit sich bringt. Der Einsatz des Programms für das geschilderte Projekt zur Vorhersage von Profilen bei Stahlwalzwerken begann in der Endphase dieser Arbeit.

6.1 Experimente

6.1.1 XOR

Das XOR-Problem für künstliche neuronale Netze ist vor allem deswegen zum Klassiker geworden, weil es als Beispiel im bahnbrechenden Artikel über Backpropagation in [RuCl] benutzt wurde. Dort war es ausgesucht worden, weil es sich nicht in einem Netz mit einem einzelnen Neuron und einer gängigen Squashingfunction lösen läßt, und dort Backpropagation für mehrschichtige Netze erklärt wurde. Ich möchte zunächst den klassischen XOR-Versuch mit DOLPHIN, an Hand dessen schon in Kapitel 5.2 die Bedienung beschrieben wurde, hier noch einmal zusammenfassen, um dann zu zeigen, wie sich das Problem bereits mit einem Neuron mit einer speziellen Aktivierungsfunktion lösen läßt.

6.1.1.1 Der klassische Aufbau

Ein Screenshot für den Aufbau des klassischen XOR-Problems aus [RuCl] findet sich bereits in Abbildung 5.4. An dem inneren Neuron und den beiden Effektoren waren Bias-Kanäle zugelassen und trainierbar. Die Aktivierungsfunktion des inneren Neurons und des Effektors war `SumUpSquashWithE`, die der Rezeptoren `SumUpIdentity`. (Zur Erläuterung der Funktionen siehe Kapitel 5.1.2).
Die restlichen Trainingsparameter waren:

Lernrate	0.5
Auffrischungmethode	Batch
Fehler	<=0,1

Tab. 6.1 XOR nach [RuCl] mit DOLPHIN

Der Verlauf des Trainings wird am besten mit der Beschreibung der Änderung des des rezepti-

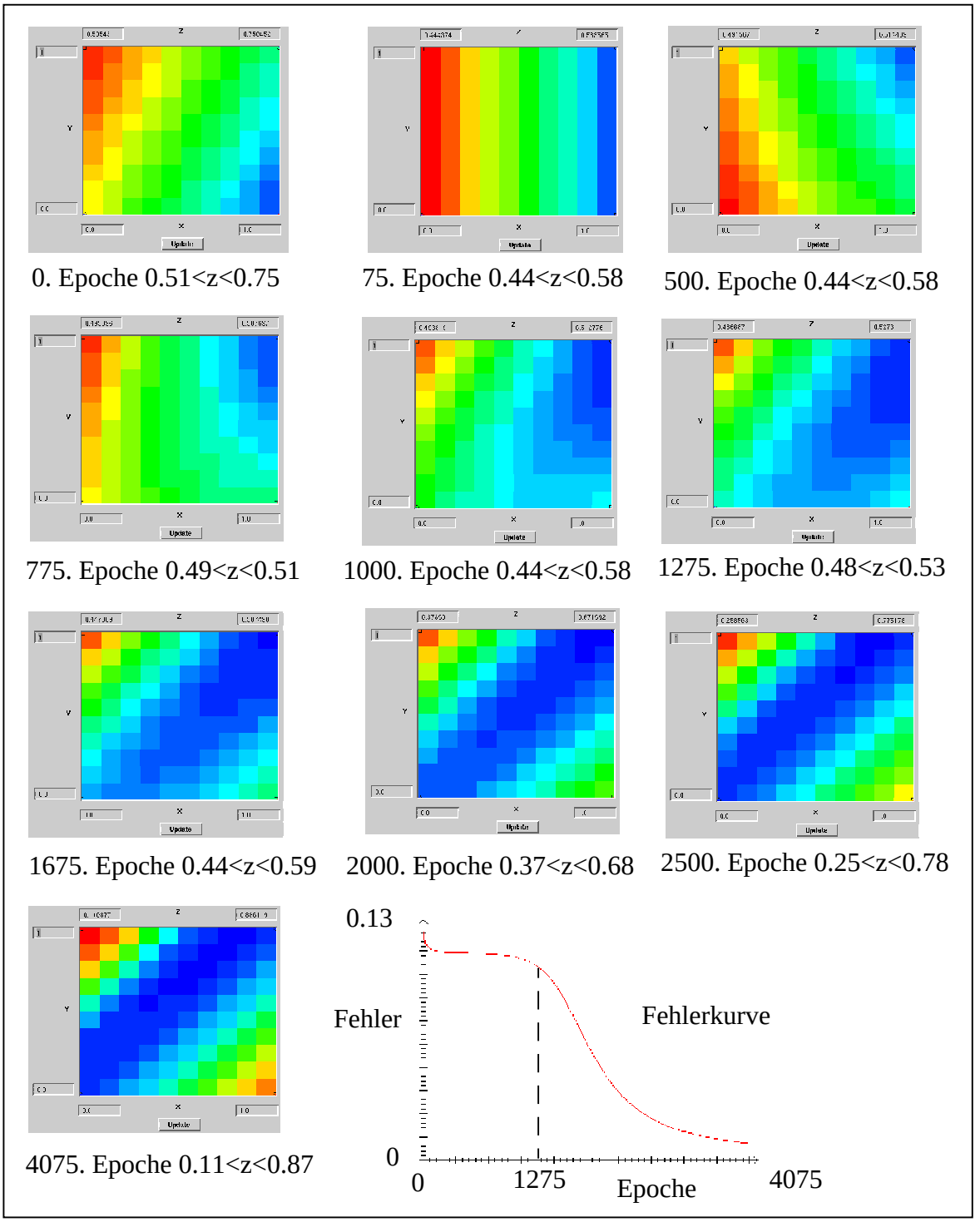


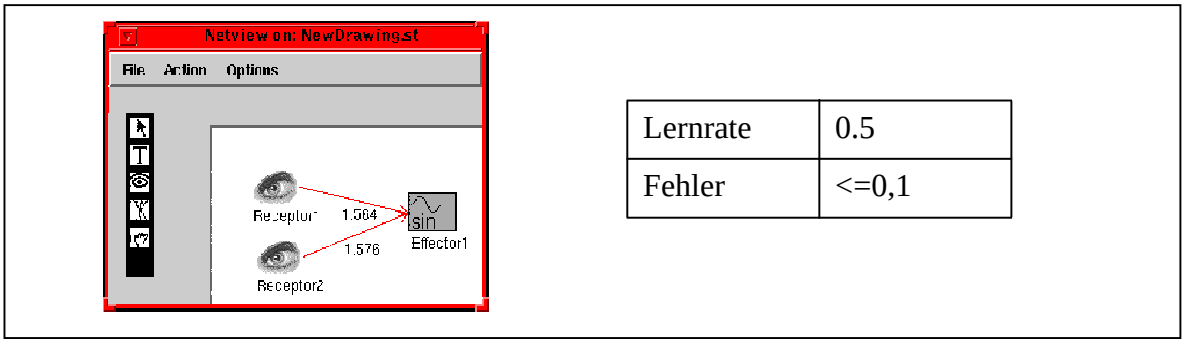
Abb. 6.1 Die Entwicklung des rezeptiven Feldes mit der entsprechenden Fehlerfunktion beim klassischen XOR-Netz

ven Feldes angegeben. Man erkennt, wie bei Epoche 1275 das Ergebnis von (0,0) und das von (1,0) sehr ähnlich (türkis) werden. Sobald der Wert bei (1,0) höher wurde, als der bei (0,0), stimmte das „Größenverhältnis“ der Ausgabewerte erstmals mit dem der Trainingspunkte überein, und es konnte bis zu einer Sättigung rasch gelernt werden. Es wurden 4075 Iterationen benötigt, um den Fehler von 0,1 zu unterschreiten.

Interessant ist auch, daß in Abbildung 5.9 die andere von zwei Lösungen des XOR-Problems bei dieser Netztopologie gefunden wurde.

6.1.1.2 Ein „hybrider“ Aufbau

Bei dieser Versuchsreihe wurde keine versteckte Schicht benutzt, es gab nur einen Effektor mit `SumUpSin` als Aktivierungsfunktion. Die Biaskanäle des Effektors wurden untrainierbar gemacht und deren Gewicht auf 0 gesetzt.



Lernrate	0.5
Fehler	<=0,1

Abb. 6.2 Ein kleines Netz für XOR

Einmal wurden die Gewichte im Batchverfahren aufgefrischt, beim anderen wurde das Onlineverfahren angewendet..

6.1.1.2.1 Batch-Update

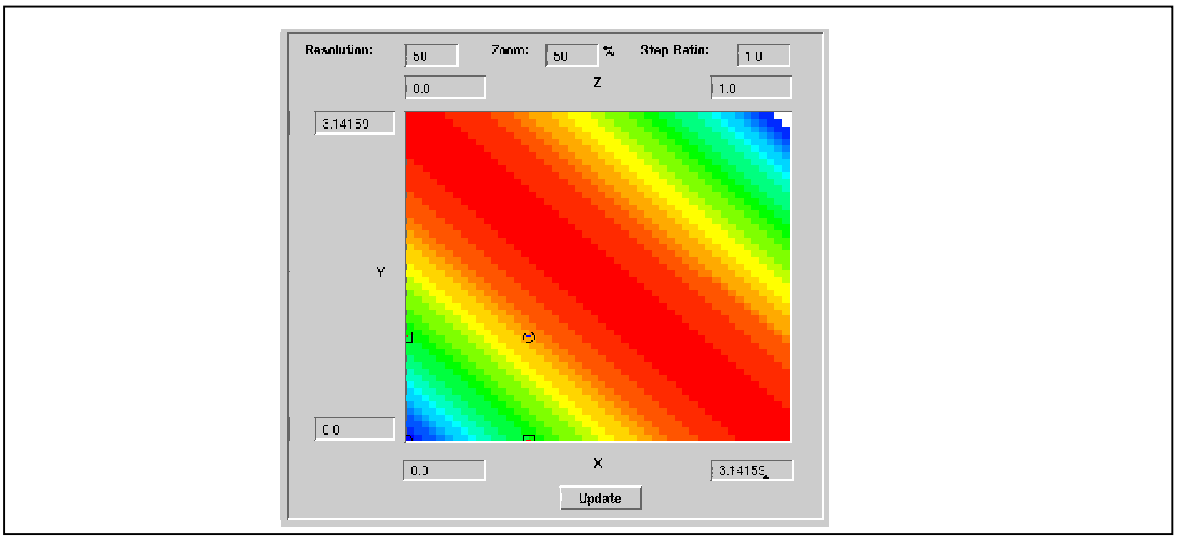


Abb. 6.3 Rezeptives Feld eines lokalen Minimums bei Batch-Update

In einer Versuchsreihe mit Batch-Update gelang es dem Netz nur in vier von zehn Läufen die Vorgabe zu lernen. Das erfolgreiche Training erfolgte nach ungefähr 200 Iterationsschritten.

In den anderern sechs Fällen blieb der Trainingsvorgang in einem lokalen Minimum hängen.

Im Folgenden möchte ich kurz an Hand des resultierenden rezeptiven Feldes aus Abbildung 6.3 meine Interpretation dieses lokalen Minimums geben.

Der Punkt (0,0) unten links wird immer im „blauen Bereich“ des rezeptiven Feldes bleiben, da beim Eingabevektor (0,0) nichts anderes als wieder 0 als Ausgabe erreicht werden kann. Wenn sich die Punkte in die horizontale Richtung ausdehnen, steht zwar (0 ,1) und (1,0) etwas besser da, was aber durch das deutlich schlechtere Abschneiden von (1,1) wieder kompensiert wird, der dann plötzlich in einem noch „heißeren“ Bereich gelandet ist. Ziehen sich die Punkte horizontal zusammen, ist zwar (1,1) in einem kälteren Bereich gelandet, jedoch sind (0 ,1) und (1,0) ebenfalls mit in den kalten Bereich zurückgerutscht. Die drei Gewichtsänderungen eines Kanals pro Epoche heben sich somit gegenseitig auf. Analog kann man auch für die vertikale Richtung argumentieren.

6.1.1.2.2 Online Verfahren

Auch hier wurde das Netz zehnmal trainiert, im Schnitt kam es nach ungefähr sieben Schritten unter einen Fehler von 0,1.

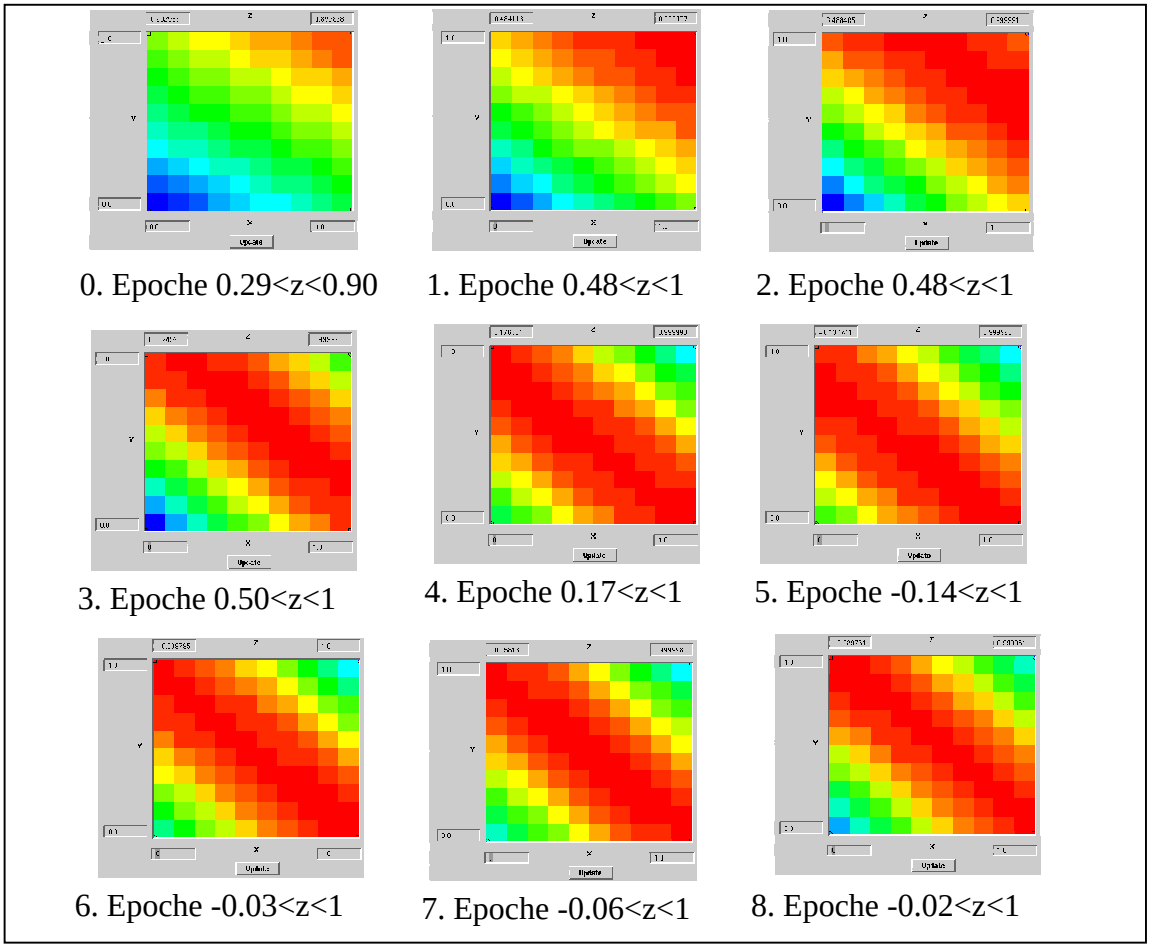


Abb. 6.4 Die Entwicklung des rezeptiven Feldes beim hybriden XOR-Netz mit Online-Update

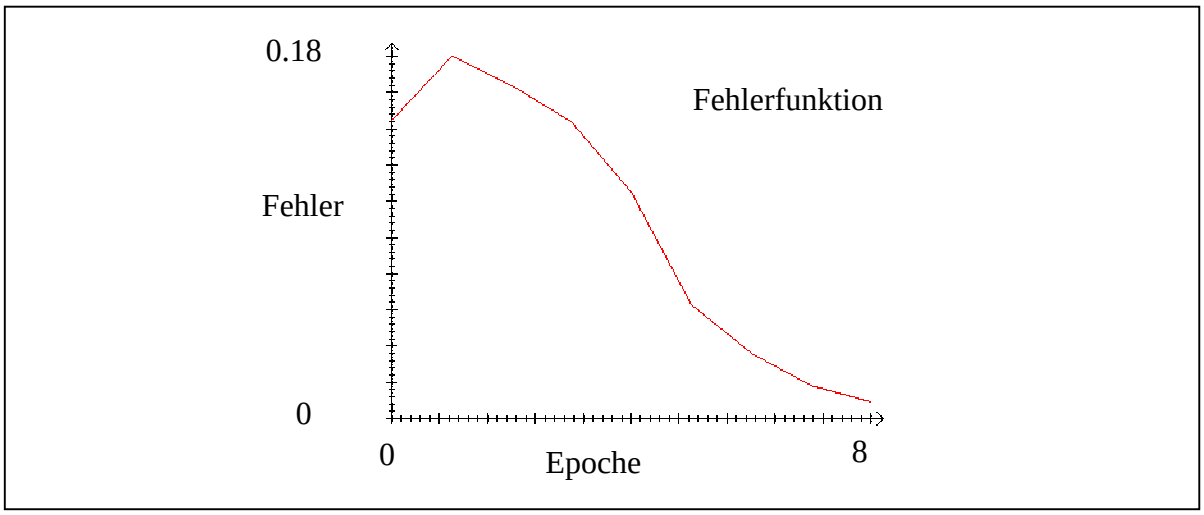


Abb. 6.5 Die Entwicklung der Fehlerfunktion beim hybriden XOR-Netz

Es hat im Gegensatz zu einem Training mit Batch-Update die Chance, den Gesamtfehler kurzzeitig zu erhöhen, wovon bei Epoche 1 des oben gezeigten Beispiellaufs sofort Gebrauch gemacht wurde. Der Fehler hat sich dort erhöht, da die Erhöhung der Ausgabe bei (1,1) von 0.9 auf 1 nicht durch die Verringerung der Ausgaben bei (0,1) und (1,0) wettgemacht werden konnten. Die Trainingsdaten wurden auch in anderer Reihenfolge präsentiert, was nichts an der durchschnittlichen Trainingsdauer änderte.

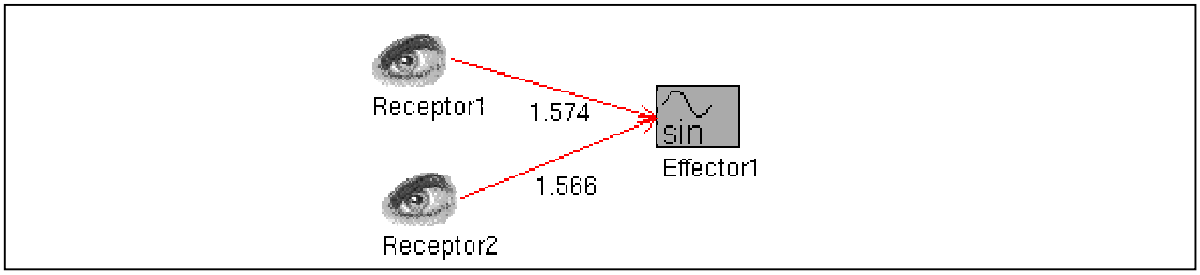


Abb. 6.6 Das hybride XOR Netz nach dem Training

In Abbildung 6.6 ist zu erkennen, daß sich die Gewichte jeweils auf $\frac{\pi}{2} \approx 1,57$ eingestellt haben. Dies ist nicht weiter verwunderlich, da $\sin\left(0\frac{\pi}{2} + 1\frac{\pi}{2}\right) = \sin\left(1\frac{\pi}{2} + 0\frac{\pi}{2}\right) = 1$ und $\sin\left(0\frac{\pi}{2} + 0\frac{\pi}{2}\right) = \sin\left(1\frac{\pi}{2} + 1\frac{\pi}{2}\right) = 0$. Man könnte auch sagen, daß das System numerisch die Lösung der Gleichungen

$$\sin(0x + 0y) = 0 \qquad \sin(0x + 1y) = 1$$

$$\sin(1x + 1y) = 0 \qquad \sin(1x + 0y) = 1$$

gefunden hat.

6.1.1.3 Ein weiterer hybrider Versuch mit XOR

Bei diesem Versuch wurde ein komplexeres Netz verwendet, um XOR zu trainieren.

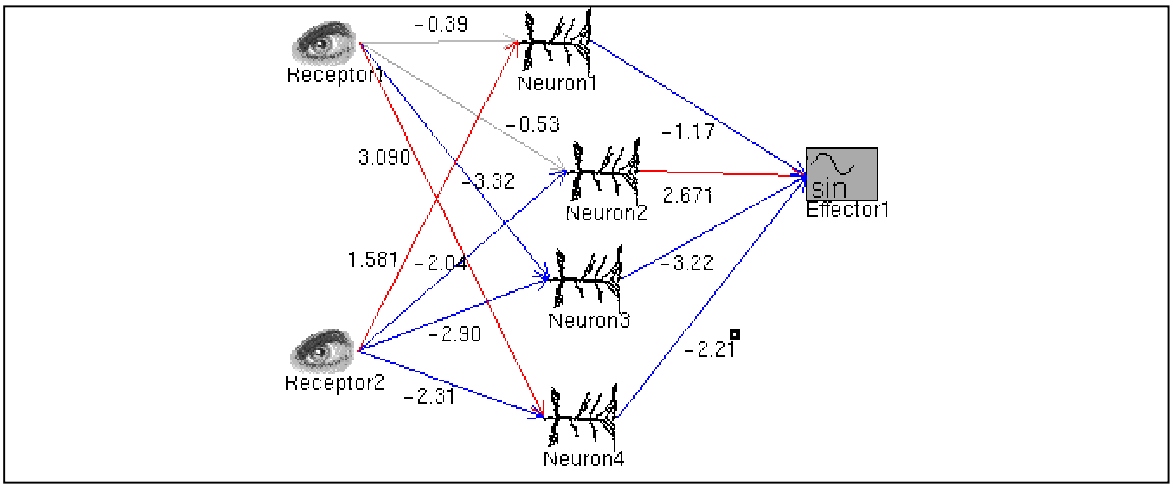


Abb. 6.7 Das Netz eines weiteren hybriden XOR-Versuchs

Die Trainingsparameter waren wie in Abbildung 6.2 und es wurde Batch-Update verwendet. Nach 350 Epochen war der Fehler unter 0,1 und es stellte sich folgendes rezeptive Feld ein.

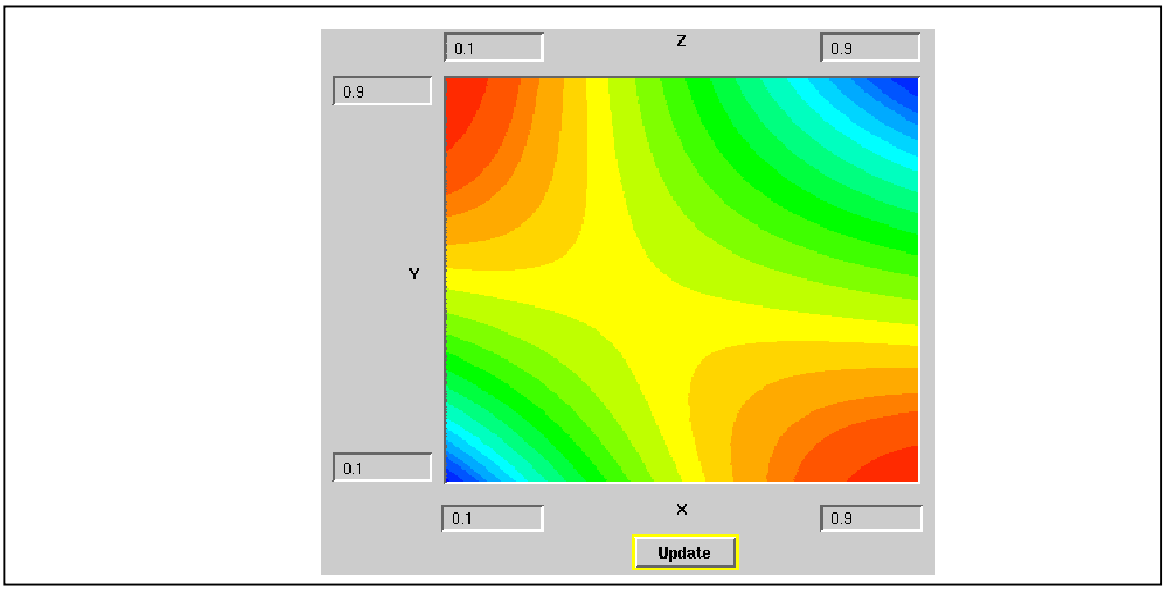


Abb. 6.8 Ergebnis eines weiteren XOR-Versuchs

Die Werte in den Ecken des rezeptiven Feldes stiegen oder fielen entsprechend monoton weiter, so daß hier von einer Extrapolation der Werte durch das Netz gesprochen werden kann. Extrapolation ist mit klassischen Sigmoiden kaum erreichbar, da einzelne Sigmoide bei einer Eingabe, die gegen Unendlich geht, selber gegen einen konstanten Wert gehen, so daß auch eine Linearkombination von Sigmoiden immer gegen einen konstanten Wert strebt.

6.1.2 Doppelspirale

Ein anderes klassisches Problem ist das Training einer Doppelspirale. Mit der Doppelspirale ge-

Lernrate	0.1
Auffrischungmethode	Batch
Fehler	<=0.1

Tab. 6.2 Parameter für ein einfache Doppelspiralproblem

lang es beim besten von 5 Experimenten 71 % der Trainingspunkte richtig zu klassifizieren. Es wurde dabei allerdings ein festgelegter Grenzwert benutzt, der genau in der „Mitte“ zwischen Rot und Blau lag.

Ein vor allem ästhetisch schönes Ergebnis DOLPHINS beim Training dieser Doppelspirale war das Ying und Yang-Zeichen, was sich im Anhang findet.

Eine einfachere Version dieser Spirale konnte allerdings erfolgreich trainiert werden.

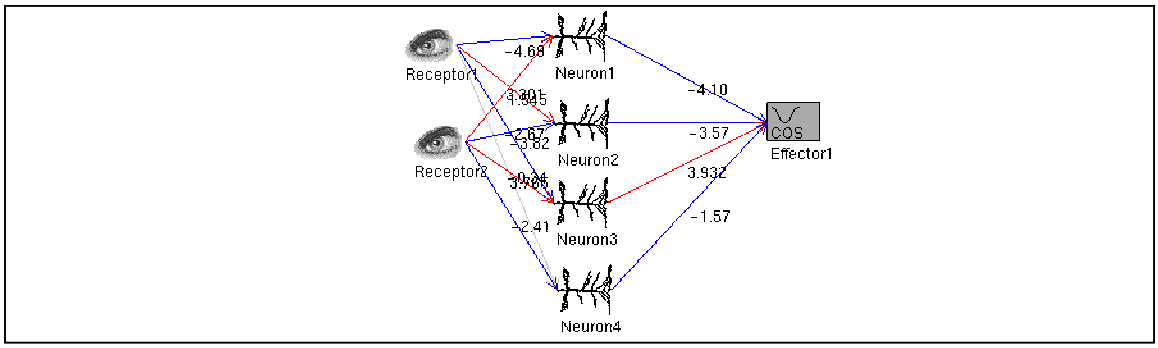


Abb. 6.9 Ein Netz zur Lösung eines einfachen Doppelspiralproblems

Als Aktivierungsfunktion des Effektors diente `SumUpCos`, da man hoffte, die bogenartigen Bewegungen der Spirale mit ihr gut nachahmen zu können.

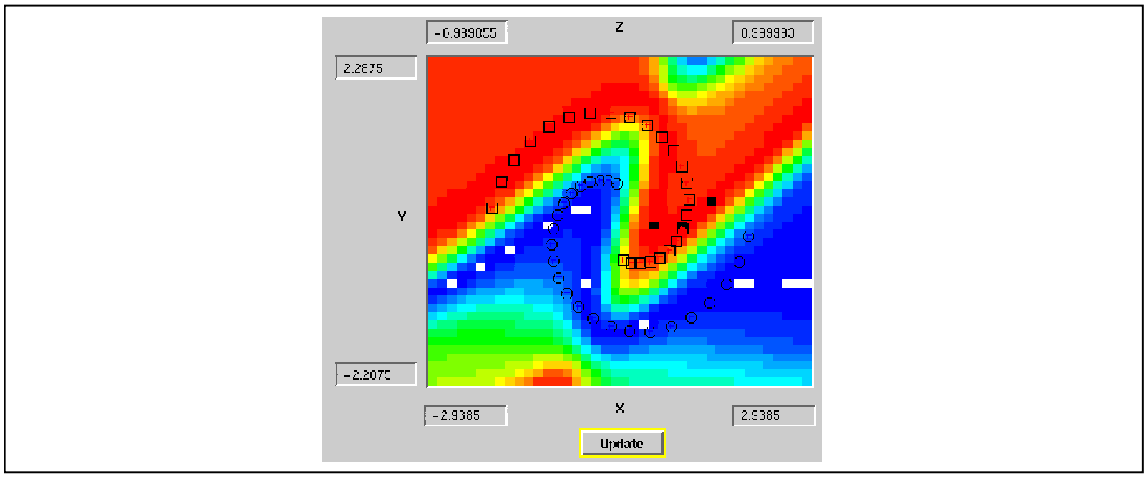


Abb. 6.10 Lösung eines einfachen Doppelspiralenproblems

Nach 300 Iterationen stellte sich die oben stehende Lösung ein.

6.1.3 Polynomiale Approximation

Als Beispielsfunktion diente $f(x):=4x-2x^2-x^3$.Gegeben waren 20 Trainingspunkte P(x,f(x)).

6.1.3.1 Approximation durch Kombination mehrerer Potenzfunktionen

Es sollte herausgefunden werden, ob ein Netz, bestehend aus einer gewichteten Summation der Faktoren 1,x, x^2, und x^3 in der Lage ist, die Koffizienten des Polynoms $4x-2x^2-x^3$ als Gewichte einzustellen.

Dazu wurde folgendes Versuchsnetz aufgebaut:

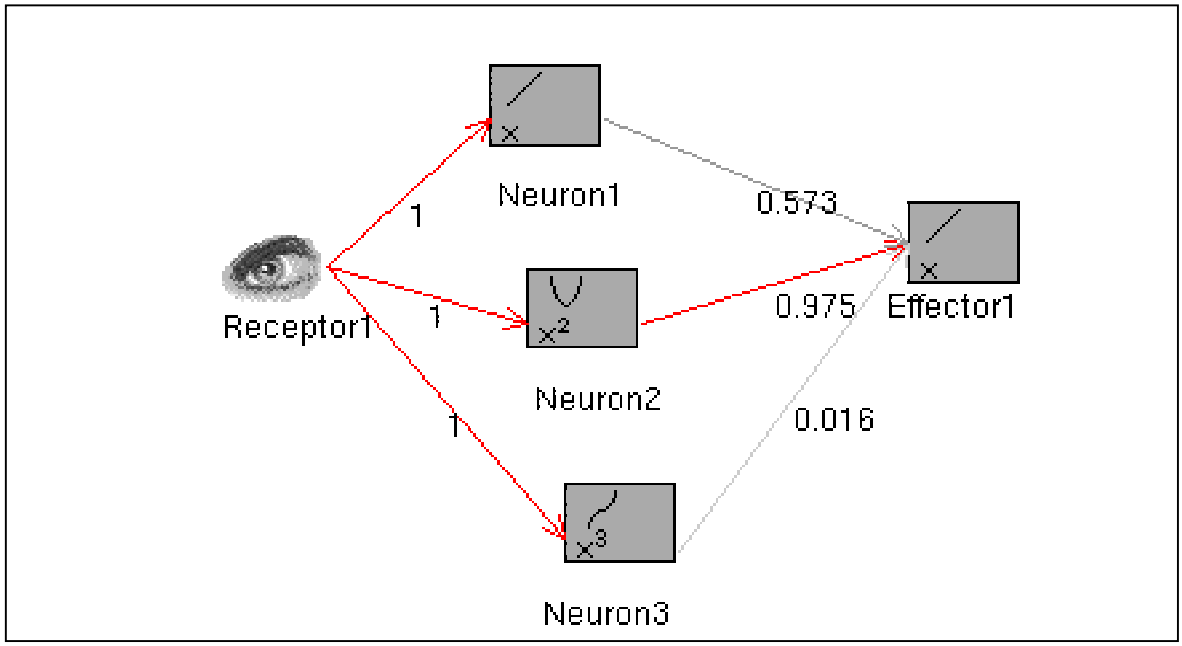


Abb. 6.11 Ein Netz zur Faktorisierung

Die Ausgangskanäle von Rezeptor1 waren alle auf 1 gesetzt und untrainierbar gemacht worden. Die Biaskanäle von Neuron1, Neuron2 und Neuron3 wurden auf 0 gesetzt und waren ebenfalls untrainierbar. Die Lernrate betrug 0,1, das Lernverfahren war Online-Update.

Als Ergebnis nach 35000 Trainingsdurchläufen stellte sich das Netz ein, wie in Abbildung 6.12 zu sehen ist. Der Biaskanal von Effektor 1 stand dabei auf 0.00616.

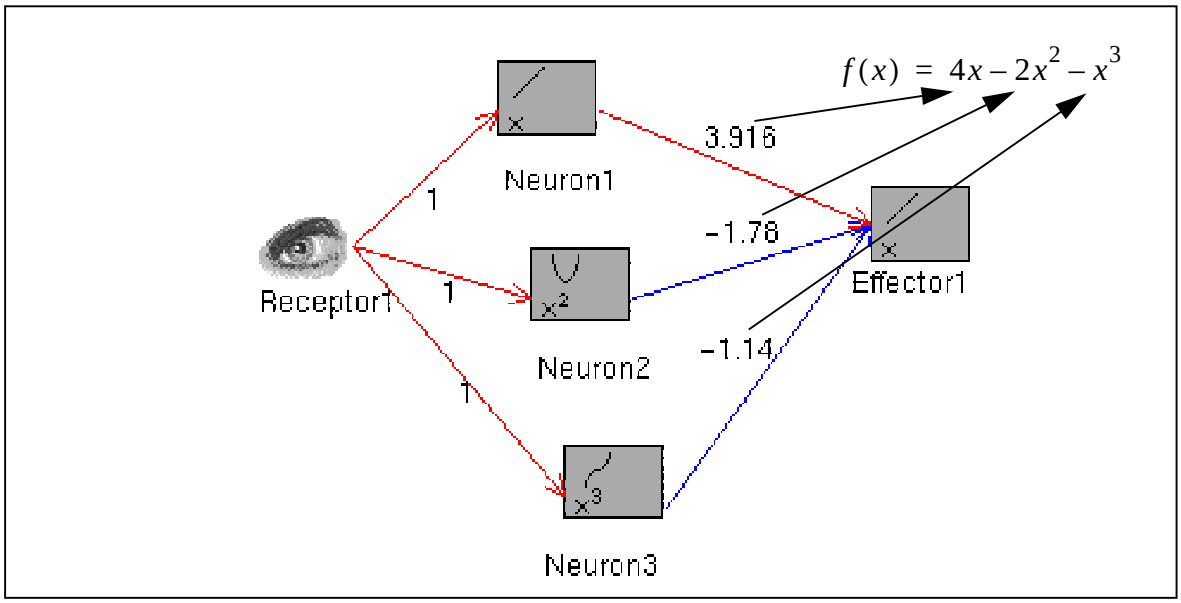


Abb. 6.12 Ein Netz aus Potenzfunktionen nach 35000 Trainingsdurchläufen

Die vorgegebene und die vom Netz erzeugte Funktion unterschieden sich nur noch kaum.

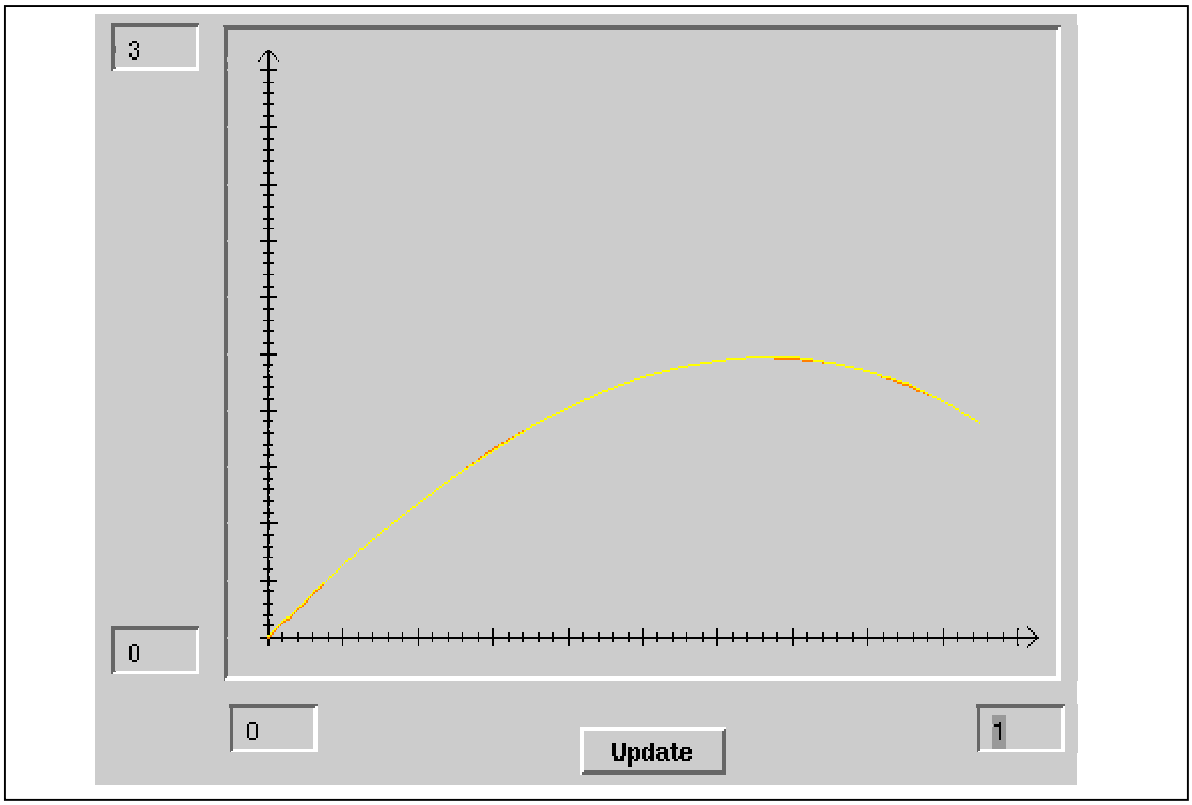


Abb. 6.13 Die vorgegene (rot) und die gelernte Kurve (gelb)

Das Netz ist in der Lage, mittels der Kombination von Potenzfunktionen als Aktivierungsfunktion die Koeffizienten eines Polynoms zu finden, welches vorgegebene Punkte annähern kann. Dabei können vom Benutzer beliebige Potenzfunktionen kombiniert werden. Für die Approximation eines Funktionsverlaufs durch ein Polynom muß somit kein anderes Werkzeug verwendet werden. Auf Grund von Zeitmangel konnten keine Vergleiche mit anderen numerischen Verfahren angestellt werden, die polynomiale Funktionsapproximation leisten, jedoch liegt bei dieser Methode der Vorteil in der Einbettbarkeit der Approximation in neuronale Netze.

6.1.3.2 Approximation durch eine generelle Polynomfunktion

Es wurde nach der Demonstration des Experimentes aus 6.1.3.1 gewünscht, eine generelle Po-

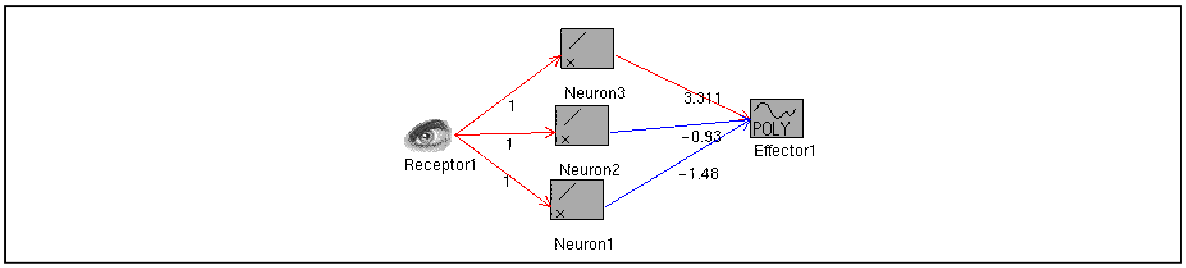


Abb. 6.14 Ein Netz mit der generellen Polynomfunktion nach 2200 Läufen

lynomfunktion zur Verfügung zu haben. Bei ihr steht der n. Eingang für x^n. Das Ergebnis aller Eingänge wird summiert. Es können beliebig viele Dendriten angehängt werden, einzelne Faktoren können ausgelassen werden, indem der entsprechende Kanal auf 0 und untrainierbar gestellt wird.

Hier offenbarte sich die Erweiterbarkeit von DOLPHIN, es brauchte ungefähr 5 Minuten, bis diese Funktionalität implementiert war.

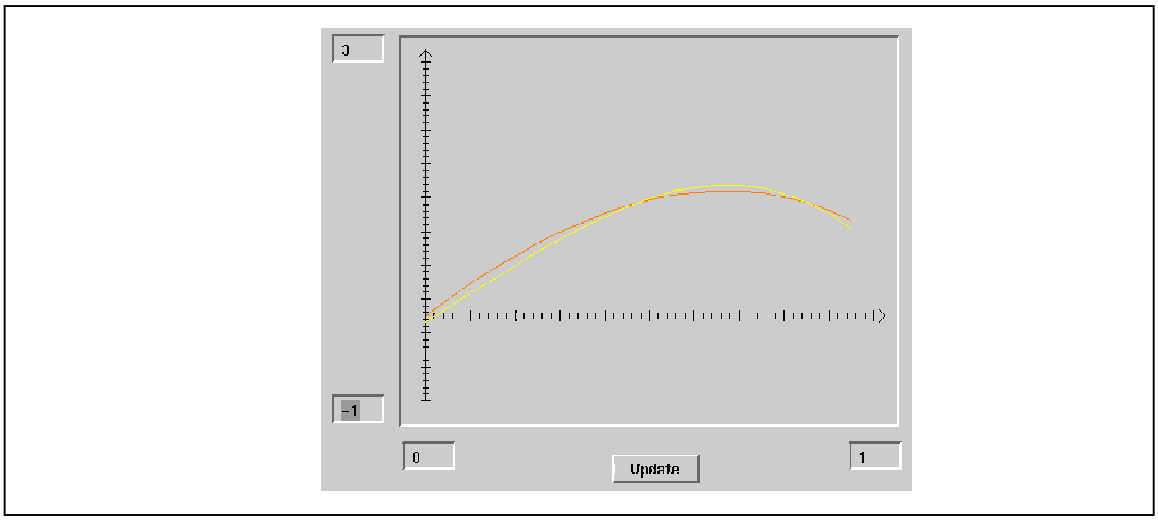


Abb. 6.15 Das Ergebnis der generellen Polynomfunktion

Das Experiment zeigt auch mit dieser Aktivierungsfunktion eine gute Annäherung an die Vorgabe, jedoch gelang mit ihm auch nach weiteren 20000 Präsentationen des Trainingsdaten die Einstellung der Koeffizienten nicht so gut wie in 6.1.3.1. Momentan ist die Visualisierung der Eingangsnummer der Dendriten noch nicht implementiert, so daß, nachdem die Verbindungen gezogen sind, nur mittels der Inspektion des entsprechenden Blattes festgestellt werden kann, welcher Eingang welchem Term zugeordnet ist.

6.2 Anwendung Walzstraße

Mit DOLPHIN konnte man bereits ein erstes Teilproblem zur Simulation der Ausgangsprofile einer Walzstraße lösen. Da der Industriepartner Siemens eine Anmeldung dieser Methode zum Patent in Erwägung zieht, darf dieses Ergebnis aus Patentschutzgründen an dieser Stelle noch nicht veröffentlicht werden.

6.3 Vergleich mit anderen NN-Simulatoren

Im Folgenden sollen kurz die Vorzüge von DOLPHIN bei Backpropagation gegenüber anderen Simulationswerkzeugen erläutert werden. Daß in keinem der besprochenen Simulatoren Shared Weights implementiert sind, soll gleich an dieser Stelle gesagt sein.

6.3.1 PDP++

PDP++ ist ein in C++ von McClelland und anderen geschriebenes, frei erhältliches Simulationswerkzeug für neuronale Netze, das wegen der Bekanntheit eines der Autoren [RuCl] und seiner Aktualität Beachtung verdient.

In der Beschreibung der Verbindungsklasse von PDP++ wird deutlich, daß das Modell der Netze schicht- und nicht einheitenorientiert ist.

„The Connection class contains the weights which represent the strengths of the relationship between units. Since there are typically many more connections than any other type of object in a simulation, these objects are treated specially to speed up processing speed and reduce their memory consumption.

The main way in which connections are different than other objects, like Units, for example, is that they are usually operated on in a group. Thus, the Con_Group class becomes very important to determining how connections behave, which is not the case with a Unit_Group, for example.

The Con_Group, and not each connection, contains a pointer to the ConSpec which governs the behavior of all of the connections in the group. Also, Con_Groups can contain algorithm-specific parameters, and in general algorithms define their own type of Con_Group. When connections are created by projections, a new Con_Group is created to hold all of the connections for each projection.

The basic Connection class (which is often abbreviated Con when new types are defined based on it, e.g. BpCon) has only the weight value, wt. ***Other algorithms will add other variables as needed.*** *(...)“* (http://www.cs.cmu.edu/Web/Groups/CNBC/PDP++/pdp-user_123.html)

Auch in PDP++ ist es damit machbar, von der durch `Con` abekürzten Klasse abzuleiten, um in den Unterklassen neue Instanzvariablen einzuführen.

„*(...) Similarly, in the error phase, first the derivative of the error* ***with respect to the activation (dEdA) of each unit*** *is computed based on current dEdNet values, and then the dEdNet values are updated based on the new dEdNet. (...)“* http://www.cs.cmu.edu/Web/Groups/CNBC/PDP++/pdp-user_188.html

Die Möglichkeit, die Aktivierungsfunktion nach jedem einzelnen Eingabewert des Neurons, also dem Ergebnis jedes Eingabekanals, abzuleiten, ist nach der Beschreibung des Backpropalgorithmus aber dennoch nicht gegeben. Es können dort nur skalare Transferfunktionen nach der durch Summation vorverarbeiteten Eingangssignale (=activation of each unit) abgeleitet werden.

6.3.2 ECANSE

ECANSE ist ein kommerzielles, ebenfalls in C++ geschriebene Simulationsumgebung für neuronale Netze. Auch hier ist es nur möglich, Ausgabefunktionen zu ändern, nicht aber ganze Aktivierungsfunktionen. Für ein Netz kann dabei nur eine einzige Ausgabefunktion eingestellt werden. [Eca, Seite 215]

6.3.3 FAST

FAST ist ein bei FORWISS in C geschriebener, auf Geschwindigkeit optimierter Werkzeugkasten zur Simulation neuronaler Netze. Auch hier sind nur auf der Summation als Integrationsfunktion beruhende Ausgabefunktionen einstellbar, die dann auch für alle Einheiten des Netzes gelten. [ArM, Seite 11]

6.3.4 SESAME

Einen wirklichen Konkurrenten auf hybridem Gebiet hat DOLPHIN mit dem objekt- und datenflußorientierten Simulator SESAME [Lin]. Dieser ist in der Lage, beliebige differenzierbare Aktivierungsfunktionen während des BP-Schrittes zu verwenden. Während der Planung und den Vorarbeiten zu DOLPHIN lag die Dissertation über SESAME noch nicht vor, so daß man sich entschied, zumindest einen Prototypen DOLPHINs auf Grund dieser Vorarbeiten fertigzustellen. „Shared Weights“ sind dort allerdings ebenfalls nicht implementiert.

7 Ausblick

Nachdem die am Anfang an das Programm gestellten Anforderungen implementiert worden sind, zeigte sich bei der Anwendung DOLPHIN‘s im Rahmen des AENEAS Projektes, daß es mit Hilfe der Shared Weights und den verallgemeinerten Neuronen tatsächlich möglich ist, Probleme bei der Simulation von Walzwerken besser zu lösen. Die detailierte Darstellung des erfolgreichen Ansatzes ist hier leider nicht möglich, da der Industriepartner Siemens eine Anmeldung dieser Methode zum Patent in Erwägung zieht und sie aus Patentschutzgründen vorher nicht veröffentlicht werden darf.

Dieser Erfolg weckte bei den Anwendern den Wunsch nach einer Erweiterung des Systems. Folgende Komponenten wären noch zu implementieren:

- Gleichzeitige Trainierbarkeit DOLPHIN‘s mit einem Trainingsset und Validation des Trainings durch einen Testset. Für beide sollte jeweils eine Fehlerkurve angezeigt werden.
- Flexiblere Handhabung des Weight Sharings, ganze Kanalschichten sollten mit anderen Kanalschichten geteilt werden können.
- Weitere Lernverfahren (Quickprop, Kohonennetze, Cascade Correlation).
- 2D-Plotter, der analog zum 3D-Farb-Plotter das Verhalten eines Neurons in Abhängigkeit von beliebigem Input beschreibt.
- 3D-Plotter, der das rezeptive Feld nicht farbig sondern dreidimensional darstellt.
- Implementierung des Verfahrens aus 3.5, so daß Netze gekapselt, als Modul gespeichert und in andere Netze eingehängt werden können.

Beim Entwurf der Netzarchitektur war der wichtigste Diskussionspunkt, ob nun zuerst das Modell des Netzes und dann eine erste Benutzeroberfläche zu implementieren ist, oder umgekehrt. Ich denke, daß ich gut daran tat, mir lange Gedanken über das Modell und dessen Implementierung zu machen, bevor ich mit der Oberflächenprogrammierung anfing. Jedoch hätte ich ohne Drängen nach ersten benutzbaren Ergebnissen wohl zu lange an einem Modell gefeilt, das dann möglicherweise doch ohne praktischen Wert gewesen wäre. Sehr gute Ideen habe ich, wenn auch erst sehr spät, aus [Lin] erhalten, der in seinem Ausblick zurecht fordert, datenflußorientierte Basisbausteine für Simulatoren neuronaler Netze zu vereinheitlichen. Über CORBA könnten dann sogar Netze oder Einheiten DOLPHINs mit SESAME kommunizieren, wichtig wäre dafür aber, daß auch BP-Netze miteinander komponierbar würden. Eine andere langfristige Perspektive könnte sein, mittels VisualWave DOLPHIN als Internet-Server zur Verfügung zu stellen, möglicherweise nur als Front-End und FAST als Back-End zur Berechnung zeitkritischer Probleme. Dazu müsste HotDraw nach VisualWave portiert werden, was sicher keine leichte aber dafür sehr lohnende Aufgabe wäre. Die Benutzer stellen das Netz zusammen, schikken die Trainingsdaten und erhalten nach kurzer Zeit ihre Ergebnisse. Wenn sie keine Kenntnisse über neuronale Netze besitzen, könnte ihnen eine Demonstration weiterhelfen, oder sie treten direkt mit einem Mitarbeiter in Kontakt. Derartige Interaktivität mit einem Kompetenzzentrum für neuronale Netze wäre sicher für viele private und geschäftliche Nutzer reizvoll.

7.1 Danksagung

Zusätzlich ließe sich so ein Werbeeffekt für die Gruppe Neuronale Netze und Fuzzy Logic vom FORWISS erzielen, bei deren Mitarbeitern Rainer Holve, Peter Protzel und Michael Tagscherer ich mich alphabetisch sehr herzlich für die sehr große Hilfsbereitschaft bei Diplomarbeit, Bewerbungsschreiben und Futtertransport vom Beck bedanken möchte. Danke auch an Herrn Prof. Görz, der bereit war, ein solches „konnektionistisches“ Thema zu betreuen. Nichtalphabetische und sehr große Danksagung geht an meinen Betreuer Lars Kindermann, der keiner Diskussion mit mir überdrüssig wurde.

8 Literatur

[AbSu] Abelson Harold, Gerald J. Sussman „Structure and Interpretation of Computer Programs“ Massachusetts MIT Press 1985

[ArMo] Arras Michael, Mohraz Karim „FAST v2.2“ Interner Bericht FORWISS 1996

[Boo] Booch, Grady „Object Oriented Design With Applications“ Redwood City 1991

[Bra] Brant John M. „HOTDRAW“ Thesis, Urbana-Champaign University of Illionois 1995

[Bur] Burbeck, Steve „Applications Programming in Smalltalk 80: How to use Model-View-Controller (MVC)“ http://uiuc.patterns.edu 1992

[Coo] ParcPlace/Digitalk „VisualWorks-Cookbook“ Sunnyvale/California 1995

[Eca] ECANSE „Object Reference Sheets“ Siemens Wien 1995

[Gal] Gallant,A White H, „There exist a neural network that does not make avoidable mistakes“ [IEEE 1988], Vol I, S. 657-664

[GHJV] Gamma Erich, Richard Helm, Ralph Johnson, John Vlissides „Design Patterns“ Massachusetts Addison Wesley 1995

[GoRo] Goldberg Adele, Robson David „Smalltalk-80 The Language“ Massachusetts Addison Wesley 1989

[GoRu] Goldberg Adele, Rubin Kenneth „Succeeding with Objects - Decision Frameworks for Project Management“ Massachusetts Addison Wesley 1995

[Joh] Johnson Ralph „Documenting Frameworks using Patterns“ OOPSALA 92

[Lew] Ted Lewis „Object-Oriented Application Frameworks“ Prentice Hall 1995

[Lin] Linden Alexander „SESAME - Ein objekt- und flußorientierter Simulator für Modelle der Neuroinformatik und angrenzender Gebiete“ St. Augustin 1995

[Lit] Littmann Enno „Strukturierung Neuronaler Netze zwischen Biologie und Anwendung“ St. Augustin 1995

[Mon] Montlick, Terry „Implementing mixins in Smalltalk“ The Smalltalk Report, New York, SIGS Publications Juli 1996

[MPGS] Martinetz T.,Protzel Peter, Gramckow O., Sörgel G., "Neural Network Control for Rolling Mills", In Proc. 2nd European Congress on Intelligent Techniques and Soft Computing - EUFIT-94, Aachen, 1994, pp. 147-152

[NoHi] Nowlan Steven, Hinton Geoffrey „Simplifying Neural Networks by Soft Weight-Sharing“ Neural Computation 4, Massachusetts Institute of Technology 1992

[RMS] Ritter Helge, Martinez Thomas, Schulten Klaus „Neuronale Netze“ Bonn 1990

[Roj] Rojas, Raul „Theorie der neuronalen Netze“ Berlin 1993

[RuCl] Rumelhart David, McClelland James „Parallel Distributed Processing“ Massachusetts MIT Press 1987

[SLS] F. Schulter, J. van Laak, L. Schmitz: SIC'95 Report and User Manual, Techn. Report 9501, Dept. of Comp. Science, UniBw München 1995

[Wil] Williams, Derek „Taking out the Garbage“ The Smalltalk Report , New York, SIGS Publications, März 1996

Anhang

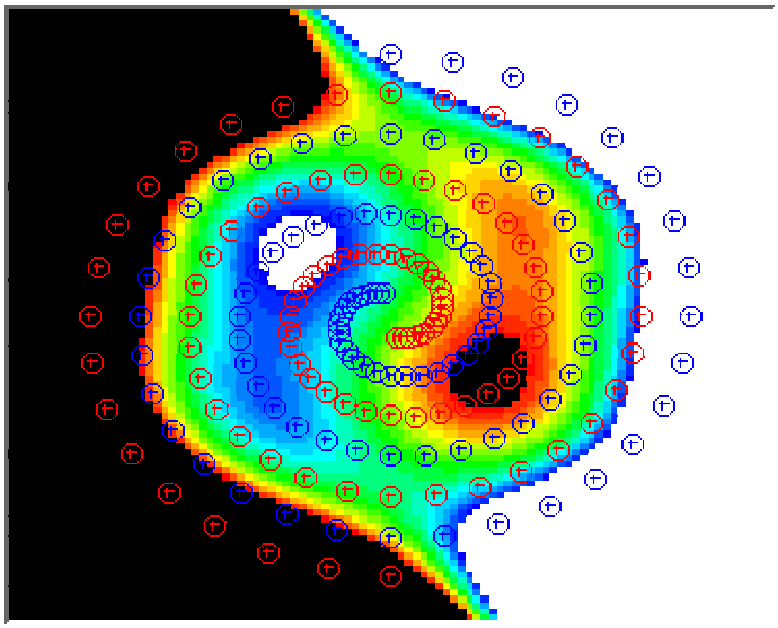


Abb. 8.1 Das Yin-Yang Symbol als ästhetisch bestes Ergebnis DOLPHINs

Index