Titel der Diplomarbeit:

Anwendung künstlicher Neuronaler Netzwerke als datengetriebenes Analysewerkzeug

Kennzahl **J151**
Matrikel–Nr.: **8651021**

Diplomarbeit

Eingereicht von

Ing. Gottfried Rudorfer

am Institut für

Informationsverarbeitung und Informationswirtschaft, Abteilung für Angewandte Informatik insbesondere Betriebsinformatik

an der Wirtschaftsuniversität Wien

Studienrichtung: **Betriebswirtschaft**

Begutachter: **o.Univ.-Prof. Dr. W. H. Janko**

Wien, am 5. September 1993

Ich versichere:

1. daß ich die Diplomarbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe,

2. daß ich diese Diplomarbeit bisher weder im In- oder Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Wien, am 5. September 1993 Ing. Gottfried Rudorfer

Inhaltsverzeichnis

Kapitel 1

Zusammenfassung

In dieser Arbeit wird das künstliche Neuronale Netzwerk als eine datengestützte Analysemethode vorgestellt.

Detailliert wird auf die Funktionsweise dieser Netzwerke ausgehend von den biologischen(natürlichen) Neuronalen Netzwerken, der Ableitung eines künstlichen Modells, bis hin zur Implementierung von Prototypen und deren Anwendung unter anderem zur Prognose von Zeitreihen eingegangen.

Anhand eines oft in der Literatur erwähnten Beispiels werden die Eigenschaften des künstlichen Netzwerks durch Veränderung seines Lernverhaltens mittels einer rechenaufwendigen Simulation gezeigt.

Resümierend läßt sich feststellen, daß sich dieser Ansatz ausgezeichnet in den durchgeführten Simulationen bewährt hat.

Kapitel 2

Einführung

Bei unserem Bestreben intelligente Maschinen zu entwickeln haben wir ein natürliches Vorbild – das menschliche Gehirn. Folglich ist die Idee, die Funktionsweise des Gehirns auf Computern zu simulieren, naheliegend.

Als Intelligent wird dabei jenes Verhalten bezeichnet, welches menschliches Handeln ermöglicht. Die Forschungsdisziplin *künstliche Intelligenz (KI)* (engl: *Artificial Intelligence (AI)*) ist Mitte der 50er Jahre in den USA entstanden und versucht Probleme, die Intelligenzleistungen voraussetzen, mit Computern zu lösen. Eine wesentliche Annahme der klassischen KI ist jene, daß intelligentes Handeln auf umfangreichem Wissen basiert. Derartige wissensbasierte Systeme bestehen aus Symbolen und Symbolstrukturen und aus ein im System gespeicherten Wissen in Form von meist bewußt formulierten ('wenn-dann') Regeln. Schlußfolgerungen, die nicht auf explizit ausformulierbaren Regeln bestehen, können durch diese Modelle nicht berücksichtigt werden. Kognitionsforscher stellten aber ein nicht ausschließlich durch Stimulus-Response ('wenn-dann' Beziehungen) induziertes Verhalten fest. Vielmehr gibt es Prozesse die unbewußt ablaufen. Viele Vorgänge des Handelns zählen zum unbewußten und assoziativen Bereich. Dorffner [Dorffner, 1991, S 10ff] spricht in diesem Zusammenhang von einer subsymbolischen AI, in der symbolische Schlußfolgerungen in einem assoziativen Grundmechanismus eingebettet sind (siehe Abb. 2.1). AI unter diesem Gesichtspunkt wird als sub-symbolische AI (SSAI) bezeichnet.

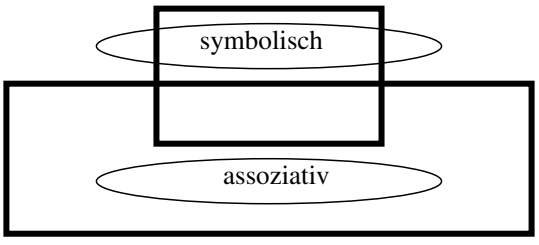

Abbildung 2.1: sub-symbolische AI

Ausgangspunkt dieses Ansatzes sind Aufgaben wie Erkennen, intuitiv Schließen und Assoziieren.

Radermacher beschäftigt sich in seinem Artikel Radermacher [1992] unter anderem mit der Einordnung Neuronaler Netze in die verschiedenen Ebenen der Erkenntnis. Eine Möglichkeit der Einsicht in die Welt ist die Symbolverarbeitung, die entweder auf einer Regelebene (Relationen, Taxonometrien, logisches Schließen) oder auf einer mathematischen Modellierung (Optimierung, Entscheidungstheorie, Differentialgleichungen) aufsetzt. Schließlich gibt es neuronal kodierte Einsichten im Sinne eines gewünschten Verhalten zwischen Sensorinformationen und bestimmten Aktionen. Neuronale Netzwerke werden unter anderem angewendet, wenn man keine tieferen Einsichten besitzt und sind als hardwarenaher Mechanismus anzusehen. Als Beispiel sei die Aufmerksamkeitssteuerung des Menschen zu nennen, die mit der Verarbeitung breiter Sensorströme verbunden ist (Bildwahrnehmung). Mithilfe eines Regelwissens findet in der mittleren Ebene der Einsicht eine Abstraktion dieser Sensorströme statt. Eine mathematische Kodierung stellt schließlich die höchste und präziseste Form der Einsicht dar, sofern die Daten vorhanden sind. Dabei muß man aber bemerken, daß man mit einem Neuronalen Netz eine Turing-Maschine bauen kann und man umgekehrt auch mit einem Computer ein Neuronales Netz simulieren kann. Es geht vielmehr um eine adäquate Beschreibung eines Problems.

Hecht-Nielsen [Hecht-Nielsen, 1990, S 1 ff] bezeichnet die technologische Disziplin 'Neurocomputing' als einen fundamentalen neuen und anderen Zugang zur Informationsverarbeitung. Es ist die erste Alternative zu dem seit 45 Jahren dominierenden klassischen *programmierten Rechnen*. Beim klassischen Zugang wird nach einem Algorithmus und/oder einer Menge von Regeln gesucht, um ein Problem zu lösen. Diese werden dann in einer passenden Programmiersprache kodiert und eventuell weiter verfeinert. Es können nur dann Programme entwickelt werden, wenn das gewünschte Verhalten durch eine Prozedur oder durch Regeln beschrieben werden kann. Das Finden von Algorithmen für neue Problemstellungen stellt sich als ein teurer und zeitaufwendiger Prozeß dar. Ist ein derartiger Algorithmus oder eine entsprechende Regelmenge bekannt, stellt sich das Problem der Implementation, das durch umfassendes Design, Testen und iterativer Verbesserung der Software versucht wird zu lösen. Die neue Disziplin 'Neurocomputing' ermöglicht die Entwicklung von Lösungen für Probleme, bei denen kein Algorithmus oder keine Regeln bekannt sind.

'Neurocomputing' ist eine technologische Disziplin, die sich mit parallelen, verteilten und adaptiven Informationsverarbeitungssystemen beschäftigt. Es werden informationsverarbeitende Fähigkeiten durch Präsentation von Informationen aus der Umgebung des Informationsverarbeitungssystems entwickelt.

Die primär informationsverarbeitende Struktur, die hier behandelt wird, sind *Neuronale Netzwerke*. Andere Klassen von adaptiven informationsverarbeitenden Strukturen sind lernende Automaten, genetische Lernsysteme, "Simulated

Annealing"-Systeme, assoziative Speicher und 'Fuzzy'-Lernsysteme, auf die in dieser Arbeit nicht eingegangen wird.

Ein Ansatz zur Modellierung von Ideen der subsymbolischen AI sind die künstlichen Neuronalen Netzwerke. Die folgenden Kapitel werden sich eingehend mit den in der Literatur behandelten Netzwerktypen und ihrem Einsatz auseinandersetzen.

Kapitel 3

Neuronale Netzwerke

In der Literatur werden Neuronale Netzwerke auch als Neurale Netzwerke (vgl. Köhle [1990]) bzw. im Englischen als "Neural Networks" bezeichnet. Im folgenden soll der Begriff Neuronale Netzwerke (NN) verwendet werden.

Ein NN besteht aus vielen Neuronen, die miteinander verbunden sind. Ein Neuron ist ein Prozessor, welcher relativ unabhängig von anderen Neuronen arbeitet. Es verwendet nur lokale Informationen von anderen Neuronen mit denen es verbunden ist. Das Neuron erhält Informationen von Neuronen und gibt Information an andere Neuronen weiter.

3.1 Begriffe aus der Biologie

Modellierungsideen für Neuronale Netzwerke werden oft aus der Biologie bzw. aus der Neurophysiologie gewonnen. In diesem Forschungsbereich sind viele Funktionen des Gehirns noch nicht erforscht. Man weiß vielmehr kleine Teile aus einem komplexen Gebilde.

3.1.1 Aufbau und Funktionsweise von Neuronen

Die Abbildung 3.1 zeigt in einer schematischen Darstellung den Aufbau eines Neurons, welches mit einem anderen Neuron verbunden ist. Als fundamentaler Grundbaustein des Nervensystems ist das Neuron eine Zelle spezieller Bauart. Dieses besteht aus den Dentriten, dem Zellkern, der von einer Membran umgeben ist und einem Ausläufer - dem sog. Axon.

Die weit verzweigten Dentriten sind die Inputs, das Soma ist die Körperzelle und das Axon ist der Outputkanal des Neurons. Jedes Neuron kann viele Dentriten und das Axon kann mit mehreren Neuronen verbunden sein.

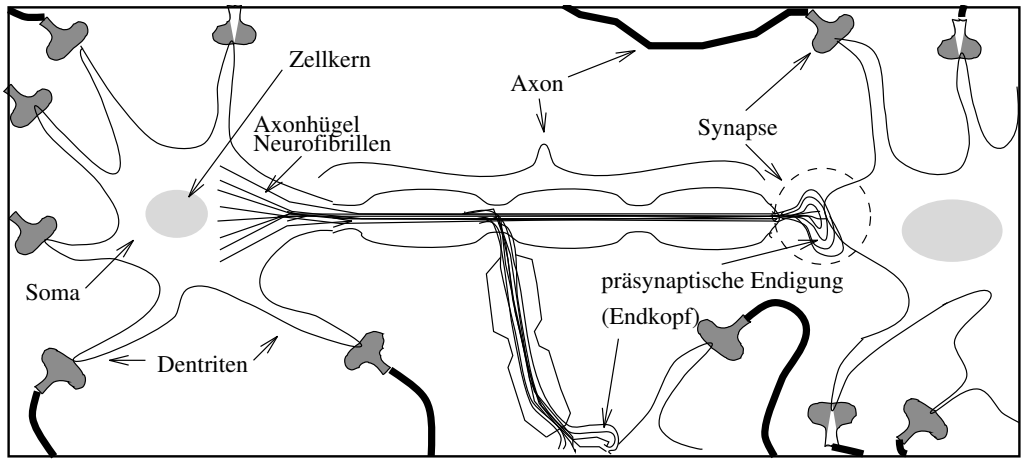

Abbildung 3.1: Schematische Darstellung eines Neurons

Erhaltungsfunktionen einer Zelle

Die Zelle ist nicht völlig von der Umgebung isoliert, sondern ist von einer Membran umgeben, die spezielle Aufgaben erfüllt. Wichtige Funktionen sind der Stoffaustausch und die Ernährung der Zelle. Das Aktivierungsniveau der Zelle wird durch eine 'Ionenpumpe' erreicht. Diese hält das Ruhepotential zwischen Zellinnerem und Zelläußerem während der Ruhephase aufrecht.

Die Zellmembran ist teilweise für Ionen (positiv oder negativ elektrisch geladene Teilchen) durchlässig. Dabei besteht eine bestimmte Konzentration von Ionen innerhalb und außerhalb der Zelle. Dadurch entsteht ein elektrisches Potential, das Ruhepotential bezeichnet wird.

Informationsverarbeitung des Zellkerns

Treffen von einem anderen Neuron Informationen ein, ändert die Zelle ihr Potential. Die Zelle antwortet mit einer elektrischen Potentialänderung – sie 'feuert'. Die Informationsübertragung erfolgt jedoch nicht elektrisch, sondern chemisch durch Ionenaustausch. Aufgrund der schlechten Leitfähigkeit der Verbindungselemente zwischen den Neuronen wäre dies gar nicht möglich. Elektrische Größen entstehen als Nebenprodukt der chemischen Prozesse. Der Verlauf des Aktionspotentials ist in Abbildung 3.3 dargestellt. Das Ruhepotential beträgt ca. -90 mV (-0.09 Volt), welches nach einer Erregung sprunghaft ansteigt und dann wieder langsam auf das Ruhepotential zurückgeht. Durch das Eintreffen eines Reizes wird dieser auf das nachfolgende Neuron über die Synapse übertragen.

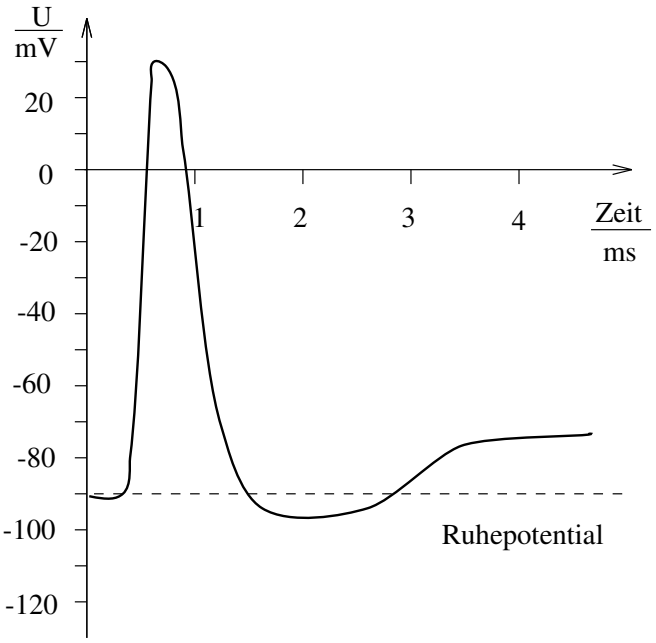

Abbildung 3.2: Phasen des Aktionspotentials

Das Verbindungselement – Die Synapse

Synapsen sind Verbindungsknoten, die schematisch in Abbildung 3.3 dargestellt sind. Dieser Baustein haftet an anderen Neuronen oder an einer Muskelzelle. Synapsen dienen der Nachrichtenübermittlung zwischen Zellen. Ein wesentliches Merkmal Neuronaler Netzwerke ist die starke Verflechtung der Neuronen untereinander. Ein Neuron ist nicht wie in 3.1 mit einem anderen Neuron verbunden, sondern mit tausenden. Die Zellen sind dabei nicht direkt miteinander verbunden, sondern der Informationsaustausch erfolgt durch chemische Substanzen. Es gibt einen kleinen Spalt von ca. 20 nm zwischen den Zellen. Die präsynaptische Zelle sendet Substanzen aus die für die postsynaptische Zelle geeignet sind. Diese öffnet nur dann die Membran für den Ionenaustausch, wenn dieser Stoff identifiziert wurde. Durch diesen Aufbau wird eine Ventilfunktion der Synapse erreicht. Eine Erregung der Ursprungszelle wird vermieden.

Wie werden viele Einzelinformationen ausgewertet?

Der Einfluß eines Input-Neurons auf das Aktivierungspotential eines Neurons ist meist gering, da viele Neuronen als Inputs für ein Neuron dienen. Erst eine entsprechende Summe von eintreffenden Informationen überschreitet den Schwellwert am postsynaptischen Neuron, welches dann 'feuert'. Es entsteht ein sog. "Exzitatorisches (erregendes) Postsynaptisches Potential" (EPSP). Die-

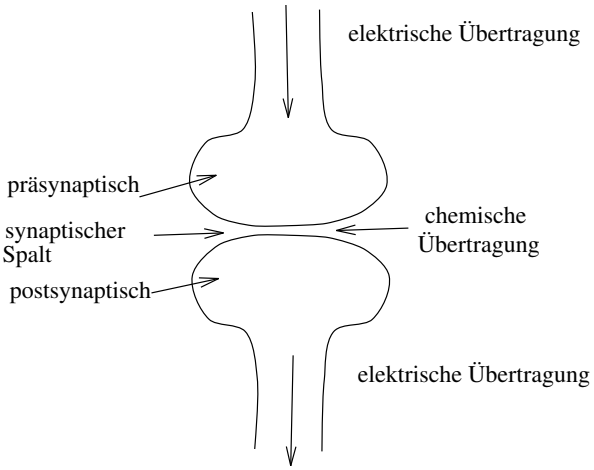

Abbildung 3.3: Informationsübertragung durch eine Synapse

ses kann nicht nur vergrößert werden, sondern negative Beiträge durch hemmende Synapsen können das Aktivierungspotential wieder verringern. Dieser hemmende Einfluß wird "Inhibitorisches (hemmendes) Postsynaptisches Potential" (IPSP) bezeichnet. Die Summe dieser Potentiale (EPSP und IPSP) entscheidet, ob ein Aktionspotential ausgelöst wird.

Das Aussehen des Neurons hat Einfluß auf seine Eingenschaften

Die Platzierung von Synapsen an einem Neuron hat Einfluß auf die Erregbarkeit des Neurons. In der nähe des Axonhügels plazierte Synapsen führen zu einer hohen, und Synapsen an den Dentriten führen zu einer geringen Beeinflussung des Aktivierungspotentials.

Weiters wurde beobachtet, daß die Zellgröße ebenfalls einen Einfluß auf die Erregbarkeit des Neurons hat. Kleine Neuronen sind leichter erregbar bzw. hemmbar als große.

Erkenntnisse aus der Neurophysiologie

Die Erkenntnisse aus der Neurophysiologie sind noch sehr lückenhaft, um die Funktion des Gehirns vollständig erklären zu können. Vielmehr sind kleine Teile aus dem komplexen Gebilde erforscht.

Dieses Kapitel wollte nicht einen exakten neurophysiologischen Abriß machen, sondern jene Konzepte näherbringen, um die Gemeinsamkeiten und Unterschiede zwischen natürlichen und künstlichen Neuronalen Netzwerken zu erkennen.

Die Modellansätze zur Modellierung künstlicher Neuronaler Netzwerke verwenden zwar die Grunderkenntnisse der Neurophysiologie, beruhen aber nicht auf tiefgreifenden Erkenntnissen aus diesem Bereich.

Rumelhart [Rumelhart et al., 1986b, S 130 ff] geht noch auf jene Eigenschaften ein, die in der Formulierung eines Modells enthalten sein soll:

1. Es gibt viele, relativ einfache Prozessoren, die gemeinsam parallel arbeiten. Neuronen arbeiten langsam (im Millisekundenbereich) im Vergleich zu konventionellen Prozessoren in Computern (im Nanosekundenbereich).

2. Ein natürliches Neuronales Netz besteht aus großen Anzahl von Neuronen. Es gibt Schätzungen über 10^{10} bis 10^{11} Neuronen im Gehirn.

3. Neuronen empfangen Signale von vielen anderen Neuronen und senden Signale an viele Neuronen. Nach Schätzungen haben Neuronen $1,000$ bis $100,000$ Synapsen an deren Dentriten und weitere $1,000$ bis $100,000$ Synapsen sind mit den Dentriten anderer Neuronen verbunden.

4. Wissen befindet sich nicht in den Neuronen, sondern in deren Verbindungen. Lernen bedeutet daher, die Stärke der Verbindungen zu verändern.

5. Neuronen kommunizieren über aktivierende oder hemmende Verbindungen. Dabei werden keine Symbole übertragen, sondern Zahlen mit geringer Genauigkeit. Es geht also um Aktivierung und Hemmung und nicht um Symbole.

6. Das Netzwerk ist in einem kontinuierlichen Informationsfluß eingebunden. Es gibt keinen Zeitpunkt, in dem das Netzwerk keinen Ausgabewert liefert. Zu einem Eingabewert gibt es auch einen Ausgabewert.

7. Jedes Neuron arbeitet lokal, d.h. es erhält Informationen über seine Dentriten und sendet seine Informationen über sein Axon an andere Neuronen weiter. Es herrscht verteilte Kontrolle und nicht globale vor. Es gibt keine zentrale Stelle, die den Informationsfluß überwacht. Es gibt keinen Teil im Gehirn, von dem alle anderen Teile des Gehirns abhängig sind.

8. Die grundlegende Verarbeitungsfunktion des Gehirns ist der Versuch, eine große Anzahl von schwachen Restriktionen durch einen iterativen Suchprozeß zu erfüllen.

3.2 Definitionen

In diesem Abschnitt werden die wichtigsten Begriffe geklärt, die in der Arbeit verwendet werden.

3.2.1 Definition: Neuronales Netzwerk

Ein Neuronales Netzwerk ist eine parallele, verteilte Informationsverarbeitungs-Struktur, welches aus Verarbeitungseinheiten (die lokalen Informationsspeicher besitzen und lokale Informationen verarbeiten) besteht, die mit unidirektionalen Signalkanälen (Verbindungen) verbunden sind. Jede Verarbeitungseinheit hat einen einzigen Ausgang, der sich in viele Verbindungen teilt. Jede Verbindung führt dasselbe Signal, welches das Ausgangssignal der Verarbeitungseinheit ist. Die Informationsverarbeitung in jeder Verarbeitungseinheit kann beliebig definiert sein, mit der Restriktion, daß sie völlig lokal sein muß. Die Informationsverarbeitung darf nur von den Eingangssignalen, die durch die hinführenden Signalkanäle an der Verarbeitungseinheit eintreffen und den gespeicherten Werten im lokalen Speicher der Verarbeitungseinheit erfolgen (vgl. [Hecht-Nielsen, 1990, S 2]).

Ein Neuronales Netzwerk ist ein paralleles, verteiltes Informationsverarbeitungssystem in Form eines gerichteten Graphen (vgl. [Hecht-Nielsen, 1990, S 22]):

1. Die Knoten des Graphen werden Verarbeitungseinheiten genannt.
2. Die Kanten des Graphen werden Verbindungen genannt. Jede Verbindung arbeitet als eine unverzögerte Signalverbindung in eine Richtung (unidirektional).
3. Jede Verarbeitungseinheit kann eine beliebige Anzahl von Eingänge haben.
4. Jede Verarbeitungseinheit kann eine beliebige Anzahl von Verbindungen zu anderen Verarbeitungseinheiten haben. Die Signale in den Ausgangsverbindungen müssen aber gleich sein.
5. Die Verarbeitungseinheiten können einen beliebig großen lokalen Speicher haben.
6. Jede Verarbeitungseinheit hat eine Transferfunktion, die lokalen Speicher verwenden/verändern kann, um das Ausgangssignal der Verarbeitungseinheit zu erzeugen.
7. Die Eingangssignale des Neuronalen Netzwerkes kommen von Verbindungen außerhalb des Netzwerkes. Die Ausgangssignale verlassen das Netzwerk und gehen in die Umgebung des Netzwerks.

3.2.2 Definition: gerichteter Graph oder Digraph

Ein Graph besteht aus einer Menge von Punkten ('Knoten') und einer Menge von Linien ('Kanten'), die diese Punkte verbinden (vgl. [Neumann, 1975, S 21 ff]). Mathematisch heißt $G = (V, E)$ ein gerichteter Graph wenn gilt:

1. V ist eine endliche nichtleere Menge, die Menge der Knoten;
2. E ist eine geordnete Menge von ein- oder zweielementigen Teilmengen von V. Ein Paar $\{u, v\} \in E$ heißt Kante, u und v sind dann adjazent. Geordnet heißt, daß E eine Teilmenge von $V \times V$ ist. Die gerichtete Kante von u nach v notiert man durch (u, v).

3.2.3 Definition: Neural Computing

"Neural computing is the study of networks of adaptable nodes which, through a process of learning from task examples, store experimental knowledge and make it available for use." [Aleksander and Morton, 1990, S 1]

3.2.4 Definition: Punkt im n dimensionalen Raum

Ein Punkt $\mathbf{X}$ in einem n dimensionalen Raum ist durch eine geordnete Menge von n Werten oder Koordinaten $(x_1, x_2, ..., x_n)$ bestimmt (vgl. [Rao, 1990, S 96]).

3.2.5 Definition: Hyperebene

In einem n dimensionalen Raum wird die Menge der Punkte, deren Koordinaten die lineare Gleichung

$$a_1 x_1 + ... + a_n x_n = \mathbf{a^T X} = b \tag{3.1}$$

erfüllen, Hyperebene genannt.

Die Hyperebene H ist daher durch

$$H(\mathbf{a}, b) = \{\mathbf{X} \mid \mathbf{a^T X} = b\} \tag{3.2}$$

gegeben.

Eine Hyperebene hat $(n-1)$ Dimensionen in einem n dimensionalen Raum. Die Menge der Punkte, deren Koordinaten eine lineare Ungleichung wie $a_1 x_1 + ... + a_n x_n \leq b$ erfüllen werden geschlossener Halbraum genannt. Geschlossen deswegen, da die Gleichheit in der obigen Ungleichung enthalten ist.

Eine Hyperebene unterteilt E^n in zwei geschlossene Halbräume

$$H^+ = \{\mathbf{X} \mid \mathbf{a^T X} \geq b\} \tag{3.3}$$

und

$$H^- = \{\mathbf{X} \mid \mathbf{a^T X} \leq b\} \tag{3.4}$$

(vgl. [Rao, 1990, S 96]).

3.2.6 Definition: Percepton

Sei $\Phi = \{\varphi_1, \varphi_2, ..., \varphi_n\}$ eine Familie von Prädikaten. Wir nennen Ψ linear in Bezug auf Φ, wenn es eine Zahl Θ und eine Menge von Zahlen $\{\alpha_{\varphi_1}, \alpha_{\varphi_2}, ..., \alpha_{\varphi_n}\}$ gibt, sodaß $\Psi(X) = 1$ wenn und nur wenn $\alpha_{\varphi_1}\varphi_1(X) + \alpha_{\varphi_2}\varphi_2(X) + ... + \alpha_{\varphi_n}\varphi_n(X) > \Theta$. Kompakter geschrieben: $\Psi(X) = 1$ wenn und nur wenn $\sum_{\varphi \epsilon \Phi} \alpha_\varphi \varphi(X) > \Theta$.

Die Zahl Θ wird als Schwellwert, die Zahlen $\alpha_{\varphi_1}, ..., \alpha_{\varphi_n}$ werden als Koeffizienten oder Gewichte bezeichnet.

Ein Percepton ist eine Verarbeitungseinheit, die alle Prädikate berechnen kann, die linear in einer gegebenen Menge von partiellen Prädikaten Φ sind. D.h. es ist eine Menge Φ gegeben, die Gewichte α_φ und der Schwellwert Θ können aber frei gewählt werden.

3.2.7 Definition: Multilayer Perceptons

Ein Multilayer-Perceptons Netzwerk besteht aus einer Menge von Simple Perceptons, die in Schichten angeordnet werden. Es gibt eine Eingabeschicht mit d Perceptons. Dann folgt eine, zwei oder mehrere[1] Schichten von Perceptons und eine Ausgabeschicht von e Perceptons. Jede Ausgabe eines Perceptons ist mit der Eingabe jedes einzelnen Perceptons in der nächsten Schicht verbunden. Jeder Verbindung der $j-$ten Verarbeitungseinheit in der Schicht $l-1$ zur $i-$ten Verarbeitungseinheit in der Schicht l wird ein synaptisches Gewicht $w_{ij}^{(l)}$ zugewiesen. Alle Neuronen sind lineare Schwellwerteinheiten, außer die Neuronen der Eingabeschicht. D.h. der Ausgabewert $x_i^{(l)}$ der $i-$ten Verarbeitungseinheit in der $l-$ten Schicht wird mit der Formel $x_i^{(l)} = g\left(\sum_j w_{ij}^{(l)} x_j^{(l-1)} - \Theta_i^{(l)}\right)$

[1] Die l Schichten zwischen Eingabe- und Ausgabeschicht werden auch als verborgene Schichten oder "hidden layers" bezeichnet. Man spricht dann von einem Netzwerk mit l verborgenen Schichten.

berechnet, wobei $x_i^{(l)}$ als Ausgabewert des $i-$ten Neurons in der $l-$ten Schicht bezeichnet wird, $\Theta_i^{(l)}$ ist der Schwellwert des $i-$ten Neurons in der $l-$ten Schicht und $\Theta(v) = +1$ für $v \geq 0$ und -1 für $v < 0$. Mit jeder Verarbeitungseinheit kann man eine Hyperebene verbunden sehen, die durch $z_j : \sum_j w_{ij} z_j = \Theta_i$ definiert ist und erzeugt den Ausgabewert von $+1$ wenn der Eingabevektor $\mathbf{y}$ an der positiven und -1 wenn $\mathbf{y}$ an der anderen Seite der Hyperebene ist (vgl. [Baum, 1988, S 194]).

3.2.8 Definition: Sigmoide Funktion

Eine Funktion $F : R \to [\xi_-, \xi_+]$ wird als sigmoide Funktion bezeichnet, wenn $\xi_+ = \lim_{\xi \to \infty} G(\xi)$ und $\xi_- = \lim_{\xi \to -\infty} G(\xi)$ und $\xi_+ \neq \xi_-$ und diese Funktion differenzierbar und deren Ableitung $G'(\xi) \neq 0$ ist.

3.2.9 Definition: Lernen in einem Neuronalen Netzwerk

Gegeben ist eine Menge von Eingabe- /Ausgabekonstellationen, die dem Netzwerk präsentiert werden. Mit einer gegebenen Netzwerkarchitektur (Anzahl der Schichten, Anzahl und Art der Neuronen, Ausgabefunktion, ...) soll das Verhalten des Netzwerks durch Änderung der Gewichte w_{ij} und Schwellwerte Θ_i so verändert werden, daß das Netzwerk durch seine gewichteten Eingabe-/Ausgabebeziehungen die Eingabe- /Ausgabekonstellation möglichst gut nachbildet.

3.2.10 Definition: Lernen als Optimierungsproblem

Das i-te Neuron in der L-ten Schicht(Ausgabeschicht) wird $\nu_i^{(L)}$ bezeichnet. Der Aktivierungszustand des Netzwerks wird mit $x_i^{(l)} = g(\sum_{j=1}^{n_{l-1}} w_{ij}^{(l)} x_j^{(l-1)} - \Theta_i^{(l)})$ berechnet.

Wähle die Gewichte w_{ij} und die Schwellwerte Θ_i so, daß der Fehler

$$\varepsilon = \sum_{i=1}^{n_L} \frac{1}{2}(x_i^{(L)} - d_i)^2 \to min$$

$$\mathbb{R}^{n_o n_1 + n_1 n_2 + \ldots + n_{L-1} n_L} \to \mathbb{R}^1$$

wird.

wobei n_o die Anzahl der Neuronen in der Eingabeschicht, n_1 die Anzahl der Neuronen in der ersten verborgenen Schicht, usw. ist.

3.2.11 Definition: Generalisierung

Meist die Menge der präsentierten Eingabemuster nicht erschöpfend, d.h. es existieren Eingabemuster, die dem Netzwerk nicht präsentiert wurden. Das Netzwerk paßt sich nur an Punktmengen im Lösungsraum an. Funktionen sind aber nicht nur auf endliche Punktmengen definiert, sondern meist in kontinuierlichen Bereichen. Durch Generalisierung erweitert das Netzwerk Teile von Hyperflächen auf kontinuierliche Bereiche.

3.2.12 Definition: Zeitreihe

Eine Zeitreihe ist eine nach der Zeit geordnete Sammlung von Beobachtungen(Observationen) über einen Zeitabschnitt. Beispiele derartiger Beobachtungen gibt es in vielen Bereichen, beginnend bei Wirtschaftsdaten bis technische Daten.

Beispiele für Zeitreihen sind

1. Ökonomische Zeitreihen
2. Zeitreihen aus der Physik
3. Zeitreihen aus dem Marketingbereich
4. Demographische Zeitreihen
5. Zeitreihen aus der Prozeßkontrolle

Zeitreihen haben eine spezielle Eigenschaft: Mehrere hintereinanderfolgende Beobachtungen sind in der Regel nicht voneinander unabhängig. Daher ist die Reihenfolge der Beobachtung von Bedeutung (vgl. [Chatfield, 1991, S 1ff]).

3.3 Das künstliche Neuron

Die Erkenntnisse aus biologischen Neuralen Netzwerken werden zur Entwicklung künstlicher Neuronaler Netzwerke verwendet. Künstlich heißt hier, daß einerseits die Funktionsweise mit Algorithmen auf Computern nachgebildet wird, andererseits spezielle Hardware in Form von integrierten Schaltkreisen entwickelt wird. Jener Forschungsbereich, welcher Rechenmodelle möglichst gut an die Funktionen des Gehirns anpaßt, wird auch als “Neural Computing” bezeichnet. Aleksander [Aleksander and Morton, 1990, S 1] weist auf die Begriffsvielfalt in der Literatur hin, die sich aus den verschiedensten Disziplinen der Wissenschaft ergibt. Die englischen Begriffe “Neural Computing”, “Neural

Networks", "Connectionism", "Parallel Distributed Processing", und "Connections Science" zielen auf eine Idee ab: Anwendung von Algorithmen, deren Funktionsweise sich an die des Gehirns anlehnt.

Abschließend versucht Aleksander dennoch eine Definition von "Neural Computing", die in 3.2.3(Seite 11) wiedergegeben ist.

Im folgenden sollen künstliche Neuronale Netzwerke kurz als Neuronale Netzwerke und künstliche Neuronen als Units oder Prozessoren bezeichnet werden. Dies ist deswegen notwendig, um die folgenden mathematischen Modelle für künstliche Neuronale Netzwerke von den komplexen natürlichen Neuronalen Netzwerken abzugrenzen.

Ein Teil der vorliegenden Arbeit befaßt sich mit einer algorithmischen Lösung einiger Erkenntnisse aus diesem Forschungsbereich.

3.3.1 Allgemeine Modellcharakteristika

Es gibt eine große Zahl verschiedener Modelle, die sich im Detail unterscheiden, aber alle eine Variation des Grundkonzeptes der Neuronalen Netzwerke sind.

Zuerst sollen die allgemeinen Modellcharakteristika eines Neuronalen Netzwerkes definiert werden. Rumelhart und McClelland [Rumelhart et al., 1986b, S 46] definieren die Hauptmerkmale eines Neuronalen Netzwerkes ("Parallel Distributed Processing Model (PDP-Model)") folgendermaßen:

1. besteht aus Verarbeitungseinheiten (Units oder synonym Prozessoren, processing units)
2. hat einen Aktivierungszustand (state of activation)
3. es gibt eine Funktion zur Erzeugung des Ausgangssignals (output function)
4. es gibt ein Verbindungsmuster (pattern of connectivity) zwischen den Units
5. es gibt eine Ausbreitungsregel (propagation rule), mit der die Aktivierung einer Unit durch das Netzwerk ausgebreitet werden kann.
6. es gibt eine Aktiverungsregel (activation rule) die ankommende Eingangssignale zusammenführt und einen neuen Aktivierungszustand für die Unit erzeugt
7. es gibt eine Lernregel (learning rule), mit welcher die Verbindungsmuster zwischen den Units durch Erfahrung geändert werden

8. es gibt eine Umgebung (environment), in der das Netzwerk arbeitet

Die Grundelemente eines Neuronalen Netzwerkes sind in Abbildung 3.4 dargestellt. Es gibt eine bestimmte Anzahl von Units (Prozessoren), die durch

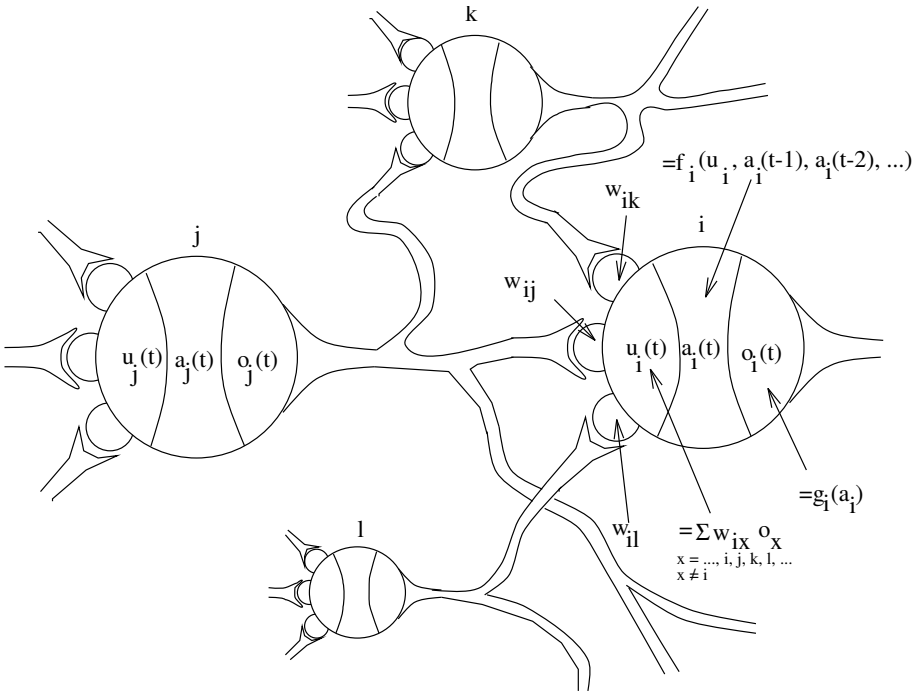

Abbildung 3.4: Grundelemente eines Neuronalen Netzwerkes

Kreise dargestellt sind. Zu jedem Zeitpunkt hat jede Unit i einen Aktivierungswert $a_i(t)$, der mit der Funktion g_i einen Ausgabewert (Output) $o_i(t)$ erzeugt. Nach Rumelhart und McClelland [Rumelhart et al., 1986b, S 47] wird dieser Ausgabewert durch eine Menge von gerichteten Verbindungen in andere Units verbreitet. Dies entspricht dem Axon im biologischen Modell. Jeder Verbindung wird eine reelle Zahl zugewiesen, die Gewicht w_{ij} bezeichnet wird, welche den Einfluß der Unit auf die nächste Unit angibt. Alle Eingangssignale müssen mit einem Operator zum Nettoinput $u_i(t)$ zusammengefaßt werden (meist über die Addition). Die Aktivierung $a_i(t)$ ergibt sich durch die Funktion f_i, der Ausgabewert durch die Funktion g_i. Jene Parameter des Modells, die sich im Zeitablauf ändern, sind die Gewichte w_{ij}.

Nun sollen die einzelnen Grundelemente etwas genauer betrachtet werden:

Die Verarbeitungseinheiten (Units oder synonym Prozessoren, processing units)

Jedes Neuronale Netz besteht aus einer Menge von Verarbeitungseinheiten. Meist wird die Anzahl und Art der Verarbeitungseinheiten über dem Zeitablauf fest gehalten. Dabei kann eine Unit einerseits für konkrete Dinge im Modell, z.B.: für eine betriebswirtschaftliche Kenngröße wie das Verhältnis zwischen Kapital und Vermögenswert, andererseits kann ein Neuron auch für abstrakte, nicht unmittelbar erklärbare Eigenschaften stehen. Diese Verarbeitungseinheiten führen die gesamte Arbeit in einem Neuronalen Netz aus, und arbeiten völlig unabhängig voneinander. Es gibt keine zentrale Kontrollstelle, die den Ablauf überwacht. Die Aufgabe einer Verarbeitungseinheit besteht darin, Informationen von anderen Units zu empfangen, zu bewerten und die verarbeiteten Informationen an die Nachbareinheiten weiterzusenden. Das System arbeitet parallel, da viele einzelne Units zur selben Zeit unabhängig voneinander arbeiten.

Aleksander [Forsyth, 1989, S 104ff] betrachtet Verarbeitungseinheiten als verteilten Wissensspeicher, und vergleicht sie mit dem herkömmlichen Hauptspeicher eines Computers. Die kleinste Speichereinheit in einem digitalen Rechner ist das 'bit', welches physisch durch eine bistabile Kippschaltung (Flip-Flop) realisiert ist. In einem Neuronalen Netzwerk besteht die Speicherkapazität aus den synaptischen Verbindungen zwischen den Verarbeitungseinheiten (Gewichten). Nun stellt sich die Frage nach der Speicherkapazität eines Neuronalen Netzwerkes. Genau kann dies nicht beantwortet werden. Nimmt man an, daß es N - binäre Inputs gibt und ein Neuron mit wahr, falsch und weiß nicht antworten kann, dann gibt es 2^N verschiedene Inputs, die es Speichern muß. Die Inputs sind die Adressen zu den möglichen Outputs (wahr, falsch und weiß nicht).

Nach der Anordnung einer Unit im Netzwerk wird zwischen drei Verarbeitungseinheiten unterscheiden, die in Tabelle 3.1 gegenübergestellt sind.

Aktivierungszustand (state of activation)

Zu jedem Zeitpunkt t gibt es einen Aktivierungszustand des Netzwerkes, der z.B. durch einen Vektor von reelle Zahlen dargestellt wird.

Der Aktivierungszustand einer Unit steht für bestimmte Eigenschaften (z.B.: der Aktivierungszustand einer Unit entspricht dem Temperaturwert, der Aktivierungszustand einer weiteren entspricht der Luftfeuchte, ...), oder steht für nicht unmittelbar interpretierbare Zustände - dies trifft besonders auf die verborgenen Units im Netzwerk zu, da hier eine interne Darstellung herrscht, die von einem Außenstehenden nicht unmittelbar interpretiert werden kann.

Bezeichnung	Anordnung
Input-Units (Eingabe-Einheiten)	Empfangen Informationen von Quellen außerhalb des Netzwerkes
Output-Units (Ausgabe-Einheiten)	Senden Signale aus dem Netzwerk hinaus
Hidden-Units (Verborgene-Einheiten)	Empfangen und senden Signale an andere Verarbeitungseinheiten im Netzwerk

Tabelle 3.1: Bezeichnung der Units nach der Anordnung im Netzwerk

Aktivierungswerte	Beispiel
stetig	$a_i(t) \in \mathbb{R}$
diskret	$a_i(t) \in \{0, 1, -1\}$
gebunden	$a_i(t) \in [0, 1]$
ungebunden	$a_i(t) \in \mathbb{R}$

Tabelle 3.2: Beispiele für zulässige Aktivierungszustände

Viele Modelle unterscheiden sich in den zulässigen Aktivierungswerten. Einige mögliche Beispiele sind in Tabelle 3.2 dargestellt.

Erzeugung des Ausgangssignals

Um mit anderen Verarbeitungseinheiten im Netzwerk kommunizieren zu können, muß sie eine Nachricht erzeugen und an die Verarbeitungseinheit senden, mit denen es Verbunden ist. Das Ausgangssignal ist abhängig von der Aktivierung des Neurons. Für jede Unit gibt es eine Ausgabefunktion ("output function"), die Aktivierungswerte in Signalwerte übersetzt. Für die Unit i gibt es also eine Ausgabefunktion $g_i(a_i(t))$, die den herrschenden Aktivierungszustand $a_i(t)$ in das Ausgabesignal $o_i(t)$ transformiert: $o_i(t) = g_i(a_i(t))$

Die Ausgabefunktionen werden in drei Gruppen unterteilt, die in 3.3 dargestellt sind.

Verbindungsmuster zwischen den Verarbeitungseinheiten

Das Verbindungsmuster, also die Art der Verbindung zwischen den Verarbeitungseinheiten hat einen wesentlichen Einfluß auf die Eigenschaften eines Neuronalen Netzwerkes. Die meisten Modelle summieren die gewichteten Eingangssignale auf. Dies ergibt den gesamten Input der Unit, der auch Nettoinput oder

Gruppe
Identitätsfunktion $g(x) = x$
Sigmoide Funktionen z.B.: $g(x) = \frac{1}{1+\exp^{-x}}$
Stochastische Funktionen

Tabelle 3.3: Unterteilung der Ausgabefunktionen ("output functions")

Erregungspotential bezeichnet wird. Für die Unit i ergibt sich der Nettoinput u_i folgendermaßen:

$$u_i = \sum_{j=1}^{N} w_{ij} o_j \tag{3.5}$$

Das Gewicht w_{ij} in der Formel 3.5 bezieht sich auf die Stärke der Verbindung nach Unit i von Unit j. In der Literatur über Neuronale Netzwerke hat sich diese Notation durchgesetzt. Beispiel dafür sind die Publikationen von Rumelhart et al. [1986b], Dorffner [1991], Ritter et al. [1990]. Dies scheint bei der Betrachtung einer Unit angebrachter zu sein, da man wissen möchte, von welchen Inputs eine bestimmte Unit gesteuert wird. In anderen Problemstellungen wie z.B. beim Transportproblem wird umgekehrt indiziert. Es werden die Transportkosten und -mengen von Quelle i nach Ziel j angeführt (siehe [Taha, 1989, S 172ff]).

Die Ausgabewerte o_j der N-Units, die Unit i erhält, werden mit den Gewichten w_{ij} multipliziert und aufsummiert. Das ergibt den Nettoinput u_i des Neurons i.

Es gibt Ansätze, die eine Differenzierung zwischen hemmenden und verstärkenden Signalen durchführen. Dies führt zu zwei Gewichtsmatrizen für hemmende und verstärkende Einflüsse. Ein weiterer Ansatz differenziert zwischen jeder Art der Verarbeitung indem es eine Menge von Gewichtsmatrizen gibt.

Ausbreitungsregel (propagation rule)

Es muß eine Ausbreitungsregel geben, die die Ausgabewerte $o_i(t)$ an die nächsten Verarbeitungseinheiten transportiert um in diesen den Nettoinput $u_i(t)$ zu erzeugen.

Aktivierungsregel (activation rule)

Es gibt eine Aktivierungsregel mit der die Nettoinputs zu einem Aktivierungswert der Unit zusammengeführt werden. Oft wird die Funktion

$$a_i(t+1) = f_i(u_i(t), a_i(t), a_i(t-1), a_i(t-2), ...) \tag{3.6}$$

Aufbau neuer Verbindungen
Abbau bestehender Verbindungen
Änderung der Gewichte der bestehenden Verbindungen

Tabelle 3.4: Möglichkeiten der Änderung von Verbindungen zwischen Verarbeitungseinheiten

verwendet. Die einfachste Funktion ist die Identitätsfunktion $a_i(t+1) = u_i(t)$. Die oben dargestellten Formeln sind deterministisch. Meist wird die Restriktion der Differenzierbarkeit der Aktivierungsfunktion hinzugefügt, wenn es um die Minimierung des Fehlers mithilfe eines Gradientenverfahrens geht.

Lernregel (learning rule)

Die freien Parameter in einem Neuronalen Netzwerk werden durch Lernregeln verändert.

Das Verbindungsmuster kann durch drei Methoden verändert werden, die in Tabelle 3.4 gegenübergestellt sind.

Meist sind die Gewichte zwischen den Verarbeitungseinheiten die freien Parameter. Die Struktur des Netzwerkes wird meist festgelegt, als einzige freie Parameter bleiben die Gewichte. Sie werden im Zeitablauf entsprechend einer Lernregel verändert. Als Spezialfall ist der Aufbau (ein bisher auf null gesetztes Gewicht bekommt einen Wert ungleich null) und der Abbau (ein bestehendes Gewicht wird auf null gesetzt) von Verbindungen enthalten.

Eine einfache Regel besagt folgendes: Wenn die Verarbeitungseinheit i ein Signal von der Verarbeitungseinheit j erhält und beide Verarbeitungseinheiten stark aktiviert sind, dann soll das Gewicht w_{ij} erhöht werden.

Auf die Lernregeln wird im Kapitel 3.5 näher darauf eingegangen.

Umgebung (environment) des Netzwerks

Die Präsentationsreihenfolge der Eingabemuster hat einen wesentlichen Einfluß auf den Lernerfolg des Netzwerkes. Viele Modelle stellen die Umgebung des Netzwerks als eine sich in der Zeit ändernde stochastische Funktion über dem Raum der Eingabemuster dar. Zu jedem Zeitpunkt t gibt es eine Wahrscheinlichkeit $p_i(t) > 0$, daß ein bestimmtes Eingabemuster an die Eingänge eines Neuronalen Netzwerks angelegt wird. Im Allgemeinen kann die Wahrscheinlichkeitsfunktion auch von vergangenen Eingabe- und Ausgabemuster des Netzwerks abhängig sein. Meist wird jedoch eine stabile Verteilungsfunktion über die möglichen Inputs definiert, die unabhängig von vergangenen Re-

aktionen des Netzwerks ist.

Die Eingaben können als Vektor betrachtet werden. Einige Modelle können nur mit orthogonalen oder linear unabhängigen Vektoren erfolgreich lernen, andere Modelle können jede Form von Inputs verarbeiten.

3.3.2 Typische Modellansätze

McCulloch und Pitts Neuronen

Die ersten Modellansätze Neuronaler Netzwerke gehen auf die vierziger Jahre zurück. Der Neurophysiologe Warren McCulloch und der Logiker Walter Pitts stellten ein Formalmodell des Neurons auf indem sie sich an dem Modell der Touring-Maschine orientierten. Im folgenden soll das Modell kurz als MCP bezeichnet werden. Sie zeigten, daß ihr Neuron im Prinzip jede allgemeine Berechnung bei entsprechendem Festlegen der Gewichte durchführen kann. Somit könnten auch Berechnungen, die digitale Rechenanlagen durchführen, durch Neuronale Netzwerke simuliert werden. Eine derartige Lösung wäre nicht so effizient und durchschaubar, ist aber im Prinzip durchführbar. Jedes Programm einer Touring-Maschine kann in einem endlichen Netzwerk aus Neuronen implementiert werden. Der Ansatz beruht auf der Annahme, daß die Aktivierung eines Neurons im Gehirn von der Außenwelt des Netzwerkes abhängig ist. Informationen von außen werden durch Sensoren an das Netzwerk übermittelt und in den Neuronen zu beliebigen endlichen Kombinationen verknüpft. Aleksander [Aleksander and Morton, 1990, S 22 ff] gibt einen guten Überblick über das MCP. Das synaptische Potential natürlicher Neuronen wird im Modell durch ein veränderliches Gewicht dargestellt, welches den Grad der Aufmerksamkeit beim Einlangen von Impulsen an der Synapse des Neurons wiederspiegelt. Das Neuron summiert alle Signale (hemmende und erregende), die es von seinen Synapsen erhält und kann somit feststellen, ob es 'feuert' oder nicht.

Ob ein Neuron feuert oder nicht kann durch die Gleichung 3.7 bzw. 3.8 dargestellt werden:

$$x_1 w_1 + x_2 w_2 + ... + x_j w_j + ... + x_n w_n > \mu \tag{3.7}$$

$$\sum_{j\,=\,1}^{n} x_j w_j > \mu \tag{3.8}$$

Dabei ist $x_j \in \{0,\ 1\}$ der Eingabewert des Neurons j, der Einfluß des Ausgabewertes wird mit $w_j \in [-1,\ 1]$ gewichtet und μ ist ein Schwellwert, der

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 3.5: Wahrheitstafel einer ODER-Logikschaltung

überschritten werden muß, damit das Neuron feuert (einen Ausgabewert $y = 1$, $y \in \{0,\ 1\}$ liefert).

Die Abbildung 3.5 zeigt den Aufbau eines MCP-Neurons.

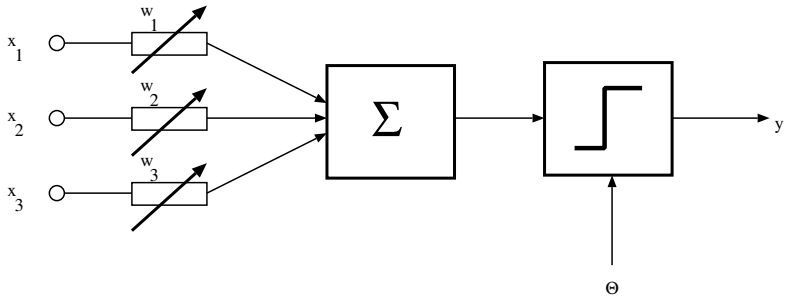

Abbildung 3.5: Aufbau eines MCP-Neurons

Es erhält die binären Eingabewerte x_1, x_2 und x_3, diese werden mit den Gewichten w_1, w_2 und w_3 und einer anschließenden Summation zu einem Aktivierungssignal zusammengefaßt. Dieser Wert wird mit einem Schwellwert μ verglichen und bei überschreiten dieses Schwellwertes feuert das MCP-Neuron indem das $y = 1$ gesetzt wird.

Bei dieser einfachen Art der Verarbeitungseinheit können die Gewichte algebrarisch durch Betrachtung der Ungleichung 3.7 ermittelt werden.

Dies sei anhand folgenden Beispiels erläutert, in dem die Funktion einer oderverknüpfenden Logikschaltung mit zwei Eingänge und einem Ausgang durch ein MCP-Neuron nachgebaut werden soll.

Die in Tabelle 3.5 dargestellte Eingabe- / Ausgabebeziehung soll das MCP-Neuron darstellen können.

Die vierte Zeile aus der Tabelle 3.5 ergibt

$1 \times w_1 + 1 \times w_2 > \mu$

bzw.

$$w_1 + w_2 > \mu \tag{3.9}$$

aus der zweiten Zeile ergibt sich

$$w_2 > \mu \tag{3.10}$$

aus der dritten Zeile ergibt sich

$$w_1 > \mu \tag{3.11}$$

und aus der ersten Zeile

$0 \times w_1 + 0 \times w_2 > \mu$

soll nicht erfüllt sein, d.h.

$$0 < \mu \tag{3.12}$$

Da nach der Definition des MCP-Neurons w_1 und w_2 im Bereich zwischen -1 und 1 liegt, muß nach Gleichung 3.9 der Schwellwert μ im Bereich zwischen -2 und 2 liegen, damit die Ungleichung unabhängig vom aktuellen Wert w_1 und w_2 erfüllt ist.

Der Schwellwert μ wird durch Gleichung 3.12 auf $\mu \in]\ 0,\ 2\ ]$ eingeengt, da $\mu > 0$ gilt.

Die Gleichungen 3.9 und 3.10 beschränken μ auf Werte kleiner als 1.

Daher liegt μ im Bereich $]\ 0,\ 1\ [$. Die Gewichte w_1 und w_2 müssen im Bereich $]\ \mu,\ 1\ ]$ liegen.

Durch Wahl des Schwellwertes μ und der Gewichte w_1 und w_2 in den oben angeführten Bereichen wird die Wahrheitstabelle erfüllt. Als Beispiel sei $\mu = 0.1$ und $w_1 = w_2 = 0.5$ angeführt.

Geometrische Veranschaulichung des Problems: Abbildung 3.6 zeigt die Funktion eines MCP-Neurons.

Dabei handelt es sich um ein Modell mit zwei Variable (x_1 und x_2), der Eingaberaum besteht also aus zwei Dimensionen.

An der Ordinate sind die Werte der Variable x_1 und and der Abszisse die Werte der Variable x_2 aufgetragen. In diesem Beispiel kann x_1 und x_2 nur die Werte null oder eins annehmen. Daher gibt es nur vier Punkte im Raum, die durch $(0, 0)$, $(0, 1)$, $(1, 0)$ und $(1, 1)$ gegeben sind.

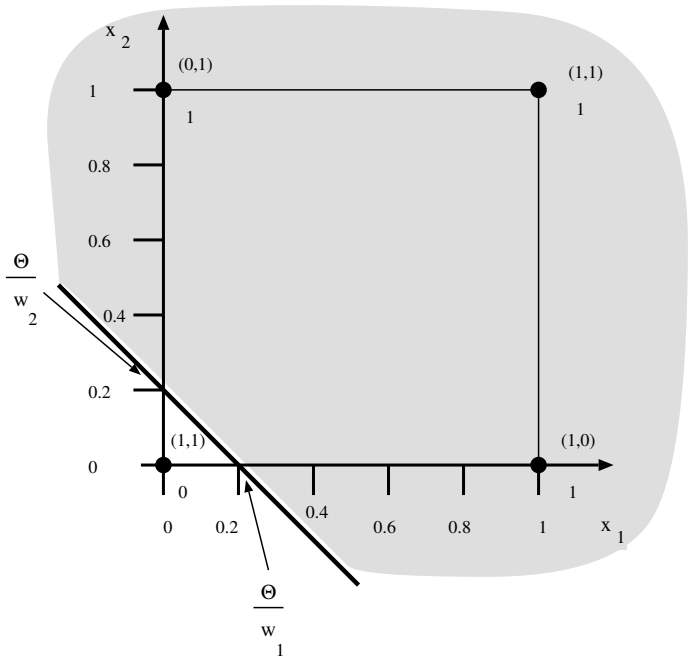

Abbildung 3.6: Geometrische Veranschaulichung der Funktion eines MCP-Neurons

Das MCP-Neuron hat nun die Aufgabe, den Raum so zu teilen, daß das gewünschte Verhalten gemäß der Tabelle 3.5 erreicht wird.

Der Eingaberaum wird durch die Gleichung 3.7 geteilt. Die Geradengleichung für unser Beispiel lautet:

$$x_1 w_1 + x_2 w_2 = \mu \tag{3.13}$$

Damit lassen sich die Stützpukte der Geraden an der Abszisse und der Ordinate berechnen:

Für die Ordinate: Mit $x_1 = 0$ und einsetzen in Gleichung 3.13 ergibt: $x_2 = \frac{\mu}{w_2} = \frac{0.1}{0.5} = 0.2$.

Für die Abszisse: Mit $x_2 = 0$ und einsetzen in Gleichung 3.13 ergibt: $x_1 = \frac{\mu}{w_1} = \frac{0.1}{0.5} = 0.2$.

Diese Gerade trennt den Eingaberaum in zwei Teile. In einem Teil wird für die Eingaben eins ausgegeben (das Neuron feuert) im anderen Teil wird null ausgegeben (das Neuron bleibt inaktiv).

Für das gezeigte Beispiel kann des MCP-Neuron das gewünschte Verhalten erzeugen, da im Lösungsraum (hier Fläche) jene Eingabewerte, bei denen das Neuron 'feuern' soll von jenen Eingabewerten, bei denen das Neuron inaktiv

binäre Schwellwertfunktion
berechnet eine gewichtete Summe seiner Inputs
feuert / feuert nicht, je nach dem, ob die gewichtete Summe einen gewissen Schwellwert überschreitet oder nicht

Tabelle 3.6: Zusammenfassung: Eigenschaften eines MCP-Neurons

bleibt, durch eine Gerade getrennt werden kann.

Allgemein kann für ein MCP-Neuron mit n-Eingabewerten folgendes gesagt werden: Das MCP-Neuron kann jede Funktion lernen, die durch eine $(n-1)$-dimensionale Hyperebene definiert werden kann und den n-dimensionalen Hyperraum teilt.

D.h. das MCP-Neuron kann nur linear separable Funktionen lernen.

Eine Funktion $f(x_1, x_2, ..., x_n)$ ist separabel, wenn sie als eine Summe von n-Funktionen mit einer Variable $f_1(x_1), f_2(x_2), ..., f_n(x_n)$ dargestellt werden kann (vgl. [Taha, 1989, S 786]):

$$f(x_1, x_2, ..., x_n) = f_1(x_1) + f_2(x_2) + ... + f_n(x_n) \tag{3.14}$$

Eine Funktion heißt linear separable Funktion, wenn sie als eine Summe von n-linearen Funktionen mit einer Variable dargestellt werden kann.

Bei nicht linear separablen Funktionen können folgende Probleme auftreten. Einerseits kann es keine Fläche (Gerade) geben, die die zwei Typen von Punkten trennt. Andererseits können die Punkte genau in einer Linie liegen. Genauer gesagt, wenn die Punkte nicht in einer allgemeinen Lage sind.

Die Eigenschaften eines MCP-Neurons sind in der Tabelle 3.6 zusammengefaßt:

Die Modellgleichung lautet:

$$x_i(t+1) = \Theta\left(\sum_j w_{ij} x_j(t) - \mu_i\right) \tag{3.15}$$

Dabei kann der Ausgabewert i zum Zeitpunkt t die Werte $x_i(t) \in \{0, 1\}$ annehmen. Dies entspricht dem 'feuern' oder nicht 'feuern' eines MCP-Neurons. t ist ein zeitdiskreter Wert, pro Zeitpunkt erfolgt ein Verarbeitungsschritt.

$\Theta(u)$ ist die sigmoide Aktivierungsfunktion und ist folgendermaßen definiert:

$$\Theta(u) = \begin{cases} 1 & \text{wenn } u \geq 0 \\ 0 & \text{sonst} \end{cases} \tag{3.16}$$

Die Gewichte w_{ij} sind die Stärke des synaptischen Einflusses von Neuron j zu Neuron i und kann positive oder negative Werte annehmen. Dies entspricht einem aktivierenden bzw. hemmenden synaptischen Potential.

Der Unterschied von MCP-Neuronen zu natürlichen Neuronen besteht im wesentlichen aus folgenden Punkten:

1. Natürliche Neuronen erzeugen eine Sequenz von Impulsen. Das MCP-Neuron erzeugt einen Ausgabewert, welcher zwei Zustände annehmen kann ('feuern', nicht 'feuern'). Erweiterungen sind kontinuierliche Ausgabewerte. Dabei geht aber eine mögliche Phaseninformation verloren, die bei Pulssequenzen enthalten ist.

2. Natürliche Neuronen haben keine Schwellwertfunktion, sondern reagieren auf die empfangenen Signale in einer kontinuierlichen Form, indem sie zumindest in Stufen auf die Signale reagieren.

3. Natürliche Neuronen führen eine Summation von Inputwerten durch, die in einer nichtlinearen Beziehung zueinander stehen. Ein derartiges Verhalten könnte durch zusammenschalten mehrerer MCP-Neuronen erreicht werden. Beispiel: Modellierung einer logischen Funktion (z.B. OR) durch mehrere MCP-Neuronen.

4. Natürliche Neuronen arbeiten nicht synchron zueinander, es gibt keine zentrale Uhr, die die Verarbeitung steuert.

5. Die Verarbeitung von Signalen bei natürlichen Neuronen wird durch stochastische Größen beeinflußt. Damit diese chemische Übertragung funktioniert, müssen Ionen vorhanden sein. Die Ionenzahl ändert sich zufällig und ist nicht vorhersagbar.

Das MCP-Neuron kann in den folgenden Eigenschaften erweitert werden:

1. Die Schwellwertfunktion wird durch eine kontinuierliche, nichtlineare Funktion ersetzt. Es kommt mehr auf die Nichtlinearität dieser Funktion an, als auf die bestimmte Form dieser Funktion.

2. Eine stochastische Verallgemeinerung des MCP-Modells berücksichtigt das stochastische Schwanken der Transmittersubstanzen bei den Synapsen.

Eine mögliche Erweiterung des MCP-Modells kann durch folgende Gleichung dargestellt werden:

$$x_i := g\left(\sum_j w_{ij} x_j - \mu_i\right) \tag{3.17}$$

Die kontinuierliche Aktivierung des Neurons wird durch die Zahl x_i ausgedrückt. Die Schwellwertfunktion $\Theta(u)$ wurde durch eine nichtlineare Funktion $g(u)$ ersetzt, die Aktivierungsfunktion, Verstärkungsfunktion oder Transferfunktion bezeichnet wird. Die Units werden nun in zufälliger Reihenfolge aktiv und berechnen von den Inputwerten den Outputwert. Es erfolgt eine asynchrone Berechnung eines neuen Systemzustandes. Daher gilt die Gleichheitsbeziehung nicht für die oben dargestellte Formel. Dies wurde durch das Zeichen := dargestellt. Mit diesem Modell wird zwar nicht das Ziel einer detaillierten Modellierung natürlicher Neuronen verfolgt, vielmehr wird versucht wichtige Eigenschaften dieser in den künstlichen Modellen zu integrieren.

Vieles baut auf die Erkenntnisse von McCulloch und Pitts auf. Es geht um veränderliche Gewichte und ihre Beziehung zu bestehenden und abgebrochenen Verbindungen. Dazu zählt die Idee, daß ein Neuron eine einfache Summations- und Schwellwertfunktion ausführt, dazu.

Kritisch bleibt zu diesem Modell anzumerken, daß es nicht die tatsächliche Funktion des Gehirns nachbildet. Lernen beinhaltet eine noch nicht erforschte intelligente Operation im Neuron. Erweiterungen des Modells bestehen z.B. darin die Inputsignale in hemmende und aktivierende Signale zu teilen.

3.4 Netzwerke von Neuronen

Ein weiterer Aspekt bei der Simulation künstlicher Neuronaler Netzwerke ist die Anordnung und Verbindung der Grundelemente - den Neuronen zu einem Netzwerk.

Viele Modelle unterstellen eine bestimmte Netzwerkstruktur. Es werden vorallem jene Netzwerke gesucht, die eine einfache Verflechtung der Neuronen untereinander haben, ohne daß die allgemeinen Eigenschaften eines vollverbundenen Netzwerks verloren gehen. Einige in den Simulationen implementierte Netzwerktopologien werden auf ihre Eigenschaften untersucht.

Aus Vereinfachungsgründen werden die Neuronen meist in Schichten angeordnet. Es gibt eine Eingabe- und eine Ausgabeschicht, dazwischen können keine, eine oder mehrere Zwischenschichten liegen. Jede Schicht erhält eine Nummer, um sie eindeutig zu identifizieren. Ein mögliches Netzwerk zeigt die Abbildung 3.7, die ein sogenanntes “feedforward”-Netzwerk darstellt, in dem nur Verbindungen in eine Richtung erlaubt sind. Die Kreise stellen die Neuronen dar, jedes Neuron der jeweiligen Schicht bekommt eine Eingabe von jedem Neuron in der darunter liegenden Schicht, außer die Neuronen in der Eingabeschicht. Diese erhalten von der Umgebung des Netzwerkes die Eingabe - hier werden die Muster dem Netzwerk präsentiert.

Die Verbindungen zwischen den Neuronen sind unidirektional, d.h. in eine

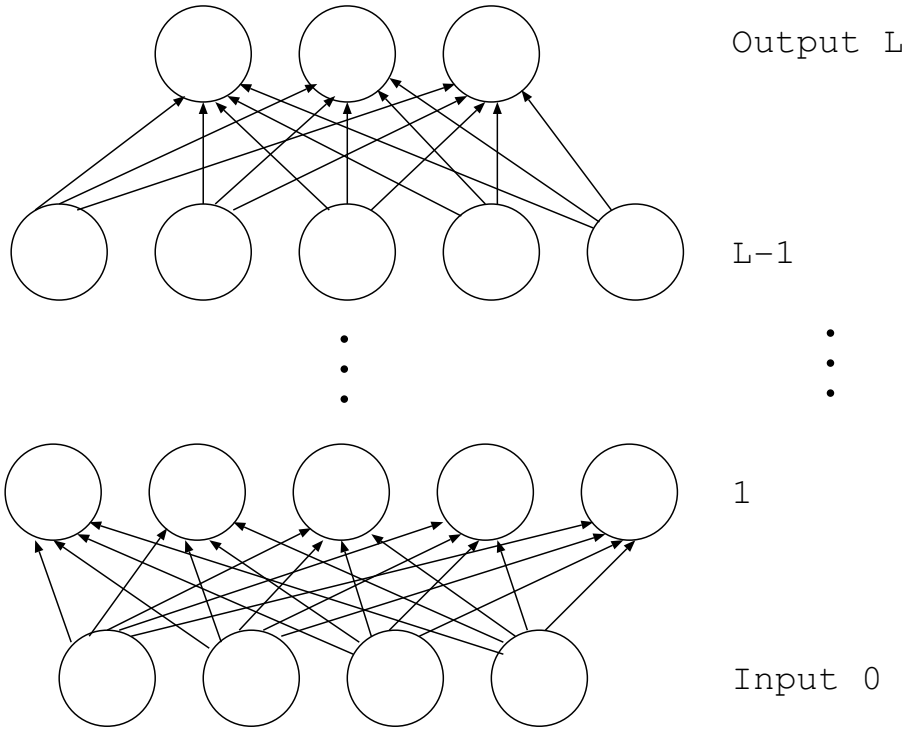

Abbildung 3.7: Anordnung der Neuronen zu einem Netzwerk

Richtung. Es gibt in diesem Modell keine Rückkoppelungsbeziehungen. Eine weitere Einschränkung besagt, daß ein Neuron nur von in der darunter liegenden Schicht befindlichen Neuronen Signale empfangen darf. Alle anderen Möglichkeiten der Verbindung von Neuronen sind verboten. Es ist z.B. unmöglich, daß ein in der Ausgabeschicht befindliches Neuron unmittelbar Signale von der Eingabeschicht empfängt, wenn dazwischen noch eine oder mehrere Schichten sind. Durch diese Einschränkung ist der Netzwerkaufbau einfacher, ohne daß die Universalität des Netzwerks verloren geht.

3.4.1 Das Einschichtenmodell

Die einfachste Netzwerkstruktur besteht aus einer Eingabe- und einer Ausgabeschicht. Es gibt keine Zwischenschicht. Ein Beispiel für ein derartiges Netzwerk ist das Percepton.

Dieser Typ von Neuron wurde von Rosenblatt [Rosenblatt, 1988, S 89ff] entwickelt. In der Entwicklungsgeschichte Neuronaler Netzwerke wurden die Perceptons lange Zeit auf ihre Eigenschaften untersucht. Perceptons wurden aufgrund ihres einfachen Aufbaus intensiv auf ihre Limitationen untersucht. Einige theoretische Einschränkungen waren von Anfang an bekannt, wie z.B.

die Erfordernis der linearen Separabilität der Datenpunkte für eine richtige Klassifikation.

Minsky und Papert [Minsky and Papert, 1988, S 162 ff] diskutierten die Limitationen der Perceptons.

Das einfache Percepton besteht aus drei Teilen: Einer Retina (Fläche an der sich Sensoren befinden), einer Menge von Assoziationseinheiten (A-Einheiten) und einer Menge von Responseeinheiten (R-Einheiten). In der Regel zeigen viele A-Einheiten auf eine R-Einheit (siehe Abbildung 3.8).

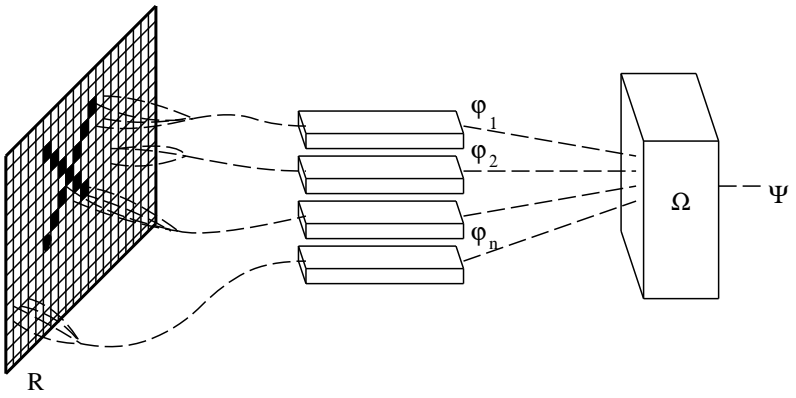

Abbildung 3.8: Das einfache Percepton

Das ist auch das simpleste Konzept einer parallelen Verarbeitung. Die Funktion $\Psi(X)$ wird in zwei Stufen berechnet. Zuerst wird unabhängig voneinander eine Menge von Funktionen $\varphi_1(X)$, $\varphi_2(X)$, ..., $\varphi_n(X)$ berechnet. Die Ergebnisse werden dann durch die Funktion Ω zum Wert Ψ zusammengefaßt.

Die Ausgabeeinheiten sind Schwellwert-Logik Einheiten. Die Ausgabe kann nur zwei Zustände annehmen. Minsky und Papert sahen diese Ausgabewerte nicht als null oder eins, sondern als falsch oder wahr an. Die Einheiten berechnen ein logisches Prädikat der Eingabewerte.

Während Rosenblatt die Funktion der Einheiten als Berechnung einer Klassifikation bezeichnete, war es für Minsky und Papert die Berechnung eines logischen Prädikats. Die A-Einheiten bilden eine Familie lokaler Prädikate Ψ, die mit der Retina verbunden sind. Die Retina besteht aus Punkten in der Ebene.

Damit war es Minsky und Papert möglich eine Analyse der Percepons mit geometrischen Figuren, die an die Retina gelegt wurden, durchzuführen.

Das einfache lineare Percepton (siehe Abbildung 3.9) summiert die gewichteten Werte der lokalen Prädikate von den A-Einheiten auf und vergleicht sie mit einem Schwellwert, und antwortet mit einem einzigen Prädikat $\Psi(X)$. Die

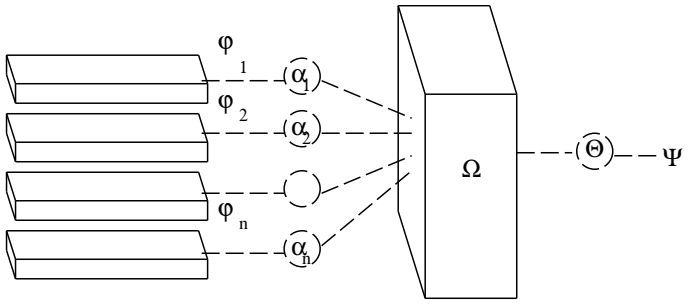

Abbildung 3.9: Die Funktionsweise des Perceptons

Berechnung des Prädikats $\Psi(X)$ kann man in zwei Stufen sehen:

Stufe I:	Die Berechnung von vielen Eigenschaften oder Merkmale φ_α. Diese sind sehr einfach zu berechnen, da jedes Merkmal nur von einem sehr kleinen Teil des Definitionsraumes R abhängig ist.
Stufe II:	Ein Entscheidungs-Algorithmus Ω kombiniert die Ergebnisse der Stufe I zu dem Prädikat Ψ. Diese Entscheidungsfunktion muß homogen, einfach programmierbar und berechenbar sein. Diese Stufe kann auch als gewichtete Abstimmung ("weighted voting") oder als Linearkombination der Prädikate der Stufe I gesehen werden. Das ist das sogenannte Perceptron-Schema (siehe Definition 3.2.6, Seite 12). Die Werte der Variable $\alpha_1, \alpha_2, ..., \alpha_n$ müssen so bestimmt werden, daß $\Psi(X) = 1$ wenn und nur wenn $\sum_{\varphi \epsilon \Phi} \alpha_\varphi \varphi(X) > \Theta$.

Die Frage, die sich bei einem derartigen Modell stellt, ist folgende: Welche logischen Funktionen können das Prädikat $\Psi(X)$ berechnen?

Minsky und Papert diskutierten das geometrische Prädikat der Verbundenheit einer geometrischen Figur. Ein derartiger zweidimensionaler geometrischer Körper kann z.B. ohne abheben des Zeichenstiftes gezeichnet werden. Die beiden Forscher zeigten, daß ein Percepton die Verbundenheit eines geometrischen Körpers nicht berechnen kann.

Es wurden zwei Limitationen der untersuchten Prädikate unterstellt: Bei "order limited" Prädikaten konnte nur eine beschränkte Anzahl von Punkte auf der Retina mit einer lokal entscheidenden Unit verbunden sein. Die zweite Limitation waren die "diameter limited" Prädikate, wo nur eine im Durchmesser beschränkte geometrische Region auf der Retina mit der lokal entscheidenden Unit verbunden ist.

Die Definition eines Perceptons ist im Kapitel Definitionen (3.2.6, Seite 12) wiedergegeben.

Der Beweis wurde mit einfachen geometrischen Figuren erbracht, die in Abbildung 3.10 dargestellt sind.

Kein im Durchmesser begrenztes Percepton kann die Verbundenheit einer geometrischen Figur feststellen. Es gibt also keinen Ausgabewert Ψ, der nur und nur dann den Wert eins liefert, wenn es sich um einen verbundene geometrische Figur handelt.

Diese Aussage kann durch folgenden Gedankengang bewiesen werden: Es wird angenommen, daß der Durchmesser des Eingabebereiches für für ein Neuron gegeben ist. Die Versuchsanordnung könnte wie in Abbildung 3.10 aussehen. Nimmt man an, daß das Percepton zusammenhängende Figuren (X_{01} und X_{10}) und nicht zusammenhängende Figuren (X_{00} und X_{11}) unterscheiden kann, muß die Summe $\sum \alpha_\varphi \varphi$ im ersten Fall größer, im zweiten Fall kleiner als Θ sein.

Für unser Beispiel ergibt sich folgende Ungleichung:

$$\left(\sum_{Gruppe\ 1} \alpha_\varphi \varphi(X) + \sum_{Gruppe\ 2} \alpha_\varphi \varphi(X) + \sum_{Gruppe\ 3} \alpha_\varphi \varphi(X) \right) - \Theta > 0 \qquad (3.18)$$

Für das Muster X_{00} muß der linke Ausdruck der Ungleichung 3.18 negativ sein, da es sich um eine nicht verbundene Figur handelt. Beim Präsentierten der nächsten Figur X_{10} ändert sich nur etwas am Eingabemuster der Neuronen der Gruppe 1. Die Summe $\sum_{Gruppe\ 1}$ muß das Gesamtergebnis so stark verändern, daß der gesamte linke Ausdruck positiv wird. Wird hingegen statt X_{00} die Figur X_{01} dem Netzwerk präsentiert, ändert sich nur das Muster für die Gruppe 2, daher muß die $\sum_{Gruppe\ 2}$ ebenfalls steigen. Beide Summen $\sum_{Gruppe\ 1}$ und $\sum_{Gruppe\ 2}$ müssen bei der Präsentation der Figur X_{11} um den selben Betrag steigen, da beide Gruppen die selbe Veränderung sehen. Den Neuronen der Gruppe 3 wird in allen Fällen die selben Muster präsentiert. Für das Muster X_{11} ist der Wert des linken Ausdruckes noch mehr positiv, als in den Fällen X_{10} und X_{01}. Tatsächlich gewünscht wäre aber ein Wert kleiner als null des linken Ausdrucks. Damit ist anhand des im Durchmesser beschränkten Perceptons bewiesen, das es keine Anordnung geben kann, die die Verbundenheit oder nicht Verbundenheit einer geometrischen Figur feststellen kann.

Die Tabelle 3.7 faßt die tatsächlichen und gewünschten Ergebnisse des Versuchsaufbaus noch einmal zusammen.

Dieser Beweis stützt sich nur auf die Geometrie der Muster und die Algebra der gewichteten Prädikate und hat nichts mit Lernen oder Wahrscheinlichkeitstheorie, noch etwas mit Hyperebenen in einem Hyperraum zu tun. Die “order limited” Perceptons können ebenfalls die Verbundenheit eines geometrischen

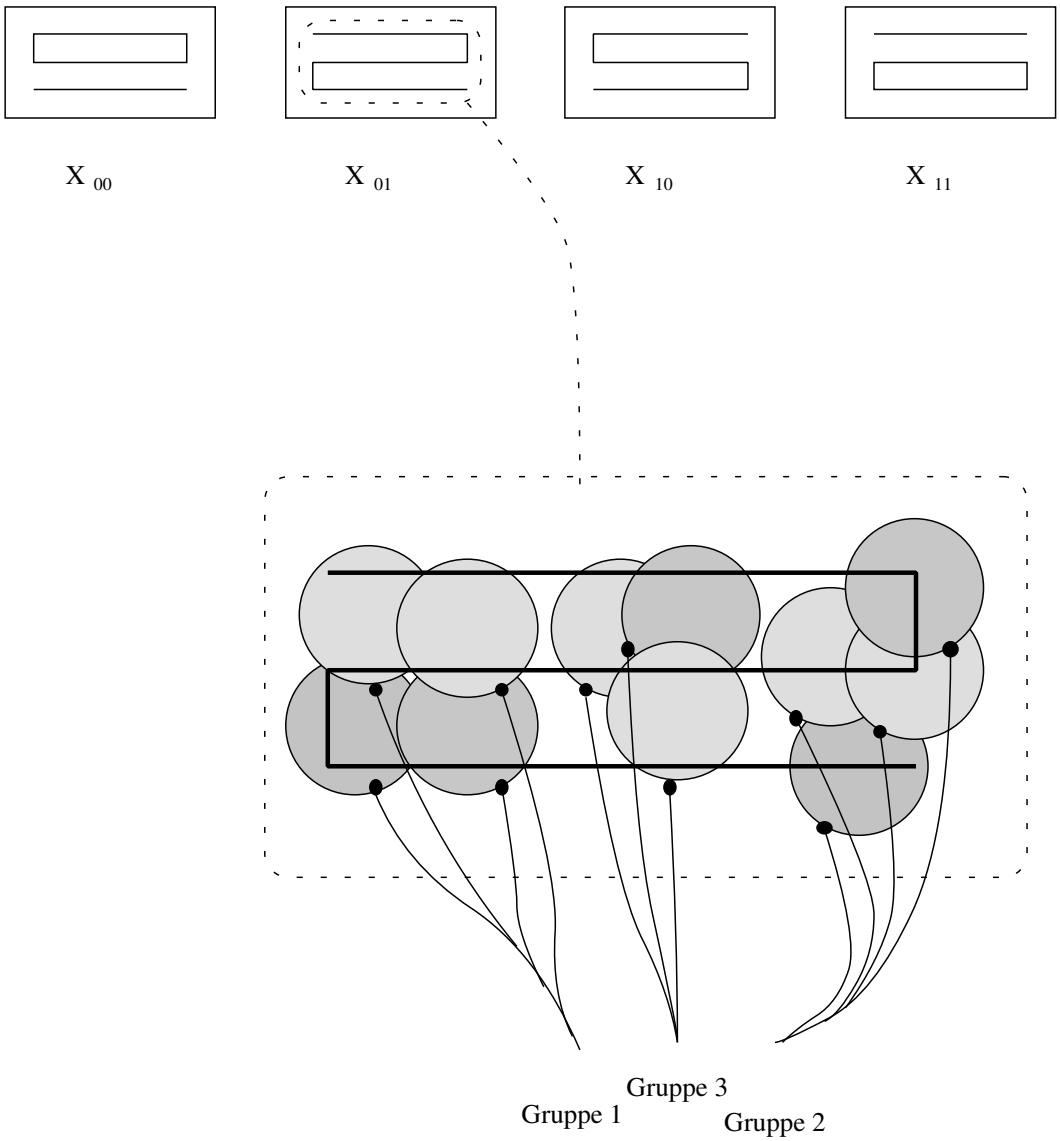

Abbildung 3.10: Geometrische Figuren für den Beweis, daß die Verbundenheit eines geometrischen Körpers nicht von einem Percepton erkannt werden kann

Muster	$\sum_{Gruppe\ 1}$	$\sum_{Gruppe\ 2}$	$\sum_{Gruppe\ 3}$	Wert des linken Ausdrucks der Ungleichung
X_{00}	-	-	-	negativ
X_{10}	positiv	-	-	positiv
X_{01}	-	positiv	-	positiv
X_{11}	positiv	positiv	-	positiv (gewünscht: negativ)

Tabelle 3.7: Zusammenfassung: Aktivierungswerte der drei Gruppen von Neuronen bei einer Versuchsanordnung mit im Durchmesser begrenzten Perceptons

Körpers nicht berechnen, der Beweis ist in diesem Fall aufwendiger und wird hier nicht näher erläutert.

3.4.2 Das Mehrschichtenmodell

Die Art und Anordnung der Neuronen im Einschichtenfall bezeichnet man als "Simple Perceptons". Sie bestehen aus einer Schicht von Verarbeitungseinheiten, da zwischen dem Eingabe- und Ausgabebereich nur ein Neuron geschaltet ist.

Eine Erweiterung der Simple Perceptons besteht in Hintereinanderschaltung mehrerer Neuronen zwischen dem Eingabe- und Ausgabebereich. Diese Art von Netzwerk wird auch "Multilayer-Perceptons Netzwerk" oder "Assoziationsnetzwerk" oder nach der angewendeten Lernregel auch "Backpropagation Netzwerk" bezeichnet. Es wird in weiterer Folge als Multilayer-Perceptons Netzwerk bezeichnet und im Kapitel Definitionen(3.2.7, Seite 12) der Begriff noch genau festgelegt. Der Frage nach dem Sinn bzw. dem Nutzen eines derartigen Netzwerkes wird in diesem Abschnitt nachgegangen. Abschließend wird noch auf die Eigenschaften dieser Art von Netzwerk eingegangen.

Ein derartiges Netzwerk besteht aus einer Eingabeschicht mit d Verarbeitungseinheiten, einer oder mehreren Zwischenschichten und einer Ausgabeschicht mit e Verarbeitungseinheiten.

Als Beispiel sei ein Netzwerk mit drei verborgenen Schichten ("hidden layers") angeführt.

Dabei handelt es sich um unidirektionale Verbindungen von der Eingabe- zur Ausgabeschicht. Eine gegebene Menge S von N Vektoren in d Dimensionen wird dem Netzwerk präsentiert. Das Netzwerk realisiert eine Funktion F in einem e dimensionalen Hyperraum. Als Beispiel sei $F : S \rightarrow \{1, -1\}^e$ angeführt. Die Funktion F wird im Spezialfall $e = 1$ Dichotomie genannt. Der Hyperraum wird durch die Funktion F in diesem Fall zweigeteilt. Dem Netz-

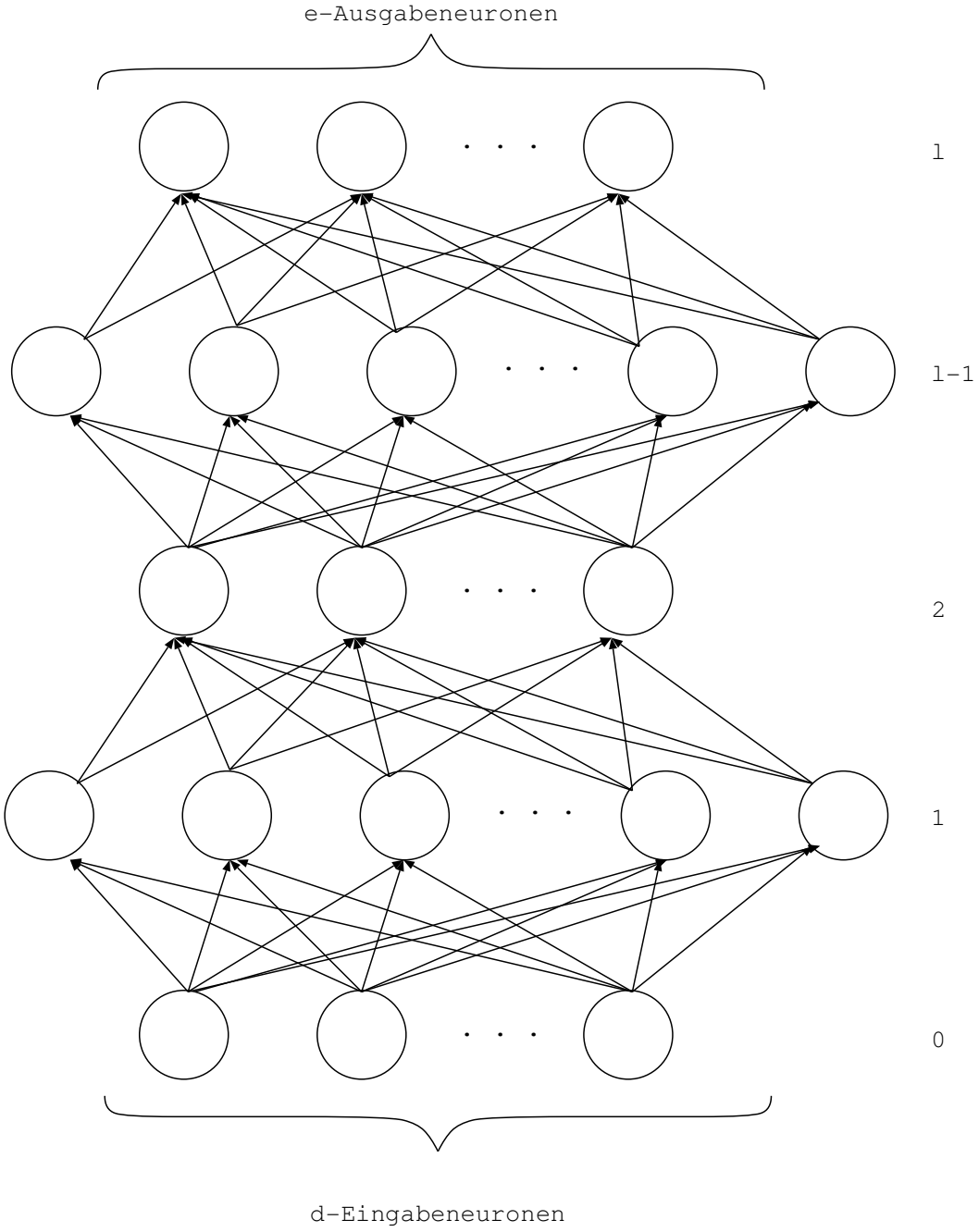

Abbildung 3.11: Beispiel eines Multilayer-Perceptons Netzwerks mit drei verborgenen Schichten

werk wird ein Eingabevektor $\mathbf{s} \in S$ präsentiert. Die Aktivierungswerte der Neuronen werden Schicht für Schicht berechnet bis die Aktivierungswerte der Ausgabeschicht berechnet werden. Es ist das Ziel, daß das Netzwerk für jeden Eingabevektor $\mathbf{s} \in S$ einen Ausgabewert erzeugt, der dem Wert von $F(\mathbf{s})$ entspricht. Ist das der Fall, dann kann das Netzwerk die Funktion *berechnen* oder *erkennen*.

Die Arbeitsweise der Neuronen ist in Abbildung 3.12 dargestellt.

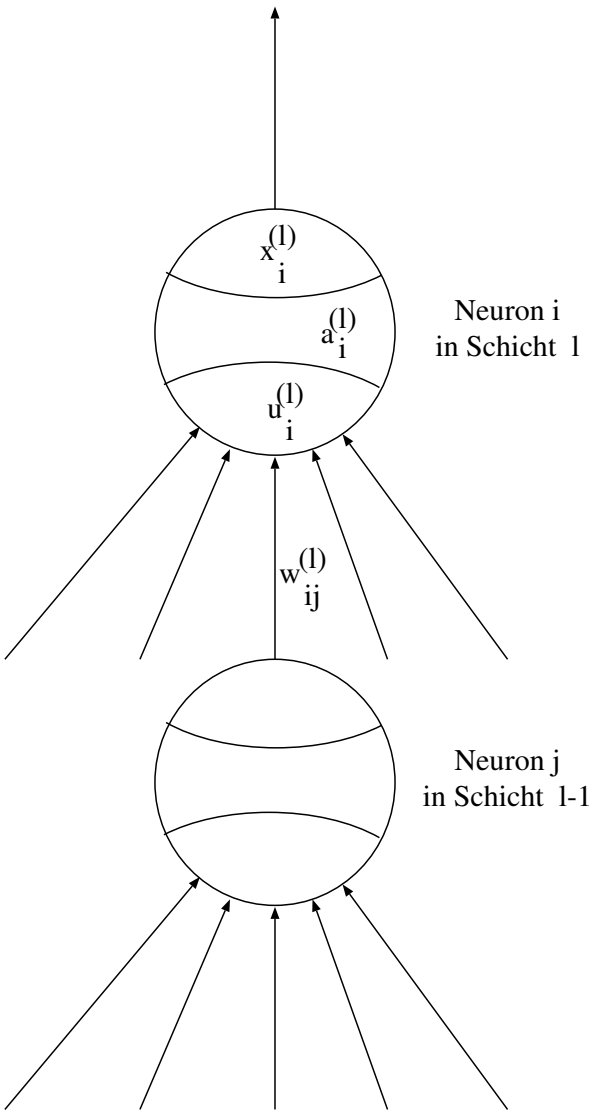

Abbildung 3.12: Detaildarstellung der der Signalausbreitung in einem Multilayer-Perceptons Netzwerk

Der Wert $x_i^{(l)}$ ist der Ausgabewert oder Zustand des Neurons i in der Schicht l

und wird mit

$$x_i^{(l)} = g(a_i^{(l)}) \tag{3.19}$$

berechnet, wobei $g(\cdot)$ eine monoton steigende Ausgabefunktion ist (siehe Ausgabefunktionen 4.1.3, Seite 49).

Der Wert $a_i^{(l)}$ ist die Aktivierung des Neurons i in der Schicht l und wird mit

$$a_i^{(l)} = f(u_i^{(l)}, a_i^{(l)}(t-1), a_i^{(l)}(t-2), ...) \tag{3.20}$$

berechnet, wobei $f(\cdot)$ die Aktivierungsfunktion (meist Identitätsfunktion) ist.

Der Wert $u_i^{(l)}$ ist der Nettoinput des Neurons i in der Schicht l und wird mit

$$u_i^{(l)} = \left(\sum_{j=1}^{n^{(l)}} w_{ij}^{(l)} x_j^{(l-1)} \right) - \Theta_i^{(l)} \tag{3.21}$$

berechnet, wobei $w_{ij}^{(l)}$ der Gewichtswert von Neuron j in Schicht $l-1$ zu Neuron i in Schicht l, $x_j^{(l-1)}$ ist der Ausgabewert des Neurons j in Schicht $l-1$ und $\Theta_i^{(l)}$ ist ein Schwellwert (Bias), der von der Summe der gewichteten Aktivierungen abgezogen wird.

Die Berechnung der Aktivierung des Netzwerks beginnt bei der Eingabeschicht und endet in der Ausgabeschicht. Der Zustand einer Schicht wird in die nächste Schicht fortgeschrieben (Vorwärtsrechnung, "forward pass").

Einen wichtigen Beweis hat Kurt Hornik im Zusammenhang mit Multilayer Feedforward Netzwerken erbracht - die Eigenschaft eines derartigen Netzwerks als universeller Approximator zu fungieren (vgl. Hornik et al. [1989]). Gewöhnliche Multilayer Feedforward Netzwerke mit einer einzigen Zwischenschicht (Hidden Layer) und beliebig gewählter Schwellwertfunktion kann jede Borel meßbare Funktion approximieren, von einem endlich dimensionalen Raum in einen anderen in jeder gewünschten Genauigkeit, vorausgesetzt, daß genügend viele Neuronen in der Zwischenschicht vorhanden sind. In diesem Sinne sind Multilayer Feedforward Netzwerke eine Klasse von universellen Approximatoren. Diese Erkenntnis hat für die theoretische und praktische Weiterentwicklung der Neuronalen Netzwerke große Bedeutung. Das entkräftet auch die Ansicht von Minsky und Papert Minsky and Papert [1988], daß die Limitationen der Simple Perceptons (ein Netzwerk ohne Zwischenschicht) auch für die Multilayered Perceptons gelten könnten (vgl. Kapitel 3.4.1).

Mögliche Designprobleme bei einem Neuronalen Netzwerk können daher nicht in der grundlegenden Unfähigkeit des Netzwerks liegen. Vielmehr müssen die Ursachen in folgenden Bereichen gesucht werden:

- falsche Lernmethode, mit der die Gewichte des Netzwerks verändert werden um das gewünschte Verhalten zu erzielen
- zu geringe Anzahl von Neuronen in der Zwischenschicht
- die dem Netzwerk präsentierten Daten sind nicht in der richtigen Form kodiert
- das Bestehen einer stochastischen statt einer deterministischen Beziehung zwischen Eingabe- und Zielwerten

In der Literatur sind auch schon Ansätze zu finden, die nach der Anzahl von Neuronen in der Zwischenschicht fragen, um eine Approximation in einer bestimmten Genauigkeit zu erreichen. Mit dieser Frage beschäftigt z.B. Baum Baum [1988]. Was ist das kleinste Multilayer Percepton Netzwerk, das eine bestimmte Funktion berechnen kann? Frühere Erkenntnisse nennen ein Netzwerk mit einer Zwischenschicht, die $N-1$ Schwellwert-Neuronen enthält. Dieses Netzwerk kann eine beliebige Dichotomie von N Punkten implementieren. Baum beweist, daß eine beliebige Dichotomie durch ein Netzwerk mit einer Zwischenschicht, die $\lceil \frac{N}{d} \rceil$ Neuronen enthält, für eine beliebige Anzahl von N Punkten in allgemeiner Lage in d Dimensionen realisierbar ist.

Netzwerke mit einer Zwischenschicht, die beliebige Funktionen in einen e dimensionalen Hyperraum implementieren, benötigen nur $\lfloor \frac{4N}{d} \rfloor \left\lceil \frac{e}{\lfloor \log_2(\frac{N}{d}) \rfloor} \right\rceil$ Neuronen in der Zwischenschicht.

Andere Ergebnisse zeigen, daß ein Netzwerk, das eine beliebige Funktion implementiert mindestens $\frac{Ne}{log_2(N)}$ Gewichte haben muß. Daher sind für ein Netzwerk mit einer Zwischenschicht mindestens $\frac{Ne}{d\ log_2(N)}$ Gewichte in der Zwischenschicht notwendig. Dürfen die Gewichte nur n_g verschiedene Werte annehmen, sind mindestens $\frac{Ne}{log_2(n_g)}$ Gewichte im Netzwerk notwendig.

3.5 Lernen in Neuronalen Netzwerken

Mit dem Lernen wird das Ein-/ Ausgabeverhalten des Neuronalen Netzwerks bzw. das Reaktionsverhalten zu seiner Umwelt verändert, bis das gewünschte Verhalten des Netzwerks erreicht ist. Das erlernte Wissen liegt in den Gewichten des Netzwerks. Daher bedeutet lernen ein Verändern der Stärke von Verbindungen zwischen Neuronen. Das Netzwerk legt nicht jedes Trainingsbeispiel explizit in seiner Struktur ab, sondern die Gewichtswerte werden durch das Lernen so verändert, daß das gewünschte Verhalten für ein erneutes Anlegen des Trainingdatensatzes erreicht wird.

Meist wird von einem durch zufällige Gewichtswerte erzeugten Anfangszustand des Netzwerks ausgegangen. Dann wird eine Lernregel angewendet, mit der die Gewichte des Netzwerks verändert werden, um das gewünschte Verhalten zu erzeugen. Durch wiederholtes Präsentieren von Eingaben an das Netzwerk lernt es anhand der Daten implizite Regeln, die aus den Daten vom Netzwerk extrahiert werden.

Eine Gewichtsveränderung im Netzwerk kann interpretiert werden als

a) Aufbau neuer Verbindungen
b) Abbruch vorhandener Verbindungen
c) Veränderung der Stärke vorhandener Verbindungen

Tabelle 3.8: Einteilung von Gewichtsveränderungen

Meist werden die Gewichte mit der Methode c) verändert, d.h. es wird eine bestimmte Netzwerkarchitektur mit zufällig generierten Gewichten festgelegt, die Gewichtswerte sind die freien Parameter.

Die Ansätze a) und b) werden neben der Methode c) in letzter Zeit zunehmend eingesetzt. Es geht bei diesen Ansätzen um die praktische Realisierung von Neuronalen Netzwerken mit minimaler Anzahl von Neuronen. Meist wird von einem Netzwerk mit genügend vielen Neuronen ausgegangen, das dann schrittweise verkleinert (engl. "node pruning") wird, indem nicht relevante Verbindungen aus dem Netzwerk entfernt werden.

Den Grundgedanken dieser Idee hat der Philosoph William von Ockham im 14. Jht. formuliert. Er hat oft ein Prinzip der methodischen Einfachheit bei der Erklärung von Sachverhalten angewendet, das als Ockham Razor bekannt wurde. Dieses Prinzip wird auch beim Entwurf symbolischer 'reasoning' Systeme angewendet: Wähle jenes System, daß ein Problem mit möglichst wenig Logikelemente beschreiben kann. Bei subsymbolischen oder konnektionistischen Problemlösungsmethoden bedeutet das folgende Vorgehensweise: Wähle jene Netzwerkarchitektur, die die wenigsten Verbindungen hat, da dieses System am besten generalisiert (vgl. Thodenberg [1991]). Ein Netzwerk generalisiert dann, wenn es auf nicht gelernte Eingaben richtig reagiert. Dies ist ein weiterer Maßsstab für die Bewertung der Leistungsfähigkeit eines Netzwerks.

Andere Ansätze, die eine sparsame Modellierung zur Zielsetzung haben sind Methoden, die die Gewichtswerte des Netzwerks aus dem saturierten Bereich der Aktivierungsfunktion herabsenken (engl. "weight decay"). Dies ist deswegen sinnvoll, da Lernregeln meist das Gradientenverfahren verwenden, und die Ableitung der Aktivierungsfunktion in diesem Bereich meist gegen null geht. Die voll aktivierten Neuronen bewirken dann eine geringe Gewichtsveränderung beim Lernen, das Verhalten des Netzwerks ändert sich kaum in Richtung des Optimums.

Der Fälle a) und b) können als Spezialfälle des Falls c) gesehen werden. Bei a) wird ein bisher auf null gehaltener Gewichtswert auf einen Wert ungleich null gesetzt und bei b) ein bisher von null verschiedener Gewichtswert auf null gesetzt. Das Ziel von algorithmischen Lösungen ist es, daß überflüssige Multiplikationen, die null ergeben, vermieden werden.

In der Literatur findet man sehr viele verschiedene Lernverfahren. Diese sind meist Abänderungen von bestehenden. Grundsätzlich kann man nach der Art der Präsentation der zu erlernenden Muster zwischen *überwachtem (supervised)* und *nicht überwachtem (unsupervised)* Lernen unterscheiden.

- Beim überwachten Lernen gibt es einen "Lehrer", der dem Netzwerk die Richtigkeit des Verhaltens mitteilt - diese Art von Lernen wird "reinforcement learning" bezeichnet, oder es wird dem Netzwerk das korrekte Verhalten präsentiert - diese Art von Lernen wird voll überwachtes Lernen ("fully supervised learning") bezeichnet. Die zu erlernenden Muster sind bekannt und sollen vom Netzwerk in Klassen eingeteilt werden, die ebenfalls schon gegeben sind. Man spricht in diesem Zusammenhang auch von Klassifizierung.

- Beim nicht überwachten Lernen ist das Netzwerk autonom. Es werden die Eingabemuster präsentiert, und das Netzwerk muß selbst die Eigenschaften des Datensatzes erkennen, die als Ausgabewerte an die Umwelt des Netzwerks gehen. Dem Netzwerk werden nur die Muster präsentiert, und soll daraus selbst Klassen bilden und finden. Man spricht in diesem Zusammenhang von Clustering.

Nach der Art des Lernens und nach der Art des Netzwerks können die Lernverfahren aufgeschlüsselt werden. Die Tabelle 3.9 nennt die Namen einiger bekannter Lernverfahren.

Aus dieser Aufstellung geht hervor, daß es je nach Netzwerktopologie und Art des Lernens ein Fülle von Lernverfahren gibt. Besondere Bedeutung hat das Lernverfahren Backpropagation Rumelhart et al. [1986a] erlangt, da es einfach zu implementieren ist und theoretische Erkenntnisse vorliegen, daß mit einem Netzwerk ohne Rückkoppelungsverbindungen (feedforward) jede beliebige Funktion in beliebiger Genauigkeit gelernt werden kann, wenn genügend Neuronen im Netzwerk vorhanden sind Hornik et al. [1989].

3.5.1 Hebb-Regel

Die Hebb-Regel [Hebb, 1988, Seite 50] war die erste Lernregel mit der ein Neuronales Netzwerk seine Gewichtswerte so verändern kann, damit es das gewünschte Verhalten erreicht. Hebb beschreibt die Vorgänge in den Zellen,

- Nicht überwachtes Lernen (unsupervised learning)
  - Netzwerke mit rückgekoppelnden Verbindungen (feedback)
    - Additive Grossberg (AG)
    - Shunting Grossberg (SG)
    - Binary Adaptive Resonance Theory (ART1)
    - Analog Adaptive Resonance Theory (ART2, ART2a)
    - Discrete Hopfield (DH)
    - Continous Hopfield (CH)
    - Discrete Bidirectional Associative Memory (BAM)
    - Temporal Associative Memory (TAM)
    - Adaptive Bidirectional Associative Memory (ABAM)
    - Kohonen Self-organizing Map (SOM)
    - Kohonen Topology-preserving Map (TPM)
  - Netzwerke mit unidirektionalen Verbindungen (feedforward)
    - Learning Matrix (LM)
    - Driver-Reinforcement Learning (DR)
    - Linear Associative Memory (LAM)
    - Optimal Linear Associative Memory (OLAM)
    - Sparse Distributed Associative Memory (SDM)
    - Fuzzy Assoziative Memory (FAM)
    - Counterpropagation (CPN)
- Überwachtes Lernen (supervised learning)
  - Netzwerke mit rückgekoppelnden Verbindungen (feedback)
    - Brain-State-in-a-Box (BSB)
    - Fuzzy Cognitive Map (FCM)
    - Boltzmann Machine (BM)
    - Mean Field Annealing (MFT)
    - Recurrent Cascade Correlation (RCC)
    - Learning Vector Quantification (LVQ)
  - Netzwerke mit unidirektionalen Verbindungen (feedforward)
    - Percepton
    - Adaline, Madaline
    - Backpropagation (BP)
    - Cauchy Machine (CM)
    - Adaptive Heuristic Critic (AHC)
    - Time Delay Neural Network (TDNN)
    - Associative Reward Penalty (ARP)
    - Avalanche Matched Filter (AMF)
    - Backpercolation (Perc)
    - Artmap
    - Adaptive Logic Network (ALN)
    - Cascade Correlation (CasCor)

Tabelle 3.9: Gruppierung bekannter Lernverfahren nach Art der Verbindung zwischen den Neuronen und nach den zwei Hauptgruppen der Lernverfahren

wenn in einem biologischen Neuronalen Netzwerk gelernt wird. Das Ergebins aus dieser Beobachtung ist die Hebb-Regel:

Ist ein Axon einer Zelle A so nahe, daß die Zelle B erregt wird und wiederholt am 'Feuern' der Zelle B maßgeblich beteiligt ist, kann die Effektivität der Verbindung verbessert werden, wenn die Stärke der Verbindung zwischen diesen Zellen erhöht wird.

Diese Formulierung ist ziemlich ungenau. Die Hebb-Regel sagt nichts über das Ausmaß der Gewichtsveränderung aus. Ebenso wird die Art der Berechnung der Aktivierungswerte der Neuronen nicht festgelegt. Bei einer Umsetzung der Regel in einen Algorithmus ergibt sich das Problem, daß die Gewichtswerte immer nur steigen und nie sinken können.

Die Abbildung 3.13 zeigt das Zusammenspiel der Neuronen. Die Stärke der

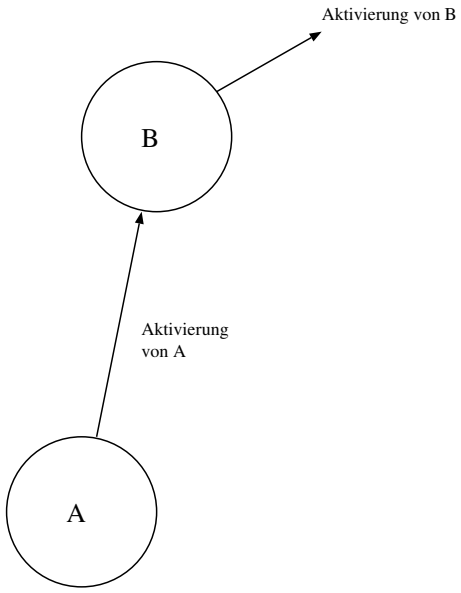

Abbildung 3.13: Das Modell für hebb'sches Lernen

Verbindung zwischen A und B wird dann erhöht, wenn die Aktivierung des Neurons positiv ist und die Aktivierung von B durch A ebenfalls positiv ist.

Die Hebbregel wurde von Grossberg zum neo-hebb'schen Lernen erweitert. Im einfachsten Fall besteht die Lernregel aus zwei dynamischen Differentialgleichungen. Eine stellt die Änderung der Aktivierung eines beliebigen Netzwerks zu einem bestimmten Zeitpunkt die andere stellt die Gewichtsänderungen einer beliebigen Verbindung im Netzwerk dar. Bei einer gegebenen Startbedingung kann die zukünftige Aktivierung und Gewichtsveränderung des zu jedem Zeitpunkt berechnet werden.

3.5.2 Delta-Regel

Diese Lernregel ist dann anwendbar, wenn der Fehler in der Ausgabeschicht direkt den Neuronen in der Eingabeschicht zugeordnet werden kann. Das Netzwerk kann daher keine (verborgenen) Neuronen zwischen der Eingabeschicht und der Ausgabeschicht haben. Der Fehler kann für verborgene Neuronen nicht mit dieser Regel berechnet werden.

Für das Netzwerk in der Abbildung 3.14 berechnet man die Gewichtsänderung

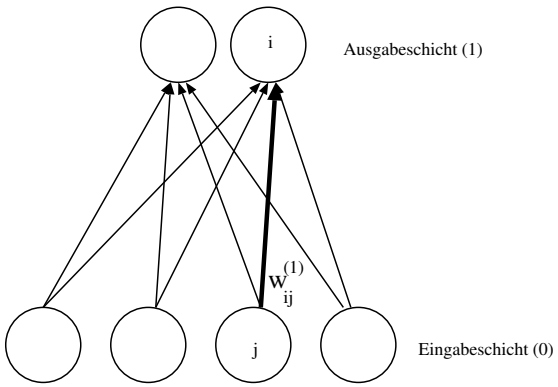

Abbildung 3.14: Das Modell für die Delta-Regel

mit

$$\Delta w_{ij}^{(1)} = \eta(d_i - x_i^{(1)})x_j^{(0)}, \tag{3.22}$$

wobei $\eta \in (0,1)$ die konstante Lernrate ist, d_i der gewünschte Ausgabewert beim Neuron i, $x_i^{(1)}$ der tatsächliche Ausgabewert des Neurons i in der Ausgabeschicht (Index 1) und $x_j^{(0)}$ der an das Netzwerk angelegte Wert am Neuron j in der Eingabeschicht (Index 0) ist.

Die Gewichte werden mit

$$w_{ij}^{neu} = w_{ij} + \Delta w_{ij} \tag{3.23}$$

verändert.

Diese Berechnung läßt sich folgendermaßen begründen: Nehmen wir an, daß die Differenz zwischen gewünschtem und tatsächlichem Ausgabewert beim Neuron i positiv $(d_i > x_i^{(1)})$, d.h. der tatsächliche Ausgabewert zu klein ist. Die Aktivierung des Neurons i ist daher zu erhöhen. Dies kann durch eine Vergrößerung bzw. Verkleinerung der Gewichtswerte $w_{ij}^{(1)}$ nach folgender Regel erreicht werden:

1. Ist $x_j^{(0)} < 0$, dann vergrößere $w_{ij}^{(1)}$.

2. Ist $x_j^{(0)} > 0$, dann verkleinere $w_{ij}^{(1)}$.

Diese Regel ist das Grundprinzip der sog. Backpropagation-Lernregel, die als Erweiterung der Delta-Regel für Netzwerke mit sog. verborgenen Neuronen (Neuronen die weder an der Eingabe- noch an der Ausgabeseite sind) gilt.

3.5.3 Back Propagation (BP)

Mit dieser Lernmethode wurde wieder verstärkt das Interesse an Neuronalen Netzwerken geweckt. Rumelhart, Hinten und Williams Rumelhart et al. [1986a] haben ein Verfahren entwickelt, mit dem auch Netzwerke in den verborgenen Schichten ihre Gewichte beim Lernen verändern können.

Netzwerke ohne verborgene Schicht können niemals nicht linear separable Probleme lösen (siehe Kapitel 3.4.1). Daher ist es bei der Lösung praktischer Probleme notwendig, Netzwerke mit weiteren Schichten zwischen der Eingabe- und der Ausgabeschicht zu verwenden. Minsky und Papert nahmen an, daß die Ergebnisse für alle Neuronalen Netze gelte und daher kein Neuronales Netzwerk linear separable Probleme lösen kann (siehe der APL-Workspace XOR im Anhang 8.2). Eine weitere Annahme war, daß es unmöglich sei, eine Lernmethode für Netzwerke mit verborgenen Neuronen zu entwickeln. Beide Annahmen stellten sich mit der Entwicklung der Backpropagation-Lernregel als falsch heraus. Netzwerke mit nichtlinearen Aktivierungsfunktionen und mit mindestens einer verborgenen Schicht von Neuronen können auch nicht linear separable Probleme lösen. Die Netzwerke mit zumindest einer verborgenen Schicht können mit der Backpropagation Lernregel trainiert werden.

Dieses Netzwerk ist eines der Standardnetzwerkarchitekturen, die in 80% der Projekte, in denen Neuronale Netzwerke verwendet werden, angewendet wird (vgl. [Caudill and Butler, 1992a, Seite 173]). Backpropagation Netzwerke sind in der Regel die erste Wahl des "Connection-Engineers", da sie einfach zu implementieren sind, viele Probleme korrekt lösen, und deren Verwendung nicht von der Problemstellung bestimmt ist.

Das Netzwerk ändert seine Gewichte der Verbindungen um die Differenz zwischen tatsächlichen und gewünschten Ausgabevektor zu minimieren. Im trainierten Netzwerk sind in den nicht zur Eingabe- oder Ausgabeschicht zählenden verborgenen Schichten wichtige Eigenschaften aus den Datensätzen gespeichert.

Bei der Back Propagation handelt es sich um eine leistungsfähige synaptische Lernregel, die in einem beliebig verbundenen Netzwerk eine für die bestimmte Problemstellung passende interne Struktur im Netzwerk entwickelt. Diese Regel wird vorallem bei Netzwerken ohne Rückkoppelungsverbindungen verwendet und zählt zur Gruppe der überwachenden Lernmethoden. Eine Pro-

blemstellung ist durch einen Zustandsvektor für die Ausgabeunits und einen Zustandsvektor für die Eingabeunits gegeben. Die im letzten Kapitel beschriebene Lernmethode eignet sich nur für Netzwerke mit direkt verbundenen Eingabeunits zu den Ausgabeunits, weil der Fehler unmittelbar den Units zugeordnet werden kann. Lernen wird aber bei Netzen mit verborgenen Schichten komplizierter, da die tatsächlichen und gewünschten Ausgabewerte nicht durch die Problemstellung festgelegt werden. Daher muß die Lernregel entscheiden, unter welchen Bedingungen die Units in den verborgenen Schichten aktiviert sein sollen, damit das gewünschte Ein- Ausgabeverhalten des Netzwerks erreicht wird.

Die Back Propagation Lernregel geht von einer bestimmten Netzwerkstruktur aus. Sie ist relativ einfach für Netzwerke mit in Schichten angeordneten Units. Es gibt eine Schicht von Eingabeunits, eine beliebige Anzahl von verborgenen Schichten (hidden layers) und eine Schicht mit Ausgabeunits.

Die Restriktion, nur linear separable Probleme lösen zu können wird durch zwei Maßnahmen aufgehoben:

- Verwendung nichtlinearer Aktivierungsfunktionen und
- Verwendung eines Netzwerks mit zumindest einer verborgenen Schicht

Die Eingabeschicht nimmt den Eingabevektor von der Umgebung auf und gibt ihn an die Zwischenschicht des Netzwerks weiter. Die Ausgabeschicht nimmt das Signal aus der Zwischenschicht auf und erzeugt das Antwortsignal auf den Eingabevektor. Die Neuronen in der Zwischenschicht fungieren als Detektor der Regularitäten (feature detector) in den Daten. In den Gewichtswerten der Neuronen in der Zwischenschicht wird der Informationsgehalt der Daten abgelegt.

Während des Netzwerktrainings erfolgt die Simulation in zwei wesentlichen Schritten. Zuerst wird ein Eingabevektor aus den Datensätzen ausgewählt und an die Neuronen in der Eingabeschicht angelegt. Die Aktivierungswerte werden schrittweise von Schicht zu Schicht berechnet, bis die Antwort an den Ausgabeneuronen des Netzwerks erzeugt wird.

Im zweiten Schritt wird der Ausgabevektor des Netzwerks mit dem gewünschten Ausgabevektor verglichen. Gibt es eine Abweichung zwischen diesen Werten, wird mit dem Fehler eine 'Schuldzuweisung' an jedes Neuron durch zurücksenden der Fehlersignale von der Ausgabeschicht an die verborgenen Schichten bis zu Eingabeschicht durchgeführt. Dabei werden die Gewichte so verändert, daß ein geringerer Fehler bei einer erneuten Präsentation des gleichen Eingabevektors entsteht (siehe Abbildung 3.15).

Ein Backpropagation-Netzwerk wird durch die Gewichte zwischen den Neuronen, den Transferfunktionen und der Anzahl der Neuronen und Schichten cha-

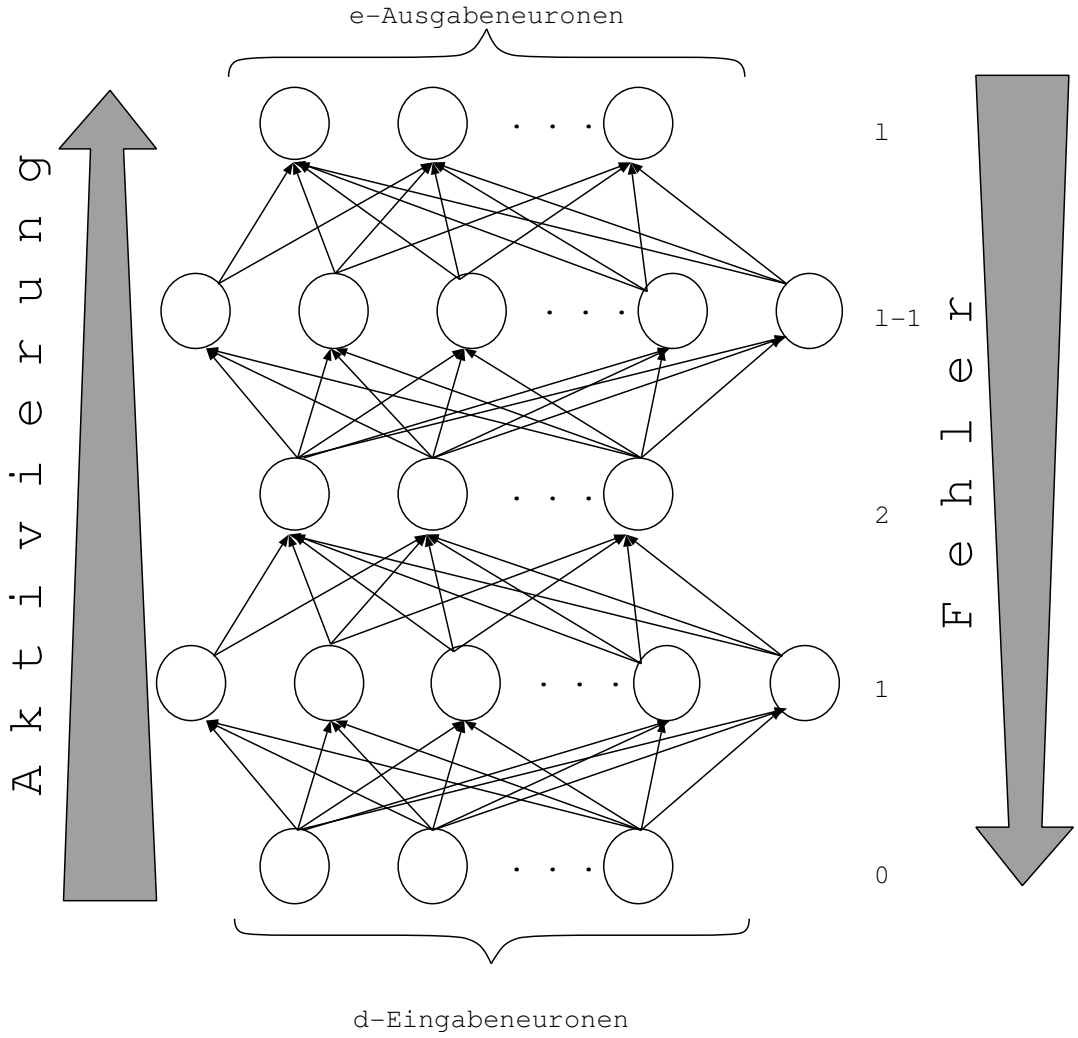

Abbildung 3.15: Ausbreitung der Aktivierung und des Fehlers in einem Backpropagation Netzwerk

rakterisiert. Typischerweise ist jedes Neuron mit jedem Neuron in der darunterliegenden Schicht verbunden, d.h es handelt sich um vollverbundene Schichten.

Die sog. "verallgemeinerte Delta-Regel" wird bei mehrschichtigen Netzwerken (Multilayered Perceptons MLP) angewendet.

Die Gewichtsänderung in der Schicht l wird mit

$$\Delta w_{ij}^{(1)} = \eta \delta_i^{(l)} x_j^{(l-1)}, \tag{3.24}$$

wobei $\eta \in (0,1)$ die konstante Lernrate, $x_j^{(l-1)}$ der über das Gewicht $w_{ij}^{(l)}$ gehende Ausgabewert von Neuron j in der darunterliegenden Schicht an das Neuron i ist.

1. Für die Ausgabeschicht (l = L) wird $\delta_i^{(L)}$ mit

$$\delta_i^{(L)} = (d_i - x_i^{(L)}) g'(u_i^{(L)}) \tag{3.25}$$

berechnet, wobei $g'(u_i^{(L)})$ die Steigung der Ausgabefunktion bei $u_i^{(L)}$ ist. Die obige Formel kann mit der Abbildung 3.16 veranschaulicht werden.

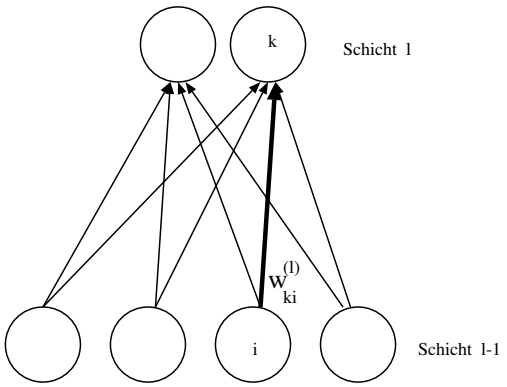

Abbildung 3.16: Veranschaulichung der allgemeinen Delta-Regel

Es wird eine Ausgabefunktion gewählt, die monoton steigend ist. Daher ist die Ableitung dieser Funktion immer positiv. Wenn der Ausgabewert $x_k^{(l)}$ des Neurons i in Schicht l zu klein ist ergibt sich ein $\delta_k^{(l)} < 0$. Der Ausgabewert des Neurons kann daher durch Erhöhung des Nettoinputs $u_k^{(l)}$ beseitigt werden, indem nach folgender Regel die Gewichtswerte verändert werden:

Wenn $x_i^{(l-1)} > 0$, dann vergrößere $w_{ki}^{(l)}$,

wenn $x_i^{(l-1)} < 0$, dann verkleinere $w_{ki}^{(l)}$.

2. Für alle unter der Ausgabeschicht liegenden Neuronen (für $l < L$) ist $\delta_i^{(l)}$ rekursiv definiert:

$$\delta_i^{(l)} = g'(u_i^{(l)}) \sum_{k=1}^{n^{(l+1)}} \delta_k^{(l+1)} w_{ki}^{(l+1)} \tag{3.26}$$

Die Gewichte werden mit

$$w_{ij}^{neu} = w_{ij} + \Delta w_{ij} \tag{3.27}$$

verändert.

Im Kapitel 4 wird das Mehrschichtenmodell von seiner implementationstechnischen Realisierung vorgestellt. Dabei wird auf notwendige Erweiterungen des bisher vorgestellten Modells eingegangen.

Kapitel 4

Lernen in Neuronalen Netzwerken mit Backpropagation

Das im vorigen Kapitel vorgestellte Modell der Backpropagation wurde als Modell zur Untersuchung der einiger Problemstellungen herangezogen. Hier soll auf Abänderungen des Lernverfahrens eingegangen werden.

4.1 Variationen der Backpropagation

4.1.1 Momentum-Term

Damit wird die Regel, mit der die Gewichte des Netzwerks verändert werden abgeändert. Es wird die letzte Gewichtsänderung berücksichtigt. Damit erreicht man eine schnellere Konvergenz des Netzwerks beim Lernen.

Die Gewichtsänderung in der Schicht l wird mit

$$\Delta w_{ij}^{(l)}(t) = \eta \delta_i^{(l)} x_j^{(l-1)} + \mu w_{ij}^{(l)}(t-1), \tag{4.1}$$

berechnet.

Die Idee stammt vom konjungierten Gradientenverfahren, bei dem die letzten Beobachtungen berücksichtigt werden, um eine Vorstellung über den Lösungsraum zu bekommen.

4.1.2 Gewichtsreduktion (weight decay)

Ein Netzwerk, das die Backpropagation Lernregel verwendet, minimiert den Fehler zwischen tatsächlichem und gewünschtem Verhalten des Netzwerks. Da beim Lernen mit der Ableitung der Aktivierungsfunktion die Gewichtswerte

geändert werden, kann es zur einem unerwünschten "hängenbleiben" des Lernvorganges beim Trainieren des Netzwerks kommen.

Eine einfache Abhilfe ist eine geringfügige Herabsetzung der Gewichtswerte durch

$$w_{ij}^{neu} = 0.999 w_{ij} \tag{4.2}$$

Multiplikation mit einer Konstanten kleiner eins.

4.1.3 Aktivierungsfunktionen

Die Aktivierungsfunktion dient zur Bewertung der vom Neuron empfangenen Signale. Dafür ist nicht jede Funktion geeignet. Bei linearen Aktivierungsfunktionen können Netze mit verborgenen Schichten durch ein äquivalentes Netzwerk ohne Zwischenschicht dargestellt werden. Für Netze ohne Zwischenschicht gelten die im Kapitel 3.4.1 dargestellten Limitationen. Es ist darauf zu achten, daß die verwendete Funktion nichtlinear ist.

Die Gruppe der sigmoiden Funktionen(siehe Definition 3.2.8, Seite 13) ist hier von besonderer Bedeutung. Die Wahl einer bestimmten sigmoiden Aktivierungsfunktion hat nur marginalen Einfluß auf die Eingenschaften des Netzwerks.

Die logistische Funktion

Diese Funktion wird am häufigsten verwendet. Sie ist definiert als

$expsigm : \mathbb{R} \rightarrow [0, 1]$

$$expsigm(x, T) = \frac{1}{1 + e^{-\frac{2x}{T}}} \tag{4.3}$$

wobei $T \, \epsilon \,]0, \infty]$ ein Temperaturwert ist. Strebt T gegen null, wird die Steigung der Flanke größer und nähert sich an die an die binäre Schwellwertfunktion an. Für T gegen ∞ wird die Steigung der Flanke geringer und nähert sich der Identitätsfunktion an. Die Abbildung 4.1 zeigt den Funktionsverlauf, wenn der Wert T variiert wird.

Die Abbildung 4.2 zeigt den Funktionsverlauf und deren erste Ableitung. Die Ableitung der Aktivierungsfunktion wird im Lernalgorithmus verwendet und hat daher große Bedeutung.

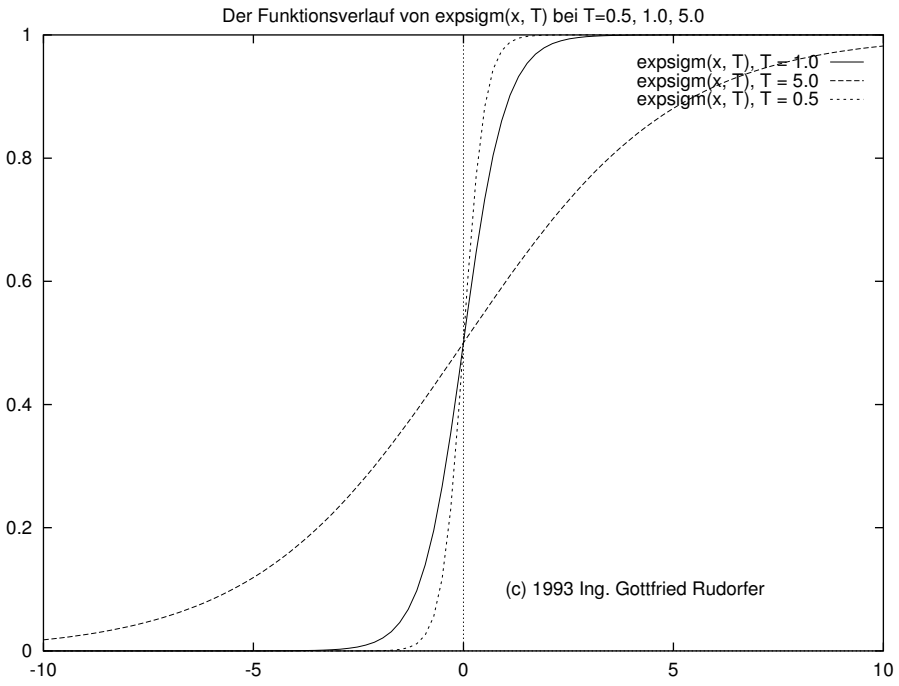

Abbildung 4.1: Funktionsverlauf der logistischen Funktion bei Variation der Temperatur T

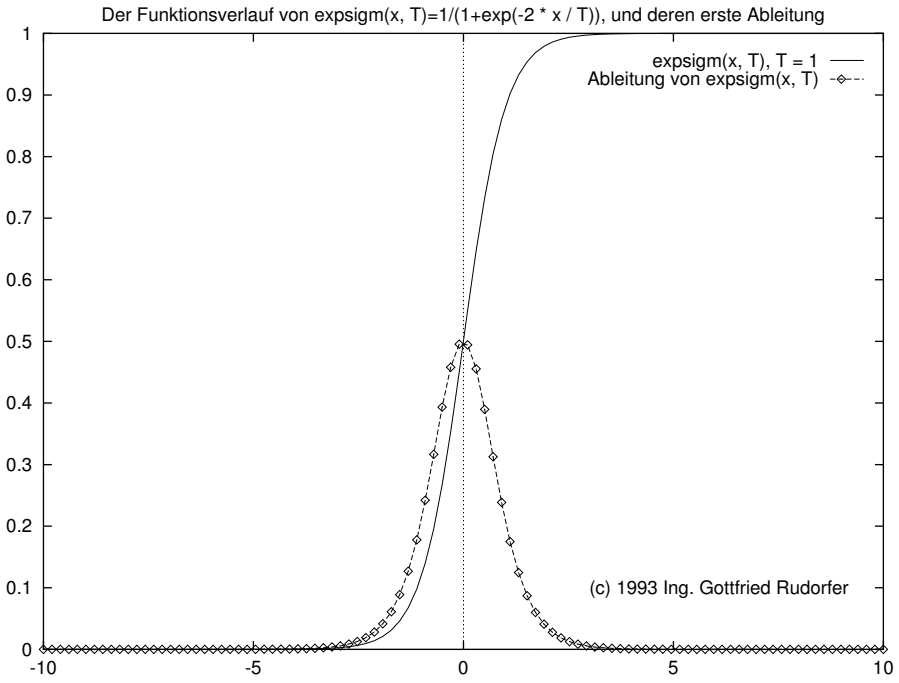

Abbildung 4.2: Der Funktionsverlauf der logistischen Funktion

Die Tangens-Hyperbolicus Funktion

Die Tangens-Hyperbolicus Funktion definiert als

$tanh : \mathbb{R} \rightarrow [-1, 1]$

$$tanh(x, T) = \frac{1 - e^{-\frac{2x}{T}}}{1 + e^{-\frac{2x}{T}}} \tag{4.4}$$

wobei $T \, \epsilon \,]0, \infty]$ der oben erwähnte Temperaturwert ist. Die Form dieser Funktion verändert sich ähnlich bei verschiedenen Temperaturwerten, wie bei der logistischen Funktion.

Die Abbildung 4.3 zeigt den Funktionsverlauf und deren erste Ableitung.

Aus der Darstellung beider Funktionen erkennt man, daß die maximale Steigung der Funktion um den Nullpukt liegt. Der Temperaturwert kann während dem Lernen von einem großen Wert ausgehend verringert werden. Damit erreicht man ein 'Abkühlen' des Netzwerks während dem Training.

Die Identitätsfunktion scheidet aufgrund der Linearität, die Sprungfunktion aufgrund der Unstätigkeitsstelle (an dieser Stelle nicht differenzierbar) für das Backpropagation-Lernverfahren aus.

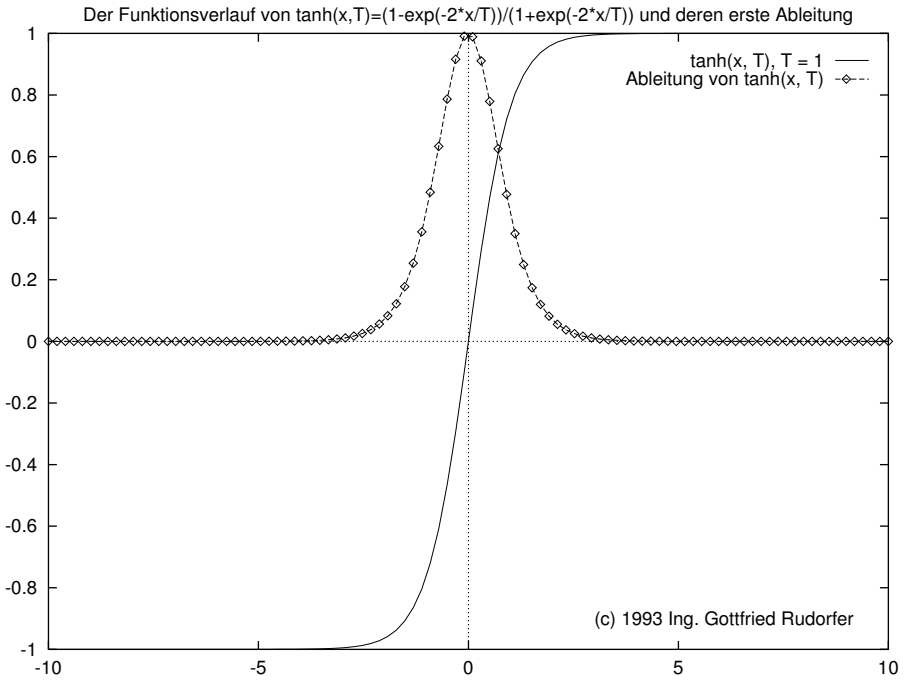

Abbildung 4.3: Funktionsverlauf der Tangens-Hyperbolicus Funktion

4.1.4 Transformation des Schwellwertes in einen Bias-Knoten

Bei der Implementation des Netzwerks ist es sinnvoll den Schwellwert in die Summe der Funktion 3.21(Seite 36) aufzunehmen.

$$u_i^{(l)} = \sum_{j=0}^{n^{(l)}} w_{ij}^{(l)} x_j^{(l-1)} \tag{4.5}$$

wobei $w_{i0} = -\Theta_i$ und $x_0 = 1$ gesetzt wird.

Das Netzwerk für das XOR-Problem (siehe Abbildung 6.1, Seite 61) wird in ein äquivalentes Netzwerk mit Bias-Knoten umgewandelt (siehe Abbildung 4.4).

4.1.5 Rekurrente Backpropagation

Rekurrente Netzwerke sind besonders bei Lernen von Mustersequenzen erfolgreich. In einem derartigen Netzwerk durchläuft ein Trainingssatz mehrmals das Netzwerk. Erst nach mehreren Schritten wird ein Ausgabewert erzeugt. (vgl. [Caudill and Butler, 1992b, Seite 79]).

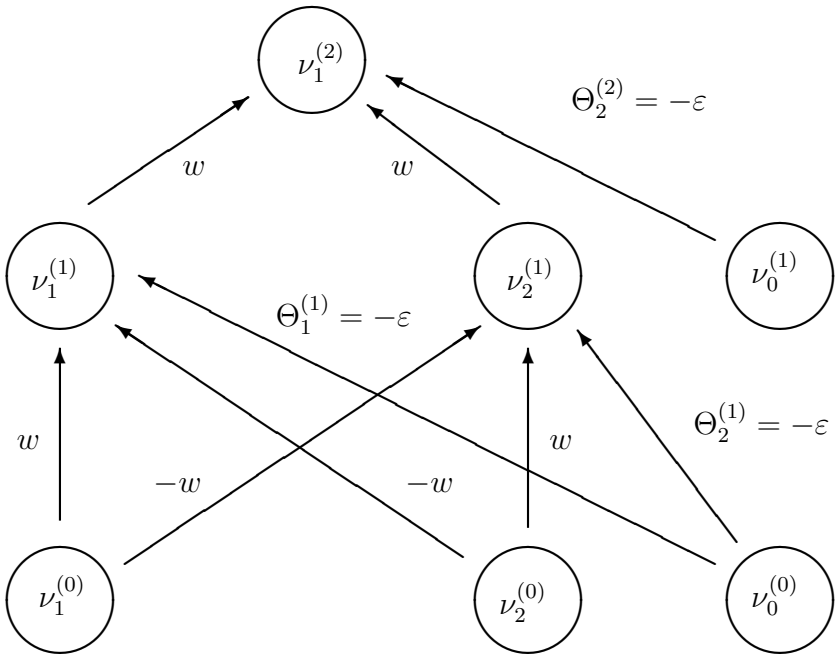

Abbildung 4.4: Ein äquivalentes XOR-Netzwerk mit Bias-Knoten

Ein Trainingssatz wird durch das Netzwerk geschickt und deren Ausgabewerte werden wieder in das Netzwerk zurückgeführt, bis das Netzwerk ein stabiles Verhalten erzeugt.

Hier geht es um eine rekurrente Erweiterung eines Backpropagation-Netzwerks. Die Abbildung 4.5 zeigt ein kleines rekurrentes Backpropagation Netzwerk.

Eine Untersuchung der Eigenschaften dieses Ansatzes würde den Rahmen dieser Arbeit überschreiten.

4.2 Vor- und Nachteile der Backpropagation

Ein "Multilayer Netzwerk" kann jede beliebige Funktion berechnen (vgl. [Knight, 1990, S 66]). Bei gegebenen Ein- Ausgabemuster erlauben es die Summations- und Schwellwertunits einfache UND, ODER und NICHT Logikelemente zu realisieren. Es kann auch jede beliebige Kombination aus diesen Grundbausteinen realisiert werden. Das Netzwerk ist aber nicht auf binäre Ein- Ausgabemuster beschränkt. Jede beliebige Funktion zu realisieren kann als ein Vorteil dieses Netzwerktyps bezeichnet werden.

Die Nachteile liegen aber beim Lernen. Die Back Propagation ist ein Gradientenverfahren, das zur langsamsten Methode für konvexe Optimierungsproble-

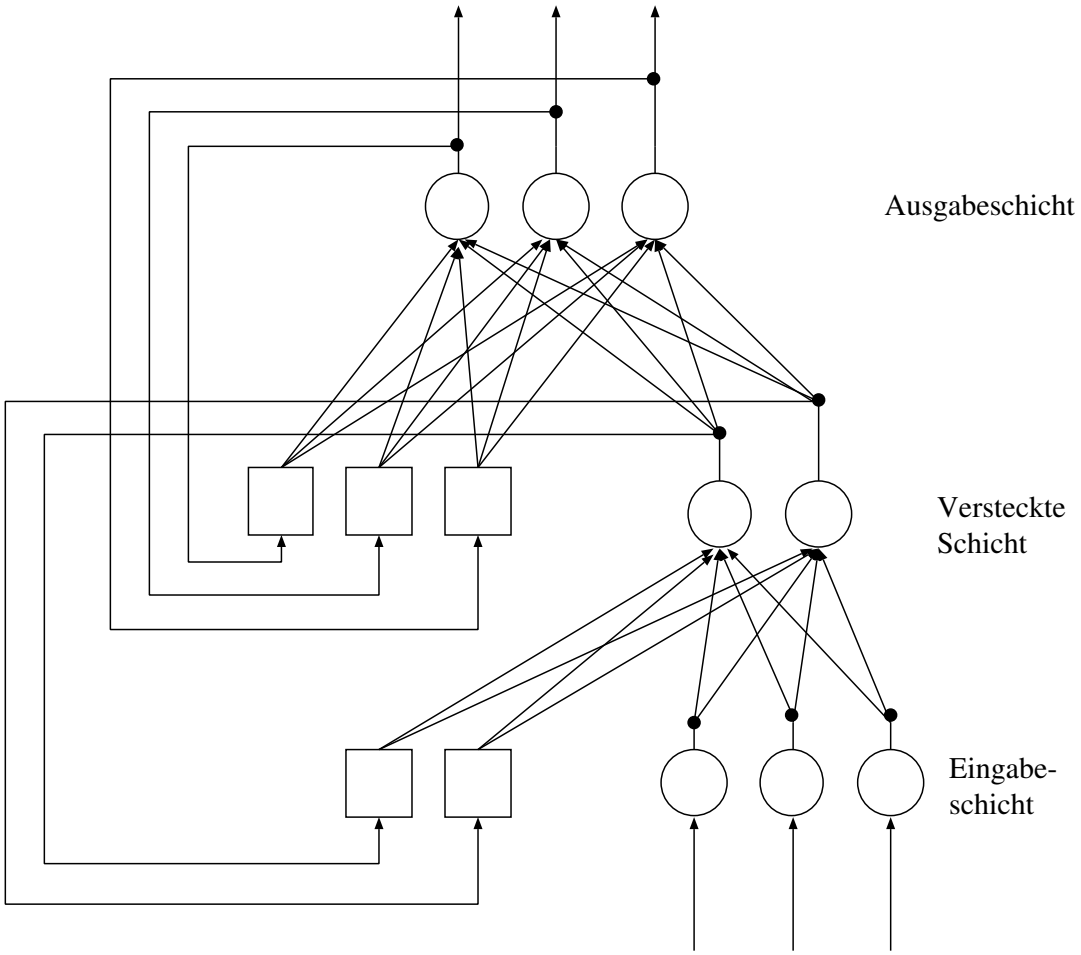

Abbildung 4.5: Ein rekurrentes Netzwerk

me zählt. Viele Probleme sind aber nicht konvex. Daher besteht die Gefahr, daß das Netzwerk in einem lokalen Optimum hängen bleibt. Glücklicherweise ist die Backpropagation aber kein reines Gradientenverfahren, da die Lernrate (Schrittweite) immer konstant bleibt.

Weil die Bewegungsrichtung auf der Fehlerhyperfläche nicht der Abstiegsrichtung des Gradientenverfahrens entspricht, besteht doch Hoffnung in das globale Optimum zu gelangen. Man nimmt an, daß es nicht viele lokale Optima gibt. Liegt ein lokales Optimum vor (der Fehler bleibt auf einem hohen Niveau), könnte das Backpropagation Lernverfahren z.B. durch Addition von Zufallszahlen zur Gewichtsänderung oder der Kopplung mit einem Genetischen Algorithmus modifiziert werden.

Kapitel 5

Lösungserarbeitung mit einem Neuronalen Netzwerk

Nicht jedes Problem kann effizient durch ein Neuronales Netzwerk gelöst werden. Daher ist es notwendig ein gegebenes Problem auf deren Eignung für eine Netzwerklösung zu untersuchen.

Nach Prem (vgl. [Prem, 1993, Seite 5f]) unterteilt sich die Erarbeitung einer Netzwerk-Lösung in folgende Schritte:

- **Problemanalyse**

  1. *Ist das Problem mit Regeln lösbar?*

     Wenn ausreichend und klar strukturiertes Wissen über das Problem vorhanden ist, besteht kein Grund, das Problem nur mit einem Neuronalen Netzwerk zu lösen. Vielmehr sollte eine Auswahl der relevanten Einflußfaktoren erfolgen. Besonders wichtig sind Informationen über Korrelationen in den Daten.

  2. *Muß das Ergebnis 'exakt' oder 'erklärbar' sein?*

     Ein Neuronales Netzwerk lernt so lange, bis die Anpassung des Netzwerks an die gewünschte Lösung für den Anwender ausreichend genau ist. Daher kann ein Netzwerk keine exakten Lösungen, sondern nur eine Approximation an die gegebenen Daten liefern.

     Meist kann das Verhalten des Netzwerks nicht oder nur mit hohem Aufwand erklärt werden. Die Frage nach dem 'wie' das Netzwerk zu diesem Ergebnis kommt kann nicht aus den Gewichtswerten des Netzwerks abgelesen werden.

     Daher scheiden Probleme, bei denen ein exaktes und/oder erklärbares Ergebnis gefordert wird aus.

  3. *Steckt die relevante Information in den Trainingsdaten?*

Diese Frage muß sich der "Connection Engineer" bei einer Lösung des Problems mit einem Neuronalen Netzwerk stellen. Die Qualität der vorhandenen Daten muß nach den folgenden Kriterien untersucht werden:

- Da ein Netzwerk anhand von Trainingsbeispielen lernt, hängt die Generalisierungsfähigkeit des Netzwerks von der Anzahl der verfügbaren Trainingsbeispiele ab.
- Bei der Einteilung der Trainingsdaten in Klassen ist zu fragen, ob genügend Daten für jede Klasse vorhanden sind.
- Abschließend muß noch die Frage geklärt werden, ob überhaupt die relevanten Informationen in den Trainingsdaten enthalten sind. Ein Neuronales Netzwerk kann zwar die relevanten Daten herausfinden, zeigt aber umgekehrt nicht die fehlenden Variable oder Meßwerte an.

4. *Wie groß ist der benötigte Eingabevektor?*

   Die Anzahl der zu berücksichtigenden Daten muß beim Entwurf einer Netzwerklösung bekannt sein, damit die Größe des verwendeten Netzwerks abgeschätzt werden kann. Ein zu großes Netzwerk hat negative Eigenschaften auf die Generalisierungsfähigkeit des Netzwerks.

- **Die Erstellung einer Lösung** Das "connection engineering"(das Erstellen einer Problemlösung mit einem Neuronalen Netzwerk) kann in folgenden Schritten ablaufen:

  1. *Datensammlung- und aufbereitung*

     Die Trainingsdaten müssen in ausreichender Menge vorhanden sein und in eine für das Netzwerk geeigneten Form konvertiert werden. Dieser Schritt wird auch als Kodierung der Daten bezeichnet. Diese Aufgabe ist sehr sorgfältig wahrzunehmen, da alle weiteren Schritte von den kodierten Daten abhängen. Eine erfolgreiche Lösung des Problems hängt daher zu einem Großteil von der Kodierung der Daten ab.

  2. *Netzwerkdesign*

     Nun ist das für die Problemstellung geeignete Netzwerk aus der fülle vorhandener Netzwerke auszuwählen. Einen groben Überblick der Netzwerke nach ihren Charakteristika gibt die Abbildung 5.1

  3. *Implementierung von Prototypen*

     Einfache Prototypen können mit Simulationsprogramme trainiert und getestet werden. Meist ist es aber unerläßlich, das trainierte Netzwerk mit konventioneller Programmierung zu realisieren, da ein Netzwerk in die Anwendung eingebunden werden muß (Ein-

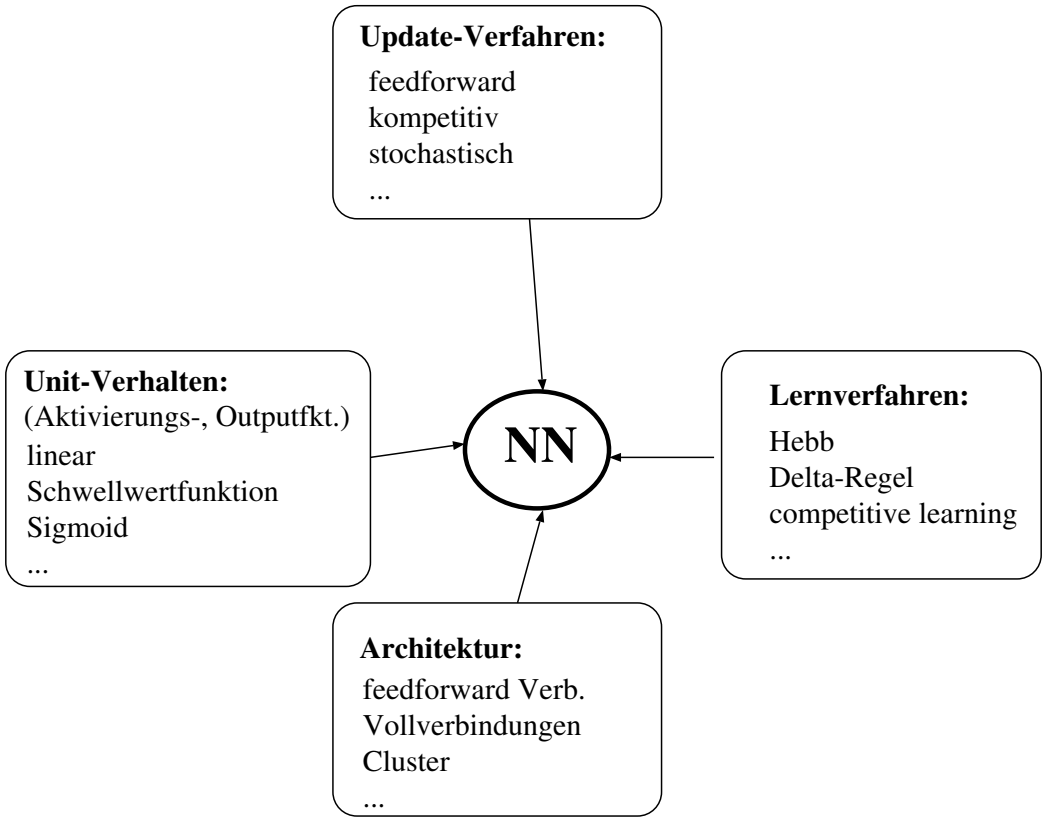

Abbildung 5.1: Zu berücksichtigende Netzwerkcharakteristika beim Netzwerkdesign

und Ausgabeschnittstelle). Dadurch ist es auch leicht möglich neue Netzwerkparadigmen in die Problemlösung einzubinden, da die Simulatoren meist nur Netzwerkgrundtypen enthalten.

4. *Auswahl des Netzwerks mit der passenden Struktur*

   Das beste Netzwerk wird meist durch Experimentieren mit verschiedenen Netzwerkarten und der Variation der Parameter (Anzahl der Schichten im Netzwerk, Anzahl der Neuronen pro Schicht, Lernrate, Anfangsinitialisierung, ...) eines Netzwerks gesucht. Hier ist es wichtig entsprechend leistungsfähige Lernalgorithmen und eventuell spezielle Hardware für Neuronale Netze zu verwenden. Der Versuch den Lernprozeß durch Spezialhardware zu verkürzen hat auch Dan Hammerstromm Hammerstrom [1993] hervorgehoben und wird in den nächsten Jahren an Bedeutung gewinnen.

Kapitel 6

Beispiele für implementierte Anwendungen

6.1 Traditionelle Anwendungsbeispiele

6.1.1 Das XOR-Problem

Hier soll das oft in der Literatur gefundene Beispiel - das XOR-Problem erwähnt werden.

Der erste Prototyp wurde in Dyalog APL (Version HP:6.1.3:X) implementiert (siehe Anhang 8.2, Seite 96). Zur Ermittlung der Auswirkung der Parametervariation (Momentum und Lernrate) war aber wegen der Laufzeiten der Programme eine Implementierung in der Programmiersprache C (siehe Anhang 8.5, Seite 126) notwendig.

Der im Anhang gezeigte Programmcode implementiert ein Backpropagation-Netzwerk, das die sog. XOR-Beziehung lernt. Das Netzwerk soll die in Tabelle 6.1 dargestellte Eingabe- Ausgabebeziehung lernen.

Nach erfolgreichem Training kann das Netzwerk die in der Abbildung 6.1 dargestellten Gewichtswerte annehmen. Es existieren noch weitere Lösungen.

$x_1^{(0)}$	$x_2^{(0)}$	$x_1^{(L)}$
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 6.1: Wahrheitstafel einer XOR-Logikschaltung

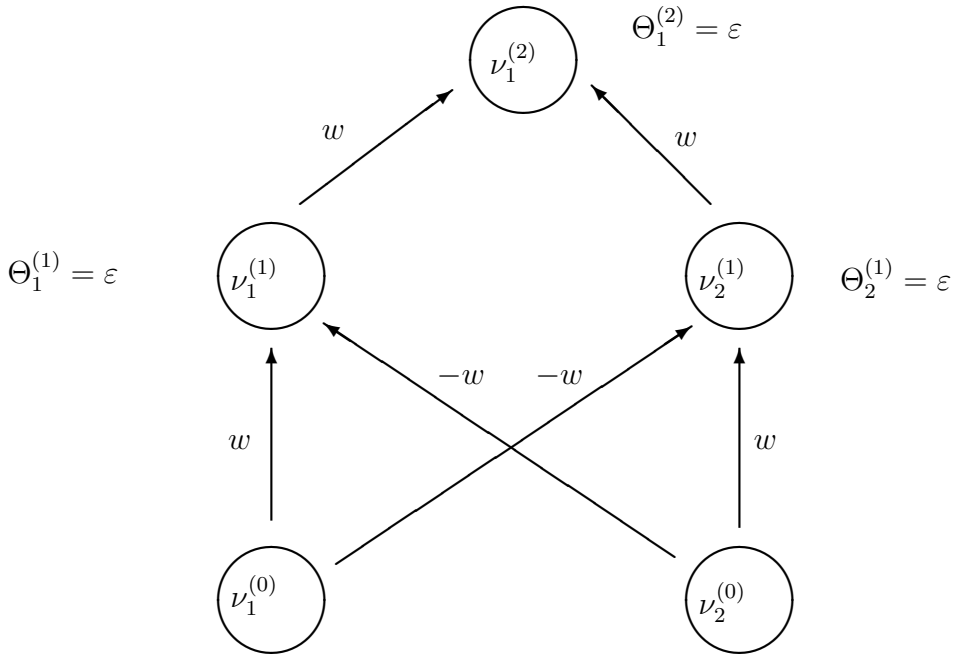

Abbildung 6.1: Ein Neuronales Netzwerk lernt die XOR-Funktion

Das n-te Neuron in der l-ten Schicht wird als $\nu_n^{(l)}$ bezeichnet. Die Schwellwerte der Neuronen nehmen den konstanten Wert ε an. Die Aktivierungsfunktion ist die Identitätsfunktion: $a_i^{(l)} = u_i^{(l)}$

Die Ausgabefunktion ist als

$$g(a_i^{(l)}) = \begin{cases} 1 & \text{wenn } a_i^{(l)} > 0 \\ 0 & \text{sonst} \end{cases} \tag{6.1}$$

definiert.

Wählen wir z.B. den Trainingssatz mit $x_1^0 = 1$, $x_2^0 = 0$, $x_1^L = 1$ und für w eine positive reelle Zahl, die größer als ε ist, kann die Arbeitsweise des Netzwerks folgendermaßen dargestellt werden:

Für das Neuron $\nu_1^{(1)}$ ergibt sich folgender Nettoinput (siehe Formel 3.21, Seite 36):

$$u_1^{(1)} = w_{11}^{(1)} x_1^{(0)} + w_{12}^{(1)} x_2^{(0)} - \Theta_1^{(1)} = w \times 1 + (-w) \times 0 - \varepsilon = w - \varepsilon \tag{6.2}$$

Die Aktivierung des Neurons ist

$$a_1^{(1)} = u_1^{(1)} = w - \varepsilon \tag{6.3}$$

Der Ausgabewert des Neurons $\nu_1^{(1)}$ ist

$$x_1^{(1)} = g(a_1^{(1)}) = g(w - \varepsilon) = 1 \tag{6.4}$$

Für das Neuron $\nu_2^{(1)}$:

$$u_2^{(1)} = w_{21}^{(1)} x_1^{(0)} + w_{22}^{(1)} x_2^{(0)} - \Theta_2^{(1)} = (-w) \times 1 + w \times 0 - \varepsilon = -w - \varepsilon \tag{6.5}$$

$$a_2^{(1)} = u_2^{(1)} = -w - \varepsilon \tag{6.6}$$

$$x_2^{(1)} = g(a_2^{(1)}) = g(-w - \varepsilon) = 0 \tag{6.7}$$

In der Ausgabeschicht ergibt sich folgender Nettoinput von der Hiddenschicht:

$$u_1^{(2)} = w_{11}^{(1)} x_1^{(0)} + w_{12}^{(1)} x_2^{(0)} - \Theta_1^{(2)} = w \times 1 + w \times 0 - \varepsilon = w - \varepsilon \tag{6.8}$$

Die Aktivierung des Neurons ist

$$a_1^{(2)} = u_1^{(2)} = w - \varepsilon \tag{6.9}$$

Der Ausgabewert des Neurons $\nu_1^{(2)}$ ist

$$x_1^{(2)} = g(a_1^{(2)}) = g(w - \varepsilon) = 1 \tag{6.10}$$

Der tatsächliche Ausgabewert, ist gleich dem gewünschten Ausgabewert.

Analog kann man für die drei restlichen Ein- Ausgabepaare zeigen, daß das Netzwerk das gewünschte Verhalten erzeugt.

Die Aufgabe des Backpropagation-Algorithmus ist es, diese Gewichts- und Schwellwerte beginnend bei einer zufälligen Gewichtsinitialisierung herauszufinden.

Der Backpropagation Algorithmus kann durch folgenden Pseudocode dargestellt werden:

Der Backpropagation Algorithmus

```
1  setze maximal zulaessigen Fehler /* Benutzerdefiniert */
2  setze maximal zulaessige Iterationszahl
3    /* Benutzerdefiniert */
```

```
 4  setze Anzahl der Schichten
 5    /* Benutzerdefiniert */
 6  setze Anzahl der Neuronen in jeder Schicht
 7    /* Benutzerdefiniert */
 8  setze Lernrate  /* Benutzerdefiniert */
 9  setze Iterationsintervall indem der Netzwerkzustand
10    gesichert wird /* Benutzerdefiniert */
11
12  initialisiere die Gewichte und Biaswerte mit Zufallszahlen
13
14  while-Schleife( Iterationszahl < max. zulaessige Iterationszahl)
15  {
16    Gesamtfehler = 0;
17    erzeuge zufaellige Indizes, nach denen die
18      Trainingsdatensaetze ausgewaehlt werden;
19    for-Schleife( ueber jedes Muster im Trainingssatz )
20    {
21      /* Vorwaertsrechnung */
22      hole den naechsten Trainingsdatensatz;
23      Trainingsdatensatz an das Netzwerk anlegen;
24
25      for-Schleife( ueber alle Schichten des Netzwerks )
26      {
27        Trainingsdatensatz an das Netzwerk anlegen
28
29        for-Schleife( ueber jedes Neuron in
30          dieser Schicht )
31        {
32          berechne den Nettoinput;
33          berechne Aktivierung des Neurons;
34        }/* Ende for Neuron */
35
36        if ( Iterationsintervall erreicht )
37        {
38          sichere das Netzwerk in eine Datei;
39        }
40
41      }/* Ende for ueber alle Schichten */
42
43    /* Rueckwaertsrechnung (nur beim Trainieren des Netzwerks) */
44
45    /* Berechne den Fehler in jedem Neuron in der Ausgabeschicht */
46      for-Schleife( ueber alle Neuronen der Ausgabeschicht )
```

```
47      {
48        Fehler = gewuenschter Ausgabewert -
49          tatsaechlicher Ausgabewert;
50        Gesamtfehler = Gesamtfehler + Fehler * Fehler;
51        /* Der Fehler kann auch anders berechnet werden */
52        /* Berechne delta nach Formel 3.25 */
53        delta = Fehler *
54          Ableitung der Aktivierungsfunktion;
55
56      }/* Ende for Neuronen in der Ausgabeschicht */
57      Gesamtfehler = Quadratwurzel(Gesamtfehler);
58
59      /*
60        Rueckwaertsrechnung (Backpropagation) des Fehlers
61        an die Eingabeschicht i
62      */
63      for-Schleife( ueber alle Schichten unterhalb
64        der Ausgabeschicht des Netzwerks )
65      {
66        /* Berechne delta fuer darunterliegende
67        Schichten nach Formel 3.26 */
68        delta = Ableitung der Aktivierungsfunktion *
69          Summe der gewichteten Fehler
70          der darueberliegenden Schicht;
71      }
72
73      /* Berechne neue Gewichtswerte */
74      for-Schleife( ueber alle Schichten des Netzwerks )
75      {
76        Gewichtsaenderung = Lernrate * delta *
77          Empfangene Aktivierung
78          aus der darunterliegenden Schicht;
79        Gewicht = Gewicht + Gewichtsaenderung;
80        Alte Gewichtsaenderung = Gewichtsaenderung;
81      }
82    } /* Ende for ueber jedes Muster im Trainingssatz */
83  } /* Ende while-Schleife I
```

Simulationsergebnisse

Bei einem einfachen Netzwerk kann deren Lernverhalten noch mitverfolgt werden. Es hat vier Trainingsmuster zu lernen indem es neun Gewichtswerte verändert. Daher wird dieses Beispiel zur Veranschaulichung des Lernens herangezogen.

Mit großem Simulationsaufwand kann der Verlauf des Fehlers in Abhängigkeit von der gewählten Lernrate einerseits und des gewählten Momentums andererseits dargestellt werden.

Die folgenden Grafiken zeigen den gemittelten absoluten Fehler (gewünschter – tatsächlicher Ausgabewert) nach der Präsentation aller Trainingsmuster, die das Netzwerk nach n-Lernschritten macht.

Die Simulationen wurden mit einem in der Programmiersprache C geschriebenen Progamm (siehe Anhang 8.5, Seite 126) erzielt und mit dem Programm Gnuplot Williams and Kelley [1993] grafisch dargestellt.

Als Ausgabefunktion wurde die *expsigm*-Funktion gewählt. Das Netzwerk wurde nach jeder Simulation mit vorher festgelegten Zufallszahlen zwischen -0.1 und 0.1 initialisiert. Mit Iteration ist folgender Ablauf im Programm gemeint: Es wird zufällig ein Trainingsmuster ausgewählt, der tatsächliche Ausgabewert des Netzwerks und der Fehler für das Trainingsmuster berechnet und die Gewichte nach dem Backpropagation- Prinzip korrigiert (Forward- und Backwardpass).

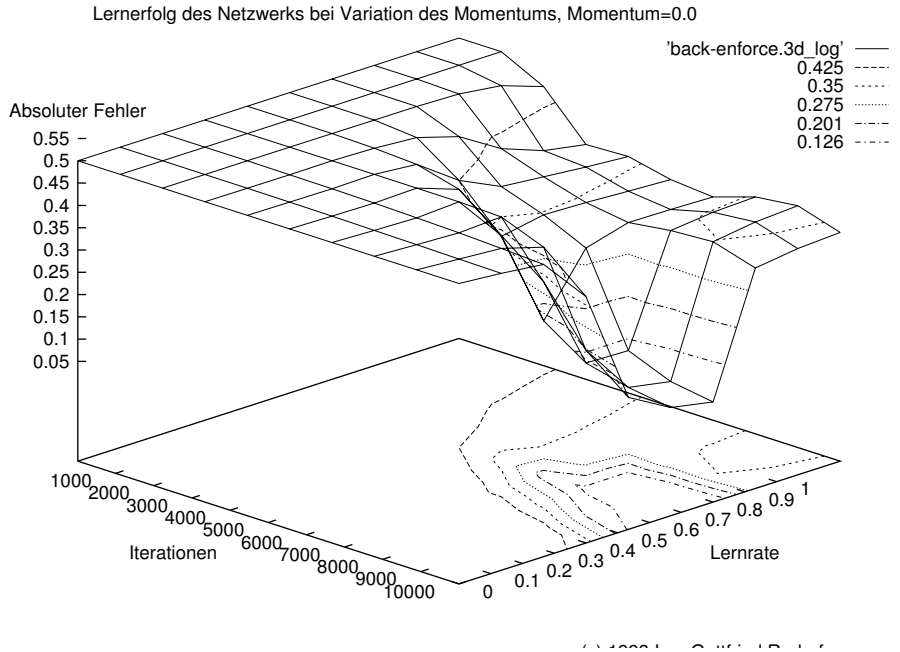

Abbildung 6.2: Lernerfolg des Netzwerks bei Variation der Lernrate ohne Berücksichtigung vergangener Gewichtsänderungen

Die erste Grafik 6.2 zeigt den Erfolg des Netzwerks bei verschiedenen Lernraten ohne Berücksichtigung vergangener Gewichtsänderungen(Momentum=0).

Eine Lernrate von 0.6 bis 0.7 führt am schnellsten zur Reduktion des gemittelten absoluten Fehlers des Netzwerks.

Bei einer zu geringen Lernrate (z.B. 0.1) ändert sich am gewichteten absoluten Fehler selbst nach 10,000-Iterationen kaum etwas. Dabei wurde jedes Trainingsmuster schon im Durchschnitt 2,500-mal präsentiert.

Wählt man die Lernrate zu hoch (z.B. 0.95), verringert sich der Fehler zwar rasch, jedoch kommt der Lernalgorithmus nicht unter einen Fehler von ca. 0.35. Warum dies so ist zeigen die Grafiken "Gewichtsänderung bei falscher Wahl der Lernrate und Momentum".

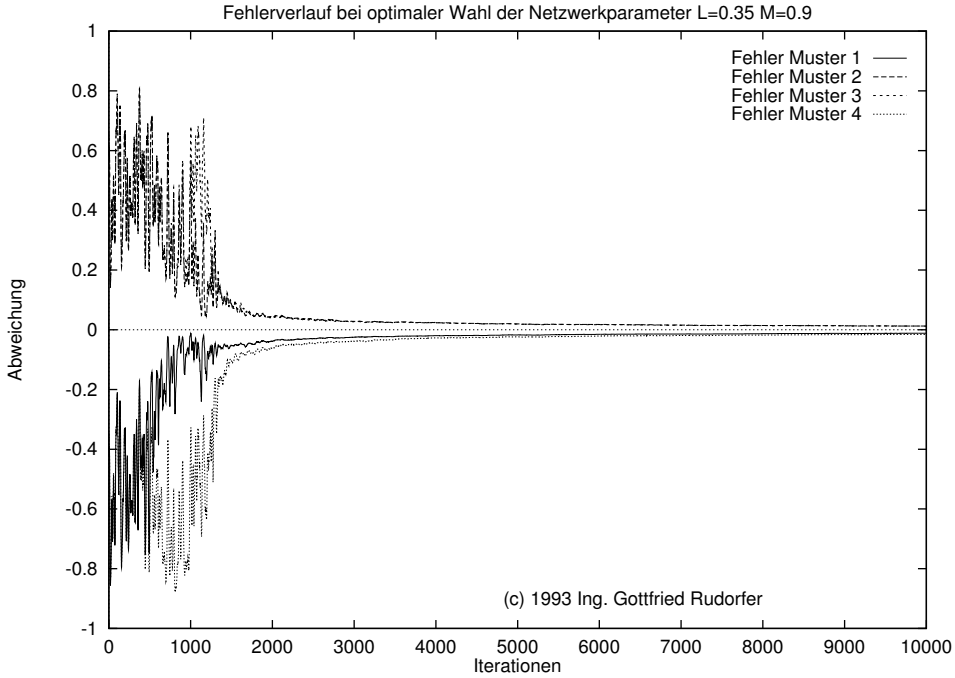

Abbildung 6.3: Fehlerverlauf für jedes Trainingsmuster beim erfolgreichen Lernen

Die Abbildung 6.3 zeigt den Verlauf des Netzwerkfehlers beim erfolgreichen Lernen. Der Fehler schwankt anfangs stark, die Varianz und Amplitude nehmen aber rasch ab. Nach ca. 3,000 Iterationen geht er schließlich gegen null.

Zu diesem Fehlerverlauf kann noch die Entwicklung der Gewichtswerte gezeigt werden. Die Abbildung 6.4 zeigt die Gewichtsveränderung in der Ausgabeschicht und die Abbildung 6.5 die Änderung in der Hidden-Schicht.

Die Gewichtswerte ändern sich beginnend von ihren Anfangswerten zwischen -0.1 und 0.1 relativ stark und konvergieren nach ca. 3,000 Iterationen zu stabilen Werten, die sich kaum bei weiterem Lernen des Netzwerks ändern.

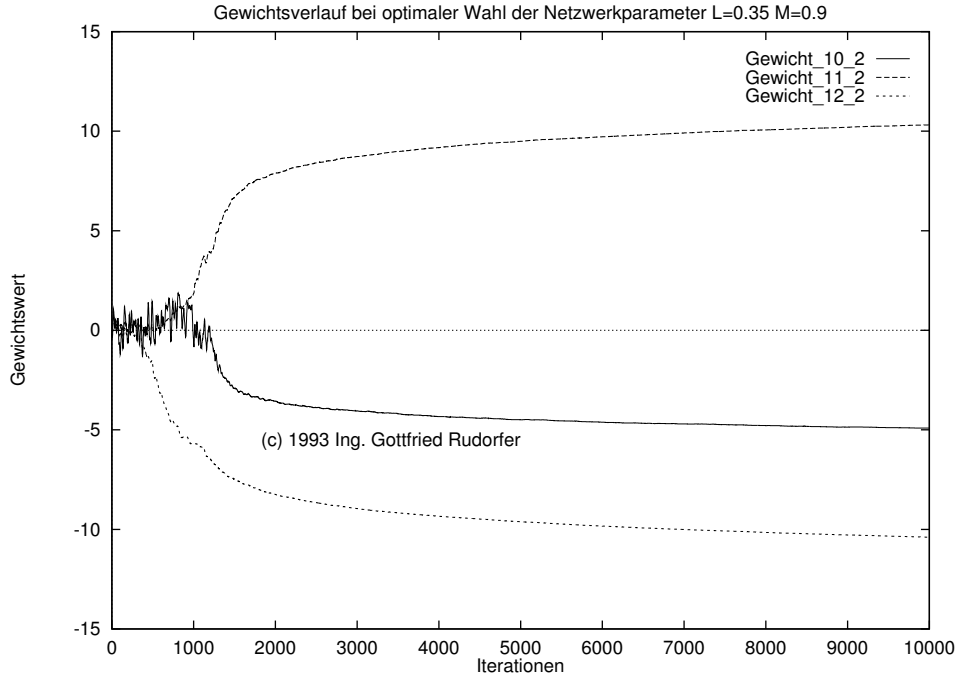

Abbildung 6.4: Gewichtsverlauf in der Hidden-Ausgabeschicht

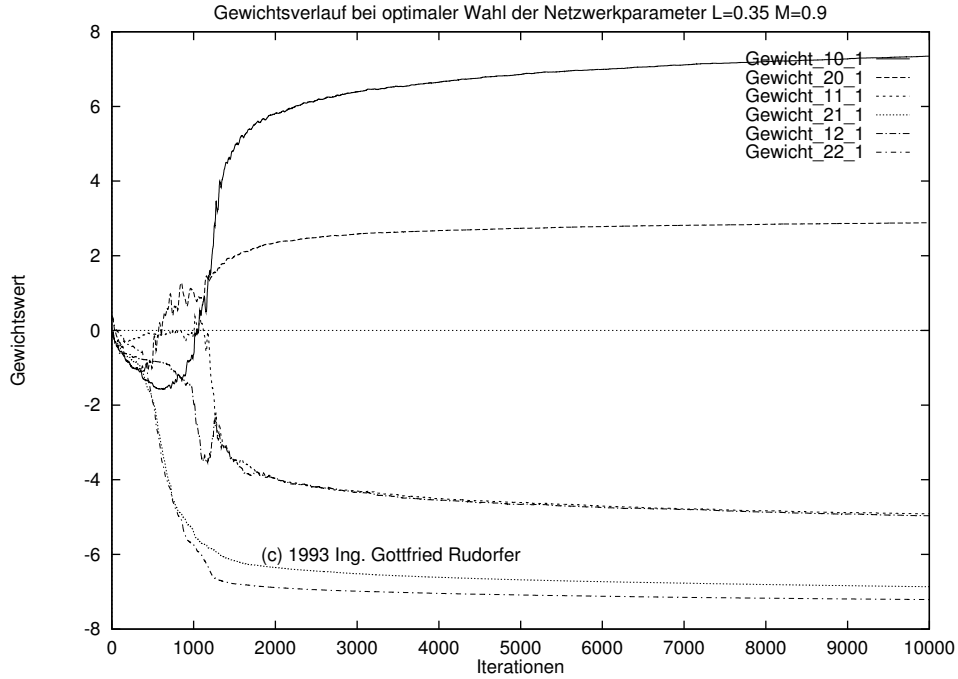

Abbildung 6.5: Gewichtsverlauf in der Eingabe-Hiddenschicht

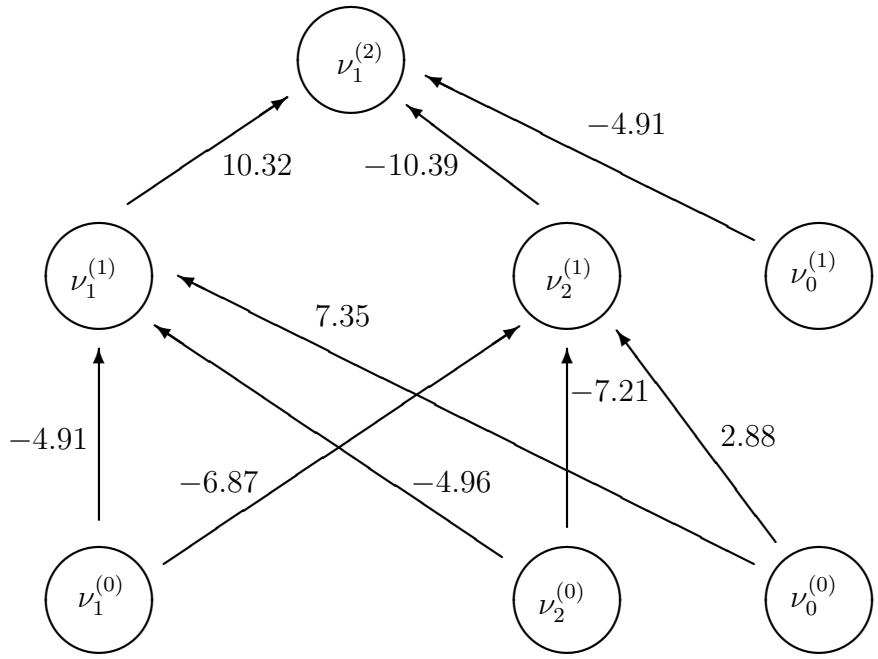

Abbildung 6.6: Gewichtswerte des XOR-Netzwerks nach erfolgreichem Lernen

Die Abbildung 6.6 zeigt noch einmal das Netzwerk und die passenden Gewichtswerte.

In der Literatur wird oft erwähnt, die letzte Gewichtsänderung zu berücksichtigen. Damit kann die Lerngeschwindigkeit erhöht werden. Die nächsten Grafiken 6.7 und 6.8 zeigen, wie sich das Lernverhalten des Netzwerks ändert, wenn die letzte Gewichtsänderung mit dem Faktor 0.5 bzw. 0.9 berücksichtigt wird.

Das Tal wird bei Erhöhung des Momentums tatsächlich größer, d.h. man kann die Lernrate in größeren Bereichen variieren ohne die Konvergenz des Netzwerks zu verlieren. Weiters fällt der Fehler bei geeigneter Wahl des Momentums sofort schon nach ca. 1,000 Iterationen stark ab. Bei einem hohen Momentum ist die 'Fehlerlandschaft' aber ziemlich zerklüftet. Die Gefahr, in einem lokalen Optimum zu landen ist relativ hoch.

Nach den bisher gezeigten Grafiken scheint eine Lernrate von 0.4 und ein Momentum von 0.9 die Lösung zu sein, bei dem das Netzwerk sehr rasch das gewünschte Verhalten lernt.

Die nächsten drei Bilder 6.9, 6.10 und 6.11 zeigen den Einfluß des Momentums auf das Lernverhalten des Netzwerks.

Bei geringer Lernrate ist der optimale Momentumwert sehr eng eingegrenzt. Er liegt bei einem Wert von ca. 0.8. Bei größeren Lernraten gibt es Sattelpunkte

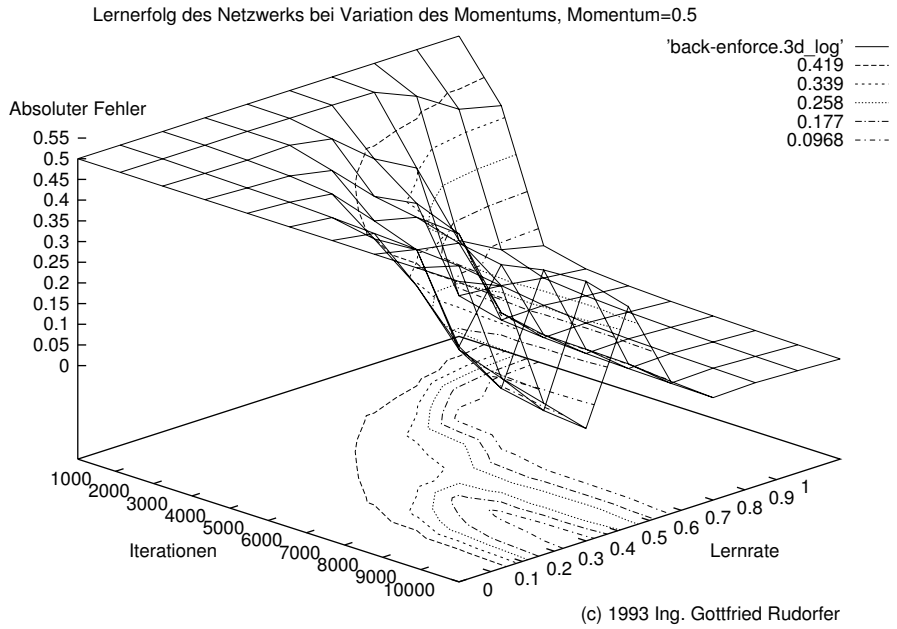

Abbildung 6.7: Lernerfolg des Netzwerks bei Variation der Lernrate unter Berücksichtigung vergangener Gewichtsänderungen mit dem Faktor 0.5

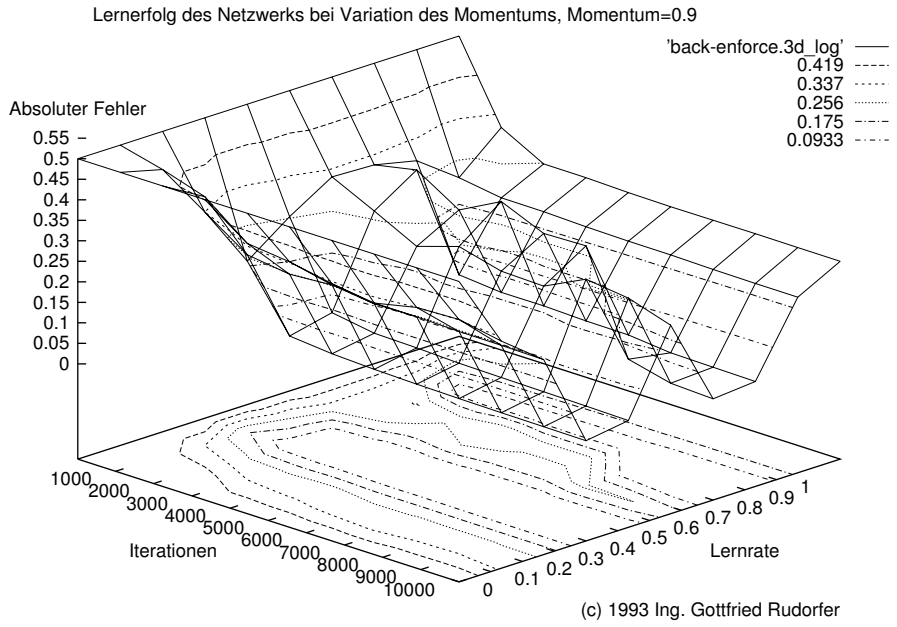

Abbildung 6.8: Lernerfolg des Netzwerks bei Variation der Lernrate unter Berücksichtigung vergangener Gewichtsänderungen mit dem Faktor 0.9

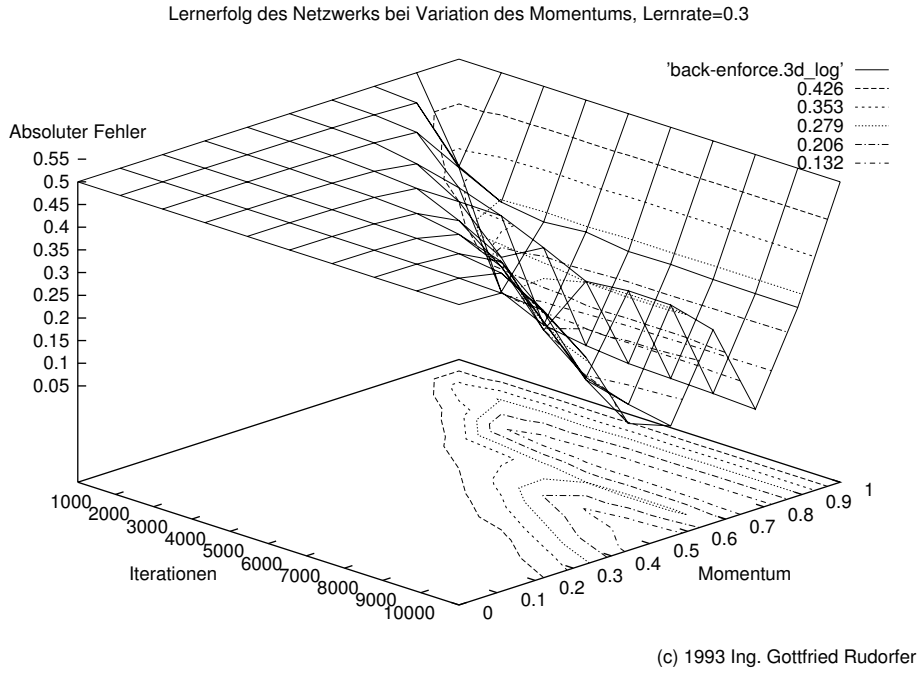

Abbildung 6.9: Lernerfolg des Netzwerks bei Variation des Momentums bei einer Lernrate von 0.3

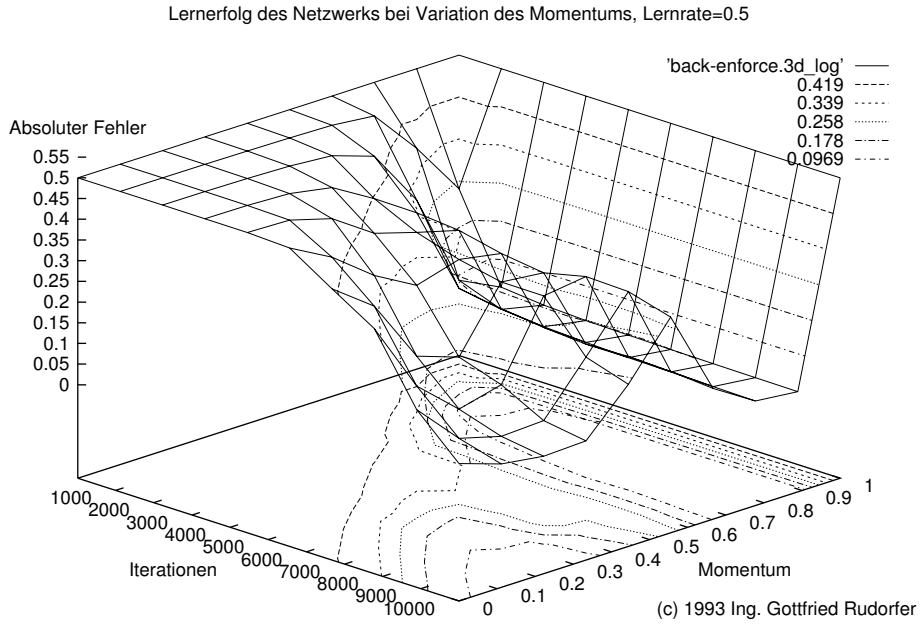

Abbildung 6.10: Lernerfolg des Netzwerks bei Variation des Momentums bei einer Lernrate von 0.5

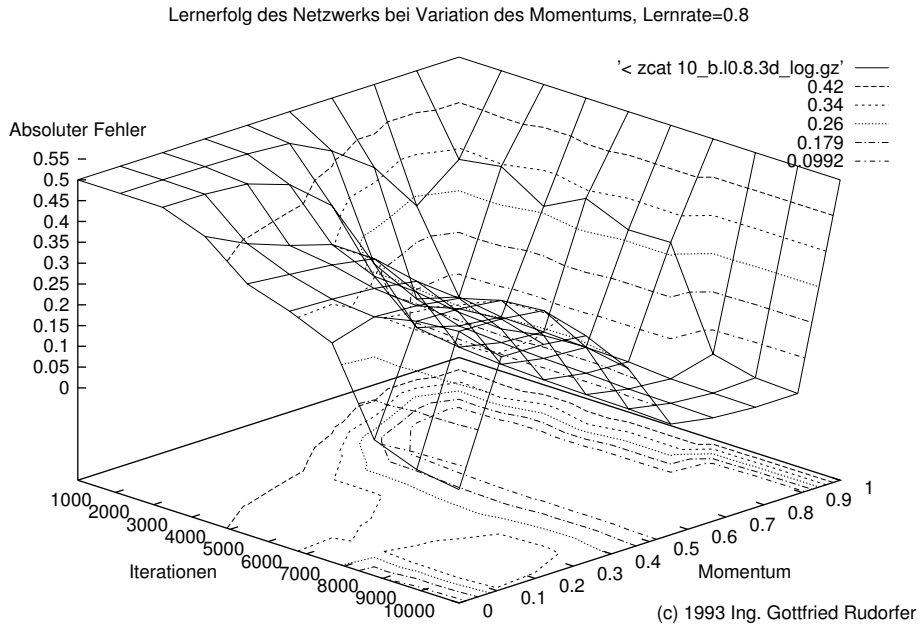

Abbildung 6.11: Lernerfolg des Netzwerks bei Variation des Momentums bei einer Lernrate von 0.8

in Fehlerfläche. Bei falscher Wahl des Momentums konvergiert das Netzwerk nicht.

Durch die Wahl des Momentums von ca. 0.8 ist ein Konvergieren des Netzwerks bei verschiedenen Lernraten gewährleistet.

Aus der Grafik können auch Netzwerkparameter abgelesen werden, die nicht zum gewünschten Verhalten des Netzwerks führen. Aus der Abbildung 6.11 führt die Wahl der Lernrate von 0.8 und ein Momentum von 0.2 zum Fehlen des Netzwerks.

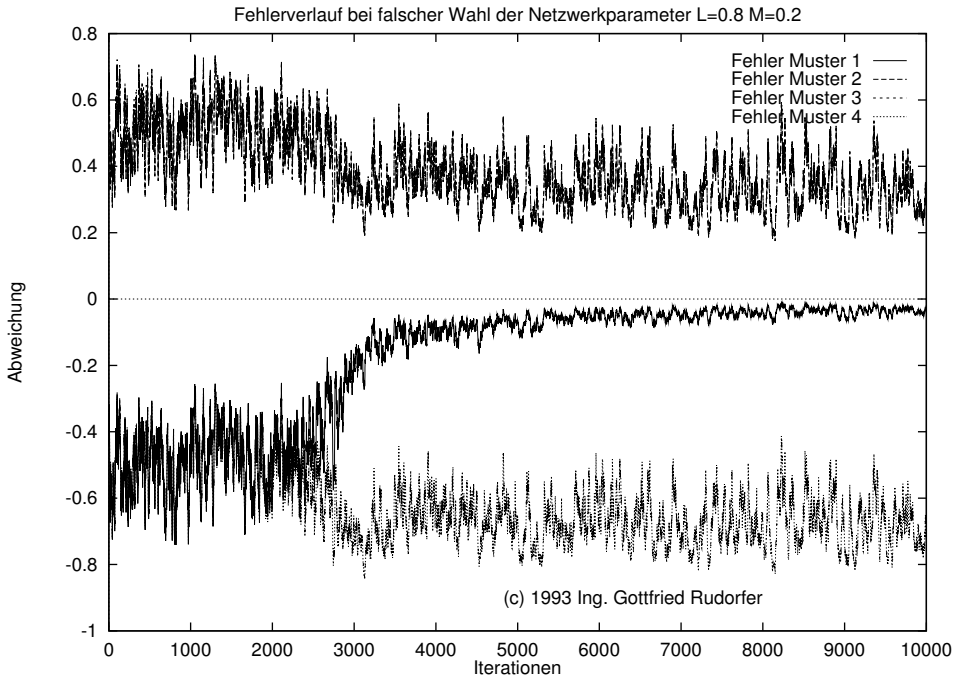

Abbildung 6.12: Fehlerverlauf bei falscher Wahl der Netzwerkparameter

Das Diagramm 6.12 zeigt den Verlauf des Fehlers für jedes Trainingsmuster. Das Netzwerk hat ein Muster richtig gelernt, bei den anderen gibt es einen oszillierenden Verlauf des Fehlers, der nicht abnimmt. Das Oszillieren kann durch die zu hohe Wahl der Lernrate begründet werden. Der Algorithmus springt mit zu großen Schritten im Lösungsraum umher.

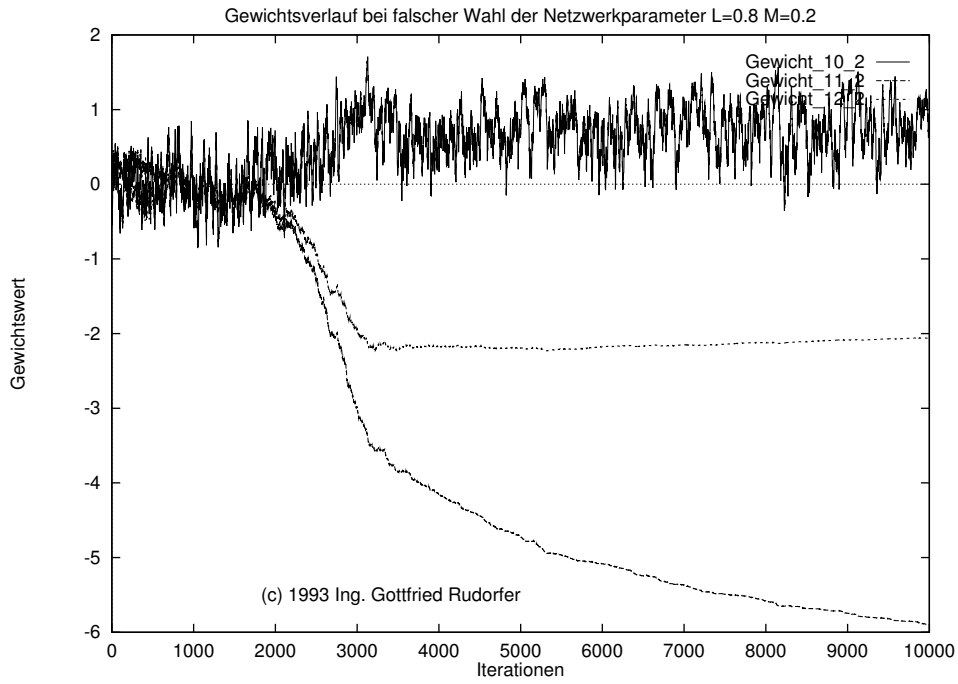

Abbildung 6.13: Gewichtsänderung zwischen Hidden-Ausgabeschicht bei falscher Wahl der Netzwerkparameter

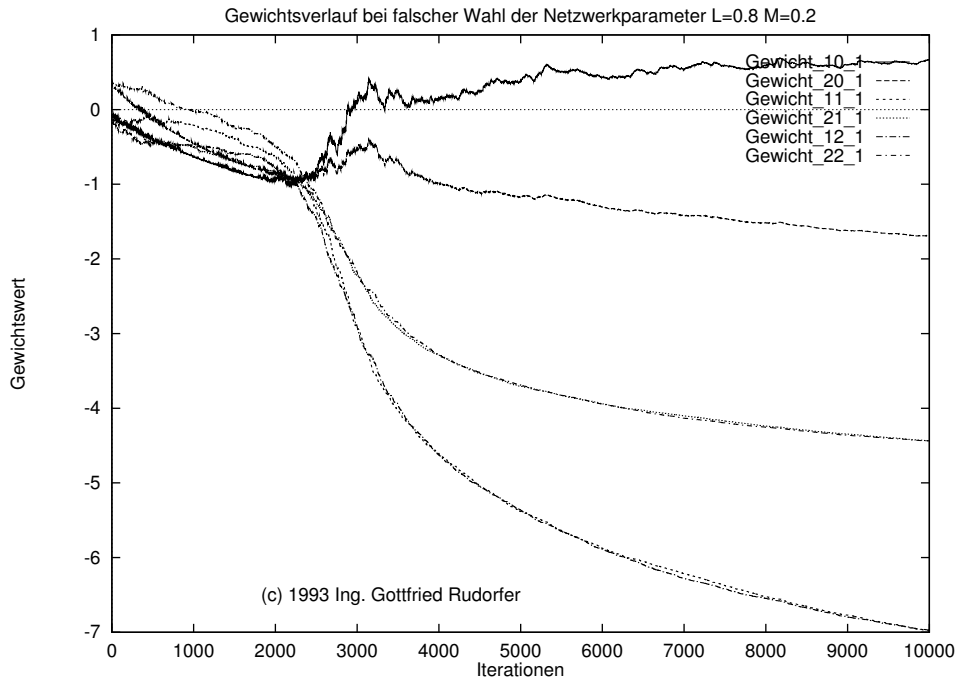

Abbildung 6.14: Gewichtsänderung zwischen Eingabe-Hiddenschicht bei falscher Wahl der Netzwerkparameter

Eingabe				Ausgabe						
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Tabelle 6.2: Zu lernende Beziehung für eine Binär zu Siebensegment Dekodierung

6.2 Erlernen einer binären Ein- Ausgabebeziehung

Das folgende Beispiel implementiert ein Neuronales Netzwerk, das eine boolsche Funktion lernt. Es soll einen sog. Binär zu Siebensegment Dekoder implementieren. Zu dem Netzwerk wurde ein Progamm zur Visualisierung des Lernerfolgs geschrieben. Es erlaubt das Ändern der Iterationsanzahl und das Einstellen des gewünschten Trainingsmusters. Dazu wird die Ausgabe des Netzwerks in Form eines Siebensegments angezeigt.

6.2.1 Modellierungsidee

Der Aufbau des Netzwerks ist ähnlich wie beim XOR-Problem. Hier hat das Netzwerk vier Eingabeneuronen für den binären Wert der Ziffer und sieben Ausgabeneuronen, die die Siebensegmentanzeige ansteuern.

Die boolsche Funktion ist in der Tabelle 6.2 gegeben.

6.2.2 APL-Funktionen

Die Simulation erfolgt wie beim XOR-Netzwerk. Daher werden diese Funktionen nicht mehr angeführt. Es wurde ein 4-7-7 Netzwerk verwendet. Neu ist jener Programmcode, der den Lernerfolg des Netzwerks visualisiert. Das Programmlistung ist im Anhang 8.3(Seite 108) zu finden.

6.2.3 Simulationsergebnisse

Nach dem Aufruf der Dyalog-APL Funktion SHOW7 erscheint ein X11-Fenster, das eine Siebensegmentanzeige zeigt. In einem Menü kann die Anzahl der Iterationen, das gewünschte Eingabe-/Ausgabemuster eingestellt werden. Der Menübaum ist in Abbildung 6.15(Seite 79) dargestellt. Es gibt einen ähnlichen Menübaum zur Visualisierung von Zeitreihen mit der TIME5 APL-Funktion. Das Netzwerk wird dann von einer Datei geladen und eine Vorwärtsrechnung mit den eingestellten Parametern durchgeführt.

Die Abbildungen 6.16, 6.17, 6.18 und 6.19 zeigen einen Ausschnitt einer APL-Sitzung. Hier setzt das Netzwerk einen binären Wert (0000 binär = 0 dezimal) in die entsprechende visuelle Darstellung um. Der Erfolg des Netzwerks ist beginnend von der Initialisierungsphase bis zum Unterschreiten eines absoluten Fehlers von maximal 0.1 (10%) dargestellt.

6.3 Zeitreihenprognose mit einem NN

Dieser Abschnitt der Arbeit hat sicherlich die höchste praktische Relevanz. Mit der Prognose einer Zeitreihe versucht man die Regularitäten eines sich in der Zeit ändernden Prozesses herauszufinden.

Der Begriff Zeitreihe ist in 3.2.12(Seite 14) definiert.

Die Zeitreihen werden nach Chatfield [Chatfield, 1991, S 5ff] mit folgenden Methoden untersucht:

1. lineare Zeitreihenmodelle

   Box & Jenkins haben rekursive Beziehungen for die Prognose von sog. Autoregressiven Moving Average (ARMA) Modellen entwickelt.

2. univariate - bivariate - multivariate Zeitreihenmodelle

   Bei der univariaten Zeitreihenanalyse wird eine Zeitreihe untersucht. Bei der bivariaten Zeitreihenanalyse werden zwei Zeitreihen und bei der multivariaten Zeitreihenanalyse mehr als zwei Zeitreihen auf deren Beziehungen untereinander untersucht. Meist erhofft man sich bei der Einbezie-

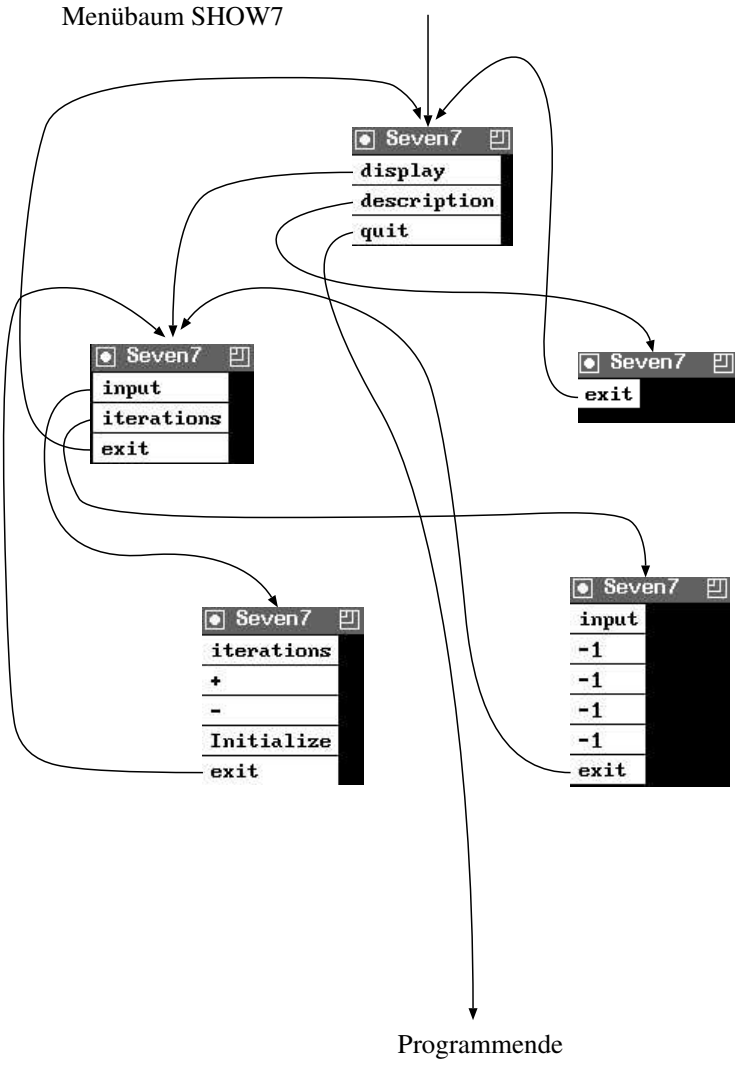

Abbildung 6.15: Der Menübaum der APL-Funktion SHOW7

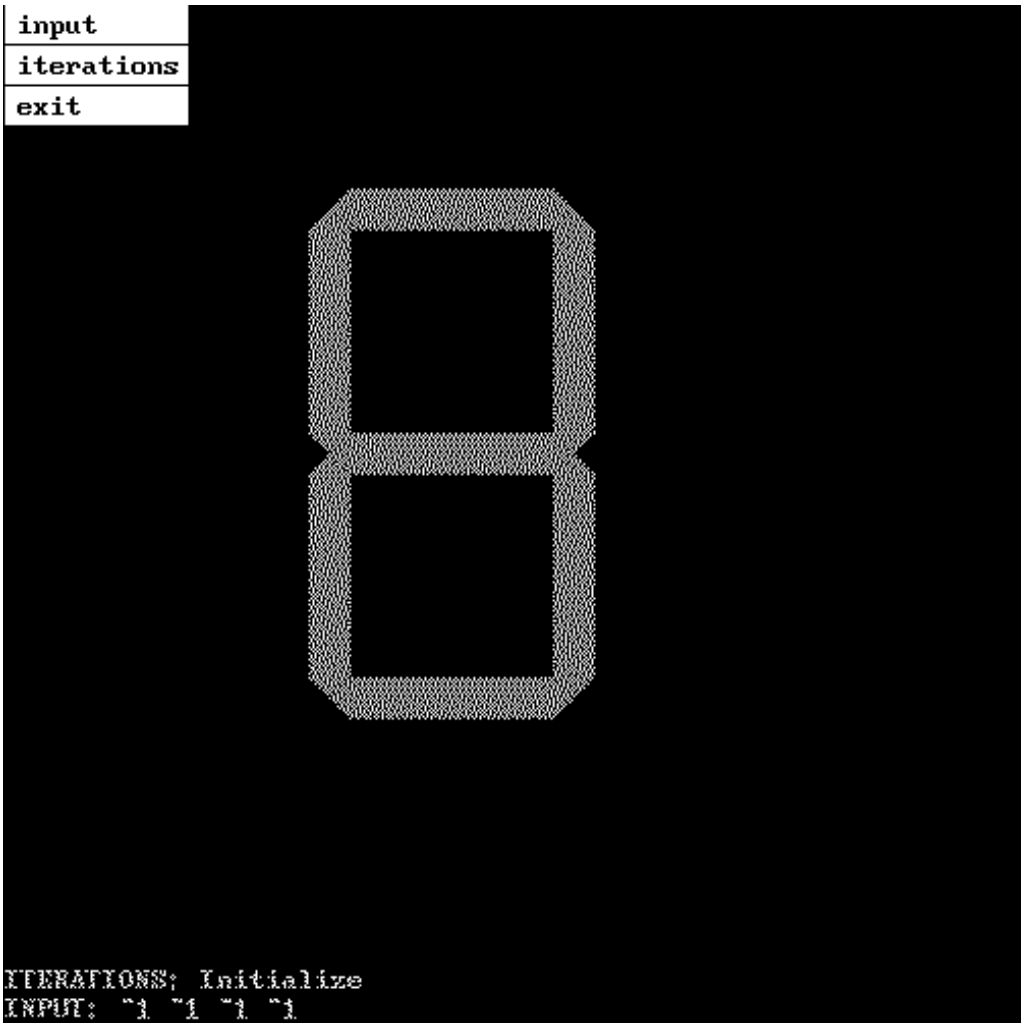

Abbildung 6.16: Ergebnis nach der Initialisierung des Netzwerks beim Eingabemuster 0000

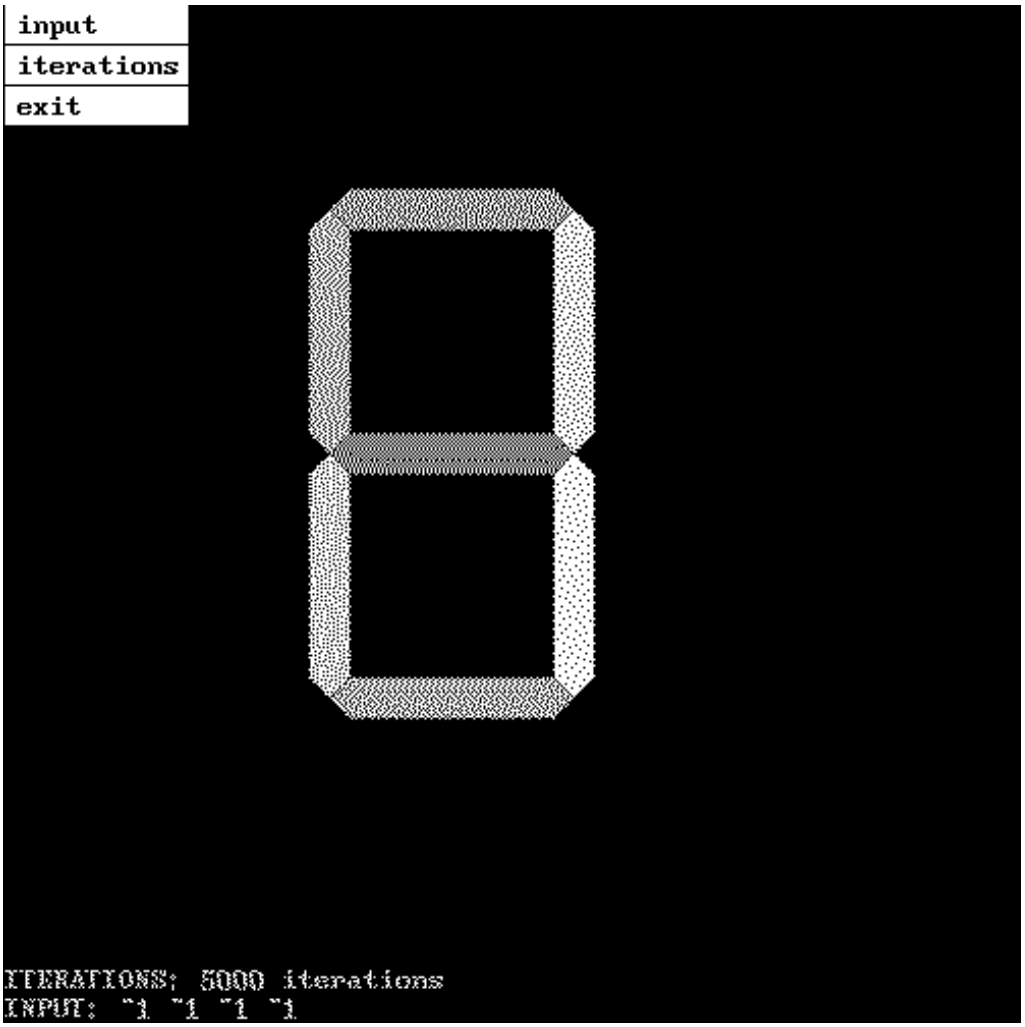

Abbildung 6.17: Ergebnis nach 5,000 Lerniterationen beim Eingabemuster 0000

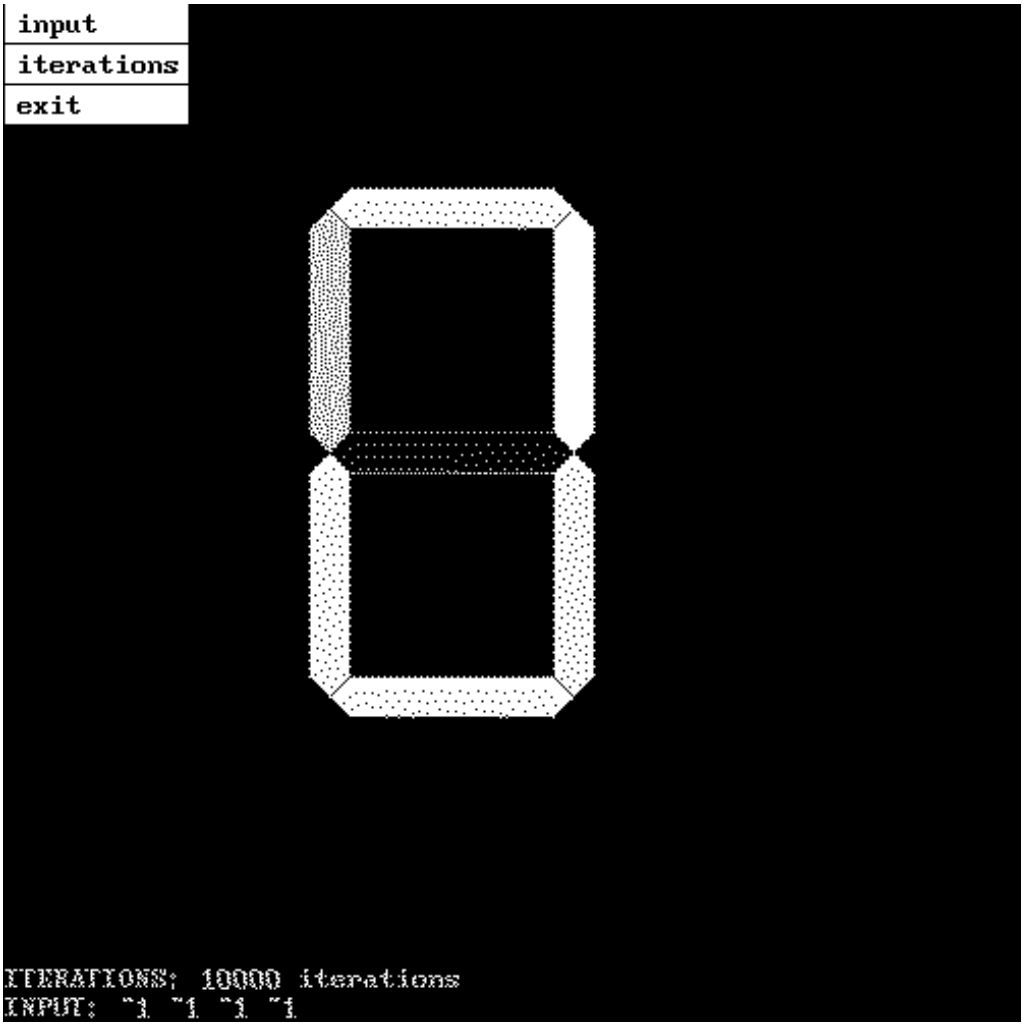

Abbildung 6.18: Ergebnis nach 10,000 Lerniterationen des Netzwerks beim Eingabemuster 0000

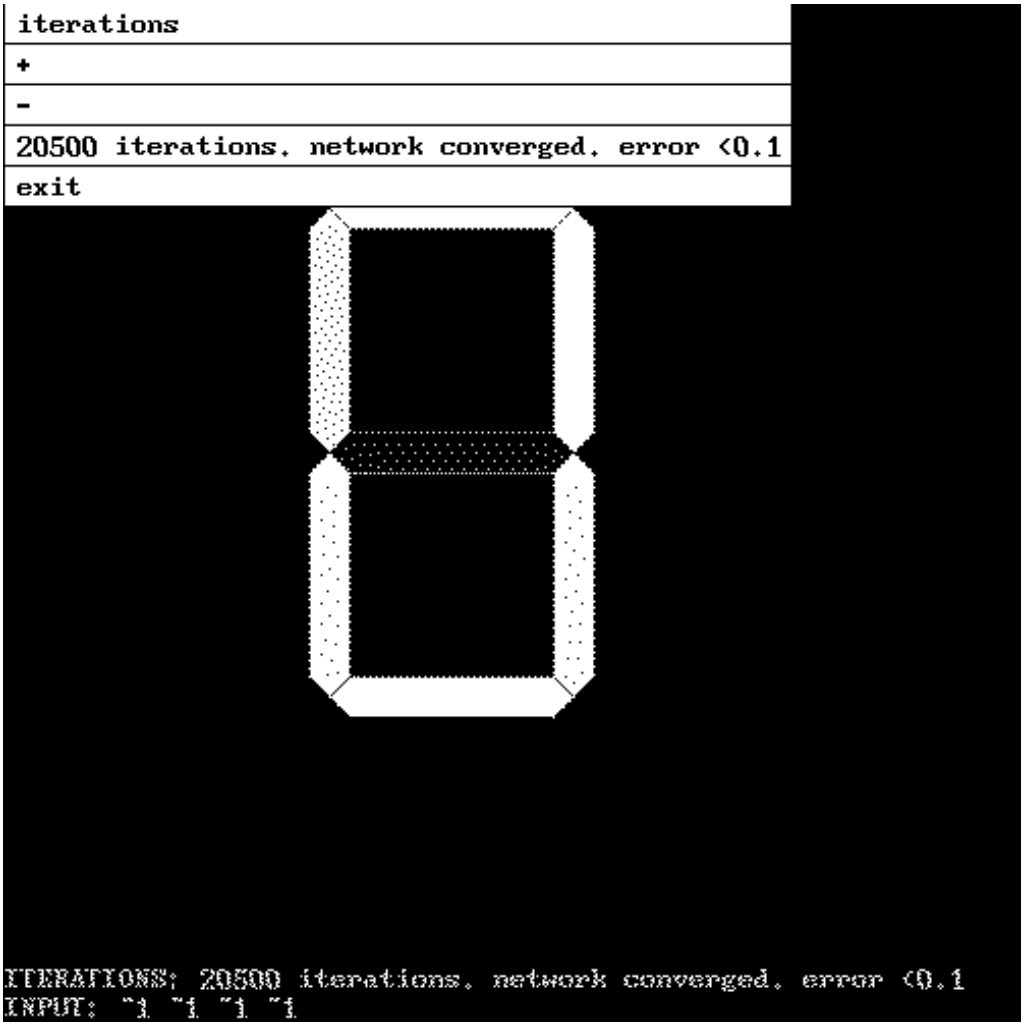

Abbildung 6.19: Ergebnis nach 20,500 Lerniterationen des Netzwerks beim Eingabemuster 0000

hung einer weiteren Zeitreihe einer verbesserte Erklärungseigenschaft des Modells.

3. Zustandsraum Modelle (Statespace Modelle)

   Dies ist eine allgemeine Klasse von Modellen, mit der auch die oben Dargestellten Zeitreihenmodelle durch eine Systemgleichung und eine Beobachtungsgleichung dargestellt werden kann.

Folgende Ziele lassen sich bei der Zeitreihenanalyse anführen:

1. Beschreibung

   Einfache deskriptive Kenndaten einer Zeitreihe (wie Mittelwert und Varianz) können schon sehr viel aussagen. Bei den oben beschriebenen Methoden geht es aber um eine aufwendigere und exaktere Untersuchung der Zeitreihe. Ziel ist es, eine gegebene Zeitreihe durch ein Modell zu so genau zu beschreiben, daß die Abweichungen zwischen der tatsächlichen Zeitreihe und der Modellzeitreihe (die Residuenzeitreihe) nur mehr rein zufällig (weißes Rauschen) sind.

2. Erklärung

   Oft kann eine Zeitreihe durch eine andere / mehrere andere Zeitreihen besser erklärt werden. Z.B. könnten Verkaufsziffern besser erklärt werden, wenn neben dem Preis auch die wirtschaftlichen Rahmenbedingungen in die Betrachtung aufgenommen werden.

3. Prognose

   Bei einer gegebenen Zeitreihe möchte man oft zukünftige Werte der Zeitreihe bestimmen. Dies wird vorallem bei Verkaufsprognosen und der Analyse wirtschaftlicher Kennzahlen durchgeführt. Die Prognose der Zukunft ist die primäre Motivation hinter der Suche nach Gesetzmäßigkeiten eines Phänomens. Ob dies möglich ist, hängt von zwei Faktoren ab:

   - die darunterliegenden Gesetzmäßigkeiten sind bekannt
   - es werden starke empirische Regularitäten in den Observationen entdeckt

   Die empirische Suche nach den Regularitäten ist oft nicht einfach, da die Zusammenhänge durch überlagerte Störungen oft nicht erkannt werden.

Zu den oben beschriebenen Untersuchungsmethoden kommt eine weitere Methode hinzu - die Analyse der Zeitreihe mit einem Neuronalen Netzwerk. So hat Chakraborty et al. [1992] Multivariate Zeitreihen mit einem Feedforward Netzwerk untersucht. Das verwendete Netzwerk hat sich als führender Konkurrent in Vergleich mit den statistischen Modellierungstechniken herausgestellt.

Ein Neuronales Netzwerk ist ein datengetriebener Ansatz. Diese Eingenschaft wird verwendet um es mit den datengetriebenen statistischen Analysemethoden zu vergleichen.

6.3.1 Modellierungsidee

Es wird das schon im vorigen Kapitel beschriebene Backpropagation-Netzwerk verwendet.

Von einer gegebenen p-variaten Zeitreihe $S = \{\langle v_1(t), ..., v_p(t)\rangle : 1 \leq t \leq N\}$ werden zwei Mengen gebildet: Das Trainingset $S_{train} = \{\langle v_1(t), ..., v_p(t)\rangle : 1 \leq t \leq T\}$ und das Testset $S_{test} = \{\langle v_1(t), ..., v_p(t)\rangle : T < t \leq N\}$. Eine Menge P_{train} von $(d+1)$-Tupel (die ersten d Elemente bilden die Eingabe des Netzwerks und das letzte Element bildet die gewünschte Ausgabe des Netzwerks) und eine Menge P_{test} von d-Tupel (jedes Element bildet die Eingabe des Netzwerks) werden dann von S_{train} und S_{test} erzeugt.

Es werden zwei Fehlermaße definiert:

- der Fehler des Netzwerks bei den Trainingdaten

$$MSE_{train} = \frac{\sum_{k=1}^{M} \left(u_k^{(train)} - \hat{u}_k^{(train)}\right)^2}{M} \tag{6.11}$$

- der Fehler des Netzwerks bei den Testdaten

$$MSE_{test} = \frac{\sum_{k=1}^{M} \left(u_k^{(test)} - \hat{u}_k^{(test)}\right)^2}{M} \tag{6.12}$$

  wobei u_k der gewünschte und $\hat{u}_k$ der tatsächliche Netzwerkausgabewert ist, $1 \leq k \leq M$ für jeden der M Trainingsmuster.

Es werden also ein paar Datenpunkte durch eine Zufallsauswahl des Startindex aus der Zeitreihe ausgewählt und an das Netzwerk als Eingabemuster angelegt. Vorher müssen die Daten noch in eine für Netzwerk passende Form gebracht werden. Das Netzwerk hat einen Ausgabeknoten. Es berechnet den Ausgabewert, der mit dem Wert der Zeitreihe der nächsten Observation verglichen wird. Die Abweichung wird verwendet, um eine Gewichtsanpassung im Netzwerk herbeizuführen.

Die Modellierungsidee wird nochmals in der Abbildung 6.20 gezeigt.

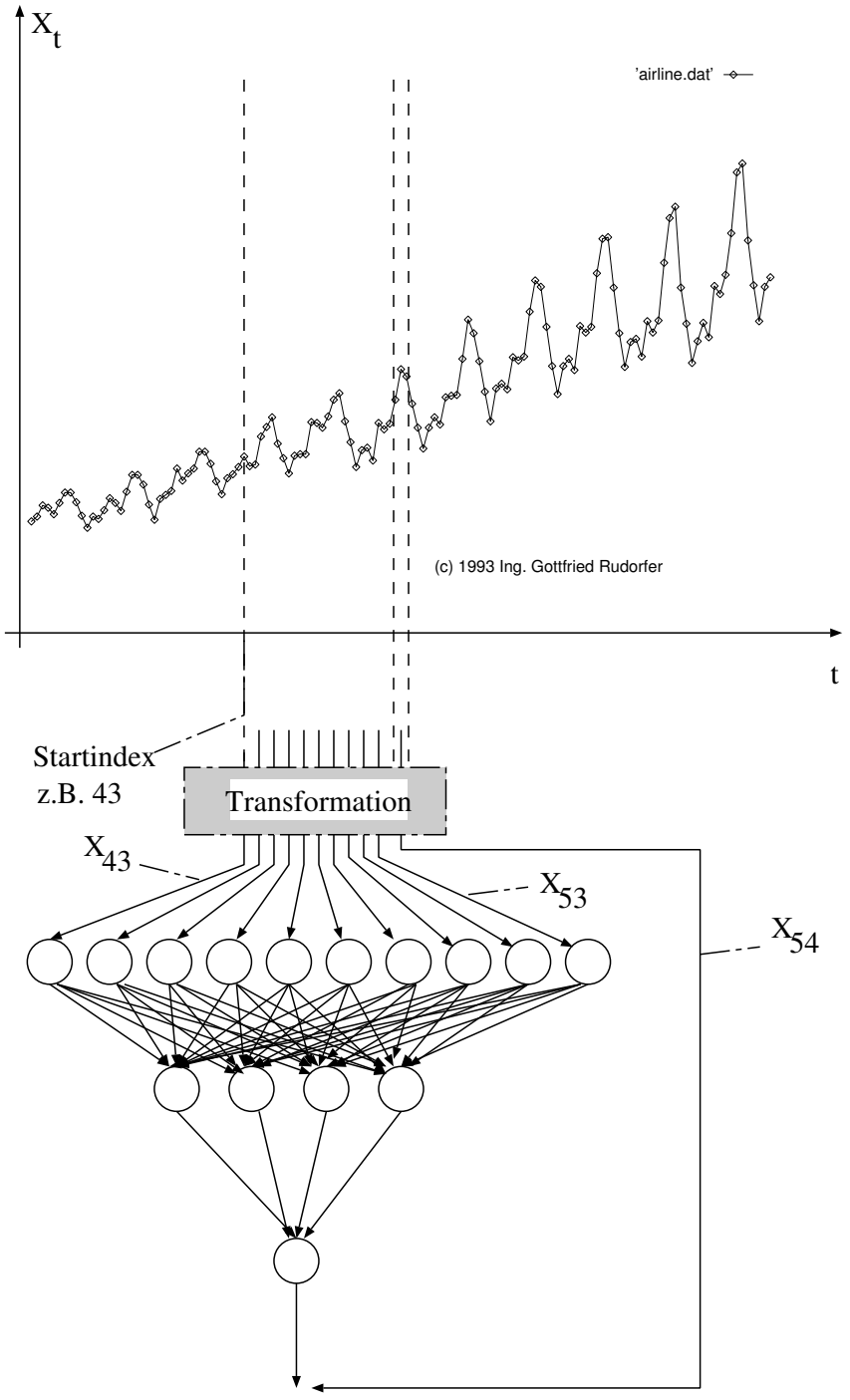

Abbildung 6.20: Die Versuchsanordnung beim Lernen der Regularitäten einer Zeitreihe

6.3.2 Transformation der Zeitreihe

Die Zeitreihenwerte (siehe Tabelle 6.3) müssen durch eine monotone Transformation in den Wertebereich der Aktivierungsfunktionen projeziert werden.

Für die Simulation wurde die logistische Funktion (siehe Formel 4.3, Seite 49) verwendet.

Die transformierte Zeitreihe Z_t wurde mit folgenden Formeln berechnet:

$$Z_t = \log(1 + X_t - \min_t(X_t)) \tag{6.13}$$

oder

$$Z_t = \frac{X_t - \min_t(X_t)}{\max_t(X_t) - \min_t(X_t)} \tag{6.14}$$

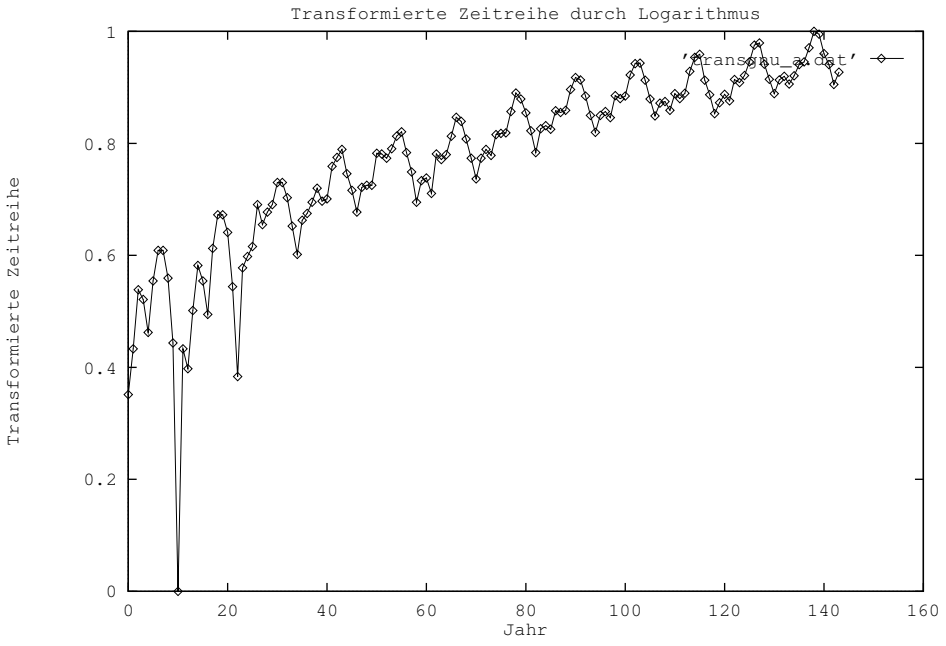

Abbildung 6.21: Transformation der Zeitreihe mit logarithmieren

Der Verlauf der transformierten Zeitreihe ist in Abbildung 6.21 und 6.22 dargestellt.

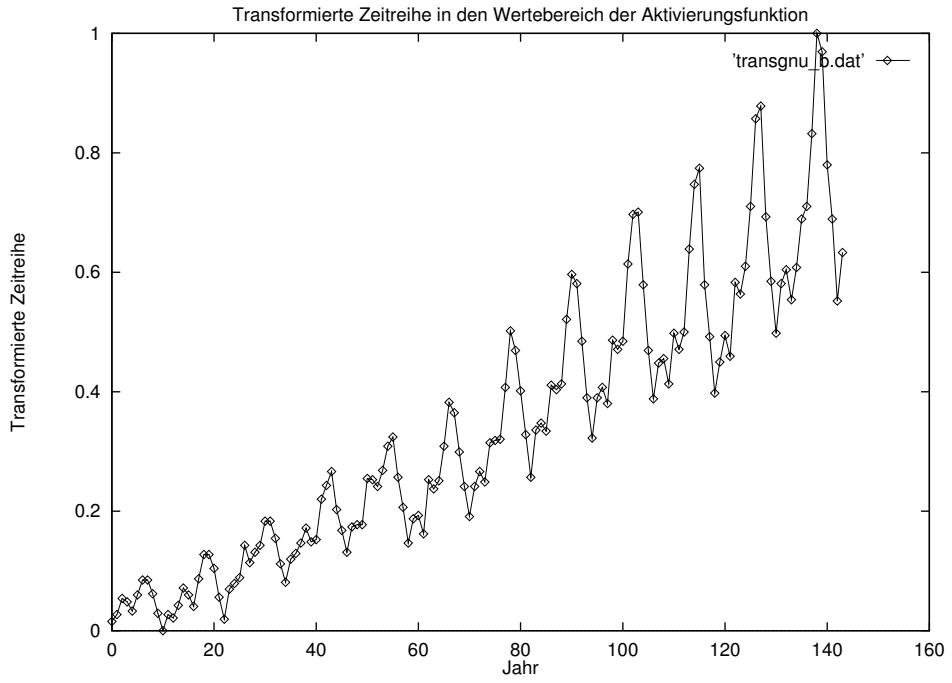

Abbildung 6.22: Transformation der Zeitreihe durch Normierung

6.3.3 APL-Funktionen

Eine APL-Funktion implementiert das oben beschriebene Netzwerk, das sich von den vergangenen Funktionen nur in Bezug auf die Schnittstelle zwischen Netzwerk und Umgebung (Datenselektion aus der Zeitreihe und der Transformation) unterscheidet.

Eine weitere APL-Funktion dient zur Visualisierung der Zeitreihe und der Ausgabe des Netzwerks. Nach dem Aufruf der Funktion erscheint ein X11-Fenster mit einer Menüleiste. Dort kann die Anzahl der Iterationen und die Position des Fensterausschnitts eingestellt werden.

6.3.4 Simulationsergebnisse

Als Beispiel zur Demonstration des Netzwerkerfolgs wurde eine bekannte Zeitreihe aus [Chatfield, 1991, S 216] herangezogen. Es sind die monatlichen Summen des internationalen Flugpassagieraufkommens vom Jänner 1949 bis Dezember 1960. Dieser bekannte Datensatz wurde schon von Box und Jenkins 1970 untersucht.

Es ist immer gut, den Verlauf der Zeitreihe in einem Diagramm darzustellen.

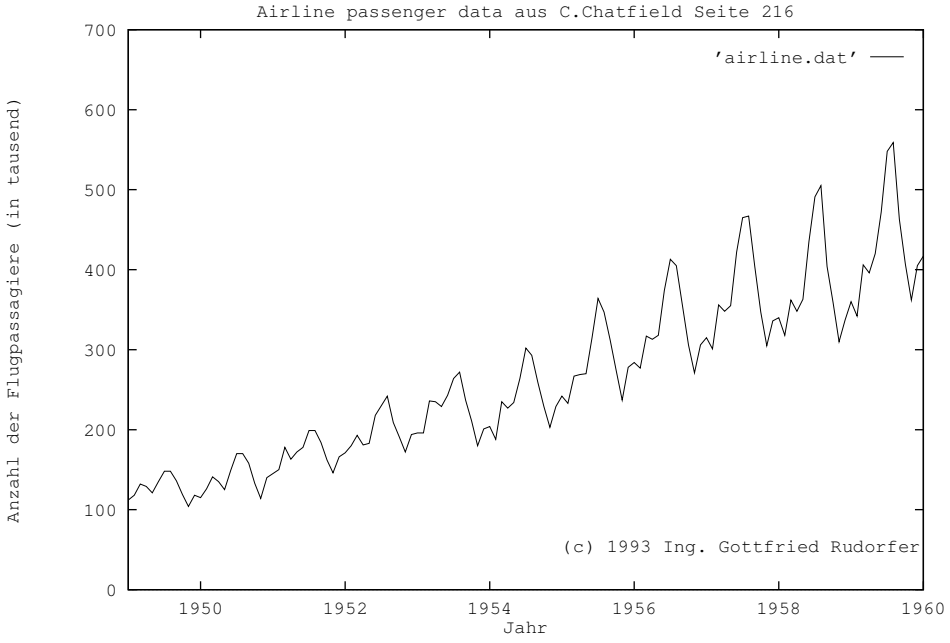

Abbildung 6.23: Monatliche Summen des internationalen Flugpassagieraufkommen vom Jänner 1949 bis Dezember 1960

Jahr	Jän.	Feb.	März	April	Mai	Juni	Juli	Aug.	Sept.	Okt.	Nov.	Dez.
1949	112	118	132	129	121	135	148	148	136	119	104	118
1950	115	126	141	135	125	149	170	170	158	133	114	140
1951	145	150	178	163	172	178	199	199	184	162	146	166
1952	171	180	193	181	183	218	230	242	209	191	172	194
1953	196	196	236	235	229	243	264	272	237	211	180	201
1954	204	188	235	227	234	264	302	293	259	229	203	229
1955	242	233	267	269	270	315	364	347	312	274	237	278
1956	284	277	317	313	318	374	413	405	355	306	271	306
1957	315	301	356	348	355	422	465	467	404	347	305	336
1958	340	318	362	348	363	435	491	505	404	359	310	337
1959	360	342	406	396	420	472	548	559	463	407	362	405
1960	417	391	419	461	472	535	622	606	508	461	390	432

Tabelle 6.3: Monatliche Summen des internationalen Flugpassagieraufkommen vom Jänner 1949 bis Dezember 1960

Iterationen	MSE_{train}
0	15632.79
500	2960.65
1000	1743.15
5000	675.95
10000	463.28

Tabelle 6.4: Verlauf der quadratischen Abweichung zwischen tatsächlichen und gewünschten Ausgabewert des Netzwerks

Die Abbildung 6.23 zeigt den Verlauf der Zeitreihe.

Die Zahlenwerte sind der Vollständigkeit halber in der Tabelle 6.3 dargestellt. Besondere Merkmale der Zeitreihe mit 144 Observationen sind ein klar erkennbarer Tend, deutliche saisonale Schwankungen und eine zunehmende Varianz.

Es wurde ein Netzwerk mit 20 Eingabeneuronen, einer Zwischenschicht mit 40 Neuronen und einer Ausgabeschicht mit einem Neuron verwendet (20-40-1). Das Netzwerk hat in Summe $20 \times 40 + 40 + 40 \times 1 + 1 = 881$ Gewichte, die vom Netzwerk mit der Backpropagation Lernregel verändert werden.

Es wurde eine geringe Lernrate von 0.1 gewählt, vergangene Gewichtsänderungen wurden nicht berücksichtigt (Momentum von 0.0).

Die Tabelle 6.4 zeigt den Verlauf des MSE_{train}. Der Lernerfolg des Netzwerks kann mit der Funktion SHOWTIME5 grafisch veranschaulicht werden.

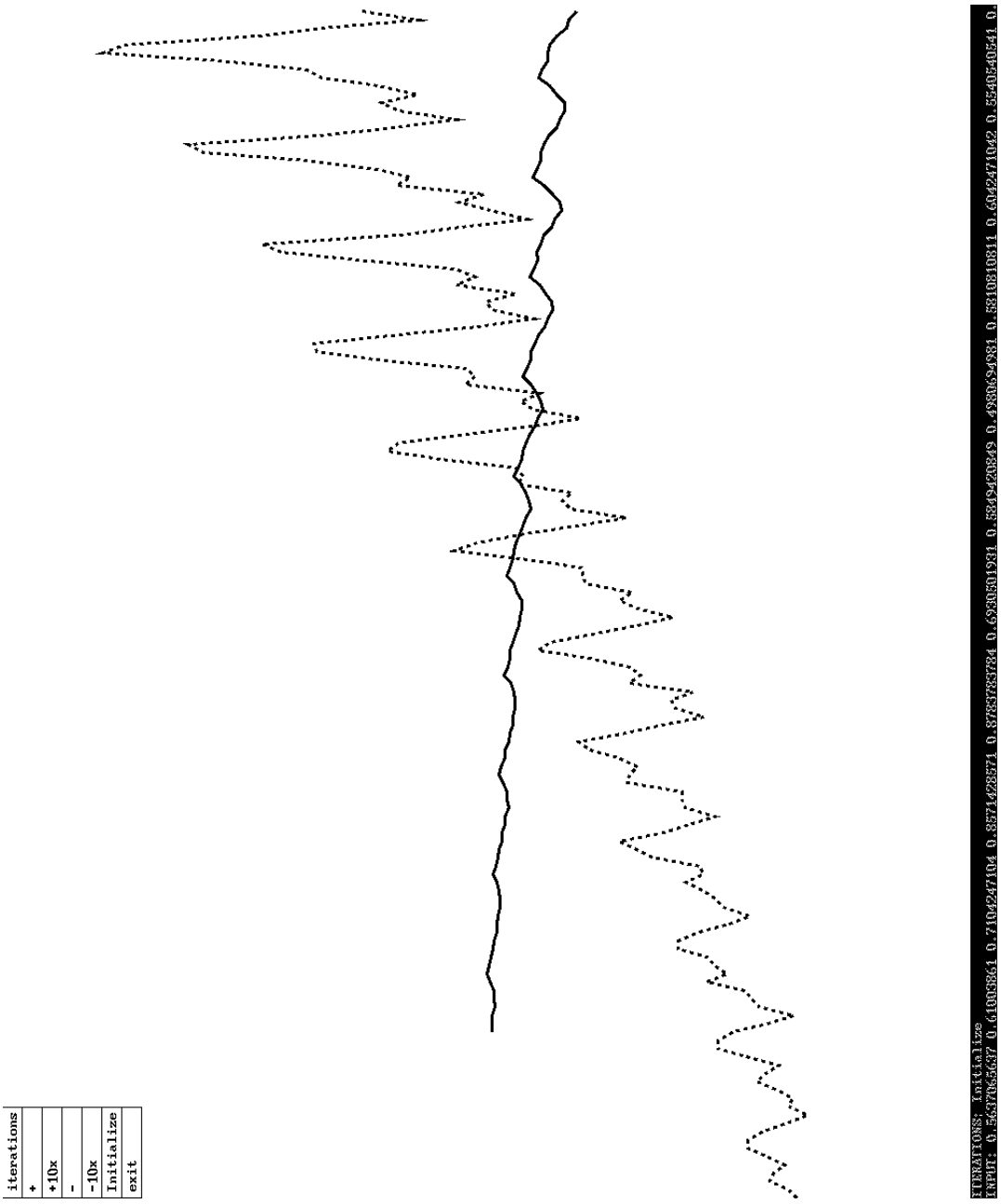

Abbildung 6.24: tatsächlicher Verlauf der Zeitreihe (strichliert) und Prognose des Netzwerks in den Trainingsdaten nach der Initialisierung

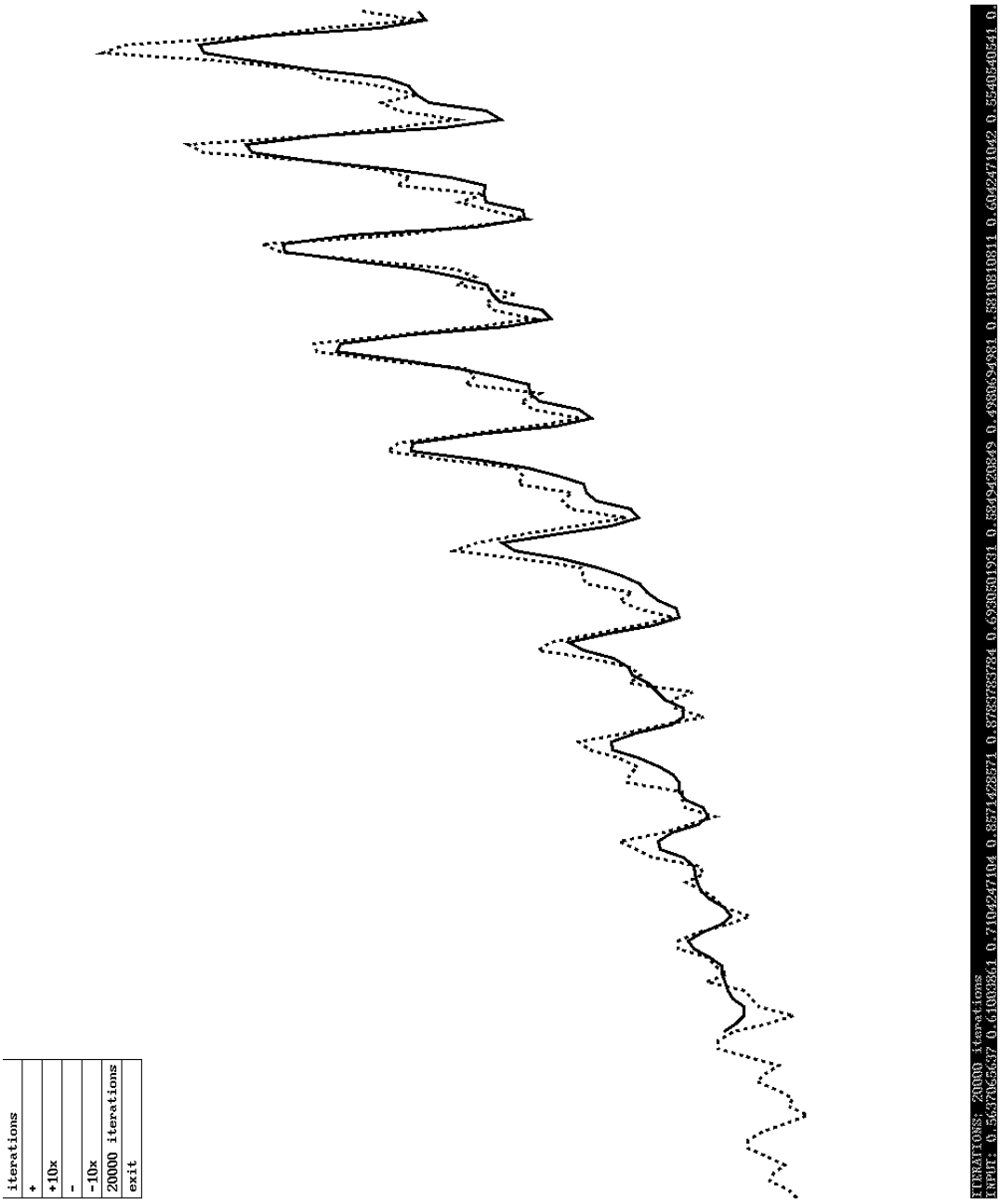

Abbildung 6.25: tatsächlicher Verlauf der Zeitreihe (strichliert) und Prognose des Netzwerks in den Trainingsdaten nach 20,000 Trainingsschritten

Die Abbildung 6.24 zeigt den Prognoseverlauf des Netzwerks nach der Initialisierung des Netzwerks. Der Verlauf der durchgezogenen Linie kann folgendermaßen erklärt werden: Das Netzwerk erhält die ersten 20 Eingabewerte $(X_1, X_2, ..., X_{20})$ und prognostiziert den nächsten Wert (X_{21}). Er ist der erste Punkt auf der durchgezogenen Geraden. Dann werden dem Netzwerk die nächsten 20 Eingabewerte$(X_2, X_3, ..., X_{21})$ präsentiert. Dies wird bis zum letzten prognostizierbaren Wert fortgesetzt.

Die Abbildung 6.25 zeigt den Verlauf der Prognose des Netzwerks in den Trainingsdaten nach $20,000$ Trainingsschritten. Die Prognose des Netzwerks hat sich schon stark an den tatsächlichen Verlauf der Zeitreihe angenähert.

Ein detaillierter Vergleich der Analysemethoden würde den über den Rahmen der Arbeit hinausgehen.

Kapitel 7

Evaluation und Ausblick

Eine stark vereinfachte Modellannahme über die natürlichen Neuronalen Netzwerke in Form des "Multi Layer Feedforward Netzwerks" ermöglichte das Erlernen beliebiger Ein- und Ausgabebeziehungen. Das ist durch die theoretischen Erkenntnisse (z.B. von Hornik et al. [1989]) gestützt.

Ein derartiger Ansatz ist bei unzureichenden exakten Wissen über die Zusammenhänge eines Phänomens und bei ausreichenden Datenmaterial zielführend. Es muß aber darauf geachtet werden, daß die gewünschte Information im Datenmaterial enthalten ist.

Voraussetzung für die Anwendung der Netzwerke ist die Implementation von Simulatoren. Für einen schnellen und einfachen Zugang sind die am Markt erhältlichen Simulatoren ausreichend. Für die Lösung eines Problems mit einen Netzwerk muß jedoch auf die konventionelle Programmierung zurückgegriffen werden, da großes Augenmerk auf die Schnittstelle zwischen dem Netzwerk und seiner Umgebung(Kodierung der Daten und Ein-/Ausgabeschnittstelle für die Daten) gelegt werden muß.

Die vorliegende Arbeit hat sich eingehend mit

- der Funktionsweise der Netzwerke ausgehend von den biologischen(natürlichen) Neuronalen Netzwerken
- der Ableitung eines künstlichen Modells
- Implementierung von Prototypen
- und deren Anwendung unter anderem zur Prognose von Zeitreihen

beschäftigt.

Tiefergehende Forschung ist in folgenden Bereichen möglich:

- Analyse eines praxisnahen Problems mit einem Modell, das sich auf Neuronale Netzwerke stützt
- Erweiterung des vorhandenen Modells mit theoretischen als auch empirischen Ergebnissen
- Entwicklung einer objektorientierten Bibliothek für Neuronale Netzwerke, die auch Klassen für die Ein- und Ausgabeschnittstelle enthält

Kapitel 8

Anhang

8.1 Anmerkungen zur Implementation in APL

Die folgenden Programmlistings enthalten wichtige Teile eines Dyalog APL-Workspaces. Die Programme verwenden auch die Erweiterungen des Dyalog APL, insbesondere die Schnittstelle zu X11 (siehe Dyadic [1991b], und Dyadic [1991a]).

8.2 APL-Funktionen XOR

```
     ▽ XOR2 INIT;⎕IO;S_INPUT;S_OUTPUT;I_OUTPUT;INPUT;OUTPUT;
       LEARNING_RATE;U;LAYER_NAMES;LAYER_UNITS;NET;RANDOM;TMP1;
       TMP2;TMP3;TMP4;TMP5;LN1;N_LAYERS;LSC;STRUCTURE;LSP;NOW;
       DSS;IDSS;LN2;WEIGHTS;B_WEIGHTS;B_INP;XJ;UI;AI;XI;LN3;
       △WEIGHTS;△B_WEIGHTS;DELTA;DELT_WEIGHT;ITCOUNT;RANDRANGE;
       OUTPUTINT;OUTPSTD;OUTPERRSER;ERRSTD;TEMPERATURE;LAST_
       INP;LAST_MOM;COMPU_TIME;FILENAME;FDIR;TEMP_MIN;TEMP_
       DECREASING;O△WEIGHTS;OB△WEIGHTS;MOMENTUM;OUTPUTFUNCTION;
       MAXIT;FNAME;RANDNUM;NEW;HIDDEN;W_DECAY
[1]    ⍝ (c) Gottfried RUDORFER 4-1992
[2]    ⍝ XOR-Problem: Ein Beispiel fuer ein nicht linear
       separables Problem
[3]    ⍝ Backpropagation mit allg. Delta-Regel
[4]    ⍝
[5]    ⍝ Aufruf: XOR2 INIT
[6]    ⍝ INIT: Ein verallgemeinertes Array, deren Elemente
       folgende Bedeutung haben:
[7]    ⍝ LEARNING_RATE MOMENTUM RANDRANGE OUTPUTINT ERRSTD
       TEMPERATURE TEMP_DECREASING TEMP_MIN OUTPUTFUNCTION MAXIT
```

```
      HIDDEN
[8]   ⍝ LEARNING_RATE: Skalar, mit dem die Lernrate festgelegt
      wird; Wertebereich: 0.0 < LEARNING_RATE < 1.0
[9]   ⍝ MOMENTUM: Skalar, mit dem vergangene Gewichtsaenderungen
      beruecksichtigt werden; Wertebereich 0.0 < MOMENTUM < 1.0
[10]  ⍝ RANDRANGE: Vektor, ρ 2, mit Zufallszahlen zwischen
      RANDRANGE[1] und RANDRANGE[2] werden die Gewichte
      initialisiert; Wertebereich 0.0 < RANDRANGE < 1.0
[11]  ⍝ OUTPUTINT: Skalar,  alle OUTPUTINT Iterationen wird der
      Zustand des Netzwerks ausgegeben,Wertebereich OUTPUTINT >
      0
[12]  ⍝ ERRSTD: Skalar, Standardabweichung vom gewuenschten
      Datensatz, die erreicht werden muss, damit der Algorithmus
      abbricht, Wertebereich: ERRSTD > 0
[13]  ⍝ TEMPERATURE: Skalar, Steilheit der Squashing−Funktion,
      Wertebereich 0.0 < TEMPERATURE < 10.0
[14]  ⍝ TEMP_DECREASING: Skalar, Wert mit dem die Temperatur bei
      jeder Iteration abnimmt, Wertebereich: TEMP_DECREASING >
      0.0
[15]  ⍝ TEMP_MIN: Skalar, Minimaler Temperaturwert,
      Wertebereich: 0.0 < TEMP_MIN < 10.0
[16]  ⍝ OUTPUTFUNCTION:Textvektor, enthaehlt den Namen der
      Funktion, die zur Berechnung der Aktivierung des Netzwerks
      verwendet wird, zulaessige Funktionen: ,EXPFUNCTION,, ,
      TANHFUNCTION,
[17]  ⍝ MAXIT: Skalar, Maximalanzahl von Iterationen, die das
      Netzwerk lernt, Wertebereich MAXIT > 0.0
[18]  ⍝ HIDDEN:Skalar, Anzahl der verborgenen Schichten im
      Netzwerk, Wertebereich: HIDDEN > 0
[19]  ⍝
[20]  ⍝ Standardnetzwerkparameter, falls das uebergebene INIT
      Array fehlende Elemente hat
[21]   LEARNING_RATE←0.1
[22]   MOMENTUM←0.9
[23]   RANDRANGE←¯0.1 0.1
[24]   OUTPUTINT←500
[25]   ERRSTD←0.05
[26]   TEMPERATURE←0.5
[27]   TEMP_DECREASING←0÷5000
[28]   TEMP_MIN←0.6
[29]   OUTPUTFUNCTION←,EXPFUNCTION,
[30]   MAXIT←20000
[31]   W_DECAY←1
```

```
[32]   B_INP←1 ⍝ INPUT TO BIAS WEIGHTS
[33]   HIDDEN←2 ⍝ ρ=number of hidden layers, vector of hidden
       layer sizes
[34]   ⍝
[35]   ⍝ Eine Fehlermeldung ausgeben, wenn falsche Netzwerkparamet
       er
[36]   ⍎(~11∈ρINIT)/,□←(⊃□SI),,,: right arg should be:
       LEARNING_RATE, MOMENTUM, RANDRANGE[2], OUTPUTINT ERRSTD
       TEMPERATURE TEMP_DECREASING TEMP_MIN OUTPUTFUNCTION[
       STRING] MAXIT HIDDEN. Using default values.,, ◇ →
       CONTINUE,
[37]   ⍝ (LEARNING_RATE MOMENTUM RANDRANGE OUTPUTINT ERRSTD
       TEMPERATURE TEMP_DECREASING TEMP_MIN OUTPUTFUNCTION MAXIT
       HIDDEN)←INIT
[38]   □←LEARNING_RATE←INIT[1]
[39]   □←MOMENTUM←INIT[2]
[40]   □←RANDRANGE←INIT[3]
[41]   □←OUTPUTINT←INIT[4]
[42]   □←ERRSTD←INIT[5]
[43]   □←TEMPERATURE←INIT[6]
[44]   □←TEMP_DECREASING←INIT[7]
[45]   □←TEMP_MIN←INIT[8]
[46]   □←OUTPUTFUNCTION←⊃INIT[9]
[47]   □←MAXIT←INIT[10]
[48]   □←HIDDEN←⊃INIT[11]
[49]   ⍝
[50]   CONTINUE:
[51]   □IO←1
[52]   ⍝ Gewuenschte Eingabe / Ausgabewerte
[53]   S_INPUT←(0 0)(1 0)(0 1)(1 1)
[54]   ⍝ S_INPUT←(¯1 ¯1)(1 ¯1)(¯1 1)(1 1)
[55]   S_OUTPUT←(1ρ⊂0)(1ρ⊂1)(1ρ⊂1)(1ρ⊂0)
[56]   LAYER_NAMES←,INPUT, ,HIDDEN, ,OUTPUT,
[57]   LAYER_UNITS←(⊃ρ¨S_INPUT),HIDDEN,(⊃ρ¨S_OUTPUT)
[58]   N_LAYERS←ρLAYER_UNITS
[59]   ITCOUNT←0
[60]   NEW←0
[61]   OUTPERRSER←0ρ0
[62]   OB△WEIGHTS←0△WEIGHTS←0ρ0
[63]   LAST_INP←0
[64]   ⍝ Dateiname fuer Simulationsergebnisse
[65]   FDIR←,/users/rudorfer/apl/component,
[66]   FILENAME←FDIR,,/,,,XOR2,
```

```
[67]   FINIT FDIR
[68]   ⍝ Name der ablaufenden Funktion
[69]   FNAME←⊃⎕SI
[70]   ⍝ Startzeit
[71]   COMPU_TIME←⊂⎕AI
[72]   ⍝ Informationen ueber das Netzwerk in Datei schreiben
[73]   PRE
[74]   ⍝
[75]   (0⍴0)DUMPFILE,erase,FILENAME
[76]   (0⍴0)DUMPFILE,erase,FILENAME
[77]   ⍝
[78]   (,Header,)(LEARNING_RATE RANDRANGE OUTPUTINT ERRSTD
       TEMPERATURE S_INPUT S_OUTPUT LAYER_NAMES LAYER_UNITS N_
       LAYERS)CW FILENAME 1
[79]
[80]   ⍝
[81]   ⍝ Vorwaertsrechnung: Die Aktivierung wird von der
       Eingabeschicht zur Ausgabeschicht ausgebreitet
[82]   ⍝ FORWARD PASS
[83]   ⍝ wij ui ai xi   ij nach i von j
[84]   ⍝  xi=g(ai)                          g=Outputfunktion
[85]   ⍝  ai=f(ui, ai(t-1), ai(t-2), ...)       f=
       Aktivierungsregel
[86]   ⍝  ui=ueber (j +/ wij xj) - theta_i
[87]   ⍝    wij ... Gewicht, theta ... Schwellwert
[88]   ⍝ oft ai=ui (kein Gedaechtnis)
[89]   ⍝
[90]   ⍝ a) ui berechnen (Summe ueber Inputs)
[91]   ⍝ b) ai berechnen (Aktivierung)
[92]   ⍝ c) xi berechnen (Out
[93]   ⍝
[94]   ⍝ STRUCTURE ist ein Array mit folgenden Elementen:
[95]   ⍝    WEIGHTS, B_WEIGHTS, XJ, UI, AI, XI, TEMPERATURE,
       0△WEIGHTS, OB△WEIGHTS  (SIZE IS 9)
[96]   ⍝
[97]   STRUCTURE←⊂(⊂,WEIGHTS,),(⊂,BIAS_WEIGHTS,),(⊂,XJ,
       ),(⊂,UI,),(⊂,AI,),(⊂,XI,),(⊂,Temperature,)(⊂,
       0△WEIGHTS,)(⊂,OB△WEIGHTS,)
[98]   ⍝ Variable, die einen Vektor mit Zufallszahlen enthaelt
[99]   RANDOM←0⍴0
[100]  ⍝
[101]  LN1←1
[102]  ⍎(0∊⎕NC FNAME,,INIT,)/,RANDOM←⍎FNAME,,,INIT←0⍴0,
```

```
      , ◇ NEW←1,
[103]  ⍋(2∈⍴NC FNAME,,INIT,)/,RANDOM←⍋FNAME,,,INIT,,,
[104] ⍝
[105] L1:→(N_LAYERS<LN1←LN1+1)/LC1A
[106] ⍝ Anzahl der Neuronen in dieser Schicht
[107]  LSC←LAYER_UNITS[LN1]
[108] ⍝Anzahl der Neuronen in der darunterliegenden Schicht
[109]  LSP←LAYER_UNITS[LN1-1]
[110] ⍝Anzahl der Gewichte
[111]  NOW←LSP×LSC
[112] ⍝ Anfangsinitialisierung der Gewichte
[113] ⍝ Erzeuge Zufallszahlen
[114]  TMP1←ρRANDOM
[115]  ⍋((1∈NEW)∧(NOW>TMP1))/,RANDOM←RANDOM,RAND RANDRANGE,
      (1000-TMP1)⌈(NOW-TMP1),
[116]  RANDNUM←NOW↑RANDOM
[117] ⍝ Initialisiere Gewichtsmatrix
[118]  ⍋(1∈NEW)/,⍋FNAME,,,INIT←(⍋FNAME,,,,,INIT,,,,),
      RANDNUM,, ◇ LN1 ,,WEIGHT,, RANDNUM ,
[119]  STRUCTURE←STRUCTURE,⊂(⊂(LSC,LSP)ρRANDNUM),(⊂,BIAS_
      WEIGHTS,),(⊂,XJ,),(⊂,UI,),(⊂,AI,),(⊂,XI,),(⊂
      TEMPERATURE),(⊂(LSC,LSP)ρ0),(⊂(LSC,1)ρ0)
[120]  RANDOM←NOW↓RANDOM
[121] ⍝
[122] ⍝ Bias Gewichte initialisieren
[123]  ⍋((1∈NEW)∧(LSC>TMP1))/,RANDOM←RANDOM,RAND RANDRANGE,
      (1000-TMP1)⌈(LSC-TMP1),
[124]  RANDNUM←LSC↑RANDOM
[125]  ⍋(1∈NEW)/,⍋FNAME,,,INIT←(⍋FNAME,,,,,INIT,,,,),
      RANDNUM,, ◇ LN1 ,,BIAS,, RANDNUM,
[126]  (2⊃LN1⊃STRUCTURE)←(LSC,1)ρB_INP×RANDNUM
[127]  RANDOM←LSC↓RANDOM
[128] ⍝
[129]  →L1
[130] LC1A:
[131] ⍝ Erzeugtes Netzwerk sichern
[132]  ((STRUCTURE)(,NIL,)(,NIL,)(,NIL,)))DUMPFILE,
      Initialize,FILENAME
[133] ⍝
[134]  IDSS←0
[135] LC1:
[136]  ITCOUNT←ITCOUNT+1
[137] ⍝ FORWARD PASS
```

```
[138] ⍝ DATA SAMPLE SIZE: DSS S_INPUT
[139]  DSS←⍴S_INPUT
[140] LC11:
[141]  TEMPERATURE←TEMPERATURE-TEMP_DECREASING
[142]  ⍎('TEMPERATURE<TEMP_MIN')/',TEMPERATURE←TEMP_MIN,
[143] ⍝ Waehle ein Trainingspaar zufaellig aus
[144]  IDSS←1↑DSS?DSS
[145] ⍝  IDSS←IDSS+1
[146] ⍝  ⍎('IDSS>DSS')/',IDSS←1,
[147] ⍝  →(LAST_INP=IDSS)/LC11
[148] ⍝ LAST_INP←IDSS
[149] ⍝
[150] ⍝ An das Netzwerk wird der zufaellig ausgewaehlte
      Trainingssatz angelegt
[151]  (6⊃1⊃STRUCTURE)←TMP1←((⍴TMP1),1)⍴TMP1←IDSS⊃S_
      INPUT
[152] ⍝[]←',IDSS,IDSS
[153] ⍝[]←IDSS⊃S_INPUT
[154]  (3⊃2⊃STRUCTURE)←TMP1
[155] ⍝ Vorwaertsrechnung (Forward Pass)
[156] ⍝ Beginne mit erster versteckter Schicht (Eingabe - 1.
      Hidden)
[157]  LN2←1
[158] ⍝ Alle weiteren Schichten
[159] L2:→(N_LAYERS<LN2←LN2+1)/LC2
[160]  WEIGHTS B_WEIGHTS XJ UI AI XI TMP1 O△WEIGHTS
      OB△WEIGHTS←LN2⊃STRUCTURE
[161]  XJ←6⊃(LN2-1)⊃STRUCTURE
[162]  UI←(B_WEIGHTS×B_INP)+(WEIGHTS+.×XJ)
[163]  AI←UI
[164]  XI←⍎'OUTPUTFUNCTION,,AI,'
[165]  (LN2⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
      TEMPERATURE O△WEIGHTS OB△WEIGHTS
[166]  →L2
[167] ⍝ Rueckwaertsrechnugn (BACKWARD-PASS)
[168] ⍝ Ausgabeschicht: △wij(l) = etha delta_i(l) xj(l-1)
[169] ⍝              delta_i(l) Fehler bei Neuron i von Schicht
      l
[170] ⍝
[171] ⍝   l=L  delta_i(L) = (di - xi(L)) g,(ai(L))
      Ausgabeschicht
[172] ⍝   l<L  delta_i(l)=g,(ai(l)) k+/ delta_k(l+1) wki(
      l+1)
```

```
[173] ⍝                                              ↑↑
[174] ⍝    k=1,...,n  n=Anzahl der Units in der Schichte l+1
[175] ⍝
[176] ⍝
[177] LC2:⍝DISPLAY STRUCTURE
[178]  WEIGHTS B_WEIGHTS XJ UI AI XI TEMPERATURE O△WEIGHTS
      OB△WEIGHTS←N_LAYERS⊃STRUCTURE
[179] ⍝ Die gewuenschte Ausgabe des Netzwerks
[180]  TMP2←((ρTMP2),1)ρTMP2←IDSS⊃S_OUTPUT
[181] ⍝
[182] ⍝
[183]  DELTA←(TMP3←TMP2−XI)×(AI DEVIATION OUTPUTFUNCTION)
[184] ⍝ Der absulte Fehler OUTPERRSER
[185]  OUTPERRSER←OUTPERRSER,⊂−TMP3
[186] ⍝ Berechnung der Gewichtsaenderung im Netzwerk
[187]  O△WEIGHTS←△WEIGHTS←(LEARNING_RATE×(DELTA+.×⌿XJ))
      +(MOMENTUM×O△WEIGHTS)
[188]  OB△WEIGHTS←△B_WEIGHTS←(LEARNING_RATE×(DELTA×1))+
      (MOMENTUM×OB△WEIGHTS)
[189] ⍝
[190]  DELT_WEIGHT←(⌿WEIGHTS)+.×DELTA
[191] ⍝ UPDATE WEIGHTS
[192]  WEIGHTS←W_DECAY×(WEIGHTS+△WEIGHTS)
[193]  B_WEIGHTS←W_DECAY×(B_WEIGHTS+△B_WEIGHTS)
[194] ⍝
[195]  (N_LAYERS⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
      TEMPERATURE O△WEIGHTS OB△WEIGHTS
[196] ⍝
[197] ⍝ Beginne mit der Ausgabeschicht und rechne den Fehler bis
      zur Eingabeschicht
[198]  LN3←N_LAYERS
[199] ⍝
[200] L3:→(2>LN3←LN3−1)/LC3
[201]  WEIGHTS B_WEIGHTS XJ UI AI XI TEMPERATURE O△WEIGHTS
      OB△WEIGHTS←LN3⊃STRUCTURE
[202]  DELTA←(AI DEVIATION OUTPUTFUNCTION)×DELT_WEIGHT
[203] ⍝
[204]  DELT_WEIGHT←(⌿WEIGHTS)+.×DELTA
[205]  O△WEIGHTS←△WEIGHTS←(LEARNING_RATE×(DELTA+.×⌿XJ))
      +(MOMENTUM×O△WEIGHTS)
[206]  OB△WEIGHTS←△B_WEIGHTS←(LEARNING_RATE×(DELTA×B_
      INP))+(MOMENTUM×OB△WEIGHTS)
[207] ⍝ Korrigiegiere die Gewichtswerte
```

```
[208]  WEIGHTS←W_DECAY×(WEIGHTS+△WEIGHTS)
[209]  B_WEIGHTS←W_DECAY×(B_WEIGHTS+△B_WEIGHTS)
[210]  (LN3⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
       TEMPERATURE 0△WEIGHTS OB△WEIGHTS
[211]  →L3
[212] LC3:⍝ Alle OUTPUTINT Iterationen wird der Zustand des
       Netzwerks ueberprueft
[213]  ⍪(0=OUTPUTINT|ITCOUNT)/,→(1∈CHECKPOINT)/0,
[214]  →LC1
    ▽

    ▽ Z←CHECKPOINT;CONVERGED;TMP1;T1;T2;T3;T4;T5;T6;T7;T8;T9;
      T10;T11;T12
[1]    ⍝ (c) 06-1992 Gottfried RUDORFER
[2]    ⍝ Idea: The status of the neural network is dumped to a file
[3]    ⍝ the last error values of the output layer are
       investigated and compared to the goal value
[4]    ⍝ ...
[5]    ⍝
[6]    ⍝ THIS FUNCTION IS CALLED FROM THE XOR, ENC, SEVEN, SEVEN2
       FUNCTIONS
[7]    ⍝
[8]    ⍝ SHOW NUMBER OF ITERATIONS SO FAR
[9]     Z←0
[10]   □←,,
[11]   □←(,ITERATIONS,)(ITCOUNT)
[12]   □←(,INPUT,)(6⊃1⊃STRUCTURE)
[13]   □←(,OUTPUT,)(S_OUTPUT[IDSS])
[14]   ⍝ SHOW STRUCTURE OF THE NET
[15]   ⍝ SHOWWEIGHTS STRUCTURE
[16]   CONVERGED←ERRORCALC OUTPERRSER
[17]   □←T1←(,STD,)((+/CONVERGED)÷ρCONVERGED)
[18]   □←T2←(,MAX. UNIT ERROR,)(⌈/CONVERGED)(,MIN. UNIT
       ERROR,)(⌊/CONVERGED)
[19]   □←T3←(,ERROR OF THIS PASS,)(¯1↑OUTPERRSER)
[20]   ⍝
[21]   →(1∈~∨/ERRSTD<CONVERGED)/END
[22]   →(ITCOUNT>MAXIT)/FAIL
[23]   ⍝
[24]   ⍝ dump to file
[25]   ((STRUCTURE)(T1 T2 T3))DUMPFILE((⍪ITCOUNT),,
       iterations,)(FILENAME)
[26]   ⍝
```

```
[27]    OUTPERRSER←0⍴0
[28]    →0
[29]   FAIL:,Network Fail!! Number of Iterations greater than ,,
       MAXIT
[30]    →REPORT
[31]   END:,Network Success!! The last ,,OUTPUTINT,, output
       standard deviations are lower than ,,ERRSTD
[32]    →REPORT
[33]   REPORT:Z←1
[34]    □←T4←,Number of Iterations:,,ITCOUNT
[35]   ⍝ SHOWWEIGHTS STRUCTURE
[36]    ,,
[37]    CONVERGED←ERRORCALC OUTPERRSER
[38]    □←T5←(,Min abs. output error:,)(⌊/OUTPERRSER)
[39]    □←T6←(,Max abs. output error:,)(⌈/OUTPERRSER)
[40]    □←,,
[41]    □←T7←(,Min std. output error:,)(⌊/CONVERGED)
[42]    □←T8←(,Max std. output error:,)(⌈/CONVERGED)
[43]   ⍝ Time-Information
[44]   ⍝
[45]    COMPU_TIME←COMPU_TIME,⊂□AI
[46]    ⍎((1=≡COMPU_TIME)∨(2≠⍴COMPU_TIME))/,→0,
[47]   ⍝
[48]    □←T9←(,Compute time: (in seconds),)((÷1000)×(2⊃2⊃
       COMPU_TIME)-(2⊃1⊃COMPU_TIME))
[49]    □←T10←(,Connect time: (in seconds),)((÷1000)×(3⊃
       2⊃COMPU_TIME)-(3⊃1⊃COMPU_TIME))
[50]    □←T11←(,Keying time: (in seconds),)((÷1000)×(4⊃2⊃
       COMPU_TIME)-(4⊃1⊃COMPU_TIME))
[51]   ⍝
[52]   ⍝ dump to file
[53]    ((STRUCTURE)(T4 T5 T6 T7 T8 T9 T10 T11))DUMPFILE((⍕
       ITCOUNT),, iterations. network converged. error <,,⍕
       ERRSTD)(FILENAME)
[54]   ⍝
[55]    →0
    ▽

    ▽ DATA DUMPFILE RI;TMP1;TMP2;TMP3;FN;TI
[1]    ⍝
[2]    ⍝ Schreibe eine APL-Variable in eine Datei
[3]    ⍝
[4]     TI FN←RI
```

```
[5]    ⍝
[6]     →(0∈ρDATA)/ERASE
[7]     TMP1←2⊃CR FILENAME 0
[8]     ⍪(0∈ρTMP1)/,TMP1←1,
[9]     ((TI)(DATA))CW(FILENAME)(⍪⍤1 TMP1)
[10]    →0
[11]   ERASE:(0ρ0)CW FILENAME 0
[12]    →0
    ▽

    ▽ {RETURN}←DATA CW NACO;TMP1;TMP2;TMP3;TMP4;FNUMS;FNAMES;
      FNAME;FNUM;COMPONENT;FSIZE;NUMERIC
[1]    ⍝
[2]    ⍝ (c) Gottfried RUDORFER
[3]    ⍝ (c) 06-1992
[4]    ⍝ (c) Functions for manipulating large nested APL Variables
[5]    ⍝     with the dyalog component file system.
[6]    ⍝
[7]    ⍝ idea: only a part of the big variable is in the workspace
[8]    ⍝       the big clumb resides in the file, only small
       elements are
[9]    ⍝       written or read.
[10]   ⍝
[11]   ⍝ Arguments: DATA:   Data Component
[12]   ⍝            NACO:   The name of the file. The component
       number.
[13]   ⍝
[14]    RETURN←1
[15]   ⍝ Check the availability of the file system (file FSCB must
       have permission 666)
[16]    →(0∈⊃FAVAIL)/NOFS
[17]   ⍝
[18]   ⍝ check given arguments
[19]    →(2≠|≡NACO)/ERR
[20]    →(2≠ρNACO)/ERR
[21]    FNAME COMPONENT←NACO
[22]   ⍝
[23]    NUMERIC←,0123456789,
[24]    →(~∧/(⍪⍤1 COMPONENT)∈NUMERIC)/NUERR
[25]   ⍝ List numbers of tied files
[26]    FNUMS←⊃FNUMS
[27]    ⍪(0∈ρFNUMS)/,FNUMS←0,
[28]   ⍝
```

```
[29]    FNAMES←(↓[2]⌷FNAMES)~¨ , ,
[30]    →(~∨/TMP1←FNAMES∈⊂FNAME)/CREATE
[31]    ⍝ exitsting tie file is updated:
[32]    FNUM←TMP1/FNUMS
[33]    FSIZE←⌷FSIZE FNUM
[34]    ⍝
[35]    →((0∈∧/⍴DATA)∧(COMPONENT<FSIZE[1]))/ERASE
[36]    →((COMPONENT>FSIZE[2])∨(COMPONENT<FSIZE[1]))/
        SIZEERROR
[37]    →(COMPONENT=FSIZE[2])/ADD
[38]    ⍝ The last case: replace component
[39]    DATA ⌷FREPLACE FNUM,COMPONENT
[40]    RETURN←0
[41]    →0
[42]    CREATE:FNUM←1+⌈/FNUMS
[43]    FNAME ⌷FCREATE FNUM
[44]    (1 3⍴⌷AI[1],¯1,0)⌷FSTAC FNUM
[45]    TMP2←DATA ⌷FAPPEND FNUM
[46]    →(1≠TMP2)/EXISTERR
[47]    RETURN←0
[48]    →0
[49]    ⍝
[50]    ADD:TMP2←DATA ⌷FAPPEND FNUM
[51]    →(TMP2≠COMPONENT)/SIZEERROR
[52]    RETURN←0
[53]    →0
[54]    ERASE:FNAME ⌷FERASE FNUM
[55]    →0
[56]    ⍝
[57]    NOFS:⍞←(,⍕⊃⌷SI),,: The file−sytem is not available (
        file FSCB). exit.,
[58]    →0
[59]    ERR:⍞←(,⍕⊃⌷SI),,: Wrong arguments given. exit.,
[60]    →0
[61]    NUERR:⍞←(,⍕⊃⌷SI),,: Second part of the right argument
        must be numeric. exit.,
[62]    →0
[63]    SIZEERROR:⍞←(,⍕⊃⌷SI),,: The no such component. exit.,
[64]    →0
[65]    EXISTERR:⍞←(,⍕⊃⌷SI),,: The file ,,FNAME,, exists. you
        may have problems.,
[66]    →0
[67]    ⍝
```

```
[68]   ⍝ (c) Gottfried RUDORFER
[69]   ⍝ (c) 06-1992
[70]   ⍝ (c) Functions for manipulating large nested APL Variables
[71]   ⍝     with the dyalog component file system.
[72]   ⍝
    ∇

    ∇ Z←{L}EXPFUNCTION A;T
[1]    ⍝ global TEMPERATURE ... temperature  T→0 ... sharp
       function
[2]     ⍎(0∊⎕NC,TEMPERATURE,)/,T←1,
[3]     ⍎(~0∊⎕NC,TEMPERATURE,)/,T←TEMPERATURE,
[4]     →(~0∊⎕NC,L,)/REQUEST_DEVIATION
[5]     Z←÷(1+⋆-(A÷T))
[6]     →0
[7]    REQUEST_DEVIATION:Z←÷(T×(⋆(A÷T))×((1+(⋆(-A÷T)))
       ⋆2))
[8]     →0
    ∇

    ∇ Z←TANHFUNCTION A
[1]    ⍝ global TEMPERATURE ... temperature  T→0 ... sharp
       function
[2]     Z←7○(A÷10)
    ∇

    ∇ PRE
[1]    ⍝ (c) 06-1992  Gottfried RUDORFER
[2]    ⍝
[3]    ⍝ Information ueber Neuronales Netzwerk
[4]    ⍝
[5]    ⍝
[6]     ,Information about NN,
[7]     (,Learning Rate:,)(LEARNING_RATE)
[8]     (,Network inititalized with weights in the range of:,)(
       RANDRANGE)
[9]     (,Status of the network is shown in every ,)(OUTPUTINT)(,
       iterations,)
[10]    (,The iteration process is stops if the std. error of all
       units is less than,)(ERRSTD)
[11]    (,The temperature is ,)(TEMPERATURE)
[12]    ,,
[13]    (,The input patterns are:,)
```

```
[14]    DISP S_INPUT
[15]    ,,
[16]    (,The output patterns are:,)
[17]    DISP S_OUTPUT
[18]    ,,
[19]    ,please be patient.....,
    ▽
```

8.3 APL-Funktionen Mapping

```
    ▽ SHOW7;COLDATA;TMP1;TMP2;TMP3;TMP4;TMP5;TMP6;TMP7;TMP8;
      BACKC;SIZE;TITLE;CONT;COLOURS;COLNAMES;HOST;RGB;BCOLOUR;
      RG;RB;GB;SRG;SRB;SGB;BRGB;COLERR;DENSCOLS;CSORT;SEGMENT;
      LCOLOUR;LINDEX;DSEG;ORIGIN;A;B;C;D;E;F;G;NCOLS;BACKI;
      LWIDTH;EQUEUE;WINID;MTEXT;FILENAME;TITLE;DATA;FSIZE;DTEXT;
      TMP9;TMP10;FONT;FONTI;FONTC;FONTCI;LASTMENU;TMP11;INPUT;
      ITERATIONS;COMPN;STRUCTURE;TMPF1;LN2;WEIGHTS;B_WEIGHTS;
      XJ;UI;AI;XI;N_LAYERS;XIC;TEMPERATURE;ERROR;FDIR;LETTER_N;
      TMP12;OUTPUTFUNCTION;MAXOUT
[1]   ⍝DATA: LEARNING_RATE RANDRANGE OUTPUTINT ERRSTD TEMERATURE
      S_INPUT S_OUTPUT LAYER_NAMES LAYER_UNITS N_LAYERS
      OUTPUTFUNCTION
[2]   ⍝ X11 SEVEN SEGMNETS
[3]    LETTER_N←(((ρLETTER_N)÷2),2)ρLETTER_N←1 1 3 1 3 6 5
      1 7 1 7 9 5 9 5 5 3 9 1 9 1 1
[4]    FDIR←,/users/rudorfer/apl/component,
[5]    FILENAME←FDIR,,/,,,SEVEN2,
[6]    FINIT FDIR
[7]   ⍝
[8]    HOST←,exaix2:0.0,
[9]    SIZE←500 500
[10]   BACKC←,black,
[11]   TITLE←,Seven7,
[12]   LWIDTH←1
[13]   MTEXT←,display segment, ,description, ,input, ,
      iterations, ,quit,
[14]   LASTMENU←1
[15]   FONT←,apl14-bold,
[16]   FONTC←,green,
[17]   INPUT←¯1 ¯1 ¯1 ¯1
[18]   COMPN←2
[19]  ⍝ segemet
```

```
[20]   SEGMENT←(((ρSEGMENT)÷2),2)ρSEGMENT←0 2 2 0 22 0 24 2
       22 4 2 4 0 2
[21]   TMP6←ρSEGMENT←SEGMENT×5
[22]   ORIGIN←TMP6ρ150 150
[23]   ⍝ dimensions of segments length with
[24]   DSEG←(⌈/SEGMENT)-(⌊/SEGMENT)
[25]   ⍝ Basic colour
[26]   BCOLOUR←,white,
[27]   ⍝ Line colour
[28]   LCOLOUR←,red,
[29]   ⍝
[30]   ⎕←,generating colour mappings ...,
[31]   COLDATA←↓[2]GETFILE,/usr/lib/X11/rgb.txt, ,ascii,
[32]   COLDATA←BLANKS¨COLDATA
[33]   COLNAMES←3 BTROP¨COLDATA
[34]   RGB←3 BTAKE¨COLDATA
[35]   ⍝
[36]   ,Xfns,⎕SH,,
[37]   ⍝
[38]   TMP1←XOPEN_DISPLAY HOST
[39]   ⍝
[40]   WINID←XCREATEWIN 0 0 0,SIZE,10,(2ρ⊂BACKC),(2ρ⊂TITLE)
[41]   ⍝
[42]   FONTI←XGETFONT FONT
[43]   ⍝ select which types of events are to be repported
[44]   TMP7←WINID XSELECTINPUT+/2*2 3 15 17
[45]   ⍝ set trap for reading event queue (weak interrupt)
[46]   ⍝⋄⎕TRAP←1002 ,E, ,→READQUEUE,
[47]   ⍝
[48]   ⍝ enable event notification
[49]   ⍝TMP8←XINTR 1
[50]   ⍝
[51]   CONT←XCREATEGC
[52]   ⍝
[53]   COLOURS←XGETCOLOUR COLNAMES
[54]   ⍝
[55]   TMP3←(COLOURS≥0)/ιρCOLOURS
[56]   ⍝
[57]   COLOURS←COLOURS[TMP3]
[58]   COLNAMES←COLNAMES[TMP3]
[59]   COLDATA←COLDATA[TMP3]
[60]   ⍝
[61]   FONTCI←(COLNAMESϵ⊂FONTC)/COLOURS
```

```
[62]   ⍝
[63]   ⍝ ADD ONE THE RGB VALUES, BECAUSE WE NEED DENOMINATORS > 0
[64]    RGB←1+⌊/¨RGB[TMP3]
[65]    ⎕←(⍴TMP3),, colours initialized.,
[66]   ⍝
[67]   ⍝ calculate closest colours to the basic colour
[68]   ⍝ squared devaition of proportions between R G B
[69]   ⍝ R:G, R:B, G:B
[70]    LINDEX←,↑(COLNAMES∊⊂LCOLOUR)/COLOURS
[71]    ⌊/(0∊⍴LINDEX)/,⎕←,,basic colour not found. exit.,, ◇
       →ERR,
[72]    BRGB←,↑(COLNAMES∊⊂BCOLOUR)/RGB
[73]    ⌊/(0∊⍴LINDEX)/,⎕←,,line colour not found. exit.,, ◇
       →ERR,
[74]    BACKI←,↑(COLNAMES∊⊂BACKC)/COLOURS
[75]    ⌊/(0∊⍴BACKI)/,⎕←,,background colour index not found.
       exit.,, ◇ →ERR,
[76]   ⍝
[77]    RGB←↑RGB
[78]    SRG←BRGB[1]÷BRGB[2]
[79]    SRB←BRGB[1]÷BRGB[3]
[80]    SGB←BRGB[2]÷BRGB[3]
[81]   ⍝
[82]    RG←RGB[;1]÷RGB[;2]
[83]    RB←RGB[;1]÷RGB[;3]
[84]    GB←RGB[;2]÷RGB[;3]
[85]   ⍝
[86]    COLERR←(((RG-SRG)⋆2)+((RB-SRB)⋆2)+((GB-SGB)⋆2))⋆
       0.5
[87]   ⍝ TAKE ALL COLORS FOR LIGHTENING WITH COLERR < 2
[88]    DENSCOLS←(COLERR=0)/⍳⍴COLERR
[89]    COLOURS←COLOURS[DENSCOLS]
[90]    COLNAMES←COLNAMES[DENSCOLS]
[91]    COLDATA←COLDATA[DENSCOLS]
[92]    RGB←RGB[DENSCOLS;]
[93]    ⎕←,  ,,(⍴NCOLS←⍴DENSCOLS),, colors used,
[94]   ⍝
[95]   ⍝ sort colours according to red value
[96]    CSORT←⍋RGB[;1]
[97]    COLOURS←COLOURS[CSORT]
[98]    COLNAMES←COLNAMES[CSORT]
[99]    COLDATA←COLDATA[CSORT]
[100]   RGB←RGB[CSORT;]
```

```
[101] ⍝
[102] ⍝ now we have colours of same colour but different
      brightness
[103] ⍝
[104] ⍝ draw segments
[105]  D←ORIGIN+(TMP6ρ(DSEG[2]÷2),0)+SEGMENT
[106]  E←ORIGIN+(TMP6ρ0,DSEG[2]÷2)+⨁SEGMENT
[107]  G←ORIGIN+(TMP6ρ(DSEG[2]÷2),DSEG[1])+SEGMENT
[108]  C←ORIGIN+(TMP6ρ(DSEG[1]),(DSEG[2]÷2))+⨁SEGMENT
[109]  B←ORIGIN+(TMP6ρ(DSEG[1]),(DSEG[1]+DSEG[2]÷2))+⨁
      SEGMENT
[110]  A←ORIGIN+(TMP6ρ(DSEG[2]÷2),(2×DSEG[1]))+SEGMENT
[111]  F←ORIGIN+(TMP6ρ0,(DSEG[1]+DSEG[2]÷2))+⨁SEGMENT
[112]  →MENU
[113] ⍝
[114] DISPL:TMP10←XCLEARWIN WINID
[115] ⍝ make a forward pass
[116]  TITLE DATA←CR FILENAME 1
[117]  N_LAYERS←10⊃DATA
[118]  TEMPERATURE←5⊃DATA
[119]  OUTPUTFUNCTION←11⊃DATA
[120]  DATA←TITLE←0ρ0
[121] ⍝ calculate range of outputfunction
[122]  MAXOUT←⌈/OUTPUTFUNCTION,,¯100 100,
[123] ⍝
[124]  TITLE STRUCTURE←CR FILENAME COMPN
[125]  STRUCTURE ERROR←STRUCTURE
[126] ⍝
[127]  (6⊃1⊃STRUCTURE)←TMPF1←((ρTMPF1),1)ρTMPF1←INPUT
[128]  (3⊃2⊃STRUCTURE)←TMPF1
[129]  LN2←1
[130] L2:→(N_LAYERS<LN2←LN2+1)/LC2
[131]  WEIGHTS B_WEIGHTS XJ UI AI XI TEMPERATURE TMP12 TMP12←
      LN2⊃STRUCTURE
[132]  XJ←6⊃(LN2-1)⊃STRUCTURE
[133]  UI←B_WEIGHTS+(WEIGHTS+.×XJ)
[134]  AI←UI
[135]  XI←⌈/OUTPUTFUNCTION,, AI,
[136]  (LN2⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
      TEMPERATURE TMP12 TMP12
[137]  →L2
[138] LC2:WEIGHTS←B_WEIGHTS←XJ←UI←AI←XI←0ρ0
[139] ⍝
```

```
[140]  TMPF1 TMPF1 TMPF1 TMPF1 TMPF1 XI TEMPERATURE TMP12 TMP12←
       N_LAYERS⊃STRUCTURE
[141] ⍝
[142] ⍝ normalize output values of neural network into the range
       0,1
[143]  XI←(XI−MAXOUT[1])÷(MAXOUT[2]−MAXOUT[1])
[144]  XIC←1+⌊(,XI)×(¯1+ρCOLOURS)
[145] ⍝
[146]  (0,COLOURS[XIC[4]])XFPLY D
[147]  (0,COLOURS[XIC[5]])XFPLY E
[148]  (0,COLOURS[XIC[7]])XFPLY G
[149]  (0,COLOURS[XIC[3]])XFPLY C
[150]  (0,COLOURS[XIC[2]])XFPLY B
[151]  (0,COLOURS[XIC[1]])XFPLY A
[152]  (0,COLOURS[XIC[6]])XFPLY F
[153] ⍝
[154]  (0,LINDEX,BACKI,LWIDTH)XPOLY D
[155]  (0,LINDEX,BACKI,LWIDTH)XPOLY E
[156]  (0,LINDEX,BACKI,LWIDTH)XPOLY G
[157]  (0,LINDEX,BACKI,LWIDTH)XPOLY C
[158]  (0,LINDEX,BACKI,LWIDTH)XPOLY B
[159]  (0,LINDEX,BACKI,LWIDTH)XPOLY A
[160]  (0,LINDEX,BACKI,LWIDTH)XPOLY F
[161]
[162] ⍝
[163]  DTEXT←⊂,ITERATIONS: ,,,⍕TITLE
[164]  DTEXT←DTEXT,⊂,INPUT: ,,,⍕INPUT
[165] ⍝ DTEXT←DTEXT,(⊃ERROR[1],⊃ERROR[2]
[166]  TMP9←(FONTI,FONTCI,0)XTEXT(0)(0)(↑DTEXT)
[167]  DTEXT←0ρ0
[168] ⍝
[169] ⍝ create the menu
[170] MENUD:TMP2←WINID(0,SIZE[2])XMENU MTEXT←,input, ,
       iterations, ,exit,
[171]  →(0∈TMP2)/MENUD
[172]  →(MTEXT∈MTEXT[TMP2])/INP,ITER,MENU
[173]  →MENUD
[174] ⍝
[175] INP:TMP2←WINID(0,SIZE[2])XMENU MTEXT←⍕¨(⊂,input,),(
       ⊂¨INPUT),(⊂,exit,)
[176]  →(0∈TMP2)/INP
[177]  →(MTEXT∈MTEXT[TMP2])/INP,CH,CH,CH,CH,DISPL
[178]  →INP
```

```
[179] ⍝
[180] CH:⍋(¯1∈INPUT[TMP2-1])/,INPUT[TMP2-1]←1 ⋄ →INP,
[181]  INPUT[TMP2-1]←¯1
[182]  →INP
[183] ⍝
[184] ITER:FSIZE←CR FILENAME 0
[185]  TITLE STRUCTURE←CR FILENAME COMPN
[186]  TMP2←WINID(0,SIZE[2])XMENU MTEXT←,iterations, ,+, ,
      -,(⍒TITLE),exit,
[187]  →(0∈TMP2)/ITER
[188]  →(MTEXT∈MTEXT[TMP2])/ITER,IPLUS,IMINUS,ITER,DISPL
[189]  →ITER
[190] ⍝
[191] IPLUS:COMPN←(COMPN+1)⌊(FSIZE[2]-1)
[192]  →ITER
[193] ⍝
[194] IMINUS:COMPN←(COMPN-1)⌈2
[195]  →ITER
[196] ⍝
[197] ⍝ create the menu
[198] MENU:TMP10←XCLEARWIN WINID
[199]  (0,FONTCI)XFPLY((⍴LETTER_N)⍴150 200)+10×LETTER_N
[200]  (0,FONTCI)XFPLY((⍴LETTER_N)⍴270 200)+10×LETTER_N
[201]  TMP2←WINID(0,SIZE[2])XMENU MTEXT←,display, ,
      description, ,quit,
[202]  →(0∈TMP2)/MENU
[203]  →(MTEXT∈MTEXT[TMP2])/DISPL,DESCRIPTION,END
[204]  →MENU
[205] ⍝
[206] END:
[207] ⍝ free graphics context
[208]  XFREEGC CONT
[209]  XFREEFONT FONTI
[210] ⍝ close window
[211]  TMP4←XDELETEWIN WINID
[212]  →0
[213] ⍝  assumtions
[214] ⍝
[215] ⍝  -------A--------
[216] ⍝  |          |
[217] ⍝  |          |
[218] ⍝  F          B
[219] ⍝  |          |
```

```
[220] ⍝  |          |
[221] ⍝  --------G---------
[222] ⍝  |          |
[223] ⍝  |          |
[224] ⍝  E          C
[225] ⍝  |          |
[226] ⍝  |          |
[227] ⍝  --------D---------
[228] ⍝
[229] READQUEUE:
[230] ⍝ read event queue
[231]  EQUEUE←XGET_EVENTS
[232] ⍝ get events for our window
[233]  □←EQUEUE←(EQUEUE[;1]∊WINID)≠EQUEUE
[234]  →□LC
[235] ⍝
[236] ERR:
[237] ⍝ exit on error
[238] ⍝ close window
[239]  TMP4←XDELETEWIN WINID
[240]  →0
[241] DESCRIPTION:TMP10←XCLEARWIN WINID
[242] ⍝DATA: LEARNING_RATE RANDRANGE OUTPUTINT ERRSTD TEMERATURE
      S_INPUT S_OUTPUT LAYER_NAMES LAYER_UNITS N_LAYERS
      OUTPUTFUNCTION
[243] ⍝ read dumped file
[244]  FSIZE←CR FILENAME 0
[245]  TITLE DATA←CR FILENAME 1
[246]  DTEXT←⊂(,'The name of the file entry is: ',,(⍕TITLE))
[247]  DTEXT←DTEXT,⊂,'Learning rate :',,(⍕1⊃DATA)
[248]  DTEXT←DTEXT,⊂,'Network initialized with weights between:
      ',,(⍕2⊃DATA)
[249]  DTEXT←DTEXT,⊂,'Output every ',,(⍕3⊃DATA),, iteration(
      s),
[250]  DTEXT←DTEXT,⊂,'maximum of unit error: ',,(⍕4⊃DATA)
[251]  DTEXT←DTEXT,⊂,'temperature: ',,(⍕5⊃DATA)
[252]  DTEXT←DTEXT,⊂,'inputs are: ',,⍕6⊃DATA
[253]  DTEXT←DTEXT,⊂,'desired outputs are: ',,⍕7⊃DATA
[254]  DTEXT←DTEXT,⊂,'layer names: ',,(⍕8⊃DATA)
[255]  DTEXT←DTEXT,⊂,'number of layer units: ',,(⍕9⊃DATA)
[256]  DTEXT←DTEXT,⊂,'number of layers: ',,(⍕10⊃DATA)
[257]  DTEXT←DTEXT,⊂,'the output function is: ',,(⍕11⊃DATA)
[258]  TMP9←(FONTI,FONTCI,0)XTEXT(0)(0)(↑DTEXT)
```

```
[259]  DTEXT←0ρ0
[260]  ⍝ create the menu
[261]  MENUE:TMP2←WINID(0,SIZE[2])XMENU MTEXT←1ρ⊂,exit,
[262]   →(MTEXTϵMTEXT[TMP2])/MENU
[263]   →MENUE
[264]  ⍝
[265]  ⍝
       ▽
```

8.4 APL-Funktionen Zeitreihenanalyse

```
     ▽ Z←TIME5;⎕IO;S_INPUT;S_OUTPUT;I_OUTPUT;INPUT;OUTPUT;
       LEARNING_RATE;U;LAYER_NAMES;LAYER_UNITS;NET;RANDOM;TMP1;
       TMP2;TMP3;TMP4;TMP5;LN1;N_LAYERS;LSC;STRUCTURE;LSP;NOW;
       DSS;IDSS;LN2;WEIGHTS;B_WEIGHTS;XJ;UI;AI;XI;LN3;△WEIGHTS;
       △B_WEIGHTS;DELTA;DELT_WEIGHT;ITCOUNT;RANDRANGE;OUTPUTINT;
       OUTPSTD;OUTPERRSER;ERRSTD;TEMPERATURE;LAST_INP;LAST_
       MOM;COMPU_TIME;FILENAME;FDIR;TEMP_MIN;TEMP_DECREASING;
       0△WEIGHTS;OB△WEIGHTS;MOMENTUM;OUTPUTFUNCTION;TRAINING;
       TEST;WINDOWSIZE_INP;WINDOWSIZE_OUT;TRAINING_INP;VHOST;
       FCONT;MATRIX;TRAINING;TEST;TRC1;TRC2;TRC3;TRANS;TRANSTR
[1]    ⍝ (c) Gottfried RUDORFER 1-1993
[2]    ⍝
[3]    ⍝ INITIALISATIONS
[4]    ⍝
[5]     LEARNING_RATE←0.3
[6]     MOMENTUM←0.5
[7]     RANDRANGE←¯0.2 0.2
[8]     OUTPUTINT←500
[9]     ERRSTD←0.1
[10]    TEMPERATURE←0 ⍝ NO TEMP
[11]    TEMP_DECREASING←0÷5000
[12]    TEMP_MIN←0
[13]    OUTPUTFUNCTION←,EXPFUNCTION, ⍝ OF_SIN TANHFUNCTION
       EXPFUNTION
[14]    ⎕IO←1
[15]    UNCOMPRESSEDFILE←,airline.data,
[16]   ⍝ Get the input from specified file
[17]   ⍝
[18]   ⍝read file
[19]    FCONT←GETFILE UNCOMPRESSEDFILE,ascii,
[20]   ⍝ the file has the following format
```

```
[21]   ⍝ Monthly totals of international airline passengers
[22]   ⍝
[23]    TRC1←TRC2←TRC3←TRANS←0⍴0
[24]    VHOST←⍬,FCONT,[2]((1↑⍴FCONT),1)⍴' '
[25]   ⍝ VHOST←1+VHOST-TRC1←⌊/VHOST
[26]   ⍝ TRANS←TRANS,⊂,VHOST←¯1+VHOST+TRC1,
[27]   ⍝ VHOST←⍟VHOST
[28]   ⍝ TRANS←TRANS,⊂,VHOST←*VHOST,
[29]    VHOST←(VHOST-⌊/VHOST)÷((TRC2←⌈/VHOST)-(TRC3←⌊/
       VHOST))
[30]    TRANS←TRANS,⊂,VHOST←TRC3+VHOST×(TRC2-TRC3),
[31]    TRANS←⍪TRANS
[32]    TRANSTR←(TRANS)(TRC1 TRC2 TRC3)
[33]
[34]    TRAINING←VHOST
[35]    TEST←0⍴0
[36]   ⍝
[37]   ⍝ ⎕SH,'compress ',UNCOMPRESSEDFILE
[38]   ⍝
[39]    S_INPUT←S_OUTPUT←0⍴0
[40]   ⍝
[41]    WINDOWSIZE_INP←10
[42]    WINDOWSIZE_OUT←1
[43]    LAYER_UNITS←WINDOWSIZE_INP,100,WINDOWSIZE_OUT
[44]   ⍝
[45]   ⍝ TRAINING_INP←WINDOWSIZE_OUT↓TRAINING
[46]    TRAINING_INP←TRAINING
[47]   ⍝ DATASETSIZE-WINDOWSIZE_INP
[48]    DSS←(⍴TRAINING_INP)-(WINDOWSIZE_INP+WINDOWSIZE_OUT)
[49]    LAYER_NAMES←'INPUT','HIDDEN','OUTPUT',
[50]    N_LAYERS←⍴LAYER_UNITS
[51]    ITCOUNT←0
[52]    OUTPERRSER←0⍴0
[53]    OB△WEIGHTS←0△WEIGHTS←0⍴0
[54]    LAST_INP←0
[55]    FDIR←'/users/rudorfer/apl/component'
[56]    FILENAME←FDIR,'/',TIME5,
[57]    FINIT FDIR
[58]   ⍝Timing
[59]    COMPU_TIME←⊂⎕AI
[60]   ⍝ Infos about net
[61]    PRE
[62]   ⍝
```

```
[63]    (0ρ0)DUMPFILE,erase,FILENAME
[64]    (0ρ0)DUMPFILE,erase,FILENAME
[65]    ⍝ Dump infos into a file
[66]    (,Header,)(LEARNING_RATE RANDRANGE OUTPUTINT ERRSTD
        TEMPERATURE VHOST S_OUTPUT LAYER_NAMES LAYER_UNITS N_
        LAYERS OUTPUTFUNCTION TRANSTR)CW FILENAME 1
[67]    VHOST←0ρ0
[68]
[69]    ⍝
[70]    ⍝ FORWARD PASS
[71]    ⍝
[72]    STRUCTURE←⊂(⊂,WEIGHTS,),(⊂,BIAS_WEIGHTS,),(⊂,XJ,
        ),(⊂,UI,),(⊂,AI,),(⊂,XI,),(⊂,Temperature,)(⊂,
        0△WEIGHTS,)(⊂,OB△WEIGHTS,)
[73]    ⍝ BUFFER FOR RANDOM VARIABLES
[74]    RANDOM←0ρ0
[75]    ⍝ START WITH FIRST LAYER (NOT INPUT LAYER)
[76]    LN1←1
[77]    ⍝
[78]    L1:→(N_LAYERS<LN1←LN1+1)/LC1A
[79]    ⍝LAYER SIZE IN CURRENT LAYER
[80]    LSC←LAYER_UNITS[LN1]
[81]    ⍝LAYER SIZE IN PREVIOUS LAYER
[82]    LSP←LAYER_UNITS[LN1−1]
[83]    ⍝NUMBER OF WEIGHTS LSP×LSC...WEIGHTS + LSC...BIAS_WEIGHTS
[84]    NOW←LSP×LSC
[85]    ⍝ initialize weights randomly
[86]    ⍝ generate random numbers and assign them to a tempary
        variable
[87]    TMP1←ρRANDOM
[88]    ⍦(NOW>TMP1)/,RANDOM←RANDOM,RAND RANDRANGE,(1000−
        TMP1)⌈(NOW−TMP1),
[89]    STRUCTURE←STRUCTURE,⊂(⊂(LSC,LSP)ρNOW↑RANDOM),(⊂,
        BIAS_WEIGHTS,),(⊂,XJ,),(⊂,UI,),(⊂,AI,),(⊂,XI,),(⊂
        TEMPERATURE),(⊂(LSC,LSP)ρ0),(⊂(LSC,1)ρ0)
[90]    RANDOM←NOW↓RANDOM
[91]    ⍝
[92]    ⍝ initialize bias−weights randomly
[93]    TMP1←ρRANDOM←0ρ0
[94]    ⍦(LSC>TMP1)/,RANDOM←RANDOM,RAND RANDRANGE,(1000−
        TMP1)⌈(LSC−TMP1),
[95]    (2⊃LN1⊃STRUCTURE)←(LSC,1)ρ¯1×(LSC↑RANDOM)
[96]    RANDOM←LSC↓RANDOM
```

```
[97]   ⍝
[98]    →L1
[99]   LC1A:
[100]  ⍝ dump net to file
[101]   ((STRUCTURE)((,NIL,)(,NIL,)(,NIL,)))DUMPFILE,
       Initialize,FILENAME
[102]  ⍝
[103]   IDSS←0
[104]  LC1:
[105]   ITCOUNT←ITCOUNT+1
[106]  ⍝ FORWARD PASS
[107]  LC11:
[108]   TEMPERATURE←TEMPERATURE-TEMP_DECREASING
[109]   ⍋(TEMPERATURE<TEMP_MIN)/,TEMPERATURE←TEMP_MIN,
[110]   IDSS←¯1+(1↑DSS?DSS)
[111]   S_INPUT←WINDOWSIZE_INP↑IDSS↓TRAINING_INP
[112]   S_OUTPUT←WINDOWSIZE_OUT↑(WINDOWSIZE_INP+IDSS)↓
       TRAINING
[113]  ⍝ IDSS←IDSS+1
[114]  ⍝ ⍋(IDSS>DSS)/,IDSS←1,
[115]  ⍝ →(LAST_INP=IDSS)/LC11
[116]  ⍝ LAST_INP←IDSS
[117]  ⍝ INPUT CONSISTS OF OUTPUT VALUES ONLY
[118]   (6⊃1⊃STRUCTURE)←TMP1←((⍴TMP1),1)⍴TMP1←S_INPUT
[119]   (3⊃2⊃STRUCTURE)←TMP1
[120]  ⍝ START WITH FIRST LAYER (NOT INPUT LAYER)
[121]   LN2←1
[122]  ⍝
[123]  L2:→(N_LAYERS<LN2←LN2+1)/LC2
[124]   WEIGHTS B_WEIGHTS XJ UI AI XI TMP1 0△WEIGHTS
       0B△WEIGHTS←LN2⊃STRUCTURE
[125]   XJ←6⊃(LN2-1)⊃STRUCTURE
[126]   UI←B_WEIGHTS+(WEIGHTS+.×XJ)
[127]   AI←UI
[128]   XI←⍋OUTPUTFUNCTION,, AI,
[129]   (LN2⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
       TEMPERATURE 0△WEIGHTS 0B△WEIGHTS
[130]   →L2
[131]  ⍝ BACKWARD-PASS / BACK-PROPAGATION
[132]  ⍝
[133]
[134]  LC2:⍝DISPLAY STRUCTURE
[135]   WEIGHTS B_WEIGHTS XJ UI AI XI TEMPERATURE 0△WEIGHTS
```

```
      OB△WEIGHTS←N_LAYERS⊃STRUCTURE
[136] ⍝ TMP2 IS DESIRED OUTPUT DATA SET
[137]  TMP2←((ρTMP2),1)ρTMP2←S_OUTPUT
[138] ⍝
[139] ⍝ DELTA AND △WEIGHTS ARE NESTED VECTORS
[140]  DELTA←(TMP3←TMP2−XI)×(AI DEVIATION OUTPUTFUNCTION)
[141] ⍝ DELTA←(TMP3←TMP2−XI)×(XI×(1−XI))
[142] ⍝
[143]  OUTPERRSER←OUTPERRSER,⊂−TMP3
[144] ⍝
[145]
[146]  0△WEIGHTS←△WEIGHTS←(LEARNING_RATE×(DELTA+.×⌽XJ))
      +(MOMENTUM×0△WEIGHTS)
[147]  OB△WEIGHTS←△B_WEIGHTS←(LEARNING_RATE×(DELTA×1))+
      (MOMENTUM×OB△WEIGHTS)
[148] ⍝ PART OF FORMULA FOR NEXT LAYER
[149]  DELT_WEIGHT←⌽(⌽DELTA)+.×WEIGHTS
[150] ⍝ UPDATE WEIGHTS
[151]  WEIGHTS←WEIGHTS+△WEIGHTS
[152]  B_WEIGHTS←B_WEIGHTS+△B_WEIGHTS
[153] ⍝
[154]  (N_LAYERS⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
      TEMPERATURE 0△WEIGHTS OB△WEIGHTS
[155] ⍝
[156] ⍝ START WITH LAST LAYER (NOT OUTPUT LAYER)
[157]  LN3←N_LAYERS
[158] ⍝
[159] L3:→(2>LN3←LN3−1)/LC3
[160]  WEIGHTS B_WEIGHTS XJ UI AI XI TEMPERATURE 0△WEIGHTS
      OB△WEIGHTS←LN3⊃STRUCTURE
[161]  DELTA←(AI DEVIATION OUTPUTFUNCTION)×DELT_WEIGHT
[162] ⍝ DELTA←(XI×(1−XI))×DELT_WEIGHT
[163] ⍝ PART OF FORMULA FOR NEXT LAYER
[164]  DELT_WEIGHT←⌽(⌽DELTA)+.×WEIGHTS
[165]  0△WEIGHTS←△WEIGHTS←(LEARNING_RATE×(DELTA+.×⌽XJ))
      +(MOMENTUM×0△WEIGHTS)
[166]  OB△WEIGHTS←△B_WEIGHTS←(LEARNING_RATE×(DELTA×1))+
      (MOMENTUM×OB△WEIGHTS)
[167] ⍝ UPDATE WEIGHTS
[168]  WEIGHTS←WEIGHTS+△WEIGHTS
[169]  B_WEIGHTS←B_WEIGHTS+△B_WEIGHTS
[170]  (LN3⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
      TEMPERATURE 0△WEIGHTS OB△WEIGHTS
```

```
[171]   →L3
[172] LC3:⍝DISPLAY STRUCTURE
[173]   ⍊(0=OUTPUTINT|ITCOUNT)/,CHECKPOINT,
[174]   →LC1
    ▽

    ▽ SHOWTIME5;TMP1;TMP2;TMP3;TMP4;TMP5;TMP6;TMP7;TMP8;BACKC;
      SIZE;TITLE;CONT;COLOURS;HOST;RGB;BCOLOUR;RG;RB;GB;SRG;
      SRB;SGB;BRGB;COLERR;DENSCOLS;CSORT;LCOLOUR;LINDEX;DSEG;
      ORIGIN;A;B;C;D;E;F;G;NCOLS;BACKI;LWIDTH;EQUEUE;WINID;
      MTEXT;FILENAME;TITLE;DATA;FSIZE;DTEXT;TMP9;TMP10;FONT;
      FONTI;FONTC;FONTCI;LASTMENU;TMP11;INPUT;ITERATIONS;COMPN;
      STRUCTURE;TMPF1;LN2;WEIGHTS;B_WEIGHTS;XJ;UI;AI;XI;N_
      LAYERS;TEMPERATURE;ERROR;FDIR;TMP12;OUTPUTFUNCTION;MAXOUT;
      VHOST;LAYER_UNITS;WINDOWSIZE_OUT;WINDOWSIZE_INP;OIA;OIB;
      OIC;XIA;XIB;XIC;XID;NPA;NPB;NPC;WIN_START;DSS;WKX1;WKX2;
      WKX3;WKY1;WKY2;WKY3;EA;EB;EC;TRC1;TRC2;TRC3;TRANS;TRANSTR;
      VHOSTTR;XXI;YYI;OT1;OT2;SCALE
[1]     ⍝DATA: LEARNING_RATE RANDRANGE OUTPUTINT ERRSTD TEMERATURE
        S_INPUT S_OUTPUT LAYER_NAMES LAYER_UNITS N_LAYERS
        OUTPUTFUNCTION
[2]     FDIR←,/users/rudorfer/apl/component,
[3]     FILENAME←FDIR,,/,,,TIME5,
[4]     FINIT FDIR
[5]     ⍝
[6]     HOST←,exaix2:0.0,
[7]     SIZE←1200 1000
[8]     BACKC←,white,
[9]     TITLE←,Time,
[10]    LWIDTH←1
[11]    MTEXT←,display time series, ,description, ,iterations,
        ,quit,
[12]    LASTMENU←1
[13]    FONT←,apl14-bold,
[14]    FONTC←,green,
[15]    COMPN←2
[16]    ⍝ segemet
[17]    ⍝ Basic colour
[18]    BCOLOUR←,black,
[19]    ⍝ Line colour
[20]    LCOLOUR←,red,
[21]    COLNAMES←BCOLOUR LCOLOUR,green, ,blue, ,black, ,red,
[22]    ⍝
```

```
[23]    ,Xfns,⎕SH,,
[24]   ⍝
[25]    TMP1←XOPEN_DISPLAY HOST
[26]   ⍝
[27]    WINID←XCREATEWIN 0 0 0,SIZE,10,(2ρ⊂BACKC),(2ρ⊂TITLE)
[28]   ⍝
[29]    FONTI←XGETFONT FONT
[30]   ⍝ select which types of events are to be repported
[31]    TMP7←WINID XSELECTINPUT+/2*2 3 15 17
[32]   ⍝ set trap for reading event queue (weak interrupt)
[33]   ⍝⋄⎕TRAP←1002 ,E, ,→READQUEUE,
[34]   ⍝
[35]   ⍝ enable event notification
[36]   ⍝TMP8←XINTR 1
[37]   ⍝
[38]    CONT←XCREATEGC
[39]   ⍝
[40]    COLOURS←XGETCOLOUR COLNAMES
[41]   ⍝
[42]    TMP3←(COLOURS≥0)/⍳ρCOLOURS
[43]   ⍝
[44]    COLOURS←COLOURS[TMP3]
[45]    COLNAMES←COLNAMES[TMP3]
[46]   ⍝
[47]    FONTCI←(COLNAMESϵ⊂FONTC)/COLOURS
[48]   ⍝
[49]    ⎕←(⍕ρTMP3),, colours initialized.,
[50]   ⍝
[51]   ⍝
[52]    TITLE DATA←CR FILENAME 1
[53]    N_LAYERS←10⊃DATA
[54]    TEMPERATURE←5⊃DATA
[55]    OUTPUTFUNCTION←11⊃DATA
[56]    LAYER_UNITS←9⊃DATA
[57]    WINDOWSIZE_INP←1↑LAYER_UNITS
[58]    WINDOWSIZE_OUT←¯1↑LAYER_UNITS
[59]    VHOSTTR←VHOST←6⊃DATA
[60]    TRANSTR←12⊃DATA
[61]   ⍝ Size of time series
[62]    DSS←ρVHOST
[63]    DATA←TITLE←0ρ0
[64]    TRANS←1⊃TRANSTR
[65]    TRC1 TRC2 TRC3←2⊃TRANSTR
```

```
[66]    ⍙¨TRANS ⍝ transform data
[67]    YYI←VHOST
[68]    →MENU
[69]   ⍝
[70]   DISPL:TMP10←XCLEARWIN WINID
[71]   ⍝ make a forward pass
[72]    XIC←50
[73]    XID←100
[74]    WIN_START←¯1
[75]    XXI←EA←EB←0⍴0
[76]   ⍝ calculate range of outputfunction
[77]    MAXOUT←⍙OUTPUTFUNCTION,,¯100 100,
[78]   ⍝
[79]    TITLE STRUCTURE←CR FILENAME COMPN
[80]    STRUCTURE ERROR←STRUCTURE
[81]   ⍝
[82]   ⍝
[83]   LO1:→((DSS-(WINDOWSIZE_INP+WINDOWSIZE_OUT))<WIN_
       START←WIN_START+1)/LOC1 ⍝ LOOP OVER ALL VALUES OF TIME
       SERIES
[84]    INPUT←WINDOWSIZE_INP↑WIN_START↓VHOSTTR
[85]    (6⊃1⊃STRUCTURE)←TMPF1←((⍴TMPF1),1)⍴TMPF1←INPUT
[86]    (3⊃2⊃STRUCTURE)←TMPF1
[87]    LN2←1
[88]   L2:→(N_LAYERS<LN2←LN2+1)/LC2
[89]    WEIGHTS B_WEIGHTS XJ UI AI XI TEMPERATURE TMP12 TMP12←
       LN2⊃STRUCTURE
[90]    XJ←6⊃(LN2-1)⊃STRUCTURE
[91]    UI←B_WEIGHTS+(WEIGHTS+.×XJ)
[92]    AI←UI
[93]    XI←⍙OUTPUTFUNCTION,, AI,
[94]    (LN2⊃STRUCTURE)←WEIGHTS B_WEIGHTS XJ UI AI XI
       TEMPERATURE TMP12 TMP12
[95]    →L2
[96]   LC2:WEIGHTS←B_WEIGHTS←XJ←UI←AI←0⍴0
[97]    EA←EA,((1↑SIZE)÷DSS)×((¯1+WIN_START+WINDOWSIZE_
       INP)+⍳WINDOWSIZE_OUT) ⍝ number of data points × vector
       ordinate
[98]    XXI←XXI,,XI
[99]    →LO1
[100]  LOC1:
[101]  ⍝ draw estrimated time series
[102]   VHOST←XXI
```

```
[103]  ⍋¨TRANS
[104]  □←¯10↑XXI←VHOST
[105]  □←¯10↑YYI
[106]  SCALE←⌈/(⌈/XXI)(⌈/YYI)
[107]  EB←XIC+(XXI÷SCALE)×((¯1↑SIZE)-(XIC+XID)) ⍝
       abszisse
[108]  (0,(1↑¯2↑COLOURS),(1↑COLOURS),3)XPOLY EA,[1.5]EB
[109]  ⍝
[110]  TMPF1 TMPF1 TMPF1 TMPF1 TMPF1 XI TEMPERATURE TMP12 TMP12←
       N_LAYERS⊃STRUCTURE
[111]  ⍝
[112]  ⍝ normalize output values of neural network into the range
       0,1
[113]  ⍝ XI←(XI-MAXOUT[1])÷(MAXOUT[2]-MAXOUT[1])
[114]  ⍝
[115]  ⍝ draw whole time series
[116]  XIC←50
[117]  XID←100
[118]  OIA←((1↑SIZE)÷DSS)×(¯1+⍳DSS) ⍝ number of data points
       × vector ordinate
[119]  XIA←XIC+(YYI÷SCALE)×((¯1↑SIZE)-(XIC+XID)) ⍝
       abszisse
[120]  (1,(1↑¯2↑COLOURS),(1↑COLOURS),3)XPOLY OIA,[1.5]XIA
[121]  ⍝ write to disk
[122]  OT1←4⍉⍪XXI
[123]  ((OT1∈,¯,)/OT1)←,-,
[124]  □SH,rm ,,((⊃□SI),,.prog,)
[125]  OT2←OT1 PUTFILE((⊃□SI),,.prog,),ascii,
[126]  OT2←4⍉⍪YYI
[127]  ((OT2∈,¯,)/OT2)←,-,
[128]  □SH,rm ,,((⊃□SI),,.real,)
[129]  OT2←OT1 PUTFILE((⊃□SI),,.real,),ascii,
[130]  OT1←(XXI-(WINDOWSIZE_INP↓YYI))
[131]  □←,size,,⍴YYI
[132]  □←,max,,⌈/YYI
[133]  □←,min,,⌊/YYI
[134]  □←,error ,,(+/OT1⋆2)÷⍴OT1
[135]  OT1←4⍉⍪OT1
[136]  ((OT1∈,¯,)/OT1)←,-,
[137]  □SH,rm ,,((⊃□SI),,.err,)
[138]  OT2←OT1 PUTFILE((⊃□SI),,.err,),ascii,
[139]  ⍝
[140]  DTEXT←⊂,ITERATIONS: ,,,⍉⍪TITLE
```

```
[141]  DTEXT←DTEXT,⊂,INPUT:  ,,,⍒INPUT
[142] ⍝ DTEXT←DTEXT,(⊃ERROR[1]),⊃ERROR[2]
[143]  TMP9←(FONTI,FONTCI,0)XTEXT(0)(0)(↑DTEXT)
[144]  DTEXT←0⍴0
[145] ⍝
[146] ⍝ create the menu
[147] MENUD:TMP2←WINID(0,SIZE[2])XMENU MTEXT←,iterations,  ,
      exit,
[148]  →(0∈TMP2)/MENUD
[149]  →(MTEXT∈MTEXT[TMP2])/ITER,MENU
[150]  →MENUD
[151] ⍝
[152] ITER:FSIZE←CR FILENAME 0
[153]  TITLE STRUCTURE←CR FILENAME COMPN
[154]  TMP2←WINID(0,SIZE[2])XMENU MTEXT←,iterations,  ,+,  ,
      +10x,  ,-,  ,-10x,(⍒TITLE),exit,
[155]  →(0∈TMP2)/ITER
[156]  →(MTEXT∈MTEXT[TMP2])/ITER,IPLUS,IPLUS10,IMINUS,IMINUS10,
      ITER,DISPL
[157]  →ITER
[158] ⍝
[159] IPLUS:COMPN←(COMPN+1)⌊(FSIZE[2]-1)
[160]  →ITER
[161] ⍝
[162] IPLUS10:COMPN←(COMPN+10)⌊(FSIZE[2]-1)
[163]  →ITER
[164] ⍝
[165] IMINUS:COMPN←(COMPN-1)⌈2
[166]  →ITER
[167] ⍝
[168] IMINUS10:COMPN←(COMPN-10)⌈2
[169]  →ITER
[170] ⍝
[171] ⍝ create the menu
[172] MENU:TMP10←XCLEARWIN WINID
[173]  TMP2←WINID(0,SIZE[2])XMENU MTEXT←,display,  ,
      description,  ,quit,
[174]  →(0∈TMP2)/MENU
[175]  →(MTEXT∈MTEXT[TMP2])/DISPL,DESCRIPTION,END
[176]  →MENU
[177] ⍝
[178] END:
[179] ⍝ free graphics context
```

```
[180]  XFREEGC CONT
[181]  XFREEFONT FONTI
[182] ⍝ close window
[183]  TMP4←XDELETEWIN WINID
[184]  →0
[185] ⍝  assumtions
[186] ⍝
[187] READQUEUE:
[188] ⍝ read event queue
[189]  EQUEUE←XGET_EVENTS
[190] ⍝ get events for our window
[191]  □←EQUEUE←(EQUEUE[;1]∈WINID)≠EQUEUE
[192]  →□LC
[193] ⍝
[194] ERR:
[195] ⍝ exit on error
[196] ⍝ close window
[197]  TMP4←XDELETEWIN WINID
[198]  →0
[199] DESCRIPTION:TMP10←XCLEARWIN WINID
[200] ⍝DATA: LEARNING_RATE RANDRANGE OUTPUTINT ERRSTD TEMERATURE
      S_INPUT S_OUTPUT LAYER_NAMES LAYER_UNITS N_LAYERS
      OUTPUTFUNCTION
[201] ⍝ read dumped file
[202]  FSIZE←CR FILENAME 0
[203]  TITLE DATA←CR FILENAME 1
[204]  DTEXT←⊂(,The name of the file entry is: ,,(⍕TITLE))
[205]  DTEXT←DTEXT,⊂,Learning rate :,,(⍕1⊃DATA)
[206]  DTEXT←DTEXT,⊂,Network initialized with weights between:
      ,,(⍕2⊃DATA)
[207]  DTEXT←DTEXT,⊂,Output every ,,(⍕3⊃DATA),, iteration(
      s),
[208]  DTEXT←DTEXT,⊂,maximum of unit error: ,,(⍕4⊃DATA)
[209]  DTEXT←DTEXT,⊂,temperature: ,,(⍕5⊃DATA)
[210]  DTEXT←DTEXT,⊂,inputs are: ,,⍕6⊃DATA
[211]  DTEXT←DTEXT,⊂,desired outputs are: ,,⍕7⊃DATA
[212]  DTEXT←DTEXT,⊂,layer names: ,,(⍕8⊃DATA)
[213]  DTEXT←DTEXT,⊂,number of layer units: ,,(⍕9⊃DATA)
[214]  DTEXT←DTEXT,⊂,number of layers: ,,(⍕10⊃DATA)
[215]  DTEXT←DTEXT,⊂,the output function is: ,,(⍕11⊃DATA)
[216]  TMP9←(FONTI,FONTCI,0)XTEXT(0)(0)(↑DTEXT)
[217]  DTEXT←0⍴0
[218] ⍝ create the menu
```

```
[219] MENUE:TMP2←WINID(0,SIZE[2])XMENU MTEXT←1ρ⊂,exit,
[220]   →(MTEXTϵMTEXT[TMP2])/MENU
[221]   →MENUE
[222] ⍝
[223] ⍝
      ▽
```

8.5 C-Programm XOR Parametervariation

Als Literatur wurde vorallem Darnell [1991] verwendet.

```
/* (c) 10-06-1993 Gottfried RUDORFER (GR)   */
/* xor problem with backpropagation      */
/* rudorfer@wu-wien.ac.at          */
/*                */
#include "back.h"


void getargs(argc, argv, bp)
int argc;
char **argv;                                                    10
BPPARAMS *bp;
{

  const int numargs = 13;
  const char *arguments[] = { "-itnumber", "-outint", "-maxcount", "-etha", "-alpha",
    "-temp", "-scale", "-wdecay", "-enforce", "-debug", "-help",
    "-errorlog", "-weightlog" };
  int index[numargs];
  int arglen, zulen, minlen, found;
  int a, b, c, d, e, f, g;                                      20
  ARGUMENT argu;

  a = 1;
 while ( a < argc ) { /* Schleife ueber alle Elemente des argv */
    /* initialisiere: alle moegl. Argumente treffen auf argv zu */
    for ( f = 0; f < numargs; f++)    index[f] = f;
    found = numargs; /* Anzahl der zutreffenden Argumente */
    b = 0;
    while ( b < found ) { /* Schleife ueber alle Argumente */
      arglen = strlen( argv[a] ); /* Laenge des a-ten argv Elements */      30
      zulen = strlen( arguments[index[b]] ); /* Lange des b-ten Arguments */
      /* wenn Laenge des argv > Lange  Arguments -> passt nicht */
      if ( arglen > zulen ) {
        for ( d = b; d < (found - 1); d++ ) index[d] = index[d + 1];
        found--;
        continue;
      }
```

```
    minlen = zulen; /* gemeinsame Laenge von argv und Argument */
    if ( arglen < zulen ) minlen = arglen;
    e = 0;                                                              40
    for ( c = 0; c < minlen; c++ ) { /* Schleife ueber die Zeichen */
      if ( argv[a][c] != arguments[index[b]][c] ) { /* keine Uebereinstimmung */
        for ( d = b; d < (found - 1); d++ ) index[d] = index[d + 1];
        found--;
        b--;
        e = 1;
        break;
      }
    }
    if (( arglen == zulen) && ( e == 0 )) { /* exakte Uebereinstimmung */  50
      found = 1;
      index[0] = index[b];
      break;
    }
    b++;
  }
  if ( found > 1 ) {
    printf("%s: Argument %s is not unique, %d matches found. exit\n",
      argv[0], argv[a], found);
    printf("%s: Arguments matching found: ", argv[0]);                  60
    for ( g = 0; g < found; g++ ) {
        printf("no:%d=%s    ", g, arguments[index[g]]);
    }
    printf("\n");
    exit(1);
  }

  if ( found == 0 ) {
    printf("%s: Argument %s not understood. exit.\n", argv[0], argv[a]);
    exit(1);                                                            70
  }

  argu = index[0];

  switch( argu ) {
    case ITNUMBER:
      ++a;
      if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
        printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
        exit(1);                                                        80
      }
      bp->itnumber = atol( argv[a] );
      break;
    case OUTINT:
      ++a;
      if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
        printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
        exit(1);
      }
```

```
    bp->outint = atol( argv[a] );
    break;
  case MAXCOUNT:
    ++a;
    if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
      printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
      exit(1);
    }
    bp->maxcount = atol( argv[a] );
    break;
  case ETHA:
    ++a;
    if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
      printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
      exit(1);
    }
    bp->etha = atof( argv[a] );
    break;
  case ALPHA:
    ++a;
    if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
      printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
      exit(1);
    }
    bp->alpha = atof( argv[a] );
    break;
  case TEMP:
    ++a;
    if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
      printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
      exit(1);
    }
    bp->temp = atof( argv[a] );
    printf("bp->temp= %f\n", bp->temp);
    break;
  case SCALE:
    ++a;
    if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
      printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
      exit(1);
    }
    bp->scale = atof( argv[a] );
    break;
  case WDECAY:
    ++a;
    if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
      printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
      exit(1);
    }
    bp->wdecay = atof( argv[a] );
    break;
  case ENFORCE:
```

```
        ++a;
        if ( argc <= a ) { /* Ein Wert zu einer Option fehlt */
          printf("Ein Wert zur Option %s fehlt. exit.\n", arguments[argu]);
          exit(1);
        }
        bp->enforce = atoi( argv[a] );
        break;
      case DEBUG:
        bp->debug = 1;
        break;
      case HELP:
        printf("%s: usage: -itnumber >0 -outint >0 -etha [0.0-1.0]\n",
          argv[0]);
       printf("%s: -alpha [0.0-1.0] -temp ]0.0- -scale ]0.0- \n", argv[0]);
        printf("%s: -wdecay [0.99-1.0] -enforce -debug\n", argv[0]);
        exit(1);
        break;
      case ERRORLOG:
        bp->errorlog = 1;
        break;
      case WEIGHTLOG:
        bp->weightlog = 1;
        break;
      default:
        printf("Unbekannte Option %d\n", argu);
        exit(1);
        break;
    } /* Ende switch */
  a++;
  } /* Ende Schleife ueber alle Elemente des argv */
} /* Ende der Funktion getargs */


void backprop_enforce(bp, progname)
BPPARAMS *bp;
char *progname;
{
  int a, b, i, j, next, count = 0;
  long sinit; /* sinit: init the random number generator with it */
  /*
        w1ij: weight maxtrix between input and hidden,
        w2ij: between hidden and output layer
   */
  double w1ij[3][3], w2ij[3][2];
  double net1[3], net2[3], act1[3], act2[3]; /* netto input values */
  double error; /* output error */
  double errors[4]; /* output error for each input/ouput pair */
  long counts[4]; /* number of each trained input set */
  double totalabserr, probability[4], maxprobability;

  double delt2j[2], delt1j[3]; /* delta  changes */
  double dw2ij[3][2], dw1ij[3][3]; /* weight changes */
```

```
char *w_file = "back-enforce.w_log", *e_file = "back-enforce.e_log",
  *d_file = "back-enforce.3d_log";
FILE *w_fp, *e_fp, *d_fp;
int w_bad, e_bad, d_bad;


const double xj[4][3] = {                                          200
  { 1.0, 0.0, 0.0 }, /* pattern 0 first is bias */
  { 1.0, 1.0, 0.0 }, /* pattern 1 first is bias */
  { 1.0, 0.0, 1.0 }, /* pattern 2 first is bias */
  { 1.0, 1.0, 1.0 }  /* pattern 3 first is bias */
};
const double oj[4][1] = {
  { 0.0 }, /* pattern 0 output */
  { 1.0 }, /* pattern 1 output */
  { 1.0 }, /* pattern 2 output */
  { 0.0 }  /* pattern 3 output */                                  210
};

sinit = time(0); /*  initialize random number generator */
/* srand48(sinit); */
/* bias weights: act1[0] ... bias weight for output layer, act2[0] unused */
act1[0] = 1.0, act2[0] = 1.0;

w_bad = 0;
if ( bp->weightlog == 1) {
  fprintf( stderr, "Gewichtsaenderungen werden in Datei %s gesichert\n",   220
    w_file);
  if ((w_fp = fopen( w_file, "w" )) == NULL ) {
    fprintf( stderr, "Kann die Datei %s nicht oeffnen\n", w_file);
    w_bad = 1;
  }
}

e_bad = 0;
if ( bp->errorlog == 1) {
  fprintf( stderr, "Fehlerverlauf wird in Datei %s gesichert\n", e_file );  230
  if ((e_fp = fopen( e_file, "w" )) == NULL ) {
    fprintf( stderr, "Kann die Datei %s nicht oeffnen\n",  e_file);
    e_bad = 1;
  }
}

d_bad = 0;
if ( bp->errorlog == 1) {
  fprintf( stderr, "Gewichtsaenderungen fuer d-plot in %s.\n", d_file );
  if ((d_fp = fopen( d_file, "a+" )) == NULL ) {                   240
    fprintf( stderr, "Kann die Datei %s nicht oeffnen\n", d_file);
    d_bad = 1;
  }
}
```

```
  if (bp->maxcount < bp->outint ) bp->outint = bp->maxcount;

  if (bp->maxcount < bp->outint ) bp->outint = bp->maxcount;

  for ( a  = 0; a <= 2; a++ ) { /* set weight changes to zero */
    for ( b  = 0; b <= 1; b++ ) {
      dw2ij[a][b] = 0.0;
    }
  }
  for ( a  = 0; a <= 2; a++ ) { /* set weight changes to zero */
    for ( b  = 0; b <= 2; b++ ) {
      dw1ij[a][b] = 0.0;
    }
  }


  if ( bp->temp == 0 ) {
    printf ("temp must be greater than zero. exit.\n");
    exit(1);
  }


  i = 2; /* number of neurons in first layer */
  j = 1; /* number of neurons in second layer */

/* initialize w1ij weights randomly */
  for ( a = 0; a <= 2; a++ ) {
    for ( b = 1; b <= i; b++ ) {
      w1ij[a][b] = (-0.5 + drand48()) * bp->scale;
      if (REPORT_CRET) printf("w1ij[%d][%d]=%f\n", a, b, w1ij[a][b]);
    }
  }

/* initialize w2ij weights randomly */
  for ( a = 0; a <= 2; a++ ) {
    for ( b = 1; b <= j; b++ ) {
      w2ij[a][b] = (-0.5 + drand48()) * bp->scale;
      if REPORT_CRET printf("w2ij[%d][%d]=%f\n", a, b, w2ij[a][b]);
    }
  }

for (count = 1; count <= bp->itnumber; count++) {
  for (next = 0; next < 4; next++) { /* recalculate the errors of the network */
    /* calculate the netto input in the hidden layer */
    net1[1] = 0.0;
    net1[2] = 0.0;
    for ( b = 1; b <= 2; b++ ) {
      for ( a = 0; a <= 2; a++ ) {
        net1[b] += w1ij[a][b] * xj[next][a];
      }
      /* calculate the actiovation of the hidden layer */
      act1[b] = 1 / ( 1 + exp( -1.0 * net1[b] / bp->temp ));
```

```
    }

    /* calculate the netto input in the output layer */
    net2[1] = 0.0;
    for ( b = 1; b <= 1; b++ ) {
      for ( a = 0; a <= 2; a++ ) {
        net2[b] += w2ij[a][b] * act1[a];
      }
      /* calculate the actiovation of the output layer */
      act2[b] = 1 / ( 1 + exp( -1.0 * net2[b] / bp->temp ));
    }
    /* error in output layer */
    error = oj[next][0] - act2[1]; /* desired minus actual value */
    errors[next] = error; /* update the current error for each pattern */
  }
  if REPORT_CRET2 {
    printf("error %f at pattern #%d\n", error, next);
    printf("error[0]%f error[1]%f error[2]%f error[3]%f\n",
      errors[0], errors[1], errors[2], errors[3]);
    printf("count[0]%d count[1]%d count[2]%d count[3]%d\n",
      counts[0], counts[1], counts[2], counts[3]);
    printf("delt2j[1] = %f \n", delt2j[1]);
  }


if REPORT_CRET2 printf("\n********** iteration no. %d\n", count);
/* forward pass */
  totalabserr = 0.0;
  for (a = 0; a < 4; a++) totalabserr += grabs(errors[a]);

  maxprobability = -1.0; /* This is a can't occur value. Ha.*/
  for (a = 0; a < 4; a++) {
    if (( counts[a] == 0 ) || ( bp->enforce == 0 )) {
      next = (int) (drand48() * 4);
      break;
    }
    probability[a] = drand48() * grabs(errors[a]) / totalabserr;
    if REPORT_CRET printf("probability[%d]=%f\n", a, probability[a]);
    if ( probability[a] > maxprobability ) {
      next = a;
      maxprobability = probability[a];
    }
  }
  counts[next]++;

  if REPORT_CRET {
      printf("Input pattern #%d\n", next);
      printf("inp: %f %f %f out: %f\n", xj[next][0], xj[next][1], xj[next][2], oj[next][0]);
  }

  /* calculate the netto input in the hidden layer */
```

```
  net1[1] = 0.0;
  net1[2] = 0.0;
  for ( b = 1; b <= 2; b++ ) {
    for ( a = 0; a <= 2; a++ ) {
      net1[b] += w1ij[a][b] * xj[next][a];
      if REPORT_CRET {
      printf("xj[%d][%d] = %f\n", next, a, xj[next][a]);
      printf("net1[%d] = %f\n", b, net1[b]);
    }
    /* calculate the actiovation of the hidden layer */
    act1[b] = 1 / ( 1 + exp( -1.0 * net1[b] / bp->temp ));
  }
  if REPORT_CRET printf("act1[%d] = %f  net1[%d] = %f\n",
     b, act1[b], b, net1[b]);
  }

  if REPORT_CRET printf("\n");

  /* calculate the netto input in the output layer */
  net2[1] = 0.0;
  for ( b = 1; b <= 1; b++ ) {
    for ( a = 0; a <= 2; a++ ) {
      net2[b] += w2ij[a][b] * act1[a];
      if REPORT_CRET {
        printf("act1[%d] = %f\n", a, act1[a]);
        printf("net2[%d] = %f\n", b, net2[b]);
      }
    }
    /* calculate the actiovation of the output layer */
    act2[b] = 1 / ( 1 + exp( -1.0 * net2[b] / bp->temp ));
    if REPORT_CRET printf("act2[%d] = %f  net2[%d] = %f\n",
       b, act2[b], b, net2[b]);
  }

  if REPORT_CRET {
    printf("\n");
    printf("net1[1] %f, net1[2] %f, act1[1] %f, act1[2] %f, net2[1] %f, act2[1] %f\n",
      net1[1], net1[2], act1[1], act1[2], net2[1], act2[1]);
    printf("\n");
  }

/* backward pass */

  /* error in output layer */
  error = oj[next][0] - act2[1]; /* desired minus actual value */
  errors[next] = error; /* update the current error for each pattern */
  /* multiply with deviation of squashing function */
  delt2j[1] = act2[1] * ( 1.0 - act2[1] ) * error;

  if ( bp->errorlog == 1 ) { /* append errors to a file */
    fprintf(e_fp, "%f %f %f %f\n", errors[0], errors[1], errors[2], errors[3]);
  }
```

```
if (( bp->errorlog == 1 ) && (( count % 10 ) == 0)) { /* append to 3d file */
  fprintf(d_fp, "%d %f %f %f\n",
    count, (totalabserr / 4), (bp->etha), (bp->alpha) );
}

if REPORT_CRET2 {
  printf("error %f at pattern #%d\n", error, next);
  printf("error[0]%f error[1]%f error[2]%f error[3]%f\n",
    errors[0], errors[1], errors[2], errors[3]);
  printf("count[0]%d count[1]%d count[2]%d count[3]%d\n",
    counts[0], counts[1], counts[2], counts[3]);
  printf("delt2j[1] = %f \n", delt2j[1]);
}


/* error in hidden layer */

for ( b = 1; b <= 2; b++ ) {
  delt1j[b] = 0.0;
  for ( a = 1; a <= 1; a++ ) {
    delt1j[b] += delt2j[a] * w2ij[b][a];
  }
  delt1j[b] = act1[b] * ( 1 - act1[b] ) * delt1j[b];
  if REPORT_CRET printf("delt1j[%d] = %f \n", b, delt1j[b]);
}


/* weight delta */
for ( b = 0; b <= 2; b++ ) {
  for ( a = 1; a <= 1; a++ ) {
    if ( bp->weightlog == 1 ) { /* append weights to a file */
      fprintf(w_fp, "%f ", w2ij[b][a]);
    }
    dw2ij[b][a] = bp->etha * delt2j[a] * act1[b] + bp->alpha * dw2ij[b][a];
    w2ij[b][a] += dw2ij[b][a];
    w2ij[b][a] *= bp->wdecay;
    if REPORT_CRET2 printf("w2ij[%d][%d]=%f ", b, a, w2ij[b][a]);
    if REPORT_CRET printf("dw2ij[%d][%d] = %f\n", b, a, dw2ij[b][a]);
  }
}
for ( b = 0; b <= 2; b++ ) {
  for ( a = 1; a <= 2; a++ ) {
    if ( bp->weightlog == 1 ) { /* append weights to a file */
      fprintf(w_fp, "%f ", w1ij[b][a]);
    }
    dw1ij[b][a] = bp->etha * delt1j[a] * xj[next][b] + bp->alpha * dw1ij[b][a];
    w1ij[b][a] += dw1ij[b][a];
    w1ij[b][a] *= bp->wdecay;
    if REPORT_CRET2 printf("w1ij[%d][%d]=%f ", b, a, w1ij[b][a]);
    if REPORT_CRET printf("dw1ij[%d][%d] = %f\n", b, a, dw1ij[b][a]);
  }
```

```
    }
    if ( bp->weightlog == 1 ) { /* end line */
      fprintf(w_fp, "\n");
    }

} /* end main for loop */
printf("\n");

if ( bp->weightlog == 1)
    fclose( w_fp );

if ( bp->errorlog == 1)
    fclose( e_fp );

} /* Ende backprop_enforce */

void main (argc, argv)
int argc;
char **argv;
{
    BPPARAMS bp = { 1000, 100, 1000, 0.1, 0.9, 1.0, 1.0, 1.0, 0 };
    char *progname = argv[0];

    /*  "-itnumber", "-outint", "-maxcount", "-etha", "-alpha",
        "-temp", "-scale", "-wdecay", "-enforce", "-debug" */

    getargs(argc, argv, &bp);
    backprop_enforce(&bp, progname);
} /* Ende main */

/*                  */
/* (c) 10-06-1993 Gottfried RUDORFER (GR)   */
/* xor problem with backpropagation      */
/* rudorfer@wu-wien.ac.at          */
```

```
/* (c) 10-06-1993 Gottfried RUDORFER (GR)   */
/* xor problem with backpropagation      */
/* rudorfer@wu-wien.ac.at          */
/*                  */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <ctype.h>
#include <string.h>

#ifndef TRUE
  #define TRUE 1
#endif

#ifndef FALSE
```

```
  #define FALSE 0
#endif

#ifndef NULL                                                    20
  #define NULL (void *) 0
#endif

#define REPORT_CRET  (((count % bp->outint) == 0) && ( bp->debug ))
#define REPORT_CRET2 ((count % bp->outint) == 0)
#define grabs(x) ((x) >= 0 ? (x) : -(x))

typedef enum {
  ITNUMBER,
  OUTINT,                                                       30
  MAXCOUNT,
  ETHA,
  ALPHA,
  TEMP,
  SCALE,
  WDECAY,
  ENFORCE,
  DEBUG,
  HELP,
  ERRORLOG,                                                     40
  WEIGHTLOG
} ARGUMENT;

typedef struct
{
  long itnumber; /* number of iterations (not epochs!) */
  long outint;  /* each number of it. the network is shown */
  long maxcount; /* maximum number of iterations */
  double etha;  /* learning rate */
  double alpha;  /* momentum term */                            50
  double temp; /* temerature value */
  double scale; /* scaling factor */
  double wdecay; /* weight decay factor */
  int enforce; /* flag for neuron-error controlled pattern presentation */
  int debug; /* debug flag */
  int errorlog; /* log errors of network into a file */
  int weightlog; /* log weightchanges into a file */
} BPPARAMS;
```

Kapitel 9

Verzeichnisse

Literaturverzeichnis

Igor Aleksander and Helen Morton. *An Introduction to Neural Computing.* Chapman and Hall, London, 1990.

Eric B. Baum. On the capabilities of multilayer perceptons. *Journal of Complexity*, 4:193–215, 1988. Academic Press, Inc.

Maureen Caudill and Charles Butler. *Understanding Neural Networks: Computer Explorations*, volume 1 Basic Networks. Massachusetts Institute of Technology, 1992a.

Maureen Caudill and Charles Butler. *Understanding Neural Networks: Computer Explorations*, volume 2 Advanced Networks. Massachusetts Institute of Technology, 1992b.

Kanad Chakraborty, Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. Forecasting the behavior of multivariate time series using neural networks. *Neural Networks*, 5:961–970, 1992. Pargamon Press.

Christopher Chatfield. *The Analysis of Time Series An Introduction.* Champman and Hall, London, 4 edition, 1991.

Peter A. Darnell. *C: A Software Engineering Approach.* Springer, New York, 1991.

Georg Dorffner. *Konnektionismus, Von neuronalen Netzwerken zu einer 'natürlichen' KI.* Leitfäden der Informatik. B. G. Teubner, Stuttgart, 1991.

Dyadic. *Dyalog APL REFERENCE MANUAL.* Dyadic Systems Limited, Hampshire, 1991a.

Dyadic. *Dyalog APL USER GUIDE.* Dyadic Systems Limited, Hampshire, 1991b.

Richard Forsyth, editor. *Learning and distributed memory: getting close to neurons*, London, 1989. Chapman and Hall.

Dan Hammerstrom. Neural networks at work. *IEEE Spectrum*, pages 26–32, Juni 1993.

Donald O. Hebb. The organization of behavior, new york: Wiley, introduction and chapter 4, "the first stage of percepton: growth of the assembly", seiten xi – xix, 60 – 78. In J. A. Anderson and E. Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 43–56, Cambridge, 1988. MIT Press.

Robert Hecht-Nielsen. *Neurocomputing*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

Kevin Knight. Connectionist ideas and algorithms. *Communications of the ACM*, 44(11):59–74, 1990.

Monika Köhle. *Neurale Netzwerke*. Springer-Verlag, Wien, 1990.

Marvin Minsky and Seymour Papert. Percepons. In J. A. Anderson and E. Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 157–169, Cambridge, 1988. MIT Press.

Klaus Neumann. *Operations Research Verfahren, Band III Graphentheorie und Netzplantechnik*. Carl Hanser Verlag, München, 1975.

Erich Prem. Die anwendung neuronaler netzwerke in der betrieblichen unternehmung. *Forschungsbericht des Österreichischen Forschungsinstituts für Artificial Intelligence*, OEFAI-TR-93-22, 1993.

F.J. Radermacher. Neuronale netze and konnektionismus: Einige anmerkungen und hinweise zu anwendungen. *ÖGOR - News*, 2:5–12, 1992. (Extended Abstract eines Vortrages an der TU Wien, 1992).

S. S. Rao. *Optimization; Theory and Applications*. Wiley Eastern Limited, New Delhi, second edition edition, 1990.

Helge Ritter, Thomas Martinetz, and Klaus Schulten. *Neuronale Netze*. Reihe Künstliche Intelligenz. Prof. Dr. Wolfgang Wahlster, Universität Saarbrücken, 1990.

F. Rosenblatt. The percepton: a probabilistic model for information storage and organization in the brain. In J. A. Anderson and E. Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 89–113, Cambridge, 1988. MIT Press.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagting errors. *Nature*, 323:533–536, Oktober 1986a.

David E. Rumelhart, James L. McClelland, and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: Foundations. MIT Press, Cambridge, Massachusetts, 1986b.

Hamady A. Taha. *Operations Research, An Introduction.* MacMillan Publishing Company, New York, 4 edition, 1989.

Hans Henrik Thodenberg. Improving generalization of neural networks through pruning. In *International Journal of Neural Systems*, 4, pages 317–326, 1 1991.

Thomas Williams and Colin Kelley. *GNUPLOT - An Interactive Plotting Program.* Thomas Williams, Colin Kelley, 1993.

Abbildungsverzeichnis

Tabellenverzeichnis