Rüdiger Brause

Adaptive Systeme

Rüdiger Brause

Adaptive Systeme

Eine Einführung in die Neuroinformatik, Fuzzy Systeme,
Evolutionäre und andere biologisch motivierte Algorithmen

Prof. Dr. rer. nat. habil. Rüdiger W. Brause

Von 1970 bis 1978 Studium der Physik und Kybernetik in Saarbrücken und Tübingen mit Diplom-Abschluß zum Thema "Stochastische Mustererkennung", 1983 Promotion mit einer Arbeit zum Thema "Fehlertoleranz in verteilten Systemen". 1985 Akad. Oberrat an der Universität Frankfurt a. M. im Fachbereich Informatik mit dem Arbeitsgebiet "Neuroinformatik und Parallele Systeme", über das er sich 1993 habilitierte. Seit 2005 Professor für Angewandte Informatik.

Er leitet eine Forschungsgruppe „Adaptive Systemarchitektur“.

Rüdiger Brause
Fachbereich Informatik und Mathematik
J. W. Goethe-Universität Frankfurt am Main
D-60054 Frankfurt

R_Brause@informatik.uni-frankfurt.de
http://www.informatik.uni-frankfurt.de/asa/

Dies ist die überarbeitete und ergänzte Version der Monographie
„Neuronale Netze“, 2. Aufl., Teubner Verlag, Stuttgart 1995, ISBN 3-519-12247-2

Vorwort

Dieses Buch handelt von "Adaptiven Systemen“ und Neuronalen Netzen" - aber was ist das? Im Unterschied zu anderen, fest abgegrenzten und etablierten Gebieten zeigen die englischen Bezeichnungen *soft computing, neural nets, artificial neural networks, connectionism, computational neuroscience* und dergleichen mehr die Vielfalt der Zugänge und Anwendungen eines Gebiets, das nach einem jahrelangen Dornröschenschlaf gerade wieder dem Jungbrunnen entsteigt und deshalb sehr schwer mit einer Schablone abzugrenzen und zu definieren ist. Für den Neurologen und Biologen ist es die theoretische Systematik, mit der er seine Ergebnisse ordnen kann; für den Experimentalpsychologen die mikroskopischen Modelle, die seine Experimente über menschliche Informationsverarbeitung erklären können. Physiker können darin neue Anwendungen physikalischer Methoden und atomarer Modelle von wechselwirkenden Atomen sehen, Nachrichten-Ingenieure hoffen auf schnelle Echtzeitnetzwerke und Informatiker vermuten darin bereits die neue, massiv parallele, ultraschnelle und intelligente Computergeneration, die endlich das Versprechen der "Künstlichen Intelligenz" einlöst.

Dieses Buch, das auf regelmäßigen Vorlesungen an der Universität Frankfurt beruht, soll dazu beitragen, das Verständnis für die tatsächlichen, aktuellen Möglichkeiten dieses Gebiets zu vertiefen.

Dazu werden anfangs kurz die wichtigsten Grundlagen und Konzepte aus den wichtigsten, beteiligten Gebieten wie der Biologie, der Mustererkennung und der Statistik referiert, um beim Leser ohne spezielles Vorwissen ein besseres Grundverständnis der präsentierten Modelle neuronaler Netze zu erreichen. Obwohl dieses Buch eine Einführung in das Gebiet darstellt, sind trotzdem über das rein intuitive Verständnis hinaus auch konkrete Formeln, Lernregeln und Algorithmen enthalten, um dem Leser einen konkreten Vergleich zwischen den verschiedenen Ansätzen zu ermöglichen und ihm/ihr die Mittel in die Hand zu geben, ein konventionelles Modell für ein gegebenes Problem passend abzuwandeln. Sehr allgemeines mathematisches Grundlagenwissen um Vektoren, Matrizen und Eigenvektoren, soweit es sich um Stoff aus der Anfängervorlesung handelt, mußte aber leider aus Platzgründen ausgespart bleiben.

Das Buch beschränkt sich dann im zweiten und dritten Kapitel auf wenige, grundlegende, immer wieder zitierte Inhalte und Arbeiten von neuronalen Netzen, die als Grundpfeiler des Gebäudes dienen, und ermöglicht so dem Leser, die Fülle der neu entstehenden Variationen und Anwendungen besser einzuschätzen. Trotzdem sind auch in den folgenden Kapiteln die relativ neuen Verbindungen neuronaler Netze zu deterministischem Chaos und evolutionären Algorithmen enthalten. Zum Abschluß wird noch kurz auf die verschiedenen Hardwarekonfigurationen und die existierenden Programmiersprachen und -systeme zur Simulation neuronaler Netze eingegangen.

Der Schwerpunkt des Buches liegt damit im Zusammenfassen und Ordnen einer Breite von Ansätzen, Modellen und Anwendungen unter wenigen, klaren Aspekten wie Netzwerkarchitektur (feedforward und feedback Netze) und Informationsverarbeitung (informationsoptimale Schichten), die sich wie ein roter Faden durch die Kapitel ziehen; für eine vertiefendes Studium sind entsprechende Literaturhinweise eingearbeitet. Ich hoffe, damit nicht nur für Informatiker den Einstieg in das Gebiet der neuronalen Netze erleichtert zu haben.

Das Fachgebiet des *soft computing*, insbesondere der neuronalen Netze und evolutionären Algorithmen, steht nicht still**.** Deshalb war es mir ein besonderes Anliegen, viele wichtige Sachthemen neu aufzunehmen und andere wegzulassen, die sich als weniger wichtig für das Grundverständnis herausgestellt haben. Auch eine neue Gliederung soll dafür sorgen, dass die didaktische Balance zwischen Übersicht und Detaillierung zu Gunsten eines besseren Verständnisses neu austariert wurde.

Rüdiger Brause

Inhaltsverzeichnis

Notation

$\mathbf{A}^T, \mathbf{w}^T$	Transponierte der Matrix **A** bzw. des Spaltenvektors **w**
$\mathbf{a}^T\mathbf{b}$	Skalarprodukt $\Sigma_i a_i b_i$ (inneres Produkt der Spaltenvektoren **a** und **b**)
$\mathbf{a}\mathbf{b}^T$	Matrixprodukt $\mathbf{A}=(a_i b_j)$ (äußeres Produkt der Spaltenvektoren **a** und **b**)
$\mathbf{x}$	Spaltenvektor (Muster) der Eingabewerte= $(x_1, \ldots, x_n)^T$
$\mathbf{x}^t$	t-tes Eingabemuster
$\mathbf{y}$	Spaltenvektor der Ausgabewerte $= (y_1, \ldots, y_m)^T$
$\mathbf{z}$	Spaltenvektor der Aktivitätswerte $= (z_1, \ldots, z_m)^T$
$\mathbf{w}_i$	Spaltenvektor der Gewichte zur Einheit i $= (w_{i1}, \ldots, w_{in})^T$
$\mathbf{W}$	$= (w_{ij})$ Matrix der Gewichte von Einheit *j* zu Einheit *i*
S(z)	Ausgabefunktion (*squashing function*)
$\mathbf{S}$	Zustand $(S(z_1), \ldots, S(z_m))^T$ der Ausgabe bzw. "Zustand des Systems"
s_i	Schwellwert (*threshold*) von Einheit *i*
N	Zahl der Muster $\mathbf{x}^1 \ldots \mathbf{x}^N$
M	Zahl der Klassen
ω_k	Ereignis "Klasse k liegt vor "
Ω_k	Menge aller Muster einer Klasse k
d(x,y)	Abstands- oder Fehlerfunktion zwischen x und y
t	Zeit, diskret oder kontinuierlich
$\langle f(x)\rangle_x$	Erwartungswert von f(x) bezüglich aller möglichen Werte von x. Ist $\langle f\rangle = 0$, so heißt f *zentriert*.
R	Zielfunktion (*objective function;* z.B. Risiko- bzw. Straffunktion, Fehlerfunktion, Energie E)
P(x)	Wahrscheinlichkeit, mit der das Ereignis x auftritt
p(x)	Wahrscheinlichkeitsdichte der Ereignisse {x}
I(x)	Information eines Ereignisses x $:= - \ln P(x)$
H(x)	Entropie oder erwartete Information $\langle I(x)\rangle_x$ einer Nachrichtenquelle x
$\mathbf{A}$	Erwartungswert der Matrix der **A**utokorrelation $= (\langle x_i x_j\rangle) = \langle \mathbf{x}\mathbf{x}^T\rangle$
$\mathbf{C}$	Erwartungswert der Matrix der **C**ovarianz $= (\langle (x_i - \langle x_i\rangle)(x_j - \langle x_j\rangle)\rangle) = \langle (\mathbf{x} - \langle \mathbf{x}\rangle)(\mathbf{x} - \langle \mathbf{x}\rangle)^T\rangle$
$\mathbf{e}_k$	k-te Eigenvektor
λ_k	Eigenwert von $\mathbf{e}_k$
δ_{ik}	Kronecker delta; hat den Wert eins, wenn i = k, sonst null.

Einleitung

Vierzig Jahre, nachdem John von Neumann das Konzept eines universellen, programmgesteuerten Automaten charakterisierte, setzt sich nun in der Informatik die Erkenntnis durch, daß sequentiell arbeitende Rechner für manche Probleme zu langsam arbeiten. Mit vielen, parallel arbeitenden Prozessoren versucht man heutzutage in Multiprozessoranlagen, den "von Neumann-Flaschenhals" zu umgehen. Dabei ergeben sich eine Menge neuer Probleme: Die Aktivität der Prozessoren muß synchronisiert werden, die Daten müssen effektiv verteilt werden und "Knoten" im Datenfluß (*hot spots*) zwischen den Prozessoren und dem Speicher müssen vermieden werden. Dazu werden Mechanismen benötigt, um auftretende Defekte in ihren Auswirkungen zu erfassen und zu kompensieren, das Gesamtsystem zu rekonfigurieren und alle Systemdaten zu aktualisieren. Bedenken wir noch zusätzlich die Schwierigkeiten, die mit einer parallelen Programmierung der eigentlichen Probleme verbunden sind, so können wir uns nur wundern, wieso wir Menschen "im Handumdrehen" und "ganz natürlich" Leistungen erbringen können, die mit den heutigen Rechnern bisher nicht nachvollziehbar waren.

Betrachten wir beispielsweise die Probleme der "künstlichen Intelligenz", besser "wissensbasierte Datenverarbeitung" genannt, so hinken die heutigen Systeme zum *Sehen, Hören und Bewegen* hoffnungslos der menschlichen Realität hinterher. Weder in der Bildverarbeitung und -erkennung, noch in der Spracherkennung oder in der Robotersteuerung reichen die Systeme in Punkto Schnelligkeit (*real-time*), Effektivität und Flexibilität (*Lernen*) oder Fehlertoleranz an die menschlichen Leistungen heran. Auch die Expertensysteme, auf denen vor einigen Jahren viele Hoffnungen ruhten, stagnieren in ihrer Entwicklung: Durch die Schwierigkeit, alles benötigte Wissen umständlich erst vorher per Hand eingeben zu müssen und die aber dennoch weiterhin vorhandenen Unterschiede zu menschlichen Expertenleistungen bleibt das Einsatzgebiet in der Praxis auf gut abgegrenzte, einfache Wissensgebiete beschränkt.

Im Unterschied zu den Informatikern, die erst seit wenigen Jahren Systeme von wenigen, parallel arbeitenden Prozessoren untersuchen, beschäftigen sich die Neurobiologen bereits seit über 40 Jahren mit Theorien, die ein massiv parallel arbeitendes Gebilde erklären wollen: unser menschliches Gehirn. Hier läßt sich ein funktionierendes System vorführen, das mit 10^{10} Prozessoren ohne Programmierungsprobleme, Synchronisations-Deadlocks, Scheduling-Probleme und

OSI-Protokolle erstaunliche Leistungen vollbringt. Dabei haben die einzelnen biologischen Prozessoren nicht nur eine 100 000 mal kleinere Taktfrequenz als unsere modernen Mikroprozessoren, sondern das System verkraftet laufend den Ausfall von einzelnen Elementen, ohne daß dies nach außen sichtbar wird.

Der Ansatz, nun direkt die Funktionen des menschlichen Gehirns zu modellieren, weckt dabei viele Hoffnungen. Der Gedanke, sich vielleicht dabei selbst besser zu verstehen, einen kleinen Homunkulus oder darüber hinaus eine großartige, neue Intelligenz zu erschaffen, fasziniert viele Menschen und bringt sie dazu, sich mit diesem Gebiet zu beschäftigen. Auch die Spekulationen darüber, was für eine Maschine wir mit unseren jetzigen schnellen Materialtechniken und hohen Taktraten bauen könnten, wenn wir nur die Prinzipien verstehen würden, nach denen die menschliche Intelligenz funktioniert, sind faszinierend.

Bei all diesen Motivationen sollten wir aber eines nicht ignorieren: die heutige Realität der neuronalen Netze sieht anders aus. Das Anspruchsvollste, was wir zurzeit mit neuronalen Netzen teilweise modellieren können, sind die menschlichen Peripherieleistungen: Bilderkennung, Spracherkennung und Motoriksteuerung. Dies ist zweifelsohne nicht wenig, ja geradezu revolutionär; aber von den "intelligenten" Funktionen in unserem Kopf sind wir noch sehr, sehr weit entfernt.

Ein wichtiger Schritt auf dem Weg dahin besteht in der systematischen Erforschung der möglichen Netzwerkarchitekturen. Genauso, wie es mehr oder weniger intelligente Menschen trotz sehr ähnlicher Gehirnstrukturen gibt, genauso können auch sehr ähnliche künstliche, neuronale Netze unterschiedliche Leistungen erbringen. Im Unterschied zur Natur, wo ab und zu durch Zufall eine besonders günstige Kombination der Parameter der Gehirnentwicklung und -architektur musische, mathematische oder sprachliche Genies hervorbringen kann, müssen wir unsere künstlichen neuronalen Netze selbst optimal gestalten. Es ist deshalb sehr wichtig, über eine naive Beschäftigung mit neuronalen Netzen hinaus Mittel und Methoden zu entwickeln, um kritisch und rational die Netze zu gestalten und zu benutzen.

Ich hoffe, daß auch eine derart eingeschränkte und bescheidenere Motivation, die in diesem Buch vermittelt wird, den Spaß an einem faszinierendem Thema erhalten kann.

1 Grundlagen

Der Traum von einer Maschine, die alle Arbeit für Menschen verrichtet, ist immer noch ein Wunschtraum geblieben. Auch im Computerzeitalter gibt es noch keine intelligente Maschinen die mitdenken können. Eine wichtige Initiative besteht deshalb darin, Maschinen zu schaffen, die nicht starr ein genau festgelegtes Programm ausführen, sondern mit der Ausführung sich an veränderte Umstände anpassen können. Die Anpassungsfähigkeit solcher Systeme ist dabei vielfältig und hängt der Anwendung ab. Die wichtigsten Arten sind

- *Adaptive Schätzung von Prozessparametern*
  Es gibt in der Industrie viele nicht-lineare chemische und physikalisch-technische Reaktionen, deren Optimierung an der fehlenden Kenntnis der gegenseitigen Parameterabhängigkeiten scheitert. Hier werden adaptive Methoden für die Produktionsoptimierung erfolgreich eingesetzt.
- *Adaptive Kontrolle und Regelung*
  Im Unterschied zur klassischen Regelung bietet adaptive Kontrolle die Möglichkeit, die Regelungsparameter an die jeweiligen Betriebsbedingungen anzupassen. Dies ermöglicht Landekontrollsysteme und Roboterkontrolle, deren Genauigkeit und Flexibilität sonst nicht möglich wäre.
- *Adaptive Klassifikation*
  Ein wichtiges Anwendungsgebiet adaptiver Techniken ist die gelernte Einordnung von Objekten in Kategorien oder Klassen. Dies wird verwendet auf den Gebieten der Qualitätskontrolle (defekt, intakt), medizinische Diagnose (krank, gesund), Bonitätsprüfung bei Banken usw.

Die Anwendungsarten adaptiver Systeme sind dabei vielfältig. Wichtig sind sie für

- *Echtzeitreaktionen*
  Schnelle, gelernte Reaktionen sind in einer Echtzeitumgebung besonders wichtig. Dies sind Anwendungen in Stahlwalzstraßen zur nichtlinearen Steuerung der Walzdicke, reaktionsschnelle Flugzeugsteuerung, usw.

- *Analytisch unbekannte Abhängigkeiten*
  Viele komplexe chemische Reaktionen sind durch ihre nichtlineare Charakteristik schwer beherrschbar. Gerade bei unbekannten Parametern und Abhängigkeiten ist eine Verlaufsschätzung und Vorhersage schwierig und muss erst auf Basis bekannter Messwerte trainiert werden. Dies betrifft beispielsweise die Polymerchemie, DNA-Schätzungen, usw.
- *Analytisch nicht zugängige Abhängigkeiten*
  Es gibt Abhängigkeiten, die auf schwer oder gar nicht messbaren Messungen beruhen. Ein praktisches Beispiel dafür sind psychische Faktoren für die Fertigung von Automobilgetrieben. Manche Getriebe klingen defekt, obwohl sie technisch intakt sind, und werden deshalb von den Kunde reklamiert. Wie soll die Qualitätskontrolle solche Getriebe aussortieren? Ein adaptives System wurde auf das Klangbild von reklamierten Getrieben trainiert, so dass nun die Getriebe getestet werden können, ohne dass man genau weiß, wie das Urteil zustande kommt.
  Eine ähnliche Lage ergibt sich bei ergebnisverändernden Messungen, etwa bei Sensoren, die auf die Messgröße bei der Messung rückwirken. Auch hier muss man lernen, aus den Daten auf die „echten“ Messwerte zu schließen, ohne die komplexe Rückwirkung kennen zu können.
- *Analytisch nur unter großem Aufwand bearbeitbare, hochdimensionale Gesetzmäßigkeiten*
  Immer dann, wenn sehr viele Wechselwirkungen bei vielen Eingabevariablen auftreten, hat der Mensch Probleme, diese Abhängigkeiten richtig zu erfassen. Dazu gehören Wechselkursabhängigkeiten oder Aktienkursanalysen, chemische Reaktionen in biologischen Zellen, Industrieprozesse mit vielen Sensoren usw. Adaptive Systeme können sich an diese Gesetzmäßigkeiten automatisch anpassen und so Dinge erfassen, die sonst schwer modellierbar sind.

Die beste adaptive System, das wir kennen ist unser Gehirn; als Vorbild ist es unübertroffen. Betrachten wir deshalb zunächst die Bausteine des Gehirns, die Neuronen, näher und versuchen, sie zu modellieren um ähnliche Leistungen für unsere künstliche Systeme zu erhalten.

1.1 Biologischer Kontext

Betrachten wir zunächst unser Vorbild, das "menschliches Gehirn", etwas näher.

Das menschliche Gehirn ist ein Gebilde von rund 10^{10} Nervenzellen, das einige Regelmäßigkeiten aufweist. Als erstes fällt die Spiegelsymmetrie auf, mit der das Gehirn in zwei Hälften geteilt ist. Verbunden sind die beiden Hälften mit einer Brücke aus Nervenfasern, dem *corpus callosum*. Trennt man diese Brücke auf, so können beide Gehirn-

teile unabhängig voneinander weiterarbeiten, allerdings mit gewissen, sehr speziellen Einschränkungen.

1.1.1 Gehirnfunktionen und Gehirnstruktur

Früher ordnete man menschliche Tugenden und Laster wie Ordnungsliebe und Neidsucht direkt einzelnen Gehirnteilen zu. Inzwischen weiß man aber, das dies nicht so möglich ist. K.S. Lashley, ein Neuropsychologe aus Harvard, versuchte Ende der vierziger Jahre, die Vorstellung des Gehirns als "Telefonvermittlungszentrale" zwischen eingehenden sensorischen Signalen und ausgehenden motorischen Signalen experimentell zu verifizieren. Seine Ergebnisse waren ziemlich entmutigend: weder fand er lokalisierte Reflexbögen zwischen Großhirn und Muskeln, noch konnte er Lernergebnisse und damit Erinnerungen experimentell im Gehirn seiner Versuchstiere lokalisieren [LASH 50].

Trotzdem sind einige, grobe Funktionszuordnungen möglich, die in einer Seitenansicht eines Gehirns (s. Abb. 1.1) mit unterschiedlichen Mustern gekennzeichnet sind. Beispiele dafür sind der Motorkortex (Areal3) und der Somato-sensorische Cortex (Areal 2), die sich entlang einer großen Furche hinziehen, sowie das visuelle Zentrum (Areale 17, 18, 19), das rechts nochmals ausführlicher im Aufschnitt an der rechten der beiden Gehirnhälften zu sehen ist.

Außer den großen Strukturen ("Lappen") bemerkt man viele kleinere Windungen des Gehirns. Man kann es sich wie ein flaches Tuch einer gewissen Dicke vorstellen, das zusammengeknüllt in einem engen Raum untergebracht ist. Dabei hat das Tuch, die Gehirnrinde, trotz seiner geringen Dicke noch eine geschichtete Struktur. Grob lässt sich die Gehirnrinde in zwei Schichten einteilen, die *grau* aussehenden Nervenzellen und der *weiße* Teil, in dem die Nervenfortsätze als kurz- und weitreichendes Verbindungsnetzwerk ("Kabelbaum") die Aktivierungsleitung sicherstellt.

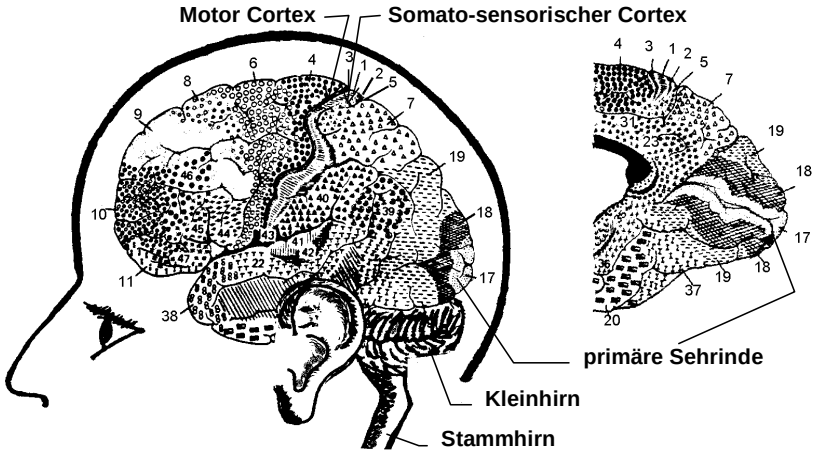


Abb. 1.1 *Gehirnareale und -funktionen (nach [BROD09])*

In einer feineren Aufteilung kann man weitere Einzelschichten unterscheiden, die in der folgenden Abbildung mit römischen Zahlen versehen sind. Je nach Anfärbemethode ("Golgi", "Nissl", "Weigert") sind je Schnitt unterschiedliche Strukturen zu sehen.

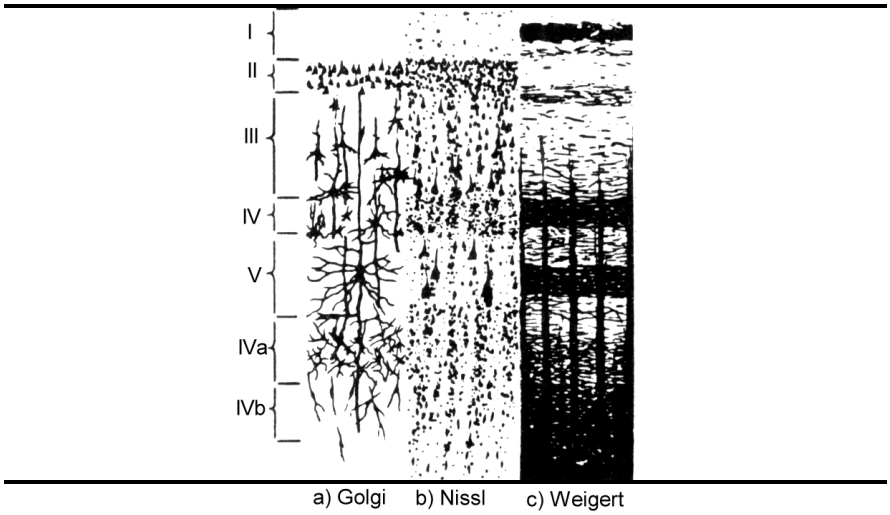


Abb. 1.2 *Schichten der Gehirnrinde und Darstellungsarten (nach [BROD09])*

1.1.2 Biologische Neuronen

Es gibt eine Vielzahl von Typen von Nervenzellen im Gehirn, die alle unterschiedliche Aufgaben erfüllen. In Abb. 1.3 ist ein Schnitt zu sehen, der von dem bekannten Physiologen Cajal mit der Golgi-Anfärbemethode hergestellt wurde. Bedingt durch die Methode, sind aus der Fülle der tatsächlich vorhandenen Neuronen nur einige wenige schwarz gefärbt sichtbar.

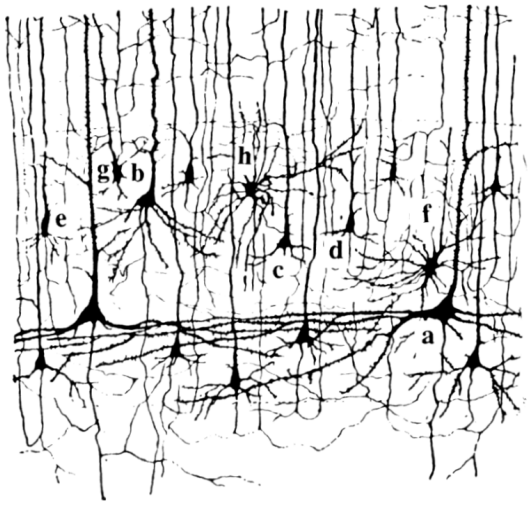


Abb. 1.3 *Biologische Neuronen (nach [CAJ55])*

Zwei Arten wollen wir genauer betrachten: die relativ großen, wie eine Pyramide aussehenden Zellen a,...,e (*Pyramidenzellen*), die sehr häufig im Gehirn vorkommen, und eine kleinere, sternförmig aussehende Sorte f,h (*Stern- oder Gliazellen*). Nach neueren Erkenntnissen gibt es eine Arbeitsteilung zwischen beiden Zelltypen: Die Pyramidenzellen verarbeiten die elektrischen Impulse und die Sternzellen sichern dabei die Stoffwechselversorgung. Durch Ausstülpungen der Sternzellen sowohl zu den Pyramidenzellen als auch zu den Blutgefäßen sind die Sternzellen Mittler zwischen Blutsystem und Pyramidenzellen (Blut-Hirn-Schranke!); außerdem tragen sie durch die Vorgabe der Wachstumsrichtungen entscheidend zur Entwicklung des Gehirns bei [KIM89]. In der folgenden Abbildung ist eine typische Pyramidenzelle gezeigt.

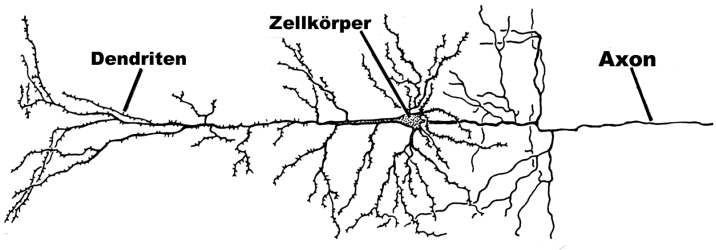


Abb. 1.4 *Eine Pyramidenzelle (nach [CAJ55])*

Die kleinen, astartigen, mit stachelartigen Stellen besetzen Auswüchse der Nervenzellen heißen *Dendriten* und leiten alle elektrische Erregung, die sie erhalten, an den eigentlichen Zellkörper (*Soma*) weiter. Überschreitet die Erregung (interne, elektrische Spannung) einen bestimmten Grenzwert, so entlädt sich die Spannung. Diese rasche Spannungsänderung bewirkt ebenfalls ein Zusammenbrechen einer durch molekulare Mechanismen auf einem dicken Zellfortsatz, dem *Axon*, entstandenen Spannung; der Impuls pflanzt sich vom Zellkörper auf dem Axon bis in die entferntesten Verzweigungen fort. Axone und benachbarte Dendriten anderer Neurone können aufeinander zuwachsen und elektro-chemische Kontaktstellen (*Synapsen*) bilden. Der Informationsfluß der Nervenzellen geht normalerweise vom Zellkern über das Axon eines Neurons durch die Synapsen auf die Dendriten zum Zellkern des anderen Neurons. Bidirektionale, rein elektrische Synapsen sind zwar bekannt; die chemischen Synapsen sind aber die Regel.

Kodierung der Information

Der oben beschriebene Vorgang kann mit Mikroelektroden elektrisch gemessen werden. In der folgenden Abb. 1.5 ist dies an einem Beispiel verdeutlicht. Eine Elektrode wird in den Zellkörper eingesteckt und ein Reizstrom für eine kurze Zeit eingeschaltet (Abbildung links). Am Axon des Neurons bewirkt dies aber nur eine Potentialerhöhung. Überschreitet dagegen der Reiz geringfügig einen Schwellwert, so folgen periodische Entladungen (*Aktionspotentiale* oder *Spikes*), deren Größe unabhängig von der Reizstromstärke ist (Abbildung rechts).

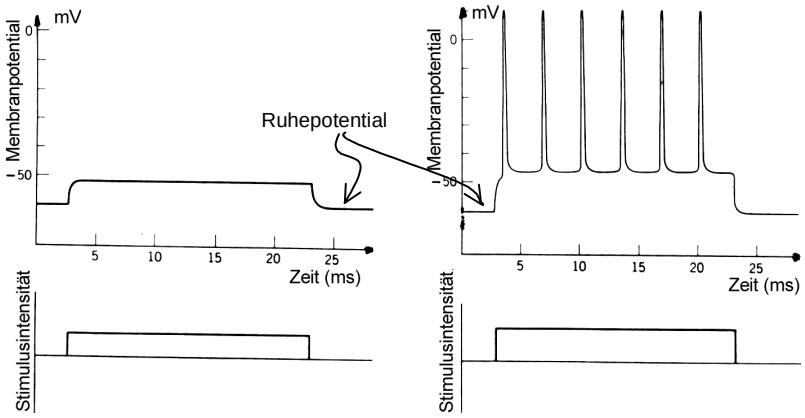


Abb. 1.5 *Erregung einer Nervenzelle (nach [STEV66])*

Obwohl das resultierende Axonsignal damit von Natur aus binär ist, lässt sich auch nicht-binäre Information darin kodieren. Betrachten wir dazu in der folgenden Abbildung eine leichte Absenkung in der Reizstromstärke oberhalb der Schwelle. Die Absenkung bewirkt eine Verkleinerung der Impulsfrequenz der Spikes bzw. eine Vergrößerung der Zeitabstände zwischen den Spikes. Nehmen wir für jeden Spike eine Einheitsladung an, die vom Axon zum Dendriten transportiert wird, so ergibt sich die zeitgemittelte Summe am Dendriten analog der Reizstromstärke.

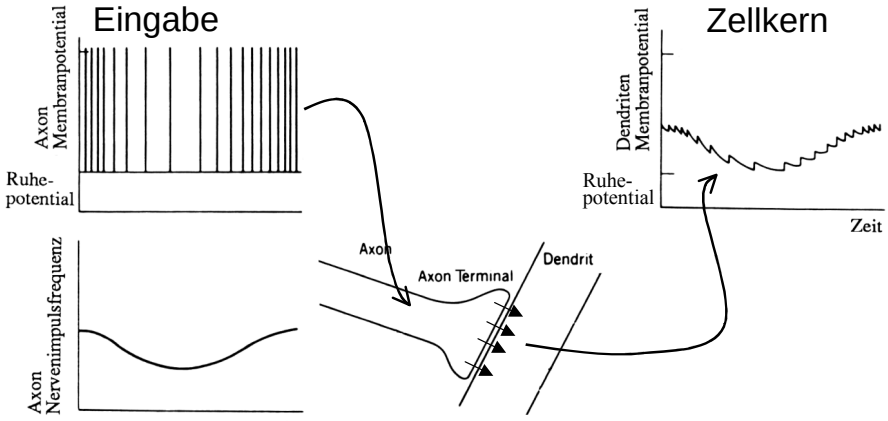


Abb. 1.6 *Frequenzmodulierung und Dekodierung (nach [STEV66])*

Wie wir aus der nächsten Abbildung entnehmen können, ist oberhalb eines Schwellwertes die Frequenzmodulation in weiten Grenzen proportional zur Stärke des angelegten Reizes; hier im Beispiel beim elektrischen Reiz an der Nervenzelle einer Krabbe.

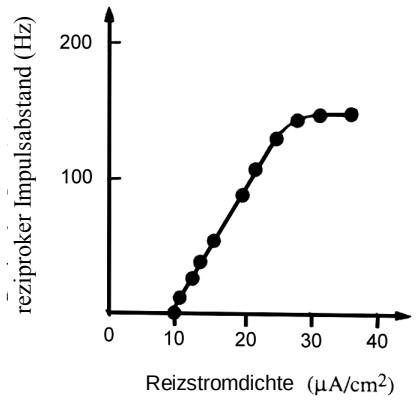


Abb. 1.7 *Die Ausgabefunktion einer Nervenzelle (nach [CHAP66])*

Bei hohen Reizstärken geht die Zellaktivität in eine *Sättigung* über, die durch eine minimal ca. 1 ms lange, nötige Regenerationszeit nach einem Aktionspotential bestimmt wird. Damit ist die maximale "Taktfrequenz" der Informationsverarbeitung auf einen für Mikroprozessoren sehr niedrigen Wert von ca 1 KHz begrenzt.
Allerdings lassen sich auch andere Formen der Kodierung denken. Betrachten wir beispielsweise die drei verschiedenen Impulsarten a), b) und c) in der Abb. 1.8

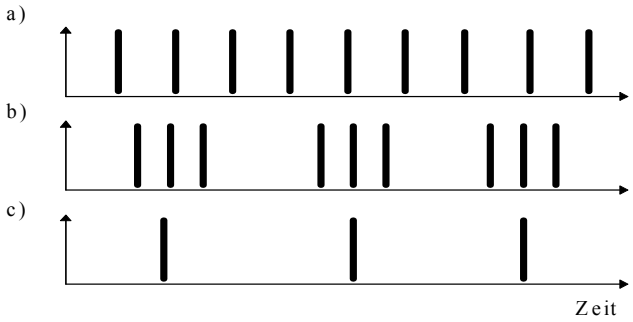


Abb. 1.8 *Kodierung und Synchronität von Impulsfolgen*

Die beiden ersten Folgen von Impulsen a) und b) haben die gleiche mittlere Frequenz von 9 Impulsen pro betrachteten Zeitabschnitt. Allerdings würden wir sie nicht unbedingt als "ähnlich" oder "synchron" betrachten, im Unterschied zu den Impulsfolgen b) und c), die mit der Kodierung "Impulse pro Sekunde" allerdings sehr unterschiedlich

bewertet werden würden. Um das Merkmal "Synchronität" zu fassen, muss hier die Kodierung geändert werden. Bisher zeigten Forschungen auf diesem Gebiet noch kein allgemeines, konsistentes Kodierungsschema, so dass wir zunächst bei der vorher eingeführten Interpretation der Reize als „Impulsfrequenz“ und damit bei einer Kodierung durch reelle Zahlen bleiben. Näheres zur Kodierung ist im Kapitel „Neural Encoding“ in [DAB01] zu finden.

Ein weiteres Problem ist die Bedeutung kleiner Signale, die die großen Spikes zusätzlich überlagern, sog. „Spikelets“. Sie stammen von den wenigen elektrischen Synapsen, die zusätzlich zu den ca. 30.000 chemischen Synapsen der Zellen, etwa im Hippocampus, existieren. Obwohl sie so klein sind, verursachen sie doch ca. 1/3 aller Aktionspotentiale und haben damit eine nicht unerhebliche Bedeutung, beispielsweise beim räumlichen Gedächtnis und der räumlichen Orientierung [ELCB10].

1.2 Modellierung der Informationsverarbeitung

Die Flut der neurobiologischen, neurophysiologischen und experimentalpsychologischen Daten lässt es einerseits nicht zu, alle Artikel über das Gehirn zu lesen und erschwert andererseits, einfache, konsistente und klare Aussagen über seine Funktionen zu machen. Ein wertvolles Instrument ist dabei die Modellbildung, um dieses Gestrüpp an experimentellen Daten zu lichten, relevantes herauszufiltern und unwichtiges oder sogar falsches zurückzustellen. Die Rückwirkungen, die von den einfachen, verständlichen Modellen auf die Experimente und Untersuchungen ausgehen, bestätigt dabei den altbekannten Satz: "Es gibt nichts Praktischeres als eine gute Theorie"!

Können biologische Elemente mit aus der Informatik bekannten Elementen der Multiprozessorsysteme verglichen werden? Im Unterschied zur traditionellen Unterteilung der Rechensysteme in die Maschine (Hardware) und in die Algorithmen, die darauf ausgeführt werden (Software), lassen sich bei den Neuronalen Netzen die beiden Aspekte nicht streng voneinander trennen. Ähnlich wie bei den systolischen Feldern sind Hardwarearchitektur und Funktionsalgorithmus als Ganzes zu sehen. Die Hardwarearchitektur implementiert dabei den Algorithmus; die Programmierung als Anpassung des allgemeinen Algorithmus an eine spezielle Aufgabe erfolgt dynamisch durch die Eingabe von Trainingsmustern. Das Gehirn als Vorbild einer neuronalen Maschine wird deshalb manchmal mit einem neuen Namen auch als *wetware* oder *brainware* bezeichnet.

1.2.1 Formale Neuronen

Untersuchen wir nun die Funktion der Prozessorelemente, der formalen Neuronen, etwas genauer.

Das Grundmodell

Im Unterschied zur Biologie bzw. Neurologie benutzen wir für unsere Neuronen-Elemente kein Modell, das alle Aspekte eines Neurons exakt beschreibt, sondern nur ein Modell, das eine sehr grobe Verallgemeinerung darstellt. Die sich damit ergebenden Netze sind auch keine Neuronen-Netze, sondern nur "neuronale", also neuronenähnliche Netze. Trotz aller vereinfachenden Annahmen erhofft man sich natürlich trotzdem, noch alle wesentlichen Funktions-Charakteristika übernommen zu haben.

Das Grundmodell eines Neurons stützt sich im Wesentlichen auf die Vereinfachungen von McCulloch und Pitts [MC43] aus dem Jahre 1943, die ein Neuron als eine Art Addierer mit Schwellwert betrachteten. Die Verbindungen (*Synapsen*) eines Neurons nehmen Aktivierungen x_i mit bestimmten Stärken w_i von anderen Neuronen auf, summieren diese und lassen dann am Ausgang y (*Axon*) des Neurons eine Aktivität entstehen, sofern die Summe vorher einen Schwellwert s überschritten hat (s. Abb. 1.9).

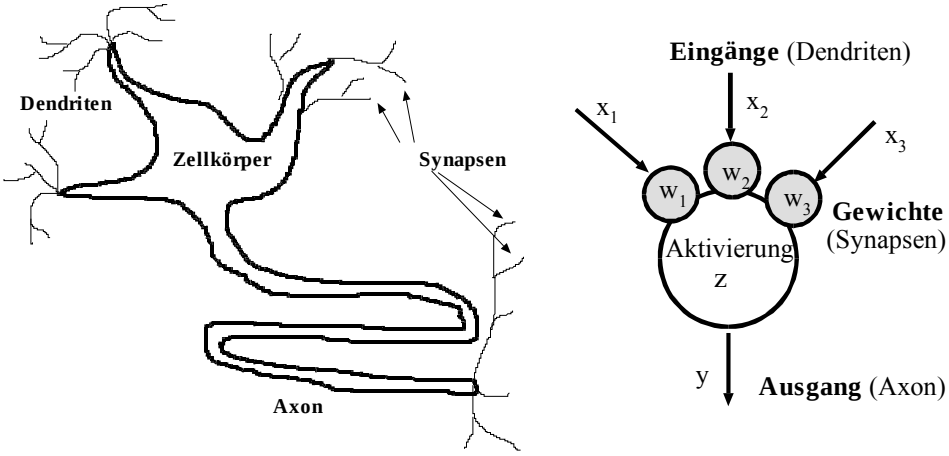


Abb. 1.9 *Ein biologisches Neuron und ein Modellneuron*

Diese Modellierung fasst die Wirkungen aller Hunderten von Synapsen zwischen den Axon-Verzweigungen eines Neurons und dem Dendritenbaum des anderen Neurons zu einem einzigen Gewicht w_i zusammen und vernachlässigt dabei natürlich solche Faktoren wie bidirektionale, elektrische Synapsen, chemische Informationswege (wie hormonelle Stimulierung bzw. Dämpfung), Energie-Versorgungsfragen (wie den Kalzium-Ionenstrom oder die Acethylcolin-Synthese) und vieles mehr. Auch die Weiterleitung der Erregung in den Dendriten und Axonen, die man exakter mit Differentialgleichungen von der Art modellieren kann, wie sie für Transatlantikkabel verwendet werden (*Kabelgleichungen*), wird intensitätsmäßig in die Gewichte projiziert; die zeitlichen Aspekte

werden (hier!) vernachlässigt. Trotzdem ermöglicht dieses einfache Modellierung einige interessante Netzfunktionen.

Die Gewichte wurden von McCulloch und Pitts noch alle als gleich angenommen; eine einzelne Inhibition (negative Gewichte) verhindert die gesamte Ausgabe, die in Übereinstimmung mit den damaligen Erkenntnissen als binär angenommen wurde, entsprechend der gleichmäßigen Spikegröße in Abb. 1.5.

Als allgemeine Aussage zeigten McCulloch und Pitts in ihrer Arbeit, dass mit diesen einfachen Elementen jeder finite logische Ausdruck berechnet werden kann. Tatsächlich entsprechen die McCulloch&Pitts-Neuronen eher den logischen Gattern der Jahrzehnte später erfundenen Computer, die mit den Regeln der Boolschen Algebra beschrieben werden können.

Funktionsmodellierung

Fassen wir die Eingabeaktivitäten x_1 .. x_n zum Eingabevektor $\mathbf{x} = (x_1, .., x_n)^T$ und die Gewichte w_1 .. w_n zum Gewichtsvektor $\mathbf{w} = (w_1, .., w_n)^T$ zusammen, so lässt sich die resultierende Aktivität *z* beispielsweise als Summe der gewichteten Eingaben im Modellneuron (*Sigma-unit*) und damit formal als Skalarprodukt (*inneres Produkt*) beider Spaltenvektoren schreiben:

$$z(\mathbf{w},\mathbf{x}) = \sum_j w_j x_j = \mathbf{w}^T\mathbf{x} \qquad \textit{Aktivitätsfunktion} \tag{1.1}$$

Die biologische Aktivitätsfunktion "sammelt" die Ladungen der dentritischen Eingänge zu einem elektrischen Potential und wird deshalb manchmal auch als "Potentialfunktion" (*potential function*) bezeichnet. In manchen Modellierungen werden Eingänge mit gleichartiger Aktivitätsfunktion zu Gruppen (*sites*) zusammengefaßt, wobei die Funktion Gl. (1.1) als *site function* bezeichnet wird. In diesem Fall wird z zur einfachen, ungewichteten Summe $\Sigma_i z_i$ aller "site-Aktivitäten" z_i.

Für theoretisch-analytische Zwecke kann man die obige Definition (1.1) auch auf Eingaben und Gewichte erweitern, die kontinuierlich in einem Intervall ("entlang einer Strecke") definiert sind. In diesem Fall wird die diskrete Summe zum Integral

$$z(w,x) = \int_{\xi_0}^{\xi_1} w(\xi)\, x(\xi)\, d\xi \tag{1.2}$$

Sehr oft muss die Aktivität erst eine Schwelle (*bias*) s überschreiten, bevor sie sich beim Ausgang (Axon) auswirkt (s. Abb. 1.7). Dies lässt sich durch die Minderung der Aktivität um den Schwellwert modellieren:

$$z(\mathbf{w},\mathbf{x}) = \mathbf{w}^T\mathbf{x} - s$$

Den zusätzlichen Term s, der hier gegenüber Gl. (1.1) auftritt, kann man allerdings mit einem Trick in der Notation wieder verschwinden lassen, indem man eine Erweiterung der Vektoren um eine Zusatzkomponente

$$\mathbf{x} \rightarrow \mathbf{x} = (x_1, .., x_n, 1)^T \quad \text{und} \quad \mathbf{w} \rightarrow \mathbf{w} = (w_1, .., w_n, -s)^T \tag{1.3}$$

vornimmt. Das Skalarprodukt ist somit wieder

$$z(\mathbf{w},\mathbf{x}) = \sum_j w_j x_j - s = (w_1, .., w_n, -s)(x_1, .., x_n, 1)^T = \mathbf{w}^T\mathbf{x} \tag{1.4}$$

Interessanterweise lässt sich die obige Aktivitätsfunktion auch anders schreiben

$$z = w^{(0)} + \sum_j w_j^{(1)} x_j \tag{1.5}$$

wobei wir mit der Notation $^{(k)}$ die Anzahl der Wechselwirkungen (Korrelationen) von **x**-Komponenten untereinander bezeichnen. Will man also auch "höhere" Wechselwirkungen an einer Synapse (*high-order synapses*) modellieren, so kann man die lineare Aktivitätsfunktion (1.1) für das i-te Neuron um zusätzliche, nichtlineare Terme erweitern:

$$z_i = w_i^{(0)} + \sum_j w_{ij}^{(1)} x_j + \sum_{jk} w_{ijk}^{(2)} x_j x_k + \sum_{jkl} w_{ijkl}^{(3)} x_j x_k x_l + ... \tag{1.6}$$

Diese Art von Neuronen werden auch als *Sigma-Pi-units* bezeichnet und können (im Unterschied zu den einfachen Modellneuronen) höhere Korrelationen in der Eingabe feststellen, für die sonst ein Netz von mehreren einfachen Neuronen benötigt werden würde [GIL87], s. auch Kapitel 4.1.3.

Die Aktivität y am Neuronenausgang wird durch die Ausgabefunktion (*activation function*) S(.), abhängig von der internen Aktivität *z*, beschrieben:

$$y = S(z) \quad \textit{Ausgabefunktion} \tag{1.7}$$

In manchen Modellierungen (s. z.B. [RUM86]) wird noch eine zusätzliche Funktion f (*output function*) angenommen, so dass y = f(S(z)) wird. Da dies keine konzeptionellen Vorteile bringt und man immer eine neue Ausgabefunktion S' = f(S) definieren kann, die die gewünschte Leistung y = S'(z) erbringt, beschränken wir uns bei unserer Modellierung auf eine einzige, einfache Ausgabefunktion. Auch die vorher erwähnten *sites* lassen sich durch zusätzliche lineare Neuronen bei der Eingabe modellieren und werden deshalb in unserem Neuronenmodell einfacherweise weggelassen.

Die gesamte Reaktion des formalen Neurons kann man auch als Ergebnis nur einer Funktion, der Transferfunktion F

$$y = S(z(\mathbf{x},\mathbf{w})) = F(\mathbf{x},\mathbf{w}) \qquad \textit{Transferfunktion} \tag{1.8}$$

auffassen. Mit diesen Überlegungen können wir ein formales Neuron ähnlich wie einen endlichen Automaten definieren:

Definition 1.1

Sei eine Eingabemenge X ={**x**}, eine Ausgabemenge Y = {y} sowie eine Zustandsmenge W = {**w**} gegeben. Ein *formales Neuron* v ist ein Tupel (X,Y,W,F,L), wobei die Transferfunktion F durch F: W×X→Y und die *Lernfunktion* L durch L: W×X×Y×Y→W definiert werden.

Diese Definition unterscheidet sich etwas von derjenigen, die man von endlichen Automaten gewohnt ist. Zum einen kann ein formales Neuron bei reellen Gewichten mit W = $\Re^n$ unendlich viele Zustände haben - ein in der Realität nicht existierender Automat, da dies unendlich viel Information (Speicherplatz!) erfordern würde. Zum anderen kann die "Zustandsüberführungsfunktion" L außer von der aktuellen Eingabe (oder z.B. einem Erwartungswert davon) und dem aktuellen Zustand auch von der zu lernenden Ausgabe und von der aktuellen (und früheren) Ausgabe abhängig sein. Ein deterministischer, endlicher Automat besitzt außerdem noch etwas, was kein formales Neuron hat: einen definierten Anfangszustand $\mathbf{w_0}$. Statt dessen wird für formale Neuronen bei Simulationen meist ein zufälliger Anfangswert angenommen.

Wichtig bei unserer Definition ist die Modellierung, dass die Ausgabe Y nur von der Eingabe X und von dem jetzigen Zustand (den Gewichten) W abhängig ist - weder von einer früheren Ausgabe (z.B. keine "Ermüdungserscheinungen ", Hysterese etc.), noch von einem früheren Zustand.

Für die Transferfunktion F sowie die Zustandsüberführung L der Gewichte, im Folgenden "Lernen" genannt, erlaubt die obige Definition viele Modelle. Beispielsweise können Aktivität und Lernen eines Neurons bei reellen Gewichten mit $\mathbf{w}\in\Re^n$ und W = $\Re^n$ nur von den eigenen Gewichten bestimmt sein. Ein Einfluss der Gewichte anderer Neuronen wird z.B. durch das kartesische Produkt W = $\Re^n\times...\times\Re^n$ modelliert.

Da man im allgemeinen sowohl durch das jetzige Ergebnis Y als auch durch frühere (direkt gespeicherte) Ergebnisse Y lernen kann, ist in der obigen Definition die Ausgabemenge zweimal vertreten. Allerdings könnten wir diesen Einfluss auch mit einem externen Speicher auslagern und diese Rückkopplung mit zusätzlichen Eingängen modellieren. Man sieht: es gibt viele Modellierungsmöglichkeiten.
Passend zu Definition 1.1 kann man ein neuronales Netz definieren:

Definition 1.2

Ein *neuronales Netz* ist ein gerichteter Graph G := (K,E) aus einer Menge von Knoten K = {v}, den neuronalen Einheiten, und einer Menge von Kanten E ⊆ K×K, den Verbindungen zwischen den Einheiten.

Unter "neuronale Einheiten" wollen wir dabei zunächst nur einzelne formale Neuronen verstehen, obwohl sich dies leicht (z.B. auf Subgraphen) ausweiten lässt.

Da ein neuronales Netz meist nicht isoliert für sich existiert, definieren wir uns noch *Eingabeneuronen*, die eine Eingabe von außerhalb des Netzes erlauben und somit nicht

als "echte" formale Neuronen, sondern nur als Datenquellen anzusehen sind, und *Ausgabeneuronen*, deren Ausgänge Daten nach außerhalb des Netzes weiterleiten und damit wie Datensenken im Netz wirken können. Beispiele für Eingabeneuronen sind Sensoren (z.B. Fotozellen, Mikrofone etc.), Datenfiles oder einfach nur Anschlussstecker; Beispiele für Ausgabeneuronen sind formale Neuronen, die an Peripheriegeräte (z.B. Lampensteuerung, Gelenkmotoren etc.), Datenfiles oder ebenfalls nur an Anschlussleitungen angeschlossen sind. Die Definitionen für Eingabe- und Ausgabeneuronen sind somit nicht symmetrisch.

Bei der Definitionen 1.1 und 1.2 beachte man, dass die Gewichte zu den Neuronen gehören und damit das neuronale Netz nur als gerichteter, aber nicht gewichteter oder bewerteter Graph definiert wurde. Die Gewichtung einer Kante mit einer Zahl (einem Gewicht) ist nicht unproblematisch, da wir uns damit auf eine Modellierung festlegen anstatt dies in der Definition eines formalen Neurons zu kapseln. Erweitern wir die Definition, beispielsweise mit der Einführung von höheren Synapsen, so haben wir im Beispiel nicht mehr nur ein Gewicht pro Graphkante: die Definition des gesamten Netzes muss nun geeignet umgeändert werden. Aus diesem Grund bleiben wir bei der Definition eines neuronalen Netzes als gerichteten Graphen, im Unterschied beispielsweise zu den Definitionen in [RUM86] und [MÜL90].

Dabei darf man nicht übersehen, dass die Definition 1.2 unvollständig ist. Es wird zwar die Verbindung der Ein- und Ausgaben von formalen Neuronen in der "Funktionsphase" beschrieben, aber nichts darüber ausgesagt, wie in der "Lernphase", falls eine solche für das betrachtete Netz überhaupt vorgesehen ist, die Gewichte "gelernt" werden. Um die wechselseitigen Einflüsse darzustellen, benötigt man einen weiteren Graphen, ein Lernnetz. Im Allgemeinen wird dies aber nicht extra hingemalt, sondern (wenn überhaupt) außer durch Angabe der Lernregeln nur durch zusätzliche, gestrichelte Kanten im Aktivitätsnetz dargestellt, obwohl das Ändern der Gewichte und damit ihre absolute Größe in der Funktionsphase entscheidend für das Verhalten des Gesamtnetzwerks ist.

Wofür kann man derartige allgemeine Definitionen wie 1.1 und 1.2 verwenden? Abgesehen von theoretischen Überlegungen kann man den Modellrahmen von 1.1 und 1.2 dazu nutzen, die Architektur eines Neurocomputers genauer festzulegen. Die Definitionen 1.1 und 1.2 beschreiben die elementaren Grundfunktionen einer neuronalen Maschine. In den Begriffen der Informatik sind es Definitionen von *Schnittstellen* von *virtuellen Maschinen* (s.[BRA97]), mit denen man komplexere Funktionen (z. B. Bild- und Spracherkennung) realisieren kann. Auf welche Art und Weise diese Maschinen implementiert werden, ob durch Simulation oder direkt als Chip in Hardware, ist letztlich von den Einsatzbedingungen, den Anforderungen und den Geldmitteln abhängig und ändert nichts an der Funktionalität.

Hat man allgemeine Strukturen von Aktivitäten und Lernregeln gefunden, die für eine möglichst große Zahl von Algorithmen zutreffen, (vgl. [RAS90]) so lässt sich dies als

Vorlage für die Organisation des Daten- und Kontrollflußes in einer spezialisierten Hardware nutzen.

Gewichtslose Neuronale Netze

Eine ganz andere Modellierung führte Igor Aleksander [ALE90], [ALE91] durch. Er ersetzte die kontinuierlichen, analogen Aktivitäts- und Ausgabefunktionen durch diskrete Funktionstabellen der Transferfunktion. Jeder der *n* Eingänge kann einen Wert aus einem diskreten Alphabet aus *N* möglichen Werten (z.B. binäre, reelle oder symbolische Werte) annehmen; die Aktivität des i-ten Neurons wird mittels einer *nxN* Matrix zu einem diskreten Aktivitätswert y_i ermittelt.

Das neuronale Netz besteht in diesem Fall aus Mengen derartiger neuronalen Einheiten (*General Neural Unit, GNU*) und einer Verbindungsmatrix oder einer Wahrscheinlichkeitsangabe über Zufallsverbindungen zwischen den Ein- und Ausgaben der GNUs. Außerdem sind Regeln (Algorithmen) nötig, um die Tabellenwerte und die Zuordnung dem Problem anzupassen.

Bezieht man einen oder mehrere Zustände des Neurons zu einem früheren Zeitpunkt in die Tabelle der Transferfunktion mit ein, so besteht der wesentliche Unterschied zu einem "normalen" endlichen Automaten und einem solchen Neuron in der Veränderung der Tabellen in Abhängigkeit von Trainingsmustern. Alle Lernalgorithmen, die in diesem Buch beschrieben sind, müssen für diese Modellierung entsprechend umformuliert werden.

1.2.2 Ausgabefunktionen

Der Wertebereich der verwendeten Variablen ist, je nach Modellvariante und Anwendungsbereich, sehr unterschiedlich.

Binäre Ausgabe

Im erweiterten Modell von McCulloch und Pitts sind nur binäre (*aktiv/nicht-aktiv*) Werte für Input x_i und Output y vorgesehen; die Gewichte w_i sind dabei reell. Es ergibt sich eine positive Aktivität erst nach dem Überschreiten eines Schwellwerts s. Dies lässt sich relativ einfach durch Erweiterung der Gewichte um den Schwellwert nach Gl.(1.3) modellieren. Somit ist für $x_i, y \in \{0,1\}$ und $w_i \in \Re$, den reellen Zahlen, die binäre Ausgabefunktion

$$y = S_B(z) := \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases} \quad \textit{Heavyside-Funktion} \tag{1.9}$$

wie in der folgenden Abb. 1.10 links gezeigt ist.

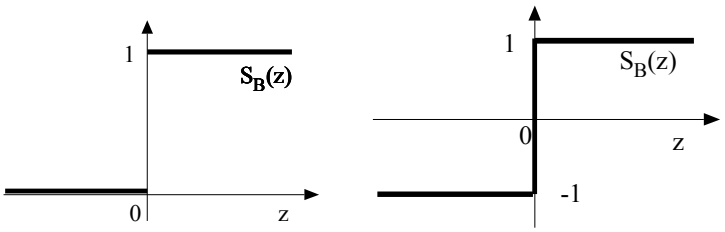


Abb. 1.10 *Binäre Ausgabefunktionen*

Anstelle von 0 wird auch für "nicht-aktiv" manchmal der Wert -1 verwendet, so dass sich mit der Transformation $x_i \rightarrow (2x_i-1)$ die Werte zu $\{+1,-1\}$ ergeben. Als Ausgabefunktion lässt dabei mit $y \in \{+1,-1\}$ die Vorzeichenfunktion *sgn(.)* verwenden

$$y = S_B(z) := \operatorname{sgn}(z) = \begin{cases} +1 & z \geq 0 \\ -1 & z < 0 \end{cases} \tag{1.10}$$

wie in Abb. 1.10 rechts gezeigt.

Beispiel *formale Neuronen und logische Gatter*

Bei *n* Eingaben x_i aus $\{0,1\}$, den Gewichten $w_i = 1/n$ und einer Schwelle von $s = 1/(n+1)$ wird aus einem Sigma-Neuron mit einer binären Ausgabefunktion S_B ein ODER Gatter, beispielsweise bei n=2:

x_1	x_2	$z=x_1/2 + x_2/2$	x_1 OR x_2
0	0	z=0	0
0	1	$z=½>1/3 \Rightarrow S_B=1$	1
1	0	$z=½>1/3 \Rightarrow S_B=1$	1
1	1	$z=1>1/3 \Rightarrow S_B=1$	1

Erhöhen wir die Schwelle auf $s = (n-1)/n$ so resultiert ein UND Gatter. Negative Gewichte erzeugen eine NICHT-Funktion.

Es gibt einen interessanten Unterschied derartiger formaler Neuronen zu den (inzwischen) konventionellen, einfachen logischen Gattern: allein die Veränderung eines analogen Parameters (der Schwelle) lässt die logische Funktion dieses neuronalen Gatters bei gleicher Architektur von einer logischen Grundfunktion zu einer anderen umschalten.

Mit der Erweiterung auf reelle Zahlen $x_i \in \Re$ kann man die binäre Ausgabefunktion zur Abbildung der reellen Variablen auf eine symbolische Größe (1 = TRUE, 0 = FALSE)

benutzen, wie sie beispielsweise für Klassifikationen (Klasse *k* liegt vor / liegt nicht vor) benötigt wird.

Begrenzt-lineare Ausgabefunktion

Nach heutigen Erkenntnissen wird allerdings Information über die absolute Größe des summierten Signals im Neuron nicht nur durch die binäre Amplitude (*spikes*), sondern auch durch die Frequenz der binären Ausgangsimpulse weitergegeben (Frequenz-Modulation); wie in Abb. 1.7 gezeigt wurde, ist die gewichtete Eingabeaktivität dabei in weiten Bereichen der Ausgangsfrequenz proportional.

Betrachten wir nun als Aktivität die Impulsfrequenzen. Diese lassen sich in bestimmten Grenzen durch positive, reelle Zahlen modellieren. Fügen wir nun noch die inhibitorische Aktivität durch negative reelle Zahlen hinzu, so erhalten wir ein Modell, bei dem Eingabe- und Ausgabesignale reell sind; die Ausgabe ist proportional zu der Eingabe:

$$y = S(z) = z \quad \textit{lineare Ausgabe} \tag{1.11}$$

Betrachten wir nochmals Abbildung Abb. 1.7. Hier gibt es zwei wichtige Werte für die Eingabeaktivität: den unteren Schwellwert s_1, der überschritten werden muss um eine Ausgabe zu erreichen, und den Wert s_2, nach dessen Überschreiten keine weitere Änderung der Ausgabe erfolgt (*Sättigung*). Mit der linearen Transformation der Variablen $z \rightarrow z\text{-}z_0$ mit $z_0 := s_1+(s_2\text{-}s_1)/2$ erfolgt die Ausgabe S(z) linear und symmetrisch um den Nullpunkt der y-Achse mit einer einheitlichen Schwelle $s=(s_2\text{-}s_1)/2$ und dem Sättigungswert z_{max} als eine *Rampenfunktion* mit $x_i, y, w_i \in \Re$

$$y = S_L(z,s) := \begin{cases} z_{max} & z > s \\ z_{max}/2 + kz & -s \le z \le s \\ 0 & z \le -s \end{cases} \qquad k=z_{max}/2s \tag{1.12}$$

Ist eine symmetrische Ausgabe nötig, so kann man mit der Transformation $y \rightarrow 2(y\text{-}y_0)$ und $y_0 := z_{max}/2$ die Ausgabefunktion auch symmetrisch um die z-Achse durch den Nullpunkt legen. Dabei kann y_0 beispielsweise der Mittelwert $\langle y \rangle$ der Ausgabe bedeuten.

$$y = S_L(z,s) := \begin{cases} z_{max} & z > s \\ kz & -s \le z \le s \\ -z_{max} & z \le -s \end{cases} \qquad k=z_{max}/s \tag{1.13}$$

In der folgenden Abb. 1.11 sind die beiden normierten Funktionen mit $z_{max} := 1$ gezeigt. Die binären Stufenfunktionen lassen sich dabei als Spezialfall der Rampenfunktionen betrachten, wenn die Geradensteigung *k* im Grenzwert gegen unendlich geht.

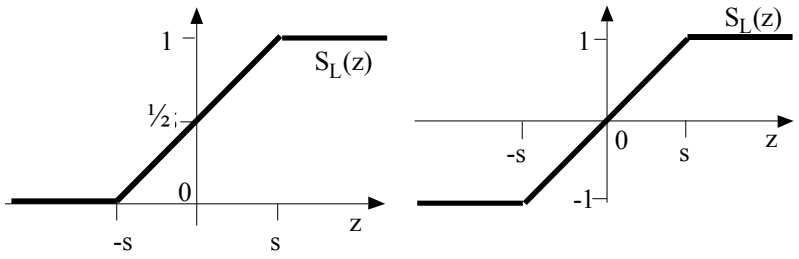


Abb. 1.11 *Begrenzt-lineare Ausgabefunktionen*

Sigmoide Ausgabefunktionen

Bei der analytischen Verwendung der Ausgabefunktionen ist es manchmal nötig, nicht nur die Ausgabefunktion selbst, sondern auch die Ableitung der Funktion zu benutzen, die im Unterschied zu den Ableitungen der binären Stufenfunktion und der Rampenfunktion stetig sein sollte. Für diesen Zweck benutzt man auch andere nichtlineare Funktionen (*squashing functions,* "Quetschfunktionen"), die auch die bei großen Signalstärken beobachteten neurologischen Sättigungseffekte modellieren. Die als *sigmoide Funktionen* bekannten Ausgabefunktionen sind dabei praktischer als die obigen Stufenfunktionen, obwohl das Verhalten der Netze interessanterweise kaum von der genauen Form der Quetschfunktionen abhängt.

Beispiele für solche Funktionen sind die aus der Physik bekannte *logistische* Funktion oder *Fermi-Funktion*

$$S_F(z) := \frac{1}{1+e^{-kz}} \tag{1.14}$$

und die um den Nullpunkt symmetrische, skalierte Version, der *hyperbolische Tangens*

$$S_T(z) := 2S_F(2z)-1 = \frac{1-e^{-kz}}{1+e^{-kz}} = \tanh(kz) \tag{1.15}$$

Eine weitere interessante Funktion ist die Kosinus-Quetschfunktion (*cosinus squasher*)

$$S_C(z) := \begin{cases} 1 & z \geq \pi/2 \\ 1/2(1+\cos(z-\pi/2)) & -\pi/2 < z < \pi/2 \\ 0 & z \leq -\pi/2 \end{cases} \tag{1.16}$$

In der folgenden Abb. 1.12 sind die Fermi- und die Kosinus-Quetschfunktion zu sehen.

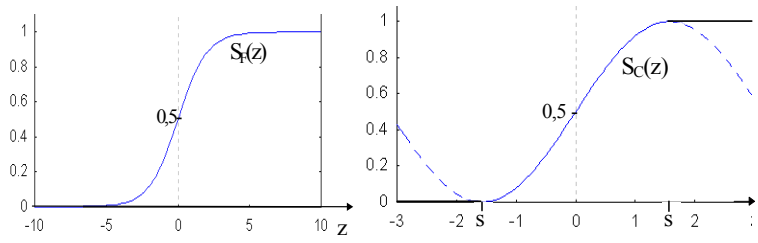


Abb. 1.12 *Beispiele sigmoidaler Quetschfunktionen*

Im Unterschied zu biologischen Neuronen ist es aber bei formalen Neuronen (die ja im Normalfall durch willkürliche Computersimulationen oder elektrische Schaltungen realisiert werden) möglich, völlig beliebige und für eine bestimmte Problemlösung günstige Neuronen neu zu definieren. Beispielsweise ist auch eine Ausgabefunktion denkbar, die die Form einer Gauß'schen Glockenkurve realisiert:

$$S_G(z) := \exp(-z^2/\sigma^2) \tag{1.17}$$

Die Aktivität kann dabei beispielsweise mit $z := |\mathbf{x}-\mathbf{w}|$ definiert sein, so dass S_G dann eine Bewertungsfunktion der Eingabe **x** bezüglich des Abstands vom Punkt **w** darstellt. Interessanterweise ist dabei die Gauß'sche Verteilungsfunktion *(Fehlerfunktion)*

$$S_V(z) := \int_{-\infty}^{z} S_G(z')\,dz' \tag{1.18}$$

eine sigmoidale Funktion.

Zur Veranschaulichung der Berechnung der Aktivität und der Ausgabefunktion ist im folgenden Beispiel für ein formales Neuron der Pseudo-Programmcode angegeben.

```
PROCEDURE   Z(w,x: ARRAY OF REAL): REAL;
    (* implementiert die Aktivierung eines Sigma-Neurons:
            Aufsummieren aller Eingaben nach Gl. (1.1) *)
VAR sum: REAL; i:INTEGER;
BEGIN
    sum:= 0;      (* Skalarprodukt bilden *)
    FOR i:=0 TO HIGH( x) DO
          sum:= sum  + w[i]*x[i];
    END;
    RETURN sum;
END Z;
```

```
PROCEDURE S(z:REAL):REAL;
    (* begrenzt-lineare Ausgabefunktion nach Gl.(1.13) *)
CONST      s=1;  Zmax=1; k=Zmax/s;
BEGIN      z:=z*k;
    IF z>s  THEN z:= Zmax  END;
    IF z<-s THEN z:=-Zmax  END;
    RETURN z;
END S;
```

Codebeispiel 1 Aktivierung und Ausgabe eines Neurons

1.2.3 Zeitmodellierung

Die Aktivitäten in neuronalen Netzen sind nicht konstant, sondern ändern sich mit der Zeit: x = x(t), z = z(t), y = y(t). Die Aktivität (1.1) unseres formalen Neurons ist somit

$$z(t) = \mathbf{w}(t)^T \mathbf{x}(t) \tag{1.19}$$

Viele Modelle von neuronalen Netzen (s. [GRO87] und Kapitel 5) sind aber zeitkontinuierlich und damit mit Differentialgleichungen beschrieben, da die Aktivität zu einem Zeitpunkt meist aus der Aktivität zu früheren Zeitpunkten hergeleitet werden kann. Nehmen wir beispielsweise an, dass der Abfluss der Ladung aus dem Zellkörper d.h. der Spannungsabfall $-\partial z/\partial t$ mit sinkender Spannung auch proportional geringer wird, so ist die entsprechende Differentialgleichung

$$\frac{\partial z(t)}{\partial t} \sim -z(t) \quad \text{oder} \quad \tau \frac{\partial z(t)}{\partial t} = -z(t) \tag{1.20}$$

mit der Proportionalitätskonstante τ. Nehmen wir noch an, dass ein konstanter Ladungszufluss bzw. konstante Spannung z_0 existiert, so modifiziert sich die Annahme zu

$$\tau \frac{\partial z(t)}{\partial t} = -z(t) + z_0 \tag{1.21}$$

Zur Lösung dieser Differenzialgleichung definieren wir uns

$$v(t) := (-z(t)+z_0)/\tau, \qquad \frac{\partial v(t)}{\partial t} = -\frac{1}{\tau}\frac{\partial z(t)}{\partial t} \tag{1.22}$$

und erhalten so mit Einsetzen aus Gl.(1.21)

$$-\tau \frac{\partial v(t)}{\partial t} = \frac{\partial z(t)}{\partial t} = (-z(t)+z_0)/\tau = v(t) \quad \text{oder} \quad \frac{\dot{v}}{v} = -\frac{1}{\tau} \tag{1.23}$$

Die Integration beider Seiten ergibt

$$\int \frac{\dot{v}}{v} dt = \ln v + const = \ln (const' \cdot v) = -\int \frac{1}{\tau} dt = -t/\tau + const'' \quad (1.24)$$

oder

$$const' \cdot v(t) = e^{-t/\tau} + \exp(const'') \quad (1.25)$$

Setzen wir die Definition (1.22) ein, so erhalten wir unter Umbenennung der Konstanten die Zeitfunktion

$$z(t) = Ae^{-(t-t_0)/\tau} + A_0 \quad (1.26)$$

was einer "Abklingkurve", beispielsweise der Entladung eines Kondensators, mit der Zeitkonstante τ entspricht. In der folgenden Abb. 1.13 ist dies dargestellt.

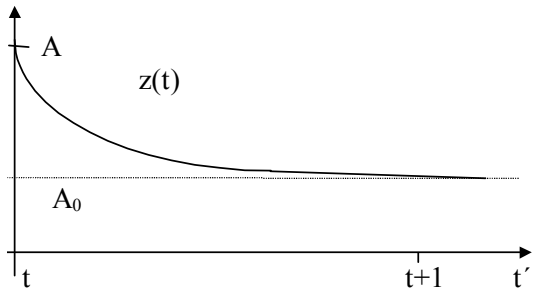


Abb. 1.13 *Visualisierung der Differenzialgleichung durch Zufluß- und Abflußverhalten*

Die ursprüngliche, zeitdiskrete Aktivität in Gl. (1.19) wird mit der zeitkontinuierlichen Formulierung zwischen den Zeitpunkten t_0 und t_1 mit einem Einschwingprozess interpoliert. Bei einer sehr kleinen Zeitkonstante τ erreichen wir, ausgehend von einem Wert $z(t_0) = A$, sehr schnell den Wert $z(t_1) \approx A_0$, so dass bei kleiner Zeitkonstante die zeitdiskrete und zeitkontinuierliche Formulierung stark übereinstimmen. Bei allen Rechnungen können wir deshalb annehmen, dass zum Zeitpunkt t+1 die Aktivität vom Zeitpunkt t abgeklungen ist und nur noch durch die neue Aktivität A_0 bestimmt wird.

Allerdings sind die zeitdiskrete und zeitkontinuierlichen Formulierungen sind nur im großen Zeitmaßstab äquivalent; bei dynamischen Vorgängen im Kurzzeitbereich ($t<\tau$) muss man bei der Computersimulation darauf achten, die Zeitschritte nicht zu groß zu machen. Der Sinn der Formulierung mit einer Zeitkonstante τ liegt dabei in einem gewissen "Trägheitseffekt", den man dem Modell damit verleiht. Im Unterschied zur zeit-

losen, sofortigen Reaktion in Gl.(1.19) lassen sich mit der "trägheitsbehafteten" Reaktion in Gl.(1.26) Zeitverzögerungen modellieren, was besonders bei Zeitsequenzen wichtig ist.

Trotzdem soll doch im weiteren Verlauf des Buches der zeitdiskreten Formulierung, falls möglich, der Vorzug gegeben werden, da die Darstellung mit einem diskreten Zeitschritt eher einer iterativen Anweisung in einem Computerprogramm ähnelt und damit die Umsetzung der abstrakten Formeln in Simulationsprogramme dem Leser erleichtert wird. Die allgemeine Umsetzung einer Differentialgleichung in eine Differenzengleichung ist nicht trivial und Gegenstand der numerischen Lösung von Differentialgleichungen [SMI78]. Der Erfolg lässt sich oft nur am konsistenten Verhalten beider Formen bei Randbedingungen nachprüfen.

1.2.4 Feedforward Netze und Schichten

Mit dem Gedanken, dass periphere Leistungen durch Verarbeitung in mehreren Stufen erbracht werden können, lassen sich die formalen Neuronen mit paralleler, gleicher Funktion zu Funktionsblöcken (Schichten) zusammenfassen und die gesamte periphere Informationsverarbeitung im Gehirn als pipeline-artige Folge von informationsverarbeitenden Schichten betrachten. Jede Schicht ist wie ein Programm-Modul, das mit Hilfe der genau festgelegten Schnittstelle (Eingänge, Ausgänge, etc) eine festgelegte Spezifikation erfüllt, wobei die eigentliche Implementation nicht entscheidend ist, solange die gewünschte Funktion tatsächlich erfüllt wird. Werden die Eingänge einer Schicht ausschließlich von den Ausgängen der Schicht davor gespeist, so wird im Unterschied zu den vollständig vernetzten Systemen diese geschichtete Struktur als *vorwärtsgerichtete* oder *feedforward* Netze bezeichnet. Die Grundidee dieser Charakterisierung ist also die Existenz von Funktionseinheiten (z.B. Schichten, Neuronen etc.), die nicht rückgekoppelt sind. Mit diesem Gedanken können wir versuchen, die Definition von "feedforward" für allgemeine Netze zu formalisieren mit Hilfe der Definition eines neuronalen Netzes als gerichteten Graphen.

Definition 1.3

> Angenommen, es liegt ein neuronales Netz nach Definition 1.2 vor. Das Netz wird genau dann als *feedforward* Netz bezeichnet, wenn der gerichtete Graph *zyklenfrei* ist.

Die Definition sagt nichts darüber aus, ob im gesamten neuronalen Netz Rückkopplungen existieren, sondern betrachtet nur die Verbindungen zwischen genau spezifizierten neuronalen Einheiten. Erweitern wir unsere Definition von "neuronalen Einheiten" von formalen Neuronen auf neuronale Schichten oder Teilnetze, so können auch in einem feedforward Netz *innerhalb* der Einheiten (innerhalb der Schichten oder neuronalen

Funktionsgruppen) Rückkopplungen vorhanden sein, die aber nach außen nicht sichtbar sind. Leider wird in der Literatur diese Sichtweise meist implizit benutzt, aber nicht explizit klar gesagt. So kann es leicht zu Begriffsverwirrungen kommen, wenn in neuronalen feedforward Netzen auf unterster Ebene Rückkopplungen existieren. In der folgenden Abb. 1.14 ist gezeigt, wie für zwei verschiedene Betrachtungsebenen (zwei verschiedene Definitionen von "Einheit") für das selbe neuronale Netz zwei verschiedene Charakterisierungen erfolgen können: In der linken Figur ist nur eine Eingabe und eine Ausgabe der schraffiert gezeichneten Einheit erkennbar; das "Netz" ist also ein "feedforward" System. Anders bei der rechten Figur; die nun erkennbare Rückkopplung zur Einheit charakterisiert das Netz als "feedback" System.

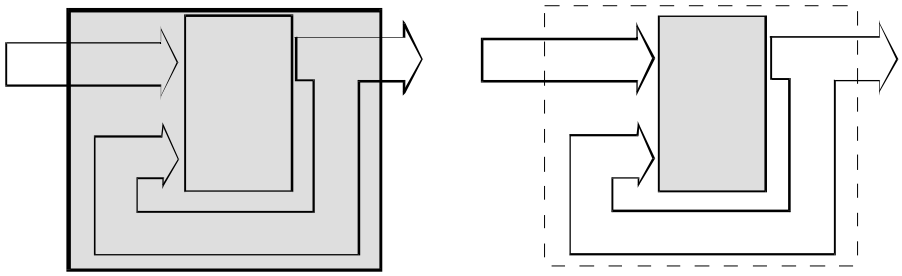


Abb. 1.14 *Feedforward und feedback Netzwerk*

Die Charakterisierung "feedforward" bezieht sich normalerweise ausschließlich auf die Informationsflussrichtung (Signalflussrichtung) in der "Funktionsphase", ähnlich wie die Definitionen 1.1 und 1.2. Der Informationsfluss in der "Lernphase", in der die Gewichte neu gesetzt werden, ist dagegen nicht festgelegt und geht meist nicht in die Netzbezeichnung ein. Aus diesem Grund enthält unsere Definition 1.2 eines neuronalen Netzes auch keine Lernalgorithmen - sie beziehen sich nur auf die Aktivitätsphase, nicht auf die Lernphase.

1.3 Einsatz formaler Neuronen

Verwenden wir lineare, formale Neuronen, so können wir die Funktion einer ganzen Gruppe von parallel arbeitenden Neuronen auch wesentlich kompakter in klassischer mathematischer Notation schreiben.

1.3.1 Lineare Transformationen mit formalen Neuronen

Wie wir in Gleichung (1.1) sehen, kann man bei linearer Aktivität und der Identität als Ausgabefunktion die Ausgabe des i-ten Neurons als Skalarprodukt wischen Gewichtsvektor und Eingabevektor schreiben.

$$y_i = \mathbf{w}_i^T \mathbf{x}$$

Angenommen, wir haben ein Netz aus *m* Neuronen, die alle die gleiche Eingabe besitzen, s. Abb. 1.15

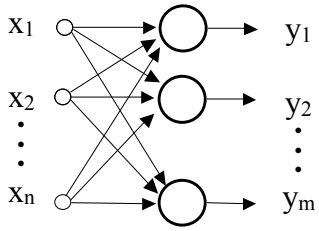


Abb. 1.15 Lineares Transformationsnetz

so lassen sich die *m* Skalarprodukte untereinander schreiben.

$$\begin{array}{l} y_1 = (w_{11},\ldots,w_{1n})\,\mathbf{x} \\ \vdots \\ y_m = (w_{m1},\ldots,w_{mn})\,\mathbf{x} \end{array} \quad \cong \quad \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{bmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

Fassen wir die *m* Werte $y_1..y_m$ zu einem Spaltenvektor **y** und die *m* Zeilenvektoren $\mathbf{w}_i^T$ zu einer Matrix **W** zusammen, so lassen sich die obigen linearen Gleichungen auch als Matrizenmultiplikation

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{1.27}$$

notieren. Lineare neuronale Netze können also alle Arten von linearen Transformationen durchführen. Wählen wir beispielsweise für **W** die Matrix

$$\mathbf{W}_r = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \tag{1.28}$$

so implementiert das neuronale Netz eine Drehung (*Rotation*) der zweidimensionalen Eingaben um den Winkel φ.
Eine Matrix, die nur die Hauptdiagonalelemente ungleich Null besitzt mit den Werten c_i bewirkt eine Dehnung oder Streckung (*Skalierung*) der Eingabe

$$\mathbf{W}_c = \begin{pmatrix} c_1 & 0 \\ 0 & c_2 \end{pmatrix}, \quad \text{so dass} \quad \mathbf{W}_c\mathbf{x} = \begin{pmatrix} c_1 x_1 \\ c_2 x_2 \end{pmatrix} \tag{1.29}$$

und eine Multiplikation mit der um eine Dimension erweiterte Matrix der Form

$$\mathbf{W}_s = \begin{pmatrix} 1 & 0 & s_1 \\ 0 & 1 & s_2 \\ 0 & 0 & 1 \end{pmatrix}, \text{ so dass } \mathbf{W}_s\mathbf{x} = \begin{pmatrix} x_1 + s_1 \\ x_2 + s_2 \\ 1 \end{pmatrix} \tag{1.30}$$

bewirkt eine Verschiebung (*Translation*) eines um eine Konstante 1 auf drei Komponenten erweiterten, 2-dim Eingabevektors, s. Gl.(1.3). Erweitert man die Rotation und Skalierung ebenfalls um eine Dimension und kombiniert Translation, Rotation und Skalierung in einer Matrix, so erhält man beispielsweise für eine Abbildung, die sich aus Skalierung, Rotation und Translation zusammensetzt, die Matrix

$$\begin{aligned} \mathbf{W} = \mathbf{W}_s\mathbf{W}_r\mathbf{W}_c &= \begin{pmatrix} 1 & 0 & s_1 \\ 0 & 1 & s_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_1 & 0 & 0 \\ 0 & c_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} c_1\cos\varphi & -c_2\sin\varphi & s_1 \\ c_1\sin\varphi & c_2\cos\varphi & s_2 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{1.31}$$

so implementiert das entsprechende Netz eine *affine Transformation*. Diese Art von Transformation wird auch in der Roboterkinematik zur Koordinatenumrechnung eines Bewegungssegments benutzt und ist dort unter dem Namen *homogene Transformation* bekannt. Damit zeigt sich schon eine Einsatzmöglichkeit neuronaler Netze zur Robotersteuerung.

Aufgabe 1. 1
Man errechne die Koeffizienten w_{ij} der Gewichtsmatrix $\mathbf{W}$ einer linearen Schicht, die als Eingabe die Koordinaten eines Dreiecks erhält und als Ausgabe die Koordinaten des um 90° gedrehten Dreiecks ausgibt. Das Dreieck habe beispielsweise die folgenden Koordinaten A = (–1,0), B = (0,–1), C = (0,1). Der Drehpunkt R ist der Nullpunkt. Die neuen Koordinaten sind also in diesem Fall A = (0,–1), B = (1,0), C = (–1,0).

1.3.2 Klassifizierung mit formalen Neuronen

Da Neuronen komplizierte, physikalisch-chemisch-biologische Gebilde sind, ergibt eine Modellierung in allen Details eine sehr komplizierte Beschreibung. Es ist erstaunlich,

dass unser relativ einfaches Modellneuron als wichtender Summierer (s. Gl. (1.1) trotzdem schon erstaunliche Leistungen wie affine Transformationen vollbringen kann.
Im Folgenden betrachten wir nun andere, nicht-lineare Fähigkeiten eines einzelnen Neurons und die einer einfachen Schicht von Neuronen ohne Rückkopplungen.

Klassifizierung und Mustererkennung

Formale Neuronen lassen sich sehr effektiv bei dem Problem der Mustererkennung zur Trennung der sogenannten *Musterklassen* verwenden. Betrachten wir dazu als Beispiel Patienten mit chronischer Bauchspeicheldrüsenentzündung, die sich von normalen Patienten durch Konzentrationen bestimmter Stoffe x_1 und x_2 im Blut unterscheiden lassen. In Abbildung 1.2.6 sind die chronischen Patienten (×) und normale Patienten (•) jeweils als Punkte $\mathbf{x}=(x_1,x_2)$ oder *Muster* repräsentiert.

Abb. 1.16 Repräsentation von Patienten als Punktmuster (nach [BLOM])

Aufgabe einer Mustererkennung ist es, zwischen den beiden Patientengruppen *(Klassen)* eine Trennungslinie *(Klassengrenze)* zu finden, welche die Punktmengen trennt. Für eine Beurteilung eines unbekannten Patienten reicht es dann aus, die Lage seines Musterpunkts diesseits oder jenseits der Klassengrenze zu bestimmen.

Nehmen wir an, es existiere als Klassengrenze eine einfache Gerade *(lineare Separierung)* wie in Abb. 1.17. Dann lautet die Geradengleichung allgemein

$$x_2 = w_1 x_1 + s \tag{1.32}$$

oder mit $\mathbf{w}^T=(w_1,w_2,w_3)$ und $w_2 := -1$, $w_3:= s$, $x_3:=1$ ähnlich wie Gl.(1.3)

$$0 = w_1 x_1 + w_2 x_2 + w_3 x_3 = \mathbf{w}^T\mathbf{x} =: g(\mathbf{x},\mathbf{w}) \tag{1.33}$$

Für alle Punkte auf der Geraden bzw. Klassengrenze gilt, dass die *Diskriminanzfunktion* $g(\mathbf{x},\mathbf{w}) = 0$ ist. Für alle Muster $\mathbf{x}'$, die bei gleichem x_1-Wert oberhalb der Gerade liegen,

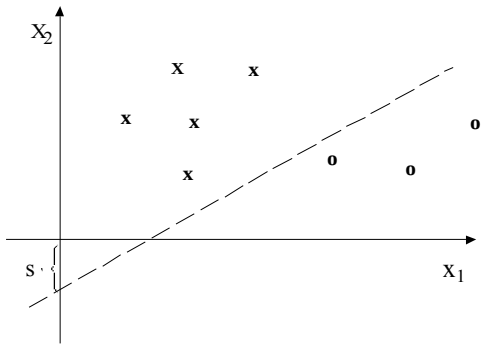


Abb. 1.17 Lineare Separierung von Musterklassen

gilt die Relation

$$x_2' > x_2 = w_1 x_1 + s \tag{1.34}$$

oder $0 > w_1 x_1 - x_2' + s = g(\mathbf{x}')$

und für die Muster aus Klasse 2 gilt entsprechend $0 < g(x')$. Die Diskriminierungsfunktion wirkt somit wie ein binärer Klassifikator von unbekannten Mustern $\mathbf{x}$.
Übrigens bleibt durch negierte Gewichte $\mathbf{w} \rightarrow -\mathbf{w}$ die Lage der Klassengrenze mit $-g(\mathbf{x}) = -\mathbf{w}^T\mathbf{x} = 0 = \mathbf{w}^T\mathbf{x} = g(\mathbf{x})$ unverändert; nur die obigen Relationen drehen sich um und $g(\mathbf{x}')$ wird für Klasse 1 positiv.
Normieren wir Gl.(1.33) für die Gerade mit dem Betrag des Vektors $\mathbf{w} = (w_1,w_2)^T$, so ist mit

$$s/|\mathbf{w}| = \text{const} = \mathbf{w}^T\mathbf{x} / |\mathbf{w}| = \mathbf{nx} \tag{1.35}$$

die Projektion von **x** auf den Einheitsvektor **n** konstant. Diese Beziehung ist als *Hessesche Normalform* der Ebene aus der linearen Algebra bekannt: eine Klassengrenze nach Gl. (1.35) hat also im allgemeinen die Form einer Hyperebene.

Vergleichen wir die Form von g(**x**) mit der Funktion des vorher eingeführten, binären Neuronenmodells, so ist ersichtlich, dass die Funktion $y(\mathbf{x}) = S_B(\mathbf{w},\mathbf{x})$ eines solchen Neurons gerade eine binäre Diskriminierungsfunktion darstellt. Ein solchermaßen definiertes, formales Neuron ist also ein Klassifikator; jedes Muster wird in eine durch die Gewichte und den Schwellwert definierte Klasse eingeordnet. Da die Trennlinie zwischen den Klassen eine Gerade bzw. eine Hyperebene darstellt, spricht man auch von einem *linearen Klassifikator* oder *linearen Separierung*. Durch entsprechende Algorithmen lassen sich die Koeffizienten w, und damit die Klassentrennung, lernen. Genauere Lernalgorithmen und -verfahren werden wir in späteren Abschnitten kennen lernen.
Allgemein lässt sich definieren:

Definition 1.4

Seien die Muster **x** und Parameter **w** in der Form von Gl.(1.3) gegeben. Zwei Klassen Ω_1 und Ω_2 des Musterraums $\Omega = \Omega_1 \cup \Omega_2$ mit $\Omega_1 \cap \Omega_2 = \varnothing$ heißen *linear separierbar*, falls eine Hyperebene $\{\mathbf{x}^*\}$ mit der Diskriminanzfunktion $g(\mathbf{x}^*) = \mathbf{w}^T\mathbf{x}^* = 0$ existiert, so daß für alle $\mathbf{x} \in \Omega_1$ gilt $g(\mathbf{x})<0$ und für alle $\mathbf{x} \in \Omega_2$ gilt $g(\mathbf{x})>0$.

Man beachte, dass die Trennung der Klassen $\Omega_1 \cap \Omega_2 = \varnothing$ zwar notwendig für "linear separierbar“ ist (Warum?), aber nicht hinreichend.
Die obige Definition linearer Separierung lässt sich übrigens verallgemeinern, s. [DUD73]. Nehmen wir an, dass die Muster **y**, die linear separiert werden können, mit $y_i = S_i(\mathbf{x})$ selbst wieder (nichtlineare) Funktionen von den Originalmustern **x** sind. Dann kann der Fall eintreten, dass die Klassen durch lineare Diskriminanzfunktionen $g(\mathbf{w},\mathbf{y}) = \mathbf{w}^T\mathbf{y}$ getrennt können, obwohl im Originalraum $\{\mathbf{x}\}$ dies nicht möglich war. Allerdings wird unsere Aufgabe dabei nicht einfacher: anstelle linearer Koeffizienten **w** müssen wir zusätzlich noch die Funktionen $S_i(\mathbf{x})$ bestimmen.

Aufgabe 1.2

Man beweise: Die folgenden Muster (x_1,x_2) sind nicht linear separierbar, so daß die Klassenzugehörigkeit durch die XOR-Funktion beschrieben wird.

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Dies bedeutet, dass die XOR-Funktion nicht durch ein Sigma-Neuron mit binärer Ausgabefunktion implementiert werden kann.

Aufgabe 1.3 **a)** Man berechne die Parameter einer affinen Transformation (und damit φ, c, s_1, s_2) so, dass die Muster (x_1,x_2) auf die folgenden neuen Koordinaten (z_1,z_2) abgebildet werden:

x_1	x_2	z_1	z_2
0	0	0	-1
0	1	1	0
1	0	-1	0
1	1	0	1

b) Wie sieht das dazu gehörende Netzwerk aus? Welche Werte haben die Gewichte?

c) Nehmen Sie als Ausgabefunktion $S_i(z)=z^2$ an. Welche logische Funktionen implementieren die Ausgaben $S_i(\mathbf{x})$?

Aufgabe 1.4

Gegeben seien 3 Klassenprototypen mit den Koordinaten $\mathbf{w}_1 = (1,3)$, $\mathbf{w}_2 = (3,3)$ und $\mathbf{w}_3 = (2,1)$. Geben Sie die Geradengleichungen an für die drei linearen Klassengrenzen, die jeweils durch $g_{ik} = \{\mathbf{x} / |\mathbf{x}-\mathbf{w}_i| = |\mathbf{x}-\mathbf{w}_k|\}$ zwischen Klasse i und k definiert seien.
Wie lautet die Koordinate des Schnittpunkts aller drei Grenzen ?

Aufgabe 1.5

Bei der Eingabe von $\mathbf{x} = (x_1,x_2)$ soll als Ausgabe der Index der Klasse aus Aufgabe Aufgabe **1.4**, in dessen Bereich **x** fällt, mit Hilfe formaler Neuronen ermittelt werden. Verwenden Sie zwei Schichten aus binären Neuronen. Die erste Schicht implementiert die Diskriminanzfunktion. Die zweite Schicht hat als Ausgabe für jede Klasse genau ein Ausgabeneuron, das genau dann 1 wird, wenn die Klasse vorliegt, und sonst null ist. Wie lauten dabei die Gewichte der Neuronen? Nutzen Sie dabei die Ergebnisse aus a). Wie ändert sich das Netz, wenn in der ersten Schicht Distanzneuronen verwendet werden?

Aufgabe 1.6

Man implementiere das Klassifizierungsnetz als Programm und gebe den Pseudocode an.

Aufgabe 1.7

Man beweise: Eine Ähnlichkeitsfunktion $f_k(\mathbf{x}) = a_k (\mathbf{x}-\mathbf{c}_k)^2$ impliziert

a) bei gleichen Koeffizienten $a_k=a_i$ eine Hyperebene als Klassengrenze
b) bei ungleichen Koeffizienten $a_k \neq a_i$ eine Hyperparabel als Klassengrenze

2 Lernen und Klassifizieren

Viele Funktionen menschlicher Informationsverarbeitung lassen sich durch eine Folge von Bearbeitungsschritten *(pipeline)* beschreiben. Jeder dieser sequentiellen Schritte kann durch ein neuronales Netz modelliert werden. In diesem Kapitel sollen die wichtigsten Modelle vorgestellt werden, bei denen die einzelnen formalen Neuronen oder Gruppen von Neuronen (eine *Schicht*) ohne Rückkopplung *(feedforward)* Informationen verarbeiten. Im einfachsten Fall führen dabei alle Elemente einer Schicht ihre Operationen parallel und unabhängig von einander durch.

Allerdings ist die Bezeichnung "ohne Rückkopplung" bei feedforward Netzen normalerweise nur auf die Hauptflussrichtung der Signale bezogen. Beispielsweise werden weder die Auswirkungen des neuronalen Ausgangssignals oder die des Fehlersignals nachfolgender Schichten auf das Lernen der Gewichte, noch die Wechselwirkungen der Einheiten innerhalb einer Schicht bei den Modellen dieses Kapitels als "Rückkopplung" angesehen.

Fast alle Modelle lassen sich unter einem gemeinsamen Aspekt betrachten: sie erfüllen die gewünschte Funktion nicht exakt, sondern sie approximieren sie nur, allerdings mit geringerem Aufwand. Dies ist für viele Anwendungen nicht nur ausreichend, sondern durchaus gewünscht. Dieser Vorgang kann durch Begriffe wie *Kodierung* oder *Datenkompression* beschrieben werden. Heutzutage sind gute, schnelle Kodierungsalgorithmen sind sehr wichtig, um trotz geringer Bandbreite (Kanalkapazität) hochauflösende Bilder (HDTV) und Musik übertragen oder speichern (DAT-Recorder) zu können. Bei gegebenen Ansprüchen an die Übertragungsgüte entscheidet damit das Kodierungsverfahren über den notwendigen technischen Aufwand und damit auch vielfach über den Preis (und Markterfolg!) des Gerätes.

2.1 Hebb'sches Lernen: Assoziativspeicher

Der in heutigen Rechnern übliche Hauptspeicher ist physikalisch als *Listenspeicher* organisiert: Auf M Adressen x^1 ... x^M werden *M* Inhalte y^1 ... y^M gespeichert. Zum Abruf von y^i präsentiert man die physikalische Adresse x^i und erhält wieder den Inhalt (oder einen Zeiger darauf) zurück. Will man einen Inhalt finden, ohne dessen physikalische

Adresse zu kennen (*inhaltsorientierte Adressierung),* so gibt es dabei allerdings ein Problem: ist die Liste vollkommen zufällig zusammengestellt, so muss im ungünstigsten Fall (*worst case*) die gesamte Liste durchsucht werden. Dies wirkt sich bei verschiedenen Anwendungen inhaltsorientierter Adressierung sehr ungünstig aus. Ein Beispiel dafür sind lernende Schachprogramme, bei denen zur Erkennung einer Spielsituation und ihrer möglichen Fortsetzung alle bisher gespeicherten Spielsituationen durchsucht werden müssen. Besitzt ein solches Programm wenig Vorwissen, mit dem die neue Situation geprüft werden muss, so geht die Entscheidung sehr schnell; ist dagegen schon ein "reicher Erfahrungsschatz" vorhanden, der laufend ergänzt wird, so vergrößern sich die Reaktionszeiten ebenso, bis das Programm "spielunfähig" wird. Der effektiven Organisation des Wissens und des schnellen Zugriffs im Speicher kommt deshalb eine wichtige Rolle zu.

Ein assoziativer Speicher ist aus diesen Gründen so organisiert, dass anstelle M physikalischer Adressen M inhaltsorientierte Schlüsselworte $\mathbf{x}^1 \ldots \mathbf{x}^M$ benutzt werden, zu denen die dazu assoziierten M Inhalte $\mathbf{y}^1 \ldots \mathbf{y}^M$ als M Tupel $(\mathbf{x}^i,\mathbf{y}^i)$ möglichst effektiv abgespeichert werden.
Für die interne Organisation gibt es nun verschiedene Möglichkeiten.

2.1.1 Konventionelle Assoziativspeicher

Herkömmliche Assoziativspeicher (*Content Addressable Memory*, *CAM*) bestehen im Wesentlichen aus normalen Listenspeichern (RAM), zwischen deren Speicherzellen aber noch die für die Suchoperationen (*Suchbefehle*) nötige Logik hardwaremäßig integriert ist. In Abb. 2.1 ist das Grundschema eines solchen Assoziativspeichers gezeigt.

Haben wir beispielsweise alle Angestellte einer Firma und ihr Gehalt in eine Liste eingetragen, so lässt sich durch Eingabe der Operation und ihrer Argumente, den Teildaten eines Listeneintrags z.B. (">", 5000), als Suchwort alle Angestellten mit einem Einkommen > 5000€ ermitteln. Dabei werden die fürs Suchen unwichtigen Teile der gespeicherten Information, etwa die Namen der Angestellten, durch eine Maske spezifiziert.

Der Aufwand, die Operationen hardwaremäßig für alle Speicherzellen jedes Listeneintrags (jeder Speicherzeile) zur Verfügung zu stellen, ist allerdings ziemlich groß. In Abb. 2.1 ist die Hardwareimplementierung einer Speicherzelle (Bit) mittels eines Flip-Flops gezeigt. Im Unterschied zu den konventionellen Speichern kommt beim CAM noch die Vergleichslogik dazu, die zusätzliche vertikale Datenleitungen für Masken- und Suchbitwerte erfordert.

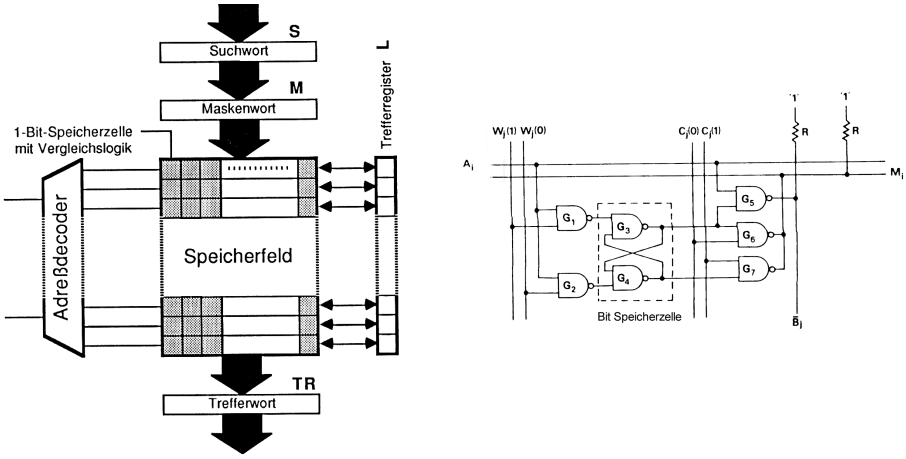


Abb. 2.1 *Architektur konventionelle Assoziativspeicher (CAM)*

Die konventionellen Assoziativspeicher (CAM) leiden unter einem für viele Anwendungen schwerwiegenden Problem: Wird nicht exakt derselbe Schlüssel präsentiert, sondern nur ein ähnlicher (z.B. ein Suchwort mit einem geänderten Buchstaben), so wird der dazu gespeicherte Inhalt nicht gefunden. Zwar lassen sich auch verschiedene Stellen des Suchworts mittels einer *Maske* als "unwichtig" erklären, aber man weiß bei einem unbekannt veränderten Suchwort nie, welche Stellen gestört sind und welche nicht.

2.1.2 Das Korrelations-Matrixmodell

Für diese Problematik haben die neuronalen Assoziativspeicher-Modelle große Vorteile. Betrachten wir dazu eines der bekanntesten der zahlreichen Modelle für assoziative Speicher, den Korrelations-Matrixspeicher. Dieses Modell hat eine lange Geschichte und existiert in verschiedenen Formulierungen, z.B. Steinbuch [STEIN61], Willshaw [WILL69], Anderson [AND72], Amari [AMA72], Kohonen [KOH72], Cooper [COOP73].

Das Modell

Seien die Eingabemuster (Ereignisse, Schlüssel) durch einen reellen Vektor $\mathbf{x} = (x_1,...,x_n)^T$ und die dazu assoziierten Ausgabemuster durch reelle $\mathbf{y} = (y_1,...,y_m)^T$ beschrieben, so lässt sich die Verknüpfung zwischen beiden Mustern linear mittels der Matrix $\mathbf{W} = (w_{ij})$ modellieren:

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{2.1}$$

mit der linearen Ausgabefunktion $y_i = S(\mathbf{w}_i^T\mathbf{x}) := \mathbf{w}_i^T\mathbf{x}$.

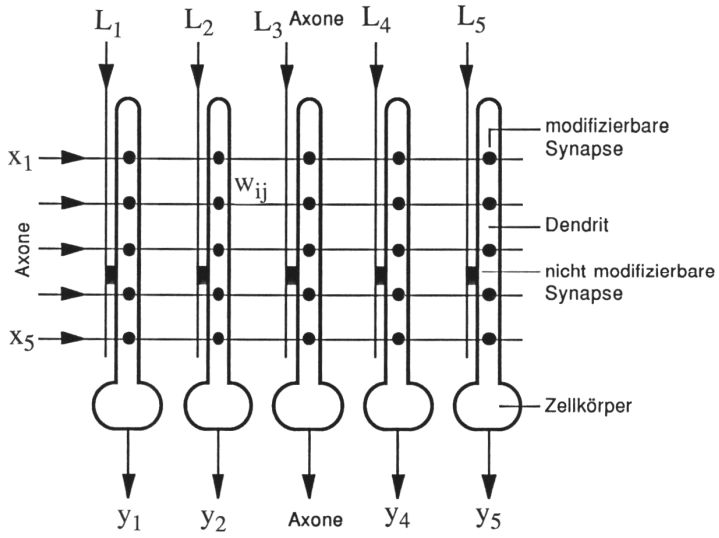


Abb. 2.2 *Hardwaremodell eines assoziativen Speichers*

In einer Implementierung würde dem Matrixkoeffizient w_{ij} hardwaremäßig die "Stärke" der Verbindung zwischen der Eingabeleitungen x_j und der Ausgabeleitung y_i entsprechen; sie als dunkler Punkt in Abb. 2.2 eingezeichnet. Die Aktivitäten (z.B. Spannungen) der einzelnen Leitungen x_j summieren sich gewichtet (z.B. als Ströme durch Widerstände) in z_i und erzeugen ein Ausgabesignal y_i.

Speichern von Mustern

Zum Speichern eines Paares $(\mathbf{x}^k,\mathbf{y}^k)$ legen wir fest, dass gleichzeitig das Schlüsselmuster $\mathbf{x}^k$ an den Eingängen und die gewünschte Lehrervorgabe $\mathbf{L}(\mathbf{x}) := \mathbf{y}^k$ direkt an dem Neuron präsentiert und die Gewichte an den Kreuzungspunkten geeignet verändert werden. Im biologisch orientierten Hardwaremodell aus Abb. 2.2 stellen die Lehrereingaben L_i sog. Kletterfasern dar, wie sie beispielsweise im Kleinhirn beobachtet werden. Welche Art der Veränderung sollen wir dabei vorsehen? Dazu orientieren wir uns an einem biologischen Vorbild. Der Physiologe *Hebb* formulierte 1949 eine Vermutung über die sy-

naptischen Veränderungen zwischen Nervenzellen, die sich als fundamental herausgestellt hat. Er schrieb in seinem Artikel [HEBB49]:

> *"Wenn ein Axon der Zelle A nahe genug ist, um eine Zelle B zu erregen und wiederholt oder dauerhaft sich am Feuern beteiligt, geschieht ein Wachstumsprozess oder metabolische Änderung in einer oder beiden Zellen dergestalt, dass A's Effizienz, als eine der auf B feuernden Zellen, anwächst."*

Diese Gedanken lassen sich auch mathematisch formulieren. Interpretieren wir die Effizienz als Gewicht zwischen A und B und die Änderung als Differenz der Gewichte zu den Zeitpunkten (t-1) und t in B, so lässt sich das Anwachsen des Gewichts w_{AB} zwischen A und B beispielsweise als Produkt der Erregungen von A und B schreiben:

$$w_{AB}(t) - w_{AB}(t\text{-}1) =: \Delta w \sim x_A\, y_B \qquad \textit{Hebb'sche Lernregel} \qquad (2.2)$$

Diese Regel lautet mit der Proportionalitätskonstanten (*Lernrate*) $\gamma(t)$

$$\Delta w(t) = \gamma(t)\, xy \qquad (2.3)$$

oder als Iterationsgleichung für jede Komponente des Eingabevektors $\mathbf{x}=(x_1,..,x_n)^T$ und des einfachen Gewichtsvektors $\mathbf{w}_i=(w_1,..,w_n)^T$ des i-ten Neurons

$$w_{ij}(t) = w_{ij}(t\text{-}1) + \gamma(t)\, y_i x_j \qquad (2.4)$$

bzw. $$w_i(t) = w_i(t\text{-}1) + \gamma(t)\, y_i x \qquad \textit{Iterative Hebb'sche Lernregel} \qquad (2.5)$$

Bei *Synapsen höherer Ordnung* wird zusätzlich zur Korrelation der j-ten Eingabekomponente x_j mit der Ausgabe y_i des i-ten Neurons auch die Korrelationen zu den anderen Komponenten x_k der Eingabe gelernt [GIL87]. Für Synapsen zweiter Ordnung modifiziert sich also zu

$$w_{ijk}(t) = w_{ijk}(t\text{-}1) + \gamma(t)\, y_i x_j x_k \qquad (2.6)$$

Ist die erwünschte Ausgabe die Lehrervorgabe L, so ist

$$\Delta w_{ij} \sim L_i x_j \qquad (2.7)$$

Nach dem Speichern von *M* Mustern $\mathbf{x}^1,...,\mathbf{x}^M$ resultieren mit der Proportionalitätskonstanten γ_k die Gewichte

$$w_{ij} = \sum_k \Delta w_{ij} = \sum_k \gamma_k L_i^k\, x_j^k \qquad w_{ij}(0) := 0 \qquad (2.8)$$

und die Gesamtmatrix $\mathbf{W}$ aller Gewichte w_{ij} ist somit definiert als

$$\mathbf{W} = \sum_k \gamma_k \mathbf{L}^k (\mathbf{x}^k)^T \qquad (2.9)$$

als äußeres Produkt der Eingabevektoren und der dazu gewünschten Ausgabevektoren darstellbar.

Haben wir mit der Definition (2.9) eine sinnvolle Gewichtsmatrix gefunden? Um dies nachzuprüfen testen wir die Matrix mit unserer Anforderung: Bei Eingabe des Schlüssels **x** muss das gewünschte Ergebnis **L** am Ausgang **y** ausgegeben werden.

Auslesen eines Musters

Wird einem solchen System ein bereits gespeicherter Schlüssel $\mathbf{x}^r$ präsentiert, so ergibt sich als Ausgabe

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{z} = \gamma_r \mathbf{L}^r(\mathbf{x}^r)^T\mathbf{x}^r + \sum_{k\neq r} \gamma_k \mathbf{L}^k \left(\mathbf{x}^k\right)^T \mathbf{x}^r \qquad (2.10)$$

assoziierte Antwort + Übersprechen von anderen Mustern

Normieren wir die Gewichte mit $\gamma_k := 1/((\mathbf{x}^k)^T\mathbf{x}^k) = |\mathbf{x}^k|^{-2}$ und verwenden wir ein System von orthogonalen Schlüsselvektoren $\mathbf{x}^k$, so wird mit $(\mathbf{x}^i)^T\mathbf{x}^j = 0$ bei $i\neq j$ der Anteil des Übersprechens von anderen, gespeicherten Mustern null und wir erhalten wieder als Antwort das zu $\mathbf{x}^r$ ursprünglich assoziierte Muster $\mathbf{y}^r = \mathbf{L}^r$. Das gleiche Ergebnis ergibt sich auch, wenn wir bei $\gamma_k := 1$ stattdessen die Eingabewerte normieren ($|\mathbf{x}^k| = 1$).
In dem nachfolgenden Programmbeispiel werden die Schlüsselmuster **x** und die gewünschte Ausgabe **L** symbolisch als ganze Tupel eingelesen. Diese Zahlentupel aus *n* bzw. *m* Zahlen kann man natürlich auch sequentiell mit einer Schleife eingeben.

```
AMEM:      (* Implementiert einen Korrelationsspeicher *)
VAR     (* Datenstrukturen *)
   x:       ARRAY[1..n] OF REAL;                (* Eingabe *)
   y,L:     ARRAY[1..m] OF REAL;                (* Ausgaben *)
   W:       ARRAY[1..m,1..n] OF REAL;           (* Gewichte *)
   γ:       REAL;                               (* Lernrate *)
BEGIN
   γ:= 0.1;                                (*  Lernrate festlegen:  |x|^2=10  *)
   initWeights( w,0.0);                    (* Gewichte initialisieren *)
   REPEAT   (* Abspeichern der Muster *)
      Read(PatternFile,x,L)(* Schlüssel & dazu gewünschte Ausgabe einlesen *)
      FOR i:=1 TO m DO
         FOR j:=1 TO n DO                  (* Gewichte verändern *)
              W[i,j] := W[i,j]+γ*L[i]*x[j] (* nach Gl.(2.8) *)
         ENDFOR;
      ENDFOR;
   UNTIL EndOf(PatternFile)

   LOOP  (* zum Schlüssel x das gespeicherte y assoziieren *)
      Input(x);
```

```
      FOR i:=1 TO m DO                (* Auslesen für alle Neuronen *)
         y[i] := S(z(W[i],x))         (* Gl.(2.1)*)
      ENDFOR;
      Print(y);
   ENDLOOP;
END AMEM.
```

Codebeispiel 2.1 *Speichern und Auslesen im Assoziativspeicher*

Fehlerhafte Eingabe

Was geschieht, wenn wir ein vom gespeicherten Schlüsselmuster $\mathbf{x}^r$ abweichendes Muster $\mathbf{x} := \mathbf{x}^r + \tilde{\mathbf{x}}$ dem linearen System präsentieren ?
Sei die Abweichung mit $\tilde{\mathbf{x}}$ bezeichnet, so resultiert als Ausgabe

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{W}(\mathbf{x}^r + \tilde{\mathbf{x}}) =: \mathbf{y}^r + \tilde{\mathbf{y}} \qquad (2.11)$$

Original + *Störterm*

die Überlagerung aus der zu $\mathbf{x}^r$ assoziierten, starken Antwort und einem kleineren "Störterm", der aus einer Linearkombination aller gespeicherten Ausgabemuster $\mathbf{y}^k$ gebildet wird.

Da bei der Ausgabe die Störungen meist schwächer als die gewünschten Originalmuster sind, kann man die korrekte Ausgabe automatisch dadurch erhalten, dass man alle Komponenten von $\mathbf{y}$ vor der Ausgabe einer Schwellwertoperation unterwirft. Ist die Schwelle s_i geeignet gewählt, so werden nur stärksten Komponenten sie überschreiten und das korrekte, nur um den Schwellwert verminderte $\mathbf{y}^r$ produzieren [AMA72]. Dies ist die Grundidee des linearen Assoziativspeichers mit Schwellwert.

Der lineare Speicher mit Schwellwert

Bisher betrachteten wir reelle Vektoren $\mathbf{x}$ und $\mathbf{y}$. Identifizieren wir die reellen Werte mit den Spikefrequenzen der Neuronen, so beschränkt sich der Wertebereich der $\mathbf{x}$ und $\mathbf{y}$ auf positive Zahlen, da keine negativen Spikefrequenzen existieren.

Betrachten wir nun den Fall, dass bei beliebigen Eingabemustern $\mathbf{x}^k$ die Ausgabemuster mit $(\mathbf{L}^k)^T\mathbf{L}^r = 0 \;\forall r \neq k$, orthogonal kodiert sind (orthogonale Projektion der $\mathbf{x}$ auf $\mathbf{y}$). Dann muss, damit $(\mathbf{L}^k)^T\mathbf{L}^r = \Sigma_i \mathbf{L}^k{}_i\mathbf{L}_i{}^r = 0$ gilt, bei jedem Produkt $\mathbf{L}^k{}_i\mathbf{L}_i{}^r$ ein Faktor oder beide null sein. Es kann für eine Ausgabekomponente *i* nur maximal ein $\mathbf{y}^r$ geben, bei dem die Komponente ungleich null ist. Damit ist in Gleichung (2.11) für jede Komponente maximal ein Term der Summe ungleich null mit

$$z_i = \gamma_r L_i{}^r (\mathbf{x}^r)^T\mathbf{x} \qquad \text{mit} \quad r \in [1..m] \qquad (2.12)$$

Verwenden wir beliebige (und nicht wie Kohonen orthogonale) $\mathbf{x}^k$ und binäre L_i, so resultiert als Aktivitätsvektor $\mathbf{z}$ ein Vektor, der in jeder Komponente z_i das innere Produkt

(Korrelation) aus dem Eingabevektor **x** und dem einzigen Schlüssel $\mathbf{x}^r$, der zu einem **y** mit der Komponente y_i ungleich null assoziiert wurde.

Damit ist das Ausgabemuster in jeder einzelnen Komponente y_i eine Funktion des inneren Vektorprodukts (Skalarprodukt), das die Kreuzkorrelation zwischen Inputmuster **x** und einem der Speichermuster $\mathbf{x}^r$ darstellt

$$y_i = S(z_i) = S(\gamma_r(\mathbf{x}^r)^T\mathbf{x}-s_i) \qquad \textit{Auslesen mit Schwellwert} \qquad (2.13)$$

beispielsweise die binäre Ausgabe $S(.) = S_B(.)$ zur Unterdrückung des Übersprechens.
Wann erhalten wir nun die „richtige" Antwort aus dem Assoziationsspeicher? Grundsätzlich haben wir das Problem, dass wir eine Klassentrennung mit einer Hyperebene nach Definition 1.4 durchführen wollen. Da für eine Klasse *r* nur die Komponenten mit $L^r = 1$ die Ausgabe $S(z_i) = 1$ haben sollen, muss

$$L_i^r\,(\mathbf{x}^r)^T\mathbf{x}-s_i > L_j^k\,(\mathbf{x}^k)^T\mathbf{x}-s_j \quad \text{für alle } k \neq r \qquad (2.14)$$

gelten. Die Korrelation des Eingabemusters mit dem gespeicherten Muster muss größer sein als mit allen anderen Mustern, um richtig eingeordnet zu werden.
Bei zwei Klassen $\mathbf{x}^r$ und $\mathbf{x}^k$ ist dies kein Problem; die Klassengrenze liegt zwischen $\mathbf{x}^r$ und $\mathbf{x}^k$. Haben wir mehr als zwei Klassen, etwa noch eine dritte und vierte Klasse $\mathbf{x}^p$ und $\mathbf{x}^q$, so ist dies nicht mehr so einfach möglich: Es muss nicht immer eine einzige Klassengrenze geben, die die betrachtete Klasse gegen alle anderen Klassen abtrennt, siehe Abb. 2.3. Hier kann die gestrichelte Linie die Klasse *p* nicht von allen anderen Klassen abtrennen; man benötigt mehrere Klassentrennlinien.

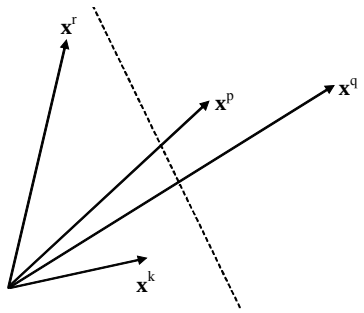


Abb. 2.3 *Klassentrennung bei mehreren Klassen*

Man kann nun nicht nur eine einzige Eins für eine Klasse in der Ausgabe definieren, sondern mehrere. Jede Komponente der Ausgabe implementiert dann eine Entscheidung, ob das Eingabemuster diesseits oder jenseits der Hyperebene liegt. Die Kombination aller Entscheidungen ist dann ein Muster für die gewünschte Klasse. In Abb. 2.4 sind für eine Ausgabe **y** mit zwei Ausgabevariablen (y_1,y_2) die vier möglichen Klassen gezeigt.

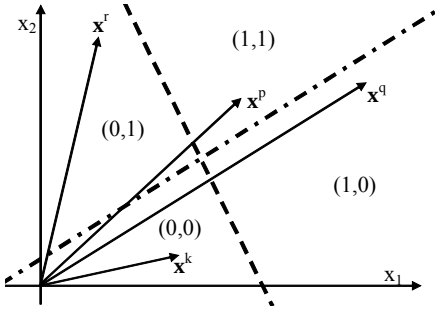


Abb. 2.4 *Klassentrennung und Klassenkodierung durch zwei Hyperebenen*

Allgemein lassen sich mit dieser Methode bei $m = \dim(\mathbf{y})$ Ausgabevariablen und damit m Hyperebenen maximal 2^m Klassen unterscheiden. Die dazu nötigen Parameter der Klassengrenzen lassen sich allerdings nicht mehr so einfach in einem Schritt über die Hebb'sche Lernregel bestimmen.
Die Trennung der Muster des Eingaberaums in verschiedene Klassen durch die Auswertung der Korrelationen ist allerdings nicht unproblematisch. Betrachten wir dazu Abb. 2.5.

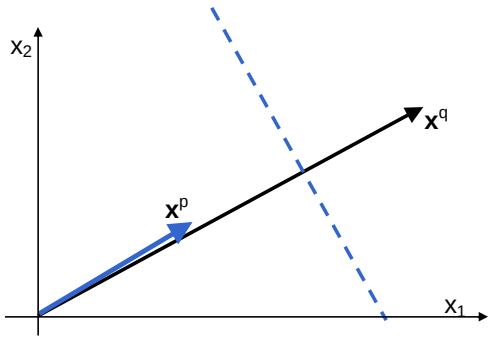


Abb. 2.5 *Klassentrennung durch Korrelation?*

Angenommen, wir möchten ein Muster, etwa den Klassenprototypen $\mathbf{x}^p$ selbst, einordnen. Zu welcher Klasse wird $\mathbf{x}^p$ zugeordnet? Wir entscheiden mit Hilfe der Korrelation nach folgender Regel

$$\begin{aligned} \mathbf{x}^{pT}\mathbf{x} > \mathbf{x}^{qT}\mathbf{x} &\Rightarrow \mathbf{x} \text{ gehört zu Klasse p,} \\ \mathbf{x}^{pT}\mathbf{x} < \mathbf{x}^{qT}\mathbf{x} &\Rightarrow \mathbf{x} \text{ gehört zu Klasse q.} \end{aligned} \tag{2.15}$$

Da für das Skalarprodukt $\mathbf{x}^{pT}\mathbf{x}^{p} \leq \mathbf{x}^{pT}\mathbf{x}^{q}$ gilt, wird sogar der Klassenprototyp $\mathbf{x}^{p}$ zur Klasse *q* zugeordnet – absolut nicht das, was wir von einer sinnvollen Mustererkennung und Klassifizierung erwarten.
Der Fehler dabei liegt darin, dass wir als Klassifizierungskriterium den Abstand des Musters zu den Klassenprototypen verwenden wollen, aber dies mit Hilfe der Korrelation umsetzen. Dem kleineren Abstand $(\mathbf{x}^{p}-\mathbf{x})$ von $\mathbf{x}$ zu dem Klassenprototypen $\mathbf{x}^{p}$ gegenüber dem Abstand $(\mathbf{x}^{q}-\mathbf{x})$ zum Klassenprototypen $\mathbf{x}^{q}$

$$|\mathbf{x}^{p}-\mathbf{x}| < |\mathbf{x}^{q}-\mathbf{x}| \tag{2.16}$$

entspricht nur dann die größere Korrelation $\mathbf{x}^{T}\mathbf{x}^{p} > \mathbf{x}^{qT}\mathbf{x}$, wenn gilt

$$(\mathbf{x}^{p}-\mathbf{x})^2 = (\mathbf{x}^{p})^2 - 2\mathbf{x}^{p}\mathbf{x} + \mathbf{x}^2 < (\mathbf{x}^{q})^2 - 2\mathbf{x}^{q}\mathbf{x} + \mathbf{x}^2 = (\mathbf{x}^{q}-\mathbf{x})^2$$

oder $$2\mathbf{x}^{p}\mathbf{x} > 2\mathbf{x}^{q}\mathbf{x} + (\mathbf{x}^{p})^2 - (\mathbf{x}^{q})^2$$

erfüllt ist. Gehen wir davon aus, dass alle Klassenprototypen gleich lang sind,

$$|\mathbf{x}^{p}| = |\mathbf{x}^{q}| \quad \forall p,q$$

so erhalten wir unabhängig von den Klassen die Bedingung

$$\mathbf{x}^{p}\mathbf{x} > \mathbf{x}^{q}\mathbf{x} \tag{2.17}$$

so dann auch kleinster Abstand (Gl. (2.22)und maximale Korrelation (Gl.(2.17)) die selbe Klasseneinteilung bewirken.
Wie erhalten wir nun Prototypen und Muster konstanter Länge? Dazu greifen wir zu einem Trick: Wir erweitern die Mustervektoren um eine Komponente x_{n+1} genau so, dass die Gesamtlänge des Vektors normiert wird

$$\mathbf{x} \rightarrow \mathbf{x}' = (x_1,x_2,...,x_n,x_{n+1}) \text{ mit } |\mathbf{x}'| = \text{const} = c \tag{2.18}$$

und somit

$$x^2_{n+1} = c^2 - (x^2_1+x^2_2+...+x^2_n) = c^2 - |\mathbf{x}|^2 \geq 0 \tag{2.19}$$

Die Norm *c* sollte groß genug sein, um positives x^2_{n+1} zu erreichen; ansonsten erhalten wir eine komplexe Zahl. Damit betten wir den *n*-dimensionalen Musterraum in einen höherdimensionalen *n+1*-dimensionalen Musterraum ein. Geometrisch können wir uns das so vorstellen, als ob eine zwei-dimensionale Fläche auf eine Kugeloberfläche abgebildet wird. Die Endpunkte aller Mustervektoren liegen auf der Oberfläche der Kugel; die Kugel hat dabei den Radius *c*. In Abb. 2.6 ist dies visualisiert.

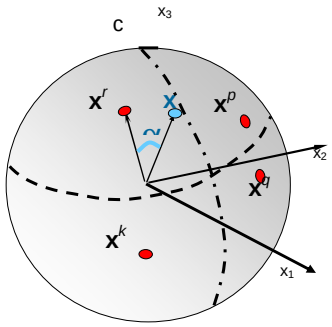


Abb. 2.6 *Projektion der Klassen auf eine Kugeloberfläche*

Die Entscheidung darüber, ob ein Muster **x** zu einer Klasse *r* gehört oder nicht, fällt damit beim Test der Korrelation $\mathbf{x}^r\mathbf{x}$. Nun gilt aber für einen Winkel α zwischen dem Mustervektor **x** und dem Klassenprototyp $\mathbf{x}^r$

$$\cos(\alpha) = \frac{\mathbf{x}^T\mathbf{x}^r}{|\mathbf{x}||\mathbf{x}^r|} = \frac{1}{c^2}\mathbf{x}^T\mathbf{x}^r \tag{2.20}$$

Der Kosinus im Bereich $[0,\pi]$ ist eine monoton fallende Funktion des Winkels α. Bei $\alpha = 0$ ist die Korrelation maximal, bei $\alpha = \pi$ minimal, so dass bei konstanten Musteraktivitäten *c* auch der Winkel α zwischen den Mustervektoren als Abstandsmaß für eine Klassentscheidung benutzt werden kann.

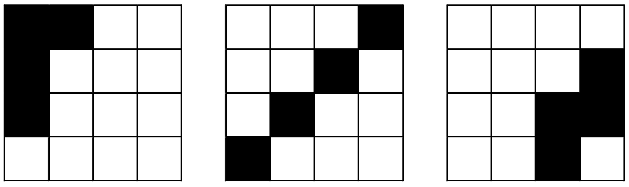
Autoassoziative Musterergänzung und Störunterdrückung

Mit der Einführung einer Schwelle und einer binären Ausgabefunktion zerfällt also die Menge aller möglichen Eingabemuster in Untermengen (Klassen), die durch die gespeicherten Muster als Klassenprototypen festgelegt sind. Die Ausleseoperation des Assoziativspeichers wird damit zu einer Mustererkennungsoperation. Hierbei erregt das Eingabemuster dasjenige Ausgabemuster, das zu dem Klassenprototypen mit der größten Ähnlichkeit zum Eingabemuster assoziiert ist. Die Tatsache, dass vom gespeicherten Muster abweichende Eingabemuster dieselbe korrekte Ausgabe bewirken, lässt sich auch als *Toleranz* gegenüber fehlerhaften Daten *interpretieren.*

Assoziiert man zu jedem Eingabemuster $\mathbf{x}^k$ das gleiche Muster als Ausgabe $\mathbf{y}^k = \mathbf{x}^k$ (Autoassoziativer Speicher), so bedeutet die Toleranz gegenüber fehlerhaften Daten eine Korrektur- und Ergänzungsoperation der Eingabedaten. Dies soll im folgenden Beispiel verdeutlicht werden.

Beispiel

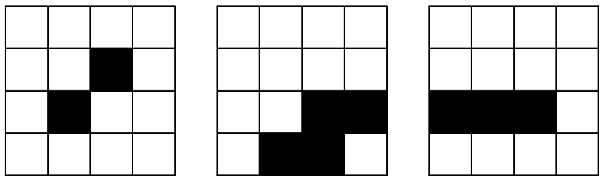
Zur Verdeutlichung der Fähigkeiten des korrelativen Assoziativspeichers geben wir uns vier binäre Muster vor, die autoassoziativ gespeichert werden sollen. Die drei Muster $\mathbf{x}^1 .. \mathbf{x}^3$ mit n = 16 seien als 4x4 Matrix in Abb. 2.7 visualisiert.



$\mathbf{x}^1$ = (1100 1000 1000 0000) $\mathbf{x}^2$ = (0001 0010 0100 1000) $\mathbf{x}^3$ = (0000 0001 0011 0010)

Abb. 2.7 *Visualisierung von drei Mustern*

Wie man sieht, sind die vier Muster *orthogonal*: sie "überlappen" sich visuell in keiner Komponente. Angenommen, die Muster seien mit $\mathbf{L}^k = \mathbf{x}^k$ autoassoziativ abgespeichert. In der folgenden Abb. 2.8 sind weitere drei verschiedene Muster für die Eingabe gezeigt.



$\mathbf{x}$ = (0000 0010 0100 0000) $\mathbf{x}$ = (0000 0000 0011 0110) $\mathbf{x}$ = (0000 0000 1110 0000)

Abb. 2.8 *Auto-Assoziative Operationen im nicht-rückgekoppelten Speicher*

Die Muster seien mit der Hebb'schen Regel nach Gl. (2.9) in der Gewichtsmatrix

$$\mathbf{W} = \mathbf{x}^1\mathbf{x}^{1T} + \mathbf{x}^2\mathbf{x}^{2T} + \mathbf{x}^3\mathbf{x}^{3T}$$

abgespeichert. Geben wir nun ein Muster ein, das einem der Gespeicherten ähnlich ist, so resultiert eine Aktivität $\mathbf{z}$.

$$\mathbf{z} = \mathbf{x}^1(\mathbf{x}^{1T}\mathbf{x}) + \mathbf{x}^2(\mathbf{x}^{2T}\mathbf{x}) + \mathbf{x}^3(\mathbf{x}^{3T}\mathbf{x}) \qquad (2.21)$$

Diese Aktivität $\mathbf{z}$ ist in einer 4×4-Matrix angeordnet für die Eingabe der drei Muster in Abb. 2.9 gezeigt. Für die Ausgabe haben wir eine einheitliche Sc hwelle von

s = 0. Die dunklen Felder des resultierenden Musters sind in die Abbildung eingezeichnet.

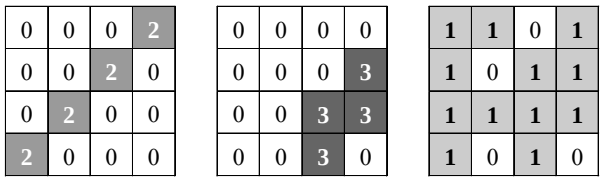


Abb. 2.9 *Aktivität der Ausgabematrix*

Wählen wir mit $\gamma = 1$

- bei dem ersten Muster die Schwellen s = 0, so erhalten wir als Ausgabe $\mathbf{x}^2$: Das Eingabemuster wurde um das fehlende Feld ergänzt.
- Beim zweiten Muster mit s = 0 erhalten wir $\mathbf{x}^3$ als Ausgabe: Es wurden nicht nur fehlende Felder ergänzt, sondern auch ein zusätzliches Feld (Rauschen) nicht weiter „beachtet".
- Erst beim dritten Muster haben wir Probleme: Es kann sowohl ein Fragment von $\mathbf{x}^1$, $\mathbf{x}^2$ oder auch von $\mathbf{x}^3$ sein. Es liegt damit auf der Grenzfläche zwischen den Klassen 1, 2 und 3 und kann keiner Klasse mit Sicherheit zugeordnet werden.

Die Erklärung für die jeweilige Zuordnung steckt in der Korrelation des Eingabemusters mit den jeweiligen Klassenrepräsentanten $\mathbf{x}^r$. In

Korrelation	Test 1	Test 2	Test 3
$\mathbf{x}^1\mathbf{x}$	0	0	1
$\mathbf{x}^2\mathbf{x}$	**2**	0	1
$\mathbf{x}^3\mathbf{x}$	0	**3**	1

Abb. 2.10 *Korrelationen der Eingabe mit den gespeicherten Mustern*

Man beachte dabei die Tatsache, dass eine kleine Verschiebung um einen Pixel aus beispielsweise $\mathbf{x}^1$ sofort ein Muster erzeugt, das die minimale Korrelation zu $\mathbf{x}^1$ hat: die visuelle Ähnlichkeit entspricht absolut nicht dem Ähnlichkeitsmaß der Korrelation. Für eine Verarbeitung von beispielsweise visuellen Daten in Assoziativspeichern muss deshalb vorher eine spezielle Kodierungsoperation erfolgen, die ähnliche Eingaben des betrachteten Problems (z.B. visuelle Ähnlichkeit) in ähnliche

Speichermuster (bzgl. Abstand etc.) übersetzt. Dieses Problem wird weiter hinten bei der *invarianten Mustererkennung* behandelt.

Den Mechanismus der Datenergänzung von unvollständigen Daten kann man dazu verwenden, Tupel von Daten, beispielsweise Relationen zwischen zwei Objekten *(Relation, Objekt1, Objekt2)* zu speichern. Ein Auslesen bzw. Ergänzen eines unvollständigen Tupels *(Relation, - , Objekt2)* wird damit zu einer relationalen Datenbankabfrage für *Objekt1*, s.[HIN81b].

2.1.3 Die Speicherkapazität

Sei eine Sequenz von M Mustertupeln $(\mathbf{x}^1,\mathbf{y}^1)$, .., $(\mathbf{x}^M,\mathbf{y}^M)$ gegeben. Wie viele dieser Tupel können in einem Korrelationsspeicher zuverlässig gespeichert werden?

Betrachten wir dazu zwei Extremfälle; zum einen, wenn $\mathbf{x}^1 = .. = \mathbf{x}^M$ gilt und zum anderen, wenn $\mathbf{y}^1 = .. = \mathbf{y}^M$. Im ersten Fall kann kein wie auch immer konstruierter, beliebiger, assoziativer Speicher bei Eingabe eines $\mathbf{x}^i$ das dazu gehörende $\mathbf{y}^i$ auslesen, so dass maximal $M_{max} = 1$ Tupelpaar gespeichert werden kann. Im zweiten Fall aber können beliebig viele Paare gespeichert werden, da jede Antwort $\mathbf{y}^i$ immer richtig ist. Die Speicherkapazität ist also nicht nur vom Speichermodell, sondern auch von der Art (Kodierung) der zu speichernden Mustern abhängig.

Kodierung

Welche Kodierung sollte man wählen? Ein gutes Kriterium ist zweifelsohne die Forderung, dass die Muster gleichmäßig unterschiedlich sein sollen, also alle Muster einen gewissen Abstand $d(\mathbf{x}^r,\mathbf{x}^k) = (\mathbf{x}^r-\mathbf{x}^k)^2$ voneinander haben sollen.
Der *maximale* Abstand bei fester Länge der Vektoren (feste Anzahl a von Einsen) ist bei orthogonalen Mustern gegeben:

$$\max d(\mathbf{x}^r,\mathbf{x}^k) = \max\left((\mathbf{x}^r)^2 + (\mathbf{x}^k)^2 - 2\mathbf{x}^r\mathbf{x}^k\right) = 2a \quad \text{mit} \quad a \equiv (\mathbf{x}^i)^2 \text{ und } \mathbf{x}^r\mathbf{x}^k = 0$$

Falls die Muster n Komponenten (Dimensionen) haben, gibt es bei n reellen Komponenten maximal n orthogonale Vektoren; sie bilden Basisvektoren im n-dimensionalen Musterraum. Bestehen die Muster nur aus Einsen und Nullen, so sind es noch weniger. Bei a Einsen pro Klassenprototyp dürfen die anderen Klassenprototypen an diesen Komponenten nur Nullen haben. Hat jeder Klassenprototyp die gleiche Zahl a von Einsen, so sind bei n Komponenten insgesamt maximal

$$M_{max} = \left\lfloor \frac{n}{a} \right\rfloor \qquad \lfloor\ \rfloor \hat{=} \text{kleinste ganze Zahl} \qquad (2.22)$$

Klassen möglich. Dies ist eine ziemlich geringe Anzahl; für beispielsweise 100 Eingänge und a = 10 Einsen pro Muster gibt es nur 10 mögliche orthogonale Muster.

Schwächen wir die Forderung, den Abstand zu maximieren, etwas ab und verlangen nur *im Mittel* einen möglichst großen Abstand bei sonst zufälligen Mustern, so ist der *maximale erwartete* Abstand bei zufälligen Mustern von *im Mittel* konstanter Aktivität

$$\langle d(\mathbf{x}^r,\mathbf{x}^k)\rangle = 2a - 2\langle(\mathbf{x}^r)^T\mathbf{x}^k\rangle = 2a - 2\sum_i \langle x_i^r\rangle\langle x_i^k\rangle$$

Mit ist

$$\langle x_i\rangle = 0\cdot P(0) + 1\cdot P(1) = \frac{1}{M} = \frac{a}{n}$$

$$\langle d(\mathbf{x}^r,\mathbf{x}^k)\rangle = 2a - 2n\ \frac{a}{n}\ \frac{a}{n} = 2\left(a - \frac{a^2}{n}\right)$$

und mit der Maximums-Bedingung

$$\frac{\partial\left\langle d\left(\mathbf{x}^r,\mathbf{x}^k\right)\right\rangle}{\partial a} I_{a=a^*} = 2\left(1 - 2\frac{a^*}{n}\right) \overset{!}{=} 0 \tag{2.23}$$

$$\text{bei } a^* = n/2 \text{ zu } d = n/2 \tag{2.24}$$

gibt es

$$M = \binom{n}{a^*} = \binom{n}{n/2} \tag{2.25}$$

verschiedene Muster.

Wie sich leicht nachprüfen lässt (Übungsaufgabe!), ist a^* nicht nur die optimale Aktivität für einen maximalen, erwarteten Abstand, sondern maximiert auch gleichzeitig die Anzahl der verschiedenen Muster.

Spärliche Kodierung in binären Speichern

Die beiden obigen Kodierungen optimieren die Fehlertoleranz bzw. die Kapazität der Kodierung. Für das Modell eines binären Speichers hat sich eine dritte Art von Kodierung als praktisch erwiesen: die *spärliche* Kodierung (*sparse coding*) mit sehr geringer Aktivität. Referieren wir dazu kurz die Eigenschaften des binären Speichers.

Für den Fall binärer Muster ($\mathbf{x}$ aus $\{0,1\}^n$ und $\mathbf{y}$ aus $\{0,1\}^m$) wird beim Speicher mit binären Gewichten $w_{ij} \in \{0,1\}$ die Speicherregel (2.8) zu

$$w_{ij} = \bigvee_k y_i^k x_j^k = \max_k y_i^k x_j^k, \tag{2.26}$$

Das Auslesen geschieht über die Schwellwertregel (2.13).

Für den Grenzfall sehr großer Speicher konnte Palm 1980 [PALM80] zeigen, dass die maximale Speicherkapazität

$$H_B = \ln 2 = 0{,}693 \text{ Bit pro Speicherzelle} \tag{2.27}$$

beträgt, wobei für die Aktivitäten $a_x := |\mathbf{x}|$ und $a_y := |\mathbf{y}|$, mit den Dimensionen *n* und *m* der Vektoren und für die Anzahl *M* der gespeicherten Muster die Beziehungen gelten

$$a_x = \mathrm{ld}\, m,\ a_y = O(\log n),\ M \approx \frac{mn \ln(2)}{a_y \mathrm{ld}(m)} \tag{2.28}$$

Bei n = 10.00 Eingabeleitungen bedeutet dies nur eine Aktivität von $a_x \approx 1\%$!
Für den *autoassoziativen* Speicher mit **x** = **y** ist die Speicherkapazität halbiert.

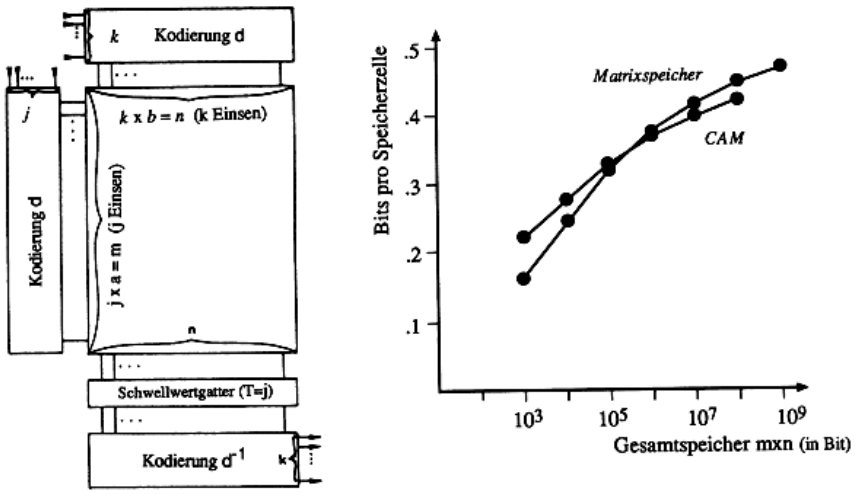


Abb. 2.11 *Funktionsblöcke und Speicherkapazität eines binären, assoziativen Speichers (nach [PALM84])*

Für den Normalfall endlicher Speichermatrizen fand Palm eine geringere Speicherkapazität. In Abb. 2.11 ist dies verdeutlicht. Links ist das Funktionsbild eines Speichers mit spärlicher Kodierung gezeigt, rechts die Information pro Speicherzelle in Abhängigkeit von der Matrixgröße für den Matrixspeicher und, zum Vergleich, für den konventionellen CAM. Die spärliche Kodierung d wird hier durch die Zuordnung jedes der *b* (bzw. *a*) möglichen Werte von einer der *k* (bzw. *j*) Eingabeleitungen von **x** (bzw. **y**) zu der Aktivierung von einer aus *b* (bzw. *a*) vorhandenen, binären, parallelen Leitungen erreicht. Damit haben die zu speichernden Muster immer genau *k* (bzw. *j*) Einsen.
Beim Auslesen wird über eine Rücktransformation das "spärliche" Datenwort wieder in ein "normal dichtes" umgewandelt. Aus Abbildung ist ersichtlich, dass dabei die Kapazität des binären, assoziativen Speichers durchaus mit einem konventionellen CAM-

Speicher vergleichbar, und bei höheren Speichermengen, wie in der Abbildung zu erkennen, sogar besser ist.

Obwohl der Fall binärer Gewichte für die Hardwareimplementierung zunächst am einfachsten und damit am aktuellsten zu sein scheint, wollen wir auch nicht-binäre, reelle Gewichte betrachten. Sehen wir die Gewichte als analoge, reelle Größen an, so lassen sich in einem endlichen Intervall [0..R], wie wir aus der mathematischen Zahlentheorie wissen, stets unendlich viele Zahlen (Zustände) unterbringen. Nach unserer Definition 1.4 von Information hat damit jedes Gewicht prinzipiell auch *unendlich viel Information*. In der Realität stimmt dies natürlich nicht, da wir den Zustand eines Gewichts nur mit endlicher Präzision festlegen können und so nur eine endliche, unterscheidbare und - je nach Implementierung - reproduzierbare Zahl von Zuständen erhalten.

Eine sinnvolle Einschränkung für die Gewichte bedeutet die Wahl von binären Variablen $\mathbf{x}$ aus $\{0,1\}^n$ und $\mathbf{y}$ aus $\{0,1\}^m$. Durch die endliche Anzahl der möglichen Paare $(\mathbf{x},\mathbf{y})$ gibt es auch nur eine endliche Anzahl möglicher Gewichtswerte der Speichermatrix, egal ob diese reelle oder nur ganzzahlige Werte annehmen können. Diese Art von Speichermatrix wurde von Bottini 1988 in [BOTT88] untersucht. Für die Speicherkapazität fand er bei spärlicher Kodierung im Grenzfall $\mu:=P(y_i=1)$ gegen Null und $M \to \infty$

$$H = ld(e)/2 = 0{,}72 \text{ Bit pro Element} \qquad (2.29)$$

was in der gleichen Größenordnung wie beim binären Speicher liegt. Eine binäre Implementierung von Gewichten bedeutet also keine wesentliche Einschränkung der Speicherkapazität. Dies trifft auch für rückgekoppelte Speichersysteme zu.

2.1.4 Andere Modelle

Das Korrelationsmodell des assoziativen Speichers ist in den Grundfunktionen vieler anderer Modelle enthalten. Im Folgenden soll dies an einigen Beispielen näher erläutert werden.

Der Konvolutionsspeicher

Da beispielsweise Korrelation und Konvolution eng zusammenhängen, hat das *Konvolutionsmodell* von Bottini [BOTT80] trotz unterschiedlicher Speichermechanismen die gleiche Speicherkapazität (2.29). Das Konvolutions-Speichermodell geht dazu von einer binären Kodierung der Schlüssel $\mathbf{x}^k \in \{-a,+a\}^n$ und der Inhalte $\mathbf{y}^k \in \{0,1\}^m$ aus.
Zum *Speichern* wird

$$\Delta w_j^k = \sum_{i=1}^{m} y_i^k x_{j-i}^k$$

gebildet, wobei nach dem Anlegen von M Mustern $\mathbf{x}^1,\ldots,\mathbf{x}^M$ die Gewichte resultieren

$$w_j = \sum_{k=1}^{M} \Delta w_j^{\,k}$$

Zum *Auslesen* bildet man

$$z_i^{\,r} = \sum_{j=1}^{n} w_j x_{j-i}^{\,r} \qquad \text{für } i = 1..m$$

Da $\langle \mathbf{x}^k \rangle = 0$ gilt, kann mit einer Schwelle *s* das gespeicherte Inhaltswort

$$y_i^{\,r} = \langle z_i^{\,r} \rangle$$

als Erwartungswert ausgelesen werden.

Bei einem in g Komponenten ungestörten Schlüssel **x** ergibt sich nach [BOTT88] für das Konvolutionsmodell als maximale Speicherkapazität im Grenzfall $\mu := P(y_i=1)$ gegen Null und $M \to \infty$

$$H = g^2(\mathrm{ld}(e)/\pi)(1-\mu) \text{ Bits pro Element} \qquad (2.30)$$

was auch für das Korrelationsmodell gilt.

Aufgaben

2.1.1) Man zeige: a^* aus Gleichung (2.24) gibt nicht nur die optimale Aktivität für einen maximalen, erwarteten Abstand, sondern maximiert auch gleichzeitig die Anzahl der verschiedenen Muster

2.1.2) Programmieren Sie einen Assoziativspeicher für 3 Muster und speichern Sie in einem Lernzyklus die Muster

$\mathbf{x}^1 = (0\ 1\ 0)$, $\mathbf{y}^1 = (1\ 0)$; $\mathbf{x}^2 = (1\ 0\ 0)$, $\mathbf{y}^2 = (0\ 1)$; $\mathbf{x}^3 = (0\ 0\ 1)$, $\mathbf{y}^3 = (1\ 1)$;

ab. Testen Sie ihren Speicher durch Eingabe der Schlüssel $\mathbf{x}^i$.

2.1.3) Versuchen Sie nun, stattdessen die Tupel $\mathbf{x}^1 = (0\ 1\ 1)$, $\mathbf{y}^1 = (1\ 0)$ und $\mathbf{x}^2 = (1\ 1\ 0)$, $\mathbf{y}^2 = (0\ 1)$ abzuspeichern und wieder auszulesen. Es geht nicht. Warum? Verwenden Sie zur Abhilfe eine binäre Ausgabefunktion und eine adäquate Lernrate γ_k.

2.2 Adaptive lineare Klassifikation

Bei der Einführung der formalen Neuronen in Kapitel 1 sahen wir, dass ein formales Neuron mit binärer Ausgabefunktion prinzipiell eine lineare Separierung der Eingabemuster in zwei Klassen vornimmt. In diesem Abschnitt betrachten wir als Beispiele solcher Klassifikatoren zwei historische Modelle, die man als durchaus noch aktuelle Vorläufer heutiger Modellvarianten ansehen kann.

2.2.1 Das Perzeptron

Das erste, genauer beschriebene, algorithmisch orientierte Modell von kognitiven Fähigkeiten wurde von dem Psychologen F. Rosenblatt 1958 [ROS58] vorgestellt und löste unter den Gehirnforschern großen Enthusiasmus aus. Endlich schien ein Modell vorhanden zu sein, das nicht nur bestimmte Reize lernen und wieder erkennen kann, sondern auch von den ursprünglichen Eingabedaten leicht abweichende Reize richtig zuordnen kann. Diese Fähigkeit zur Klassifizierung schien eine echte Intelligenz-Leistung zu sein.

Das Grundschema der verschiedenen, existierenden Perzeptronvarianten besteht aus einer Eingabe (*künstliche Retina*) S, einer festen Kodierung (*Assoziationsschicht*) A in der jede Einheit (*Neuron*) dieser Schicht mehrere Bildpunkte der Retina beobachten kann, und einer Verarbeitungsschicht (*Responseschicht*) R, in der jede Einheit lernen soll, nur genau auf eine Klasse von Eingabemustern der Retina anzusprechen (s. Abb. 2.12). Die Verbindungen zwischen den Schichten S und A wurden als speziell ausgebildete, mehr oder weniger zufällig vorhandene, feste Zuordnungen verstanden; die Verbindungen zwischen A und R dagegen als modifizierbar angenommen.

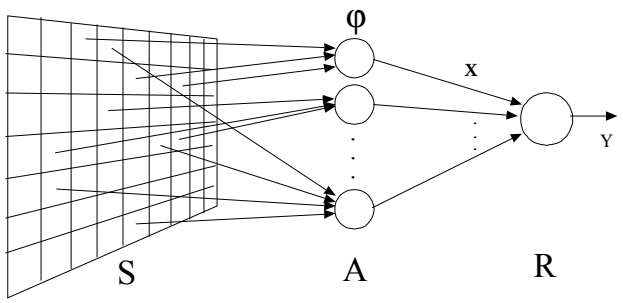


Abb. 2.12 *Grundarchitektur des Perzeptrons*

Bezeichnen wir mit $\mathbf{x} = (x_1,...,x_n)^T = (\varphi_1(S),...,\varphi_n(S))^T$ die Ausgabe der Assoziationseinheiten als Funktion φ_i der Eingabe in S, und mit $\mathbf{w} = (w_1,...,w_n)^T$ die Stärke der Verbindungen (Gewichte) zwischen den Schichten A und R, so wird die binäre, logische Ausgabe y_i (*Prädikat*) in der Responseschicht genau dann WAHR

$$y = \begin{cases} \text{TRUE} & \text{wenn } \mathbf{w}^T\mathbf{x} > s \\ \text{FALSE} & \text{wenn } \mathbf{w}^T\mathbf{x} \le s \end{cases} \tag{2.31}$$

wenn die Aktivität einen Schwellwert *s* überschreitet. Eine Einheit der Response-Schicht kann somit als ein binäres, formales Neuron angesehen werden.

Die faszinierende, neue Idee dieses Ansatzes bestand nun darin, durch Eingabe verschiedener Trainingsmuster und ihrer Bewertung durch einen externen Lehrer die Maschine dazu zu bringen, selbst die Gewichte für eine korrekte Klassentrennung richtig einzustellen.
Einer der bekanntesten Lernalgorithmen dafür lautet folgendermaßen:

> Bezeichnen wir aus der Menge $\Omega := \{\mathbf{x}\}$ aller möglichen Muster diejenige Teilmenge mit Ω_1, bei deren Mustern immer das Prädikat y = TRUE lauten soll, und die andere Menge von y = FALSE mit $\Omega_2 := \Omega - \Omega_1$. Alle Muster von Klasse 1 sind also in Ω_1, und alle Muster aus Klasse 2 in Ω_2. Unser Lernziel besteht darin, die Trennung zwischen den Mustern beider Klassen mit **w** so zu lernen, dass für jedes **x** aus Klasse Ω_1 die Bedingung $\mathbf{w}^T\mathbf{x} > s$ gilt.

Dies lässt sich auch allgemeiner formulieren: Gruppieren wir die Schwelle *s* als zusätzliches Gewicht in **w** und erweitern die Muster **x** zu

$$\mathbf{w} = (w_1,...,w_n,s)^T \quad , \quad \mathbf{x} = (x_1,...,x_n,1)^T \tag{2.32}$$

so gilt für jedes **x** aus Klasse Ω_1 stattdessen die Bedingung $\mathbf{w}^T\mathbf{x} > 0$.

Wie erreichen wir das Lernziel? Ist das Gewichtsprodukt (die Korrelation zwischen **x** und **w**) zu klein, so können wir durch einen kleinen Zuwachs $\gamma\mathbf{x}$ (mit $0 < \gamma < 1$)

$$\mathbf{w}(t) = \mathbf{w}(t-1) + \gamma\mathbf{x} \qquad \textit{Perzeptron-Lernregel} \tag{2.33}$$

für den Gewichtsvektor erreichen, dass mit $\mathbf{w}(t)^T\mathbf{x} = (\mathbf{w}(t\text{-}1) + \gamma\mathbf{x})^T\mathbf{x} = \mathbf{w}(t\text{-}1)^T\mathbf{x} + |\mathbf{x}|^2 > \mathbf{w}(t\text{-}1)^T\mathbf{x}$ die gewünschte Relation mindestens verbessert oder vielleicht sogar erreicht wird. Dies gibt uns eine Lernregel, aber noch keinen Beweis, dass sich das Lernziel erreichen lässt.

Die Lernregel können wir auch in einem Algorithmus in einer Pseudo-Programmiersprache formulieren:

```
PERCEPT1:
    Wähle zufällige Gewichte w  zum Zeitpunkt t:=0.
    REPEAT
        Wähle zufällig ein Muster x aus Ω1∪ Ω2;     t:= t+1;
        IF (x  aus Klasse  Ω1)
            THEN  IF wTx < 0
                    THEN  w = w + γx
                    ELSE  w = w
                  END
            ELSE  IF wTx > 0
                     THEN  w = w - γx
                     ELSE  w = w
                  END
         END
    UNTIL (alle x richtig klassifiziert)
```

Codebeispiel 2.2 *Der Perzeptron-Algorithmus*

Dieser Algorithmus lässt sich auch einfacher formulieren. Dazu bilden wir die neue Menge $\Omega^- := \{\mathbf{x} \mid -\mathbf{x}$ aus Klasse $\Omega_2\}$ der negierten Trainingsmuster. Für sie gilt statt der Eigenschaft $\mathbf{w}^T\mathbf{x} \le 0$ der Muster aus Klasse Ω_2 die Bedingung $\mathbf{w}^T\mathbf{x} > 0$. Somit gilt auch für die (negierten) Muster aus Klasse 2 die Lernregel $\mathbf{w}(t) = \mathbf{w}(t-1) + \gamma\mathbf{x}$. Mit der speziellen Trainingsmenge Ω^- lässt sich so der Algorithmus PERCEPT1 formulieren:

```
PERCEPT2:
        Wähle zufällige Gewichte w  zum Zeitpunkt t:=0.
        REPEAT
         Wähle zufällig ein Muster x aus Ω1U Ω⁻;      t:= t+1;
         IF wTx  ≤ 0
            THEN  w(t) = w(t–1) + γx
            ELSE  w(t) = w(t–1)
        END
        UNTIL (alle x richtig klassifiziert)
```

Codebeispiel 2.3 *Der vereinfachte Perzeptron-Algorithmus*

Der Algorithmus verwendet die Lehrerentscheidung ("Soll-Wert") nur indirekt durch die Konstruktion von Ω^-. Dies können wir auch explizit tun. Dazu definieren wir das Prädikat *y* mit den numerischen Werten 0,1 anstatt mit den logischen Attributen TRUE, FALSE. Die Ausgabe (2.31) wird so zu

$$y = \begin{cases} 1 & \text{wenn } \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \text{wenn } \mathbf{w}^T\mathbf{x} \le 0 \end{cases} \tag{2.34}$$

Weiterhin führen eine Lehrerentscheidung ein

$$L(\mathbf{x}) = \begin{cases} 1 & \text{wenn } \mathbf{x} \in \Omega_1 \\ 0 & \text{sonst} \end{cases} \tag{2.35}$$

Jetzt können wir das "`IF...THEN`" Konstrukt in PERCEPT2 weggelassen und stattdessen einheitlich immer die Lernregel

$$\mathbf{w}(t) = \mathbf{w}(t-1) + \gamma\,(L(\mathbf{x}) - y)\mathbf{x} \qquad \textit{Fehler-Lernregel} \tag{2.36}$$

benutzen.
Die Lernregel verwendet für die Korrektur der Gewichte die Differenz aus Lehrervorgabe und tatsächlicher Entscheidung, also den Fehler des Systems. Rosenblatt nannte diese "rückgekoppelte" Lernregel *back-coupled error correction* [ROS62].

Schön wäre es, wenn wir diese spezielle Lernregel verallgemeinern könnten zu einem allgemeinen Lernprinzip. Betrachten wir dazu die Menge aller Klassenentscheidungen. Die obige Lernregel (2.36) bei falsch klassifizierten Mustern lässt sich als Spezialfall eines Algorithmus ansehen, der versucht, den Fehler R(**w**) des Systems bei den Gewichten **w** zu minimieren

$$R(\mathbf{w}) = \sum_{\mathbf{x}\in\Omega_F} -\mathbf{w}^T\mathbf{x} \qquad \textit{Perzeptron-Zielfunktion} \tag{2.37}$$

über alle mit dem Gewichtsvektor (Klassengrenze) **w** fehlklassifizierten Muster $\{\mathbf{x}\}_{\text{falsch}} =: \Omega_F$, die erst bei richtiger Klassifikation aller Muster null wird [DUD73]. Die Funktion R wird als *Zielfunktion* bezeichnet. Beim Lernen wird diese Funktion minimiert. Klassifizieren wir zuerst alle Muster und verbessern dann die Gewichte, so erfolgt das Lernen nach Gl. (2.33) durch alle falsch eingeordneten Muster

$$\mathbf{w}(t) = \mathbf{w}(0) + \gamma \sum_{\mathbf{x}\in\Omega_F} \mathbf{x} \tag{2.38}$$

Dies lässt sich auch als negative Ableitung (Gradient) der Perzeptron-Zielfunktion Gl. (2.37) schreiben

$$\mathbf{w}(t) = \mathbf{w}(0) - \gamma\,\nabla_{\mathbf{w}} R(\mathbf{w}) \tag{2.39}$$

Dies können wir auch für einen einzelnen Zeitschritt anwenden:

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma\,\nabla_{\mathbf{w}} R(\mathbf{w}(t-1)) \tag{2.40}$$

Damit kennen wir das Lernziel des Perzeptron-Algorithmus: Der Algorithmus versucht, die Strafe R bei falscher Einordnung der Muster zu minimieren. Dabei wird bei der Iteration ein festes Inkrement (Lernrate) γ = const verwendet, was allerdings zu einer immerwährenden Verschiebung führen kann. Ist die Konvergenz dennoch gewährleistet ?

Das berühmte *Perzeptron-Konvergenztheorem* ([MIN88], Theorem 11.1) besagt nun, dass der obiger Algorithmus tatsächlich nach einer endlichen Anzahl von Iterationen mit konstantem Inkrement (z. B. γ = 1) erfolgreich terminiert, falls es ein **w*** gibt, das beide Klassen mit der Entscheidung (2.34) von einander trennt. Im Sinne unserer früheren No-

tation müssen die beiden Klassen somit *linear separierbar* sein. Andernfalls, so zeigen Simulationen, wird die Klassengrenze bei der Iteration nur periodisch hin und her geschoben.

Obwohl man wusste, dass ein formales Neuron nur linear separierbare Klassen voneinander trennen kann, erhoffte man sich trotzdem "intelligentere" Leistungen von dem Perzeptron. Der Schlüssel dafür sollte die Funktion $\varphi(S)$ sein, die die Eingangsdaten "geeignet" auf linear separierbare Mustermengen abbildet. Natürlich stellt sich sofort die Frage: Lässt sich für alle Probleme eine solche Funktion $\varphi(S)$ finden? Welche Art von Objekten (geometrischen Figuren auf der Retina), welche Art von Musterklassen kann das Perzeptron prinzipiell erkennen und welche nicht ? Diese Frage untersuchten Minsky und Papert in ihrem berühmten Buch [MIN88] genauer. Die Möglichkeit der Klassentrennung ist beim Perzeptron entscheidend von der Abbildung φ_i der Retina-Schicht auf die Merkmale geprägt. Für Perzeptronarten mit Merkmalsfunktionen, die

- nur Bildpunkte aus einem begrenzten Radius enthalten *(diameter-limited perceptrons)*
- von maximal *n* (beliebigen) Bildpunkten abhängig sind *(order-restricted perceptrons)*
- eine zufällige Auswahl aller Bildpunkte erfassen *(random perceptrons)*

konnten sie zeigen, dass sie prinzipiell *keine* korrekte Klassifizierung von Punktmengen $X := \{\mathbf{x}\}$ (Bildpixeln) für topologische Prädikate durchführen können, wie beispielsweise "X ist ein Kreis", "X ist eine konvexe Figur" oder "X ist eine zusammenhängende Figur". Stattdessen kann ein solches Perzeptron nur ein einziges Prädikat "X hat die Eulerzahl E" erkennen. Die Eulerzahl macht eine globale Aussage über die Anzahl zusammenhängender Punktmengen (schwarze Bildpixel, die benachbart sind) und ist definiert als

$$E(X) := \quad K(X) - \text{Anzahl der Löcher} \tag{2.41}$$

wobei K(X) die Anzahl der *Komponenten* des Bildes ist, also der Punktmengen eines Bildes, die dadurch charakterisiert sind, dass alle Punkte in einer Menge miteinander zusammenhängen, der Mengenschnitt untereinander aber leer ist. Die "Löcher" lassen sich dann einfach als Komponenten der komplementären Punktmenge X definieren.

Beispiel

In der folgenden Abb. 2.13 beispielsweise sind links zwei zusammenhängende, schwarz markierte Punktmengen in die Bildpunkte X eingezeichnet, so dass $K(X) = 2$ ist. In der rechten Abbildung ist das Komplement davon zu sehen, bei dem nur ein zusammenhängender Bereich (also Zahl der Löcher im Urbild = 1) erkennbar ist, so dass für Abb. 2.13 die Eulerzahl $E(X) = 2–1= 1$ ist.

Ein Perzeptron vom ersten Typ kann beispielsweise erkennen, ob ein Bild vollkommen schwarz oder weiß ist, ob die bedeckte Fläche mehr als *s* Bildpunkte ausmacht oder eine ganz bestimmte Figur an der selben Retinastelle präsentiert wird, nicht aber, ob eine Fi-

gur verbunden ist. Ursache dieser Unfähigkeit ist die Tatsache, dass mit einer Schicht nur ein sehr begrenztes Maß der Fähigkeit zu verwirklichen ist, eine Figur trotz starker Veränderungen (*Deformationsinvarianz*) wiederzuerkennen. Beispielsweise lässt sich zeigen, dass mit einer Schicht von Neuronen mit Synapsen erster Ordnung keine Unempfindlichkeit gegenüber Verschiebungen (*Translationsinvarianz*) oder Drehungen (*Rotationsinvarianz*) möglich ist

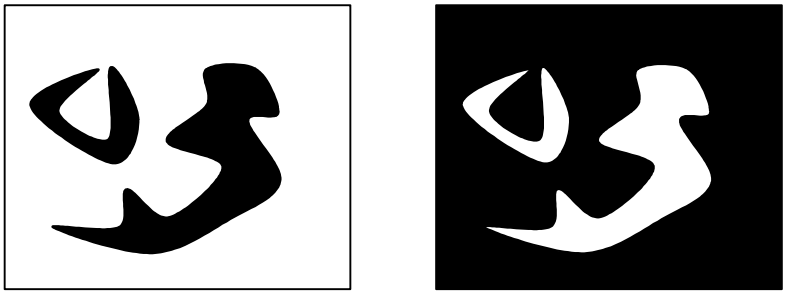


Abb. 2.13 *Die Eulerzahl einer Bildpixelmenge*

Die Untersuchungen von Minsky und Papert zeigten, dass einfache Perzeptrons prinzipiell nicht die "höheren" Leistungen ermöglichen, die man von ihnen erwartet hatte, und ernüchterten so viele Forscher. Abgesehen von den Neurologen und Gehirnforschern wandten sich das Interesse vieler Forscher der "Künstlichen Intelligenz" in den Jahren 1961-63 den formalen, logischen Methoden (Symbolverarbeitung) zu, die mit Hilfe der neu entwickelten Computer versprachen, die höheren, "intelligenten" Funktionen direkt als Programme umzusetzen. Erst die Schwierigkeiten in den 80-ger Jahren, die widersprüchlichen Ereignisse der realen Welt in die logischen symbol-verarbeitende Systeme zu integrieren, zeigten die Vorteile der alten Ansätze.

Multilayer-Perzeptrons

Zu den Perzeptron-Architekturen, die Minsky und Papert untersuchten, gehörten auch die *Gamba-Perzeptrons*, bei denen die Eingabefunktionen φ_i selbst wieder eine vollständige Perzeptron-Funktion (formales Neuron) darstellen. Fassen wir alle Funktionen φ_i zu einem Block (Schicht) zusammen, so lässt sich ein Gamba-Perzeptron als ein zweischichtiges Netzwerk ansehen, das in der ersten Schicht *n* nicht-lineare formale

Neuronen enthält und in der zweiten Schicht ein Neuron (Perzeptron) mit *n* Eingangsvariablen.

Da beide Schichten nicht-lineare Ausgabefunktionen enthalten, ließ sich über die Grenzen und Möglichkeiten der Gesamtfunktion damals relativ wenig aussagen. Obwohl wir heute wesentlich mehr über mehrschichtige Netzwerke wissen, ist dies auch heute noch ein sehr unerforschtes Gebiet, besonders was die Gesamtfunktion vieler unterschiedlicher Schichten (*Multilayer-Perzeptrons*) angeht.

Ein größeres Problem bei der Anwendung des Perzeptrons ist die Tatsache, dass die Eingabe und Ausgabe nur binär kodiert ist. Möchte man statt dessen beispielsweise bei Echtzeit-Controllern metrische, analoge Eingaben wie [Volt] oder [cm] verwenden, so reicht es nicht aus, die Analogwerte vorher zu digitalisieren und die einzelnen Binärstellen (Bits) der Zahlendarstellung als binäre Einzeleingaben zu behandeln. Vielmehr muss man eine Kodierungsstufe vorsehen, bei der die Abbildung $\varphi(S)$ die Ähnlichkeit von Analogwerten (z.B. geringe Differenz) auch in eine Ähnlichkeit der digitalen Zahl (z.B. geringer Hammingabstand) umsetzt. Ein Beispiel für ein solches Vorgehen ist das CMAC-System von J. Albus [ALB75], bei dem eine Version des Perzeptron-Lernalgorithmus zum Erlernen der Positionskontrolle eines Roboter-Manipulatorarms mit 7 Freiheitsgraden eingesetzt wird.

2.2.2 Adaline

Eine der ersten Maschinen zur stochastischen Mustererkennung war das *ADALINE* (*ADA*ptive *LIN*ear *E*lement) von Widrow und Hoff (1960). Ziel des Projekts war es, einen adaptiven, linearen Filter zu entwickeln, der mit Hilfe von dargebotenen Mustern eine Klassifizierung der Eingabedaten (binäre Ausgabe) erlaubt. In der folgenden Abb. 2.14 ist ein Funktionsschema zu sehen.

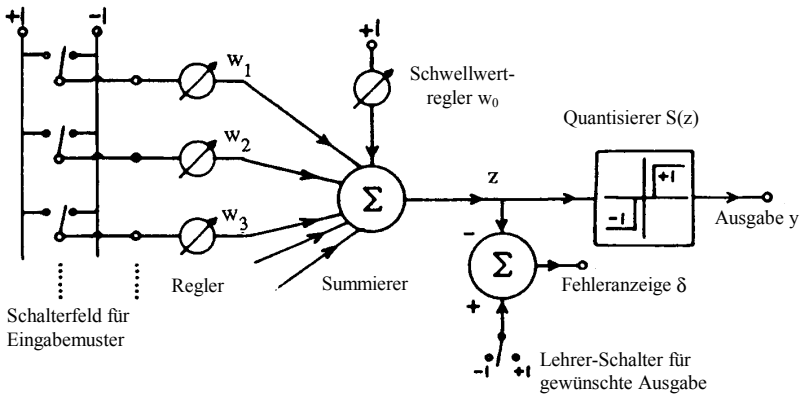


Abb. 2.14 Funktionsschema von Adaline (nach [WID60])

Der Apparat selbst wurde damals aus diskreten, elektro-mechanischen Bauteilen aufgebaut: die 16 Schalter der Eingabe sind in einer 4x4 Matrix an der Frontplatte angeordnet, die Gewichte sind als Drehwiderstände (Potentiometer) ausgeführt und bestimmen die Größe der elektrischen Ströme, die im Summierer zusammengeführt und auf einen Verstärker gegeben werden. Die Fehleranzeige ist ein Zeiger-Meßinstrument. Nach jeder Eingabe eines Musters (Umlegen der Schalter) mussten mit der Hand die entsprechenden Potentiometer soweit gedreht werden, bis die Fehlermessung ein Minimum anzeigte.

Wie man leicht sieht, implementiert das Funktionsschema des Adaline ebenso wie das Perzeptron formale Neuronen: Durch Schalter erzeugte, binäre Eingabesignale x_i werden mit Koeffizienten w_i gewichtet und summiert. Ist die Gesamtsumme > 0, so wird +1 ausgegeben, andernfalls -1. Damit ist die Ausgabe

$$y = \begin{cases} +1 \text{ wenn } \mathbf{w}^T\mathbf{x} > 0 \\ -1 \text{ wenn } \mathbf{w}^T\mathbf{x} \le 0 \end{cases} \qquad \begin{matrix} w_i \in \Re^+ \\ y, x_i \in \{-1,+1\} \end{matrix} \tag{2.42}$$

Betrachtet man die konstante Kodierung in der Assoziationsschicht des Perzeptrons als konstante Vorverarbeitungsstufe, so ist die gelernte Verarbeitung des Adaline in der Form eines binären, formalen Neurons der des Perzeptron äquivalent.

Beim Lernen gibt es allerdings einen Unterschied: Bei der Adaline-Maschine zum Lernen einer Musterklasse werden die Gewichte so verändert, dass gegenüber der Lehrervorgabe $L(\mathbf{x})$ der Klassifizierungsfehler möglichst klein wird. Die Lerngleichung für die Veränderung der Gewichte lässt sich dabei aus diesem Gütekriterium herleiten. Sei der erwartete, quadratische Fehler R als Abweichung der tatsächlichen Aktivität $z(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ von der gewünschten Ausgabe $L(\mathbf{x})$ des Lehrers definiert

$$R(\mathbf{w},L) := \left\langle (z(\mathbf{x}) - L(\mathbf{x}))^2 \right\rangle_{\mathbf{x}} = \left\langle (\mathbf{w}^T\mathbf{x} - L(\mathbf{x}))^2 \right\rangle_{\mathbf{x}} \tag{2.43}$$

Angenommen, wir verringern die Gewichte, wenn der Fehler bei Vergrößern der Gewichte zunimmt; die Funktion $R(\mathbf{w})$ also eine positive Ableitung in allen Richtungen hat. Dies erreichen wir beispielsweise durch Addition der negativen Ableitung

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma(t)\ \frac{\partial}{\partial \mathbf{w}} R(\mathbf{w}(t-1)) \tag{2.44}$$

Die äußere Ableitung lässt sich direkt komponentenweise errechnen

$$\begin{aligned}
\frac{\partial}{\partial w_i} R(\mathbf{w}) &= \frac{\partial}{\partial w_i} \left\langle (\mathbf{w}^T\mathbf{x} - L(\mathbf{x}))^2 \right\rangle_{\mathbf{x}} \\
&= \left\langle 2(\mathbf{x}^T\mathbf{w} - L(\mathbf{x}))\ \frac{\partial}{\partial w_i} (\mathbf{x}^T\mathbf{w} - L(\mathbf{x})) \right\rangle \\
&= \left\langle 2(\mathbf{x}^T\mathbf{w} - L(\mathbf{x}))\ \frac{\partial}{\partial w_i} \Big(\sum_j x_j w_j - L(\mathbf{x}) \Big) \right\rangle \\
&= \left\langle 2(\mathbf{x}^T\mathbf{w} - L(\mathbf{x}))\ x_i \right\rangle
\end{aligned} \tag{2.45}$$

Damit ist die Ableitung für alle Komponenten des Vektors in symbolischer Schreibweise

$$\frac{\partial}{\partial \mathbf{w}} R(\mathbf{w}) = \langle 2(\mathbf{x}^T\mathbf{w} - L(\mathbf{x}))\mathbf{x} \rangle \tag{2.46}$$

und Gl.(2.44) lautet für ein $\mathbf{x}$ ohne Erwartungsklammern

$$\mathbf{w}(t) = \mathbf{w}(t\text{-}1) - \gamma(t)(\mathbf{w}^T\mathbf{x} - L(\mathbf{x}))\mathbf{x} \qquad \textit{stochastische Approximation} \tag{2.47}$$

Wollen wir mit dem neuen Gewicht $\mathbf{w}(t)$ den Fehler $\delta := \mathbf{w}^T(t\text{-}1)\mathbf{x} - L(\mathbf{x})$ möglichst vollständig kompensieren, so sollte die neue Aktivität $\mathbf{w}^T(t)\mathbf{x}$

$$\mathbf{w}^T(t)\mathbf{x} = \mathbf{w}(t-1)^T\mathbf{x} - [\gamma(t)(\mathbf{w}^T\mathbf{x} - L(\mathbf{x}))\mathbf{x}]^T\mathbf{x} \overset{!}{\sim} L(\mathbf{x}) \tag{2.48}$$

gleich oder mindestens direkt proportional zum geforderten Verhalten $L(\mathbf{x})$ sein. Dies können wir erreichen, indem wir den Faktor γ als Produkt $\gamma =: \gamma' \cdot \alpha$ auffassen mit positivem, endlichem $\alpha := (\mathbf{x}^T\mathbf{x})^{-1} = |\mathbf{x}|^{-2}$, $0 < \alpha < G$. Setzen wir dies in Gl. (2.48) ein, so ergibt sich wie gefordert $\mathbf{w}^T(t)\mathbf{x} = \gamma' L(\mathbf{x})$. Bei $\gamma' = 1$ würde zwar der Fehler vollständig kompensiert, aber nur für das zufällige $\mathbf{x}$ am Zeitpunkt t–1, nicht für alle anderen Zeitpunkte. Deshalb ist es sinnvoll, mit $\gamma' < 1$ nur einen kleinen Schritt als Verbesserung vorzusehen.

Mit diesem Gedanken und der Umbenennung $\gamma' \rightarrow \gamma$ lässt sich (2.47) verändern zu

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma(t)(\mathbf{w}^T\mathbf{x} - L(\mathbf{x}))\frac{\mathbf{x}}{|\mathbf{x}|^2} \qquad \textit{Widrow-Hoff Lernregel} \qquad (2.49)$$

Da der Lernschritt proportional Fehler bzw. zur Differenz δ ist, wird die Widrow-Hoff Lernregel auch als *Delta-Lernregel* bezeichnet. Das für das Lernen erforderliche Fehlersignal wird dabei direkt aus der reellen Aktivität $z(\mathbf{x})$ und nicht aus dem binären Ausgangssignal $y(\mathbf{x})$ gewonnen. Die binäre Ausgabe anstelle einer allgemeinen reellen Ausgabe ist deshalb eher historisch bedingt und wird je nach Anwendungsfall modifiziert. Im folgenden Codebeispiel ist das allgemeine Vorgehen verdeutlicht. Als Eingabe dienen wieder die *n*-dimensionalen Mustertupel **x** und die *m*-dimensionale gewünschte Ausgabe **L** dazu.

```
ADALINE:        (* Implementierung des Widrow-Hoff Algorithmus *)
VAR  (* Datenstrukturen *)
     x:          ARRAY[1..n] OF  REAL;          (* Eingabe *)
     z,y,L:      ARRAY[1..m] OF  REAL;          (* IST und SOLL-Ausgaben *)
     w:          ARRAY[1..m,1..n] OF  REAL;     (* Gewichte *)
     γ:          REAL                           (* Lernrate *);   x2: REAL;
BEGIN
  γ:= 0.1;                              (* Lernrate festlegen *)
  initWeights(w,0.0);                   (* Gewichte initialisieren *)
  REPEAT
     Read(PatternFile,x,L)              (* Eingabe der Tupel *)
     x2 := Z(x,x)                       (* |x|^2 *)
(* Aktivität bilden im Netz *)
     FOR i:=1 TO m DO                   (* Ausgabe für alle Neuronen *)
        z[i] :=  Z(w[i],x)              (*  Aktivität *)
        y[i] :=  S(z[i])                (*  Nicht-lin. Ausgabe *)
     END;
(* Lernen der Gewichte *)
     FOR i:=1 TO m DO                   (* Für alle Neuronen *)
     FOR j:=1 TO n DO                   (* und alle Dimensionen *)
        w[i,j] := w[i,j]-γ*(z[i]-L[i])*x[j]/x2   (* Gewichte verändern *)
     END;
  END;
UNTIL EndOf(PatternFile)
END ADALINE.
```

Codebeispiel 2.4 *Aktivierung und Lernen in Adaline*

Bei beiden Lernregeln ist der Erwartungswert der Lehrerentscheidung bzw. des Fehlers für das Konvergenzziel entscheidend, nicht die Zahlenrepräsentation des Ausgangssignals. Der Widrow-Hoff Lernalgorithmus (2.49) konvergiert immer, egal, ob die Mustermengen Ω_1 und Ω_2 linear separierbar sind oder nicht. Die sinnvolle Interpretation des Konvergenzziels ist allerdings ein anderes Problem.

Das Schema des adaptiven, linearen Systems (*adaptiven Filters*), das von Widrow und Hoff damals präsentiert und weiterentwickelt (s. z.B. [WID85],[TSYP73]) wurde, fand in den darauffolgenden Jahren verschiedene Nutzanwendungen. Eine der wichtigsten ist zweifelsohne die Anwendung in interkontinentalen Telefonanlagen, wo digitale, adaptive Filter die Wirkungen parasitärer Kapazitäten zwischen den Kupferleitungen von sehr langen Kabeln, etwa der Transatlantikkabel, kompensieren. In der folgenden Abb. 2.15 ist dies an einem digitalen Signal verdeutlicht.

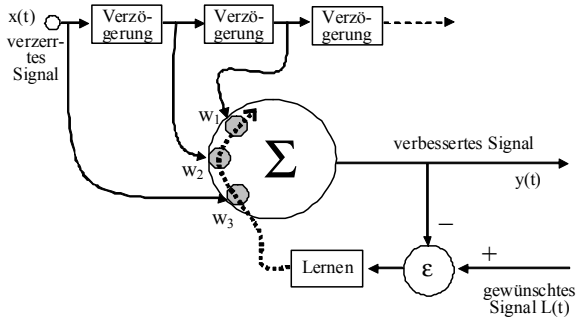


Abb. 2.15 *Adaptive Signalverbesserung (nach [WID88])*

Dabei erscheint das digitale Signal mit dem ursprünglich binären, durch verzögerte Überlagerungen des Ursprungssignals "verschmierten" Pegel am Eingang einer Verzögerungskette, beispielsweise einer elektronischen Schaltung aus Widerständen und Kondensatoren („Eimerkettenschaltung“). Nach jedem Verzögerungsglied wird das Signal abgetastet. Die verschiedenen Signalwerte zu verschiedenen Zeitpunkten werden dem Neuron präsentiert, und es entscheidet dann aus dem jetzigen Signalwert und den verzögerten, früheren Signalwerten, ob das Signal nur gestört ist oder der Pegel tatsächlich umgeschaltet wird. Durch Training mit einem Normsignal („gewünschtes Signal“) im Vergleich mit dem durch das Kabel geleitete und verbesserte Signal wird der Fehler bestimmt und die Gewichte nach der Widrow-Hoff-Regel verändert.
Durch den Einsatz von solchen adaptiven Filtern zur phasenrichtigen Regeneration des Pegels ist es möglich, die Fehlerrate von 10% auf 10^{-6} zu drücken oder aber bei gleicher Fehlerrate hierdurch eine Steigerung der Übertragungsrate um den Faktor 4 zu ermöglichen.

Perzeptron und Adaline als Assoziativspeicher

Angenommen, wir geben ein binär kodiertes $\mathbf{x}^k$ an die Eingänge des i-ten Neurons einer Perzeptron-Schicht und wünschen dazu eine orthogonal kodierte Assoziation $y_i^k := L_i^k$. Mit der Lernregel (2.36) des Perzeptrons

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma(L_i(\mathbf{x})–y_i)\mathbf{x} \qquad \text{Fehler-Lernregel}$$

ist

$$\mathbf{w}_i(1) = (L_i(\mathbf{x}^k)–y_i)\mathbf{x}^k = L_i^k\,\mathbf{x}^k \qquad \text{bei } \mathbf{w}_i(0) = 0 \text{ und folglich } y_i^k(0) = 0. \quad (2.50)$$

Da in allen folgenden Zeitschritten $L_i^k = 0$ und mit passender Schwelle $\mathbf{w}_i^T\mathbf{x} < 0$ sein wird (Übungsaufgabe: warum?), speichert der Gewichtsvektor $\mathbf{w}_i$ gerade ein Muster $\mathbf{L}^k$ mit L_i^k=1 ab, das bei der Eingabe von $\mathbf{x}^k$ an y_i wieder ausgegeben wird. Damit implementiert das Perzeptron bei geeigneter Inhaltskodierung und Anfangswerten einen Assoziativspeicher.

Auch in Adaline implementiert mit der Lernregel

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma(t)(L(\mathbf{x})–z_i)\mathbf{x} \quad (2.51)$$

bei t=1

$$\mathbf{w}_i(1) = (L(\mathbf{x})–z_i)\mathbf{x} = L_i^k\,\mathbf{x}^k \qquad \text{bei } \mathbf{w}_i(0) = 0 \text{ und folglich } z_i^k(0) = 0. \quad (2.52)$$

mit dem oben gesagten einen Assoziativspeicher.

Damit sehen wir, dass viele Modelle neuronaler Netze, die eine korrelative Aktivitärts- und Lernregel (Hebb-Regel) verwenden, bei geeigneter Musterkodierung einen korrelativen Assoziativspeicher darstellen. Der Assoziativspeicher kann also als eine Art "Grundfunktion" oder "erste Näherung" vieler neuronaler Modelle angesehen werden. Eine Übersicht und Vergleich mit anderen Modellen ist auch in [CAT92] zu finden.

2.2.3 Symbolik und Subsymbolik

Nachdem *Minski* und *Papert* zeigten, dass das Perzeptron selbst so einfache Probleme wie die XOR Funktion prinzipiell nicht lernen kann, wandten sich in den 60-ger Jahren des letzten Jahrhunderts viele Forscher enttäuscht von den neuronalen Netzen ab und versuchten, "intelligente" Systeme nicht mehr indirekt durch Imitation der Natur zu verwirklichen, sondern sahen in den neu aufkommenden Computern die Maschinen, mit denen sie direkt ihre Ideen umsetzten konnten. Die Grundidee, die sich noch heute in vielen Systemen der "künstlichen Intelligenz" (KI) widerspiegelt, besteht darin, neue Erkenntnisse aus einer Reihe von eingegebenen Daten (*Fakten*) und (logischen) Verarbeitungs*regeln* zu erschließen. Hat man nur genügend viele Fakten und Regeln eingegeben, so der Anspruch, wird sich die Maschine zu einem "intelligenten" Lebewesen (*General Problem Solver, GPS*) entwickeln. Manche Autoren sahen schon dabei die Menschen als "Geburtshelfer", die einer mächtigen Intelligenz auf den Weg hilft und nach einigen Computergenerationen, in denen jede Generation eine neue, noch Intelligentere gebaut habe, als geduldete Idioten am Rande einer maschinellen Zivilisation dahinvegetieren.

Wie wir heute wissen, ist dies alles nicht eingetreten. Stattdessen hat sich die Hoffnung, aus Fakten und Regeln Intelligenz zu erschließen, als fataler Irrtum erwiesen. Die so konzipierten Systeme leiden alle unter demselben Problem: sie können selbst keine

neuen Daten aufnehmen (neue Fakten definieren) und sind unfähig, mit unvorhergesehenen Situationen fertig zu werden. Obwohl verschiedene Anstrengungen unternommen wurden, um dem abzuhelfen, ist doch ein großes Problem ungelöst: die Unfähigkeit dieser Systeme, zu abstrahieren, zu generalisieren und Analogien zu bereits bekannten Daten oder Lösungen zu finden. Es ist deshalb nicht selten zu beobachten, dass regelbasierte Systeme um Echtzeitkomponenten (Schnittstellen zur "Realen Welt") erweitert werden, indem man ein neuronales Netz als Vorverarbeitungsstufe davor schaltet.

Seitdem sind viele Anstrengungen gemacht worden, durch die Integration statistischer Methoden und die Einführung sog. „Zielfunktionen“ mit Nebenbedingungen (*constraints*) die obige Problematik zu entschärfen.

Die Fähigkeiten solcher KI-Systeme sind dabei durchaus mit den Leistungen neuronaler Netze vergleichbar: beide versuchen, mit ihren Mitteln (ihren Algorithmen), dasselbe Ziel, die Optimierung einer Funktion, zu erreichen. Welche Methoden dabei schneller zum Erfolg führen, muss man am konkreten Optimierungsproblem überprüfen. Erfahrungsgemäß enthalten aber die neuronalen Netze den (traditionellerweise) weiter entwickelten statistischen Ansatz, so dass sie dann auch bessere Ergebnisse vorweisen können.

Die Ursache für dieses Problem liegt prinzipiell in dem Ansatz selbst, willkürlich die "Intelligenz" in einem Rechner zu programmieren. Alle Personen, Zustände oder Situationen sind hierbei als *Symbole* im Rechner repräsentiert. Sucht der Rechner zu einer Person oder einer Situation ein Analogon, so ist dies ohne Aussicht auf Erfolg, wenn keinerlei weitere Daten über die Beziehungen und Ähnlichkeiten zwischen den Symbolen eingegeben wurden. Alles, was an Analogschlüssen in einem solchen System möglich ist, muss also mühsam per Hand vom Programmierer explizit eingegeben werden. Dies ist aber, ebenso wie andere Fakten, eine stete Quelle von Ungenauigkeiten und Fehlschlüssen in einer sich stetig verändernden Welt. Aus diesem Grund sind symbolische Expertensysteme meist nur für abgeschlossene, genau definierbare "Welten" geeignet, von denen es nicht sehr viele gibt.

Im Gegensatz dazu lassen sich manche Funktionen neuronaler Netze als "Abstraktion" oder "Generalisierung" deuten; ein wesentliches Element in neuronalen Netzen sind gerade die Lernregeln zur Verarbeitung von neuen Dingen. Der wesentliche Unterschied zu den symbolverarbeitenden Systemen liegt dabei in der Möglichkeit der neuronalen Netze, Dinge und Situationen adäquat zu kodieren. Die interne *Kodierung*, die bei dem symbolischen Ansatz keine Rolle spielt, ist hier entscheidend. Das folgende, einfache Beispiel von Hinton [HIN89b] soll dies illustrieren. Angenommen, wir haben einen autoassoziativen Speicher. Wir speichern das Wissen, dass der Elefant Erni grau ist, allgemein Elefanten grau und Personen rosa sind durch das Abspeichern der aneinander gereihten Tupel

```
(Erni, Farbe, Grau)
(Elefant, Farbe, Grau)
(Person, Farbe, Rosa)
```

Nun bewirken wir eine assoziative Ergänzung durch Eingabe eines unvollständigen Tupels (Person, Farbe, ?). In symbolverarbeitenden Systemen würde dies zu einer Rückverfolgung (*backtracking*) führen, bis alle in Frage kommenden Werte abgearbeitet sind. Hier führt dies stattdessen in einem Schritt zum Tupel (Person, Farbe, Rosa). Was passiert nun, wenn man über die Datenbankfunktion hinaus unbekannte Namen eingibt? Ist das Tupel in symbolverarbeitenden Systemen nicht abgespeichert, so muss das System aufgeben. Hier dagegen, wenn Clyde ein Elefant und Bill ein Mensch ist, so ergänzt das System folgendermaßen:

(Clyde, Farbe, ?) → (Clyde, Farbe, Grau)
(Bill, Farbe, ?) → (Bill, Farbe, Rosa)

Wieso? Woher hat das System das zusätzliche Wissen und kann zwischen dem (unbekannten!) Namen für einen Elefanten und einen Menschen unterscheiden? Die Lösung des Rätsels liegt in der Kodierung der Symbole durch ihre Bits, die in der folgenden Tabelle 6.1 aufgelistet ist.

```
 ?       000000   000000
Elefant  111000   000000
Person   000111   000000
Clyde    111000   111000
Erni     111000   000111
Scott    000111   101010
Bill     000111   010101
```

Tabelle 6.1 Kodierung des 1.Objekts

Wie wir sehen, impliziert die *subsymbolische* Kodierung eine Ähnlichkeit zwischen bekannten Daten und entscheidet durch ein inhärentes Ähnlichkeitskriterium (Hammingabstand), welche Farbe die unbekannte Sache "Clyde" hat.

Aufgaben

1) Was ist der Hauptunterschied zwischen der Perzeptron-Lernregel und der Adaline-Lernregel ? Wann stimmen sie überein und bewirken das Gleiche?

Stimmen in diesem Fall auch die Zielfunktionen überein?

2) Man implementiere ein Netz, das die Klassentrennungen aus Aufgabe 1.4 lernt. Dazu erzeuge man zufällige Muster $\mathbf{x} = (x_1,x_2)$, wobei x_i eine uniform verteilte Zufallszahl aus [0,4] sein soll, und benutze das in Aufgabe 1.4 geschriebene Programm als Lehrer, der für jedes $\mathbf{x}$ das $L(\mathbf{x})$ erzeugt und an den Lernalgorithmus weitergibt. Als Algorithmen verwende man die Perzeptron-Lernregel und die Widrow-Hoff Lernregel. Man teste das System mit mehreren Lernraten γ. Welche Lernregel bewirkt eine schnellere Konvergenz und warum ?

2.3 Lernen und Zielfunktionen

Die Leistung vieler Systeme hängt von Parametern ab, deren Auffinden Schwierigkeiten bereitet. Diese Problematik beschränkt sich nicht nur auf die unbekannten Gewichte in neuronalen Netzen, sondern ist ein allgemeines Problem. Eine wichtige Lösungsidee besteht darin, die Parameter nicht direkt auszurechnen (was oft sehr zeitaufwendig oder unmöglich ist), sondern sie sukzessive solange zu verbessern, bis die gewünschte Systemleistung erreicht ist. Ein Beispiel dafür ist das im vorigen Abschnitt eingeführte Verfahren der Gewichtsverbesserung durch den Widrow-Hoff Lernalgorithmus. Dieses Verfahren wollen wir im folgenden Abschnitt verallgemeinern.

Den Vorgang der schrittweisen Verbesserung der Gewichte bezeichnen wir mit *Adaption* und ist eine Implementation von *Lernen*. Der Begriff des Lernens beschränkt sich hier auf die Verbesserung der Systemleistung gemäß einem Lernziel *(Zielfunktion)*. Im psychologischen und soziologischen Kontext ist der Begriff „Lernen“ allerdings weitaus vielschichtiger und komplexer. Unabhängig von der in der Psychologie und Pädagogik bekannten Terminologie wollen wir hier besonders folgende Lernformen betrachten:

- **Überwachtes Lernen** (interaktives Lernen, beratendes Lernen)

  Diese Form des Lernens geht davon aus, das eine Art Lehrer (menschlich oder nur in Form einer spezifisch reagierenden Umgebung) existiert, der das Lernen steuert. Unterformen dieses Lernens sind

  - Beispiel-basiertes Lernen (*example based learning, feedback learning*)

    Es wird zu jeder Eingabe auch die gewünschte Ausgabe bereitgestellt. Der Lernende kann dann versuchen, die Differenz zwischen beiden im Laufe des Lernens besonders klein zu machen.

  - Erklärungs-basiertes Lernen (*explanation based learning EBL*)

    Der Lehrer stellt zur Eingabe noch das Ziel sowie Regeln, das Ziel zu erreichen, und Beispielpaare bereit. Beim Lernen werden die Beispiele generalisiert. Dieses Verfahren [MK86],[DM86] ist eher bei regelbasierten Systemen üblich, nicht bei neuronalen Netzen.

  - Score-basiertes Lernen (*reinforcement learning*)

    Die Umgebung (der Lehrer) stellt nur ein skalares Gütemaß ("gut", "schlecht", mit Abstufungen dazwischen) über die Leistung des Lernenden zur Verfügung. Der Lernende muss daraus selbst sehen, was an der Ausgabe zu ändern ist.

- **Unüberwachtes Lernen** (*observation based learning, emotion based learning, similarity learning*)

  Wenn der Lernende keine explizite Rückmeldung über die Güte seines Lernens erhält, so muss er sie sich selbst erschließen. Dies geschieht durch die Beobachtung der Umwelt und dem Vergleich der gewünschten Auswirkungen mit den tatsächlich beobachteten Auswirkungen.

  Die Mechanismen dafür sind vielfältig. So kann der Lernende den (intern) erwarteten Zustand, die Vorhersage (*prediction*), mit der Realität vergleichen und daraus die Parameter der Ausgabe geeignet verändern. Es lässt aus mehreren Einzelbeobachtungen ein interner Score aufstellen und dieser wie beim Score-basierten Lernen verwenden. Man kann aber auch die Ähnlichkeit (*similarity*) des Ausgabemusters mit einem beobachteten Muster vergleichen und damit in Beobachtungswelt in Cluster (Klassen) einteilen.

Ein Beispiel für die obige Systematik bietet die Beurteilung von Aufsätzen in der Schule. Bei der Korrektur eines Aufsatzes kann man entweder genau die Formulierungen vorschreiben, was man hätte schreiben sollen (Beispiel-basiertes Lernen). Oder man gibt Beispiele an und sagt damit, wie ein Aufsatz allgemein zu konstruieren sei (Erklärungsbasiertes Lernen). Man kann aber auch nur eine Note darunter schreiben ("gut", "mittelmäßig") und hofft auf das Score-basierte Lernen.

Im schlechtesten Fall aber wird nichts korrigiert (unüberwachtes Lernen), sondern der Lernende muss durch vieles Lesen von Aufsätzen selbst erkennen, was einen "guten" Aufsatz ausmacht (c*lustering*). Dies erfordert aber zum einen internen Sinn (internen Kritiker) für "gut" zur Auswahl der Vorbilder und zum anderen die Fähigkeit, die Abweichung des eigenen Werks (*similarity*) von der Abstraktion der Vorbilder zu erkennen.

Nach diesen allgemeinen Bemerkungen stellt sich nun die Frage: Wie lässt sich konkret für ein Problem ein Lernalgorithmus konstruieren?

2.3.1 Zielfunktionen und Gradientenabstieg

Am Anfang eines Lernalgorithmus steht eine geeignete Definition des Ziels, das erreicht werden soll. Dies kann beispielsweise wie beim feedback-Lernen die Minimierung des Fehlers sein, den das System macht: dabei nimmt man gern den absoluten Fehler oder das Fehlerquadrat. Beispielsweise konzipierten Widrow und Hoff die iterative Gewichtsveränderung als Methode, das Minimum des quadratischen Fehlers zu finden. Dieser stochastisch-analytische Ansatz basiert auf Arbeiten, die damals der bekannte Kybernetiker Norbert Wiener über Filter durchgeführt hatte.

Die Frage des Lernens reduziert sich mit dieser Annahme auf die Frage, wie wir iterativ die optimalen Parameter $(w_1^*,...,w_n^*) = \mathbf{w}^*$ für das Minimum einer solchen Zielfunktion finden. Angenommen, die zu minimierende Zielfunktion $R(\mathbf{w})$ habe ein lokales, relatives Minimum bei $\mathbf{w}^*$. Ausgehend von einem initialen $\mathbf{w}$ in der Nähe dieses lokalen Minimums versuchen wir, schrittweise iterativ uns dem optimalen $\mathbf{w}^*$ anzunähern (s. Abb. 2.3.1).

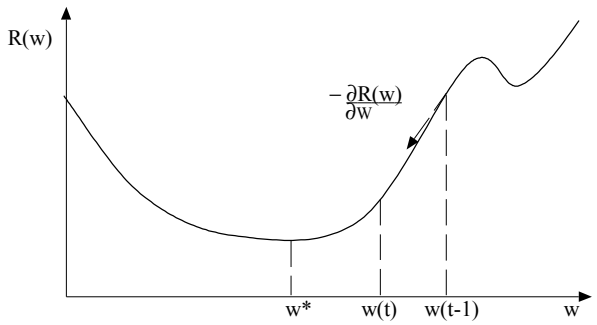


Abb. 2.3.1 Die Gradientensuche

Die Ableitungen nach allen Komponenten können wir in einem Vektor $(\partial R/\partial w_1,...,\partial R/\partial w_n)^T$ zusammenfassen und bezeichnen dies mit dem *Gradienten* grad $R(\mathbf{w})$ der Funktion $R(\mathbf{w})$. Dies lässt sich auch mit dem *Nabla-Operator* $\nabla_w:=(\partial/\partial w_1,...,\partial/\partial w_n)^T$als "$\nabla_w R(\mathbf{w})$" notieren. Der so definierte Vektor der Richtungsableitungen zeigt in der Nähe des Minimums in die Richtung des stärksten Anstiegs der Funktion; der negative Gradient also in Richtung des Minimums. Mit dieser Überlegung soll die Differenz von $\mathbf{w}(t-1)$ beim (t–1)-ten Schritt zu $\mathbf{w}(t)$ vom nächsten Schritt t proportional zum negativen Gradienten sein:

$$\Delta\mathbf{w} := (\mathbf{w}(t) - \mathbf{w}(t\text{-}1)) \sim -\nabla_w R(\mathbf{w}(t-1)) \tag{2.53}$$

oder mit der schrittabhängigen Proportionalitätskonstanten γ

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma(t)\,\nabla_w R(\mathbf{w}(t-1)) \tag{2.54}$$

Eine solche Iterationsgleichung zur Optimierung von Parametern einer Zielfunktion wird auch als *Lernregel*, die Proportionalitätskonstante $\gamma(t)$ dabei als *Lernrate* bezeichnet. Im allgemeinen Fall ist γ eine Matrix, die eine Amplituden- und Richtungskorrektur vornimmt; im Normalfall ist eine skalare Funktion ausreichend.

Man beachte, dass diese Art der Optimierung nicht unbedingt zwingend ist. Beispielsweise hat das Gradientenverfahren die unangenehme Eigenschaft, dass die Ableitung in der Nähe des Extremums sehr klein ist und deshalb mit kleiner Schrittweite die Iteration nur sehr langsam konvergiert. Aus diesem Grunde werden auch manchmal die Ableitungen höherer Ordnungen, falls bekannt, in der Iterationsgleichung (2.54) zusätz-

lich verwendet. Dies bedeutet eine bessere, nicht-lineare Annäherung an den gewünschten Extremwert R(**w***), beispielsweise in Form einer abgebrochenen Taylor-Entwicklung einer Funktion f(x+Δx)

$$f(x+\Delta x) = f(x) + \frac{\partial}{\partial x} f(x)\,\Delta x + \frac{1}{2!}\frac{\partial^2}{\partial x^2} f(x)\,(\Delta x)^2 + \ldots \tag{2.55}$$

zu

$$R(\mathbf{w}+\Delta\mathbf{w}) - R(\mathbf{w}) = \nabla_w R(\mathbf{w})^T\,\Delta\mathbf{w} + \tfrac{1}{2}\Delta\mathbf{w}^T\boldsymbol{R}\,\Delta\mathbf{w} + \ldots \tag{2.56}$$

mit der Hesse-Matrix $\boldsymbol{R} = (\partial^2 R/\partial w_i \partial w_j)$. Sie bedeuten allerdings einen zusätzlichen Rechenaufwand. Ein Überblick darüber ist z.B. in [LEE88] zu finden.

In der numerischen Optimierungstechnik ist auch eine Methode bekannt, die notwendigen Koeffizienten γ_1, γ_2 für die Überlagerung selbst wieder iterativ zu bestimmen. Bei der Methode des *conjugate gradient* [HES90] geht man davon aus, dass man genau dann den besten Schritt Δ**w*** gefunden hat, wenn

$$R(\mathbf{w}+\Delta\mathbf{w}^*) - R(\mathbf{w}) = (\nabla_w R(\mathbf{w})^T + \tfrac{1}{2}\Delta\mathbf{w}^{*T}\boldsymbol{R})\,\Delta\mathbf{w} = 0 \tag{2.57}$$

was bei

$$\nabla_w R(\mathbf{w})^T + \Delta\mathbf{w}^{*T}\boldsymbol{R} = 0 \tag{2.58}$$

eintritt. Man muss nun an dieser Stelle bei dem resultierenden *n*-dim Vektor das homogene, *n*-dim. Gleichungssystem in *n* Schritten lösen, um eine effiziente Schrittweite zu errechnen. Da dies zusätzliche Rechnungen bedeutet und die Rechen-Komplexität der Algorithmen vergrößert, ohne unser Verständnis der grundsätzlichen Zusammenhänge zu fördern, soll hier nicht weiter auf diese Technik eingegangen werden. Ein Vergleich verschiedener Varianten ist beispielsweise in [MOL93] zu finden. Für die folgenden Betrachtungen reicht das einfache Verfahren des Gradientenabstiegs vollkommen aus.

Eine andere Methode, die Konvergenz zu beschleunigen, besteht in einer Reduzierung von γ(t), je näher wir dem Ziel $\mathbf{w}^*$ kommen. Als Anzeichen dafür lässt sich beispielsweise eine Änderung im Vorzeichen des Gradienten deuten, wie sie zustande kommt, wenn die Schätzung „über das Ziel hinausschießt" und wir jenseits des angestrebten w* auf dem anderen Teil der Zielfunktion gelandet sind. Die Regel für γ ist etwa

$$\gamma(t) = \begin{cases} \gamma(t-1) * 2 & \text{wenn } \operatorname{sign}(\nabla_w R(t)) = \operatorname{sign}(\nabla_w R(t\text{-}1)) \text{ oder } R(t) \le R(t\text{-}1) \\ \gamma(t-1)/2 & \text{sonst} \end{cases} \tag{2.59}$$

Bei diesem Verfahren bleibt die Lernrate somit immer nur für eine endliche Zahl von Iterationen konstant, bis der Gradient sich im Vorzeichen ändert oder die Zielfunktion nicht mehr abnimmt. Eine erfolgreiche Strategie besteht auch darin, zusätzlich für jede Gewichtskomponente eine eigene Lernrate vorzusehen [SIL90]. In diesem Fall ist γ(t) eine Matrix, die nur in der Hauptdiagonalen von Null verschieden ist.

Interessanterweise lässt sich für das Gradientenverfahren direkt eine wichtige Eigenschaft zeigen: Da das Verfahren ein Absinken der Zielfunktion bewirkt, ist bei nach un-

ten beschränkten Zielfunktionen auch eine Konvergenz der Iteration garantiert. Die Stabilitätsbedingung an den Systemzustand für diskrete Zeitschritte,

$$R(t+1) \leq R(t) \tag{2.60}$$

oder für kontinuierliche Zeit

$$\frac{\partial R}{\partial t} \leq 0 \quad \text{wobei } \exists\, R_{min} \text{ mit } R_{min} \leq R(t) \tag{2.61}$$

wird bei Zielfunktionen, die nach unten beschränkt sind, als *Ljapunov-Bedingung* und die monoton fallende Zielfunktion als *Ljapunov-Funktion* bezeichnet.
Ist die Zielfunktion von einem Parameter **w** abhängig, so folgt dabei

$$\frac{\partial R}{\partial t}(\mathbf{w}(t)) = \nabla_w R(\mathbf{w}) \frac{\partial \mathbf{w}}{\partial t} \leq 0 \tag{2.62}$$

Für diese Bedingung ist beispielsweise hinreichend, wenn

$$\frac{\partial \mathbf{w}}{\partial t} = -\gamma \nabla_w R(\mathbf{w}) \qquad \text{mit } \gamma > 0 \tag{2.63}$$

ist, da $(\partial \mathbf{w}/\partial t)^2 \geq 0$ gilt. Die Stabilitätsbedingung transformiert sich interessanterweise in die Aussage, dass eine Konvergenz gewährleistet ist, wenn sich die Gewichte mit dem negativen Gradienten der Energie ändern.

Allerdings trifft dies jeweils nur für ein **w** und damit nur für eine sequentielle Parameterverbesserung zu; die Konvergenz für parallel oder gleichzeitig verbesserte Parametervektoren $\mathbf{w}_i$ (parallel lernende Neuronen) muss extra gezeigt werden.

Das Gradientenverfahren zur Bestimmung der Parameter für ein (lokales) Minimum oder Maximum einer Zielfunktion lässt sich immer dann gut einsetzen, wenn die Zielfunktion bzw. deren Ableitung $\nabla_w R(\mathbf{w})$ bekannt ist. Dies ist dann gegeben, wenn alle Muster $\{\mathbf{x}\}$ vorliegen und damit $p(\mathbf{x})$ numerisch errechnet werden kann. Was kann man aber tun, wenn eine Entscheidung bereits für jedes neu beobachtete Muster getroffen werden muss, obwohl wir die Wahrscheinlichkeitsdichte noch gar nicht richtig kennen? Diese Fragestellung wurde durch das Verfahren der *stochastischen Approximation* beantwortet.

2.3.2 Stochastische Approximation

Meist ist uns der Erwartungswert $R(\mathbf{w}) = \langle r(\mathbf{w},\mathbf{x})\rangle_{\mathbf{x}}$ der verrauschten Zielfunktion $r(\mathbf{w},\mathbf{x})$ und damit der Gradient $\nabla_w R(\mathbf{w})$ der Zielfunktion nicht bekannt, sondern nur ihr zufallsbedingter Gradient $\nabla_w r(\mathbf{w},\mathbf{x})$. Die Aufgabe besteht nun darin, mit jedem der sequentiell beobachteten Muster **x** eine neue Schätzung **w** zu erreichen, die einen geringeren Abstand zu dem optimalen $\mathbf{w}^*$ mit $\nabla_w R(\mathbf{w}^*) = 0$ hat.

Dies ist gleichbedeutend mit der Aufgabe, die Nullstelle der Funktion $F(\mathbf{w}) := \nabla_w R(\mathbf{w})$ zu finden, wobei allerdings nur deren "verrauschte" Funktionswerte $f(\mathbf{x},\mathbf{w})$ mit $F(\mathbf{w}) =$

$\langle f(\mathbf{x},\mathbf{w})\rangle_{\mathbf{x}}$ bekannt sind, siehe Abb. 2.16 für den eindimensionalen Fall. Wie lässt sich trotzdem, ohne alle Muster **x** abwarten zu müssen um p(**x**) und damit F(**w**) zu bilden, ein verbesserter Wert für $\mathbf{w}^*$ erreichen?
Diese Frage wurde zuerst von *Robbins* und *Monro* (1951) beantwortet. Sie erkannten, dass auch die einfache, ungemittelte Iteration

$$\mathbf{w}(t) = \mathbf{w}(t\text{-}1) - \gamma(t)\, f(\mathbf{w}(t\text{-}1),\mathbf{x}(t)) \tag{2.64}$$

die Nullstelle der Funktion F(**w**), und in unserem Fall damit das Minimum der Funktion R(**w**), "findet".

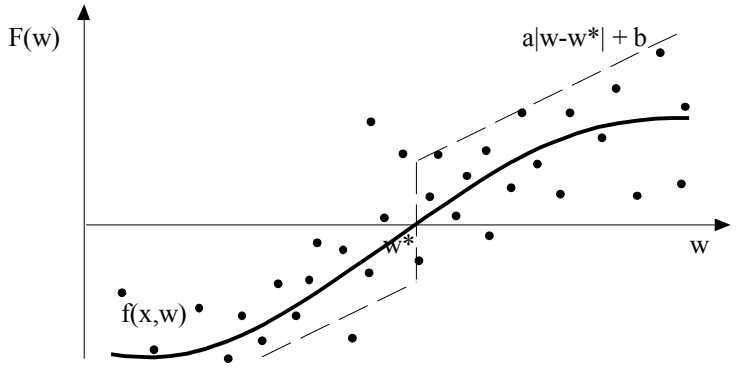


Abb. 2.16 Die Nullstelle einer verrauschten Funktion

Allerdings machten sie dies von einigen Bedingungen über die Koeffizienten und die „Gutartigkeit“ der verrauschten Funktion abhängig. Betrachten wir das Intervall $[\mathbf{w}_0..\mathbf{w}^*..\mathbf{w}_1]$. Dann wird
vorausgesetzt, dass für alle Komponenten (2.65)

- die Funktion $F(w) := \langle f(x,w)\rangle_x$ ist *zentriert*, d.h. insbesondere $F(w^*) = 0$
- F(w) ist *ansteigend*, d.h. $F(w<w^*) < 0$, $F(w>w^*) > 0$.
- F(w) ist *beschränkt* mit $|F(w)| \leq a|w-w^*| + b < \infty$ $a,b > 0$
- f(x,w) hat *endliche Varianz*, d.h. $\sigma^2(w) = \langle (F(w) - f(x,w))^2\rangle_x < \infty$
- $\gamma(t)$ *verschwindet*, $\gamma(t) \to 0$
- $\gamma(t)$ wird *nicht zu schnell klein* $\sum_{t=1}^{\infty} \gamma(t) = \infty$
- $\gamma(t)$ wird *nicht zu groß* $\sum_{t=1}^{\infty} \gamma(t)^2 < \infty$

wird garantiert, dass

- die mittlere quadratische Konvergenz ex.: $\lim_{t\to\infty} \langle (w(t) - w^*)^2 \rangle = 0$ *(Robbins-Monro)*
- die Konvergenz mit Wahrscheinlichkeit eins erfolgt: $P(\lim_{t\to\infty} w(t)=w^*) = 1$ *(Blum)*

s.[ROB51], [BLUM54], [KIEF52] und die Übersicht in [FU68].

Ein Beispiel für $\gamma(t)$ ist die Lernrate $\gamma(t) = 1/t$. Sie ist nicht die einzig mögliche Wahl; man kann beispielsweise auch diese Lernrate für eine endliche Zahl von Iterationen konstant lassen (z.B. bis zum Vorzeichenwechsel des Gradienten) und dann erst herabsetzen. Nehmen wir an, dass $\hat{\gamma}$ für maximal *N* Iterationen jeweils konstant ist, so gilt mit $\hat{\gamma} \geq \gamma(t)$

$$\infty = \sum_{t=1}^{\infty} \gamma(t) \leq \sum_{t=1}^{\infty} \hat{\gamma} \qquad \text{sowie} \quad \sum_{t=1}^{\infty} \hat{\gamma}^2 \leq \sum_{t=1}^{\infty} N\gamma(t)^2 < \infty \tag{2.66}$$

und $\hat{\gamma}(t) \to 0$ die Konvergenzbedingungen ebenso erfüllt.

Auch ist Lernrate von $\gamma(t) = 1/t$ nicht immer die beste Wahl. Wie Pfaffelhuber [PFAF73] zeigte, ist diese Wahl ein Kompromiss, um frühere und spätere Ereignisse gleich stark den jetzigen Zustand von **w** beeinflussen zu lassen, d.h. ein unendliches Gedächtnis zu erreichen. Ist die Lernrate größer, beispielsweise konstant, so werden die früheren Ereignisse schwächer gewichtet und der Ereignisspeicher wird endlich. Wichtig ist dies besonders bei zeitlich schwankenden, nicht stationären Zufallsprozessen (Ereignisverteilungen): hier muss man bewusst durch die Benutzung einer anderen Lernrate als 1/t altes, nicht mehr aktuelles Wissen "überschreiben", beispielsweise durch eine konstante Lernrate oder einen gleitenden Mittelwert anstelle eines "echten". Kann man anstelle des stochastischen Gradienten $\nabla_w r(\mathbf{w},\mathbf{x})$ nur die stochastische Zielfunktion $r(\mathbf{w},\mathbf{x})$ messen (etwa beim Score-basierten Lernen), so lässt sich trotzdem das Extremum iterativ finden. In diesem Fall kann man den Gradienten durch den Differenzenquotienten $(r(\mathbf{w}+\mathbf{d},\mathbf{x})-r(\mathbf{w}-\mathbf{d},\mathbf{x}))/2\mathbf{d}$ ersetzen [KIEF52]. Eine allgemeine Untersuchung über dieses Gebiet ist z.B. in [LJUNG77] zu finden.

Stochastisches Lernen

Diese Aussagen lassen sich nun leicht auf die Vektorvariable **w** anwenden und für unser Problem nutzen, das Minimum einer Zielfunktion zu finden. Sind die Voraussetzungen für die Ableitung der Zielfunktion (begrenzte Varianz und Erwartungswerte) und für die Lernrate $\gamma(t)$ im betrachteten Gebiet erfüllt, so wissen wir nun, dass anstelle der Suche durch den Erwartungswert des Gradienten (2.54) auch die stochastische Approximation (2.64) durch den Gradienten zum Ziele führt:

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma(t)\, \nabla_{\mathbf{w}}\, r(\mathbf{w}(t-1),\mathbf{x}(t)) \tag{2.67}$$

Dies lässt sich an einem Beispiel verdeutlichen. Betrachten wir dazu als Kostenfunktion den quadratischen Fehler $r(\mathbf{w},\mathbf{x}) := \frac{1}{2}(\mathbf{w}-\mathbf{x})^2$ bei der Einordnung eines Musters **x** in eine von zwei Klassen.

$$\begin{cases} r(w_1,x) < r(w_2,x) & \text{x ist aus Klasse 1} \\ r(w_1,x) > r(w_2,x) & \text{x ist aus Klasse 2} \end{cases} \tag{2.68}$$

Die Trennlinie $\{\mathbf{x}^*\}$ zwischen beiden Klassen ist mit der Bedingung $r(\mathbf{w}_1,\mathbf{x}^*) = r(\mathbf{w}_2, \mathbf{x}^*)$ gegeben, so dass gilt

$$\tfrac{1}{2}(\mathbf{w}_1-\mathbf{x}^*)^2 = \tfrac{1}{2}(\mathbf{w}_2-\mathbf{x}^*)^2 \quad \Rightarrow \quad d_1 = |\mathbf{w}_1-\mathbf{x}^*| = |\mathbf{w}_2-\mathbf{x}^*| = d_2$$

Die Klassengrenze ist eine Gerade (Hyperebene), die zwischen beiden Klassenprototypen senkrecht auf der Verbindungsgeraden $\mathbf{w}_1 - \mathbf{w}_2$ steht und sie in zwei gleich große Strecken unterteilt, siehe Abb. 2.17 (Übungsaufgabe: Warum? Beweisen Sie dies.).

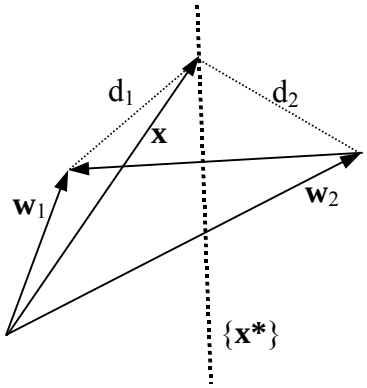


Abb. 2.17 *Klassentrennung durch eine Gerade {x*}*

Die Lernrate $\gamma(t) := 1/t$. erfüllt die Bedingungen von (2.65) so dass die Lernregel oder stochastische Iterationsgleichung für jeden Klassenprototyp lauten kann

$$\mathbf{w}_i(t) = \mathbf{w}_i(t-1) - \gamma(t)\,\nabla_w r(\mathbf{w}_i(t-1),\mathbf{x}(t)) = \mathbf{w}_i(t-1) - 1/t\,(\mathbf{w}_i(t-1)-\mathbf{x}(t)) \tag{2.69}$$

Der Programmcode für das stochastische Lernen einer solchen Klassentrennung ist auszugsweise

```
VAR  w: ARRAY[1..2, 1..2] OF REAL;
     x: ARRAY[1..2] OF REAL;   γ: REAL; t,k,i:INTEGER;
BEGIN
     t:=1;                          (* erster Zeitschritt *)
REPEAT
     (* Eingabe oder Erzeugung des Trainingsmusters x *)
     Read(PatternFile, x);
```

```
    γ := 1/t;                    (* zeitabh. Lernrate  *)
    (*suche Klasse mit minimalem Abstand *)
    IF Abstand(x,w[1]) > Abstand(x,w[2])
       THEN k:=2       ELSE k:= 1
    END;
    (* verändere den Gewichtsvektor *)
    FOR i:=1 TO 2 DO             (* Approximation des Mittelwerts *)
       w[k,i]:= w[k,i] - γ*(w[k,i]-x[i]);
    END;
    t:= t+1;                     (* nächster Zeitschritt und Muster *)
UNTIL EndOf (PatternFile);
```

Codebeispiel 2.5 *Stochastische Approximation einer Klassentrennung*

Nach dem ersten Schritt (t=1) ist mit $\mathbf{w}(1) = \mathbf{x}(1)$ die Iteration unabhängig von dem Startwert $\mathbf{w}(0)$. Wohin konvergiert $\mathbf{w}(t)$?
Es ist $\mathbf{w}(1) = 1/1\ \mathbf{x}(1) = \overline{\mathbf{x}}(1)$, der Mittelwert von $\mathbf{x}$. Ist dies zufällig so oder gilt es auch allgemein ?
Für einen Induktionsbeweis müssen wir nun für beliebiges *t* beweisen, dass Mittelwert und Gewicht übereinstimmen, nachdem klar ist, dass es für den ersten Schritt mit t = 1 gilt.
Angenommen, es gilt bereits für t-1 Schritte

$$\mathbf{w}(t\text{-}1) = \overline{\mathbf{x}}(t\text{-}1) = \frac{1}{t-1}\sum_{i=1}^{t-1}\mathbf{x}(i) \qquad \textit{Induktionsvoraussetzung}$$

Dann ist für den *Induktionsschritt*

$$\overline{\mathbf{x}}(t) = \frac{1}{t}\sum_{i=1}^{t}\mathbf{x}(i) = \frac{\mathbf{x}(t)}{t} + \frac{t-1}{(t-1)t}\sum_{i=1}^{t-1}\mathbf{x}(i)$$

$$= \frac{\mathbf{x}(t)}{t} + \frac{t}{(t-1)t}\sum_{i=1}^{t-1}\mathbf{x}(i) - \frac{1}{(t-1)t}\sum_{i=1}^{t-1}\mathbf{x}(i)$$

$$= \frac{\mathbf{x}(t)}{t} + \overline{\mathbf{x}}(t\text{-}1) - 1/t\ \overline{\mathbf{x}}(t\text{-}1) = \mathbf{w}(t\text{-}1) - 1/t\,(\mathbf{w}(t\text{-}1) - \mathbf{x}(t)) = \mathbf{w}(t)$$

die Behauptung erfüllt, so dass die Behauptung für alle *t* gilt, was zu beweisen war.

Für die quadratische Kostenfunktion ist also bei der stochstischen Iteration nach jedem Iterationsschritt der geschätzte Parameter $\mathbf{w}$ der Mittelwert aller bisherigen Beobachtungen $\{\mathbf{x}\}$; der Parameter konvergiert zum Mittelwert aller $\mathbf{x}$, die in diese Klasse eingeordnet werden. Damit haben wir nicht nur das Konvergenzziel der quadratischen Zielfunktion kennen gelernt, sondern auch eine Möglichkeit, einen Mittelwert iterativ zu berechnen.

Bei der Klassentrennung im Codebeispiel 2.5, kann allerdings eine ungünstige, initiale Lage der Klassenprototypen $\mathbf{w}_i$ eine andere Klassifikation der Eingabe bewirken als dies bei dem perfekt konvergierten Endzustand geschieht. $\mathbf{w}_i$ ist also nicht der Erwartungswert aller Muster $\mathbf{x}$ von Ω_i, sondern der Erwartungswert einer (etwas) anderen Teilmenge von Ω. Da die anfangs fehlerhaft eingeordneten Muster das Ergebnis "verfälschen", ist es sinnvoll, bei derartigen Problemen, bei denen die beobachtete Verteilung der Muster sich beim Lernen verändert, frühere Eingaben geringer zu bewerten.

Das Fachgebiet der Mustererkennung ist, unabhängig von der Entwicklung der neuronalen Netze, in den 60-ger und 70-ger Jahren entwickelt worden, s. z.B. die Lehrbücher [FU68], [FUK72], [DUD73], [TSYP73], [TOU74].

Aufgaben

2.1) Im Abschnitt 2.3.1 haben wir das Gradientenverfahren zur Verbesserung eines Parameters w für die Nullstelle einer Funktion kennen gelernt. Betrachten wir nun ein anderes Verfahren. Dazu gehen wir von dem Mittelwertsatz aus. Für den Differenzenquotienten der Funktion f existiert ein Punkt ξ, an dem der Wert der Ableitung gleich dem Quotienten ist

$$f'(\xi) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Kennen wir also die analytische Form der Ableitung, so können wir den Wert x_2 ausdrücken als

$x_2 = x_1 + (f(x_2)-f(x_1))/f'(\xi)$

Haben wir nur kleine Intervalle mit $x_1 \approx x_2$, so gilt

$x_2 \approx x_1 + (f(x_2)-f(x_1))/f'(x_1)$

so dass wir als neue Iterationsvorschrift für den Parameter w_{k+1} und die gesuchte Nullstelle $f(w_{k+1})=0$ formulieren können

$w_{k+1} = w_k - f(w_k)/f'(w_\kappa)$ $\qquad f'(w) \neq 0$

Aufgabe a)

Benutzen Sie die oben entwickelte Formel, um das Minimum der Funktion $R(w) = 3w^2 - 12w + 12$ zu erhalten. Welchen Wert erhalten Sie für das Minimum?

Aufgabe b)

Ersetzen Sie die Ableitung, die nicht in allen Fällen bekannt ist, statt dessen durch den beobachteten Differenzenquotienten der letzten beobachteten Werte $f(w_k)$ und $f(w_{k-1})$. Wie lautet die Iterationsgleichung dann ?

Führen Sie mit den Startwerten $w_0=0$, $w_1=1$ Iterationsschritte durch bis $|w_k-w_{k-1}| < 0{,}01$ erreicht ist.

2.2) Zeigen Sie, dass die Iteration von Gl. (2.69) bei konstanter Lernrate $\gamma < 1$ alle $\mathbf{x}(i)$ nicht gleich gewichtet, sondern ihr Einfluß auf den Gewichtsvektor nimmt exponentiell ab. Dazu zeigen Sie durch vollständige Induktion, dass im k-ten Schritt

$$w(k) = (1-\gamma)w(k-1) + \gamma\, x(k) = (1-\gamma)^k w(0) + \gamma \sum_{i=1}^{k} (1-\gamma)^{k-i}\, x(i)$$

gilt. Was folgt beim Grenzübergang $\lim_{k\to\infty}\langle \mathbf{w}(k)\rangle = ?$

2.3) Angenommen, Sie haben die Punkte A_1=(0,3 0,7), B_1=(-0,6 0,3). Aus den beiden Klassen A und B. Ihr Gewichtsvektor Ihres ADALINE ist initial $\mathbf{w}_0$ = (-0,6 0,8). und γ=0,5, wobei die Ausgabe +1 bei Klasse A und –1 bei Klasse B sein soll. Zeichnen Sie sich die Situation (Eingaben und Gewichtsvektor) auf und führen Sie eine jeweils Iteration mit dem Widrow-Hoff-Algorithmus durch.

a) Geben Sie als Eingabe $\mathbf{x}(1) = A_1$ vor. Was passiert? Ist die Klassifizierung korrekt?

b) Geben Sie nun als Eingabe $\mathbf{x}(2) = B_1$ vor. Was passiert?

c) Zeichnen Sie nun die resultierende Lage von $\mathbf{w}(1)$ und $\mathbf{w}(2)$ ins Diagramm ein. Klassifiziert ADALINE mit diesem Gewichtsvektor die Muster A_1 und B_1 korrekt?

2.4 Klassifizierung stochastischer Muster

Bei der Grundmodellierung im ersten Kapitel abstrahierten wir von konkreten Ereignissen und faßten die für ein Ereignis typischen Werte in einem *Mustervektor* **x** zusammen. Dabei stellte sich die Frage, wie man die Ereignisart oder *Klasse* ermittelt, zu der **x** gehört. Wie wir sahen, kann ein formales Neuron eine Trennung zwischen zwei Mustermengen (Klassen) Ω_1 und Ω_2 vornehmen und implementiert damit eine *Mustererkennung*. Die dazu notwendigen Gewichte oder Koeffizienten w_i der Klassengrenze müssen allerdings bekannt sein. Aufgabe der Lernalgorithmen des vorigen Abschnitts war es, die Koeffizienten und damit die Klassengrenze durch die Lage der beobachteten Muster **x** sukzessive zu bestimmen. Da die beobachteten Muster von zufälligen Einflüsse bestimmt werden, wird die korrekte Klassifizierung der Muster als *stochastische Mustererkennung* bezeichnet. Im diesem Abschnitt wollen wir die grundsätzlichen Probleme und ihre Lösung etwas näher kennen lernen. Die Ausführungen gelten dabei nicht nur für neuronale Netze, sondern sind allgemein für alle Arten von Klassifizierungssystemen gültig, etwa für alle medizinischen Diagnosesysteme oder technische Qualitätskontrollen.

2.4.1 Stochastische Mustererkennung

Die Grundproblematik der stochastischen Mustererkennung lässt sich mit Hilfe eines Modells, das die Erzeugung und Erkennung von Mustern als Problem eines Systems aus Nachrichtensender und -empfänger ansieht, folgendermaßen skizzieren:

Seien *M* Quellen von Ereignissen **x** gegeben, die jeweils mit der *a priori*- Wahrscheinlichkeit $P(\omega_i)$ auftreten, wobei ω_i das Ereignis "**x** ist aus Klasse Ω_i" darstellt. Es sei Dann wird ein **x** mit der *likelihood*-Wahrscheinlichkeitsdichte $p(\mathbf{x}|\omega_i)$ produziert und einem Empfänger übermittelt, s. Abb. 2.18.

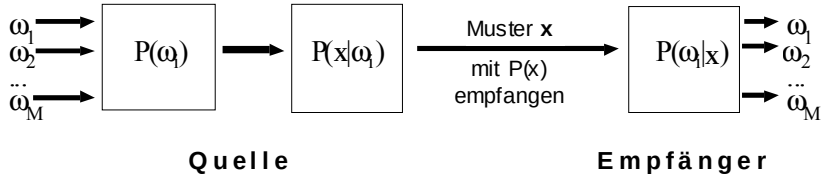


Abb. 2.18 *Mustergenerierung und -erkennung*

Seien für ein vorliegendes Muster **x** die Wahrscheinlichkeiten $P(\omega_i|\mathbf{x})$ bekannt. Es ist klar, dass der geringste Fehler dann gemacht wird, wenn als Klassifizierungsstrategie für jedes **x** diejenige Klasse Ω_k zu wählen, für die $P(\omega_k|\mathbf{x})$ maximal wird:

$$\Omega_k: P(\omega_k|\mathbf{x}) = \max_i P(\omega_i|\mathbf{x}) \qquad \textit{Bayes-Klassifikation} \tag{2.70}$$

Der Beobachter kann allerdings nur $p(\mathbf{x})$ messen und muss die *a posteriori*-Wahrscheinlichkeit $P(\omega_i|\mathbf{x})$ irgendwie herausfinden, um auf die gesuchte Klasse Ω_k schließen zu können. Welche Möglichkeiten hat er dafür?

Bekannte Quelle

Ist dem Beobachter die Quelle mit ihren Wahrscheinlichkeiten $P(\omega_i)$ und $P(\mathbf{x}|\omega_i)$ bekannt, so können wir auf das gesuchte $P(\omega_i|\mathbf{x})$ zu schließen

$$P(\omega_i|\mathbf{x}) = P(\mathbf{x}|\omega_i)\, P(\omega_i) / P(\mathbf{x}) \tag{2.71}$$

wobei gilt

$$P(\mathbf{x}) = \sum_i P(\mathbf{x}|\omega_i)\, P(\omega_i) \tag{2.72}$$

Woher haben wir diese Gleichungen? Den Weg dahin können wir mit Basiskenntnissen der Wahrscheinlichkeitstheorie nachvollziehen. Es ist bekannt, dass für zwei Teilmengen A und B der Grundmenge Ω die Beziehung (Definition!) über die bedingten Wahrscheinlichkeiten gilt

$$P(A,B) = P(A \cap B) = P(B|A)P(A) = P(A|B)P(B) \qquad \textit{Bayes-Regel} \tag{2.73}$$

Also ist

$$P(B|A) = P(A|B)P(B) / P(A) \tag{2.74}$$

Setzen wir für $B=\Omega_i$ und für $A=\{\mathbf{x}\}_{beob}$ die beobachteten Muster, so ist erhalten wir direkt die zu zeigende Gl.(2.71).
Weiterhin gilt für einen Musterraum $\Omega = \{\mathbf{x}\}$ aus *M* Teilmengen Ω_i, der in *M* Klassen zerfällt

$$\Omega = \bigcup_{i=1}^{M} \Omega_i \text{ und } \bigcap_{i=1}^{M} \Omega_i = \varnothing \tag{2.75}$$

Mit $P(\omega_i) := P(\mathbf{x} \in \Omega_i) = P(\Omega_i)$ gilt

$$P(A) = P(\Omega \cap A) = P(\bigcup_{i=1}^{M} \Omega_i \cap A) = \sum_{i=1}^{M} P(\Omega_i \cap A) = \sum_{i=1}^{M} P(A|\omega_i)P(\omega_i) \tag{2.76}$$

wobei das zweite Gleichheitszeichen nur mit Gl.(2.73) gilt. Die Beziehung Gl.(2.72) ergibt sich dann mit der obigen Zuordnung der beobachteten Muster für A.

Unbekannte Quelle

In den allermeisten Fällen sind aber die Quellwahrscheinlichkeiten unbekannt und wir müssen sie erst erschließen. Die Musterwahrscheinlichkeit $P(\mathbf{x})$ lässt sich leicht durch Beobachtung errechnen, aber alle anderen Wahrscheinlichkeiten, insbesondere die *a posteriori*-Wahrscheinlichkeit $P(\omega_i|\mathbf{x})$ müssen erst iterativ ermittelt werden. Dabei muss eine Information über den augenblicklichen Lernzustand ermittelt werden, sei es durch einen externen Lehrer mit einer Lehrerbewertung (Bewertung des vorliegenden Musters mit einer Klassenbezeichnung). Dies führt über die implizite oder explizite Bildung von $P(\omega_i)$ zu der gewünschten Bayes-Klassifikation.
Die Verfahren zum Lernen der korrekten Klassifikation unterscheiden sich dabei durch die Information, die sie aus dem Auftreten der Muster $\mathbf{x}$ und der Klassen ω_i ziehen. Ohne ein zusätzliches Wissen über die Klassenzugehörigkeit sei es in Form von Beispielen oder in Form von messbaren Zielfunktionen ist es allerdings nicht möglich, bei unbekannten Quellwahrscheinlichkeiten eine korrekte Zuordnung zu lernen.

Aufgabe 2.4)

Sei eine Datenquelle gegeben, die vier Muster $x_1, .. ,x_4$ aus drei Klassen ω_1, ω_2 ω_3 hervorbringt. Die Wahrscheinlichkeiten sind in der Tabelle aufgeführt.

P(xi|ωk)	ω1	ω2	ω3	P(xi)
x1	0,3	0,7	0	0,41
x2	0,2	0,1	0,3	0,18
x3	0,2	0,2	0,3	0,23
x4	0,3	0	0,4	0,18
P(ωk)	0,2	0,5	0,3	

Wie lautet der günstigste Klassifizierer für $x_1, .. ,x_4$?

2.4.2 Beurteilung der Klassifikationsleistung

Angenommen, wir haben nun ein System zur Klassifizierung von Mustern und konnten durch gute Lernalgorithmen die Wahrscheinlichkeit einer korrekten Diagnose $P(\omega_i|\mathbf{x})$ maximieren – haben wir damit die Aufgabe vollbracht und ein optimales System geschaffen? Leider muss man hier sagen: Nein! Es reicht nicht aus, die bedingte Wahrscheinlichkeit zu maximieren. Tatsächlich gibt es in einer Klassifizierungssituation noch mehr zu beachten. Betrachten wir dazu eine typische Klassifikation, etwa eine Diagnose eines Patienten durch einen Arzt, der wegen der Symptome $\mathbf{x}$ vermutet, dass der Patient an einer Krankheit ω leidet. Für die Fälle, dass die Krankheit vorliegt oder nicht vorliegt (ω oder $\neg\omega$), gibt es jeweils zwei mögliche Diagnosen $D(\mathbf{x})$, also insgesamt vier Fälle für das Auftreten eines Ereignisses ω bzw. $\neg\omega$, für die bei einem bestimmten Arzt (be-

stimmten Diagnosesystem) vier verschiedene Wahrscheinlichkeiten zugeordnet werden können:

Situation (Diagnose, Realität)	**Wahrscheinlichkeit**	**Name**
$(D(x)= \omega \mid \omega)$	$P_L = P(D(x)=\omega \mid \omega)$	Sensitivität
$(D(x)= \neg\omega \mid \omega)$	$P_I = P(D(x)= \neg\omega \mid \omega)$	Ignoranz
$(D(x)= \omega \mid \neg\omega)$	$P_A= P(D(x)=\omega \mid \neg\omega)$	Fehlalarm
$(D(x)= \neg\omega \mid \neg\omega)$	$P_K= P(D(x)= \neg\omega \mid \neg\omega)$	Spezifität

Tabelle 1 *Diagnosewahrscheinlichkeiten*

Dabei wird die Wahrscheinlichkeit für Fehlalarm auch als FRR (*False Rejection Rate*) und die Wahrscheinlichkeit für Ignoranz als FAR (*False Acceptance Rate*) bezeichnet. Es ist klar, dass $P_K+P_A = 1$ und $P_L+P_I = 1$ gelten.

Die Diagnosewahrscheinlichkeiten sind in der Medizin sehr wichtig und werden direkt über die beobachteten Fälle errechnet. Die obige Liste der Situationen erhält die Bezeichnungen

Situation (Diagnose, Realität)	**Name**	Anzahl der beobachteten Fälle
$(D(x)= \omega \mid \omega)$	True Positive	TP
$(D(x)= \neg\omega \mid \omega)$	False Negative	FN
$(D(x)= \omega \mid \neg\omega)$	False Positive	FP
$(D(x)= \neg\omega \mid \neg\omega)$	True Negative	TN

Tabelle 2 *Diagnosesituationen*

wobei beispielsweise mit „*True Positive*“ gemeint ist, dass die Entscheidung auf „*Positive*“ (Klasse ω liegt vor) richtig (*True*) war. Die Gesamtmenge *M* der Patienten unterteilt sich in vier Mengen M_1,M_2,M_3,M_4, die jeweils die Patienten einer Situation enthalten. Es gelten mit den obigen Bezeichnungen

$$|M_1| = TP,\ |M_2| = FN,\ |M_3| = FP,\ |M_4| = TN \text{ mit } |M| = TP+FN+FP+TN$$

Die Verbundwahrscheinlichkeit einer Situation ist

$$P(D(x)=\omega,\omega) = \frac{TP}{TP+FN+FP+TN},\ P(D(x)=\neg\omega,\neg\omega) = \frac{TN}{TP+FN+FP+TN}$$

$$P(D(x)=\neg\omega,\omega) = \frac{FN}{TP+FN+FP+TN},\ P(D(x)=\omega,\neg\omega) = \frac{FP}{TP+FN+FP+TN}$$

Für die gemeinsamen Mengen $M_1 \cup M_2$ sowie $M_3 \cup M_4$ gelten

$$P(\omega) = \frac{TP+FN}{TP+FN+FP+TN},\ P(\neg\omega) = \frac{TN+FP}{TP+FN+FP+TN},$$

Damit können wir nun die Sensitivität und Spezifität errechnen zu

$$\begin{aligned} P_L &= P(D(x)=\omega \mid \omega) \equiv \frac{P(D(x)=\omega,\omega)}{P(\omega)} = \frac{TP}{TP+FN} \\ P_K &= P(D(x)=\neg\omega \mid \neg\omega) \equiv \frac{P(D(x)=\neg\omega,\neg\omega)}{P(\omega)} = \frac{TN}{TN+FP} \end{aligned} \tag{2.77}$$

Seien die Wahrscheinlichkeiten der übrigen Ereignisse mit $P_A = P(\text{Fehlalarm}) = 1-P_K$ und $P_I = P(\text{Ignoranz}) = 1-P_L$ notiert. Im Idealfall sind die Wahrscheinlichkeiten P_L und P_K der Sensitivität und Spezifität für die Diagnose eins, und die beiden Wahrscheinlichkeiten P_A und P_I sind null. Leider ist dies aber nicht möglich: Alle Diagnosesysteme machen Fehler. Meist kann man eine der beiden Wahrscheinlichkeiten (P_L bzw. P_K) immer nur auf Kosten der anderen minimieren. Die gegenseitige Abhängigkeit lässt sich als Grafik visualisieren, bezeichnet als ROC-Kurve (*Receiver Operating Characteristic*). Ein Beispiel dafür ist in Abb. 2.19 zu sehen. Die ideale ROC-Kurve ist ein Rechteck, das als gepunktete Line eingezeichnet ist.

Jedes Diagnosesystem ist mit seinen Parametern so eingerichtet, dass es an einem Punkt (P_K,P_L) auf der Kurve, dem Arbeitspunkt, betrieben wird. Die Wahl des Arbeitspunktes ist dabei typisch für die Anwendung und hängt von den Benutzervorgaben ab: Bei einem Bombensuchsystem wird man Wert auf hohe Entdeckungswahrscheinlichkeit legen und dafür eine hohe Zahl von Fehlalarmen einplanen; viele medizinische Diagnosegeräte (z.B. Atemmonitore) werden aber nicht mehr genutzt, wenn sie zu oft einen Fehlalarm generieren.

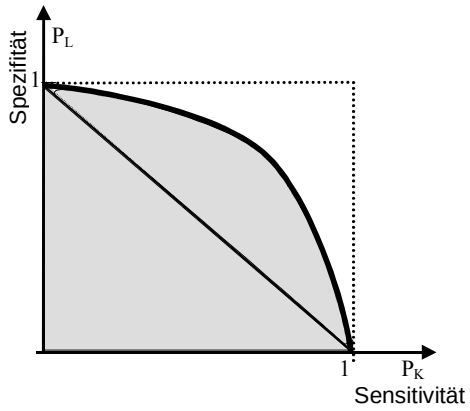


Abb. 2.19 *Typische ROC-Kurve und AUC (schraffiert) eines Diagnosesystems*

Jede ROC-Kurve, die bei gegebener Sensitivität P_K eine höhere Spezifität P_L aufweist, ist an dieser Stelle besser; ihr entspricht eine bessere Diagnose. Da dies aber an anderer Stelle schlechter sein kann, wird üblicherweise die allgemeine Güte des Diagnosesystems durch die Gesamtkurve mittels der Fläche unter der ROC-Kurve (*area under curve* AUC) charakterisiert. Die AUC ist in Abb. 2.19 schraffiert dargestellt.
Ist eine Diagnose unabhängig von der Realität, also $P_L = P(D(x)=\omega|\omega) = P(D(x)=\omega|\neg\omega)$, so ist mit $P(D(x)=\omega|\neg\omega) = 1- P(D(x)=\neg\omega|\neg\omega) = 1- P_K$ die Diagnoseline eine Gerade mit der Steigung –1, also eine Diagonale von (0,1) nach (1,0). Es ist damit klar, dass für nicht-zufällige Diagnosesysteme die AUC der ROC größer als 0,5 sein muss. Ist sie dies nicht der Fall, so muss man nur die Diagnoseentscheidungen umdefinieren, also auf die komplementären Variablen P_A und P_I übergehen, da ja mit $P_K+P_A+P_L+P_I = 1+1 = 2$ bei $P_K+P_L < 1$ sofort $P_A+P_I > 1$ gelten muss und die entstehende ROC über der Diagonalen liegt und damit einen AUC > 0,5 zur Folge hat.

Als Konsequenz von dem, was hier gesagt wurde, müssen wir uns hüten, Diagnosesysteme mit Aussagen wie „Diagnosesystem A hat 90% Erfolg, Diagnosesystem B hat nur 80% Erfolg. Also ist System A besser als System B“ charakterisieren zu wollen. Wenn $P(D_A(\mathbf{x})=\omega|\omega) = 0{,}9$ und $P(D_B(\mathbf{x})=\omega|\omega) = 0{,}8$ gelten, so muss man immer fragen: und die anderen Wahrscheinlichkeiten? Liegt beispielsweise der Arbeitspunkt von System A bei (0,3 0,9) und von System B bei (0,5 0,8), so ist wahrscheinlich System B besser obwohl die Spezifität schlechter ist; endgültig entscheidet dies aber der AUC-Wert der Systeme.

Eine Alternative zum AUC-Wert ist die Angabe für P_L am Arbeitspunkt mit $P_L = P_K$, dem Punkt der *equal error rate* (EER). Manchmal wird auch stattdessen der Mittelwert $P_K+P_L/2$ genommen, was aber nicht so genau wie der AUC-Wert ist.

Man beachte bei dieser Modellierung, dass die ROC-Kurve von einem realen System meist nur als statistische Näherung vieler Messungen, also als „verrauschte“ Kurve, gemessen werden kann und deshalb in der Praxis nur näherungsweise in Optimierungsversuche eingehen kann. Ein Beispiel dafür ist in Abb. 2.20 zu sehen.

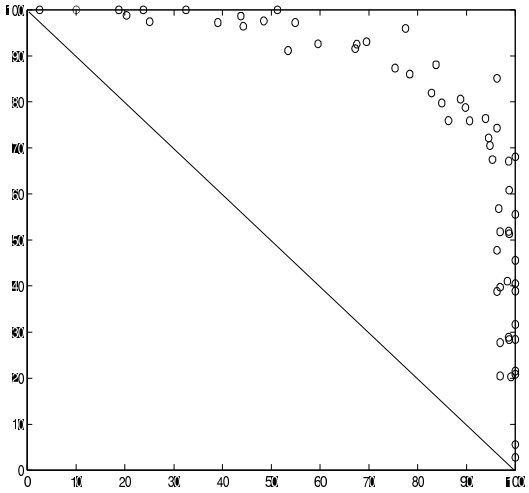


Abb. 2.20 *ROC-Kurve eines medizinischen Diagnosesystems*

Dabei sind die Diagnosewahrscheinlichkeiten „Sensitivität“ und „Spezifität“ auf verschiedenen Datenbasen (Trainingsdaten) und für verschiedene Parameterwerte (Entscheidungsschwellen θ) gemessen worden. Das Ergebnis (P_K, P_L) ist als Punkt im Diagramm eingezeichnet. Die mittlere ROC-Kurve erhält man dann durch Mittelwertbildung über alle Punkte.

2.4.3 Risikobehaftete Klassifizierung

An unserem Beispiel der Klassifizierung von Tumorpatienten in Kapitel 1.3.2 sehen wir ein weiteres Problem: Es reicht nicht aus, nur die Trennung zwischen den Klassen nach der größten Wahrscheinlichkeit (Bayes-Klassifikation) durchzuführen, wir müssen auch die Folgen bedenken, die eine falsche Einordnung mit sich bringt. Bei diesem Beispiel sind sie asymmetrisch: eine falsche Beurteilung der Tumorpatienten verhindert eine schnelle Operation und senkt ihre Lebenserwartung; eine falsche Einordnung der chronisch entzündeten Patienten dagegen bewirkt eine unnötige Operation mit all ihren Risiken. Um beide Entscheidungen gegeneinander abwägen zu können, müssen beide Risi-

koarten in einem gemeinsamen Maßsystem (Sterberisiko etc) numerisch konkretisiert werden, was allerdings in jedem Krankenhaus unterschiedlich ausfallen wird.

Wie müssen nun die Klassengrenzen gewählt werden, um bei bekannten Einzelrisiken das erwartete Gesamtrisiko einer Fehlentscheidung zu minimieren?
Angenommen, unser Musterraum $\Omega := \{\mathbf{x}\}$ ist in Untermengen (Klassen) Ω_i unterteilt und wir bezeichnen das Ereignis ω_i, dass $\mathbf{x}$ aus Ω_i stammt, mit "Klasse Ω_i liegt vor". Dann sei die *Strafe*, das *Risiko* oder die *Kosten* dafür, dass für Klasse Ω_i entschieden wird, obwohl Ω_j vorliegt, mit r_{ij} bezeichnet. Das Risiko, ein $\mathbf{x}$ fälschlicherweise in eine Klasse Ω_i einzuordnen anstatt in Ω_j, ist dann

$$r_i(\omega_i|\mathbf{x}) = \sum_j r_{ij}\, P(\omega_j|\mathbf{x}) \qquad \textit{Bayes-Risiko} \tag{2.78}$$

Mit dem Erwartungswertoperator $\langle\ \rangle$

$$\langle f(\mathbf{x})\rangle := \sum P(\mathbf{x})\, f(\mathbf{x}) \tag{2.79}$$

ist das erwartete Risiko für alle $\mathbf{x}$ aus Ω_i

$$R_i = \langle r_i(\omega_i|\mathbf{x})\rangle_{x\in\Omega i} = \sum_{\Omega i} \sum_j r_{ij}\, P(\omega_j|\mathbf{x})\, P(\mathbf{x}) \tag{2.80}$$

und das *erwartete Gesamtrisiko* aller $\mathbf{x}$ aus Ω ist bei der Klasseneinteilung mit $\mathbf{w}$

$$R(\mathbf{w}) = \sum_i R_i = \sum_i \sum_{\Omega i} \sum_j r_{ij}\, P(\omega_j|\mathbf{x})\, P(\mathbf{x}) \tag{2.81}$$

Ist die Strafe nur abhängig von der falsch gewählten Klasse, so ist $r_i = r_{ij}$. Mit der Beziehung

$$\begin{aligned} \Sigma_j P(\omega_j|\mathbf{x}) &= \frac{1}{P(\mathbf{x})} \Sigma_j P(\omega_j|\mathbf{x})P(\mathbf{x}) = \frac{1}{P(\mathbf{x})} \Sigma_j P(\Omega_j \cap \mathbf{x}) \\ &= \frac{1}{P(\mathbf{x})}\, P\left(\bigcup_j \Omega_j \cap \mathbf{x}\right) = \frac{1}{P(\mathbf{x})}\, P(\Omega \cap \mathbf{x}) = \frac{1}{P(\mathbf{x})}\, P(\mathbf{x}) = 1 \end{aligned} \tag{2.82}$$

lässt sich Gl.(2.81) vereinfachen zu

$$R(\mathbf{w}) = \sum_i \sum_{\Omega i} r_i\, P(\mathbf{x}) = \sum_i \langle r_i(\mathbf{w},\mathbf{x})\rangle_{x\in\Omega i} := \sum_i R_i(\mathbf{w}) \tag{2.83}$$

mit dem erwarteten Einzelrisiko $r_i := r_i\, P(\mathbf{x})$, $\mathbf{x}$ aus Ω_i.

Das erwartete Gesamtrisiko ist damit nur noch von den Klassengrenzen und dem beobachteten $P(\mathbf{x})$ abhängig und nicht mehr explizit von den Wahrscheinlichkeiten $P(\omega_i)$ der Quelle. Das Ziel des Lernprozesses wird damit, eine derartige optimale Klasseneinteilung zu finden, dass das erwartete Gesamtrisiko minimal wird:

Suche w so, dass bei gegebenen r_i und $p(\mathbf{x})$*

$$R(w^*) = \min_{\mathbf{w}} R(w) \tag{2.84}$$

Es lässt sich zeigen ([FU68], Kap.7.3), dass eine Variation der Klassengrenzen keinen Anteil an der Minimierung der Zielfunktion hat. Für ein lokales Minimum ist es damit hinreichend, ein $\mathbf{w}^*$ zu finden, für das jedes einzelne $J_i(\mathbf{w})$ minimal wird.

2.5 Klassifikation mit Multilayer-Perzeptrons

In ersten Kapitel sahen wir, dass formale, binäre Neuronen die Fähigkeit besitzen, Muster zu klassifizieren; die Gewichte bestimmen die Klassengrenzen. Sind die Gewichte entsprechend eingestellt, so lässt sich damit eine Bayes-Klassifikation aus Abschnitt 2.4.1 durchführen.

Die Gewichte und damit die Klassengrenzen lassen sich trainieren; die ersten Algorithmen dazu wurden von Rosenblatt für das berühmte Perzeptron mit binären Neuronen entwickelt, s. Kapitel 2.2.1.

In diesem Abschnitt beleuchten wir nun genauer, welche Fähigkeiten ein System aus mehreren, hintereinander geschalteten, *binären* Klassifikationsschichten besitzt und welche Trainingsalgorithmen dafür existieren. Betrachten dazu zunächst die Problematik, mit binären Neuronen die XOR-Funktion darzustellen.

2.5.1 Nichtlineare Schichten und das XOR Problem

Eine der wichtigsten nichtlinearen Funktionen eines Systems ist die Klassifikation. Im Kapitel 1lernten wir mit binären formalen Neuronen auch den Begriff der "linearen Separierbarkeit" von Mustermengen kennen. Betrachten wir nun in Abb. 2.21 eine kompliziertere Klasseneinteilung, die zwei nicht-zusammenhängende Klassen {X} und {◊} enthält.

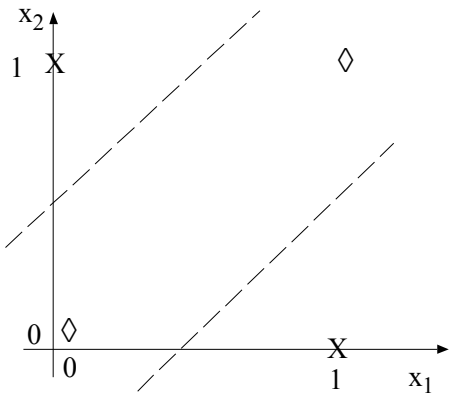
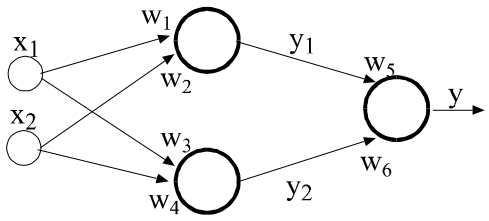


Abb. 2.21 *Zwei nichtlinear trennbare Klassen*

Diese Situation ist beispielsweise für die Funktion "exklusiv-oder" $XOR(x_1,x_2) = \overline{x}_1 x_2 + x_1 \overline{x}_2$ der beiden Variablen x_1 und x_2 gegeben. Wie man leicht sehen kann, lassen sich die beiden Mengen der Klassen XOR(.) = 0 und XOR(.) = 1 nicht durch eine Gerade trennen, sie sind *nicht linear separierbar.* Für dieses Problem gibt es also keine Koeffizienten **w**, die eine solche Klassifizierungsfunktion XOR mit einem einzigen solchen Neuron implementieren würde. Dieser Sachverhalt, der typisch für einschichtige Perzeptrons ist, führte in den 60-Jahren zur Enttäuschung über das Versagen neuronaler Netze selbst bei derart "einfachen" Problemen und einem Rückzug vieler Forscher aus diesem Gebiet.

Erst erweiterte Konzepte, wie das der mehrschichtigen Netzwerke, zeigten später Wege auf, das obige Problem zu überwinden. Dabei geht man davon aus, dass jede Gerade eine Klassengrenze darstellt, die von einem formalen Neuron implementiert werden kann. Betrachten wir dazu unser Beispiel der XOR Funktion. Zwei parallel arbeitende Neuronen können mit einer UND Verknüpfung die Unterscheidung zwischen den Mengen der Klassenpunkte {(0,1)} und {(0,0), (1,0), (1,1)} sowie zwischen {(1,0)} und {(0,0), (0,1), (1,1)} realisieren. Aus den Zuständen dieser Neuronen lässt sich nun wiederum mit Hilfe einer ODER Verknüpfung auf die Klasse schließen, so dass dieses neuronale Netz direkt die Bool'sche Funktion des XOR implementiert. In Abb. 2.22 ist ein solches Netzwerk gezeigt.



$y_1 := x_1 \text{ AND } \bar{x}_2$
$y_2 := \bar{x}_1 \text{ AND } x_2$
$y_{XOR} := y_1 \text{ OR } y_2$

$\Rightarrow w_1=w_4=w_5=w_6= ½$
$w_2 = w_3 = – ½$
$s_1= s_2 = ?\quad s = ?$

Abb. 2.22 *Ein Neuronales Netz für die XOR Funktion*

Die Konstruktion von Mehrschichten-Netzwerken erlaubt also, auch kompliziertere Klassenformen durch stückweise lineare Separierung als Flächenstücke ("Patchwork") zu trennen. Die Anzahl der Schichten und die Zahl der Neuronen pro Schicht hängt, wie an dem Beispiel deutlich wird, stark von dem betrachteten Klassifizierungsproblem ab. Betrachten wir zunächst einfache Fälle.

2.5.2 Multilayer-Klassifikation durch Hyperebenen

Ein binäres Neuron, wie es beim Perzeptron verwendet werden, implementiert eine lineare Separierung. Dabei unterteilt es durch seine binäre Ausgabefunktion

$$y = S_B(z) := \begin{cases} 1 & z > 0 \\ 0 & z \le 0 \end{cases} \tag{2.85}$$

den Eingaberaum mit einer Hyperebene in zwei Mengen (Klassen). Wollen wir Punkte (Muster) in der Nähe eines Klassenprototypen $\mathbf{w}_A$ zu einer Klasse zusammenfassen, so lässt sich dies über lineare Diskriminanzfunktionen erreichen. Anschaulich gesprochen werden dazu mehrere Klassengrenzen (Geraden bzw. Hyperebenen) um $\mathbf{w}_A$ errichtet, wobei die Ausgabe S_B jeder einzelnen Klassenentscheidung gerade so gewählt wird, dass sie auf der Seite von $\mathbf{w}_A$ positiv ist und auf der anderen Seite der Grenze negativ. In Abb. 2.21 ist die Situation in der Ebene gezeigt.

Eine solche Klassifizierungsmechanismus wird auch als "stückweise lineare Maschine" [NIL65] bezeichnet. Allerdings impliziert diese Art linearer Separierung einen Vorgang der Maximumsbildung, der in der einen Neuronenschicht nicht

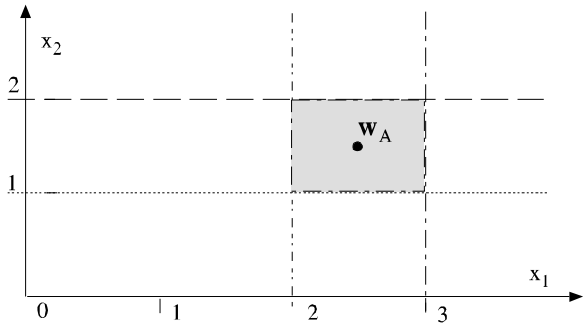


Abb. 2.23 *Klassentrennung durch Hyperebenen*

enthalten ist. Um eine nicht-lineare Klassengrenze (z. B. eine Begrenzung für eine "Insel" von Datenpunkten) zu erreichen, muss man mindestens eine zweite Schicht vorsehen. In Abb. 2.24 ist dazu ein Netz gezeigt.

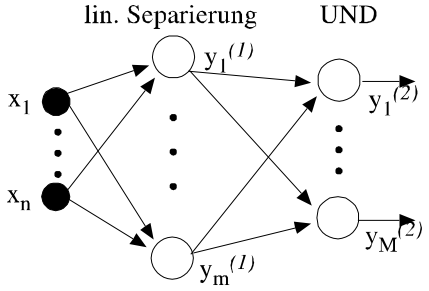


Abb. 2.24 *Ein Netz für Klassifizierung mit Hyperebenen*

Man kann sich die Funktion eines Netzes von binären Neuronen so vorstellen, dass die erste Schicht den Eingaberaum durch viele, sich schneidende Hyperebenen in Gebiete oder Parzellen unterteilt, die sich überlappen oder auch enthalten können. Erst in der zweiten Schicht wird mit einer Art "UND"-Verknüpfung das gewünschte Gebiet definiert. Gehören mehrere, getrennte Flecken zur selben Klasse, so können wir in einer weiteren, dritten binären Schicht mit einer "ODER"-Verknüpfung die Zusammenziehung mehrerer Klassenflecken zu einer einzigen Klassenentscheidung bewirken. In der folgenden Abbildung ist dies schematisch für eine schraffiert gezeigte Punktmenge gezeigt.

Alle drei Entscheidungen für jeweils eine Hyperebene wirkenüber UND zusammen wie eine Entscheidung für die Punktmenge innerhalb des Dreiecks.

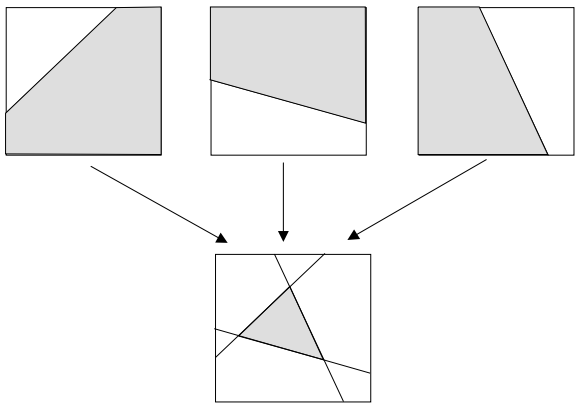


Abb. 2.25 *Zusammenwirken von drei Entscheidungen an Hyperebenen*

Dies lässt sich auch durch eine Überlagerung von sigmoidalen Ausgabefunktionen erreichen. Bei der Funktion $z = \mathbf{w}^T\mathbf{x} - s$ bedeutet eine Veränderung des Parameters s eine Verschiebung (Translation) des Funktionswerts. Dabei bildet der Übergang von null auf eins eine Kante; im dreidimensionalen ist dies eine unendlich lange Kante ähnlich einer Treppenstufe. Zeichnen wir nun für das XOR-Problem die sigmoidale, binäre Ausgabefunktion $y = 1/2(y_1+y_2)$ als Überlagerung der beiden Eingaben y_1 und y_2, also der beiden Stufenfunktionen $S_B(z_1)$ und $S_B(z_2)$, als

$$y = 1/2 \left(S_B(z_1(\mathbf{x})) + S_B(z_2(\mathbf{x})) \right) \tag{2.86}$$

in einer Grafik, so ergibt sich als drei-dimensionale Visualisierung der Klassengrenzen die Überlagerung der beiden Stufen in Abb. 2.26.

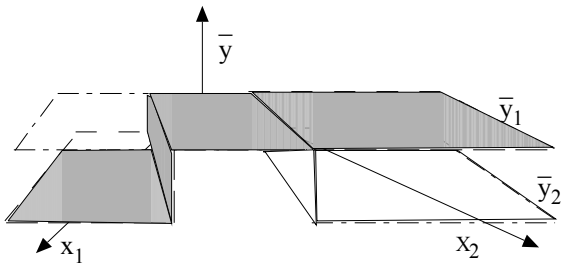


Abb. 2.26 *Die (negierte) Ausgabefunktion des XOR-Netzwerks*

Wie wir beobachten können, verläuft die Ausgabefunktion von z_1 (grau schraffiert) durch $\mathbf{w}_1 = -\mathbf{w}_2$ gerade invers zu der von z_2; zusammen bildet die Summe der Stufenfunktionen einen Bereich ("Balken"), der die mittlere Zone "herausschneidet". Die passende Verschiebung der Stufenfunktionen, um eine passende Überlappung zu erreichen, wird dabei durch die Größe der Schwellwerte erreicht; die Gewichte kodieren die Orientierung der Stufenkanten (Lage der Geraden im Raum (x_1,x_2)).
Diesen Gedanken können wir auch auf die nicht-binären, sigmoidalen Ausgabefunktionen übertragen. In Abb. 2.27 ist links eine zweidimensionale sigmoidale Funktion zu sehen.

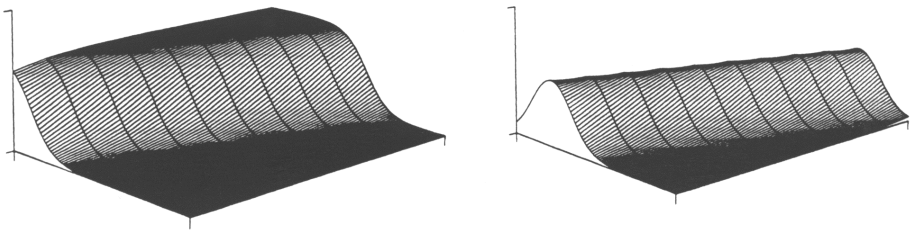


Abb. 2.27 *Sigmoidale Ausgabefunktion und eine Überlagerung* (aus [LAP88])

Überlagern wir in einem zweischichtigen Netzwerk zwei solcher "abgerundeten Stufenfunktionen" als Ausgabe zweier Neuronen der ersten Schicht in geeigneter Weise (negatives Gewicht und geeignetes s), so erhalten wir eine "Welle", siehe Abb. 2.27 rechts.

Die Überlagerung von vier Neuronen kann als Überlagerung von zwei solcher Wellen angesehen werden, die zu einer starken Erhöhung am überlappenden Kreuzungspunkt der Wellen führt, siehe Abb. 2.28 links.

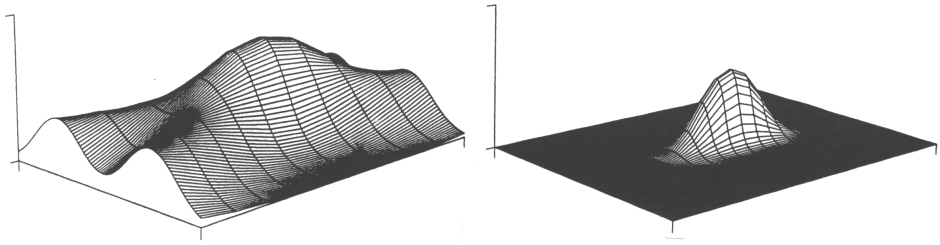


Abb. 2.28 *Überlagerung von vier Ausgabefunktionen (aus [LAP88])*

Subtrahieren wir von dieser Aktivität einen konstanten Pegel (Schwellwert) und schneiden in der nachfolgenden, zweiten Neuronenschicht mit Hilfe der Ausgabefunktion den negativen Anteil ab, so erhalten wir als Eingabemenge, für die dieses Neuron der zweiten Schicht empfindlich ist, eine kleine Untermenge von {**x**}, s. Abb. 2.28 rechts. Die nicht-linearen Funktionen wirken wie ein "Abtastmechanismus" *(Sampling)*, der für die Ausgabe der zweiten Schicht nur einen schmalen Bereich von {**x**} erfaßt.

Damit hat die zweite Schicht die Aufgabe, durch eine lineare Überlagerung lokal definierter, nicht-linearer Kurvenstücke *(splines)* die gewünschte nicht-lineare Funktion vom gesamten Raum {**x**} angenähert zusammenzusetzen. Die lineare Überlagerung nicht-linearer Basisfunktionen ist die Grundidee der Mehrschichtennetzwerke in den folgenden Abschnitten dieses Kapitels.

2.5.3 Stückweise lineare Approximation einer nicht-linearen Funktion

Angenommen, ein zweischichtiges Netzwerk soll die quadratische Abbildung

$$F(x) = ax^2+b \tag{2.87}$$

approximieren. Es gibt also nur eine Eingabekomponente (n=1) und eine Ausgabe (m=1) beim Netz, s. Abb. 2.29. Zur Vereinfachung des Beispiels sollen *lineare*

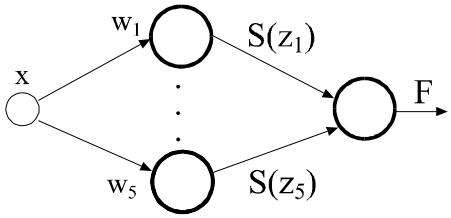


Abb. 2.29 *Das zweischichtige Netzwerk zur Approximation von F(x)*

splines verwendet werden: die Ausgabefunktion eines Neurons der ersten Schicht soll also begrenzt-linear sein.

$$y = S_L(z,T) := \begin{cases} z_{max} & z_T \le z \\ kz & 0 < z < z_T \quad k=z_{max}/z_T \\ 0 & z \le 0 \end{cases} \tag{2.88}$$

(üblich: $z_{max}:=1$, $z_T=1$,$k:=1$)

Die Ausgabefunktion des zweiten Neurons soll die Identität sein.
Diese Form der Ausgabefunktion ist deshalb besonders interessant, weil sie ohne große Probleme in VLSI implementiert werden kann; bei z = wx + s entspricht sie einem linearen Analogverstärker mit den durch die Versorgungsspannung gegebenen Verstärkungsgrenzen 0 und z_{max}.

Mit k:=1 und z_{max}:=1 ist die Bedingung $S(\infty) = 1$, $S(-\infty) = 0$ des Satzes von Hornik, Stinchcomb und White aus Abschnitt 2.5.5 erfüllt. Überdecken sich die Intervalle der linearen Ausgabe der einzelnen Neuronen vollständig, so resultiert wieder eine lineare Ausgabe, so dass in diesem Fall die Parabel sehr schlecht durch eine einfache Gerade approximiert werden könnte. Erst die Überlagerung des linearen Bereichs einer Ausgabefunktion für ein Eingabeintervall mit einer konstanten, nicht-linearen Ausgabe eines zweiten Neurons ermöglicht ein weiteres Geradenstück mit einer anderen Steigung und damit eine bessere Approximation. Allgemein erreichen wir bei diesem Beispiel also eine gute Approximation durch *m* Polygonzüge, wenn wir das Eingabeintervall $[X_0, X_1]$ in *m* Teilintervalle Δx aufteilen (gleiche Intervallbreiten ergeben gleichen, minimalen Maximalfehler, s. [BRA93c]) und in jedem Teilintervall gerade ein Neuron linear aktivieren. In Abb. 2.30 ist die Funktion f(x), überlagert mit der approximierenden Netzwerkausgabe F(x) im k-ten Intervall

$$F(x) = \sum_{i=1}^{m} W_i \, S(z_i) + T = \sum_{i=1}^{k-1} W_i + W_k \, S(z_k) + T \tag{2.89}$$

für *m* = 5 Neuronen zu sehen. Die vertikalen, gepunkteten Linien sind die Intervallgrenzen; die gestrichelte Linie die quadratische Funktion. Man sieht, dass selbst bei einer

solch groben Intervallteilung kein großer Unterschied zwischen der Originalfunktion und der approximierenden Funktion besteht.

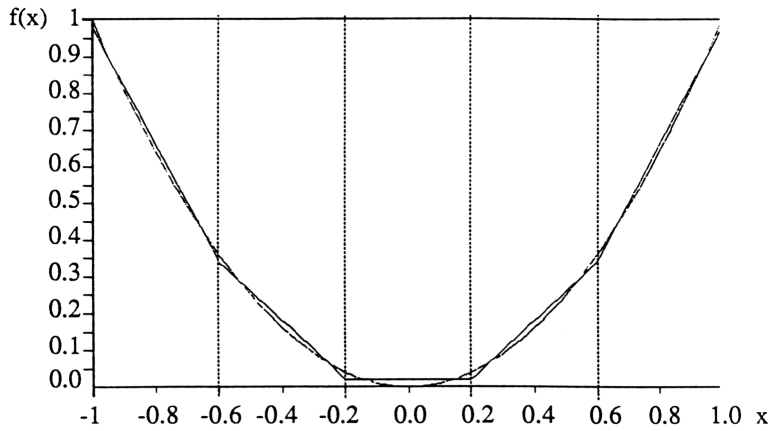


Abb. 2.30 *Die quadratische Funktion und ihre Approximation durch m= 5 Neuronen*

Wie verhalten sich die Neuronen in der Approximation? In der Abb. 2.31 sind die einzelnen Ausgabefunktionen der Neuronen separat gezeichnet. Die Ausgaben sind über einander durchgehend schwarz gezeichnet; die Nulllinien jeweils gepunktet. Jede Ausgabe steigt innerhalb des Intervalls Δx linear von 0 auf 1 an, siehe Gleichung (2.89), und bildet mit einem Gewicht W_k (Steigung) genau eine Gerade des Polygonzuges. Alle anderen Neuronen sind entweder nicht aktiv oder im "Sättigungsbereich" der Ausgabefunktion und bilden einen konstanten, additiven Term. Woher erhalten wir die Parameter, die Gewichte des Netzwerks?

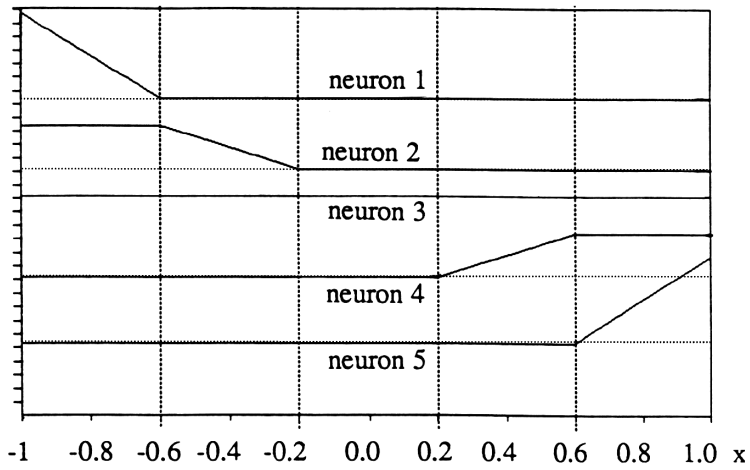


Abb. 2.31 *Die Einzelfunktionen der m=5 Neuronen*

Wir können in diesem Fall direkt analytisch die Gewichte und Schwellen w_i und s_i der ersten Schicht und W_i und T der zweiten Schicht ausrechnen. Dies geht folgendermaßen: Mit den Bedingungen aus Gl.(2.88) erhalten wir

$$z|_{xi-\Delta x/2} = 0 \quad z|_{xi+\Delta x/2} = 1 \tag{2.90}$$

und mit den vorigen Überlegungen erhalten wir für $z = wx + s$

$$\begin{aligned} w_i &= 1/\Delta x = m / (x_1-x_0) \\ s_i &= -w_i(x_i-\Delta x/2) = (x_0/\Delta x)+1-i = -mx_i/(x_1-x_0) +1/2 \end{aligned} \tag{2.91}$$

Die Gewichte W_i der zweiten Schicht wählen wir so, dass in jedem Intervall die approximierende Gerade die Tangente von f(x) in x_i darstellt

$$\frac{\partial f(x_i)}{\partial x} = \frac{\partial\left(ax^2 + b\right)}{\partial x} x_i = 2ax_i := \Delta y/\Delta x \tag{2.92}$$

Da die Ausgabe S(z) der Neuronen der ersten Schicht zwischen Null und Eins normalisiert ist, sind die Gewichte W_i als normalisierte Tangente $\Delta y/1$

$$W_i := \Delta y/1 = 2ax_i\,\Delta x \tag{2.93}$$

Der allgemeine konstante Term T von F(**x**) wird damit die Drift im Punkt x_0. Mit dem linearen Fehler d_{lin} am Intervallrand x_0 von [BRA91] (s. Abb. 4.12.5) erhalten wir

$$T = f(x_0) - d_{lin} = ax_0^2 + b - a/2\,(\Delta x/2)^2 \tag{2.94}$$

Ein solches analytisches Vorgehen vereinfacht dabei eine VLSI-Implementation des neuronalen Controllers, da wir anstelle der zur Zeit noch technologisch sehr problemtisch realisierbaren Lernverfahren für Gewichte die fest ausgerechneten Gewichtswerte in konventioneller ROM-Technik implementieren können.

Die Gewichte der ersten Schicht skalieren das Teilintervall Δx der Eingabe auf das Intervall [0,1], und der konstante Term s_i jedes Neurons verschiebt den Nullpunkt der Ausgabefunktion geeignet. Die so normierte, linear transformierte Ausgabe wird beim Ausgabeneuron erneut linear transformiert, diesmal so, dass der entstehende Geradenausschnitt parallel zur Tangente (Ableitung) der Funktion im Mittelpunkt des Teilintervalls ist und mit den anderen Geradenstücken einen durchgehenden Polygonzug bildet.

Die Approximation einer quadratischen Funktion durch einen Polygonzug stellt einen Spezialfall des allgemeinen Vorgehens dar, beliebige Kurven stückweise durch Teilstücke besonderer Kurvenformen (*splines*) anzunähern. Im multi-dimensionalen Fall sind dies Hyperflächen; die Geraden des Polygons werden dabei zu Hyperebenen, s. Abb. 2.32.

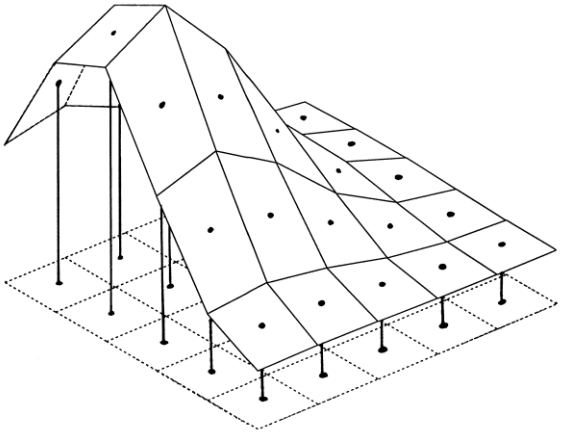
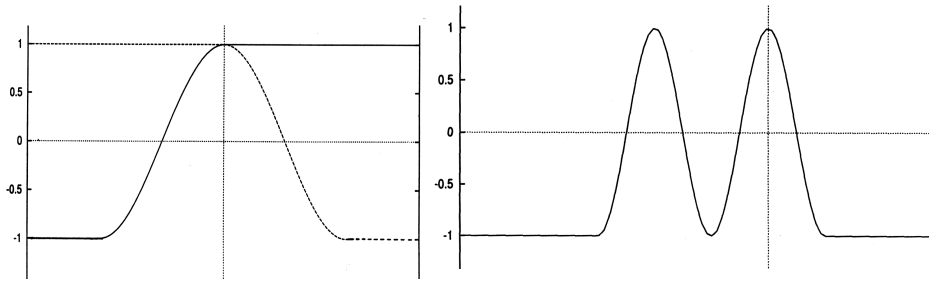


Abb. 2.32 *Stückweise Approximation durch Hyperebenen*

Die allgemeine Approximationstheorie kennt eine Vielzahl von *spline*-Formen mit unterschiedlichen Eigenschaften. Da die Implementierung solcher speziellen Hyperflächen auf einem Rechner eine hohe Rechenleistung fordert, werden für praktische Probleme meist besondere *splines* (z.B. B-*splines*) eingesetzt. Allerdings führen solche nichtlinearen Interpolationen nicht unbedingt zu besseren Ergebnissen; vielfach ist die Zahl der Stützstellen und nicht die Kurvenform für den Approximationsfehler ausschlaggebend [WAT83].

2.5.4 Approximation durch Wellenfunktionen

Betrachten wir erneut die Entwicklung nach Basisfunktionen in der Basisgleichung (2.99). Wir wissen, dass mit einer unendlichen Summe von gewichteten Kosinus (bzw. Sinus-) funktionen eine Fouriertransformation implementiert wird, mit der man jede gewünschte periodische Funktion exakt darstellen kann. Ist eine nicht-periodische Funktion auf ein endliches Intervall beschränkt, so kann man durch eine fiktive Replikation der Funktion in den Nachbarintervallen aus der nicht-periodischen Funktion eine periodische machen, so dass man in der Praxis jede beliebige Funktion durch eine diskrete Fouriertransformation mit einer endlichen Summe approximieren kann. Wäre die Quetschfunktion eine periodische Sinus- (oder Kosinus-)funktion, so wäre damit ein konkretes Approximationsnetz mit den bekannten Approximationseigenschaften der Fouriertransformation gegeben.
Diese Voraussetzung lässt sich nun leicht mit Hilfe der Kosinus-Quetschfunktion $S_C(z)$ aus dem ersten Kapitel herstellen. Fügen wir zu einer Funktion $S_C(z)$ noch eine weitere hinzu, die ein negatives Argument hat, so bildet die Summe aus beiden, jeweils um den richtigen Betrag verschoben, gerade eine Kosinuswelle, s. Abb. 2.33 (a).



(a) (b)

Abb. 2.33 *Überlagerung von (a) zwei und (b) vier Kosinusfunktionen*

Verschieben wir die Ausgabe eines solchen Paares um eine Periodenlänge und addieren wir die Ausgabe dazu, so resultiert ein Sinus (oder Kosinus) mit zwei Perioden. Wiederholungen dieser Prozedur führt schließlich zu einem kontinuierlichen Sinus (oder Kosinus) im gesamten, betrachteten Intervall, siehe Abb. 2.33 (b).

$$\cos(x)-1 = \sum_{k=-p}^{+p} 2\,(S_C(-x-2\pi k+\pi/2) + S_C(x+2\pi k-3\pi/2) -1) \qquad (2.95)$$

$$\text{für } -2p\pi \le x \le 2\pi(p+1)$$

Unterschiedliche Konstanten führen zu unterschiedlichen Kosinuswellen, so dass das gesamte Netz aus zwei Schichten als Linearkombination von Kosinusfunktionen eine *diskrete Kosinustransformation* bzw. mit Sinus-Einheiten (um 90 Grad phasenverschobene Kosinuseinheiten) eine *diskrete Fouriertransformation* darstellt [GAL88].

$$f(x) = a_0/2 + \sum_{k=1}^{\infty} [a_k \cos(k\omega x) + b_k \sin(k\omega x)] \qquad a_0 = \text{offset}, \quad \omega = 2\pi/T$$

$$= a_0/2 + \sum_{k=1}^{\infty} c_k \sin(k\omega x + \varphi_k) \qquad \text{mit } c_k = (a_k^2 + b_k^2)^{1/2}, \quad \tan\varphi_k = a_k/b_k \quad (2.96)$$

In einem ähnlichen Ansatz [GIR91] interpretierten andere Autoren die Paare von sigmoidalen Funktionen, deren Überlagerung nach Gl.(2.86) die zu approximierende Funktion ergibt, als lokale Wellenpakete, wie sie beispielsweise in der Quantenmechanik verwendet werden. Die nicht-lokalisierten, ebenen Wellen, die durch die Replikation entstehen, haben im Unterschied zu den lokalisierten "bumps" fehlertolerante Eigenschaften: da die Fourierbasis leicht überbestimmt ist (die Fourierkoeffizienten sind leicht korreliert), hat ein Ausfall einzelner Neuronen keine solchen starken Folgen wie im lokalisierten Fall.

Sind nicht nur die Funktionswerte, sondern auch die analytische Form der zu approximierenden Funktion f(**x**) bekannt, so lässt sich interessanterweise bei diesen Ansätzen durch eine Fouriertransformation von f(**x**) leicht die Fourierkoeffizienten und damit die Gewichte w_i im Netzwerk von F(**x**) nicht durch Lernen, sondern direkt analytisch bestimmen [GIR91].

Die Wellenpakete, deren lineare Überlagerung die unbekannte Funktion approximieren soll, lassen sich in zwei Kategorien einteilen, je nachdem, ob bei den höheren Frequenzen die Pakete schmaler und höher werden (*Wavelets*, s. Lehrbuch [LMR94])

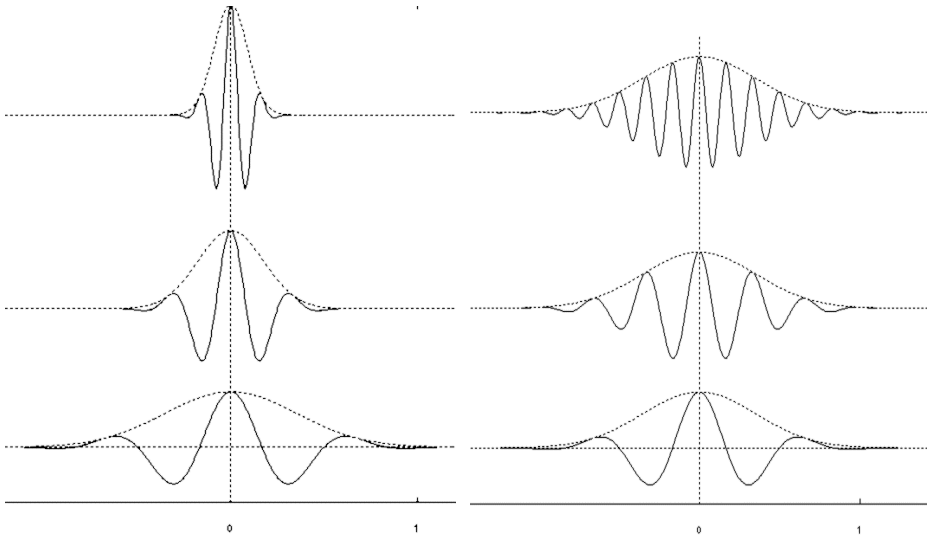


Abb. 2.34 *Basisfunktionen: Wavelets und Gaborfunktionen*

oder gleich bleiben (*Gaborfunktionen*, s.[DAU88]). Um für die Entwicklung einer Funktion nur eine Art von Wavelets benutzen zu müssen, verwendet man eine *Multi-Skalen-Analyse* [LMR94], bei der die großen Wavelets am Platz k, die grobe Details beschreiben, durch Verdopplung der Argumente und der Skalierung aus den kleinen hervorgehen: $\varphi(x,m)_k = 2^{-m/2}\varphi(2^{-m}x-k)$. In Abb. 2.34 sind beide Basisfunktionsarten für die Frequenzverhältnisse 1:2:4 im eindimensionalen Fall mit der Funktion

$$\varphi(x,n)=A(n)\exp(-x^2n^2/\sigma^2)\cos(n\omega x) \qquad (2.97)$$

für n = 1,2,4 dargestellt, wobei mit $\sigma = 0.5$, $\omega = 3\pi$ für die Wavelets $S_j(x) = \varphi(x,n)$, $A(n) = n^{1/2}$ und für die Gaborfunktionen $S_j(x) = \varphi(x,n)$, $A(n) = 1$ gilt.

Ein neuronales Netz, das Wellenpakete als Basisfunktionen verwendet, hat im Prinzip unsere bekannte zweischichtige, in Abb. 2.35 gezeigte Struktur. Die Neuronen der ersten Schicht können als Ausgabefunktionen statt direkt eine Wellenfunktion auch eine Überlagerung mehrerer allgemeiner sigmoidaler Quetschfunktionen haben; beispielsweise ergibt die Überlagerung

$$S(x) = S_F(x+2) + S_F(x-2) - 2S_F(x) \quad \text{mit z.B. } S_F = \text{Fermifunktion} \qquad (2.98)$$

die Grundfunktion eines Wellenpakets [PAK91], das verschoben und skaliert überlagert die Funktionsapproximation nach Gl. (2.99) bildet.

2.5.5 Allgemeine Eigenschaften mehrschichtiger Netze

Die Kombination verschiedener Schichtarten bringt uns neue Funktionen des Gesamtnetzes, die mit einer Schicht vorher nicht möglich waren. Diese Funktionen lassen sich mit dem Begriff der *Approximation* kennzeichnen. Betrachten wir dies genauer.

Die Funktion vieler bisher vorgestellter Netzwerke lässt sich als eine Lösung des Problems charakterisieren, aus einer Darbietung von N Trainingsmustern $\mathbf{x}_i$ und N dazu gewünschten Ausgaben L_i eine allgemeine Funktion, für die bei den Trainingsmustern *(Stützstellen)* $f(\mathbf{x}_i) = L_i$ gilt, mit Hilfe der Funktion F des Netzwerks angenähert zu konstruieren, etwa die Klassifikation eines Musters. Der Fehler, der bei dieser Annäherung *(Approximation)* noch beobachtet wird, kann dazu verwendet werden, die Genauigkeit der Näherung zu verbessern.

Die mathematische Wirkung der neuronalen Approximation kann sehr unterschiedlich sein; von einer einfachen Überlagerung von kontinuierlichen Funktionen bis hin zu einer stückweisen Approximation in Intervallen durch Standard-Kurvenzüge *(Splines)* ist vieles denkbar.

Wir wissen, dass einschichtige Netze ziemlich begrenzte Eigenschaften haben. Fügen wir weitere Schichten hinzu, so können wir die Möglichkeiten des Gesamtnetzes erweitern. Wieviele Schichten benötigen wir nun für ein gegebenes Problem? Betrachten wir dazu zunächst nur Netze aus zwei Schichten, einer nicht-linearen Schicht, die in einer zweiten Schicht von einem linearen Neuron pro Ausgabevariable $y = F(\mathbf{x})$ abgeschlossen wird. In Abb. 2.35 ist dies dargestellt.

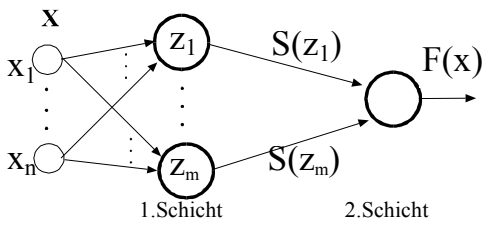


Abb. 2.35 *Ein zweischichtiges Netz*

Da die Ausgabe S_j der Neuronen der ersten Schicht nicht direkt beobachtet werden kann, nennt man sie auch "versteckte Einheiten" (*hidden units*). Formal lässt sich die Netzfunktion schreiben als

$$F(\mathbf{x}) = \sum_{j=1}^{m} w_j S(\mathbf{x}) \tag{2.99}$$

wobei w_j und $F(\mathbf{x})$ aus $\Re$, $\mathbf{x}$ aus $\Re^n$ sind. Die Funktion $F(\mathbf{x})$ lässt sich also durch eine gewichtete Summe aus den Ausgaben *m* formaler Neuronen oder, mathematisch formuliert, als *Linearkombination von* m *Basisfunktionen* S(.) darstellen.

Es ergeben sich dabei zwei Fragen. Zum einen ist unklar, was sich durch ein solches zweischichtiges Netz überhaupt erreichen lässt und zum anderen, was man als Basisfunktionen wählen kann. Untersuchen wir zunächst das erste Problem.

Betrachten wir stellvertretend für viele Arbeiten die bekannte Arbeit von Hornik, Stinchcombe und White [HSW89]. Ähnlich dem Stone-Weierstraß Theorem, das die beliebig dichte Approximation von Funktionen durch Polynome garantiert, zeigten Hornik, Stinchcombe und White, dass mit Hilfe von neuronalen Funktionen eines Netzwerks aus zwei Schichten sich eine vorgegebene Menge von diskreten Funktionswerten direkt darstellen oder in einem Intervall eine kontinuierliche Funktion beliebig dicht approximieren lässt.

Um die verheißungsvollen Aussagen genauer formulieren zu können, betrachten wir eine Klasse von Funktionen, genannt *Sigma-Funktionen* $\Sigma^n(S)$

$$\Sigma^n(S) := \{F / F : \Re^n \to \Re \quad \text{mit } F(\mathbf{x}) = \sum_{j=1}^{m} w_j^{(2)} S(z_j(\mathbf{x}))\} \qquad (2.100)$$

$$\text{wobei } w_j^{(2)} \text{ aus } \Re, \mathbf{x} \text{ aus } \Re^n$$

Die Sigma-Funktionen sind genau die Menge von Funktionen, die mit einem zweischichtigem, neuronalen Netz wie in Abb. 2.35 erzeugt werden. Betrachten wir nun als Aktivitätsfunktion z_i des *i*-ten Neurons der ersten Schicht eine *affine Funktion* wie sie in Kapitel 1 eingeführt wurde, d.h. sie kann Größenänderungen, Drehungen und Verschiebungen eines Vektors $\mathbf{x}$ bewirken

$$z_j \in z^n := \{ z \mid z(\mathbf{x}) = \mathbf{w}^{(1)T}\mathbf{x}+b \} \qquad \textit{affine Funktionen } \Re^n \to \Re \qquad (2.101)$$

wobei $b \in \Re$ die negative Schwelle und $\mathbf{w}^{(1)}$ der Gewichtsvektor eines Neurons der ersten Schicht ist. Die Ausgabefunktion $S(\mathbf{x})$ ("Quetschfunktion") kann dabei beliebig sein, vorausgesetzt, sie erfüllt die Bedingungen

$$\lim_{x\to\infty} S(x) = 1, \quad \lim_{x\to\infty} S(x) = 0 \qquad (2.102)$$

Die zweite Schicht hat wieder eine lineare Aktivitätsfunktion mit den Gewichten $\mathbf{w}^{(2)}$; die Ausgabefunktion ist die Identität $F(x) = S(z^{(2)}) := z^{(2)}$.

Als Maß für die Abweichung zwischen der approximierenden Netzausgabe $F(\mathbf{x})$ und der zu lernenden Funktion $f(\mathbf{x})$ nehmen wir die $L_s(p)$-Norm

$$\|f(\mathbf{x}) - F(\mathbf{x})\| = \left(\int |f(\mathbf{x})-F(\mathbf{x})|^s dP(\mathbf{x})\right)^{1/s} \qquad L_s(p)\text{-Norm} \qquad (2.103)$$

was z.B. für zufällige $\mathbf{x}$ mit der Wahrscheinlichkeitverteilung $P(\mathbf{x})$ der erwarteten Abweichung entlang des ganzen Kurvenzuges von $f(\mathbf{x})$ entspricht; im Fall s = 2 ist dies unser erwarteter quadratischer Fehler. Im Gegensatz dazu entspricht der *uniforme* Abstand

$$||f(\mathbf{x}) - F(\mathbf{x})|| = \sup_x |f(\mathbf{x}) - F(\mathbf{x})| \quad (2.104)$$

dem maximalen Fehler, der bei der Approximation überhaupt auftreten kann, unabhängig von seiner Häufigkeit P(**x**).
Dann gilt:

1) Für die Funktionswerte *jeder beliebigen Funktion* $f(\mathbf{x}) : \Re^n \rightarrow \Re$ von *N* Mustern $\mathbf{x}^1 .. \mathbf{x}^N$ gibt es eine Sigma-Funktion F, so dass für alle Muster $\mathbf{x}^i$ mit i = 1..N gilt

$$F(\mathbf{x}^i) = f(\mathbf{x}^i)$$

Natürlich gilt dies auch für eine ganze Schicht, d.h. für $\mathbf{F} = (F_1, ... ,F_m)$, den Ausgabevektor aus Sigma-Funktionen.

2) Jede beliebige, stetige Funktion f(**x**) in einem kompakten Intervall ("kompakte Teilmenge des $\Re^n$ ") kann beliebig dicht (*uniform dicht* im Sinne der L_s-Norm in der Menge C^n aller stetigen Funktionen und ρ_p-*dicht* in der Menge der Borel meßbaren Funktionen) durch eine Sigma-Funktion F(**x**) approximiert werden.

Die obigen Aussagen gelten übrigens auch für den allgemeineren Fall, wenn die zweite Schicht aus *Sigma-Pi-Units* besteht. In einer späteren Arbeit zeigte Hornik [HORN91], dass für die Quetschfunktion sogar die Eigenschaft "stetig, begrenzt und nicht-konstant" ausreicht (s.a.[KRE91]). Damit ist auch die Klasse der *radialen Basisfunktionen* wie die Gaußsche Glockenkurve in den möglichen Ausgabefunktionen enthalten. Eine Sammlung der Arbeiten dieser Forschungsgruppe ist in [WHI92] zu finden.

Die Ergebnisse der zitierten Arbeiten sind nicht überraschend. Die Grundfunktion des obigen Netzes besteht darin, eine gewünschte Funktion mit Hilfe vieler linear überlagerter Ausgabefunktionen von Neuronen zu erreichen. Dies ist aber ein in der Mathematik bekanntes Problem. Wie bereits Kolmogorov 1956 mit seinem berühmten Theorem [KOL56] zeigte, lässt sich jede beliebige, stetige Funktion mehrerer Variablen durch eine Überlagerung einfacherer Funktionen jeweils einer Variablen darstellen. Allerdings beschränkt sich diese Aussage, genau wie auch viele folgende Arbeiten anderer Autoren [HN87b], [HN89], [CYB88], [CYB89], [FUN89], [HSW89], [HORN91], [IRI88], die auch auf die spezielle Situation neuronaler Netze eingehen, auf die Aussage, dass die Approximation einer Funktion und ihrer Ableitungen [HSW90], [GAL92] durch neuronale Netze möglich ist. Dabei wird allerdings offen gelassen, wie man ein konkretes Netz (Art der Ausgabefunktionen, Zahl der Schichten, Zahl der Einheiten pro Schicht etc) für eine gegebene Aufgabe am Günstigsten auswählt.

Obwohl einige Autoren nicht nur Existenzbeweise, sondern auch konstruktive Beweise präsentieren (s. z.B. [ITO1a], [ITO91b],[ITO92]), die als Algorithmus für die Konstruktion eines Netzes gelten sollen, sind die Konstruktionsmethoden nicht unbedingt einfach und praktikabel.

Trotzdem haben wir eine Antwort auf unsere Frage nach der Zahl der Schichten erhalten: wir wissen nun, dass ein zweischichtiges Netzwerk ausreicht. Allerdings kann

man aus verschiedenen Gründen auch mehr Schichten vorsehen und dabei die Aufgabe des einzelnen Neurons einschränken, wie im folgenden Abschnitt gezeigt wird. Die Entscheidung, ob man wenige Schichten mit vielen Einheiten oder viele mit wenigen Einheiten benutzt, ist weitgehend problemabhängig.

Auch auf die Frage nach der optimalen Zahl der *hidden units* bei zwei Schichten gibt es Hinweise. White zeigte in seiner Arbeit [WHI90], dass die optimale Zunahme der Zahl *m* der *hidden units*, um ein *overfitting* zu vermeiden, von der Statistik und der Zahl *N* der Trainingsmuster $\{\mathbf{x}^i\}$ sowie den Gewichten $\mathbf{w}^{(1)}$ und $\mathbf{w}^{(2)}$ abhängt. Sei eine Zahl d_t gegeben, für die beim *t*-ten Lernschritt gilt

$$d_t \geq 1/m_t \sum_{i=1}^{m} \sum_{j=0}^{n} \left| w_{ij}^{(1)} \right| \qquad \text{bzw. } d_t \geq \sum_{j=0}^{n} \left| w_{ij}^{(1)} \right| \quad \text{und} \quad d_t \geq \sum_{i=1}^{m} \left| w_i^{(2)} \right| \tag{2.105}$$

wobei die Gewichtsnorm d_t nur mit der Ordnung $d_t = O(N^{1/4})$ zunehmen darf. Dann gilt für unabhängige, identisch verteilte Trainingsmuster

$$d_t^4 m_t \log (d_t m_t) = O(N) \tag{2.106}$$

und für gemischte, stationäre Zufallsprozesse (nicht-zeitabhängige Verteilungen), beispielsweise durch mehrere Klassen, die Restriktion

$$d_t^2 m_t \log (d_t m_t) = O(N^{1/2}) \tag{2.107}$$

Die beiden Bedingungen stellen Grenzen für die Zuwachsrate der Zahl der *hidden units* (*Komplexität des Netzes*) dar, wenn das Netzwerk versucht, den quadratischen Fehler der Approximation zu minimieren. Interessanterweise ist die sinnvolle Zuwachsrate für die Zahl der *hidden units* für gemischte Prozesse geringer, da neue Information über den selben Prozess mit einer geringeren Rate eintrifft und deshalb weniger Gewichte zur Auswertung benötigt werden. Sind andererseits nur geringe Gewichte (geringe Zahl von Gewichtszuständen) möglich, so kann man dafür mehr Neuronen vorsehen: die maximal speicherbare Information ist die gleiche.

2.6 Lernalgorithmen in Multi-Layer-Perzeptrons

Bisher gingen wir davon aus, dass die für eine mehrschichtige Klassifikation nötigen Gewichte prinzipiell existieren. Mit allen mathematischen Zusicherungen über die prinzipiellen Möglichkeiten neuronaler Netze im Rücken stellt sich trotzdem die Frage: Wie findet man sie aber im konkreten Fall ?

2.6.1 Erstellen von Entscheidungsbäumen zur Klassifizierung

Liegen uns alle Trainingsmuster gleichzeitig vor und wir müssen nur dazu ein Klassifikationsnetz bauen, so bietet sich dafür auf sequentiellen Computern als einfachste und schnellste Lösung die *Klassifizierung mit Entscheidungsbäumen* [BFO84] an. Das Prin-

zip davon ist relativ einfach: jedes Klassengebiet (Trainingsmuster) lässt sich durch Entscheidungen bezüglich der Merkmale charakterisieren. Die Entscheidungen bilden einen Binärbaum, der direkt vom Computer oder einem Netz aus binären Neuronen evaluiert werden kann. In Abb. 2.36 ist ein solcher Baum für die Klassifikation der Muster von Abb. 2.23 gezeigt.

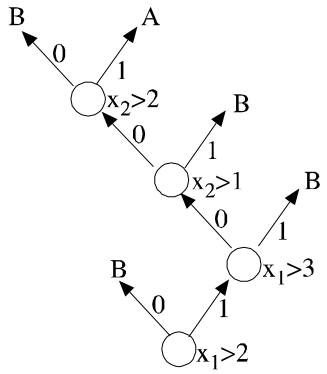


Abb. 2.36 *Binärbaum der Klassifikation aus Abb. 2.23*

Zum Aufbauen des Baums müssen nur die Koordinaten (Merkmale) der Trainingsmuster aufsteigend angeordnet werden und in Mustermengen entsprechend zusammengefasst werden, was relativ schnell geschehen kann; wesentlich schneller als bei adaptiven Algorithmen. Ein Algorithmus, der ein Netzwerk entsprechend dem Binärbaum erstellt, wurde von Frean [FRE90] vorgeschlagen.

2.6.2 Sequentielle Netzerstellung

Auch Klassifikationsnetze lassen sich schrittweise erstellen. Dazu schaffen z.B. wir für jedes Trainingsmuster, das falsch eingeordnet wird, eine neue Klasse. Sind zwei Klassen sehr dicht beieinander, so lassen sie sich auch miteinander verschmelzen. Ein Verfahren, das ein Verschieben, Erzeugen und Verschmelzen von Klassengrenzen bzw. Prototypen beschreibt und damit (in Abhängigkeit von entsprechenden Steuerungsparametern) ein Klassifikationsschema aus stückweise linearen Klassengrenzen bewirkt, wurde von Kong und Noetzel vorgestellt [KON90].

Eine andere Idee stammt von Marchand [MAR90]. Hier wird zuerst ein Neuron solange mit dem Perzeptron-Algorithmus trainiert, bis für alle Trainingsmuster, bei denen für eine bestimmte Klasse (z.B. Klasse 1) entschieden wird, dies auch richtig ist. Diese

richtig erkannten Trainingsmuster werden der Trainingsmenge entzogen. Es können aber noch weitere Muster der Klasse 1 existieren, die nicht erkannt wurden. Dazu wird ein weiteres Neuron hinzugefügt, das wieder solange trainiert wird, bis für alle Muster, für die auf Klasse 1 entschieden wird, dies auch korrekt ist. Auch diese Muster werden der Trainingsmenge entzogen. Das Hinzufügen neuer Neuronen und deren Trainieren wird solange fortgesetzt, bis alle Muster der Klasse 1 korrekt klassifiziert worden sind und keine weiteren mehr in der Trainingsmenge existieren. Die dann vorhandene Trainingsmenge enthält nur noch Muster der übrigen Klassen. Für Klasse 2 und darauf für die restlichen Klassen wird das gleiche Verfahren angewendet, bis die Trainingsmenge leer ist und somit alle Muster korrekt eingeordnet werden. Am Ende liegt ein Netzwerk aus einer Schicht vor, das mit einer zweiten bzw. dritten Schicht, jeweils nach dem gleichen Prinzip erstellt und trainiert wie die erste, eine Klassifikation durchführt.

2.6.3 Back-Propagation-Netzwerke

Beim Perzeptron und bei Adaline wurde mit Erfolg der Ausführungsfehler (die Differenz zwischen der gewünschten Ausgabe und der tatsächlichen Ausgabe des Netzwerks) dazu benutzt, die Gewichte der Netzwerkschicht für die gewünschte Funktion besser einzustellen. Diese Idee wird beim berühmten *error back-propagation* Algorithmus auf Mehrschichten-Architekturen (*multilayer feedforward*) ausgedehnt und das überwachte Lernen in allen Schichten durchgeführt. Diese Idee wurde zuerst von Rosenblatt in seinem Buch [ROS62] für MultilayerPerzeptrons skizziert und dort als *back-propagating error correction procedures* bezeichnet. Der Algorithmus stellt einen wichtigen Standardalgorithmus dar und wurde deshalb bisher als "das Arbeitspferd der neuronalen Netze" bezeichnet. Damit lernen wir einen Algorithmus kennen, den wir immer dann einsetzen können, wenn wir ein neuronales Netz mit Beispielen trainieren wollen, um eine unbekannte Funktion $\mathbf{y} = f(\mathbf{x})$ möglichst gut zu approximieren.

Betrachten wir nun den Algorithmus in der Form, wie er von Rumelhart und McClelland in ihrem bekannten Buch "Parallel Distributed Processing" [RUM88] formuliert wurde.

Die Funktionsarchitektur

Obwohl der Algorithmus beliebig viele Schichten gestattet, geht man üblicherweise nur

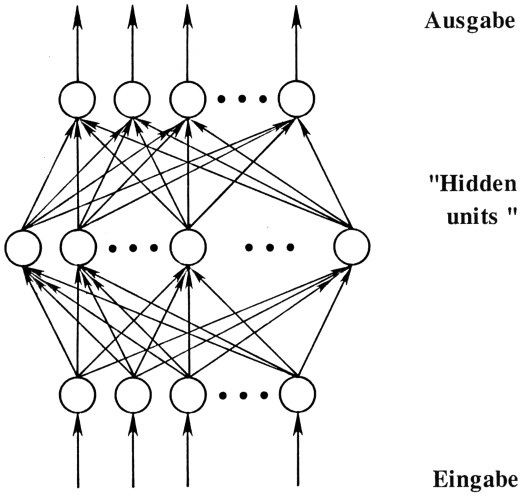


Abb. 2.37 *Grundstruktur des Backpropagation-Netzwerks*

von zwei Schichten aus, die aber, wie wir aus Abschnitt 2.5.5 wissen, für die Approximation einer beliebigen Funktion prinzipiell ausreichen. Die Eingänge beider Schichten sind, wie in Abb. 2.37 zu sehen, jeweils vollständig miteinander vernetzt.

Die Eingabeschicht (*input units*) als dritte Schicht besteht dabei allerdings nur aus Signalverteilerpunkten, so dass sie keine "echte", lernfähige Schicht darstellt. Da die Ausgänge der Einheiten der ersten Schicht nicht beobachtet werden können, werden sie auch als "verborgene" Einheiten oder *hidden units* bezeichnet.

Die Gewichte der hidden units und der Ausgabeeinheiten werden beim Backpropagation-Algorithmus durch Training mit Eingabemustern $\mathbf{x}$ und den dazu gewünschten Ausgabemustern $\mathbf{L}$ (Lehrervorgabe) solange verbessert, bis das Netzwerk die gewünschte Leistung (die gewünschte Ausgabe bei einer Eingabe) mit genügender Genauigkeit erbringt. Dazu wird nach dem Durchlaufen (*propagation*) des Eingabesignals $\mathbf{x}$ durch die Netzschichten der Fehler $\mathbf{y}(\mathbf{x}) - \mathbf{L}(\mathbf{x})$ des Ausgangssignals $\mathbf{y}$ bezüglich der gewünschten Ausgabe $\mathbf{L}$ ermittelt und durch alle Schichten zurückgeführt (*error back-propagation*), wie dies in Abb. 2.38 schematisch gezeigt ist.

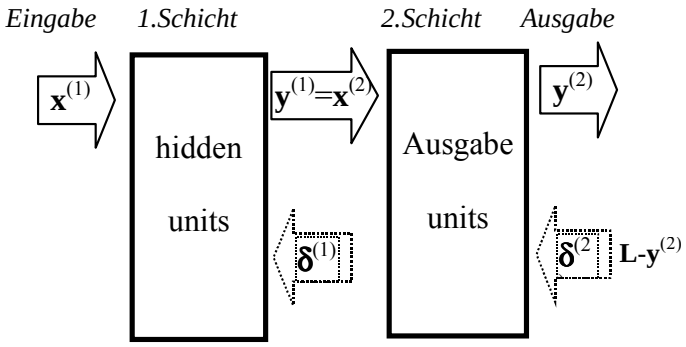


Abb. 2.38 *Fehlerrückführung im Mehrschichtensystem*

Die Lernregeln

Als Lernziel geben wir uns vor, dass der Algorithmus den erwarteten Fehler für alle Trainingsmuster minimieren soll. Die Zielfunktion R als Erwartungswert einer stochastischen, für ein Muster **x** gültigen Zielfunktion R_x lautet also mit einem Proportionalitätsfaktor 1/2

$$R = \langle R_x \rangle_x = ½ \langle (\mathbf{y}(\mathbf{x}) - \mathbf{L}(x))^2 \rangle_x \tag{2.108}$$

über alle Trainingsmuster **x**. Zur Verminderung der Zielfunktion wollen wir den einfachen stochastischen Gradientenabstieg aus Abschnitt 2.3.1 nutzen. Der Gradientenalgorithmus aus Gl. (2.54) für die Iteration des Gewichts w_{ij} von Neuron *j* zu Neuron *i* ist auch bei der stochastischen Approximation

$$w_{ij}(t) = w_{ij}(t\text{-}1) - \gamma(t)\, \partial R_x / \partial w_{ij} \tag{2.109}$$

mit der Lernrate $\gamma(t)$. Präsentieren wir die einzelnen Trainingsmuster hinter einander, so ist die Gewichtsänderung nach jedem einzelnen Trainingsmuster **x**

$$\Delta w_{ij}(x) := w_{ij}(t) - w_{ij}(t\text{-}1) = -\gamma \frac{\partial R_x}{\partial w_{ij}} = -\gamma \frac{\partial R_x}{\partial z_i} \frac{\partial z_i}{\partial w_{ij}} \tag{2.110}$$

Die Beeinflussung des Fehlers durch die Änderung der Gewichte im System lässt sich also aufspalten in einen Anteil, der auf die Aktivität zurück geht und einen, der die Abhängigkeit der Aktivität von den Gewichten widerspiegelt. Mit der Notation für die partielle Ableitung der Ausgabe

$$\frac{\partial y_i}{\partial z_i} = \frac{\partial S(z_i)}{\partial z_i} =: S'(z_i) \tag{2.111}$$

ist $$\delta_i := \frac{-\partial R_x}{\partial z_i} = \frac{-\partial R_x}{\partial y_i}\frac{\partial y_i}{\partial z_i} = \frac{-\partial R_x}{\partial y_i} S'(z_i), \quad \frac{\partial z_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k w_{ik} x_k = x_j \qquad (2.112)$$

Die Gewichtsänderung kann man mit den obigen Abkürzungen sehr einfach schreiben:

$$\Delta w_{ij}(x) = \gamma \delta_i x_j \qquad \textit{Delta-Regel} \qquad (2.113)$$

Nun wollen wir diese Lernregel für die beiden Schichten anwenden und konkret formulieren. Dazu müssen wir das oben definierte δ_i evaluieren. Für die Neuronen der zweiten Schicht, deren Ausgabe wir beobachten können, gilt der Teilausdruck

$$\frac{\partial R_x}{\partial y_i} = \frac{\partial}{\partial y_i} \tfrac{1}{2}(y_i - L_i)^2 = (y_i - L_i) \qquad (2.114)$$

für den beobachteten Fehler.

Bezeichnen wir zur Unterscheidung die Variablen der Neuronen der ersten Schicht mit dem Index $^{(1)}$ und die der zweiten Schicht mit $^{(2)}$, so ist für die Ausgabeschicht mit der Definition aus Gl. (2.112)

$$\delta_k^{(2)} = -(y_k^{(2)} - L_k^{(2)})\, S'(z_k^{(2)}) \qquad (2.115)$$

Für die anderen Schichten, beispielsweise für die erste Schicht ("hidden units"), gilt ein komplizierteres Delta, da dann auch die Auswirkungen der Fehler der ersten Schicht auf die Ausgabe der zweiten Schicht berücksichtigt werden müssen. Betrachten wir dazu ein Neuron *i*, das in der folgenden Schicht andere Neuronen *k* beeinflusst, siehe Abb. 2.39.

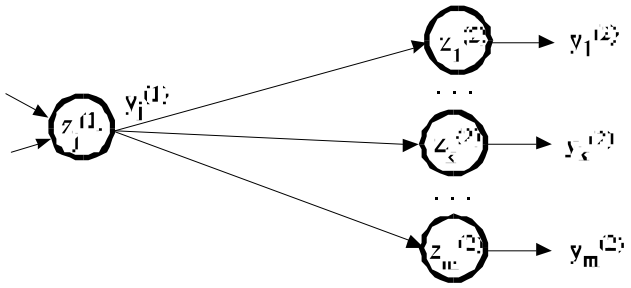


Abb. 2.39 *Propagierung der Aktivität von Neuron* i *zu* m *Neuronen* k

Es lässt sich der beobachtete Fehler im *i*-ten Neuron der ersten Schicht rekursiv durch die Folgefehler der zweiten Schicht darstellen. Dazu wird der Fehler aufgespaltet in die einzelnen Komponenten

$$R_x = |\,\mathbf{y}(\mathbf{x}) - \mathbf{L}(\mathbf{x})\,|^2 = \sum_{k=1}^{m} (y_k(\mathbf{x}) - L_k(\mathbf{x}))^2 = \sum_{k=1}^{m} R_x(k) \tag{2.116}$$

und der Einfluss der ersten Schicht auf die Aktivität isoliert

$$\frac{\partial R_x}{\partial y_i^{(1)}} = \sum_k \frac{\partial R_x(k)}{\partial z_k^{(2)}} \frac{\partial z^{(2)}}{\partial y_i^{(1)}} \tag{2.117}$$

Durch die Definition von Gl. (2.112) und den Teilausdruck

$$\frac{\partial z_k^{(2)}}{\partial y_i^{(1)}} = \frac{\partial}{\partial y_i^{(1)}} \sum_j w_{kj}^{(2)} x_j^{(2)} = w_{ki}^{(2)} \tag{2.118}$$

erhalten wir beim Einsetzen von Gl. (2.112), (2.118) und (2.111) in Gl. (2.117) den Ausdruck für das Delta der ersten Schicht:

$$\delta_i^{(1)} = \frac{-\partial R_x}{\partial y_i^{(1)}} \frac{\partial y_i^{(1)}}{\partial z_i^{(1)}} = \left(\sum_{k=1}^{m} \delta_k^{(2)} w_{ki}^{(2)} \right) S'\left(z_i^{(1)}\right) \tag{2.119}$$

Setzen wir nun noch sigmoidale Fermi-Ausgabefunktion S_F aus Kapitel 1 voraus mit

$$S'(z) = \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{+1-1+e^{-z}}{\left(1+e^{-z}\right)^2} = \left(1 - S(z)\right) S(z) \tag{2.120}$$

als Ableitung, so sind die Fehlerinkremente $\delta_i^{(2)}$ und $\delta_i^{(1)}$ bestimmt und wir haben damit die zwei Delta-Lernregeln, eine für die Ausgabeschicht

$$\Delta w_{ij}^{(2)}(x) = \gamma\, \delta_i^{(2)} x_j^{(2)} = -\gamma\, (y_i^{(2)} - L_i)\, (1 - S(z_i^{(2)}))\, S(z_i^{(2)})\, x_j^{(2)} \tag{2.121}$$

und eine für die Schicht der "hidden units"

$$\Delta w_{ij}^{(1)}(x) = \gamma\, (\textstyle\sum_k \delta_k^{(2)} w_{ki}^{(2)}) (1 - S(z_i^{(1)}))\, S(z_i^{(1)})\, x_j^{(1)} \tag{2.122}$$

vollständig hergeleitet.

Für die allgemeine Rekursionsformel des Fehlerinkrements für eine beliebige Zahl von Schichten reicht es, in (2.119) die Ersetzung der $^{(2)}$ durch $^{(n)}$ und der $^{(1)}$ durch $^{(n-1)}$ vorzunehmen, so dass

$$\delta_i^{(n-1)} = \left(\textstyle\sum_k \delta_k^{(n)} w_{ki}^{(n)}\right) S'(z_i^{(n-1)}) \tag{2.123}$$

resultiert.

Mit den Lernregeln Gl.(2.121) und (2.122) errechnet der Algorithmus für jedes Trainingsmuster die notwendige Korrektur der Gewichte $\Delta\mathbf{w}$ und speichert sie zunächst nur als Summe. Erst nach dem letzten Trainingsmuster erfolgt dann tatsächlich die Korrektur der Gewichte mit

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w} \tag{2.124}$$

Da die einzelnen Korrekturbeiträge auf der Basis der alten Gewichte ohne Kenntnis der bereits errechneten Korrekturen errechnet wurden, wird dies als *OFF-Line*-Version bezeichnet. Im Codebeispiel 2.6 ist die Grundstruktur eines OFF-Line Backpropagation-Algorithmus zum einmaligen Training von *n* Eingabeeinheiten, *p hidden units* und *m* Ausgabeeinheiten gezeigt, wobei in dem Beispiel die Trainingsmuster in einer Datei `Pattern File` fertig vorliegen. Im Code wird für die Fehlerprozedur der Gl. (2.119) die Abkürzung `SumProd(i,m,`$\delta^{(2)}$`,`$w^{(2)}$`)`$:= \sum_{k=1}^{m} \delta_k^{(2)} w_{ki}^{(2)}$ verwendet. Voraussetzung für das Training ist wieder eine Sammlung von Beispielen `PatternFile`, bei denen für jedes Eingabemuster **x** die gewünschte Ausgabe **L** vorliegt.

BackProp: *(* Lernen im Backpopagation-Netzwerk *)*

```
x1: ARRAY[1..n] OF REAL; x2: ARRAY[1..p] OF REAL;
y2,L,d2:ARRAY[1..m] OF REAL;                    (* y_i^(2),L_i,δ_k^(2) *)
w1,dw1: ARRAY[1..p] OF ARRAY[1..n] OF REAL;     (* w_ij^(1), Δw_ij^(1)*)
w2,dw2: ARRAY[1..m] OF ARRAY[1..p] OF REAL;     (* w_ij^(2), Δw_ij^(2)*)
REPEAT (* jeweils einen Trainingszyklus *)
dw1:=0.0; dw2:=0.0; γ:=0.5;
 REPEAT   (* Für alle Trainingsmuster im PatternFile *)
     Read(PatternFile,x1,L)(* Einlesen der Eingabe und Ausgabe *)
 (* Ausgabe errechnen 1. und 2. Schicht *)
     FOR i:=1 TO p DO        (* Für alle hidden units der 1.Schicht *)
         x2[i]:= S(z(w1[i],x1))
      END
      FOR i:=1 TO m DO     (* Für alle Ausgabeneuronen: Ausgabe+Fehler*)
          y2[i]:= S(z(w2[i],x2))
          d2[i]:= -(y2[i]-L[i])*(1-y2[i])*y2[i]          (* Gl.(2.115) *)
      END

(* Gewichtsveränderungen in 2. Schicht *)
  FOR i:=1 TO m DO        (* Für alle Ausgabeneuronen *)
    FOR j:=1 TO p DO      (* Für alle Eingabemusterkomp. 2. Schicht *)
      dw2[i,j]:= dw2[i,j]+γ*d2[i]*x2[j]          (* Nach Gl.(2.121)*)
    END;
  END
```

```
   (* Gewichtsveränderungen in 1. Schicht *)
    FOR i:=1 TO p DO        (* Für alle hidden units der 1.Schicht *)
      FOR j:=1 TO n DO      (* Für alle Eingabemusterkomp. 1. Schicht *)
        dw1[i,j]:=dw1[i,j] +γ*SumProd(i,m,d2,w2)*(1-x2[i])*x2[i]*x1[j]
      END;
    END
  UNTIL   ( EOF( PatternFile))
  w1:=w1+dw1;               (* Korrektur der Gewichte *)
  w2:=w2+dw2;
UNTIL   Fehler_klein_genug
```

Codebeispiel 2.6 *OFF-Line Training eines Backpropagation-Netzwerks*

Wollen wir im Unterschied dazu die Gewichte sofort nach jedem Trainingsmuster korrigieren (*ON-Line*-Version), so können wir statt Gl.(2.109) die stochastische Version verwenden

$$w_{ij}(t) = w_{ij}(t\text{-}1) - \gamma \, \partial R_x / \partial w_{ij} \tag{2.125}$$

Von der stochastischen Approximation wird uns in Abschnitt 2.3.2 eine gute Konvergenz garantiert, wenn wir anstelle einer konstanten Lernrate γ = const eine den Bedingungen (2.65) gehorchende Lernrate, beispielsweise $\gamma(t) = 1/t$, einsetzen. Allerdings gilt dies nur, wenn δ eine stochastische Variable mit zeitunabhängiger Verteilung ist. Das ist aber bei uns nicht gegeben: durch die Gewichtskorrektur verschiebt sich trotz konstanter Verteilung der **x** die Verteilung der y und damit die des Fehlers δ. Es ist also deshalb sinnvoll, beispielsweise anfangs eine konstante Lernrate zu verwenden, bis keine großen Änderungen mehr stattfinden, und dann auf $\gamma(t) = 1/t$ überzugehen, um die Gewichtsänderungen zu null konvergieren zu lassen.

2.6.4 Anwendung: NETtalk

Eines der wichtigsten ersten Beispiele für die Anwendung des Backpropagation Algorithmus, das die Möglichkeiten und Probleme des Verfahrens zeigte, war das NETtalk Projekt, über das Sejnowski und Rosenberg 1986 [SEJ86] berichteten.

Die Aufgabe des Systems bestand darin, vorhandenen ASCII-Text in Lautschrift (Phoneme) umzuwandeln, die über einen Sprachchip und einer nachgeschalteten Verstärker-Lautsprecher Kombination als verständliche Sprache hörbar sein sollte. Vorläufer des Systems war ein Expertensystem der Firma Digital Equipment (DECtalk), das mit einem Aufwand von 20 Personen-Jahren 95% Genauigkeit erreichte. Demgegenüber konnte das NETtalk-System in nur 16 CPU-Stunden so trainiert werden, dass es eine Genauigkeit von 98% erreichte.

Wie erreichte die Architektur des Systems so schnell eine so gute Genauigkeit ? Betrachten wir dazu die Architektur des Systems genauer. Das NETtalk System besteht aus einem normalen, zweischichtigen Backpropagation Netzwerk mit zusätzlichen Eingabeeinheiten ("dreischichtiges Netzwerk"). Die Anzahl der Eingabe- und Ausgabeeinheiten war dabei durch die Kodierung gegeben: Jeder der Textbuchstaben, der eingegeben wird, kann eines der 26 Buchstaben des Alphabets oder eines der drei Zeichen für Punktuation und Wortgrenzen sein, so dass bei einer binären Kodierung 29 binäre Eingänge ("Eingabeunits") pro Buchstaben benötigt werden.

Da jeder Buchstabe im Kontext der vor- und nachstehenden Buchstaben gesehen werden muss, werden die drei vorhergehenden und die drei nachfolgenden Buchstaben zusätzlich dem System als Eingabe präsentiert. Diese insgesamt sieben Buchstaben mit je 29 binären Eingabekomponenten bilden also $7 \cdot 29 = 203$ Eingabedaten, die mit 80 Neuroneneinheiten in der "versteckten Schicht" ("hidden units") bearbeitet werden und eine Ausgabe auf den 26 binären Ausgabeeinheiten bewirken, die 23 Laut- und drei Artikulationsmerkmale (Continuation, Wortgrenze, Stop) kodieren. Das gesamte Netzwerk besitzt somit über 18.000 Gewichte.

In Abb. 2.40 ist ein Funktionsschema des Netzwerks gezeigt.

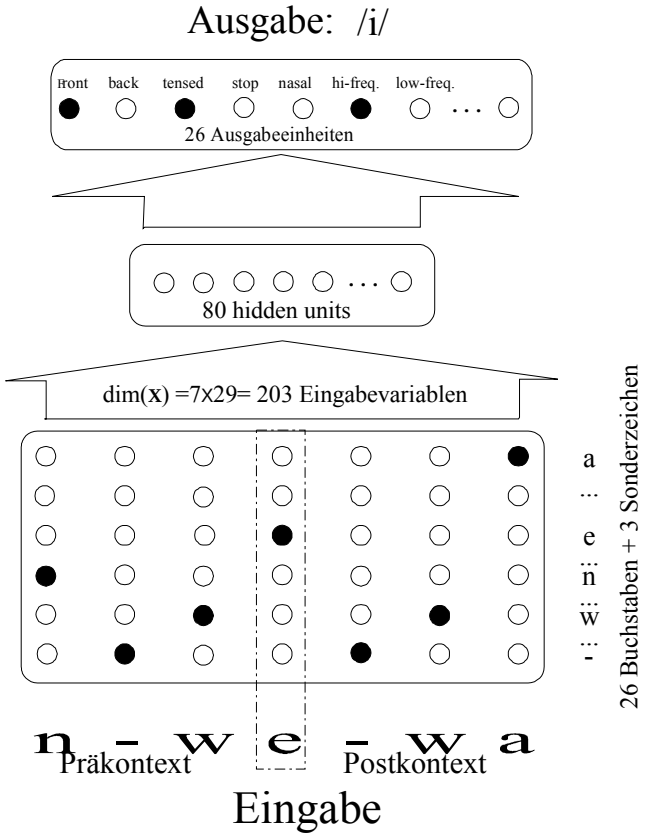


Abb. 2.40 *Kodierung im NETtalk Netzwerk*

Dabei sind die Eingabeneuronen zum besseren Überblick in Spalten angeordnet. Jede Spalte mit 29 binären Eingabeeinheiten kodiert einen der 7 Textbuchstaben, wobei der eigentlich betrachtete Buchstabe genau in der Mitte steht. Bei der Texteingabe wird der Text ("..then we wanted..") fortlaufend, wie bei einer Laufschrift, durch das "Fenster" von 7 Buchstaben geschoben. Nach jedem Verschiebeschritt werden die Gewichte neu bestimmt und die Abweichung gespeichert.

Der Lernalgorithmus bedient sich einer geringfügig modifizierten Version des Backpropagation-Algorithmus (2.121)

$$w_{ij}(t)^{(n)} = \alpha\, w_{ij}(t\text{-}1)^{(n)} - (1\text{-}\alpha)\gamma\, \delta_i^{(n+1)}\, S(z_j)^{(n-1)} \qquad (2.126)$$

mit der konstanten Lernrate $\gamma \approx 2{,}0$ und dem empirischen Parameter $\alpha \approx 0{,}9$, der den Gradientenzuwachs harmonisieren soll.

Die Trainingsmenge bestand aus zwei Datensorten: zum einen aus protokollierten, von Kindern gesprochenen Sätzen, zum anderen aus zufallsmäßig eingegeben Worten eines Wörterbuchs mit 20.000 Einträgen. Bei beiden Textmaterialien war die phonologische Transkription verfügbar und diente als Lehrervorgabe **L**(**x**), mit der die Ausgabe verglichen wurde. Dabei wurde in der phonetischen Umschreibung ein besonderes Zeichen (Continuation) immer dann eingefügt, wenn ein Buchstabe nicht gesprochen wurde.

Der Erfolg des Trainings kann an dem Prozentsatz der richtig ermittelten Betonungen ("Stresses") und Lauten ("Phoneme") in Abb. 2.41 abgelesen werden.

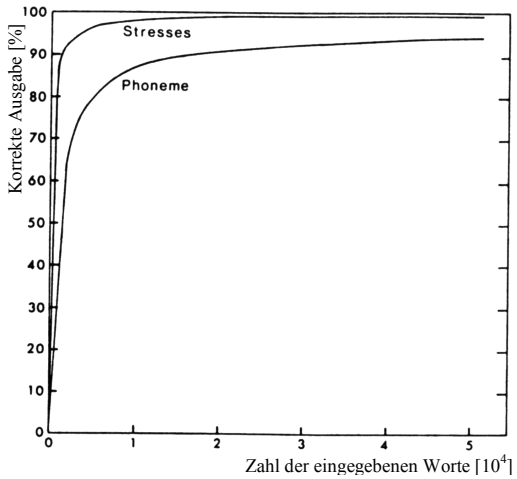


Abb. 2.41 *Lernen der phonetischen Übersetzung (nach [SEJ86])*

Beim Abhören der Laute, die mit den Ausgabedaten des NETtalk Netzes von dem Sprachsystem erzeugt wurden, konnten die Forscher drei Phasen der Sprachentwicklung unterscheiden, die man auch von der kindlichen Entwicklung her kennt:

- Zuerst wurden die Konsonanten und Vokale als Klassen getrennt. Innerhalb der Klassen blieben die Phoneme aber noch gemischt, so dass sich die Laute wie "Babbeln" anhörten.
- Dann wurden die Wortgrenzen als Merkmale entwickelt, so dass "Pseudoworte" erkennbar wurden.
- Zuletzt, nach ca. 10 Durchgängen pro Wort, entstand eine verständliche Sprache, die sich mit fortlaufender Erfahrung weiter verbesserte.

Die Konstellation der Gewichte ist dabei durchaus kritisch: werden alle zufallsmäßig auch nur wenig verändert, so sinkt die Erfolgsquote stark. Mit erneutem Training und erneutem Lernen "erholt" sich aber das Netz schnell wieder, da ja alle Gewichte vorher bereits fast optimale Werte hatten.

2.6.5 Analyse der Funktion der "hidden units"

In der Abbildung 4.2.6 sind die Gewichte von zwei verschiedenen Neuronen der "hidden units" visualisiert (*Hinton Diagramm*).

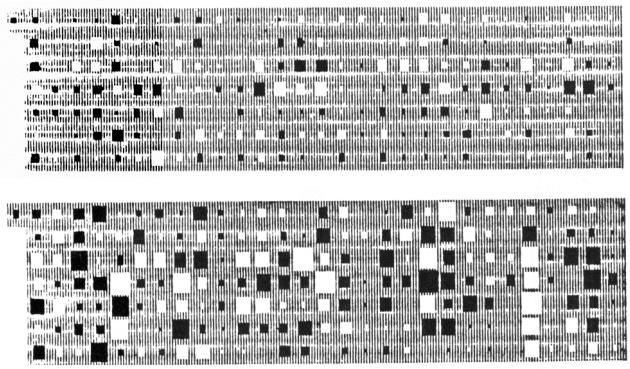


Abb. 2.42 *Visualisierung der Gewichte (nach [SEJ86])*

Dabei stehen weiße Vierecke für positive und schwarze Vierecke für negative Gewichte; die Flächengröße jedes Vierecks ist dem Wert des Gewichts proportional. Das erste Gewicht links oben ist dabei das Verschiebungs-Gewicht (*bias*), also der Schwellwert. Bei der Betrachtung der Abbildung stellen sich einige Fragen: Was bedeuten die Gewichte und wie lässt sich ihre Verteilung interpretieren? Welche Aufgaben übernehmen die "hidden units", wohin konvergieren die Gewichte und wie groß ist die optimale Zahl dieser Einheiten?

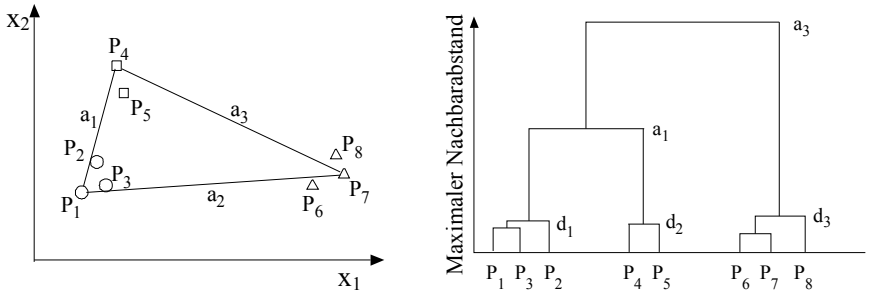
Die Beantwortung dieser Fragen ist auch bis jetzt noch nicht vollständig aufgeklärt, aber seit Erscheinen der Arbeit ist doch vieles klarer geworden. Zweifelsohne scheinen die formalen Neuronen mit den Gewichten eine interne Repräsentation der Eingabedaten aufzubauen, also eine Art Umkodierung und Datenkompression von vielen (203) Eingabedaten auf wenige (80) Ausgabedaten der *hidden units*. Aber um was für eine Art der Repräsentation handelt es sich dabei ?

Man kann die Gewichte auf verschiedene Weise untersuchen. Statistisch gesehen bilden die Gewichte von NETtalk eine Normalverteilung mit Mittelwert null [HAN90]. Dies zeigt, dass die eigentliche Information in der Struktur der Gewichtsverteilung des einzelnen Neurons liegt, nicht in der Gesamtheit.

Clusteranalyse

Eine genauere Analyse kann versuchen, "Typen" von Neuronen zu finden. Ein gängiger Ansatz dafür fasst alle Muster, die das betrachtete Neuron stärker als eine vorgegebene Schwelle aktivieren, zu einer für dieses Neuron spezifischen Mustermenge zusammen. Wird jedes Muster als Punkt im n-dimensionalen Eingaberaum aufgefasst, so bildet die besagte Mustermenge einen Haufen (*cluster*). Die Analyse der Neuronengewichte wird also zu einer *Clusteranalyse* des Eingaberaums. Fasst man mehrere, benachbarte Cluster zu einem neuen Cluster zusammen, z.B. wenn ein Neuron für die Muster aus mehreren Clustern empfindlich ist, so ergibt sich eine *Clusterhierarchie*.

Für den Aufbau einer Clusterhierarchie ist die Kenntnis von zwei Ähnlichkeits- bzw. Abstandsmaßen nötig: zum einen das Ähnlichkeitsmaß *(Abstand)* zwischen zwei Punkten, zum anderen das Entscheidungskriterium für eine Gruppenzugehörigkeit. Das erstere ist beispielsweise der Euklidsche Abstand, der Hamming-Abstand, der Manhattan (*city block*) Abstand usw. Für das zweite wird meist der Abstand zum Clustermittelpunkt (Erwartungswert aller Muster eines Clusters), zum am weitesten entfernten Clustermitglied, oder zum nächst gelegensten Clustermitglied verwendet. In Abb. 2.43 ist ein solcher Clusterprozess verdeutlicht.



a) Punkte im Musterraum **b)** Hierarchie

Abb. 2.43 *Aufbau einer Clusterhierarchie*

In Abb. 2.43a sind im 2-dim Musterraum 9 Muster (Punkte $P_1..P_9$) zu sehen. Misst man den Abstand zwischen den Punkten, so ergeben sich drei Cluster mit dem jeweiligen Durchmesser (Abstand zwischen den entferntesten Mitgliedern) d_1, d_2 und d_3. Suchen

wir nun zu jedem Cluster dasjenige andere Cluster, welches bei einer Verschmelzung den kleinsten max. Mitgliederabstand im neuen Cluster bringen würde, so ergibt sich eine Reihenfolge für die Verschmelzungen der einzelnen Cluster miteinander, die durch das Anwachsen des max. Nachbarabstandes im Cluster (*Clusterdurchmessers*) gekennzeichnet ist. Dies ist in Abbildung Abb. 2.43b) dargestellt. Hier ist ein Verschmelzen durch eine horizontale Linie zwischen den Punkten dargestellt; eine senkrechte Linie verdeutlicht das Anwachsen des Clusterdurchmessers bis zum nächsten Verschmelzungspunkt.

Diese Clusterung im Eingaberaum wurde von Hanson und Burr [HAN90] als "in vivo"-Clusterung bezeichnet. Im Kontrast dazu schlugen sie ein "in vitro"-Verfahren vor, das im Raum der *hidden-unit* Aktivierungsausgaben ($S_{1,}.., S_m$) arbeitet. Hierbei wird eine Standardeingabe (z.B. (1,..,1)) auf alle Neuronen gegeben und der Aktivitätsunterschied als Abstand zwischen den Neuronen aufgefasst. Die Neuronen mit geringem Abstand (Aktivierungsunterschied) lassen sich hier als "Funktionsgruppen" auffassen. Allerdings ist dies nur zutreffend, wenn eine Lokalisierung von Funktionen und damit eine Zuordnung von Neuronen zu bestimmten Funktionen vorliegt. Mit der Eingabe von äquidistanten Eingabemustern lässt sich so der Eingaberaum "abrastern" und die sensitiven Bereiche charakterisieren.

Sensitivitätsanalyse

Welche der Eingabevariablen sind ausschlaggebend für das Ergebnis und damit besonders wichtig? Dieser Frage kann man versuchen, über eine Analyse der Abhängigkeit des Ausgabefehlers (Approximationsfehlers) von den Eingabevariablen nachzugehen. Dazu wird für alle Testmuster die Variable weggelassen (das Eingabegewicht null gesetzt) und der zusätzliche, mittlere Fehler ermittelt. Dies führt zu einer absteigenden Liste des jeweiligen zusätzlichen Fehlers, die auch gleichzeitig die Prioritätenliste der Eingabevariablen darstellt.
Allerdings ist dieses Verfahren von der Skalierung der Eingabevariablen abhängig, so dass sich hier vorher eine Normierung der Eingabevariablen (Korrektur des Mittelwerts und der Varianz) als Vorstufe des Netzwerks anbietet. Ein zusätzliches, ungelöstes Problem tritt auf, wenn die unnötigen, zufallsbedingten Gewichte (der unnötigen Eingabevariablen) zu groß sind, um beim Weglassen den Ausgabefehler unverändert zu lassen. In diesem Fall versagt das Verfahren.

Hauptkomponentenanalyse

Mehr Licht in diese Zusammenhänge brachte die Arbeit von Baldi und Hornik 1989 [BAL89]. Betrachten wir ein Netzwerk, das aus zwei Schichten (*n* Eingabeunits, *p* hidden units, *n* Ausgabeunits) besteht. Wird versucht, das Netzwerk so zu trainieren, dass die selben Eingabemuster auch bei der Ausgabe reproduziert werden, so bilden die we-

nigen *p* Einheiten einen "Flaschenhals" für den Informationsstrom durch das Netz; das Netz wird "gezwungen", durch die *hidden units* eine kompakte interne Repräsentation der Eingabe aufzubauen. Ein solches Netzwerk wirkt wie ein *Kodierer* und bewirkt eine Datenkompression.
Angenommen, wir betrachten nur den linearen Teil der Ausgabefunktionen. Dann besteht das Netz aus zwei aufeinanderfolgenden, linearen Abbildungen mit den Matrizen **A** und **B**. Die resultierende lineare Transformation $\mathbf{y} = \mathbf{ABx} = \mathbf{Dx}$ der Eingabemuster hat das Ziel, den kleinsten quadratischen Fehler bei der Kodierung und Dekodierung zu erzielen. Die Matrizen **A** und **B** sind dabei nicht eindeutig bestimmt, da ja mit $\mathbf{D} = \mathbf{AB} = (\mathbf{AU})(\mathbf{U}^{-1}\mathbf{B}) = \mathbf{A'B'}$ auch andere Matrizen $\mathbf{A'} = \mathbf{AU}$ und $\mathbf{B'} = \mathbf{U}^{-1}\mathbf{B}$ die Leistung erfüllen.

Für ein solches System zeigten Baldi und Hornik [BAL89] die folgenden Eigenschaften:

- Die Fehlerfunktion hat ein *einziges*, lokales und globales Minimum (s. Abb. 2.44)
- Dieses Minimum wird angenommen, wenn die *p* Gewichtsvektoren von **B** den Raum der *p Eigenvektoren* der Kovarianzmatrix $\mathbf{C}_{xx}$ der *N* Trainingsmuster

$$\mathbf{C}_{xx} = \left\langle (\mathbf{x}-\overline{\mathbf{x}})(\mathbf{x}-\overline{\mathbf{x}})^T \right\rangle = (C_{ij}) = \left(\frac{1}{N} \sum_{k=1}^{N} \left(x_i^k - \left\langle x_i^k \right\rangle \right) \left(x_j^k - \left\langle x_j^k \right\rangle \right) \right) \qquad (2.127)$$

  aufspannen, wobei dies die Eigenvektoren mit den größten Eigenwerten sind.

Der erste Punkt beruht im Wesentlichen auf der angenommenen Linearität der Ausgabefunktion; bei den tatsächlich vorhandenen Nichtlinearitäten müssen wir dagegen immer mit lokalen Minima rechnen, in denen das System "stecken" bleiben kann. Der zweite Punkt ist ein klassisches Ergebnis, siehe.

Da die Eigenschaft der Minimierung des Fehlerquadrats im linearen Teil der sigmoidalen Ausgabefunktion durch die Transformation auf Eigenvektoren (vgl. auch Kapitel 3) näherungsweise gilt, können wir stark vermuten, dass die erste Schicht des nicht- linearen Backpropagation-Netzwerks für kleine Eingabewerte ähnlich einer Transformation der Eingabedaten auf das "natürliche", durch ihre Statistik bestimmtes Koordinatensystem der Eigenvektoren wirkt.

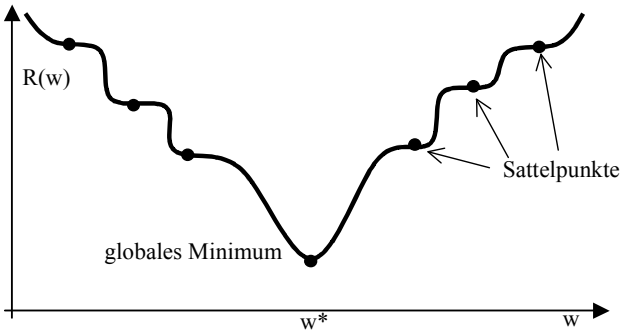


Abb. 2.44 *Das Minimum des quadratischen Fehlers*

Die Konvergenz verlangsamt sich an den "kritischen Punkten" (Sattelpunkten), an denen für die Fehleränderungen $\partial R/\partial a_{ij} = \partial R/\partial b_{ij} = 0$ gilt, s. Abb. 2.44.
Damit gilt: Die Zahl der Eigenvektoren und damit die Zahl der "hidden units" ist immer kleiner oder gleich der Zahl der Eingabeunits, da die Zahl der Eingabevariablen (Dimensionen der Gewichtsvektoren) die Maximalzahl der Basisvektoren des Eingaberaums und damit auch der Eigenvektoren als neue Basis bedeutet. Weniger "hidden units" beschränkt also die Zahl der möglichen Eigenvektoren und damit die Genauigkeit der Approximation. Allerdings nur dann, wenn ihre Zahl kleiner als der Rang der Kovarianzmatrix $\mathbf{C}_{xx}$ ist. Die mögliche Einschränkung wird durch die Datenkompression als "Generalisierung" oder "Abstraktion" empfunden. Werden andererseits mehr "hidden units" als nötig installiert, so konvergieren auch sie zu Vektoren im Raum der *p* Eigenvektoren und tragen nichts wesentliches bei.
Die Nichtlinearität der Ausgabefunktionen kodiert zusätzlich die nichtlinearen Anteile der gewünschten Funktion.

2.6.6 Heuristische Verbesserungen des Algorithmus

Der ursprüngliche, nicht-lineare Backpropagation-Algorithmus weist, wie bereits erwähnt, verschiedene Nachteile auf:

- Die Konvergenz ist relativ *langsam*, besonders bei mehreren Schichten
- Das System kann in einem *lokalen* Optimum "stecken" bleiben
- Trotz guter Trainingsleistung zeigt der Test *schlechte Ergebnisse*

Es gibt viele Arbeiten in der Literatur für jeden dieser Punkte, um mit einem geänderten Algorithmus die Ergebnisse zu verbessern. Im Folgenden soll aus der Menge dieser Arbeiten kurz verschiedene Lösungsmöglichkeiten der Probleme angedeutet werden.

Bessere Konvergenz durch geeignete Initialisierung der Gewichte

Wären die Ausgabefunktionen der ersten Schicht linear, so könnte man beide Schichten zu einer einzigen zusammenfassen; die Gewichte könnten dabei leicht nach dem stochastischen Lernalgorithmus so gelernt werden, dass der quadratische Fehler wie bei Adaline mit Hilfe der Widrow-Hoff-Regel in Gl.(2.49) minimiert würde. Wie im vorigen Abschnitt 2.5.5 ausgeführt wurde, ist aber eine nicht-lineare Ausgabefunktion für die Eigenschaft des neuronalen Netzes als universeller Approximator unerlässlich. Dies bedeutet, dass es beim Backpropagation-Algorithmus im Unterschied zum Minimum des quadratischen Fehlers bei linearer Ausgabefunktion nicht nur ein, sondern mehrere lokale Minima des Fehlerquadrats geben kann. Um zu verhindern, dass der Gradientensuchalgorithmus in einem lokalen, aber nicht globalen Minimum der Zielfunktion "stecken" bleibt, müssen die initialen Gewichte und die Lernraten beider Schichten gut aufeinander abgestimmt werden.

Ein guter Ansatz besteht darin, bei der Initialisierung der Gewichte alles Wissen zu verwenden, das verfügbar ist. Besitzen wir alle Trainingsmuster, so kann man mit den Eigenvektoren ihrer Kovarianzmatrix als erste Näherung die Gewichte initialisieren.

Schnellere Konvergenz durch heuristische Anpassungen

Aber auch kleinere Änderungen ermöglichen eine Konvergenzbeschleunigung. Beispielsweise bringt die Wahl der Lernrate γ als Hauptdiagonalmatrix (für jedes Gewicht eine eigene Lernrate, s. [JAC88], [SIL90]) und Änderung der Rate erst bei Vorzeichenänderung (vgl. Abschnitt 1.4.1) schon eine Beschleunigung von ca. 100.

Es sind noch viele weitere, heuristische Änderungen am Backpropagation Algorithmus möglich. Eine Literaturübersicht darüber und eine Bewertung für ein gegebenes Problem ist in [SCHI92] gegeben. Interessanterweise erwies sich dabei von allen Modifikationen der Quickprop- Algorithmus von Fahlman [FAH88] am effizientesten. Dieser sieht im Wesentlichen folgendes vor:

- Addiere 0.1 zum Faktor $S'(z) = (1-S)S$ in (2.120) .

  Damit wird der Faktor bei 0 oder 1 nicht zu klein.

- Ist der Faktor <0.1, so wird er bei den Ausgabeneuronen auf Null gesetzt und verhindert so zu kleine, zufällige Schwankungen

- Der so veränderte Gradient wird durch einen Abklingterm $\alpha w_{ij}(t-1)$ und eine Schrittinterpolation $\beta(t)\Delta w_{ij}(t-1)$ ergänzt:

$$\Delta w_{ij}(t) = -\gamma(t)R_x(t)/\partial w_{ij} - \gamma(t)\alpha w_{ij}(t-1) + \beta(t)\Delta w_{ij}(t-1) \qquad (2.128)$$

wobei der Faktor $\beta(t)$ sehr speziell ermittelt wird.

Die heuristischen Algorithmen haben dabei das Problem, dass sie meist viele problemabhängige Konstanten besitzen, die erst mühsam ermittelt werden müssen, um eine befriedigende Leistung zu erhalten. Besser sind deshalb Ansätze, die grundlegende, für alle Problemarten gültige Betrachtungen berücksichtigen. Beispielsweise fällt bei der Klassifikation mit dem Kriterium der Information (s. unten) anstelle des mittleren quadratischen Fehlers der Term *(1-S)S* aus Gl. (2.120) in Gl.(2.115) automatisch weg und muss nicht mehr berücksichtigt werden.

Bei vielen Schichten beruht die langsame Konvergenz im Wesentlichen auf der Tatsache, dass bei dem Verfahren die Gewichte nicht durch direkte Beobachtungen (Fehler etc.) gelernt werden, sondern indirekt über die Beobachtung des Fehlers durch mehrere Schichten hindurch. Eine wichtige Möglichkeit zur Konvergenzbeschleunigung besteht deshalb darin, die Adaption (falls möglich) für jede Schicht unabhängig von den anderen durchzuführen. Für manche Probleme kann es vernünftig sein, die Schicht der hidden units durch eine Schicht zu ersetzen, die eigenständig eine Kodierung lernt. Beispielsweise kann man auf die Rückführung des Fehlers verzichten und stattdessen direkt mit einem PCA-Netz (s. Kapitel 3) die Eigenvektoren der Kovarianzmatrix der Eingabedaten lernen lassen oder direkt bei wenigen, bekannten Trainingsmustern numerisch bestimmen. Die zweite Schicht lernt dann die passende Kopplung der Eingabestatistik (die Koeffizienten der Linearkombination der Eigenvektoren für ein gewünschtes Ergebnis) an die Lehrervorgaben mit Hilfe der Widrow-Hoff Lernregel Gl.(2.49) wesentlich (sogar exponentiell, s. [BAL88] !) schneller.

Erreichen des globalen Optimums

Ein direkter Ansatz, dem System die Möglichkeit zu geben, aus einem lokalen Optimum zu entkommen, besteht darin, das ganze System in regelmäßigen Zeitabständen zu stören (zufällige Gewichtsveränderungen) und wieder neu konvergieren zu lassen. Sind die zufälligen Störungen groß genug, so kann das System das lokale Maximum der Zielfunktion ("Potentialbarriere"), die bei nicht-linearen Ausgabefunktionen an Stelle des Sattelpunktes tritt, überwinden. Dies entspricht einer Strategie des "simulierten Erhitzens" (*simulated annealing*), analog zum realen Erhitzen und Abkühlen wie es in der Physik zum Entfernen von Gitterfehlstellen in Kristallen (lokale Energieminima!) üblich ist.

Ein anderer Ansatz versucht, die Potentialbarriere zu "durchtunneln", indem die Systemveränderung nur mittels einer Wahrscheinlichkeit durchgeführt wird. Dies ermöglicht dem System, mit einer bestimmten Wahrscheinlichkeit auch gegen die durch die Zielfunktion vorgegebenen Richtung "den Berg hinauf" sich zu verändern..

Ein dritter Ansatz versucht, die Zahl der "hidden units" stückweise nach Bedarf zu erhöhen. Diese Idee geht davon aus, dass sich beim Hinzufügen von Einheiten die Leis-

tung des Netzes verbessert. Ist das bestehende Netz bereits optimal trainiert, so lässt sich das neu hinzugefügte Neuron separat optimal einrichten, so dass die Gesamtleistung steigt. Dies lässt sich sequentiell solange durchführen, bis das gewünschte Verhältnis zwischen Aufwand und Ergebnis erreicht ist. Leistungskriterium der vorhandenen Einheiten kann dabei sowohl die erzielte Ausgabeentropie [BICH89] über alle Eingabemuster sein, die bei Hinzufügen und Training einer Einheit absinken sollte, als auch der Fehler [REF91a,b], der durch jede zusätzliche Einheit vermindert werden kann oder der Fehler bei linearer Separierung [KPD90].

2.6.7 Training, Validierung, Testen

Wird ein System trainiert, so kann es „überangepasst“ (*overfitting*) werden, wenn es die Daten „speichert“ anstatt zu generalisieren. Wie muss man sich das vorstellen? In der folgenden Abb. 2.45 ist dies an der Approximation einer Funktion visualisiert. Die Funktion (durchgezogene schwarze Linie) ist nicht direkt bekannt, sondern nur einzelne, mit zufälligen Abweichungen behaftete Funktionswerte (schwarze Punkte). Wird das Netz nur mit diesen Punkten trainiert, so ist die Abweichung der Netzfunktion (gestrichelte Linie) davon bei Überanpassung sehr gering; der Trainingsfehler ist minimal. Beachten wir aber noch andere, mit zufälligen Abweichungen versehenen Beobachtungswerte (Kreise) der unbekannten Funktion, so nimmt der auf dieser Validierungsmenge beobachtete Fehler wieder zu, wenn die vermutete Funktion von der tatsächlichen Funktion abweicht und sich an die zufälligen Schwankungen anpasst.

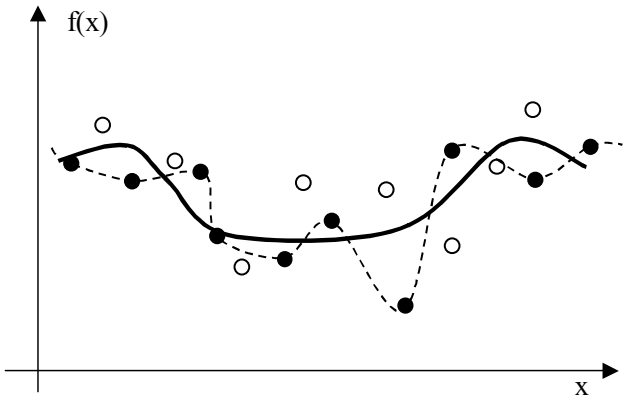


Abb. 2.45 *Generalisierung und Überanpassung bei einer Funktionsapproximation*

Wie kann man feststellen, ob im konkreten Fall eine Überanpassung vorliegt? Dazu testet man die Leistung des Netzes nach jeder Lernperiode ΔT mit Daten, die vom Training unabhängig sein sollen. Diese Daten, die man zur Validierung des Trainingserfolges benötigt, kann man als „Validierungsmenge" bezeichnen. Erhöht sich der Fehler nach einer Trainingsepoche wieder, so liegt eine Überanpassung vor; das Training sollte gestoppt werden („*stopped training*"), siehe Abb. 2.46. Hier ist die Fehlerentwicklung eines Netzes während des Trainings gezeigt, verdeutlicht an dem Fehler, den ein Trainingsmuster beim Lernschritt t macht. Der Fehler, den das trainierte Netz auf der gesamten Validierungsmenge erbringt, ist in Abständen ΔT jeweils als schwarzer Punkt ebenfalls eingetragen. Mißt man ihn nicht in großen Schritten, sondern nach jedem Lernschritt (was sehr aufwendig ist), so ergibt sich die durchgezogene Kurve. Obwohl der Trainingsfehler deutlich abnimmt, kann der Fehler dagegen auf einer unabhängigen Validierungsmenge zunehmen, sobald eine Überanpassung durchgeführt wird. Der Zeitpunkt, an dem der Fehler der Validierungsmenge am geringsten ist, bietet sich dazu an, das Training abzubrechen.

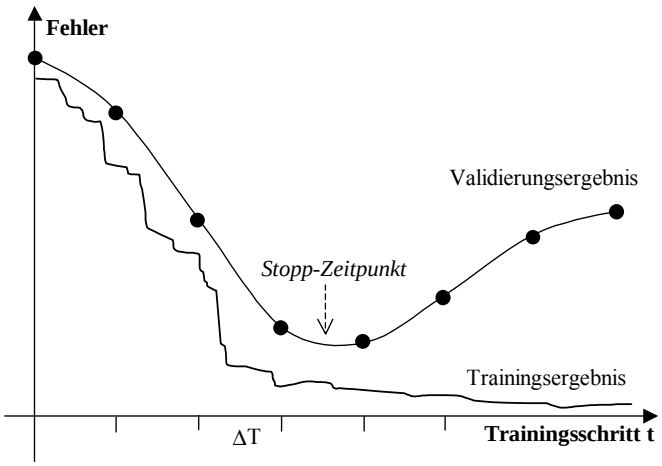


Abb. 2.46 *Überanpassung beim Lernen*

Das überwachtes Training bezeichnet man als „Kreuzvalidierung" (*cross validation*) und kann noch verfeinert werden. Der Fehler auf der Validierungsmenge und damit das Trainingsende kann von der zufälligen Zusammensetzung der Validierungsmenge abhängen. Um die tatsächliche Güte des trainierten Netzes zu bestimmen sollte ein einheit-

licher Test für alle Versionen des Netzes existieren, der auch von der Validierung unabhängig ist: der Test mit einer unabhängigen Mustermenge, der Testmenge.
So plausibel dieses mehrstufige Qualitätsssicherungsschema sein mag, so unpraktisch und unlogisch ist es auch:

- Die Trainingsmenge sollte so gewählt werden, dass alle wichtigen Variationen der Eingabemuster darin enthalten sind. Ist dies erfüllt, so kann das Validierungsmenge von den Daten her nichts mehr beitragen. Es zeigt nur die Abhängigkeit des Lernens von der Präsentationsreihenfolge, was sich auch ohne Validierungsmenge und Testmenge durch eine hohe Zahl zufälliger Wiederholungen bei kleinen Lernraten minimieren lässt.
- Auch die Validierungsmenge sollte alle Variationen enthalten, ebenso wie die Testmenge. Ist diese Voraussetzung erfüllt, so kann auch die Testmenge kein anderes Ergebnis (keinen anderen Fehler) als die Validierungsmenge liefern; sie ist prinzipiell überflüssig.
- Ein großer Unterschied zwischen den Ergebnissen der Mengen deutet darauf hin, dass die Voraussetzungen nicht erfüllt sind. Dann gibt es keinen Grund zu glauben, dass die Ergebnisse der Testmenge besser sind als die der Validierungsmenge. Es ist in diesem Fall egal, welche der beiden Mengen man zum Trainingsabbruch heranzieht; die Korrektheit der Ergebnisse sind fraglich.
- In den meisten Anwendungen gibt es nur sehr wenig (z.B. 45) Datentupel aus denen sich kaum noch guten Gewissens neben den Trainingsdaten auch Test- und Validierungsdaten abtrennen lassen. Man greift deshalb meist zu dem Trick, die Daten zu kopieren, zu mischen und dann die Mengen zufällig daraus zu wählen (*bootstrap*-Methode, s. Kapitel über Simulation in diesem Buch). In unserem Beispiel werden so aus 45 Daten 4500, die in drei Gruppen zu je 1500 Daten eingeteilt werden können. Die dabei beobachteten Unterschiede im Trainingsverhalten sind aber eher zufälliger Natur. Das Anwachsen des Fehlers des Netzes beim Testen gegenüber Training und Validierung bedeutet nicht, dass das Netz schlecht ist, sondern dies kann auch an den zufälligen Abweichungen der synthetisch generierten Mengen liegen. Im Endeffekt lässt sich keine verbindliche Aussage über das Lernzustand des Netzes treffen. Nur wenn Test- und Validierungsmengen ähnliche Resultate bringen, die deutlich vom Trainingserfolg abweichen, lässt sich auf ein Überlernen schließen, sonst nicht. In der Praxis sollte man deshalb im Fall weniger Daten sich auf Trainings- und Testmenge beschränken; beim Training durch kopierte Daten reicht auch die Trainingsmenge allein als Testmenge aus.

2.6.8 Pruning gegen Overfitting

Ist die Systemleistung nach Tests schlechter als die Leistung bei den Trainingsdaten, so ist das System durch die nichtlinearen Anteile zu speziell an die Trainingsdaten angepasst (*overfitting*) und hat nicht genügend generalisiert. Bei "verrauschten", mit zufälligen Abweichungen behafteten Daten, wie sie in der Praxis meist vorliegen, geschieht dies unter dem Ziel einer Minimierung des quadratischen Fehlers zwangsläufig, wie wir

uns leicht klarmachen können. Sei $L + \eta$ der mit der zufälligen, unkorrelierten Abweichung η behaftete Trainingswert der Lehrervorgabe L und y die Ausgabe des Netzwerks. Der erwartete quadratische Fehler ist

$$R = \langle (L+\eta-y)^2 \rangle = \langle (L-y)^2 \rangle + 2\langle L-y \rangle \langle \eta \rangle + \langle \eta^2 \rangle \qquad (2.129)$$

Angenommen, die Ausgabe approximiert den wirklichen Wert L perfekt. Dann ist bei $L - y = 0$ die Zielfunktion mit $R = \langle \eta^2 \rangle$ ungleich null und damit noch nicht bei dem absoluten Minimum angelangt. Umgekehrt bedeutet dies, dass bei $R = 0$ und nichtzentriertem Rauschen die Ausgabe y nicht den wahren Wert L approximiert, sondern einen anderen, durch die nicht-verschwindende Rauschintensität $\langle \eta^2 \rangle$ bestimmten Wert.

Es gibt verschiedene Methoden, mit diesem Problem umzugehen. Eine davon ist das *stopped training*. Allerdings ist dabei nicht unbedingt die bestmöglichste Leistungsfähigkeit gegeben.

Eine andere Möglichkeit, die Generalisierung zu erhöhen, liegt in einer Begrenzung der Zahl der "hidden units" durch Entfernen bestimmter, "unnötiger" Neuronen (*neuron pruning*) oder Weglassen des Einflusses von Eingabevariablen durch Entfernen der korrespondierenden Gewichte (*weight pruning*) [WEI90], [HUB91]. Das Kriterium für "unnötig" ist dabei problemabhängig [HER92] und muss gesondert ermittelt werden. Ein Ansatz sieht beispielsweise Gewichte mit nur geringer Varianz $\Sigma_i (w_i - \langle w_i \rangle)^2$ als solche an, da Neuronen mit großer Varianz der Gewichte größere Fähigkeiten zugeschrieben werden, Reize zu unterscheiden als solche mit kleinen. Dies lässt sich elegant verallgemeinern, indem man die Zielfunktion um einen Strafterm für die Zahl bzw. Größe der Gewichte erweitert [WEI92]. Allerdings können die Werte der Gewichte nicht durch die Netzleistung, sondern nur durch die Proportionen der Eingabe (z.B. Gewichtsvarianz durch die Datenvarianz) bedingt sein und so zu falschen Schlüssen führen.

Bei zufälligen Anfangsgewichten lassen sich *aktive* und *inaktive* Gewichte dadurch unterscheiden, dass mit einem Faktor $\alpha<1$ ein Gewichtsabfall (*weight decay*) $\mathbf{w} \rightarrow \alpha \mathbf{w}$ eingebaut wird [KRO92], der nur über aktives Lernen kompensiert werden kann. Dabei wird vermutet, dass generell Gewichte, die fast Null sind, die relative Unabhängigkeit der Repräsentation von diesen Eingabevariablen anzeigen. Auch wenn man eine Normierung der Eingabevariablen voraussetzt, ist dies sicher nicht immer so. Bessere Ergebnisse bringt in diesem Fall eine Sensitivitätsanalyse.

Ein komplizierterer, aber erfolgreicher [REH92] Ansatz von Finnoff und Zimmermann [FIZ92] sieht vor, die relative Variation der Gewichtsvariablen beim Training zu beobachten und mit diesem Testkriterium die Liste aller Variablen in "lebendige", für das Netzwerk benutzte Gewichte, und "tote" Variablen zu unterteilen. Steigt der Fehler beim Training wieder an, so werden die "toten" Gewichte überprüft und ein Prozentsatz davon in die "lebendige" Liste einsortiert.

Mit diesen Überlegungen lassen sich Strategien finden, "überflüssige" Neuronen und Gewichte wegzulassen. Dabei erhöht sich meist nicht nur die Generalisierungsfähigkeit sondern auch die Konvergenzgeschwindigkeit des Netzwerks.

2.6.9 Information als Zielfunktion der Klassifikation

Eine Neudefinition der Risikofunktion, wie sie beispielsweise mit Hilfe des Informationskriteriums möglich ist, kann nicht nur helfen, lokale Minima zu vermeiden, sondern auch die Konvergenz zu beschleunigen. Dies wollen wir an einem Beispiel näher betrachten.

Beispielsweise kann man für die Anpassung des Systems nur die tatsächliche Information benutzen, die in den Trainingsdaten enthalten ist. Dieser Ansatz für Klassifikationsprobleme [BRI90a,b], [ONI92], [BAW88], [YAI90] verwendet anstelle des Ziels des minimalen quadratischen Fehlers das Kriterium, für das Erreichen des Lernziels die Wahrscheinlichkeit der richtigen Klassifikation zu maximieren bzw. die für die richtige Klassifikation noch nötige Information zu minimieren. Dies kann für Klassifikation realer Daten durchaus zu anderen, besseren Ergebnissen führen.

Betrachten wir dazu die Wahrscheinlichkeit P_{kj}, dass eine Klassifikation ω_j bei der Eingabe des Trainingsmusters $\mathbf{x}^k$ korrekt war. Ist die bedingte Wahrscheinlichkeit $P(\omega_j|\mathbf{x}^k)$ für das Auftreten der Klasse ω_j gegeben, so kann P_{kj} mit der Kenntnis des gewünschten Ergebnisses $L_j(\mathbf{x}^k) = 0$ (falsche Klasse) oder 1 (richtige Klasse) als

$$P_{kj} = \begin{cases} P(\omega_j \mid \mathbf{x}^k) & \text{bei } L_j = 1 \\ 1 - P(\omega_j \mid \mathbf{x}^k) & \text{bei } L_j = 0 \end{cases} \tag{2.130}$$

oder als Produkt der Wahrscheinlichkeiten für falsche und richtige Klassifikation geschrieben werden

$$P_{kj} = P(\omega_j|x^k)^{L_j} \, (1\text{-}P(\omega_j|x^k))^{1\text{-}L_j} \tag{2.131}$$

Die Wahrscheinlichkeit P_k, ein $\mathbf{x}^k$ bei M unabhängigen Klassenentscheidungen richtig zu klassifizieren, ist die Produktwahrscheinlichkeit aus allen M Entscheidungen

$$P_k = \prod_{j=1}^{M} P_{kj} = \prod_{j=1}^{M} P\left(\omega_j \middle| \mathbf{x}^k\right)^{L_j} \left(1 - P\left(\omega_j \middle| \mathbf{x}^k\right)\right)^{1-L_j} \tag{2.132}$$

Dabei kann man bei $L_j \in [0,1]$ dies auch als mittlere Lehrerentscheidung, als Rate oder (auf eins normiert) als Wahrscheinlichkeit dafür interpretieren, dass die Entscheidung richtig wahr.

Die Aufgabe, die Wahrscheinlichkeit P_k einer richtigen Klassifikation zu maximalisieren, ist gleichbedeutend mit der Aufgabe, den Logarithmus davon zu maximieren. Entsprechend reicht es für ein gewünschtes Minimum aus, den negativen Logarithmus zu minimieren

$$\begin{aligned} R_x &:= I(x^k) = -\log P_k \\ &= -\sum_{j=1}^{M} L_j \log P(\omega_j|x^k) + (1\text{-}L_j) \log (1\text{-}P(\omega_j|x^k)) \overset{!}{=} \min \end{aligned} \tag{2.133}$$

Damit haben wir uns eine neue Zielfunktion definiert, die den Abstand zur Lehrerinformation zu minimieren sucht. Betrachten wir $P(\omega_j|\mathbf{x}^k)$ als die mittlere Ausgabe $\langle y_j \rangle$ von

$$y_j = \begin{cases} 1 & \text{wenn } \omega_j \text{vorliegt} \\ 0 & \text{sonst} \end{cases} \tag{2.134}$$

so wird im stochastischen Fall

$$R_x = -\sum_{j=1}^{M} L_j \log y_j + (1-L_j) \log (1-y_j) \tag{2.135}$$

und für die stochastischen Lernalgorithmen aus den Gl. (2.121) und (2.122) ändert sich das $\delta^{(2)}$ aus Gl. (2.115) zu

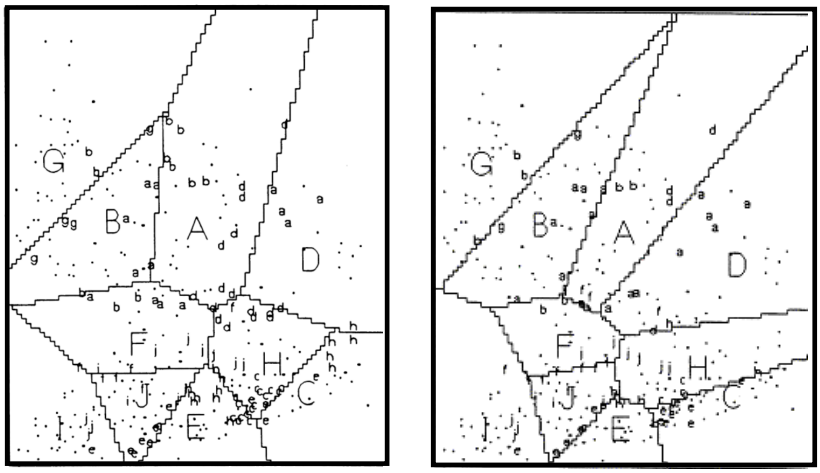
$$\delta^{(2)} = \frac{-\partial R_x}{\partial y_i} S'(z_i) = \left(\frac{L_i}{y_i} - \frac{1-L_i}{1-y_i} \right) S'(z_i) = \frac{L_i - y_i}{y_i(1-y_i)} S'(z_i) \tag{2.136}$$

Für die sigmoidale Fermi-Ausgabefunktion gilt mit Gl. (2.120) die Beziehung $S'(z_i) = y_i(1-y_i)$, so dass für die neue Zielfunktion in Gl. (2.133) alle alten Iterationsgleichungen unverändert gelten bis auf die um den Faktor $S'(z_i)$ verkleinerte Formel (2.136)

$$\delta^{(2)} = L_i - y_i \tag{2.137}$$

Die veränderte Delta-Regel vermeidet den problematischen Bereich, wenn $S(z_i)$ nahe den Extremwerten 1 oder 0 ist und dadurch die Fehlerkorrektur trotz stark falschen z_i nur noch sehr gering ausfällt [ONI92]. Damit wird die Konvergenz des Algorithmus wesentlich beschleunigt.

In der folgenden Abbildung sind die Ergebnisse einer Approximation mit dem Kriterium des quadratischen Fehlers für nicht-Gaußverteilte Daten gezeigt.



a) mit kleinstem Fehler b) mit max. Information

Abb. 2.47 *Klassifikation von Phonemdaten (nach [BRI90a])*

Aufgabe dabei war es, Vokalphoneme zu klassifizieren, die durch ihre 1. und 2. Formantenfrequenz gekennzeichnet waren. Korrekt klassifizierte Muster sind als Punkte abgebildet, nicht korrekt eingeordnete mit dem kleinen Buchstaben der zugehörigen Klasse bezeichnet. Als Klassenprototypen ergaben sich hierbei der Erwartungswert der Muster in der Klasse. In der Abb. 2.47a ist einerseits das Ergebnis eines Trainings gezeigt, das auf dem Kriterium des quadratischen Fehlers aufbaut. Im Gegensatz dazu ist andererseits in Abb. 2.47b das Ergebnis der Klassifikation, die auf maximaler Information beruht: Mit den neuen Klassenprototypen, die nicht mehr die Mustermittelwerte darstellen, ergibt sich eine Verbesserung der Klassifikation von 68% auf 78% [BRI90a].

Aufgaben

1) Trainieren Sie ein XOR-Netz aus Abb. 2.22 mit Backpropagation. Welches Problem ergibt sich ? Lösen Sie es mit sigmoidalen Funktionen. Was passiert, wenn Sie die Steilheit k der Fermi-Funktion erhöhen?

2) Lösen Sie die gleiche Aufgabe wie in 1) mittels der Zielfunktion der maximalen Klassifikationswahrscheinlichkeit aus Gl.(2.133).

3 Adaptive lineare Transformationen

Was ist *Lernen*? Im ersten Kapitel definierten wir "Lernen" allgemein als eine Funktion, bei der die Parameter eines formalen Neurons geändert werden. Dabei konkretisierten wir den Begriff "Lernen" mit einem iterativen Algorithmus, bei dem ein Systemparameter **w** (der Gewichtsvektor eines formalen Neurons) derart verändert wird, dass eine vorgegebene Zielfunktion, beispielsweise die Kosten einer Mustereinordnung, optimiert wird. Dazu haben wir uns in den ersten beiden Kapiteln einfache Klassifikationssysteme gebaut, ausgehend von den kleinen, einfachen Einheiten der formalen Neuronen.
Die Basis dieser Klassifikationen sind Merkmale. Typischerweise hängen Lerngeschwindigkeit (Konvergenzverhalten der iterativen Algorithmen) stark von der Güte der Merkmale ab: repräsentieren sie das zu lernende Objekt gut genug, oder verwirren zu viele unnötige Merkmale das System zu sehr, lenken es vom eigentlich Wichtigen ab und verlangsamen das Lernen? Jegliche „Intelligenz“ ist also mit der Fähigkeit verknüpft, Wichtiges von Unwichtigem zu unterscheiden. In diesem Kapitel wollen wir nun Methoden kennen lernen, mit einfachen linearen Operationen (linearen Schichten) wichtige Merkmale aus den beobachteten Datenmustern zu erschließen.

3.1 Lineare Schichten

Angenommen, wir verwenden eine lineare Ausgabefunktion S(z) = z. Dann ist nach unserer Grundmodellierung aus Kapitel 1 die Ausgabe y_i

$$y_i = \mathbf{w}_i^T \mathbf{x} \tag{3.1}$$

und in Vektornotation für alle Ausgaben $\mathbf{y} = (y_1, \ldots, y_m)$

$$\mathbf{y}^{(1)} = \mathbf{W}^{(1)} \mathbf{x} \tag{3.2}$$

mit der Matrix $\mathbf{W}^{(1)} = (w_{ij})$ der Gewichte der ersten Schicht, notiert mit dem geklammerten Index oben„$^{(1)}$“. Die Klammernotation soll dabei verhindern, dass wir den Index mit einem Exponenten verwechseln. Werden mehrere lineare Schichten hintereinander gesetzt, so wirkt die Ausgabe der einen Schicht als Eingabe für die nächste. Die Ausgabe der zweiten Schicht ist somit

$$\mathbf{y}^{(2)} = \mathbf{W}^{(2)}\mathbf{x}^{(2)} = \mathbf{W}^{(2)}\mathbf{y}^{(1)} = \mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x} = \mathbf{W}\mathbf{x} \qquad \text{mit } \mathbf{W} := \mathbf{W}^{(2)}\mathbf{W}^{(1)} \quad (3.3)$$

Die Funktionen beider linearen Schichten $\mathbf{W}^{(2)}$ und $\mathbf{W}^{(1)}$ lassen sich also zu der Funktion einer einzigen, linearen Ersatzschicht $\mathbf{W}$ zusammenfassen. Dies gilt aber ebenfalls für alle weiteren Schichten, so dass sich die Funktion einer beliebig langen Sequenz von linearen Schichten immer äquivalent zu einer einzigen, linearen Schicht ist.

Es reicht also aus, für die Eigenschaften auch mehrerer linearer Schichten immer nur eine einzige Schicht zu betrachten.

3.2 Hebb'sches Lernen und Merkmalssuche

In diesem Abschnitt werden wir eine Art von Lernregeln kennen lernen, die neue, unerwartete Einsichten in die Fähigkeiten formaler, linearer Neuronen eröffnen, wesentliche Merkmale der beobachteten Muster $\{\mathbf{x}\}$ zu ermitteln. Diese Eigenschaften des Lernens wird uns ermöglichen, durch Weglassen unwesentlicher Merkmale die Daten zu komprimieren: das System führt eine *Generalisierung* oder *Abstraktion* durch.

3.2.1 Beschränktes Hebb'sches Lernen

Im zweiten Kapitel haben wir die Hebbsche Lernregel kennengelernt. Die Hebbsche Lernregel lautet mit der Proportionalitätskonstanten (*Lernrate*) $\gamma(t)$

$$\Delta\mathbf{w}(t) = \gamma(t)\,\mathbf{x}y \qquad (3.4)$$

oder $\Delta\mathbf{W}(t) = \gamma(t)\,\mathbf{x}\mathbf{y}^T$

Dieses inkrementelle Anwachsen der Gewichte bei der Aktivität von Eingang und Ausgang ist aber ziemlich problematisch: Bei anhaltender Aktivität beider Zellen wachsen die Gewichte ins Unendliche; die Zellen kennen kein "Vergessen". Diese Annahme ist nicht realistisch. Wie verschiedene Experimente gezeigt haben (s. z.B. [SING85]), schwächen aktive Verbindungen andere, unbenutzte Verbindungen. Die Ursache dafür könnte beispielsweise in einer nur begrenzt verfügbaren, molekularen Ressource (chem. Reaktionsgleichgewicht!) liegen, die bei unbenutzten Synapsen zugunsten der stärker benutzten wieder abgebaut wird. Dies könnte beispielsweise bei der Nährstoffversorgung durch die Sternzellen geschehen.

Für die genaue Formulierung eines solchen verbindungs-abbauenden, selektiven Mechanismus gibt es verschiedene Vorschläge.

Nicht-lineares Abklingen der Synapsen

In der *zeitkontinuierlichen Form* kann man ein "Überschreiben" der Gewichte mit einem Term $-\mathbf{w}(t)$ berücksichtigen, wobei die Regel bei konstanten $\mathbf{x}$ und y als Grenzwert erscheint

$$\gamma^{-1}\frac{\partial}{\partial t}\mathbf{w}(t) = -\mathbf{w}(t) + \mathbf{x}(t)y(t) \quad \textit{Zeitkontinuierliche Hebb'sche Lernregel} \qquad (3.5)$$

oder $$\mathbf{w}(t) = (1-\gamma)\,\mathbf{w}(t-1) + \gamma(t)\mathbf{x}y \qquad (3.6)$$

in der iterativen Version. Die Zeitkonstante des „Überschreibens“ oder „Abklingens“ ist dabei γ.

Dabei müssen wir zwei verschiedene Zeitmaßstäbe (Mechanismen) unterscheiden: Die kurzzeitige Aktivität z(t), die sich bei Änderung der Eingabe relativ schnell ändern kann, hat in der zeitkontinuierlichen Form eine kleinere Zeitkonstante $\tau = \gamma^{-1}$ als die sich langsam ändernden Gewichte, die während der Aktivität praktisch konstant bleiben und sich aus einer Mittelung der Aktivität ergeben.
Ein plausibles Bild sieht jede Synapse wie einen undichten Kondensator: dieser entlädt sich mit der Zeit, wenn keine neue Ladung aufgebracht wird. Ein *linearer* Abklingterm wie in Gl.(3.5) führt bei einem linearen System mit $y = \mathbf{x}^T\mathbf{w}$ zu

$$\frac{\partial}{\partial t}\mathbf{w}(t) = \gamma_1\mathbf{x}(t)\cdot y(t) - \gamma_2\mathbf{w}(t) = \gamma_1\mathbf{x}\mathbf{x}^T\mathbf{w} - \gamma_2\mathbf{w} \qquad (3.7)$$

Im Erwartungswert $\langle . \rangle$ entspricht dies mit der Autokorrelationsmatrix $\mathbf{A} = \langle \mathbf{x}\mathbf{x}^T \rangle$ der Gleichung

$$\frac{\partial}{\partial t}\langle \mathbf{w}(t) \rangle = \gamma_1\langle \mathbf{x}\mathbf{x}^T \rangle\,\mathbf{w} - \gamma_2\mathbf{w} = \gamma_1\mathbf{A}\mathbf{w} - \gamma_2\mathbf{w} \qquad (3.8)$$

Für die Fixpunkte $\mathbf{w}^*$ gilt dabei $\partial\mathbf{w}(t)/\partial t = 0$. Wie man leicht sieht, erfüllen alle *Eigenvektoren* $\mathbf{e}$ der Matrix $\mathbf{A}$ mit der *Eigenvektorgleichung* $\mathbf{A}\mathbf{e} = \lambda\mathbf{e}$ ($\lambda := \gamma_2/\gamma_1$) die obige Fixpunktgleichung. Sind sie auch stabile Fixpunkte? Im eindimensionalen Fall mit $r := \langle x^2 \rangle$ entspricht die Differenzialgleichung dem Ausdruck

$$\frac{\partial}{\partial t}w(t) = \gamma(r-1)\,w(t) \qquad \text{bei } \gamma_1 = \gamma_2 = \gamma \qquad (3.9)$$

der als Lösung

$$w(t) = w_0 e^{\gamma(r-1)t} \qquad (3.10)$$

hat. Mit fortschreitender Zeit geht w(t) für $r > 1$ gegen unendlich und für $r < 1$ gegen null. Also ist $r = 1$ kein stabiler Fixpunkt; die Eigenvektoren von $\mathbf{A}$ erfüllen zwar die Bedingung $\partial w/\partial t = 0$, aber sie bedeuten keine stabilen Fixpunkte.
Erst ein nicht-linearer Abklingterm, beispielsweise in der Form einer *n*-ten Potenz,

$$\frac{\partial}{\partial t} w_i(t) = \gamma_1 x_i y - \gamma_2 w_i^n \qquad n>1 \qquad (3.11)$$

führt, wie Riedel und Schild zeigten [RIE92], zu nicht-trivialen, endlichen Fixpunkten.

Normierung der Gewichte

Eine anderes Modell macht die Annahme, dass zusätzlich zu der Hebb'schen Regel ein Normierungsmechanismus für die Gewichte eingebaut ist, um das Anwachsen der Gewichte durch die Hebb'sche Lernregel zu begrenzen:

$$\sum_i \hat{w}_i^2 \stackrel{!}{=} 1 \qquad (3.12)$$

wobei die normierten Gewichte mit $\hat{w}$ bezeichnet sind. Dann sei

$$\hat{w}_i = \frac{w_i(t)}{|\mathbf{w}(t)|}\ , \text{ so dass } \sum_i \hat{w}_i^2 = \frac{1}{|\mathbf{w}(t)|^2} \sum_i w_i^2 = 1 \text{ gilt} \qquad (3.13)$$

Was ist das Konvergenzziel der Hebb'schen Lernregel mit normierten Gewichten? Betrachten wir dazu eine Zielfunktion, die die Hebb'sche Lernregel aus Gl. (3.1) als stochastische Gradientenabstiegsregel hat. Wie man durch partielles Ableiten leicht nachprüfen kann, lautet mit dem erwarteten Gradienten

$$\frac{\partial R(\mathbf{w})}{\partial \mathbf{w}} = \langle \mathbf{x} y \rangle = \langle \mathbf{x}\mathbf{x}^T \rangle \mathbf{w} = \mathbf{A}\mathbf{w} \qquad \text{bei } y = \mathbf{x}^T\mathbf{w} \quad \textit{lin. Neuron} \qquad (3.14)$$

die Zielfunktion

$$R(\mathbf{w}) = \tfrac{1}{2}\, \mathbf{w}^T\mathbf{A}\mathbf{w} \quad \text{mit } |\mathbf{w}| = \text{const} = 1 \qquad (3.15)$$

Was ist das Extremum dieser Zielfunktion, das mit dem Gradientenalgorithmus erreicht wird? Das Problem, den Extremwert einer Funktion unter Nebenbedingungen zu finden, wird durch die Methode der „Langrang'schen Parameter“ erreicht. Dazu gehen wir folgendermaßen vor:
Zuerst bilden wir eine Hilfsfunktion L, die aus der zu maximierenden Funktion und der mit einem Hilfsparameter μ gewichteten Nebenbedingung besteht, Die Nebenbedingung ist in einer Form aufgeschrieben, die null ergibt, hier: $\mathbf{w}^2 - 1 = 0$.

$$L(\mathbf{w},\mu) = R(\mathbf{w}) + \mu\, (\mathbf{w}^2 - 1) \qquad (3.16)$$

Die für ein Extremum notwendigen Bedingungen sind die Ableitungen dieser Hilfsfunktion

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \text{ und } \frac{\partial L}{\partial \mu} = 0 \tag{3.17}$$

In unserem Fall sind dies

$$\frac{\partial L}{\partial \mathbf{w}}(\mathbf{w},\mu) = \frac{\partial R(\mathbf{w})}{\partial \mathbf{w}} + \mu 2\mathbf{w} = \mathbf{A}\mathbf{w} + \mu 2\mathbf{w} = 0$$

oder $\mathbf{A}\mathbf{w} = \lambda\mathbf{w}$ mit $\lambda := -2\mu$ (3.18)

Dies ist eine *Eigenwertgleichung* für **A**; die Lösungen $\mathbf{w}^*$ sind die Eigenvektoren der Autokorrelationsmatrix **A**. Die Zielfunktion (3.15) wird bei dem Eigenvektor $\mathbf{w}^*$ mit dem größten Eigenwert maximal und beim Eigenvektor mit dem kleinsten Eigenwert minimal. Welche der beiden Extreme erreicht wird hängt von der Lerngleichung ab: Verwenden wir einen Gradientenabstieg, so wird das Minimum gelernt, beim Gradientenaufstieg das Maximum.

Die Tatsache, dass beim Lernen der Korrelation von Eingabe **x** und linearen Aktivität z bei beschränkten Gewichten der Eigenvektor mit dem größten Eigenwert gelernt wird, war übrigens schon vorher bekannt (s. [AMA77], S.179), aber nicht wie bei Oja für eine spezielle Lernregel genutzt worden. Oja konnte übrigens für seine Lernregel beweisen, dass die Lernregel nicht nur eine Approximation des Hebbschen Lernens bei beschränkten Gewichten darstellt, sondern als Konvergenzziel tatsächlich den Eigenvektor mit dem größten Eigenwert hat.

Verwendet man mehrere derartige Neuronen und koppelt sie geeignet miteinander, so kann ein solches System *alle* Eigenvektoren von **A** als Gewichte extrahieren. Damit wirken die formalen Neuronen wie statistische Analysatoren: Das Muster **x** wird von seiner Darstellung in *n* alten Basisvektoren auf *n* neue Basisvektoren, die Eigenvektoren, transformiert. Dabei ist die neue Basis durch die Statistik der Daten selbst und deren inneren Zusammenhänge bestimmt. Was verbirgt sich hinter der neuen Darstellung der Daten?

3.2.2 Merkmalssuche mit PCA

In der statistischen Analyse von Daten in der Psychologie und Soziologie stellte sich schon früh die Frage, wie man das Material der teilweise korrelierten Variablen x_1, x_2 so transformieren kann, dass nur noch unkorrelierte ("orthogonale") Variable existieren, also Variable, deren Kovarianz

$$C_{12} = \langle (x_1 - \langle x_1 \rangle)(x_2 - \langle x_2 \rangle) \rangle \qquad \text{DEF } \textit{Kovarianz} \tag{3.19}$$

mit $C_{12} = 0$ verschwindet. Durch den Wechsel in der Beschreibungsbasis *(Transformation der Basisvektoren)* auf "natürliche", dem Problem angemessene Variable erhofft man sich nun eine verbesserte, von der Meßmethode möglichst unabhängigere Formulierung der betrachteten Zusammenhänge.

Dazu ermittelt man zunächst den Richtungsvektor der stärksten Datenänderung (größten Varianz σ_i^2) als ersten neuen Basisvektor. Dann wird der Basisvektor in Richtung der größten Varianz orthogonal dazu gesucht, und so fort bis alle *n* alte Basisvektoren (Variablen) auf *n* neue Basisvektoren *(principal variables)* abgebildet wurden. Die Komponenten $\lambda_i = \sigma_i^2$ dieser neuen Basis sind die *principal components*. Das Verfahren zur Ermittlung dieser Komponenten heißt *Hauptkomponentenanalyse (principal component analysis*, *PCA).*

Durch die Anpassung an die Daten erlaubt das neue Koordinatensystem, die Beschreibungsbasis der Daten zu verbessern. In der Abb. 3.1 ist ein solches Beispiel zu sehen, bei der zwei Normalverteilungen überlagert sind, so dass sie sich weder in der x_1 noch in der x_2 Komponente trennen lassen, da es kein lokales Minimum der Projektion der Summenfunktion auf eine der beiden Koordinatenachsen *(Marginalverteilungen)* gibt. Nimmt man als Richtungen des transformierten Koordinatensystems dagegen die maximalen Varianzen, so ist nun eine Trennung der beiden Verteilungen durch die $\mathbf{e}_1$-Variable gut möglich. Die in Abb. 3.1 skizzierten Marginalverteilungen auf den zu den Basisvektoren $\mathbf{e}_1$ und $\mathbf{e}_2$ parallelen Geraden 1 und 2 illustrieren dies qualitativ.

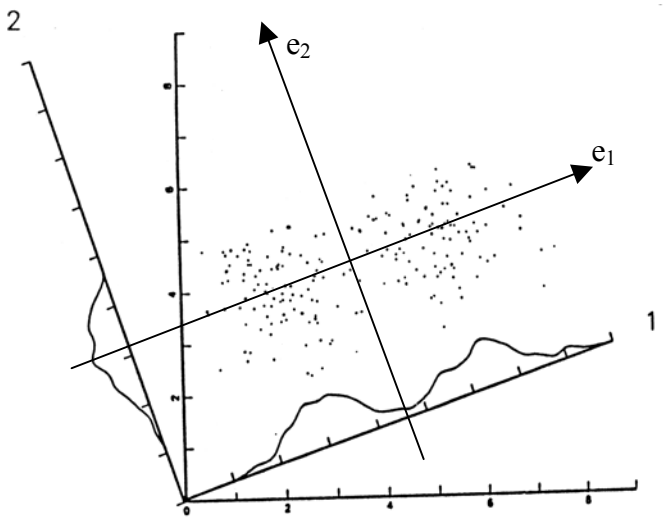


Abb. 3.1 Hauptkomponentenanalyse zweier Normalverteilungen

Es zeigte sich bald, dass dieses Problem auch in anderen Disziplinen unter anderem Namen, beispielsweise in der Physik unter dem Namen *Hauptachsentransformation,* bekannt ist. Die Lösung des Problems ist identisch mit dem Problem, alle Eigenvektoren $\mathbf{e}_i$ *(principal variables, Hauptachsen)* und Eigenwerte λ_i *(principal components)* der Kovarianzmatrix C_{xy} der n-dimensionalen Variablen **x** und y zu finden. Dabei ist die Kovarianzmatrix zwischen **x** und **y** wie folgt definiert:

$$\mathbf{C}_{xy} = \begin{bmatrix} \langle x_1 - \langle x_1 \rangle \rangle \langle y_1 - \langle y_1 \rangle \rangle & \cdots & \langle x_1 - \langle x_1 \rangle \rangle \langle y_n - \langle y_n \rangle \rangle \\ \vdots & \ddots & \vdots \\ \langle x_n - \langle x_n \rangle \rangle \langle y_1 - \langle y_1 \rangle \rangle & \cdots & \langle x_n - \langle x_n \rangle \rangle \langle y_n - \langle y_n \rangle \rangle \end{bmatrix} \tag{3.20}$$

Dies lässt sich auch mit dem äußeren Produkt abkürzen

$$\mathbf{C}_{xy} = \langle \mathbf{x} - \langle \mathbf{x} \rangle \rangle \langle \mathbf{y}^T - \langle \mathbf{y}^T \rangle \rangle \qquad \textit{Kovarianzmatrix} \tag{3.21}$$

oder als Tupel aller Matrixelemente schreiben.

$$\mathbf{C}_{xy} = [C_{ij}] = \left[\langle x_i - \langle x_i \rangle \rangle \langle y_j - \langle y_j \rangle \rangle \right] \tag{3.22}$$

Betrachten wir nur einen einzigen n-dimensionalen Eingaberaum $\{\mathbf{x}\}$, so bezieht sich die Kovarianzmatrix auf $\mathbf{x} = \mathbf{y}$, also die Kovarianz zwischen allen Elementen von $\mathbf{x}$. Ist außerdem noch der Erwartungswert $\langle \mathbf{x} \rangle = \mathbf{0}$, so wird die Kovarianzmatrix als *Autokorrelationsmatrix* bezeichnet

$$\mathbf{A} = \langle \mathbf{x}\mathbf{x}^T \rangle \qquad \textit{Autokorrelationsmatrix} \tag{3.23}$$

Man beachte, dass der Erwartungswert im diskreten Fall über alle vorliegenden Muster $\{\mathbf{x}\}$ gebildet werden muss. Haben wir also beispielsweise M=100 unterschiedliche Muster und wollen die Autokorrelationsmatrix $\mathbf{A}$ bilden, so müssen wir für jeden Koeffizienten a_{ij} der Matrix $A = (a_{ij})$ die mittlere Summe mit dem k-ten Muster $\mathbf{x}(k)$, k=1..M

$$a_{ij} = \langle x_i x_j \rangle_x = \frac{1}{M} \sum_{k=1}^{M} x_i(k) x_j(k) \tag{3.24}$$

ausrechnen.

Lineare Aktivierung und lineare Ausgabefunktionen bei formalen Neuronen bedeutet allgemein ein lineares Systemverhalten. In mechanischen, linearen Systemen tritt oft das Phänomen auf, dass die resultierende Erregung der Anregung gleicht. Dies ist in vielen Gebieten eine wichtige Erscheinung, beispielsweise im Flugzeugbau, wo bei einer Resonanz der Flugzeugflügel durch Luftwirbel die Flügel im Fluge abbrechen können. Die Eigenvektoren ("Eigenfunktionen") der linearen Verbiegung entsprechen hier Anregungen mit bestimmten Frequenzen ("Eigenfrequenzen") und müssen bei der Tragflügelkonstruktion berücksichtigt werden. Bei den formalen Neuronen entspricht die Resonanz einer besonders starken Antwort auf Eingabemuster mit bestimmten Merkmalen.

Die Transformation auf das Basissystem von Eigenvektoren hat, wie in Anhang A näher ausgeführt, eine interessante Eigenschaft: sie hat *maximale Informationserhaltung* für Gauß-verteilte Eingabedaten $\{\mathbf{x}\}$ bei linearen Systemen.

Dies gibt Anlass zu den philosophischen Überlegungen, dass eine "intelligente" Informationsverarbeitung, die typische Merkmale der Daten findet, nur dann möglich zu sein scheint, wenn nicht nur gute Aktivitäten unterstützt werden, sondern auch die verwendeten Ressourcen begrenzt sind. Lässt man dagegen beliebige Ressourcen zu, so wächst ein System unter anderem zwar auch in die "richtige" Richtung, aber wesentlich langsamer. Der Mechanismus der Umverteilung, der aus den begrenzten Ressourcen folgt, scheint die kontrastive Orientierung des Systems an der Realität der Eingabedaten stark zu unterstützen. Die denkbaren Folgerungen aus diesen Überlegungen für Politik, Behörden und Wirtschaft sind allerdings wesentlich weniger gesichert.

Was sind die mathematischen Eigenschaften von Eigenvektoren? Rekapitulieren wir für ein besseres Verständnis dieser speziellen Merkmale kurz die wichtigsten Eigenschaften eines Eigenvektorsystems.

Eigenvektoren und Eigenwerte

Wie wir aus der linearen Algebra wissen, bilden die mit

$$\mathbf{B}\mathbf{e}_k = \lambda_k \mathbf{e}_k \qquad \textit{Eigenwertgleichung} \qquad (3.25)$$

definierten Eigenvektoren $\mathbf{e}_k$ einer Matrix **B** ein System von linear unabhängigen (aber nicht unbedingt orthogonalen) Basisvektoren. Ist die Matrix hermitesch (im reellen Fall: symmetrisch), so sind die Eigenwerte reell und die Eigenvektoren orthogonal, aber nicht normiert, da ja mit $\mathbf{e}_k$ auch jeder andere Vektor $c\mathbf{e}_k$ mit reeller Konstante c Eigenvektor ist. Ist der Spalten- bzw. Zeilenrang rang(**B**) kleiner als die Dimension der Matrix, so gibt es linear abhängige Zeilen bzw. Spalten in **B** (" degenerierte Matrix") und es gibt n - rang(**B**) gleiche Eigenwerte; die dazu gehörenden (orthogonalisierbaren) Eigenvektoren werden als "degeneriert" bezeichnet.

Die Eigenvektoren besitzen unter anderem folgende für uns interessante Eigenschaften:

- Für die Eigenvektoren der Matrix **B** gilt die Eigenwertgleichung, so dass mit der Matrix **E** aus den Spalten ($\mathbf{e}_1$, ..., $\mathbf{e}_m$) der Eigenvektoren folgt

$$\mathbf{B}\,\mathbf{E} = \mathbf{E}\,\Lambda \qquad \Lambda := \begin{pmatrix} \lambda_1 & & 0 \\ & \bullet\bullet\bullet & \\ 0 & & \lambda_m \end{pmatrix} \qquad (3.26)$$

bzw. $\mathbf{B} = \mathbf{E}\,\Lambda\,\mathbf{E}^{-1}$ und $\mathbf{E}^{-1}\mathbf{B}\,\mathbf{E} = \Lambda$ (3.27)

Umgekehrt ist auch die Matrix **B** durch ihre Eigenwerte und -vektoren eindeutig festgelegt. Haben wir (3.26) vorliegen, so ergibt sich durch Rückrechnung (3.25): die Matrix **E** muss die Matrix der Eigenvektoren sein. Für symmetrische, reelle Matrizen (z.B. die Kovarianz- und Autokorrelationsmatrizen **C** und **A**) gilt allge-

mein $\mathbf{E}^{-1} = \mathbf{E}^{T}$, d.h. die Eigenvektoren sind nicht nur linear unabhängig, sondern auch orthogonal.

- Die Eigenwertgleichung (3.25) lässt sich auch mit der Einheitsmatrix **I** um stellen zu

$$\mathbf{A}\mathbf{e}_k - \lambda_k\mathbf{e}_k = (\mathbf{A} - \lambda_k\mathbf{I})\,\mathbf{e}_k = 0 \qquad (3.28)$$

  Bilden wir von diesem Ausdruck die Determinante, so muss für die nichttriviale Lösung $\mathbf{e}_k \neq \mathbf{0}$ des Gleichungssystems gelten

$$\det(\mathbf{A} - \lambda_k\mathbf{I}) = |\mathbf{A} - \lambda_k\mathbf{I}| = 0 \quad \textit{Charakteristische Gleichung} \qquad (3.29)$$

Bekannterweise lässt sich jede Determinante auch als Polynom entwickeln, in diesem Fall als charakteristisches *Polynom* $P(\lambda) = 0$, das als Nullstellen gerade die Eigenwerte der *charakteristischen Matrix* $(\mathbf{A}-\lambda\mathbf{I})$ besitzt. Sind die λ_k bekannt, so lässt sich durch Einsetzen in Gl.(3.25) die Form der Eigenvektoren gewinnen. Man beachte, dass durch Gl.(3.25) nur die Richtung der Eigenvektoren definiert wird und nicht ihre Länge; jeder beliebig skalierte Eigenvektor $\mathbf{e}' = c\mathbf{e}$ erfüllt ebenfalls Gl.(3.25) und ist damit Eigenvektor.

Beispiel 1

Angenommen, wir beobachten Muster $\mathbf{x} = (x_1,x_2)$, deren beide Komponenten x_1 und x_2 miteinander korreliert sind durch die Beziehung $x_2 = ax_1$, so dass $a\langle x_1^2\rangle = \langle x_1x_2\rangle \neq 0$ gegeben ist. In welchen Richtungen verlaufen dann die Hauptachsen ? In der nachfolgenden Abbildung ist diese Situation am Beispiel einiger Musterpunkte gezeigt.

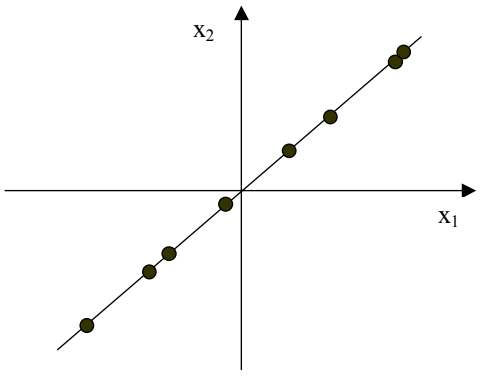


Abb. 3.2 Rauschfrei korrelierte Muster

Anstatt die Kovarianzmatrix aller Punkte {**x**} zu bilden, können wir die Kenntnis der Systematik (lineare Abhängigkeit $x_2 = ax_1$) benutzen, um die Matrix zu errechnen. Mit $\langle x_1 \rangle = \langle x_2 \rangle = 0$ ist die Kovarianzmatrix **C** gleich der Autokorrelationsmatrix

$$\mathbf{A} = \begin{pmatrix} \langle x_1{}^2 \rangle, & \langle x_1 x_2 \rangle \\ \langle x_2 x_1 \rangle, & \langle x_2{}^2 \rangle \end{pmatrix} = \langle x_1{}^2 \rangle \begin{pmatrix} 1, & a \\ a, & a^2 \end{pmatrix}$$

so dass mit $b := \langle x_1{}^2 \rangle$ die charakteristische Gleichung lautet

$$|\mathbf{A} - \lambda \mathbf{I}| = \begin{vmatrix} b-\lambda, & ba \\ ba, & ba^2-\lambda \end{vmatrix} = (b-\lambda)(ba^2-\lambda) - b^2a^2 = \lambda^2 - \lambda b(a^2+1)$$

mit dem charakt. Polynom $\lambda\,(\lambda - b(a^2+1)) = 0$ und den Lösungen $\lambda_1 = 0$, $\lambda_2 = b(a^2+1)$. Die erste Lösung zeigt, dass die Variablen linear abhängig sind („degenerierte“ Eigenvektoren). Setzen wir die zweite Lösung in die Eigenwertgleichung ein, so ergibt sich $\mathbf{Aw} = \lambda \mathbf{w}$ oder

$$\mathbf{A}\,\mathbf{w} = \begin{pmatrix} bw_1 & + & baw_2 \\ baw_1 & + & ba^2w_2 \end{pmatrix} = \begin{pmatrix} b(a^2+1)w_1 \\ b(a^2+1)w_2 \end{pmatrix} = \lambda\,\mathbf{w}$$

und mit der ersten (und zweiten!) obigen Gleichung der Zusammenhang $w_2 = aw_1$. Der einzige nicht-degenerierte Eigenvektor (und damit die Hauptachse) liegt also genau auf der Gerade durch die Musterpunkte in Abb. 3.2. Die zweite, dazu senkrechte Achse des zweiten Eigenvektors hat einen Eigenwert (Varianz) von null. Bei diesem Beispiel kann man also, ohne Information zu verlieren, die zwei Eingabevariablen durch eine einzige ersetzen, die der Projektion auf den Eigenvektor mit dem größten Eigenwert entspricht.

Faktorenanalyse

Es gibt auch noch andere mögliche Transformationen, die ebenfalls interessante Eigenschaften haben. Beispielsweise sagt eine Transformation von n Variablen auf n Eigenvektoren nichts darüber aus, ob dabei auch wirklich alle Variablen nötig sind und nicht durch weniger Variablen ("gemeinsame Ursachen") ersetzt werden können. Ein Beispiel dafür sind Schulnoten von Schülern, die meist positiv miteinander korreliert sind. Je nach Arbeitshypothese lassen sich solche Faktoren wie "Sprachbegabung" und "Mathematische Begabung" als neue Variablen extrahieren, von denen die aktuellen Noten in den verschiedenen Fächern nur zufällige Ausprägungen einer gemeinsamen Ursachengröße darstellen. Die Suche nach neuen, den Variablen zugrunde liegenden, gemeinsame Faktoren wird *Faktorenanalyse (factor analysis)* genannt und ist stark von der Arbeitshypothese (Zahl der erwarteten Faktoren) abhängig. In unserem Beispiel könnte man auch anstelle von zwei Faktoren nur einen gemeinsamen Faktor erwarten und das Datenmaterial

(Schulnoten) als zufällige Abweichungen von dem gemeinsamen Faktor "Intelligenz" betrachten.

Allgemein formuliert transformiert die Hauptkomponentenanalyse die n alten Basisvektoren auf eine gleiche Zahl von $m = n$ neuen Basisvektoren, während die Faktorenanalyse $m < n$ Faktoren annimmt und die gesamte Varianz darauf verteilt. Näheres über die Faktorenanalyse ist in der Literatur, beispielsweise in [LAW71] zu finden.

Unabhängige Komponenten

Sucht man nach der gemeinsamen Ursache mehrerer, von einander abhängiger Beobachtungen, so bedeutet dies, von n stochastisch abhängigen Variablen auf $m \leq n$ unabhängige Variable überzugehen. Existieren diese Variable, so sind sie auch durch die Bedingung unabhängiger Auftrittswahrscheinlichkeiten

$$p(x_1,x_2,..,x_m) = p(x_1)p(x_2)..p(x_m) \qquad \textit{Unabhängigkeit} \qquad (3.30)$$

hinreichend charakterisiert.

Aus der Unabhängigkeit zweier Variabler z_i, z_k folgt

$$\langle z_i\, z_k\rangle = \langle z_i\rangle\langle z_k\rangle \qquad (3.31)$$

und damit für die Kovarianz von $z_i := x_i - \langle x_i\rangle$, $z_k := x_k - \langle x_k\rangle$

$$\begin{aligned}\langle (x_i-\langle x_i\rangle)(x_k-\langle x_k\rangle)\rangle &= (\langle x_i-\langle x_i\rangle\rangle)(\langle x_k-\langle x_k\rangle\rangle) \\ &= (\langle x_i\rangle-\langle x_i\rangle)(\langle x_k\rangle-\langle x_k\rangle) \;= 0\end{aligned} \qquad (3.32)$$

Also folgt aus der Unabhängigkeit die Unkorreliertheit der Variablen. Trotzdem sind aber beide nicht identisch; aus der Unkorreliertheit folgt umgekehrt *nicht* die Unabhängigkeit. Je nach Anwendung kann eine PCA vollkommen falsche und unerwünschte Ergebnisse für abhängige, aber dekorrelierte Merkmale liefern.

Aufgaben

3.1) Es seien 3-dim. Muster $\{\mathbf{x}\}$ gegeben, die eine Punktwolke von Ei-förmiger Gestalt im 3-dim Raum bilden. In welche Richtung müßte der Eigenvektor der Kovarianzmatrix mit dem größten Eigenwert (*principal component*) zeigen? Angenommen, die Punktwolke habe scheibenförmige Gestalt. Welches ist die Richtung des Eigenvektors mit dem kleinsten Eigenwert ?

3.2) Zum Nachprüfen der Funktionsfähigkeit simulierter Neuronaler Netze ist es oft hilfreich, die erwarteten Lösungen der numerischen Iteration ('Lernen') für kleine, überschaubare Beispiele vorher analytisch zu bestimmen.

Es seien die Muster $\mathbf{x}^1 = (1,0)$ und $\mathbf{x}^2 = (1,1)$ gegeben. Gesucht sind die Eigenvektoren und Eigenwerte der Autokorrelationsmatrix dieser Muster.
Dazu bestimme man

a) die Autokorrelationsmatrix **A**
b) die charakt. Gleichung
c) das charakt. Polynom

und löse dieses Polynom 2. Ordnung. Die beiden Lösungen λ_1 und λ_2 setze man in die Eigenwertgleichung ein und bestimme die beiden Eigenvektoren. Man prüfe, ob auch $\mathbf{w}_1 \perp \mathbf{w}_2$ ist.

3.3) Zeigen Sie, dass die erwartete Ausgabe $\langle y^2 \rangle$ in einem linearen System $y = \mathbf{w}^T\mathbf{x}$ mit $\langle \mathbf{x} \rangle = 0$ besonders klein wird, wenn der normierte Gewichtsvektor **w** dem Eigenvektor der Autokorrelationsmatrix **A** mit dem kleinsten Eigenwert entspricht.

3.3 Lineare Transformation unter Nebenbedingungen

Eine Möglichkeit, Daten zu komprimieren, besteht in einer Reduzierung der *n* Eingabekanäle auf *m* Ausgabekanäle (Datenwege) der Schicht. Haben wir mit $n > m$ weniger neuronale Signale bei der Ausgabe als bei der Eingabe, so findet eine Datenreduktion statt. In Abb. 3.3 ist diese Situation veranschaulicht. Eine Eingabe mit *n* Kanälen wird mit einer Matrix **W** der Parameter transformiert, komprimiert, gespeichert bzw. übertragen und dann wieder dekomprimiert und rücktransformiert. Dabei werden nur die Werte von *m* Kanälen, also *m* Koeffizienten y_i, übertragen; die restlichen *n-m* Koeffizienten werden als konstant angenommen.

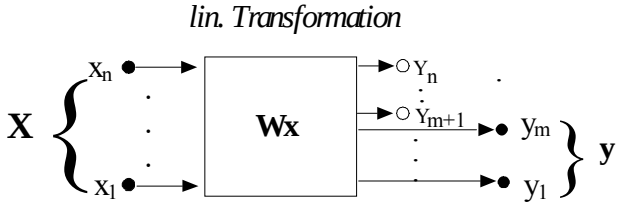


Abb. 3.3 Lineare, kodierende Merkmalstransformation

Die Kompression geschieht nicht durch die Transformation, sondern nur durch das Ersetzen der aktuellen Werte $y_{m+1},...,y_n$ mit konstanten Werten $c_{m+1},...,c_n$. Dabei machen wir natürlich durch die Kompression einen Fehler. Angenommen, wir möchten ein Muster **x**, beispielsweise ein Bild, möglichst genau wieder rekonstruieren, nachdem wir es wie oben beschrieben in einen Vektor **y** komprimiert haben: Wie müssen wir dabei die Parametermatrix **W** der Transformation wählen, um den durch die Kompression entstandenen Fehler so klein wie möglich zu machen ?

3.3.1 Lineare Approximation mit kleinstem Fehler

Ein mögliches Maß für den Fehler zwischen dem Original **x** und der Rekonstruktion $\hat{\mathbf{x}}$ ist die Differenz $(\mathbf{x}-\hat{\mathbf{x}})$. Dies eignet sich aber nicht zum Minimieren: das Minimum des Ausdrucks ist bei $-\infty$. Quadriert man aber den Ausdruck, so kann er nur minimal null werden. Wir wählen uns als Ziel, dieses Fehlermaß im Erwartungswert so klein wie möglich zu machen.

$$\min_{\mathbf{w}} R(\mathbf{W}) = \min \left\langle (\mathbf{x}-\hat{\mathbf{x}})^2 \right\rangle \quad \textit{least mean squared error} \quad \text{(LMSE)} \tag{3.33}$$

Ein wichtiger Spezialfall einer Transformation ist eine lineare Transformation.

$$\mathbf{y} = \mathbf{W}\mathbf{x} \qquad \text{bzw.} \quad y_i = \mathbf{w}_i^T \mathbf{x} \tag{3.34}$$

Jede einzelne Komponente y_i der Ausgabe $\mathbf{y}$ entspricht dabei einer Projektion des Vektors $\mathbf{x}$ auf einen Zeilenvektor $\mathbf{w}_i$ von $\mathbf{W}$. Sind die $\mathbf{w}_i$ linear unabhängig, so bilden sie eine Basis; die Zahl der Zeilen der Matrix $\mathbf{W}$ ist gleich ihrem Rang.

Das ursprüngliche Muster $\mathbf{x}$ lässt sich aus $\mathbf{y}$ nur dann wieder fehlerfrei rekonstruieren, wenn die Zahl der linear unabhängigen Basisvektoren in beiden Räumen gleich groß ist. Mit $n = \dim(\mathbf{x}) = \dim(\mathbf{y}) = m$ ist dann

$$\mathbf{x} = \mathbf{W}^{-1}\mathbf{y} \;\text{ oder }\; \mathbf{x} = \sum_{i=1}^{n} y_i \mathbf{b}_i \;\text{ mit }\; (\mathbf{b}_i^T) = \mathbf{B} = \mathbf{W}^{-1} \tag{3.35}$$

Bei orthonormalen Basisvektoren $\mathbf{w}_i$

mit
$$\mathbf{w}_i^T \mathbf{w}_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{3.36}$$

ist die Matrix $\mathbf{W}$ orthogonal und es gilt

$$\mathbf{B} = \mathbf{W}^{-1} = \mathbf{W}^T. \tag{3.37}$$

Damit lassen sich das Muster $\mathbf{x}$ und seine Rekonstruktion $\hat{\mathbf{x}}$ mit Hilfe der Basisvektoren $\mathbf{w}_i$ (Spaltenvektoren) ausdrücken

$$\mathbf{x} = \sum_{i=1}^{n} y_i \mathbf{w}_i = \sum_{i=1}^{m} y_i \mathbf{w}_i + \sum_{i=m+1}^{n} y_i \mathbf{w}_i \quad \text{und} \quad \hat{\mathbf{x}} = \sum_{i=1}^{m} y_i \mathbf{w}_i + \sum_{i=m+1}^{n} c_i \mathbf{w}_i \tag{3.38}$$

wobei das rekonstruierte Signal $\hat{\mathbf{x}}$ mit Hilfe der gespeicherten bzw. übermittelten Koeffizienten y_i und der konstanten, für alle Muster gleichen Koeffizienten c_i beschrieben wird.

Der erwartete Fehler der Rekonstruktion hängt dabei nicht nur von der Zahl m der Koeffizienten ab, sondern auch von der Wahl der Basis $\{\mathbf{w}_i\}$ und der Konstanten $\{c_i\}$. Beide wollen wir so bestimmen, dass unsere Zielfunktion in Gl. (3.33) möglichst klein wird. Beginnen wir mit den Konstanten. Das Minimum bezüglich der Wahl der Konstanten ist

$$\min_{\mathbf{c}} R(\mathbf{c}) = \min_{\mathbf{c}} \left\langle (\mathbf{x} - \hat{\mathbf{x}})^2 \right\rangle$$

In Anhang B zeigt sich, dass die besten Konstanten c_i die Erwartungswerte der Koeffizienten y_i sind

$$\langle y_i \rangle = c_i \quad \text{oder} \quad \mathbf{w}_i^T \langle \mathbf{x} \rangle = c_i \tag{3.39}$$

Wie sollten wir uns die Basisvektoren $\{\mathbf{w}_i\}$ wählen? Unsere Forderung ist

$$\min_{\mathbf{w}} R(\mathbf{w}) = \min_{\mathbf{w}} \langle (\mathbf{x} - \hat{\mathbf{x}})^2 \rangle$$

Die Lösung in Anhang B führt uns auf die Eigenvektoren $\mathbf{e}_i$ der Kovarianzmatrix $\mathbf{C}_{xx}$. In diesem Fall ist der Fehler die Summe der kleinsten Eigenwerte λ_i (s. Anhang B)

$$R(\mathbf{e}_1,..,\mathbf{e}_n) = \sum_{i=m+1}^{n} \lambda_i \qquad (3.40)$$

Um den Fehler zu minimieren ist es deshalb sinnvoll, für die vernachlässigten *(n-m)* Komponenten als Basisvektoren diejenigen Eigenvektoren zu verwenden, die die *kleinsten* Eigenwerte besitzen. Die tatsächlich zur linearen Transformation von **x** verwendeten *m* Basisvektoren $\mathbf{w}_i$ sollen also denjenigen Unterraum *(subspace)* aufspannen, der dem Raum der *m* Eigenvektoren mit den *größten* Eigenwerten entspricht.

Dabei können die neuen Basisvektoren

(a) sowohl selbst die Eigenvektoren darstellen $\mathbf{w}_i = \mathbf{e}_i$

(b) oder auch nur eine Linearkombination sein $\mathbf{w}_i = \sum_{k=1}^{n} a_{ik}\mathbf{e}_k$

Im ersten Fall (a) erfüllen die Koeffizienten y_i eine wichtige Voraussetzung für die Unabhängigkeit: sie sind *unkorreliert* (s. Gl. (3.19)) bzw. *dekorreliert.*

Datenapproximation mit Hyperebenen

Die Minimierung des quadratischen Fehlers bei der Approximation durch ein lineares System ist eng verwandt mit dem Problem, die verrauschten Funktionswerte einer unbekannten Funktion y = f(**x**) in einem Intervall durch eine Gerade (allgemein: Hyperebene) F(**x**) mit dem kleinsten quadratischen Fehler zu approximieren. Betrachten wir dazu in Abb. 3.4 einen verrauschten Messpunkt **x** der unbekannten Funktion. Mit dem üblichen Kriterium des quadratischen Fehlers wird in diesem Fall versucht, die erwartete Abweichung $\langle (F(\mathbf{x})-f(\mathbf{x}))^2 \rangle$ so klein wie möglich zu machen. Dies ist aber nicht optimal, da das Ergebnis von der Lage der Koordinatenachsen abhängt. Besser ist es, anstelle des jeweils vertikalen Abstand der Datenpunkte den erwarteten tatsächlichen Abstand der Punkte senkrecht zu der Geraden (Hyperebene) zu minimieren (*Total Least Mean Squared Error* TLMSE), der unabhängig von der Lage des Koordinatensystems ist. Dazu betrachten wir die geometrischen Verhältnisse in Abb. 3.4 genauer.

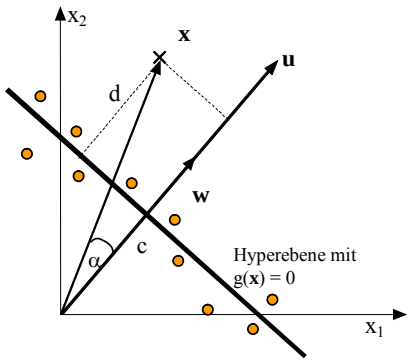


Abb. 3.4 *Lineare Approximation einer verrauschten Funktion*

Für einen beliebigen Datenpunkt $\mathbf{x} = (x_1,..,x_n,f(\mathbf{x}))^T$ ist die Projektion auf einem beliebigen Vektor **u**, der senkrecht auf der Geraden (Hyperebenen) steht, mit $\mathbf{x}^T\mathbf{u}$ gegeben. Normieren wir den Vektor **u** auf die Länge eins, so ist die Projektion $\mathbf{x}^T\mathbf{u}/|\mathbf{u}|$ auf den normierten Einheitsvektor $\mathbf{w} = \mathbf{u}/|\mathbf{u}|$ gerade die Strecke $c + d$ in der Zeichnung. Für alle **x**, die auf der Geraden (Hyperebenen) liegen, ist diese Projektion genau der Abstand c vom Nullpunkt zur Geraden. Für alle Punkte $\mathbf{x}^*$ der Geraden gilt somit die Geradengleichung

$$g(\mathbf{x}^*) = \mathbf{x}^{*T}\frac{\mathbf{u}}{|\mathbf{u}|} - c = 0 \quad \textit{Hesse'sche Normalform der Hyperebene} \tag{3.41}$$

Der Abstand d eines Datenpunktes **x** von der Geraden ist gerade derjenige Anteil der Projektion von **x** auf den normierten Abstandsvektor (Vektor der Flächennormalen) $\mathbf{w} = \mathbf{u}/|\mathbf{u}|$, der c übersteigt

$$d = \mathbf{x}^T\frac{\mathbf{u}}{|\mathbf{u}|} - c = g(\mathbf{x}) = \mathbf{x}^T\mathbf{w} - c \tag{3.42}$$

Damit lautet das Fehlerkriterium

$$\min R(\mathbf{a},b) = \min_{\mathbf{w},c}\left\langle d^2\right\rangle = \min_{\mathbf{w},c}\left\langle g(\mathbf{x})^2\right\rangle = \min_{\mathbf{w},c}\left\langle\left(\mathbf{x}^T\mathbf{w}\text{-}c\right)^2\right\rangle \quad |\mathbf{w}| = 1 \tag{3.43}$$

Das Minimum der Zielfunktion $R(\mathbf{w},c) = \langle d^2\rangle$ bezüglich des *Abstandes* b ist erreicht, wenn

$$\frac{\partial R(\mathbf{w},b)}{\partial c} = 2\langle\mathbf{x}^T\mathbf{w} - c\rangle = 0 \text{ oder } c = \langle\mathbf{x}^T\mathbf{w}\rangle \tag{3.44}$$

so dass das Ziel zu

$$\min_{\mathbf{w},c} R(\mathbf{w},c) = \min_{\mathbf{w},c} \left\langle \left(\mathbf{x}^T \mathbf{w} - c \right)^2 \right\rangle = \min_{\mathbf{w},c} \left\langle \left(\left(\mathbf{x}^T - \left\langle \mathbf{x}^T \right\rangle \right) \mathbf{w} \right)^2 \right\rangle \text{ bei } |\mathbf{w}| = 1 \quad (3.45)$$

wird. Das Minimum bezüglich der *Neigung* **w** der Hyperebene ist bei

$$\min_{\mathbf{w}} R\ (\mathbf{w}) = \min_{\mathbf{w}}\ \left\langle \mathbf{w}^T \left(\mathbf{x} - \langle \mathbf{x} \rangle \right)\ \left(\mathbf{x}^T - \left\langle \mathbf{x}^T \right\rangle \right)\ \mathbf{w} \right\rangle = \min_{\mathbf{w}} \mathbf{w}^T \mathbf{C}_{xx}\ \mathbf{w}$$

$$\text{mit der Kovarianzmatrix } \mathbf{C}_{xx} = \left(\mathbf{x} - \langle \mathbf{x} \rangle \right) \left(\mathbf{x} - \langle \mathbf{x} \rangle \right)^T \quad (3.46)$$

und der Nebenbedingung $|\mathbf{w}| = 1$ gegeben. Dieses Problem wird in Anhang B gelöst. Ein Extremum wird erreicht bei dem Eigenvektoren $\mathbf{e}_i$ der Kovarianzmatrix $\mathbf{C}_{xx}$ der Eingabemuster **x**

$$\min_{\mathbf{w}} R\ (\mathbf{w}) = \min_{\mathbf{w}} \mathbf{w}^T \mathbf{C}_{xx}\ \mathbf{w}\ = = \min_{i}\ e_i^T \lambda_i\ e_i = \min_{i} \lambda_i \quad (3.47)$$

dessen zugehöriger Eigenwert λ_i den kleinsten Wert hat.

Für die Lösung dieses Minimisierungsproblems (Parameter der Hyperebene) wählen wir also den Eigenvektor mit dem kleinsten Eigenwert, da dann als Extremum das Minimum vorliegt. Geometrisch können wir uns dies in Abb. 3.4 verdeutlichen. Würden alle Messwerte **x** auf der Geraden liegen, so hat der Eigenvektor mit dem größten Eigenwert die Richtung der Geraden und geht durch den Mittelwert auf der Geraden. Der zweite Eigenvektor ist senkrecht dazu und damit in Richtung von **w**, wie gewünscht.
Man kann auch leicht zeigen, dass es nur ein einziges Maximum und ein Minimum von (3.46) gibt (alle anderen Lösungen sind Sattelpunkte, s. z.B. [BRA92]), so dass die eindeutige Lösung des Problems durch den Eigenvektor mit dem *kleinsten* Eigenwert gegeben ist; es wird als *eigenvector line fitting* [DUD73] bezeichnet und ist im Gegensatz zum konventionellen kleinsten erwarteten quadratischen Fehler unabhängig vom Koordinatensystem. Der Vorteil dieses TLMSE Verfahrens ist gut (vgl.[XU92]) und dabei umso größer, je ungünstiger beim normalen LMSE Verfahren das Koordinatensystem der Variablen $x_1,..,x_n$ gewählt wurde.

Angenommen, wir haben die Koeffizienten **w** und c bestimmt – wie erhalten wir dann bei gegebenem x den Funktionswert der Hyperebene? Das Ergebnis für die Approximation der Funktion $f(x_1,..,x_n)$ mit einer Hyperebene $g(x_1,..,x_n) = 0$ kann mit $\mathbf{x} := (1,x_1,..,x_n,f(x_1,..,x_n))$ und $\mathbf{w} := (w_0,w_1,..,w_n,w_{n+1})$ für die Neuronenausgabe $y = S(\mathbf{x}) := g(\mathbf{x})$ geschrieben werden als

$$\langle y \rangle = \langle g(\mathbf{x}) \rangle\ = \langle \mathbf{x}^T \mathbf{w} \rangle\ = \langle w_0 + \sum_i x_i w_i + w_{n+1} f(x_1,..,x_n) \rangle = 0 \quad (3.48)$$

wobei die Koeffizienten w_1, ..., w_{n+1} durch den Eigenvektor mit kleinstem Eigenwert festgelegt sind, siehe Gl. (3.47). Wir können den zu den Variablen $x_1,..,x_n$

gehörenden, approximierten Funktionswert $F(x_1,..,x_n)$ vom Netzwerk ausgeben lassen, indem wir nur das unvollständige Tupel $\mathbf{x}' = (1, x_1, .., x_n, 0)$ eingeben, und mit Gl. (3.41) wird sich die Ausgabe

$$y = g(x') = w_0 + \sum_i x_i w_i = - w_{n+1} F(x_1,..,xn) \tag{3.49}$$

einstellen. Normieren wir die Ausgabe außerdem mit dem Faktor $-1/w_{n+1}$, so gibt das Netz direkt den unbekannten, geschätzten Funktionswert aus. Wie bei den korrelativen Assoziativspeichern kann man dies als *Ergänzung* des unvollständigen Musters $\mathbf{x}'$ ansehen und deshalb ein derartiges Netz als analogen, *kontinuierlichen Assoziativspeicher* betrachten.

3.3.2 PCA-Netze für minimalen Approximationsfehler

Wie wir sahen, sind die Probleme zur Minimierung des quadratischen Fehlers bei der linearen Reproduktion von Signalen sowie bei der linearen Approximation von Funktionen eng miteinander verknüpft. Bei beiden Problemen müssen die Eigenvektoren (oder ein Unterraum davon) ermittelt werden. Normalerweise ist dies eine Methode, die zwar für jede Art von Signalen den quadratischen Fehler minimiert, da sie datenabhängig arbeitet, aber doch umständlich und rechenintensiv ist und deshalb in der Praxis gegenüber anspruchsloseren Methoden (Kosinus- bzw. Fouriertransformation bei der Sensorkodierung, einfache quadratische Fehler bei der Approximation) zurückgestellt wird.

Mit dem Konzept des iterativen Lernens erhalten wir nun die Möglichkeit, Netze direkt die benötigte PCA-Transformation lernen zu lassen. Wie könnte eine Lernregel dafür lauten? Eine Möglichkeit dazu würde bestehen, wenn wir die Hebb'sche Lernregel und die Beschränkung der Gewichte durch Normierung zu einer einzigen Lernregel zusammenfassen würden. Betrachten wir dazu Gl. (3.13) und setzen die Hebb-Lernregel Gl. (3.1) darin ein, so erhalten wir

$$\hat{w}_i(t) = \frac{w_i(t-1) + \gamma x_i y}{|\mathbf{w}(t)|} \tag{3.50}$$

Entwickeln wir die Funktion $f(\gamma) := 1/|\mathbf{w}(t)| = [\Sigma_i (w_i(t-1)+\gamma(t) x_i y)^2]^{-1/2}$ in einer Taylorreihe

$$f(\gamma) = f(0) + \gamma f'(0) + \tfrac{1}{2} \gamma^2 f''(0) + ... \tag{3.51}$$

so wird mit Beachtung von $|\mathbf{w}(t-1)| = 1$ sowie $y = \mathbf{w}^T\mathbf{x}$

$$f(0) = [\sum_i (w_i)^2]^{-1/2} = 1$$

$$f'(\gamma=0) = -\tfrac{1}{2}\,[\textstyle\sum_i (w_i+\gamma x_i y)^2]^{-3/2}\,[\sum_i 2(w_i+\gamma x_i y)]\,x_i y$$

$$= -\tfrac{1}{2}\,[\textstyle\sum_i (w_i)^2]^{-3/2}\,\sum_i 2 w_i x_i y = -y^2$$

und unter Vernachlässigung der Summanden mit dem Faktor $\gamma^2(t)$ und höheren Potenzen von γ die Gleichung (3.50) zu

$$\hat{w}_i(t) = [w_i(t-1) + \gamma(t)\, x_i y]\;\; f(\gamma) \approx [w_i(t-1) + \gamma(t)\, x_i y]\,(1-\gamma(t) y^2) \qquad (3.52)$$

$$= w_i(t-1) + \gamma(t)\, x_i y - w_i(t-1)\gamma(t) y^2 - \gamma^2(t) y^3 x_i$$

Vernachlässigen wir wieder das Glied mit γ^2, so erhalten wir eine Lernregel, die nicht nur die Hebb'sche Regel beinhaltet, sondern auch eine Normierung der Gewichte durchführt:

$$\mathbf{w}(t) = \mathbf{w}(t-1) + \gamma(t)\, y(\mathbf{x}(t) - \mathbf{w}(t-1)y) \qquad \textit{Oja Lernregel} \qquad (3.53)$$

Diese Lernregel wurde 1982 von Oja entwickelt und gilt für alle Eingaben mit $\langle \mathbf{x} \rangle = 0$. Eine Verallgemeinerung auf höhere Synapsen ist in [TAC93] zu finden.

In Gl. (3.53) konvergiert der Gewichtsvektor durch die Hebb-Regel zu einem Extremum, hierbei dem Eigenvektor mit dem maximalen Eigenwert. Entsprechend lernt ein Neuron mit einem Gradientenabstieg, d.h. einer negativen Hebb-Regel (*Anti-Hebb-Regel*) bei normierten Gewichten, den Eigenvektor mit dem kleinsten Eigenwert für das Minimum der Varianz. Für unser Problem der Funktionsanpassung mit TLMSE bedeutet dies für den stochastischen Fall die Oja-Lernregel für den Abstieg

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma(t)\, y(\mathbf{x}(t) - \mathbf{w}(t-1)y) \qquad (3.54)$$

Bisher haben wir Lernregeln kennen gelernt, mit der wir den Eigenvektor mit dem größten bzw. kleinsten Eigenwert iterativ lernen können. Für eine erfolgreiche Minimierung des quadratischen Fehlers benötigen wir aber nicht nur einen, sondern möglichst alle Eigenvektoren. Natürlich kann man dazu ein Netz von parallel arbeitenden Neuronen aufsetzen, um die Eigenvektoren lernen lassen. Allerdings muss man dabei eine Kommunikation in Form eines Signalaustauschs zwischen den Neuronen vorsehen, um zu verhindern, dass alle Neuronen den selben Eigenvektor lernen. In den folgenden Abschnitten werden verschiedene Systeme mit ihren Lernregeln vorgestellt, um die Gewichte für eine vollständige Eigenvektorzerlegung zu erreichen. Trotz verschiedener Lernregeln ist die eigentliche lineare Aktivität (3.34) auf der Basis der iterierten Gewichtsvektoren immer gleich, wobei die Zeilen von $\mathbf{W}$ aus den neuen, gelernten Basisvektoren bestehen.

Dabei wollen wir die Lernregeln in zwei Gruppen unterteilen: in der einen Gruppe werden die neuen Basisvektoren derart gelernt, dass sie orthogonal sind und den gesuchten Unterraum aus den Eigenvektoren aufspannen, in der anderen

Gruppe stellen die neuen Basisvektoren die Eigenvektoren selbst dar und minimieren damit nicht nur den Fehler einer Reproduktion, sondern implementieren auch eine Hauptkomponentenanalyse. Die Auswahl eines der Lernverfahren hängt dabei von den Einsatzbedingungen des betrachteten Problems ab.

3.3.3 Der PCA-Unterraum

Der einfachste Ansatz zur vollständigen Eigenvektorzerlegung besteht darin, die im vorigen Abschnitt entwickelte Methode von Oja, den Eigenvektor mit dem größten Eigenwert von einem Neuron lernen zu lassen, auf mehrere Neuronen und damit mehrere Eigenvektoren auszudehnen.
Die Lernregel für ein Neuron war bei linearer Ausgabefunktion y = S(z) = z

$$\mathbf{w}(t) = \mathbf{w}(t\text{-}1) + \gamma(t)\, y\, [\mathbf{x}(t) - \mathbf{w}(t\text{-}1)y] \quad \textit{Oja Lernregel} \tag{3.55}$$

wobei die Normung $|\mathbf{w}|^2 = 1$ der Gewichte durch den Korrekturterm in der Klammer

$$\mathbf{w}(t\text{-}1)y =: \mathbf{x}^- \tag{3.56}$$

erreicht wurde. Haben wir nun *m* Neuronen, die alle die gleiche Eingabe **x** erhalten, so ist die Lerngleichung für das i-te Neuron

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma(t)\, y_i[\mathbf{x} - \mathbf{x}^-] \tag{3.57}$$

Setzen wir als negativen, inhibitorischen Korrekturterm x_j^- für die Eingabe x_j nicht nur den Beitrag $w_{ij}y_i$ des eigenen Neurons wie in Gl. (3.55) an, sondern die Überlagerung aller anderen dazu [OJA89], so wird ein Neuron *i* gehindert, für eine Komponente x_j bei starken Gewichten w_{kj} der anderen Neuronen k sein eigenes Gewicht w_{ij} auszubilden. Ist also bereits ein Neuron besonders "empfindlich" für eine Eingabekomponente, so macht es alle anderen Neuronen darin "unempfindlich". Formal bedeutet dies, das Korrekturglied $\mathbf{x}^-$ in (3.57) durch

$$x_j^- := \sum_{i=1}^{m} w_{ij} y_i \text{ oder } \mathbf{x}^- = \sum_{i=1}^{m} \mathbf{w}_i y_i = \mathbf{W}\,\mathbf{y} \tag{3.58}$$

zu ersetzen, wobei die Spalten von $\mathbf{W} = (\mathbf{w}_1, \ldots, \mathbf{w}_m)$ aus den *m* Gewichtsvektoren gebildet werden.

Die durch Plausibilitätsbetrachtungen begründete Lernregel (3.57) bzw. (3.58) kann man auch anders systematisch herleiten. Betrachten wir unser Ziel, den Reproduktionsfehler $R(\mathbf{w}_i) = \langle(\mathbf{x} - \hat{\mathbf{x}})^2\rangle$ mit einer Änderung von $\mathbf{w}_i$ zu minimieren, so ist der Lernalgorithmus mit einer Gradientensuche auf dem Risiko nach Gl. (3.42)

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) - \gamma'(t)\nabla_w R(\mathbf{w}_i) = \mathbf{w}_i(t\text{-}1) - \gamma'(t)\nabla_w\langle(\mathbf{x} - \textstyle\sum_j \mathbf{w}_j\, y_j)^2\rangle \tag{3.59}$$

$$= \mathbf{w}_i(t\text{-}1) + 2\gamma'(t)\langle(\mathbf{x}-\sum_j \mathbf{w}_j\, y_j)y_i\rangle = \mathbf{w}_i(t\text{-}1) + 2\gamma'(t)\langle y_i[\mathbf{x}-\mathbf{x}^-]\rangle$$

Mit $\gamma(t) = 2\gamma'(t)$ ist die stochastische Version davon ohne Erwartungswertklammern $\langle . \rangle$ gerade die Lernregel (3.57) bzw. (3.58).

Die Beeinflussungs-Situation ist in der Abb. 3.5 schematisch widergegeben. Nach einer linearen Beeinflussung in einer Phase werden die Aktivitäten y_i über die entsprechenden Gewichte w_{ij} auf die Eingabe"neuronen" x_j zurückpropagiert.

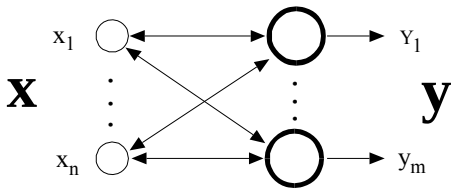


Abb. 3.5 *Die Wechselwirkungen im "subspace"-Netzwerk*

In der zweiten Phase wird dann die Summe x_j^- aller Rückflüsse von der Eingabe im nächsten Zeitschritt abgezogen. Die Neuronen haben beim Lernen in diesem Modell also eine Wechselwirkung (Inhibition) ihrer Ausgabe y_i mit der Eingabe **x** über ihre Gewichte, was allerdings durchaus nicht den Möglichkeiten unserer Modellneuronen aus Kapitel 1 entspricht und deshalb biologisch problematisch ist. Zwar ist bei der praktischen Anwendung in der Simulation auf einem Computer dieser Aspekt weniger wichtig, in VLSI-Implementationen bedeutet es aber zusätzliche, nötige Leitungen und damit Flächenbedarf.

Was ist nun das Lernziel des Systems? A. Krogh und J. Hertz zeigten in [KRO90], dass auch beim Erwartungswert von Gl. (3.57) die Spalten $\mathbf{w}_i$ der Matrix **W** zu dem Raum (*subspace*) aus den Eigenvektoren der Korrelationsmatrix $\mathbf{A} = \langle \mathbf{x}\mathbf{x}^T \rangle$ mit den größten Eigenwerten konvergieren. Wählt man allerdings für die Länge der einzelnen Gewichtsvektoren leicht unterschiedliche Werte, so konvergieren sie tatsächlich zu den Eigenvektoren und nicht nur zu irgendwelchen Basisvektoren des Unterraums [OOW92a, OOW92b]. Diesen Effekt kann man auch bei der Verallgemeinerung des Oja-Netzes auf mehrere Schichten beobachten [XU93]. Eine Anwendung des Verfahrens in der Bildverarbeitung zur Texturerkennung ist in [OJA89] beschrieben.

Damit haben wir Verfahren und Netze gefunden, den quadratischen Fehler der Reproduktion der Eingabe aus der kodierten Ausgabe zu minimieren. Ein ähnliches Netz dafür wurde auch von Williams [WILL85] entwickelt; ein symmetrisches Netz mit lateraler Inhibition, das ebenfalls den Unterraum lernt, wurde von Földiak [FÖL89] vorgeschlagen.

3.3.4 PCA und Dekorrelationsnetze

Bei den neuronalen Modellen des vorigen Abschnitts wurde der quadratische Fehler minimiert. Strebte man zusätzlich die Maximierung der Ausgabevarianz $\langle \mathbf{y}^2 \rangle$ an, so lernte das Netz als Gewichtsvektoren eine orthonormale Basis im Unterraum der Eigenvektoren von **A** mit den größten Eigenwerten. Dies ist für manche Anwendungen nicht ausreichend. Möchte man eine echte PCA durchführen, also nicht nur den Unterraum, sondern die Eigenvektoren selbst lernen, so sind die bisher vorgestellten Modelle zum Teil nicht geeignet.

Ein weiteres Problem für manche Anwendungen stellt die Tatsache dar, dass die Wechselwirkungen aller Neuronen für eine lineare Transformation völlig symmetrisch und gleichartig konstruiert wurden. Dadurch hängt es für dieses System weitgehend von den initialen Werten der Gewichte (Anfangsbedingungen) ab, welcher der Gewichtsvektoren zu welchem Basisvektor konvergiert.

Möchte man also eine echte Eigenvektorzerlegung (PCA) erhalten, um den vernachlässigten Fehler abschätzen zu können (datenabhängige Bestimmung der nötigen Zahl *m* der Ausgabekanäle), oder sollen die Ausgaben bei minimalem Rekonstruktionsfehler auch dekorreliert sein, so ist es nötig, als Gewichtsvektoren (Basisvektoren) direkt die Eigenvektoren zu lernen.

PCA mit geordneter Zerlegung

Eine Methode für eine geordnete Zerlegung besteht darin, ein asymmetrisches Netz zu verwenden, bei dem sich die Eigenvektoren nach der Größe ihrer Eigenwerte geordnet ergeben.

Ein solcher Lernalgorithmus wurde von Terence Sanger [SAN88] entwickelt. Er geht dazu von der Grundidee des Gram'schen Orthogonalisierungsverfahrens aus. Die Aufgabe besteht darin, aus einer Menge von *n* linear unabhängigen, vorgegebenen Vektoren eine orthogonale Basis, also *n* orthogonale Vektoren zu konstruieren. Dazu geht man wie folgt vor: Beginnend mit einem der vorgegebenen Vektoren errechne man einen zweiten, dazu orthogonalen Vektor. Hat man ihn, so suche man einen dritten, der zu den beiden bereits vorhandenen ebenfalls orthogonal ist, und so weiter, bis man eine zwar willkürliche, aber vollständige Basis gefunden hat.

Angenommen, wir gehen von der linearen Transformation (3.34) aus. Betrachten wir nun Gleichung (3.48). Wie man leicht nachprüfen kann, ist der Vektor ($\mathbf{x}$-$\mathbf{x}^{\sim}$) orthogonal zu $\mathbf{w}$. Subtrahieren wir also $\mathbf{x}^{\sim}$ von der Eingabe $\mathbf{x}$, so erhalten wir alle Anteile von $\mathbf{x}$, die nicht mit $\mathbf{w}_1$ dargestellt werden können. Nehmen wir diese um alle Komponenten bezüglich des ersten Gewichtsvektors reduzierte Eingabe $\tilde{\mathbf{x}}$ und bieten sie dem zweiten Neuron mit der selben Lernregel als Eingabe an, so wird das zweite Neuron von der reduzierten Eingabe ebenfalls für sich lokal als Gewichtsvektor $\mathbf{w}_2$ den Eigenvektor mit dem stärksten Eigenwert aus der reduzierten Eingabe bestimmen. Da die reduzierte Eingabe $\tilde{\mathbf{x}}$ orthogonal zu $\mathbf{w}_1$ ist, wird auch $\mathbf{w}_2$ orthogonal zu $\mathbf{w}_1$ und damit orthogonal zum Konvergenzziel von $\mathbf{w}_1$, dem

Eigenvektor $\mathbf{e}_1$. Da außerdem bei regulärem $\mathbf{C}$ mit $\mathbf{w}_1 \neq \mathbf{w}_2$ auch $\lambda_1 \neq \lambda_2$ und damit $\lambda_{max} = \lambda_1 > \lambda_2 > \lambda_i$, $i \neq 1,2$ gilt, haben wir mit $\mathbf{w}_2$ den zweiten Eigenvektor $\mathbf{e}_2$ gefunden.

Das Verfahren lässt sich iterativ fortsetzen, so dass sich als Lerngleichung für das i-te Neuron die Hebb'sche Regel

$$\mathbf{w}_i(t) = \mathbf{w}_i(t-1) + \gamma(t)\, y_i \mathbf{x}_i \qquad \textit{"Generalisierte" Hebb-Regel} \qquad (3.60)$$

mit $\mathbf{x}_{k+1} := \mathbf{x}_k + a\mathbf{w}_k(t-1)y_k$ und $\mathbf{x}_1 := \mathbf{x}$, $a := -1$, $k = 1..i-1$ (3.61)

angeben lässt.

Setzt man (3.61) in (3.60) ein und wiederholt dies für alle *k*, so erhält man einen Ausdruck mit einer Summe anstelle einer Rekursion

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma(t)\, y_i \left(\mathbf{x} - \sum_{k=1}^{i-1} \mathbf{w}_k y_k\right) \qquad (3.62)$$

In der Abb. 3.6 ist dieser sukzessive Lernprozeß *(pipeline)* der Eingabe mit einem Blockschema veranschaulicht.

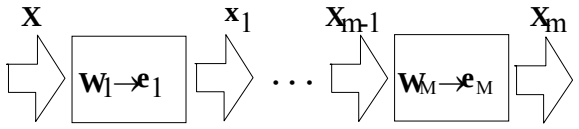


Abb. 3.6 Sequentielle Zerlegung der Eingabe

Die Formel (3.62) kann damit als sequentielle, asymmetrische Version des Oja-subspace-Lernregel (3.57) bzw.(3.58) angesehen werden. Die Aktivierung y_i wird dabei verschieden errechnet: im Lernmodus nur aus der reduzierten Eingabe $\mathbf{x}_i$, im Funktionsmodus nach dem Lernen dagegen voll parallel aus der unreduzierten Eingabe $\mathbf{x}$.

Durch die asymmetrische Formulierung in (3.62) sind die Wechselwirkungen der Gewichtsveränderungen im Netzwerk hochgradig asymmetrisch und für biologische Anwendungen weniger plausibel, um so mehr aber für die technischen Anforderungen zur Datenkompression. Sehen wir bewusst $m<n$ Neuronen im Netzwerk vor, so ist der nach der Konvergenz der $\mathbf{w}_i$ aus den Ausgabevariablen y_i gebildete Vektor $\mathbf{y}$ eine diskrete, endliche Karhunen-Loève-Entwicklung der Eingabe $\mathbf{x}$.

Ein ähnliches, sequentielles Netzwerk wurde von Brause [BRA92] vorgeschlagen, um die Eigenvektoren mit dem *kleinsten* Eigenwert *zuerst* zu lernen. Allerdings ist das Problem nicht symmetrisch: um das Minimum der Varianz zu lernen,

reicht es nicht aus, in Gl.(3.60) die negierte Hebb-Regel (Anti-Hebb-Regel) zu verwenden. Eine tiefere Analyse [BRA92] zeigt, dass in diesem Fall die Konstante *a* in Gl. (3.61) der Bedingung

$$a > (\lambda_{max}/\lambda_{min})^{1/2} - 1 \qquad (3.63)$$

gehorchen muss.

PCA durch laterale Inhibition

Bei den bisher vorgestellten Methoden wurde davon ausgegangen, dass die neuronalen Wechselwirkungen von dem Ausgangssignal y_i über die eigenen Gewichte w_{ij} und die eigenen Eingänge x_j auf das Eingangssignal x_j der anderen Neuronen erfolgen soll eine biologisch nicht sehr einleuchtende Annahme, da der Informationsfluss zum Signalfluss gerade umgekehrt ist. Eine andere Methode der geordneten Eigenvektorzerlegung, die stärkere Bezüge zu neurologischen Beobachtungen hat, wurde von Jeanne Rubner, Klaus Schulten und Paul Tavan entwickelt [RUB90].

Die Autoren nahmen an, dass das erste Neuron nach der Methode von Oja durch eine Hebb-Lernregel bei normierten Gewichten und zentriertem **x** den ersten Eigenvektor $\mathbf{e}_1$ lernt. Alle anderen Neuronen erhalten ebenfalls die gleiche, ungefilterte Eingabe **x**. Die Ausgabe dieses Neurons unterdrückt außerdem über einen zusätzlichen Eingang im zweiten Neuron alle Korrelationen mit seiner Aktivität, so dass das zweite Neuron daran gehindert wird, den ersten Eigenvektor zu lernen und stattdessen den stärksten Eigenvektor der unkorrelierten (orthogonalen) Aktivität $\tilde{\mathbf{x}}$ lernt. Führt man diesen Gedanken weiter, so erhält jedes Neuron *i* einseitig das Ausgangssignal y_k aller vorherigen Neuronen $k < i$ als Gegenkopplung über Gewichte u_{ki}. Das Gesamtnetz der Eigenvektorzerlegung ist in Abb. 3.7 gezeigt. Diese Art der Kopplung wird in der Biologie als *laterale Inhibition* bezeichnet und ist durchaus in vielen Nervensystemen anzutreffen, allerdings in vollsymmetrischer Anordnung.
Bezeichnen wir die Gewichte eines Neurons *i*, das Ausgaben anderer Neuronen *j*

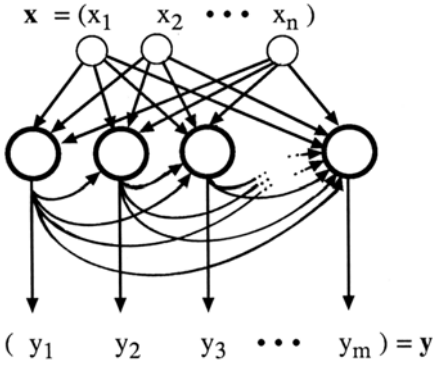


Abb. 3.7 *Geordnete Eigenvektorzerlegung durch asymmetrische laterale Inhibition*

erhält, zur Unterscheidung von w_{ij} mit u_{ij}, so ist die Aktivierungsgleichung als Summe wieder linear

$$y_i = \mathbf{w}_i^T\mathbf{x} + \sum_{k<i} u_{ik}\, y_k \;=\; (\mathbf{w}_i + \sum_{k<i} u_{ik}\mathbf{w}_k)^T\mathbf{x} \;=: \tilde{\mathbf{w}}_i^T\mathbf{x} \tag{3.64}$$

Die normalen Gewichte werden mit der Hebb'sche Lerngleichung (1.5.1) verändert

$$\Delta\mathbf{w}_i = \gamma\, \mathbf{x} y_i$$

und die Gegenkopplungsgewichte $u_{ik} = u_{ki}$ erhalten eine besondere Lernregel

$$\Delta u_{ik} = -\mu\, y_i y_k \qquad \textit{Anti-Hebb Lernregel} \tag{3.65}$$

mit der die korrelierten Aktivitätsanteile subtrahiert werden. Konvergieren die Gewichtsvektoren $\mathbf{w}_i$ zu den Eigenvektoren $\mathbf{e}_i$, so werden sie unkorreliert und die Gewichte u_{ik} konvergieren nach null.

Die Lernregel Gl.(3.65) ist nicht neu; schon Kohonen und Oja erkannten 1976 [KOH76], [KOH84] in einem anderen Zusammenhang, dass eine negative Rückkopplung der Form Gl.(3.65) eine Orthogonalisierung der gespeicherten Vektoren zur Folge hat. Die Form der Lernregel Gl.(3.65) kann man, wie H. Kühnel und P. Tavan in [KÜH90] gezeigt haben, auch aus dem Prinzip der *kleinsten gemeinsamen Information* $H(y_k,y_i)$ zweier Ausgänge y_k und y_i erhalten. Für die Minimierung der Zielfunktion

$$\begin{aligned} H(y_k,y_i) &= H_k + H_i - H_{ki} \\ &= \left\langle \ln\frac{1}{P(y_k)} \right\rangle + \left\langle \ln\frac{1}{P(y_i)} \right\rangle - \left\langle \ln\frac{1}{P(y_k, y_i)} \right\rangle \end{aligned} \tag{3.66}$$

setzten sie einen Gradientenalgorithmus an und erhielten für die stochastische Version bei einer Gauß'sche Normalverteilung als Eingabeverteilung p(**x**)

$$\Delta u_{ik} = -\gamma(t)\frac{\partial H}{\partial u_{ik}} = \frac{-\gamma(t)}{\left\langle y_k^2 \right\rangle} y_i y_k \tag{3.67}$$

so dass wir den Wert von μ in Gleichung (3.65) damit gut abschätzen können.

Eine Analyse der meisten vorgeschlagenen Algorithmen zur Eigenvektorzerlegung wurde von Hornik und Kuan [HOR92] vorgenommen. Sie fanden, dass vom Standpunkt der Konvergenz und Stabilität die asymmetrischen Netze den symmetrischen vorzuziehen seien.

Diese Analyse berücksichtigt allerdings nicht das symmetrische PCA Netz von Brause [BRA93a,b]. Mit der zu minimierenden Zielfunktion

$$R(\mathbf{w}) = ¼\sum_i \sum_j \langle y_i y_j \rangle^2 \; - ½\sum_i \langle y_i^2 \rangle \tag{3.68}$$

die nicht nur die Varianz (Autokorrelation) $\langle y_i^2 \rangle$ maximiert, sondern auch die Kreuzkorrelation $\langle y_i y_j \rangle$ minimiert, ergibt sich als stochastische Lernregel nach dem Gradientenabstieg

$$\mathbf{w}_i(t) = \mathbf{w}_i(t-1) - \gamma(t)\; \mathbf{x}\left(\beta \sum_{j \neq i} u_{ij} y_i\right) \tag{3.69}$$

bei normiertem **w**. Für die lateralen Gewichte

$$u_{ij} = -\langle y_i y_j \rangle \tag{3.70}$$

muss die Lernregel sicherstellen, dass sie den Erwartungswert der Kreuzkorrelation annehmen. Nimmt man allerdings hierzu eine Lernregel zum Lernen des Mittelwerts, etwa in der Form

$$u_{ij}(t) = u_{ij}(t\text{-}1) - 1/t\,(u_{ij}(t\text{-}1) + y_i y_j) \tag{3.71}$$

so ist die Voraussetzung einer stationären Verteilung der Zufallsvariablen $v := y_i y_j$ nicht erfüllt: die Verteilung der Variablen v ändert sich mit ändernden $\mathbf{w}_i$ und $\mathbf{w}_j$ trotz stationärer Verteilung der Eingabe **x**. In einem solchen Fall ist es besser, entweder eine konstante Lernrate γ_2 zu verwenden, die außerdem mit $\gamma_2 > \gamma$ eine schnellere Konvergenzgeschwindigkeit der lateralen Gewichte u_{ij} im Vergleich zu den direkten Gewichten w_{ij} bewirken muss, oder aber eine gleitende zeitliche Mittelung über das Zeitintervall T

$$u_{ij}(t+1) = \frac{1}{T}\sum_{k=1}^{T} y_i(t+k)\; y_j(t+k) \tag{3.72}$$

zu bilden, um stark abweichende Anfangswerte zu eliminieren.
In Abb. 3.8 ist ein lateral inhibiertes Netz nach diesem Modell zu sehen.

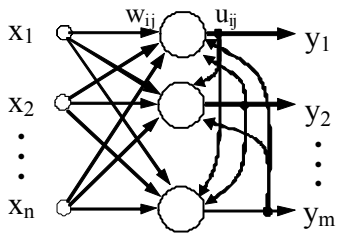


Abb. 3.8 Eigenvektorzerlegung durch symmetrische laterale Inhibition

Obwohl der Gewichtsvektor mit der Lernregel als Gradient einer beschränkten Zielfunktion nach Kapitel 1 konvergieren muss, bedeutet dies nicht, dass als Konvergenzziel verschiedene Eigenvektoren resultieren müssen. Um dies für die Differenzengleichung mit endlichen Schrittweiten zu garantieren, ist es nötig [BRA93b], dass für die Lernrate γ und das Verhältnis zwischen Kreuzkorrelation und Autokorrelation β die Relationen gelten

$$\gamma < 2/\lambda^2_{max} \qquad \beta > 2/\lambda_{min} \tag{3.73}$$

3.3.5 Störunterdrückung durch Dekorrelation und Normierung

Angenommen, zusätzlich zu der Dimensionsverringerung wie im vorigen Abschnitt wirkt sich auch noch eine Störquelle aus, die bei der Übertragung der Signale gleichmäßig auf allen Kanälen die Signale überlagert. Wie muss dann die lineare Transformation vor der Übertragung beschaffen sein, um die Signalstörung möglichst zu kompensieren? In Abb. 3.9 ist eine solche Situation gezeigt.

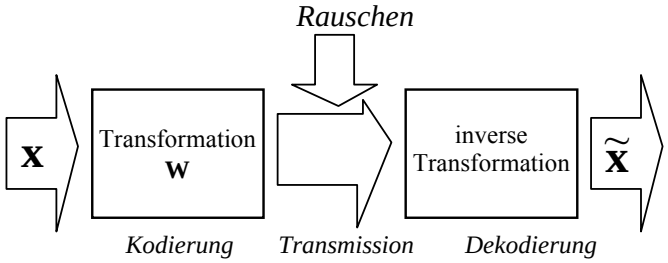


Abb. 3.9 *Kodierung zur Störungsunterdrückung*

Ein ähnliches Problem behandelte schon Shannon [SHA49b] für einen Kanal. Erlöste es, indem er das Signal einer Fouriertransformation unterwarf und das

Spektrum in unabhängige Kanäle aufteilte. Aus der Forderung, dass die durch Gauß'sches Rauschen gestörten Kanäle nach der Rekonstruktion maximale Information enthalten sollten errechnete er als Optimalitätskriterium, dass alle Frequenzkanäle vor der Störung gleiche Energie, also gleiche Varianz der Amplitude, haben sollten. Die dazu nötige Transformation W wird „Weißen" genannt; der Filter heißt „*whitening filter*". In der analogen Rauschunterdrückungstechnik (z.B. Dolby-System) hat sich dies als Anhebung der hohen Töne bei der Tonbandaufnahme und Absenken der Höhen bei der Wiedergabe durchgesetzt, um das Rauschen bei leisen Musikpassagen zu unterdrücken.
Für den Fall von vielen parallelen Kanälen und Gauß'schem Rauschen führt die Forderung nach maximaler Information ebenfalls zu der Notwendigkeit einer weißenden Transformation, bei der vor der Störung alle Kanäle y_i unkorreliert sein sollen und gleiche Varianz haben [BRR98].

$$\left\langle y_i y_j \right\rangle = 1 \text{ bei } i = j, \quad \text{und 0 sonst; also } \langle \mathbf{y}\mathbf{y}^T \rangle = \mathbf{I} \tag{3.74}$$

Für die lineare Transformation $\mathbf{y} = \mathbf{W}\mathbf{x}$ bedeutet dies

$$\mathbf{I} = \langle \mathbf{y}\mathbf{y}^T \rangle = \langle \mathbf{W}\mathbf{x}\mathbf{x}^T\mathbf{W}^T \rangle = \mathbf{W}\langle \mathbf{x}\mathbf{x}^T \rangle \mathbf{W}^T = \mathbf{W}\mathbf{A}\mathbf{W}^T$$

mit der Autokorrelationsmatrix $\mathbf{A} = \langle \mathbf{x}\mathbf{x}^T \rangle$. Eine Lösung dieses Problems ist durch orthonormale Matrizen $\mathbf{W}$ mit $\mathbf{W}^{-1} = \mathbf{W}^T$ gegeben. In diesem Fall gilt

$$\mathbf{W}^T = \mathbf{W}^T (\mathbf{W}\mathbf{A}\mathbf{W}^T) = \mathbf{A}\mathbf{W}^T$$

und damit für jede Spalte von $\mathbf{W}^T$, für jeden Basisvektor $\mathbf{w}_k$, die charakteristische Gleichung

$$\mathbf{w}_k = \mathbf{A}\mathbf{w}_k \tag{3.75}$$

Die Lösung von Gl. (3.75) sind die Eigenvektoren von $\mathbf{A}$ mit dem Eigenwert $\lambda_i = 1$. Beachten wir Gl. (B.22) aus Anhang B, so reicht es für das Weißen, eine PCA vorzunehmen und dann die Länge $|\mathbf{e}_k|$ der Basisvektoren auf $|\mathbf{e}_k|^2 = \lambda_k^{-1}$ neu zu skalieren. Die so veränderte Transformationsmatrix $\widetilde{\mathbf{W}}$ erzeugt eine dekorrelierte Ausgabe $\mathbf{y}$ mit Varianz eins.
Wirkt auf ein solches Signal eine Störung, so werden Kanäle mit kleiner Varianz nicht mehr von der Störung verdeckt, sondern lassen sich nach der Rücktransformation wieder nutzen. Die Störung wird mit der notwendigen Verkleinerung der Varianz (Verkleinerung der Amplitude) ebenfalls verkleinert.
Die Wirksamkeit diesen Mechanismus aus Dekorrelation und Varianznormierung ist in den folgenden Abbildungen am Beispiel eines Bildes gezeigt. In Abb. 3.10 ist das Originalbild „Zoe" gezeigt.
Das gesamte Bild wurde in Blöcke von 8×8 Bildpunkten (Pixeln) zerlegt. Jeder Block hat somit 64 Variablen, die im Bild einer Statistik unterworfen sind. Die PCA hat also 64 Kanäle, deren Anzahl zur Kompression verringert werden können.

Abb. 3.10 *Das Beispielbild "Zoe"*

In Abb. 3.11 ist die Rekonstruktion des Bildes zu sehen: zum einen, wenn **W** eine PCA bewirkt und danach nur die 16 stärksten Kanäle verwendet werden, und zum anderen, wenn **W** eine normalisierte PCA (NPCA) bewirkt und ebenfalls die ersten 16 Komponenten verwendet werden. Bei beiden Systemen überlagert eine Störung aus normalverteilten Abweichungen mit einer starken Varianz die erhlatenen 16 Komponenten, bevor aus diesem komprimierten Code durch Rücktransformation die Approximation des Originalbild erzeugt wird.

a) PCA Rekonstruktion b) NPCA Rekonstruktion

Abb. 3.11 *Rekonstruktionen des mit einem Gauß'schen Rauschen ($\sigma^2 = 0{,}01$) gestörten Bildes aus m = 16 Komponenten aus 64*

Man sieht, dass im Fall b) von den Störungen kein Rauschen, sondern nur die fehlenden 48 Komponenten an den viereckigen Flecken im Bild (veränderte Blockinformation) bemerkbar sind. In Abb. 3.12 ist das Gesamtverhalten des Systems bei drei verschiedenen Rauschstärken (σ^2 = 0.0, 0.001 und 0.01) und verschiedener Anzahl von erhaltenen Komponenten aufgetragen.

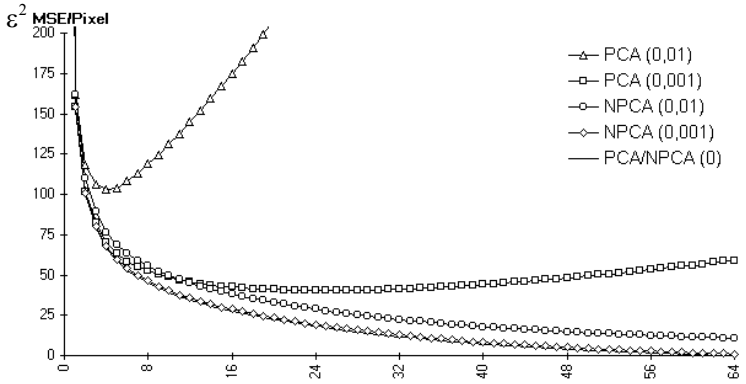


Abb. 3.12 *Der Rekonstruktionsfehler bei verschiedener Zahl von Koeffizienten (aus [BRR98])*

Man bemerkt einen interessanten Effekt: Bei der PCA fällt mit steigender Anzahl der erhaltenen Komponenten der Fehler erwartungsgemäß ab, steigt aber ab einer gewissen Anzahl wieder an. Warum? Der Gesamtfehler ε^2 setzt sich aus zwei Anteilen zusammen: aus dem Rekonstruktionsfehler durch *n–m* fehlende Komponenten und dem Rauschen auf den *m* vorhandenen Komponenten

$$\varepsilon^2 = \sum_{i=m+1}^{n} \lambda_i + m\sigma^2 \tag{3.76}$$

Da die λ_i mit dem Index *m* abfallend angeordnet sind ($\lambda_1 \gg \lambda_m$), nimmt bei wachsendem *m* der erste Term durch die sehr unterschiedlichen λ_i nicht-linear ab, während der zweite Term mit *m* linear ansteigt: Die zusätzlichen Komponenten verringern den Rekonstruktionsfehler nur wenig, aber tragen viel zum Störfehler bei, so dass der größte Fehler beim unkomprimierten, aber gestörten Bild vorhanden ist. Im Gegensatz dazu wirken sich bei der NPCA die zusätzlichen Komponenten nur positiv aus: Bei der Rücktransformation wird die Varianz mit dem λ_i jeder Komponente multipliziert, so dass der Fehler ε^2 aus dem Rekonstruktionsfehler und dem mit λ_i multiplizierten Rauschen besteht.

$$\varepsilon^2 = \sum_{i=m+1}^{n} \lambda_i + \sum_{i=1}^{m} \lambda_i \sigma^2 = \sum_{i=m+1}^{n} \lambda_i + \sigma^2 \sum_{i=1}^{m} \lambda_i$$

$$= \sum_{i=1}^{n}\lambda_i - \sum_{i=1}^{m}\lambda_i + \sigma^2\sum_{i=1}^{m}\lambda_i = P - (1-\sigma^2)\sum_{i=1}^{m}\lambda_i \qquad (3.77)$$

Ausgehend von der gesamten Energie P verringert der Fehler sich also nicht-linear mit wachsendem *m* und damit wachsendem zweiten Term in Gl.(3.75.
Damit schließen wir unseren motivierenden Einstieg in die Möglichkeiten der Störunterdrückung durch dekorrelierende und normierende Transformationen ab und stellen die Frage: können wir so etwas auch durch neuronale Netze lernen lassen ?

3.3.6 Netze zur Dekorrelation und Normierung der Daten

Möchte man die Daten mit einem symmetrischen Netz dekorrelieren und normieren, so müssen nicht unbedingt die Eigenvektoren als Basis- und damit Gewichtsvektoren resultieren, wie in Gl. (B.22) in Anhang B gezeigt wurde. Stattdessen können wir die Forderungen auch direkt lernen lassen.
Das Verfahren von Silva und Almeida [SIL91] orthogonalisiert ($\langle y_i y_j \rangle = 0$) und normalisiert ($\langle y_i^2 \rangle = 1$) die Ausgabedaten. Sie gingen von der Überlegung aus, dass es für die Orthogonalisierung (Dekorrelierung) zweier Basisvektoren $\mathbf{w}_i$ und $\mathbf{w}_j$ ausreichend ist, wenn sich die Projektion eines Basisvektors $a_{ij} := \mathbf{w}_i^T\mathbf{w}_j$ (Korrelation) auf den anderen vermindert. Dazu wird jeder Vektor um eine Komponente vermindert, der in die gleiche Richtung wie der andere Vektor zeigt, s. Abb. 3.13.

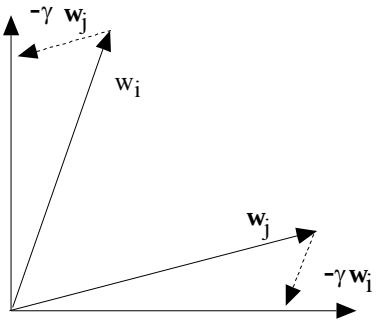


Abb. 3.13 Iterative Orthogonalisierung zweier Vektoren

Dies entspricht der folgenden Lernregel

$$\begin{aligned}\mathbf{w}_i(t) &= \mathbf{w}_i(t\text{-}1) - \gamma(t)\, a_{ij}\mathbf{w}_j(t\text{-}1) \\ &= \mathbf{w}_i(t\text{-}1) - \gamma(t)\,(\mathbf{w}_i^T(t\text{-}1)\mathbf{w}_j(t\text{-}1))\,\mathbf{w}_j(t\text{-}1)\end{aligned} \qquad (3.78)$$

Da dies für alle anderen Basisvektoren ebenfalls gilt, ist die gesamte Änderung eine Summe aller Anteile an allen anderen Richtungen

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) - \gamma(t) \sum_{j \neq i} (\mathbf{w}_i^T(t\text{-}1)\mathbf{w}_j(t\text{-}1)) \, \mathbf{w}_j(t\text{-}1)$$

Da nun nicht die Gewichtsvektoren, sondern ihre Wirkungen dekorreliert werden sollen, benutzen wir anstelle der direkten Korrelation (Skalarprodukt $\mathbf{w}_i^T\mathbf{w}_j$) ihre Korrelation $\langle y_i y_j \rangle$ im Ausgaberaum.

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) - \gamma(t) \sum_{j \neq i} (\langle y_i y_j \rangle \, \mathbf{w}_j(t\text{-}1)$$

Erweitern wir die Summe um einen Term für j = i, so müssen wir den Anteil $(1 - \langle y_i^2 \rangle)$ wieder addieren

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) - \gamma(t) \sum_{j} (\langle y_i y_j \rangle \, \mathbf{w}_j(t\text{-}1)) \; + \gamma(t)(1 - \langle y_i^2 \rangle)\mathbf{w}_i(t\text{-}1)$$

$$= \mathbf{w}_i(t\text{-}1) - \gamma(t) \, [\sum_{j} (\mathbf{w}_j \mathbf{w}_j^T \mathbf{C}) \; - \; (1\text{-}\mathbf{w}_i^T \mathbf{C} \mathbf{w}_i)\mathbf{I} \,] \, \mathbf{w}_i(t\text{-}1) \qquad (3.79)$$

Dieses Verfahren konvergiert ziemlich rasch bei geeigneten Parametern (z.B. $\gamma <$ ½) und lässt sich auch in Mehrphasen-Form ähnlich dem *subspace*-Modell mit speziellen Eingabeneuronen einsetzen.

Möchte man im Gegensatz zu dieser heuristischen, nicht aus einer Zielfunktion ableitbaren Methode eine Dekorrelation mit einem symmetrischen, lateral inhibierten Netz nach Abb. 3.8 erreichen, so kann man eine gegenüber Gl.(3.68) leicht veränderte, normierenden Zielfunktion, bei der sowohl die Korrelation $\langle y_i y_j \rangle$ als auch die Abweichung von der Norm $\langle y_i^2 \rangle - 1$ minimiert werden, aufstellen

$$R(\mathbf{w}) = 1/4 \, \beta \sum_i \sum_j \langle y_i y_j \rangle^2 + 1/4 \sum_i (\langle y_i^2 \rangle\text{-}1)^2 \qquad (3.80)$$

und daraus die Lernregel eines stochastischen Gradientenabstiegs für die Gewichte herleiten [BRR98]

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma(t)\mathbf{x}(\beta \sum_j u_{ij} y_j + y_i) \qquad (3.81)$$

Dabei wird die Länge der Gewichte durch die Normierungsforderung $\langle y_i^2 \rangle = 1$ iterativ festgelegt. Die Lernregel für die lateralen Gewichte $u_{ij} = - \langle y_i y_j \rangle$ ist analog zum nicht-normierten Fall beispielsweise mit (3.71) gegeben.

Ein anderer Ansatz, mit einem lateral inhibierten Netz eine Datenorthonormalisierung zu erreichen zeigte Plumbley in [PLUM93]. Sein Netz besitzt eine direkte, jedem Neuron exklusiv zugeordnete Eingabe; die Dekorrelation und Normalisie-

rung wird durch Lernregeln auf den lateralen Gewichten erreicht. In Abb. 3.14 ist das Netz abgebildet.

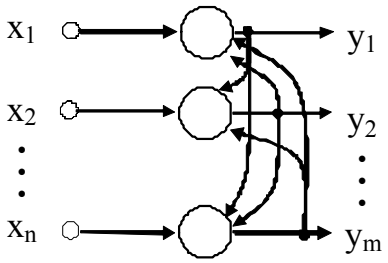


Abb. 3.14 *Ein symmetrisches, lateral inhibiertes Dekorrelationsnetz*

Für die Wechselwirkungen im Netzwerk definierte er die (positiven) lateralen Gewichte mit einer Gewichtsmatrix **U**, wobei die Aktivität mit der Vektorgleichung

$$\mathbf{y} = \mathbf{x} - \mathbf{U}\mathbf{y} \tag{3.82}$$

definiert wurde. Dies bedeutet, dass das Netzwerk eine lineare Transformation ausführt mit

$$\mathbf{y} = (\mathbf{I}+\mathbf{U})^{-1}\mathbf{x} = \mathbf{W}\,\mathbf{x}$$

Die Lernregel lässt sich aus einer Zielfunktion herleiten, bei der das Maximum der Transinformation bei fester Gesamtvarianz (Signalenergie P) gesucht wird. Sie lautet

$$u_{ij}(t+1) = u_{ij}(t) + \begin{cases} \gamma\left(y_i^2(t)\right) & i = j \\ \gamma\, y_i(t) y_j(t) & \text{sonst} \end{cases} \tag{3.83}$$

Die beiden lateral inhibierten Netze unterscheiden sich dabei hauptsächlich durch die Konvergenzziele: beim Netz von Plumbley sind die Eingangsgewichte fest mit $w_{ij} = \delta_{ij}$ fixiert, so dass die lateralen Gewichte am Ende der Iteration die Transformationskoeffizienten enthalten. Beim symmetrischen Netz von Brause und Rippl [BRR98] nach Abb. 3.8 konvergieren die lateralen Gewichte am dagegen zu null; die Transformation der Eingabe wird nur noch durch die Eingabegewichte w_{ij} durchgeführt.

3.3.7 Nichtlineare PCA

Bisher haben wir nur Problemstellungen betrachtet, die sich durch eine lineare Transformation lösen ließen. Dabei benutzten wir vorwiegend eine Hauptachsentransformation oder PCA, was einer Rotation des Koordinatensystems zu den Richtungen $\mathbf{e}_i$ größter Datenstreuung bzw. Varianz (*principal components)* bedeutet, s. Abb. 3.15(a).

Es gibt nun Datenverteilungen, für die diese lineare Vorgehensweise nicht geeignet sind. In Abb. 3.15(b)ist eine Datenverteilung gezeigt, die eine krummlinige, nichtlineare Transformation benötigen: hier ist die einfache PCA, die sich ja mit numerischen Standardmethoden weit schneller durchführen lässt als mit langsam konvergierenden neuronalen Netzen, überfordert und führt zu falschen Resultaten.

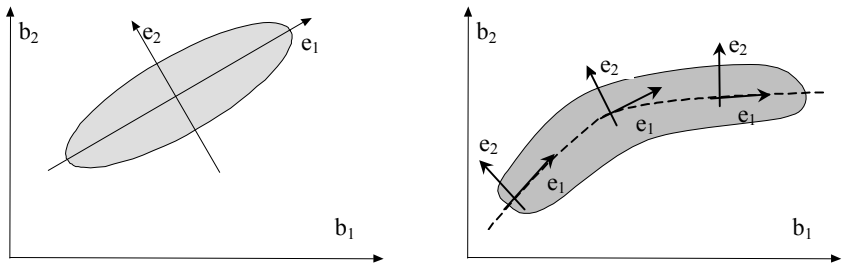


Abb. 3.15 (a) lineare PCA (b) nichtlineare PCA

Man kann verschiedene Ansätze wählen, um diesem Problem mit adaptiven Systemen gerecht zu werden. Beispielsweise kann man das Gebiet in lokale Stücke unterteilen, in denen jeweils die nötige nichtlineare Transformation linear angenähert wird. Oder man setzt die neuen Basisvektoren als Funktion der Koordinaten an.

Bisherige Versuche, das Problem anzugehen, beschränken sich allerdings darauf, anstelle der linearen Neuronen nicht-lineare Ausgabefunktionen $y = S(\mathbf{x},\mathbf{w})$ zu verwenden. Beispielsweise fügte Sanger eine nicht-lineare Schicht einem normalen PCA Netz hinzu, um eine Approximation nicht-linearer Funktionen zu erreichen [SAN91]. Oja et. al. ersetzte in [OOW91] die lineare Funktion $y = \mathbf{w}^T\mathbf{x}$ in den Lerngleichungen (3.57) und (3.58) des *subspace*-Netzes durch eine sigmoidale Funktion $y = S_F(\mathbf{w}^T\mathbf{x})$. Die neuen Basisvektoren weichen von den Eigenvektoren ab [OJA94] und ermöglichen eine bessere Trennung von verrauschten Sinus-Signalen [OOW91]. Dies wird damit erklärt, dass die sigmoidale Funktion überhöhte Daten ("Ausreißer") unterdrücken und dadurch eine bessere Analyse ermöglichen.

In [SHP92] wird ein Neuron mit nicht-linearer Ausgabefunktion $\{y = S(z) = (z)^b$ bei $z = \mathbf{w}^T\mathbf{x} > 0$, sonst $y = 0\}$ auf die Fähigkeit untersucht, mit Hilfe der Lern-

gleichung $\Delta\mathbf{w} = \langle y(\mathbf{x}\text{-}z\mathbf{w})\rangle_x$ und dem Training zweier verschiedener Muster (hier: Buchstaben aus 64x64 Pixeln) diese *unüberwacht* zwischen unabhängigen Eingaben zu unterscheiden und den Gewichtsvektor dabei auf ein Muster zu spezialisieren. Es ergibt sich, dass dazu $\mathbf{x}_1^T\mathbf{x}_2 < (b\text{-}1)/(b\text{+}1)$ erfüllt sein muss. Die Nichtlinearität ist also essentiell für eine Unterscheidungsfähigkeit zwischen korrelierten Mustern: ein lineares Neuron führt nur zu einer PCA; in diesem Fall zu einem Gewichtsvektor, bestehend aus einer Mischung der Muster.

In [PALMI94] wird die Idee verfolgt, ein lateral inhibiertes Netz mit einer sigmoidalen Ausgabefunktion einzusetzen. Aus den Zielfunktionen für minimalen Fehler folgt die Veränderung der feedforward-Gewichte des *i*-ten Neurons über die Hebb-Lernregel

$$\Delta w_{ij} = \gamma\,(x_{ij} - \bar{x}_{ij})y_i \qquad \bar{x}_{i\cdot} := \sum_j u_{ij}y_j \tag{3.84}$$

und die der lateralen Gewichte u_{ij} über die Anti-Hebb Lernregel (3.65).
Auch ein direkter Ansatz mit der Zielfunktion

$$R(\mathbf{w}) = \sum_i \langle g[\mathbf{x}\text{-}\mathbf{w}_i y_i]\rangle, \qquad y_i = S(\mathbf{w}^T\mathbf{x}) \tag{3.85}$$

mit beispielsweise $g(\mathbf{z}) = \mathbf{z}^2$ (quadrat. Fehler) führt zu Lernregeln, die solch eine nicht-linearen Abweichung der neuen Basisvektoren von den Eigenvektoren bewirken [KAJ94].

3.4 Abhängigkeitsanalyse ICA

Angenommen, wir haben eine lineare Mischung von Signalen, beispielsweise die Überlagerung der Stimmen zweier Sprecher, die von zwei Mikrofonen aufgefangen werden. In Abb. 3.16 ist eine solche Situation gezeigt.

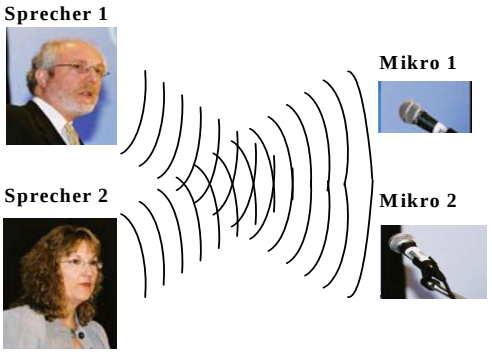


Abb. 3.16 *Lineare Überlagerung zweier Signale*

Wir erhalten also bei jedem Mikrofon M als Überlagerung das Mischsignal x aus unabhängigen Quellen (Sprechern) s_1 und s_2, wobei die Anteile mit einem Gewicht a_i versehen sind, etwa mit dem Faktor der geringeren Lautstärke durch einen größeren Abstand zwischen Mikrofon und Sprecher. Also können wir dies modellieren mit

$$x_1 = a_{11}s_1 + a_{12}s_2$$
$$x_2 = a_{21}s_1 + a_{22}s_2 \tag{3.86}$$

Fassen wir die beiden beobachteten Variablen der Mischung x_i zu dem Vektor $\mathbf{x} = (x_1,x_2)$ und die Quellen zu $\mathbf{s}$ zusammen, so entspricht dies der linearen Mischung

$$\mathbf{x} = \mathbf{As} \tag{3.87}$$

Gesucht ist die Matrix $\mathbf{W}$, die es ermöglicht, die ungestörten Quellen aus der Mischung zu extrahieren. Dies ist zweifelsohne eine Matrix mit der Eigenschaft

$$\mathbf{s} = \mathbf{Wx} \text{ oder } \mathbf{W} = \mathbf{A}^{-1} \tag{3.88}$$

Wie finden wir diese Matrix? Bevor wir uns daran machen, iterative Lerngleichungen des linearen neuronalen Netzes für diese Aufgabe zu formulieren, sollten wir uns vorher klarmachen, unter welchen Umständen diese Aufgabe überhaupt lösbar ist. Einziges Kriterium, ob wir die Aufgabe erfüllt haben oder nicht, kann nur sein, ob die neu gewonnen Quellen auch unabhängig sind oder nicht. „Unabhängigkeit“ bedeutet dabei „stochastische Unabhängigkeit“, also die Proportion der Dichtefunktionen

$$p(s_1, s_2, ..., s_n) = p(s_1)\cdot p(s_2)\cdot ... \cdot p(s_n) \tag{3.89}$$

Daraus folgen einige Eigenschaften, die zu folgenden einschränkende Feststellungen führen:

1. Wir können *keine eindeutige Reihenfolge* der Quellen bestimmen.
   Haben wir ein $\mathbf{y}$ mit $\mathbf{y} = \mathbf{Wx}$ gefunden, bei dem $p(y_1y_2...y_n) = p(y_1)\cdot p(y_2)\cdot ... \cdot p(y_n)$ gilt, so wissen wir nicht, ob $y_1 = s_1$ und $y_2 = s_2$ gilt; es kann auch $y_1 = s_2$ und $y_2 = s_1$ sein. Allgemein können wir nur eine Matrix $\mathbf{W}$ finden, die bis auf eine Permutation $\mathbf{P}$ der Indizes der gesuchten Matrix $\mathbf{A}^{-1}$ entspricht: $\mathbf{s} = \mathbf{Py}$ oder $\mathbf{A}^{-1} = \mathbf{PW}$.

2. Wir können die *Varianz* der Quellen *nicht bestimmen*.
   Angenommen, die zentrierte Quelle s_1 habe die Varianz $\langle s^2\rangle$. Dann hat der Term $v = a_{11}s_1$ aus $x_1 = a_{11}s_1 + a_{12}s_2$ die Varianz $\langle v^2\rangle = \langle a_{11}^2 s_1^2\rangle = a_{11}^2 \langle s_1^2\rangle$. Allerdings kann die Varianz auch von einem Koeffizienten $b^2 = a_{11}^2 \langle s_1^2\rangle$ und einer Quelle der Varianz eins herrühren oder von einem Koeffizienten $c^2 = b^2/4$ und einem Signal der Varianz vier: Aus der beobachteten Varianz können wir nicht eindeutig auf die Varianz der Quelle schließen. Man umgeht deshalb dieses Problem dadurch, dass man annimmt, dass die Quel-

len die Varianz $\sigma^2 = 1$ haben und alle Matrixkoeffizienten für diese Annahme bestimmt werden. Per Definition muss also für das Ergebnis **y** des Entmischens $\langle \mathbf{y}\mathbf{y}^T \rangle = \mathbf{I}$, die Einheitsmatrix mit allen Hauptdiagonaleinträgen gleich eins, gelten.

3. Wir können zwei Quellen mit Normalverteilung *nicht* voneinander trennen. Dieses Problem folgt aus einer ganz besonderen Eigenschaft der Normalverteilung: Nach der Dekorrelation zweier Mischungen sind die erhaltenen Variablen nicht nur dekorreliert, sondern auch unabhängig. Dies können wir leicht bei einer *n*-dimensionalen Normalverteilung zeigen.

Mit $p(x_1,...,x_n) = A\, e^{-\mathbf{x}^T\mathbf{C}^{-1}\mathbf{x}}$, $\qquad A = \dfrac{1}{\sqrt{(2\pi)^n \left|\det \mathbf{C}\right|}}$

gilt bei dekorrelierten Komponenten von **x** die *Kovarianzmatrix*

$$\mathbf{C} = \langle (\mathbf{x}-\mathbf{x}_0)(\mathbf{x}-\mathbf{x}_0)^T \rangle = \begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_n^2 \end{bmatrix}, \quad \mathbf{C}^{-1} = \begin{bmatrix} \sigma_1^{-2} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_n^{-2} \end{bmatrix}$$

und so mit $|\det \mathbf{C}| = \prod_{i=1}^{n} \sigma_i^2$ ist $A = \prod_{i=1}^{n} \dfrac{1}{\sigma_i \sqrt{(2\pi)}}$,

$$e^{-\mathbf{x}^T\mathbf{C}^{-1}\mathbf{x}} = e^{-\frac{x_1^2}{\sigma_1^2} \cdots -\frac{x_n^2}{\sigma_n^2}} = \prod_{i=1}^{n} e^{-\frac{x_i^2}{\sigma_i^2}}$$

und wir erhalten die Beziehung

$$p(x_1,...,x_n) = A\, e^{-\mathbf{x}^T\mathbf{C}^{-1}\mathbf{x}} = \prod_{i=1}^{n} \frac{1}{\sigma_i \sqrt{(2\pi)}} \prod_{i=1}^{n} e^{-\frac{x_i^2}{\sigma_i^2}}$$

$$= \prod_{i=1}^{n} A_i\, e^{-\frac{x_i^2}{\sigma_i^2}} = p(x_1)\cdots p(x_n) \qquad (3.90)$$

das heißt, die Zufallsvariablen x_i sind unabhängig. Normieren wir noch zusätzlich die Varianz auf eins, so erhalten wir bei jeder Mischung von Normalverteilung nach der Dekorrelation und Normierung wieder eine multidimensionale Normalverteilung. Diese hat in allen Richtungen die gleiche Varianz und ist in allen Richtungen dekorreliert: Es gibt keine eindeutigen

Hauptachsen mehr, jede Lage eines orthogonalen Basissystems hat unabhängige Zufallsvariablen. Damit können wir unsere ursprüngliche Mischung nicht mehr entmischen.

Eine Dekorrelation ist im allgemeinen aber nicht identisch mit der Unabhängigkeit der beteiligten Variablen; die Ergebnisse sind anders. Abgesehen von dem formalen Unterschied lässt sich dies inhaltlich auch visualisieren. Betrachten wir dazu den Unterschied der Transformationen zwischen einer PCA (Dekorrelation) und einer ICA (Unabhängigkeitsanalyse) Abb. 3.17 am Beispiel einer rautenförmigen, schraffiert gezeichneten uniformen Verteilung

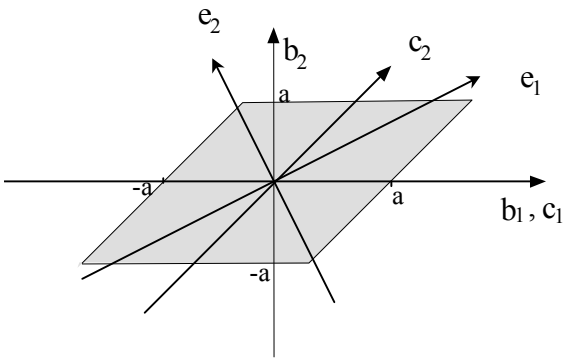


Abb. 3.17 Lineare Transformation auf Hauptkomponenten {**e**} und unabhängige Komponenten {**c**}

von Daten. Sie besteht aus der Verteilungsmenge { $\mathbf{x} = (x_1,x_2) \mid x_1 = \eta+x_2,\ x_2 = \eta'$}, die im Raum der Basis $\mathbf{b}_1$und $\mathbf{b}_2$ gezeigt ist. Die Zufallsvariablen η und η' mit der uniformen Dichte $p = 1/(4a^2)$ sind jeweils aus dem Intervall [-a,a] gezogen. Die PCA ermittelt als Richtungen größter Varianz die Diagonalen $\mathbf{e}_1$ und $\mathbf{e}_2$ in der Raute und dekorreliert damit die Variablen x_1 und x_2; die ICA aber ermittelt davon abweichend die Richtungen $\mathbf{c}_1$ und $\mathbf{c}_2$ beider unabhängigen Variablen, die im Unterschied zur PCA nicht senkrecht aufeinander stehen. Die in Abschnitt 3.3.7 erwähnten Verfahren der nicht-linaren PCA erzeugen keine ICA, s. [KAJ94]; es werden stattdessen besondere ICA-Verfahren benötigt.

Die Verfahren der ICA sind inzwischen zahlreich [HKO1]. Unter den Verfahren gibt es zwei Hauptrichtungen: Zum einen Verfahren, bei denen die Transinformation zwischen den Kanälen minimiert wird und zum anderen Verfahren, die Extremwerte einer statistischen Größe, der Kurtosis, suchen. Aus beiden Ansätzen soll in den nächsten Abschnitten jeweils eine Arbeit vorgestellt werden.

3.4.1 ICA durch maximale Transinformation

Eine wichtige Größe für die Beschreibung von Signalen ist ihre statistische Ausprägung, gemessen in Größen der Signal-Auftrittswahrscheinlichkeiten. Dabei betrachten wir als Zufallsvariable $X = \{x_k\}$ die Menge aller diskretisierten Messwerte. Haben wir nur endliche viele, diskrete Messwerte, so können wir jedem Messwert eine Auftrittswahrscheinlichkeit $P(x_k)$ zuordnen. Dies gilt auch für eine Verbundvariable $X = \{\mathbf{x}_i \mid \mathbf{x}_i = (x_1,...,x_n)\}$ aus mehreren, zu $\mathbf{x}_i$ zusammengefassten Messwerten. Die Information *H* einer solchen Messwertverteilung X ist nach Anhang A.1

$$H(X) = -\sum_i P(\mathbf{x}_i) \ln P(\mathbf{x}_i) \tag{3.91}$$

Die Information zwischen der Eingabe **X** und der Ausgabe **Y**, die Transinformation $H(\mathbf{Y};\mathbf{X})$, ist nach Anhang A.4

$$H(Y;X) = H(Y) - H(Y|X) \qquad \textit{Transinformation} \tag{3.92}$$

Wir stellen nun an die transformierende Schicht die Forderung, bei der Transformation die Transinformation durch die Schicht zu maximieren. Wann ist das der Fall?
In Anhang A wird gezeigt, dass die Transinformation im wesentlichen durch den Term H(Y) bestimmt wird, da wir ein deterministisches System haben und der Term H(Y|X) deshalb konstant ist. Unser Ziel, die Transinformation von der Eingabe zur Ausgabe zu maximieren, wird also durch Maximierung der Zielfunktion

$$R(\mathbf{w}) := H(Y)$$

erreicht. Dies ist nach Anhang A.3

$$R(\mathbf{w}) = H(X) + \int_{-\infty}^{+\infty} p(\mathbf{x}) \ln |\det \mathbf{J}| d\mathbf{x} \tag{3.93}$$

Der einfache Gradientenaufstieg zum Optimieren der Gewichte der Ausgabeschicht ist im eindimensionalen Fall

$$w(t+1) = w(t) + \gamma \frac{\partial R(w)}{\partial w} = w(t) + \int_{-\infty}^{+\infty} p(x) \frac{\partial}{\partial w} \ln \left|\det \frac{\partial y}{\partial x}\right| dx \tag{3.94}$$

Angenommen, wir haben einen hyperbolischen Tangens als Ausgabefunktion $y = S_T(z) = \tanh(z)$, so ist mit $z = wx + w_0$

$$\frac{\partial}{\partial w} \ln \left|\det \frac{\partial y}{\partial x}\right| = \left[\left|\det \frac{\partial y}{\partial x}\right|\right]^{-1} \frac{\partial}{\partial w} \left|\det \frac{\partial y}{\partial x}\right| = \left[\frac{\partial y}{\partial x}\right]^{-1} \frac{\partial}{\partial w} \frac{\partial y}{\partial x}$$

wobei $\frac{\partial y}{\partial x} = (1 - \tanh^2(z))w$, $\frac{\partial}{\partial w}\frac{\partial y}{\partial x} = (1 - \tanh^2(z)) + (0 - \frac{\partial}{\partial w}\tanh 2(z))\cdot w$

$= (1/w - x\cdot 2\cdot\tanh(z))\cdot(1 - \tanh^2(z))\cdot w$

Damit wird

$$\left[\frac{\partial y}{\partial x}\right]^{-1}\frac{\partial}{\partial w}\frac{\partial y}{\partial x} = \frac{1}{w} - 2\cdot x\cdot\tanh(z)$$

und die Lernregel Gl.(3.94) lautet

$$w(t+1) = w(t) + \gamma(\frac{1}{w} - 2\cdot y\cdot x) \tag{3.95}$$

Die Lernregel zeigt einen Anti-Hebb-Term, der einem reines Anwachsen von w entgegenwirkt. Im multidimensionalen Fall lautet die Gleichung dann [Bell95]

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \gamma(\left[\mathbf{W}^T\right]^{-1} - 2\mathbf{y}\mathbf{x}^T) \tag{3.96}$$

Kombinieren wir sie mit dem Ansatz eines „natürlichen Gradientenaufstiegs“ von Amari [AMA85], der als Veränderung der Gewichtsmatrix vorschlug

$$\frac{d\mathbf{W}}{dt} = \Delta\mathbf{W} = \gamma(t)\,\frac{\partial R(\mathbf{W})}{\partial \mathbf{W}}\mathbf{W}^T\mathbf{W} \tag{3.97}$$

so erhalten wir die Lernregel

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \gamma(\mathbf{I} - 2\mathbf{y}\mathbf{x}^T\mathbf{W}^T)\mathbf{W} = \mathbf{W}(t) + \gamma(\mathbf{I} - 2\mathbf{y}\mathbf{z}^T)\mathbf{W} \tag{3.98}$$

Eine ähnliche Lernregel kann man auch durch eine Entwicklung der Dichtefunktionen nach ihren Momenten erreichen [AMA96].
Man beachte, dass die Herleitung der Lerngleichung (3.98) auf der Tangens-Hyperbolicus-Funktion der Ausgabe aufbaut. Nehmen wir eine andere Funktion an, etwa die Fermifunktion oder die Gaußsche Fehlerfunktion, so resultiert auch eine andere Lerngleichung.

3.4.2 ICA durch Extremwerte der Kurtosis

Im vorigen Abschnitt wurde erwähnt, dass man die Quellenseparierung durch die Unabhängigkeit aller Ausgaben erreichen kann. Eine Möglichkeit dazu besteht darin, nicht-lineare Ausgabefunktionen zu verwenden, die implizit höhere Momente der Wahrscheinlichkeitsdichte erzeugen und diese zur Trennung der Quellen verwenden. In diesem Abschnitt sehen wir uns die Möglichkeit an, eine Quellentrennung durch die explizite Verwendung höherer Momente zu erreichen.
Um welche Momente geht es? Zur Unterscheidung verschiedener Wahrscheinlichkeitsverteilungen werden in der Statistik Zahlen definiert, die „typische“ Ei-

genschaften von Verteilungen wiedergeben, die Momente. Man kann sie als (unvollständige) Kurzbeschreibungen der Verteilungen auffassen und folgendermaßen formal definieren:

k-tes Moment: $m_k := \langle x^k \rangle$

k-tes Zentralmoment : $mz_k = \langle (x-m_1)^k \rangle$

In der folgenden Tabelle sind die wichtigsten Momente einer Zufallsvariablen x gezeigt.

Definition	Bezeichnung
$m_1 = \langle x \rangle = \bar{x} = \Sigma\, P(x)\, x$	Erwartungswert
$mz_2 = \sigma^2 = \langle (x-\bar{x})^2 \rangle$	Varianz
$mz_3 = \langle (x-\bar{x})^3 \rangle$	Schiefe (*Skew*)
$mz_4 = \langle (x-\bar{x})^4 \rangle$	Exzeß, Wölbung (*Kurtosis*)
$K(x) = (\langle (x-\bar{x})^4 \rangle - 3\sigma^4) / \sigma^4$	relative Kurtosis

Die relative Kurtosis ist so definiert, dass sie bei der Normalverteilung $N(\bar{x},\sigma)$ null wird. Bei allen Verteilungen, die spitzer sind als die Normalverteilung („supergaussisch"), ist sie positiv, etwa bei Musiksignalen. Bei allen Verteilungen mit geringer Wölbung, etwa die zweigipflige Verteilung von reinen Sinussignalen, ist sie negativ. In Abb. 3.18 ist dies visualisiert.

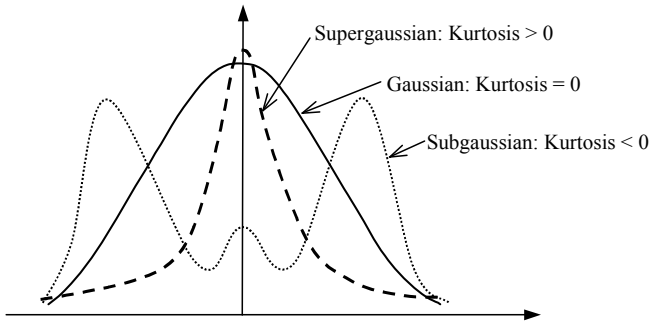


Abb. 3.18 *Arten von Verteilungsdichten und ihre Kurtosis*

Es lässt sich zeigen, dass die Kurtosis als Kriterium ausreicht, um unabhängige Quellen unterscheidbar zu machen.Wie lässt sich die Aussage praktisch verwenden? Dazu unterwerfen wir die Mischungen zunächst einigen Vorverarbeitungsstufen, in denen sie für das Trennen vorbereitet werden.

Bei der Vorverarbeitung werden meist zwei Stufen unterschieden: das Zentrieren der Daten (Abziehen des Mittelwerts) und das Weissen, also das Dekorrelieren der Kanäle und die Skalierung auf eine Varianz von eins. In Abb. 3.19 ist eine solche Stufenverarbeitung gezeigt.

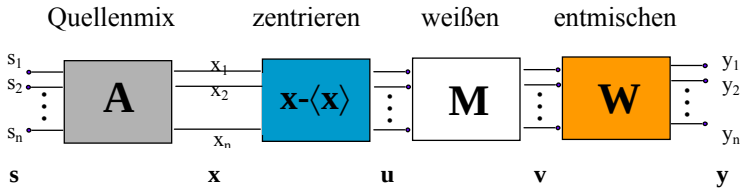


Abb. 3.19 *Die Stufen der Vorverarbeitung*

Die Bildung des Mittelwertes lässt sich auch online durchführen durch eine Iteration der Art

$$\mathbf{x}_0(t+1) = \mathbf{x}_0(t) - \gamma\,(\mathbf{x}_0-\mathbf{x}),\ \ \gamma = 1/t \quad \text{so dass } \mathbf{x}_0 \rightarrow \langle \mathbf{x} \rangle \tag{3.99}$$

Die Dekorrelation und Normierung auf Varianz eins kann man durch eines der Orthonormalisierungsverfahren aus Abschnitt 3.3.5 erreichen. Üblicherweise wird dies durch eine PCA mit einem numerischen Standardverfahren erreicht, so dass die $\mathbf{b}_i$ die Eigenvektoren der Matrix $\mathbf{C} = \langle \mathbf{u}\mathbf{u}^T \rangle$ mit $|\mathbf{b}_i| = 1$ und Eigenwerten λ_i bilden, und anschließender Normierung der Eigenvektoren auf die Länge $\mathbf{b}_i/\lambda_i^{1/2}$. Dies führt bei zu einer normierten Ausgabevarianz $\langle v^2 \rangle = \mathbf{b}_i^T \langle \mathbf{u}\mathbf{u}^T \rangle \mathbf{b}_i = \mathbf{b}_i^T \lambda_i \mathbf{b}_i = 1$. Also ist der Erwartungswert des äußeren Produkts der Vektoren $\mathbf{z}$ gleich der Einheitsmatrix $\langle \mathbf{v}\mathbf{v}^T \rangle = \mathbf{I}$. Mit unserer Forderung $\langle \mathbf{y}\mathbf{y}^T \rangle = \mathbf{I}$ und dem Zusammenhang $\mathbf{y} = \mathbf{W}\mathbf{v}$ ergibt sich also für $\mathbf{W}$

$$\langle \mathbf{y}\mathbf{y}^T \rangle_v = \langle \mathbf{W}\mathbf{v}\mathbf{v}^T\mathbf{W}^T \rangle_v = \mathbf{W} \langle \mathbf{v}\mathbf{v}^T \rangle_v \mathbf{W}^T = \mathbf{W}\mathbf{W}^T = \mathbf{I} \tag{3.100}$$

Also ist $\mathbf{W}$ eine orthonormale Matrix, deren Inverse mit der Transponierten übereinstimmt. Die anschließende Entmischung hat nur noch zur Aufgabe, eine orthonormale Matrix $\mathbf{W}$ zu finden, die die Quellen trennt.

Der ICA-Fixpunktalgorithmus

Dazu beschränken wir uns zuerst auf die Aufgabe, die Komponente zu finden, die am meisten von der Kurtosis null der Gaußfunktion abweicht, also positiv maximal oder minimal im Negativen ist. Haben wir sie gefunden, so können wir uns auf die nächste Komponente konzentrieren, die unabhängig davon ist, und so weiter, bis alle Komponenten bestimmt sind. Als iteratives Verfahren jeder Komponente verwenden wir einen Fixpunktalgorithmus, wie er in Anhang C beschrieben ist.

Dazu wählen wir uns als Zielfunktion die relative Kurtosis der Komponente. Angenommen, wir wollen das Minimum der relativen, zentrierten Kurtosis K bei Varianz eins

$$K(\mathbf{w}) = \langle(\mathbf{w}^T\mathbf{v})^4\rangle - 3\,\langle\,(\mathbf{w}^T\mathbf{v})^2\rangle^2 \qquad (3.101)$$

der beobachteten Signale $\mathbf{y} = \mathbf{w}^T\mathbf{v}$ bestimmen. Der Gradient dieser Zielfunktion ist

$$\text{grad } K(\mathbf{w}) = \langle 4(\mathbf{w}^T\mathbf{v})^3\mathbf{v}\rangle - 3\cdot 2\langle(\mathbf{w}^T\mathbf{v})^2\rangle\cdot 2\cdot\langle(\mathbf{w}^T\mathbf{v})\,\mathbf{v}\rangle \qquad (3.102)$$

Da wir geweißte Daten mit $\langle\mathbf{v}\mathbf{v}^T\rangle = \mathbf{I}$ verwenden, so erhalten wir

$$\langle(\mathbf{w}^T\mathbf{v})^2\rangle = \langle(\mathbf{w}^T\mathbf{v})(\mathbf{v}^T\mathbf{w})\rangle = \langle\mathbf{w}^T(\mathbf{v}\,\mathbf{v}^T)\mathbf{w}\rangle = \mathbf{w}^T\langle\mathbf{v}\,\mathbf{v}^T\rangle\mathbf{w} = |\mathbf{w}|^2$$

und $$\langle(\mathbf{w}^T\mathbf{v})\,\mathbf{v}\rangle = \cdot\langle\mathbf{v}(\mathbf{v}^T\mathbf{w})\rangle = \langle(\mathbf{v}\,\mathbf{v}^T)\mathbf{w}\rangle = \langle\mathbf{v}\mathbf{v}^T\rangle\,\mathbf{w} = \mathbf{I}\mathbf{w} = \mathbf{w}$$

so dass wir erhalten

$$\text{grad } K(\mathbf{w}) = 4\,(\langle(\mathbf{w}^T\mathbf{v})^3\mathbf{v}\rangle - 3|\mathbf{w}|^2\,\mathbf{w}) \qquad (3.103)$$

Für den Fixpunkt gilt grad $K(\mathbf{w}^*) = 0$ und bei normiertem $|\mathbf{w}| = 1$ ist

$$\langle(\mathbf{w}^{*T}\mathbf{v})^3\mathbf{v}\rangle - 3\mathbf{w}^* \stackrel{!}{=} 0 \quad \text{oder} \quad \mathbf{w}^* = \langle(\mathbf{w}^{*T}\mathbf{v})^3\mathbf{v}\rangle - 2\mathbf{w}^*$$

Dieser Fixpunkt ist auch bei der Iterationsvorschrift

$$\mathbf{w}_{t+1} = g(\mathbf{w}_t), \quad |\mathbf{w}_t|=1 \qquad \text{mit } g(\mathbf{w}_t) := \langle(\mathbf{w}_t^T\mathbf{v})^3\mathbf{v}\rangle - 2\,\mathbf{w}_t \qquad (3.104)$$

vorhanden. Damit haben wir eine Iterationsvorschrift gefunden. Ein ähnlicher Fixpunktalgorithmus stammt von Hyvärinen [HK01]

$$\mathbf{w}_{t+1} = g(\mathbf{w}_t), \quad |\mathbf{w}_t|=1 \qquad \text{mit } g(\mathbf{w}_t) := \langle(\mathbf{w}_t^T\mathbf{v})^3\mathbf{v}\rangle - 3\,\mathbf{w}_t \qquad (3.105)$$

Konvergenz des Fixpunktalgorithmus

Frage: Konvergiert der so erhaltene Fixpunktalgorithmus, und wenn ja, unter welchen Umständen ?

Um dies zu untersuchen, formen wir ihn etwas um mit Hilfe der als unabhängig und zentriert vorausgesetzten Quellen **s**. Sei die Mischung der Quellen mittels der Matrix **A** und die Weißung mit **M** erfolgt, so ist $\mathbf{v} = \mathbf{M}\cdot\mathbf{A}\cdot\mathbf{s}$. Dies lässt sich auch durch $\mathbf{v} = \mathbf{B}\mathbf{s}$ schreiben mit $\mathbf{B} = \mathbf{M}\mathbf{A}$ wobei **B** orthonormal ist: $\mathbf{B}^T\mathbf{B} = \mathbf{B}\mathbf{B}^T = \mathbf{I}$. Wir suchen dabei durch Iteration die Matrix **W** derart zu finden, dass die Ausgabe $\mathbf{y} = \mathbf{W}\mathbf{v}$ zu den Quellen konvergiert.

Angenommen, wir definieren uns spezielle Parametervektoren **z** derart, dass $\mathbf{z} = \mathbf{B}^T\mathbf{w}$. Bei eindeutigem **B** gibt es zu jedem Zustand von **w** auch ein entsprechendes **z**. Es reicht also, anstelle der Konvergenz von **w** die Konvergenz der entsprechen-

den Fixpunktgleichung für **z** zu betrachten. Multiplizieren wir unsere Fixpunktgleichung (3.104) mit $\mathbf{B}^T$, so erhalten wir

$$\mathbf{z} = \mathbf{B}^T\mathbf{w}_{t+1} = \langle(\mathbf{w}_t{}^T\mathbf{B}\mathbf{B}^T\mathbf{v})^3\mathbf{B}^T\mathbf{v}\rangle - 2\,\mathbf{B}^T\mathbf{w}_t = \langle(\mathbf{z}^T\mathbf{s})^3\mathbf{s}\rangle - 2\,\mathbf{z}_t \tag{3.106}$$

Die k-te Komponente (ohne Iterationsindex) davon ist

$$z_k = \sum_l\sum_m\sum_n \left\langle (z_{kl}s_l)(z_{km}s_m)(z_{kn}s_n)s_k \right\rangle - 2\,z_k$$

$$= \sum_l\sum_m\sum_n z_{kl}z_{km}z_{kn}\left\langle s_m s_l s_n s_k \right\rangle - 2\,z_k \tag{3.107}$$

Über die Quellen s_k wissen wir, dass sie unabhängig voneinander sind mit Varianz eins, also $\langle s_j\rangle = 0$ und $\langle s_i^2\rangle = 1$ gelten. Dies bedeutet, dass für den Term $\langle s_l s_m s_n s_k\rangle$ folgendes gilt:

- Sind alle vier Indizes gleich, so ist $\langle s_k^4\rangle$ ungleich null.
- Sind drei Indizes gleich und einer nicht, so lässt sich der Term abspalten und mit $\langle s_j\rangle = 0$ ist der ganze Ausdruck null.
- Sind zwei Indizes gleich und die restlichen zwei nicht, so fällt der Term ebenfalls weg.
- Sind zwei Indizes gleich und die restlichen zwei ebenfalls, so ergibt sich eine Einfachsumme $\sum_{l\neq k} z_l^2 z_k \left\langle s_l^2 s_k^2 \right\rangle$, wobei $\langle s_l{}^2\, s_k{}^2\rangle = \langle s_l{}^2\rangle\,\langle s_k{}^2\rangle = 1$ ist. Dies gilt für alle drei Indizes l=m,n.
- Sind alle Indizes ungleich, so ist der Ausdruck null.

Damit wird Gl. (3.107) zu

$$z_k = z_k{}^3\langle s_k{}^4\rangle + 3\sum_{l\neq k} z_l^2 z_k \; - 2\,z_k \tag{3.108}$$

Da **z** durch

$$|\mathbf{z}|^2 = \sum_l z_l^2 = \mathbf{w}^T\mathbf{B}\mathbf{B}^T\mathbf{w} = \mathbf{w}^T\mathbf{I}\mathbf{w} = |\mathbf{w}|^2 = 1$$

ebenfalls normiert ist, erhalten wir

$$(1\text{-}z_k{}^2) = \sum_{l\neq k} z_l^2$$

und damit

$$z_k \; = z_k{}^3\langle s_k{}^4\rangle + 3z_k(1\text{-}z_k{}^2) - 2\,z_k = z_k{}^3\,(\langle s_k{}^4\rangle - 3) + z_k$$

$$= z_k^3 \, K(s_k) + z_k = g(z_k) \tag{3.109}$$

Wir sehen, dass jede Komponente für sich iteriert wird, unabhängig von den anderen. Unser Kriterium für einen stabilen Fixpunkt aus Anhang C wird damit zu

$$|g'(z)| = |3z^2 \, K(s_k) + 1| < 1 \tag{3.110}$$

Ist dies immer gegeben? Bei positiver Kurtosis trifft dies nicht zu: Der Fixpunktalgorithmus konvergiert nicht. Der Algorithmus konvergiert also nur für ausreichend negative Kurtosis, etwa bei Sinus-Signalen. Verwenden wir dagegen den Algorithmus Gl. (3.105), so ergibt sich anstelle von Gl. (3.110) die Bedingung

$$|g'(z)| = |3z^2 \, K(s_k)| < 1 \tag{3.111}$$

die bei kleinem z bei positiver und negativer Kurtosis erfüllt ist.

3.4.3 Anwendungen der ICA

Die Trennung vermischter Signale lässt sich in vielen Bereichen einsetzen. Der Bereich der akustischen Signalentmischung, etwa der Separierung von Sprache in einer gestörten Umgebung, ist nicht so einfach, wie man denken könnte: Akustische Reflexionen an Wänden und Gegenständen führen zu Überlagerungen (Echos), die zu der Mischung verschiedener, verzögerter Signale führt. Ein Entmischung ohne Elemente der Zeitverzögerung ist deshalb nicht mehr möglich. Es gibt aufwändige Verfahren, die dem auch Rechnung tragen [TOR96a,b], aber hier nicht weiter behandelt werden sollen. Stattdessen wollen wir als Beispiel die Entmischung von EEG-Signalen betrachten.

Bei der Ableitung der EEG-Spannungen am Kopf tritt normalerweise ein Problem auf: Die gemessenen Spannungen im Millivolt-Bereich sind eine Überlagerung der Aktivität verschiedener Gehirnteile. Leider überlagert sich dazu noch die wesentlich kräftigeren Signale der Muskelaktivitäten, etwa der Augenmuskeln, der Gesichtsmuskeln und der Halsmuskulatur. Hier zeigt eine ICA gute Erfolge. Auch wenn die Herkunft und Bedeutung der ICA-Quellen beim EEG unklar ist, kann man die Kenntnis der Störkanäle dazu nutzen, die EEG-Signale aller anderen Kanäle zu entstören. Dazu werden nur alle als Muskelartefakte identifizierten Kanäle null gesetzt und dann die Ausgabe $\mathbf{y}$ mit der inversen Matrix $\mathbf{W}^{-1}$ zurücktransformiert. Es ergeben sich die ursprünglich aufgezeichneten EEG-Signale, aber ohne Muskelartefakte. In Abb. 3.20 ist links das EEG der gestörten und rechts das der ungestörten Signale zu sehen. Die korrelierten Aktivitätsmuster mehrerer Kanäle sind nun deutlich besser erkennbar.

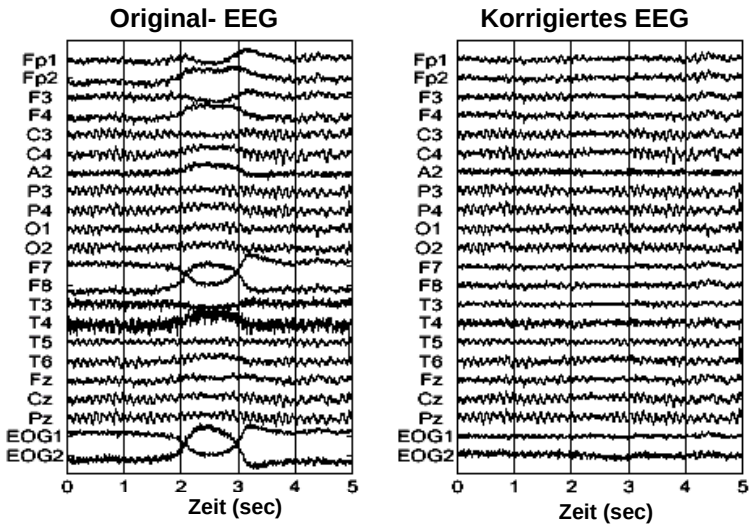


Abb. 3.20 *Gestörte und entstörte EEG-Signale (nach [JUN98])*

In Abb. 3.21 ist das Resultat einer solchen Entmischung gezeigt. Die vier Kanäle mit Muskelartefakten sind extra nach ihrer Herkunft aufgeführt.

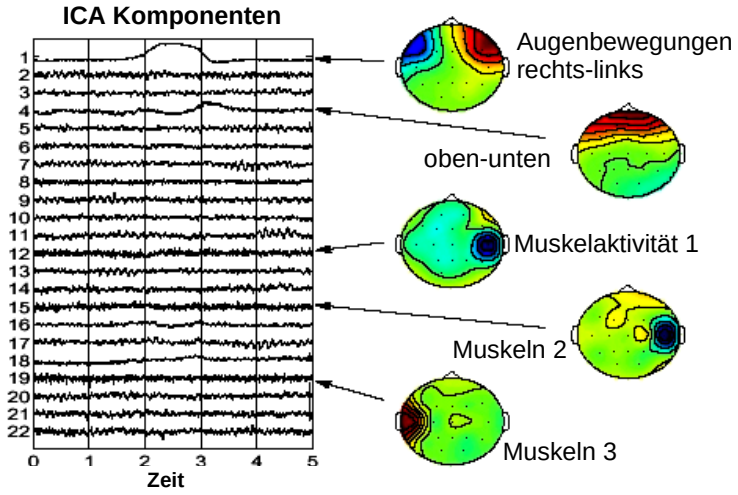


Abb. 3.21 *Vom EEG entmischte Muskelsignale(nach [JUN98])*

Aufgaben

1) Sei ein lineares Neuron mit $y = \mathbf{w}^T\mathbf{x}$ gegeben. Zeigen Sie, dass der durch $\mathbf{x}' := \mathbf{x} - y\mathbf{w}$ definierte Eingabevektor senkrecht auf $\mathbf{w}$ steht, wenn $\mathbf{w}$ auf eins normiert ist.

2) Man zeige, dass für den quadratischen Fehler bei orthonormaler Basis, linearen Neuronen und zentrierter Eingabe die Gleichung gilt

$$R(\mathbf{w}) = \langle(\mathbf{x} - \hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2 - 2(\mathbf{x}^T\hat{\mathbf{x}}) + (\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2\rangle - \langle(\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2\rangle - \langle(\hat{\mathbf{y}})^2\rangle$$

Der Fehler der Eingabeapproximierung entspricht dem Fehler der Ausgabe.

3) Man simuliere ein PCA Netzwerk nach Wahl für n = 2.

Als Eingabe verwende man Zufallsvariable einer 2-dim. Gaußverteilung, die man sich generiert, und drehe die so entstehende Verteilung mit einer linearen Transformation um einen Winkel.

Das PCA Netzwerk wird damit trainiert. Wie ist die Lage der so gefundenen Eigenvektoren und warum? Welche Werte kann man für die Lernrate γ verwenden?

4 Konkurrentes Lernen

In den vorigen Kapiteln betrachteten wir Netze formaler Neuronen, die parallel zueinander ohne Wechselwirkung funktionierten. Zwar war manchmal beim Lernen eine Koordination oder Wechselwirkung nötig, aber bei der Erzeugung der Aktivität kamen meist keine Wechselwirkungen zum Tragen.
In diesem Kapitel wollen wir nun die Effekte betrachten, die bei der Kopplung der Aktivität der Neuronen zusätzlich auftreten und für das Ergebnis essentiell sind. Eines der wichtigsten besteht darin, die Zuständigkeiten für die Eingabedaten unter den Neuronen aufzuteilen durch eine Klassifizierungsentscheidung. Dies bedeutet eine Extrahierung wichtiger Kategorien aus den Eingabedaten.

4.1 Klassifikation und Vektorquantisierung

Viele Merkmale, an denen man Ereignisse oder Objekte erkennen kann, sind nicht exakt, sondern eher unscharf definiert: trotz starker Schwankung der Merkmalsvariablen erkennen wir als Objekt beispielsweise einen "Tisch" und nicht einen "Hocker". Trotzdem gibt es aber eine Grenze in den Proportionen, ab der das Objekt eher als "Hocker" angesehen wird. Im Prinzip bilden wir aus einer Menge von Objekten eine Kategorie "Tisch", die beispielsweise im Merkmalsraum durch Fußhöhe und Plattengröße von einer Kategorie "Hocker" getrennt ist. Dieser Vorgang lässt sich als *Klassifikation* von Merkmalstupeln ansehen, wie wir sie in Kapitel 2 kennen gelernt haben. Die Ausbildung eines Einzelmerkmals (z.B. "Lehne existiert") kann selbst wieder Gegenstand einer Klassifikationsoperation sein kann.
Die klassischen Algorithmen zur Klassifikation (s. z.B. [DUD73]) werden durch die Algorithmen der neuronalen Netze ergänzt, die besonders bei unüberwachter, sequentiellem, adaptivem Aufbau einer Klassifizierung die Schwächen der traditionellen Ansätze überwinden und deshalb interessant werden. Im Unterschied zur *probabilistischen Klassifikation*, die besonders bei der Kenntnis der Verteilungen der Muster in den Klassen erfolgreich sind, haben wir hier meist kein Vorwissen. Das System muss die Klassifikation selbst erarbeiten. Ein Überblick über klassische und neuronale Klassifikationsarten und Methoden ist in [LIP89] zu finden.

In diesem Kapitel wollen wir uns näher mit adaptiven, selbstorganisierenden Ansätzen zu der Problematik der Klassifikation beschäftigen. Dazu führen wir zunächst die Grundbegriffe der Klassifikation von kontinuierlichen Merkmalen und Signalen, der Vektorquantisierung, ein.

Für die zuverlässige Übertragung und Speicherung von Signalen war der Vorgang der Klassifikation oder *Quantisierung* für eine Unterdrückung von Störungen sehr wichtig und wurde für den eindimensionalen Fall genau untersucht. Ein wichtiges Beispiel dafür ist die Analog/Digital Wandlung, bei der kontinuierliche Signalwerte *x* durch diskrete Zahlen w_i beschrieben werden. In Abb. 4.1 ist eine solche Zuordnung $w_i = Q(x)$ gezeigt.

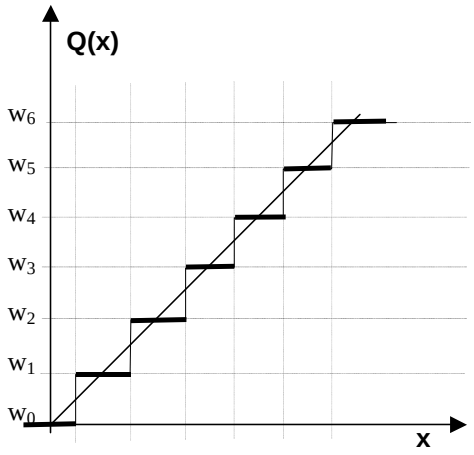


Abb. 4.1 *Quantisierung einer reellen Variablen*

Die Variable *x* wird dazu in Intervalle unterteilt (*quantisiert*), wobei alle Werte von *x* in einem Intervall einem Pegelwert $w_i = Q(x)$ zugewiesen werden, bezeichnet mit einem Symbol d_i. Eine Quantisierung ordnet also wie eine Klassifikation einer Menge von Punkten jeweils einem Symbol d_i (*Codewort*) zu. Zusätzlich wird ein Punkt w_i auserkoren, der im mehrdimensionalen Fall als *Codebuchvektor* $\mathbf{w}_i$ wie ein Klassenprototyp die ganze Klasse repräsentiert. In der Signalkodierungstheorie wird die Menge $\{d_i\}$ als *Codebuch* bezeichnet. Im mehrdimensionalen Fall von Vektoren **x** heißt die Klassifizierung hier *Vektorquantisierung*.

Sind alle Quantisierungsintervalle gleich groß, so spricht man von *uniformer Quantisierung*. Dies ist gewöhnlich der Fall, wenn die Variable x uniform verteilt ist, also $p(x) = \text{const}$. Für den kleinsten erwarteten Fehler $\langle (w_i-x)^2 \rangle$ der Quantisierung ist es in diesem

Fall auch am günstigsten, den Codebuchvektor w_i so zu wählen, dass der Abstand zu beiden Intervallgrenzen gleich ist.
Bei einer nicht-uniformen Eingabeverteilung ist es sinnvoll, die Intervallgrößen und die Lage der x_i für eine *nicht-uniforme Quantisierung* zu ändern. In diesem Fall gilt für das kleinste Fehlerquadrat nach [JAY84]

- die Prototypen x_i sind die Schwerpunkte (*centroids*) der $p(\mathbf{x})$ im Intervall der Klasse
- die Klassengrenzen sind in der Mitte zwischen den Prototypen x_i
- für die Intervalldichte (Klassendichte M(x)) gilt im 1-dim Fall $M(x) \sim p(x)^{1/3}$
- die Varianzen aller Intervalle (Fehlerabweichungen) sind gleich

Allgemein wird jeder Ereigniswert $\mathbf{x}$ in die Klasse ω_i so eingeordnet, dass das Fehlerquadrat $(\mathbf{x}\text{-}\mathbf{w}_i)^2$ besonders klein ist. Alle Punkte $\{\mathbf{x}\}$, für die dies gilt, formen eine Teilmenge Ω_i des Eingaberaums mit

$$\Omega_i = \{\mathbf{x} \mid |\mathbf{x}\text{-}\mathbf{w}_i| \leq |\mathbf{x}\text{-}\mathbf{w}_j| \ \forall j \ \} \ \textit{Voronoi-Teilung} \tag{4.1}$$

Die Gleichheit beim Abstandsvergleich gilt dabei für alle Punkte auf der Grenze zur Klasse ω_j, so dass die Punkte der Grenze gemeinsam für beide benachbarten Klassen *i* und *j* sind. Die Grenze bildet eine Gerade, die in halbem Abstand die Verbindungslinie zwischen den Klassenprototypen $\mathbf{w}_i$ und $\mathbf{w}_j$ senkrecht schneidet. Die Relation aus Gl.(4.1) teilt dabei den gesamten Eingaberaum durch Gerade in einzelne, nicht-überlappende Teilmengen ein. Die Art der Aufteilung kann dabei sehr unterschiedlich sein, siehe die Formen in Abb. 4.2. Diese Art von Aufteilung heißt *Voronoi-Teilung* (*Voronoi tesselation*). Für den 2-dim. Fall ist bekannt, dass die sechseckige Aufteilung den geringsten Fehler verursacht; die günstigste Form für den n-dim. Fall ist unbekannt [MAK85].

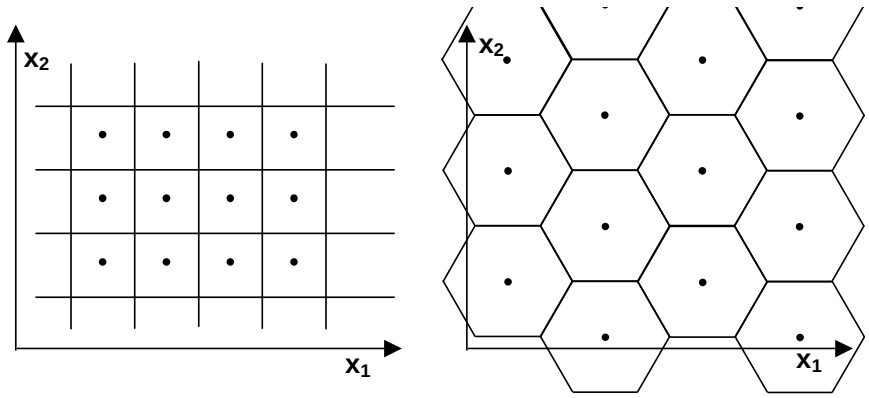


Abb. 4.2 *Formen möglicher Aufteilungen*

Eine solche Aufteilung ist als Klassifizierung immer dann sinnvoll, wenn alle Muster einer Klasse im Eingaberaum dicht beieinander liegen.
Wir haben den Abruf von gespeicherten Mustern im Assoziativspeicher auf die Problematik der Klassifikation eines Musters **x** zurückgeführt. Da für eine lineare Trennung die Klassenmengen Ω_i mindestens disjunkt sein müssen, teilen wir den Musterraum aller möglichen Eingaben $\Omega = \{\mathbf{x}\}$ in disjunkte Teilmengen

$$\Omega_i \in \Omega \quad \text{mit } \Omega = \bigcup_i \Omega_i \ , \ \bigcap_i \Omega_i = \varnothing \tag{4.2}$$

ein. In jeder Teilmenge Ω_i gibt es genau einen Klassenprototypen: das gespeicherte Muster $\mathbf{x}^i$.
Was für eine Form hat die Klassengrenze und was ist ihr Abstand zum Klassenprototypen bei der Voronoi-Teilung? Mit der Relation von Gl. (4.1) folgt folgende Klassifikationsregel für die Einordnung des Musters **x**: Wähle diejenige Klasse ω_r, für deren Klassenprototyp $\mathbf{x}^r$ gilt

$$\left|\mathbf{x} - \mathbf{x}^r\right| = \min_k \left|\mathbf{x} - \mathbf{x}^k\right| \tag{4.3}$$

Damit wird ein Klassifikationsschema und damit eine Aufteilung des Musterraums in Klassen nach dem Kriterium des kleinsten Abstands definiert. In Abb. 4.3 ist dies an zwei Klassen *r* und *k* verdeutlicht.

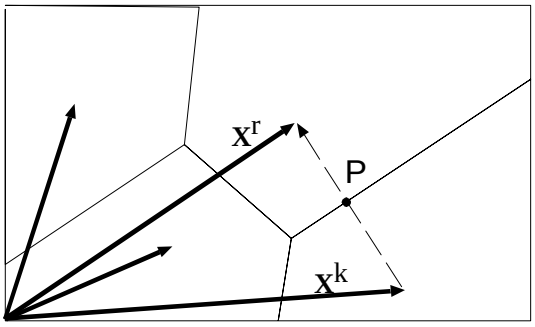


Abb. 4.3 *Aufteilung des Musterraums bei minimalem Abstand*

Für alle Punkte **x*** der Grenzfläche gilt

$$(\mathbf{x}^* - \mathbf{x}^r)^2 = (\mathbf{x}^* - \mathbf{x}^k)^2 \Leftrightarrow (\mathbf{x}^r)^2 - (\mathbf{x}^k)^2 + 2(\mathbf{x}^k - \mathbf{x}^r)^T \mathbf{x}^* = 0 \tag{4.4}$$

oder $(\mathbf{x}^k - \mathbf{x}^r)^T \mathbf{x}^* - c = 0$ mit $c := ((\mathbf{x}^k)^2 - (\mathbf{x}^r)^2)/2$

Mit dem Differenzvektor $\mathbf{d}_{rk} := \mathbf{x}^r - \mathbf{x}^k$ hat die Gleichung die Form

$$\mathbf{d}_{rk}^{T}\mathbf{x}^{*} - c = 0 \text{ oder } \mathbf{n}_{rk}^{T}\mathbf{x}^{*} - s = 0 \text{ mit } \mathbf{n} = \mathbf{d}/|\mathbf{d}| \text{ und } s = c/|\mathbf{d}| \tag{4.5}$$

was als „Hesseschen Normalform der Ebene“ bekannt ist. Damit ist gezeigt, dass die Trennfläche eine Hyperebene ist, die den Nullpunkt im Abstand *s* passiert. Legen wir den Nullpunkt genau auf den Schnittpunkt *P* zwischen der Verbindungsgerade $\mathbf{d}_{rk}$ und der Hyperebene, so ist mit s = 0 auch c = 0 und wir sehen, dass wegen $\mathbf{d}_{rk}^{T}\mathbf{x}^{*} = 0$ die Verbindungsgerade und die Hyperebene immer einen rechten Winkel bilden müssen. Für den Abstand der Klassenprototypen $\mathbf{x}^{k}$ bzw. $\mathbf{x}^{r}$ von P gilt hier mit Gl. (4.4) $(\mathbf{x}^{r})^2 = (\mathbf{x}^{k})^2$ oder $|\mathbf{x}^{r}| = |\mathbf{x}^{k}|$. Also schneidet die Trennebene die Verbindungsgerade bei halbem Abstand $\mathbf{d}_{rk}/2$.

4.2 Competitive Learning

Im vorigen Kapitel sahen wir, dass durch die Wechselwirkung der Einzelneuronen wichtige Funktionen der Mustererkennung und Merkmalsgewinnung, wie beispielsweise eine Hauptachsentransformation, verwirklicht werden können. Eine der bekanntesten, lokalen Wechselwirkungen bestand dabei in einer gegenseitigen Hemmung (*laterale Inhibition*) der formalen Neuronen. Die Hemmung wirkt sich dabei so aus, dass von allen Neuronen nur dasjenige Neuron mit der größten Aktivität ausgewählt wird und die maximale Ausgabe $y_i = 1$ annimmt (*winner-take-all Netzwerk*); die Ausgabe alle anderen bleibt null. Dies entspricht einer *Klassifizierung* oder Vektorquantisierung aus dem letzten Abschnitt: Eine Menge von Eingabemustern wird auf einen Klassenprototypen (Neuronengewichtsvektor) abgebildet. Die Auswahlregel lässt sich relativ einfach *global* bei einer Simulation auf einem sequentiellen Computer durchführen; als *lokale* Entscheidung ist sie durch unabhängige Neuronen nicht so einfach zu implementieren.

Ist das "Lernen" (Verändern der Gewichte) abhängig von der Größe der eigenen Aktivität und derjenigen der anderen Neuronen, so spricht man von "Wettbewerbs-Lernen" (*competitive learning*).

Grundmechanismen

Sei eine Menge *(Cluster)* von formalen Neuronen gegeben, die binäre Ein- und Ausgabe besitzen. Aus allen Neuronen des Clusters wird dasjenige herausgesucht, das die größte Aktivität z_r hat

$$z_r = \max_i z_i \qquad \text{mit } z_i = \mathbf{w}^{T}\mathbf{x} \qquad \textit{Auswahlregel} \tag{4.6}$$

wobei die Ausgabefunktion lautet

$$y_i = S(z_i) := \begin{cases} 1 & r = i \\ 0 & r \neq i \end{cases} \qquad \textit{winner-take-all} \qquad (4.7)$$

Mit einer auf eins normierten Ausgabefunktion

$$y_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \qquad \textit{soft winner-take-all} \qquad (4.8)$$

und einer Zielfunktion aus der subjektiven Information lässt sich auch eine "weiche" winner-take-all Klassifizierung durchführen, beispielsweise in einem Backpropagation-Netzwerk [BRI90a].

Aber zurück zu unserem einfachen Modell. Man kann nun mehrere Cluster nebeneinander auf derselben Eingabe **x** arbeiten lassen und das Ergebnis aller Ausgänge aller Cluster zu einem Ausgabevektor **y** zusammenfassen. Die Ausgabeinformation **y**, die soviel Einsen hat wie es Cluster auf dieser Ebene gibt, kann im nächsten Schritt von einer weiteren Cluster-Gruppe verarbeitet werden und so fort. Verringert man bei jedem Verarbeitungsschritt die Zahl der Neuronen pro Cluster und die Zahl der Cluster pro Ebene bis auf eins, so erhält man schließlich eine diskrete Entscheidung über das präsentierte Muster **x** *(Klassifizierung)*.
Eine solche Situation ist in Abb. 4.4 gezeigt, wobei in runden Kreisen der Zustand y_i jeder Einheit als Schwärzung aufgetragen ist. In jedem Cluster wird ein Gewinner gesucht, so dass in jedem Cluster nur für ein Neuron $y_i = 1$ gilt.

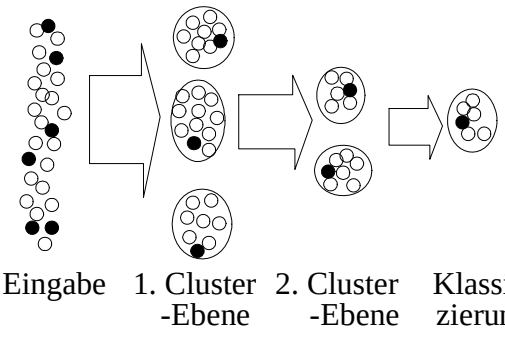


Abb. 4.4 *Cluster-Entscheidungsfolge bei Competitive Learning*

Bisher hatten wir die Cluster willkürlich bestimmt. Man kann aber auch auf jeder Stufe alle Neuronen mit ähnlicher Aktivität *z* bei der Eingabe eines **x** zu einem Cluster zusammenfassen. In diesem Fall entspricht dies der Vektorquantisierung, wobei der Gewichtsvektor des Gewinners dem Klassenprototypen entspricht.

Die Regeln zur Gewichtsveränderungen resultieren aus der Forderung, den Abstand zum Eingabevektor zu verringern und dabei die Gewichte normiert zu halten:

Aus $|\mathbf{x}\text{-}\mathbf{w}| \rightarrow 0$, d.h. $\Delta\mathbf{w} \sim (\mathbf{x}\text{-}\mathbf{w})$ bzw. $\Delta\mathbf{w}_i = \gamma(c\mathbf{x}\text{-}\mathbf{w}_i)$

und $$\sum_j w_{ij} = 1 \Rightarrow \sum_j \Delta w_{ij} = \sum_j w_{ij}(t) - \sum_j w_{ij}(t\text{-}1) = 0 \tag{4.9}$$

folgt für den Koeffizienten c mit

$$0 = \sum_j \Delta w_{ij} = \sum_j \gamma(cx_j - w_{ij}) = \gamma(c\sum_j x_j - 1) \Rightarrow c = 1/(\sum_j x_j) \tag{4.10}$$

Im Binärfall ist die Zahl der Einsen $\sum_j x_j$ gleich dem Betragsquadrat $|\mathbf{x}|^2$.
Die Lerngleichung für die Gewichte des *r*-ten Neurons ist also

$$\mathbf{w}_r(t) = \mathbf{w}_r(t\text{-}1) + \Delta\mathbf{w}_r = \mathbf{w}_r(t\text{-}1) + \gamma[\mathbf{x}/|\mathbf{x}|^2 - \mathbf{w}_r(t\text{-}1)] \tag{4.11}$$

und $$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) \text{ für } i \neq r\,. \tag{4.12}$$

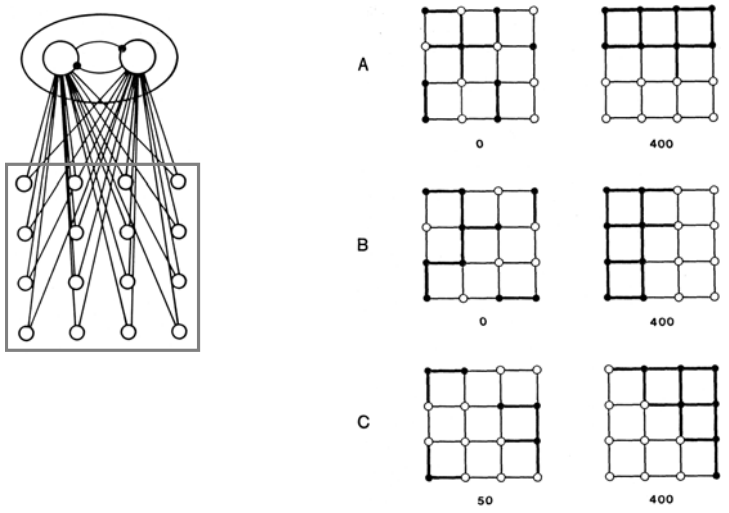
Das Modell des *competitive learning* ist besonders für Probleme geeignet, wo sich nur eine eindeutige Lösungskonfiguration durchsetzen darf. Beispiele dafür sind Klassenentscheidungen für Muster oder die Entscheidung, auf welchem Prozessor ein Programmstück ablaufen soll (Schedulingproblem). Das Modell des *competitive learning* lässt sich als Studienobjekt betrachten, um isoliert die Wirkungsweise einer hemmenden Wechselwirkung (Konkurrenzprinzip) zu erproben.

Historisch entwickelte sich der Mechanismus aus Modellen, die die Veränderung der Gewichte durch einen Gewichtsaufbau beim selektierten Neuron ("Gewinner") und ein Abbau bei allen anderen Neuronen vorsahen [GRO72], den Effekt der Normalisierung der Gewichte durch gleichmäßigen Abbau der Gewichte oder durch eine explizite Gewichtsnormalisierung [MAL73] ersetzten und schließlich noch eine Normalisierung der Eingabe vorsahen [GRO76]. Genauer ist diese Entwicklung in [GRO87] beschrieben. Eine der populärsten Formulierungen stammt von Rumelhart und Zipser (1985) [RUM85] und ist in dem bekannten Lehrbuch [RUM86] enthalten .

Beispiel: Dipol-Korrelationen

Von den vielen Anwendungen, die in der Literatur beschrieben sind, soll eine der ersten, klassischen Beispiele aus [RUM86] stellvertretend beschrieben werden.
Gegeben sei ein *Competitive Learning*-Netzwerk mit einer Schicht und zwei Ebenen: einer Eingangsebene mit 16 binären Eingängen und einer Klassifizierungsebene mit zwei sich gegenseitig hemmenden Einheiten. Die Eingänge werden dabei als zweidimensionales Feld der Maße 4×4 betrachtet. In Abb. 4.5 (a) ist dies gezeigt. Das Netz-

werk wurde mit zufälligen Anfangsgewichten initialisiert und dann als Trainingsmuster 400 Punkt-Paare von benachbarten Punkten im 4x4 Feld eingegeben.
Die Ergebnisse von drei Experimenten A, B und C sind in Abb. 4.5 (b) rechts gezeigt. Das linke Gitter zeigt dort den Zustand der Gewichte vor dem Training und das rechte Gitter den Zustand nach einem Training von 400 Iterationen. Dunkle, vollausgefüllte Gitterpunkte symbolisieren die Tatsache, dass Einheit 1 das größere Gewicht für diese Eingabe hat und weiße Gitterpunkte, dass dies eher für Einheit 2 der Fall ist. Entsprechend für zwei gleichzeitig aktivierte Punkte (Dipol) signalisiert eine dicke, schwarze Linie die Dominanz der Einheit 1 und eine dünne Linie die Dominanz der Einheit 2 für diesen Eingabe-Dipol.



(a) Eingabefeld und Netzwerk (b) Start- und Endzustände

Abb. 4.5 *Das Dipol-Experiment (nach [RUM86])*

Wie man sieht, sind anfangs die Gewichte, und damit die Präferenzen, ziemlich gleichmäßig verteilt. Bereits nach wenigen Iterationen ändert sich aber die Verteilung und am Schluss reagiert jeweils eine Einheit auf eine zusammenhängende Menge der Hälfte aller Punkte, obwohl der Zusammenhang nur im Training hergestellt wurde und nicht in der Architektur tatsächlich vorhanden ist. Wie lassen sich diese Ergebnisse interpretieren?

Diskussion

Betrachten wir unsere Lerngleichungen (4.11), (4.12) und die Aktivierungsgleichung (4.7), so können wir die beiden Gleichungen in (4.11) und (4.12) miteinander kombinieren und zu einer Lerngleichung für beliebige Neuronen *i*

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma\, y_i\, [\mathbf{x}/|\mathbf{x}|^2 - \mathbf{w}_i(t\text{-}1)] \tag{4.13}$$

zusammenfassen. Bei normierter Eingabeaktivität $\mathbf{x}' = \mathbf{x}/|\mathbf{x}|^2$ wird dies zu

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma\, y_i\, [\mathbf{x}' - \mathbf{w}_i(t\text{-}1)] \tag{4.14}$$

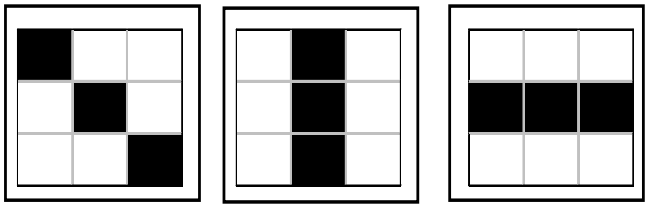
oder auch $\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + \gamma\, y_i\, [\mathbf{x}' - \mathbf{w}_i(t\text{-}1) y_i]$ da hier $y_i = 1$

Diese Lernregel ist der Oja-Lernregel aus Kapitel 3 sehr ähnlich, bei der als Lernziel die Gewichtsvektoren $\mathbf{w}_i$ zu den Eigenvektoren $\mathbf{e}^i$ der erwarteten Autokorrelationsmatrix $\mathbf{A}$ der Eingabewerte konvergieren! Man könnte also schlussfolgern, dass *competitive learning* gerade die Eigenvektoren als Merkmale in den Eingabemustern entdeckt und damit eine Eigenvektorzerlegung durchführt. Mit dieser Erklärung erscheint das Ergebnis im Beispiel der Dipol-Korrelation in einem anderen Licht: Das System lernt durch die Korrelation jeweils zweier Punkte die vollständige Ebene als Eingabe und bildet daraufhin zwei Eigenvektoren, die jeweils die Hälfte der Wahrscheinlichkeitsverteilung beschreiben. Sie entsprechen somit in etwa einer binären Version des Eigenvektors $\mathbf{e}^2$ und einem dazu orthogonalen (komplementären) Vektor $\bar{\mathbf{e}}^2$, wobei das gesamte Basisvektorsystem nur bis auf eine Drehung festgelegt ist.

Aus den obigen Ausführungen lässt sich folgern, dass *Competitive Learning* durch sein *winner-take-all* Prinzip zwar unkorrelierte Ausgänge und damit die Bildung von Eigenvektoren fördert, dies aber durch seinen ungenügend entwickelten Algorithmus nicht konsistent durchführt: Weder werden die Eigenvektoren systematisch entwickelt noch nach ihrer Wichtigkeit (Information!) bewertet. Für den gleichen Zweck sind deshalb binäre Versionen der Algorithmen aus Kapitel 3 durchaus brauchbarer.

Aufgabe 4.1

Programmieren Sie ein Netz, das als Eingabe die folgenden drei Bilder akzeptiert.



Die drei Ausgangsneuronen sollen dabei so reagieren, dass zu jedem Bild jeweils nur ein einziges die Ausgabe 1 produziert, die anderen die Ausgabe 0. Dies entspricht einem "winner-take-all" Algorithmus.

Trainieren Sie das Netz, bis die drei Bilder fehlerfrei klassifiziert werden. Versuchen Sie verschiedene Lernraten γ. Vergleichen Sie die erhaltenen Gewichtsvektoren mit den Eingabevektoren.

Welche Reaktion hat das Netzwerk, wenn ein unbekanntes Bild eingegeben wird ?

4.3 Lokale Wechselwirkungen und Nachbarschaft

Eine der wichtigsten Wechselwirkungen zwischen Neuronen ist die gegenseitige Hemmung innerhalb einer neuronalen Funktionsgruppe (Schicht). Im vorigen Abschnitt betrachteten wir ein einfaches Modell dazu: das "*competitive learning*" oder "*winner-take-all*" Modell, bei dem dasjenige Neuron einer Gruppe (*Cluster*) aktiv wird und lernt, das den zur Eingabe ähnlichsten Gewichtsvektor besitzt.

Dieses Modell kann man verfeinern, indem man allen Neuronen regelmäßig angeordnete Punkte eines Raumes (Koordinaten) zuordnet, beispielsweise die Kreuzungspunkte in einer Gitterstruktur. Mit einer solchen Zuordnung werden *Abstände* und *Nachbarschaften* zwischen den Neuronen definiert Die neue Anordnung hat besondere Möglichkeiten: alle Mitglieder eines Clusters, oder genauer: einer Nachbarschaft, reagieren ähnlich auf ähnliche Eingaben. Diese Eigenschaft bedeutet eine Abbildung von hochdimensionalen Musterpunkten auf Punkte (Neuronenorte) einer niederdimensionalen Anordnung: benachbarte Eingabemusterpunkte werden auch auf benachbarte Neuronen abgebildet.

Solche Abbildungen sind auch aus der Neurobiologie bekannt: Im Nervensystem höherer Tiere wurden sehr viele derartiger Zuordnungen von benachbarten Körperarealen zu benachbarten Gehirnarealen experimentell beobachtet. In Abb. 4.6 ist eine solche Zuordnung, wie sie für verschiedene Sensorik-Arten (auditiv, taktil, etc) bekannt ist, für die menschliche Motorik abgebildet.

Eine solche Zuordnung ist durchaus biologisch plausibel: unbekannte Eingaben lassen sich so leichter adäquat behandeln und führen meist zu biologisch sinnvollen Reak-

tionen, etwa das Wegziehen der ganzen Hand bei dem Schmerz an einer unbekannten Stelle an der Hand.

Aber auch in der Entwicklungsgeschichte der Mustererkennung stellte sich schon früh die Notwendigkeit heraus, die Struktur von Mustermengen {**x**}, die nur als Punkt-Wolken im *n*-dimensionalen Raum definiert sind, für das menschliche Vorstellungsvermögen erfaßbar und begreifbar auf der 2-dimensionalen Fläche einer Zeichnung abzubilden [BLOM]. Eine Abbildung, die den Musterraum in der Dimensionalität so drastisch reduziert, verliert dabei sicher Information. Um die Probleme bei der Trennung der hochdimensionalen Musterklassen weiterhin zu verdeutlichen, sollte allerdings eine solche Abbildung die wichtige Eigenschaft besitzen, dass benachbarte Punkte im Musterraum auch im Bildraum benachbart bleiben.

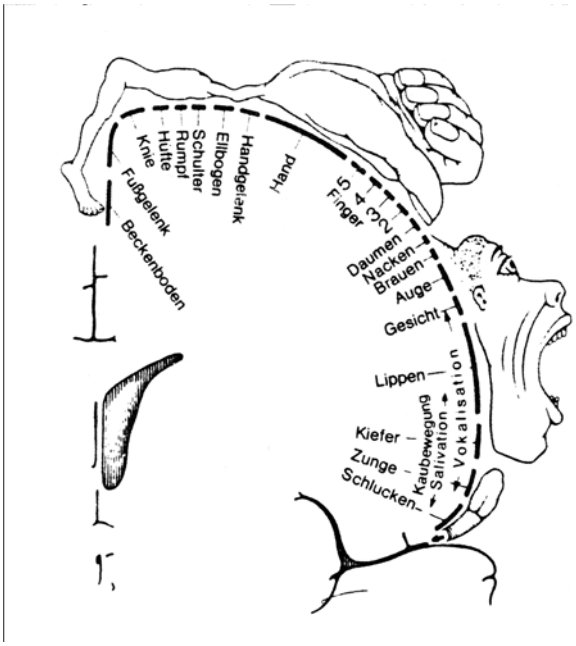


Abb. 4.6 *Die Somatopie im Cortex*

Diese Nachbarschaftserhaltung kann auch mit dem Begriff *Topologie-Erhaltung* beschrieben werden, der in der Mathematik die Erhaltung zusammenhängender Punktmengen ausdrückt. Bei den hier betrachteten Algorithmen und Anwendungen wird dieser Begriff allerdings nicht im qualitativen, mathematischen Sinn, sondern eher im

quantitativen Sinne eines Gütekriteriums benutzt: Obwohl eine Kugel topologisch nicht äquivalent einem Torus ist [ARN64], lässt sie sich trotzdem mit einer approximierenden Abbildung auf die Punktmenge eines Torus abbilden, wobei der mittlere Fehler dabei minimal, aber endlich, wird. Der Begriff "topologie-erhaltend" sollte deshalb besser durch den mathematisch weniger präzise definierten, vagen Begriff "nachbarschafts-erhaltend" (*neighborhood conserving*) oder treffender "topologie-approximierend" ersetzt werden, was aber in der Literatur meist nicht beachtet wird.

Lokale Wechselwirkungen in Neuronenfeldern

Das parallele Modell einer nachbarschafts-erhaltenden, lokalen Abbildung stützt sich im Wesentlichen auf das Modell einer nachbarschaftlichen, lokalen Kopplung zwischen den Neuronen, wie es beispielsweise als physiologisches Phänomen der *lateralen Inhibition* (Hemmung) bekannt ist. Hierbei erhalten die Neuronen nicht nur parallel zueinander eine Eingabe, sondern sie hemmen sich gegenseitig in der weiteren Nachbarschaft, so dass aktiv nur wenige, direkt benachbarte Neuronen übrigbleiben.
Die Grundidee des Modells besteht darin, ein kontinuierliches Feld *V* von Neuronen anzunehmen, bei dem an jedem Punkt **v** ein formales Neuron lokalisiert ist, siehe Abb. 4.7 (a).

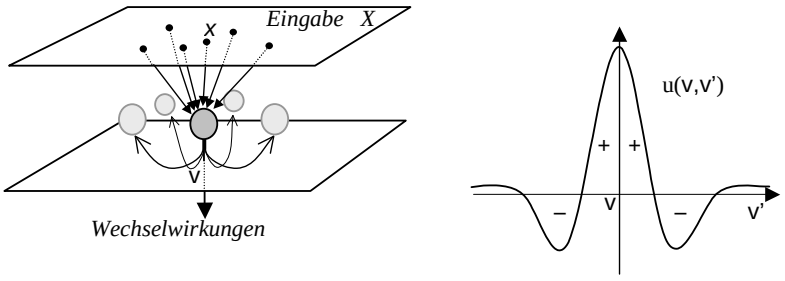


Abb. 4.7 *(a) Neuronenfelder und (b)Wechselwirkung mit Nachbarn*

Jedes dieser Neuronen erhält auch eine Eingabe von einem Feld X von Orten **x** und gewichtet sie mit Gewichten w(**x**,t). Seine Ausgabeaktivität $y = S(z)$ koppelt es zu allen anderen Neuronen in der Nachbarschaft, die sie mit den Gewichten u(**v**,**v'**) bewerten. Die Kopplungsgewichte werden als feste laterale Kopplung betrachtet und nicht verändert. In siehe Abb. 4.7 (b) ist die fördernde bzw. hemmende Funktion eines Neurons vom Platz **v** an alle Nachbarn am Platz **v'** aufgetragen. Wie man sieht, ähnelt die Funktionsform dem Schnitt durch einen Sombrero, so dass sie den Namen *Mexikanerhut*-Funktion erhalten hat. Interessanterweise wird diese Funktion auch in der Bildverarbeitung als Bildoperator zur Kontrast- und Konturbildung benutzt, vgl. [LEV85], S.180.

Damit lässt sich die zeitliche Aktivität $z(\mathbf{v},t)$ eines Neurons als Summe der gewichteten Eingabe $x(\mathbf{x},t)w(\mathbf{z},\mathbf{x},t)$, der Schwellaktivität $s(\mathbf{v},t)$ und der Summe aller Nachbarschaftseinflüssen $u(\mathbf{v}\text{-}\mathbf{v}')y(\mathbf{v}',t)$ mit einer Zeitkonstante τ in der differentiellen Form formulieren

$$\tau \frac{\partial}{\partial t} z(\mathbf{v},t) = -z(\mathbf{v},t) + \int_X w(\mathbf{v},\mathbf{x},t) x(\mathbf{x},t)\,d\mathbf{x} - s(\mathbf{v},t) + \int_V u(\mathbf{v}-\mathbf{v}') y(\mathbf{v}',t)\,d\mathbf{v}' \qquad (4.15)$$

Eingabe für Neuron $\mathbf{v}$ *– Schwelle + laterale* K*opplungen*

Die Gewichte $w(\mathbf{v},\mathbf{x},t)$ werden beim Training mit der Zeitkonstanten τ_1

$$\tau_1 \frac{\partial}{\partial t} w(\mathbf{v},\mathbf{x},t) = -w(\mathbf{v},\mathbf{x},t) + \gamma_1\, y(\mathbf{v},t)\, x(\mathbf{x},t) \qquad (4.16)$$

und die Schwelle $s(\mathbf{z},t)$ mit der Zeitkonstanten τ_2

$$\tau_2 \frac{\partial}{\partial t} s(\mathbf{v},t) = -s(\mathbf{v},t) + \gamma_2\, y(\mathbf{v},t)\, x_0 \qquad (4.17)$$

verändert, wobei $y = S(z)$ und für die Zeitkonstanten $\tau_1,\tau_2 \gg \tau$ gilt: Die Aktivität ändert sich wesentlich schneller als die Gewichte.

Die Lernregeln (4.16) und (4.17) reflektieren mit dem Term "*yx*" die Hebb'sche Lernregel. Es zeigt sich, dass das System stabil ist und bei kleinen Reizen in $\mathbf{x}$ auch kleine, lokalisierte Erregungszustände annimmt [AMA80]. Werden dagegen Reize mit endlicher Ausdehnung angelegt, so ist die Punktlösung nicht mehr stabil und es ergibt sich ein Erregungszustand ebenfalls endlicher Ausdehnung [TAK79].
Nehmen wir als Eingabe eine ungewichtete, sich addierende Erregung an, so lässt sich zeigen, dass das vollständig vernetzte System nur dann stabile Gewichtszustände ("Langzeitgedächtnis") herausbildet, wenn die Ausgabefunktion nicht-linear ist [SLH90]. Bei linearen Neuronen kann dieses System also nur als Filter wirken ("Kurzzeitgedächtnis"), nicht als Assoziativspeicher.

Eines der ersten Modelle für lateral gekoppelte Neuronen, das nachbarschaftserhaltende Eigenschaften aufwies, stammt von v.d.Malsburg [MAL73] und Willshaw [WIL76]. Das leicht modifizierte Modell wurde von Amari [AMA77], [AMA78], [TAK79], [AMA80] analysiert. Eine Übersicht und Einordnung der verschiedenen Arbeiten ist in [AMA83] enthalten.

Lokale Wechselwirkungen diskreter Neurone

Betrachten wir nun den Fall diskreter, zusammenhängender Neuronen, wie er von Kohonen [KOH84] formuliert wurde, s. Abb. 4.8. Die differentielle Aktivität z_i am Platz $\mathbf{v}_i$ nach Gl.(4.15) wird mit der Euler-Methode zur Integration ($\Delta t = 1$) eine Diffe-

renzengleichung. Bei $\tau := 1$ und $s_i = 0$ bzw. einer Kodierung der Schwelle als Gewicht erhalten wir für das Neuron *i*

$$z_i(t) - z_i(t\text{-}1) = -z_i(t\text{-}1) + \sum_j w_{ij}(t)\, x_j(t) + \sum_j u_{ij} y_j(t) \tag{4.18}$$

Nach dem "Abklingen" der Aktivität ergeben sich die Gleichungen

$$z_i(t) = \sum_j w_{ij}(t) x_j(t) + \sum_j u_{ij} y_j(t) \tag{4.19}$$

und $$w_{ij}(t) = w_{ij}(t\text{-}1) + \gamma x_j y_i \tag{4.20}$$

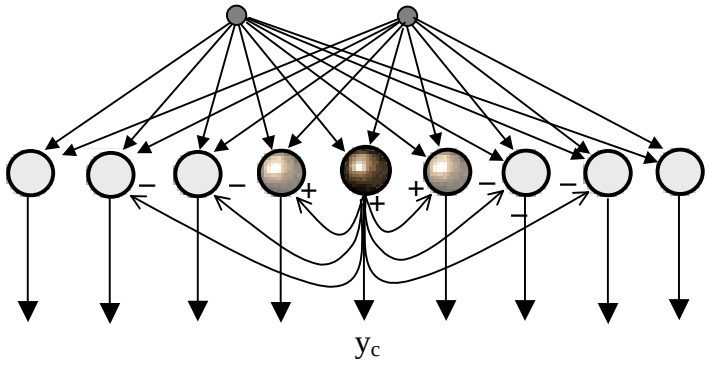


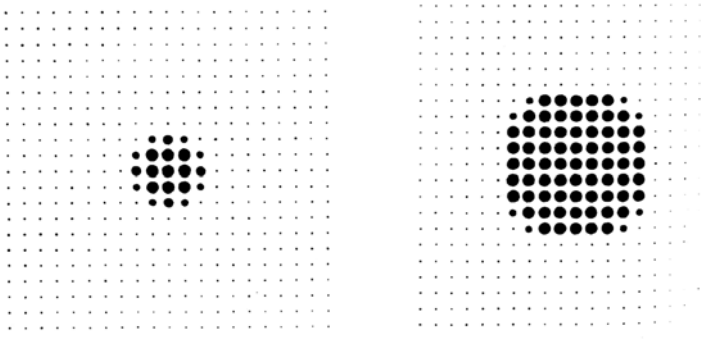
Abb. 4.8 *Diskrete Laterale Inhibition in Neuronenfeldern*

Um das unbeschränkte Anwachsen der Gewichte in Gl. (4.20) zu verhindern, ergänzte Kohonen die einfache Hebb'sche Regel durch einen Term "$-B(y_i)w_{ij}$", der in nichtlinearer Abhängigkeit von der Ausgabe y_i ein "Vergessen" bewirken soll. Dies entspricht der Idee der "synaptischen Konkurrenz": nicht benutzte Gewichte werden vermindert, wenn durch andere Gewichte das Neuron aktiviert wird. Mit diesem Term und der Ausgabefunktion $y = S(z)$ werden Gl. (4.19),(4.20) zu

$$y_i(t) = S\left(\sum_j w_{ij}(t)\, x_j(t) + \sum_j u_{ij} y_j(t) \right) \tag{4.21}$$

und $$w_{ij}(t) = w_{ij}(t\text{-}1) + \gamma[x_j y_i - B(y_i) w_{ij}] \tag{4.22}$$

Für die diskrete Formulierung sind in der folgenden Abb. 4.9 zwei Beispiele der Erregung eines zwei-dimensionalen Netzes mit einem Eingabemuster **x** bei geringen (a) und bei starken (b) Schwellwerten zu sehen, wobei die Größe der Aktivität eines Neurons durch die Größe der Punktschwärzung visualisiert wird.



(a) hoher Schwellwert (b) geringer Schwellwert

Abb. 4.9 *Parallele Aktivierung und Hemmung der Neuronen (aus [KOH84])*

Die Netzschicht wird aus gitterförmig angeordneten Neuronen gebildet und ist von oben betrachtet. Man sieht deutlich, dass die Erregung sich in zusammenhängenden Gebieten (Cluster, "Blasen") gruppiert. Die Neuronen eines solchen Clusters zeichnen sich dadurch aus, dass ihre Aktivität überhalb einer Schwelle liegt. Dies sind besonders diejenigen Neuronen, deren Gewichtsvektor eine starke Korrelation mit dem Eingabemuster aufweist. Sieht man diese Gruppe von Neuronen als "Gewinner" an, so ist ihre Aktivität eine Gruppenversion *(population coding)* der "*winner-take-all*" Regel aus dem vorigen Abschnitt.
Da die Erregung selbsttätig auch ohne Eingabe weiterbesteht, muss das gesamte System nach jeder Eingabe **x** durch ein Reset-Signal (Kontrollsignal) in der Aktivität zurückgesetzt ("gelöscht") werden, was beispielsweise im kontinuierlichen Modell mit Gleichung (4.17) durch Anheben der Schwellwerte mittels x_0 geschehen kann. Andere *reset*-Modelle, beispielsweise durch die integrierende, hemmende Aktivität zusätzlicher Neuronen, sind beispielsweise in [KOH93] zu finden. Für alle gilt, dass die *reset*-Aktivität in einem Zyklus verläuft, der jeweils vollständig in der Zeitspanne zwischen zwei unterschiedlichen Eingabemustern (einem Zeitschritt) ablaufen muss.
In verschiedenen Arbeiten (s. [KOH84]) zeigte nun T.Kohonen, dass sich diese Art von lokalisierter Erregung für verschiedene Anwendungen gut einsetzen lässt. Dazu vereinfachte er zunächst den diskreten Algorithmus in geeigneter Weise.

4.3.1 Selbstorganisierende Karten

Seien N Punkte $\mathbf{w}_1 ... \mathbf{w}_N$ gegeben, die beispielsweise in einer zwei-dimensionalen Nachbarschaft zusammenhängen sollen. Assoziieren wir mit jedem Punkt (Gewichts-

vektor) eine Prozessoreinheit (Neuron), so hat bei einer einfachen rechteckigen Netzmasche jedes Neuron vier direkte Nachbarn. In Abb. 4.10 ist eine solche Konfiguration gezeigt. Jedes Neuron bekommt parallel die gesamte Eingabeinformation des *n*-dim. Mustervektors **x** von oben und gibt eine Ausgabe y nach unten weiter. Zur Vereinfachung ist nur die Eingabe einer Variablen x_1 eingezeichnet.

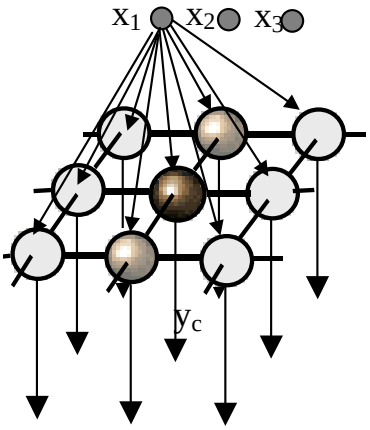


Abb. 4.10 *Nachbarschaftskonfiguration der Neuronen* $v = (i_1,i_2)$

Nach der vorherigen Abb. 4.9 lässt sich nun als "Mittelpunkt einer Aktivitätsblase" ein einziges Neuron *c* mit den Koordinaten $\mathbf{v}_c = (i_1,i_2)$ definieren, das als "selektiertes Neuron" mit der stärksten Aktivierung $z_c = \mathbf{w}^T_c\mathbf{x} = \max$ betrachtet werden kann und die anderen Neuronen der Nachbarschaft über die kurzreichweitige Mexikanerhut-Funktion erregt hatte. Die Ausgabe y_c dieses Neurons (und einiger Neuronen in der Nachbarschaft, die ebenfalls stark erregt werden) ist maximal und damit im Sättigungsbereich der Quetschfunktion $y = S(z) \approx 1$. Mit der Definition B(1):=1 wird aus Gl.(4.22) für die Gewichte des Clusters

$$w_{ij}(t) = w_{ij}(t\text{-}1) + \gamma[x_j\text{-}w_{ij}] \tag{4.23}$$

und für alle anderen Neuronen, die außerhalb des Clusters sind und mit y = 0 keine Aktivität aufweisen, für die Gewichte bei B(0):=0

$$w_{ij}(t) = w_{ij}(t\text{-}1) \tag{4.24}$$

Nehmen wir an, dass die Länge $|\mathbf{w}|$ des Gewichtsvektors für alle Neuronen gleich ist, so ist die Korrelation $|\mathbf{w}^T_c\mathbf{x}|$ für eine bestimmte Eingabe **x** beim gleichen Neuron *c* maximal bei dem auch der Abstand $|\mathbf{w}_c\text{-}\mathbf{x}|$ minimal ist: Anstelle des Neurons mit der größ-

ten Korrelation $\mathbf{w}^T_c\mathbf{x}$ kann man dasjenige Neuron *c* auswählen, dessen Gewichtsvektor $\mathbf{w}_c$ den kleinsten Abstand zum Eingabemuster hat

$$|\mathbf{x} - \mathbf{w}_c| = \min_k |\mathbf{x} - \mathbf{w}_k| \quad \textit{nearest neighbourhood classification} \tag{4.25}$$

Das Cluster von Neuronen wird in dieser Vereinfachung von Kohonen durch die Auswahl eines einzigen Neurons ersetzt, um das noch zusätzlich eine Nachbarschaft (Menge von Nachbarneuronen) definiert wird. Die Auswahlregel der maximalen Korrelation wird zur Auswahlregel des minimalen Abstands. Die Aktivierung des Neurons nach dem Konkurrenzprinzip (*winner-take-all*)

$$y_k := \begin{cases} 1 & \text{wenn } k = c \\ 0 & \text{sonst} \end{cases} \tag{4.26}$$

bedeutet eine Quantisierung des Eingabemusters (*Vektorquantisierung*): alle geringfügigen Variationen dieses Musters werden auf ein und denselben Gewichtsvektor abgebildet. Mit der Vorschrift aus Gl.(4.25) wird somit eine Aufteilung des Musterraums in einzelne *Klassen* vorgenommen; der Gewichtsvektor ist der *Klassenprototyp.*
Allerdings ist hier ein Bruch in der "Vereinfachung des Algorithmus" vorhanden: Wir erhielten aus dem Auswahlkriterien "maximale Korrelation" nur dadurch das Kriterium "minimaler Abstand", indem wir annahmen, dass bei gegebener Aktivität **x** die Gewichtslänge |**w**| genormt ist, oder aber aktivitätsabhängige Schwellen vorhanden sind. Für die Auswahloperation Gl.(4.25) ist aber beides nicht gegeben, so das die "Vereinfachung" tatsächlich einen *neuen* Algorithmus darstellt; der parallele Algorithmus (4.21) und (4.22) hat keine nachbarschaftserhaltenden Eigenschaften [ACK90] und ist für die weiteren Betrachtungen ungeeignet.
Die Auswahloperation Gl.(4.25) wird üblicherweise sequentiell vorgenommen. Eine parallele Hardwareversion ist nur in ziemlich komplizierter Weise denkbar, beispielsweise durch einen Ergebnis-Bus aller Einheiten mit einer Rückkopplung, bei dem alle Einheiten mit geringerem Ergebniswert ihre Ausgabe von selbst abschalten. Im Allgemeinen ist die Auswahloperation schwierig vollparallel mit Einzelneuronen durchzuführen, da es eine globale Operation ist.
Nehmen wir zum Beginn des Algorithmus Zufallswerte für die Gewichtsvektoren an, so geschieht der eigentliche Mechanismus der nichtlinearen Abbildung durch eine Korrektur (Lernen) aller Gewichtsvektoren durch die Eingabemuster **x** zum Zeitschritt t+1. Fassen wir die Gleichungen (4.23) und (4.24) mit Hilfe einer *Nachbarschaftsfunktion*

$$h(t+1, c, k) := \begin{cases} 1 & \text{wenn Neuron } k \text{ aus der Nachbarschaft von } c \text{ ist} \\ 0 & \text{sonst} \end{cases} \tag{4.27}$$

zusammen, so ergibt sich die Lernregel der *Kohonen map*

$$\mathbf{w}_k\,(t+1) = \mathbf{w}_k\,(t) + \gamma(t+1)\; h(t+1,c,k)\; [\mathbf{x} - \mathbf{w}_k(t)] \tag{4.28}$$

Die obige Gleichung ohne den Faktor h(t+1,c,k) beschreibt eine *stochastische Approximation*, deren Konvergenz unter bestimmten Voraussetzungen für **x** und γ (t) gegeben ist.
Die wichtige Idee des Lernschritts für die Gewichte in Gleichung (4.28), die ihn von den üblichen Verfahren (und damit der Konvergenzgarantie) der stochastischen Approximation sowie des "competitive learning" abhebt, ist die Einführung einer *Nachbarschaft* um das ausgewählte Neuron c. Mit Hilfe der Nachbarschaftsfunktion h(t+1,c,k) werden nicht nur das Neuron *c*, sondern parallel dazu alle Neuronen *k* innerhalb der Nachbarschaft am Lernprozeß beteiligt. Die Gewichtsänderung auch in der Nachbarschaft entspricht dabei dem positiven Anteil der Mexikanischen Hutfunktion; der negative Anteil wird weggelassen. Im Unterschied zum "competitive learning" gewinnt hier also nicht ein einzelnes Neuron, sondern eine ganze Gruppe. Dabei kann die "Nachbarschaft" diskret definiert sein, beispielsweise als "alle Neuronen innerhalb eines Radius d (Abstand) um das Neuron c", oder kontinuierlich, beispielsweise durch die Gaußfunktion

$$h(t,c,k) := \exp\mathbf{(}-(\mathbf{v}_c-\mathbf{v}_k)^2/2\sigma^2(t)\mathbf{)} \tag{4.29}$$

mit der Standardabweichung ("Nachbarschaftsradius") σ, wobei der Abstand $\mathbf{v}_c$-$\mathbf{v}_k$ der Neuronen durch die Differenz der Koordinaten (Indizes) modelliert wird. Üblicherweise wird die Nachbarschaft während der Iteration in Stufen langsam eingeengt, so dass am Schluß nur noch das selektierte Neuron allein übrigbleibt.
Interessanterweise ist das Verhalten des Algorithmus ziemlich unabhängig von der Art der Nachbarschaft; bei einer Computersimulation ist die Gewichtsveränderung in einer diskreten Nachbarschaft allerdings schneller berechnet, da im Unterschied zur Gaußfunktion keine zeitraubende Berechnung der Exponentialfunktion anfällt.

Im vereinfachten Programmierbeispiel "Codebeispiel 4.1" für den Algorithmus ist die diskrete Nachbarschaft im 2-dim Gitter ausgenutzt. Der "Nachbarschaftsradius" σ wird gezielt benutzt, um die durch den Index festgelegten Nachbarn zu ermitteln und die Gewichtsveränderungen auf wenige, ausgewählte Gewichte zu beschränken.
Mit der Einführung einer Auswahloperation wird eine Eingabe **x** nicht mehr von allen Neuronen gleichartig verarbeitet. Da sich die Klassengrenzen im Laufe der Iteration mit Änderung der Gewichte $\mathbf{w}_i$ verschieben, ist die Wahrscheinlichkeitsverteilung der **x**, wie sie von einem Neuron "gesehen" wird, durch einen sich verändernden Ausschnitt aus der Verteilung p(**x**) der Eingabe gegeben. Konvergenzziel und -bedingungen sind hier deshalb schwierig exakt zu bestimmen.

TOPO: (* topologie-erhaltende Abbildung auf ein m×m Gitter *)

```
VAR x:   ARRAY [1..n] OF REAL; (* Muster*)
    w:   ARRAY[1..m,1..m] OF ARRAY [1..n] OF REAL; (* Gewichte*)
    vc:  RECORD i,j : INTEGER END; (* sel. Neuron *)
 BEGIN
   σ :=σmax ;                                    (* initial: max. Nachbarschaft *)
   FOR t:=1 TO tmax DO
    Read(PatternFile,x)                          (* Muster erzeugen oder einlesen *)
    (* Neuron selektieren *)
    Min:= ABS(x-w[1,1]);                         (* initialer Wert *)
    FOR i:=1 TO m DO                             (* In allen Spalten *)
       FOR j:= 1 TO m DO                         (* und Zeilen *)
           Abstand:= ABS(x-w[i,j]);   (* suche Minimum *)
           IF Abstand < Min
              THEN Min:=Abstand; vc.i:=i; vc.j:=j;
           END
       END
    END
    (* Gewichte adaptieren *)
    FOR i:= vc.i-σ TO vc.i+σ DO                  (* Evaluiere 2-dim. *)
      FOR j:= vc.j-σ TO vc.j+σ DO                (* Nachbarschaft um c*)
        IF i>0 AND i<= m AND j>0 AND j<=m THEN   (* Vorsicht am Rand *)
              w[i,j] = w[i,j] + 1.0/FLOAT( t)*( x-w[i,j])
        END
      END
    END (*i*)
    GrafikAnzeige(t,w)                  (* Visualisierung der Iteration *)
    upDate(σ,t)                         (* Verkleinerung des Nachbarschaftsradius σ *)
   END (*t*)

 END TOPO.
```

Codebeispiel 4.1 *Nachbarschaftserhaltende, 2-dim. Abbildung*

In der folgenden Abb. 4.11 ist die Entwicklung und Konvergenz einer solchen topologie-approximierenden Abbildung am Beispiel eines begrenzten, zweidimensionalen Musterraums (Rechteckfläche) gezeigt. Die Eingabemuster sind uniform verteilt über den Musterraum, dessen Grenzen mit der rechteckigen Umrandung angedeutet ist. In dem Musterraum sind die Gewichtsvektoren als Punkte eingezeichnet; Gewichtsvektoren direkt benachbarter Neuronen sind jeweils mit einer Linie verbunden.

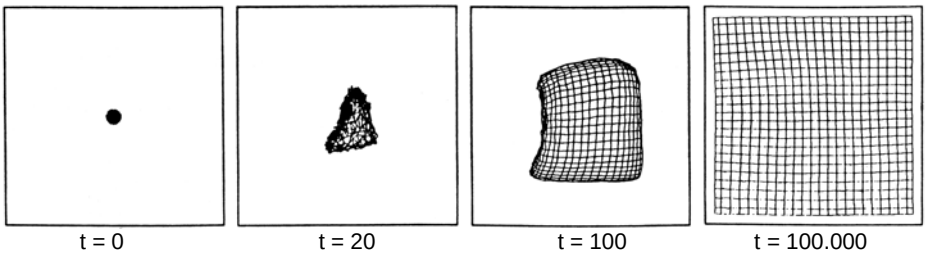


Abb. 4.11 *Selbstorganisation und Konvergenz der topol.-erh. Abbildung* (nach [KOH84])

Am Anfang der Iteration sind die Gewichtsvektoren initial nur in einem kleinen Bereich zufällig verteilt um den Nullpunkt in der Mitte. Schon bald aber wirken sich die über die Nachbarschaft der Neuronen definierten Zusammenhänge aus und die Gewichtsvektoren ordnen sich entsprechend im Musterraum. Dabei wirken sich zwei verschiedene Mechanismen aus: Zum einen werden die Gewichtsvektoren benachbarter Neuronen in die gleiche Richtung korrigiert, so dass die Gewichtsvektoren benachbarter Neuronen die Tendenz haben, ähnlich zu werden (*Selbstordnung*); zum anderen erhalten die Neuronen am Gitterrand als Eingabe mindestens die Punkte **x** am Rande der Verteilung, so dass ihre Gewichtsvektoren "an den Rand streben" und das Netz aus den Gewichtsvektoren "wie ein Netz aus Gummibändern" auseinanderziehen (*Entfaltung*).

Die Tendenz zur Selbstordnung lässt sich relativ einfach zeigen: Werden zwei Neuronen mit der selben Eingabe iteriert, so lernen beide Gewichtsvektoren $\mathbf{w}_1$ und $\mathbf{w}_2$; sie streben dichter zusammen und werden dadurch zu Nachbarn. Ausgehend von den Iterationsgleichungen (4.28) mit h = 1 für beide Gewichtsvektoren ist

$$\mathbf{w}_1(t+1) = \mathbf{w}_1(t) + \gamma\,(\mathbf{x}(t)-\mathbf{w}_1(t)) = (1-\gamma)\mathbf{w}_1(t) + \gamma\,\mathbf{x}(t)$$

und $$\mathbf{w}_2(t+1) = \mathbf{w}_2(t) + \gamma\,(\mathbf{x}(t)-\mathbf{w}_2(t)) = (1-\gamma)\mathbf{w}_2(t) + \gamma\,\mathbf{x}(t)$$

mit dem gleichen Faktor γ. Ihr Abstand $|\mathbf{w}_1-\mathbf{w}_2|$ zum Zeitpunkt t ist nach der Iteration

$$|\mathbf{w}_1(t+1)-\mathbf{w}_2(t+1)| = |(1-\gamma)\mathbf{w}_1(t) - (1-\gamma)\mathbf{w}_2(t)| = |\,(1-\gamma)[\mathbf{w}_1(t) - \mathbf{w}_2(t)]|$$

Für $0 < \gamma \leq 1$ ist also

$$|\mathbf{w}_1(t+1) - \mathbf{w}_2(t+1)| < |\mathbf{w}_1(t) - \mathbf{w}_2(t)|$$

der Abstand der Gewichtsvektoren benachbarter Neuronen nach der Iteration geringer als vorher; die Gewichtsvektoren (Klassenprototypen) werden immer ähnlicher.
Die genauen mathematischen Zusammenhänge dafür sind sehr schwierig zu erfassen und bisher noch nicht hinreichend analysiert worden. Die Verteilung der Gewichtsvek-

toren nach der Selbstordnungsphase dagegen ist verschiedentlich näher untersucht worden (s. [KOH82a], [RITT88],) und Gegenstand von Kontroversen (vgl. [RITT86]).

Eine ganz andere Situation ergibt sich, wenn wir für die Nachbarschaftsfunktion h(t,c,k) anstelle einer Rechteckfunktion oder Gaußfunktion die ursprüngliche mexikanische Hutfunktion wählen. Hier werden alle "Rivalen" des gewinnenden Neurons mit einem Abrücken vom Ziel **x** "bestraft", so dass sich der Kontrast noch verstärkt. Dieser Mechanismus wurde von [XUKO93] ausgenutzt, um für eine Clusterung des Eingaberaums (*Clusteranalyse*) sicherzustellen, dass zu jedem Cluster nur ein Neuron konvergiert. Dabei wird speziell mit h(t,c,k) = +1 für c = k, h(t,c,k) = –1 für das Neuron mit dem zweitnächsten Abstand (der "Rivale") und null für alle anderen erreicht. Nach einer gewissen Zahl von Iterationen "landen" nicht benötigte Neuronen am Rande des Eingabefeldes und werden nicht mehr iteriert. Die Zahl der "aktiven" Neuronen entspricht somit der gesuchten Anzahl der Cluster; ihre Gewichte den Clustermittelpunkten.

Netzformation mit maximaler Information

Wir können die nachbarschafts-erhaltenden Abbildungen als die Schicht eines Systems betrachten, das maximal viel Information anbieten soll. Wir wissen, dass die Information über das Auftreten einer aus *M* Klassen ω_i maximal wird, wenn alle Klassen die gleiche Auftrittswahrscheinlichkeit haben

$$P(\omega_i) = P(\omega_j) \overset{!}{=} 1/M \tag{4.30}$$

Für kontinuierliche Dichtefunktionen p(**x**) der Muster im Eingaberaum kann man bei sehr vielen Klassen approximativ auch eine Dichtefunktion M(**x**) der Klassen ω aufstellen. Sie lässt sich als Grenzwert definieren für ein Raumelement ΔA

$$M(\mathbf{x}) := \lim_{\Delta A \to 0} \frac{\text{Klassenzahl k}}{\Delta A} \tag{4.31}$$

Haben alle Klassen die gleiche Wahrscheinlichkeit, so ist die Zahl der Klassen in einem Raumsegment gerade die Wahrscheinlichkeit, dass ein Ereignis dort stattfindet, geteilt durch die Wahrscheinlichkeit einer Klasse:

$$k = \frac{P(\mathbf{x} \in \Delta A)}{P(\omega)} = \frac{P(\mathbf{x} \in \Delta A)}{1/M} \tag{4.32}$$

Unser Grenzwert wird somit

$$M(\mathbf{x}) = \lim_{\Delta A \to 0} M \frac{P(\mathbf{x} \in \Delta A)}{\Delta A} = M\, p(\mathbf{x})$$

zur Wahrscheinlichkeitsdichte am Ort **x**. Also gilt für die Netzformation mit maximaler Information

$$M(\mathbf{x}) \sim p(\mathbf{x}) \tag{4.33}$$

Die Funktion $M(\mathbf{x})$ wird „Vergrößerungsfaktor“ (*magnification factor*) genannt und beschreibt, dass der Musterraum in Zonen dichterer "Punktwolken" durch mehr Klassen stärker unterteilt wird und die Ausgabe damit genauer wird.

Hat nun der nachbarschafts-erhaltende Algorithmus von Kohonen diese Optimalitätseigenschaften? Kohonen fand bei einer Untersuchung [KOH82a] allgemein die Proportion $M(\mathbf{x}) \sim p(\mathbf{x})$. Demgegenüber zeigten aber Ritter und Schulten [RITT86] mathematisch und durch Simulation, dass dies nicht allgemein der Fall ist. Beispielsweise ergeben sich im eindimensionalen Fall $M(\mathbf{x}) \sim p(\mathbf{x})^{2/3}$ und für höhere Dimensionen kompliziertere Ausdrücke. Nur im zweidimensionalen (komplexen) Fall gilt obige Relation $M(\mathbf{x}) \sim p(\mathbf{x})$.
Wie können wir den Kohonen-Algorithmus abändern, um ihn optimal zu machen? Beispielsweise kann man die relative Häufigkeit P_k = (Anzahl der Klassenwahl k / Anzahl aller Muster) bestimmen, mit der eine Klasse ω_k gewählt wurde, und als Abstandsmaß den mit der relativen Häufigkeit gewichteten, Euklidschen Abstand nehmen [AKCM90]. Als Neuron c wird dann statt Gl. (4.25) das Neuron gewählt mit

$$\left|\mathbf{x} - \mathbf{w}_c\right| P_c = \min_k \left|\mathbf{x} - \mathbf{w}_k\right| P_k \tag{4.34}$$

Ist ein Neuron schon sehr oft gewählt worden, so erhöht sich P_k entsprechend und ein anderes Neuron, dass zwar weiter entfernt liegt, aber weniger oft gewählt wurde, gewinnt die Entscheidung. Die Lernregeln bleiben dabei die bekannten Regeln (4.28) der "Kohonen map".

Aufgabe 4.2

Zeigen Sie analytisch: Nach der Lernoperation hat der Gewichtsvektor des selektierten Neurons immer noch den kürzesten Abstand zum Eingabemuster **x** gegenüber allen anderen, ebenfalls iterierten Nachbarneuronen.

Aufgabe 4.3

Lassen Sie die drei Klassenprototypen mit den Koordinaten $\mathbf{w}_1 = (1,3)$, $\mathbf{w}_2 = (3,3)$ und $\mathbf{w}_3 = (2,1)$mit einem Kohonennetz lernen. Wählen Sie dazu eine Eingabe drei Gaußverteilungen, die auf den Klassenprototypen zentriert sind.

Überdecken Sie den 2-dim. Eingaberaum mit einem Gitter aus 10×10=100 Neuronen und initialisieren Sie dazu passend die Neuronengewichte. Hinweis: Ein Gitter lässt sich grafisch leicht visualisieren, wenn man von jedem Gewichtsvektor (Punkt

im Eingaberaum) eine Line nach oben und nach rechts zum Punkt des Nachbarneurons im Gitter zieht (MoveTo, DrawTo-Befehle). Nur in der letzten Spalte bzw. Zeile muss man aufpassen.

Aufgabe 4.4

Wie Sie wissen, soll für eine maximale Informationsübertragung jede Klasse die selbe Auftrittswahrscheinlichkeit haben. Testen Sie unter diesem Gesichtspunkt den Kohonen-Algorithmus. Simulieren Sie dazu im 1-dim Fall für ein Netz aus M = 100 Neuronen zunächst eine uniforme Eingabeverteilung p(x) = const der x aus [0,1] und ermitteln Sie als Fehlermaß die mittlere Abweichung der Selektionswahrscheinlichkeit (= normierte Zahl der Selektionen) eines Neurons von der Wahrscheinlichkeit 1/M bei jedem Iterationsschritt und beobachten Sie die Fehlerabnahme. Die Nachbarschaft jedes Neurons soll bei einer Iteration unverändert bleiben.

Danach simulieren Sie eine linear ansteigende Dichte p(x) = 2x durch eine Transformation der Zufallsvariablen mittels $x = \sqrt{z}$ mit der uniform verteilten Zufallsvariablen *z*. Ist die Differenz zu 1/m nun abhängig vom Klassenindex ?

Modifizieren Sie den Algorithmus so, dass die Größe h(.) der Nachbarschaft mit fortschreitender Iteration abgesenkt wird. Ändert sich das Ergebnis?

4.3.2 Überwachte, adaptive Vektorquantisierung

Die selbstorganisierenden Karten des vorigen Abschnitts lernen eine Vektorquantisierung, die auf den statistischen Eigenschaften der Eingabemuster beruhen (*unüberwachtes Lernen*). Anders liegt aber der Fall, wenn die korrekte Einordnung der Muster bekannt ist. Diese Information kann man direkt in den Iterationsprozeß für die Klassenprototypen $\mathbf{w}_i$ und damit für die Klassengrenzen einbeziehen. Einige Möglichkeiten dazu sind im Folgenden beschrieben.
Der einfachste Ansatz besteht darin, die bekannte Klassenzugehörigkeit an Stelle der Nachbarschaftsfunktion h(t,c,k) der Lerngleichung (4.28) zu notieren. Definieren wir

$$h(t,c,k) := \begin{cases} 1 & k = c, \text{Klasse}(\mathbf{w}_c) = \text{Klasse}(\mathbf{x}) \\ -1 & k = c, \text{Klasse}(\mathbf{w}_c) \neq \text{Klasse}(\mathbf{x}) \\ 0 & k \neq c \end{cases} \tag{4.35}$$

so schieben wir den richtigen Prototypen zum Häufungspunkt der **x**, den falschen entfernen wir davon. Auch hier bewirkt eine eigene Lernrate pro Gewichtsvektor eine Konvergenzbeschleunigung.

Eine andere Idee bezieht auch noch den zweit-ähnlichsten Gewichtsvektor $\mathbf{w}_{c'}$ mit ins Lernen ein, um das Lernen durch Muster, die dicht an einer Klassengrenze liegen, zu verbessern.

$$h(t,c,k) := \begin{cases} 1 & k = c, c' \text{ und Klasse}(\mathbf{w}_k) = \text{Klasse}(\mathbf{x}) \\ -1 & k = c, c' \text{ und Klasse}(\mathbf{w}_k) \neq \text{Klasse}(\mathbf{x}) \\ 0 & k \neq c, c' \end{cases} \tag{4.36}$$

Die Algorithmen stammen aus der Arbeitsgruppe von Kohonen und sind als LVQ (*learning vector quantization*) bekannt [KOH88b].

Aufgabe 4.5

Realisieren Sie einen Lernalgorithmus für die Klassenprototypen mit den Koordinaten $\mathbf{w}_1 = (1,3)$, $\mathbf{w}_2 = (3,3)$ und $\mathbf{w}_3 = (2,1)$ mit der überwachten Kohonen Lernregel.

4.3.3 Neuronale Gase

Der Algorithmus von Kohonen bildet einen n-dimensionalen Eingaberaum in einen Unterraum ab. Die Zahl D der Dimensionen des Unterraums muss der Benutzer dabei selbst festlegen. Sie entscheidet über die Art der Nachbarschaft und damit über die Auswahl der Neuronen, deren Gewichtsvektor zusätzlich zum selektierten Neuron ebenfalls verbessert wird.

Abbildungsdimension und Informationsdimension

Die Abbildung des höher-dimensionalen Eingaberaums in den geringer-dimensionalen Unterraum ist aber nicht immer unproblematisch. Abb. 4.12 zeigt das Ergebnis einer Abbildung einer 3-dim Datenmenge auf eine 2-dim Fläche.

Man sieht, dass die 2-dim Ebene nicht "faltenfrei", sondern leicht "zerknittert" in dem Eingaberaum enthalten ist. Die "Falten" äußern sich als starke Variationen in den Nachbarschaftsrelationen und können bei großen Dimensionsunterschieden zu Konvergenzproblemen führen [SRR94]. Dabei ist nicht die formale Differenz (*n-D*) zwischen den Ein- und Ausgabedimensionen entscheidend, sondern die innere Struktur der Datenwolken. Besitzen sie nur wenig Inhomogenitäten, so ist eine "faltenfreie" Abbildung leicht möglich. Wie können wir nun bei gegebenen Trainingsdaten feststellen, wie groß die Dimension des passenden Ausgaberaumes sein sollte?

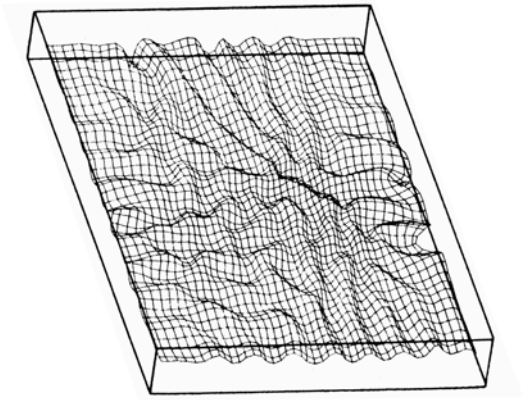


Abb. 4.12 *Abbildung einer 3-dim Datenmenge auf eine 2-dim Ausgabe (nach [KOH84])*

Automatische Wahl der Ausgabedimension

Da die Zahl der Dimensionen des Unterraums eine willkürlich gewählte Konstante ist, die nicht unbedingt den Eigenschaften der Trainingsdaten im Eingaberaum entspricht, wäre es schön, wenn der Algorithmus die Nachbarschaft selbsttätig ermitteln würde.
Diese Überlegung liegt dem Algorithmus des *Neuronalen Gases* zugrunde, der von Martinetz et. al. [MAR91], [MAR93] entwickelt wurde. Ursprünglich sind alle Neuronen ohne Nachbarschaft, also "wie in einem gasförmigen Zustand", vorhanden. Ziel des Algorithmus ist es, im Laufe der Iterationen stabile Nachbarschaftsverbindungen zu formen. Identifizieren wir die Nachbarschaftsrelationen des neuronalen Netzes mit einem Graphen, so versuchen wir, die Adjazenzmatrix $\mathbf{A} = (A_{ij})$ des Graphen, die mit $A_{ij} = 1$ eine direkte Nachbarschaftsverbindung zwischen den Neuronen i und j beschreibt, zu lernen. Dazu schätzen wir die Nachbarschaft mit dem "Nachbarschaftsgrad" $B_{ij} \in \Re^+$ ab. Haben wir noch eine geeignete Abstandsfunktion $y_i = S(|\mathbf{x}\text{-}\mathbf{w}_i|)$, die mit zunehmendem Argument monoton abnimmt (z.B. eine Glockenfunktion), so lässt sich die Entscheidung, wer nächster Nachbar ist, damit folgendermaßen definieren:

Wähle Neuron *k* mit

$$y_k = \max_i y_i \qquad \textit{winner take all} \tag{4.37}$$

und den nächsten Nachbarn *r* mit

$$y_k y_r = \max_j y_k y_j \tag{4.38}$$

Verbessere den Nachbarschaftsgrad von beiden mit

$$B_{kr}(t+1) = B_{kr}(t) + \gamma\, y_k y_r \tag{4.39}$$

Dabei wird der Gewichtsvektor des Gewinners *k* wie üblich verbessert. Im Unterschied zur festen, vorher festgelegten Nachbarschaft der Kohonen-Karte bildet sich aber hier die Nachbarschaftsrelation durch die Aktivität heraus; die Nachbarn passen nicht ihren Gewichtsvektor an die Nachbarschaft an, sondern die Nachbarschaft wird nach der Ähnlichkeit der Aktivitäten und damit der Gewichtsvektoren gewählt.
Es lässt sich zeigen [MAR93], dass mit fortschreitender Zeit der aufgerundete Wert $S_B(B_{ij}(t))$ zum Adjazenzkoeffizienten A_{ij} strebt.

Die Fortschritte eines solchen Algorithmus ist in der folgenden Abb. 4.13 zu sehen.

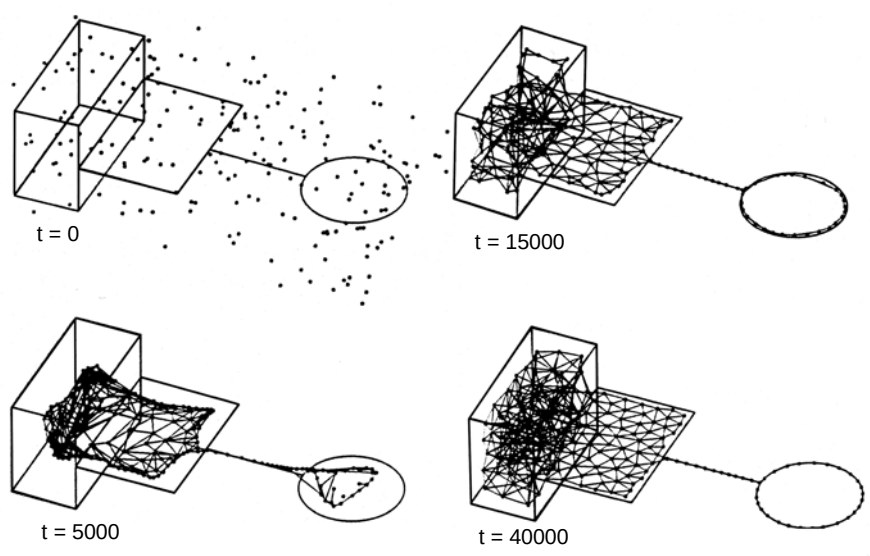


Abb. 4.13 *Anpassung der Nachbarschaft bei neuronalen Gasen (aus [MAR91])*

Hier sind im drei-dimensionalen Raum die Trainingsdaten aus drei verschiedenen Teilmengen mit verschiedenen Dimensionen entnommen: einem Kubus, einer Fläche und einem Ring. Mit sieht, wie mit fortschreitender Iteration die Gewichtsvektoren (abgebildet als Punkte) und ihre direkten Nachbarn (durch Linien verbunden) sich der tatsächlichen Dimensionalität der Eingaben anpassen.
Die gelernte Nachbarschaft der neuronalen Gase geht von den Nachbarschaften der zufälligen Anfangsgewichte der Neuronen aus und vermeidet so die Umordnungsphase,

die bei fester Nachbarschaft den Kohonen-Algorithmus anfangs bestimmt und dabei die Konvergenzgeschwindigkeit herabsetzt.

Aufgabe 4.6
Erproben Sie den Algorithmus der neuronalen Gase. Wählen Sie dazu als Eingaberaum die 2-dim Fläche und eine 1-dim Punktgerade als Ziel-Ausgaberaum für ein Netz aus 100 Neuronen. Es ist also $\mathbf{x} = (x_1,x_2)$ mit x_1 = UniformRandom aus [0,1] und $x_2 = ax_1 + b$.

4.3.4 Andere Varianten

Möchte man die Auswahl des zu aktivierenden Neurons und die Veränderung seines Gewichtsvektors als Regeln zusammenfassen, so kann man die Auswahlregel als Aktivität in die Lernregel indirekt mit einbeziehen, beispielsweise über die reine *winner-take-all* Gleichung (4.14). Allerdings bleibt damit die Nachbarschaft nur auf das gewinnende Neuron beschränkt. Man kann deshalb zusätzlich noch als Lernziel einbeziehen, die Abweichungen $(\mathbf{w}_k\text{-}\mathbf{w}_i)^2$ der Gewichtsvektoren innerhalb einer Nachbarschaft N_k des Neurons *k* nicht zu groß werden zu lassen. Betrachten wir den negativen Gradienten davon als Term einer Lernregel und addieren ihn zu der Lerngleichung (4.14), so erhalten wir für alle sich aufsummierenden Eingaben $\{\mathbf{x}_j\}$

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \alpha y_k\left[\mathbf{x} - \mathbf{w}_k(t)\right] + \beta \sum_{i \in N_k} (\mathbf{w}_k - \mathbf{w}_i) \tag{4.40}$$

Mit der normierten Ausgabe

$$y_k(\sigma_x) = \frac{S_k(\mathbf{x})}{\sum_i S_i(\mathbf{x})} \qquad \textit{"soft" winner-take-all Auswahl} \text{ aus Gl.(4.8)} \tag{4.41}$$

und der Gauß'schen Ausgabefunktion $S_i = \exp(-(\mathbf{x}\text{-}\mathbf{w}_i)^2/\sigma_y^2)$ ist dies die verallgemeinerte Lernregel des *elastischen Netzes* [DUM90],[DUW87] mit dem "Elastikparameter" β.
Anstelle des "Elastikanteils" lässt sich allerdings auch hier direkt wie in Gl.(4.29) eine Nachbarschaftsfunktion $h(i,k,\sigma_y)$ im Raum der Ausgabeneuronen, beispielsweise eine Gaußfunktion, ansetzen, mit der die Gewichtsunterschiede zwischen Nachbarn möglichst klein gehalten werden [SPWG94]

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \gamma(t+1)\left\langle \sum_i y_i(\sigma_x) h(i,k,\sigma_y)\left[\mathbf{x} - \mathbf{w}_k(t)\right]\right\rangle_x \tag{4.42}$$

Die Summe $\Sigma_i y_i(\sigma_x)h(i,k,\sigma_y)$ misst dabei die mit der Nähe im Ausgaberaum h(i,k) gewichteten Aktivitäten y_i der Nachbarn *i*, inklusive der von Neuron *k*. Eine tiefere

mathematische Analyse zeigt, dass beide Modelle im Fall $\beta/\alpha = \sigma_p^2\sigma_x^2/2\sigma_y^2$ übereinstimmen, wobei hier neben der Weite σ_x des Eingabefeldes und der Weite σ_y der Ausgabenachbarschaft auch die Varianz σ_p der Eingabeverteilung eine Rolle spielt.
Eine weiterte Variation ist die Idee, das Netz sukzessive den Daten entsprechend wachsen zu lassen [FRIT92a].
Es gibt noch sehr viele weitere Varianten, auf die aber aus Platzgründen verzichtet werden muss.

4.4 Anwendungen

Die Methode der nachbarschafts-erhaltenden Abbildungen lässt sich auf die verschiedensten Probleme anwenden. Sie ist nicht beschränkt auf die ursprünglichen Anwendungen in der Mustererkennung, sondern kann in verschiedenen Bereichen eingesetzt werden. Beispielhaft für viele Bereiche sind nachstehend einige wichtige Beispiele in ihren prinzipiellen Ansätzen kurz beschrieben; für Details sei auf die jeweilige Literatur verwiesen.

4.4.1 Das Problem des Handlungsreisenden

Eines der "Standardprobleme" der Informatik und Prüfstein vieler Ansätze ist das *Problem des Handlungsreisenden*: Ein Handlungsreisender möchte auf seiner Reise alle Kunden in allen Städten seines Bezirks hintereinander besuchen, beispielsweise in Deutschland (s. Abb. 4.14). Dabei möchte er aber Zeit und Kosten möglichst gering halten. Ist außer den Streckenlängen zwischen den Städten nichts weiter über die Reise bekannt (z.B. verbindungsabhängige Kosten etc), so besteht sein Problem darin, eine möglichst kurze Reiseroute zu wählen, die alle Städte umfasst.
Dieses berühmte Problem ist als Problem der *Optimierung* bekannt; es gehört zu der Klasse der *NP-vollständigen* Probleme. Wie man sich leicht überlegen kann, hat der Reisende nach der Wahl der ersten Stadt genau *N-1* Möglichkeiten, seine Reise fortzusetzen. Hat er eine davon gewählt, so verbleiben ihm bei der zweiten Stadt noch *N-2* Alternativen für seine dritte Station und so fort. Da die erste Stadt in allen Rundreisen vorkommen muss, ist ihre Wahl nicht typisch für eine Rundreise. Die Zahl der möglichen Reisen ist also

$$M = (N-1)(N-2)(N-3)... = (N-1)! \; . \tag{4.43}$$

Mit der Stirling'schen Formel

$$N! \approx (2\pi N)^{1/2} N^N e^{-N}$$

ist $M = (N-1)! = N!\, N^{-1} \approx (2\pi)^{1/2}\, N^{N-1/2}\, e^{-N} = (2\pi)^{1/2}\, e^{(N-1/2)\ln N - N}$ (4.44)

Die Zahl M der Rundreisen nimmt also mit wachsender Städtezahl exponentiell zu, so dass selbst moderne Rechner bei kleinen Problemgrößen wie N = 1000 hoffnungslos überlastet sind: Bei der Rundreise aus Abb. 4.14 durch N = 120 Städte gibt es mit ca. $M = 6x10^{196}$ möglichen Reisen mehr Rundreisen als Elementarteilchen im Universum.

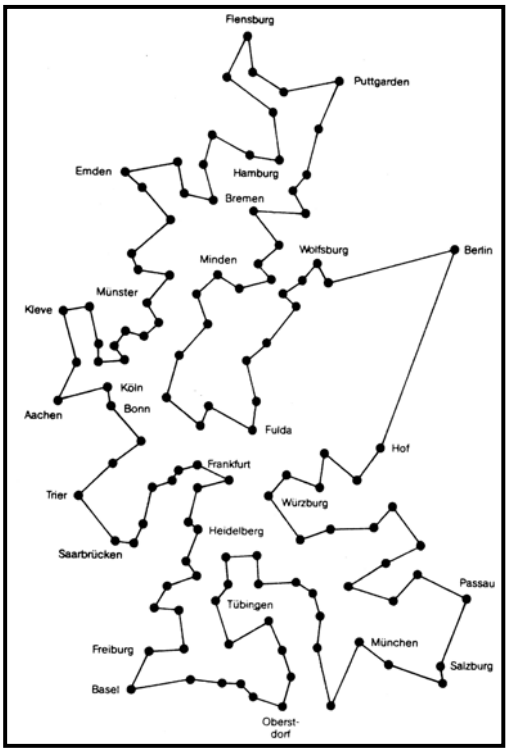


Abb. 4.14 *Eine Rundreise durch 120 Städte (nach[AB87])*

Deshalb rücken für die praktische Anwendung Verfahren in den Mittelpunkt des Interesses, die nicht unbedingt nach langer Rechenzeit die überhaupt mögliche allerbeste Rundreise (*Optimum*) finden, sondern bereits nach kurzer Zeit eine noch nicht ganz so gute Reise (*Suboptimum*) anbieten. Ein Beispiel dafür ist das Problem, bei einer automatischen Platinenbestückung mit Bauteilen durch möglichst kurze Wege für den Bestückungsarm möglichst kurze Bestückungszeiten zu erzielen.

Interessanterweise kann man nun zur Lösung dieser (und ähnlich gelagerter) Probleme auch die Mechanismen der Kohonen Map einsetzen. Bei dem Ansatz von B. Angéniol, et al. [ANG88], werden die Stadtkoordinaten $\mathbf{x} = (x_1,x_2)$ als Eingabemuster aufgefaßt, so dass die Menge $\{\mathbf{x}\}$ aller Städte die Trainingsmustermenge bildet. Bei jedem Iterationszyklus werden alle Städte bzw. Eingabemuster in einer einmal festgelegten Reihenfolge sequentiell eingegeben. Ausgehend von einer Kette aus m = 1 Neuronen wird nun für jedes Trainingsmuster das Neuron mit dem "kürzesten Abstand" zwischen Gewicht und Muster mit Gl.(4.25) gesucht. Wurde dieses Neuron bereits für eine Stadt ausgewählt, so wird eine Kopie des Neurons (Kopie des Gewichts) erzeugt, beide Neuronen "ausgeschaltet" und die Kopie als Nachbarneuron in die geschlossene Kette (ein-dimensionale Nachbarschaft) gehängt. Ein selektiertes, aber ausgeschaltetes Neuron erzeugt dabei keine Gewichtsänderungen im Netz und wird bei der nächsten Eingabe wieder eingeschaltet. Damit wird sichergestellt, dass ein dupliziertes Neuronenpaar nicht gleichartig iteriert wird, sondern durch die unterschiedliche Nachbarschaft die Gewichtsänderungen verschieden ausfallen.
Diese Kette ("Gummiband") wird nun Schritt für Schritt für die eingegebenen Städte erweitert, so dass am Ende der Iteration für jede Stadt ein formales Neuron existiert: der zwei-dimensionale Eingaberaum (Menge aller Städte) wird auf eine ein-dimensionale Kette (Reiseroute) abgebildet, wobei die Nachbarschaftsbeziehungen (benachbarte Städte sind auch in der Kette benachbart) so gut wie möglich (mit dem kleinsten quadratischen Fehler) erhalten bleiben.

Die Lerngleichung für das i-te von insgesamt *m* Neuronen lautet hier

$$\mathbf{w}_i(t) = \mathbf{w}_i(t\text{-}1) + h(\sigma,d)\,(\mathbf{x}\text{-}\mathbf{w}_i) \tag{4.45}$$

mit der Gauß'schen Nachbarschaftsfunktion $h(\sigma,d) := (1/2^{1/2})\,\exp(-d^2/\sigma^2)$ von dem Index-Abstand $d := \inf(\,|i\text{-}i_c|,\, m\text{–}|i\text{-}i_c|\,)$.

Nach jedem Trainingsmuster-Zyklus wird die Nachbarschaft verkleinert, ähnlich wie in (4.28). Um "Konzentrationen" von Neuronengewichten in "leeren" Gebieten zu vermeiden, existiert außerdem noch ein Löschmechanismus, bei dem jedes Neuron, das in 3 aufeinanderfolgenden Zyklen von keiner Stadt ausgewählt wurde, aus dem Ring entfernt wird. In der Abb. 4.15 ist die Konvergenz eines Rings aus 30 Neuronen auf 30 Städte gezeigt.

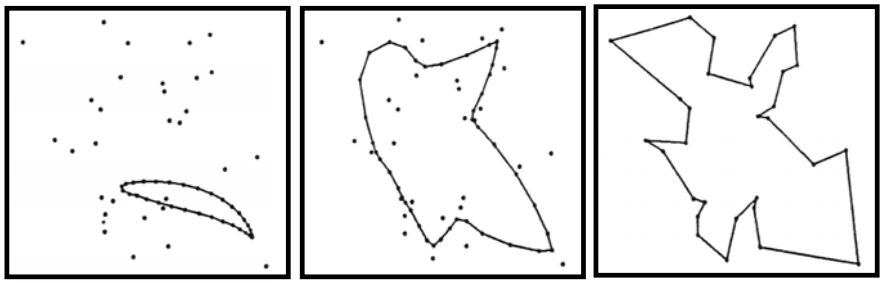


Abb. 4.15 *Nachbarschafts-erhaltende Abbildung einer Reiseroute (aus [ANG88])*

Das Elastische Netz

Betrachten wir wieder eine Anordnung der Neuronen zu einer 1-dim. Kette, wobei jeweils ein Neuron einer Stadt zugeordnet wird. Das Netz führt dabei die Aufgabe durch, die 2-dim. Menge der Stadtkoordinaten $\{\mathbf{x}_i\}$ durch eine 1-dim. Kette $\{\mathbf{w}_i\}$ möglichst gut zu approximieren. In diesem Kontext bedeutet "gut", wieder möglichst viel von den Nachbarschaftsbeziehungen zu berücksichtigen, also in der Ebene benachbarte Städte auch auf der Kette benachbart sein zu lassen.
Gehen wir von der einfachen Lerngleichung der *Kohonen map* aus

$$\mathbf{w}_k\,(t+1) = \mathbf{w}_k\,(t) + \gamma(t+1)\,h(c,k,t)\,[\mathbf{x} - \mathbf{w}_k(t)] \qquad (4.46)$$

Hierbei erlaubt zum einen eine Nachbarschaftsfunktion h(.) das Lernen nur in der fest definierten Nachbarschaft von Neuronen *k*, zum anderen wird zuerst über eine *winner-take-all*-Entscheidung das aktivste Neuron ausgesucht.

Ermitteln wir die Entscheidung über das aktivste Neuron durch die stetige Softmax-Regel von $y_{kj} = S_k(\mathbf{x}_j)/\Sigma_i S_i(\mathbf{x}_j)$ mit der Gauß'schen Ausgabefunktion $S_k = \exp(-(\mathbf{x}_j - \mathbf{w}_k)^2/\sigma_k^2)$, so generiert dies einen hohen Wert nur in der Umgebung von $\mathbf{w}_k$. Für ein System mit der Neuronenaktivität y_{kj} beim j-ten Eingabemuster $\mathbf{x}_j$ lässt sich deshalb die Nachbarschaftsfunktion direkt durch die Ausgabe y_{kj} ersetzen und aus Gl.(4.46) wird

$$\mathbf{w}_k\,(t+1) - \mathbf{w}_k\,(t) = \Delta\mathbf{w}_k\,(t) = \alpha\, y_{kj}\,[\mathbf{x}_j - \mathbf{w}_k(t)]$$

Da alle Städte $\mathbf{x}_j$ Einfluß auf die geschätzten Stadtkoordinaten, die Gewichte $\mathbf{w}_j$, haben sollen, erhalten wir

$$\Delta\mathbf{w}_k\,(t) = \alpha \sum_j y_{kj}\,[\mathbf{x}_j - \mathbf{w}_k(t)] \qquad (4.47)$$

Im Unterschied zu der *Kohonen map* Gl. (4.46) haben wir allerdings keinerlei feste Nachbarschaft der Neuronen definiert. Um trotzdem eine Kette zu erhalten, definieren wir dafür einen Fehler $R_j = \Sigma_j(\mathbf{w}_{j+1} - \mathbf{w}_j)^2$ der Nachbarabstände, der ebenfalls klein wer-

den soll, und fügen den negativen Gradienten $-\partial R_j/\partial \mathbf{w}_k$ der Lerngleichung (4.47) hinzu, so dass sowohl der mittlere Abstand der Gewichtsvektoren zu den Städten als auch unter einander minimal wird

$$\Delta \mathbf{w}_k\,(t) = \alpha \sum_j y_{kj}\,[\mathbf{x}_j - \mathbf{w}_k(t)] + \sigma\beta\,(\mathbf{w}_{k+1}+\mathbf{w}_{k-1}-2\mathbf{w}_k) \qquad (4.48)$$

Reduzieren wir im Laufe der Iteration die Weite σ der RBF, so wird auch der korrektive Einfluß der Nachbarn von $\mathbf{w}_k$ reduziert und die Kette, die am Anfang als kleiner Ring vorhanden ist, kann sich am Ende "voll ausdehnen" und ausspezialisieren. Dieses Modell mit der Lernregel Gl.(4.48) heißt deshalb *elastisches Netz* [DUW87] mit dem "Elastikparameter" β. Für M = 100 verschiedene Städte ist eine Iteration mit dem Lernalgorithmus in Abb. 4.16 gezeigt.

Abb. 4.16 *Approximation der optimalen Städtetour durch das Elastische Netz mit den Parametern $\sigma(0) = 0{,}2$, $\sigma(7000) = 0{,}01$,$\alpha = 0{,}2$,$\beta = 2{,}0$*

Gezeigt sind die drei verschiedenen Zustände bei σ = 0,2 , σ = 0,13 und σ = 0,01, wobei σ(t) nach jeweils Δt = 25 Iterationen um 1% verringert wurde.

4.4.2 Spracherkennung

Eine der interessantesten Anwendungen, die von Kohonen selbst initiiert wurde, ist die Abbildung von Sprachlauten auf sprachliche Merkmale. Die digitalisierten Sprachlaute (physikalischen Schwingungen) werden dazu alle 10ms einer Fouriertransformation (26ms-Intervall) unterworfen. Das entstandene Kurzzeitspektrum wird in 15 Frequenzbereiche unterteilt. Die Intensität jedes Frequenzbereichs wird als eine der Eingabekomponenten für ein neuronales Netz betrachtet, so dass alle 10ms ein neuer, insgesamt 15-dim Eingabevektor $\mathbf{x}(t)$ zur Verfügung steht. Für alle M = 21 finnischen Phoneme (die vorher bekannt waren!) wurden aus Sprachproben der Phoneme für je 50 Traingsmuster der Klassenprototyp $\mathbf{x}^1,..,\mathbf{x}^M$ ermittelt und ein Netzwerk aus Neuronen mit 2-dim

Nachbarschaft mit dem sequentiellen, nachbarschaftserhaltenden Algorithmus von Gl.(4.28) des vorigen Abschnitts trainiert. Als Ergebnis formte sich eine 2-dim Karte, die an verschiedenen Stellen auf verschiedene Phoneme besonders stark ansprach, s. Abb. 4.17.

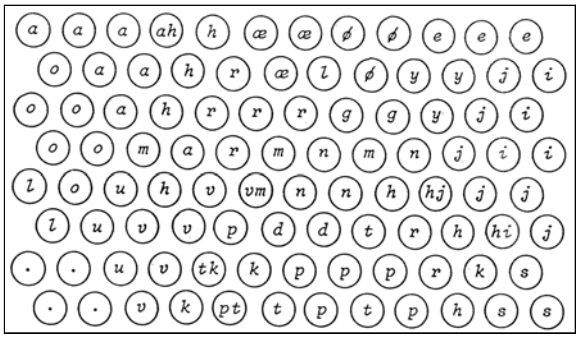


Abb. 4.17 *Phonemkarte des Finnischen (nach [KOH88])*

Interessanterweise gibt es Einheiten, die keinem der vorgegebenen Phoneme eindeutig zuzuordnen sind und den Übergang zwischen den Lauten andeuten.

Eine weitere Besonderheit in diesem System stellt die fehlende Zeitrepräsentation dar. Die Sprache wird als "Folge von sequentiellen Einzelphonemen" betrachtet, was in der Phonemkarte als Weg erscheint. In Abb. 4.18 ist eine solche Sequenz für das Wort /*humppila*/ eingezeichnet; jeder Pfeil zeigt von der Koordinate des für **x**(t) ermittelten Neurons auf die nächste Koordinate des Neurons für **x**(t+10ms). Die Repräsentation der drei tonlosen Konsonanten /k,p,t/, die sich durch ihre zeitlichen Frequenzschwankungen charakterisieren, lassen sich deshalb mit der Phonemkarte nicht darstellen und müssen, abgesetzt von den 19 verbleibenden Phonemen, mit konventionellen Methoden gesondert extrahiert und erkannt werden.Mit dem vorgestellten System wurde also eine Abbildung zwischen gesprochener Sprache und der dazu gehörenden Lautschrift gelernt. Würde man nun noch zusätzlich die Zuordnung zwischen Lautschrift und geschriebenem Wort (Umkehrung von NETtalk, siehe Kapitel „Backpropagation“) lernen, so hätte man im Prinzip eine "Neuronale Schreibmaschine".

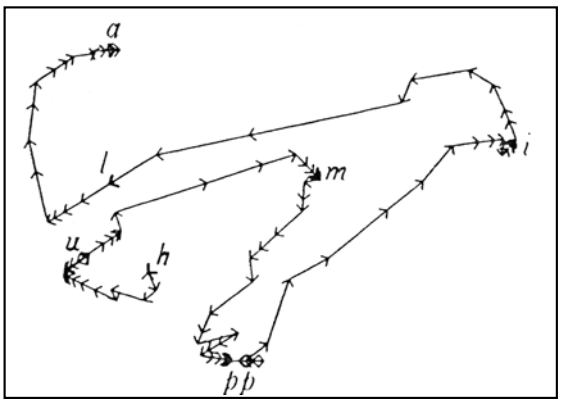


Abb. 4.18 *Phonemsequenz für /humppila/ (nach [KOH88])*

In der Praxis wird dies mit Systemen durchgeführt, die den gesamten Wortschatz gespeichert haben und ihn mit dem unbekannten Wort vergleichen können. Im Unterschied zum Finnischen ergibt sich nämlich normalerweise die Schreibweise nicht automatisch aus dem Gesprochenen, sondern ist etwas assoziativ-willkürlich Gelerntes, das wir uns erst mühsam im Laufe der Jahre eingeprägt haben. Beispielsweise existieren im TANGORA System von IBM ca. 20.000 Wortformen, die zusätzlich über assoziierte Wahrscheinlichkeiten nach dem "Hidden-Markov-Modell" miteinander kombiniert erkannt werden können [SPI93]. Diese Funktion ist in dem Vektorquantisierungsschritt von Kohonen nicht enthalten und ist in ihrer Komplexität nicht zu unterschätzen.

4.4.3 Lokal verteilte, konkurrente Sensorkodierung

Angenommen, wir haben ein gleichmäßiges Feld von Sensorelementen, etwa die Sensorpunkte eines Bildverarbeitungschips. Zwischen die Sensorpunkte platzieren wir nun informationsverarbeitende Elemente, etwa formale Neuronen, und verbinden jedes Neuron mit den Sensorpunkten seiner lokalen Nachbarschaft. In Abb. 4.19 ist ein solches Schema gezeigt, das sich durch seine mögliche Wiederholung gut als Schablone für eine VLSI-Fertigung eignet. Im Mittelpunkt ist ein Neuron als Viereck eingezeichnet; seine Verbindungen in der Nachbarschaft als dicke Linien. Ein achteckiger Ring schließt alle Sensorpunkte der Nachbarschaft ein. Man sieht, dass durch repetitive, überlappende Anordnung der Nachbarschaft die gesamte Sensorfläche abgedeckt wird.

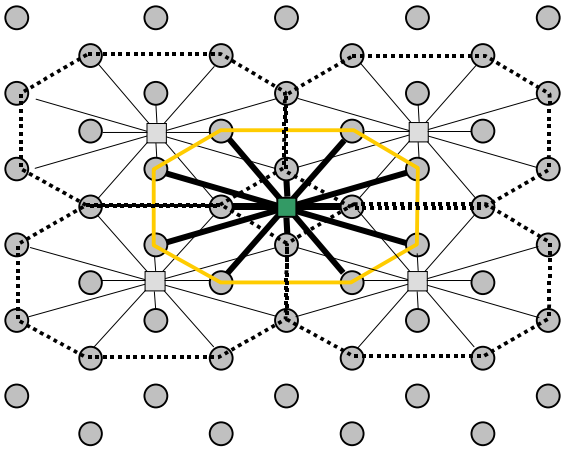


Abb. 4.19 *Repetitive Anordnung von Zellen*

Ein Neuron, seine Gewichte, seine Aktivitäts- und Lernfunktion bildet eine Zelle, die man räumlich abgrenzen kann. Alle Zellen zusammen ergeben ein „zellulares, neuronales Netz“ (*Cellular Neural Net* CNN), eine informationsverarbeitende Maschine, die durch ihre Parallelverarbeitung der Sensoreingabe Zehnerpotenzen an Rechenleistung eines sequentiellen Computers erreichen kann.

Als Beispiel für diese Klasse von neuronalen Netzen wollen wir ein selbstorganisierendes System betrachten, das zusätzlich zu der Sensoreingabe auch eine Hemmung der Neuronen in nächster Nachbarschaft vorsieht entsprechend dem Modell nach Abb. 4.8. Das Lernen der Neuronen soll als Ziel den quadratischen Fehler der Kodierung minimieren. Dies entspricht den Lernregeln des lateral inhibierenden Netzes aus Abschnitt 3.3, bei dem der Gewichtsvektor jedes Neurons zum größten Eigenvektor der Eingabevarianz konvergiert. Die entsprechenden Lerngleichungen

$$\mathbf{w}_i(t) = \mathbf{w}_i(t-1) - \gamma(t)\mathbf{x}\left(\beta \sum_{j \neq i} u_{ij} y_i\right) \tag{4.49}$$

und

$$u_{ij}(t) = u_{ij}(t\text{-}1) - 1/t\ (u_{ij}(t\text{-}1) + y_i y_j) \tag{4.50}$$

wirken allerdings bei dieser Anwendung nur auf die lokalen Gewichte $\{u_{ij}\}$, die den Einfluss der Nachbarn auf ein Neuron modellieren.

Was können wir von solch einem System erwarten? In Abb. 4.20 sind zwei mögliche Nachbarschaften als Graphen sowie die Simulationsergebnisse gezeigt. Die Doppelpfei-

le symbolisieren die Wechselwirkungen zwischen Nachbarn durch die *Anti-Hebb*-Regel; die Zahlen sind die Nummern der Eigenvektoren, zu denen der Gewichtsvektor des Neurons konvergiert.

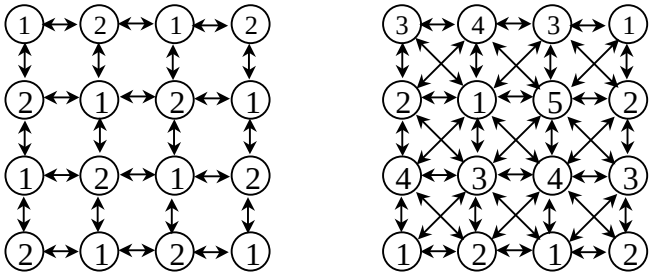
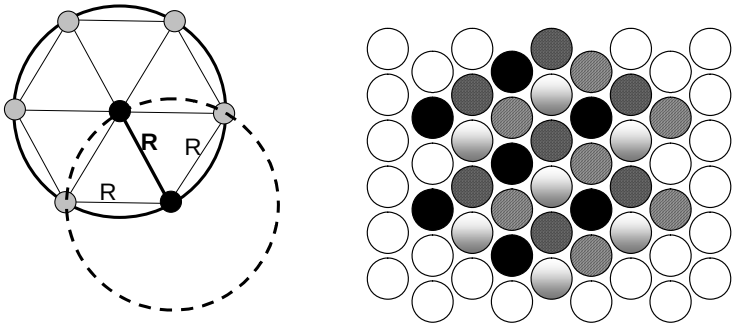


Abb. 4.20 *Verschiedene inhibitorische Nachbarschaftsradien* r *und ihre Simulationswirkung*

Bei geringer Nachbarschaft mit R = 1 beobachten wir, dass die Gewichtsvektoren wechselseitig den Eigenvektor mit dem größten Eigenwert und mit dem zweitgrößten Eigenwert, symbolisiert durch die Zahlen 1 und 2, lernen. Dies ist durchaus verständlich: Zum einen konvergiert jedes Neuron zum Eigenvektor mit dem größten Eigenwert, zum anderen verhindert die inhibitorische Lernregel, dass zwei Neuronen mit dem selben Eigenvektor Nachbarn sind. Für den indirekten Nachbarn, zu dem keine direkte Verbindung existiert, sind damit keine Einschränkungen vorhanden, so dass er zu dem selben Eigenvektor konvergieren kann. Die Gesamtkonfiguration ist damit bei einem alternierenden Muster stabil.

Wie kann man nun das Ergebnis bei R = 1,5 interpretieren? Die erweiterte Nachbarschaft verhindert in einem Gebiet alle weiteren Eigenvektoren mit dem selben Eigenwert. In Abb. 4.21 (a) ist dies als Kreis um ein Neuron in der Mitte symbolisiert. Innerhalb des Kreises kann kein weiteres Neuron mit diesem Eigenvektor als Gewichtsvektor liegen, nur außerhalb und auf dem Kreis. Nehmen wir ein weiteres Neuron mit diesem Konvergenzziel an, so bildet es den Abstand *R* zum ersten Neuron. Ein drittes Neuron mit dem gleichen Eigenvektor ist ebenfalls möglich, aber nur im Abstand R zum zweiten Neuron, so dass diese drei Neuronen ein gleichschenkliges Dreieck bilden. Ergänzen wir dieses Bild durch weitere Neurone, so sind insgesamt sechs weitere Neurone möglich, die eine stabile Konfiguration in Form eines Sechsecks um das zentrale Neuron bilden. Man beachte, dass die resultierende Geometrie unabhängig von der Anzahl der möglichen Neuronenpositionen ist und durch die endliche Nachbarschaftskonkurrenz bedingt wird.



(a) Konvergenz zum gleichen EV (b) Konvergenz zu verschiedenen EV

Abb. 4.21 *Die selbstorganisierende Hexstruktur der Sensorkodierung*

Wir können nun noch weitere Neuronenpositionen innerhalb des Kreises annehmen, auf denen Neuronen sitzen, die andere Eigenvektoren, etwa den Eigenvektor mit dem zweitgrößten Eigenwert, lernen. Für sie gilt das gleiche wie für den ersten Eigenvektor, so dass auch für sie eine Hexstruktur resultiert. In Abb. 4.21 (b) ist dies an einer Beispielstruktur gezeigt. Hier symbolisieren die schwarzen Punkte Neuronen mit einem Gewichtsvektor, der zum Eigenvektor mit den ersten Eigenwert konvergiert sind, die grauen Punkte Neuronen mit dem zweiten Eigenwert usw.

Das gesamte, konvergierte Netz kodiert also die Sensorpunkte als eine Eigenvektorentwicklung oder PCA, siehe Kapitel 3. Dabei gehören immer nur die Komponenten aus einer Nachbarschaft zusammen, wobei die Nachbarn an jedem Punkt des Sensorfeldes neu bestimmt werden müssen. Die Kodierung ist also lokal verteilt. Ein ähnliches Schema ist die Kodierung eines Fernsehbildes aus blauen, roten und grünen Leuchtpunkten: Zusammen ergeben sie den lokalen Helligkeitswert, wobei die zusammengehörenden Komponenten von Ort zu Ort wechseln.

4.4.4 Robotersteuerung

Eine Anwendung der nachbarschafts-erhaltenden Abbildungen ist in der Steuerung von Robotermanipulatoren zu finden. Wie wir aus Abb. 4.6 sehen können, treten interessanterweise auch bei der menschlichen Muskelsteuerung nachbarschafts-erhaltende Abbildungen auf. Dabei lassen sich die verschiedenen Kontroll- und Sensorschichten der Modelle der menschlichen Muskelkontrolle den Schichten der Roboterkontrolle gegenüberstellen. In Abb. 4.22 ist dies für eine relativ grobe Modellierung gezeigt.

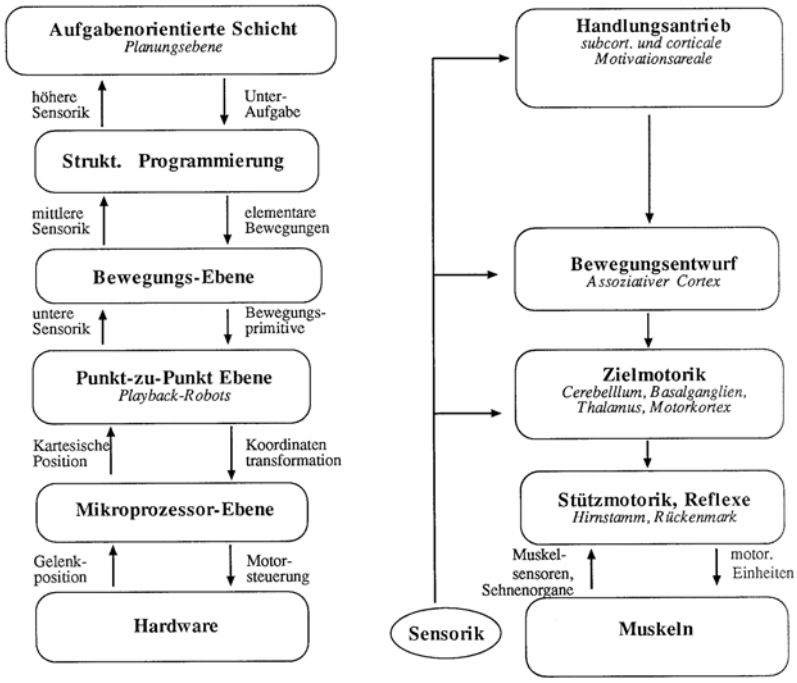


Abb. 4.22 *Muskelkontrolle und Roboterarmkontrolle*

Für die Roboterkontrollschicht der direkten Positionskontrolle lässt sich nun ein direkter Ansatz angeben, um mit Hilfe der nachbarschafts-erhaltenden Abbildungen eine Kontrolle der Gelenkwinkel der Manipulatorgelenke im kartesischen Raum zu erlernen, anstatt sie durch analytisch-geometrische Transformationen zu errechnen.

Angenommen, wir betrachten einen Roboterarm aus mehreren Segmenten, ähnlich den des bekannten PUMA Roboters, s. Abb. 4.23. Das Grundproblem der statischen Positionskontrolle oder *Kinematik* besteht darin, bei vorgegeben Gelenkwinkeln θ_i die Arm- bzw. Handwurzelposition **x** zu errechnen, und ist relativ leicht zu lösen. Ordnen wir jedem Manipulatorsegment eine sog. *homogene Transformation*smatrix **T** [DEN55] zu, mit der für jedes Segment die Bewegungen im Raum (Translationen und Rotationen) beschrieben werden, so können wir die tatsächliche kartesische Position der Hand **x** direkt durch sukzessives Ausmultiplizieren der Transformationsmatrizen aller Segmente erhalten.

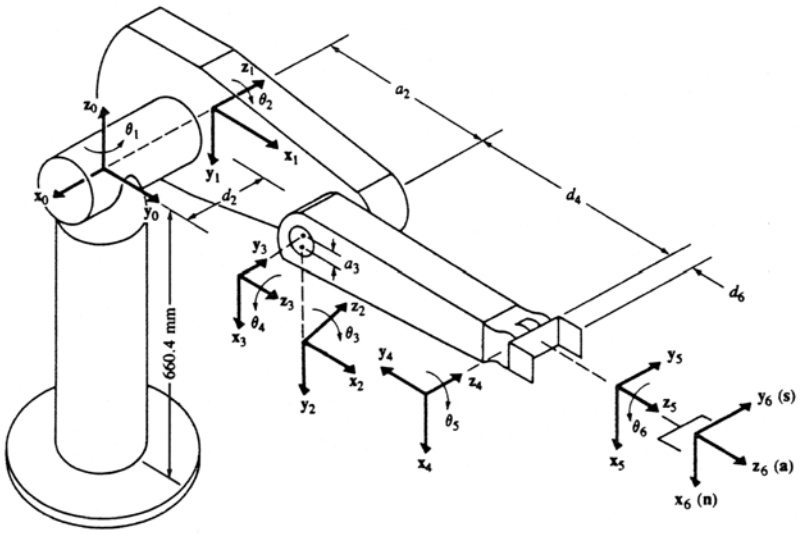


Abb. 4.23 *Der PUMA Roboter (nach [FU87])*

Jede Transformationsmatrix reflektiert dabei die Geometrie des Segments, zu der sie gehört.

$$\mathbf{x} = \mathbf{T}_1 \cdot \mathbf{T}_2 \cdot \cdot \mathbf{T}_n \, \mathbf{a} \qquad \textit{Kinematik} \tag{4.51}$$

Das umgekehrte Problem, aus der vorgegeben kartesischen Position **x** die Gelenkwinkel θ_i zu bestimmen

$$\theta_i = f_i(\mathbf{x}) \qquad \textit{inverse Kinematik} \tag{4.52}$$

ist dagegen wesentlich schwerer analytisch zu lösen. Jede Armgeometrie benötigt eine besondere Lösung, die auch nur mit akzeptablen Aufwand zu erreichen ist, wenn die Armgeometrie bestimmten Restriktionen (parallele oder orthogonale Gelenkachsen) erfüllt.

Dieses Problem lässt sich vermeiden, indem die Abbildung der inversen Kinematik nicht analytisch errechnet, sondern mit den nachbarschafts-erhaltenden Abbildungen gelernt wird. Dazu unterteilen wir die Armbewegung in zwei Teile: eine *grobe* Armbewegung, die im wesentlichen die nicht-lineare inverse Kinematik beinhaltet, und in eine *feine* Armbewegung, die mit Hilfe der gewünschten Kartesischen Position (Eingabemuster) zwischen den groben Positionsrastern linear interpoliert.

Die grobe Positionierung

Für die grobe Schätzung der Parameter wird der gesamte Arbeitsraum des Roboterarms in Raumzellen (Punktmengen) unterteilt, siehe Abb. 4.24. Jeder Zelle wird ein Neuron mit seinem Gewichtsvektor zugeordnet, wobei der Gewichtsvektor **w** den Mittelpunktskoordinaten der Zelle entspricht. Jedes Neuron v der drei-dimensionalen Nachbarschaft ist durch sein Tripel $\mathbf{v} = (i_1,i_2,i_3)$ von Indizes innerhalb des Netzes gekennzeichnet. In Abb. 4.24 ist dies am Beispiel des Neurons (7,5,5) illustriert.

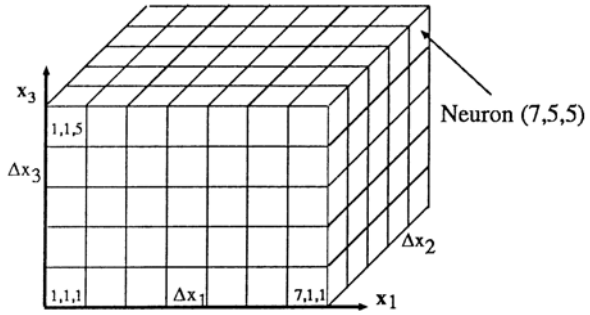


Abb. 4.24 *Aufteilung des Arbeitsraums*

Zu jedem Neuron v_k mit dem Gewichtsvektor $\mathbf{w}_k$ des kartesischen Raums sei auch ein entsprechender Vektor $\Theta_k = (\theta_1,\theta_2,\theta_3)$ des Gelenkkoordinatenraums assoziiert. Wird nun eine Kartesische Position $\mathbf{x}_{soll}$ gewünscht, so wird wie beim Kohonen-Algorithmus (4.25) mit einer *winner-take-all*-Entscheidung festgestellt, welches Neuron v_c "Gewinner" ist und damit die Abbildung der inversen Kinematik vollbringt. Mit dem Index *c* lässt sich sofort auf die Position Θ_c in Gelenkkoordinaten schließen, so dass die Transformation der inversen Kinematik hier nur aus einer einfachen, schnellen Tabellenausleseoperation besteht.

Allerdings ist die Realisierung nicht unproblematisch: Wählen wir eine kleine Zahl von Neuronen, etwa einen würfelförmigen Arbeitsraum mit Kantenlänge 70 cm unterteilt in 10×10×10 = 1000 Neuronen, so entsteht ein großer, nichtakzeptabler Positionierungsfehler (ca.12 cm); für eine ausreichende Genauigkeit (0,121mm) müssen dagegen sehr viele Neuronen (ca. 10^{12}) vorhanden sein, was zu erheblichem Speicherbedarf (12 Gigabyte!) für die Tabelle aller $\mathbf{w}_k$ und Θ_k führt. Eine Möglichkeit, diese Probleme zu umgehen, besteht in der Benutzung einer groben Tabelle und einer anschließenden linearen Korrektur der groben Bewegung.

Die Feinpositionierung

Wie schon Ritter und Schulten in [RITT3] vorschlugen, lässt sich der Fehler, der durch die Aufteilung des Arbeitsraums in endliche Zellen gemacht wird, durch eine Interpolation (lineare Approximation) für die Gelenkposition $\Theta(\mathbf{x})$ innerhalb einer Zelle verkleinern:

$$\Theta(\mathbf{x}) = \Theta_c + \Delta\Theta = \Theta_c + \mathbf{A}_c\,(\mathbf{x}\text{-}\mathbf{w}_c) \tag{4.53}$$

Dabei ist $\mathbf{A}_c$ eine Matrix, deren Koeffizienten für jedes Neuron extra gelernt werden muss. Zwar bedeutet das auch zusätzlichen Speicherbedarf pro Neuron, es verringert aber trotzdem den Gesamtspeicherbedarf, da so eine geringere Zahl von Neuronen notwendig ist, um den gleichen geringen Fehler zu erhalten.

Das gesamte Netzwerk besteht aus zwei Schichten, wobei die erste Schicht die Grobpositionierung durchführt und die zweite Schicht für die Feinpositionierung zuständig ist. In Abb. 4.25 ist dies schematisch gezeigt.

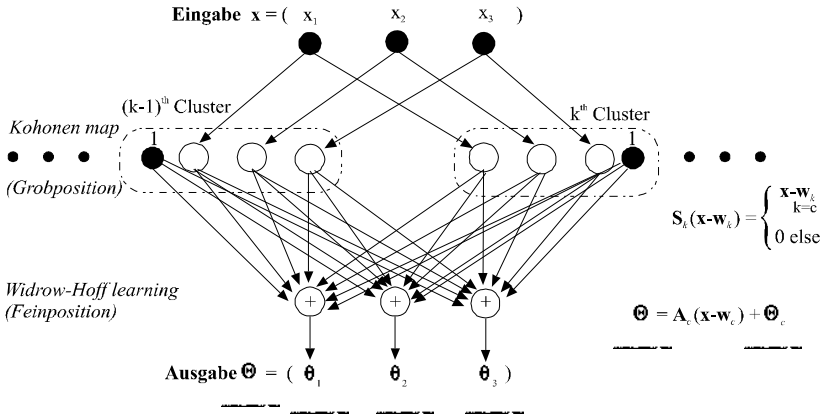


Abb. 4.25 *Das Approximationsnetzwerk für die Robotersteuerung* (nach [BRA93c])

Der Lernalgorithmus

Das Lernen der Gewichte für $\mathbf{w}_k$ geschieht entweder durch direkte Zuweisung zu einem regelmäßigen Gitter, oder mittels stochastischer Approximation der beobachteten kartesischen Positionsanforderungen.

Die übrigen Werte Θ_k und $\mathbf{A}_k$, also die 3 Gelenkkoordinaten vom Spaltenvektor Θ_k und die 9 Matrixkoeffizienten A_{11}, ..., A_{33}, fassen wir zu der allgemeinen 3×4 Gelenkparametermatrix $\mathbf{U}_k = (\mathbf{A}_k,\Theta_k)$ zusammen. Angenommen, wir erweitern die Spaltenvektoren

$\Delta\mathbf{x}$:= $(\mathbf{x}\text{-}\mathbf{w}_c)$ mit einer weiteren Komponente zu $\Delta\mathbf{x}\rightarrow(\Delta\mathbf{x}^T,1)^T$, so lässt sich der lineare Ansatz (4.53) auch als

$$\Theta = \Delta\Theta + \Theta_c = (\mathbf{A}_c,\Theta_c)(\mathbf{x}^T\text{-}\mathbf{w}_c^T,1)^T = \mathbf{U}_c\Delta\mathbf{x} \tag{4.54}$$

formulieren. Die Matrix **U** lässt sich koeffizientenweise folgendermaßen iterieren:

$$\mathbf{U}_c(t+1) = \mathbf{U}_c(t) + h(.)\gamma(t+1)[\mathbf{U}_c^* - \mathbf{U}_c(t)] \tag{4.55}$$

mit der Nachbarschaftsfunktion h(.) und der (t+1)-ten Schätzung $\mathbf{U}_c^*$ von $\mathbf{U}_c$.
Wie kommen wir aber zu einer guten Schätzung $\mathbf{U}_c^* = (\mathbf{A}_c^*,\Theta_c^*)$?

Aus Kapitel 2.2 kennen wir eine gute Lernregel, die den quadratischen Fehler zwischen der Vorgabe L und der tatsächlichen Ausgabe $\mathbf{w}^T\mathbf{x}$ minimiert

$$\mathbf{w}(t) = \mathbf{w}(t\text{-}1) - \gamma(t)\frac{(\mathbf{w}^T\mathbf{x} - L)\mathbf{x}}{\|\mathbf{x}\|^2} \qquad \textit{Widrow-Hoff Lernregel} \tag{4.56}$$

In der Gesamtmatrix **W** der Gewichte bilden die Gewichtsvektoren $\mathbf{w}^T$ die Zeilen, so dass (4.55) transponiert für eine ganze Matrix **W** lautet

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \gamma(t+1)\frac{(\mathbf{W}\mathbf{x} - \mathbf{L})\mathbf{x}^T}{\|\mathbf{x}\|^2} \tag{4.57}$$

Die Korrektur der Matrix **W** erfolgt hier durch den Fehler **Wx–L** .

Ersetzen wir die Lernrate $\gamma(t+1)$ durch die mit der Nachbarschaftsfunktion kombinierte Lernrate $h(.)\gamma(t+1)$, so erhalten wir mit der Notation $\mathbf{W}\rightarrow\mathbf{U}_c$, $\mathbf{x}\rightarrow\Delta\mathbf{x}$ durch die Gleichsetzung des Fehlers $\mathbf{U}_c^*-\mathbf{U}_c(t)$ aus Gl.(4.55) mit dem Fehler $-[\mathbf{U}_c\Delta\mathbf{x}-\mathbf{L}]\,\Delta\mathbf{x}^T/|\Delta\mathbf{x}|^2$aus Gl.(4.57)die Schätzung mit dem kleinsten quadratischen Fehler für die Gelenkparameter

$$\begin{aligned}\mathbf{U}_c^* &= \mathbf{U}_c(t) + [\mathbf{L}\text{-}\mathbf{U}_c\Delta\mathbf{x}]\,\Delta\mathbf{x}^T/|\,\Delta\mathbf{x}|^2 \\ &= \mathbf{U}_c(t) + [\Delta\Theta_{soll}\text{-}\Delta\Theta_{ist}]\,\Delta\mathbf{x}^T/|\,\Delta\mathbf{x}|^2\end{aligned} \tag{4.58}$$

mit der gewollten Bewegung $\mathbf{L} = \Delta\Theta_{soll}$ und der tatsächlichen Näherung $\Delta\Theta_{ist} := \mathbf{U}_c\Delta\mathbf{x}$. Angenommen, wir können außer der tatsächlichen Endposition $\mathbf{x}_F$ auch die tatsächliche Position $\mathbf{x}_I$ nach der groben Bewegung feststellen. Dann kann man die beobachtete Bewegung $(\mathbf{x}_F\text{-}\mathbf{x}_I)$ als die Eingabe für das lineare System auffassen, die durch $\mathbf{U}_c$ bzw. $\mathbf{A}_c$ auf die Ausgabe $\Delta\Theta_{ist} := \mathbf{A}_c(\mathbf{x}_F\text{-}\mathbf{x}_I)$ linear abgebildet würde. Stattdessen aber wurde

$$\Delta\Theta_{soll} := \mathbf{A}_c^*(\mathbf{x}_F\text{-}\mathbf{x}_I) = \mathbf{A}_c(\mathbf{x}\text{-}\mathbf{w}_c)$$

verlangt, so dass aus Gl. (4.58) für $\mathbf{A}_c^*$ die Iteration

$$\mathbf{A}_c^* = \mathbf{A}_c + \mathbf{A}_c[\,(\mathbf{x}\text{-}\mathbf{w}_c) - (\mathbf{x}_F\text{-}\mathbf{x}_I)\,]\,(\mathbf{x}_F\text{-}\mathbf{x}_I)^T/\,|(\mathbf{x}_F\text{-}\mathbf{x}_I)|^2 \tag{4.59}$$

wird. Ist die Matrix $\mathbf{A}_c$ gut gelernt, so lässt sich eine Schätzung von Θ_c beispielsweise durch den restlichen Fehler $\Delta\mathbf{x}\rightarrow(\mathbf{x}\text{-}\mathbf{x}_F)$ zwischen gewünschter Position $\mathbf{x}$ und tatsächlicher Position $\mathbf{x}_F$ erreichen.

$$\Theta_c^* = \Theta_c + \mathbf{A}_c(\mathbf{x}\text{-}\mathbf{x}_F) \tag{4.60}$$

Die Iteration nach Gl. (4.60) wird deshalb erst nach einigen Iterationsschritten für $\mathbf{A}_c^*$ begonnen.

Eine ausführliche mathematische Behandlung und viele interessante Simulationen dieses Ansatzes sind in dem empfehlenswerten Buch von H.Ritter, T.Martinetz und K.Schulten [RITT90] zu finden.

Ausblick

In den vorigen Abschnitten wurde gezeigt, wie sich nachbarschafts-erhaltende Abbildungen zur Robotersteuerung einsetzen lassen. Die analytisch fundierte Roboterpositionierung wird bei dem vorgestellten Verfahren durch das Lernen einer Positionierungstabelle ersetzt. Dies hat folgende Vorteile:

- Wie bei allen tabellierten Funktionen wird die *Kontrolle sehr schnell*, da das Ausrechnen der Funktionen durch ein Nachschlagen in Tabellen (*memory mapping*) ersetzt wird.
- Da *keine analytischen Lösungen* für die inverse Kinematik benötigt werden, lassen sich mit dieser Methode auch sehr unkonventionelle Architekturen steuern, bei denen die Gelenkachsen nicht orthogonal oder parallel zueinander orientiert sind oder die mehr Gelenke (Freiheitsgrade) als Kartesische Koordinaten besitzen. Dies bedeutet auch, dass Veränderungen der Armgeometrie für den Einsatz beim Benutzer keine umständlichen, zeitraubenden und kostspieligen Rückfragen beim Hersteller nötig machen; der Benutzer kann ohne tiefere Roboterkenntnisse die Manipulatorgeometrie direkt seinen Bedürfnissen anpassen.
- Die *Auflösung* (Neuronendichte) der nachbarschafts-erhaltendenden Abbildung ist ortsabhängig und hängt von der Zahl der Trainingspositionierungen (Wahrscheinlichkeitsdichte) ab. Der Lernalgorithmus ermöglicht damit automatisch eine Anpassung der Positionierungsauflösung an den eigentlichen Einsatzort. Dabei lassen sich unproblematisch auch Nebenbedingungen (minimale Energie, minimale Abweichung vom mittleren Winkel) einführen, indem die Lerngleichungen (4.56) bzw. (4.58) entsprechend angepasst werden. Als Beispiel dafür ist die gelernte 2D-Positionierung für eine häufig aufgesuchte Kreisfläche gezeigt. Die Knoten zeigen die Gewichtspositionen, die Kanten die Nachbarschaften.

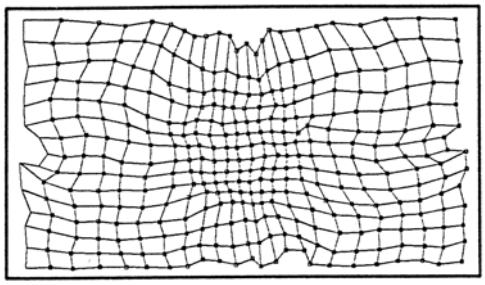


Abb. 4.26 *Häufigkeitsangepasste Eingabe (nach [RITT90])*

- Die *Lerngeschwindigkeit* ist durch die Nachbarschaft deutlich *beschleunigt*: Da zusätzlich zum Gewinner auch die Nachbarn lernen, lernt jedes Neuron bei gleicher Ereigniszahl mit Nachbarschaft öfters als ohne Nachbarschaft. Dies ist in Abb. 4.27 gezeigt. Hier ist der Fehler in Abhängigkeit von der Zahl der Lernschritte aufgetragen. Man sieht, dass bei einer Nachbarschaft von $\sigma = 2$ der Fehler des Netzes deutlich schneller absinkt als wenn mit $\sigma = 0$ nur ein einziges Neuron lernt.

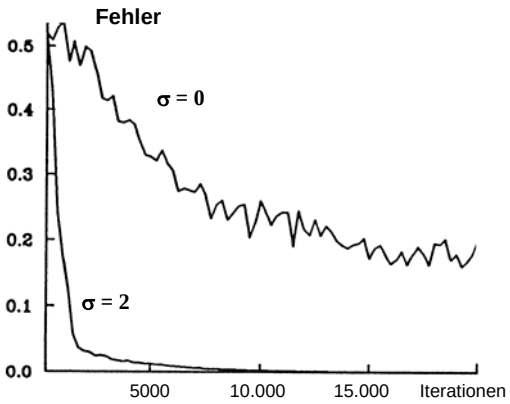


Abb. 4.27 *Lerngeschwindigkeit und Nachbarschaft(nach [Ritt90])*

Allerdings sind auch einige Nebenbedingungen zu beachten:

- Ein charakteristisches Problem von tabellierten Funktionen ist der *Speicherbedarf*, um eine gute Auflösung zu erreichen. Auch diese speziellen, nachbar-

schafts-erhaltenden Abbildungen bilden dabei, wie gezeigt wurde, keine Ausnahme. Trotzdem scheint es durch einen speicheroptimierten Ansatz [BRA89b] [BRA93c] möglich, bei realistischen Problemen durch Bestimmung der optimalen Zahl von Neuronen und der Auflösung der Gewichte mit relativ moderaten Speichergrößen auszukommen.

- Ein weiteres wichtiges Problem ist die *Zeit*, um den Algorithmus durchzuführen. Nehmen wir beispielsweise die obige Konfiguration mit ca. n = 40 Neuronen in jeder Dimension, also insgesamt 64000 Neuronen.

  Betrachten wir nur den Aufwand für die Selektion eines Neurons. Benutzen wir den sequentiellen Algorithmus, so muss für jede Positionierung vorher das Abstandsminimum der Position **x** zu allen 64000 Gewichtsvektoren gesucht werden, was einen großen Rechenaufwand bedeutet. Nimmt man anstelle einer beliebigen Wahrscheinlichkeitsverteilung p(**x**) dagegen eine Gleichverteilung an und verzichtet auf eine problemangepasste Auflösung, so kann man die Minimumsuche bei den resultierenden, gleichartigen Zellen durch das direkte Ausrechnen des Neuronenindexes ersetzen.

  Ein Ausweg aus diesem Dilemma bietet ein paralleler Algorithmus, der eine parallele Operation von vielen einfachen, in Hardware implementierten Prozessoren erlaubt.

Die erlernte Positionierung mittels nachbarschafts-erhaltender Abbildungen lässt allerdings auch einige Probleme außer Acht:

- Die erlernten Transformationstabellen sind *fest*, ähnlich der eines Assoziativspeichers, und müssen bei jeder kleinsten Änderung der Manipulatorgeometrie, z.B. einer leichten Schiefstellung, neu erlernt werden. Es gibt keine Bewegungsprimitive, die bei einer solchen Umstellung erhalten bleiben und als Wissen benutzt werden können.
- *Zeitsequenzen* können nicht erlernt werden; sie obliegen den höheren Kontrollschichten. Damit gibt es auf dieser Ebene auch keine Möglichkeit, beispielsweise dieselbe Bewegung an einer anderen Position in eine andere Richtung zu wiederholen.
- Es gibt keine *Generalisierung* oder Abstraktion einer Bewegung. Von Menschen wissen wir aber, dass sie Bewegungskonzepte (wie z.B. die Handschrift) auch mit anderen Bewegungsprimitiven durchführen können, wenn beispielsweise anstatt auf Papier klein mit Bleistift auf eine Tafel groß mit Kreide geschrieben wird. Dies ist bei dem beschriebenen Ansatz aber prinzipiell nicht möglich.

Zusammenfassend kann gesagt werden, dass die nachbarschafts-erhaltenden Abbildungen einen interessanten Ansatz zur Roboterkontrolle auf neuronaler Ebene mit vielen interessanten Eigenschaften darstellen. Trotzdem gibt es aber noch einige wichtige Probleme für eine zufriedenstellende, neuronale Roboterkontrolle zu lösen.

Aufgabe 4.7

Errechnen Sie für die quadratische spline-Approximation $F(x) = a_0+a_1x+a_2x^2$ die Koeffizienten a_0,a_1,a_2 aus der Kenntnis von $F(x_1)$, $F(x_2)$, $F(x_3)$.

Aufgabe 4.8

Approximieren Sie eine kubische Funktion. Dazu verwenden Sie ein 1-dim. Kohonen-Netz, lassen bei jedem Neuron zusätzlich eine Stützstelle lernen und interpolieren Sie linear zwischen den Stützstellen. Dazu müssen Sie keine Koeffizienten extra lernen lassen, im Unterschied zur Robotersteuerung.

Erweitern Sie das Verfahren auf 2 Dimensionen. Wo liegt das Problem?

5 Netze mit Radialen Basisfunktionen (RBF)

Angenommen, wir möchten eine Menge von Mustern im Musterraum klassifizieren und benutzen dazu die Eigenschaft, dass meist alle Muster einer Klasse ähnlich zueinander sind, also im gleichen Gebiet im Musterraum liegen: Sie bilden meist einen Haufen oder Cluster.

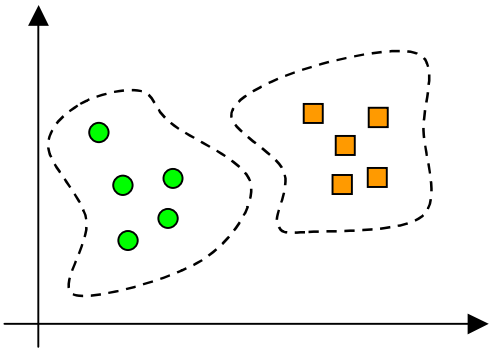


Abb. 5.1 *Clusterbildung im Musterraum*

Anstelle nun eine Trennungslinie zu finden zwischen beiden Klassen in Abb. 5.1 können wir die Aufgabe auch umformulieren und jede Klasse durch die Lage und Ausmaß einer Funktion charakterisieren, deren Funktionswert $S(|\mathbf{x}-\mathbf{x}_i|)$ mit steigendem Abstand $r = |\mathbf{x}-\mathbf{x}_i|$) von der Klassenmitte (Klassenprototyp $\mathbf{x}_i$) abnimmt. Die Klassengrenze ist dann dort gegeben, wo die Funktionswerte für beide Klassen gerade gleich groß sind.
Der gesamte Eingaberaum wird so von einem System von Funktionen („Basisfunktionen“) abgedeckt und in einzelne Klassen eingeteilt.
Solche Basisfunktionen S(r) werden auch „radiale Basisfunktionen“ RBF genannt und können außer zur Klassifikation auch zur Approximation der Dichte der Muster verwendet werden: Bei einem hohen Funktionswert beim Klassenprototypen ist die Auf-

trittswahrscheinlichkeit für die Muster hoch, sonst niedriger. Verwenden wir einfache RBF, so können wir uns kompliziertere Dichtefunktionen aus einfachen durch Überlagerung konstruieren, wobei mehrere RBF pro Klasse vorhanden sind. Allgemein können wir eine beliebige n-dimensionale Funktion f(x) durch eine Überlagerung aus RBF im gesamten Musterraum approximieren, wobei wir dabei die Aufgabe haben, Anzahl, Lage und Ausdehnung der einzelnen RBF so zu bestimmen, dass die Überlagerung möglichst gut die gewünschte Funktion f annähert.

Beim Aufbau eines Approximationsnetzes mit radialen Basisfunktionen (RBF) stellen sich folgende Fragen:

- Welche der vielen möglichen RBF-Funktionen sind besonders gut? Welche sollte man verwenden?
- Wie viele RBF soll man wählen? Wie viele sind für einen noch akzeptablen Ausgabefehler mindestens nötig?
- Welche Lagen sollten die Zentren der einzelnen RBF im Eingaberaum haben?
- Wie groß sollte der maximale Radius *r* der Eingabe (das „rezeptive Feld“) gewählt werden? Ist er abhängig von der Gesamtzahl der RBF?

Die folgenden Abschnitte versuchen, praktisch und theoretisch diese Problematik zu erhellen. Einfache und eindeutige Antworten auf die Fragen sind allerdings zur Zeit noch nicht möglich. Trotzdem haben die RBF inzwischen so viele Anwendungen und Erfolge gefunden, dass sie die Backpropagation-Netzwerke in ihrer Bedeutung als "Arbeitspferd der Neuronalen Netze" ablösen.

5.1 Glockenfunktionen als Basisfunktionen

Die Idee, Basisfunktionen S_i zu verwenden, die nur lokal von einer Untermenge der möglichen Eingaben $\{\mathbf{x}\}$ abhängen, kann man verallgemeinern. Eine derartige Funktion, die im Unendlichen jeweils null und an einer Stelle *a* einen endlichen, positiven reellen Wert annimmt, nennen wir eine *Glockenfunktion (bell shaped function)*.
Nach [CAR92] lässt sich dies formal folgendermaßen definieren:

Definition Eine Funktion S_G mit den Eigenschaften

- $S_G(x) \geq 0$ für reelles x, $S_G(-\infty) = S_G(\infty) = 0$,
- $\int_{-\infty}^{\infty} S_G(x)dx < \infty$ und $\neq 0$
- Es ex. ein reelles *a* mit $S_G(a) \neq 0$ und $S_G(z)$ nicht anwachsend $\forall z \in [a,\infty)$
  nicht abfallend $\forall z \in (-\infty,a)$

nennen wir eine *Glockenfunktion*.

Dabei ist $S_G(a) > 0$ und bildet das Maximum von $S_G(x)$. Mit der obigen Definition lassen sich viele Beispiele für Glockenfunktionen anführen. So entstehen Glockenformen aus

- *Kombinationen von Quetschfunktionen S_Q*

$$S_G(x_1,..,x_n) = \max\left(0,\ 1+\sum_{i=1}^{n} b(x_i)\text{-}1\right) \textit{ mit } b(x_i) = \frac{S_Q(1+x_i)+S_Q(1-x_i)-1}{2S_Q(0)-1} \quad (5.1)$$

Beispiel: Mit $S_Q = S_B$ Heavyside-Funktion ergibt sich ein Rechteckimpuls.

- *Ableitungen von Quetschfunktionen*

$$S_G(x) = \frac{\partial S_Q}{\partial x} \quad (5.2)$$

Beispiel: Mit $S_Q = S_T$ hyperbolischer Tangens ist z.B. $S_G = \partial S_T(z)/\partial z = S'_T(z)$ eine Glockenfunktion

- *Produkte von Glockenfunktionen*

$$S_G(x_1,..,x_n) = S_G(x_1) \cdot \ldots \cdot S_G(x_n) \quad (5.3)$$

Beispiel: Gaußfunktion $S_G(\mathbf{a},\mathbf{x}) = \exp(-(\mathbf{a}-\mathbf{x})^2) = \exp(-(a_1-x_1)^2) \cdot \ldots \cdot \exp(-(a_n-x_n)^2)$

- *allgemeine Radiale Basisfunktionen*

Für alle Funktionen h(r), die monoton fallend von einer positiven, reellen Zahl r (dem Radius) abhängen, lässt sich mit $r = |\mathbf{x}|$ eine Glockenfunktion

$$S_G(\mathbf{x}) = h(|\mathbf{x}|), \qquad \mathbf{x} \in \Re^n \quad (5.4)$$

konstruieren.

- *aus Intervallen zusammengesetzte Funktionen*

Beispiel: Quadratische *spline*-Funktion zur Interpolation

$$S_G(z) = (1-z^2)^{2n} \text{ im Intervall } z \in [-1,+1], \text{ sonst null.} \quad (5.5)$$

Dies entspricht einer an der x-Achse gespiegelten Parabel, die nur im positiven Bereich definiert ist.

Für Glockenkurven nach obiger Definition kann man zeigen [CAR92], dass ein Netzwerk, das aus linear überlagerten, normierten Glockenfunktionen

$$f(\mathbf{x}) = \sum_i w_i \tilde{S}_i (\mathbf{x},\mathbf{c}_i) \tag{5.6}$$

mit $\tilde{S}_i (\mathbf{x},\mathbf{c}_i) = S_i(\mathbf{x},\mathbf{c}_i)/(\sum_k S_k(\mathbf{x},\mathbf{c}_k))$

besteht, jede beliebige Funktion beliebig dicht approximieren kann. Wenn die Gewichte dabei beschränkt bleiben, werden auch die Ableitungen gleichzeitig beliebig dicht approximiert, was in Netzen für Kontrollaufgaben eine wichtige Rolle spielt. Das dazugehörige Netzwerk ist in Abb. 5.2 gezeigt, wobei die RBF-Einheiten mit dem Sybol "Ω" gekennzeichnet sind. Das ganze Netzwerk ist ein Zwei-Schichten-Netz; die erste Schicht enthält als Parameter die Zentren $\mathbf{c}_i$ und die Breite σ der Basisfunktionen, und die zweite Schicht die Gewichte w_i.

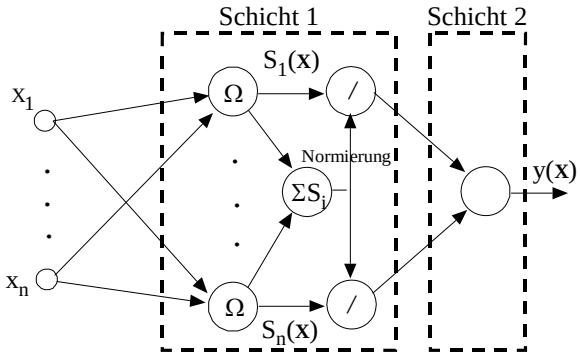


Abb. 5.2 *Das Netzwerk normierter Basisfunktionen*

Nachdem wir eine allgemeine Definition von Glockenfunktionen und ihre wichtige Approximationseigenschaft kennen gelernt haben, stellt sich für uns immer noch die Frage: Welche RBF- oder Glockenfunktion ist die Beste? Welche sollten wir für ein Problem wählen?

5.2 Basisfunktionen maximaler Entropie

Angenommen, wir können nur den reellen Mittelwert c und die Streuung σ eines Datensatzes im ein-dimensionalen Fall beobachten: Was ist dann die geeignetste Basisfunktion, um die Dichtefunktion p(x) der Muster zu approximieren ? Eine Antwort auf dieses Frage wurde von Shannon [SHA49] aus seiner Sicht gegeben: Suchen wir die Funktion, die die meiste Information über die Muster repräsentiert, so lässt sich dies wie folgt errechnen.
Wir suchen diejenige Funktion p(x), die die Information H(x) über alle möglichen Funktionen h(x) maximiert

$$p = \arg\max H(h(x))$$

wobei bei zentrierter Eingabe $\langle x \rangle = c = 0$ die beiden Nebenbedingungen gelten müssen

$$\sigma^2 = \langle x^2 \rangle = \int p(x)\, x^2\, dx \quad \text{oder} \quad g_1(x) := \int p(x) x^2 dx - \sigma^2 \tag{5.7}$$

und

$$\int p(x)\, dx = 1 \quad \text{oder} \quad g_2(x) := \int p(x) dx - 1 \tag{5.8}$$

Dies ist ein Optimierungsproblem mit Nebenbedingungen, das sich mit Hilfe der Methode der Lagrange'schen Multiplikatoren lösen lässt, siehe Anhang D. Hierzu stellen wir eine neue Funktion L auf, die aus der ursprünglichen, zu maximierenden Funktion H(p) und den Nebenbedingungen besteht

$$L(p, \mu_1, \mu_2) := H(p) + \mu_1 g_1(p) + \mu_2 g_2(p) \tag{5.9}$$

wobei das Verschwinden der Ableitungen dieser Funktion nach p, μ_1 und μ_2 die notwendigen Bedingungen für die gesuchten Extremwerte liefert. Für den Parameter *p* ist dies

$$\frac{\partial L(p)}{\partial p} = \frac{\partial}{\partial p} \int [-p \log p + \mu_1 p\, x^2 + \mu_2\, p]\, dx \tag{5.10}$$

$$= \int [-\log p + \mu_1 x^2 + \mu_2 - 1]\, dx = 0$$

Im allgemeinen Fall ist dies nur erfüllt, wenn das Argument der Integraloperation Null ist, so dass mit

$$\mu_1 x^2 + \mu_2 - 1 = \log p \tag{5.11}$$

sich die gesuchte Funktion zu

$$p(x) = \exp(\mu_1 x^2 + \mu_2 - 1) = \exp(\mu_2 - 1) \exp(\mu_1 x^2) = A \cdot \exp(\mu_1 x^2) \tag{5.12}$$

ergibt. Die Ableitungen von L(.) nach μ_1 und μ_2 ergeben wieder die Nebenbedingungen $g_1(p) = 0$ und $g_2(p) = 0$. Nach Einsetzen der obigen Lösung Gl.(5.12) in Gl.(5.7) und (5.8) können die Ausdrücke für die Lagrange'schen Multiplikatoren ermittelt werden. Die Gl.(5.12) wird so mit $\mu_1 = -(2\sigma^2)^{-1}$ und dem Skalierungsfaktor $A = \exp(\mu_2-1) = [(2\pi)^{1/2}\sigma]^{-1}$ zu

$$p(x) = A \exp(-x^2/2\sigma^2) \qquad \textit{Gauß'sche Glockenkurve} \tag{5.13}$$

Die beste, die gestreuten Daten approximierende Basisfunktion im Sinne maximaler mittlerer Information bzw. Entropie ist also die Gauß'sche Glockenkurve, falls nur Mittelwert und Streuung der Daten bekannt ist.
Interessanterweise verändert sich bei festen Intervallgrenzen die optimale Funktion. Bei festen Intervallgrenzen [a,b] benötigen wir die Annahme der endlichen Varianz nicht mehr, so dass in den Gleichungen (5.9)-(5.12) der Term mit μ_1 wegfällt und es sich ergibt

$$p(x) = \exp(\mu_2 -1) = \exp(\mu_2-1) = A = \text{const} \tag{5.14}$$

und mit Einsetzen aus der Nebenbedingung $g_2(x)$ folgt

$$p(x) = \begin{cases} \frac{1}{b-a} & x \in [a,b] \\ 0 & \text{sonst} \end{cases} \qquad \textit{uniforme Verteilung} \tag{5.15}$$

einer uniformen Verteilung, was einer Rechteck-Glockenfunktion als Basisfunktion entspricht.

Bei der Messung mehrerer Varianzen und Mittelwerte lässt sich die gesuchte Wahrscheinlichkeitsverteilung als eine lineare Überlagerung von Basisfunktionen (Kernfunktionen, *kernel functions*) formulieren. Sind die Basisfunktionen K(.) alle gleich (gleiche Varianz bzw. Breite allgemeiner Glockenfunktionen), so werden sie als *Parzen windows* bezeichnet [DUD73]. Der beobachtete Messwert wird als Mittelwert einer Basisfunktion betrachtet, so dass die gesamte geschätzte Verteilung p(x) eine Überlagerung von *M* Basisfunktionen ist:

$$p(x) = \frac{1}{M}\sum_{i=1}^{M} K(x_i - x) \tag{5.16}$$

Für wenige Messwerte x ist die *Parzen window* Methode einfach, schnell und liefert gute Resultate, obwohl die perfekte Approximation nur im Grenzwert mit

$$\lim_{M\to\infty} \sigma(M) = 0, \qquad \lim_{M\to\infty} M\,\sigma(M)^n = \infty \tag{5.17}$$

garantiert ist [DUD73]. Dies bedeutet, dass man mit großem σ startet und mit wachsender Zahl von Trainingsmustern die Anpassung an den unregelmäßigen Funktionsverlauf durch Verkleinern von σ verbessern kann. Alternativ zu dem Anpassungsaufwand für σ

verwendet man eine normierte Gaußfunktion mit $\sigma = 1$ [MOD89] und setzt dafür mehr Basisfunktionen zur Approximation ein.
Für wenige Trainingsdaten kann man mit dem Ansatz des *Parzen window* ohne iterative Verbesserung in einem Durchgang (*one-pass* algorithm) das Approximationsnetzwerk aufstellen und initialisieren. Als Beispiel sind in der Abb. 5.3(a) links die Trainingsdaten und rechts in (b) die Konturlinien des approximierenden "Funktionsgebirges" zu sehen.

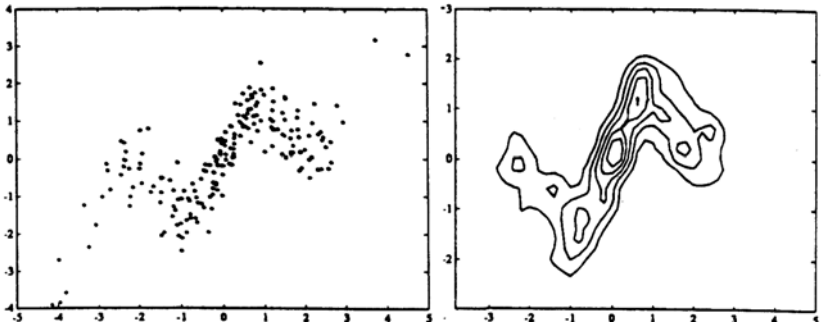


Abb. 5.3 (a) *Daten und (b) die Konturen der Approximation (nach [SCH92])*

Die *Parzen window*-Methode ermöglicht also, aufgrund weniger Messwerte eine kontinuierliche Wahrscheinlichkeitsverteilung zu schätzen. Dies lässt sich auch für die Schätzung der Entropie eines Netzes mit der Ausgabe *y* ausnutzen. Die mittlere Information oder Entropie einer Zufallsvariablen *y* ist

$$H(y) = \langle -\ln p(y) \rangle \tag{5.18}$$

Berücksichtigen wir dies nur zum Zeitpunkt *k*, so können wir die Erwartungswertklammern weglassen und erhalten den stochastischen Schätzwert, dessen Verteilung wir durch die *Parzen window*-Methode als Überlagerung von Kernfunktionen K aus Gl.(5.16) ansetzen

$$H(y) \approx -\ln p(y_k) = -\ln \frac{1}{M} \sum_{i=1}^{M} K(y_i - y_k) \tag{5.19}$$

Damit erhalten wir einen Ausdruck, den wir später zum Lernen verwenden können.

5.3 Approximation mit Glockenfunktionen

Angenommen, wir modellieren für einen n-dimensionalen Argumentwert $\mathbf{x}^i$ den Funktionswert f(**x**) als Mittelwert einer Variablen y mit einer unbekannten Dichtefunktion p(**x**,y). Dann ist dies der bedingte Erwartungswert der Zufallsvariablen y, wenn $\mathbf{x}^i$ gegeben ist

$$f(\mathbf{x}) = \langle y|\mathbf{x}^i\rangle = \int y\, p(y \mid \mathbf{x}^i)\, dy = \frac{\int y\, p(y,\mathbf{x}^i)\, dy}{p(\mathbf{x}^i)} = \frac{\int y\, p(y,\mathbf{x}^i)\, dy}{\int p(y,\mathbf{x}^i)\, dy} \tag{5.20}$$

Fassen wir beide Zufallsvariablen zu einer gemeinsamen Variablen **z** = (**x**,y) zusammen, so ist eine gute Approximation (*Schätzer*) $\hat{p}(\mathbf{z})$ für p(**z**), wie wir oben sahen, die Überlagerung der Gauß'schen Glockenfunktionen

$$\hat{p}(\mathbf{x},y) = \hat{p}(z) = \frac{1}{M}\sum_{i=1}^{M} \frac{1}{\left[(2\pi)^{1/2}\sigma\right]^{(n+1)}} \exp\left(-(\mathbf{z}-\mathbf{z}^i)^T(\mathbf{z}-\mathbf{z}^i)/2\sigma^2\right) \tag{5.21}$$

Mit den beobachteten Werten $y^i := y(\mathbf{x}^i)$ und der Abkürzung

$$d_i^2 := (\mathbf{x}-\mathbf{x}^i)^T(\mathbf{x}-\mathbf{x}^i)$$

ist $$(\mathbf{z}-\mathbf{z}^i)^T(\mathbf{z}-\mathbf{z}^i) = d_i^2 + (y-y^i)^2 \tag{5.22}$$

so dass nach dem Einsetzen von Gl.(5.22) in Gl.(5.21), dem Hinausziehen der Summen aus den Integralen und unter Beachtung von der Normierung $\int p(y)\, dy = 1$ und dem Erwartungswert $\int y\, p(y,y^i)\, dy = y^i$ der geschätzte Approximationswert lautet

$$f(\mathbf{x}^i) = \sum_i y^i \frac{\exp\left(-d_i^2/2\sigma^2\right)}{\sum_k \exp\left(-d_k^2/2\sigma^2\right)} = \sum_i y^i \frac{S_i(\mathbf{x})}{\bar{S}} \tag{5.23}$$

Die Gl. (5.23) lässt sich wieder als zweischichtiges Netzwerk deuten [SPE91], [SCH92]. Die erste Schicht zur Approximation der unbekannten Funktion *f* enthält Neuronen mit Glockenkurven als Ausgabefunktionen $S_G(\mathbf{x},\sigma)$, die, mit bekannten, beobachteten Werten y^i gewichtet, in der zweiten Schicht aufsummiert werden. Die Ausgabefunktionen sind als *bedingte Wahrscheinlichkeitswerte* zusätzlich noch mit $\bar{S}$ normiert, so dass $\Sigma_i S_i(\mathbf{x}) = 1$ gilt. Eine Klassifikation, die auf der maximalen bedingten Wahrscheinlichkeit (*Bayes-Klassifikation*) beruht, ist mit einem solchen Netz leicht zu implementieren [SPE88].

Es sind auch Netzwerke mit dem einfachen Abstandsbetrag (L_1-Abstandsmaß) anstatt des quadratischen Euklidschen Abstands (L_2-Abstandsmaß) denkbar. Diese Art von Neuronen ersparen bei numerischer Berechnung das aufwendige Quadrieren der Komponenten in der Exponentialfunktion [SPE91].

Ist eine ein-eindeutige Abbildung zwischen Eingabe und Ausgabe gegeben, so lässt sich genauso gut auch die Umkehrfunktion lernen – ein wichtiger Vorteil für Kontrollaufgaben [SPE91].
Die Normierung S_i / $\bar{S}$ der Ausgabeaktivität in (5.23) ist dabei ziemlich wichtig. In Abb. 5.4a ist die Ausgabe $y = w_1S_1(\mathbf{x}) + w_2S_2(\mathbf{x}) + w_3S_3(\mathbf{x})$ eines RBF-Netzes aus drei RBF-Neuronen in der ersten Schicht und einem linearen Neuron in der zweiten Schicht gezeigt.

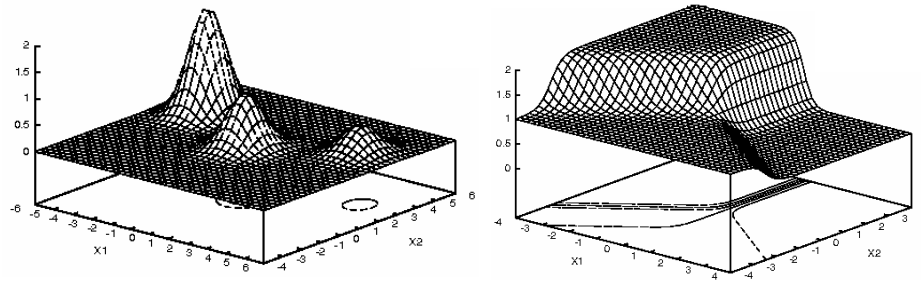


Abb. 5.4 *Einfache (a) und normierte Netzaktivität (b) aus drei RBF Neuronen*

Normieren wir zusätzlich die Ausgabe $S_i = S_G(\mathbf{x})$ zu $\tilde{S}_i = S_i/(S_1+S_2+S_3)$, so ergibt sich das Verhalten in Abb. 5.4b. Hier ist deutlich zu sehen, dass die normierte, in der zweiten Schicht mit w_i gewichtete Ausgabe eines RBF Neurons über den gesamten "Zuständigkeitsbereich" konstant ist. Die Grenze zum benachbarten Zuständigkeitsbereich definiert sich ziemlich scharf durch den Aktivitätsquotienten (die relative Aktivität), die an der Grenze schnell auf Null abfällt. Damit ist es im Unterschied zu (a) relativ einfach, eine Hyperfläche zu approximieren.

Die Ansätze des vorigen Abschnitts sind nur bei einer kleinen Anzahl von Trainingsmustern sinnvoll. Bei einer größeren Zahl von Mustern kann man von den Einzelmustern absehen und nur noch wenige, aber typische Basisfunktionen verwenden, die nur durch wenige Parameter festgelegt sind und damit eine Datenreduktion (*Generalisierung*) bedeuten. Im Unterschied zu der Methode der *Parzen windows* des vorigen Abschnitts besitzen hier die Basisfunktionen nicht nur unterschiedliche Mittelwerte $\mathbf{c}^i$, sondern auch unterschiedliche Varianzen σ_i, die iterativ gelernt werden müssen. Hauptanliegen dieses Abschnittes ist es, dafür geeignete Verfahren zu zeigen.

Normierung der Eingabevariablen

Bei vielen Problemen aus der realen Welt ist von vornherein nicht klar, welche die adäquate Skalierung der Variablen ist. Sollte man eine Länge in *Metern* oder in *cm* messen, einen Druck in *Pascal* oder *Hektopascal*? Und wenn wir uns auf eine Einheit festgelegt haben, welches ist dann die passende andere der zweiten Variablen? Und der

dritten, vierten ,usw.? Beachten wir dies nicht, so werden die Punkthaufen im Musterraum zu Elypsen verzerrt; für ein schnelles Lernen der Parameter müssen die Lernschritte dann in den unterschiedlichen Dimensionen der Variablen unterschiedlich groß sein. Diese Komplikation können wir vermeiden, indem wir alle Variablen so normieren, dass ihr Mittelwert den Wert null und ihre Varianz den Wert eins annimmt. Die Zusammenhänge der Variablen werden dadurch nicht berührt, nur ihre Darstellung.
Jede Variable wird also vor der eigentlichen Verarbeitung mit der Transformation $d = (x–c)/\sigma$ um ihren Mittelwert c und ihre mittlere Abweichung σ korrigiert. Im multidimensionalen, allgemeinen Fall ist die Länge d^2 der um den Punkt *c* zentrierten und auf die Varianz eins skalierten Zufallsvariablen **x**

$$d^2 = (\mathbf{x}–\mathbf{c})^T\mathbf{C}^{-1}(\mathbf{x}–\mathbf{c}) \qquad \textit{Mahalanobis Abstand} \qquad (5.24)$$

mit $\mathbf{C}^{-1}$ als der Inversen der Kovarianzmatrix $\mathbf{C} = \langle(\mathbf{x}–\mathbf{c})(\mathbf{x}–\mathbf{c})^T\rangle$ gegeben. Geometrisch lässt sich dies so interpretieren, dass die Variable **x** einer Verschiebung um **c** und einer Drehung und Skalierung durch eine Matrix (hier $\mathbf{C}^{-1}$) unterworfen wird. Stimmen die Koordinatenachsen mit den Hauptachsen der Punktwolke {**x**} überein, so ist $\mathbf{C}^{-1} = \Lambda^{-1}$, eine Diagonalmatrix. Die Hauptdiagonale besteht aus den Eigenwerten $\lambda_i^{-1} = \sigma_{ii}^{-2}$: Es findet dann keine Drehung mehr statt, sondern nur noch eine Skalierung mit $1/\sigma_{ii}^2$.

Der mit Gl. (5.24) korrigierte Abstand (**x**–**c**) zwischen dem Punkt **x** und dem Zentrum **c** wird als *Mahalanobis* Entfernung $d_M(\mathbf{x},\mathbf{c})$ bezeichnet und stellt einen statistisch korrigierten Euklidschen Abstand dar. Die Gaußsche Glockenfunktion wird in diesem Fall

$$S_G(\mathbf{x}) = A \exp(-(\mathbf{x}–\mathbf{c})^T\mathbf{C}^{-1}(\mathbf{x}^-\mathbf{c})) \qquad (5.25)$$

wobei die Normierungsbedingung (5.8) für eine *n*-dimensionale Normalverteilung den Wert $A = [(2\pi)^n \det\mathbf{C}]^{-1/2}$ bestimmt.
Die Transformation eines Musterhaufens auf eine zentrierte Verteilung mit gleicher Varianz in allen Richtungen ist in Abb. 5.5 illustriert. Hierbei sind alle Punkte **x** einer zwei-dimensionalen Gaußverteilung, die den gleichen Funktionswert $S_G(\mathbf{x})$ ergeben ("Höhenlinien"), durch eine Linie verbunden. Wie man sich leicht klarmachen kann, ist dies im Allgemeinen eine Elypse, die nach der Transformation in einen Kreis mit dem Einheitsradius übergeht. Die Abbildung zeigt aus der Vogelperspektive, wie die Transformation Gl.(5.24) sich auch formal durch eine allgemeine Verschiebung, Drehung und Skalierung erreichen lässt.

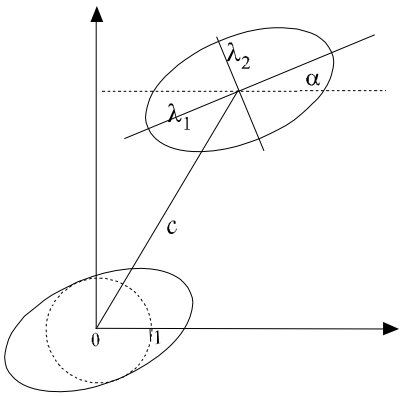


Abb. 5.5 *Transformation auf Normalverteilung*

Die Skalierung **S** und die Drehung **D** lassen sich als eine Matrizenmultiplikation formulieren, ebenso die Verschiebung **V**, wenn wir die Eingabe **x** um eine konstante Komponente $\mathbf{x} \rightarrow (\mathbf{x}^T,1)^T$ erweitern. Im zwei-dimensionalen Fall ist dies also

$$\mathbf{x} \rightarrow \mathbf{z} = \mathbf{SDVx} \text{ mit } \mathbf{S}=\begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix}, \mathbf{D} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}, \mathbf{V}=\begin{pmatrix} 1 & 0 & -c_1 \\ 0 & 1 & -c_2 \end{pmatrix} \quad (5.26)$$

oder

$$\mathbf{x} \rightarrow \mathbf{z} = \mathbf{Mx} \quad \text{mit } \mathbf{M}=\begin{pmatrix} s_1\cos\alpha & -s_1\sin\alpha & -c_1 s_1(\cos\alpha - \sin\alpha) \\ s_2\sin\alpha & s_2\cos\alpha & -c_2 s_2(\cos\alpha + \sin\alpha) \end{pmatrix}$$

und die Transformation Gl.(5.24) wird zu

$$d^2 = \mathbf{z}^2 = \mathbf{z}^T\mathbf{z} = \mathbf{x}^T\mathbf{M}^T\mathbf{Mx} \quad (5.27)$$

Hier kann der Mahalanobis-Abstand mit **Mx** nicht nur als die Auswirkung der Transformation **M** gesehen werden, sondern es ist die geometrische Bedeutung der einzelnen Koeffizienten M_{ij} der Matrix **M** direkt klar. Beispielsweise ist mit $\cos^2\alpha + \sin^2\alpha = 1$ der Skalierungskoeffizient $s_1>0$ durch $M_{11}^2 + M_{12}^2 = s_1^2$ gegeben. Dadurch sind über $\cos\alpha$, $\sin\alpha$, der Drehwinkel α, und damit auch die Verschiebung c_1, bestimmt.
Mit der Aufbereitung der Eingabe auf Mittelwert null und Varianz eins haben wir nun die Voraussetzungen geschaffen, um mit einem Lernverfahren alle Eingabedimensionen einheitlich zu behandeln und daraus die gewünschten Parameter zu bestimmen.

5.4 Lernverfahren

Ein RBF-Netzwerk besteht meist aus zwei oder mehr Schichten. Jede Schicht hat ihre Parameter, die ihre Funktion festlegen. Die Lernverfahren, mit denen diese Parameter bestimmt werden können, lassen sich in zwei Ansätze unterteilen:

1) dem Anpassen zuerst der ersten Schicht, also Lage $\mathbf{c}_i$ und Varianz $\mathbf{C}$ der RBF-Neuronen, und dann Anpassen der Gewichte $\mathbf{w}_j$ nächsten Schicht bei konstanter Lage

2) dem gleichzeitigen Anpassen aller Parameter aller Schichten bei gegebenen Anzahlen der RBF-Neuronen in der ersten Schicht und der Neuronen in den weiteren Schichten.

5.4.1 Lernen der Parameter der Schichten

Die getrennte Optimierung der Parameter der ersten und zweiten Schicht hat verschiedene Vorteile. So lassen sich zum einen verschiedene, voneinander unabhängige Methoden zur Optimierung der ersten und der zweiten Schicht einsetzen, zum anderen ist die Konvergenz dieser Verfahren für die Anpassung einer einzelnen Schicht deutlich besser als bei der gleichzeitigen Anpassung beider Schichten. Dies hängt damit zusammen, dass der Suchraum bei beiden Schichten (die Anzahl aller möglichen Parameterwertekombinationen) exponentielle größer ist als der Beschränkung auf die Parameter nur einer Schicht; die Dimensionszahl beider Suchräume addieren sich.

5.4.2 Anpassung der ersten Schicht

Ein gängiger Ansatz für die erste Schicht besteht darin, die Wahrscheinlichkeitsdichten der Eingabemuster durch die Lage der RBF-Neuronen zu approximieren. Ein Lernalgorithmus für die Approximation mit Glockenfunktionen besteht grundsätzlich aus zwei Schritten:

- einer initialen Verteilung (Anzahl, Lage und Form) der Glockenfunktionen
- der iterativen Adaption der Parameter an die Trainingsdaten

Während die Adaption der Parameter relativ leicht durchzuführen ist, ist die initiale Phase für das Lernen entscheidend und der häufigste Grund, wenn das System nach dem Training nicht die gewünschte Leistung zeigt, sondern in lokalen Minima "hängen" bleibt oder nur langsame bzw. keine Konvergenz aufweist.

Wichtigstes Ziel der Initialisierung ist eine Aufteilung der "Datenwolke" in unterschiedliche Haufen (*cluster*), die jeweils von einer RBF "besetzt" werden. Als Initialisierung ist also eine parallele (OFF-LINE) oder sequentielle (ON-LINE) Clusteranalyse nötig. Für eine effiziente Initialisierung müssen wir nun unterscheiden, ob die Trainingsdaten bekannt oder unbekannt sind.

- **Bekannte Trainingsdaten**

  In diesem Fall können wir eine grobe, statistische Vorverarbeitung der Gesamtmenge der Trainingsdaten durchführen. Dazu wird zuerst mit einem einfachen Clustersuchalgorithmus (z.B. k-mean, s. [DUD73]) eine Unterteilung der Daten in Haufen (*cluster*) vornehmen [WEM91], [MOD89]. Bei Klassifizierungsproblemen wird dies durch die Annnahme der Existenz von Muster-Haufen gleicher Klassenzugehörigkeit gerechtfertigt. Dann wird in wenigen Rechner-Minuten die Streuung (Eigenwerte) der Cluster bestimmt und damit die Parameter **M** bzw. **c** und **C** der dazu gehörenden Glockenfunktionen initialisiert [MACFH92], [MKAC93].

- **Unbekannte Trainingsdaten**

  Der einfachste Ansatz einer sequentiellen Clusteranalyse besteht darin, die initiale Verteilung der Glockenfunktionen so zu wählen, dass der Raum der Eingabemuster $\{\mathbf{x}\}$ entweder systematisch in periodischen Abständen [BROL88] oder zufällig durch die Glockenfunktionen überdeckt wird. Fängt man mit wenigen Glocken funktionen großer Reichweite (großer Überdeckung, d.h. großem σ) an und verkleinert dann die Radien und erhöht die Anzahl der Funktionen, so kann sich das System mit diesem sukzessiven Netzaufbau die Wahrscheinlichkeitsdichten approximativ "einfangen". In Abb. 5.6 ist die iterative Verkleinerung der Radien und Erhöhung der Neuronenzahl für eine handgeschriebene Zahl "3" gezeigt.

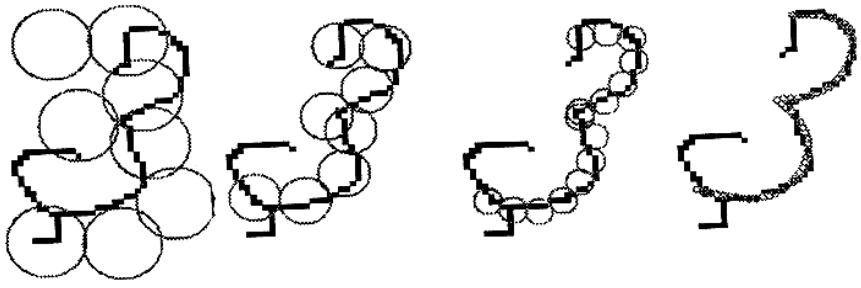


Abb. 5.6 *Annäherung von Anzahl und Überdeckung von Glockenfunktionen an die Intensitätsverteilung eines Bildes (nach [HIN92]).*

In beiden Situationen, bei bekannten und unbekannten Daten, lässt sich die Komplexität des Netzwerks und die Trainingszeiten deutlich verringern, wenn man das Netzwerk durch die sukzessive Erzeugung neuer Glockenfunktionen (Hinzufügen von Neuronen) verbessert.

Betrachten wir nun nach der Phase der Initialisierung die Phase der Verbesserung der Parameter des festen Netzes. Die Formeln für eine iterative Verbesserung lassen sich danach einteilen, ob sie die Schichten *unabhängig voneinander* oder *als Ganzes* berücksichtigen.

Überdeckung durch regelmäßiges Raster

Ein interessanter Ansatz bestimmt die Lage und Ausdehnung der Cluster über eine Überlagerung fest definierter Glockenfunktionen. In Abb. 5.7 ist dieser Gedanke verdeutlicht.

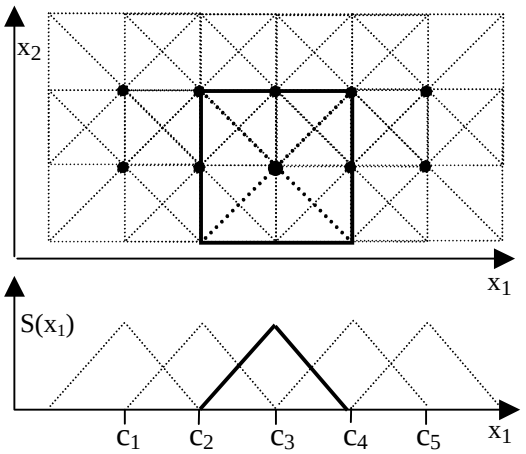


Abb. 5.7 *Überdeckung des Eingaberaums durch genormte dreieckige Glockenfunktionen*

Jede Dreiecks-Glockenfunktion verdeutlicht die Lage und die Ausdehnung des rezeptiven Feldes eines potentiellen RBF-Neurons. Am Anfang gibt es kein Neuron im System. Tritt nun ein Eingabemuster **x** auf, das keinem existierendem Neuron zuzuordnen ist, so wird die Glockenfunktion, die maximalen Wert S(**x**) hat, gewählt und an der Stelle ein RBF-Neuron platziert. Auf diese Weise werden nur die tatsächlich benötigten Neuronen auf einer Liste gehalten; alle nicht aktivierten Glockenfunktionen werden nicht berücksichtigt. Die Gesamtapproximation ergibt sich dann als gewichtete Überlagerung aller Glockenfunktionen, wobei die Gewichte die Häufigkeit der Aktivierung der einzelnen Funktion angibt, s. [ALEX03].

Überdeckung durch Fehlerminimierung

Kriterium für die Lage des neuen RBF-Neurons bei diesem Verfahren ist der Punkt, an dem der größte Fehler gemacht wird. Vermutet man an Orten mit größerem mittleren Fehler auch mehr Information (z.B. bei der Adaption von RBF-Bereiche an Bildkonturen, s. Abb. 5.6 und [SCTW91]), so ist die Erzeugung und damit die Konzentration neuer Neuronen gerade an diesen Häufigkeitspunkten sinnvoll.

Bei bekannten Daten kann man nach jedem Trainingslauf den Ort des größten Fehlers des Netzes feststellen und dort ein Neuron platzieren [SCH92]. Sein Gewicht in der zweiten Schicht wird gerade so gewählt, dass es den Fehler an dieser Stelle kompensiert. Das Training bricht ab, wenn der maximale Fehler klein genug ist.

Gibt man einen festen Fehlerwert als Schranke vor, bei dessen Überschreiten jeweils ein Neuron eingefügt wird, so kann man bei jedem Trainingslauf auch mehrere Neuronen zum Netz hinzufügen [SCTW91], [PLA91]. Dazu kann man auch die Strategie einer Fehlerschranke mit der einer Abstandsschranke (Überschreiten des Abstands $|\mathbf{x}-\mathbf{c}_m|$ der Eingabe $\mathbf{x}$ vom Zentrum $\mathbf{c}_m$ des nächsten Neurons) kombinieren, um eine einigermaßen konsistente Überdeckung des Eingaberaumes zu erreichen. Diese Art von Strategien empfehlen sich auch bei sequentieller Approximation unbekannter Daten, wobei mit fortlaufender Zeit auch die Schranken erniedrigt und somit die Ansprüche erhöht werden können.

Ein überwachter Clusteralgorithmus zum Erzeugen und Anpassen von Glockenkurven ist beispielsweise auch von Lee und Kil [LEEK91] vorgestellt worden. Hier werden solange neue Glockenfunktionen erzeugt, bis alle Eingabemuster innerhalb eines sog. "effektiven Radius" fallen und somit durch eine Glockenfunktion "abgedeckt" werden. Tritt keine Verkleinerung des Fehlers mehr ein ("Sättigung") und ist der Fehler noch zu hoch, so werden die Weiten σ der Glockenfunktionen verkleinert und Training und Erzeugung neuer Funktionen erneut durchgeführt, bis der vorgegebene Maximalfehler unterschritten wird.

Clusteranalyse durch Kohonen-Netze

Da das gleichzeitige Lernen aller Parameter, also der Zahl und Lage (Mittelpunkt und Ausdehnung) der Basisfunktionen sowie der Gewichte der 2. Schicht, im allgemeinen Schwierigkeiten macht, beschränkt man sich meistens auf wenige Parameter und initialisiert die anderen mit plausiblen Werten. Beispielsweise wird gern bei fester Anzahl von RBF's ein gleichmäßige Ausdehnung in alle Richtungen angenommen, also $S_G(\mathbf{x},\mathbf{c}) = S_G(|\mathbf{x}-\mathbf{c}|)$ mit $\mathbf{C}^{-1} = \sigma^{-2}\mathbf{I}$, was durch eine Transformation der Eingabevariablen auf gleiche Varianz unterstützt werden kann.

Dazu kann man für die erste Schicht eine iterative Clusteranalyse durchführen, beispielsweise mit dem Kohonen-Algorithmus.

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + h(i,k,\sigma_y)[\mathbf{x} - \mathbf{w}_k(t)] \tag{5.28}$$

Dabei ist die Nachbarschaftsfunktion für das gewinnende Neuron *k* definiert als

$$h(i,k,\sigma_y) = \begin{cases} 1 & \text{Neuron } i \text{ ist aus Nachbarschaft von Neuron } k \\ 0 & \text{sonst} \end{cases} \tag{5.29}$$

Eine interessante Variante wurde dazu von Xu, Krzyzak und Oja [XUKO93] vorgeschlagen. Dabei wird der Grundgedanke des *Competitive Learning* und des Kohonen-Netzes, das Neuron mit der kürzesten Entfernung $|\mathbf{x} - \mathbf{c}|$ als Gewinner auszuwählen und dem Trainingspunkt $\mathbf{x}$ zu nähern, durch die Vorschrift ergänzt, den "Rivalen" des Gewinners mit dem zweitnächsten Abstand mit einer Abstandsvergrößerung "abzuschrecken":

$$h(i,k,\sigma_y) = \begin{cases} +1 & i = k \\ -1 & \text{Neuron } i \text{ ist nächster Nachbar von Neuron } k \\ 0 & \text{sonst} \end{cases} \tag{5.30}$$

Die Nachbarschaftsfunktion h(c,k) ist also +1 beim Gewinner *k*, –1 beim nächsten Verlierer und 0 bei allen anderen. Dies führt dazu, dass bei jedem Cluster nur ein RBF-Neuron existiert; nicht benötigte neuronale Gewichte $\mathbf{c}_i$ wandern an den Rand und werden nicht mehr iteriert. Sondern wir nach einer gewissen Iterationszahl diese "toten" Neuronen aus, so erhalten wir eine gute Clusteraufteilung [XUKO93].

Eine weitere, wichtige Variante ist der Ansatz, bei gegebener Größe σ und Anzahl *m* der RBF ihre Lagen $\mathbf{c}_i$ der zu approximierenden Wahrscheinlichkeitsverteilung anzupassen. Dazu lässt sich beispielsweise der Algorithmus der *Kohonen map* so abwandeln, dass wir anstelle der *winner-take-all* Entscheidung die *soft winner-take-all* Regel verwenden. In diesem Fall ist die Nachbarschaftsfunktion für alle gleich

$$h(i,k,\sigma_y) = \frac{e^{z_i}}{\sum_j e^{z_j}} \quad \text{mit } z_i = \mathbf{x}^T\mathbf{c}_i \text{ und } z_k = \max_i z_i \tag{5.31}$$

Dabei wird der Nachbarschaftseinfluss h(c,k,t) genutzt, um die Gewichte der Nachbarn denen des Gewinners anzupassen, was zu einer schnelleren Konvergenz und zu einer homogeneren Abdeckung der Gesamtverteilung führt.

Versucht man den *maximalen* Fehler zu minimieren, so ist dies nicht unbedingt identisch mit dem *mittleren* Fehler, der eng mit der Häufigkeit der Punkte in einem Gebiet zusammenhängt. Aus diesem Grund ist für den kleinsten maximalen Fehler eine Initialisierung der RBF Neuronen Gewichte $\mathbf{c}_i$ mit einer Kohonen-Karte nicht empfehlenswert, da hier die Auftrittswahrscheinlichkeitsdichte $p(\mathbf{x})$ eine Rolle spielt und nicht der Fehler allein.

5.4.3 Anpassung der zweiten Schicht

Die Approximation mit Radialen Basisfunktionen verbindet verschiedene Mechanismen miteinander. In der ersten Schicht werden Muster aus dem Eingaberaum auf Reaktionen von RBF-Neuronen abgebildet. Durch ihre Eigenschaften wird trotz geringen Variationen die Eingabe auf gleiche Ausgabe abgebildet; dies bedeutet eine Tolerierung von Abweichungen. Beispielsweise wird eine Linie im zwei-dimensionalen Eingaberaum bei radialsymmetrischer RBF immer auf die gleiche Ausgabeerregung abgebildet, egal in welchem Winkel sie im rezeptiven Feld erscheint, was große Vorteile, z.B. bei der Erkennung von handgeschriebenen Buchstaben, bedeutet. Die erste Schicht lässt sich dabei als deformationsinvariante Mustervorverarbeitung betrachten, die von der zweiten Schicht, einem adaptiven Filter, weiterverwendet wird.

Für die Verbesserung der Gewichte w_i dieser zweiten, linearen Schicht mit der Ausgabe

$$y = z = (x_1,...,x_n)(w_1,...,w_n)^T = \mathbf{x}^T\mathbf{w} \tag{5.32}$$

lässt sich nun, wie für einen adaptiven, linearen Filter üblich, entweder die Widrow-Hoff Lernregel verwenden, oder besser noch das in Kapitel 3.3.1 beschriebene Verfahren des "Eigenvektor fitting". Hierbei wird der kleinste, erwartete quadratische Fehler TLMSE von der Approximation zur gemessenen Kurve dadurch erreicht, dass man durch Anwendung der negativen Oja-Lernregel

$$\mathbf{w}(t) = \mathbf{w}(t\text{-}1) - \gamma(t)\, y\, [\mathbf{x}(t) - \mathbf{w}(t\text{-}1)y] \quad \textit{negative Oja Lernregel} \tag{5.33}$$

oder durch Verwendung der Hebb'schen Lernregel mit Gewichtsnormierung für die Abweichungen den Eigenvektor mit dem kleinsten Eigenwert stochastisch iterativ lernt. Voraussetzung ist dabei, dass die Eingabe **x** für die zweite Schicht zentriert wird. Dies lässt sich durch den Abzug des Mittelwerts $\mathbf{x}_0$ von der Aktivität *z* erreichen

$$z = (\mathbf{x}\text{-}\mathbf{x}_0)^T\mathbf{w} = \mathbf{x}^T\mathbf{w} - \mathbf{x}_0{}^T\mathbf{w} = \mathbf{x}^T\mathbf{w} - w_0 \qquad w_0 = \mathbf{x}_0{}^T\mathbf{w} \tag{5.34}$$

Ergänzen wir wieder den Gewichtsvektor **w** durch eine Komponente w_0 und die Eingabe **x** durch eine Komponente $x_0 = 1$, so wird die Mittelwertskorrektur aus Gl.(5.38) automatisch in der linearen Transformation Gl.(5.32) vorgenommen. Den Mittelwert $\mathbf{x}_0$ können wir entweder vorher über alle Eingaben ermitteln, oder aber ebenfalls iterativ über die stochastische Approximation aus Abschnitt 2.3 lernen

$$w_0(t) = w_0\,(t\text{-}1) \; - 1/t \left(w_0\,(t\text{-}1) - \sum_{i=1}^{n} w_i x_i(t) \right) \tag{5.35}$$

Ein anderes Modell verwendet das Verfahren des "Competitive Learning" aus Abschnitt 4.2. Abhängig vom Erfolg wird der Gewichtsvektor desjenigen Neurons der zweiten Schicht verbessert, der bei Eingabe eines Musters die größte Aktivität bewirkt. Damit werden die Neuronen der zweiten Schicht auf Ereignisse "spezialisiert"; sind es

binäre Neuronen, so lernen sie die Klassenprototypen von Ereignisklassen. In der Aktivitätsphase stellt das gesamte Netzwerk dann einen Assoziativspeicher mit binärer Ausgabe dar, dessen Eingabemuster in der ersten Schicht deformationsinvariant vorbereitet in der zweiten Schicht quantisiert werden. Deshalb ist dieses Netz auch als Klassifikator verwendbar.

Verwenden wir in der zweiten Schicht lineare Neuronen, so ist es besser, anstelle des einfachen "Competitive Learning" mit der Ausrichtung der Neuronen der zweiten Schicht auf spezielle Ereignisse direkt eine Datenorthogonalisierung bzw. Dekorrelation ihrer Ausgabe mit Hilfe einer Hauptkomponentenanalyse (PCA) vorzunehmen, bei der die Neuronen unüberwacht "Elementarereignisse" lernen. Eine darauf folgende dritte Schicht, beispielsweise ein Klassifikator, hat durch die dekorrelierten Variablen eine wesentlich besseres Konvergenz bei überwachter Gewichtsanpassung mit der Widrow-Hoff Lernregel.

Eine interessante Alternative besteht darin, anstelle des üblichen kleinsten quadratischen Fehlers die Gewichte der zweiten Schicht so zu wählen, dass die mittlere Information der Ausgabe maximal oder minimal wird, je nach Fragestellung. Für das Gradienten-Lernen

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma \text{grad}\, H(y(\mathbf{w})) \qquad \textit{Minimisierung der Entropie} \qquad (5.36)$$

wird die Ableitung der Entropie benötigt, die auf eine Schätzung der Wahrscheinlichkeitsverteilung der $y(\mathbf{w})$ aufbaut.

$$\text{grad}_w\, H(y(\mathbf{w})) = -\frac{\partial}{\partial \mathbf{w}} \langle \ln p(y) \rangle = -\left\langle p(y)^{-1} \frac{\partial p(y)}{\partial y} \frac{\partial y}{\partial \mathbf{w}} \right\rangle$$

Zur Schätzung von p(y) verwenden wir die *Parzen window*-Methode aus Gl.(5.16) für eine diskrete Ausgabe $y = y_i$ in Abhängigkeit vom Gewicht w_j mit den M früheren Ausgaben y_k und approximieren damit die Information, siehe Gl. (5.19)

$$-\ln p(y_i) = -\ln \frac{1}{M} \sum_{k=1}^{M} K(y_i - y_k)$$

Wir erhalten so einen Ausdruck für eine geschätzte, stochastische Version des Gradienten

$$-\frac{\partial}{\partial w_j} \ln p(y_i) = -\left[\sum_{k=1}^{M} K(y_i - y_k) \right]^{-1} \sum_{k=1}^{M} K'(y_i - y_k) \frac{\partial (y_i - y_k)}{\partial w_j}$$

Da die zweite Schicht mit $y = \mathbf{w}^T\mathbf{x}$ linear ist, gilt für n Gewichte

$$\frac{\partial (y_i - y_k)}{\partial w_j} = \frac{\partial}{\partial w_j} \sum_{s=1}^{n} w_s (x_{is} - x_{ks}) = x_{ij} - x_{kj}$$

Gehen wir bei der Schätzung mit unserer stochastischen Approximation durch M = 1 auf den letzten vorher beobachteten Wert zurück, so erhalten wir mit der Abkürzung

$$f(y_i\text{-}y_k) = \frac{K'(y_i - y_k)}{K(y_i - y_k)} \tag{5.37}$$

die Lerngleichung

$$w_i(t) = w_i(t-1) + \gamma\, f(y_i\text{-}y_k)\,(x_{ij} - x_{kj}) \qquad \textit{allgem. Hebb'sche Regel} \tag{5.38}$$

Diese Lerngleichung lässt sich als verallgemeinerte Hebb'sche Lernregel ansehen [ERD02], die aber im Unterschied zur üblicherweise verwendeten reinen Korrelationsregel nicht den Approximationsfehler minimiert, sondern die Entropie. Auch hier ist für eine schnellere Konvergenz die Normierung der Gewichte $|\mathbf{w}| = \text{const}$ nach jedem Lernschritt eine gute Bedingung.
Verwenden wir die Gauß'sche Glockenfunktion als Kernfunktion zur Approximation, so wird mit

$$K'(y_i\text{-}y_k) = \exp'\left(-\frac{(y_i - y_k)^2}{\sigma^2}\right) = -\frac{2(y_i - y_k)}{\sigma^2}\exp\left(-\frac{(y_i - y_k)^2}{\sigma^2}\right)$$
$$= -\frac{2(y_i - y_k)}{\sigma^2} K(y_i\text{-}y_k)$$

die Funktion in Gl.(5.52) zu

$$f(y_i\text{-}y_k) = \frac{K'(y_i - y_k)}{K(y_i - y_k)} = -\frac{2(y_i - y_k)}{\sigma^2} = -c\,(y_i\text{-}y_k) \qquad \gamma' = \gamma c$$

Damit vereinfacht sich die Lernregel für ein Gewicht (mit der Umbenennung $\gamma' \rightarrow \gamma$) zu

$$w_j(t) = w_j(t-1) - \gamma(y_i\text{-}y_k)\,(x_{ij} - x_{kj}) \qquad \textit{Hebb'sche Regel} \tag{5.39}$$

Man beachte, dass dies natürlich auch für Linearkombinationen davon gilt [ERD03]. Wollen wir beispielsweise nicht die Entropie der Ausgabe minimieren, sondern des Fehlers $\varepsilon_i = y_i - L_i$ und damit die Abweichung vom vorgegebenen Sollwert L_i minimieren, so wird Gl. (5.39) zu

$$w_j(t) = w_j(t-1) - \gamma(\varepsilon_i\text{-}\varepsilon_k)\,(x_{ij} - x_{kj}) \qquad |\mathbf{w}| = \text{const} \tag{5.40}$$

Bei der Abschätzung der Entropie durch die veränderte Hebb'sche Regel baut das Lernen im Unterschied zur ursprünglichen Regel nicht allein auf den aktuellen Werten auf, sondern bezieht mindestens einen Wert aus der Vergangenheit zur Abschätzung der aktuellen Wahrscheinlichkeitsdichte mit ein.

5.4.4 Codebeispiel

Bei den Lernverfahren gilt also, dass es nicht sinnvoll ist, mit Fehler-Backpropagation beide Schichten gleichzeitig zu trainieren. Die Konvergenz der zweiten Schicht wird wesentlich beschleunigt, wenn in der ersten Schicht feste Parameter eingesetzt werden. Aus diesem Grund ist in dem einfachen Codebeispiel 4.6.1 einheitliche, kugelsymmetrische RBF gewählt, die man bei der Initialisierung aber auch abweichend vom Codebeispiel - initial gleichmäßig über den Eingaberaum verteilen und/oder mit der Kohonen map (s. Codebeispiel 2.4.1) trainieren kann.

RBF: (* Approximation einer Funktion L(**x**) mit einem RBF-Netz f(**x**) *)

```
TYPE VEKTOR = ARRAY [1..n] OF REAL;
 CONST sigma2 = 1.0 (* σ²*);
       s = 100    (* Anzahl der Tabellenwerte von e–x² *);
       r = 0.001  (* RBF-Reichweitenschwelle *)

VAR       (* Datenstrukturen *)
 e         : ARRAY[1..s] OF REAL;
 x         : VEKTOR;                  (* Eingabe *)
 x0        : ARRAY[1..m] OF VEKTOR;   (* Lage der RBF Zentren *)
 y,w       : ARRAY[1..m] OF REAL;     (* Eingabe und Gewichte der 2. Schicht*)
 f,L,Sum   : REAL;                    (* IST und SOLL-Ausgaben, S' *)
 γ         : REAL;                    (* Lernrate *);
BEGIN
   a:=0.0; da:=-ln(r)/s;              (* RBF Funktionstabelle e[.] erstellen *)
   FOR i:=1 TO s DO  e[i]:=exp(–a);  a:=a+da;  END;
   γ:= 0.1;                           (* Lernrate festlegen *)

   FOR i:=1 TO m DO                   (* Gewichte initialisieren  *)
     Read( PatternFile,x0[i],w[i])    (* mit zufälligen Trainingsmustern *)
   END
   REPEAT
     Read( PatternFile,x,L)           (* Eingabe *)

   (* Aktivität bilden im Netz *)
      Sum:= 0.0;
     FOR i:=1 TO m DO                 (* Für alle Neuronen der 1. Schicht *)
        y[i] : =   Srbf(x-x0[i])      (* Nicht-lin. Ausgabe*)
        Sum : =    Sum+y[i];         (* Gesamtaktivität bilden *)
     END;
     f:= Z(w,y);                      (* Aktivität 2.Schicht: f(x)=w^T S_RBF(x–x0)*)
     f:= f/Sum;                       (* und normieren *)

    (* Lernen der Gewichte der 2.Schicht *)
     y2:= Z(y,y)                                  (* |y|² *)
     FOR i:=1 TO m DO                             (* Für alle Dimensionen *)
          w[i]:= w[i] – γ*(f–L)*y[i]/y2           (* Gewichte verändern *)
     END;
    UNTIL EndOf(PatternFile)
END RBF.
```

```
PROCEDURE Srbf(x:VEKTOR):REAL;      (* Glockenfunktion, definiert als Tabelle *)
BEGIN
   index := TRUNC((Z(x,x)/sigma2)/da);
   IF index < s THEN RETURN e[index] ELSE RETURN 0.0 END
END Srbf;
```

Codebeispiel 5.1 *Aktivierung und MSE-Lernen in einem RBF-Netz.*

5.5 Verbundenes Lernen beider Schichten

Den Nachteil einer längeren Anpassungsdauer bei der gleichzeitigen Optimierung beider Schichten kann man durch geeignete Verfahren begegnen. Betrachten wir zuerst den direkten Ansatz, durch einen einfachen Gradientenabstieg beide Schichten zu optimieren.

5.5.1 Lernen mit Backpropagation

Im Fall einer ganzheitlichen Lernregel wird meist der normale Backpropagation-Algorithmus verwendet, wobei anstelle sigmoidaler Ausgabefunktionen die Glockenfunktionen eingesetzt werden. Meist werden dabei die nicht-normierten RBF Netze verwendet, obwohl die Konvergenz hauptsächlich für die normierten gezeigt wurde [CAR92], [YUKY94], da sich hier die Lerngleichung leicht für die Koeffizienten M_{ij} errechnen lassen.

Betrachten wir als Kriterium den kleinsten erwarteten quadratischen Fehler $R(M) = \langle(f(\mathbf{x},M)–F(\mathbf{x}))^2\rangle = \langle r(\mathbf{x},M)\rangle$ zwischen der Ausgabe f(.) des neuronalen Netzes und dem gewünschten Funktionswert F(.), gemittelt über alle Muster $\{\mathbf{x}\}$, so ist als Lernalgorithmus für die Parameter M_{ij} der RBF-Neuronen der Gradientenalgorithmus für das k–te Neuron in seiner stochastischen Version

$$M^k_{ij}(t+1) = M^k_{ij}(t) - \gamma \frac{\partial}{\partial M^k_{ij}} r(\mathbf{x},\mathbf{M}^k) \tag{5.41}$$

mit $$\frac{\partial}{\partial M^k_{ij}} r(\mathbf{x},\mathbf{M}^k) = \frac{\partial}{\partial M^k_{ij}} (f(\mathbf{x},\mathbf{M}^k) - F(\mathbf{x}))^2 = 2\,(f(\mathbf{x},\mathbf{M}^k) - F(\mathbf{x})) \frac{\partial}{\partial M^k_{ij}} f(\mathbf{x},\mathbf{M}^k)$$

$$= 2\,(f(\mathbf{x},\mathbf{M}^k) - F(\mathbf{x}))\, w_k\, S_G{}^k(\mathbf{x}) \frac{\partial}{\partial M^k_{ij}} (\mathbf{M}^k\mathbf{x})^2$$

$$= 2\,(f(\mathbf{x},\mathbf{M}^k) - F(\mathbf{x}))\; w_k\; S_G{}^k(\mathbf{x})\; 2\;(\mathbf{M}^k{}_i\mathbf{x})\; x_j$$

mit der Ausgabefunktion S_G eines *hidden* Neurons und dem Skalarprodukt $\mathbf{M}^k{}_i\mathbf{x}$ aus der i-ten Zeile der Matrix $\mathbf{M}^k$ und der Eingabe $\mathbf{x}$.

Da die Koeffizienten M_{ij} nicht unabhängig voneinander sind, kann es bei der iterativen, voneinander unabhängigen Verbesserung der Koeffizienten dazu kommen, dass in bestimmten Situationen das Endergebnis nicht besser, sondern sogar schlechter sein kann. Dieses auf den ersten Blick verblüffende Resultat erklärt sich durch die Korrelation und damit wechselseitige Beeinflussung der Koeffizienten. Eine einfache Methode solche Probleme zu vermeiden besteht darin, auf unabhängige oder mindestens dekorrelierte Parameter (z.B. Drehwinkel α, Skalierung s_i, Verschiebung c_j) für die iterative Verbesserung überzugehen.

Ein weiteres Problem stellt die zu lernende Weite σ_i der RBF-Funktionen (Skalierung s_i) dar. Beim verbundenen Training gibt es keine direkte Restriktion, sie klein zu halten, um eine lokale Empfindlichkeit zu gewährleisten. Stattdessen vergrößern sich die σ_i leicht beim Training und führen zu langsamer Konvergenz und unvorhersehbaren Lösungen [MOD89].

5.5.2 Lineare Klassifizierung durch RBF-Neurone

Im ersten Kapitel lernten wir eine Klassifizierung von Mustern durch Hyperebenen kennen. Zweifelsohne ist es nicht möglich, direkt mit einer einzigen Hyperebene einen Punkthaufen im Musterraum abzutrennen wenn er von Punkten (Mustern) anderer Klassen umgeben ist, siehe Abb. 5.8(a).

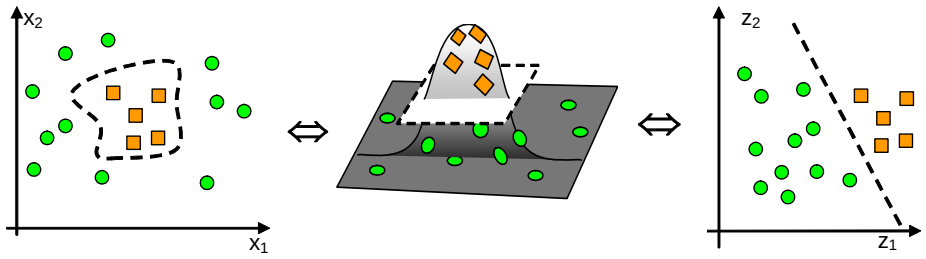


Abb. 5.8 *(a) Hyperfläche, (b) Einbettung und (c) Hyperebene zur Klassentrennung*

Könnten wir die Klassengrenze in die Form bringen

$$f_i(\mathbf{z}) = \mathrm{sgn}(\mathbf{w}_i^T\mathbf{z} + b) = \begin{cases} +1 & \text{wenn } \mathbf{z} \text{ aus Klasse } i \\ -1 & \text{sonst} \end{cases} \tag{5.42}$$

so könnten wir alle Lernverfahren für lineare Klassengrenzen anwenden, etwa den Perzeptron-Algorithmus. Dazu müssen wir zuerst den ursprünglichen Musterraum $\{\mathbf{x}\}$ aus Mustern $\mathbf{x}$ derart nicht-linear transformieren, dass für die transformierten Muster $\mathbf{z} = \boldsymbol{\varphi}(\mathbf{x})$ und $\mathbf{w}_i = \boldsymbol{\varphi}(\mathbf{c}_i)$ die Beziehung (5.42) gilt

$$f_i(\mathbf{x}) = \mathrm{sgn}(\boldsymbol{\varphi}(\mathbf{c}_i)^T\boldsymbol{\varphi}(\mathbf{x}) + b) = \begin{cases} +1 & \text{wenn } \mathbf{x} \text{ aus Klasse } i \\ -1 & \text{sonst} \end{cases} \tag{5.43}$$

Dieses Situation ist in Abb. 5.8(c) visualisiert, in der die nichtlinear-transformierten Muster in einem höher-dimensionalen Vektorraum (z.B. Kugeloberfläche im 3-dim. Raum) abgebildet sind, in dem sie leicht durch eine Hyperebene getrennt werden können, siehe Abb. 5.8(b).

Die Frage, wie wir eine lineare Klassifikation mithilfe von RBF-Neuronen durchführen können, reduziert sich so auf die Frage, wie wir solche nichtlineare Funktionen finden und ob wir dabei RBF-Neuronen verwenden können. Tatsächlich ist das Problem damit nicht einfacher geworden; eine solche nichtlineare Funktion ist nicht einfach zu finden. Glücklicherweise können wir aber das Problem umgehen: Gl.(5.43) lässt sich auch schreiben als

$$f_i(\mathbf{x}) = \mathrm{sgn}(\, K(\mathbf{c}_i,\mathbf{x}) + b) = \begin{cases} +1 & \text{wenn x aus Klasse } i \\ -1 & \text{sonst} \end{cases} \tag{5.44}$$

wobei das Skalarprodukt

$$K(\mathbf{c}_i,\mathbf{x}) = \boldsymbol{\varphi}(\mathbf{c}_i)^T\boldsymbol{\varphi}(\mathbf{x}) \quad \textit{kernel function} \tag{5.45}$$

als Kernfunktion (*kernel function*) bezeichnet wird. Für eine solche Kernfunktion müssen wir die Form der nichtlinearen Funktion $\boldsymbol{\varphi}$ nicht mehr explizit wissen, es reicht, die Kernfunktion zu kennen.

Beispiel:

Sei $\boldsymbol{\varphi}(\mathbf{x}) = (\, x_1^2,\ \sqrt{2}\, x_1x_2,\ x_2^2)^T$

dann ist

$$\boldsymbol{\varphi}(\mathbf{c})^T\boldsymbol{\varphi}(\mathbf{x}) = c_1^2x_1^2 + 2c_1c_2x_1x_2 + c_2^2x_2^2$$

$$= (c_1x_1)^2 + 2(c_1c_2x_1x_2) + (c_2x_2)^2 = (c_1x_1 + c_2x_2)^2 = (\mathbf{c}^T\mathbf{x})^2 = K(\mathbf{c},\mathbf{x})$$

Wir können also K(**c**,**x**) berechnen, ohne **φ** explizit zu kennen!

Im Allgemeinen wird uns dies durch Mercer's Theorem aus der Funktionalanalysis zugesichert:

Sei eine Funktion K(x,y) gegeben. Erfüllt sie die Bedingung

$$\int_{-\infty}^{\infty} K(x,y)g(x)g(y)dxdy \geq 0 \quad \text{für alle Funktionen g mit} \quad \int_{-\infty}^{\infty} g^2(x)dx < \infty \qquad (5.46)$$

so wird sie „positiv definierte Kernfunktion" (*positiv definite kernel*) genannt und es gibt eine Funktion **φ**(**x**) mit K(**x**,**y**) = **φ**(**x**)^T**φ**(**y**), einem Skalarprodukt in einem hochdimensionalen Raum.

Ist K(**x,y**) = Aexp(-(**x**-**y**)^{2}a) eine Gaußfunktion, so ist **φ**(**x**) unendlich-dimensional, wobei **φ**(**x**) nicht explizit bekannt sein muss. Typische Kernfunktionen, die die Anforderungen aus Gl.(5.46) von Mercer's Theorem erfüllen, sind beispielsweise

$$K(\mathbf{x},\mathbf{y}) = e^{-|\mathbf{x}-\mathbf{y}|^2 a} \qquad \textit{Gaußfunktion} \qquad (5.47)$$

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}^T\mathbf{y} + 1)^d \qquad \textit{Polynom vom Grad d} \qquad (5.48)$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(\mathbf{x}^T\mathbf{y} - \theta) \qquad \textit{Multi-layer-Perzeptron} \qquad (5.49)$$

Wie bekommen wir nun die optimalen Parameter für eine optimale lineare Klassentrennung?

5.5.3 Klassifikation durch *support vector* - Maschinen

Für den speziellen Fall der linearen Klassentrennung gibt es eine Theorie, die verspricht, optimale Resultate zu liefern, die nicht nur für eine Klassifikation, sondern allgemein für die Approximation einer Funktion optimal sind. Dazu definieren wir uns die Zielfunktion, die für eine optimale Klasifizierungsfunktion f_α bei Eingaben z,y im Erwartungswert minimal wird

$$R(\alpha) = \int \tfrac{1}{2} |f_\alpha(z) - y| \, dp(z,y) \qquad (5.50)$$

wobei über die unbekannte Wahrscheinlichkeitsdichte p(z,y) integriert wird. Für *N* diskrete Messwerte (beobachtete Muster) ist dies

$$R_e(\alpha) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}|f_\alpha(z_i) - y_i| \tag{5.51}$$

Für diese Zielfunktionen stellte nun Vapnik [VAP79] folgende Abschätzung unter der Überlegung auf, dass für eine beliebige Funktion f_α bei *N* Trainingsmustern mit der Mindestwahrscheinlichkeit $1-\eta$ gilt

$$R(\alpha) \le R_e(\alpha) + \phi\left(\frac{h}{N}, \frac{\log\eta}{N}\right) \tag{5.52}$$

$$\text{mit } \phi(a,b) = \sqrt{a(\log(2/a)+1) - b + (\log 4)/N}$$

Der minimale Fehler ist also nicht direkt erreichbar, sondern wird durch den beobachteten Fehler sowie einem Term ϕ erhöht, der nicht nur von der Anzahl *N* der Trainingsmuster und der geforderten Gewissheit abhängt, sondern auch von einer Konstanten *h*. Diese Konstante *h* ist typisch für die Diagnosekapazität der klassifizierenden Maschine bei vorhandenen Daten, also der auf den Mustern möglichen Klassifikations-Funktionen f_α und wird „**Vapnik-Chervonenkis"(VC)-Dimension** genannt. Bei der Entscheidung zwischen zwei Klassen ist *h* die maximale Anzahl der Punkte, die durch die Diagnosemaschineauf 2^h Arten in zwei Klassen geteilt werden können. Dies bedeutet, dass für jede der 2^h Trennmöglichkeiten eine Funktion f_α, $\alpha = 1..2^h$ auf der Diagnosemaschine existiert.

Beispiel VC-Dimension

Nehmen wir an, wir haben h = 3 Punkte. Dann lassen sich $2^3 = 8$ Lagen einer Trenngerade einzeichnen. In Abb. 5.9 sind vier mögliche Geraden eingezeichnet, von denen jede Einzelne jeweils eine Zerteilung der Mustermengen in zwei Klassen A und B bewirkt.

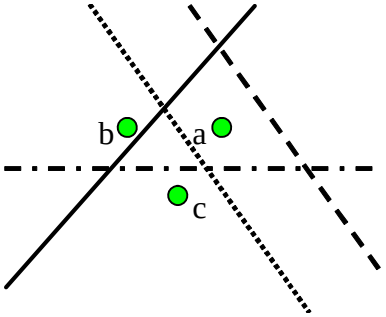


Abb. 5.9 *Mögliche Trenngeraden zweier Klassen*

Bezeichnen wir die Elemente mit *a*, *b* und *c*, so entspricht dies den Teilmengenkonfigurationen (A={},B={a,b,c}); (A={a},B={b,c}); (A={a,b},B={c}) und (A={a,c},B={b}). Vertauschen wir die Klassenbezeichnungen A und B, so erhalten wir vier weitere Konfigurationen, die wir noch nicht hatten. Also gibt es insgesamt $2^3 = 8$ mögliche Diskriminanzfunktionen f_α. Ein 1-Schicht-Perzeptron, mit dem wir eine solche Klassentrennung durchführen können, hat also die VC-Dimension h = 3. Versuchen wir, für h = 4 die 4 Punkte durch $2^4 = 16$ Arten zu trennen, also die 8 Geraden zu finden, um alle Teilmengen zu erstellen, so erleiden wir Schiffbruch: Es gibt Teilmengen, die wir nicht mit einer Geraden abtrennen können, etwa die Klassifizierung für die Funktionswerte von XOR, siehe Kapitel 2. Also ist für n = 2 Dimensionen die VC-Dimension $h = 3 < 4$. Dies gilt auch allgemein: Für Hyperebenen als Trennfunktionen im *n*-dimensionalen Raum ist h = n+1.

Wir betrachten nun die Menge *Z* aller trennenden Hyperebenen

$$Z = \{\mathbf{z} \mid \mathbf{w}^T\mathbf{z} + b = 0, \quad \mathbf{z},\mathbf{w} \in \Re^n\}$$

wobei verlangt wird, dass der Mindestabstand der Trainingspunkte $\mathbf{z}_i$ zu der Hyperebene eins sein soll

$$\min_i | \mathbf{w}^T\mathbf{z}_i + b | = 1 \tag{5.53}$$

Diese Forderung versucht der Tatsache Rechnung zu tragen, dass wir nicht wissen, wo die nächsten beobachteten Muster erscheinen werden. Wählen wir eine Trennfunktion, die möglichst viel Platz lässt zwischen der Klassengrenze und den nächstgelegenen Mustern, so ist die Wahrscheinlichkeit groß, dass neue Muster in der Nähe der alten gerade richtig klassifiziert werden. Diese Vorgabe ist als „Minimierung des strukturellen Risikos“ bekannt [VAP95].
Mit der Forderung

$$f_{w,b}(\mathbf{z}_i) = \mathrm{sgn}(\mathbf{w}^T\mathbf{z}_i + b) \overset{!}{=} y_i \in \{+1,-1\}$$

erhalten wir aus Gl.(5.53)

$$(\mathbf{w}^T\mathbf{z}_i + b)\, y_i \geq 1 \tag{5.54}$$

Berücksichtigen wir noch die Tatsache, dass einige Muster $\mathbf{z}_i$ den Mindestabstand nicht wahren können, sondern um ein $\xi \geq 0$ abweichen, so wird dies zu

$$(\mathbf{w}^T\mathbf{z}_i + b)\, y_i \geq 1 - \xi_i \tag{5.55}$$

Das Ziel des Verfahrens besteht nun darin, sowohl den beobachteten Fehler R_e aus Gl.(5.51) und den Diagnose-Zusatzterm ϕ aus Gl.(5.52) zu minimieren, als auch die Nebenbedingungen (5.55) für den Mindestabstand einzuhalten und die „Schlupfvariablen“ ξ_i zu minimieren.
Angenommen, alle Punkte befinden sich in einem Cluster. Sei nun *r* der minimale Radius einer Kugel um einen Punkt **a**, der auch die Punkte $\mathbf{z}_1 ... \mathbf{z}_n$ enthält

$$|\mathbf{z}_i - \mathbf{a}| < r$$

und $f_{w,b}(\mathbf{z}_i)$ die Entscheidungsfunktion auf diesen Punkten. Dann wissen wir mit Vapnik [VAP95], dass die Menge derartiger Entscheidungsfunktionen bei endlicher Vektorlänge des Gewichtsvektors $|\mathbf{w}| \leq A$ eine VC-Dimension hat, die durch die Längen begrenzt ist

$$h < r^2A^2 + 1 \tag{5.56}$$

Anstatt die VC-Dimension direkt zu reduzieren, um die Terme in Gl. (5.52) zu minimieren, reicht es, die Länge des Gewichtsvektors zu minimieren. Wir können also eine neue, zu minimierende Zielfunktion

$$T(\mathbf{w}, \xi_i) = \tfrac{1}{2}\,\mathbf{w}^2 + \gamma \sum_{i=1}^{N} \xi_i \tag{5.57}$$

aufstellen, wobei gleichzeitig die Nebenbedingungen aus Gl. (5.55) erfüllt sein müssen. Die Minimierung des ersten Terms bedeutet eine implizite Minimierung der VC-Dimension der resultierenden Diagnosemaschine. Der zweite Term ist eine Abschätzung für die Anzahl der Fehlklassifikationen der Trainingsmenge.
Nehmen wir Schlupfvariable an, für die in Gl. (5.55) die Gleichheit gilt, so können wir die Nebenbedingungen umformulieren

$$g(\mathbf{w},i) = 1 - (\mathbf{w}^T\mathbf{z}_i + b)y_i - \xi_i = 0 \tag{5.58}$$

und damit eine Lagrangefunktion L aufstellen

$$L(\mathbf{w},\mu_1,..,\mu_N) = T(\mathbf{w}, \xi_i) + \sum_{i=1}^{N} \mu_i g(\mathbf{w}, i) \tag{5.59}$$

Die Extremwerte dieser Funktion werden angenommen bei

$$\frac{\partial}{\partial \mathbf{w}} L = \mathbf{w} - \sum_{i=1}^{N} \mu_i y_i \mathbf{z}_i = 0 \quad \text{oder} \quad \mathbf{w} = \sum_{i=1}^{N} \mu_i y_i \mathbf{z}_i \tag{5.60}$$

Die Grenze **w** ist also bestimmt durch alle Trainingsmuster (y_i, $\mathbf{z}_i$), bei denen mit $\mu_i \neq 0$ die Nebenbedingung erfüllt ist, die $\mathbf{z}_i$ werden deshalb „**Support-Vektoren**“ genannt.
Die Koeffizienten μ_i dazu erhalten wir durch eine neue Zielfunktion *W*, die nicht nur kurze **w** erzeugt, sondern auch nur wenige Grenzpunkte mit großem μ_i berücksichtigt

$$W(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \mu_i - ½ \mathbf{w}^2 = \sum_{i=1}^{N} \mu_i - ½ \sum_{i=1}^{N} \sum_{j=1}^{N} \mu_i \mu_j y_i y_j \mathbf{z}_i \mathbf{z}_j \overset{!}{=} \max \tag{5.61}$$

mit den Nebenbedingungen

$$0 < \mu_i < \gamma, \; i = 1,...,N \text{ und } \sum_{i=1}^{N} \mu_i y_i = 0 \tag{5.62}$$

wobei die μ_i nur für die Punkte $\mathbf{z}_i$ ungleich null sind, die der Gl. (5.55) genügen. Diese Punkte reichen also aus, die Grenzfläche zwischen den Klassen festzulegen; alle anderen sind irrelevant. Alle anderen Punkte (Trainingsmuster) liefern Nebenbedingungen, die das Ergebnis (das Extremum) nicht beeinflussen; die Nebenbedingungen und damit auch die Punkte sind redundant und fallen deshalb mit $\mu_i = 0$ weg. Diese typische Eigenschaft des Lagrange-Mechanismus ist sehr praktisch für uns und unterstützt das Ziel, nur den Abstand der am dichtesten zur Hyperfläche gelegenen Punkte durch die Lage der Hyperfläche zu maximieren.
Die Zielfunktion W kann man bei bekannten Trainingsmustern mit Standardmethoden der quadratischen Optimierung numerisch minimieren. Man erhält abschließend die Koeffizienten μ_i für eine optimale Klassifizierung eines Musters $\mathbf{x}_k$ mit der Funktion

$$f(\mathbf{x}) = \text{sgn}\left(\sum_{i=1}^{N} \mu_i y_i \mathbf{z}_i^T \mathbf{z} + b \right) = \text{sgn}\left(\sum_{i=1}^{N} \mu_i y_i K(\mathbf{x},\mathbf{c}_i) + b \right) \tag{5.63}$$

Dabei werden nur die Trainingsmuster als Supportvektoren verwendet, bei denen $\mu_i \neq 0$ gilt und die damit als direkte Nachbarn der Klassengrenze wichtig für deren Verlauf sind. Für den Gauß’schen Kern ergibt sich somit als Klassifizierungsfunktion

$$f(\mathbf{x}) = \mathrm{sgn}\left(\sum_{i=1}^{N} \mu_i y_i \; e^{-|\mathbf{x}-\mathbf{c}_i|^2 a} + b\right) \tag{5.64}$$

Dies entspricht einem RBF-Netz aus zwei Schichten, wobei die erste Schicht als Zentren $\mathbf{c}_i$ die Trainingsmuster mit $\mu_i \neq 0$ verwendet, und die zweite Schicht aus einem binären Neuron mit der Signumfunktion als Ausgabefunktion besteht, das als Gewichte die μ_i besitzt. Erstaunlicherweise wird mit den Supportvektoren nicht nur die Zentren, sondern auch die optimale Anzahl der RBF-Neuronen bestimmt.

In Abb. 5.10 (a) sind zur Veranschaulichung die Muster von zwei Klassen als schwarze Punkte und weiße Kreise eingezeichnet. Die Trenngerade wird dabei nur durch die zwei umkreisten Muster festgelegt. Dies ist ein wichtiger Unterschied zu der Funktion von traditionellen RBF-Algorithmen, bei denen das RBF-Zentrum ins Dichtezentrum der Muster gelegt wird und nicht an den Rand. In Abb. 5.10 (b) ist die nichtlineare Klassengrenze für eine andere Klassenverteilung gezeigt. Die RBF-Zentren sind wieder nicht in den Dichtezentren, sondern sind am Rand im Hinblick auf optimale Klassentrennung und Diskriminierung gewählt und legen die Lage der Hyperfläche fest.

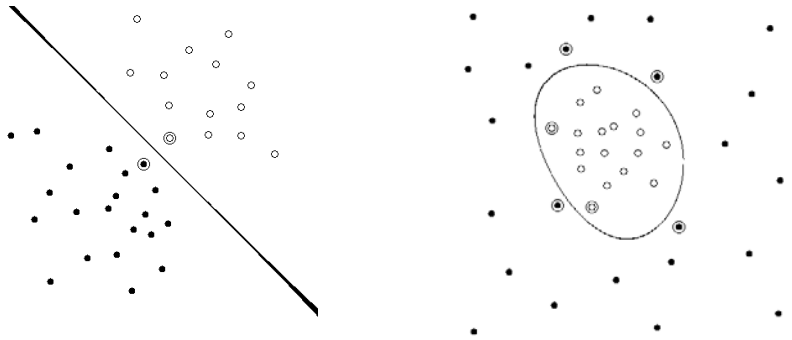


Abb. 5.10 *(a) Trennung durch Hyperebene (b) Trennung durch Hyperfläche*

Die Leistung solch optimierter Diagnosemaschinen ist bei günstiger Wahl der Kernfunktion (etwa des Gaußkernels) durch die gleichzeitige Berücksichtigung aller Trainingsmuster meist etwas besser als das konventioneller, adaptiver RBF-Approximationen.

Beispiel:

Für die Klassifizierung von handgeschriebenen Ziffern 0..9 der US-Post Datenbank wurden drei verschiedene Diagnosemaschinen konstruiert: Ein konventionelles

RBF-Netz, ein RBF-Netz, dessen RBF-Zentren aus Supportvektoren besteht, und eine reine Supportvektormaschine. Für einen fairen Vergleich wurde als Anzahl der RBF-Neuronen die Anzahl der errechneten Supportvektoren verwendet. Die Ergebnisse sind in der folgenden Tabelle aus [SSB97] zusammengefasst:

Daten	Klassifikations-Fehlerrate		
US Postal Service	Klass. RBF	RBF mit SV Zentren	Reine SV-Maschine
Training (7291 Muster)	1,7 %	0,0 %	0,0 %
Test (2007 Muster)	6,7 %	4,9 %	4,2 %

5.6 Netzaufbau und Approximationsfehler

Bisher wurde versucht, die eingangs gestellten Fragen über Anzahl und Art der Glockenfunktionen sowie über den optimalen Radius des rezeptiven Feldes durch die Angabe theoretisch motivierter Lernverfahren zu beantworten.

Interessanterweise gibt es auch theoretische Ergebnisse, die aus der mathematischen Approximationstheorie stammen und auf die RBF anwendbar sind. Eine der Arbeiten dazu wurde von Xu, Kryzak und Yuille verfaßt [XUKY94] und erweitert die Ergebnisse für *Parzen-window*-Basisfunktionen, genannt *kernel regression estimators (KRE)*, auf allgemeine, nach (4.6.1b) normierte Glockenfunktionen.
Die wichtigsten Ergebnisse, möglichst einfach formuliert, sind folgende:

- *Zusicherung der Konvergenz*

  Sei $y = f^*(\mathbf{x})$ ein Schätzwert für die unbekannte Funktion $f(\mathbf{x})$. Es existieren Schätzfunktionen, basierend auf Linearkombinationen von RBF, so dass der Fehler $d(\mathbf{x}) = |f^*(\mathbf{x})–f(\mathbf{x})|$ mit steigender Anzahl m der RBF für alle Punkte $\mathbf{x}$ kleiner wird (*punktweise konvergiert*).

- *Fehlerverminderung*

  Betrachten wir die Klasse von RBF, die stetige, quadratisch integrierbare Ableitungen der Ordnung q besitzen. Dann vermindert sich der Fehler mit der Ordnung $O(m^{-2q/(2q+n)})$ bei m Basisfunktionen und $n = \dim(\mathbf{x})$. Dies bedeutet, dass der Fehler umso größer ist, je größer die Eingabedimension ist, und umso kleiner, je glatter die Basisfunktion ist.

- *Optimale Form der RBF*

  Sei ein "rezeptives Feld" definiert als (multidimensionales) Eingabeintervall (kompakte Punktmenge), für das $SG(x,Rm) > s$ für die konstante Schwelle s gilt.

  Sei $R_m = m–\tau$ mit $0 < \tau < 1/n$. Dann gilt:

Sind die RBF auf kompakten Punktmengen (endliche Intervalle) definiert, wie beispielsweise Rechteckfunktionen etc., so vermindert sich der Fehler bei punktweiser Konvergenz und n=1 bestenfalls mit der Ordnung $O(m^{-1/3})$, bei Gaußfunktionen mit $O(m^{-\tau} \ln m)$ und bei Polynomen mit einem "Schwanz" mit $O(m^{-1/5})$. Dies bedeutet, dass bei fester Anzahl die verwendeten RBF für einen kleinen Fehler möglichst kurze "Reichweiten" haben sollten. Bei Gauß-Funktionen lässt sich dies durch geringes σ und durch Null setzen der Funktion bei geringen Funktionswerten ("abschneiden") erreichen.

- *Optimaler Radius des rezeptiven Feldes*

  Der Fehler $d(\mathbf{x})$ besteht aus der Summe zweier Anteile der Ordnungen $O(R_m^q)$ und $O(1/(mR_m^n)^{-1/2})$, so dass sowohl bei zu kleinem rezeptivem Feld als auch bei zu großem der Fehler anwächst. Das ausgewogene Optimum ist von der Ordnung

  $$R_m^* = O[(g(\mathbf{x})n^2/q^2m)^{1/(2q+n)}]$$

  wobei $g(\mathbf{x})$ eine Funktion ist, die von $f(\mathbf{x})$, der Wahrscheinlichkeitsdichte der $\mathbf{x}$ und statistischen Proportionen von $y(\mathbf{x})$ abhängt. Da wir über diese Angaben meist nicht verfügen, lässt sich der optimale Radius nicht einfach errechnen; wir bleiben auf unsere adaptiven Verfahren angewiesen. Allerdings können wir trotzdem Schlüsse aus der obigen Formel ziehen: Je mehr Basisfunktionen wir verwenden, um so kleiner können wir den Radius jeder Basisfunktion wählen. Außerdem sind "Rauheit" q der RBF und Eingabedimension n wichtig: Ist der Quotient n/q konstant beim Übergang von einem Netz zum anderen, so lassen sich bei den glatteren Funktionen größere Radien verwenden.

Ähnliche Aussagen lassen sich übrigens auch für den Erwartungswert des quadratischen Fehlers $d^2(\mathbf{x})$ für den gesamten Eingaberaum $\{\mathbf{x}\}$ aufstellen [XUKY94].

Die obigen Angaben betrachten dabei Klassen von konstruierbaren Funktionen, die die gewünschten Leistungen erbringen. Dies besagt aber nicht, dass man im speziellen Fall mit speziellen RBF's nicht noch bessere Ergebnisse erbringen könnte: die Konvergenzraten sind immer nur als schlechteste Raten für die konstruierten Funktionen zu interpretieren.

5.7 Anwendungen

Es gibt sehr viele Anwendungen von radialen Basisfunktionen. An dieser Stelle seinen einige herausgegriffen, um die Vielfalt der Anwendungsmöglichkeiten dieser Approximationstechnik zu zeigen. Für spezielle Probleme sei auf die einschlägigen Konferenzen verwiesen.

5.7.1 Erkennen von 3-dim. Objekten

Das Erkennen der räumlichen Lage (horizontaler Drehwinkel θ und vertikaler Drehwinkel φ) eines Objekts nur durch das "Anschauen" von 2-dimensionalen Projektionen des Objekts erscheint für Menschen sehr einfach, für Computer aber als schwierige Aufgabe. Obwohl es theoretisch möglich ist, anhand von drei Ansichten eines Objekts bei Kenntnis der Referenzpunkte das gesamte 3-dim. Objekt zu rekonstruieren [ULL89], zeigten Poggio und Edelman [POGE90], dass das Problem mit RBF Netzen auch praktisch relativ einfach lösbar ist. Dazu nahmen sie als Objekt ein Drahtmodell, dessen 3-dim. Konfiguration durch wenige Referenzpunkte eindeutig bestimmt ist, wobei sich die Referenzpunkte nicht verdecken, s. Abb. 5.11. Wären dabei Punkte verdeckt, so würden die entsprechenden Neuronengewichte im Schnitt nur weniger trainiert, aber nicht ausgeschlossen. Die Figur in Abb. 5.11 besteht aus 6 Punkten P_1,..,P_6, die durch Linien verbunden sind, und lässt sich durch Drehen und Kippen mit den Winkeln θ und φ um den Schwerpunkt verändern. Diese beiden Winkel des Beobachters, der Längen- und der Breitengrad auf der Kugel um die Figur, sollen aus der 2-dim. Projektion erschlossen werden, ebenso die Figur im "Originalzustand" bei θ = φ = 0.

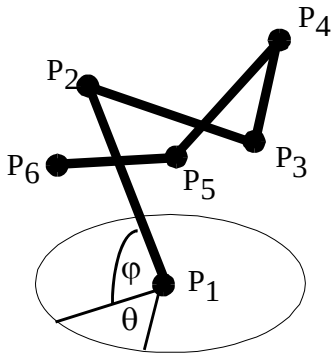


Abb. 5.11 *Eine 3-dim Figur mit 6 Referenzpunkten*

In jeder Projektion lassen sich die Koordinaten der 6 Punkte $P_1 = (x_1(1), x_2(1))$, …, $P_6 = (x_1(6), x_2(6))$ angeben. Falls die Winkel eine direkte Funktion $f(P_1,..,P_6)$ der Projektion sind, können wir sie auch approximieren, z.B. mit einem RBF-Netz. Dazu verwendeten Poggio und Edelman ein einfaches, nicht-normiertes RBF-Netz von der Struktur in Abb. 5.12.

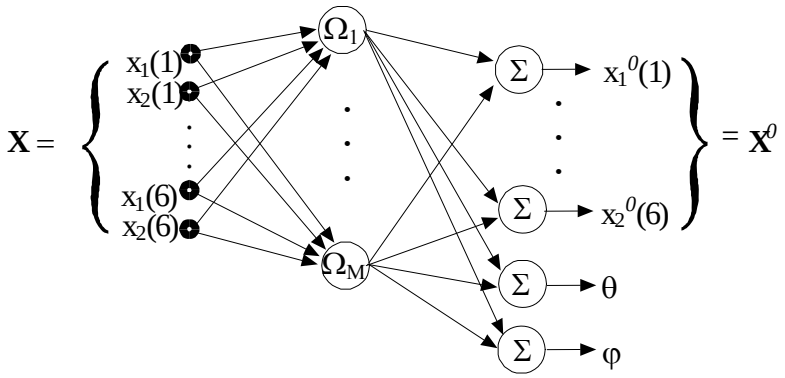


Abb. 5.12 *Struktur des nicht-normierten, wachsenden RBF-Netzes*

Die zufällig erzeugte Trainingseingabe **x**, bestehend aus den Koordinaten P_1,..,P_6, wird eingegeben. Für jedes Trainingsmuster wird ein extra RBF-Neuron Ω_i erzeugt und durch Fehlerkorrektur (Delta-Lernregel) die neuen Gewichte angepasst. Das so erzeugte Netz hat bei *M* Trainingsmustern auch *M* versteckte Neuronen. Die linearen Ausgabeneuronen Σ_i erzeugen zum einen die "Originalfigur" $\mathbf{x}^0$ als auch die gesuchten Winkel θ und φ. Bei der Simulation reichten 40-100 Trainingsmuster aus, um über den ganzen Winkelbereich von $\theta = 180$ Grad eine gute Vorhersage zu erreichen; bei einem Teilausschnitt bis 45 Grad sogar nur mit zwei RBF, trainiert mit 40 Mustern.

Die Größenordnung der Trainingsmuster für eine Wiedererkennung im Gesamtraum ist dabei durchaus kompatibel zu menschlichen Leistungen: Um ein neues Objekt nach einer Drehung wiederzuerkennen, soll sie für Menschen nicht mehr als 30 Grad betragen. Der gesamte 180°×360° Bereich besteht aber gerade aus 72 Segmenten der Größe 30°×30°, so dass mindestens 72 Trainingsmuster für Menschen nötig sind.

Beim RBF-Netz ist sogar dabei eine Klassifikation möglich: Vergleicht man die rücktransformierte Figur $\mathbf{x}^0_?$ mit der gespeicherten Originalfigur $\mathbf{x}^0_\alpha$, so kann man den Euklid'schen Abstand $|\mathbf{x}^0_? - \mathbf{x}^0_\alpha|$ als Ähnlichkeitskriterium verwenden und mit einem Schwellwert s eine Klassifikationsentscheidung treffen: Ist $|\mathbf{x}^0_? - \mathbf{x}^0_\alpha| < s$, so liegt eine (evtl. leicht deformierte) Version des Originals vor, sonst nicht. Damit ist die beim Assoziativspeicher sonst problematische Klassifikation affin transformierter Muster hier möglich.

Aufgaben

1) Man errechne die Lerngleichungen für den Drehwinkel α, die Skalierung s_i und die Verschiebung $\mathbf{c}_j$ mit dem Gradientenalgorithmus, wobei man von der Beziehung

$$\frac{\partial \mathrm{r}\left(\mathbf{x}, \mathbf{M}^{\mathrm{k}}(\alpha)\right)}{\partial \alpha} = \frac{\partial \mathrm{r}\left(\mathbf{x}, \mathbf{M}^{\mathrm{k}}\right)}{\partial \mathrm{M}_{\mathrm{ij}}^{k}} - \frac{\partial \mathrm{M}_{\mathrm{ij}}^{k}(\alpha)}{\partial \alpha}$$

Gebrauch machen kann.

2) Man erstelle ein Computerprogramm, um mit 10 RBF die Funktion f(x) = sin(x) in einem Intervall zu approximieren. Hierbei verwende man a) normierte und b) nicht normierte RBF.

Wodurch machen sich die Unterschiede bemerkbar? Wie werden Konvergenzgeschwindigkeit und Approximationsgenauigkeit beeinflusst?

6 Rückgekoppelte Netze

Die Teile des menschlichen Gehirns, die mit Aufgaben der sensorischen Verarbeitung (*periphere Leistungen*) beschäftigt sind, haben meist einen sehr speziellen, geordneten Aufbau. Dies lässt sich mit einem Schichtenmodell erklären, bei dem die sensorische Verarbeitung in mehreren Stufen (Schichten) erfolgt (*pipeline Modell*).

Betrachtet man in die zeitliche Folge, mit der eine sensorische Erregung diese Schichten durchläuft, so bleibt pro Schicht so wenig Zeit, dass bei einer minimalen Taktrate der Neuronen von ca. 1 ms nur noch eine fast ausschließliche *feed-forward* Verarbeitung denkbar ist. Beispielsweise lässt sich nach einem "Klick"-Laut am Ohr in den verschiedenen Stufen des auditiven Systems eine Erregungswelle ableiten, die ihr Maximum nach 5 ms in der Cochlea, nach 7 ms in dem Olivenkernkomplex, nach 9 ms im *colliculus inferior*, nach 13 ms im Genicularischen Körper und nach 24 ms im Auditorischen Cortex annimmt [KUFF53].

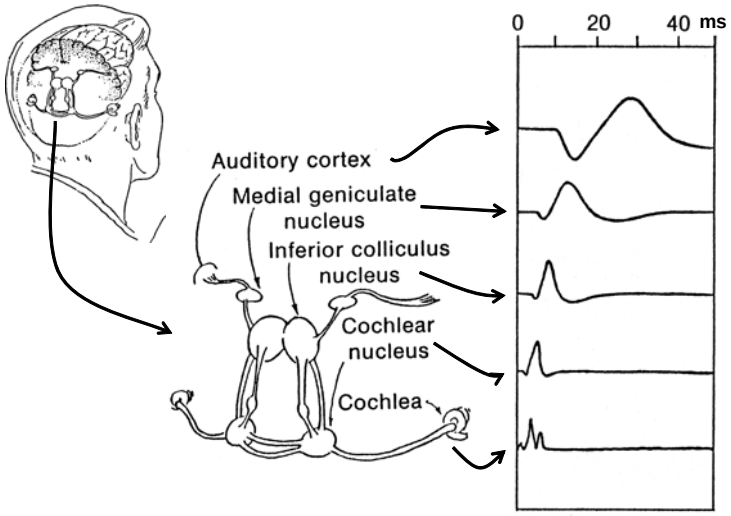


Abb. 6.1 *Schema der auditiven Verarbeitung im Gehirnstamm (nach [KUFF53])*

Anders dagegen sieht es für das Großhirn (*Cortex*) aus, von dem wir sehr wenig wissen. Einige physiologische Hinweise besagen, dass im Unterschied zu den peripheren Arealen (z.B. visuelle Zentren, Areale 17,18,19 in Abb. 1.1) beispielsweise beim Mäusecortex mit ca. einer Million Eingangsfasern und ca. 20 Millionen Zellen, von denen die Mehrheit (85%) einheitliche, exitatorische Pyramidenzellen sind, anstelle einer Schichtenstruktur eine ungeordnete, lose Struktur von rückgekoppelten, gleichartigen Neuronen vorliegt [BRAI89]. Einige Forscher glauben deshalb, dass die Funktion des menschlichen Großhirns eher der eines großen Assoziativspeichers ähnelt, bei dem die sensorischen, stark kodierten Informationen miteinander und mit einer großen Menge bereits gespeicherter Informationen in Verbindung gebracht werden. In den folgenden Abschnitten wollen wir uns deshalb näher mit den Modellen und Eigenschaften rückgekoppelter, parallel arbeitender Systeme (*recurrent* oder *feedback networks*) aus kleineren Einheiten befassen.

6.1 Lineare assoziative Speicher

Es gibt verschiedene Modelle für Systeme aus rückgekoppelten, formalen Neuronen. Die Eigenschaft, assoziativ Muster zu speichern, ist eines der Grundproportionen neuronaler Modelle. Betrachten wir deshalb in diesem Abschnitt zuerst die Modelle und Eigenschaften assoziativer Speicher.

6.1.1 Brain-state-in-a-box

Eines der klassischen Modelle besteht aus einer Erweiterung des korrelativen Assoziativspeichers. Im Unterschied zu dem dort vorgestellten Modell wirken die Ausgänge hier auch auf die Eingänge zurück, so dass sich eine Rückkopplung ergibt.

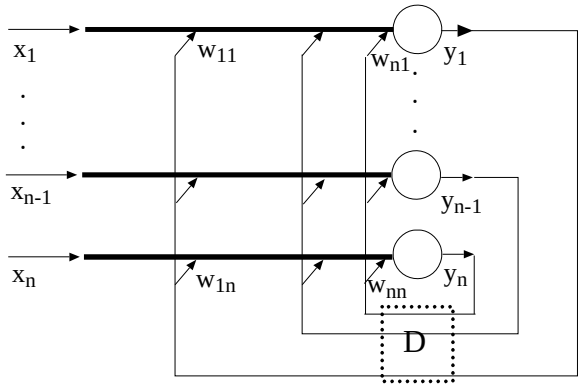


Abb. 6.2 *Ein rückgekoppelter Assoziativspeicher*

In der Abb. 6.2 ist gezeigt, dass die Eingabekomponenten jeweils separat an den Eingang (dicker Strich) eines Neurons herangeführt und nicht gewichtet sind. Nach der Summenbildung mit den gewichteten Rückkopplungen wird das Ausgangssignal um einen Zeittakt verzögert (gepunktetes *delay*-Element) und gelangt wieder an die Eingabe. Die diagonalen Pfeile verdeutlichen eine Ankopplung der Ausgangsleitungen über Gewichte w_{ij} an ein Neuron. Betrachten wir zunächst die Stabilität des rückgekoppelten Systems, ohne eine Eingabe vorzunehmen, bei möglichen, internen Zuständen $\mathbf{x}(t) = \mathbf{y}(t-1)$.

6.1.2 Speicherung von Mustern

Ein an den Eingängen initial vorhandenes, nicht eingespeistes Signal $\mathbf{x}(t)$ wird in diesem System linear auf die Ausgabe $\mathbf{y}(t)$ abgebildet, so dass in diesem linearen Modell zunächst die Ausgabe als Spaltenvektor

$$\mathbf{y} = \mathbf{S}(\mathbf{z}) = \mathbf{W}\mathbf{x} \qquad \text{mit } \mathbf{S}(\mathbf{z}) = \mathbf{z} \tag{6.1}$$

vorhanden ist mit der Identität als Ausgabefunktion. Im Unterschied zu den einfachen Assoziativspeichern ist hier nicht nur eine autoassoziative Kodierung $(\mathbf{x}^k,\mathbf{x}^k)$ des k-ten zu speichernden Ein/Ausgabepaares vorhanden, sondern die Aktivität wirkt auch wieder auf sich zurück und beschränkt dadurch die Art der speicherbaren Muster $\mathbf{x}$.
Betrachten wir dazu zunächst nur das rückgekoppelte System *ohne* zusätzliche Eingabe. In diesem System lassen sich nur dann Muster $\mathbf{x}^k$ autoassoziativ stabil speichern, wenn die Bedingung

$$\mathbf{x}^k(t+1) = \mathbf{y}^k(t) = \mathbf{W}\,\mathbf{x}^k(t) \stackrel{!}{=} \lambda_k \mathbf{x}^k(t) \tag{6.2}$$

erfüllt ist, also die Ausgabevektoren in die gleiche Richtung zeigen wie die Eingabevektoren: Das System ist in *Resonanz.* Die speicherbaren Muster, die Resonanzzustände, stellen damit die *Eigenvektoren* der Gewichtsmatrix **W** dar und ihre Verstärkung λ_k die Eigenwerte; ihre Anzahl ist gleich dem Rang der n×n Matrix **W**, also maximal *n*.

Wir wissen nun, dass die Matrix **W** durch ihre Eigenvektoren festgelegt ist. Wie erhalten wir **W** ? Dazu betrachten wir wie beim *feedforward*-Assoziativspeicher die Speicherung über die Hebb'sche Regel. Die Erzeugung der Elemente von **W** in stabilem Systemzustand mit $x_i(t) = x_i(t+1) = y_i(t)$ ist mit dem Gewichtszuwachs

$$\Delta w_{ij} = \gamma y_i x_j = \gamma y_i y_j = \gamma y_j y_i = \Delta w_{ji} \tag{6.3}$$

und

$$w_{ij} = \sum_k \Delta w_{ij}$$

so wird die resultierende Autokorrelationsmatrix bei initialen Anfangsgewichten $w_{ij}(0) = 0$ von dem äußeren Produkt der beiden Vektoren aufgespannt

$$\mathbf{W} = \mathbf{W}^T = \sum_k \gamma_k \mathbf{y}^k (\mathbf{y}^k)^T = \sum_k \gamma_k \mathbf{x}^k (\mathbf{x}^k)^T \tag{6.4}$$

W ist symmetrisch und ihre orthogonalen Eigenvektoren haben damit reelle Eigenwerte λ. Die Speichermuster $\mathbf{x}^k$ können in diesem Fall also nur dann die Eigenvektoren und damit "stabile Zustände" darstellen, wenn sie nicht nur *linear unabhängig*, sondern auch *orthogonal* kodiert sind.

$$(\mathbf{x}^j)^T \mathbf{x}^k = \begin{cases} (\mathbf{x}^k)^2 & \text{für } j = k \\ 0 & \text{sonst} \end{cases} \tag{6.5}$$

Wählen wir uns unsere zu speichernden Muster also orthogonal (nicht unbedingt auch orthonormal), so können wir sie direkt mit Hilfe der Hebb'schen Regel abspeichern. Allerdings ist die Konfiguration nur dann stabil, wenn die Länge des Vektors $\mathbf{x}^k$ nicht mit der Zeit wächst, sondern konstant bleibt. Dies bedeutet, dass in Gl. (6.2) die Eigenwerte $\lambda_k = 1$ sein müssen. Dies bedeutet, mit Gl. (6.4) und Gl.(6.5)

$$\lambda_k \mathbf{x}^k = \mathbf{x}^k = \mathbf{W}\mathbf{x}^k = \sum_i \gamma_i \mathbf{x}^i (\mathbf{x}^i)^T \mathbf{x}^k = \gamma_k \mathbf{x}^k (\mathbf{x}^k)^2 \;\Leftrightarrow\; \gamma_k = (\mathbf{x}^k)^{-2} \tag{6.6}$$

Was passiert nun mit einem beliebigen Muster, einer Abweichung von den bereits gespeicherten $\mathbf{x}^k$? Die gespeicherten Eigenvektoren bilden ein Basissystem, so dass ein beliebiger Zustand **x** als Linearkombination dieser Basis dargestellt werden kann

$$\mathbf{x}(t) = \sum_k a_k(t)\mathbf{x}^k \quad \text{mit } (\mathbf{x})^T\mathbf{x}^k = \sum_k a_k\,(\mathbf{x})^T\mathbf{x}^k = a_k(\mathbf{x}^k)^2 \;\Leftrightarrow\; a_k = \frac{\mathbf{x}\mathbf{x}^k}{\left(\mathbf{x}^k\right)^2} \tag{6.7}$$

Wird ein beliebiges $\mathbf{x}$ in das System eingespeist, so ist die Ausgabe mit $\mathbf{y}(t) =: \mathbf{x}(t+1)$

$$\mathbf{x}(t+1) = \mathbf{W}\mathbf{x}(t) = \sum_k \gamma_k \mathbf{x}^k(\mathbf{x}^k)^T\Big(\sum_j a_j\mathbf{x}^j\Big) = \sum_k \mathbf{x}^k\, a_k(t) = \mathbf{x}(t) \tag{6.8}$$

Jedes Muster bleibt als Linearkombination der $\mathbf{x}^k$ genauso erhalten und stellt einen stabilen Zustand dar, den wir aber so nicht speichern können (Warum nicht?).

6.1.3 Mehrfache Speicherung und Ausgabe

Betrachten wir weiterhin den Fall orthogonaler Muster $\mathbf{x}^k$. Nachdem jedes Muster $\mathbf{x}^k$ einige Male (genau b^k-mal) präsentiert und gespeichert wurde, ergibt sich die Gewichtsmatrix aus Gl.(6.4) zu

$$\mathbf{W} = \sum_k b_k\gamma_k\mathbf{x}^k(\mathbf{x}^k)^T \tag{6.9}$$

Nehmen wir an, dass mit Gl. (6.6) $\gamma_k = (\mathbf{x}^k)^{-2}$ gilt, so ist bei der Eingabe des Musters $\mathbf{x}^r$ mit

$$\mathbf{W}\mathbf{x}^r = b_r\mathbf{x}^r \qquad (\mathbf{x}^k)^T\mathbf{x}^r = 0 \text{ bei } r \neq k \tag{6.10}$$

bei fester Gesamtzahl $M = \Sigma_k b_k$ an Speicherungen die Ausgabe wieder ein Eigenvektor, so dass $b_r = \lambda_r$ gilt und damit die relativen Eigenwerte $\lambda_r/\Sigma_k\lambda_k$ proportional zur **Auftrittswahrscheinlichkeit** eines gespeicherten Musters $\mathbf{x}^i$

$$P_i = b_i / M \tag{6.11}$$

sind; die relativen Größenunterschiede der Eigenwerte bedeuten hier einen entsprechenden Unterschied in den Auftrittswahrscheinlichkeiten. Haben alle gespeicherten Muster die gleiche Auftrittswahrscheinlichkeit, so sind auch alle Eigenwerte gleich. In diesem Fall ist auch jede beliebige Linearkombination der $\mathbf{x}^k$ Eigenvektor und damit stabiler Zustand, wie man leicht mit Gl.(6.8) nachprüfen kann.

Haben die gespeicherten Vektoren einen Erwartungswert null, etwa als

$$\sum_k P_k\,\mathbf{x}^k = \langle\mathbf{x}\rangle_k = \mathbf{0} \tag{6.12}$$

so ist der Erwartungswert der gesamten Speichermatrix

$$\langle\mathbf{W}\rangle = \sum_k P_k\,\mathbf{x}^k(\mathbf{x}^k)^T = \langle\mathbf{x}\mathbf{x}^T\rangle - \mathbf{0} = \langle\mathbf{x}\mathbf{x}^T\rangle - \langle\mathbf{x}\rangle\mathbf{x}^T - \mathbf{x}\langle\mathbf{x}\rangle^T + \langle\mathbf{x}\rangle\langle\mathbf{x}\rangle^T \tag{6.13}$$

$$= \langle (\mathbf{x} - \langle \mathbf{x} \rangle)(\mathbf{x} - \langle \mathbf{x} \rangle)^T \rangle = \mathbf{C}$$

proportional der positiv-semidefiniten Kovarianzmatrix **C**, so dass alle Eigenwerte $\lambda_r > 0$ sind.

Betrachten wir nun einen beliebigen, zufälligen Vektor **x**. Wie wir aus Gl.(6.8) wissen, bedeutet die lineare Abbildung mit der Matrix **W** eine Projektion auf die Eigenvektoren $\mathbf{x}^k$. Geht aber **W** in die Kovarianzmatrix **C** der Eingaben **x** über, so bedeutet nun die Projektion auf die Eigenvektoren von **C** eine **Hauptkomponentenanalyse** (PCA) der Eingabe.

Damit maximiert die Projektion $y_1 = \mathbf{x}^T\mathbf{e}_1$ des Vektors **x** auf den Eigenvektor mit dem größten Eigenwert die Varianz

$$\text{var}\,(y_1) = \langle (y_1 - \langle y_1 \rangle)^2 \rangle = \langle y_1^2 \rangle = \lambda_1 \qquad \langle y_i \rangle = 0 \tag{6.14}$$

Konstruieren wir orthogonal dazu in Richtung der zweitgrößten Varianz den zweiten normierten Eigenvektor und so fort, so erhalten wir eine Darstellung von **x** durch ein orthonormales Basissystem von Eigenvektoren, die als die wesentlichen, informationstragenden Merkmale der Eingabe $\{\mathbf{x}\}$ angesehen werden können. Die Varianz des zufälligen Vektors **x** ist damit

$$\text{var}\,(\mathbf{x}) = \langle |\mathbf{x} - \langle \mathbf{x} \rangle|^2 \rangle = \langle |\mathbf{x}|^2 \rangle = \langle \, (\sum_k y_k \mathbf{x}^k)\,(\sum_k y_k \mathbf{x}^k)^T \, \rangle = \langle \, \sum_k y_k^2 \, \rangle$$

$$= \sum_k \text{var}\,(y_k) = \sum_k \lambda_k \tag{6.15}$$

die nun maximal ist, da die Variablen y_i nicht mehr korreliert sind. Bei normierten Vektoren mit $|\mathbf{x}| = 1$ sind es die Eigenvektoren, die die größten Antworten liefern: unter allen **x** bringt $\mathbf{x}^1$ mit $|\mathbf{y}|^2 = \lambda_1$ die größte Antwort; $\mathbf{x}^2$ mit $|\mathbf{y}|^2 = \lambda_2$ die zweitgrößte und so fort.

6.1.4 Eingabe und Rückkopplung

Bisher betrachteten wir das gesamte System als ein geschlossenes, rückgekoppeltes System, in dem nur wenige der injizierten Zustände bei der Rückkopplung "überleben" können. Angenommen, wir betrachten einen diskreten Zeitpunkt t, an dem gleichzeitig Eingabe und Rückkopplung wirken sollen. Dann ist dieser Zustand nach einem Zeitschritt mit Gl.(6.8) und Gl. (6.2)

$$\mathbf{x}(t+1) = \mathbf{y}(t) \equiv \mathbf{x}(t) + \mathbf{y}(t\text{-}1) = \mathbf{x}(t) + \mathbf{W}\mathbf{x}(t\text{-}1)$$
$$= \sum_k a_k(t)\,\mathbf{x}^k + \mathbf{W}(\sum_k a_k(t-1)\,\mathbf{x}^k)$$

$$= \sum_k (a_k(t) + a_k(t-1)\,\mathbf{W})\,\mathbf{x}^k = \sum_k \left(a_k(t) + \lambda_k\, a_k(t-1)\right)\mathbf{x}^k \tag{6.16}$$

Da die Eigenwerte λ_k alle positiv sind, gilt sowohl für positive als auch für negative Koeffizienten a_k

$$\left(a_k(t) + \lambda_k\, a_k(t-1)\right)^2 \geq \left(a_k(t)\right)^2 \quad \text{und somit}$$

$$|\mathbf{x}(t+1)| \geq |\mathbf{x}(t)| \tag{6.17}$$

Unter diesen Bedingungen sind nur die sog. "degenerierten" Eigenvektoren mit identischem $\lambda_j = 0$ Fixpunkte im System. Für alle anderen Eingaben ist das System instabil: die Vektoren werden immer länger; das System „schaukelt sich auf".

Hier zeigt sich die Notwendigkeit, eine Wachstumsbeschränkung in dem rückgekoppelten System zur Stabilisierung einzuführen. Bei biologischen Neuronen kann die Variable x_i nur einen endlichen Wert, etwa aus dem Intervall [0,Z] annehmen; es gibt eine begrenzte Spikerate von maximal ca. 1000Hz. Fordern wir dies als Beschränkung für alle Komponenten des Zustandsvektors

$$y_i \in [-Z, +Z] \tag{6.18}$$

so sind alle Zustände **y** des Systems in einer abgeschlossenen Menge des *m*-dimensionalen Raums enthalten; im dreidimensionalen Fall wäre dies ein Kasten der Kantenlänge 2Z. Identifizieren wir **y** mit einem Zustand im Gehirn, so lässt sich dieses Modell mit dem Schlagwort *brain-state-in-a-box* ("Gehirnzustand im Kasten", s. Anderson, Silverstein, Ritz und Jones [AND77]) kennzeichnen . Der "Kasten" hat dabei im 3-dimensionalen Fall $8 = 2^3$ Ecken, im m-dimensionalen Fall 2^m Ecken (Warum?). Der Einschränkung des Zustandsraums, die ja nicht nur als Restriktion für die Eingabe **x**, sondern auch für die Ausgabe **y** aufzufassen ist, entspricht damit einer begrenzt-linearen Ausgabefunktion S(**y**).

Interessanterweise ist dies nicht unbedingt neu: Das Modell von Oja impliziert mit der Forderung normierter Gewichte $|\mathbf{w}|^2 = 1$ trotz linearer Ausgabefunktion $y = \mathbf{W}^T\mathbf{x}$ ebenfalls bei begrenzter Eingabe $|\mathbf{x}| \leq Z$ eine begrenzte Ausgabefunktion S(z):

$$\max(y) = \max(\mathbf{W}^T\mathbf{x}) = \max(\,|\mathbf{w}|\,|\mathbf{x}|\cos(\mathbf{w},\mathbf{x})\,) = +Z \tag{6.19}$$

und $\min(y) = \min(\,1\cdot|\mathbf{x}|\cos(\mathbf{w},\mathbf{x})\,) = -Z$

so dass die effektive Ausgabefunktion mit der begrenzt-linearen Ausgabe übereinstimmt.

Die Begrenzung der Ausgabe, d.h. die "Ecken" des Zustandsraums sind ziemlich wichtig für die Stabilität des Systems. Der Ausgabevektor wird Gl. (6.17) solange wachsen, bis er "stabil" ist, beispielsweise "in einer Ecke landet". Die Frage ist: in welcher?

Sei **x** ein Aktivitätsmuster. Dann gibt es unter allen Ecken $\mathbf{x}^j$ eine, die maximale Korrelation mit **x** hat:

$$\mathbf{x}\mathbf{x}^k = \max_j \mathbf{x}\mathbf{x}^j \quad \text{mit } \mathbf{x}^j = (x_1^j,...,x_n^j),\ x_i^j \text{ aus } \{ +L, -L \}$$

Diese Ecke $\mathbf{x}^k$ zeichnet sich dadurch aus, dass ihre Vorzeichen der Komponenten die gleichen sind wie bei den korrespondierenden Komponenten von **x**

$$\text{sgn}(x_i^k) = \text{sgn}(x_i) \text{ für alle } i, \text{ so dass } \mathbf{x}\mathbf{x}^k = \sum_{i=1}^{n} x_i L = L \sum_{i=1}^{n} |x_i|$$

Auch bei einem Wachstum $\lambda\mathbf{x} \rightarrow \mathbf{x}$ aller Komponenten von **x**, die nicht mit *L* begrenzt sind, bleibt diese Eigenschaft der Vorzeichen erhalten. Beim Wachstum der Komponenten wird eine nach der anderen an die Wachstumsgrenze stoßen und sich nicht weiter verändern, bis am Schluss alle Komponenten sowohl im Vorzeichen als auch im Wert mit denen der Ecke $\mathbf{x}^k$ übereinstimmen: Die Aktivität ist in die Ecke mit der größten Korrelation konvergiert.
Ist das System also schließlich in einer zu einem der *m* Eigenvektoren gehörenden Ecke gelandet, so bleibt der Systemzustand stabil; die Eigenvektoren stellen *Resonanzzustände* des Systems dar.

6.1.5 Anwendung: Kategorische Sprachwahrnehmung

Das Modell des *brain-state-in-a-box* wurde in der gleichen Arbeit [AND77] zur Erklärung der experimentalpsychologischen Beobachtung der "kategorischen Sprachwahrnehmung" eingesetzt. Hierbei handelt es sich um die Beobachtung, dass Versuchspersonen beim Anhören und Einordnen von synthetischen Sprachlauten in Lautkategorien (Klassifizieren) trotz langsamer, kontinuierlicher Veränderung der Laute, beispielsweise von stimmhaftem */b/* zu stimmlosem */p/* durch Verzögerung des Stimmeinsatzes, über weite Verzögerungsbereiche keine qualitative Veränderung (Änderung der Lautkategorie) wahrnehmen; dann aber, in einem sehr kleinen Übergangsbereich ("Schwellwert t_0") auf der Zeitachse, schlagartig einen Wechsel des Phonemtyps registrieren. In Abb. 6.3 ist ein solcher Umschlag an den Lauten */bah/* und */pah/* bei $t_0 = 30$ ms gezeigt.

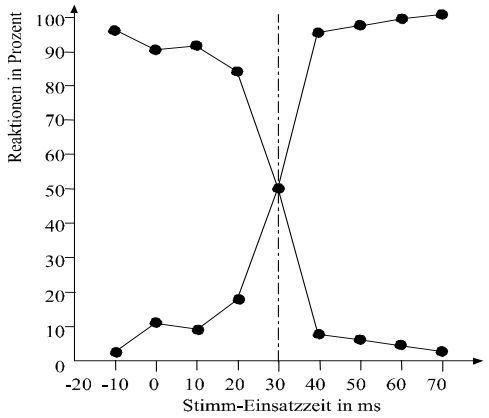


Abb. 6.3 Die Kategorisierung variierter Phonemlaute (nach [EIM85])

Es wird vermutet, dass in der Sprachverarbeitung zuerst ein vor-kategorischer Speicher der Sprachlaute und danach eine Kategorienbildung (Klassifikation) vorhanden ist. Bei Konsonanten fällt durch die kurze Dauer der Sprachlaute die Kategorisierung besonders klar auf, da die Klassifizierung über typische Frequenzen (*Formanten*) durch die unregelmäßigen Lautformen hier schwer möglich ist.

Bei der Simulation wählten sich die Autoren in [AND77] zwei 8-dim Vektoren **a** und **b** als Sprachmerkmale aus, die als Eigenvektoren zwei stabile Ecken in dem normierten System (Z=1) darstellen. Dann brachten sie das System in einen instabilen Zustand **x**, der auf einem Einheitskreis zwischen den beiden Vektoren lag, und ließen es konvergieren. Wurden dem initialen Zustand noch ein Gaußverteiltes Rauschen (Zufallsabweichung um den Nullpunkt) überlagert, so war das Konvergenzziel nicht mehr deterministisch vorgegeben. In Abb. 6.4 ist die initiale Konfiguration der beiden Kategorien A und B sowie die 16 Startpositionen dazwischen als 2-dim Schnitt des 8-dim Vektorraums gezeigt. Das Ergebnis der Simulation zeigt die prozentuale "Wahl" (Konvergenz) der Kategorie A bei jeweils 100 Iterationsläufe von einer gegebenen Startposition aus, wobei als Parameter die Standardabweichung σ der Zufallsabweichung in allen 8 Dimensionen gewählt wurde. Wie deutlich zu sehen ist, wird die Unterscheidung zwischen den zwei Kategorien erst bei großen Störungen ($\sigma = 0.4$) schwieriger.

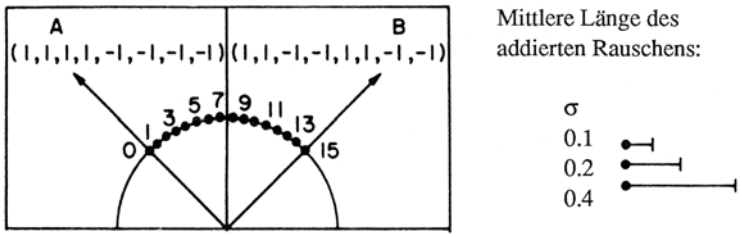


Abb. 6.4 Die Ausgangspositionen und Konvergenzziele der Iteration (nach [AND77])

In Abb. 6.5 ist das Ergebnis der Simulation gezeigt. Der Anteil von jeweils 100 Simulationsläufen, der zu Kategorie A konvergiert ist, ist in Abhängigkeit von der Startposition aufgetragen. Diese Daten ähneln sehr stark denen in Abb. 6.3, die Experimentalpsychologen und Sprachforscher gemessen haben.

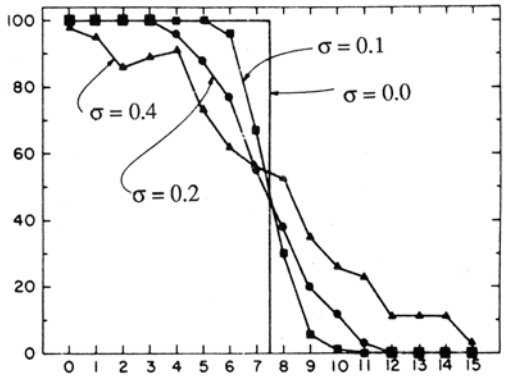


Abb. 6.5 Die Kategorisierung bei gestörter Eingabe (nach [AND77])

Wir können daraus schließen, dass schon ein einfaches, lineares rückgekoppeltes Netz durchaus ein ähnliches Verhalten wie die Sprachkategorisierung beim Menschen aufweisen kann.

6.1.6 Andere Modelle

Ein weiteres, klassisches Modell eines einfachen, rückgekoppelten Assoziativspeichers ist in den Arbeiten von Amari 1972 [AMA72] und 1977 [AMA77] beschrieben. Aus-

gehend von zeitdiskreten, rückgekoppelten Aktivität eines parallel arbeitenden, vollvernetzten Systems

$$\mathbf{x}(t+1) = \mathbf{S}_B(\mathbf{W}\mathbf{x}(t) - \mathbf{s}) \tag{6.20}$$

mit $S_B(z) = 1$ bei $z>0$, $S_B(z) = 0$ sonst

ergeben sich Gleichgewichtszustände ("*Kurzzeitgedächtnis*") nur für bestimmte Muster $\mathbf{x}^k$. Im Unterschied zu dem vorigen Modell, bei der die rückgekoppelte, lineare Signalverstärkung durch die Begrenzung ("Sättigung") des Signals x(t) stabilisiert wird, benutzt Amari in Gl. (6.20) direkt eine binäre, nichtlineare Ausgabefunktion S_B.

Für das Erlernen der Gewichte **W** ("*Langzeitgedächtnis*") gab Amari drei verschiedene Lernregeln an. Mit der Zeitkonstante α^{-1}, die ein Gewichtsbegrenzung und ein "Vergessen" durch das Abklingen der Gewichtsgröße bewirkt, lauten sie

$$w_{ij}(t+1) = (1-\alpha)\, w_{ij}(t) + \beta v_{ij}(t) \tag{6.21}$$

$$v_{ij}(t) = x_i(t)y_j(t) \quad \textit{(Hebb-Regel)} \tag{6.22}$$

$$v_{ij}(t) = x_i(t)L_j(t) \quad \textit{("erzwungenes" Lernen)} \tag{6.23}$$

$$v_{ij}(t) = x_i(t)z_j(t) \quad \textit{(Potential-Lernen)} \tag{6.24}$$

Der Lernprozess hat also zwei verschiedene Zeitmaßstäbe: Nach der Eingabe eines Musters **x** stabilisiert sich das System nach wenigen Rückkopplungen, wobei sich die Gewichte kaum verändern (Kurzzeitverhalten). Geben wir sehr viele Muster ein, so konvergieren allmählich die Gewichte zu bestimmten Werten (Langzeitverhalten).

Für jede Lernregel lässt sich eine Zielfunktion r(**W**,**x**,z) konstruieren, aus der man das Konvergenzziel für die Iteration Gl. (6.21) direkt ableiten kann. Die Änderung der Gewichte $\Delta\mathbf{W} = -\alpha\, \partial r/\partial \mathbf{W}$ wird dabei nach dem Gradientensuchalgorithmus für jeden einzelnen Gewichtsvektor $\mathbf{w}_i$ (*i*te Zeile von **W**) durchgeführt. Die Lernregeln (6.22), (6.23), (6.24) folgen dabei unterschiedlichen Anforderungen: In der ersten wird die Korrelation zwischen der Eingabe $\mathbf{x}_i$ und der erwünschten Ausgabe y_i, in der zweiten zwischen der Eingabe und einem Lehrer-Vorgabesignal L_i und in der dritten zwischen der Eingabe und der Neuronenaktivität z_i gelernt. Dabei lässt sich zeigen [AMA72], dass im dritten Fall der Gewichtsvektor **w** bei normierter Länge zum Eigenvektor der Autokorrelationsmatrix $\langle \mathbf{x}(t)\mathbf{x}^T(t)\rangle$ mit dem größten Eigenwert konvergiert. Dies ist in Übereinstimmung mit den Ergebnissen von Oja.

Wählen wir uns für die Lernregel (6.23) einen stabilen Zustand $\mathbf{x}^k$ durch $L_j := x_j^k$ als Lernziel (*Speichern* von $\mathbf{x}^k$), so hat diese auto-assoziative Speicherung bestimmte Eigenschaften. Durch die Überlagerung der Korellationen konvergieren die Gewichte zum Erwartungswert

$$w_{ij} = \beta/\alpha \, \langle x_i(t) x_j(t) \rangle_t \tag{6.25}$$

so dass selbst bei der Eingabe von gestörten Versionen der stabilen Muster ("verrauschte Signale") stabile Zustände entstehen [AMA72]. Die Zahl der speicherbaren Muster richtet sich dabei nach der Stärke der überlagerten Störung und der mittleren Korrelation der zu speichernden Muster untereinander und ist, wie auch beim Modell des *brain-state-in-a-box*, maximal *n* bei orthogonalen Mustern. Bei völlig unkorrelierten Mustern können immer alle Muster wieder ausgelesen werden (stabile Zustände), wobei mit wachsender Musterzahl das Einzugsgebiet Ω_k jedes Klassenprototypen $\mathbf{x}^k$ abnimmt.

6.1.7 Die Speicherkapazität

Die Informationskapazität eines rückgekoppelten Netzwerks ist bei *n* binären, orthogonalen Mustern nach [AMA72] leicht zu bestimmen: das erste Muster hat mit *n* Dimensionen auch *n Freiheitsgrade* und deshalb *n* Bits; das zweite hat allerdings neben den *n* Dimensionen noch die Nebenbedingung der Orthogonalität zum ersten Muster zu erfüllen, so dass nur noch *n–1* Freiheitsgrade und damit *n–1* Bits zur Verfügung stehen. Entsprechendes gilt auch für das dritte Muster mit zwei Nebenbedingungen, so dass die *n* Muster (Basisvektoren) insgesamt $n+(n-1)+(n-2)+..+1 = n(n+1)/2$ Bits darstellen. In der Speichermatrix gibt es bei *n* Dimensionen n^2 Gewichte, zusammen mit den *n* dazu gehörenden Schwellwerten also $(n+1)n$ Gewichte. Also beträgt die Informationskapazität des Netzwerks pro Gewicht

$$H = \frac{n(n+1)/2}{(n+1)n} = 0{,}5 \left[\frac{\text{Bit}}{\text{Gewicht}} \right] \tag{6.26}$$

Die Gewichte sind durch die symmetrische Korrelation der Hebb-Lernregel mit $w_{ij} = w_{ji}$ zur Hälfte gleich. Also ist

H = 1 Bit pro unabhängiges Gewichtspaar

6.1.8 Eine Simulation

Eine Veranschaulichung der autoassoziativen Eigenschaften zeigte die Simulationen von Willwacher [WIL76]. In ein rückgekoppeltes Netzwerk nach Abb. 6.2, das als Ausgabefunktion für $z > 0$ eine sigmoide Funktion (Sättigungskurve) der Form

$$S(z) = \begin{cases} a\left(1 - e^{-z/b}\right) & z > 0 \\ 0 & \text{sonst} \end{cases} \qquad a, b > 0 \tag{6.27}$$

enthielt, speicherte er verschiedene Muster ein. Ein Muster **x** wurde dabei als die Aneinanderreihung der Zeilen einer 10×10 Matrix betrachtet, in der verschiedene Buchstaben eingespeichert wurden, siehe Abb. 6.6. Um ein "Überlaufen" der Gewichte zu verhindern, wurde nicht wie in (6.18)die Eingabe begrenzt, sondern die Gewichte selbst mit der Funktion (6.27) nicht-linear bewertet. Die Schwellwerte der n = 100 Neuronen wurden durch die Gesamtaktivität geregelt.

In der Abb. 6.6 ist die Musterergänzung im System gezeigt, nachdem alle Buchstaben eingespeichert worden sind. Die eingespeicherten Muster sind oben gezeigt; die zeitliche Stabilisierung des rückgekoppelten Systems ist bei vier verschiedenen Eingaben ist darunter gezeigt. Bei der Eingabe eines unvollständigen oder anders von dem Gespeicherten abweichenden Musters "wählt" das System dasjenige gespeicherte Muster $\mathbf{x}^r$ aus, das die größte Korrelation $\mathbf{x}^T\mathbf{x}^r$ mit der Eingabe **x** aufweist.

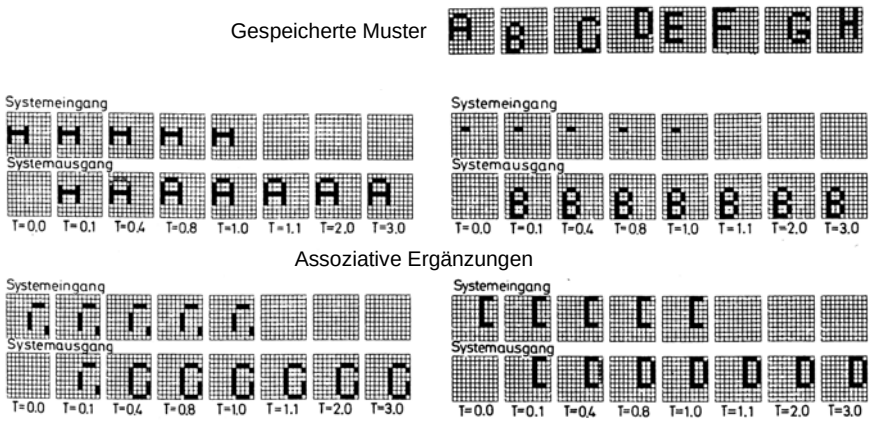


Abb. 6.6 Musterergänzung im autoassoziativen Speicher (aus [WIL76])

6.2 Das Hopfield-Modell

Einer der wichtigen Ereignisse für die Entwicklung von Gebieten der neuronalen Netze war die Tatsache, dass verstärkt theoretische Physiker Interesse an diesem Gebiet fanden und ihr Handwerkszeug, ein Spektrum physikalisch-mathematischer Methoden und Modelle, auf die neuen Probleme anwendeten. Ausgelöst wurde dieses Interesse nicht zuletzt 1982 durch einen Artikel von John Hopfield [HOP82], in dem er eine Verbindung zwischen den magnetischen Anomalien in seltenen Erden (*Spingläsern*) und Netzen aus rückgekoppelten, binären Neuronen aufzeigte und die Zielfunktion, die für das Verhalten des Systems ausschlaggebend ist, als *Energie* interpretierte. Mit dem *Hopfield-Modell* war zwar prinzipiell nichts Neues gefunden worden, aber die Zugangsschwelle zu dem Gebiet der neuronalen Netze war für die Physiker entscheidend gesenkt worden. Da inzwischen eine Flut von Veröffentlichungen über dieses Modell und mögliche Varianten existiert, seien an dieser Stelle nur einige der wichtigsten Grundzüge erläutert. Für ausführlichere detaillierte Rechnungen und Hinweise sei auf die Lehrbücher von Amit [AMIT89] und von Müller und Reinhardt [MÜ90] verwiesen.

Viele physikalischen Modelle für Spingläser gehen von der Vorstellung von Atomen aus, die sich gegenseitig beeinflussen. Die Atome wirken dabei wie kleine Magnete (*Spins*), die in einem äußeren, konstanten Magnetfeld entweder parallel oder antiparallel orientiert sein können. Den Zustand der Dipolmagnete kann man auch mit "Spin auf" und "Spin ab" oder den Zahlen +1 und –1 bezeichnen. Bezeichnen wir die Kopplungskoeffizienten zwischen dem Atom *i* und dem Atom *j* mit w_{ij}, so ist das lokale Magnetfeld z_i beim Atom *i* eine Überlagerung aus den Zuständen y_j aller anderen Atome und der Wirkung eines äußeren Feldes s:

$$z_i = \sum_j w_{ij}\, y_j - s_i \quad \text{für } i \neq j \tag{6.28}$$

Sind alle $w_{ij} > 0$, so ist das Material *ferromagnetisch;* wechselt das Vorzeichen von w_{ij} periodisch im Kristallgitter, so ist das Material *antiferromagnetisch;* bei einer zufälligen Verteilung des Vorzeichens spricht man von *Spingläsern.*

Dieses Modell hat das gleiche Verhalten wie ein System aus *n* binären Neuronen mit einer zentrierten Eingabe und Ausgabe

$$y_i,\ x_i \in \{-1,+1\} \Rightarrow |\mathbf{x}|^2 = n$$

$$y_i(t) = S(z_i(t-1)) := \operatorname{sgn}(z_i(t-1)) = \begin{cases} +1 & z_i \geq 0 \\ -1 & z_i < 0 \end{cases} \tag{6.29}$$

wobei der Übergang von der manchmal ebenfalls verwendeten Darstellung $x_i \in \{0,1\}$ zu der Darstellung $x_i \in \{-1,+1\}$ durch die Umrechnung $x_i \rightarrow (2x_i - 1)$ erreicht werden kann.

Dabei wird der Wert $S_i := y_i$ der Ausgabe $S_i(z_i)$ eines Neurons als sein *Zustand* betrachtet, so dass der Vektor $\mathbf{S} = (S_1,...,S_n)$ als *Systemzustand* interpretiert werden kann. Der Algorithmus des Modells lässt sich mit Hilfe der diskreten Verbindungen folgendermaßen formulieren:

```
HOPF1 :(* sequentielle Aktivierung rückgekoppelter Neuronen *)
        Wähle w_ij gemäß dem Problem      (* Problemkodierung *)
        Setze den Startzustand S(0); t:=0;
      REPEAT
          t  := t+1;
          i  := RandomInt(1,n)        (* wähle zufällig ein Neuron, i aus 1..n *)
          S_old:=S_i                  (* merke alten Zustand *)
          z_i  := z(wi,S )            (* Bilde die Aktivität *)
          S_i := S(z_i)               (* Bilde die Ausgabefunktion  *)
          MarkNeuron(i)               (* setze die "alter Zustand" Markierung *)
          IF Si#S_old THEN ResetAllMarks END
      UNTIL AllNeuronsMarked   (* Abbruchkriterium: stabiler Zustand *)
      Gebe Endzustand S(t) aus
```

Codebeispiel 6.1 *Der sequentielle Algorithmus für das Hopfield-Modell*

Dabei wird in diesem Codebeispiel für das Abbruchkriterium ein Mechanismus benutzt, der alle Neuronen mit gleich bleibenden Zuständen markiert. Sind alle Neuronen markiert, so kann sich im Netz nichts mehr ändern; das System ist in einem stabilen Zustand angelangt. Natürlich sind auch andere Abbruchkriterien denkbar; beispielsweise eine ausreichend gute Lösung oder ähnliches.

Für das zeitliche Systemverhalten betrachtet Hopfield zu einem Zeitpunkt nur eine Zustandsänderung an einem einzigen Neuron. Diese sequentiellen Zustandsübergänge sind leichter auf einem sequentiellen Rechner zu simulieren, verändern aber das Modell gegenüber einem echt parallelen, synchronen Modell [LIT78].
Die Gewichte sind bei symmetrischen Kopplungen ebenfalls symmetrisch:

$$w_{ij} = w_{ji} \tag{6.30}$$

Die Zielfunktion (*Energie*) lässt sich durch die Forderung definieren, dass ein Aktivitätszuwachs eines Neurons *i* und damit eine Zustandsänderung nur mit einer Energieminderung verbunden sein soll:

$$z_i \sim -\partial E(S_i)/\partial S_i \tag{6.31}$$

somit $E(S_i) := -z_i S_i$ (6.32)

Addieren wir alle Wechselwirkungsenergien der einzelnen Neuronen, so müssen wir darauf achten, die Wechselwirkung zwischen Neuron *i* und *j* nur einmal zu zählen. Dies bedeutet mit Gl.(6.28)

$$E(t) = \sum_i E(S_i) = -\sum_i \sum_{j>i} w_{ij} S_i(t) S_j(t) + \sum_i s_i S_i(t) \tag{6.33}$$

Die Energie für die parallele Version von Little und Shaw ist interessanterweise fast identisch mit Gl. (6.33) bis auf den Faktor $S_j(t)$, der hier $S_j(t–1)$ lauten muss. Allerdings kann man damit E(t) nicht mehr als physikalische Energie zum Zeitpunkt *t* interpretieren, sondern nur noch als eine Art Stabilitätsfunktion (sog. *Ljapunov-Funktion*) des Systems.
Da die Energie in Gl. (6.33) zwischen *i* und *j* in der Doppelsumme zweimal gezählt wird, muss man entweder mit der Nebenbedingung j > i summieren oder den Faktor ½ vorsehen, was auch öfters benutzt wird, aber natürlich äquivalent zu (6.33) ist:

$$E(t) = -½ \sum_i \sum_j w_{ij} S_i(t) S_j(t) + \sum_i s_i S_i(t) \tag{6.34}$$

In Gl. (6.33) ist der Fall der Wechselwirkung w_{ii} mit sich selber bei i = j mit der Nebenbedingung j > i ausgeschlossen. Da aber der Term $S_i(t)S_j(t)$ für i = j konstant eins ist, trägt dieser Term in der Doppelsumme von (6.34) nichts zur Dynamik des Systems bei und kann als Konstante zur Bequemlichkeit einbezogen werden, so dass die Randbedingung $i \neq j$ entfallen kann, ohne dass man dazu $w_{ii} = 0$ definieren müsste.

Betrachten wir die Schwelle s_i ebenfalls als Gewicht eines konstanten Neurons, so lässt sich Gl. (6.34) mit $s_i = 0$ als

$$E(t) = -½ \sum_i \sum_j w_{ij} S_i(t) S_j(t) =: \sum_i E_i(t) \tag{6.35}$$

schreiben.

Angenommen, alle Gewichte sind vorgegeben und man überlässt das System sich selbst: Wie entwickelt sich der Systemzustand mit der Zeit? Betrachten wir dazu die Änderung der oben definierten Energie, wenn sich beim Zeitschritt t+1 der Zustand eines Neurons *i* ändert. In magnetischen Systemen entspricht dies der "Glauber Dynamik" [GLA63] für die sequentielle Änderung der Spins.
Mit Gl. (6.35) ist für das Neuron *i*

$$E_i(t) = -z_i(t) S_i(t) \tag{6.36}$$

Angenommen, die Aktivität $z_i(t)$ aus dem Netz hatte sich derart geändert, dass sich auch der Zustand $S_i(t) \rightarrow S_i(t+1)$ ändert. Damit ändert sich auch die Energie des Neurons mit Gl. (6.29) zu

$$E_i'(t+1) = -z_i(t) S_i(t+1) = -z_i(t) S_i(z_i(t)) = -z_i(t)\, \mathrm{sgn}(z_i(t)) = -|z_i(t)| \leq 0 \tag{6.37}$$

Da sich S_i geändert hatte, haben $z_i(t)$ und $S_i(t)$ unterschiedliche Vorzeichen, so dass ihr Produkt negativ ist und damit

$$E_i'(t+1) \leq 0 \leq -z_i(t)S_i(t) = E_i(t) \tag{6.38}$$

gilt. Da Neuron *i* nicht mit sich selbst verbunden ist, tritt in diesem Zeitschritt keine Rückkopplung von S_i auf z_i auf, so dass $z_i(t+1)=z_i(t)$.

$$E_i'(t+1) = E_i(t+1) \leq E_i(t) \tag{6.39}$$

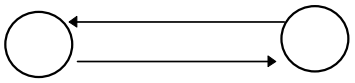
Die Energie für ein Neuron wird also bei jedem Zeitschritt und jeder Zustandsänderung dieses Algorithmus nur kleiner oder bleibt gleich, so dass die *Ljapunov-Bedingung* der stetigen Verminderung der Zielfunktion aus Kapitel 2 erfüllt ist. Man beachte, dass eine Aktivitätsänderung durch den geänderten Zustand von Neuron *i* bei einem Neuron *k* die gleiche obige Argumentation nach sich zieht, so dass sich sequentiell auch dann die Energie vermindert.

Da $\Delta E_i = E_i(t+1) - E_i(t) = -z_i(S_i(t+1) - S_i(t)) = -z_i \Delta S_i$ ist bei konstanter Energie ($\Delta E_i = 0$) der Zustand konstant ($\Delta S_i = 0$) und umgekehrt. Dies gilt nur für den sequentiellen Algorithmus und bedeutet nicht notwendigerweise, dass es nicht zwei verschiedene Zustände mit gleicher globaler Energie E(**S**) geben kann: Beispielsweise ist die negierte Form –**S** eines Zustands **S** unterschiedlich zu **S**; seine Energie ist aber durch die Beziehung $S_iS_j = -S_i-S_j$ gleich. Da der Algorithmus sequentiell funktioniert, kann der Übergang von **S** auf _**S** und zurück, wie bewiesen, nicht stattfinden. Es existiert somit ein Zustand minimaler Energie, der stabil ist. Es lässt sich zeigen [HOP84], dass dies hauptsächlich auf die Symmetriebedingung $w_{ij} = w_{ji}$ zurückzuführen ist.

Im Unterschied dazu lässt sich zeigen (Übungsaufgabe!), dass die Energie des parallelen Systems auch konstant bleibt ($\Delta E = 0$), wenn $S_i(t+1) - S_i(t-1) = 0$ ist, so dass das System zwischen zwei Zuständen hin und her oszillieren kann. Bezieht man noch die Beeinflussung einer Synapse durch mehrere, andere Neuronen (z.B. Synapsen höherer Ordnung) ein, so kann das System zwischen vielen Zuständen oszillieren.

Beispiel 6.2.1

Betrachten wir ein Hopfieldnetz aus n = 2 Einheiten.



$\mathbf{S} = (S_1, S_2)$
$S_1 = \text{sign}(wS_2)$
$S_2 = \text{sign}(wS_1)$
$w>0$

Abb. 6.7 *Ein Hopfieldnetz mit n = 2 Einheiten*

Dann kann der Systemzustand $\mathbf{S} = (S_1,S_2)$ bei symmetrischen Gewichten $w_{12} = w_{21} =: w$ nur dann stabil bleiben, wenn $\mathbf{S} = (+1,+1)$ oder $\mathbf{S} = (-1,-1)$ ist. Nehmen wir nämlich an, dass $\mathbf{S}(t) = (-1,+1)$ sei, so ergibt sich im nächsten Schritt bei der Auswahl eines Neurons (sequentielle Dynamik)

$$z_1(t) > 0 \quad \text{und damit } \mathbf{S}(t+1) = (+1,+1) \tag{6.40}$$

oder $z_2(t) < 0$ und damit $\mathbf{S}(t+1) = (-1,-1)$.

Bei $\mathbf{S}(t) = (+1,-1)$ gilt dies natürlich analog ebenso. Ist dagegen $\mathbf{S} = (+1,+1)$ oder $\mathbf{S} = (-1,-1)$, so reproduzieren sich die Zustände immer wieder, da mit Gl. (6.28) folgt

$$\begin{aligned} &(S_1(t)>0) \Rightarrow (z_2(t)>0) \Rightarrow (S_2(t+1)>0), \\ &(S_2(t)>0) \Rightarrow (z_1(t)>0) \Rightarrow (S_1(t+1)>0) \end{aligned} \tag{6.41}$$

und
$$\begin{aligned} &(S_1(t)<0) \Rightarrow (z_2(t)<0) \Rightarrow (S_2(t+1)<0), \\ &(S_2(t)<0) \Rightarrow (z_1(t)<0) \Rightarrow (S_1(t+1)<0) \end{aligned}$$

Bei der parallelen Dynamik wechseln beide Neuronen gleichzeitig das Vorzeichen, so dass auch ein Pendeln zwischen $\mathbf{S}(t) = (+1,-1)$ und $\mathbf{S}(t) = (-1,+1)$ stabil sein kann ("Oszillation").

Der Algorithmus für das Hopfield-Modell lässt sich, anstatt mit einer diskreten Dynamik eines Netzes (Codebeispiel 6.1), auch nur mit Hilfe der Systemenergie im Codebeispiel 6.2 formulieren.

```
HOPF2 :   (* Sequentieller, Energie-gesteuerter Algorithmus *)
          Wähle w_ij gemäß dem Problem     (* Problemkodierung *)
          Setze den Startzustand S(0); t:=0;
          Bilde E                          (* nach Gleichung (6.35)*)
          REPEAT
            t:= t+1
            i:= RandomInt(1,n )            (* wähle zufällig ein Neuron i aus 1..n *)
            ΔE:= E_i(–S_i) – E_i(S_i)      (* Energiediff. bei Zustandsänderung *)
            IF  ΔE ≤ 0  THEN  E := E + ΔE;  S_i:= –S_i   (* ändere Zustand *)
          UNTIL E ≤ E_min                  (* Abbruchkriterium *)
          Gebe Endzustand S(t) aus
```

Codebeispiel 6.2 *Energie-Algorithmus des Hopfieldmodells*

Er sieht nach einer initialen Wahl der Gewichte, die durch das betrachtete Problem gegeben sind, solange eine sequentielle Zufallsauswahl und Zustandsverbesserung eines Neurons vor, bis die Energie unter eine vorgegebene Schwelle sinkt. Ein diskretes Netzwerk wird für diesen Algorithmus nicht mehr explizit benötigt; implizit ist es aber in der Energiedefinition enthalten. Ist die minimale Energie E_{min} nicht bekannt, so reicht es auch, solange zu iterieren, bis für alle Neuronen $\Delta E_i = 0$ ist. Das Konvergenzverhalten eines solchen Systems ist in Abb. 6.8 visualisiert.

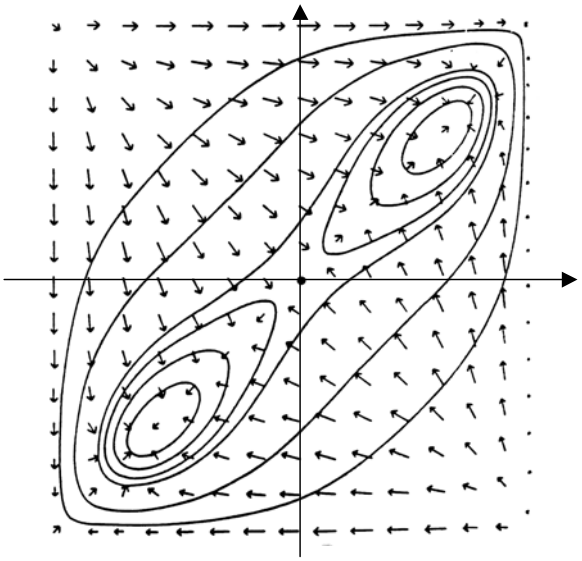


Abb. 6.8 *Das Attraktorverhalten stabiler Fixpunkte (aus [HOP84])*

Für den Fall von zwei rückgekoppelten Neuronen bei paralleler Aktualisierung sind die 2-dim. stabilen und labilen Fixpunkte $\{(S_1,S_2)^*\}$ eingetragen. Da dies bei der binären signum-Ausgabefunktion nur vier Punkte wären (s. Beispiel 6.2.1), wurde zur besseren Sichtbarkeit der Dynamik die sigmoidale Ausgabefunktion S_T gewählt, die auch reellwertige Zwischenwerte aufweist. Ausgehend von einer großen Menge von Zuständen $\mathbf{S}(t)$, die im regelmäßigen Raster als Punkte in einem Koordinatensystem in den Intervallen [–1,+1], [–1,+1] aufgetragen sind, berechnete man den Zustand im Zeitpunkt t+1 und trug den Punkt $\mathbf{S}(t+1)$ ebenfalls in das Koordinatensystem ein. Für jede Zustandsänderung $\mathbf{S}(t) \rightarrow \mathbf{S}(t+1)$ wurden nun beide Zustandspunkte mit einem Pfeil verbunden.
Zusätzlich wurden Punkte gleicher Energie miteinander durch eine Linie verbunden und bilden so die "Höhenlinien" des "Energiegebirges". Man beachte, dass die Änderungen des Systems nicht unbedingt senkrecht zu diesen Höhenlinien sind: Die Schritt-

weiten, und damit die Konvergenz, sind nicht optimal. Da es aussieht, als ob die Fixpunkte den Systemzustand "anziehen" würden, werden diese Art von Netzen auch als *Attraktornetze* bezeichnet. Man beachte, dass hier die Fixpunkte durch die sigmoidalen statt binären Ausgabefunktionen nicht genau in den Eckpunkten liegen.
Man beachte auch, dass der Nullpunkt zwar ein Fixpunkt, aber nicht stabil ist; jede kleine Abweichung führt sofort zu einem der beiden stabilen Fixpunkte.

6.2.1 Stabile Zustände und Energiefunktionen

Da jede Einheit wiederum von vielen anderen Einheiten und damit auch wieder von sich selbst abhängig ist, lässt sich bei der daraus resultierenden, komplexen Dynamik über das Gesamtsystem durch Einzelbeobachtungen (Beispiele) meist relativ wenig aussagen; eine analytische Lösung der gekoppelten Differenzialgleichungssysteme ist meist aussichtslos oder sehr schwierig. Stattdessen hat es sich als sinnvoll erwiesen, Systeminvariante zu betrachten, die sich einheitlich auf das Verhalten der Einzelelemente auswirken. Eines der wichtigsten Größen ist dabei die *Energie E* des Systems. Ähnlich wie in physikalischen Systemen ist der stabile Zustand *S*, den das System nach einer gewissen Konvergenzzeit (hoffentlich) erreicht, durch ein Minimum an Gesamtenergie E(t) gekennzeichnet, sofern die Zustandsänderung jeder Einheit nur im Sinne einer Verminderung der lokalen Einzelenergie erfolgen darf. Hat die Energie die Eigenschaft einer Ljapunov-Funktion bezüglich der Zeit, so garantiert der daraus abgeleitete Änderungsalgorithmus für die Aktivität eines Neurons einen stabilen Zustand des Gesamtsystems. Andererseits bedeutet ein daraus abgeleiteter Änderungsalgorithmus für die Gewichte $\mathbf{w}$ auch eine Konvergenz beim Lernen. Bei den rückgekoppelten Netzwerken sind also Systemstabilität und Lernstabilität eng miteinander verbunden.

Allerdings muss man dabei beachten, dass die Gradientensuche nur die Stabilität für den einen Parameter $\mathbf{w}$ garantiert, der geändert wird. Dies entspricht bei mehreren Parametern $\mathbf{w}_i$ (mehreren Neuronen) einer sequentiellen Verbesserung der Parameter. Sollen die Parameter dagegen parallel (gleichzeitig) geändert werden, so bedeutet eine lokale Verbesserung bei $\mathbf{w}_i$ nicht auch automatisch eine globale Verbesserung, d.h. ein Absenken, von E. Wie beim Hopfieldmodell gezeigt werden wird, kann auch beispielsweise ein Oszillieren resultieren.

Natürlich handelt es sich bei formalen Neuronen nicht um Atome und damit auch nicht um echte Energie; der Name "Energiefunktion" ist nur eine andere Bezeichnung für eine *Zielfunktion*, wie wir sie bereits bei der Mustererkennung kennen gelernt haben. Da die Forschung über rückgekoppelte Modelle stark von Physikern betrieben wurde, behalten wir in diesem Abschnitt die Bezeichnung "Energie" für eine Zielfunktion zur Veranschaulichung bei. Wesentlich für die Funktion dieser Modelle ist damit nicht nur ihre Vernetzungsart und die Lernregeln für ihre Gewichte, sondern auch die Definition ihrer Energiefunktion.

Die Hauptarbeit, um solche Systeme zum Lösen einer bestimmten Aufgabe zu programmieren, besteht damit nicht so sehr im Entwurf einer Netzwerktopologie und im Zusammenstellen einer Menge von Trainingsmustern, sondern insbesondere in der Konstruktion einer Energiefunktion, die das gewünschte Resultat bewirkt.

6.2.2 Anwendung: Das Problem des Handlungsreisenden

Im Unterschied zu den feedforward-Netzen entspricht die "Programmierung" des Netzwerks nicht etwa einer Auswahl einer Lernregel und der Aufstellung von Trainingsmustern, sondern der initialen Festlegung der Gewichte im Netzwerk sowie des Systemzustands $\mathbf{S}(t)$ zum Zeitpunkt $t = 0$.
Wie werden nun die Gewichte festgelegt ? Dies ist sicher vom Verwendungszweck des Netzwerks bestimmt. Betrachten wir dazu das uns bekannte klassische Beispiel des Handlungsreisenden, das für eine Vielzahl ähnlicher Probleme steht.

Das Hopfield-Modell betrachtet die Wechselwirkung vieler, miteinander gekoppelter Einheiten und versucht, durch die Definition einer Zielfunktion ("Energie") die Konvergenz der Zustandsfolge vom initialen Zustand zu einem stabilen Zustand zu zeigen. Dieser stabile Zustand entspricht einem lokalen Minimum der Zielfunktion. Durch die binären Ausgabefunktionen und die Nebenbedingungen eines Problems, die sich in der Definition der Gewichte widerspiegeln, existieren meist mehrere lokale Minima, die nicht mit dem globalen Minimum übereinstimmen müssen.

Das Hopfield-Modell lässt sich also besonders gut bei derartigen Probleme anwenden, bei denen viele einzelne, unabhängige Parameter existieren und sich miteinander koordinieren müssen, um gemeinsam eine optimale Lösung zu finden. Abgesehen von der Rechenzeit, die nötig ist, um auf traditionellen Rechnern die Wechselwirkung sehr vieler Einheiten zu simulieren, besteht das wesentliche Problem bei diesem Ansatz, das globale bzw. ein möglichst gutes suboptimales, lokales Minimum zu finden.

Betrachten wir dazu als Beispiel das Problem des Handlungsreisenden. Bei der Benutzung des Hopfield-Modells sind verschiedene Schritte nötig, um das vorhandene Problem auf den Mechanismus des Modells abzubilden. Dazu wählen wir uns zuerst die Kodierung der N Städte, die vom Handlungsreisenden auf einer Reise hintereinander besucht werden sollen, auf die Art und Weise, wie sie von Hopfield und Tank [HOP85] vorgeschlagen wurden. Hierbei wird eine Matrix aufgestellt, die als Spalten die Städte und als Zeilen die Positionsnummer der Stadt innerhalb der Reise enthalten. In der Abb. 6.9 ist dies verdeutlicht.

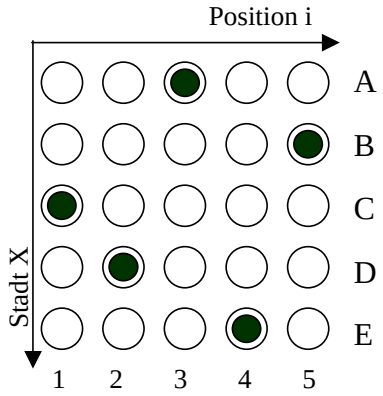


Abb. 6.9 *Kodierung der Reise C→D→A→E→B→C*

Alle Zeilen der Matrix werden, hintereinander gehängt, als Zustandsvektor **S** des Systems aufgefasst; ein dunkler Punkt in der Matrix bei Zeile X und Spalte i bedeutet, dass auf der Rundreise Stadt X als *i*-te Station (Position *i*) zu besuchen sei und wird im Vektor mit "1" notiert, alle leeren Kreise mit "0". Wir benutzen hier also die binäre Notation $S_i \in \{0,1\}$ anstelle der symmetrisch-dualen Spin-Notation $S_i \in \{-1,+1\}$.

Alle $n = N^2$ Komponenten des Zustandes **S** können wir nun im Sinne der Matrixnotation doppelt indizieren, um sowohl die Reihe (Stadt) als auch die Spalte (Position) festzuhalten. Mit S_{xi} ist also der Zustand (das Matrixelement) festgelegt, bei dem die Stadt X an *i*-ter Position in der Rundreise steht und mit $w_{xi,yj}$ das Gewicht zwischen S_{xi} und S_{yj}. Die Reise auf dem Weg zwischen den Städten X und Y ist mit der Bedingung

$$S_{x,i}\, S_{y,i+1} = 1 \text{ oder } S_{x,i}\, S_{y,i-1} = 1 \tag{6.42}$$

charakterisiert, wobei die Position bei *N* Städten Modulo *N* gerechnet wird. Die Gesamtwegstrecke ist bei bekanntem Abstand d(X,Y) zwischen den Städten X und Y

$$E_d(S) := 1/2 \sum_x \sum_{y \neq x} \sum_i d(X,Y)\, S_{x,i}\, (S_{y,i+1} + S_{y,i-1}) \tag{6.43}$$

wobei die Summierung alle möglichen Städte und Positionen berücksichtigt. Da dabei alle Wege in beiden Richtungen gezählt werden, müssen wir mit d(X,Y) = d(Y,X) ein Faktor 1/2 davor setzen.

Die obige Formel gibt zwar die Länge einer Rundreise, unterscheidet allerdings dabei nicht, ob die Matrix bzw. der Zustand **S** tatsächlich auch eine Rundreise darstellt. Um ungültige Reisen (unerwünschte Matrizen) auszuschließen, definieren wir uns noch die Nebenbedingungen

- Keine Stadt darf mehrfach vorkommen (nur eine 1 in jeder Zeile)
- An jeder Position darf *max. eine* Stadt stehen (max. eine 1 in jeder Spalte)
- An jeder Position muss *mind. eine* Stadt stehen (mind. eine 1 in jeder Spalte)

Mit diesen Nebenbedingungen reduzieren wir die Zahl $M = 2^{N2}$ aller durch eine N×N Matrix beschreibbaren Reisen auf die Zahl $M = (N-1)!/2$ der tatsächlich sinnvollen Reisen. Die Nebenbedingungen lassen sich durch Terme einer Energiefunktion ausdrücken, die nur bei Erfüllung der Nebenbedingung null wird

$$\begin{aligned} E(\mathbf{S}) := \quad & A/2 \sum_x \sum_{i \neq j} \sum_i S_{x,i} S_{x,j} && \textit{keine Stadt mehrfach} \\ & + B/2 \sum_x \sum_{y \neq x} \sum_i S_{x,i} S_{y,i} && \textit{nicht mehr als eine Stadt pro Position} \\ & + C/2 \left(N - \sum_x \sum_i S_{x,i}\right)^2 && \textit{mind. eine Stadt pro Position} \\ & + D\, E_d(\mathbf{S}) && \textit{minimaler Reiseweg} \end{aligned} \tag{6.44}$$

mit dem jeweiligen Term-Faktor "1/2" wegen der Doppelzählung und den noch zu bestimmenden Parametern A,B,C,D > 0. Die so definierte Energiefunktion steht nicht etwa im Gegensatz zu der bereits definierten allgemeinen Energiefunktion Gl. (6.34) des Hopfieldmodells, sondern muss mit ihr identisch sein. Um diese Identität auch tatsächlich herzustellen, errechnen wir uns nun die dafür benötigten Gewichte w_{ij}. Die allgemeine Energiefunktion Gl. (6.34) modifiziert sich mit der obigen Doppelindex-Notation zu

$$E(t) = -1/2 \sum_u \sum_v \sum_i \sum_j w_{ui,vj}\, S_{ui}(t) S_{vj}(t) + \sum_u \sum_i s_{ui} S_{ui}(t) \tag{6.45}$$

Mit der Definition

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{sonst} \end{cases} \qquad \textit{Kronecker-Delta} \tag{6.46}$$

ergeben sich dann die Gewichte $w_{ui,vj} = w_{ui,vj}^{(1)} + w_{ui,vj}^{(2)} + w_{ui,vj}^{(3)} + w_{ui,vj}^{(4)}$ für den

ersten Term: Für u = v und i ≠ j ist der Term A, also

$$w_{ui,vj}^{(1)} = -A\, \delta_{uv} (1 - \delta_{ij}) \tag{6.47}$$

zweiten Term: Für u ≠ v und i = j ist der Term B, also

$$w_{ui,vj}^{(2)} = -B\, \delta_{ij} (1 - \delta_{uv})$$

dritten Term: Für alle anderen Terme einer nicht-sinnvollen Reise soll

$$w_{ui,vj}^{(3)} := -C \quad \text{gelten.}$$

Dabei lässt sich die Energie auch umformen in

$$E = C/2 \left(\sum_u \sum_i S_{ui}\right)\left(\sum_u \sum_i S_{ui}\right) + s \left(\sum_u \sum_i S_{ui}\right)$$

so dass mit der passenden Definition s:= –CN folgt

$$E = C/2\,[(\sum_u\sum_i S_{ui})(\sum_u\sum_i S_{ui}) - 2N\,(\sum_u\sum_i S_{ui})]$$

was bis auf einen konstanten, fehlenden Term $C/2\,N^2$ identisch mit der Vorgabe (6.34) ist.

vierten Term: Für $u \neq v$, $i = j+1$ und $i = j-1$ ist der Term gleich $-D\,d(u,v)$, also

$$w_{ui,vj}^{(4)} := -D\,(1-\delta_{uv})(\delta_{i,j+1}+\delta_{i,j-1})d(u,v)$$

Der im dritten Term fehlende Ausdruck $C/2\,N^2$ verändert das dynamische Verhalten des Systems nicht, da er nur die Energie um einen festen, vom Systemzustand unabhängigen Betrag verschiebt; wir können also bei der Gewichtsbestimmung ohne Probleme auf ihn verzichten.

Die Wahl der Parameter A,B,C und D ist durchaus nicht beliebig. Beispielsweise beruhten die Schwierigkeiten von Wilson und Pawley [WIL88], die nachfolgend beschriebene Simulation von Hopfield und Tank nachzuvollziehen, im wesentlichen auf diesem Problem. Bedingungen für diese Parameter lassen sich beispielsweise aus der Tatsache herleiten, dass die Energiedifferenz größer null für benachbarte Zustände einer vernünftigen Reise ist [LAN88], [KAM90]

$$C > 2D \max_{\substack{u,v,w \\ u\neq v}} (d(u,v)+d(u,w)) \text{ und daraus } A > C/2, \tag{6.48}$$

In einer analogen Simulation mit sigmoidalen statt binären Ausgabefunktionen zeigten Hopfield und Tank [HOP85], dass der Algorithmus tatsächlich das Gewünschte leistet.

In Abb. 6.10a ist die Verteilung der Längen von zufällig generierten, "vernünftigen"

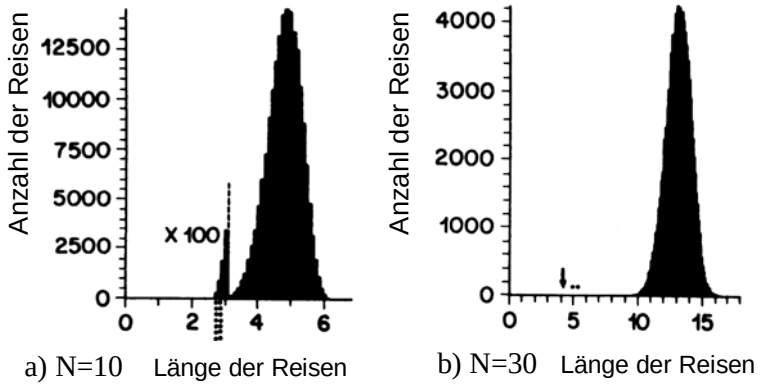


Abb. 6.10 *Verteilung der Längen von Zufallsreisen (nach [HOP85])*

Rundreisen (Startwerte) von N = 10 Städten sowie die Ergebnisse der Simulationen als Histogramm zu sehen. Der vergrößerte Ausschnitt (*x100*) zeigt eine gute Konvergenz des Algorithmus bei den kleinsten Reiselängen (Häufigkeitspunkte unterhalb der Achse).

In Abb. 6.10b ist die statistische Verteilung der Reiselängen bei N = 30 gezeichnet, wobei hier die minimale Reiselänge durch einen Pfeil angezeigt ist. Obwohl statistisch die wenigsten (ca. 10^8) der $4.4 \cdot 10^{30}$ möglichen Rundreisen kürzer sind als 7, konvergiert das System regelmäßig in diesen Bereich. Man beachte, dass bei größerer Städtezahl nur weniger Simulationen möglich sind.

Die Reiserouten selbst sind in Abb. 6.11 in auf eins normierten Koordinaten visualisiert. Wird die Nichtlinearität ("Verstärkung") der sigmoidalen Ausgabefunktionen nach einem zufälligen Anfangszustand (Abb. 6.11a) nur langsam erhöht, so bleibt das System im Konvergenzbereich eines suboptimalen Fixpunkts [KAM90] und konvergiert dahin (Abb. 6.11b)). Die bestmögliche Reise ist mit dem deterministischen Lin-Kernighan Algorithmus [LIN73] errechnet und in Abb. 6.11c) eingetragen.

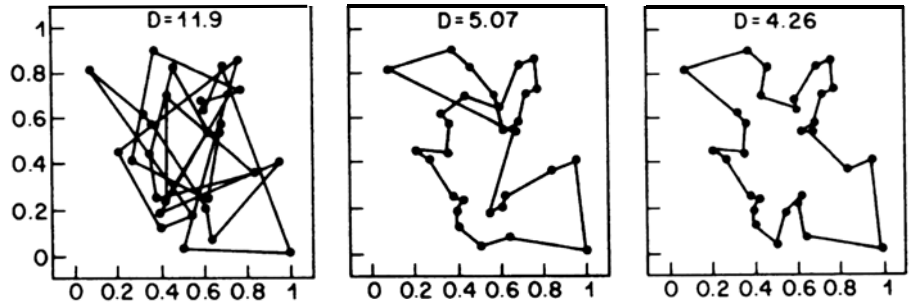


Abb. 6.11 *a) zufällige, b) suboptimale und c) beste Rundreise (nach [HOP85])*

Die Anwendung des Hopfield-Modells auf das Problem des Handlungsreisenden ist nur ein Beispiel aus der Problemklasse der NP-vollständigen Probleme. Ein anderes Beispiel besteht aus dem Problem, *n* verschiedene Arbeiten auf *m* Maschinen so zu verteilen, dass alle Arbeiten möglichst schnell verrichtet werden. Dies ist nicht nur für die Arbeitsorganisation in Fabriken interessant, sondern auch für das Problem der Lastverteilung in Multiprozessoranlagen. In der Arbeit von Protzel et al. [PRO93] wurde dies für eine fehlertolerante, adaptive Lastverteilung im fehlertoleranten SIFT-Multiprozessorsystem gezeigt. Bei der Modellierung von defekten Neuronen (defekte Schedulingprogramme) durch feste Zustände S_i konvergierte der Systemzustand selbst bei defekten Prozessoren immer zu einem stabilen Zustand für ein sinnvolles Schedulingschema.

Anwendungen des Hopfield-Modells sowie anderer neuronaler Netze im Telekommunikationsbereich (dynamische Zuteilung von Funkkanälen und *routing*-Wegen in Computernetzen) sind in [FMT93] beschrieben.

6.2.3 Die Speicherkapazität

Angenommen, wir wollen das Hopfield-Netz im Gegensatz zum vorher behandelten Problem des Handlungsreisenden nun für die Aufgabe verwenden, Muster zu speichern. Wie wir für den Spezialfall autoassoziativer Speicherung wissen, lassen sich besonders gut Muster abspeichern, wenn sie orthogonal sind. In diesem Fall reicht es, für das Abspeichern von *M* Mustern $\mathbf{S}^k$ die Gewichte w_{ij} nicht über spezielle Energiefunktionen zu bestimmen, sondern dafür die Hebb'sche Regel zu verwenden

$$w_{ij} = 1/n \sum_{k=1}^{M} S_i^{\,k} S_j^{\,k} \quad \text{bzw. } \mathbf{W} = 1/n \sum_{k=1}^{M} \mathbf{S}^k \left(\mathbf{S}^k\right)^T \tag{6.49}$$

Ist der Anfangszustand $\mathbf{S}^r$ zum Zeitpunkt t gegeben, so ergibt sich der Zustand zum Zeitpunkt t+1 mit Gl. (6.28) und Gl.(6.29) durch

$$\mathbf{S}(t+1) = \mathbf{y}(t+1) = \mathbf{sgn}(\mathbf{W}\mathbf{S}^r) \tag{6.50}$$

mit der Vektorfunktion **sgn**(**x**), die für einen Vektor komponentenweise mit $\mathrm{sgn}(x_i)$ das Vorzeichen bestimmt.
In diesem Modell ist der Zustand binär $S_i \in \{-1,+1\}$ und damit ist die Länge $|\mathbf{S}|^2 = n$ konstant Da die Zustandsvektoren $\mathbf{S}^k$ und $\mathbf{S}^r$ nach Voraussetzung orthogonal sind, ist $(\mathbf{S}^k)^T\mathbf{S}^r = 0$ für $k \neq r$ und es ergibt sich mit $\mathbf{S}^T\mathbf{S} = |\mathbf{S}|^2 = n$

$$\mathbf{S}(t+1) = \mathbf{sgn}\left(1/n \sum_k \mathbf{S}^k(\mathbf{S}^k)^T\mathbf{S}^r\right) = \mathbf{sgn}(\mathbf{S}^r) = \mathbf{S}^r \tag{6.51}$$

so dass sich der gespeicherte Zustand reproduziert; er wirkt als *stabiler Zustand*.

Um zu prüfen, ob auch die Nachbarzustände in ihn übergehen und er damit als *Attraktor* für seine Umgebung wirkt, speichern wir nur ein einziges Muster $\mathbf{S}^r$ in das Netzwerk ein. Dann verwenden wir als initialen Zustand **S**(0) eine in *p* der *n* Komponenten gestörte Form des gespeicherten Musters $\mathbf{S}^r$. Damit ist die Aktivierung des *i*-ten Neurons

$$z_i(0) = \sum_j w_{ij} S_j(0) = 1/n\, S_i^r \sum_j S_j^r S_j(0) \tag{6.52}$$

In der Summe von Gl. (6.52) gibt es *p* Summanden, bei denen das Produkt $S_j^{\,k} S_j(0)$ aus ungleichen Faktoren besteht mit $S_j^{\,k} S_j(0) = -1$, und an *n–p* Stellen gleiche Faktoren aufweist mit $S_j^{\,k} S_j(0) = +1$. In der Summe ist also *p* mal (–1) und *(n–p)* mal +1, so dass die Summe gleich *n–2p* ist. Aus Gl. (6.52) wird so

$$z_i(0) = 1/n\ S_i^r\ (n–2p) = S_i^r(1–2p/n) \tag{6.53}$$

Der Netzzustand nimmt somit trotz gestörten Initialzustands den ursprünglich gespeicherten Zustand $\mathbf{S}^r$ an, wenn

$$\mathbf{S} = \mathbf{sgn}\ (\mathbf{z}) = \mathbf{sgn}(\mathbf{S}^r(1–2p/n)) = \mathbf{S}^r \tag{6.54}$$

ist. Dies ist genau dann der Fall, wenn $(1–2p/n) > 0$ oder $n/2 > p$ ist. Dies bedeutet, dass bei einem gespeicherten Muster eine "Wiedererkennung" oder Musterergänzung nur möglich ist, wenn die Zahl der gestörten Stellen und damit der *Hamming-Abstand* zum gewünschten Muster nicht zu groß ist. Durch die konstante Aktivität $|\mathbf{S}|^2 = n$ im Netzwerk sind die Ähnlichkeitskriterien der maximalen Korrelation und des minimalen Abstands einander gleich; für eine korrekte Klassifikation reicht eine konstante Schwelle aus.

In der nachfolgenden Abb. 6.12 ist die Konvergenz anhand vier ausgewählter Zustände von der Eingabe eines stark gestörten Musters "A" zu dem vorher abgespeicherten Muster "A" gezeigt. Der Vektor $\mathbf{S}^k$ ist als 20×20 Matrix dargestellt, wobei ein Punkt für +1 und ein Leerzeichen für –1 gezeichnet wurde.

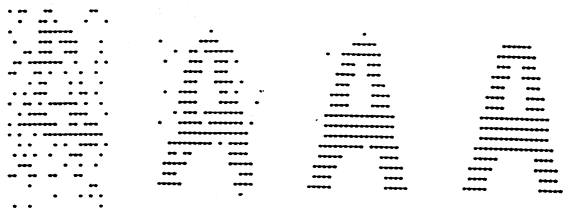


Abb. 6.12 *Musterergänzung und Rauschunterdrückung (nach [KIN85])*

Allerdings sollte an dieser Stelle ein wichtiger Unterschied zwischen dem einfachen und dem rückgekoppelten Assoziativspeicher erwähnt werden: Durch die Rückkopplung gibt es eine Abstimmung der Neuronen untereinander, so dass trotz konstanter Schwelle immer ein gespeichertes Muster existiert, zu dem der Systemzustand konvergiert. Es gibt also im Unterschied zum einfachen Assoziativspeicher keine Situation, in der das eingegebene Muster so stark von den gespeicherten Mustern abweicht, dass das System mit der Ausgabe des Nullvektors "Muster unbekannt" signalisieren kann. Diese Unfähigkeit zum "Nicht-Wiedererkennen" ist ein ernsthaftes Problem in der Klasse der rückgekoppelten Netzwerke und muss mit speziellen Maßnahmen (s. z.B. das ART-Modell) behandelt werden.

Das Speichervermögen

Die *Speicherkapazität* des Hopfield-Modells ist stark von der Kodierung der zu speichernden Muster geprägt. Betrachten wir nur *orthogonale* Muster, so gibt es gerade M = *n* Muster (*n* Basisvektoren!), die von *m* = *n* Neuronen zuverlässig gespeichert werden können. Jedes der *n* Muster hat *n* Bit, wobei nach Amari [AMA72] maximal H = 1 Bit pro unabhängigem Speicherelement (Gewicht) gespeichert werden kann.

Für einen Abruf eines gespeicherten Musters (Musterergänzung bzw. Konvergenz zum gespeicherten Muster mit geringstem Hammingabstand) ist allerdings der Konvergenz- oder Einzugsbereich nicht mehr so groß wie im vorigen Beispiel mit einem Muster. Es lässt sich zeigen (s. z.B. [MÜ90]), dass gespeicherte Muster nur dann zuverlässig "ausgelesen" werden können, wenn die Abweichung (Störung) des initialen Zustands $\mathbf{S}(0)$ nicht mehr als p = 13,8% der *n* Stellen erfasst.

Für *nicht-orthogonale* Muster gelten allerdings schlechtere Verhältnisse. Betrachten wir beispielsweise die Fähigkeit, *M* beliebige Muster zu speichern, so zeigten Abu-Mostafa und St. Jaques [ABU85], dass allgemein $M \leq n$ gelten muss. Darüber hinaus lässt sich zeigen, dass

$$M \approx n/(4 \ln n) \qquad \text{[MCE87]} \quad \text{und} \tag{6.55}$$

sogar $M < n/[2 \ln(n) - \ln \ln (n)]$ [AMA88]

gilt. Letzteres bewies Amari sogar, ohne die Spin-Glass Analogie zu benutzen.

Eine andere Situation ergibt sich, wenn wir zulassen, dass die Zustände des Systems, die sich nach gewisser Zeit "einschwingen", *nur im Mittel* den gewünschten Zuständen (gespeicherten Mustern) entsprechen sollen. Mit dieser Erweiterung erlauben wir auch fehlerhafte Zustände, die sich durch das "Übersprechen" von Mustern (Kreuzkorrelationen) ergeben. Mit dem *Speichervermögen* $\alpha := M/n$ lässt sich eine Gruppe von Modellen charakterisieren, bei denen die zu speichernden Muster $\mathbf{S}^k$ als Zufallsmuster mit einem Kreuzkorrelations-Erwartungswert $\langle (\mathbf{S}^k)^T \mathbf{S}^r \rangle = 0$ angenommen werden. Für diese gilt bei Hebb'scher Speicherregel, dass bei zunehmender Zahl von abgespeicherten Mustern und damit zunehmendem α der Auslesefehler exponentiell ansteigt und bei

$$\alpha_{\text{Hebb}} = 0{,}138 \qquad \text{[AMIT85]} \tag{6.56}$$

mit einer 50%gen Fehlerrate ein Auslesen sinnlos werden lässt. Ordnen wir der zunehmenden Unordnung im Speicher eine "Temperatur" zu, so lässt sich das neuronale Netz mit einem magnetischen System vergleichen, das erhitzt wird und bei einer "Sprungtemperatur" plötzlich alle magnetischen Eigenschaften verliert. Die Vorgänge im Inneren dieses Systems lassen sich dabei durch Betrachtungen über Mittelwerte von Wechselwirkungen (*mean field theory, MFT*) approximieren.

Erhöhung der Speicherkapazität

Versuchen wir sehr viele Muster mit der Hebb'schen Regel zu speichern, so bewirkt die Veränderung der Gewichte ein "Überladen" der Speicherkapazität und ein zu starkes Wachstum der Gewichte. Um diesen Effekt zu begrenzen, ist es sinnvoll, zusätzlich einen Mechanismus zur Wachstumsbeschränkung einzubauen.

Eine Methode der Gewichtsregulierung besteht darin, die Größe der Synapsengewichte auf einen festen Maximalbetrag zu begrenzen, beispielsweise durch eine nicht-lineare Quetschfunktion. Schränken wir die Zahl der möglichen Zustände der Synapsen auf zwei ein (*clipped synapses*)

$$w_{ij} = M^{1/2}/n \qquad S\left(\sum_{k=1}^{M} M^{-1/2} S_i^{\,k} S_j^{\,k}\right) = +/- M^{1/2}/n \tag{6.57}$$

mit $S(z) := \mathrm{sgn}(z)$

so lässt sich zeigen [HEM87],[HEM88b] , dass α nur unwesentlich kleiner ist

$$\alpha_{clipped} = 0{,}102 \tag{6.58}$$

wobei für allgemeine, nicht-lineare Funktionen S(.) immer $\alpha \leq \alpha_{Hebb}$ gilt [HEM88a].

Dies ist eine für die VLSI-Technik sehr interessante Tatsache: Binäre Gewichte reichen also für eine Implementierung dieses Modells normalerweise aus. Auch für den Fall der begrenzt-linearen Funktion $S_L(.)$ fand Parisi [PAR86], dass der Speicher trotz der Gewichtsbegrenzung niemals die Fähigkeit verlor, neue Muster zu lernen. Damit kann diese Art von Speicher als *Kurzzeitspeicher* (Kurzzeitgedächtnis) angesehen werden, der immer wieder überschrieben wird. Die Existenz dieser Art von Speichern wird auch in der sensorischen Verarbeitung beim Menschen vermutet.

Eine andere Methode besteht darin, die Instabilitäten, die eine Überladung der Gewichte mit sich bringt, zu korrigieren. Bezeichnen wir die *normierte Stabilität* der *i*-ten Komponente eines Musters $\mathbf{S}^k$ mit

$$\beta_i^{\,k} = S_i^{\,k} \frac{\mathbf{w}_i^{\,T}}{\left|\mathbf{w}_i^{\,T}\right|} \mathbf{S} \tag{6.59}$$

so ist $\beta_i^{\,k} > 0$ genau dann der Fall, wenn nach Anlegen des Musters $\mathbf{S}(0) = \mathbf{S}^k$ die *i*-te Komponente erhalten bleibt. Bezeichnen wir mit κ die minimal gewünschte Stabilität ("Störabstand"), dann lässt sich mit der Lernregel

$$\Delta w_{ij} = \frac{1}{M} \sum_{k=1}^{M} S_B\left(\kappa - \beta_i^{\,k} S_i^{\,k} S_j^{\,k}\; F\left(\beta_i^{\,k}, w_{ij}\right)\right) \tag{6.60}$$

eine stabilitätsfördernde Korrektur erreichen. Die einfache Form mit F(.) = 1 von Gardner [GARD88] reicht dabei völlig aus, wie ein simulativer Vergleich [MÜWA92] zeigte.

Betrachten wir nur linear unabhängige Muster, so ist bei geeigneter Energiefunktion sogar ein α=1 im Grenzwert möglich [KAN87]. Mit einigen Kunstgriffen lässt sich das Speichervermögen sogar auf $\alpha = 2$ steigern [OP88]. Dies wird allerdings mit extremer Instabilität bei Abweichungen der Muster erkauft. Bei besserer Stabilität sinkt dagegen das Speichervermögen bei der Speicherung von Zufallsmustern auf die üblichen α = 0,138. Selbst unendliche Speicherkapazität ist rein rechnerisch möglich. Allerdings sollte man von derartigen Ergebnissen in der Praxis nicht zu viel erwarten. Ähnlich wie jede reelle Zahl *w* unendlich viele Werte (Zustände) annehmen kann, aber nur durch endlich viele Bits auf einem Rechner repräsentiert wird, ist die praktisch verwertbare Speicherkapazität durch die physikalische Genauigkeit der Gewichte (thermisches Rauschen!) auf endliche Werte begrenzt. Die Speicherkapazität ist durch unterschiedliche Definitionen auch oft nicht direkt vergleichbar. Eine Übersicht verschiedener Ergebnisse ist in [ENG93] enthalten.

Bei nicht-orthogonalen, beliebigen Mustern, die gespeichert werden sollen, wird die Sachlage (wie schon im Fall nicht-rückgekoppelter Assoziativspeicher) wesentlich problematischer. Hier kann man versuchen, den Fehler dadurch möglichst klein zu machen, dass man die günstigste Matrix **W** (kleinste quadratische Fehler) für die lineare Abbildung **y** = **WS** sucht. Nach Abschnitt 2.1 ist hierbei **W** durch die Moore-Penrose Pseudoinverse $\underline{\mathbf{S}}^{+}$ der Mustermatrix $\underline{\mathbf{S}} = (\mathbf{S}^{1},...,\mathbf{S}^{M})$ bestimmt. Die so gefundene Matrix **W** lässt sich als gute Näherung der ursprünglich gesuchten Matrix auffassen, obwohl eine nicht-lineare Ausgabefunktion vorliegt.

Die Speicherkapazität des Hopfield-Modells hängt aber nicht nur von der Kodierung, sondern auch von der Modellierung der Kodierung ab. Beispielsweise bewirkt eine nicht-zentrierte Binärkodierung mit $x_i \in \{0,1\}$ gegenüber einer zentrierten Binärkodierung mit $x_i \in \{-1,+1\}$ eine schlechtere Speicherkapazität [AMARI77], was auf den Unterschied im Lernverhalten unter der Hebb'schen Regel bei unterschiedlichen Modellen zurückzuführen ist. Im ersten Fall ist bei $S_i = 0$, $S_j = 1$ das Gewichtsinkrement $\Delta w = S_i S_j = 0$, im zweiten Fall dagegen bei $S_i = -1$, $S_j = 1$ ist $\Delta w = S_i S_j = -1$; das Gewicht ändert sich bei jedem Muster und kodiert mit seinem veränderten Zustand Information.

Ausgedünnte Netze

Nehmen wir an, dass nach jeder Konvergenz ("Anregung eines Zustandes") die betreffenden Synapsengewichte auch etwas mit der negativen Hebb'schen Regel vermindert werden ("Vergessen"), so werden nicht gelernte, schwache, durch "Übersprechen" von Mustern entstandene Fehlzustände aus dem System entfernt [KEE87] ("Gehirn-Reorganisation durch Träumen").

Dies können wir auch gezielt machen: Mit der Definition einer Schwelle s wird mit der Stufenfunktion S_B und der Vorschrift

$$w_{ij} = w_{ij}\, S_B(|w_{ij}| - s) \qquad \textit{annealed dilution} \qquad (6.61)$$

jede Synapse aus dem System entfernt, die kleiner als die Schwelle s ist. Dies führt zu einem *Ausdünnen* (*dilution*) des rückgekoppelten Netzes. Der Vorteil dieser Vorgehensweise liegt einerseits darin, dass wir weniger Gewichte brauchen, um die Muster abzuspeichern (Speicheraufwand), und andererseits die Komplexität des Speicheralgorithmus von $O(m^2)$ auf $O(mN)$ bei jeweils N Nachbarn pro Neuron sinkt.

Im simulativen Vergleich zeigt sich [MÜL92], dass die Strategie aus Gl. (6.61) bessere Ergebnisse bringt als nur zufällig oder geometrisch-symmetrisch ausgedünnte Netze, da mit den kleinen Gewichten auch nur die kleinen, unbedeutenden Autokorrelationen weggelassen werden. Besonders effizient ist die Anwendung auf Sensordaten (Bilder, Töne), bei denen die Korrelationen zwischen zwei Punkten (Komponenten von $\mathbf{S}^k$) mit wachsender Entfernung abnehmen. Hier "lernt" das Netz beim Speichern durch die Ausdünnung die den Daten innewohnenden Korrelationen; das Netz wird dabei den Daten entsprechend "zurechtgeschneidert". Eine Übersicht über die Speicherkapazität ausgedünnter Netze ist auch in [ENG93] zu finden.

Aufgaben

(1) In Abb. 6.8 sind die Fixpunkte nicht genau in den Ecken, sondern etwas entfernt. Was sind die genauen Koordinaten der Fixpunkte? Die Ausgabefunktionen seien mit $S_T(k,z)$ vorgegeben.

(2) Zeigen Sie: Die Energie des Hopfield-Netzes bleibt beim parallelen Algorithmus konstant, wenn $S_i(t+1) = S_i(t-1)$ gilt.

(3) Definieren Sie eine Energiefunktion, um mit dem Hopfield-Modell das "Damen-Problem" zu lösen. Gesucht ist dabei eine der vielen Stellungen, in denen auf einem *nxn* Schachbrett n=8 Damen so positioniert werden, dass sie sich nicht direkt schlagen können.
Dies wird durch folgende Bedingungen gewährleistet:
a) Auf jeder Zeile des Brettes soll genau eine Dame stehen
b) In jeder Spalte des Brettes soll genau eine Dame stehen
c) Auf jeder von links oben nach rechts unten führenden Diagonale darf höchstens eine Dame stehen
d) Auf jeder von links unten nach rechts oben führenden Diagonale darf höchstens eine Dame stehen

Hinweis: Nehmen Sie eine Funktion an, die für jedes Feld *x,y* des Brettes "1" liefert, wenn es mit einer Dame besetzt ist, sonst "0". Speziell die dritte und vierte Bedingung sind schwer zu fassen; eine Diagonale von links oben nach rechts un-

ten wird zum Beispiel dadurch beschrieben, dass die Koordinaten x und y die Bedingung $x–y = i$, $\forall i$ mit $2 \leq i \leq 2n$ erfüllen.

7 Zeitsequenzen

Bei den bisher behandelten Netzwerken war die Aufmerksamkeit mehr auf die "eigentlichen" Funktionen wie Musterspeicherung, Funktionsapproximierung, Klassifizierung und dergleichen gerichtet, wobei die real existierende Zeit entweder nur als Hilfsvariable einer Differenzialgleichung diente, die "eigentliche Funktion" zu implementieren, oder aber als diskrete Iterationsvariable benutzt wurde.

In vielen Systemen der realen Welt ist aber der genaue zeitliche Verlauf einer Funktion äußerst wichtig. Beispiele dafür bilden die *Analyse* zeitlicher Funktionen, die in der Wirtschaft für die Analyse und Vorhersage von Aktienkursen und Firmenentwicklungen benutzt werden, und die *Synthese* von Zeitsequenzen für die Steuerung von Roboterbewegungen. In beiden Problemkreisen lässt sich die Fähigkeit der neuronalen Netze vorteilhaft einsetzen, die Systemparameter der Zeitfunktion zu lernen anstatt sie explizit (von Hand) zu programmieren.

7.1 Zeitreihenanalyse

Bei der Anwendung des Backpropagation-Algorithmus in NETtalk bestand die Grundidee der Eingabe- und Ausgabekodierung darin, den Zustand der Eingabevariablen zu diskreten Zeitpunkten gleichzeitig dem Netz als Eingabe anzubieten. Diese Idee lässt sich für die Verarbeitung von zeitlichen Funktionen verallgemeinern.
Sei ein Signal x(t) gegeben, das sich entlang einer Richtung mit endlicher Geschwindigkeit ausbreitet. In echten neuronalen Systemen wird die geringe Geschwindigkeit meist durch die relativ langsamen, physiologischen Erregungsmechanismen der Nervenzellen (Stofftransport an Synapsen) und die Parasitärkapazitäten entlang der Axone bewirkt. Das Signal kann dann an *n* Punkten (Anzapfstellen) beobachtet und auf (schnellen) Wegen an einer Verarbeitungseinheit zusammengeführt werden. Damit wird es möglich, das Signal x(t) an *n* Zeitpunkten $x(t_1)$ .. $x(t_n)$ parallel zu beobachten. Wählen wir uns jeweils gleiche zeitliche Abstände $\Delta t := t_i - t_{i-1}$ so entspricht dies einer *Abtastung* (Diskretisierung) des Signals x(t) durch eine *Haltespeicherkette* (*tapped delay line*). In Abb. 7.1 ist dies verdeutlicht.

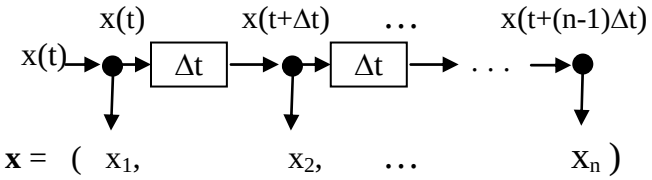


Abb. 7.1 *Parallelisierung eines Zeitsignals*

Das so definierte Signal **x**(t) stellt ein *zeitliches Fenster* über die Funktion x(t) dar; die Dimension *n* von **x** entscheidet dabei über den maximalen zeitlichen Abstand (n-1)Δt zwischen zwei Ereignissen (zwei Werten x(t) und x(t+(n-1)Δt)), bei dem die Ereignisse gerade noch im selben Muster **x** enthalten sind und sie parallel (und damit ihre Korrelation) beobachtet werden können. Ein neuronales Netz, das die Information von **x** nutzt, "errechnet" seine Ausgabe also immer nur auf der Basis der *n* neuesten Werte von x(t); das gesamte Training und die Vorgeschichte des Lernvorgangs dient nur dazu, diese Zuordnung so gut wie möglich zu machen. Gibt es dagegen - problembedingt - keine solche eindeutige, implizite oder explizite Abhängigkeit, so wird die Ausgabe des neuronalen Netzes zwangsläufig fehlerhaft sein.

7.1.1 Chaotische Sequenzen

Betrachten wir als erstes Beispiel einer Analyse von Zeitsequenzen ein schwieriges Problem: die Vorhersage von deterministischen, chaotischen Zeitreihen. Hier reichen schon kleinste Unterschiede in den Anfangsbedingungen aus, um bei gleicher, deterministischer Iteration völlig verschiedene Funktionen zu erzeugen.

Eine der bekannten chaotischen Zeitreihen wird durch die nicht-lineare, verzögerte Differenzialgleichung von *Glass-Mackey* erzeugt

$$\frac{\partial x}{\partial t} = \frac{a\, x(t-\tau)}{1 + x^{10}(t-\tau)} - b\, x(t) \qquad t \geq 0 \tag{7.1}$$

Die entsprechende Iterationsgleichung hat als Anfangsbedingungen die Werte $\{x(t) | -\tau \leq t \leq 0\}$: da die Werte $x(t-\tau)$ um τ verzögert werden, liegen sie im Anfangszeitraum $0 \leq t \leq \tau$ noch nicht vor und müssen vorgegeben werden. Im diskreten Fall mit *n* Zeitpunkten bilden die Anfangswerte ein endliches Tupel von *n* Zahlen, also einen *n*-dimensionalen Vektor. Im kontinuierlichen Fall bilden die Anfangswerte eine Funktion;

man spricht auch von einem "unendlich-dimensionalen Phasenraum". In Abb. 7.2 ist für konstante Anfangsbedingungen x ($-\tau \le t \le 0$) = const die Funktion x(t) bei a = 0.2, b = 0.1, τ = 30 aufgezeichnet.

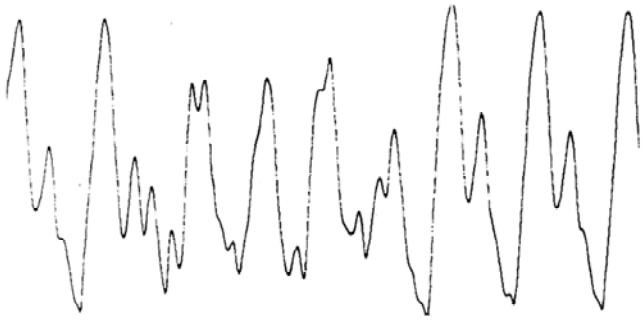


Abb. 7.2 *Die Glass-Mackey Zeitreihe (nach [LAP88])*

Die Zahl *n* der für eine Vorhersage benutzten Werte $x(t_1)$, .., $x(t_n)$ ist hier bei diskreten Zeitschritten sinnvollerweise nicht größer als τ.Lässt sich mit einer solchen Zeitreihe ein neuronales Netz derart trainieren, dass es bei *n* vorgegebenen Werten den *n+1*-ten Wert mit genügender Genauigkeit vorhersagen kann und damit den internen Mechanismus der Zeitreihe "analysiert" hat ?

Diese Frage stellten sich Lapedes und Farber [LAP88] und wählten sich dazu neuronale Netze mit der Backpropagation-Struktur und als Zielfunktion den kleinsten, quadratischen Fehler aus. Als Ausgabefunktion S(.) benutzten sie anstelle der ursprünglichen Fermi-Funktion die um den Nullpunkt symmetrische *tanh*-Funktion und trainierten jedes Netzwerk mit 500 Ein/Ausgabetupeln. Mit einer zweiten Menge von ebenfalls 500 Tupeln testeten sie die Netzwerke und notierten den beobachteten Fehler. Die Ergebnisse sind in Abb. 7.3 gezeigt.

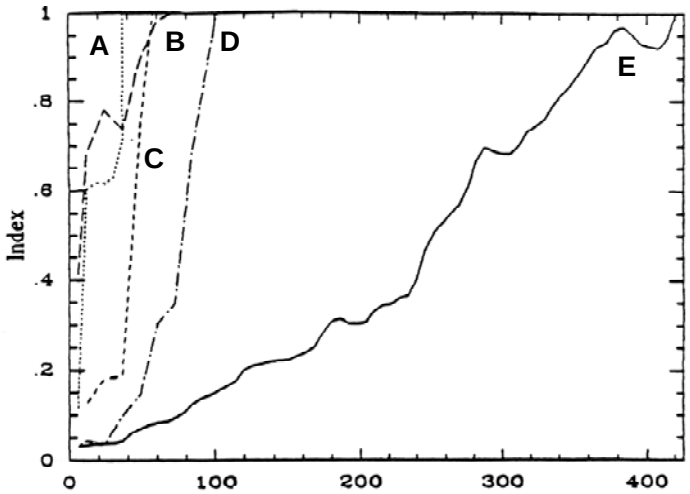


Abb. 7.3 Relativer Fehler der vorhergesagten Funktion

In der Zeichnung ist der Fehlerindex (der Betrag der mittleren Abweichung, geteilt durch die Standardabweichung) einer Prognose in Abhängigkeit von ihrem zeitlichen Abstand (Zahl der Zeitschritte) zum Zeitpunkt der Prognose für verschiedene Modelle A,B,C,D und E aufgetragen. Im Modell A wurden konventionelle, iterierte Polynome verwendet, im Modell B ein Widrow-Hoff-System (einschichtiges Backpropagation-Netzwerk), im Modell C jeweils ein nicht-iteriertes Polynom, in D Backpropagation Netze mit verschiedenen Eingabelängen *n* von 6 bis 100 Schritten, die jeweils extra trainiert wurden, und schließlich in E ein kleines iteriertes 6-Schritt Backpropagation-Netz. Es ist offensichtlich, dass das kleine, iterierte Netz den Mechanismus der iterierten Version von Gleichung (5.1.1) am besten approximiert.

Ein anderer Ansatz zur Lösung des Problems wurde von Deco und Schürmann [DES95] eingeführt. Dazu fasst man den Wert zum Zeitpunkt *t+1* als ein Wert auf, der von den Werten an den Zeitpunkten t-T, t-T+1, .., t abhängig ist. In Abb. 7.4 ist dies dargestellt.

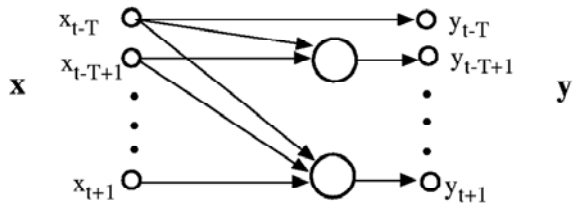


Abb. 7.4 Eine asymmetrische Architektur zur Zeitdekorrelation

Wie früher gezeigt wurde, kann man dies als Aufgabe auffassen, die T+1 Variablen x_{t-T}, .., x_t unabhängig voneinander zu machen. Für dieses Ziel reicht es in vielen Fällen bereits aus, aus den Ausgabesignalen alle Vielfachkorrelationen zu entfernen. Setzt man die Neuronenfunktion mit höheren Synapsen nach (2.1.36) mit

$$y_i = S_i(x_1,..,x_n) = x_i + \sum_{j<i} w_{ij}^{(1)} x_j + \sum_{j,k<i} w_{ijk}^{(2)} x_j x_k + ... \qquad (7.2)$$

an mit $t\text{-}T \le i \le t+1$, so modelliert man die Korrelationen x_i, $x_i x_j$, $x_i x_j x_k$ usw. mit ihren ersten und zweiten Momenten (Mittelwert und Varianz) und approximiert damit die unbekannte Wahrscheinlichkeitsdichte $p(\mathbf{x})$. Da der Wert x_{t+1} als Funktion $f(x_{t-T},..,x_t)$ von den anderen Zeitwerten abhängig ist, ist ein von allen Abhängigkeiten befreites y_{t+1} konstant. Dadurch lässt sich aus y_{t+1} = const mit Gl.(7.2) auf eine Aktivität z_{t+1} der höheren Synapsen

$$z_{t+1} = y_{t+1} - x_{t+1} = \text{const} - x_{t+1} = \text{const} - f(x_{t-T},..,x_{t+1}) \qquad (7.3)$$

schließen: bei Eingabe von $(x_{t-T},..,x_t)$ in das trainierte Netzwerk erhalten wir direkt die gesuchte Funktion

$$\begin{aligned} f(x_{t-T},..,x_{t+1}) &= \text{const} - z_{t+1} \\ &= \text{const} - \sum_{j<i} w_{ij}^{(1)} x_j + \sum_{j,k<i} w_{ijk}^{(2)} x_j x_k + ... \end{aligned} \qquad (7.4)$$

als numerischen Wert. Simulationen für die Glass-Mackey-Gleichung zeigen die guten Approximationseigenschaften des Konzepts [DES95].
Zur Illustration des Konzepts betrachten wir besser die chaotische Abbildung

$$x_{t+1} = 1 - 1{,}4x_t^2 + 0{,}3x_{t-1} \qquad (7.5)$$

die als 2-dim Abbildung (x,y) mit Gl.(7.5) und der Ersetzung und Iterationsgleichung $y_t := 0{,}3x_{t-1}$ als *Henon*-Abbildung bekannt ist. In Gl.(7.5) sehen wir, dass in diesem Falle der Wert zum Zeitpunkt *t+1* gerade von zwei Werten der Zeitpunkte t und t-1 abhängig ist. Zur Modellierung reichen also mit T = 1 zwei formale Neuronen aus. In Abb. 7.5 sind links die drei ersten Kanäle (Werte der Variablen) y_1, y_2 und y_3 untereinander gezeigt; rechts im Gegensatz dazu die Ergebnisse bei einer

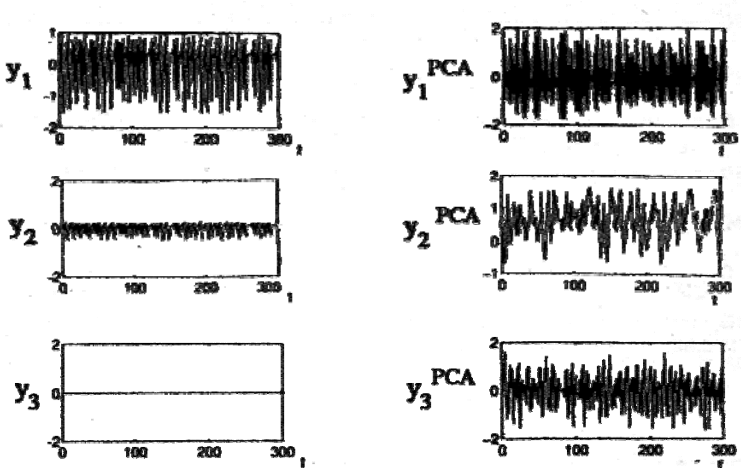


Abb. 7.5 *Vielfachdekorrelation und PCA (nach [DES95])*

Beschränkung auf Einfachkorrelationen, d.h. einer einfachen PCA wie sie in Kapitel 3 eingeführt wurde. Man sieht, dass nur Mehrfachkorrelationen die Ausgabevariablen dekorrelieren und damit die Abhängigkeiten korrekt nachbilden können.

7.1.2 Börsenkurse

Ein weiteres, sehr populäres Beispiel von Zeitreihen bilden die Börsenkurse. Gerade hier, wo man durch richtige Prognosen viel Geld verdienen kann, konzentrieren sich die Hoffnungen von manchen Leuten, die von einem "intelligenten, neuronalen System" sich große Vorteile versprechen. Das neuronale Netz soll - *deus ex machina* - über Nacht eine Aufgabe erfolgreich durchführen, an der schon manche Wirtschaftswissenschaftler und Börsenpraktiker "sich die Zähne ausgebissen" haben. Viele Seminare, Tagungen und Untersuchungen [TUR92] werden zur Zeit zu diesem Thema durchgeführt. Können neuronale Netze diese Hoffnungen erfüllen ?

Betrachten wir dazu beispielsweise die Untersuchungen von E. Schöneburg [SCHÖ90]. Neben verschiedenen anderen Modellen verwendete auch er ein Backpropagation-Netzwerk mit einem 10-Tages-Zeitfenster (n = 10). Das Netzwerk wurde darauf trainiert, den 11. Wert vom darauffolgenden Tag vorherzusagen. In Abb. 7.6 ist das Ergebnis für Mercedes-Aktien für den Zeitraum vom 18.5.89 bis 13.7.89 gezeigt.

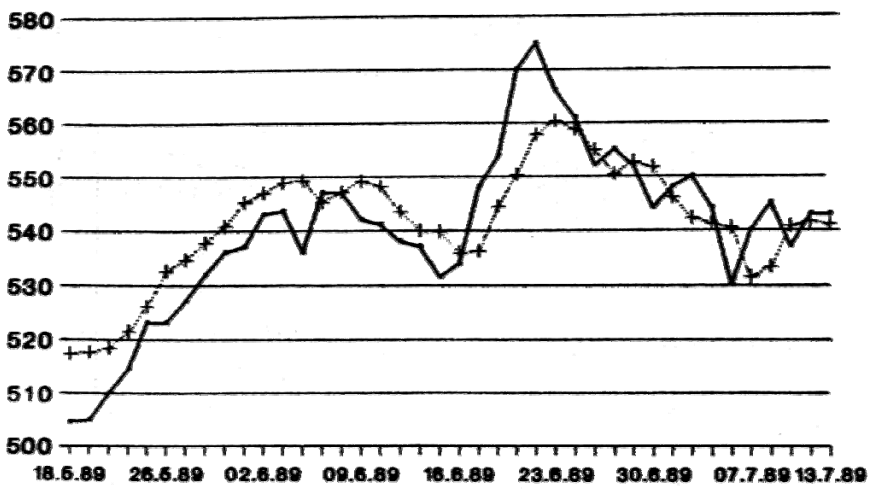


Abb. 7.6 *Vorhersage von Aktienkursen (nach [SCHÖ90])*

Die eingegebenen Werte bewirken zwar eine Korrektur der Prognose, es bleibt aber immer ein Prognosefehler. Man gewinnt leicht den Eindruck, dass die Prognose gerade um einen Tag (das fehlende Wissen!) versetzt dem tatsächlichen Kurs hinterher läuft. Der Autor erklärt das damit, dass das System wohl "die alte Börsenregel entdeckt" habe, für die Vorhersage den aktuellen Tageswert plus eine kleine Veränderung in die richtige Richtung anzusetzen.

Es ist zweifelsohne eine *mögliche* Strategie, durch eine lineare Interpolation bzw. abgebrochene Taylorentwicklung (Tageswert plus erste Ableitung) eine Prognose für eine unbekannte Funktion zu erreichen. Die Prognose müsste aber besser werden, wenn der tatsächliche, interne Mechanismus eines Aktienkurses vom neuronalen Netz gelernt wird - *wenn* es überhaupt einen solchen Mechanismus gibt. An dieser Stelle kommen wir nun zu dem eigentlichen Problem der Aktienkursanalyse: Wenn wir nur den Kursverlauf der Vergangenheit eingeben und daraus erwarten, dass das neuronale Netz eine fehlerfreie Prognose abgibt, so setzen wir voraus, dass alle für eine fehlerfreie Prognose nötigen Informationen bereits vorliegt. Dies ist aber genau *nicht* der Fall, wie wir aus der täglichen Praxis wissen: Der Aktienkurs kommt als Resultierende vieler menschlichen Aktionen zustande, die von einer Menge anderer Informationen wie beispielsweise Kursverlauf anderer Aktien, Goldpreis, Arbeitslosigkeit, Zinsniveau, öffentliche Daten der betreffenden Firma und – *last not least* – internes Wissen und Mutmaßungen, die nicht öffentlich sind. Ein neuronales Netz kann - wenn überhaupt - nur mit Hilfe der Informationen einen zuverlässige Prognose durchführen, die den "Machern" der Börsenkurse ebenfalls bekannt ist. Dabei handelt allerdings überhaupt nicht mystisch: Wie

wir aus Kapitel 2 wissen, lässt sich die lineare Grundfunktion des Backpropagation-Netzwerks mit einer Hauptkomponentenanalyse vergleichen.

In diesem Sinne lässt sich nun die am Anfang gestellte Frage beantworten: Falls es einen nachvollziehbaren Mechanismus zur Prognose von Börsenkursen gibt, so ist ein neuronales Netz ein nützliches Hilfsmittel, um schnell und bequem mit Hilfe eines einfachen Algorithmus eine statistische Analyse vorliegender Daten vorzunehmen. Dies kann man aber auch mit anderen bewährten, mathematischen Hilfsmitteln erreichen, falls man sie hat und beherrscht. Das "intelligente, neuronale System" hat dabei nur einen Intelligenzgrad, der mit einer Küchenschabe verglichen werden kann durchaus ausreichend, um eine begrenzte statistische Analyse der Umgebungsereignisse durchzuführen, aber überhaupt nicht vergleichbar mit menschlicher Intelligenz. Man muß deshalb im konkreten Fall immer mit konkreten Daten ermitteln, ob die so zwangsläufig begrenzte Treffsicherheit neuronaler Prognosen für den angestrebten Zweck ausreichend ist und man sich überhaupt mit dieser Art von Analyse und Prognose zufrieden gibt.

Neuere, seriöse Untersuchungen dieses Problemsbereichs leiden dabei allerdings unter einem problemspezifischen Dilemma. Findet man (z.B. im Auftrag einer Bank) einen Netzalgorithmus, der einige wenige Prozentpunkte bessere Leistungen als die bisher üblichen Methoden bringt (was bei Millionenumsätzen durchaus schon interessant ist), so bleibt dies Geheimnis des Auftraggebers, da ja sonst sein Wettbewerbsvorteil dahin schmilzt und die Investition in die Forschung sich nicht auszahlt. Aus diesem Grund, so argumentieren Wirtschaftswissenschaftler, wird es niemals ein verlässliches, öffentlich zugängiges Verfahren für Finanzprognosen geben, da das Wissen, das damit alle haben, automatisch die Vorhersage invalidiert. Die genaue Vorstellung eines erfolgreichen Verfahrens an dieser Stelle (z. B. der Erfolge der Arbeiten der Siemens-Gruppe) ist deshalb prinzipiell problematisch. Alle genaueren Veröffentlichungen stammen von öffentlichen Forschungsgruppen, s. [REF93], [REF94], aber auch [TR92] und [CAL94].

Dies gilt aber nicht für alle Bereiche: Beispielsweise erhoffen Banken und Versicherungen aus der Diagnose von finanziellen Daten Erkenntnisse zu gewinnen, die Ihnen eine bessere Beurteilung der Kunden sichern [SLB92], [REH92] oder mittelfristige Prognosen verbessern [REH90a,b].

Da bei dem finanziellen Anwendungsbereich meist bestehende Algorithmen verwendet werden, sei als neuere Übersicht auf die Arbeit von Hollick [HOL93] verwiesen.

7.2 Feed-forward Assoziativspeicher

Das vorige Abschnitt beschäftigte sich vorwiegend mit der *Analyse* von Zeitsequenzen. In diesem Abschnitt gehen wir davon aus, dass uns die Bildungsgesetze und Vorhersa-

ge der Zeitsequenzen nicht weiter interessieren. Vielmehr verlangen wir von einem System, dass es vorgegebene, willkürliche Zeitsequenzen speichert, so dass man sie beliebig abrufen kann. Die Fragestellung nach der *Synthese* von Zeitsequenzen ist vor allen Dingen bei Kontroll- und Steuerungsaufgaben wichtig; ein Beispiel dafür ist die Steuerung der Gelenkmotoren eines Roboters, um eine bestimmte Bewegung auszuführen.

Der Unterschied zwischen der Analyse und Synthese von Zeitsequenzen ist dabei fließend: Genauso wie die Prognose eines Zeitreihenwertes als Musterergänzung und damit als Abrufen eines gespeicherten Musters gesehen werden kann, lässt sich der Lernprozess nach der Eingabe von mehreren, verrauschten Versionen der selben Zeitsequenz als "Analyse" bezeichnen.

Eines der einfachsten Systeme, eine Sequenz von Mustern zu lernen, ist die "Sternlawine" (*outstar avalanche*) von Grossberg [GRO69]. Die Grundidee besteht darin, zunächst einzelne Mustervektoren **x** (*räumliche* Muster) assoziativ zu lernen. In einem zweiten logischen Schritt kann man dann mehrere Muster über eine Verzögerungsleitung (s. Abb. 5.1.1) in einer zeitlichen Reihenfolge auslösen (*räumlich-zeitliche* Muster). Betrachten wir zunächst den einfachen, assoziativen Grundmechanismus.

7.2.1 Die OUTSTAR-Konfiguration

Durch ein *Kontrollsignal* lässt sich ein "räumliches" Muster abrufen, wenn man beide miteinander assoziiert. Denken wir an unseren assoziativen Speicher aus Kapitel 2.1, so besteht hier das Eingabemuster **x** aus einer einzigen Komponente x_0; das dazu assoziierte Muster **y** aus dem zu lernenden räumlichen Muster **x**(t). In Abb. 7.7 ist diese Konfiguration für ein Muster "U" dargestellt.

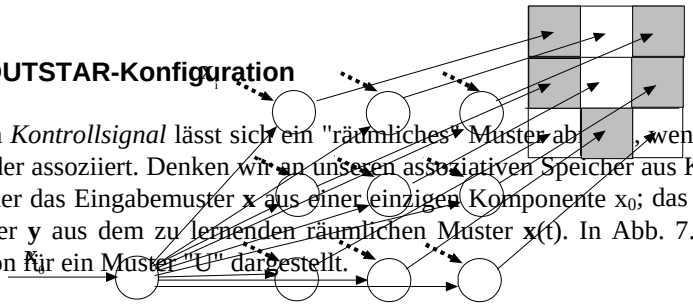


Abb. 7.7 *Die Outstar-Konfiguration*

Ordnet man die Speicherneuronen kreisförmig um den Kontrollknoten an, so gehen alle Kontrollsignale *sternförmig* vom Kontrollknoten aus; Grossberg bezeichnet deshalb diese Anordnung als *Ausgabestern* (*outstar*).

Wie in allen Arbeiten von Großberg (vgl. Kapitel 2.7) sind die Funktionen der Einheiten durch Differentialgleichungen gegeben. Die Bedeutung und die Umsetzung solcher kontinuierlichen Differentialgleichungen in diskrete Differenzengleichungen wurde bereits in Abschnitt 1.2.1 näher erläutert. Im Folgenden sind die zeitlichen Ableitungen mit einem Punkt "?" gekennzeichnet.

Die Aktivität z_0 des Kontrollknotens v_0 stabilisiert sich langfristig ($t>>\alpha^{-1}$) auf dem Kontrollsignal $x_0(t)$

$$\dot{z}_0(t) = -\alpha\, z_0(t) + x_0(t) \tag{7.6}$$

und die der *m* kontrollierten Assoziativknoten auf der gewichteten, um τ verzögerten Eingabe

$$\dot{z}_i(t) = -\alpha\, z_i(t) + \beta\, z_0(t-\tau)\ \bar{w}_i(t) + x_i(t) \tag{7.7}$$

mit den Konstanten α und β und den normierten Gewichten

$$w_i(t) = w_i(t) / \sum_j w_j(t)$$

Die Gewichte werden mit der Hebb'schen Regel (1.5.2b) gelernt

$$\dot{w}_i(t) = -\gamma_1\, w_i(t) + \gamma_2 z_0(t-\tau) z_i(t) \tag{7.8}$$

Welche Leistungen kann nun ein solches System lernen? Dazu stellte Grossberg einen Satz auf, der Folgendes besagt [GRO69]:

SATZ: Angenommen, die *m* Komponenten des Signal **x**(t) (z.B. die Grauwerte eines Bildes) seien *relativ* nach der Wahrscheinlichkeit $\mathbf{p} = (p_1,..,p_m)$ verteilt, also

$$p_i := x_i(t) / \sum_j x_j(t) \text{ mit } x_i>0\ ,\ \sum_j p_j=1 \tag{7.9}$$

wobei $x_0(t)$ in einem endlichen Zeitraum T_0 in einer "Mindeststärke" (= Integral, s. [GRO69]) vorhanden (und danach unbestimmt) ist und das zu assoziierende Muster **x**(t) ebenfalls in einer Mindeststärke während dieser Zeit anliegt

$$x_i(t) = \begin{cases} a_i \left| \mathbf{x}(t) \right| & t < T_m \\ 0 & t \geq T_m \end{cases} \qquad T_m > T_0 \tag{7.10}$$

Dann gilt u.a.

$$\lim_{t\to\infty} \bar{z}_i(t) := z_i(t) / \Sigma_j\, x_j(t) = p_i \qquad \text{und} \qquad \lim_{t\to\infty} \bar{w}_i(t) = p_i\ . \tag{7.11}$$

Mit dem System nach Gl.(7.9)-(7.11) wird also nicht ein "Bildsignal" **x**(t) *absolut* assoziiert, sondern nur in seiner *relativen* Ausprägung; das Bildsignal kann in seiner absoluten Stärke (fast) beliebig schwanken und wird trotzdem exakt mit dem Kontrollsignal x_0 assoziiert; allein die Eingabe von x_0 reicht aus, das Muster **x** zu erregen (Musterergänzung bzw. Auslesen des Assoziativspeichers).

7.2.2 Die Sternlawine

Möchten wir nun eine Sequenz von Mustern lernen, so können wir den Verzögerungsmechanismus aus Abb. 7.1 nutzen. Hier wird ein Signal x(t) auf einer Wegstrecke verzögert und in gleichmäßigen Abständen abgetastet. Nach jedem Zeitabschnitt Δt erreicht das Signal einen neuen Knoten v_0^t, so dass die Knotenfolge $v_0^1, v_0^2, \ldots, v_0^M$ durchlaufen wird. Benutzen wir jeden Knoten v_0^t als Kontrollknoten eines Sternsystems, so werden in den dazugehörenden *M* Speichern *M* Muster gespeichert, die im zeitlichen Abstand Δt (korreliert mit dem Kontrollsignal) angeboten wurden. Ein Signalimpuls x_0, der am Anfang der Kette eingeben wird, löst dann beim Durchlaufen nacheinander an den angeschlossenen Kontrollknoten die Ausgabe des dort jeweils assoziierten Musters aus, so dass mit diesem System *M* Zeitwerte eines beliebigen, zeiträumlichen Mustersignals **x**(t) gespeichert und abgerufen werden können. Dies entspricht einem "Abtasten" (*Sampling*) des Mustersignals, falls es sich nicht zu schnell verändert. In Abb. 7.8 ist eine solche Konfiguration für getrennte, separate Ausgaben (Matrix) gezeigt; in Abb. 7.9 das Entsprechende für eine gemeinsame Ausgabe.

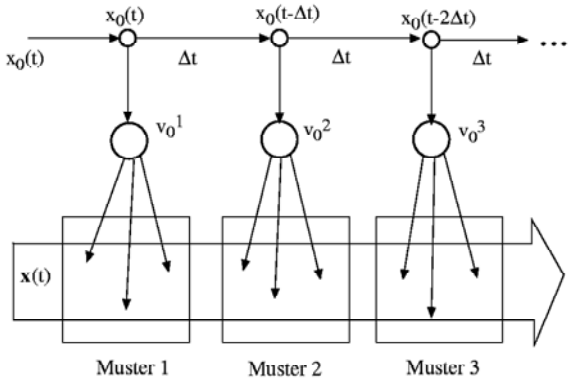


Abb. 7.8 Zeitlich-räumliche Sequenz von Mustern

Die Differentialgleichungen (7.6)bis (7.8) gelten für die Konfiguration bei separaten Ausgaben sinngemäß für jede einzelnen Speicher. Bei gemeinsamer Ausgabe münden dagegen alle Kontrollsignale in den Knoten des gleichen Speichers, allerdings in unterschiedlichen Gewichten. Jedes der *M* Muster wird also in einem anderen Satz von Gewichten gespeichert, so dass die Speicherung der Muster unabhängig voneinander

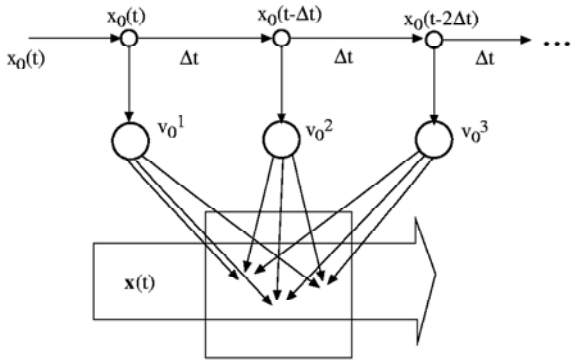


Abb. 7.9 Zeitliche Sequenz von Mustern

ist und keine Wechselwirkungen zwischen Mustern auftreten können. Im Unterschied zu den anderen Modellen im Abschnitt 7.3 können die Muster also beliebig sein.

Für den *k*-ten Kontrollknoten v_0^k und seinen Speichergewichten w_i^k im Gesamtzeitraum T mit M:=⌊T/Δt⌋+1 Mustern gilt

$$\dot{z}_0^k(t) = -\alpha\, z_0^k(t) + x_0(t-k\Delta t) \qquad k = 0 \,..\, M-1 \tag{7.12}$$

$$\dot{z}_i(t) = -\alpha\, z_i(t) + \beta \sum\nolimits_k \bar{w}_i^k(t)\, z_0^k(t-\tau) + x_i(t)$$
$$\text{mit } \bar{w}_i^k(t) := w_i^k(t) \,/\, \sum\nolimits_j w_j^k(t) \tag{7.13}$$

$$\dot{w}_i^k(t) = -\gamma_1\, w_i^k(t) + \gamma_2 z_0^k(t-\tau) z_i(t) \tag{7.14}$$

Die zeitverzögerte Auslösung einer Sequenz von gespeicherten Musterwerten $\mathbf{x}(t_1)$, $\mathbf{x}(t_2)$, .. erinnert stark an den Mechanismus des Kleinhirns, bei dem in den langsamen, parallelen Fasern (Zeitverzögerung!) hintereinander die Aktivitäten zur Muskelsteuerung ausgelöst werden können und der deshalb als "Timer" im Millisekundenbereich angesehen wird [BRAI67]. Die Impulsfolgen der Parallelfasern im Kleinhirn wurden von dem berühmten Gehirnforscher Ram'on y Cajal als "Lawine" bezeichnet, so dass Großberg in Anlehnung daran das obige Modell als *outstar avalanche* oder "Sternlawine" bezeichnete.

Stabilisierung

Bei dem oben vorgestellten Modell wird genau dann im Intervall [kΔt, kΔt+dt] mit Gl.(7.14) gut gelernt, wenn $x_0(t)$ und damit $z_0(t)$ nur in diesem Intervall ungleich null sind. Bei einer beliebigen Funktion von $x_0(t)$, die nicht nur die Werte eins und null

annimmt (Rauschen!), führt diese Abweichung von der idealen Rechteckimpulsform ("Sample"-bzw. Triggerimpuls) zu "ungenauem" Lernen; beim Abbrechen des Lernvorgangs nach endlicher Zeit bleibt noch eine relativ große Abweichung vom Idealzustand. Um den Lernprozeß zu verbessern, ist es deshalb sinnvoll, kleine Signalstärken ("Schmutzeffekte") durch nicht-lineare Ausgabefunktionen S(.) mit den Schwellen s_i^k zu unterdrücken. Die Gleichungen (7.12) und (7.13) werden dann zu

$$\dot{z}_0^k(t) = -\alpha\, z_0^k(t) + S(x_0(t-k\Delta t)-s) \tag{7.15}$$

$$\dot{z}_i^k(t) = -\alpha\, z_i^k(t) + \beta \sum_k S(z_0^k(t-\tau)-s_0^k)\ \bar{w}_i^k(t) + x_i(t) \tag{7.16}$$

Es lässt sich zeigen, dass der vorher vorgestellte Satz auch für dieses modifizierte System gilt. Weitere Modifikationen zur Stabilität lassen sich auch durch zusätzliche, inhibierende Verbindungen schaffen [GRO67].

7.3 Rückgekoppelte Assoziativspeicher

In Kapiteln 3.1 und 3.2 wurden verschiedene rückgekoppelte Netze vorgestellt, mit denen man Muster auto-assoziativ speichern konnte. Gab man Teile eines gespeicherten Musters ein, so konvergierte die Ausgabe unter bestimmten Voraussetzungen zu dem gesamten, gespeicherten Muster; das System ergänzte die fehlenden bzw. gestörten Musterteile. Der Grundmechanismus dieser Musterergänzung beruht dabei auf der verwendeten Hebb'schen Speicherregel: Alle Teile des Musters wurden miteinander korreliert gespeichert, so dass von jedem Teilstück über die stärkste Korrelation das Restmuster erschlossen werden kann. Begrenzend in diesem Schema wirken sich nur die Musterähnlichkeiten bzw. Korrelationen mit den anderen, ebenfalls gespeicherten Mustern aus.

Der Grundmechanismus der Speicherung und Wiedergabe von Zeitsequenzen $\mathbf{x}^1,\mathbf{x}^2,...\mathbf{x}^M$ mit rückgekoppelten Assoziativspeichern beruht nun gerade auf der Korrelation von Mustern untereinander: korreliert man ein Muster $\mathbf{x}^k$ mit dem darauf folgenden Muster $\mathbf{x}^{k+1}$ und verzögert die Rückkopplung, so wird nach der Ausgabe des k-ten Musters beim nächsten Zeittakt das Muster *k+1* erregt.

7.3.1 Sequenzen ohne Kontext

Neben der Speicherung von statischen Mustern (s. Abschnitt 2.1) untersuchte Amari 1972 [AMA72] auch die Speicherung von Mustersequenzen. Abweichend von der Lernregel

$$\mathbf{w}_i(t+1) = (1-\alpha)\,\mathbf{w}_i(t) + \beta\mathbf{x}(t)\,x_i(t) \tag{7.17}$$

bzw. $$w_{ij}(t+1) = (1-\alpha)\,w_{ij}(t) + \beta x_i(t)\,x_j(t) \tag{7.18}$$

die ein einzelnes Muster stabilisiert, speicherte er mit

$$w_{ij}(t+1) = (1-\alpha)\,w_{ij}(t) + \beta x_i(t+1)\,x_j(t) \tag{7.19}$$

auch eine Sequenz ab. Die Gewichtsmatrix konvergiert auch hier zur Überlagerung aller *M* Muster

$$\mathbf{W} = 1/n \sum_k \mathbf{x}^{k+1}(\mathbf{x}^k)^T \tag{7.20}$$

Dabei gilt (s. [AMA72]):

- Alle Einzelmuster (und damit die ganze Sequenz) sind nur dann stabil, wenn die unerwünschten Korrelationen genügend klein sind, beispielsweise wenn alle Muster orthogonal zueinander sind.
- Wird *ein* Muster irgendwo aus der Sequenz eingegeben, so wird der Rest der Sequenz ausgegeben- der Anfang fällt weg.
- Ist ein Zustand nicht stabil, da er beispielsweise durch ähnliche Muster "überladen" wird (*Stabilitäts-Plastizitätsdilemma*), so kann die Sequenz "umkippen" und in einer anderen, öfters gespeicherten ("dominierenden") Sequenz aufgehen.
- Es ist also unmöglich, mit dem Modell der paarweisen Korrelation von Mustern das gleiche Muster in zwei verschiedenen Sequenzen oder mehrmals in der selben Sequenz zu verwenden, da diese dann nicht stabil werden.
- Wird zum letzten Muster einer Sequenz das erste assoziiert, so wird die gesamte Sequenz zyklisch ausgegeben: Das System wirkt als Rhythmusgenerator.

Auch Zeitsequenzen können durch die Simulationen von Willwacher 1976 [WIL76] illustriert werden. Dazu betrachten wir eine Sequenz, die in Abb. 7.10 visualisiert ist.

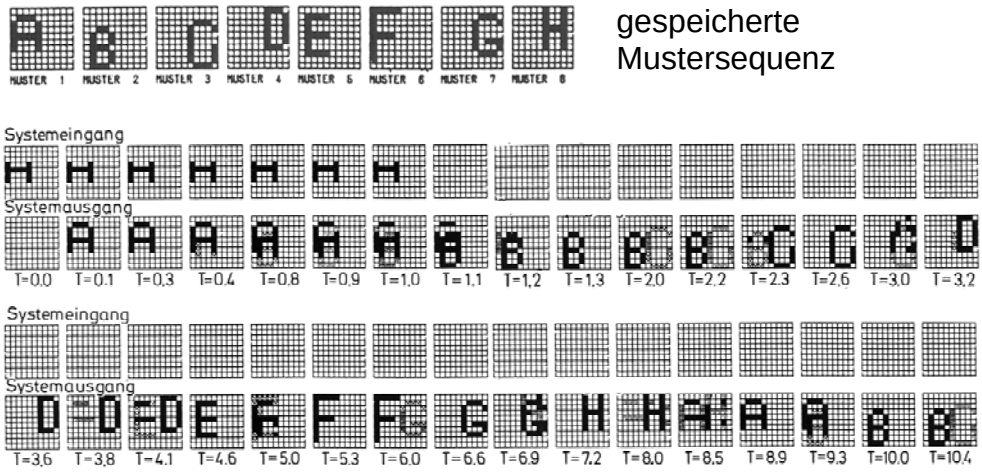


Abb. 7.10 *Sequenz von Buchstaben (aus [WIL76])*

Ausgehend von den Korrelationen nach Gl. (7.20) wurden die in der ersten Reihe abgebildeten Pixelmuster eingespeichert (vgl. Kapitel 2.1). Die Eingabe von Teilen des ersten Muster "A" führte dann sofort zu der Ausgabe einer Sequenz. Der zeitliche Abstand der Muster wird weitgehend von den Zeitkonstanten der simulierten Systemarchitektur bestimmt.

Wenn man den Betrag der Ausgabeaktivität über eine Schwellwertregelung in der Simulation verändert, ergeben sich interessante Situationen. In der folgenden Abb. 7.11 ist gezeigt, wie das System bei schrittweiser Erhöhung der Aktivität (Erniedrigung der Schwellen) bei einer Anregung reagiert.

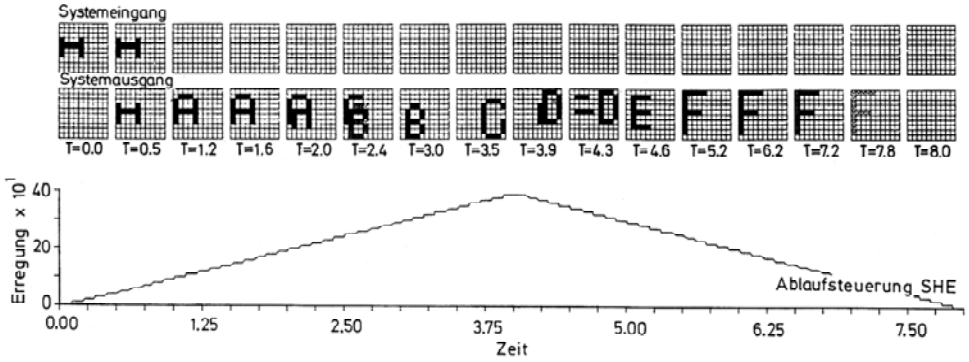


Abb. 7.11 *Parallele und sequentielle Muster (aus [WIL76])*

Bei kleiner Aktivität SHE werden alle Assoziationen unterdrückt (T = 0.0 in Abb. 7.11). Bei größerer Aktivität (T = 0.5) wird gerade das angebotene Muster reproduziert. Bei noch kleineren Schwellen (T = 1.2) reichen die Korrelationen aus, das gesamte Muster zu assoziieren und dann (bei T = 2.2) auch das folgende Muster dazu. Bei diesem Schwellwert geht also der parallele Assoziationsspeicher in den sequentiellen Modus über: die nächsten Muster werden sequentiell reproduziert. Wird die Aktivität wieder abgesenkt und die Schwellen erhöht, so stoppt die Sequenz (T = 6.2) und selbst das gerade assoziierte Muster wird schließlich unterdrückt (T = 7.8).

Geben wir in dem Modell von Amari ein Muster ein, so wird sofort mit dem nächsten Zeitschritt das nächste Muster der Sequenz assoziiert. Um dies zu verhindern und die Stabilität des bestehenden Musters für einige Zeitschritte zu erhalten, kann man beispielsweise für die Berechnung der Gewichte im Hopfieldmodell den normalen Hebb-Term, der die Stabilität eines Musters bewirkt, mit einer Assoziation zu dem darauf folgenden Muster verbinden

$$\mathbf{W} = \alpha\mathbf{W}^{(1)} + \beta\mathbf{W}^{(2)} = \alpha/n \sum_k \mathbf{x}^k(\mathbf{x}^k)^T + \beta/n \sum_k \mathbf{x}^{k+1}(\mathbf{x}^k)^T \qquad (7.21)$$

Der Faktor 1/n dient dazu, bei einer Multiplikation $\mathbf{W}\,\mathbf{x}$ den Zahlenwert der entstehenden Korrelation $(\mathbf{x}^j)^T\mathbf{x}^k$ auf das Intervall [-1,+1] zu normieren, da im Skalarprodukt maximal *n* Komponenten +1 oder -1 sind.

Die Überlagerung zweier Komponenten in Gl. (7.21) gibt aber Probleme: wählen wir β zu klein, so wird die Sequenz nicht sauber realisiert. Wählen wir es aber zu groß, so wird das folgende Muster "gleichzeitig" überlagert. Selbst bei einer wahrscheinlichkeitsgesteuerten, verzögerten Ausgabe kann sich Chaos ergeben [RIE88]. Es ist deshalb sinnvoll, auch die Ausgabe aus zwei Komponenten zusammenzusetzen: einer Aktivität $z_i^{(1)}(t)$, die den augenblicklichen Zustand stabilisiert und einer verzögerten Aktivität

$z_i^{(2)}(t)$, die das nächste Muster der Sequenz assoziiert. Sompolinski und Kanter [SOM86] sowie Kleinfeld [KLE86] gaben dazu ähnliche Lösungen an.

Im binären Fall mit $y(t+1) = S(z) = \mathrm{sgn}(z(t))$ ist nach Sompolinski und Kanter

$$z_i(t) = z_i^{(1)}(t) + z_i^{(2)}(t) = \sum_j w_{ij}^{(1)} S_j + \sum_j w_{ij}^{(2)} \bar{S}_j \tag{7.22}$$

mit dem gemittelten, verzögerten Signal

$$\bar{S}_j := \int_{-\infty}^{t} p(t-t')\, S_j(t')\, dt' \tag{7.23}$$

Das Zeitsignal $S_j(t)$ wird über die Gesamtzeit gemittelt, wobei die Gewichtungsfunktion $p(t-t')$ mit ${}_0\int^t p(t)\, dt = 1$ als Wahrscheinlichkeitsdichte aufgefaßt werden kann. Beispiele für die Gewichtungsfunktion sind

$p(t) = 1/\tau$ *Stufenfunktion* (7.24)

oder $p(t) = 1/\tau \exp(-t/\tau)$ *exponentieller Abfall* (7.25)

oder $p(t) = \delta(t-\tau)$ *Zeitverzögerung* (7.26)

Als hinreichendes Kriterium für eine stabile Zeitsequenz mit dem Zeittakt τ_0 fanden die Autoren, dass bei $\alpha = 1$ der Parameter β der Gleichung

$$\int_{\tau_0}^{2\tau_0} p(t)\, dt = 1/2\,(1-1/\beta) \tag{7.27}$$

genügen muss. Der Parameter β muss ≥ 1 sein und schränkt die Wahl des zeitlichen Abstandes τ_0 zweier Muster ein.

- Für die *Stufenfunktion*, die als Konstante bei $t = \tau$ verschwindet, ergibt sich aus Gl.(7.24) und (7.27) für $\beta>1$ die Gleichung $\tau_0 = (\tau/2)(1+1/\beta)$, so dass von minimalem zu großem β die Verzögerungszeit von τ auf $\tau/2$ herabsinkt.
- Für den *exponentiellen Abfall* aus Gl.(7.25) ergibt sich die Situation, dass mit $\tau_0 = \tau \{ \ln 2 - \ln [1-(2/\beta-1)^{1/2}] \}$ der Parameter β maximal 2 sein darf ($\tau_0 = \tau \ln 2$) und bei $\beta \rightarrow 1$ unendliches τ_0 hervorbringt.
- Die *Deltafunktion* aus Gl.(7.26) schließlich modelliert nur eine einfache Zeitverzögerung mit $\tau_0 = \tau$, $\beta > 1$.

Für den kontinuierlichen Fall modellierte Kleinfeld ein Neuron durch einen elektronischen Verstärker mit einer begrenzt-linearen, symmetrischen Ausgabefunktion S_L von (1.2.3b). Die Aktivität eines Neurons (eines Verstärkers) ist dann mit einer Differenti-

algleichung in der Form (1.2.0c) gegeben, die nun wie in (5.3.3) zusätzlich aus zwei additiven Komponenten besteht:

$$\tau_i \dot{z}_i(t) = - z_i(t) + \sum_j w_{ij}^{(1)} S_j + \sum_j w_{ij}^{(2)} \bar{S}_j \tag{7.28}$$

wobei die Zeitkonstante τ_i die "Ladezeit" des Verstärkers anzeigt. Im Grenzfall $\tau_i \to 0$ geht Gl.(7.28) in Gl. (7.22) über. Für den Fall des einfachen, mit Gl.(7.26) zeitverzögerten Signals zeigte Kleinfeld, dass sich, ähnlich wie im Hopfieldmodell, auch hier eine Energiefunktion aufstellen lässt, mit deren "Bergen" und "Tälern" sich der Übergang von einem stabilen Muster zu einem anderen erklären lässt.

Welche Rolle spielt nun die Länge der Zeitverzögerung bzw. die Verteilung p(t) der Verzögerungszeiten über die Synapsen bei der Stabilität der Zeitsequenzen ? In einer interessanten Arbeit zeigte A. Herz [HER88a], [HER88b], [HER89] mit einigen Mitarbeitern, dass der Einfluss wesentlich geringer ist, als man es für möglich halten würde. Sie gingen von einer kontinuierlichen Version von Gl.(7.22) aus, bei der sich eine Eingabe mit verschieden verzögerten Anteilen in einem Neuron als Überlagerung (Mittelung) einstellt

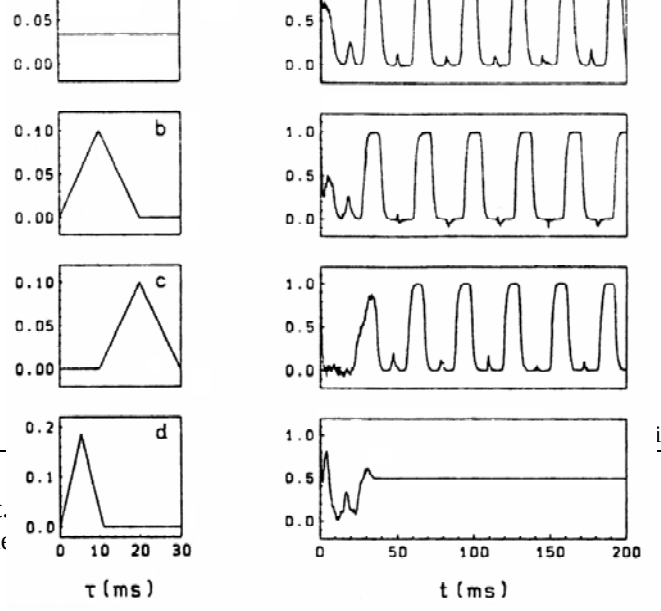
$$z_i(t) = \sum_{j \neq i} \bar{w}_{ij} \bar{S}_j(t) \tag{7.29}$$

mit Gl. (7.28) und der zeitgemittelten Form der Hebb'schen Regel

$$\bar{w}_{ij} := \frac{1}{nT} \int_0^T S_i(t) \bar{S}_j(t) \, dt \tag{7.30}$$

Die Ergebnisse für eine Sequenz aus drei Mustern sind in der nachfolgenden Abb. 7.12 für verschiedene Verteilungen von Verzögerungen gezeigt, die beim Lernen in Gl. (7.30) benutzt wurden. Links sind die vier verschiedenen Verteilungsfunktionen a,b,c und d für Zeitverzögerungen τ aufgetragen; rechts davon die Reaktion des Systems (Überlappung bzw. Korrelation der Ausgabe **z** mit dem gespeicherten Muster $\mathbf{x}^1$ der Sequenz). Jede Verzögerung τ wird dabei durch eine Verteilung nach Gl. (7.25) bzw. (7.26) erzeugt.

Zwei Dinge sind bemerkenswert: zum einen wird auch ein stabiler Rhythmus produziert, selbst wenn nur eine konstante Verteilung von einfach verzögerten Signalen (Verteilung a) vorliegt; zum anderen aber ist eine Mindestverzögerung τ_{min} = 10ms (größer als die zeitliche Dauer eines Musters der Folge) nötig, um eine zeitliche Korrelation und damit eine Sequenz zu erzeugen (Verteilungen *a,b,c*). Die genaue Verteilungsfunktion der Verzögerungen ist unkritisch, da die Korrelation mit dem jeweiligen zeitlichen Signal und damit die "richtige" Verzögerung (die "richtigen" Synapsengewichte) gelernt wird; alle Signale zum "falschen" Zeitpunkt werden nicht korreliert und damit



ivspeicher

nicht beachtet. Musterfolge
auch nicht gele

Abb. 7.12 *Einfluß der Verzögerungszeiten (nach [HER89])*

Interessanterweise konnten Buhmann und Schulten 1987 [BUH87] zeigen, dass auch eine verzögerungs*freie* Aktivität gute Zeitsequenzen ergeben kann. Dazu speicherten sie die Muster nicht nur wie in Gl. (7.21) mit der Korrelation zu sich selbst und zum nachfolgenden Muster ab, sondern stabilisierten die Sequenz noch durch zwei zusätzliche Terme

$$\mathbf{W} = \mathbf{W}^{(1)} + \mathbf{W}^{(2)} + \mathbf{W}^{(3)} + \mathbf{W}^{(4)} \tag{7.31}$$

mit den Abkürzungen

$$\mathbf{W}^{(1)} = \alpha \sum_k \mathbf{x}^k (\mathbf{x}^k)^T \qquad \textit{Eigenkorrelation}$$

$$\mathbf{W}^{(2)} = \beta \sum_k \mathbf{x}^k (\mathbf{x}^{k-1})^T \qquad \textit{Sequenzkorrelation}$$

$$\mathbf{W}^{(3)} = -\gamma \sum_k \mathbf{x}^k (\mathbf{x}^{k+1})^T \qquad \textit{Anti-inverse Sequenzkorr.}$$

$$\mathbf{W}^{(4)} = -\eta \sum_k \sum_{\substack{j \neq k-1, \\ k,k+1}} \mathbf{x}^k (\mathbf{x}^j)^T \qquad \textit{Anti-Kreuzkorrelation}$$

Der dritte Term verhindert den Übergang vom nächsten Muster zurück zum vorhergehenden und der vierte Term vermindert alle Korrelationen, die mit den sonst noch vorhandenene Mustern existieren. Beide Terme erinnern stark an die Anti-Hebb Regel aus Abschnitt (2.1.3), die eine Orthogonalisierung der Gewichtsvektoren bewirkt. Definiert man noch zusätzlich eine wahrscheinlichkeitsbedingte, aktivitätsabhängige Ausgabefunktion gemäß (3.2.9) mit $\Delta E := z_i(t)$, so reicht schon eine leichte Überlagerung der Aktivität mit zufälligen Störungen (Rauschen) aus, die Sequenz hintereinander stabil ablaufen zu lassen.

7.3.2 Sequenzen mit Kontext

Die bisher vorgestellten Verfahren beschränken sich darauf, die Abfolge von jeweils zwei Zeichen sicherzustellen. Dabei können aber alle Sequenzen, in denen das selbe Zeichen zweimal unterschiedlich vorkommt, prinzipiell nicht gespeichert werden. Bei-

spielsweise können zwar N*E*ST und R*E*ST gespeichert werden, nicht aber N*E*IN und R*E*ST. Hier müssen neue Verfahren gefunden werden.

Eine Idee besteht darin, nicht nur die Verbindung (Korrelation) von zwei Zeichen, sondern von mehreren Zeichen zu verwenden. Dies lässt sich beispielsweise durch die Verwendung von Synapsen höherer Ordnung herstellen, bei der mehrere Zeichen miteinander korreliert werden. Ein Beispiel dafür ist die Arbeit von R.Kühn, van Hemmen und Riedel [KÜH89a],[KÜH89b], in der eine Sequenz der Form ABACADAE... generiert werden soll, wobei die einzelnen Symbole A,B,C, .. jeweils selbst wieder eine Sequenz darstellen sollen. Der Übergang von einem Zeichen zum nächsten innerhalb einer Sequenz wird wieder durch gespeicherte Korrelationen der Form nach Gl. (7.21) bewirkt. Für den Übergang von einer Sequenz über das immer gleiche "Zwischenglied" A zur nächsten Sequenz ist allerdings außer der *2-Korrelation* (des letzten Musters $\mathbf{x}\alpha L$ einer Sequenz α mit dem ersten Muster $\mathbf{x}A0$ der Zwischensequenz A) eine höhere *3-Korrelation* zwischen den beiden Sequenzen α und α+1 sowie der Zwischensequenz A nötig. Dazu wird die Aktivität aus Gl.(7.22) um einen dritten Anteil ergänzt

$$z_i(t) = z_i^{(1)}(t) + z_i^{(2)}(t) + z_i^{(3)}(t) \tag{7.32}$$

$$\text{mit } z_i^{(3)}(t) = \sum_j w_{ijk}^{(3)}\ \bar{S}_j(t,\tau_1)\ \bar{S}_k(t,\tau_2)$$

Im dritten Anteil wirken sich zwei Verzögerungszeiten aus: τ_1 bewirkt einen Übergang vom letzten Zeichen A_L der Sequenz A zum ersten Zeichen $(\alpha+1)_0$ der nächsten Sequenz und $\tau_2 > \tau_1$ ermöglicht als Verzögerung eine "Erinnerung" an das letzte Zeichen α_L der vorigen Sequenz α. Damit ist ein eindeutiger Kontext "$\alpha_L A_L$" für das erste Muster $(\alpha+1)_0$ der Folgesequenz da. Die Gewichte dieses Sequenzen-Übergangs werden dabei durch die "Synapsen 2. Ordnung" gebildet

$$w_{ijk}^{(3)} = \gamma/n^2 \sum_\alpha x_i^{(\alpha+1)0}\, x_j^{AL}\, x_k^{\alpha L} \tag{7.33}$$

Die Ausgabefunktion S(z) ist dabei wieder binär symmetrisch S(z) = sgn(z) und die Erwartungswerte der verzögerten Signale $\bar{S}_j(t,\tau)$ sind mit Gleichungen (7.23) und (7.26) gegeben.

Charakteristisch für die Methode, höhere Korrelationen einzusetzen, ist die "lokale", durch die längste Zeitverzögerung beschränkte (s. Abb. 7.12!) Eigenschaft der Signal-Korrelationen, als Kontext zu wirken. Für andere Fragestellungen ist es einfacher, den Kontext dauerhaft für die gesamte Sequenz anzubieten und damit verschiedene Sequenzen mit gleichen Mustern zu ermöglichen. Ist beispielsweise der Kontext in unserer Sequenz "NEIN" mit "#" gegeben und in der Sequenz "REST" mit "*", so folgt auf den Sequenzanfang #(NE) eindeutig "I" und auf *(RE) "S", ohne dass es eine Verwechslungsmöglichkeit geben kann. In Abbildung 5.3.4 ist ein solches System in der Übersicht zu sehen, wie es von Kohonen [KOH84] vorgeschlagen wurde.

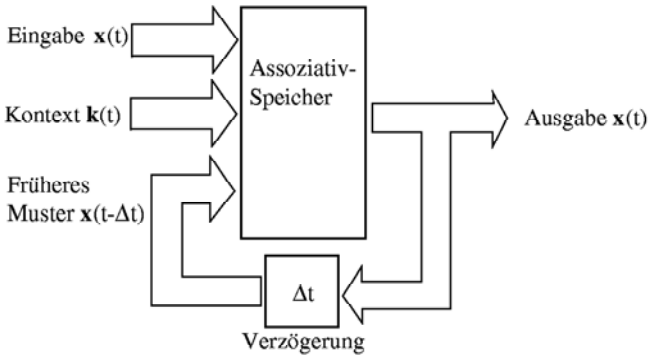


Abb. 7.13 *Kontextsensitive Sequenzspeicherung und -generierung*

Werden an Stelle *einer* Verzögerung *mehrere* angeboten, so lassen sich über diese zusätzlichen, zeitlichen Kontexte auch Muster in der selben Sequenz wiederholt verwenden, wie z.B. in der Sequenz "KELLER", ohne höhere Synapsen einzusetzen.

8 Fuzzy Systeme

Ende der achziger Jahre etablierte sich eine neue Denkschule in den Ingenieurswissenschaften: Viele Probleme können (Mangels exaktem Wissen) nicht exakt beschrieben werden, sondern nur mit vagen, ungenauen Begriffen. Man brauchte also eine Rechenmethode, mit der man auch mit ungenauen Sensorwerten gute Ergebnisse für Regelungen und Steuerungen erhielt. Dazu kamen die Arbeiten von L.A. Zadeh gerade richtig, der die exakte Logik bereits seit 1965 auf eine ungenaue, vage Logik verallgemeinerte.

8.1 Einführung

Die mittlerweile klassischen, symbolischen Systeme der künstlichen Intelligenz (KI) haben einen großen Nachteil: sie versuchen, die Realität mit einem System von nur zwei Zuständen zu beschreiben. Dies ist Ursache für manches klassisches Paradoxon. Beispielsweise ist der Satz "Dieser Satz ist unwahr" ein Paradoxon: ist er wahr, so bedeutet dies, dass ein unwahr ist. Ist er aber unwahr, so ist er wahr. In diese Kategorie von Paradoxen gehören auch die klassischen Beispiele vom Kreter, der sagt, dass alle Kreter lügen (lügt auch er?) oder vom Dorfbarbier, der alle im Dorf rasiert, die sich nicht selbst rasieren (Wer rasiert den Barbier?).

8.1.1 Fuzzy-Variable

Dieses Paradoxon lässt sich mit den Wahrheitswerten {0,1} anstelle von {`FALSCH`, `WAHR`} folgendermaßen darstellen:

Mit "Satz s ist wahr"≅ (s = 1) und "s ist unwahr"≅ (s = 0) gilt für die obigen widersprüchlichen Sätze die Folgerungen

$$\left.\begin{array}{lll} s = 1 & \Rightarrow & s = 0 \\ s = 0 & \Rightarrow & s = 1 \end{array}\right\} \Rightarrow s = 1 - s \qquad (8.1)$$

Die obige Schlussfolgerung bedeutet in dem zweiwertigen Logiksystem einen Widerspruch: eine Aussage kann nicht gleichzeitig wahr und falsch sein. Erweitern wir dagegen die Zahl der möglichen Logikwerte, beispielsweise auf alle reellen Zahlen aus dem Intervall [0,1], so ist Aussage (8.1) gleichbedeutend mit 2s = 1 oder s = 0,5. Der Aussagewert liegt in der Mitte zwischen den Extremen 0 und 1 und ist "unscharf". Diese Form von Logik wird deshalb "unscharfe Logik" oder *Fuzzy Logik* genannt.

Es gibt noch eine zweite Situation, in der die binäre Logik zu Problemen führt: die Charakterisierung eines Systems, beispielsweise eines Modells der realen Welt, durch reellwertige Variable. Eine nur geringe Veränderung der Situation (z.B. $\{x = x_1\} \rightarrow \{x = x_2\}$) wird entweder durch die unveränderten, logischen Fakten nicht ausreichend reflektiert ($\{x = x_1\}$ bleibt `WAHR`) oder aber verlangt die Definition mindestens einer neuen Variablen ($\{x = x_2\}$ wird `WAHR`). Die Modellierung eines Vorgangs, beispielsweise der Übergang von einem Zustand {Sandhaufen = `WAHR`} zu einem anderen {Sandhaufen = `FALSCH`} durch langsame Zustandsänderung (Abtragen von einzelnen Körnchen) kann dadurch logisch sehr problematisch werden .

Die Probleme der binären Logik lassen sich beseitigen, indem man anstelle der scharfen Zuordnung reeller Variablen (Zuständen) zu logischen Variablen als "unscharfe" Zuordnung eine kontinuierliche Zugehörigkeitsfunktion $m_A(x)$ eines logischen Attributs A von einer Größe x einführt. Diese Funktion ist also nicht auf die Werte null oder eins beschränkt; sie wird so definiert, dass sie bei x_A eins ist und mit zunehmendem Abstand der Variablen x vom Wert x_A geringer wird. Mit dieser Überlegung wird eine "Ähnlichkeit" zwischen x und x_A ausgedrückt. In Abb. 8.1 ist eine solche Funktion visualisiert.

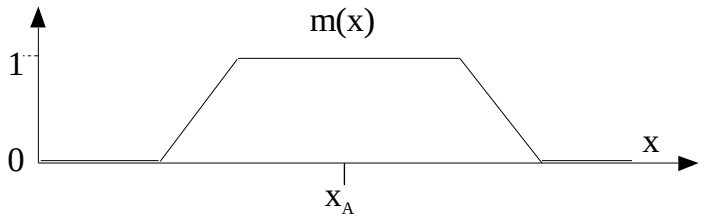


Abb. 8.1 *Zugehörigkeitsfunktionen für die Variable x*

Üblicherweise haben die Zugehörigkeitsfunktionen meist eine Dreiecks- oder Trapezform, da dies einfacher numerisch ausgewertet werden kann als Funktionen wie etwa die Gaußfunktion; prinzipiell ist aber jede Form denkbar, die zuerst monoton zu- und dann abnimmt. Damit lassen sie sich als *Glockenfunktionen* beschreiben wie sie vorher definiert wurden.

8.1.2 Interpretationen und Inferenzen

Ähnlich wie in der formalen Logik kann man auch in der Fuzzy Logik mit Hilfe von Regeln Schlussfolgerungen aus vorliegenden Fakten ziehen. Betrachten wir als Beispiel die Definition für "schönes Wetter". Machen wir diese Definition von den zwei Variablen x_1 = "Helligkeit" und x_2 = "Feuchte" abhängig, so lassen sich x_1 in "dunkel", "normal", "sehr hell" und x_2 in "nass" und "trocken" unterteilen. In Abb. 8.2 sind für die Variable x_i die Zugehörigkeitsfunktionen m_{ij} dazu gezigt, sowie ein diskreter Wetterzustand $\mathbf{b} = (b_1,b_2)$ eingezeichnet.

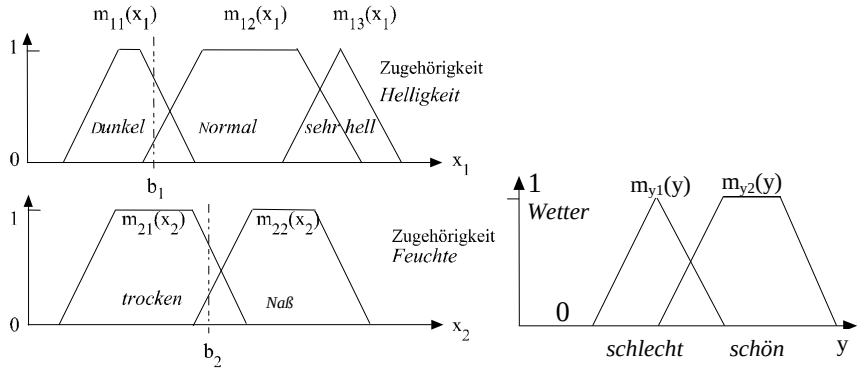


Abb. 8.2 *Zugehörigkeitsfunktionen fürs Wetter*

Im Unterschied zu den scharfen, diskreten Regeln der Logik lassen sich die Folgerungen aus den Fakten, in unserem Beispiel ob das Wetter „schön“ oder „schlecht“ ist, mehr als *Prinzipien* interpretieren, die meist, aber nicht immer gelten. Beispielsweise können wir uns aus unserer Alltagserfahrung die folgenden Prinzipien fürs Wetter formulieren:

`WENN` (Helligkeit = normal) `UND` (Feuchte = trocken) `DANN` (schönes Wetter)
`WENN` (Helligkeit = dunkel) `UND` (Feuchte = naß) `DANN` (schlechtes Wetter)

Dieses Prinzipien lauten dann für unseren konkreten Wetterzustand $\mathbf{b} = (b_1,b_2)$

`WENN` $(m_{12}(x_1=b_1))$ `UND` $(m_{21}(x_2=b_2))$ `DANN` (schönes Wetter)
`WENN` $(m_{11}(x_1=b_1))$ `UND` $(m_{22}(x_2=b_2))$ `DANN` (schlechtes Wetter)

Allerdings ist nicht immer ein logisches Attribut („schlecht“, „schön“) gefragt, sondern auch ein konkreter numerischer Wert der resultierenden Variablen. In Abb. 8.2 erhalten wir für den Wetterzustand (b_1,b_2) numerische Zugehörigkeitswerte von logischen Attri-

buten „dunkel“, „normal“, „sehr hell“ usw. Wie erhalten wir nun den gewünschten numerischen Wert des Resultats *y*?

Bei der Auswertung der Fuzzy-Regeln unterscheiden wir zwei Operationen: zum einen die Auswertung der UND-Terme einer Regel und zum anderen die Auswertung der Ergebnisse aller, parallel geltenden Regeln. Betrachten wir zunächst den ersten Fall.

8.1.3 Auswertung der Terme einer Regel

Jede Regel *i* mit *n* durch "UND" verbundenen Termen definiert eine Abbildung $y_i = S_i(x_1..x_n)$. Wie ergibt sich die Abbildung aus den Einzeltermen? Betrachten wir dazu eine 2-dim. Punktmenge {**x**}, die in der Fläche illustriert werden kann. Über die 2-dim. Funktion $S(x_1,x_2)$ wissen wir nur, dass ihre 1-dim. Projektionen, die Zugehörigkeitsfunktionen $m_1(x_1)$ und $m_2(x_2)$, Trapezform haben. Zeichnen wir die Funktionen $m_1(x_1)$ und $m_2(x_2)$ jeweils um 90 Grad gedreht zueinander in eine Ebene, so erhalten wir Abb. 8.3. Die dunkel schraffierte Fläche in der Mitte des Vierecks in Abb. 8.3 enthält alle Punkte $\mathbf{x} = (x_1,x_2)$, für die $\{m_1(x_1) = 1$ und $m_2(x_2) = 1\}$ gilt. Sie ist die Überlagerung von den zwei hell schraffierten Streifen mit jeweils $\{m_1(x_1) = 1\}$ bzw. $\{m_2(x_2) = 1\}$. In dem dunkel schraffierten Kerngebiet (*core region*) ist deshalb $S(\mathbf{x}) = 1$; innerhalb des gesamten Vierecks (*support region*) gilt $S(\mathbf{x}) > 0$.

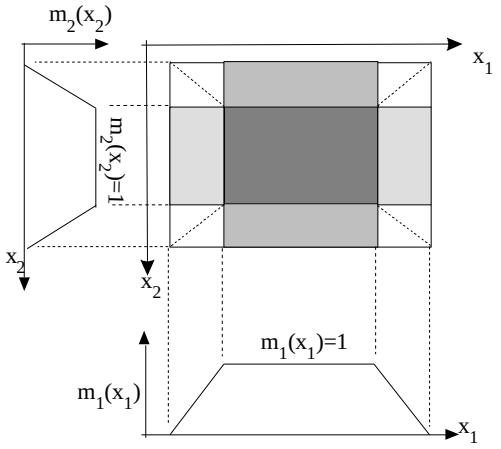


Abb. 8.3 Einzelterme und Funktionswert einer Regel

Welchen Wert hat S(**x**) in den hell schraffierten Gebieten? In den hell schraffierten horizontalen Gebieten mit $m_1(x_1) = 1$ steigt bzw. fällt $m_2(x_2)$. Hier sollte S(**x**) ebenfalls zunehmen bzw. abnehmen, um für $m_2 = 1$ den Wert $S(x) = 1$ bzw. bei $m_2 = 0$ den Wert $S(x) = 0$ anzunehmen und damit konsistent mit der binären Logik ($S = m_1$ UND m_2) zu bleiben. Das Analoge gilt für die Gebiete mit $m_2(x_2) = 1$ und ansteigendem bzw. abfallendem $m_1(x_1)$. Damit lässt sich in den schraffierten Gebieten beispielsweise $S(x_1,x_2) = \mathit{min}\ (m_1(x_1), m_2(x_2))$ definieren. Für die restlichen vier Gebiete in den vier Ecken lässt sich dies kontinuierlich ergänzen, so dass die gestrichelten Diagonalen die Gleichheit der Terme $m_2(x_2) = m_1(x_1)$ bedeutet.

Verallgemeinern wir die Auswertung der UND-Terme auf alle *n* Dimensionen, muss also in Fuzzy-Systemen bei der Auswertung der Prinzipien von den mit "UND" verbundenen Fakten der numerisch kleinste Term gewählt und als Funktionswert S_i(**x**) des *i*-ten Prinzips (der *i*-ten Regel) betrachtet werden. In unserem Wetterbeispiel ist somit $S_1(b_1,b_2) = m_{12}(b_1)$ und $S_2(b_1,b_2) = m_{22}(b_2)$.

Die Minimumsbildung ist eine weitgehend durch den mathematischen Unterbau der Fuzzy-Logik motivierte Operation. Alternativ dazu erhalten wir in den hell schraffierten Gebieten die gleiche Funktion $S(x_1,x_2)$, wenn wir sie statt als Minimum $\mathit{min}(m_1(x_1), m_2(x_2))$ als Produkt beider Funktionen $m_1(x_1)\cdot m_2(x_2)$ definieren. In den vier Ecken sind die beiden Funktionen „Minimum“ und „Produkt“ zwar nicht identisch, aber da wir frei in der Definition der Zugehörigkeitsfunktion S(**x**) sind, ist dies nicht wesentlich. Im Produktfall ist die Funktion S(**x**) nicht nur eine Glockenfunktion, sondern hat auch die Proportion der multi-dimensionalen Gaußfunktion $S_G(\mathbf{x}) = S_G(x_1)\cdot \ldots \cdot S_G(x_n)$.

Wieviele Regeln gibt es in einem solchen System? In Abb. 8.4 sind die beiden Variablen mit ihren Zugehörigkeitsfunktionen aus Abb. 8.2 am Rand aufgetragen. In der Mitte im zweidimensionalen Musterraum sind alle denkbaren Regeln mit ihren Kerngebieten zu sehen; sie decken den Musterraum überlappend ab.

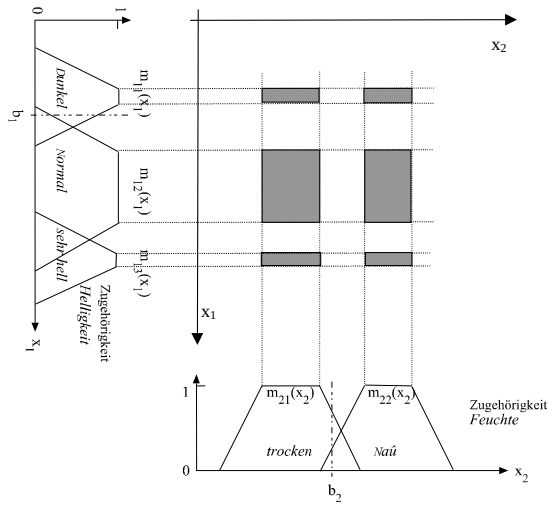


Abb. 8.4 Die Kernmengen (schraffiert) aller Regeln im 2-dim Musterraum

Bei drei Kategorien für Helligkeit und zwei für Feuchte gibt es also 2·3 = 6 mögliche Kombinationen von Helligkeit und Feuchte. Allgemein bei r_j definierten Zugehörigkeitsfunktionen pro Variable ist die maximale Anzahl N der damit bildbaren Regeln

$$N = \prod_{j=1}^{n} r_j \tag{8.2}$$

8.1.4 Auswertung aller Regeln

Betrachten wir nun die Auswertung aller N parallel gültigen Prinzipien. Bei Fuzzy-Variablen ist keines der "unscharfen" Prinzipien absolut falsch, sondern wird nur unterschiedlich bewertet. Wie lässt sich aus mehreren Prinzipien (Überlagerung mehrerer Fuzzy Mengen) die tatsächliche Ausgabe y berechnen?

Für den Prozeß der *Defuzzifikation* wird zunächst aus dem ermittelten Funktionswert S_i der Fuzzy-Regel und der j-ten Zugehörigkeitsfunktion $m_{yj}(y)$ für die Ausgabevariable y (hier: j = 1:"schönes Wetter", j = 2:"schlechtes Wetter") eine neue Zugehörigkeitsfunktion $M_{ij}(y)$ gebildet. Die Überlagerung aller neuen $M_{ij}(y)$ ergibt für eine Funktion $M_y(y)$, deren Erwartungswert (Schwerpunkt) schließlich der gesuchte Ergebniswert y ist.

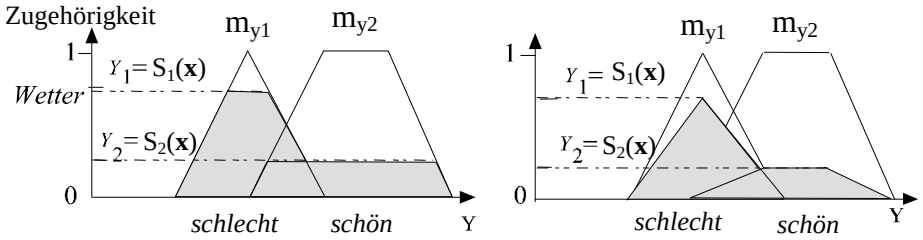
Betrachten wir dies nun im einzelnen. Für die Bildung der neuen $M_{ij}(y)$ gibt es verschiedene Methoden. Die populärste, die *Korrelations-Minimum-Kodierung*, beschränkt die Ausgabefunktion auf alle Werte kleiner als S_i bzw. m_{yi}

$$M_{ij}(y) = \min (S_i, m_{yj}(y)) \quad \textit{correlation minimum encoding} \tag{8.3}$$

Dies entspricht ein „Auffüllen“ der Funktion $m_{yj}(y)$ bis zum Wert S_i, siehe Abb. 8.5a). Eine andere Methode, die *Korrelations-Produkt-Kodierung*, besteht darin, als Ausgabefunktion $M_{ij}(y)$ die mit S_i gewichtete Zugehörigkeitsfunktion m_{yi} der Fuzzy Menge zu verwenden

$$M_{ij}(y) = S_i \cdot m_{yj}(y) \quad \textit{correlation product encoding} \tag{8.4}$$

Dies bedeutet eine Verkleinerung der Funktion m_{yj} mit dem Faktor S_i, siehe Abb. 8.5b.



a) correlation minimum encoding b) correlation product encoding

Abb. 8.5 *Defuzzifikation der Ausgabe*

Bildet man bei jedem Wert von y das Maximum aller Funktionswerte $M_{ij}(y)$ von allen Regeln *i* und Zugehörigkeitsfunktionen *j*, so erhält man eine Funktion M(y). Sie lässt sich als Gewichtungsfunktion bzw. nicht-normierte Wahrscheinlichkeitsdichte der Ausgabevariablen y interpretieren. Als diskreten Ausgabewert $y = S(\mathbf{x})$ wählen wir uns den Erwartungswert der Ausgabevariablen bei dem aktuellen Fuzzy-Kontext.

$$S(\mathbf{x}) = \langle y \rangle = \int p(y)\, y\, dy \tag{8.5}$$

Der Erwartungswert $\langle y \rangle$ wird über die Gewichtung mit der auf eins normierten Auftrittsdichte p(y) gebildet

$$p(y) = \frac{M(y)}{\int M(y)\, dy} \tag{8.6}$$

und erhalten so die Ausgabe

$$S(\mathbf{x}) = \frac{\int M(y)\, y\, dy}{\int M(y) dy} \tag{8.7}$$

Der Erwartungswert der Ausgabevariablen ist identisch mit dem y-Anteil des Schwerpunkts aus den schraffierten Flächen unter M(y), s. Abb. 8.5a). Man sieht, dass in unserem Beispiel beide Zugehörigkeitsfunktionen für schönes und für schlechtes Wetter bis zu den Funktionswerten S_i "aufgefüllt" wurden und dann aus der Überlagerung beider (in Abb. 8.5 schraffiert dargestellten) Fuzzy-Mengen die gemeinsame Zugehörigkeitsfunktion $M(y) = max(m_{y1}, m_{y2}) \approx M_1 + M_2$ bei geringer Überlappung gebildet und davon der gewichtete Mittelwert (Schwerpunkt bzw. Zentroid der schraffierten Flächen) errechnet wird.

Die Defuzzifizierung können wir auch noch weiter umformen und vereinfachen. Zunächst ist mit (8.7)

$$S(\mathbf{x}) = \langle y \rangle = \frac{\int M(y)\, y\, dy}{\int M(y) dy} \approx \frac{\sum_j \int M_{ij}(y)\, y\, dy}{\sum_l \int M_{kl}(y) dy} \tag{8.8}$$

approximiert, wobei die Approximation umso genauer ist, je weniger die Zugehörigkeitsfunktionen sich überlappen.
Bezeichnen wir der Einfachheit halber das Maximum $max_i(S_i)$ aller Regelwerte für den j-ten Ausgabebereich mit S_j, so ist mit Gl.(8.4)

$$S(\mathbf{x}) \approx \frac{\sum_j \int M_j(y)\, y\, dy}{\sum_k \int M_k(y) dy} = \frac{\sum_j S_j \int m_{yj}(y)\, y\, dy}{\sum_k S_k \int m_{yk}(y) dy} \tag{8.9}$$

Mit den Definitionen für die Fläche F_i (Gewichtungsmasse) unter den Zugehörigkeitsfunktionen m_{yi} und den Schwerpunkt w_i der einzelnen Zugehörigkeitsfunktionen (dem Mittelpunkt bei symmetrischen Dreiecken)

$$F_i = \int m_{yi}(y)\, dy, \qquad w_i = \frac{\int m_{yi}(y)\, y\, dy}{\int m_{yi}(y) dy} \Rightarrow \int m_{yi}(y)\, y\, dy = w_i F_i \tag{8.10}$$

erhalten wir die Ausgabe $S(\mathbf{x})$ in der Form

$$S(\mathbf{x}) \approx \frac{\sum_j S_j w_j F_j}{\sum_k S_k F_k} \tag{8.11}$$

Da w_j und F_j konstant sind, lässt sich das Produkt $S_j(\mathbf{x})F_j$ als Variable y_i ansehen, die in L_1-normierter Form

$$y_i = \frac{S_i F_i}{\sum_k S_k F_k} \tag{8.12}$$

bei der Defuzzifikation mit den Gewichten w_i bewertet wird

$$f(\mathbf{x}) = \sum_i w_i \, y_i \tag{8.13}$$

Diese Formen kommen uns sehr bekannt vor: sie entsprechen dem Mechanismus der zweischichtigen Approximationsnetze mit radialen Basisfunktionen aus Abschnitt 4.6. Das Fuzzy System mit Korrelations-Produkt Kodierung kann also als zweischichtiges neuronales Netz mit speziellen Radialen Basisfunktionen, die "Prinzipien" oder "Fuzzy-Regeln" genannt werden, betrachtet werden.
In der folgenden Abb. 8.6 ist dies schematisch gezeichnet.

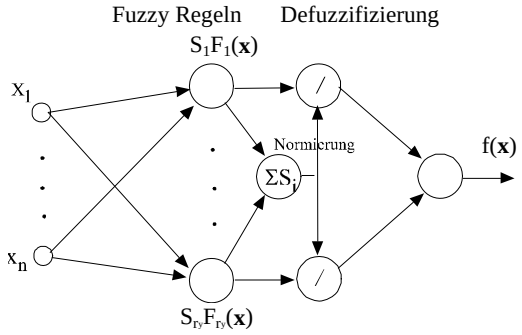


Abb. 8.6 *Fuzzy–Regeln als neuronales Netz*

8.2 Adaptive Fuzzy Systeme

Ein großes Problem der Fuzzy Systeme besteht darin, dass die Prinzipien, also Existenz, Form und Lage der Zugehörigkeitsfunktionen, von einem menschlichen Experten erzeugt werden müssen. Dieses Wissen ist aber meist nicht exakt; die Parameter müssen erst an die Wirklichkeit angepasst werden. Da die Fuzzy-Systeme nur Spezialfälle neuronaler Netze sind, lassen sich hier auch die normalen, uns bekannten Lernalgorithmen neuronaler Netze einsetzen.

Dazu machen wir uns die Ähnlichkeit von Gl.(8.12) und Gl.(8.13) mit den Radialen Basisfunktionen zu Nutze. Jede Regel wird mit einem RBF-Neuron identifiziert, indem

das *correlation encoding* verwendet wird: die Ausgabefunktion ist das Produkt aller beteiligten eindimensionalen Zugehörigkeitsfunktionen $m_r(x_1),\ldots,m_k(x_n)$

$$S(x_1,...,x_n) = m_r(x_1)\cdot\ldots\cdot m_k(x_n)$$

$$S_{RBF}(x_1,...,x_n,y_i) = S(x_1,...,x_n)\cdot m_i(y)$$

$$f(x_1,...,x_n) = \sum_i w_i\, S_{RBF}(x_1,...,x_n,y_i) \qquad \textit{Schwerpunktsgewichtung} \tag{8.14}$$

Jeder Term einer Regel entspricht einer eindimensionalen Glockenfunktion; alle Terme multiplikativ zusammen ergeben die Ausgabefunktion des RBF-Neurons, wobei auch die Zugehörigkeitsfunktion $m_i(y)$ der WENN..DANN-Schlussfolgerung enthalten ist.

Das Gesamtnetz ergibt sich aus der Zusammenschaltung aller Fuzzy-Regeln (RBF-Neuronen) zu einem zweischichtigen Netz. Die zweite Schicht entspricht der Schwerpunktbildung im Fuzzy-System.

Auf ein solches Netz können wir nun alle uns bekannten RBF-Lernalgorithmen anwenden. Hat das Netz befriedigend gelernt, so lassen sich danach die so erhaltenen RBF-Neuronenparameter (Zentren und Breiten der RBF-Ausgabefunktionen) wieder zurückprojizieren auf den Raum der eindimensionalen Fuzzy-Zugehörigkeitsfunktionen und die so in der Lage und Breite veränderten Zugehörigkeitsfunktionen betrachten.

Was ist der Vorteil eines solchen Fuzzy-Neuro-Ansatzes gegenüber einem reinen Ansatz mit neuronalen Netzen? Warum nimmt man nicht gleich ein neuronales Netz anstatt zuerst Fuzzy-Proportionen zu definieren und dann doch wieder ein äquivalentes RBF-Netz zu trainieren? Die Antwort darauf liegt in der relativ einfachen Initialisierung durch die Fuzzy-Notation der Netzparameter bei der Benutzung durch Menschen.

Beispiel *Medizindiagnostik*

In der Medizin werden üblicherweise Diagnosen und Beurteilungen nicht in exakten Zahlen angegeben, sondern in vagen Begriffen wie "leicht erhöhter Blutzucker". Möchte man diese Angaben etwas genauer fassen um sie mit Rechnern zu verwerten, so ist es sinnvoll, die vage, umgangssprachliche Bezeichnung passend auf einen logischen Mechanismus abzubilden. Hier ist die Fuzzy-Logik sehr willkommen. Mit Hilfe der Zugehörigkeitsfunktionen kann man sehr unscharfe, vage Einsachätzungen handhabbar machen. In Abb. 8.7 ist eine solche Notation, die entsprechende symbolische Noation mit -,+,++ usw. sowie die exakten Intervallzahlen für Blutzucker aufgeführt.

Medizinische Notation

semi-quant. Notation	Werteintervall [g/dl]	ling. Notation
–	0 – 0.1	niedrig
	0.1 – 0.2	normal
+	0.2 – 0.3	leicht erhöht
++	0.3 – 0.45	erhöht
+++	0.45 – 2	stark erhöht
++++	> 2	sehr stark erhöht

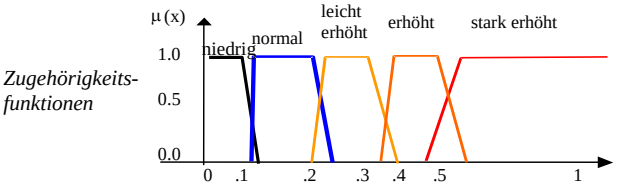


Abb. 8.7 *Medizinische Notation und Fuzzy–Zugehörigkeit bei Blutzucker*

Diese Form der initialen Eingabe von Wissen gestattet es nicht nur, einen guten Startpunkt für den adaptiven Algorthmus des RBF-Netzes zu wählen, sondern auch umgekehrt, die Ergebnisse des Trainings wieder in der Notation der Mediziner darzustellen und damit begreifbar und verständlich zu machen. In Abb. 8.7 ist das Schema eines solchen Austauschs gezeigt.

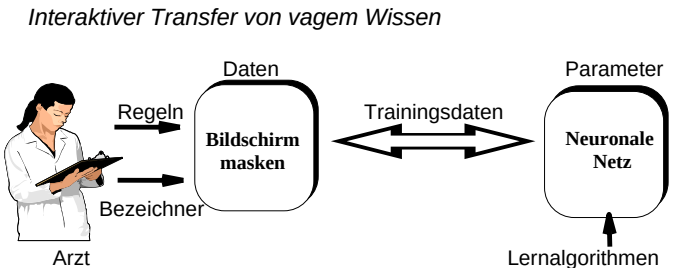


Abb. 8.8 *Interaktion zwischen Nutzer und Diagnosemaschine*

Die Vorteile liegen auf der Hand:

- Keine exakte Aussage über Art der Zugehörigkeitsfunktion nötig (und möglich!)
- Direkte Initialisierung *verschiedenartigsterer* RBF-Netze

Allerdings hat auch dieser Ansatz Probleme: Neue, algorithmenbedingte RBF-Neurone und stark abweichende Kategorien benötigen neue, synthetische Bezeichnungen.

Mit dem Ansatz, den Fuzzy-Regeln äquivalente neuronale Netze zu konstruieren, die Netzgewichte mit bekannten Lernalgorithmen anzupassen und danach wieder die korrespondierenden, veränderten Fuzzy-Prinzipien zu konstruieren, erhalten die Fuzzy-Systeme eine interessante Anwendung für die Konstruktion neuronaler Netze: Sie dienen als einfache, dem menschlichen Benutzer gemäße, verständliche Schnittstelle, um bereits bekanntes Wissen zur Initialisierung der neuronalen Netze verwenden zu können. Ihr Formalismus von Prinzipien und Fakten ist somit nur dann sinnvoll, solange der menschliche Benutzer diese Art von Schnittstelle als "einfach" und "problemgemäß" empfindet. Ist dieses für einen Problemkreis nicht sinnvoll, so muss man sich andere Mechanismen für den Wissenstransfer zur Initialisierung überlegen.

8.3 Fuzzy-Kontrolle

Eine der wichtigsten Anwendungen der Fuzzy Regeln liegt bei Kontroll- und Regelungsaufgaben. Das Grundschema einer solchen Funktion soll in diesem Abschnitt kurz skizziert werden. Eine weitergehender Überblick ist beispielsweise in [BUCK92] zu finden.

Ein wichtiges Grundschema der Regelung ist in Abb. 8.9 skizziert. Ausgehend von einem Sollzustand vergleicht der Regler den durch einen Sensor am System erfühlten Ist-Zustand und errechnet sich einen geeignet korrigierten Stellwert für den Effektor, der das System beeinflusst und die Differenz zwischen Soll- und Istwert reduziert.
Ein Beispiel für ein solches Regelsystem ist die Temperaturregelung eines industriellen Schmelzprozesses.

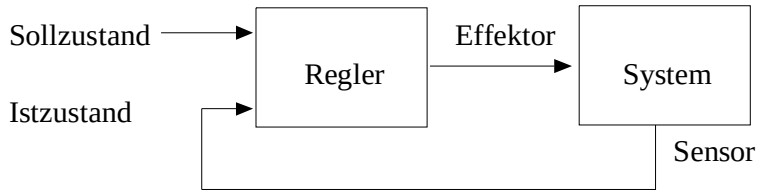


Abb. 8.9 *Ein einfacher Regelkreis*

Erwarten wir, dass der Prozeß bei einer Temperatur von 100°C stattfindet mit 50% der maximalen Heizleistung, so könnten die Zugehörigkeitsfunktionen einer Fuzzy Regelung aussehen wie Abb. 8.10.

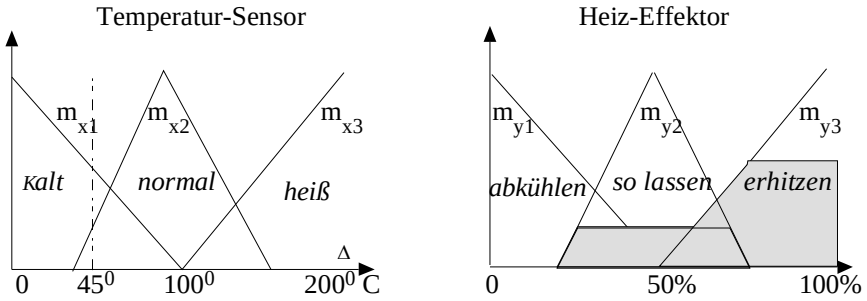


Abb. 8.10 *Temperaturbereiche und Heizbereiche*

Im linken Diagramm sind die drei Bereiche "kalt", "normal" und "heiß" der gemessenen Temperatur (*Sensorik*) gezeigt, in der Rechten Abbildung die drei Aktionsbereiche "abkühlen", "so_lassen" und "erhitzen", die durch Einstellen der Heizleistung (*Effektor*) bewirkt werden können.

Der eigentliche Regler ist beispielsweise durch folgende Prinzipien definiert:

`WENN` (Temperatur = heiß) `DANN` abkühlen (8.15)

`WENN` (Temperatur = normal) `DANN` so_lassen (8.16)

`WENN` (Temperatur = kalt) `DANN` erhitzen (8.17)

Für die Funktion der Regelung betrachten wir den Fall, dass eine zu niedrige Temperatur von 45°C gemessen wurde. Dann sei m_{x1}= 0,45 und m_{x2} = 0,29. Dann ergeben die Prinzipien (8.15),(8.16),(8.17) die Werte $y_1 = 0$, $y_2 = m_{x2}$ und $y_3=m_{x1}$. Mit der Korrelation-Minimum Methode wird als Ergebnis der Schwerpunkt der schraffierten Bereiche in Abb. 8.10 gebildet, was zu einer erhöhten Heizleistung führt und die Temperatur ansteigen lässt..

Aufgabe

Man errechne für das den Regelkreis in Abschnitt 6.4 die Heizleistung y aus, wenn eine Temperatur von 120°C gemessen wurde. Dazu ermittle man y_1,y_2,y_3 der Regeln (8.15),(8.16),(8.17), errechne sich Schwerpunkte c_i und Flächen F_i der Funktionen in Abb. 8.10 und bilde dann $\langle y \rangle$ mit der Korrelations-Produkt Kodierung nach Formel Gl.(8.13).

9 Evolutionäre und genetische Algorithmen

Eine der wichtigsten Aufgaben von neuronalen Netzen besteht darin, durch ihre Architektur und die Wahl der Gewichte eine vorgegebene Aufgabe so gut wie möglich zu erledigen. Die Gewichte werden dabei meist mittels einer Zielfunktion, die auf die Aufgabenstellung zugeschnitten ist, optimiert („gelernt“). Allgemein entspricht dies der Situation, eine vorgegebene Zielfunktion zu optimieren; neuronale Algorithmen lassen sich deshalb als eine der Gruppen in der Vielzahl von *Algorithmen für Optimierungsaufgaben* ansehen. Typisch für ein Optimierungsproblem ist, dass eine Anzahl von Parametern $(g_1,...,g_n) = \mathbf{g}$ existieren, deren günstige Wahl die Zielfunktion $R(\mathbf{g})$ optimiert und damit eine Lösung für das Problem darstellt. Wie finden wir nun eine solche Lösung, wenn der Wert von $R(\mathbf{g})$ für ein konkretes $\mathbf{g}$ zwar bestimmt werden kann, die Zielfunktion R(.) aber selbst nicht analytisch explizit bekannt ist ? Wie entkommen wir unter solchen Voraussetzungen der Gefahr, bei vielen lokalen Optima der Zielfunktion das globale Optimum zu übersehen? Die bisher betrachtete Gradientenmethode bietet dafür keine Lösung.

Eine der ungewöhnlichsten Methoden, Optimierungsaufgaben zu lösen, besteht in der Nachahmung biologischer Optimierungsmethoden. Dies sind beispielsweise Algorithmen, die den erfolgreichen Funktionsprinzipien der Jahrmillionenalten, biologischen Evolution nachempfunden sind. Die Grundidee dieser *evolutionären* oder *genetischen Algorithmen* besteht darin, die Parameter $\mathbf{g}$ als Bauplan (Gene) eines Lebewesens zu betrachten. In der freien Natur müssen sich die Gene („Genotyp“) der Lebewesen derart anpassen, dass Aussehen, Eigenschaften und Ausprägungen („Phänotyp“) des Lebewesens möglichst gutes Überleben sichert, ohne die Funktion $R(\mathbf{g})$ („Fitness“) explizit zu kennen. Die evolutionären Algorithmen bieten damit eine Antwort auf unsere Frage. Genauso, wie in der freien Natur die Lebewesen sich durch Veränderungen der Gene den gegebenen Umweltbedingungen anpassen müssen oder untergehen, sollen die Parametertupel $\mathbf{g}$ Veränderungen unterworfen werden und nur die besten Lösungstupel akzeptiert und als "Saat" für noch bessere verwendet werden.

Die evolutionären Algorithmen versuchen also, ein Zahlentupel zu finden, mit dem eine vorgegebene Zielfunktion $R(\mathbf{g})$ maximiert (minimiert) wird. Dabei unterscheiden sich die evolutionären Algorithmen hauptsächlich in der Zahl der Individuen, die sich fortpflanzen oder "sterben", sowie in der Art und Weise, wie die Veränderungen an

ihrem Bauplan vorgenommen werden und wie ihre Überlebensfähigkeit beschrieben wird.

Identifizieren wir die Gene mit unseren Gewichten in einem neuronalen Netz, so bietet dieser Ansatz die Möglichkeit, die Gewichte **g** ohne Gradientenverfahren nur auf Basis der beobachteten Leistung R(**g**) des Netzes zu bestimmen. Darüber hinaus bieten evolutionäre Algorithmen die Möglichkeit, die neuronalen Netze selbst zu optimieren: Neben der Optimierung (Lernen) der Gewichte in neuronalen Netzen haben wir hier sogar den Mechanismus, auch die Architektur und Struktur der Netze selbst zu lernen.

Voraussetzung für diese Art von Algorithmen ist die Repräsentation des Problems durch ein Tupel von Elementen **g** und die Bewertung dieses Tupels mit einer Gütefunktion R(**g**). Gibt es solch eine Tupeldarstellung oder solch eine objektive Gütefunktion nicht, so lassen sich die evolutionären Methoden für das betreffende Problem auch nicht verwenden.

Bei der Anwendung der evolutionären Methoden lassen sich mehrere Strategien verfolgen: die evolutionäre, durch Mutation und Selektion gesteuerte Entwicklung eines einzigen Lebewesens (einer Lösung) und die parallele Entwicklung mehrerer Lebewesen (Lösungen). Betrachten wir zunächst den einfachen Fall.

9.1 Verbesserung einer Lösung: Die Mutations-Selektions-Strategie

Eines der einfachsten Verfahren der evolutionären Optimierung wurde von Rechenberg [RE73] "Mutations-Selektions-Strategie" genannt (*creeping random search method*), weiterentwickelt und der Nutzen an verschiedenen, praktischen Problemen demonstriert. Die Methode besteht darin, ein einziges Tupel solange zufällig zu verändern, bis es eine "genügend gute" Lösung des gegebenen Problems darstellt oder die Veränderungen keine Verbesserung mehr bringen.

```
MuSe:    (* Suche das Maximum einer Funktion R(g) *)
         Kodiere das Problem im Tupel g = (g1, ..., gn)
         REPEAT   g' := Mutation(g )      (* Erzeuge neues, verändertes Tupel *)
                  IF   R(g')>R(g)          (* Wenn es besser ist ..  *)
                    THEN g:= g' ENDIF      (* .. übernimm es.  *)
         UNTIL Abbruchkriterium
```

Codebeispiel 9-1 *Die Mutations-Selektions-Strategie*

Die Veränderungsoperation `Mutation(g)` selbst sollte, so ermittelte Rechenberg, nicht zu groß sein, um ein "Überschießen" über das Ziel hinaus zu vermeiden, und nicht zu klein sein, um die Gefahr zu vermeiden anstelle in einem globalen Optimum nur in einem lokalen Optimum festzusitzen. Als günstigste Veränderung postulierte er als

neues Tupel **g'** einen *n*-dimensionalen Punkt aus einer *n*-dimensionalen Binomialverteilung bzw. Gauß-Verteilung ("Punktwolke") um **g** herum.

Für die Anwendung des Verfahrens kommen besonders diejenigen Probleme in Betracht, für die analytische Lösungen nur sehr schwer oder gar nicht gefunden werden konnten. Beispielsweise wurde das beste Querschnittsprofil einer Zweiphasendüse (flüssig/gasförmiger Zustand in der Düse gleichzeitig vorhanden) für chemische Reaktionen ermittelt. Dazu wurde eine solche Düse aus einer Reihe von $n=9$ Scheiben mit verschiedenem, zentrierten Lochdurchmesser aufgebaut, die in eine Reihe gebracht die gesamte Düse ergeben. Das Tupel $\mathbf{g} = (g_1,...,g_9)$ bestand damit aus einem geordneten Zahlentupel aller 9 Lochdurchmesser; der Funktionswert R(**g**) für eine diskrete Scheibenkombination wurde durch experimentelle Untersuchung der Effizienz (chemische Reaktionsausbeute) nach dem Durchgang der Substanz durch das metallische Gebilde ermittelt [Re73]. Mit Hilfe eines Computers wurde dann ein neues Tupel generiert, die entsprechenden Scheiben eingebaut und die so konstruierte neue Düse erneut vermessen.

Das evolutionäre Düsensystem wurde bei einer einfachen konischen Form gestartet und konvergierte dann zu einer ungewöhnlichen, doppelbauchigen Form (Serie A in Abb. 9.1), die von keiner Theorie vorhergesagt wurde.

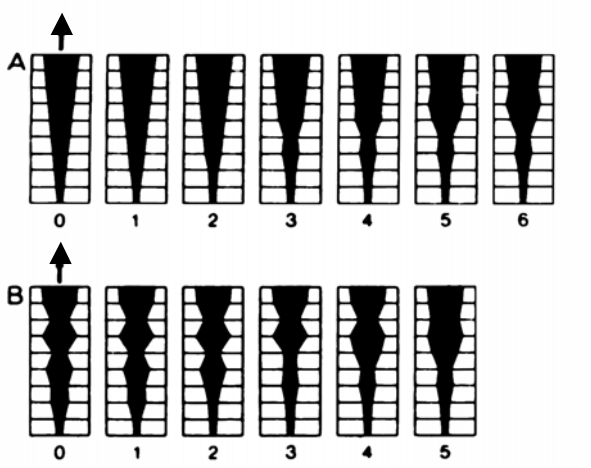


Abb. 9.1 *Evolutionäre Entwicklung einer Düsenform. Die Düse ist aus 9 Scheiben aufgebaut, die im Bild waagerecht angeordnet sind. Das chem. Substanzgemisch wird von unten durchgepreßt.*

Auch der Start mit einer zufälligen, welligen Form (Versuch B in Abb. 9.1) beeinflusste nicht das Endergebnis, das so wie in Versuch A aussah. Auf diese Art und Weise wurde

experimentell ein Düsenprofil, das (fast) dem maximalen, theoretisch möglichen Wert entsprach, gefunden, ohne die analytischen Zusammenhänge zwischen Form (Genotyp **g** bzw. Phänotyp) und Wirkung R(**g**) zu kennen.

Ein anderes Beispiel ist die Suche nach dem besten Mischungsrezept, um mit einem Verchromungsbad einen besonders dauerhaften Metallauftrag hoher Güte zu erreichen. Das Tupel für die Mutations-Selektionsstrategie wird in diesem Fall durch die *n* prozentualen Anteile der *n* möglichen Zutaten gegeben; die Zielfunktion R(**g**) muß durch einen Gütetest an den verchromten Werkstücken implementiert werden.

Ein interessantes Anwendungsbeispiel, dessen Lösung auch mit anderen Lösungsmethoden aus den Kapiteln 2, 3 und 4 verglichen werden kann, ist das Optimierungsproblem des Handelsreisenden, der die kürzeste Rundreise durch *n* vorgegebene Städte sucht. Kodieren wir uns die Rundreise A→B→C→D→E durch ein Tupel **g** = (A,B,C,D,E), so besteht die Aufgabe darin, dasjenige Tupel zu finden, das die Gesamtlänge der Rundreise, die Zielfunktion R(**g**), minimiert. Das Problem läßt sich mit den Begriffen der evolutionären Algorithmen formulieren.
Sei eine Reihe von Städten gegeben, die auf einer gedachten Kreislinie liegen. Drei mögliche Reisen sind in Abb. 9.2 gezeigt. Die dazu gehörenden Tupel sind $\mathbf{g}_1$ = (A,B,D,C,E), $\mathbf{g}_2$ = (A,D,C,B,E) und als beste Lösung $\mathbf{g}_3$ = (A,B,C,D,E). Sie haben eine größere Güte als diejenigen Zufallstupel, die keine Abfolge von nebeneinander auf der Kreislinie liegenden Städten (z.B. AB, CD etc.) enthalten. Die Tupel $\mathbf{g}_1$ und $\mathbf{g}_2$ enthalten Lösungselemente wie (A ... E) oder (AB...), die damit allein schon eine gewisse Güte garantieren und in einer günstigen Kombination, wie sie $\mathbf{g}_3$ darstellt, zum Optimum führen.

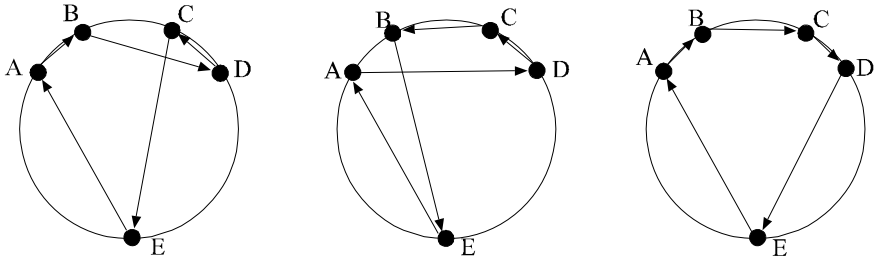


Abb. 9.2 Mögliche Rundreisen

Die verschiedenen Tupel der Rundreisen lassen sich als "Chromosomen" **g** interpretieren; die Stellen g_i im Tupel **g** = (g_1, ... ,g_n) als *Gene*. In biologischer Sprechweise ist somit das Tupel **g** der Genotyp der Rundreise, die Reise selbst der Phänotyp und die Länge der Reise die Fitneß R(**g**) des Phänotyps bzw. Genotyps.

Entscheidend für die Güte und die Schnelligkeit, mit der eine Lösung des Problems gefunden wird, ist zweifelsohne die Art der Veränderungsoperation auf dem Tupel.

Paul Ablay fand dazu in [ABL87] zwei verschiedene Arten der Mutation: das Aufbrechen von *s* Kanten des Rundreisegraphs (*Kantenmutation*) und das Vertauschen von *p* Positionen von Städten innerhalb einer Rundreise (*Positionsmutation*). In Abb. 9.3 ist eine Rundreise (AECDBF) gezeigt, die aus (ABCDEF) sowohl durch Aufbrechen und Neubesetzen der s = 4 Kanten A→B, B→C, D→E, E→F erzeugt sein kann, als auch durch Vertauschen der beiden Positionen B und E mit p = 2.

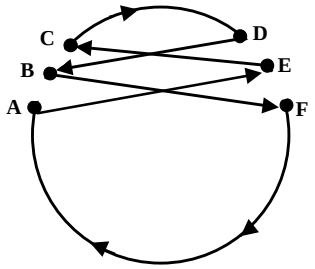


Abb. 9.3 Kanten- bzw. Positionsmutation

Zwar fand er heraus, dass der Algorithmus im allgemeinen mit der Kantenmutation besser als mit der Positionsmutation konvergiert; eine dritte Strategie funktionierte aber noch besser: die *gerichtete* bzw. *gewichtete Mutation*. Hierbei wird das Wissen über lokale, suboptimal verbessernde Veränderungsschritte in die Mutation eingebaut, so dass die Mutation nicht "blind", sondern überwiegend in die "richtige" Richtung erfolgt. Im Fall der Kantenmutation könnte dies für s = 3 Kanten in Abb. 9.4 bedeuten (s. [ABL87])

- Breche die zufällig gewählte Kante $[X_i, X_{i+1}]$ zwischen den Städten X_i und X_{i+1} auf.
- Suche ein zufälliges Paar $\{X_j, X_{j-1}\}$ so, dass der Abstand $| [X_i, X_j] |$ klein ist („gewichtete Zufallsauswahl“)
- Suche unter allen Möglichkeiten dasjenige Paar $\{X_k,X_{k+1}\}$, bei dem die beiden Verbindungen mit X_{j-1} und X_{i+1} besonders kurz werden.

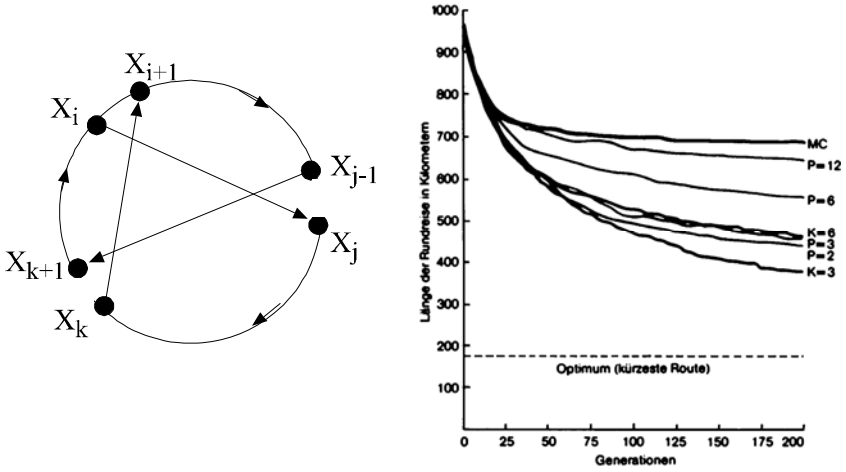


Abb. 9.4 Gewichtete Kantenmutation und ihre Konvergenz (nach [ABL87])

In Abb. 9.4 ist links die Kantenkonfiguration und rechts der Konvergenzverlauf der Iteration mit den obigen drei Schritten zu sehen.

Die gewichtete Mutation beschleunigt zwar die Konvergenz, vermeidet aber nicht das Problem der lokalen Optimierungsmethoden, in einem lokalen Optimum "stecken" zu bleiben. Ein gutes Verfahren, dieses Problem zu vermeiden, ist die Änderung der Mutationsrate, die zyklisch vergrößert und verkleinert wird. Eine kleine Mutationsrate bewirkt die gute Anpassung innerhalb eines Optimums; eine große Mutationsrate ermöglicht ein Überwechseln von einem lokalen Optimum in ein anderes.

9.2 Parallele Evolution mehrerer Lösungen

Die Grundidee der parallel arbeitenden evolutionären Algorithmen kommt aus der Populationsbiologie. In einer gegebenen Umwelt (Randbedingungen des Problems) gibt es mehrere Lebewesen (Problemlösungen), die durch ihre Gene $\mathbf{g} = (g_1 \ldots g_n)$ charakterisiert werden. Jedes Lebewesen hat eine bestimmte Überlebensfähigkeit bzw. *Fitness* (Güte der Lösung) R(**g**) besitzt. Im Unterschied zu dem Mutations-Selektions-Ansatz des vorigen Abschnitts versucht man nun, nicht nur durch eine Veränderung (*Mutation*), sondern auch durch eine geschickte Kreuzung (*cross over*) der Erbanlagen erfolgreicher Lebewesen miteinander (Kombination der Parametern von Lösungen hoher Güte), besser angepasste Lebewesen (bessere Lösungen für das Problem) zu finden. Im Unterschied zu den Mutations-Selektionsalgorithmen, bei denen die Verbesserung vorzugsweise sequentiell von Generation zu Generation erfolgt (*vertikaler Informationstransfer*), wird bei den Algorithmen mit parallel existierenden Lebewesen die Tatsache

ausgenutzt, dass gleichzeitig mehrere gute Lösungen existieren, um Verbesserungen zu erreichen (*horizontaler Informationstransfer*).

In der Sprache der Optimierungsalgorithmen verschafft man sich durch die parallele Existenz von mehreren, bewerteten Teillösungen ("Population" von Individuen bzw. "Pool" von Genen) die Möglichkeit, aus den Parametern Teillösungen zu extrahieren und sie zu "besten" Parameterkombinationen, zu neuen, besseren Lösungen mit Hilfe von *genetischen Operationen* zusammenzusetzen.

9.2.1 Gene

Den Begriff des „Gens“ für einen Funktionsparameter sollte man allerdings nicht zu eng fassen. Genauso wie im biologischen Fall als Gen ein funktionaler Abschnitt einer Sequenz von Aminosäuren aufgefasst wird, bei der die gesamte Sequenz die Erzeugung eines Enzyms regeln, kann für uns ein Gen auch ein Funktionselement eines Programms, beispielsweise eine Regel in einem logischen Programm [FUJI87] oder für eine elementare Verhaltensweise (Sequenz von Manövrierkommandos) innerhalb einer Flugzeug-Abwehrstrategie [GRE90] stehen. Eine Evolution der Population entspricht in diesem Fall einer Evolution eines Programms bzw. einer Strategie.

9.2.2 Schemata

Ein wichtiger Mechanismus für die Entwicklung solcher Populationen ist das Finden und Kombinieren von Lösungselementen (*building blocks*) wie in unserem Beispiel (A ... E) und (AB ...). Nach John F. Holland bezeichnet man die Menge aller Tupel mit den gleichen Lösungselementen (.. g_i, ... , g_j,...) als *Schema*. Betrachten wir ein solches Schema als spezielle Lösungsmenge in einem n-dimensionalen Lösungsraum $\{g_i\}^n$, so stellt diese Punktmenge eine Hyperfläche dar; die Schnittmenge der verschiedenen Schemata bzw. Hyperflächen enthält die gesuchte Lösung. Bezeichnen wir mit dem Symbol "#" die für ein Schema uninteressanten Gene ("don't care"), so lassen sich die Lösungselemente unseres Beispiels als Schemata S_1 = (A # # # E) und S_2 = (AB # # #) schreiben. Die *Länge* $L(S)$ eines Schemas ist dabei die Zahl der Stellen minus eins, die ein Schema zur positionsunabhängigen Charakterisierung benötigt; für unser Beispiel also $L(S_1) = 4$ und $L(S_2) = 1$.

Dabei ist für die Definition eines Schemas die Kodierung der Gene wichtig. Haben wir beispielsweise 1000 mögliche Lösungen, so lässt die dezimale Kodierung mit drei Stellen (g_1,g_2,g_3) weniger Schemata zu als die binäre Kodierung mit 10 Stellen, da jedes "don't care" Symbol mit 10 statt 2 möglichen Werten die fünffache Menge an Tupeln umfaßt. Aus diesem Grund wird traditionellerweise bei genetischen Algorithmen die binäre Kodierung eingesetzt; viele Konvergenzbeweise wurden für die Binärkodierung erstellt. Dem Vorteil der starken Veränderbarkeit bei binärer Kodierung steht die Tatsache entgegen, dass bei einer binären Kodierung durch Bits die Mutation einer Zahl

sehr unterschiedliche Auswirkungen hat, je nachdem ob ein niederwertiges Bit oder ein höherwertiges Bit angesprochen wurde. Es ist deshalb sinnvoll, die Mutationsrate bzw. Mutationswahrscheinlichkeit bei den höherwertigen Binärstellen kleiner zu wählen als bei den niederwertigen Bits.
Man sollte schon bei der Kodierung des Problems überlegen, wie die Lösungselemente des Problems - und damit mögliche Schemata - aussehen werden, und dann danach die Kodierung konstruieren.

9.2.3 Reproduktionspläne

Wie sieht nun eine evolutionäre Entwicklung für erfolgreiches Schema aus ?

In der Populationsbiologie wird Fortpflanzung und Überleben von zwei wichtigen Faktoren bestimmt: Einerseits durch das Entstehen neuer Individuen durch bereits vorhandene Individuen {**g**} mittels genetischer Operationen und andererseits durch die Selektion gemäß der Fitness R(**g**). Der Algorithmus, der beides miteinander verbindet, wird *Reproduktionsplan* genannt. Der genetische Algorithmus lautet dann folgendermaßen:

```
Repro:(* Allgemeiner Reproduktionsplan *)
    Erzeuge eine zufällige Population G:={g}
    REPEAT
       GetFitness(G)        (* Bewerte die Population *)
       SelectBest(G)        (* Wähle nur die Besten zur Reproduktion *)
       GenOperation(G)      (* Erstelle neue Individuen *)
    UNTIL Abbruchkriterium;
    Gebe g mit maximaler Fitness aus.
```

Codebeispiel 9-2 *Grundstruktur eines Reproduktionsplans*

In der ersten Prozedur `GetFitness(G)` werden die einzelnen Tupel bezüglich der Zielfunktion R(**g**) bewertet, und in der zweiten Prozedur ihre Auswahlwahrscheinlichkeit errechnet. Der Pseudocode dafür von M Tupeln **g** mit jeweils S Genen g_i könnte lauten

```
PROCEDURE GetFitness(G:ARRAY OF TUPEL)
BEGIN
  FOR i:=1 TO M DO          (* Bewerte jedes Tupel der Population *)
     fitness[i]:= R(G[i])
  ENDFOR;
  Rmin := MIN(fitness); Rmax := MAX(fitness);
  FOR i:=1 TO M DO              (* Bilde Fitness auf 0..1 ab *)
     fitness[i]:= (fitness[i]-Rmin)/(Rmax-Rmin) ;
  ENDFOR
END GetFitness;

PROCEDURE SelectBest(G:ARRAY OF TUPEL);
BEGIN                           (* Markiere die relativ Besten: *)
  N := M;
  FOR i:=1 TO M DO
     SelectProb:= fitness[i];
     IF Random(0,1) > SelectProb
          THEN delete(G,i); N:=N-1, ENDIF
  ENDFOR
END SelectBest;

PROCEDURE GenOperation(G:ARRAY OF TUPEL);
(* Erzeuge neue Tupel *)
BEGIN P := N;
WHILE P < M DO  (* ergänze fehlende Tupel *)
    (* Überkreuzung von gi und gj an Gen k *)
     i:=Random(1,N); j:=Random(1,N); k:=Random(1,S);
     cross_over(G[i],G[j],k,G[P+1],G[P+2]); (* 2 neue Tupel *)
     P:=P+2;

   (* Mutation von gi an Stelle j *)
     i:=Random(1,N); j:=Random(1,S);
     mutate(G[i],j,G[P+1]);    (* 1 neues Tupel *)
     P:=P+1;
ENDWHILE
END GenOperation;
```

Codebeispiel 9-3 *Prozeduren des Reproduktionsplans*

Die Normierung der Fitness auf das Intervall [0,1] garantiert dabei, dass die Fitness einer positiven Überlebenswahrscheinlichkeit zugeordnet werden kann. In der Auswahloperation `SelectBest` wird die Fitness dann dazu verwendet, die schlechtesten Individuen zu löschen. Die entstehende Lücke wird durch die Reproduktion mittels genetischer Veränderungsoperationen in `GenOperation(G)` wieder geschlossen.

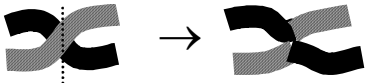
9.2.4 Genetische Operatoren

Die genetischen Operationen, mit deren Hilfe die neuen Tupel (und damit die "Individuen") entstehen und die schlechteren ersetzen, sollten dabei der Problemkodierung

angemessen gewählt werden. Es gibt einige genetische "Standardoperationen", die aus der Molekularbiologie stammen und sich auch bei Optimierungsproblemen bewährt haben.
Die Operationen sind:

- **Überkreuzung** (*Cross-over*)

  Zwei Tupel werden an einer Stelle auseinandergeschnitten und überkreuzt zusammengesetzt.

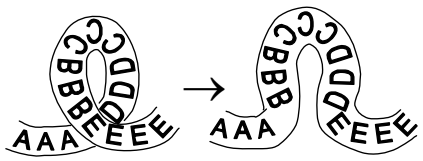


g_1= (AB|DCE) → (ABCBE)
g_2= (AD|CBE) → (ADDCE)

  In der Abbildung sind sie oben schematisch an Chromosomensträngen und darunter am Beispiel der Tupel der Rundreisen verdeutlicht. Eine Behandlung des Rundreiseproblems mit genetischen Algorithmen ist übrigens in [OLI87] enthalten.

- **Invertierung** (*Inversion*)

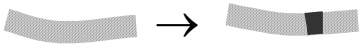
  Teilabschnitte des Chromosomen (Tupel) werden in der Reihenfolge der Gene invertiert.



g_2= (A|DCB|E) → (ABCDE)

- **Mutation**

  Einige Gene werden zufällig geändert und erhalten jeweils einen anderen, möglichen Genwert.



g_2= (ADCBE) → (ABCCE)

Der Überkreuzungsoperator hat vorzugsweise Einfluss auf alle Schemata, die weit auseinander liegende Stellen haben (z.B. (A...E), nicht aber (AB..)). Der Inversionsoperator hat dabei einen gegenläufigen Effekt, da er weit auseinander liegende Stellen zusammenbringt.

Außer den hier vorgestellten Operationen sind auch andere String-Operationen wie Löschen sowie An- und Einfügen (*Rekombination*) von Tupelstücken üblich.

Beispiel: *Der digitale Schmetterling (nach [RE73])*

An einem einfachen Beispiel wollen wir den Unterschied zwischen vertikalem Informationstransfer durch die Mutations-Selektionsstrategie und horizontalem Informationstransfer durch Überkreuzen und Rekombinieren zwischen Individuen gleicher Population verdeutlichen. Dazu unterteilen wir die Flügeloberfläche eine Schmetterlings in eine Matrix aus 9×9 = 81 einzelnen Flächenelementen, siehe Abb. 9.5. Jedes Element hat eine Farbe: weiß (0) oder schwarz (1). Die Matrix aus Nullen und Einsen charakterisiert also die Farbverteilung der Flügeloberfläche. Ordnen wir die Zeilen dieser Matrix hintereinander an, so entsteht eine Zeichenkette aus Nullen und Einsen: der Bauplan der Flügelfärbung.

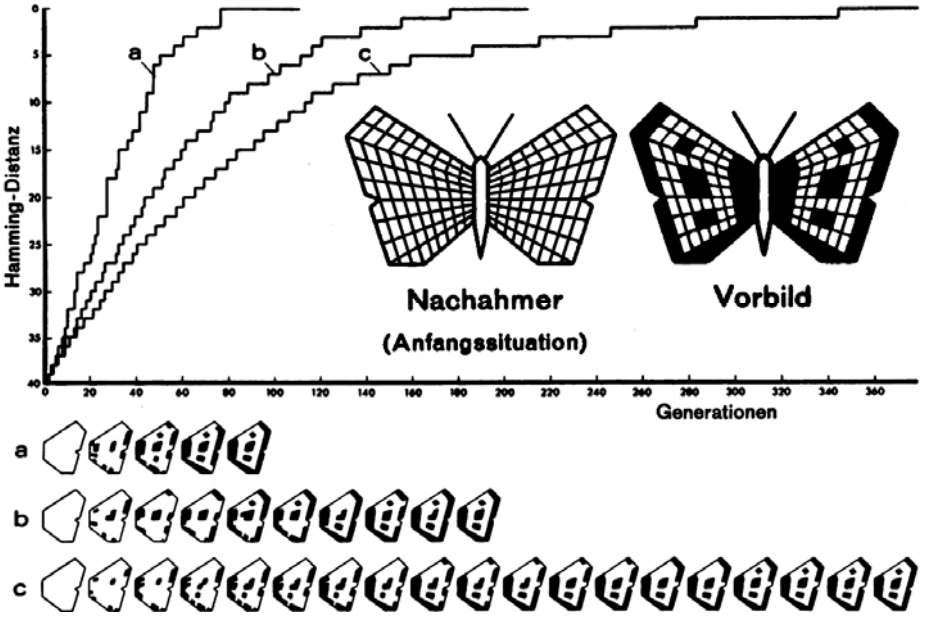


Abb. 9.5 *Vertikaler und horizontaler Informationstransfer beim digitalen Schmetterling* (nach [RE73])

Nun beobachten wir in der Natur, dass es Schmetterlinge gibt, die nicht so stark gefressen werden, weil sie giftig sind oder schlechter schmecken. Diese Schmetterlinge haben eine charakteristische Färbung $\mathbf{g}_0$. Jeder Schmetterling, der so ähnlich aussieht, wird leicht von den Vögeln mit dem problematischen Schmetterling verwechselt und deshalb weniger stark gefressen werden. Wir können deshalb wir die Fitnessfunktion durch die Unähnlichkeit der Schmetterlinge $\mathbf{g}_0$ (Original) und $\mathbf{g}$ (Nachahmer) modellieren, welche für gute Nachahmer besonders klein ist. Verwenden wir dazu den Ham-

mingabstand h($\mathbf{g}_0$,$\mathbf{g}$), welcher die Anzahl der unterschiedlichen Stellen zweier Tupel $\mathbf{g}_0$ und $\mathbf{g}$ angibt, so können wir damit testen, wie sich die Schmetterlingsevolution bei unterschiedlichen evolutionären Systemen entwickelt.
Wir betrachten dazu drei Systeme: (a) eines, das aus einer Population von 10 Individuen besteht und durch Mutation, Rekombination und Selektion geformt wird, (b) ein weiteres, das ebenfalls aus 10 Individuen besteht und nur Mutation/Selektion verwendet, und (c) ein Einzelindividuum, das Mutation und Selektion unterworfen ist. Die Leistung (*Fitness*) eines solchen Systems ist in Abb. 9.5 aufgetragen.
Zusätzlich zur Fitness ist auch der Zustand des besten Individuums jeweils alle 20 Generationen eingezeichnet. Wie man sieht, passt sich das System mit allen genetischen Operationen auf einer Population schneller an als das reine Mutations-Selektions-System, das aber schneller ist als das System eines einzigen Individuums. Hier zeigt sich der Vorteil des horizontalen Informationstransfers: gute Teile (mehrere gleich kolorierte Flächenabschnitte) können schnell weitergegeben werden und ergeben mit den bereits vorhandenen guten Teillösungen eine bessere Lösung. Beim rein vertikalen Informationstransfer werden im Vergleich die meisten Generationen für die optimale Lösung benötigt.
Allerdings muss man dieses Ergebnis kritisch betrachten: Ist unser Ziel eine möglichst rasche Konvergenz bei wenigen Generationen so wie in der Natur, so ist als Lösung sicher die parallele Strategie (a) vorzuziehen. Anders dagegen, wenn wir die dazu nötigen Operationen zur Berechnung auf einem sequentiellen Computer betrachten: Bei (a) sind für 10 Schmetterlinge 80 Generationen, also insgesamt 800 Schmetterlinge für die Lösung zu berechnen, bei (c) nur 380, wobei die Anzahl der Fehlversuche (verworfenen Änderungen) nicht eingerechnet wurde.

9.3 Diskussion

Für die Daten der Rundreise sehen wir beim *cross over* auf S.318, dass ein aus der genetischen Operation entstehendes Tupel nicht unbedingt besser als das ursprüngliche sein muss. Wann ist uns garantiert, dass sich ein günstiges Schema wie beim Schmetterling bei den entstehenden Populationen auch wirklich durchsetzt und nicht ausstirbt?

9.3.1 Schemareproduktion und Konvergenz

Diese Frage untersuchte J. Holland für die Überkreuzungsoperation und fand dazu folgenden, den als "Fundamentalsatz der genetischen Algorithmen" bezeichneten Zusammenhang:

> Sei m_s die Anzahl der Individuen, die das Schema *S* zum Zeitpunkt *t* besitzen. Dann sind zum Zeitpunkt *t+1* mindestens

$$m_s(t+1) \geq m_s(t)\, r_s\, P_s \tag{9.1}$$

Individuen mit dem Schema *S* als Teilmenge G_S aus *G* vorhanden, wobei die relative Fitness des Schemas hier definiert ist als

$$r_s := \frac{\text{mittl. Fitness des Schemas } R_s}{\text{mittl. Fitness der Population } R} = \frac{\frac{1}{m_s}\sum_{G_s} R(g)}{\frac{1}{m}\sum_{G} R(g)} \tag{9.2}$$

und die Wahrscheinlichkeit P_s, dass das Schema bei der Operation erhalten bleibt,

$$P_s = 1 - P_c\, L(S)/(n-1) \tag{9.3}$$

mit der Überkreuzungswahrscheinlichkeit P_c und der Wahrscheinlichkeit $L(S)/(n-1)$, dass die Überkreuzung in der Schemalänge $L(S)$ bei der Gesamttupellänge *n* stattfindet.

Betrachten wir Formel **Fehler! Verweisquelle konnte nicht gefunden werden.**, so sehen wir, dass ein Schema nur dann sich in der Population durchsetzen kann, wenn $r_s P_s > 1$ gilt, das Schema also eine überdurchschnittliche Fitness und eine geringe Schemalänge für eine geringe Veränderungswahrscheinlichkeit besitzt. Eine umfassendere Version dieses Satzes, in dem auch eine Neubildung der selben Schemata durch Überkreuzung sowie andere Einflüsse wie Mutation etc. zusätzliche Terme in Gl. **Fehler! Verweisquelle konnte nicht gefunden werden.** liefern, ist beispielsweise in [GOL89] enthalten. Da die Konvergenz eines genetischen Algorithmus nicht selbstverständlich ist, muss für jeden selbst definierten Operator prinzipiell auch ein ähnlicher Satz bewiesen werden.

Die Gruppe der Algorithmen mit binär kodierten Genen und Überkreuzung wurde zuerst von Holland in den 70-ger Jahren näher untersucht und in der amerikanischen Literatur als „genetische Algorithmen“ (*genetic algorithms*) bezeichnet. Der stärker an derartigen genetischen Algorithmen interessierte Leser sei auf die Lehrbücher von Holland [HOL75] und Goldberg [GOL89] verwiesen.

9.3.2 Kodierung und Konvergenz

Die genetischen Verfahren zur Kombination "guter" Lösungselemente (Schemata, *building blocks*) zu größeren Einheiten lassen sich nur dann sinnvoll bei Optimierungsproblemen einsetzen, wenn die Probleme geeignet strukturiert sind. Voraussetzung für diese Art von Optimierungsstrategie ist dabei eine derartige Kodierung der Gene, dass die genetischen Operationen sinnvoll zusammenarbeiten können und eine Kombination guter Lösungen auch eine bessere und nicht eine schlechtere Lösung ergibt. Existieren

aber beispielsweise bei gegebener Problemkodierung gar keine sinnvoll kombinierbaren Lösungselemente, so kann der genetische Algorithmus auch kein Optimum finden. Ein einfaches Beispiel für dieses Problem bildet die Fitnessfunktion R(**g**) in Abb. 9.6, die als Überlagerung einer ansteigenden Geraden und einer Sinus-Funktion definiert sei.

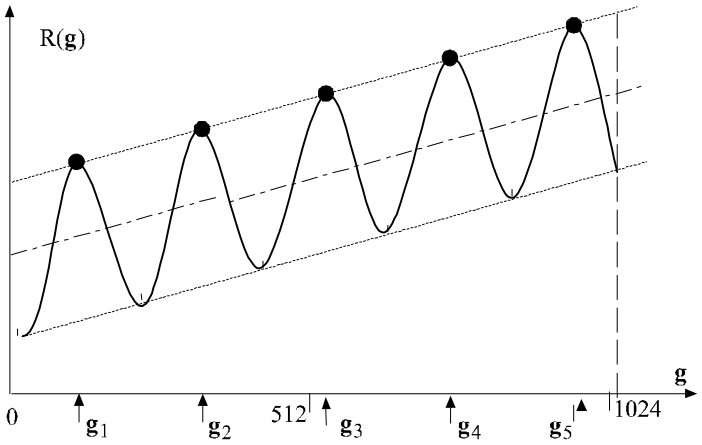


Abb. 9.6 *Eine Fitnessfunktion*

Werden die Individuen durch die Gentupel $\mathbf{g} \in \{0,1\}^{10}$ als binäre Repräsentation (*Genotyp*) der ganzen Zahlen *x* (*Phänotyp*) aus dem Intervall [0,1023] gebildet ("10Bit Integer"), so kann weder durch Überkreuzen noch durch Inversion aus den "guten" Bitstrings $\mathbf{g}_1$ und $\mathbf{g}_2$ deterministisch ein "besserer" Bitstring $\mathbf{g}_3$ oder $\mathbf{g}_4$ gewonnen werden, da der numerische Wert *x* und seine binäre Kodierung als 0-1 Bitfolge keine Beziehung zueinander haben, die man für die numerische Regelmäßigkeit der Fitnessfunktion (Sinusperiode) durch Überkreuzen oder Inversion ausnutzen könnte. Das Verharren ("Inzucht") der genetischen Population bei einem suboptimalen Wert wird nur durch eine zufällige Mutation überwunden; auch Überkreuzen und Inversion wirken hier nur als Zufallsoperationen ohne systematischen Gehalt. Die Suche nach dem globalen Optimum könnte man in diesem Fall also auch mit der einfachen Mutations-Selektions-Strategie ganz ohne den Ballast genetischer Operatoren durchführen.

9.3.3 Evolutionäre Rekombination

Die Kodierung von Parametern mit binären Codes, deren Bits als Gene angesehen werden, führt mit den genetischen Operatoren wie Rekombination und *crossover* nicht automatisch zu guten Ergebnissen, sondern kann auch nur wie einfache Mutationsoperationen wirken. Diese Erkenntnis bewog Rechenberg neben starker Polemik [Re94] dazu, ein eigenes Rekombinationsmodell zu schaffen. Bei seiner Kombination werden

als Genwerte die Zahlen der Parameter von Eltern-Tupeln ausgetauscht und zu neuen Kindern zusammengefasst (*rekombiniert*). Ein Beispiel ist in Abb. 9.7 gezeigt, wobei die Tupel (x,y) aus zwei Parametern als Punkte visualisiert sind. Für die zwei Tupel $\mathbf{g}_1 = (x_1,y_1)$ und $\mathbf{g}_2 = (x_2,y_2)$ ergibt eine solche Rekombination die Kinder $\mathbf{g}_3 = (x_2,y_1)$ und $\mathbf{g}_4 = (x_1,y_2)$.

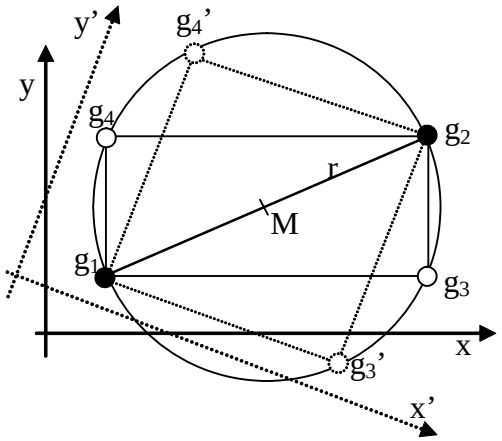


Abb. 9.7 *Die Rekombination zweier Elterntupel (•) zu zwei Kindtupeln (∘)*

Alle vier Tupel bilden zusammen ein Rechteck. Allerdings ist die Lage der Kinder von der Lage des Koordinatensystems abhängig. Würde statt der durchgezogenen Koordinatenachsen mit x, y das gestrichelt angedeutete Koordinatensystem mit x', y' vorliegen, so würden dann die gestrichelt eingezeichneten Kinder $\mathbf{g}_3' = (x_2',y_1')$ und $\mathbf{g}_4' = (x_1',y_2')$ vorliegen. Was haben die doch verschiedenen Kinder aus verschiedenen Koordinatensystemen gemeinsam? Da die Verbindungsgeraden zu den Elterntupeln jeweils einen rechten Winkel einschließen, liegen alle möglichen Kinder auf einem Kreis, dem sog. *Thales*-Kreis. Im *n*-dimensionalen Raum der *n* Gene bzw. *n* Parameter liegen die Kind-Rekombinationen also auf einer Hyperkugel um den Halbierungspunkt *M* zwischen den beiden Eltern. Der Durchmesser der Kugel ist der Elternabstand. Bei sehr vielen Dimensionen gibt es mit $(2^n - 2)$ auch sehr viele mögliche Positionen von Kindern. Im kontinuierlichen Fall einer „kontinuierlichen Rekombination" steht die ganze Fläche der Hyperkugel dafür zur Verfügung. Würde man eine zufällige Mutation konstruieren, die von *M* aus eine Position nach einer Normalverteilung $N(M,\sigma)$ mit $n\sigma^2 = r^2$ für ein Kindstupel erwürfelt, so wäre der Effekt sehr ähnlich. Eine derartige, auf diskrete Parameter beschränkte Rekombination wirkt also wie eine Mutation mit einer Schrittweite $\sigma = \sqrt{n}\, r$, proportional zum Abstand der Elterntupel und kann so mit dem Mutations-Selektionsmodell ohne das Schema-Modell von Holland beschrieben werden.

9.3.4 Genetische Drift

Ein weiteres Problem genetischer Algorithmen bildet die Begrenztheit der Populationsgröße. Durch die Zufallsnatur der Selektion können bei geringer Zahl von Individuen leicht Tupel entstehen, die nur aus "1" bzw. "0" bestehen und deshalb durch Überkreuzen oder Inversion nicht mehr zu verändern sind. Obwohl in der initialen Population ein Gleichgewicht vorhanden war, kann die Akkumulation von Selektionsfehlern einen Extremzustand herbeiführen (*Genetische Drift*). Erst die Einführung einer Mutation kann dies verhindern [GOL87].

9.4 Genetische Operationen mit neuronalen Netzen

In den bisherigen Abschnitten wurde der genetisch-evolutionäre Ansatz wie eine Alternative präsentiert zu dem in den vorigen Kapiteln betrachteten neuronalen Netzen. Dies ist aber nicht der Fall: Es ist vielmehr eine Art Meta-Algorithmus. Beispielsweise ist bekannt, dass Singvögel beim Lernen des Gesangs ihren eigenen Gesang an den Gehörten von Artgenossen anpassen. Dabei aktivieren die neuronale Gehirnregionen, die für die Motorsteuerung zuständig sind, den Gesang in immer neuen Variationen. Jede Variante wird mit dem Gehörten verglichen. Entstehen dabei Varianten, die ähnlicher zum Vorbild geworden sind, so werden sie beibehalten; diese Verschaltungen werden gefestigt und der Gesang wird weiter variiert solange, bis eine starke Ähnlichkeit erreicht ist.

Genetische Operationen lassen sich, wie bereits erwähnt, auf verschiedene Art und Weise auf neuronale Netze anwenden und helfen auch dabei, die Grenzen einer bestehenden neuronalen Netzarchitektur zu überwinden.

9.4.1 Evolution der Gewichte

Kodieren wir beispielsweise die Gewichte der Gewichtsmatrix **W** im Hopfield-Modell (vollständig vernetzte Neuronen) aus Kapitel 7 als Gene, so kodiert jeder Genotyp $\mathbf{g} = (W_{11}, \ldots, W_{nn})$ den Lernzustand eines Netzwerks. Geben wir noch die geeignete Fitnessfunktion vor, beispielsweise die Fehlerrate beim Auslesen gespeicherter Muster, so bedeutet die genetische Evolution der Gewichte ein "Lernen" der Gewichte für eine minimale Auslese-Fehlerrate. Die genetischen Operatoren (Überkreuzen, Mutation) verhindern dabei im Unterschied zu den Gradientenalgorithmen ein Verharren des Systems im suboptimalen, lokalen Minimum.

9.4.2 Evolution der Netzarchitektur

Im Unterschied zu allen bisher besprochenen Lernverfahren kann man mit evolutionären Algorithmen nicht nur die Gewichte eines vorgegebenen Netzwerks bei bekanntem Lernziel ermitteln, sondern darüber hinaus auch das Netzwerk selbst [SJW92],[HARP90],[DOD90]. Dazu muss man nur die Struktur und die Anzahl der Neuronen geeignet in dem Tupel kodieren.
Beispielsweise kann man ein Mehrschichten-Netzwerk mit *n* Schichten in einem Tupel von *n* Genen kodieren. Jedes Gen besteht wiederum aus zwei Unterabschnitten: den Daten für die Neuronen (Zahl, geometrischen x-y Eingabebereich, Lernraten, Schwellwerte, Ausgabefunktion etc) und den Verbindungsdaten (Verbindungsdichte, Projektionsadresse, x/y Radius des lokalen Ausgabebereichs, etc) für das Netzwerk [HARP90]. Überkreuzungs- und Inversionsoperatoren wirken dabei natürlich nur auf bedeutungsmäßig gleichartigen Genstücken.
Eine andere Kodierung, die nur die Verbindungen beschreibt, kann aus einem Tupel ((Neuronenindex1, Zielindex1, Zielindex2,..; Neuronenindex2,....) bestehen. Ein einfaches XOR-Netz wird so durch den Bauplan (1,3,4|2,3,4|3,5|4,5) beschrieben.

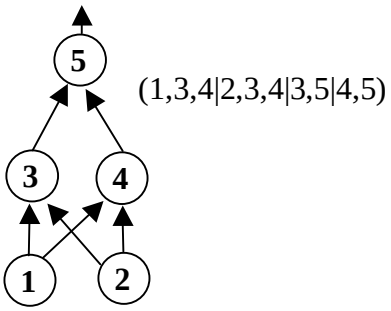


Abb. 9.8 Ein XOR-Netz und seine Strukturkodierung

Ein Gene besteht hier aus jeweils einem Neuron mit all seinen Verbindungen. Die Operationen wie *cross over* etc. wirken dann nur auf die Verbindungen von zwei zufällig gewählte Neuronen (Genen) zweier Baupläne. Der grundsätzliche Algorithmus für eine Netzarchitekturevolution ist ein Reproduktionsplan nach Codebeispiel 9-2. Die Fitnessfunktion kann man allerdings nur relativ aufwendig gewinnen; sie besteht im Prinzip aus der Leistungsfähigkeit des trainierten Netzwerks. Wollen wir also die besten Architekturversionen bestimmen, so müssen wir zuerst einen Trainingsalgorithmus definieren, den wir auf alle Netzwerkvarianten anwenden. Im Anschluss daran werden alle Varianten getestet. Die Fitnessfunktion nimmt also folgende Form an:

```
PROCEDURE R (g:TUPEL)      (* Fitnessfunktion für ein Netzwerk *)
  BEGIN
```

```
    FOR pattern:=1 TO N DO        (* Trainiere das Netzwerk *)
      TrainNet(g,pattern)
    END
     FOR test:=1 TO N DO           (* Teste das Netzwerk *)
      TestNet(g,test,results[test])
    END
    R(g):=Evaluate(results)       (* Berechne die Güte d. Netzes*)
END R;                             (* aus den Testergebnissen  *)
```

Codebeispiel 9-4 Fitnessfunktion für Netzwerkevolution

Mit Hilfe der Netzbeschreibung, dem Reproduktionsplan (Codebeispiel 7.2.1) und der oben definierten Fitnessfunktion kann man ein Softwaresystem aufbauen, mit dem man für eine gegebene Anwendung mit gegebenen Trainings- und Testdaten das dafür am besten geeignetste Netzwerk findet. Steven Harp und Mitarbeiter [HARP90] konstruierten sich ein solches Werkzeug, das sie NeuroGENESYS nannten. Dabei fand beispielsweise das System für einen ihrer Datensätze, dass sogar ein einlagiges Netz ausreichte - die Daten waren "linear separierbar", ohne dass die Autoren dies vorher bemerkt hatten.
Als Beispiel sind in der folgenden Abbildung die 10 zufällig generierten Startwerte für ein Netzwerk gezeigt, das die XOR-Funktion mittels Backpropagation lernen soll.

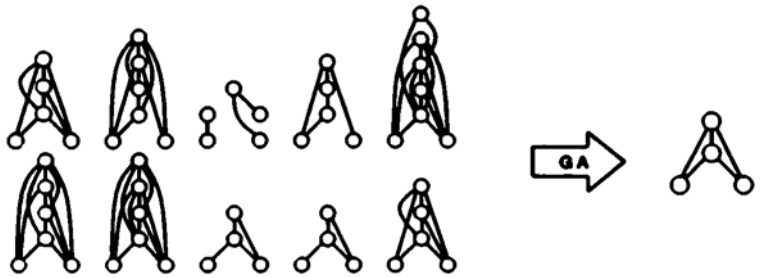


Abb. 9.9 Die 10 Netze beim Start und das Endprodukt (nach [SJW92]).

Der Endzustand nach 1000 Epochen ist ebenfalls gezeigt: das günstigste Netz ist die aus Kapitel 4 bekannte Lösung. Man beachte, dass bei dem Algorithmus automatisch auch das kleinste Netz gesucht wird, wenn nur eine feste Anzahl von Trainingssequenzen pro Netz verwendet wird: Kleine Netze konvergieren schneller und liefern deshalb auch im Mittel die größte Fitneß für eine gegebene Anzahl von Lernschritten.

Phänotyp, Genotyp und Konvergenz

Angenommen, wir gehen von einem zufälligen Anfangstupel des Parametersatzes aus und suchen nur durch Mutation analog einer Gradientensuche neue Tupel (Genotyp),

deren Erscheinungsbild (Fitness) eine gestellte Aufgabe optimal lösen, so können wir, wie Dodd [DOD90] fand, eine Überraschung erleben: das gleiche Netzwerk, das vorher sehr gute Ergebnisse brachte, zeigt nach einem erneuten Training mit neuen, zufälligen Anfangsparametern (Gewichten) wesentlich schlechtere Ergebnisse. Im Unterschied dazu findet ein Überkreuzungsoperator ein stabiles Optimum der Netzarchitektur, das unabhängig von den Anfangsgewichten immer gleich gute Testergebnisse erbringt. Die unterschiedliche Wirkungsweise der beiden Operatoren beruht wohl auf den unterschiedlichen Randbedingungen, die beide Operatoren schaffen: der Überkreuzungsoperator begünstigt eine Netzwerkarchitektur, die auch bei vielfältiger Vertauschung der Gewichte optimal bleibt, was zufällig wechselnden Gewichten entspricht, während die Mutation eine vorgegebene Gewichtskombination optimiert. Der Autor interpretiert dieses Ergebnis als Unterschied zwischen der Optimierung des Phänotyps durch die Mutation und des Genotyps durch den Überkreuzungsoperator.

Zusammenfassend lässt sich sagen, dass die genetischen Algorithmen ein interessantes Werkzeug bilden, um optimale Parameter für Systeme zu finden, die eine nichtstetige Zielfunktion besitzen und deshalb mit den üblichen Gradientenverfahren nur schwer bearbeitet werden können. Dabei sollte man aber durchaus kritisch darauf achten, ob eine adäquate Kodierung vorliegt, die eine Existenz von Lösungselementen (*building blocks*) garantiert.

Eine automatische Generierung einer Netzstruktur durch genetische Algorithmen ist möglich, verlangt aber für größere Netze eine hohe Rechenleistung, so dass dieses Verfahren nur für kleinere Netze (Module) spezieller Funktionalität sinnvoll ist.

10 Schwarmintelligenz

In der Natur können wir das Phänomen beobachten, dass nicht nur einzelne Lebewesen intelligentes Verhalten zeigen können, sondern auch mehrere, die zusammenarbeiten können. Dies ist etwa bei Termitenstaaten der Fall, wo jede Termite eine genau festgelegte Rolle erfüllt und erst durch das Zusammenwirken aller ein soziales Gebilde, ein Termitenstaat entsteht, der allen das Überleben ermöglicht. Aber auch bei losen Zusammenschlüssen, etwa einem Fischschwarm oder einem Vogelschwarm auf dem herbstlichen Weg in sonnigere Gefilde, lässt sich „Schwarmintelligenz“ beobachten. Dabei entsteht aus den dezentralen Aktionen aller etwas Neues, das Verhalten des gesamten Kollektivs.

10.1 Schwärme

Diese Idee einer Schwarmintelligenz ist im Genre des *Science Fiction* ein bekanntes Motiv. Schon um 1930 beschrieb etwa der englische Autor Olaf Stapledon eine insektenartige Lebensform auf dem Mars, die sich in Schwärmen ebenfalls zu großen Überintelligenzen verbinden konnte und (natürlich) letztendlich sogar versuchte, die Erde zu erobern. Stanislaw Lem schildert in seiner Geschichte „Der Unbesiegbare“ (1964) eine seltsame Erscheinung. Kleine Metallkristallformationen, die an Insekten erinnern, werden von unscheinbaren Pflanzen erzeugt und bevölkern in loser Form den Planeten und sind die einzig überlebende Form eines lang andauernden Krieges. Bei Gefahr können sich die an sich harmlosen und simplen Gebilde zu bedrohlichen und scheinbar intelligent handelnden, 'schwarzen Wolken' zusammenfinden, die ihre in riesigen Schlachten gewonnenen und auf die Einzelkristalle verteilt gespeicherten Informationen im Bedarfsfall zusammenkoppeln. Bei Vernichtung einzelner Teile können diese sofort wieder ersetzt werden, ohne die Funktion des Ganzen zu beeinträchtigen.

Eine solche Idee dezentralen Verhaltens liegt allen Algorithmen zur Schwarmintelligenz zugrunde. Obwohl die Idee selbst faszinierend ist, ist sie doch schwer zu realisieren. Neuere Ansätze versuchen dies in peer-to-peer (P2P)Computernetzen für die dezentrale, ausfallsichere Datenhaltung umzusetzen. Unabhängig davon gibt es aber einige Ansätze, die Idee der Schwarmintelligenz in Optimierungsproblemen zu nutzen. Zwei Ansätze, die Behandlung des klassischen Problems des Handlungsreisenden durch

Ameisenalgorithmen und das Schedulingproblem durch Bienenalgorithmen, sollen im folgenden Abschnitt vorgestellt werden.

10.2 Ameisenalgorithmen

Bei natürlichen Ameisen kann man beobachten, dass sie bei der Nahrungssuche einen optimalen Weg wählen. Er entsteht, wenn viele Ameisen den selben Weg gehen und dabei durch Duftstoffe (Pheromone), die sie dauernd absondern, eine Spur hinterlassen. Diesen chemischen Fährten folgen andere Schwarmmitglieder, die ebenfalls nach Futter suchen und mit einer Präferenz dem stark duftenden Weg folgen. Nach einer Weile wird der kürzere Weg von weit mehr Ameisen benutzt als der längere Weg zum Futter.

Warum? Die Ameisen, die den kürzeren Weg genommen haben, sind als erste wieder zurück am Ameisenhaufen. Sie haben damit in der gleichen Zeit mehr Pheromonspuren auf ihrem Weg hinterlassen, wie die Ameisen, die den längeren Weg gewählt haben. Aufgrund der kürzeren Strecke schaffen sie in einer bestimmten Zeit den Hin- und Rückweg schneller, so dass zu einem Zeitpunkt eine höhere Duftstärke auf dem kürzeren Weg besteht. Ameisen, die jetzt loslaufen, werden sich mit höherer Wahrscheinlichkeit für den kürzeren Weg mit mehr Pheromonspuren entscheiden und damit die Duftstärke weiter erhöhen: Es entsteht eine Rückkopplung, die eine schnelle Konvergenz zum kürzesten Weg bewirkt. Dies tritt sogar auf, wenn beide Wege gleich lang sind: Jede größere statistische Fluktuation der Ameisenzahl auf einem Weg kann das System in einen stabilen, asymmetrischen Zustand bringen. In Abb. 10.1 ist links die Versuchsanordnung für die Ameisenart *Linepithema humile* und rechts das Ergebnis als relative Anzahl der Ameisen pro Weg in Abhängigkeit von der Zeit gezeigt. Man sieht, dass nach einer Fluktuation ein stabiler Zustand erreicht wird, der nur etwas schwankt.

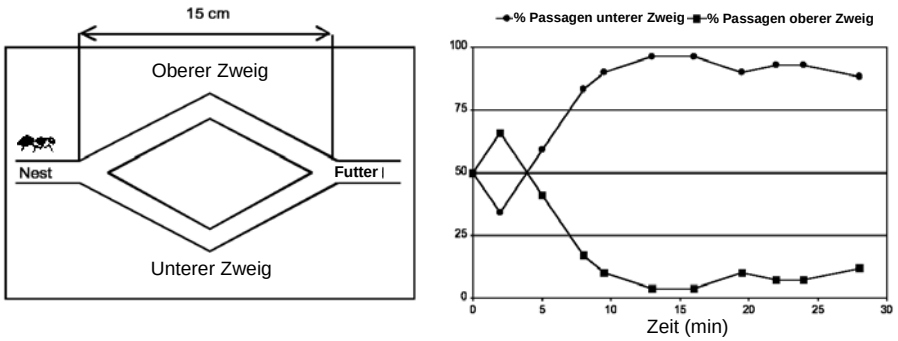


Abb. 10.1 *Versuchsanordnung und Anzahl der Ameisen auf dem Weg (nach [DOG99])*

Im Interaktionssystem ist die Duftstärke des Wegs damit zum Wert der globalen Variablen „Weg“ geworden, mittels dem das verteilte System sich koordiniert. Eine solche gegenseitige Beeinflussung über physische Variablen außerhalb der Tiere wird „sementic interaction“ genannt.

Diesen Vorgangs kann man folgendermaßen beschreiben: Sei U_m die Anzahl der Ameisen, die bisher den oberen Weg und L_m die den unteren Weg gewählt haben. Dann ist P_U die Wahrscheinlichkeit dafür, dass die *m*-te Ameise den oberen Weg wählt gegeben durch

$$\text{rel. Häufigkeit oben} = \frac{\text{Anzahl oben}}{\text{Gesamtzahl oben + unten}}$$

$$P_U(m) = \frac{(U_m + k)^h}{(U_m + k)^h + (L_m + k)^h} \tag{10.1}$$

mit den empirischen Konstanten k = 20 und h = 2.

Die biologischen Ameisen lösen das Problem, den kürzesten Weg zwischen zwei Punkten zu finden. Diese Leistung ist für Informatiker interessant: Sie kann dazu verwendet werden, das Problem des Handlungsreisenden zu behandeln. Dazu definieren wir uns künstliche Ameisen, die zwei zentrale Ideen der natürlichen Ameisen verwenden:

1. Die Entscheidung für einen Weg ist lokal und wahrscheinlichkeitsgestützt
2. Die Belegung der Wege durch Pheromone (*Stigmergy*) ist der zentrale Koordinationsmechanismus.

Im Unterschied zu biologischen Ameisen hat unsere Welt allerdings diskrete Zustände, bei denen jeweils von einem Ort zum nächsten gewandert werden kann; die Ameisen haben ein Gedächtnis und die Art der Pheromonabsonderung kann unnatürlich sein. Typischerweise werden für die praktischen Optimierungsalgorithmen der Ameisenkolonien noch weitere, nicht-natürliche Algorithmenelemente eingebaut wie lokale Optimierung, back-tracking usw.

Als Beispiel eines solchen Ansatzes betrachten wir den Ameisenalgorithmus von Dorigo et al. [DOG97][DOG99] für das Problem des Handlungsreisenden. Die Grundstruktur ist relativ einfach:

```
FOR t = 1 TO t_max
    FOR ant = 1 TO m
        Bilde eine TSP-Tour
    ENDFOR
    update_pheromone
ENDFOR
```

Codebeispiel 1 *Der Ameisenalgorithmus*

Am Anfang werden alle Ameisen zufällig auf die Wegpunkte (Städte) verteilt. Bei jedem Zeitschritt der Iteration müssen nun alle Ameisen parallel sich einen Weg zu suchen. Nach der Wegewahl werden die Wegstücke bewertet („mit Pheromonen belegt“) entsprechend ihrer Güte, die der resultierende Gesamtpfad der einzelnen Ameise hatte.
Dabei wird die Tour jeder einzelnen Ameise folgendermaßen ermittelt:

Bilde eine TSP-Tour:

- Bilde *einmal* für die Kante von Stadt *i* nach Stadt *j* die relative Gewichtung

$$a_{ij}(t) = \frac{w_{ij}(t)^{\alpha}\left(1/d_{ij}\right)^{\beta}}{\sum_{r\in N_i} w_{ir}(t)^{\alpha}\left(1/d_{ir}\right)^{\beta}} \qquad \text{mit } w_{ij} = \text{Pheromonmenge auf der Kante} \tag{10.2}$$

  für *alle* Städte *i* und *j* mit ihrem jeweiligen Abstand d_{ij}. Hier sind α und β feste Koeffizienten, mit denen das Verhalten der Algorithmen gelenkt werden kann. Ist $\alpha = 0$, so erhalten wir einen klassischen Greedy-Algorithmus: Alle Ameisenübergänge sind durch die Weglängen festgelegt. Ist dagegen $\beta = 0$, so ist nur die Pheromonlenkung mit ihren zufälligen Abweichungen aktiv.

- Starte nun bei zufälliger Stadt mit Ameise *k*

- Errechne nun Übergangswahrscheinlichkeit p_{ij} von Ameise *k* für Kanten bisher nicht besuchter Städte

$$p_{ij}^{k}(t) = \frac{a_{ij}(t)}{\sum_{r\in N_i^k} a_{ir}(t)} \qquad \text{mit } N_i^k = \{\text{alle von Ameise } k \text{ noch nicht besuchten Knoten}\} \tag{10.3}$$

  Die Normierung durch den Quotient wird nötig, da hier $\sum_r a_{ir} \neq 1$ ist. Man beachte, dass die Einführung von mit N_i^k einem privaten, individuellen Gedächtnis jeder Ameise entspricht, was bei biologischen Ameisen so nicht existiert.

- Lagere Pheromone ab. Dies kann nach lokal jedem einzelnen Wegstück der Tour geschehen, ist aber effizienter, wenn es erst am Ende mit der Tourbewertung vorgenommen wird. Dabei kann man verschiedene Strategien verwenden, etwa

  $\Delta w_{ij} = \text{const}$ *Dichtestrategie*

  oder $\Delta w_{ij} = 1/d_{ij}$ *Mengenstrategie*

Update_pheromone

Nach jeder Tourensuche aktualisiere für alle Kanten der gefundenen Tour die Gewichte

$$w_{ij}(t+1) = (1-\rho)\, w_{ij}(t) + \rho\, \Delta w_{ij}(t) \qquad \textit{Tour markieren}$$

$$\text{mit} \quad \Delta w_{ij}(t) = \sum_{k=1}^{m} \Delta w_{ij}^{k}(t) \qquad \textit{Bewertung aller Ameisen} \tag{10.4}$$

$$\text{und } \Delta w_{ij}^{k}(t) = \begin{cases} 1/L^{k}(t) & \text{Tourlänge} \\ 0 & \text{wenn nicht besucht} \end{cases}$$

Der positive Koeffizient $\rho < 1$ bewirkt dabei ein "Vergessen" und gibt an, wie viel Anteil an der alten Gewichtung erneuert werden soll.
Testet man diesen Ameisenalgorithmus an Aufgaben mit einer großen Anzahl an Dimensionen, so ergeben sich einige Konvergenzschwierigkeiten. Dies führte zu einer Modifikation des Algorithmus, nun unter dem Namen "Ant Colony System" ACS firmiert [DOG97]. Dabei zeigten sich folgende drei Änderungen sinnvoll:

a) *Ausnutzen vs. Erkunden*

Es wird $\alpha=1$ gesetzt und die Wahrscheinlichkeit des Übergangs wird mit Hilfe einer Zufallszahl $\xi \in [0,1]$ und einem festen Schwellwert θ definiert zu

$$p_{ij}^{k}(t) = \begin{cases} 1 & \text{wenn } \xi < \theta \text{ und } j = \arg\max_{r} a_{ir} \\ \text{wie in (9.3)} & \text{wenn } \xi \geq \theta \end{cases} \tag{10.5}$$

Ist etwa $\theta = 0{,}9$, so wird in 90% aller Fälle die (mit $p_{ij}=1$ deterministisch) beste Verbindung gewählt; aber in 10% der Fälle wird eine davon abweichende Verbindung ausprobiert. Über die Schwelle θ wird so die Balance zwischen der greedy-Strategie der bekannten besten Verbindung ("*exploitation*", Ausnutzung) und dem Erforschen einer neuen Verbindung ("*exploration*", Erkundung) gehalten. Dieser Gegensatz zwischen Optimierung einer fester Struktur und Ausprobieren einer Strukturänderung entspricht dem Problem, zwischen einem lokalen Optimum und einem (vielleicht) vorhandenen globalen Optimum zu wählen. Es ist auch unter dem Namen "Stabilitäts-Plastizitäts"-Problem bekannt.

b) Stigmergy: *global trail update*
Nach allen Tourensuchen aktualisiere für die Kanten der **besten** Tour mit der Länge L^{+} die Gewichte zu

$$w_{ij}(t+1) = (1-\rho)\, w_{ij}(t) + \rho\, \Delta w_{ij}(t) \qquad \textit{Tour markieren mit}$$

$$\text{mit} \quad \Delta w_{ij}(t) = 1/L^{+} \qquad \textit{bester Tourlänge aller Ameisen} \tag{10.6}$$

c) *Spur verwischen (local trail update)*

Bei dem obigen Algorithmus gibt es ein Problem: Zu viele Ameisen wählen den gleichen Weg. Um die nachfolgenden Ameisen stärker abzulenken, wird deshalb nach jedem Übergang der Gewichtswert verändert zu

$$w_{ij}(t+1) = (1-\gamma)\, w_{ij}(t) + \gamma\, w_0 \qquad (10.7)$$

mit einem festen Wert w_0, etwa $w_0 = \frac{1}{nL_{nn}}$

wobei L_{nn} die heuristische, durch die Verbindung zum jeweils nächsten Nachbarn gefundene kurze Tour zwischen *n* Städten ist. Diese Strategie entspricht einem Verdunsten von Pheromonen, was allerdings in der Realität erst nach Tagen, nicht nach Minuten eintritt.

Wie ist dieser Algorithmus im Vergleich mit anderen Algorithmen zu bewerten? In Abb. 10.2 sind die Ergebnisse eines Vergleichs des Ameisenalgorithmus ACO mit anderen approximativen Verfahren (*simulated annealing* SA, *elastic net* EN, *self-organizing map* SOM und *farthest insertion* FA) für verschiedene Konfigurationen mit 50 Städten zu sehen.

Problem	ACS	SA	EN	SOM	FI
City set 1	**5.86**	5.88	5.98	6.06	6.03
City set 2	6.05	**6.01**	6.03	6.25	6.28
City set 3	**5.57**	5.65	5.70	5.83	5.85
City set 4	**5.70**	5.81	5.86	5.87	5.96
City set 5	**6.17**	6.33	6.49	6.70	6.71

Abb. 10.2 *Die kürzeste gefundene Tour verschiedener Algorithmen(aus [DOG97])*

Die jeweils kürzeste Tour ist fett gedruckt. Man sieht, dass das ACO-Verfahren durchaus akzeptable Ergebnisse liefert. Weitere Anwendungen beim Graph-Färbeproblem, Fahrzeug-Routing, Job-Shop-Scheduling und Netzwerk-Routing siehe [DOG99].
Interessanterweise lässt sich dieser Algorithmus nicht so einfach parallelisieren, da er durch die globalen Pheromondaten einer ständige Synchronisierung lokaler Aktivität bedarf. Entsprechende Versuche zeigten eine schlechte Skalierbarkeit. Stattdessen bewährte es sich, Unterkolonien von Ameisen zu bilden, sie parallel auf identischen Graphen laufen zu lassen und anschließend die Ergebnisse in globaler Kommunikation auszutauschen.

Es gibt übrigens interessante Ähnlichkeiten zu dem Ansatz der formalen neuronalen Netze, Probleme zu lösen. So kann man jede Ameise als Signal ansehen, das in einem Netz (dem Graphen) von einem Neuron (einem Zustand) über Gewichte (Kantenbewertungen) zu anderen Neuronen (Zuständen) übergeht. Allerdings gibt es auch Unterschiede: Die Auswahl der Kante beim Ameisenalgorithmus ist eine stochastische Transferfunktion und nicht deterministisch. Dies würde einem "umschaltenden" Neuron ent-

sprechen, was als Modell nicht existiert. Die Lernregeln wiederum sind durchaus plausibel: Gewichte zu "guten" Lösungen werden verstärkt.

10.3 Bienenalgorithmen

Auch aus dem Verhalten von Bienen lassen sich interessante Algorithmen abstrahieren. Betrachten wir dazu zuerst das Verhalten biologischer Bienen.

- Bei den Bienen gebe es *m* Futtersucher. Jede Biene, die zurückkommt, teilt ihr Ergebnis durch einen Schwänzeltanz mit. Sei die Güte der Futterquelle eine Funktion R(Distanz zum Bienenstock, Futtermenge, Futterqualität), so ist neben der Futterprobe, die mitgebracht wird, die zeitliche Länge des Schwänzeltanzes proportional zu R.

- Weitere Futtersucher beobachten zufällig einen der Tänze und fliegen zur Futterquelle los.

- Damit wird eine positive Rückkopplung erzeugt: Je länger der Tanz dauert, umso mehr Futtersucher fliegen los. Zurückkehrende Futtersucher mit guten Ergebnissen erzeugen weitere Aktivität, so dass gute Futterquellen einen zunehmenden Anteil der Futtersucher beschäftigen, relativ schlechtere einen abnehmenden Anteil.

Diese Mechanismen verwendeten Nagrani und Tovey in [NaTo03], um ankommende http-Anfragen auf die Server eines Internet-Clusters zu verteilen. Üblicherweise werden solche Anfragen auf virtuelle Server, die aus mehreren, zusammengeschalteten realen Servern bestehen, geleitet. Jede derartige themenorientierte Zusammenschaltung kostet allerdings Zeit und Ressourcen; die Zahl der Schaltungen sollte deshalb minimisiert werden. Wie lässt sich das erreichen?
Die Autoren stellten dazu folgende Zuordnung auf:

Warteschlange	=	Blumenbeet (es ändert sich laufend in der Größe, Zusammensetzung usw. je nach Wetterlage)
Server	=	Futtersuchbiene
Virt. Server	=	Eine Gruppe der Futtersuchbienen
Menge aller Server	=	Bienenstock
Servicequalität	=	Futterqualität
Servicemenge	=	Futtermenge
Migrationskosten	=	Aufwand, den Futterort zu wechseln.
Globaler Speicher	=	Tanzboden
Inter-Server-Nachricht	=	Schwänzeltanz

Wir nehmen an, dass es *m* Gruppen von virtuellen Servern $VS_0,...,VS_{m-1}$ gibt, die War-

teschlangen $Q_0,...,Q_{m-1}$ haben. Pro Auftrag in der Warteschlange Q_j gebe es einen Ertrag C_j. Es gibt jeweils *n* Server pro Gruppe.

Mit diesen Definitionen und Zuordnungen wurde folgender Bienenalgorithmus konzipiert: Jeder Server S_i aus VS_j

- Arbeitet eine Anfrage der Warteschlange Q_j seiner Gruppe ab.
- Heftet mit Wahrscheinlichkeit *p=0,1* seine Anfrage ans schwarze Brett.
- Diese bleibt eine Zeitdauer D erhalten, die proportional zum Ertrag C_j mit dem Faktor a ist: $D = a\, C_j$
- Wenn es ein **Futtersucher** ist, liest er mit Wahrscheinlichkeit r_i eine andere Nachricht vom schwarzen Brett. Seine Profitrate P_i wird dazu mit der Anzahl R_i der Anfragen bei Server S_i der Servergruppe VS_j in der Zeit T_i errechnet.

$$r_i = f(P_i,P_c) \qquad \text{mit } P_i = \text{“Profitrate”} = C_j R_i / T_i \tag{10.8}$$

$$\text{und } P_c = \frac{1}{T_c}\sum_{j=0}^{m-1} C_j V_j \quad \text{mittlere Profitrate aller Gruppen}$$

  Die Profitrate P_c der gesamten Kolonie ergibt sich aus der Anzahl V_j der bearbeiteten Aufträge des virtuellen Servers VS_j im Zeitintervall T_c und der Erträge C_j. Die Wahrscheinlichkeit r_i errechnet sich dann aus dem relativen Profit $p_i = P_i/P_c$ des einzelnen Servers anhand einer vorgegeben Stufenfunktion $f(P_i,P_c) = f(p_i)$.

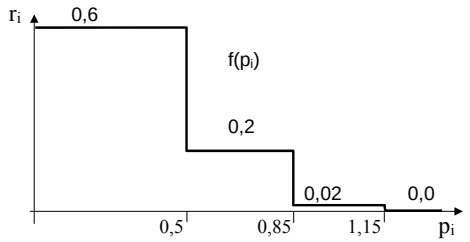


Abb. 10.3 *Wahrscheinlichkeit, andere Anfragen zu bearbeiten in Abhängigkeit vom relativen Ertrag*

  Man sieht, dass die Funktion $r_i = f(.)$ so konstruiert wurde, dass bei überdurchschnittlicher Profitrate nur die eigenen Anfragen bearbeitet werden. Ist die Profitrate aber zu gering, so werden auch Aufträge anderer Servergruppen mit großer Wahrscheinlichkeit übernommen.
- Wenn es der (eine) **Scout** der Gruppe ist, wird zufällig eine ganze Warteschlange einer anderen Gruppe ausgewählt.

Die Leistungen dieses Algorithmus wurde am Beispiel der Anforderungen innerhalb eines 24-Stunden-Intervalls für Webseitenaufruf an einen Pool von 50 Servern simu-

liert. Dabei wurde C_i = 0,5 cent/Anfrage, eine mittlere Antwortzeit von 15ms und eine Zuordnungszeit von 15 Min. angenommen. In Abb. 10.4 ist die zeitliche Abhängigkeit der mittleren Anzahl von Web-Anfragen pro Sekunde von drei verschiedenen Internetdiensten A,B und C gezeigt.

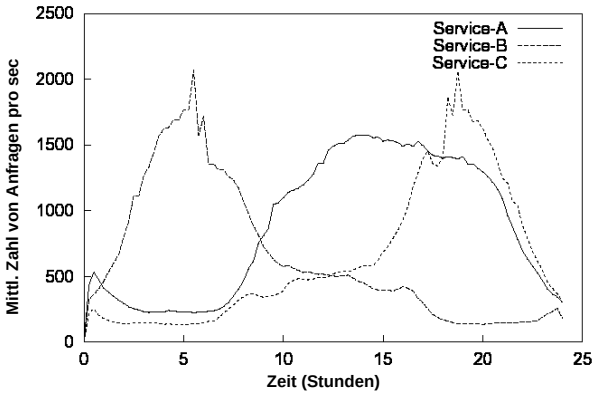


Abb. 10.4 *http-Anfrageverteilung, aufgegliedert nach Internetdiensten*

Aufgabe der Serverkolonie ist es nun, unter sich eine dynamische Lastverteilung der Anfragen zu erreichen. Dazu wurden jeweils für ein 2-Gruppen-System die Daten der Dienste B und C und für ein 3-Gruppen-System alle drei Dienste A, B und C verwendet. Außerdem wurde der Profit der Server für diesen Zeitraum ermittelt. Die Ergebnisse sind in Tabelle 1 gezeigt.

Algorithmus	**2 VS-Gruppen**	**3 VS-Gruppen**
Allwissend	1,066	1,337
Bienen-Alg.	1,050	1,238
Greedy-Alg.	1,043	0,818
Allwissend-statisch	0,845	1,108

Tabelle 1 *Leistungsvergleich von vier Algorithmen. Angaben in Mill.$*

Die vier Algorithmen im Vergleich waren

- *Allwissend* – ermittelt über aufwändige dynamische Programmierung die optimale Verteilung unter Einbeziehung der Migrationskosten. Dies ist in Echtzeit nicht praktikabel, dient aber als Vergleichsmaßstab, was optimal möglich ist. In Tabelle 1 hat dieser Algorithmus deshalb die besten Ergebnisse.
- *Bienen* – Der Bienen-Algorithmus weist keine großen Unterschiede zum theoretisch besten Algorithmus auf.
- *Greedy* – Wähle die Zuordnung, die in den vergangenen Zeitintervallen den meis-

ten Profit gebracht hatte. Wie in Tabelle 1 zu sehen, führt dies zwar zu guten Ergebnissen, aber der Algorithmus fängt sich manchmal in einem lokalem Optimum, aus dem nur eine Zufallsentscheidung wie beim Bienen-Algorithmus herausführen würde.

- *Allwissend-statisch* – Wähle die Zuordnung, die den Zeithorizont aus N Zeitabschnitten optimiert. Dazu werden einmal aus jeweils *N* Zeitabschnitten die optimalen Zuordnungen pro Zeitabschnitt mittels dynamische Programmierung errechnet. Dieser statische Ansatz ist machbar, da er immer nur die optimale Zuordnung für einen geringen Zeithorizont errechnen muss. Leider führt dieser Ansatz bei nichtwechselnden, statischen Anfrageverteilungen nicht zum optimalen Ergebnis. Wie wir in Abb. 10.4 und damit in Tabelle 1 sehen, führt dies auch bei dem gegebenen Problem zu deutlich suboptimalen Lösungen.

11 Simulationssysteme neuronaler Netze

In den vergangenen Jahren gab es viele Ansätze und Projekte, die Modelle und Algorithmen der neuronalen Netze durch besondere Maschinen und Programmiersprachen einer breiteren Öffentlichkeit nutzbar zu machen.

Der sinnvolle Einsatz einer bestimmten Hardware-Software Kombination hängt dabei von vielen Faktoren ab, die auf den ersten Blick ziemlich verwirrend wirken. In diesem Kapitel wollen wir uns deshalb die Merkmale von Rechnern und Programmen für die Simulation neuronaler Netze genauer ansehen.

11.1 Parallele Simulation

Obwohl die Modellvorstellung eines neuronalen Netzes als Funktionseinheiten bereits viele, parallel arbeitende Einheiten vorsieht, wird im Regelfall der neuronale Algorithmus nicht parallel auf einem speziellen Neuro-Chip ablaufen, sondern meist als Simulation oder Emulation ausgeführt, da weder das Chipdesign unproblematisch durchgeführt werden kann, noch über die grundsätzlichen Mechanismen und Algorithmen und damit über die grundsätzlich geeignete neuronale Architektur Einigkeit besteht. Forschung und Entwicklung gehen rasant voran; es vergeht bisher keine größere Konferenz zu diesen Themen, bei der nicht grundsätzliche, neuartige Ideen präsentiert werden. Aus diesem Grund empfiehlt es sich, sowohl für Grundlagenforschungen als auch für reelle Anwendungen, die in Frage kommenden Algorithmen in einer kontrollierten, mit Hilfsmitteln und Werkzeugen gut ausgestatteten, konventionellen Simulationsumgebung auf einem sequentiellen von-Neumann Computer auszutesten und die optimalen Parameter für das spezifische Problem zu bestimmen.

Funktioniert der Algorithmus erst zufriedenstellend für das Problem, so ist der weitere Schritt, ihn auf parallele Hardware zu übertragen oder in VLSI zu realisieren, nicht mehr ganz so problematisch. Vielfach reicht es für die Real-time Anwendung sogar aus, den Algorithmus in paralleler Version durch ein Netzwerk von konventionellen von-Neumann Prozessoren ausführen zu lassen.

Multiprozessor-Architekturen

Der Hauptunterschied in der Programmierung (und damit auch in der Benutzersicht) zwischen den verschiedenen Architekturen heutiger Multiprozessorsysteme ist die Behandlung von Daten, auf die alle Prozessoren unabhängig voneinander zugreifen müssen (*globale Daten*). Da der gleichzeitige Zugriff von vielen Prozessoren auf solche Daten (und Programme) sehr problematisch ist (s. z.B. [ALM89]), unterscheiden sich die Rechnerarchitekturen hauptsächlich in dem Grundansatz, dieses Dilemma von Datenzugriff und Datenkonsistenz befriedigend zu lösen.

Die Methoden dafür lassen sich in drei Ansätze aufteilen: den "Tanzsaal" Ansatz, bei dem jeder Prozessor sich ein Speicher-"Partner" wählen kann (vollständige Vernetzung), dem "Vorzimmer"-Ansatz, bei dem der Zugang zu einem Prozessor/ Speicher-Paar über einen gemeinsamen Kommunikationsweg ("Vorzimmer") erfolgt, und dem Netzwerk-Ansatz, bei dem jeder Prozessor nur einen Zugang zu einer lokal definierten, begrenzten Nachbarschaft hat, s. Abbildung 8.1.1.

Abb. 8.1.1 "Tanzsaal"- ,"Vorzimmer"- und Netzwerk- Architekturen

Am einfachsten läßt sich die Datenkonsistenz globaler Daten dadurch erreichen, daß man alle Komponenten miteinander direkt verbindet (volle Vernetzung der "Tanzsaal"- Architektur). Dies bedeutet aber einen großen Hardwareaufwand (Bei *N* Prozessoren und *N* Speichern $O(N^2)$ Verbindungen!) und ist nur schwer erweiterbar.

Das andere Extrem besteht darin, Prozessor-Speicher-Paare mit einem sehr niedrigen Vernetzungsgrad ("Vorzimmer"-Architektur) in einer Kette, in einem Ring oder in sternförmiger Art miteinander zu verbinden. In diesen Strukturen werden durch den indirekten Datenaustausch die Konsistenz der globalen Daten durch Nachrichtenaustausch erreicht, was einen relativ großen Software- und Zeitaufwand bedeutet.

Die "Netzwerk"-Architektur mit lokaler Kommunikation erlaubt sowohl nachrichten-orientierte Kopplungen im MIMD-Betrieb (*Multiple Instruction- Multiple Data Stream*) als auch einen synchronen, durch einen externen *Host*-Hauptprozessor gesteuerten SIMD-Betrieb (*Single Instruction- Multiple Data Stream*) beispielsweise durch ein Prozessorfeld (*systolische Felder*).

Die tatsächlich genutzten Multiprozessorsysteme verwenden meist Verbindungsarten, die in dem Aufwand zwischen der vollen Vernetzung und der Linienkette liegen. Betrachten wir im Folgenden eine Auswahl der gebräuchlichsten Netztopologien.

Singleprozessor-Architekturen

Die einfachste Hardware für Neurosimulatoren bildet der konventionelle Rechner mit einem Prozessor. Es existiert ein Spannungsfeld zwischen den Tatsachen, daß einerseits eine Applikation auf einem Multiprozessorsystem schneller "laufen" könnte, und andererseits der finanzielle und zeitliche Aufwand enorm ist, es für die Parallelverarbeitung komplett umzuschreiben. Es ist deshalb vielfach einfacher und billiger, für eine

Leistungssteigerung bei unverändertem Programm einen schnelleren Prozessor vorzusehen. Dieses Spannungsfeld zwischen Mono- und Multiprozessoranwendung wird etwas durch die Anwendung von *Koprozessoren* gemildert, die spezielle Programmteile, z.B. Fließkommaoperationen (*Floating-Point Coprocessor*) Matrixoperationen (*Vector-Coprocessor*) oder graphische Operationen *(Graphic boards*) bearbeiten. Die Umschaltung zu diesen Koprozessoren geschieht meist über spezielle Instruktionen (*Ausnahmebehandlung* von unbekannten Anweisungen) oder besondere Softwarebibliotheken.

Multiprozessor-Bus Architekturen

Läßt man mehrere Koprozessoren (Boards) an einem (Multi-Master) Systembus arbeiten, so erhält man ein System nach der "Vorzimmer"-Architektur. Hier gibt es einen Host-Prozessor, der alle Koprozessoren mit Arbeit versorgt (*Farming-Modell,* s. Abschnitt 8.1.2). In der folgenden Abbildung ist der Vorschlag für solch ein System

7 cm

Abb. 11.1 Ein Multi-Prozessor Neurocomputer (nach [HN86])

abgebildet; eine Implementierung davon ist beispielsweise das Mark III-System [TRE89].

Das leistungsbegrenzende Architekturmerkmal ist in diesem Fall der einzelne Broadcast-Bus, der eine Kommunikation der Koprozessoren untereinander empfindlich einschränkt.

11.1.1.1 Feldrechner und Hypercubes

Als ein interessantes Beispiel einer SIMD-Netzwerk-Rechnerarchitektur, die sehr viele Prozessoren sowohl in einer Gitterstruktur ("Aneinanderfügen" der Prozessoren-Chips zu einer großen Prozessor-Ebene durch direkte Datenleitungen an den vier Kanten, angedeutet durch die vier Pfeile in Abb. 8.1.3 links) als auch in einem nachrichtenorientierten Verbindungsnetzwerks eines Hypercubes (Abb. 8.1.3 rechts für 8 Prozessor-Chips gezeigt) einsetzt, zeigt sich die Connection Machine CM-1 [HILL85]. Sie wurde ursprünglich dazu entwickelt, konnektionistische, symbolorientierte Modelle und Programme effizient auszuführen. Bald jedoch zeigten sich als Hauptanwendungen der Einsatz in physikalischen Problemen (Finite Elemente, *ray tracing* etc), was zu einem Redesign der Maschine führte. Bei dem Nachfolgemodell CM-2 wurde nicht nur das I/O-System verbessert, sondern auch Fließkomma-Coprozessoren eingebaut, was zu einer theoretischen Höchstleistung von 2,5 GFLOPS führte.

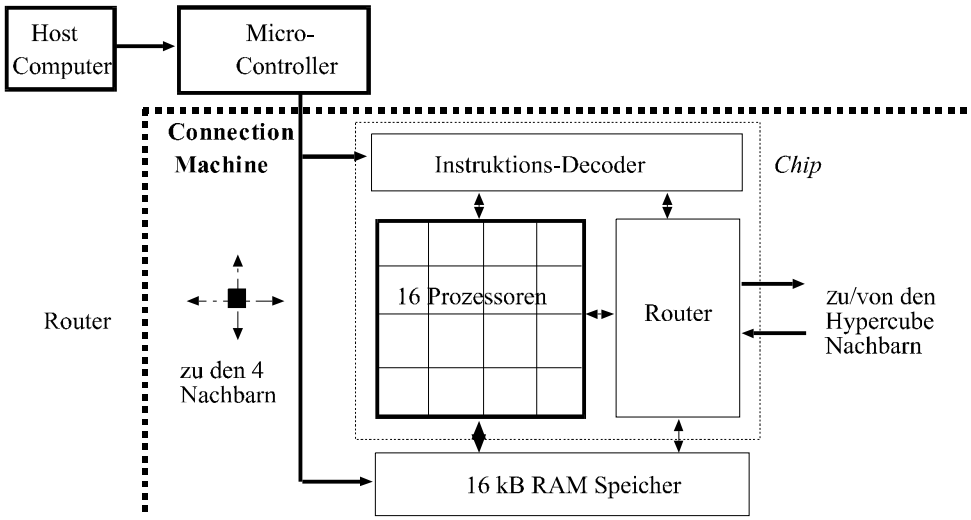


Abb. 8.1.3 Die connection machine und das Hypercube-Netzwerk

Die Programmierung der parallelen 1-Bit-Operationen wird durch die Existenz von Lisp, C und FORTRAN Compilern erleichtert. Die Menge der 1-Bit-Daten wird zu Vektoren ("xector") zusammengefaßt und es werden elementare Datenoperationen darauf ermöglicht, die durch die Compiler unterstützt werden. Beispielsweise läßt sich die Addition von Vektoren leicht komponentenweise lokal durch die den einzelnen Komponenten zugeordneten Prozessoren durchführen; die Summation (Betragsbildung der Vektoren) durch Addieren und Verschieben (routing) der resultierenden Daten zwischen den Prozessoren. Die zur Durchführung der Summation nötige Kommunikation zwischen den Daten wird damit auf die Kommunikation zwischen den Bits der Daten, also auf die Kommunikation zwischen den Prozessoren abgebildet. Dabei spielt es aber durchaus eine Rolle, ob die Kommunikation nur zwischen den Prozessoren im Gitternetzwerk oder mit erheblich höherem Aufwand zwischen zwei Prozessoren verschiedener Chips über das Hypercubenetzwerk stattfindet. Deshalb machen die meisten schnellen Algorithmen keinen Gebrauch von dem Hypercube-Netzwerk, sondern benutzen vorzugsweise die diskrete Nachbarschaft des Gitterrasters.

Neuronale Chips

Seit Beginn der Untersuchungen neuronaler Netze gibt es Versuche, die Funktion der Algorithmen direkt durch eine dedizierte Maschine ausführen zu lassen (s. z.B. Adaline, Abschnitt 2.2.2). Durch die gut entwickelte VLSI-Technologie gibt es in neuerer Zeit viele Versuche, neuronale Netze (Assoziativspeicher, Hopfieldnetze, Bolzmannmaschinen, Bildverarbeitungsnetze etc) in Hardware zu implementieren. Allerdings müssen dabei verschiedene Schwierigkeiten überwunden werden. Die beiden Hauptprobleme sind

- *Implementierung der Gewichte*
  Zwar können feste Gewichte leicht als PROMS fest implementiert werden, aber variable, für die Lernalgorithmen benötigten Gewichte bereiten etliche Schwierigkeiten [GOS87].

- *Implementierung der Verbindungen*
  Leitungen und Gewichte haben einen erheblichen Flächenbedarf auf Chips. Werden sehr viele Verbindungen benötigt, beispielsweise bei der vollständigen Vernetzung im Hopfield-Modell, so steigt der Flächenbedarf überproportional an. Hier benötigt ein neuronaler Chip der Firma AT&T, der 54 vollständig vernetzte Neuronen (Verstärker) beinhaltet, bereits 90% der Chipfläche nur für die Verbindungen [GRA88].

Im Gegensatz dazu besteht eine der interessantesten Proportionen der eigentlichen, neuronalen Prozessoren darin, daß sie oft direkt in einfacher, analoger VLSI-Technik realisiert werden können anstelle aufwendiger Digitaltechnik [CARD89], [VITT89a]. Dabei kann man die physikalischen Effekte der Analogtechnik ausnutzen, um nicht nur die Funktionen der formalen Neuronen zu approximieren, sondern sogar die Funktionen ganzer Schichten von Neuronen. Ein Beispiel dafür ist die Entmischung einer linearen Überlagerung unbekannter Signalquellen mit Hilfe eines Netzwerks [VITT89b].

Eine Übersicht und weitere Einzelheiten über neuere Hardwarearchitekturen ist im Buch [GLEP94] zu finden, wobei der Schwerpunkt auf die VLSI-Implementation gelegt wird.

Diskussion: Die Auswahl der Hardwarekonfiguration

Welche der bisher vorgestellten Architekturen ist nun am besten geeignet für neuronale Algorithmen ?

Für die Anwendung unterscheiden sich die Algorithmen hauptsächlich durch ihre unterschiedliche Benutzerfreundlichkeit und Flexibilität. Konventionelle Single-Prozessor Systemeerlauben meist nur mäßige Leistungen, aber eine einfachere Bedienung gegenüber den Multiprozessor-Systemen, die zwar schneller sind, aber nur unter Schwierigkeiten programmiert werden können. Bei den NeuroChips haben wir schließlich im Extremfall eine sehr hohe Leistung bei einem bestimmten Algorithmus, also bei geringer Flexibilität.

Die Wahl einer geeigneten Rechnerarchitektur hängt damit stark von der gegebenen Problemsituation ab. Im Wesentlichen lassen sich zwei Extremsituationen unterscheiden, zwischen denen der normale Benutzer seine Situation ansiedeln wird:

- Die Aufgabe lautet, zu einem gegebenen Problem einen gut funktionierenden, neuronalen Algorithmus als Lösung zu finden.

  Hier muß man den Erfolg verschiedener Algorithmen vergleichen und verschiedene Veränderungen am Algorithmus und seinen Parametern erproben. Dazu ist meist ein Universalrechner besser geeignet, da die Ausstattung mit Grafikbildschirmen, Haupt- und Massenspeichern, Druckern etc. durchweg besser und der Softwareanschluß unproblematisch ist.

- Der Algorithmus und seine Parameter stehen fest; gesucht wird die schnellste Ausführungsmöglichkeit.

  Hier muß ein Kompromiß zwischen Ausführungszeit und Hardware kosten ausgearbeitet werden. Die preiswerteste Lösung ist sicher ein schneller Prozessor (z.B. Signalprozessoren, RISC-Chips) oder die einfache Kopplung weniger Prozessoren (z.B. Transputerring, s. Abb. 8.1.4). Erst bei größeren Anforderungen

an das zeitliche Verhalten lohnen sich Architekturen wie die Connection-machine oder speziell gefertigte Neuro-Chips.

Im allgemeinen Fall wird man die bewährte Strategie der üblichen Entwicklungssysteme übernehmen: Auf einem normalen Rechner (*Host*) wird das Programm geschrieben, getestet und compiliert und dann als binäre Daten über eine Leitung auf den Spezialrechner übertragen (*download*). Kommunikationsmechanismen und Programm-Monitore sorgen dort für eine Überwachung und evtl. Debugging. Dabei kann der Spezialrechner durchaus physisch im Hostcomputer integriert sein (z.B. als Coprozessor-Board).

Partitionierung der Algorithmen

Welche Probleme stellen sich bei der Programmierung der neuronalen Algorithmen auf den Multiprozessoranlagen ?

Betrachtet man die in den bisherigen Kapiteln vorgestellten Algorithmen, so bemerkt man, daß viele Algorithmen von neuronalen Netzen eine Mischung aus sequentiellen und parallelen Konstrukten darstellen. Dabei läßt sich bezüglich ihrer Wechselwirkungen (Kopplungen) mit Hey [HEY87] folgende Einteilung vornehmen:

Wechselwirkungsfreie Parallelarbeit

Betrachten wir zunächst die Algorithmen, bei denen die Daten in Bereiche (packets) aufgeteilt unabhängig voneinander bearbeitet werden können. Diese Aufgabe kann leicht von einem Hauptprozessor (*master* oder *farmer*) erledigt werden, der mehr Aufgabenpakete schnürt, als Prozessoren verfügbar sind. Immer dann, wenn der beauftragte Prozessor (*worker*) sein Ergebnis abliefert, bekommt er eine neue Aufgabe. Beispiele für dieses *farming* sind die Funktion einer Perzeptron-Schicht oder die Algorithmen zur Berechnung von Lichtspiegelungen -und brechungen in einer gegebenen Szene (*ray tracing*).

Sind weniger Aufgabenpakete als Prozessoren da, so läßt sich der Durchsatz dadurch erhöhen, daß die Ergebnisse als neue Pakete für die nach-

folgende Bearbeitung den freien Prozessoren übergeben werden (*pipeline*).

Nachbarschaftsabhängige Parallelarbeit

Bei vielen Algorithmen hängt die Lösung des Problems nicht nur von den eingegebenen Datenpunkten ab, sondern auch von den Lösungen, die für die unmittelbaren Nachbarpunkte gefunden werden. Bei diesem Problem läßt sich eine gute Prozessorauslastung dadurch erreichen, daß jeweils eine zusammenhängende Datenregion einem einzelnen Prozessor übergeben wird. Die aus der Nachbarschaft benötigten Daten am Rande der Regionen müssen über die Interprozessorkommunikation ausgetauscht werden.

Beispiele dafür sind die parallelen Algorithmen der topologieerhaltenden Abbildungen aus Abschnitt 2.4 oder die numerische Integration partieller Differenzialgleichungen.

Vollvernetzte Parallelarbeit

Werden im Algorithmus die Wechselwirkungen großer Teile des Systems berechnet, wie dies beispielsweise für den autoassoziativen Speicher, die Hopfield-Netze oder das *N*-Körper Problem der Astrophysik bzw. das Wechselwirkungsproblem bei der Molekülmodellierung durch Atome nötig ist, so lassen sich erfahrungsgemäß gute Resultate erzielen, wenn die Neuronen, Körper oder Atome gleichmäßig den Prozessoren zugeteilt werden und die Prozessoren innerhalb einer Ringschaltung die Eingangsgrößen und Ausgangsgrößen zirkulieren lassen [HILL87].

Zur Veranschaulichung sei in Abbildung 8.1.4 die Hardwarestruktur eines Transputerrings zur Modellierung eines biologischen Photosynthesemoleküls wiedergegeben. Von den vier seriellen Verbindungen jedes Transputers sind je zwei als Eingänge und zwei als Ausgänge geschaltet. Durch die entsprechende Verbindung mit den Nachbartransputern entstehen zwei gegenläufig zirkulierende Pipelines; eine für die Eingabegrößen (Koordinaten der Atome) und eine für die Ausgabegrößen (resultierende Kräfte).

Abb. 8.1.4 Hardwarestruktur für eine Molekülmodellierung (nach [GRUB88])

Andere Ansätze sehen kompliziertere, hierarchische Kommunikationsmuster vor [HILL87]. Die Softwarestruktur geht von einem Auftraggeber (T414) aus, der auch die Ergebnisse wieder einsammelt und bei Fehlern aktiv wird und den einzelnen Transputerknoten, die sowohl ihre eigenen Ergebnisse verschicken als auch die der Nachbarn weiterreichen.

Abb. 8.1.5 Softwarestruktur eines Transputer-Knoten (nach [GRUB88])

11.1.1.2 Parallelisierung der Simulation

Bei diesen grundsätzlichen Überlegungen wird allerdings nichts darüber ausgesagt, wie ein konkreter Algorithmus mit seinem neuronalen Netz tatsächlich partitioniert werden soll. Betrachten wir beispielsweise den viel verwendeten Back-Propagation Algorithmus (s. Abschnitt 2.3). Er besteht aus mehreren Schichten eines Feed-forward Netzes, bei der jede Schicht mit der nachfolgenden vollvernetzt ist. Im Unterschied zum Multi-Layer Perzeptron (s. Abschnitt 2.1.1) ist der Lernalgorithmus jeder Schicht durch den jeweils erzielten Fehler (Differenz zwischen gewünschtem und tatsächlichem Ergebnis) in einer stochastischen Approximation gegeben.

Es gibt nun verschiedene Möglichkeiten, diesen Algorithmus auf Multiprozessorsysteme zu verteilen. Ohne auf Details einzugehen sind in Abbildung 8.1.6 zwei von drei prinzipiellen Möglichkeiten schematisch gezeichnet, um ein dreischichtiges Netzwerk zu parallelisieren.

Abb.10 Aufteilungen des Back-Propagation Algorithmus

Die erste Aufteilung ordnet jede Schicht einem Prozessor zu, der in einer bidirektio nalen Pipeline mit den anderen Prozessoren verbunden ist. Jedes Trainingsmuster, das eingegeben wird, löst eine Aktivität aus, die bis zur Ausgabe geht und als Fehler durch die Schichten bis zur Eingabe zurückgeführt wird. Erst nach Eingabe aller Trainingsmuster werden die Gewichte verändert.

In der *zweiten* Aufteilung werden die bidirektionalen Aktivitäten getrennt und auf extra Prozessoren verteilt, so daß eine Prozessorstruktur ähnlich einem Hypercube entsteht.

Die *dritte* Aufteilung partitioniert die Trainingsmuster auf die Prozessoren und führt die Iterationen aller Schichten für diese Teilmenge auf jeweils dem selben Prozessor durch (farming).

In ihrer Arbeit [CECI88] zeigten L.Ceci, P.Lynn und P.Gardner, daß jede Aufteilung für eine bestimmte Hardwarekonfiguration geeignet ist; dieses Schema ist in Abbildung 8.1.7 gezeigt. Als Kommunikationskosten wurden die in einem INTEL Hypercube angenommen.

Abb.10 Optimale Aufteilung des Algorithmus (nach [CECI88])

Die Betrachtungen zeigen, daß ein universell-optimales Verteilungsschema nicht einmal für einen einzigen Algorithmus eindeutig existieren muß, so daß man noch viel weniger *eine* Hardware/Software Kombination bestehender Systeme als optimal für *alle* Anwendungen ansehen darf. Für jede Anwendung, jeden Algorithmus und jedes Multiprozessor-System muß deshalb individuell eine günstige Aufteilung gefunden werden. Dabei ist es sicher einfacher, für (fast) wechselwirkungsfreie Parallelarbeit (z.B. genetische Algorithmen, s. [BET76],[SPI90]) eine optimale Aufteilung des Algorithmus zu finden, als für vollvernetzte Parallelarbeit, da der größte Arbeitsaufwand neuronaler Algorithmen - wie Untersuchungen an der Connection Machine zeigen [DEP89] - in der Kommunikation der Prozessoren untereinander besteht.

Darüber hinaus zeigten die Studien von Murre [MUR93], daß die Topologie der Prozessorverbindungen bei vollständig oder zufällig verbundenen neuronalen Netzen nur wenig die Kommunikationskosten senken kann. Viel wichtiger erwies sich die Tatsache, ob die neuronalen Netze modular mit nur lokaler Kommunikation organisiert waren oder nicht. Er schloß daraus, daß für eine sinnvolle Ressourcenorganisation auch im menschlichen Gehirn die Funktionen modular organisiert sein müßten.

11.2 Sprachsysteme und Simulationsumgebungen

Für die Simulation neuronaler Netze gibt es eine Vielzahl von Simulationsumgebungen, die sich durch verschiedene Merkmale unterscheiden. Fast jeder, der sich ernsthaft mit neuronalen Netzen und deren Simulation beschäftigt und über Computerkenntnisse verfügt, schreibt sich seine eigenen Simulationsprogramme und neuronale Simulatoren. Ohne den Sinn oder Unsinn solcher Entwicklungen zu diskutieren, wollen wir uns

fragen: Was macht denn eine gute Simulationsumgebung nun aus ? Für die Beantwortung dieser Frage betrachten wir zunächst den Gegenstand, der modelliert werden soll, etwas näher.

Die Anforderungen

Als Beispiel für die Modellierung und Simulation neuronaler Strukturen ist das folgende Schema des visuellen Kortex einer Katze abgebildet. Wie würde sich diese vorgeschlagene Architektur zur Prüfung ihrer Eigenschaften am besten simulieren lassen?

Abb.10 Schema des visuellen Katzen-Cortex (aus [SEG82])

Es ist meist sehr zeitaufwendig und mühsam, ein Konzept einer neuronalen Architektur direkt zur Erprobung in VLSI-Technik zu realisieren oder als Multiprozessorprogramm bis ins Detail auszuprogrammieren. Spielen die Ausführungszeiten gegenüber den qualitativen Merkmalen des gewünschten Algorithmus eine untergeordnete Rolle, so läßt sich eine neuronale Architektur am schnellsten zunächst als Simulation auf einem herkömmlichen von-Neumann Computer realisieren.

Komponenten der Simulation

Alle Simulationsprogramme für neuronale Netze müssen ähnliche Funktionen erfüllen, da die zugrunde liegenden Algorithmen meist den Standard-Algorithmen entsprechen, die in den Kapiteln 1 bis 5 vorgestellt wurden. Alle Simulationen müssen dazu Trainings- und Testmuster entwerfen, diese dem Algorithmus zur Verfügung stellen und die Ergebnisse anzeigen. In Abbildung 8.2.2 ist eine solche typische Architektur gezeigt.

Abb.10 Daten- und Kontrollfluß einer typischen Simulation

Alle Komponenten bestehen selbst wiederum aus einzelnen Modulen. Will man nun die einmal geschriebenen Module in den verschiedenen Simulationen wiederverwenden (*reusable software*), so bringt man sie günstigerweise in einer besonderen Modulsammlung (*module library*) unter. Untersuchen wir nun die Funktion der einzelnen Komponenten etwas näher.

Die Simulationskontrolle

Schreibt man ein Simulationsprogramm für einen Algorithmus, nachdem man bereits ein Simulationsprogramm für einen anderen Algorithmus geschrieben hat, so bemerkt man sehr schnell, daß bestimmte Funktionen sich wiederholen und deshalb von dem ersten Programm übernommen werden können. Dies sind insbesondere die Kontrollfunktionen wie *Simulation anhalten, Parameter verändern, Gewichte initialisieren, speichern, verändern, Simulation weiterführen* etc. Diese Funktionen lassen sich deshalb leicht herausziehen und in der *Simulationskontrolle* vereinen. Beispielsweise besteht das NeuralShell [AHA90] System aus einer Ablaufkontrolle, von der aus die verschiedenen Algorithmen (Hopfield-Netz, Backpropagation, etc.) als Befehle (Programme) aufgerufen werden können.

Der Trainings/Testmuster-Generator

Bekannterweise besteht die Programmierung einer neuronalen Funktion nicht nur aus der Programmierung des neuronalen Algorithmus (neuronalen Netzarchitektur), sondern auch aus dem Training des Algorithmus mit Trainingsdaten und anschließendem Test des trainierten Systems mit (möglichst unabhängigen) Testmustern. Da zum einen reale Daten (z.B. Videodaten, Sprachmuster, Sensorwerte) nicht immer verfügbar sind und zum anderen auch Systeme für die Verarbeitung von Real-Welt Daten erst getestet und "justiert" (Parametereinstellung) werden müssen, werden oft "künstliche", unter kontrollierten Bedingungen generierte Daten benötigt. Dies sind beispielsweise aus Einzelpixeln zusammengesetze

Buchstaben (s. Abb. 3.1.5), mit der Hand korrigierte Sprachdaten oder allgemeine Mustervektoren, die bestimmten Bedingungen (Orthogonalität etc., s. z.B. Abb. 3.1.3) gehorchen müssen. Auch Zufallsmuster (s. z.B. Abb. 1.5.1.) gehören in diese Kategorie.

Diese Muster werden mit einem speziellen Programm, dem Muster-Generator erzeugt, das meist graphisch-interaktiv funktioniert und über Einzelfunktionen verfügt, die auf die Art der zu generierenden Daten zurechtgeschnitten sind. Im on-line Betrieb werden die Muster direkt erzeugt (z.B. multidimensionale Zufallszahlen) und dem neuronalen Algorithmus eingegeben; im off-line Betrieb werden die erzeugten Muster (z.B. Buchstabenpixel) zunächst auf einem Massenspeicher für den späteren Bedarf abgespeichert.

Display

Da es für Menschen sehr schwierig ist, Zahlentupel direkt zu beurteilen, gewinnt die Darstellung der Ausgabemuster z.B. als Klassenentscheidung, Prototypen, korrigierte Eingabemuster und ähnliches eine große Bedeutung. Viele Algorithmen sind erst durch ihre gute Visualisierung bekannt geworden (vgl. z.B. [KOH84]). Meist ist die visuelle Darstellung (Display) der Ausgabemuster eine inverse Funktion der Musterkodierung: Hier werden beispielsweise aus eindimensionalen Zahlentupeln wieder zweidimensionale Pixelbilder gemacht.

Resultate

Eng an die visuelle Darstellung der Muster ist auch die Speicherung und Darstellung der Resultate angelehnt. Üblicherweise wird man hier die Genauigkeit der Funktions-Approximation ("Lernkurven") durch das neuronale Netz (z.B. mit dem mittleren Fehlerquadrat), den Verlauf von Parameterwerten mit der Zeit, die Darstellung der Gewichte (z.B. durch Hinton-Diagramme, s. Abb. 4.2.6), die Histogramme über Musterklassen und dergleichen mehr verdeutlichen. Ebenso wie dem Display kommt dieser Visualisierung eine nicht zu unterschätzende Bedeutung bei der Simulation zu.

Der *Netzwerk*-Editor

Die Wiederverwendung des neuronalen Algorithmus wird entscheidend gefördert, wenn das Programm nicht nur aus einem monolithischen Block, sondern aus kleineren Einheiten besteht. Vielfach werden die neuronalen, semantischen Einheiten (Neuronen, Schichten, Cluster, Verbindungen) auf syntaktische Einheiten einer *Netzbeschreibungssprache* abgebildet. Damit werden die Einheiten zu *virtuellen Maschinen*, die portabel sind und leicht auf unterschiedlichen Multiprozessorsystemen implementiert werden können. Für jede unterschiedliche Hardware (Prozessornetz, Neuro-Chip etc.) muß dann nur noch ein Treiber erstellt werden, der das Zwischenformat (*intermediate code*) der Beschreibung des neuronalen Netzes interpretiert bzw. ausführt. Aufgabe des *Netzwerk-Editors* ist es, die Erstellung einer solchen Netzwerkbeschreibung zu unterstützen.

Die Netzwerkbeschreibung

Für die Netzbeschreibungssprache gibt es hauptsächlich zwei Ansätze: die Beschreibung durch *Programmiersprachen* und die *graphische Programmierung*. Betrachten wir zunächst den konventionellen Ansatz, die Funktionseinheiten der neuronalen Netze mittels spezieller Datentypen in bereits bestehende Sprachen einzubinden. Hier gibt es verschiedene Sprachsysteme wie das prozedurale, C-orientierte Pygmalion [AZE90] oder das MetaNet-System [MUR90], das Lisp-orientierte Spread-3 [DIE87], das deklarative Nesila [KOR89], funktionale Programmierung [KOO90] oder den bekannten RCS Simulator [GOD87], die spezielle Sprachkonstrukte zur Beschreibung der Einheiten und ihrer Verbindungen untereinander bereitstellen. Für jede Spracherweiterung gibt es einen Compiler oder Präprozessor, der die Spracherweiterungen in Programmkonstrukte der ursprünglichen Sprache übersetzt, so daß der resultierende Quellcode auf jedem Prozessortyp verwendet werden kann, sofern dort ein Compiler oder Interpreter für diese Sprache existiert. Der Netzwerk-Editor besteht in diesem Fall aus einem reinen Texteditor, der eine manuelle Modifikation der textuellen Netzwerkbeschreibung gestattet. In

Abbildung 8.2.3 ist als Beispiel die Beschreibungshierarchie in Pygmalion zu sehen.

Diese Art der Netzwerkspezifikation ist allerdings sehr mühsam und außerdem (gerade bei größeren Netzen) sehr fehleranfällig. Der Ansatz läßt sich mit dem Assembler-Konzept der Maschinenprogrammierung vergleichen: jedes Detail der

Abb. 8.2.3 Fenstersichten und Datentypen in Pygmalion (nach [TRE90])

neuronalen Maschine muß genau beachtet und ausprogrammiert werden, was bei größeren Programmen (s. obige Abbildung 8.2.1!) selbst bei Anwendung von Modultechnik leicht zu Inkonsistenzen und Fehlern führt.

Demgegenüber ist bei einer *graphischen Programmierung* sofort zu sehen, wie die vorgegebenen Einheiten verbunden sind. Die grundsätzlichen Befehle des graphischen Editors wie *move, copy, insert, delete, help, ...* sollten orthogonal [SMI82] sein und sollten Operationen auf einer beliebigen Teilmenge der Netze bzw. Subnetze erlauben. Damit läßt sich nur durch Anwendung der Kopier- und Verbindungsoperationen aus einer Kollektion vordefinierter Standardeinheiten (Standardalgorithmen) leicht ein spezielles Netzwerk konstruieren. Dies ist eine graphische, interaktive Konfiguration bestehender Module im Sinne einer graphischen Programmierung und ermöglicht so eine Simulation eines Netzes auf verschiedenen Betrachtungsebenen. In der folgenden Abbildung 8.2.4 ist als Beispiel die graphische Spezifikation des Datenflusses von einer Videokamera, künstlich überlagert durch Störrauschen, über ein Bildverarbeitungssystem ("decode blocksworld") zu Anzeige- und Protokolleinheiten gezeigt.

Bei der graphischen Programmierung läßt sich beispielsweise das Netzwerk aus Abbildung 8.2.1, das aus Schichten, Arealen, Funktionsgruppen und Neuronen besteht, sowohl auf verschiedenen Ebenen als hierarchisches Subnetz realisieren, bei dem die virtuellen Einheiten mit den Arealen, Funktionsgruppen und Neuronen identisch sind, als auch im Ganzen als eine virtuelle Einheit, bei dem alle Netzverbindungen zwischen Einzelneuronen per Hand am Bildschirm programmiert sind. Wel-

cher Grad zwischen den beiden Unterteilungs-Extremen gewählt wird, hängt von verschiedenen, benutzertypischen Faktoren ab. Anfangs wird man nur solche Funktionseinheiten als virtuelle

Abb.10 Graphische Spezifikation einer Bildverarbeitung

Einheiten wählen, die in ihrer Funktion gut bekannt sind und das unbekannte Zusammenspiel der bekannten Einheiten über Monitor- und Protokolliermöglichkeiten beobachten. Speziell bei neuronalen Netzen werden zum *Debugging* besondere Einheiten benötigt, die den von einer Einheit ausgegebenen Code wieder invertieren, also 'lesbar' machen [KIN89].

Eine dritte Möglichkeit der Programmierung besteht darin, beide Ansätze miteinander zu verbinden und die textuelle Beschreibung des Netzes nur als "low level" Zwischensprache zu sehen, von der bei Bedarf zu einer graphischen, interaktiv editierbaren Repräsentation übergewechselt werden kann (und zurück), wie es beispielsweise in MetaNet [MUR90] vorgesehen ist.

Ein anderer Aspekt ist die Unterstützung der *Verteilung* der virtuellen Einheiten (Neuronen, Schichten, Trainingsmuster), wie sie im vorigen Abschnitt 8.1.2 diskutiert wurde, auf vorhandene Hardware. Existieren im System spezielle neuronale Chips, beispielsweise als Coprozessoren, oder Netzzugänge zu Multiprozessorsystemen, so formuliert man günstigerweise die Netzhierarchie derart, daß die Abbildung der virtuellen Einheiten auf diese Hardware durch eine passende Einbindung bestimmter, vordefinierter Einheiten (Treibersoftware zum Ansprechen der Hardware) unterstützt wird. Entsprechend modifizierte Simulatoren können dann die Verteilung der Einheiten auf die Multiprozessorkonfiguration vornehmen.

Die kommerziell erhältlichen, graphikorientierten, neuronalen Simulatoren wie MacBrain™, Cognitron™, ExploreNet 3000™, Plexi usw. genügen leider meist nicht allen angesprochenen Kriterien. Da es häufig nur fertige, binäre Versionen angeboten werden, genügen sie durchweg nur am Anfang den vom Käufer gestellten Erwartungen. Sobald man

neue Architekturen, neue Displayarten oder neue Hardware integrieren will, stößt man auf das Problem mangelnder Flexibilität. Außerdem beinhalten sie aber neben der Schwierigkeit, die gewünschte Architektur auch mit den fest eingebauten, vorhandenen Grundfunktionen realisieren zu können (obwohl sie teilweise auch hierarchisch gegliederte, neuronale Netzwerke zulassen), auch das Problem der mangelnden Effizienz (Simulationsgeschwindigkeit) bei großen neuronalen Netzen.

Möchte man die Vorteile modularer, graphischer Programmierung mit ihren Nachteilen abwägen, so fallen folgende Punkte positiv auf:

- Die Netzspezifikation und Programmierung ist interaktiv graphisch möglich
- Die Wiederverwendung (*reusability*) der Software wird gefördert
- Die Einbindung von benutzerspezifischer, paralleler Hardware ist relativ einfach

Demgegenüber ist Folgendes zu bedenken:

- Es ist *keine* feinabgestufte, zeit- oder ereignisgesteuerte Simulation wie beispielsweise mit SIMULA möglich, da der gesamte Ablauf eines Moduls als ein Zeitschritt betrachtet wird und externe Ereignisse sich nur in dem Programmteil auswirken können, das gerade die Prozessorkontrolle hat.
- Die Verteilung der Prozesse in Multiprozessorsystemen wird, durch die graphi sche Netzspezifikationen vorgegeben, vom Simulationssystem durchgeführt. Die günstigste Aufteilung des Algorithmus muß vom Benutzer selbst vorgenommen werden und wird nicht automatisch generiert.

  Dies ist bei kleineren Systemen sicher der effektivste Ansatz, um einen gegebenen Algorithmus auf spezielle Hardware (Multi-Transputersysteme etc) zu verteilen; bei größeren Systemen ist dies aber voraussichtlich zu fehleranfällig und zu arbeitsintensiv. Hier ist die fein "granulierte" Einteilung durch entsprechende Compiler für Netzbeschreibungen in OCCAM oder paralleles C von Vorteil gegenüber der grafischen Methode.

Zusammenfassend kann gesagt werden, daß neuronale Simulatoren die heutige Problematik der Programmierung paralleler Hardware gut widerspiegeln. Hier zeigen sich alle Probleme, ursprünglich parallele Algo-

rithmen effektiv und zugleich benutzerfreundlich sowohl zu beschreiben als auch auszuführen.

Eine besondere Bedeutung zur Lösung dieser Problematik kommt dabei den Ansätzen der visuellen Programmierung zu, wie sie beispielsweise in dem Buch von Shu [SHU88] ausführlicher erläutert werden.

11.3 Simulationstechniken

Möchte man in ein gegebenes Simulationssystem um eigene Module erweitern oder etwa kein vorgefertigtes Simulationssystem verwenden, so stößt man früher oder später auf ganz bestimmte, praktische Probleme, von denen hier einige näher erörtert und Tips zu ihrer Lösung oder Umgehung gegeben werden sollen.

Stochastische Mustererzeugung

In vorigen Abschnitten sahen wir, wie man stochastische Muster klassifizieren und wiedererkennen kann. Um solche Algorithmen und Verfahren durchzuführen und ihre Wirksamkeit (Fehlersuche in den Simulationsprogrammen!) zu testen, benötigen wir allerdings erst einmal Muster. Angenommen, wir verfügen nicht von vornherein über die hunderttausende von Musterbeispielen, um eine gute stochastische Konvergenz zu erreichen: Woher bekommen wir dann die Muster?
Dazu unterscheiden wir zwei Fälle: normierte Muster zu Testzwecken und Muster aus Anwendungen.

Synthetische Muster

Angenommen, wir wollen die Eigenschaften eines Algorithmus zur Mustererverarbeitung und -erkennung zeigen oder die Fehler in einem Simulationsprogramm finden. Dann ist es sehr sinnvoll, nur solche Muster einzugeben, deren statistische Proportionen (und damit die Auswirkungen) man sehr genau kennt. Zur Mustererzeugung eignet sich damit beispielsweise die Belegung aller Komponenten eines Mustervektors $\mathbf{x}$ mit gleichverteilten Zufallszahlen zwischen Null und Eins, wie sie in den meisten Programmierbibliotheken schon vorhanden ist. Aus diesen läßt sich dann im Prinzip über die *Monte-Carlo Methode* (s. z.B. [Yak77]) die

Gleichverteilung in (fast) jede gewünschte Verteilung transformieren. Bei einer der wichtigsten und auch am meisten verwendeten Verteilungen, der Gauß-Verteilung, hat man allerdings Schwierigkeiten, die dafür benötigte Fehlerfunktion analytisch zu errechnen. Anstelle nun eine tabellierte, numerisch bekannte Version zu verwenden, gibt es einen einfachen Trick, um Gauß-verteilte Zufallszahlen zu errechnen. Nach dem zentralen Grenzwertsatz wissen wir, daß die Verteilung für die zentrierte Summenvariable x_s zur *Normalverteilung* N(0,1)

$$\lim_{n\to\infty} x_s(n) = N(0,1) \qquad \text{mit } x_s(n) = \sum_{i=1}^{n}(x_i - c)/\bar{\sigma} \qquad (8.3.1)$$

konvergiert, wobei die mittlere Varianz aus den einzelnen Varianzen mit

$$\bar{\sigma} = (\sigma_1^2+...+\sigma_n^2)^{1/2} = n^{1/2}\sigma = n^{1/2}\,\big(\,\langle x_i^2\rangle - c^2\,\big)^{1/2} \qquad (8.3.2)$$

bei gleicher Varianz σ der Zufallswerte x_i gegeben ist.

Angenommen wir verfügen über einen Zufallsgenerator der uns gleichverteilte x aus dem Intervall [-d,+d] mit dem Mittelwert c=0 liefert (s. Codebeispiel 8.3.1). Dann ist die Wahrscheinlichkeitsdichte p(x)=1/2d und die erwartete Varianz von x_s ist

$$\bar{\sigma}^2 = n\,\langle x_i^2\rangle = n\int_{-d}^{+d} p(x)\,x^2 dx = n\,d^{2/3}$$

Eine vorgegebene Varianz σ läßt sich also mit einer Summe von $n=3\sigma^2/d^2$ gleichverteilten, zentrierten Zufallszahlen erreichen; eine Normalverteilung (Varianz der Größe eins im Ausdruck (8.3.1)) wird bei $x_i \in [-1/2,1/2]$ erreicht, wenn n=12 ist. Damit haben wir einen einfachen Algorithmus gefunden, normalverteilte Zufallsvariable zu generieren; im Codebeispiel 8.3.1 ist dies verdeutlicht.

```
VAR seed:CARDINAL;      (* Stat. Variable, muß anfangs init. werden *)

PROCEDURE Random():REAL;
      (* generiert einen Zufallswertgleichverteilt aus [0,1]
         mit dem "linear congruential random generator" (s.
[YAK77]) *)
CONST a=4; b=25543; modulus=7415;
```

```
VAR random:REAL;
BEGIN
    seed := (a * seed + b)MODmodulus;
    random := FLOAT(seed)/FLOAT(modulus);
RETURN random
END Random;

PROCEDURE GaussRandom(c,sigma:REAL):REAL;
      (* generiert einen normalverteilten Zufallswert
         mit Mittelwert c und Streuung  sigma nach (8.3.1) *)
VAR n,sum:REAL;
BEGIN
    sum:=0;
    FOR i:=1 TO 12 DO
     sum :=sum + Random() - 0.5
    END;
RETURN sum*sigma + c;

END GaussRandom;
```

Codebeispiel 8.3.1 Erzeugung von Zufallszahlen

Möchte man anstelle einer normalverteilten Zufallsvariablen z mit $\sigma=1$ eine Normalverteilung mit anderer Streuung erhalten,

$$p(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2} \rightarrow p(x,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-c)^2}{2\sigma^2}}$$

so ist es ungünstig, die Anzahl der Summanden zu verändern; bei geringer Streuung könnte weniger als ein Summand übrigbleiben. Stattdessen ist es besser, die Transformation von $z = x/\sigma$ auf x dadurch zu gehen, daß der normalverteilte Zufallswert z mit der Varianz zu $x = z\sigma$ multipliziert wird. Für schmale Verteilungen „stauchen“ wir durch die Skalierung die Normalverteilung oder „ziehen“ sie für breite Verteilungen auseinander.

Trainings- und Testmuster

Für die mustererkennenden Systeme ist es üblich, die Gesamtmenge der bekannten Muster in zwei willkürlich zusammengestellte Mengen aufzuteilen: in die Menge der Muster, mit denen das System trainiert wird und in die Menge der Muster, mit denen das System nach dem Training getestet wird (*cross-validation*). Meist ist die Erkennungsleistung bei den

unbekannten Testmustern schlechter als bei den bekannten Trainingsmustern.

Angenommen wir verfügen aber nur über wenige, bekannte Musterdaten der Anwendung (z.B. 15 Stück) und möchten trotzdem das System stochastisch lernen lassen, obwohl die Anzahl für eine Aufteilung zu klein ist. Dann gibt es mehrere statistische Tricks, um aus den wenigen Daten viele zu erzeugen, die (fast) gleiche statistische Eigenschaften haben [EF82]. Eine einfache Variante der cross-validation besteht darin, Trainingsmengen aus allen möglichen Mengen-Halbierungen zu gewinnen und dann diese sequentiell einzugeben. Ist der Algorithmus empfindlich für die Reihenfolge der Mustereingabe (unterschiedliche zeitliche Bewertung der Muster), so führt dies zum gewünschten Ergebnis: unsere "synthetische" Statistik hat fast die gleichen Eigenschaften (Streuung, Mittelwert, Korrelationen etc) wie es eine "echte", tatsächlich von der Anwendung gewonnene Statistik hätte.

Andere Möglichkeiten sind die Auswahl nicht der Hälfte, sondern einer kleineren Trainingsmenge; nimmt man alle bis auf jeweils ein weggelassenes Muster, so wird dies als "Taschenmesser"Verfahren (*jack-knife*) bezeichnet. Auch die Partitionen müssen nicht alle gleich groß sein.

Eine andere Idee ist das "Münchhausen" (*bootstrap*)-Verfahren, bei dem alle Daten sehr oft (z.B. 10^6 mal) vervielfältigt werden. Wählen wir davon eine Untermenge an Mustern aus, in der auch ein Muster mehrmals vorkommen kann, so haben wir wieder eine synthetische Traings- bzw. Testmenge erhalten [DIA83].

Darstellung der Ergebnisse

Eine der wichtigsten Aufgaben bei einer Simulation liegt in der plausiblen, überzeugenden Darstellung der Ergebnisse. Dies ist in der Regel in der Form einer Grafik, weniger in Form von Zahlentabellen. Beispiele dafür sind

- Grafische Darstellung der Eingabemuster, z.B. Eingaben $\{\mathbf{x}\}$ in Form von Buchstaben, Bildern oder Musterwolken. Sinnvoll ist z.B., visuell einfach in der Ebene darstellbare, 2- oder 3-dim Eingaben.

- Grafische Darstellung der Ausgabemuster, z.B. als Funktionswert (3.Dimension) von zweidimensionalen Eingaben.
- Visualisierung der Gewichte bzw. der Gewichtsentwicklung. Dies kann absolut als Muster im Eingaberraum geschehen wie z.B. in Abb. 2.4.9, als Helligkeitswert in Matrixform (Hinton-Diagramm, s. Abb. 4.2.6), als Helligkeitswert in der Anordnung der Eingabegeometrie, s. Abb. 4.3.5 oder aber relativ zu einem vorgegebenen (oder erhofften) Konvergenzziel.
- Visualisierung der Fehlerentwicklung und der Konvergenz, z.B. der quadratische Fehler als Funktion der Zeit bzw. des Iterationsindex t.

Da man den Aufwand einer grafischen Programmierung nicht unterschätzen sollte (er ist meist höher als der des eigentlichen neuronalen Algorithmus) hat sich als sinnvoll erwiesen, bestehende grafische Software dafür einzusetzen. Eine preiswerte Lösung für Standard- 2D und 3D Grafiken ist das GNU-Plot-Programm, das kostenlos im Internet über ftp erhältlich ist. Aber auch kommerzielle Programme (Exel, Mathematika, Maple V etc.) sind gut für Standardaufgaben zu verwenden. Spezielle Grafik hingegen (z.B. Abb. 2.4.12 oder 2.4.15) muß auch weiterhin per Hand programmiert werden.

Fehlersuche in Neuronalen Netzen

Ist die Leistung eines programmierten Netzes nicht befriedigend, so gibt es zwei mögliche Ursachen: entweder ist der Netztyp für das Problem nicht geeignet, oder das Programm enthält Fehler. Den ersten Fall, das Designproblem, kann man oft durch eine geeignete Auswahl lösen; hierfür waren die bisherigen Kapitel des Buches gedacht. Für den zweiten Fall gibt es andere Wege.

Um die üblichen Programmierhilfsmittel wie Source Code Debugger, Hilfsausdrucke und Monitoren sinnvoll einsetzen zu können, muß man sich zuerst im Klaren sein, mit welcher Strategie man das Programm testen will. Dazu gibt es einige, aus praktischen Erfahrungen heraus gewonnen Vorschläge:

- Besteht das Netz aus mehreren Schichten oder Teilnetzen, so unteile man das Netze und teste jede Schicht bzw. jedes Modul einzeln unanbhängig voneinander.
- Man reduziert das Programm auf die einfachsten, unbedingt notwendigen Grunoperationen. Graphische Anzeigen und Steuerelemente, Mausoperationen etc. und alle aufwendigen Ein-und Ausgabeoperationen lasse man weg oder ersetze durch plausible Konstanten, so daß nur noch der Netz-Grundalgo rithmus mit Aktivitäten und Lernregeln übrigbleibt.
- Für den Test einer einfachen Schicht erzeuge man sich so einfache Testdaten, daß man sie auch per Hand in die Aktivitäts- und Lerngleichungen einsetzen kann. Ist der Fehler absolut nicht anderweitig zu finden, so ist das Durch rechnen per Hand und Taschenrechner und der Vergleich mit den Computer ausdrucken der Kern der Fehlersuche. Hat man hier Erfolg, so läßt sich Stück für Stück die Grafik usw. dazuschalten, so bootstrap-artig ausgehend von einem intakten Kern das Gesamtprogramm korrigieren.

  Ist auch bei der Handrechnung der Fehler vorhanden, so ist die verwendete Formel falsch; oft sieht man dann mit Hilfe der einfachen Beispiele, was man am Konzept, d.h. am Algorithmus, besser machen könnte.
- Ist die Lerngleichung aus einer Zielfunktion gewonnen worden, so hilft es auch, anstelle der Netzzustandsvektoren den Wert der Zielfunktion zu den Iterations zeitpunkten t auszudrucken, möglichst aufgeschlüsselt nach den einzelnen Anteilen, aus denen sie besteht. Aus der Differenz von erwartetem Verlauf (z.B. Absinken) und tatsächlichem Verlauf eines Anteils kann man einen Hinweis auf den Fehler erhalten.

12 Literaturreferenzen

12.1 Public-Domain Programme, Simulatoren und Reports

Es gibt Software, die kostenlos über internationale Netze (Internet) kopiert und verwendet werden kann. Die folgende (nicht-vollständige) Aufstellung listet jeweils (soweit bekannt) den Namen, die Internet-Adreßnummer sowie das Unterverzeichnis, in dem sich Software, Reports & Veröffentlichungen befinden und den login-Namen (z.B. *anonymous, guest*). Unter UNIX wird meist zum Suchen von Dateien das Programm "Archie" benutzt (Suchbegriff "neural"); zum Kopieren das Filetransferprogramm "ftp". Files mit der Endung ".Z" sind komprimierte Dateien (Unix-Programm compress/uncompress) bzw. ".zip" für die MS-DOS Programme ZIP/UNZIP; die Endung ".ps" bezeichnet druckbare Postscript-Dateien. Es gibt auch Dateien von oft gestellten Fragen ("neural-net-faq"), z.B. bei ftp.bath.ac.uk in /info/faqs. Alle Angaben sind (wie die Software) ohne Gewähr.

ftp.funet.fi		/pub/sci/neural/02INFODIR	*GeneralArchiv*
archive.cis.ohio-state.edu	128.146.8.62	/pub/neuroprose	*Reports*
cogsci.indiana.edu	129.79.238.6	/pub	*Reports*
inf.informatik.uni-stuttgart.de	129.69.211.2	/pub/SNNS*Simulator*	
menaik.cs.ualberta.ca	129.128.4.241	/pub	
cs.ucsd.edu		/pub/GAucsd	
pt.cs.cmu.edu		/afs/cs/project/connect/bench	
b.gp.cs.cmu.edu	128.2.242.8	/usr/connect/connectionists/archives /usr/connect/connectionists/bibliographies	
cs.rochester.edu	192.5.53.209	/pub/simulator	
crl.ucsd.edu	132.239.63.1	/pub/neuralnets	
	128.146.8.60	/pub/condela/condela.tar	
hplpm.hpl.hp.com	15.255.176.205	/pub, /pub/Neuron-Digest, /pub/Neuron-Software	
amazonas.cs.columbia.edu	128.59.16.72	/pub/learning.ps	
princeton.edu		/pub/harnad	
iubio.bio.indiana.edu	129.79.1.101	/biology, /molbio, /chemistry, /science	
ics.uci.edu		/pub/machine-learning-databases	
tutserver.tut.ac.jp	133.15.240.3	/pub/misc	
cochlea.hut.fi	130.233.168.48	/pub/lvq_pak, /pub/som_pak, /pub/ref	
(Progr. & ca. 1000 Referenzen über adaptive Vektorquant. und Selbstorg. Abbildungen)			
boulder.colorado.edu	128.138.240.1	/pub/generic-sources	
b.gp.cs.cmu.edu	128.2.242.8	/connectionists/archives /connectionists/bibliographies	
pt.cs.cmu.edu	128.2.254.155	/afs/cs/project/connect/code	
dartvax.dartmouth.edu	129.170.16.4	/pub/mac	
polaris.cognet.ucla.edu	128.97.50.3	/alexis	
cattel.psych.upenn.edu		/pub/Neuron-Digest	

hope.caltech.edu /pub/mackay

Aktuelle Informationen finden sich auch in der *News*-Gruppe *comp.ai.neural-nets*; wissenschaftliche Informationen etc. sind in der Liste *Connectionists* zu bekommen, der man durch email an *Connectionists-Request@cs.cmu.edu* beitreten kann.

ExtraTip: Im Internet mit WWW bei http://www.neuronet.ph.kcl.ac.uk unter software nachsehen!

12.2 Lehrbücher

D. Amit: *Modeling Brain Functions;* Cambridge Univ. Press, Cambridge (UK) 1989

Wie der Untertitel "the world of attractor neural networks" schon andeutet, beschäftigt sich dieses Buch ausschließlich mit rückgekoppelten Netzwerken.

J.A. Anderson, E. Rosenfeld (Eds): *Neurocomputing I,II: Foundations of Research* MIT Press, Cambridge USA, London, 1988 *und* 1992

Kein Lehrbuch im engen Sinn, sondern eine interessante, zweibändige Sammlung wichtiger Veröffentlichungen zum Thema "Neuronale Netze" von 1890-1992

R. Beale, T. Jackson: *Neural Computing;* Adam Hilger, Bristol, England 1990

Ein sehr einfache, klar geschriebene Einführung.

K. Beerns, T. Kolb: Neuronale Netze für technische Anwendungen; Springer Verlag 1994

A. Carling: *Introducing Neural Networks;* Sigma Press, Wilmslow, England 1992

Eine sehr intuitive, mathematik-freie Einführung.

M. Caudill, C. Butler: *Understanding Neural Networks;* MIT press, London 1992

Eine zweibändige, didaktisch gut gemachte, klare Einführung mit Simulatordisketten.

P. Dayan, L.F. Abbott: Theoretical Neuroscience; MIT Press, London 2001

Eine stark mathematisierte Annäherung an elementare neurobiologische Beobachtungen

J. Freeman, D. Skapura: Neural Networks: Algorithms, Applications, and Programming Techniques; Addison-Wesley Publ., Reading.

Die wichtigsten Modelle, ihre Differenzialgleichungen und ihr Pseudocode

A. Grauel: *Neuronale Netze;* BI-Verlag, Mannheim 1992

P. Hamilton: *Künstliche neuronale Netze;* vde-verlag Berlin 1993

S. Haykin: Neural networks, a comprehensive foundation. Prentice Hall, New York, 3.ed. 2008

Das zur Zeit inhaltlich und didaktisch beste Buch zum Thema Neuronale Netze auf dem Markt. Man merkt, dass Simon Haykin jahrelange Erfahrung in Signaltheorie und in der Lehre hat.

J.A. Hertz, A. Krogh, Palmer: Introduction to the Theory of Neural Computation; Addison-Wesley 1991

Ein mathematisch klares und profundes Lehrbuch, aber ohne graph. Visualisierung, Beispiele oder Aufgaben. Schwerpunkt sind rückgekoppelte Netze

N. Hoffmann: Simulation Neuronaler Netze; Vieweg Verlag 1991

Eine auf den beiliegenden, in Turbo-Pascal geschriebenen Simulator ausgerichtete Darstellung, die nur mit einem zusätzlichen, einführenden Buch benutzt werden sollte.

T. Khanna: *Foundations of Neural Networks;* Addison-Wesley 1990

Kinnebrock: *Neuronale Netze;* Oldenbourg Verlag München 1992

Eine kleine, klare, übersichtliche Einführung.

M. Köhle: *Neuronale Netze;* Springer Verlag Wien 1990

B. Müller, J. Reinhardt: *Neural Networks;* Springer Verlag Berlin 1990

Ein Lehrbuch, das hauptsächlich rückgekoppelte Netze (Stabilität und Kapazität des Hopfield-Modells) mit physikalischen Methoden behandelt. Eine beiliegende Diskette ermöglicht die Simulation der wichtigsten Algorithmen, programmiert in C.

D. Patterson: *Künstliche neuronale Netze*. Prentice Hall 1997

Dan Patterson ist umÜbersicht und Plausibilität bemüht. Dazu wird die Mathematik auf ein Minimum beschränkt, was nicht immer hilfreich für das Verständnis ist.

R. Rojas: *Theorie der neuronalen Netze;* Springer Verlag Berlin 1993

Eines der ausführlichsten deutschen Lehrbücher (außer dem vorliegenden natürlich), das besonders durch seine graphischen Veranschaulichungen besticht.

D.E. Rumelhart, J.L. McClelland: Parallel Distributed Processing; Vol I,II, III MIT press, Cambridge, Massachusetts 1986

Diese klassische Lehrbuchreihe enthält einige Modelle (Comp.Learn.,Back-prop. etc) und Veranschaulichungen sowie die Harmony-Theorie von Sejnowsky. Zum dritten Band wird ein kleiner Simulator mitgeliefert.

E. Schöneburg: *Neuronale Netze*; Markt & Technik Verlag, München 1990

Eine kleine, leichtverständliche Einführung mit einer MS-DOS Diskette, die einen Simulator enthält.

Patrick K. Simpson: *Artificial Neural Systems;* Pergamon Press, Oxford 1989

Y. Takefuji: *Neural Network Parallel Computing;* Kluwer Academic Publ. 1992

Hier wird gezeigt, wie die wichtigsten Standardprobleme, die als Benchmarks für Parallelrechner gelten (N-Queen Problem, crossbar switch scheduling problem, Four-coloring map, Graphenabbildung, Kanal routing, Sortieren und Suchen etc.) mit Neuronalen Netzen gelöst werden können.

A. Zell: Simulation Neuronaler Netze; Addison-Wesley 1994

Ein Buch, das für die praktische Simulation sehr viel Unterstützung bietet. In 26 Kapiteln, werden die wichtigsten Algorithmen kurzgefaßt präsentiert. Der kleinere (ca. 250 Seiten!), aber durchaus beachtenswerte Teil zeigt Simulationstechniken, Hardware und Anwendungen.

J. Zurada: Introduction to Artificial Neural Systems; West Publ. Comp., 1992

und viele weitere Bücher (s. Referenzen), die aber als Aufsatzsammlungen für Laien nur einen facettenhaften Eindruck des Gebiets vermitteln.

12.3 Zeitschriften

Neural Networks, Pergamon Press	zweimonatlich
Offizielles Organ der INNS, ENNS und JNNS	
Biological Cybernetics, Springer Verlag	monatlich
Biologisch-mathematische Beiträge	
Connection Science, Carfax Publ. Comp., Mass.,USA	vierteljährlich
IEEE-Transactions on Neural Networks	zweimonatlich
International Journal of Neural Systems,	vierteljährlich
World Scientific Publishing Co., Singapur	
Network: Computation in Neural Systems	vierteljährlich
Blackwell Scientific Publications, Bristol, UK	
Neural Computation, MIT Press, Boston	halbjährlich
Grundlagenbeiträge	
Neural Computing & Applications, Springer Verlag	vierteljährlich
Anwendungen	
NeuroComputing, Elsevier Science Publ.	zweimonatlich

Außerdem enthalten viele herkömmliche Zeitschriften (pattern recognition, machine intelligence) regelmäßig Beiträge über neuronale Netze.

12.4 Konferenzen

Fast alle Konferenzen über Mustererkennung, computer vision, Kybernetik, Künstliche Intelligenz, VLSI, Robotik, Parallelverarbeitung, u.ä. enthalten Beiträge über Anwendungen neuronaler Netze. Besonders sind allerdings die Fachkonferenzen zu nennen:

Auf internationaler (*amerikanischer*) Ebene halbjährlich

Winter: INNS- WCONN (World Congress On NN , Pergamon Press)

Sommer: IEEE - IJCNN (Int. Joint Conf. on NN, IEEE press)

Auf internationaler (*europäischer*) Ebene ist eine einheitliche Konferenz der European Neural Network Society (ENNS) eingerichtet. Dies ist

ICANN Int. Conf. on Artificial Neural Networks

Den Schwerpunkt auf (*industrielle*) Anwendungen legt die europäische Konferenz

Neuro-Nimes

die seit 1988 jeweils im November in Nîmes (Südfrankreich) tagt.

Eine kleine, aber wichtige Konferenz ist auch

Advances in *Neural Information Processing Systems* (NIPS),
(Morgan Kaufmann Publ., USA)

12.5 Vereinigungen

Die drei wichtigsten Zusammenschlüsse sind die *International Neural Network Society INNS* für (Nord-) Amerika, die *European Neural Network Society ENNS* auf europäischer Ebene und die *Japanese Neural Network Society JNNS* für den ostasiatischen Raum. Allen dreien gemeinsam ist die wissenschaftliche (Mitglieder) Zeitschrift *Neural Networks*.

In Deutschland gibt es die *German Neural Network Society GNNS* als Untergruppe der ENNS sowie die *GI-Fachgruppen* FG 0.0.2 "Neuronale Netze" (Rundbrief N^3) und FG1.1.2 "Konnektionismus", wobei allerdings der gesamte Fachbereich 1 der GI inhaltlich relevante Themen behandelt.

12.6 Literatur

[AARTS89] E. Aarts, J.Korst: *Simulated Annealing and Boltzman Machines;* J. Wiley & Sons, Chichester, UK 1989

[ABL87] Paul Ablay: *Optimieren mit Evolutionsstrategien;* Spektrum der Wissenschaft, Juli 1987

[ABU85] Y.S. Abu-Mostafa, J.-M. St.Jaques: *Information Capacity of the Hopfield Model;* IEEE Trans. on Inf. Theory, Vol IT-31,No4, pp.461-464 (1985)

[ACK85] D.Ackley, G. Hinton, T. Sejnowski: *A learning algorithm for the Boltzmannmachines;* Cognitive Science, Vol 9, pp. 147-169, (1985); auch in [ANDR88]

[ACK90] Reinhard Acker, Andreas Kurz: *On the Biologically Motivated Derivation of Kohonen's Self-Organizing Feature Maps;* in [ECK90], pp. 229-232

[AHA90] Stanley Ahalt, Prakoon Chen, Cheng-Tao Chou: *The Neural Shell: A Neural Network Simulation Tool;* IEEE Proc. Tools for Art. Intell. TAI-90, pp.118-122

[AGM79] N. Agmon, Y.Alhassid, R.D.Levine: *An Algorithm for finding the Distribution of Maximal Entropy;* Journal of Computational Physics, Vol.30, 1979, pp.250-258

[AKCM90] Stanley Ahalt, Astor Krishnamurthy, Prakoon Chen, Douglas Melton: *Competative Learning Algorithms for Vector Quantization;* Neural Networks, Vol.3, pp.277-290, 1990

[ABR64] M. Aizerman, E. Braverman, and L. Rozonoer. *Theoretical foundations of the potential function method in pattern recognition learning*. Automation and Remote Control, 25:821--837, 1964.

[ALB72] A. Albert: *Regression and the Moore-Penrose pseudoinverse;* Academic Press, New York 1972

[ALB75] J.S. Albus: *A New Approach To Manipulator Control CMAC;* Transactions of the American Society of Mechanical Eng. (ASmE), Series G: Journal of Dynamic Systems, Measurement and Control, Vol 97/3 (1975)

[ALE90] I. Aleksander, H.B. Morton: *An Overview of Weightless Neural Systems;* Porc. IJNNC, Washington (1990)

[ALE91] I. Aleksander: *Connectionism or Weightless Neurocomputing?;* Proc. ICANN-91, in: T.Kohonen, K.Mäkisara, O. Simula, J. Kangas (Eds.): Artificial Neural Networks, Elsevier Sc. Publ. (1991)

[ALEX03] A.Alexandridis, H.Sarimveis, G.Bafas: *A new algorithm for online structure and parameter adaption of RBF networks*, Neural Networks 16, pp.1003-1017 (2003)

[ALM89] G. Almasi, A. Gottlieb: *Highly Parallel Computing;* Benjamin/Cummings Publ. Corp., Redwood City CA 1989

[AMA71] S. Amari: *Characteristics of Randomly Connected Threshold-Element Networks and Network Systems;* Proc. IEEE, Vol 59, No. 1, pp. 35-47 (1971)

[AMA72] S. Amari: *Learning Patterns and Pattern Sequences by Self-Organizing Nets of Threshold Elements;* IEEE Trans. on Comp., Vol. C-21 No.11, pp. 1197-1206 (1972)

[AMA77] S. Amari: *Neural Theory of Association and Concept-Formation;* Biol. Cyb. Vol 26, pp.175-185 (1977)

[AMA77b] S. Amari: *Dynamics of Pattern Formation in Lateral-Inhibition Type Neural Fields;* Biol. Cyb. Vol 27, pp.77-87 (1977)

[AMA78] S. Amari, A. Takeuchi: *Mathematical Theory on formation of category detecting nerve cells;* Biol. Cyb., Vol 29, pp. 127-136 (1978)

[AMA79] s. [TAK79]

[AMA80] S. Amari: *Topographic organization of nerve fields;* Bulletin of Mathematical Biology, Vol 42, pp. 339-364 (1980)

[AMA83] S. Amari: *Field Theory of Self-Organizing Neural Nets;* IEEE Trans. System, Man and Cybernetics, Vol SMC-13 No.5, pp.741-748 (1983)

[AMA85] S. Amari: *Differential-Geometrical Methods in Statistics,* Lecture Notes in Statistics, Vol.28, Springer 1985

[AMA88] S. Amari, K. Maginu: *Statistical Neurodynamics of Associative Memory*; Neural Networks, Vol 1, pp.63-73 (1988)

[AMA89] S. Amari : *Characteristics of Sparsely Encoded Associative Memory;* Neural Networks, Vol.2, pp. 451-457 (1989)

[AMA96] S. Amari, A.Cichocki, H.Yang: *A New Learning Algorithm for Blind Signal Seperation.* in: D.Touretzky, M.Mozer, M.Hasselmo (eds.), Advances in Neural Information Processing Systems 8, MIT Press, Cambridge MA, pp.757-763, (1996)

[AMIT85] D. Amit, H. Gutfreund: *Storing infinite Numbers of Patterns in a Spin-Glass Model of Neural Networks;* Phys. Rev. Lett., Vol 55, Nr 14, pp. 1530-1533 (1985)

[AMIT89] D. Amit: *Modeling Brain Functions;* Cambridge University Press, Cambridge (England) 1989

[AND72] J.Anderson: *A simple neural network generating an interactive memory;* Math. Biosc. , Vol 14, pp 197-220 (1972); auch in [ANDR88]

[ANDR88] J.A.Anderson, E.Rosenfeld (Eds): *Neurocomputing: Foundations of Research;* MIT Press, Cambridge USA, London, 1988

[AND88] Diana Z. Anderson (Ed): *Neural Information Processing Systems - Natural and Synthetic;* American Institute of Physics, New York 1988

[ANG88] B. Angéniol, G. de la Croix Vaubois, J.-Y. le Texier: *Self-Organizing Feature Maps and the Travelling Salesman Problem;* Neural Networks Vol 1, pp.289-293, Pergamon Press, New York 1988

[APO87] Special Issue on Neural Networks, Applied Optics, Vol 26, No. 23 (1987)

[ARN64] B. Arnold: *Elementare Topologie;* Van den Hoek & Ruprecht , Göttingen 1964

[ATBO91] Christopher Atkeson, Sherif Botros: *Generalization Properties of Radial Basis-Functions;* in: R. Lippmann, J. Moody, D. Touretzky(Eds): Advances in Neural Information Processing Systems 3, Morgan Kaufmann Publ., San Mateo 1991, pp. 707-713

[BAK90] Gregory Baker, Jerry Gollub: *Chaotic dynamics: an introduction;* Cambridge University Press, 1990

[BALD89] P. Baldi, K. Hornik: *Neural Networks and Principal Component Analysis;* Neural Networks, Vol 2, pp. 53-58, Pergamon Press 1989

[BALL82] D. H. Ballard, Ch. Brown: *Computer Vision;* Prentice Hall 1982

[BAR90] Etienne Barnard, David Casasent: *Shift Invariance and the Neocognitron;* Neural Networks, Vol.3, pp.403-410 (1990)

[BARN86] M.F. Barnsley, V. Ervin, D.Hardin, J. Lancaster: *Solution of an inverse problem for fractals and other sets*; Proc. Natl., Acad. Sci. USA, Vol 83, pp.1975-1977 (1986)[BARN88a] M. Barnsley, A. Sloan: *A Better Way to Compress Images;* Byte, pp. 215-223 (January 1988)

[BARN88b] Michael F. Barnsley: *Fractals everywhere;* Academic Press 1988

[BAW88] E.Baum, F.Wilczek: *Supervised Learning of Probability Distributions by Neural Networks;* in: Dana Anderson (ed.): Neural Information Processing Systems, American Inst. of Physics, New York 1988

[BELF88] L. Belfore, B. Johnson, J. Aylor: *The Design of Inherently Fault-Tolerant Systems;* in: S.Tewksbury, B.Dickinson, S.Schwartz (Eds); Concurrent Computations, Plenum Press, New York 1988

[BELL95] A.Bell, T. Sejnowski: An information-maximisation approach to blind seperation and blind deconvolution. Neural Computation, 7,6 1004-1034 (1995)

[BERT88] J-M. Bertille, J-C. Perez: *Le modele neuronal holographique chaos fractal;* Bases Théorique et applications industrielles, Proc. Neuro-Nimes, Nimes 1988

[BERT90] J-M. Bertille, J-C. Perez: *Dynamical Change of Effective Degrees of Freedom in Fractal Chaos Model;* Proc. INNC-90, pp. 948-951, Kluwer Academic Publ. 1990

[BET76] Albert D. Bethke: *Comparison of genetic algorithms and gradient-based optimizers on parallel processors: Efficient of use of processing capacity;* Logic of Computers Group, Technical Report No. 197, The University of Michigan, Comp. and Com. Sc. Dep. 1976

[BFO84] L. Braimann, J.H.Friedman, R.A. Olshen, C.J. Stone: *Classification and Regression Trees;* Wadsworth International Group, Belmont, CA, 1984

[BICH89] M. Bichsel, P. Seitz: *Minimum Class Entropy: A Maximum Information Approach to Layered Networks;* Neural Networks, Vol 2, pp.133-141 (1989)

[BLOM] R. Blomer, C. Raschewa, Rudolf Thurmayr, RoswithaThurmayr: *A locally sensitive mapping of multivariate data onto a two-dimensional plane,* Medical Data Processing, Taylor & Francis Ltd., London

[BLUM54] J. Blum: *Multidimensional stochastic approximation methods;* Ann. Math. Stat., Vol 25, pp.737-744 (1954)

[BLUM72] E. K. Blum: *Numerical Analysis and computation: Theory and practice;* Addison-Wesley, Reading, MA, (1972)

[BLI91] E. K. Blum, L. K. Li: *Approximation Theory and Feedforward Networks;* Neural Networks, 4, pp. 511-515 (1991)

[BOTT80] S. Bottini: *An Algebraic Model of an Associative Noise-like Coding Memory;* Biol. Cybernetics, Vol 36, pp. 221-228 (1980)

[BOTT88] S. Bottini: An After-Shannon Measure of the Storage Capacity of an Associative Noise-Like Coding Memory; Biol. Cybernetics, Vol 59, pp. 151-159 (1988)

[BRAI67] V. Braitenberg: *Is the cerebellar cortex a biological clock in the millisecond range?* in: C.A.Fox, R.S. Snider (eds.), The cerebellum, Progess in brain research, Vol 2, Elsevier, Amsterdam 1967, pp.334-346

[BRAI89] V. Braitenberg, A. Schütz: Cortex: *hohe Ordnung oder größtmögliches Durcheinander?* Spektrum d. Wissensch., pp. 74-86 (Mai 1989)

[BRA79] R. Brause, M. Dal Cin: *Catastrophic Effects in Pattern Recognition;* in: Structural Stability in Physics, Ed. W.Güttinger, H.Eickemeier, Springer Verlag Berlin, Heidelberg 1979

[BRA88a] R. Brause: *Fehlertoleranz in intelligenten Benutzerschnittstellen;* Informationstechnik it 3/88, pp.219-224, Oldenbourg Verlag 1988

[BRA88b] R. Brause: *Fault Tolerance in Non-Linear Networks;* Informatik Fachberichte 188, pp.412-433, Springer Verlag 1988

[BRA89a] R. Brause: *Neural Network Simulation using INES;* IEEE Proc. Int. Workshop on tools for AI, Fairfax, USA 1989.

[BRA89b] R. Brause: *Performance and Storage Requirements of Topology-conserving Maps for Robot Manipulator Control*; Interner Bericht 5/89 des Fachbereichs Informatik der J.W. Goethe Universität Frankfurt a. M., 1989 und in: Proc. INNC-90, pp. 221-224 , Kluwer Academic Publ. 1990

[BRA91] R. Brause: *Approximator Networks and the Principle of Optimal Information Distribution;* Interner Bericht 1/91 des Fachbereichs Informatik der J.W. Goethe

Universität Frankfurt a. M., 1991 und in: Proc. ICANN-91, Elsevier Science Publ., North Holland 1991

[BRA92] R. Brause: *The Minimum Entropy Network*; Proc. IEEE Tools for Art. Intell. TAI-92

[BRA92b] R. Brause: *Optimal Information Distribution and Performance in Neighbourhood-conserving Maps for Robot Control;* Computers and Artificial Intelligence, Vol. 11, 1992, No.2, 173-199

[BRA93a] R. Brause: *A Symmetrical Lateral Inhibited Network for PCA and Feature Decorrelation;* Proc. Int. Conf. Art. Neural Networls ICANN-93, Springer Verlag, pp. 486-489

[BRA93b] R. Brause: *Transform Coding by Lateral Inhibited Neural Nets;* Proc. IEEE Tools for Art. Intell. TAI-93

[BRA93c] R. Brause: *The Error-Bounded Descriptional Complexity of Approximation Networks*; Neural Networks, Vol.6, pp.177-187, 1993

[BRR98] R. Brause, M. Rippel: Noise Suppressing Sensor Encoding and Neural Signal Orthonormalization, IEEE Transact. on Neural Networks Vol.9, No.4, 1998, pp.613-628

[BRI90a] John Bridle: *Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationship to Statistical Pattern Recognition;* in: NATO ASI Ser. F68, F.Fogelman Soulié, J. Hérault (Eds): Neurocomputing, pp.227-236, Springer Verlag Berlin 1990

[BRI90b] John Bridle: *Training Stochastic Model Recognition Algorithms as networks can lead to Maximum Mutual Information Estimation of Parameters;* in: David S. Touretzky (Ed.): Neural Information Processing Systems 2, pp.211-217, Morgan Kaufmann Publ., San Mateo 1990

[BROD09] K. Brodman: *Vergleichende Lokalisationslehre der Großhirnrinde in ihren Prinzipien dargestellt auf Grund des Zellenbaues*, J.A. Bart, Leipzig 1909

[BROL88] D. Broomhead, D. Loewe : Multivariate functional interpolation and adaptive networks; Complex Systems, Vol.2, pp.321-355 (1988)

[BUCK92] James Buckley: *Theory of the Fuzzy Controller: A Brief Survey;* in: Cybernetics and Applied Systems, C. Negoita (Ed.), Marcel Dekker Inc., New York 1992

[BUH87] J.Buhmann, K.Schulten: *Noise-driven Temporal Association in Neural Networks;* Europhysics Letters, Vol 4 (10), pp. 1205-1209 (1987)

[BUR92] G. Burel: *Blind Separation of Sources: A Nonlinear Neural Algorithm;* Neural Networks, Vol. 5, pp.937-947 (1992)

[CAJ55] Ramon y Cajal: *Histologie du Systéme Nerveaux II;* C.S.I.C. Madrid 1955

[CAL94] R. Caldwell (Ed.): *NEUROVE$T JOURNAL;* (2-Monat. Journal für Neuronale Netze etc. zur Anwendung in Finazmärkten) bestellbar bei R. Caldwell, rbcwell@delphi.com

[CARD89] H.C.Card, W.R. Moore: *VLSI Devices and Circuits for Neural Networks;* Int. Journ. of Neural Syst. Vol 1, No. 2 (1989), pp 149-165

[CAR87a] Gail A Carpenter, S. Grossberg: *A Massively Parallel Architecture for Self-Organizing Neural Pattern Recognition Machine;* Computer Vision, Graphics, and Image Processing Vol 37, pp.54-115, Academic Press 1987; auch in [GRO88]

[CAR87b] Gail A Carpenter, S. Grossberg: *ART2: Self-organization of stable category recognition codes for analog input patterns;* Applied Optics Vol 26, pp. 4919-4930

[CAR90] Gail A. Carpenter, S. Grossberg: *ART3: Hierarchical Search Using Chemical Transmitters in Self-organizing Pattern Recognition Architectures;* Neural Networks Vol 3, pp. 129-152, Pergamon Press 1990

[CAR91a] G. Carpenter, S. Grossberg, John Reynolds: *ARTMAP: Supervised Real-Time Learning and Classification of Nonstationary Data by a Self-Organizing Neural Network;* Neural Networks, Vol.4, pp. 565-588, (1991)

[CAR91b] G. Carpenter, S. Grossberg, David Rosen: *Fuzzy ART: Fast Stable Learning and Categorization of Analog Patterns by an Adaptive Resonance System;* Neural Networks, Vol.4, pp. 759-771, (1991)

[CAR92] Pierre Cardaliaguet, Guillaume Euvrard: *Approximation of a Function and its Derivative with a Neural Network;* Neural Networks, Vol.5, pp.207-220 (1992)

[CAT92] D. Casent, B. Telfert: *High Capacity Pattern Recognition Associative Processors;* Neural Networks, Vol.5, pp.687-698 (1992)

[CECI88] L. Ceci, P. Lynn, P. Gardner: *Efficient Distribution of Back-Propagation Models on Parallel Architecture;* Report CU-CS-409-88, University of Colorado, Sept. 1988 und Proc. Int. Conf. Neural Networks, Boston, Pergamon Press 1988

[CHAP66] R. Chapman: *The repetitive responses of isolated axons from the crab carcinos maenas;* Journal of Exp. Biol., Vol. 45 (1966)

[CHER91] Vladimir Cherkassky, Hossein Lari-Najafi: *Constrained Topological Mapping for Nonparametric Regression Analysis;* Neural Networks, Vol.4, pp.27-40, (1991)

[CHOU88] P.A. Chou: *The capacity of the Kanerva associative memory is exponential;* in: [AND88]

[COOL89] A.C.C. Coolen, F.W.Kujik: *A Learning Mechanism For Invariant Pattern Recognition in Neural Networks;* Neural Networks Vol. 2, pp.495-506 (1989)

[COOP73] L.N. Cooper: *A possible organization of animal memory and learning* Proc. Nobel Symp. on Collective Prop. of Physical Systems, B.Lundquist, S.Lundquist (eds.), Academic Press New York 1973; auch in [ANDR88]

[COOP85] Lynn A. Cooper, Roger N. Shepard: *Rotationen in der räumlichen Vorstellung;* Spektrum d. Wiss., pp. 102-109, Febr. 1985

[COTT88] R.M.Cotterill (ed.): *Computer simulation in brain science;* Cambridge University Press, Cambridge UK (1988)

[COT91] T.Cover, J.Thomas: *Elements of Information Theory*, John Wiley, New York 1991

[CRUT87] J. Crutchfield, J.Farmer, N. Packard, R. Shaw: Chaos; Spektrum d. Wissenschaft, Februar 1987

[DAB01] P. Dayan, L.F. Abbott: *Theoretical Neuroscience*; MIT Press, London 2001

[DAU88 J. Daugman: *Complete Descrete 2-D Gabor Transforms by Neural Networks for Image Analysis and Compression;* IEEE Transactions on Acoustics, Speech and Signal Processing Vol 36, No 7, pp.1169-1179 (1988)

[DAV60] H.Davis: *Mechanism of exication of auditory nerve impulses;* G.Rasmussen, W.Windle (Hrsg), Neural Mechanism of the Auditory and Vestibular Systems, Thomas, Springfield, Illinois, USA 1960

[DEN55] J. Denavit, R.S. Hartenberg: *A Kinematic Notation for Lower-pair Mechanismen Based on Matrices;* Journ. Applied Mech., Vol 77, pp.215-221 (1955)

[DEN86] J.S. Denker (Ed.): *Neural Networks for Computing;* American Inst. of Physics, Conf. Proc. Vol. 151 (1986)

[DES95] G. Deco, B. Schürmann: *Learning time series evolution by unsupervised extraction of correlations;* Physical review E, Vol.51, No.2, Febr. 1995

[DM86] DeJong, G.F. & Mooney, R.J. (1986). *Explanation Based Learning: An Alternative View*, Machine Learning, Vol. 1. No. 2, pp 145-176.

[DEP89] Etienne Deprit: *Implementing recurrent Back-Propagation on the Connection machine;* Neural Networks, Vol.2, pp. 295-314 (1989)

[DIA83] P. Diaconis, B. Efron: *Statistik per Computer: Der Münchhausen-Trick;* Spektrum der Wissenschaft; S. 56-71, Juli 1983

[DIE87] J. Diederich, C. Lischka: *Spread-3. Ein Werkzeug zur Simulation konnektionisti scher Modelle auf Lisp-Maschinen;* KI-Rundbrief Vol.46, pp.75-82, Oldenbourg Verlag 1987

[DOD90] Nigel Dodd: *Optimisation of Network Structure Using Genetic Techniques;* Proc. Int. Neural Network Conf. INNC 90, Kluwer Acad. Publ., 1990

[DOG97] M. Dorigo and L. M. Gambardella: *Ant colonies for the traveling salesman problem*. BioSystems, 43:73–81, 1997.

[DOG99] Marco Dorigo, Gianni Di Caro, Luca M. Gambardella: *Ant Algorithms for Discrete Optimization*, Artificial Life, 1999

[DOW66] J.E.Dowling, B.B. Boycott: *Organization of the primate retina: Electron microscopy;* Proc. of the Royal Society of London, Vol B166, pp.80-111 (1966)

[DUD73] R.Duda, P.Hart: *Pattern Classification and Scene Analysis;* John Wiley & Sons, New York 1973

[DUW87] R. Durbin, D. Willshaw: *An analogue approach to the travelling salesman problem using an elastic net method;* Nature 326, pp. 689-691 (1987)

[DUM90] R. Durbin, G. Mitchison: *A dimension reduction framework for understanding cortical maps;* Nature 343, pp.644-647 (1990)

[DUW87] R. Durbin, D. Willshaw: *An analogue approach to the travelling salesman problem using an elastic net method*; Nature, Vol.326, No.16, April 1987, pp. 689-691

[ECK90] R.Eckmiller, G.Hartmann, G.Hauske (eds.): *Parallel Processing in Neural Systems and Computers;* North Holland, Amsterdam 1990

[EIM85] P. D. Eimas: *Sprachwahrnehmung beim Säugling;* Spektrum d. Wissenschaft, März 1985

[EF82] B. Efron: *The Jackknife, The Bootstrap and Other Resampling* SIAM Monograph No. 38, Society for Industr. and Applied Mathematics, 1982

[ELL88] D. Ellison: *On the Convergence of the Albus Perceptron;* IMA, Journal of Math. Contrl. and Inf., Vol. 5, pp.315-331 (1988)

[ENG93] H. Englisch, Y. Xiao, K. Yao: *Strongly Diluted Networks With Selfinteraction;* Neural Networks, Vol.6, pp.681-688, 1993

[ERD02] D. Erdogmus, J. Principe, K. Hild II: Do Hebbian Synapses Estimate Entropy? in: Neural Networks for Signal Proc. (NNSP '02), IEEE, pp. 199-208, Sept. 2002.

[ERD03] D. Erdogmus, K. Hild II, J. Principe: Online Entropy Manipulation: Stochastic Information Gradient, IEEE Signal Proc. Letters vol. 10, No.8, pp.242-245 (2003)

[FAH88] Scott Fahlmann: *An Empirical Study of Learned Speed in Back-Propagation Networks;* Technical Report CMU-CS-88-162, Dep. of Comp. Sc, Carnigan Mellon University, Pittsburgh 1988

[FELD80] J.A. Feldman, D.H. Ballard: *Computing with connections;* University of Rochester, Computer Science Department, TR72, 1980

[FIZ92] W.Finnoff, H.G. Zimmermann: *Detecting Structure in Small Datasete by Network Fitting under Complexity Constraints;* Arbeitsreport, Siemens AG, München (1992)

[FÖL89] P. Földiák: *Adaptive Network for Optimal Linear Feature Extraction;* IEEE Proc. Int. Conf. Neural Networks; pp.I/401-405 (1989)

[FMT93] T. Fritsch, M. Mittler, P. Tran-Gia: *Artificial Neural Net Applications in Telecommunication Systems;* Neural Computing & Applications, Vol1, pp.124-146 (1993)

[FRE90] Frean: The Upstart Algorithm: *A method for constructing and training feedforward Neural Networks;* Neural Computation2, pp.198-209, 1990

[FRIT92a] B. Fritzke: *Growing cell structures - a self-organizing network in k dimensions;* in: I. Aleksander, J. Taylor (eds.): Art. Neural Networks II, North-Holland, pp.1051-1056, 1992

[FRIT92b] B. Fritzke: *Wachsende Zellstrukturen - ein selbstorganisierendes neuronales Netzwerkmodell;* Diss. a.d. Techn. Fakultät, Universität Erlangen-Nürnberg 1992

[FU68] K.S. Fu: *Sequential Methods in Pattern Recognition;* Academic Press, New York 1968

[FU87] Fu, Gonzales, Lee: *Robotics: Control, Sensing, Vision and Intelligence;* McGraw-Hill 1987

[FUCHS88] A. Fuchs, H. Haken: *Computer Simulations of Pattern Recognition as a Dynamical Process of a Synergetic System*; in: H.Haken (ed.), Neural and Synergetic Computers, pp.16-28,Springer Verlag Berlin Heidelberg 1988

[FUJI87] Cory Fujiki, John Dickinson: *Using the Genetic Algorithm to Generate Lisp Source Code to Solve the Prisoners Dilemma;* in [GRE87], pp. 236-240

[FUK72] K.Fukunaga: *Introduction to Statistical Pattern Recognition;* Academic Press, New York 1972

[FUK80] K. Fukushima: *Neocognitron: A Self-Organized Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position;* Biolog. Cybernetics, Vol. 36, pp. 193-202 (1980)

[FUK84] K. Fukushima: *A Hierarchical Neural Network Model for Associative Memory;* Biolog. Cybernetics, Vol 50, pp. 105-113 (1984)

[FUK86] K. Fukushima: *Neural Network Model for selective Attention in Visual Pattern ecognition*; Biological Cybernetics 55, pp 5-15, (1986)

[FUK87] K. Fukushima: *A neural network model for selective attention in visual pattern recognition and associative recall;* Appl. Optics, Vol.26 No.23, pp.4985-4992 (1987)

[FUK88a] K. Fukushima: *A Neural Network for Visual Pattern Recognition;* IEEE Computer, pp. 65-75, March 1988

[FUK88b] K. Fukushima: *Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition;* Neural Networks, Vol.1, pp.119-130, (1988)

[FUK89] K. Fukushima: *Analysis of the Process of Visual Pattern Recognition by the Neo cognitron;* Neural Networks, Vol. 2, pp.413-420, (1989), Pergamon Press

[GAL88] A. R. Gallant, H. White: *There exists a neural network that does not make avoidable mistakes;* IEEE Sec. Int. Conf. on Neural Networks, pp. 657-664, 1988

[GAL92] A. R. Gallant, H. White: *On Learning the Derivatives of an Unknown Mapping With Multilayer Feedforward Networks;* Neural Networks, Vol. 5, pp. 129-138 (1992)

[GALL88] S. I. Gallant: *Connectionist Expert Systems;* Comm. ACM, Vol 31/2, pp. 152-169 (Febr. 1988)

[GARD88] E. Gardner: *The space of interactions in neural networks;* Journal of Physics, A21, pp.257-270

[GIL87] C. Lee Giles, Tom Maxwell: *Learning, invariance, and generalization in high-order neural networks;* Applied Optics, Vol 26 No.23, pp. 4972-4978 (1987)

[GIL88] C. Lee Giles, R.D. Griffin, Tom Maxwell: *Encoding geometric invariances in high-order neural networks;* in [AND88], pp. 301-309

[GIR91] B. Giraud, L.C. Liu, C. Bernard, H.Axelrad: *Optimal Approximation of Square Integrable Functions by a Flexible One-Hidden-Layer Neural Network of Exitatory and Inhibitory Neuron Pairs;* Neural Networks, Vol.4,pp.803-815 (1991)

[GLA63] R. Glauber: *Time-Dependent Statistics of the Ising Model;* Journal of Math. Physics, Vol 4, p. 294 (1963)

[GLEP94] M. Glesner, W. Pochmüller: *Neurocomputers;* Chapman&Hall, London 1994

[GLU88] M. Gluck, G. Bower: *Evaluating an adaptive network model of human learning;* Jounal of memory and language 27, (1988)

[GOD87] N. Goddard: *The Rochester Connectionist Simulator, User Manual and Advanced rogramming Manual*; Dep. of Comp. Sci., Univ. of Rochester, USA, April 1987

[GOL87] David E. Goldberg, Philip Segrest: *Finite Markov Chain Analysis of Genetic Algorithm;* in [GRE87]

[GOL89] David Goldberg: *Genetic algorithm in search, optimization and machine learning;* Addison Wesley, 1989

[GOS87] U. Rueckert, I. Kreuzer, K.Goser: *A VLSI concept for adaptive associative matrix based on neural networks;* IEEE- Proc. EuroComp '87, pp. 31-34 (1987)

[GRA88] Hans P. Graf, Lawrence D. Jackel, Wayne E. Hubbard: *VLSI Implementation of a Neural Network Model;* IEEE Computer, March 1988

[GRE87] John Grefenstette (Ed.): *Genetic Algorithms and their applications;* Proc. Second Int. Conf. Genetic Alg., Lawrence Erlbaum Ass., 1987

[GRE90] John J. Grefenstette, Alan C. Schultz: *Improving Tactical Plans with Genetic Algorithms;* IEEE Proc. Tools for AI TAI-90, pp. 328-334, Dulles 1990

[GRO69] S. Grossberg: *Some Networks That Can Learn, Remember, and Reproduce Any Number of Complicated Space-Time Patterns, I;* Journal of Mathematics and Mechanics, Vol 19, No. 1, pp.53-91 (1969)

[GRO72] S. Grossberg: *Neural Expectation: Cerebellar and retinal analogs of cells fired by learnable or unlearned pattern classes.;* Kybernetik, Vol.10, pp. 49-57 (1972)

[GRO76] S. Grossberg: *Adaptive pattern classification and universal recoding I + II;* Biological Cybernetics, Vol.23, Springer Verlag (1976)

[GRO87] S. Grossberg: *Competitive Learning: From Interaction to Adaptive Resonance;* Cognitive Science, Vol 11, pp.23-63 (1987); auch in [GRO88]

[GRO88] S. Grossberg (ed.): *Neural Networks and Natural Intelligence;* MIT Press, Cambridge, Massachusetts 1988

[GRO88b] S. Grossberg: *Nonlinear Neural Networks;* Neural Networks,Vol.1,pp.17-61 (1988)

[GRUB88] H. Grubmüller, H. Heller, K. Schulten: *Eine Cray für "jedermann";* mc 11/88, Franzis Verlag, München 1988

[GRU89] A. Grumbach: *Modeles connexionistes du diagnostic;* Proc. Journées d'électron., Ecole Polytechnique Fédérale, Lausanne 1989

[GUEST87] C. Guest, R. TeKolste: *Designs and devices for optical bidirectional associative memories;* in: [APO87], pp. 5055-5060

[GUL78] S. Gull, G. Daniell: *Image reconstruction from incomplete and noisy data;* Nature, Vol. 272, April 1978, pp.686-690

[HAB71] A. Habibi, Paul A. Wintz: *Image Coding by Linear Transformation and Block Quantization;* IEEE Transactions on Communication Technology, Vol. COM-19, No. 1, pp. 50-62 (1971)

[HAK88] H. Haken: *Information and Self-Organization;* Springer Verlag Berlin Heidelberg 1988

[HAN90] S. Hanson, D.Burr: *What connectionist models learn: Learning and representation in connectionist networks;* Behavioral and Brain Sciences Vol 13, 1990, pp 471-518

[HARP90] S. A. Harp, Tariq Samad, Aloke Guha: *Designing Application-Specific Neural Networks Using the Genetic Algorithm;* in: David S. Touretzky, Advances in Neural Information Processing Systems 2, Morgan Kaufmann Publishers, 1990

[HAS89] M. H. Hassoun: *Dynamic Heteroassociative Neural Memories;* Neural Networks, Vol 2, pp. 275-287 (1989)

[HEBB49] D.O. Hebb: *The Organization of Behavior;* Wiley, New York 1949

[HEISE83] W. Heise: *Informations- und Codierungstheorie;* Springer Verlag 1983

[HEM87] J.L. van Hemmen: *Nonlinear Neural Networks Near Saturation;* Phys. Rev. A36, pp.1959 (1987)

[HEM88a] J.L. van Hemmen, D.Grensing, A.Huber, R.Kühn: *Nonlinear Neural Networks;* Journal of Statist. Physics, Vol.50, pp 231 u. 259 (1988)

[HEM88b] J.L. van Hemmen, G. Keller, R. Kühn: *Forgetful Memories;* Europhys. Lett., Vol. 5, pp. 663 (1988)

[HER88a] A. Herz: *Representation and recognition of spatio-temporal objects within a generalized Hopfield Scheme;* Connectionism in Perspective, Zürich Oct. 1988

[HER88b] A.Herz, B. Sulzer, R.Kühn, J.L.van Hemmen: *The Hebb Rule: Storing Statioc and Dynamic Objects in an Associative Neural Network;* Europhys. Letters Vol. 7, pp. 663-669 (1988)

[HER89] A. Herz, B. Sulzer, R. Kühn, J.L van Hemmen: *Hebbian Learning Reconsidered: Representation of Static and Dynamic Objects in Associative Neural Nets;* Biol. Cybernetics, Vol 60, pp. 457-467 (1989)

[HER92] F. Hergert, H, Zimmermann, U.Kramer, W. Finnoff: *Domain Independent Testing and Performance Comparisons for Neural Networks;* in: I. Aleksander, J. Taylor (Eds.), Art. Neural Networks 2, Elsevier Sc. Publ. 1992

[HES90] M. Hestenes: *Conjugate direction methods in optimization;* Springer Verlag NewYork 1990

[HEY87] A. Hey: *Parallel Decomposition of large Scale Simulations in Science and Engineering;* Report SHEP 86/87-7, University of Southampton 1987

[HILL85] D. Hillis: *The Connection Machine;* MIT Press, Cambridge, Massachusetts, 1985

[HILL87] Daniel Hillis, Joshua Barnes: *Programming a highly parallel computer;* Nature Vol. 326, pp. 27-30 (1987)

[HIN81a] G. Hinton, Anderson (eds): *Parallel Models of Associative Memory;* Lawrence Erlbaum associates, Hillsdale 1981

[HIN81b] G. Hinton: *Implementing Semantic Networks in Parallel Hardware;* in [HIN81a]

[HIN91] G. Hinton (Ed.): *Connectionist Symbol Processing;* MIT press 1991

[HIN92] G. Hinton, C. Williams, M. Revow: *Combining Two Methods of Recognizing Hand-Printed Digits*; in: I. Aleksander, J. Taylor (Eds), Art. Neural Systems 2, Elsevier Sc. 1992, pp.53-60

[HN86] Robert Hecht-Nielsen: *Performance Limits of Optical, Electro-Optical, and Electronic Neurocomputers;* Optical and Hybrid Computing SPI, Vol. 634, pp. 277-306 (1986)

[HN87] R. Hecht-Nielsen: *Counterpropagation networks;* IEEE Proc. Int. Conf. Neural Networks, New York 1987; auch in [APO87]

[HN87b] R. Hecht-Nielsen: *Kolmogorov's mapping neural network existence theorem* Caudill, Butler(Eds) IEEE First Int. Conf. on Neural Networks, 3, 11-14

[HN88] R. Hecht-Nielsen: *Applications of Counterpropagation Networks;* Neural Networks, Vol. 1, pp.131-139 (1988), Pergamon Press

[HN89] R. Hecht-Nielsen: *Theory of the back propagation neural network;* Proc. Int. Joint Conf. on Neural Networks, I pp.593-606, SOS printing, San Diego 1989

[HO65] Y. Ho, R.L. Kashyap: *An Algorithm for linear inequalities and its application;* IEEE Trans. on Electronic Computers, Vol EC-14, pp. 683-688 (1965)

[HOL75] J. H. Holland: *Adaption in Natural and Artificial Systems;* University of Michigan Press, Ann Arbor, MI, 1975

[HOL93] M. Hollik: *Einsatzmöglichkeiten von Neuronalen Netzen im Bankbereich*; Diplomarbeit am Inst. f. Betriebswirtschaftslehre, Universität Frankfurt a. M., 1993

[HOP82] J.J. Hopfield: *Neural Networks and Physical Systems with Emergent Collective Computational Abilities;* Proc. Natl. Acad. Sci, USA, Vol 79, pp.2554-2558 (1982)

[HOP84] J.J. Hopfield: *Neurons with graded response have collective computional properties like those of two-state neurons;* Proc. Natl. Acad. Sci. USA, Vol 81, pp 3088-3092 (1984)

[HOP85] J.J. Hopfield, D.W. Tank: *'Neural' Computation of Dercisions in Optimization Problems;* Biological Cybernetics, Vol.52, pp. 141-152 (1985)

[HOR89] H. Horner: *Neural Networks with Low Levels of activity: Ising vs. McCulloch-Pitts Neurons;* Zeitschr. f. Physik , B 75, S. 133 (1989)

[HOR92] K. Hornik, C.-M. Kuan: *Convergence Analysis of Local Feature Extraction Algorithms*; Neural Networks, Vol.5, pp. 229-240 (1992)

[HOP93] R. Hoptroff: *The Principles and Practice of Time Series Forecasting and Business Modelling Using Neural Nets;* Neural Computing and Application, vol.1 pp.59-66 (1993)

[HSW89] K. Hornik, M. Stinchcombe, H. White: *Multilayer Feedforward Networks are Universal Approximators;* Neural Networks, Vol 2, pp. 359-366, Perg. Press 1989

[HSW90] K. Hornik, M. Stinchcombe, H. White: *Universal Approximation of an Unknown Mapping and Its Derivat*ives *Using Multilayer Feedforward Networks;* Neural Networks, Vol 3, pp. 551-560, Perg. Press 1990

[HORN91] K. Hornik: *Approximatiobn Capabilities of Multilayer Feedforward Networks;* Neural Networks, Vol.4, pp.251-257 (1991)

[HUB91] B. Huberman, D. Rumelhart, A. Weigend: *Generalisation by Weight Elimuination with Application to Forecasting*, in: Lippmann, Moody (eds), Advances in Neural Information Processing III, Morgan Kaufman (1991)

[HKO1] Hyvärinen, A., Karhunen, J., and Oja, E., Independent Component Analysis. J. Wiley 2001

[ING92] D. Ingman, Y. Merlis: *Maximum Entropy Signal Reconstruction with Neural Net works;* IEEE Transactions on Neural Networks Vol. 3, No.2, pp.195-201 (1992)

[IRI88] B. Iri, S. Miyake: *Capabilities of three layer perceptrons;* IEEE Second Int. Conf. on Neural Networks I, pp. 641-648, SOS printing, San Diego 1988

[ITO91a] Y. Ito: *Representation of Functions by Superpositions of a Step or Sigmoid Function and Their Applications to Neural Network Theory;* Neural Networks, Vol. 4, pp.385-394 (1991)

[ITO91b] Y. Ito: *Approximation of Functions on a Compact Set by Finite Sums of a Sigmoid Function Without Scaling;* Neural Networks Vol. 4, pp. 817-826 (1991)

[ITO92] Y. Ito: *Approximation of Continuous Functions on R^d by Linear Combinations of Shifted Rotations of a Sigmoid Function With and Without Scaling;* Neural Networks, Vol. 5, pp. 105-115, (1992)

[JAC88] Robert Jacobs: *Increased Rates of Convergence Through Learning Rate Adaption, Neural Networks;* Vol1, pp. 295-307 (1988)

[JAY82] E. Jaynes: *On the Rationale of Maximum-Entropy Methods;* Proc. IEEE, Vol.70, No.9, Sept. 1982, pp.939-961

[JAY84] N.S. Jayant, P.Noll: *Digital Coding of Waveforms: Principles and Applications to Speech and Video;* Prentice Hall, Englewood Cliffs,NJ 1984

[JUD90] S. Judd: *Neural Network Design and the Complexity of Learning;* MIT press 1990

[JUH91a] C. Jutten, J. Herault: *Blind seperation of sources, Part I: An adaptive algorithm based on a neuromimetic architecture;* Signal Processing, Vol.24 (1), pp.1-10 (1988)

[JUH91b] P. Comon, C. Jutten, J. Herault: *Blind seperation of sources, Part II: Problem statement;* Signal Processing, Vol.24 (1), pp.11-20 (1988)

[JUN98] T.-P. Jung, C. Humphries, T.-W. Lee, S. Makeig, M. McKeown, V. Iragui, T. Sejnowski: Removing Electroencephalographic Artifacts, Comparison between ICA and PCA, Proc. Neural Networks for Signal Processing, (1998)

[KAL75] S.Kallert: *Einzelzellverhalten in verschiedenen Hörbahnteilen;* W.Keidel (Hrsg.), Physiologie des Gehörs, Thieme Verlag, Stuttgart 1975

[KAM90] Behzad und Behrooz Kamgar-Parsi: *On Problem Solving with Hopfield Neural Networks;* Biol. Cybernetics, Vol. 62, pp.415-423 (1990)

[KAN87] I.Kanter, H. Sompolinsky: *Associative recall of memory without errors;* Physical Review A, Vol 35, No 1, p. 380 (1987)

[KAN86] P. Kanerva: *Parallel Structures in human and computer memory;* in: [DEN86]

[KAJ94] J. Karhunen, J. Joutsensalo: *Representation and Seperation of Signals Using Nonliner PCA Type Learning;* Neural Networks, Vol.7, No.1, pp.113-127 (1994)

[KEE87] J.D. Keeler: *Basin of Attraction of Neural Network models;* in:[DEN86]

[KEE88] J.D. Keeler: *Capacity for patterns and sequences in Kanerva's SDM as compared to other associative memory models;* in: [AND88]

[KIEF52] J. Kiefer, J.Wolfowitz: *Stochastic estimation of the maximum of a regression function;* Ann. Math. Stat., Vol 23, pp. 462-466 (1952)

[KIM89] H. Kimelberg, M.Norenberg: *Astrocyten und Hirnfunktion;* Spektrum der Wissenschaft, Juni 1989

[KIN89] J. Kindermann: *Inverting Multilayer Perceptrons;* Proc. DANIP Workshop on Neural Netw., GMD St. Augustin, April 1989

[KIN85] W. Kinzel: *Spin Glasses as Model Systems for Neural Networks;* Int. Symp. Complex Syst., Elmau 1985, Lecture Notes, Springer Series on Synergetics

[KIRK83] S. Kirkpatrick, C.D. Gelatt, Jr, M.P. Vecchi: *Optimization by simulated annealing;* Science, Vol 220, pp.671-680 (1983); auch in [ANDR88]

[KLE86] David Kleinfeld: *Sequential state generation by model neural networks;* Proc. Natl. Acad. Sci. USA, Vol. 83, pp. 9469-9473, (1986)

[KMYS93] H. Kobayashi, T. Matsumoto, T. Yagi, T. Shimmi: *Image Processing Regularization Filters on Layered Architecture;* Neural Networks, Vol.6, pp.327-350 (1993)

[KOH72] T. Kohonen: *Correlation Matrix Memories;* IEEE Transactions on Computers, Vol C21, pp.353-359, (1972); auch in [ANDR88]

[KOH76] T. Kohonen, E. Oja: *Fast Adaptive Formation of Orthogonalizing Filters and Associative Memory in Recurrent Networks of Neuron-Like Elements;* Biological Cybernetics, Vol. 21, pp. 85-95 (1976)

[KOH77] T. Kohonen: *Associative Memory;* Springer Verlag Berlin 1977

[KOH82a] T. Kohonen: *Analysis of a simple self-organizing process;* Biological Cybernetics, Vol. 40, pp. 135-140 (1982)

[KOH82b] T. Kohonen: *Self-Organized formation of topology-correct feature maps;* Biological Cybernetics, Vol. 43, pp. 59-69 (1982)

[KOH84] T. Kohonen: *Self-Organisation and Associative Memory;* Springer Verlag Berlin, 1984

[KOH88] T. Kohonen: *The "Neural" Phonetic Typewriter of Helsinki University of Technology;* IEEE Computer, March 1988

[KOH88b] T. Kohonen: *Learning Vector Quantization;* in: Abstracts of the First annual INNS meeting; Pergamon Press 1988, p. 303

[KOH93] T. Kohonen: *Physiological Interpretation of the Self-Organizing Map Algorithm*; Neural Networks, Vol.6, 895-905, 1993

[KOL56] A.N. Kolmogorov: *On the representation of continuous functions of several variables by superpositions of continuous functions of a smaller number of variables* (English. Russian original) [J] Am. Math. Soc., Transl., II. Ser. 17, 369-373 (1961); translation from Dokl. Akad. Nauk SSSR 108, 179-182 (1956).

[KON90] Y. Kong, A. Noetzel: *A Training Algorithm for a Piecewise Linear Neural Network;* Proc. Int. Neural Netw. Conf. INNC-90, Kluwer Academic Publ., Dordrecht 1990

[KOR89] T. Korb, A. Zell: *A declarative Neural Network Description Language;* Proc. Euromicro, Köln 1989, Microprogr. and Microproc., Vol 27/1-5, North- Holland

[KOS87] Bart Kosko: *Adaptive bidirectional associative memories;* in: [APO87]

[KOS92] B.Kosko: *Neural Networks and Fuzzy Systems;* Prentice-Hall, Englewood Cliffs, N.J. USA 1992

[KOO90] P.Koopman, L.Rutten, M. van Eekelen, M. Plasmeijer: *Functional Descriptions of Neural Networks;* Proc. INNC-90, pp. 701-704, Kluwer Academic Publ. 1990

[KRE91] V. Kreinovich: *Arbitrary Nonlinearity Is Sufficient to Represent All Functions by Neural Networks: A Theorem;* Neural Networks, Vol. 4, pp. 381-383 (1991)

[KRO90] A. Krogh, J. Hertz: *Hebbian Learning of Principal Components;* in: [ECK90], pp.183-186

[KRO92] A. Krogh, J. Hertz: *A Simple Weight Decay Can Improve Generalization;* in: G.Moody, S.Hanson, R.P. Lippmann, Advances in Neural Proc. Syst. 4, San Mateo 1992

[KPD90] S. Knerr, L. Personnaz, G. Dreyfus: *Single-layerlearning revisited: A stepwise procedure for building and training a neural network;* Fogelman Soulié, Hérault (Eds.): Neurocomputing; NATO ASI Series Vol.F68, Springer Verlag 1990

[KÜH89a] R. Kühn, J.L. van Hemmen, U. Riedel: *Complex temporal association in neural networks;* J. Phys. A: Math. Gen. Vol 22 (1989), pp. 3123-3135

[KÜH89b] R. Kühn, J.L. van Hemmen, U. Riedel: *Complex temporal association in neural nets;* Proc. Conference nEuro'88, L. Personnez, G.Dreyfus (Hrsg), I.D.S.E.T., Paris 1989, pp. 289-298

[KÜH90] H. Kühnel, P.Tavan: *The Anti-Hebb Rule derived from Information Theory;* in: [ECK90], pp.187-190

[KÜR88] K.E. Kürten, J.W. Clark: *Exemplification of chaotic activity in non-linear neural networks obeying a deterministic dynamics in continous time;* in [COTT88]

[KUFF53] S.W.Kuffler: *Discharge Patterns and Functional Organization of Mammalian Retina;* Journal of Neurophys., Vol 16, No1, pp.37-68, (1953)

[KUL59] S. Kullback: *Information Theory and Statistics*; Wiley, New York (1959)

[LAN88] D. Lang: *Informationsverarbeitung mit künstlichen neuronalen Netzwerken;* Dissertation am Fachbereich Physik der Universität Tübingen, 1988

[LAP87] A. Lapedes, R. Farber: *Nonlinear Signal Processing using Neural Networks: Prediction and System Modelling;* Los Alamos preprint LA-UR-87-2662 (1987)

[LAP88] A. Lapedes, R. Farber: *How Neural Nets Work;* Report LA-UR-88-418, Los Alamos Nat. Lab. 1988; und in [LEE88b]

[LASH50] K.S. Lashley: *In search of the engram;* Soc. of Exp. Biol. Symp. Nr.4: Psycholog. Mech. in Anim. Behaviour, Cambridge University Press, pp.454, 468-473, 477-480 Cambridge 1950; auch in [ANDR88]

[LAW71] D.N. Lawley, A.E. Maxwell: *Factor Analysis as a Statistical Method;* Butterworths, London 1971

[LEE88] Y.C. Lee: *Efficient Stochastic Gradient Learning Algorithm for Neural Network;* in [LEE88b], pp.27-50

[LEE88b] Y.C. Lee (Ed.): *Evolution, Learning and Cognition;* World Scientific, Singapore, New Jersey, London 1988

[LEEK91] S. Lee and R. Kil: *A Gaussian Potential Function Network With Hierarchicallly Self-Organizing Learning;* Neural Networks, Vol. 4, pp. 207-224, 1991

[LEV85] M. Levine: *Vision in man and machine;* McGraw Hill 1985

[LIN73] S. Lin, B.W. Kernighan: *An effective heuristic algorithm for the traveling salesman problem;* Operation Research, Vol 21, pp. 498-516 (1973)

[LIN86] R. Linsker: *From Basic Network Principles to Neural Architecture;* Proc. Natl. Academy of Science, USA, Vol. 83, pp7508-7512, 8390-8394, 8779-8783

[LIN88a] R. Linsker: *Self-Organization in a Perceptual Network;* IEEE Computer, pp. 105-117, (March 1988)

[LIN88b] R. Linsker: Towards an Organizing Principle for a Layered Perceptual Network; in [AND88]

[LIN88c] R. Linsker: *Developement of feature-analyzing cells and their columnar organization in a layered self-adaptive network;* in [COTT88]

[LIP89] R. Lippmann: *Pattern Classification Using Neural Networks;* IEEE Communications Magazine, Nov. 1989, pp. 47-64

[LIT78] W.A. Little, G.L. Shaw: *Analytic Study of the Memory Storage Capacity of a Neural Network;* Math. Biosc., Vol 39, pp. 281-290 (1978); auch in [SHAW88]

[LJUNG77] L. Ljung: *Analysis of Recursive Stochastic Algorithms;* IEEE Transactions on Automatic Control, Vol AC-22/4, (August 1977)

[LONG68] H.C. Longuet-Higgins: *Holographic model of temporal recall;* Nature 217, p.104 (1968)

[LOOS88] H.G.Loos: *Reflexive Associative Memories;* in [AND88]

[MAAS90] Han van der Maas, Paul F. Verschure, Peter C. Molenaar: *A Note on Chaotic Behavior in Simple Neural Networks;* Neural Networks, Vol.3, pp.119-122 (1990)

[MACFH92] M. Musavi, W. Ahmed, K. Chan, K. Faris, D. Hummels: *On the Training of Radial Basis Function Classifiers*; Neural Networks, Vol.5, pp.595-603, (1992)

[MKAC93] M. Musavi, K. Kalantri, W.Ahmed, K.Chan: *A Minimum Error Neural Network (MNN)*; Neural Networks, Vol.6, pp.397-407, (1993)

[MC43] W. S. McCulloch, W. H. Pitts: *A Logical Calculus of the Ideas Imminent in Neural Nets;* Bulletin of Mathematical Biophysics (1943) Vol 5,pp. 115-133; auch in: [ANDR88]

[MAR90] Marchand, Golea, Rujan: *A Convergence Theorem for Sequential Learning in Two-Layer Perceptrons;* Europhysics Letters 11, pp.487-492, 1990

[MAR91] T. Martinetz, K. Schulten: *A "Neural-Gas" Network Learns Topologies;* in: T.Kohonen, K. Mäkisara, O.Simula, J.Kangas (Eds.): Artif. Neural Networks, North Holland, pp. 397-402, 1991

[MAR93] T. Martinetz: *Competitive Hebbian Learning Rule Forms Perfectly Topology Preserving Maps;* Proc. ICANN-93, Springer Verlag 1993

[MAL73] C. von der Malsburg: *Self-organization of orientation sensitive cells in the striate* cortex; Kybernetik, Vol 14 pp. 85-100 (1973); auch in: [ANDR88]

[MAL88] C. von der Malsburg: *Pattern Recognition by Labeled Graph Matching;* Neural Networks, Vol.1, pp.141-148, (1988)

[MAW91] C.M. Marcus, F.R. Waugh, R.M. Westervelt: *Connection Topology and Dynamics in Lateral Inhibition Networks;* in: R.Lippmann, J. Moody, D. Touretzky (Eds.): Advances in Neural Information Proc. Syst. 3, pp.98-104, Morgan Kaufmann Publ., San Mateo 1991

[MCE87] R.J. McEliece, E.C. Posner, E.R. Rodemich, S.S.Venkatesh: *The Capacity of the Hopfield Associative Memory;* IEEE Trans. on Inf. Theory, Vol IT-33, No.4, pp.461-482 (1987)

[MEHS93] A. Meyer-Bäse, W. Endres, W. Hilberg, H. Scheich: *Modulares Neuronales Phonemerkennungskonzept mit Radialbasisfunktionen;* in: S.Pöppl, H.Handels (Hrsg.): Mustererkennung 1993, Springer Verlag 1993, pp. 670-677

[MET53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller: Equation of *State Calculations by Fast Computing Machines;* The Journal of Chemical Physics, Vol 21, No.6, pp.1087-1092 (1953)

[MIN88] M. Minsky, Papert: *Perceptrons;* MIT Press, 1988

[MK86] Mitchell, T.M., Keller, R., and Kedar-Cabelli, S. (1986). *Explanation Based Generalisation: A Unifying View*, Machine Learning, Vol 1, No. 1, pp 47-80.

[MOD88] J. Moody, C. Darken: *Learning with Localized Receptive Fields;* Research Report YALEU/DCS/RR-649, Yale University, Dep. of Comp. Science, Sept. 1988

[MOD89] J. Moody, C. Darken: *Fast Learning in Networks of Locally-Tuned Processing Units;* Research Report YALEU/DCS/RR-654, Yale University, Dep. of Comp. Science, Revised March. 1989 und in: Neural Computation, Vol.1, pp.281-294, 1989

[MOL93] M. Moller: A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning; Neural Networks, Vol.6,pp.525-533, 1993

[MOO89] J. Moody: *Fast Learning in Multi-Resolution Hierarchies;* Research Report YALEU/DCS/RR-681, Yale University, Dep. of Comp. Science, Feb. 1989

[MÜ90] B. Müller, J. Reinhardt: *Neural Networks;* Springer Verlag Berlin 1990

[MÜL92] K.-R. Müller: *On Systematically Diluted Hopfield Networks;* in.: I. Aleksander, J.Taylor(Eds), Art. Neural Networks, Elsevier Sc. Publ., pp. 95-100, 1992

[MÜWA92] K.-R. Müller, A. Waldenspuhl: *Performance Comparison of Learning Algorithms in Hopfield Networks*; in.: I. Aleksander, J.Taylor(Eds), Art. Neural Networks, Elsevier Sc. Publ., pp. 961-964, 1992

[MUR90] Jacob Murre, Steven Kleynenberg: *The MetaNet Network Environment for the Developement of Modular Neural Networks;* Proc. INNC-90, pp. 717-720, Kluwer Academic Publ. 1990

[MUR93] Jacob Murre: *Transputers and Neural Networks: An Analysis of Implementation Constraints and Performance;* IEEE Transaction on Neural Networks, Vol.4, Nr.2, pp.284-292

[NaTo03] Sunil Nakrani, Craig Tovey: *On Honey Bees and Dynamic Allocation in an Internet Server Colony*, Proc. 2nd Int. Workshop on the Mathematics and Algorithms of Social Insects, 2003, *http://www.insects.gatech.edu*/ [Febr.2005]

[NaPa94] J.-P.Nadal, N.Parga: *Non-linear neurons in the low noise limit: a factorial code maximizes information transfer*. Network 5 (4), pp.565-581, 1994

[NEW88] F. Newbury: *EDGE: An Extendible Directed Graph Editor;* Internal Report 8/88, Universität Karlsruhe, West Germany

[NIL65] N. Nilson: *Learning machines: Foundations of trainable pattern-classifying systems,* McGraw-Hill, New York 1965

[NIPS3] R. Lippmann, J. Moody, D. Touretzky(Eds): Advances in Neural Information Processing Systems 3, Morgan Kaufmann Publ., San Mateo 1991

[OJA82] E. Oja: *A simplified Neuron Model as Principal Component Analyzer;* J. Math. Biology, Vol. 15, pp. 267-273 (1982)

[OJA85] E. Oja, J. Karhunen: *On Stochastic Approximation of the Eigenvektors and Eigenvalues of the Expectation of a random Matrix;* Report TKK-F-A458, Helsinki Univ. of Techn., Dept. Techn. Phys.(1981); und in J. M. A. A., Vol. 106, pp.69-84 (1985)

[OJA89] E. Oja: *Neural Networks, Principal Components, and Subspaces;* Int. Journ. Neural Syst. Vol1/1, pp. 61-68, World Scientific, London 1989

[OOW91] E. Oja, H. Ogada, J. Wangviwattana: *Learning in nonlinear, constained Hebbian networks;* in: Kohonen et al. (eds), Artificial Neural Networks, Proc. ICANN'91, North-Holland, Amsterdam 1991, pp. 385-390

[OOW92a] E. Oja, H. Ogada, J. Wangviwattana: *Principal Component Analysis by Homogenous Neural Networks, Part I: The Weighted Subspace Criterion;* IEICE Trans. Inf. and Systems, Vol. E75-D, No. 3, pp.366-375 (1992)

[OOW92b] E. Oja, H. Ogada, J. Wangviwattana: *Principal Component Analysis by Homogenous Neural Networks, Part II:Analysis and Extensions of the Learning Algorithm;* IEICE Trans. Inf. and Systems, Vol. E75-D, No. 3, pp.376-382 (1992)

[OJA94] E. Oja: *Beyond PCA: Statistical Expansions by Nonlinear Neural Networks;* in: ICANN'94, Springer Verlag 1994, Vol.2, pp.1049-1054

[OLI87] I.M. Oliver, D.J. Smith, J.R.C. Holland: *A Study of Permutation Crossover Operators on the Travelling Salesman Problem;* in [GRE87], pp. 224-230

[ONI92] A.van OOyen, B.Nienhuis: *Improving the Convergence of the Back-Propagation Algorithm;* Neural Networks, Vol5, pp.465-471 (1992)

[OP88] M. Opper: *Learning Times of Neural Networks: Exact Solution for a PERCEPTRON Algorithm;* Phys. Rev. A38, p.3824 (1988)

[PAK91] Y. Pati, P. Krishnaprasad: *Discrete Affine Wavelet Transforms For Analysis And Synthesis Of Feedforward Neural Networks;* in: [NIPS3], pp. 743-749

[PALM80] G. Palm: *On Associative Memory;* Biolog. Cybernetics, Vol 36, pp. 19-31 (1980)

[PALM84] G. Palm: *Local synaptic modification can lead to organized connectivity patterns in associative memory;* in: E.Frehland (Ed.), Synergetics: from microscopic to macroscopic order, Springer Verlag Berlin, Heidelberg, New York 1984

[PALMI94] F. Palmieri: *Hebbian Learning and Self-Association in Nonlinear Neural Networks;* Proc. IEEE World Congress on Comp. Intell., 1994

[PAR86] G. Parisi: *A memory which Forgets;* Journal of Physics, Vol. A19, L 617 (1986)

[PAW91] K. Pawelzik: *Nichtlineare Dynamik und Hirnaktivität;* Dissertation Universität Frankfurt, FB Physik 1991, Verlag Harri Deutsch, Frankfurt 1991

[PEHI86] Barak. A. Pearlmutter, Geoffrey E. Hinton: *G-Maximization: an Unsupervised Learning Procedure for Discovering Regularities;* in: [DEN86], pp. 333-338

[PERL89] M. Perlin, J. -M. Debaud: *MatchBox: Fine grained Parallelism at the Match Level;* IEEE TAI-89, Proc. Int. Workshop on tools for AI, USA 1989.

[PER88] J.-C. Perez: *De nouvelles voies vers l'intelligence artificielle;* Editions Masson, 1988

[PER88b] J.-C. Perez: *La memoire holographique fractale;* IBM France, Montpellier 1988

[PFAF72] E. Pfaffelhuber: *Learning and Information Theory;* Int. J. Neuroscience, Vol 3, pp. 83-88, Gordon and Breach Publ. , 1972

[PFAF73] E. Pfaffelhuber, P. S. Damle: *Learning and Imprinting in Stationary and Nonstationary Environments;* Kybernetik Vol.13, pp.229-237 (1973)

[PIN88] S. Pinker, J. Mehler (Eds): *Connections and Symbols;* MIT press 1988

[PLUM91] M. Plumbley: *On Information Theory and Unsupervised Neural Networks;* Technical Report CUED/F-INFENG/TR.78, Cambridge University Engineering Department, UK (1991).

[PLUM93] M. Plumbley: Efficient Information Transfer and Anti-Hebbian Neural Networks; Neural Networks, Vol.6, pp.823-833 (1993)

[POGT85] T. Poggio, V. Torre, Ch. Koch: *Computational vision and regularization theory;* Nature Vol. 317, pp. 314-319, Sept. (1985)

[POGE90] T. Poggio, S. Edelman: *A network that learns to recognize three-dimensional objects;* Nature Vol. 343, pp. 263-266 (1990)

[POGG90a] T. Poggio, F. Girosi: *Networks for Approxi*mation and Learning; Proc. IEEE Vol. 78, No.9, pp.1481-1496 (1990)

[POGG90b] T. Poggio, F. Girosi: *Regularization Algorithms for Learning That Are Equivalent to Multilayer Networks;* Science, Vol.247, pp.978-982, (1990)

[POGG93] T. Poggio, F. Girosi, M. Jones: *From Regularization to Radial, Tensor and Additive Splines;* in: Kammet.al.: Neural Networks for Signal Processing III, Proc. IEEE-SP Workshop, pp. 3-10, 1993

[PRO93] P. Protzel, D. Palumbo, M. Arras: *Performance and Fault-Tolerancwe of Neural Networks for Optimization;* IEEE Transactions on Neural Networks, Vol.4, No.4, pp. 600-614, 1993

[RE73] Ingo Rechenberg: *Evolutionsstrategie;* problemata frommann-holzboog, Stuttgart 1973

[RE94] Ingo Rechenberg: *Evolutionsstrategie '94;* frommann-holzboog, Stuttgart 1994

[REF91a] A. Refenes: *CLS:An Adaptive Learning Procedure and Its Application to Time Series Forecasting;* Proc. IJCNN-91, Singapore, (1991)

[REF91b] A. Refenes, S. Vithlani: *Constructive Learning by Specialisation;* Proc. ICANN-91, Kohonen et. al. (eds.), Elsevier Sc. Publ. (1991), pp.923-929

[REF93] A. Refenes, M. Azema-Barac, L. Chen, S. Karoussos: *Currency Exchange Rate Prediction and Neural Network Design Strategies;* Neural Computing and Applic., vol1, pp.46-58,(1993)

[REF94] A. Refenes, M. Azema-Barac: *Neural Network Applications in Financial Asset Management;* Neural Computing and Applic., vol2, pp.13-39,(1994)

[REH90a] H. Rehkugler, T.Podding: *Statistische Methoden versus Künstliche Neuronale Netzwerke zur Aktienkursprognose;* Bamberger Betriebswirtschaftliche Beiträge 73/1990, Universität Bamberg 1990

[REH90b] H. Rehkugler, T.Podding: *Entwicklung leistungsfähiger Prognosesysteme auf Basis Künstliche Neuronaler am Beispiel des Dollars;* Bamberger Betriebswirtschaftliche Beiträge 76/1990, Universität Bamberg 1990

[REH92] H. Rehkugler, T.Podding: *Klassifikation von Jahresabschlüssen mittels Multilayer-Perceptrons;* Bamberger Betriebswirtschaftliche Beiträge 87/1992, Universität Bamberg 1992

[RIE88] U. Riedel, R.Kühn, J.L. van Hemmen: *Temporal sequences and chaos in neural nets;* Physical review A, Vol 38/2, pp. 1105-1108 (1988)

[RIE92] H. Riedel, D. Schild: *The Dynamics of Hebbian Synapses can be Stabilzed by a Nonlinear Decay Term;* Neural Networks, Vol.5, pp.459-463, (1992)

[RITT86] H. Ritter, K. Schulten: *On the Stationary State of Kohonen's Self-Organizing Sensory Mapping;* Biolog. Cyb. Vol 54, pp. 99-106, (1986)

[RITT88 H. Ritter, K. Schulten: *Convergence Proporties of Kohonen's Topology conserving Maps;* Biological Cyb., Vol 60, pp. 59 ff, (1988)

[RITT89] H. Ritter, T. Martinetz, K. Schulten: *Topology-Conserving Maps for Learning Visuomotor-Coordination;* Neural Networks, Vol 2/3, pp. 159-167, (1989)

[RITT90] H. Ritter, T.Martinetz, K.Schulten: *Neuronale Netze;* Addison-Wesley, Bonn 1990

[RITT93] H. Ritter: *Parametrized Self-Organizing Maps;* in: S.Gielen, B. Kappen (Eds.), ICANN'93, Springer Verlag London 1993, pp. 568-575

[ROB51] H.Robbins, S.Monro: *A stochastic approximation method;* Ann. Math. Stat., Vol 22, pp. 400-407 (1951)

[ROD65] R.W. Rodieck: *Quantitative Analysis of Cat Retinal Ganglion Cell Response to Visual Stimuli;* Vision Research, Vol.5, No.11/12, pp.583-601 (1965)

[ROS58] F. Rosenblatt: *The perceptron: a probabilisitic model for information storage and organization in the brain;* Psychological Review, Vol 65, pp. 386-408 (1958) auch in [ANDR88]

[ROS62] F.Rosenblatt: *Principles of neurodynamics;* Spartan Books, Washington DC, 1962

[RUB90] J. Rubner, K. Schulten, P. Tavan: *A Self-Organizing Network for Complete Feature Extraction;* in [ECK90], pp. 365-368

[RUM86] D.E. Rumelhart, J.L. McClelland: *Parallel Distributed Processing;* Vol I, II, III MIT press, Cambridge, Massachusetts 1986

[RUM85] D.E. Rumelhart, D. Zipser: *Feature discovery by competitive learning;* Cognitive Science Vol 9, pp. 7-112

[SAN88] Terence D. Sanger: *Optimal Unsupervised Learning in a Single-Layer Linear Feedforward Neural Network;* Proc. of the Int. Conf. Neural Networks, Boston 1988, Pergamon press; und in Neural Networks, Vol. 2, pp.459-473 (1989)

[SAN90] T. Sanger: *Analysis of the Two-Dimensional Receptive Fields Learned by the Generalized Hebbian Algorithm in Response to Random Input;* Biol. Cybernetics, V0.63, pp.221-228 (1990)

[SAN91] T. Sanger: *Optimal hidden units for two-layer nonlinear feedforward neural networks;* Int. J. of Pattern Recognition and Art. Int., Vol.5, pp. 545-561, (1991)

[SPWG94] O. Scherf, K. Pawelzik, F. Wolf, T.Geisel: Unification of Complementary Feature Map Models; Proc. Int. Conf. on Art. Neural Networks ICANN 94, pp.338-341 (1994)

[SCH92] H. Shioler, U. Hartmann: *Mapping Neural Network Derived from Parzen Window Estimator*; Neural Networks, Vol.5, pp.903-909 (1992)

[SCHI92] W. Schiffmann, M. Joost, R. Werner: *Optimierung des Backpropagation Algorithmus zum Training von Multilayer Perceptrons;* Fachbericht 15/1992, Universität Koblenz, 1992

[SCH76] R. F. Schmidt, G. Thews: *Einführung in die Physiologie des Menschen;* Springer Verlag, Berlin 1976

[SSB97] B. Schölkopf, K-K. Sung, C.Burges, F. Girosi, P. Niyogi, T. Poggio, V. Vapnik: Comparing Support Vector Machines with Gaussian Kernels to Radial Basis Function Classifiers, IEEE Trans. Signal Processing, Vol 45, No. 11, (1997), pp 2758-2765

[SCHÖ90] E. Schöneburg: *Stock price prediction using neural networks: A project report;* Neurocomputing, Vol 2, pp. 17-27 ; Elsevier Science Publ. 1990

[SCHU84] H. G. Schuster: *Deterministic Chaos: An Introduction;* Physik-Verlag, Weilheim 1984

[SCTW91] A. Saha, J. Christian, D.Tang, Ch. Wu: *Oriented Non-Radial Basis Functions for Image Coding and Analysis;* in: [NIPS3], pp. 728-734

[SEG82] Segraves, Rosenquist: *The afferent and efferent callosal connections of retinotopic defined areas in cat cortex;* J. Neurosci., Vol 8, pp. 1090-1107 (1982)

[SEJ86] T.J. Sejnowski, C.R. Rosenberg: *NETtalk: A parallel network that learns to read aloud;* John Hopkins University, Electrical Engeneering and Computer Science Technical Report JHU/EE CS- 86/01; auch in [ANDR88]

[SEJ86b] T.J. Sejnowski: *High-Order Boltzmann Machines;* AIP Conf. Proc. Vol 151, 398 (1986)

[SHA49] C.E. Shannon, W.Weaver: *The Mathematical Theory of Information;* University of Illinois Press, Urbana 1949 und C.E. Shannon, W.Weaver: *Mathematische Grundlagen der Informationstheorie;* Oldenbourg Verlag, München 1976

[SHA49b] C. E. Shannon: *Communication in the Presence of Noise*; Proceedings of the IRE, Vol. 37, pp. 10-21, January 1949

[SHP92] J. Shapiro, A. Prügel-Bennet: *Unsupervised Hebbian learning and the shape of the neuron activation function;* in: Aleksander, Taylor (eds), Artificial neural Networks 2, Proc. ICANN'92, North Holland, Amsterdam 1992, pp.179-182

[SHAW88] G.L.Shaw, G. Palm (eds.): *Brain Theory;* World Scientific, Singapur 1988

[SHU88] Nan C. Shu: *Visual Programming;* van nostrand Reinhold Comp., New York 1988

[SIL90] F. Silva, L. Almeida: *Speeding up Backpropagation;* R. Eckmiller (ed.), Advanced Neural Computers, pp.151-158 (1990)

[SIL91] F. Silva, L. Almeida: *A distributed solution for data orthonormalization;* in: T. Kohonen et. al. (Eds.): Artificial Neural Networks, Elsevier Sc. Publ., 1991

[SING85] W. Singer: *Hirnentwicklung und Umwelt;* Spektrum der Wissenschaft, März pp. 48-61 (1985)

[SJW92] W. Schiffmann, M.Joost, R. Werner: *Synthesis and Performance Analysis of Multilayer Neural Network Architectures*; Technical Report 16/1992, Univ. Koblenz-Landau, Institut f. Physik (1992)

[SLB92] M. Schumann, T. Lohrbach, P. Bährs: *Versuche zur Kreditwürdigkeitsprognose mit Künstlichen Neuronalen Netzen;* Arbeitspapier Nr.2, Univ. Göttingen, Abt. Wirtschaftsinformatik, Göttingen 1992

[SMI82] D.Smith, C. Irby, R.Kimball, B. Verplanck: *Designing the STAR User Interface;* Byte, April 1982, pp. 242-282

[SOM86] H.Sompolinsky, I. Kanter: *Temporal Association in Asymmetric Neural Networks;* Physical review Letters, Vol.57, No. 22, pp.2861-2864 (1986)

[SONA91] N. Sonehara, K. Nakane: *A Random IFS (Iterated Function System) Estimation of a Gray Scale Image Using Square Error Criterion and Reconstruction Characteristics Evaluation;* The Journal of the Institute of Television Engineers of Japan, Vol. 45, No. 8, pp. 1008-1012 (1991)

[SONA92] N. Sonehara, K. Nakane: *A Parallel Image Generation by an IFS (Iterated Function System) and an Adaptive IFS Estimation of the Gray Scale Image;* The Journal of the Institute of Image Electronics Engineers of Japan, Vol. 21, No. 5, pp. 486-493 (1992)

[SONA93] N. Sonehara, K. Nakane: *Image Generation and Inversion based on a Probabilistic Recurrent Neural Model;* Proc. IEEE-SP Workshop on Neural Networks for Signal Processing III, pp. 271-280 (1993)

[SPE91] D. Specht: *A General Regression Neural Network;* IEEE Transactions on Neural Networks, Vol.2, No.6, Nov. 1991, pp.568-567

[SPI90] Piet Spiessens, Bernard Manderick: *A Genetic Algorithm For Massively Parallel Computers;* in [ECK90], pp. 31-36

[SPI93] Marcus Spies: *Spracherkennung mit TANGORA: Medizinische Sprachmodelle und ihre Erweiterung auf Komposita in der deutschen Sprache;* in: S.J. Pöppl, H. Handels (Hrsg.) Mustererkennung 1993, Springer Verlag 1993

[SRR94] H. Speckmann, G Raddatz, W. Rosenstiel: *Considerations of geometrical and fractal dimension of SOM to get better learning results;* Proc. ICANN94, pp.342-345, Springer Verlag 1994

[STA91] J. Stark: *Iterated Function Systems as Neural Networks;* Neural Networks, Vol.4, pp. 679-690 (1991)

[STEE85] L. Steels, W. Van de Welde: *Learning in second generation expert systems;* in: Kowalik (ed), Knowledge based problem solving, Prentice Hall 1985

[STEIN61] K. Steinbuch: *Die Lernmatrix;* Kybernetik, Vol 1, pp.36-45 (1961)

[STEV66] C.F. Stevens: *Neurophysiology: A primer;* Wiley, New York 1966, pp. 21-22

[TAC93] J.G. Taylor, S. Coombes: *Learning High Order Correlations;* Neural Networks, Vol.6, pp.423-427, 1993

[TAK79] T. Tacheuchi, S.-I. Amari: *Formation of topographic maps and columnar microstructure in nerve fields;* Biological Cybernetics, Vol 3, pp.63-72 (1979)

[TIK77] A.Tikhonov, V.Arsenin: *Solutions of Ill-posed Problems;* Winston, Washinghton DC (1977)

[TON60] J.Tonndorf: *Dimensional Analysis of cochlear models;* J. Acoust. Soc. Amer., Vol 32, pp 293 (1960)

[TOR96a] K. Torkkola: Blind seperation of delayed sources based on information maximization. Proc. IEEE Int. Conf. on Acoustics, Speech & Signal Processing, Atlanta 1996

[TOR96b] K. Torkkola: Blind seperation of convolved sources based on information maximization. IEEE Workshop on Neural Networks for Signal Processing, Kyoto 1996

[TOU74] J.T. Tou, R.C. Gonzalez: *Pattern Recognition Principles;* Addison-Wesley Publ. Comp., 1974

[TR92] R. Trippi, E. Turban (Eds.): *Neural Networks in Finance and Investing;* Probus, Chicago Ill., 1992

[TRE89] Philip Treleaven: *Neurocomputers;* Int. Journ. of Neurocomputing, Vol 1/1, pp.4-31, Elsevier Publ. Comp. (1989)

[TRE90] M. Azema-Barac, M. Hewetson, M. Reece, J. Taylor, P. Treleaven, M. Vellasco: *PYGMALION Neural Network Programming Environment;* Proc. INNC-90, pp. 709-712, Kluwer Ac. Publ. 1990

[TSYP73] Tsypkin: *Foundations of the Theory of Learning Systems;* Academic Press, New York 1973

[TUR92] E. Turban, R.Trippi (eds.): *Neural Network Applications in Investment and Finance Services;* Probus Publishing, USA (1992)

[UES72] Uesa, Ozeki: *Some properties of associative type memories;* Journ. Inst. of El. and Comm. Eng. of Japan,Vol 55-D, pp.323-330 (1972)

[ULL89] S. Ullmann, Cognition Vol.32, pp.193-254 (1989)

[VAP79] V.Vapnik: *Estimation of Dependences Based on Empirical Data,* [in Russian] Nauka, Moscow 1979; (transl.) Springer Verlag, New York 1982

[VAP95] V Vapnik.: *The Nature of Statistical Learning Theory*, Springer-Verlag, New York (1995).

[VITT89a] E. Vittoz: *Analog VLSI Implementation of Neural Networks;* Proc. Journées d'électron., Ecole Polytechnique Fédérale, Lausanne 1989

[VITT89b] E. Vittoz, X. Arreguit: *CMOS integration of Herault-Jutten cells for seperation of sources;* in: C. Mead, M. Ismail (eds), Analog Implementation of Neural Systems, Kluwer Academic Publ. , Norwell 1989

[WAL91] J. Walter, T. Martinetz, K. Schulten: *Industrial Robot Learns Visuo-motor Coordination by Means of "Neural-Gas" Network;* in: T.Kohonen, K.Mäkisara, O.Simula, J. Kangas (Eds.): Artificial Neural Networks, Elsevier Sc. Publ., 1991, pp. 357-364

[WAN90] L.Wang, J.Ross: *On Dynamics of Higher Order Neural Networks: Existences of Oscillations and Chaos;* Proc. INNC-90, pp.945-947 (Paris 1990)

[WAT83] L. Watson, R. Haralick, O. Zuniga: *Constrained Transform Coding and Surface Fitting;* IEEE Transact. on Comm., Vol. COM-31, No. 5, pp. 717-726 (1983)

[WECH88] Harry Wechsler, George L. Zimmermann: *Invariant Object Recognition Using a Distributed Associative Memory;* in [AND88], pp. 830-839

[WEI90] A. Weigaend, D. Rumelhart, B. Huberman: *Generalization by Weight Elimination with Application to Forecasting;* in: [NIPS3]

[WEI92] A. Weigaend, B. Huberman, D. Rumelhart : *Predicting Sunspots and Exchange Rates with Connectionist Networks;* in: M. Casdagli, S.Eubank (Eds.), Nonlinear Modeling and Forecasting, SFI Studies in the Sciences of Complexity, Proc. Vol. XII, Addison Wesley 1992

[WEL90] A. Webb, D. Lowe: T*he Optimised Internal Representation of Multilayer Classifier Networks Performs Nonlinear Discriminant Analysis;* Neural Networks, Vol.3, pp. 367-375, 1990

[WEM91] N. Weymaere, J-P. Martens: *A Fast and Robust Learning Algorithm for Feedforward Neural Networks;* Neural Networks, Vol.4, pp. 361-369, (1991)

[WHI90] H. White: *Connectionist Nonparametric Regression: Multilayer Feedforward Networks Can Learn Arbitrary Mappings;* Neural Networks, Vol.3, pp.535-549 (1990)

[WHI92] H. White: *Artificial Neural Networks - Approximation and Learning Theory;* Blackwell, Oxford UK 1992

[WID60] B.Widrow, M. Hoff: *Adaptive switching circuits;* 1960 IRE WESCON Convention Record, New York: IRE, pp.96-104; auch in [ANDR88]

[WID85] B.Widrow, S. Stearns: *Adaptive Signal Processing;* Prentice Hall, Englewood Cliffs, N.J., 1985

[WID88] B.Widrow, R.Winter: *Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition;* IEEE Computer, Vol 21 No 3, pp.25-39 (1988)

[WILL69] D. Willshaw, O.Buneman, H. Longuet-Higgins: *Non-holographic associative memory;* Nature, Vol 222, pp. 960-962 (1969); auch in [ANDR88]

[WILL71] D. Willshaw: *Models of distributed associative memory;* Unpublished doctoral dissertion, Edinburgh University (1971)

[WILL76] D. Willshaw, C.van der Malsburg: *How patterned neural connections can be set up by self-organization;* Proc. Royal Soc. of London, Vol B-194, pp. 431-335 (1976)

[WILL85] R. Williams: *Feature Discovery through Error-Correction Learning*; ICS Report 8501, University of California and San Diego (1985)

[WIL76] G. Willwacher: *Fähigkeiten eines assoziativen Speichersystems im Vergleich zu Gehirnfunktionen;* Biol. Cybernetics, Vol. 24, pp. 181-198 (1976)

[WIL88] G.V. Wilson, G.S. Pawley: *On the Stability of the Travelling Salesman Problem Algorithm of Hopfield and Tank;* Biol. Cybernetics, Vol 58 pp.63-70 (1988)

[WIN89] Jack H. Winters, Christopher Rose: *Minimum Distance Automata in Parallel Networks for Optimum Classification;* Neural Networks,Vol.2,pp.127-132 (1989)

[WINTZ72] Paul A. Wintz: *Transform Picture Coding;* Proc. of the IEEE, Vol.60, No. 7, pp.809-820 (1972)

[XU92] Lei Xu, Erkki Oja, Ching Suen: *Modified Hebbian Learning for Curve and Surface Fitting;* Neural Networks, Vol. 5, pp.441-457 (1992)

[XU93] Lei Xu: *Least Mean Square Error Reconstruction Principle for Self-Organizing Neural-Nets;* Neural Networks, Vol.6, pp. 627-648 (1993)

[XUKO93] Lei Xu, A. Krzyzak, E. Oja: *Rival Penalized Competitive Learning for Clustering Analysis, RBF Net, and Curve Detection;* IEEE Transactions on Neural Networks, Vol. 4, No.4, pp. 636-649, July 1993

[XUKY94] Lei Xu, Adam Krzyzak, Alan Yuille: *On Radial Basis Function Nets and Kernel Regression: Statistical Consistency, Convergence Rates, and Receptive Field Size;* Neural Networks, Vol. 7, No.4, pp.609-628, (1994)

[Yak77] S.Yakowitz: *Computational Probability and Simulation;* Addison-Wesley (1977)

[ZUR65] R. Zurmühl: *Praktische Mathematik*; Springer Verlag Berlin 1965

13 Musterlösungen

Aufgabe 1.1) *Man errechne die Koeffizienten w_{ij} der Gewichtsmatrix **W** einer linearen Schicht, die als Eingabe die Koordinaten eines Dreiecks erhält und als Ausgabe die Koordinaten des um 90° gedrehten Dreiecks ausgibt. Das Dreieck habe beispielsweise die folgenden Koordinaten A = (–1,0), B = (0,–1), C = (0,1). Der Drehpunkt R ist der Nullpunkt. Die neuen Koordinaten sind also in diesem Fall A = (0,–1), B = (1,0), C = (–1,0).*

Es ist $x_A = (-1,0)^T$, $y_A = (0,-1)^T$
$x_B = (0,-1)^T$, $y_B = (1,0)^T$
$x_C = (0,1)^T$, $y_C = (-1,0)^T$

und

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Dies ergibt für jeden Punkt zwei Gleichungen mit vier Unbekannten. Mit zwei Punkten lassen sich also die vier Gewichte ausrechnen:

A: $0 = -1\ w_{11} + 0\ w_{12} = -w_{11}$
$-1 = -1\ w_{21} + 0\ w_{22} = -w_{21}$
B: $1 = 0\ w_{11} + -1\ w_{12} = -w_{12}$
$0 = 0\ w_{21} + -1\ w_{22} = -w_{22}$

Also ist $W = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$, was der Drehmatrix mit $\alpha = 90°$, $\sin\alpha = 1$, $\cos\alpha = 0$ entspricht. Die Probe mit Punkt C ist korrekt.

Aufgabe 1.2) *Man beweise: Die folgenden Muster (x_1,x_2) sind nicht linear separierbar, so daß die Klassenzugehörigkeit durch die XOR-Funktion beschrieben wird.*

x_1	x_2	XOR
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Dies bedeutet, dass die XOR-Funktion nicht durch ein Sigma-Neuron mit binärer Ausgabefunktion implementiert werden kann.

Die Eingabemenge (x_1,x_2) der XOR-Funktion

Fall	x_1	x_2	XOR(x_1,x_2)
a)	0	0	0
b)	0	1	1
c)	1	0	1
d)	1	1	0

ist genau dann linear separierbar, wenn es eine Gerade in der Ebene gibt der Form

$$g(\mathbf{w},\mathbf{x}) = w_1x_1 + w_2x_2 + w_3 ,$$

und >0 bei Klasse 1 mit XOR=1 und <0 bei Klasse 2 mit XOR=0 ist.
Angenommen, es existiert eine solche Gerade. Versuchen wir, ihre Koeffizienten zu bestimmen. Aus den Werten für x_1,x_2 in der Tabelle für alle vier Fälle folgt für $g(\mathbf{x})$

Fall	$g(\mathbf{x})$
a)	$w_3 < 0$
b)	$w_2+w_3 > 0$
c)	$w_1+w_3 > 0$
d)	$w_1+w_2+w_3 < 0$

Aus b)+c) folgt $w_1 + w_2 + 2w_3 > 0$ oder $w_1 + w_2 + w_3 > -w_3 > 0$ mit a). Dies ist allerdings ein Widerspruch zu d), so dass es keine Gerade mit den angenommenen Eigenschaften geben kann: die Annahme der linearen Separierbarkeit trifft nicht zu.

Aufgabe1.3

a) Man berechne die Parameter einer affinen Transformation (und damit φ, c, s_1, s_2) so, dass die Muster (x_1,x_2) auf die folgenden neuen Koordinaten (z_1,z_2) abgebildet werden:

x_1	x_2	z_1	z_2
0	*0*	*0*	*-1*
0	*1*	*1*	*0*
1	*0*	*-1*	*0*
1	*1*	*0*	*1*

b) Wie sieht das dazu gehörende Netzwerk aus? Welche Werte haben die Gewichte?

c) Nehmen Sie als Ausgabefunktion $S_i(z)=z^2$ an. Welche logische Funktionen implementieren die Ausgaben $S_i(\mathbf{x})$?

a) Es gibt vier mögliche Eingabekombinationen und damit vier Muster $\mathbf{x}_i$, die jeweils auf einen Aktivitätsvektor $\mathbf{z}_i$ abgebildet werden. Dazu werden die Eingaben um eine Komponente konstant 1 erweitert auf den Spaltenvektor $\mathbf{x} = (x_1,x_2,1)^T$. Wir erhalten für jede der vier möglichen Eingabemuster eine Gleichung $\mathbf{z}_i = \mathbf{W}\mathbf{x}_i$, wobei $\mathbf{W}$ eine affine Transformationsmatrix der Dimension 3×3 ist. Fassen wir die vier Eingabemuster zu einer Matrix $\mathbf{X}$ aus vier Spalten zusammen und die Ausgabe ebenfalls zu einer Matrix $\mathbf{Z}$, so lautet die Matrixgleichung der affinen Transformation mit allen vier Mustern $\mathbf{Z} = \mathbf{W}\mathbf{X}$ oder

$$\begin{bmatrix} 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Dies lässt sich sukzessiv lösen. Die erste Gleichung für i=1 lautet

$$0 = 0\cdot w_1 + 0\cdot w_2 + 1\cdot w_3 \Rightarrow w_3 = 0$$

$$-1 = 0\cdot w_4 + 0\cdot w_5 + 1\cdot w_6 \Rightarrow w_6 = -1$$

Die zweite Gleichung mit i=2 ergibt

$$1 = 1\cdot w_1 + 0\cdot w_2 + 1\cdot w_3 \Rightarrow w_1 + w_3 = w_1 = 1$$

$$0 = 1\cdot w_4 + 0\cdot w_5 + 1\cdot w_6 \Rightarrow w_4 - 1 = 0 \Rightarrow w_4 = 1$$

Und die dritte Gleichung mit i=3 ergibt

$$-1 = 0\cdot w_1 + 1\cdot w_2 + 1\cdot w_3 \Rightarrow w_2 + w_3 = w_2 = -1$$

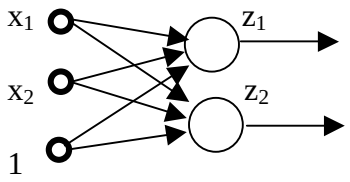
$$0 = 0\cdot w_4 + 1\cdot w_5 + 1\cdot w_6 \Rightarrow w_5 - 1 = 0 \Rightarrow w_5 = 1$$

Also ist

$$\mathbf{W} = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

Im Vergleich mit der allgemeinen affinen Transformationsmatrix ist klar, dass $c_1=c_2=c$ ist und $\cos\varphi = \sin\varphi$. Also ist $\varphi=45°$ und $c=\sqrt{2}$.

b) Das entsprechende Aktivitätsnetz der formalen Neuronen ist



und die Aktivitätsgleichungen lauten

$$z_1 = (x_1 - x_2)$$

$$z_2 = (x_1 + x_2) - 1$$

c) Die Ausgabefunktionen sind mit $y = S(z) = z^2$

x_1	x_2	z_1	z_2	y_1	y_2
0	0	0	–1	0	1
0	1	–1	0	1	0
1	0	1	0	1	0
1	1	0	1	0	1

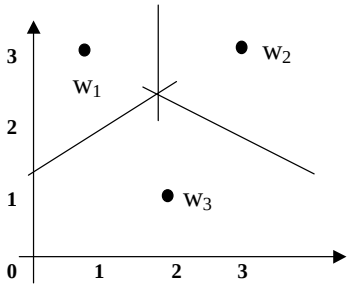
Im Vergleich zeigt sich, dass y_1 die XOR-Funktion ist und y_2 die inverse XOR-Funktion.

Aufgabe 1.4

Gegeben seien 3 Klassenprototypen mit den Koordinaten $w_1 = (1,3)$, $w_2 = (3,3)$ und $w_3 = (2,1)$. Geben Sie die Geradengleichungen an für die drei linearen Klassengrenzen, die jeweils durch $g_{ik} = \{\boldsymbol{x} / |\boldsymbol{x}\text{-}\boldsymbol{w}_i| = |\boldsymbol{x}\text{-}\boldsymbol{w}_k|\}$ zwischen Klasse i und k definiert seien.

Wie lautet die Koordinate des Schnittpunkts aller drei Grenzen ?

Die Klasseneinteilungen sind wie folgt:



$\mathbf{w}_1 = (1,3)^T$, $\mathbf{w}_2 = (3,3)^T$, $\mathbf{w}_3 = (2,1)^T$

Die Geradengleichungen sind :

$g_{12} = \{\mathbf{x}|x_1=2, x_2 \in \Re\}$

$g_{13} = \{\mathbf{x}|x_2-a_1x_1-b_1=0\}$ ist senkrecht auf der Verbindungsgerade zwischen $\mathbf{w}_1$ und $\mathbf{w}_2$, also $\mathbf{h}(\mathbf{w}_1-\mathbf{w}_2)=0$, so daß $-h_1+2h_2=0$ oder $2h_2=h_1$. Dies entspricht einer Steigung von $a_1=1/2$. Einsetzen vom bekannten Punkt zwischen beiden Zentren auf (1.5,2) ergibt $b_1=1.25$.

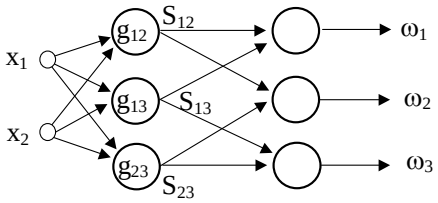
Das gleiche gilt für g_{23}. Hier ist $a_2=-1/2$ und $b_2=3{,}25$.

Der gemeinsame Schnittpunkt ist bei $g_{12}=g_{13}$ oder bei $x_1=2$, $x_2=2.25$.

Aufgabe 1.5 *Klassentrennung*

*Bei der Eingabe von $\mathbf{x} = (x_1,x_2)$ soll als Ausgabe der Index der Klasse aus Aufgabe **Fehler! Verweisquelle konnte nicht gefunden werden.**, in dessen Bereich $\mathbf{x}$ fällt, mit Hilfe formaler Neuronen ermittelt werden. Verwenden Sie zwei Schichten aus binären Neuronen. Die erste Schicht implementiert die Diskriminanzfunktion. Die zweite Schicht hat als Ausgabe für jede Klasse genau ein Ausgabeneuron, das genau dann 1 wird, wenn die Klasse vorliegt, und sonst null ist. Wie lauten dabei die Gewichte der Neuronen? Nutzen Sie dabei die Ergebnisse aus a). Wie ändert sich das Netz, wenn in der ersten Schicht Distanzneuronen verwendet werden?*

Das neuronale Netz aus zwei Schichten sieht folgendermaßen aus:



Aus den drei Gleichungen für die Grenzen folgen direkt die drei Gewichte jedes Neurons.

g_{12} : $w_1=+1$, $w_2=0$, $w_3=-2$

g_{13} : $w_1=-2$, $w_2=4$, $w_3=-5$

g_{23} : w_1=+2, w_2=4, w_3=−13

Es ergibt sich also folgende Funktionstabelle:

	g_{12}	g_{13}	g_{23}
ω_1	< 0	> 0	*
ω_2	> 0	*	> 0
ω_3	*	< 0	< 0

Die binäre Ausgabe $S_{bin}(\mathbf{w},\mathbf{x})$ der drei Neuronen ist =1, wenn $g(x)\geq 0$ und sonst 0. Sei $S_{ij}=S(g_{ij})$, so bedeutet die obige Tabelle für die binäre Ausgabe

	S_{12}	S_{13}	S_{23}	Klasse liegt vor, wenn...
ω_1	0	1	*	NOT(S_{12}) AND S_{13}
ω_2	1	*	1	S_{12} AND S_{13}
ω_3	*	0	0	NOT(S_{13}) AND NOT(S_{23})

Bekanntlich kann man jede logische Funktion mit einem binären Neuron und entsprechenden Gewichten implementieren. Die Gewichte der zweiten Schicht lassen sich direkt aus den obigen logischen Ausdrücken ablesen.

ω_1: w_1=−1, w_2=+1, w_3=1, ω_2: w_1=+1, w_2=+1, w_3=2, ω_3: w_1=−1, w_2=−1, w_3=0

Mit Distanzneuronen werden in der ersten Schicht die Eingaben direkt mit den Klassenprototypen verglichen. Das Ergebnis (der Abstand) muß dann in der zweiten Schicht untereinander verglichen werden. Das Ergebnis dieser drei Vergleiche muß dann in einer weiteren Schicht wie oben ausgewertet werden.

Aufgabe 1.6

Man implementiere das Klassifizierungsnetz als Programm und gebe den Pseudocode an.

Ein Programm in Pseudocode, das das obige neuronale Netz implementiert, kann folgendermaßen geschrieben sein:

```
Program  Klassifikation;
(* implementiert eine Klassifikation von drei Klassen *)

   CONST n:=3; M:=3;
   TYPE  Vector =  ARRAY[1..n] OF REAL;
   VAR      w1,w2:  ARRAY[1..M]OF Vector; (* weights 1.+2.layer *)
    x1,x2: Vector;         (* input 1.+2.layer *)

   Function S(w,x:Vector): INTEGER;
   (* Neuronenfunktion *)
   BEGIN Sum:=0.0;
      FOR j:=1 TO n DO                    (* Skalarprodukt*)
         Sum:=Sum+(w[j]*x[j])
      END FOR
      IF Sum>=0 THEN S:=1 ELSE S:=0; (* Binäre Ausgabe *)
   END
```

```
LOOP
  Input x1[1],x1[2]; x1[3]:=1;
  FOR i:=1 TO M DO              (*Aktivität 1.Schicht *)
      x2[i]= S(w1[i],x1);
  ENDFOR
  FOR i:=1 TO M DO              (*Aktivität 2.Schicht *)
   IF S(w2[i],x2)=1 THEN WriteLn(„Klasse",i,"liegt vor.")
  ENDFOR
ENDLOOP

BEGIN (*main*)
 w1:=((1, 0,–2),(–2, 4,–5),(2, 4,–13));
 w2:=((1, 1, 1),(1, 1, 2),(–1,–1, 0))
END
```

Man beachte, daß jede Schicht modular für sich funktioniert und damit Zeitparallelität für alle Neuronen einer Schicht simuliert wird. Die Eingabedaten und die Gewichte sind als Datenstrukturen getrennt, um die sich ändernden und die konstanten Daten zu trennen.

Aufgabe 1.7

Man beweise: Eine Ähnlichkeitsfunktion $f_k(\mathbf{x}) = a_k (\mathbf{x}\text{-}\mathbf{c}_k)^2$ impliziert

a) bei gleichen Koeffizienten $a_k = a_i$ eine Hyperebene als Klassengrenze

b) bei ungleichen Koeffizienten $a_k \neq a_i$ eine Hyperparabel als Klassengrenze

Auf der Klassengrenze zwischen den Klassen *k* und *i* gilt für alle Punkte $\{\mathbf{x}\}$

$$a_i(\mathbf{x}-\mathbf{c}_i)^2 = f_i(\mathbf{x}) = f_k(\mathbf{x}) = a_k(\mathbf{x}-\mathbf{c}_k)^2$$

$$\text{oder} \quad \mathbf{x}^2(a_i-a_k) - \mathbf{x}\cdot 2(a_i\mathbf{c}_i-a_k\mathbf{c}_k) + (a_i\mathbf{c}_i^2-a_k\mathbf{c}_k^2) = 0$$

a) Bei gleichen Koeffizienten $a_k = a_i$ vereinfacht sich dies mit

$$(a_i\mathbf{c}_i^2-a_k\mathbf{c}_k^2) = r,\ 2(a_i\mathbf{c}_i-a_k\mathbf{c}_k) = \mathbf{s}$$

zu der Gleichung

$$\mathbf{x}\cdot\mathbf{s} + r = 0$$

was nach der Division durch $|\mathbf{s}|$ der Hesse'schen Normalform einer Hyperebene entspricht.

b) Bei ungleichen Koeffizienten $a_k \neq a_i$ ist dies eine quadratische Gleichung oder Hyperparabel.

Kapitel 2

Aufgabe 2.1

*Fehler! Verweisquelle konnte nicht gefunden werden.**.1)** Man zeige: a^* aus Gleichung **Fehler! Verweisquelle konnte nicht gefunden werden.** gibt nicht nur die optimale Aktivität für einen maximalen, erwarteten Abstand, sondern maximiert auch gleichzeitig die Anzahl der verschiedenen Muster*

*Fehler! Verweisquelle konnte nicht gefunden werden.**.2)** Programmieren Sie einen Assoziativspeicher für 3 Muster und speichern Sie in einem Lernzyklus die Muster*

$$\boldsymbol{x}^1=(0\ 1\ 0), \boldsymbol{y}^1=(1\ 0);\ \boldsymbol{x}^2=(1\ 0\ 0), \boldsymbol{y}^2=(0\ 1);\ \boldsymbol{x}^3=(0\ 0\ 1), \boldsymbol{y}^3=(1\ 1);$$

ab. Testen Sie ihren Speicher durch Eingabe der Schlüssel $\boldsymbol{x}^i$.

*Fehler! Verweisquelle konnte nicht gefunden werden.**.3)** Versuchen Sie nun, stattdessen die Tupel $\boldsymbol{x}^1=(0\ 1\ 1)$, $\boldsymbol{y}^1=(1\ 0)$ und $\boldsymbol{x}^2=(1\ 1\ 0)$, $\boldsymbol{y}^2=(0\ 1)$ abzuspeichern und wieder auszulesen. Es geht nicht. Warum?*

Verwenden Sie zur Abhilfe eine binäre Ausgabefunktion und eine adäquate Lernrate γ_k.

Aufgabe2.2 Newton-Iteration

*Im Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** haben wir das Gradientenverfahren zur Verbesserung eines Parameters w für die Nullstelle einer Funktion kennengelernt. Betrachten wir nun ein anderes Verfahren. Dazu gehen wir von dem Mittelwertsatz aus. Für den Differenzenquotienten der Funktion f existiert ein Punkt ξ, an dem der Wert der Ableitung gleich dem Quotienten ist*

$$f'(\xi) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Kennen wir also die analytische Form der Ableitung, so können wir den Wert x_2 ausdrücken als

$x_2 = x_1 + (f(x_2)-f(x_1))/f'(\xi)$

Haben wir nur kleine Intervalle mit $x_1 \approx x_2$, so gilt

$x_2 \approx x_1 + (f(x_2)-f(x_1))/f'(x_1)$

so dass wir als neue Iterationsvorschrift für den Parameter w_{k+1} und die gesuchte Nullstelle $f(w_{k+1})=0$ formulieren können

$$w_{k+1} = w_k - f(w_k)/f'(w_\kappa) \qquad f'(w) \neq 0$$

a) Benutzen Sie die oben entwickelte Formel, um das Minimum der Funktion $R(w) = 3w^2 - 12w + 12$ zu erhalten. Welchen Wert erhalten Sie für das Minimum?

Die Funktion ist $R(w) = 3w^2 - 12w + 12 = 3(2-w)^2$, also eine Parabel mit dem Minimum 0 bei w=2. Die Ableitung ist $R'(w) = f(w) = 6w-12$ und die zweite Ableitung $R''(w) = f'(w) = 6$.

Die Iteration ist also $w_{k+1} = w_k - (6w_k-12)/6 = 2$. Die Lösung ergibt sich also in einem Schritt.

b) Ersetzen Sie die Ableitung, die nicht in allen Fällen bekannt ist, statt dessen durch den beobachteten Differenzenquotienten der letzten beobachteten Werte $f(w_k)$ und $f(w_{k-1})$. Wie lautet die Iterationsgleichung dann ?

Führen Sie mit den Startwerten $w_0 = 0$, $w_1 = 1$ Iterationsschritte durch bis $|w_k-w_{k-1}| < 0{,}01$ erreicht ist.

Der Differenzenquotient ist bei

$$f'(w_k) \approx \frac{f(w_k) - f(w_{k-1})}{w_k - w_{k-1}} \text{ so dass die Formel zu}$$

$$w_{k+1} = w_k - f(w_k)/f'(w_\kappa) = w_k - f(w_k)\ \frac{w_k - w_{k-1}}{f(w_k) - f(w_{k-1})}$$

wird.

Mit $w_1=1$, $w_0=0$ und $f(w)=6w-12$ ist $f(w_0)=-12$, $f(w_1)=-6$

$w_2 = 1 -(6-12)\cdot(1-0)/(-6+12) = 1+6/6 = 2$, $f(w_2)= 0$

und

$w_3 = 2 -(12-12)\cdot(2-1)/(+6) = 1+0 = 2$, $f(w_3)= 0$

so daß mit $w_2 = w_3$ die Abbruchbedingung erreicht ist.

Aufgabe 2.3 Lernrate

Zeigen Sie, dass die Iteration von ***Fehler! Verweisquelle konnte nicht gefunden werden.*** *bei konstanter Lernrate $\gamma < 1$ alle $\mathbf{x}(i)$ nicht gleich gewichtet, sondern ihr Einfluss auf den Gewichtsvektor nimmt exponentiell ab. Dazu zeigen Sie durch vollständige Induktion, dass im k-ten Schritt*

$$w(k) = (1-\gamma)w(k-1) + \gamma x(k) = (1-\gamma)^k w(0) + \gamma\Sigma^k_{i=1}(1-\gamma)^{k-i}x(i)$$

gilt. Was folgt beim Grenzübergang $\lim_{k\to\infty}\langle \mathbf{w}(k)\rangle = ?$.

Es ist die stochastische Approximation

$$\mathbf{w}(k) = \mathbf{w}(k-1) + \gamma\,(\mathbf{x}(k)- \mathbf{w}(k-1))$$

Start: Nach der ersten Iteration ist

$$\mathbf{w}(1) = \mathbf{w}(0) + \gamma\,(\mathbf{x}(1)- \mathbf{w}(0)) = (1-\gamma)\mathbf{w}(0) + \gamma\,\mathbf{x}(1)$$

Durchführung: Sei nach dem k-ten Schritt

$$\mathbf{w}(k) = (1-\gamma)\mathbf{w}(k-1) + \gamma\,\mathbf{x}(k) = (1-\gamma)^k\mathbf{w}(0) + \gamma\Sigma^k_{i=1}(1-\gamma)^{k-i}\mathbf{x}(i)$$

gültig. Dann ist im k+1-ten Schritt

$\mathbf{w}(k+1) = (1-\gamma)\mathbf{w}(k) + \gamma\,\mathbf{x}(k+1) = (1-\gamma)\,[(1-\gamma)^k\mathbf{w}(0) + \gamma\Sigma^k_{i=1}(1-\gamma)^{k-i}\mathbf{x}(i)] + \gamma\,\mathbf{x}(k+1) = [(1-\gamma)^{k+1}\mathbf{w}(0) + \gamma\Sigma^k_{i=1}(1-\gamma)^{k+1-i}\mathbf{x}(i)] + \gamma\,(1-\gamma)^{k+1-(k+1)}\mathbf{x}(k+1)$

$= (1-\gamma)^{k+1}\mathbf{w}(0) + \gamma\Sigma^{k+1}_{i=1}(1-\gamma)^{k+1-i}\mathbf{x}(i)$

q.e.d.

Einfluß von γ: Sei $\alpha=(1-\gamma)$. Dann ist

$$\mathbf{w}(k) = (1-\gamma)^k\mathbf{w}(0) + \gamma\Sigma^k_{i=1}(1-\gamma)^{k-i}\mathbf{x}(i) = \alpha^k\mathbf{w}(0) + \gamma\Sigma^{k-1}_{i=1}\alpha^{k-i}\mathbf{x}(i) + \gamma\mathbf{x}(k)$$

$$= \alpha^k\mathbf{w}(0) + \gamma[\,\mathbf{x}(k) + \alpha^1\mathbf{x}(k-1) + \alpha^2\mathbf{x}(k-2) + \ldots + \alpha^{k-1}\mathbf{x}(1)]$$

Neben dem Anfangsgewicht, das mit wachsendem k exponentiell geringer addiert wird, geht in das Gewicht der exponentiell mit k geringer gewichtete Anteil der vorherigen Muster $\alpha^t\mathbf{x}(k-t)$ ein. Damit ist der Einfluß jeden Musters mit dem Faktor $\alpha<1$ geringer als der des nachfolgenden Musters; absolut gesehen verringert sich der Einfluß zusätzlich bei jeder Iteration um den Faktor α.

Der Grenzwert des Erwartungswertes ist

$$\lim_{k\to\infty}\langle\mathbf{w}(k)\rangle = \lim_{k\to\infty}\langle\alpha^k\mathbf{w}(0) + \gamma\sum_{i=1}^{k}\alpha^{k-i}\,\mathbf{x}(i)\rangle$$

$$= \lim_{k\to\infty}\alpha^k\mathbf{w}(0) + \lim_{k\to\infty}\langle\mathbf{x}\rangle\gamma\sum_{i=1}^{k}\alpha^{k-i}$$

$$= 0 + \lim_{k\to\infty}\gamma\,\langle\mathbf{x}\rangle\sum_{i=0}^{k-1}\alpha^k$$

$$= \langle\mathbf{x}\rangle\lim_{k\to\infty}\gamma\,\frac{1-\alpha^k}{1-\alpha} = \langle\mathbf{x}\rangle\lim_{k\to\infty}1-\alpha^k = \langle\mathbf{x}\rangle$$

der Erwartungswert der Eingabe.

Aufgabe 2.4 Adaline

Angenommen, Sie haben die Punkte $A_1=(0{,}3\ 0{,}7)$, $B_1=(-0{,}6\ 0{,}3)$. Aus den beiden Klassen A und B. Ihr Gewichtsvektor Ihres ADALINE ist initial $\mathbf{w}_0 = (-0{,}6\ 0{,}8)$. und $\gamma=0{,}5$, wobei die Ausgabe +1 bei Klasse A und –1 bei Klasse B sein soll. Zeichnen Sie sich die Situation (Eingaben und Gewichtsvektor) auf und führen Sie eine jeweils Iteration mit dem Widrow-Hoff-Algorithmus durch.

a) Geben Sie als Eingabe $\mathbf{x}(1)=A_1$ vor. Was passiert? Ist die Klassifizierung korrekt?

b) Geben Sie nun als Eingabe $\mathbf{x}(2)=B_1$ vor. Was passiert?

*c) Zeichnen Sie nun die resultierende Lage von **w**(1) und **w**(2) ins Diagramm ein. Klassifiziert ADALINE mit diesem Gewichtsvektor die Muster A_1 und B_1 korrekt?*

Aufgabe 2.5 Bayesdiagnose

Sei eine Datenquelle gegeben, die vier Muster $x_1, .. ,x_4$ aus drei Klassen ω_1, ω_2 ω_3 hervorbringt. Die Wahrscheinlichkeiten sind in der Tabelle aufgeführt.

P(xi|ωk)	ω1	ω2	ω3	P(xi)
x1	0,3	0,7	0	0,41
x2	0,2	0,1	0,3	0,18
x3	0,2	0,2	0,3	0,23
x4	0,3	0	0,4	0,18
P(ωk)	0,2	0,5	0,3	

Wie lautet der günstigste Klassifizierer für $x_1, .. ,x_4$?

Es ergibt sich folgende Tabelle für die Likelihood-Werte $P(\omega_i|x_j)$:

P(wi|xj)	w1	w2	w3
x1	0,15	0,85	0,00
x2	0,22	0,28	0,50
x3	0,17	0,43	0,39
x4	0,33	0,00	0,67

Also ist der Günstigste Klassifizierer für x_1 und x_3 die Klasse 2 und für x_2 sowie x_4 die Klasse 3. Die Klasse 1 sollte nie gewählt werden.

13.1.1 Aufgaben

***1)** Trainieren Sie ein XOR-Netz aus **Fehler! Verweisquelle konnte nicht gefunden werden.** mit Backpropagation. Welches Problem ergibt sich ? Lösen Sie es mit sigmoidalen Funktionen. Was passiert, wenn Sie die Steilheit k der Fermi-Funktion erhöhen?*

***2)** Lösen Sie die gleiche Aufgabe wie in 1) mittels der Zielfunktion der maximalen Klassifikationswahrscheinlichkeit aus Gl.**Fehler! Verweisquelle konnte nicht gefunden werden.**.*

Aufgabe: Realisierung von XOR mit einer Schicht höherer Neuronen

Die nicht linear separierbaren XOR-Funktionswerte können mit einfachen Neuronen also in mehreren Schichten separiert werden. Es gibt aber eine Möglichkeit, doch noch mit Hilfe einer einzigen Schicht das Ziel zu erreichen. Betrachten wir dafür zunächst die Funktionstabelle der XOR-Funktion; zuerst in der üblichen 0,1 Kodierung und dann in der Kodierung $1\rightarrow$-1, $0\rightarrow$+1.

x_1	x_2	XOR	$\rightarrow$	x_1	x_2	XOR
0	0	0		+1	+1	+1
0	1	1		+1	-1	-1
1	0	1		-1	+1	-1
1	1	0		-1	-1	+1

(4.1.6)

In der zweiten Tabelle bemerken wir eine Regelmäßigkeit: Die XOR-Funktion ist gerade das Produkt aus den beiden Eingaben x_1 und x_2: $XOR(x_1,x_2)=x_1x_2$. Ein Neuron, das eine solche Korrelation bemerken kann, haben wir aber bereits in Abschnitt 1.2 kennengelernt: Es ist ein Neuron mit Synapsen zweiter Ordnung (Sigma-Pi-Unit). Die Aktivität dieses Neurons ist

$$z_i = w_i^{(0)} + \sum_j w_{ij}^{(1)} x_j + \sum_{jk} w_{ijk}^{(2)} x_j x_k$$

und für unser spezielles XOR Problem mit der Ausgabefunktion (1.2.1b)

$$y_{XOR}=\mathrm{sgn}(z) = \mathrm{sgn}(w^{(2)} x_1 x_2)$$

Es reichen die Bedingungen $w_i^{(0)}=w_i^{(1)}=0$, $w^{(2)}>0$ völlig aus, mit diesen einem Neuron die XOR-Funktion zu implementieren und demonstrieren damit die Möglichkeit, mehrstufige Netze aus einfachen Neuronen durch wenige Neuronen mit höheren Synapsen zu ersetzen. Dies ist auch ein Beispiel für die in Kapitel 1.2.3 erwähnte verallgemeinerte lineare Separierung: die lineare Diskriminanzfunktion ist $g(\mathbf{x})=w^{(2)}y$ mit $y(\mathbf{x})=x_1x_2$.

Im XOR Beispiel wurden binäre Korrelationen anstatt durch Mehrschichtennetze durch höhere Synapsen in einer Schicht realisiert. Umgekehrt lassen sich aber auch die nicht-binären Korrelationen von höheren Synapsen durch Mehrschichtennetze realisieren. Beispielsweise kann man bei Neuronen mit der Ausgabefunktion $S(z) = z^2$ mit der ersten Schicht die Terme x_1^2, x_2^2 und $(x_1+x_2)^2$ bilden; in der zweiten Schicht erfolgt eine linear gewichtete Addition dieser Terme zu $y= +(x_1+x_2)^2-x_1^2 -x_2^2 = x_1x_2$. In Abbildung 4.1.6 ist ein solches Netz zu sehen.

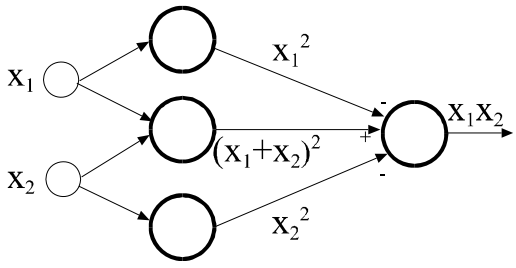


Abb 4.1.6 Realisierung einer Synapse zweiter Ordnung

Aufgabe 1.1
Mit einem Distanzneuron muß nur in *n* Schritten der kleinste Abstand von den gegebenen Prototypen festgestellt werden. Ein Distanzneuron hat den Code

```
Function S(w,x:ARRAY OF REAL)
BEGIN Sum:=0.0;
   FOR j:=1 TO n DO
      Sum:=Sum+(w[j]-x[j])*(w[j]-x[j])
   END FOR
   S = Sum;
END
```

Der Hauptcode ist

```
LOOP
   Input x1,x2
   Class = 0; min:=MAXREAL;
   FOR i:=1 TO M DO           // Für alle Klassen
        dist.= S(A[i],x);
     IF dist<min THEN min:=dist; Class:=i;
   ENDFOR
ENDLOOP
```

Aufgabe 4.1 *Ein ADALINE aus Papier*

Angenommen, Sie haben die Punkte $A_1=(0{,}3\ 0{,}7)^T$, $B_1=(-0{,}6\ 0{,}3)^T$. Aus den beiden Klassen A und B. Ihr Gewichtsvektor Ihres ADALINE ist initial $\mathbf{w}_0 = (-0{,}6\ 0{,}8)^T$ und $\gamma=0{,}5$, wobei die Ausgabe +1 bei Klasse A sein soll. Zeichnen Sie sich die Situation (Eingaben und Gewichtsvektor) auf und führen Sie jeweils eine Iteration mit dem Widrow-Hoff-Algorithmus durch.

a) Geben Sie als Eingabe A_1 vor. Was passiert? Ist die Klassifizierung korrekt?

b) Geben Sie nun als Eingabe B_1 vor. Was passiert?

c) Zeichnen Sie nun die resultierende Lage von $\mathbf{w}_1$ und $\mathbf{w}_2$ ins Diagramm ein. Klassifiziert ADALINE mit diesem Gewichtsvektor die Muster A_1 und B_1 korrekt?

Es ist $\mathbf{w}_0 = (-0{,}6\ 0{,}8)^T$ und das erste Muster der Klasse A ist $A_1=(0{,}3\ 0{,}7)^T$.

a) Die Klassifizierung errechnet sich mit

$z(1) = \mathbf{w}^T(0)\cdot\mathbf{x}(1) = -0{,}6\times0{,}3 + 0{,}8\times0{,}7 = -0{,}18+0{,}56 = +0{,}38$ und $S_{bin}(z) = +1$ zu Klasse A. Dies ist korrekt. Mit dem Lernalgorithmus

$\mathbf{w}(t) = \mathbf{w}(t-1) - \gamma(t)(\mathbf{w}^T\mathbf{x}-L(\mathbf{x}))\mathbf{x}/|\mathbf{x}|^2$ *Widrow-Hoff Lernregel*

und $|A_1|^2= 0{,}09+0{,}49=0{,}58$ ist der neue Gewichtsvektor

$w(1) = (-0{,}6\ 0{,}8)^T - 0{,}5\cdot (0{,}38-1)\cdot (0{,}3\ 0{,}7)^T/0{,}58 = (-0{,}6\ 0{,}8)^T + 0{,}53\cdot(0{,}3\ 0{,}7)^T$

$= (-0{,}44\ \ 1{,}17)$

b) Mit $B_1=(-0{,}6\ 0{,}3)^T$ ist die Klassifizierung

$z(2) = \mathbf{w}^T(1)\cdot\mathbf{x}(2) = 0{,}44\times0{,}6 + 1{,}17\times0{,}3 = 0{,}264+0{,}351 = +0{,}615$ und $S_{bin}(z) = +1$

zu Klasse A. Dies ist falsch.

Also muss der Gewichtsvektor verbessert werden. Mit $|B_1|^2= 0{,}36+0{,}09=0{,}45$ und $L(B)= -1$ ist

$\mathbf{w}(2) = \mathbf{w}(1) - 0{,}5(0{,}615+1)\ B_1/0{,}45 = (-0{,}44\ \ 1{,}17)^T - (-0{,}6\ \ 0{,}3)^T\cdot0{,}8075/0{,}45$

$= (-0{,}6\ 0{,}8)^T + (1{,}076\ -0{,}538)^T = (0{,}476\ \ 0{,}261)^T$

c) Mit $\mathbf{w}(2) = (0{,}476\ \ 0{,}261)^T$ werden klassifiziert

A_1: $z = \mathbf{w}^T(1)\mathbf{x}_A = 0{,}476\times0{,}3 + 0{,}261\times0{,}7 = 0{,}1428 + 0{,}1827 = +0{,}3255$ und $S_{bin}(z) = +1$ korrekt zu Klasse A

B_1: $z = \mathbf{w}^T(1)\mathbf{x}_B = -0{,}476\times0{,}6 + 0{,}261\times0{,}3 = -0{,}2856 + 0{,}0783 = -0{,}2073$ und $S_{bin}(z) = -1$ korrekt zu Klasse B.

Aufgabe 4.2

Sei ein lineares Neuron mit $y=\mathbf{w}^T\mathbf{x}$ gegeben. Zeigen Sie, daß der durch $\mathbf{x}' := \mathbf{x}-y\mathbf{w}$ definierte Eingabevektor senkrecht auf $\mathbf{w}$ steht, wenn $\mathbf{w}$ auf eins normiert ist.

Das Skalarprodukt aus $\mathbf{x}'$ und $\mathbf{w}$ ist $\mathbf{w}^T\mathbf{x}' = \mathbf{w}^T\mathbf{x}-y\mathbf{w}^T\mathbf{w} = y - y = 0$ da $\mathbf{w}^T\mathbf{w} = 1$. Dies bedeutet, dass $\mathbf{x}'$ senkrecht auf $\mathbf{w}$ steht.

Aufgabe 4.3

Man zeige, dass für den quadratischen Fehler bei orthonormaler Basis und linearen Neuronen die Gleichung gilt

$$R(\mathbf{w}) = \langle(\mathbf{x}-\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2-2(\mathbf{x}^T\hat{\mathbf{x}})+(\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2\rangle-\langle(\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2\rangle-\langle(\hat{\mathbf{y}})^2\rangle$$

Der Fehler ist

$$R(\mathbf{w}) = \langle(\mathbf{x}-\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2-2(\mathbf{x}^T\hat{\mathbf{x}})+(\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2\rangle-\langle 2(\mathbf{x}^T\hat{\mathbf{x}})\rangle+\langle(\hat{\mathbf{x}})^2\rangle$$

Bei zentrierter Eingabe sind die konstanten Koeffizienten $a_i = \langle y_i\rangle = \langle \mathbf{w}_i^T\mathbf{x}\rangle = \mathbf{w}_i^T\langle\mathbf{x}\rangle =$ null, so dass das Skalarprodukt

$$\langle \mathbf{x}^T\hat{\mathbf{x}}\rangle = \left\langle\left(\sum_{i=1}^{n} y_i\mathbf{w}_i\right)\left(\sum_{i=1}^{m} y_i\mathbf{w}_i + \sum_{j=m+1}^{n} a_j\mathbf{w}_j\right)\right\rangle =$$

$$\left\langle\left(\sum_{i=1}^{n} y_i\mathbf{w}_i\right)\left(\sum_{i=1}^{m} y_i\mathbf{w}_i + 0\right)\right\rangle$$

wird. Mit der Bedingung der orthonormalen Basis $\mathbf{w}_i^T\mathbf{w}_k = \delta_{ik}$ wird dies zu

$$= \left\langle\sum_{i=1}^{m} y_i^2\right\rangle =$$

$$\left\langle\left(\sum_{i=1}^{m} y_i\mathbf{w}_i\right)\left(\sum_{i=1}^{m} y_i\mathbf{w}_i\right)\right\rangle = \left\langle\left(\sum_{i=1}^{m} y_i\mathbf{w}_i + \sum_{j=m+1}^{n} a_j\mathbf{w}_j\right)\left(\sum_{i=1}^{m} y_i\mathbf{w}_i + \sum_{j=m+1}^{n} a_j\mathbf{w}_j\right)\right\rangle =$$

$$\langle \hat{\mathbf{x}}^T\hat{\mathbf{x}}\rangle \qquad (1)$$

so dass

$$\langle(\mathbf{x}-\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2-2(\mathbf{x}^T\hat{\mathbf{x}})+(\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2\rangle - \langle 2(\hat{\mathbf{x}}^T\hat{\mathbf{x}})\rangle+\langle(\hat{\mathbf{x}})^2\rangle = \langle(\mathbf{x})^2\rangle - \langle(\hat{\mathbf{x}})^2\rangle$$

und mit (1) folgt

$$\langle \hat{\mathbf{x}}^T\hat{\mathbf{x}}\rangle = \left\langle\sum_{i=1}^{m} y_i^2\right\rangle = \left\langle\sum_{i=1}^{m} y_i^2 + \sum_{i=m+1}^{n} a_i^2\right\rangle = \left\langle\hat{\mathbf{y}}^2\right\rangle$$

so dass schließlich

$\langle (x - \hat{\mathbf{x}})^2 \rangle = \langle (x)^2 \rangle - \langle (\hat{\mathbf{y}})^2 \rangle$ folgt, q.e.d.

13.1.2 Aufgaben

1) *Trainieren Sie ein XOR-Netz aus* ***Fehler! Verweisquelle konnte nicht gefunden werden.*** *mit Backpropagation. Welches Problem ergibt sich ? Lösen Sie es mit sigmoidalen Funktionen. Was passiert, wenn Sie die Steilheit k der Fermi-Funktion erhöhen?*

2) *Lösen Sie die gleiche Aufgabe wie in 1) mittels der Zielfunktion der maximalen Klassifikationswahrscheinlichkeit aus Gl.****Fehler! Verweisquelle konnte nicht gefunden werden.****.*

Anhang A Information und Schichtenstruktur

Welche Leistungen sollte eine Schicht erfüllen, um "optimal" zu funktionieren ?
Eine Vermutung besteht darin, dass jede Schicht eine Umkodierung der eingehenden Information dahingehend vornimmt, dass alle Redundanz und unwesentliche Information (Variation der Formen usw.) eliminiert wird.

Dies wollen wir präzisieren, um Folgerungen daraus ziehen zu können. Dazu müssen wir zunächst klären: Was verstehen wir unter „Information“ und „Redundanz“?

A.1 Information und Entropie

In der Informatik kennen wir als Informationsspeicher beispielsweise den Hauptspeicher (RAM) oder den Massenspeicher (Magnetplatten). Die Informationsmenge wird dabei, wie wir wissen, in *Bit* gemessen. Ein Speicherplatz mit *n* Bits kann dabei eine von 2^n Zahlen enthalten. Je mehr Bits ein Speicherplatz hat, umso mehr Information lässt sich darin abspeichern. Also können wir intuitiv vom Verständnis her die Menge an Information I als proportional zu der Zahl der Bits definieren. Mit Hilfe des dualen Logarithmus ld(.) bedeutet dies

$$I \sim n = ld(2^n) = ld \text{ (Zahl der möglichen Daten)}$$

Wird nun jede der möglichen Daten bzw. Speicherzustände mit der gleichen Wahrscheinlichkeit $P = 1/2^n$ eingenommen, so ist die Information eines Zustands

$$I \sim ld(1/P) \qquad [Bit]$$

Wählen wir anstelle des dualen Logarithmus ld(.) den davon um einen konstanten Faktor verschiedenen natürlichen Logarithmus ln(.) = ln(2)/ld(.), so messen wir die Information nicht mehr in Bit, sondern in "natürlichen Einheiten". Wir können nun die Information $I(x^k)$ allgemein für ein konkretes Ereignis x^k aus der Menge aller Ereignisse X mit der Wahrscheinlichkeit $P(x^k)$ definieren:

$$\text{DEF} \quad I(X) := \ln(1/P(x^k)) = -\ln(P(x^k)) \qquad \textit{Information} \qquad (A.1)$$

Diese Definition von Information trägt der Beobachtung Rechnung, dass ein Ereignis, das dauernd auftritt, als „langweilig“ empfunden wird und deshalb wenig Information bringt; ein seltenes, unerwartetes Ereignis dagegen Überraschung auslöst und deshalb als

sehr informativ angesehen wird. Hier wird „öfters“ und „selten“ mit Wahrscheinlichkeiten gemessen und so die mittlere Information über Wahrscheinlichkeiten definiert. Es gibt aber auch andere Definitionsmöglichkeiten: Beispielsweise kann man „erwartet“ und „unerwartet“ als Kriterium ansehen und die Information nicht über die absoluten, sondern über die relativen, von einer Erwartungshaltung (Hintergrundkontext) abhängigen Wahrscheinlichkeiten definieren. In diesem Skript werden wir aber nur die erste Definition verwenden.
Die obige Definition zeigt eine Eigenschaft der Information, über die man sie ebenfalls definieren kann: Die Information zweier unabhängiger Ereignisse ist die Summe der Informationen der Einzelereignisse. Die Menge X besteht aus den möglichen Zahlenwerten, die jeweils mit unterschiedlicher Wahrscheinlichkeit $P(x^k)$ auftreten. Die Menge X wird auch als „Zufallsvariable“ (*random variable*) bezeichnet.

Die mittlere Information von *N* Ereignissen x^k mit gleicher Wahrscheinlichkeit $P(x^k) = 1/N$ lässt sich als Mittelwert errechnen:

$$H(X) = \frac{1}{N} \sum_k I(x^k) = \sum_k \frac{1}{N} I(x^k) \qquad \text{(A.2)}$$

Treten die x^k allerdings mit mit ungleichen Wahrscheinlichkeiten auf, so müssen wir stattdessen die tatsächlichen Wahrscheinlichkeiten $P(x^k)$ verwenden und anstelle der mittleren die *erwartete Information pro Ereignis*, die *Entropie*, berechnen:

$$\text{DEF} \quad H(X) := \sum_k P(x^k) I(x^k) = \langle I(x^k) \rangle_k \qquad \textit{Entropie} \qquad \text{(A.3)}$$

wobei wir uns für die Kurzschreibweise des Erwartungswert-Operators "⟨ ⟩" mit

$$\langle f(x_k) \rangle_k := \sum_k P(x_k)\, f(x_k) \qquad \text{(A.4)}$$

bedienen. Analog dazu lässt sich für den kontinuierlichen Fall der Wahrscheinlichkeitsdichte p(x) die differenzielle Entropie definieren

$$\text{DEF} \quad H(X) := \int_{-\infty}^{+\infty} p(x) \ln p(x)^{-1} dx \qquad \textit{differenzielle Entropie} \qquad \text{(A.5)}$$

Die Entropie H(X) einer Zufallsvariablen („Nachrichtenquelle“) X entspricht also der erwarteten Information pro Zeichen der Quelle. Damit haben wir für die Musterverarbeitung in Schichten neuronaler Netze ein quantitatives Maß eingeführt, das von dem bekannten Informationstheoretiker Claude E. Shannon 1949 zur Charakterisierung der Übertragungseigenschaften von Nachrichtenverbindungen eingeführt wurde [SHA49]. Mit diesen Begriffen versuchen wir nun, die Frage genauer zu beantworten, wie eine optimale Schicht auszusehen hat. Weiteres über die Anwendung des Informationsbegriffs ist beispielsweise in dem populären Buch von Heise [HEISE83] enthalten.

A.2 Bedingungen optimaler Informationstransformation

Mit dem oben eingeführten Begriff der Information können wir nun präzisieren, was wir unter "optimaler Informationsverarbeitung" einer neuronalen Schicht verstehen:
Eine neuronale Einheit (z.B. eine Schicht) soll *informationsoptimal* genannt werden, wenn ihre Architektur und ihre Gewichte (ihre Funktionsparameter) derart gewählt werden, dass sie die Verarbeitung der Eingangssignale *(Eingabemuster)* zu den Ausgabesignalen *(Ausgabemustern)* gemäß einem Informations-Gütekriterium optimal durchführt.

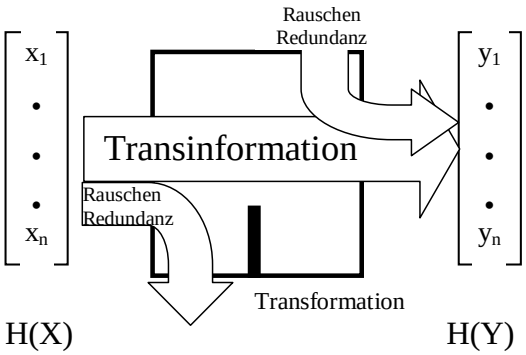


Abb. A.1 *Informationsfluss in einer Schicht*

In der Abb. A.1 ist diese Situation veranschaulicht. Sie entspricht dem Problem, das von Shannon 1949 [SHA49] untersucht wurde. Er betrachtete die Umstände, unter denen die Information eines *Senders* trotz eines gestörten Übertragungsweges *(Kanal)* besonders gut beim *Empfänger* ankommen. Die Maximierung der übertragenen Information bezeichnete er als *Maximierung der Übertragungsrate* eines Kanals (*Kanalkapazität*).

Dies wurde als ein Gütekriterium für informationsoptimale Schichten neuronaler Netze zuerst von Linsker [LINS86], [LINS88] (*infomax* Prinzip) vorgeschlagen und von ihm als fundamentales biologisches Funktionsprinzip bewertet. Nach H. Haken handelt es sich bei der Informationsoptimierung sogar um ein allgemeines, grundlegendes Funktionsprinzip der Natur [HAK88].

Bevor wir weitere Gütekriterien kennen lernen, betrachten wir dieses Kriterium etwas näher. Fassen wir zunächst alle Eingabesignale x_i der Einzelfasern, beispielsweise am Sehnerv, zu einem Tupel oder Vektor $\mathbf{x} = (x_1,..,x_n)$ und die Ausgangssignale y_j der informationsverarbeitenden Schicht zu $\mathbf{y} = (y_1,..,y_m)$ zusammen. Dann ist die Entropie H(X,Y) der beiden Verbund-Zufallsvariablen X = {**x**} und Y = {**y**}, also der Gesamtmenge {**x**,**y**} aller Ereignisse, *nicht* gleich der Summe H(X) + H(Y) der Einzelentropien. Dies resultiert daraus, dass die Ereignisse **x** der Eingabe *(Sender)* und der beobachteten Ausgabe **y** *(Empfänger)* im Allgemeinen nicht unabhängig voneinander sind. Wären sie

es, so würde keine Information von einem Sender auf einen Empfänger übertragen werden, sondern alle Ereignisse $\mathbf{y}^k$ werden unabhängig von $\mathbf{x}^k$ durch interne Mechanismen (Störsignale, Rauschen!) des Kanals produziert. Stattdessen müssen wir die Austauschinformation (*mutual information*) zwischen Sender und Empfänger, die *Transinformation* H(X;Y) berücksichtigen. Sie ist die Information, die X und Y gemeinsam haben, also die Information, die von X auf Y übertragen wurde. Es gilt nach Shannon [SHA49]

$$I(X;Y) := H(X) + H(Y) - H(X,Y) \quad \textit{Transinformation} \tag{A.6}$$

mit

$$I(X,Y) = -\sum_i \sum_k P(\mathbf{y}^k,\mathbf{x}^i) \ln\left(P(\mathbf{y}^k,\mathbf{x}^i)\right) \tag{A.7}$$

was durch die Verbundwahrscheinlichkeiten abhängiger Ereignisse bedingt ist. Bei der Notation werden im Argument der Transinformation H(X;Y) die Variablen mit einem Semikolon anstelle eines Kommas getrennt, so dass die Entropie H(X,Y) und die Transinformation H(X;Y) auch ohne Subskript unterschieden werden können.

Die Transinformation ist also die Differenz zwischen den Entropien der Eingabe und der Ausgabe. Das Maximum der Transinformation pro Zeiteinheit durch einen Kanal

$$\begin{aligned} I(X;Y) &= H(X) + H(Y) - H(X,Y) \\ &= \left\langle -\ln p(x) \right\rangle_x + \left\langle -\ln p(y) \right\rangle_y - \left\langle -\ln p(x,y) \right\rangle_{xy} \\ &= \left\langle -\ln p(x) \right\rangle_{xy} + \left\langle -\ln p(y) \right\rangle_{xy} - \left\langle -\ln p(x,y) \right\rangle_{xy} \\ &= \left\langle \ln p(x,y) / (p(y)p(x)) \right\rangle_{xy} \end{aligned} \tag{A.8}$$

$$\overset{!}{=} \max \qquad \textit{maximale Transinformation}$$

wird auch als *Kanalkapazität* bezeichnet, wobei das Maximum bei allen möglichen Eingabeverteilungsdichten p(X) gesucht wird.

Es existieren auch noch andere Gütekriterien für die Funktion eines informationsverarbeitenden Systems. Beispielsweise ist die Informationsdivergenz (*I-divergence, cross entropy*, *Kullback-Leibler Informationsmaß*) zwischen der mit der vermuteten Wahrscheinlichkeitsdichte q(X) erwarteten, subjektiven Information, gemittelt mit der objektiv beobachtbaren Wahrscheinlichkeitsdichte p(x)

$$H_{sub}(X) = \left\langle -\ln q(x) \right\rangle_x \qquad \textit{subjektive Information} \tag{A.9}$$

und der objektiven, erwarteten Information H, vgl. [KUL59],[PFAF72]

$$D(P||Q) = H_{sub}(X) - H_{obj}(X) \qquad \textit{Informationsdivergenz} \tag{A.10}$$

$$= \left\langle -\ln q(x) \right\rangle_x + \left\langle \ln p(x) \right\rangle_x = \left\langle \ln p(x)/q(x) \right\rangle_x \qquad \text{(A.11)}$$

Minimiert man die Informationsdivergenz, so passt man die subjektive Beurteilung von beobachten Daten x an eine objektive Realität an. Es lässt sich leicht zeigen, dass D(P,Q) immer ≥ 0 ist und sich deshalb durchaus als Leistungskriterium benutzen lässt. Da D(P,Q) = D(Q,P) gilt, ist es allerdings nicht als Abstandsmaß brauchbar.

Die Informationsdivergenz zwischen dem vermuteten Zusammenhang von Eingabe X und Ausgabe Y und dem tatsächlichen, beobachteten Zusammenhang ist $D(P(X,Y)||Q(X,Y)) = \langle \ln p(x,y)/q(x,y) \rangle_{x,y}$. Für die subjektive Einschätzung, dass X und Y unabhängig seien, geht mit $q(x,y) = p(x)\cdot p(y)$ die Informationsdivergenz formal in die Transinformation (A.8) über. Wie man leicht überlegen kann, entspricht dies auch inhaltlich der Formel (A.7).

Ein weiteres interessantes informationstechnisches Kriterium ist diejenige Information über ein Merkmal α, die ein Muster x nach dem Durchgang durch eine Schicht darin einbüßt

$$\Delta H_\alpha(x;y) = I_{trans}(x;\alpha) - I_{trans}(y;\alpha) \overset{!}{=} \min \qquad \textit{min. Informationsverlust} \qquad \text{(A.12)}$$

Da $I_{trans}(x;\alpha)$ festliegt, entspricht dies einer Maximierung von $I_{trans}(y;\alpha)$ durch Optimierung der Schichtparameter, was meist dem Kriterium der maximalen Transinformation entspricht [PLUM91]. Im weiteren wollen wir deshalb mit dem hier eingeführten, durchaus brauchbaren Kriterium der maximalen Transinformation arbeiten.

A.3 Effiziente Kodierung paralleler Signale

Bei der Informationsverarbeitung von Signalen **x** durch eine neuronale Schicht lassen sich mit dem Informationskriterium einige allgemeine Bedingungen ableiten, die wichtige Gütekriterien einer Schicht darstellen. Betrachten wir dazu die Abb. A.2 für die parallele Bearbeitung der Signale.

Mathematisch gesehen lassen sich die Signale $x_1,..,x_n$ bzw. $y_1,..,y_n$ als reellwertige, zufällige (*stochastische*) Variable ansehen, die sich zu einer Verbundvarianblen X bzw. Y, einem Zufallsvektor, zusammenfassen lassen. Die Transformation der mittleren Information ist in diesem System durch die Auftrittswahrscheinlichkeit der Variablen bestimmt. Bereits Shannon führte in [SHA49] aus, dass eine Menge von diskreten Ereignissen nach einer linearen Transformation auf eine andere, vollständige Basis weiterhin die gleiche Entropie enthält: H(x) = H(y). Die Ereignisse sind immer noch unterscheidbar und behalten ihre Auftrittswahrscheinlichkeit P_i bei.

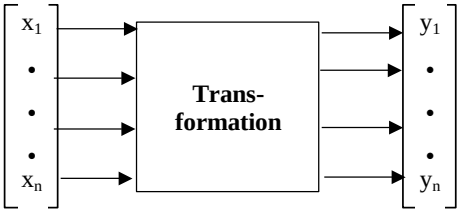


Abb. A.2 *Transformation paralleler Signale in einer Schicht*

Anders ist dagegen der Fall, wenn eine kontinuierliche Wahrscheinlichkeitsverteilung vorhanden ist.

A.3.1 Transformation von Wahrscheinlichkeitsdichten

Eine der einfachsten Arten von Zufallsvariablen sind Variablen mit einer konstanten, uniformen Verteilung. In Abb. A.3 ist die **Wahrscheinlichkeitsdichte** p(x) (*probability density function pdf*) gezeigt. Sie ist hier von x = 0 bis x = 1 konstant eins und geht ab x = 1 auf null zurück.

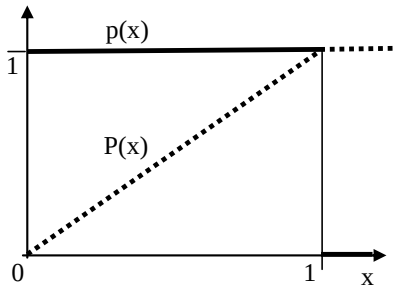


Abb. A.3 Die Wahrscheinlichkeitsdichte einer uniformen Verteilung

Formal kann man sie definieren als

$$p(x) = \begin{cases} 1 & \text{für } 0 \le x \le 1 \\ 0 & \text{sonst} \end{cases} \qquad \text{(A.13)}$$

wobei die Gesamtwahrscheinlichkeit $P(0 \le x \le 1)$ das Integral ist, die Fläche unter der Kurve p(x). Sie wird auch als „Wahrscheinlichkeitsmasse“ bezeichnet.

$$P = \int_0^1 p(x)dx = x \Big|_0^1 = 1 \qquad \text{(A.14)}$$

Alle Ereignisse geschehen wie gewünscht im Intervall von null bis eins. Die Wahrscheinlichkeit P(w), dass ein Ereignis (Beobachtung der Zufallsvariablen) den Wert $x \leq w$ hat, ist die Summe aller Wahrscheinlichkeiten in diesem Intervall, also

$$P(w) = \int_{-\infty}^{w} p(x)dx \qquad \textit{Verteilungsfunktion} \qquad \text{(A.15)}$$

was für unsere uniforme Verteilung bedeutet

$$P(w) = \int_0^w 1\,dx = x \Big|_0^w = w \qquad \text{(A.16)}$$

eine lineare Funktion in *w*, die ab x = 1 konstant bleibt. Sie ist in Abb. A.3 gestrichelt eingezeichnet und heißt **Verteilungsfunktion** der Zufallsvariablen X *(cumulative distribution function* cdf). Sie charakterisiert vollkommen die Verteilung und kann anstelle der Dichtefunktion angegeben werden. Für die uniforme Verteilung kürzen wir sie mit $U(\mu,\sigma)$ (hier: U(0,1)) ab.
Jede Verteilungsfunktion F(x) nach Gl.(A.15)

$$F(x) = \int_{x_0}^{x} p(t)dt \qquad \text{(A.17)}$$

verläuft vom Wert $F(x_0) = 0$ bis zum Wert $F(x_1) = 1$, und hat als Wertemenge von y = F(x) das Intervall [0,1] . Ziehen wir also mit der uniformen Verteilung U(0,1) zufällig einen Wert Y = P(y) der *cdf* aus dem Intervall [0,1], so entspricht dies einem Wert F(x) und, da F(x) monoton wachsend ist, auch eineindeutig einem Wert *x*. Es existiert also eine inverse Funktion $F^{-1}(Y) = x$, mit deren Hilfe wir aus dem uniform verteilten Wert *y* einen Wert *x* finden. In Abb. A.4 ist dies gezeigt.

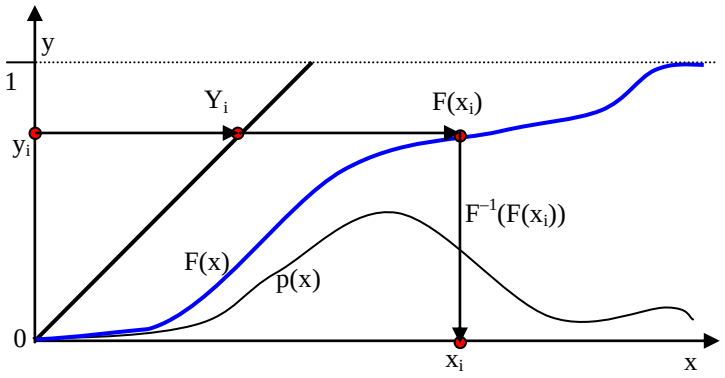


Abb. A.4 Transformation durch inverse Verteilung

Warum ist dieser so gefundene Wert *x* nach F(x) verteilt? Schließlich war ja *y* aus U(0,1) nur uniform verteilt. Der Beweis dafür ist kurz: Die Verteilungsfunktion von *x* ist

$$P(x \le x_i) = P(x \le F^{-1}(Y_i)) = P(F(x) \le F(F^{-1}(Y_i)))$$

$$= P(y \le Y_i) = Y_i = F(x_i) \qquad \text{(A.18)}$$

Inhaltlich bedeutet dies, dass eine Änderung ΔF der Wahrscheinlichkeitsmasse von F gerade einer Änderung ΔY der uniformen Verteilung entspricht. Wenn beide Änderungen gleich sind, so ist auch die Wahrscheinlichkeit von y in ΔY zu sein, genauso groß wie von x in ΔF zu sein; die „Wahrscheinlichkeitsmassen" sind gleich. Die Wahrscheinlichkeitsverteilung einer Variablen y := F(x) der Verteilungsfunktion F ist also immer uniform unabhängig von der Funktion F; die *pdf* ist konstant.

Damit haben wir nicht nur eine Aussage über die *cdf* einer Zufallsvariablen gemacht, sondern können auch umgekehrt eine Zufallsvariable x mit der *cdf* F(x) bzw. der *pdf* p(x) aus einer uniform verteilten Zufallsvariablen y generieren:

$$x = F^{-1}(y) \qquad \text{(A.19)}$$

Betrachten wir nun die Auswirkungen der Transformation auf die Information.

A.3.2 Transformation der Information

Mit der aus der Analysis bekannten Formel

$$\int\!\!\ldots\!\int p(x_1,\ldots,x_n)\, dx_1 \ldots dx_n = 1 = \int\!\!\ldots\!\int p(x_1(\mathbf{y}),\ldots,x_n(\mathbf{y}))\, \left|\det\left(\frac{\partial \mathbf{x}}{\partial \mathbf{y}}\right)\right| dy_1 \ldots dy_n \qquad \text{(A.20)}$$

die einer Variablentransformation der Variablen $x_1,..,x_n$ zu den Variablen $y_1,..,y_n$ in den Integralen entspricht, lässt sich eine neue Dichte

$$p(y_1,...,y_n) : = p(x_1(\mathbf{y}),...,x_n(\mathbf{y}))\ |\det\left(\frac{\partial \mathbf{x}}{\partial \mathbf{y}}\right)| \quad \text{oder} \quad p(\mathbf{y}) = p(\mathbf{x})\ |\det \mathbf{J}|^{-1} \qquad \text{(A.21)}$$

mit $$\mathbf{J} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_{n1}} \end{pmatrix} \quad \text{und} \quad |\det\left(\frac{\partial \mathbf{x}}{\partial \mathbf{y}}\right)| \cdot |\det\left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)| = 1$$

für den Übergang zwischen den *n* unabhängigen Variablen mithilfe der *Jacobi-Determinante* det **J** = det(∂**y**/∂**x**) (Determinante der Matrix aus den Ableitungen der Funktion) definieren. Für die erwartete Information gilt damit

$$H(\mathbf{Y}) = -\int_{-\infty}^{+\infty} p(\mathbf{y}) \ln p(\mathbf{y})\, d\mathbf{y} \quad = -\int_{-\infty}^{+\infty} p(\mathbf{x}) \ln [p(\mathbf{x})/|\det \mathbf{J}|\,]\, d\mathbf{x}$$

$$= -\int_{-\infty}^{+\infty} p(\mathbf{x}) \ln p(\mathbf{x})\, d\mathbf{x} \; + \int_{-\infty}^{+\infty} p(\mathbf{x}) \ln |\det \mathbf{J}| d\mathbf{x}$$

$$= H(\mathbf{x}) \; + \int_{-\infty}^{+\infty} p(\mathbf{x}) \ln |\det \mathbf{J}| d\mathbf{x} \qquad \text{(A.22)}$$

Im linearen Fall mit **y** = **W**x ist der Rang der Matrix rang(**W**) = n und die Jacobi-Determinate ist det(**W**), so dass sich die Wahrscheinlichkeitsdichte p(**X**) mit dem skalaren Faktor det(∂**x**/∂**y**) = det(∂**y**/∂**x**)$^{-1}$ = 1/det(**W**) des Raumvolumens zu p(**Y**) transformiert. Dies bedeutet z.B. für eine Gauß-verteilte Variable **x**, die linear transformiert wird, dass auch die Variable **y** = **W**x eine Gauß-verteilte Zufallsvariable ist.
Für die Entropie gilt im linearen Fall

$$H(\mathbf{Y}) = H(\mathbf{X}) + \log |\det(\mathbf{W})| \qquad \text{(A.23)}$$

Bei einer *maßerhaltenden Transformation* (Drehung etc) mit det(**W**) = 1 bleibt auch die Entropie erhalten.

Wann enthalten die parallelen Kanäle die maximal mögliche Information? Sei H_{max} die maximal über alle Kanäle gleichzeitig (bzw. pro Zeiteinheit) übertragbare Information; beispielsweise bei *m* Kanälen und 2 diskreten Zuständen pro Kanal ist dies mit 2^m Zuständen gerade H_{max}= ld(2^m) = m Bit. Dann lässt sich die Redundanz in den Kanälen definieren als Differenz zwischen maximaler und tatsächlicher mittlerer Information

$$H_{red} = H_{max} - H(y_1,...,y_m) \qquad \textit{Redundanz} \qquad \text{(A.24)}$$

Die maximal mögliche mittlere Information bei kontinuierlichen Kanalvariablen y_i festzustellen ist gar nicht so einfach. Eine Möglichkeit besteht darin, alle Kanäle unabhängig voneinander zu beobachten und dann für jeden Kanal die mittlere Information auszu-

rechnen. Ist die Gesamtinformation $H(y_1,...,y_m)$ kleiner als die Summe der Einzelinformationen $\Sigma_i H(y_i)$, so ist zweifelsohne Redundanz vorhanden. Eine effiziente Kodierung der Information in den Variablen $y_1,...,y_m$ dann gegeben, wenn ihre gemeinsame Information (Transinformation) möglichst klein ist. Mit der Verallgemeinerung von Gleichung (A.7) gilt

$$I(y_1;...;y_m) = H(y_1) + H(y_2) +...+ H(y_m) - H(y_1,...,y_m) \qquad (A.25)$$

Da bei allgemeinen, zufälligen Variablen gilt

$$p(y_1,...,y_m) = p(y_1)\, p(y_2|y_1) \ldots p(y_n|y_1,..,y_{m-1}) \qquad (A.26)$$

und man mit einigen Umformungen (Übungsaufgabe!) daraus erhält

$$H(y_1,...,y_m) = H(y_1) + H(y_2|y_1) +...+ H(y_n|y_1,..,y_{m-1}) \qquad (A.27)$$

so ist die Transinformation dann minimal, wenn

$$H(y_i) = H(y_i|y_1,..,y_{i-1}) \quad \text{bzw. } p(y_i) = p(y_i|y_1,..,y_{i-1}) \qquad (A.28)$$

gilt, also *alle Variablen stochastisch unabhängig voneinander* sind. Ein wichtiges Ziel einer Transformation $\mathbf{x}\rightarrow\mathbf{y}$ soll also sein, die Unabhängigkeit der $\{x_i\}$ zu erhalten bzw. bei den y_i herzustellen.

A.4 Optimale Informationsübertragung einer Schicht

Angenommen, wir wollen Signale durch eine Schicht von formalen Neuronen übertragen. Was ist die beste nicht-lineare Ausgangsfunktion S(z), die wir uns vorstellen können? Angenommen, wir wenden das Kriterium der Transinformation für ein einzelnes Neuron an. Die Information zwischen der Eingabe X und der Ausgabe Y, die Transinformation I(Y;X), ist nach Gl.(A.6)

$$I(X;Y) := H(X) + H(Y) - H(X,Y)$$

Dies lässt sich auch anders schreiben. Mit

$$H(Y,X) = H(Y|X) + H(X) \qquad \text{da } p(Y,X) = p(Y|X)p(X) \qquad (A.29)$$

erhalten wir als andere Formulierung

$$I(Y;X) = H(Y) - H(Y|X) \qquad \textit{Transinformation} \qquad (A.30)$$

Ist diese Abbildung auch noch umkehrbar, so ist jedem diskreten Ereignis $\mathbf{x}_i$ genau ein $\mathbf{y}_i = g(\mathbf{x}_i)$ zugeordnet, das immer zusammen mit $\mathbf{x}_i$ auftritt. Es ist also $P(\mathbf{y}_i|\mathbf{x}_i) = 1$ bei allen Ereignissen und damit mit

$$H(Y|X) = -\sum_i P(\mathbf{y}_i|\mathbf{x}_i)P(\mathbf{x}_i) \ln P(\mathbf{y}_i|\mathbf{x}_i)$$

ist $H(Y|X) = 0$. Für kontinuierliche Verteilungen lässt sich $H(Y|X)$ durch ein additives Rauschen der Dichte $p(y\text{-}S(x))$ modellieren [NaPa94], das unabhängig von den Gewich-

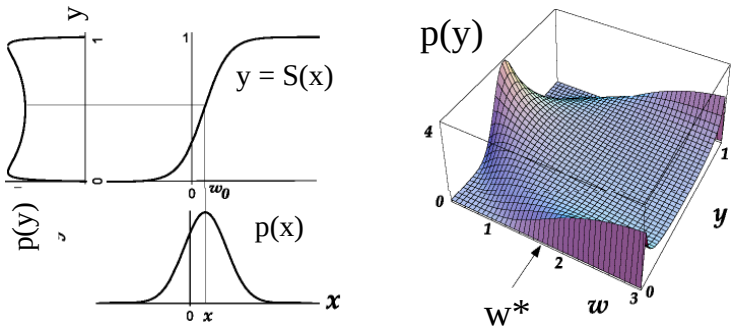
ten und der Ausgabefunktion ist und deshalb eine konstante Information besitzt. Die Transinformation wird also genau dann maximiert, wenn H(Y) maximal wird.
Mit Gl.(A.5) und Gl. (A.21) ist

$$H(Y) = -\int_{-\infty}^{+\infty} p(y)\ln p(y)\,dy \;=\; -\int_{-\infty}^{+\infty} p(x)\ln\frac{p(x)}{\partial y/\partial x}\,dx$$

und mit $\partial y/\partial x = S'(x)$ erhalten wir

$$H(Y) = -\int_{-\infty}^{+\infty} p(x)\ln\frac{p(x)}{S'(x)}\,dx \;= \left\langle -\ln\frac{p(x)}{S'(x)}\right\rangle_x \qquad (A.31)$$

Damit nimmt die Transinformation die Form der negativen Informationsdivergenz von Gl. (A.11) an: Sie erreicht genau dann ein Maximum, wenn ihr negativer Wert, die Informationsdivergenz, minimal wird, wenn also die Dichtefunktion p(x) mit der Ableitung S'(x) der Ausgabefunktion übereinstimmt.



(a) Transformation der pdf bei festem *w* **(b)** pdfs bei verschiedenem $S_F(w,x)$

Abb. A.5 *Transformation von Signalen in einem nichtlinearen Neuron (nach [Bell95])*

In der Abb. A.5 (a) links ist für eine Normalverteilung $p(x) = N(w_0,\sigma)$ die Dichtefunktion aufgetragen. Sie wird durch S(x) zu einer zweiten Dichtefunktion p(y) transformiert, die um 90 Grad gekippt dargestellt ist. Durch die sigmoidale Ausgabefunktion, die durch 0 und 1 in ihren minimalen und maximalen Werten begrenzt ist, werden die ursprünglichen x aus dem unendlichen Intervall auf das [0,1]-Intervall abgebildet. Verwendet man beispielsweise eine Fermi-Funktion mit $S_F(w,x) = 1/(1+\exp(-wx-w_0))$, so lässt sich durch Veränderung von w_0 der Mittelwert der Verteilung auf die Mitte von S(.) einstellen, siehe das Diagramm in Abb. A.5 (b). Verändern wir w, so wirkt sich dies auf die

Steilheit von S(.) aus. Ein zu geringer Anstieg von S(.) bedeutet eine eingipflige Kurve für p(y), ein zu steiler Anstieg eine zweigipflige Verteilung. Nur dann, wenn bei einer „richtigen“ Steigung die Ereignisse x optimal getrennt und nicht zu wenig oder zuviel aufgelöst werden, ist das Maximum der Transinformation erreicht. Es ist durch einen uniforme Verteilung gegeben: die Verteilungsdichte, die theoretisch in einem Intervall die maximale Information enthält. Jeder der beobachteten Eingabedichten p(x) entspricht also eine optimale Ausgabefunktion S(x) mit der Proportion $p(x) = S'(x)$.

A.5 Gestörte Signale

Oftmals sind die Kanäle durch thermisches Rauschen, elektromagnetische Koppelungseffekte zwischen den Einzelkanälen (Übersprechen) und andere Störungen, wie sie beispielsweise im biologischen Nervengewebe existieren, nur eingeschränkt nutzbar.

A.5.1 Klassifizierung

Eine Korrekturmöglichkeit besteht darin, zusätzlich zum Signalmuster $\mathbf{x}^i$ auch allen (nicht zu großen) Abweichungen von $\mathbf{x}^i$ den selben Wert $\mathbf{y}^k$ zuzuweisen. Wie muß dieser Klassifizierungsprozeß gestaltet werden, um die übertragene Information zu maximieren? Der Erwartungswert der Information der Ausgabe $\mathbf{y}^k$ über alle Eingaben, falls die jeweilige Eingabe $\mathbf{x}^i$ wohlbekannt ist, entspricht dem unbekannten, nicht nutzbaren Anteil der übertragenen Information (z.B. Störsignale) und wird als *bedingte Entropie* bezeichnet:

$$H(Y|X) := \langle I(\mathbf{y}^k/\mathbf{x}^i)\rangle_{i,k} = -\sum_i \sum_k P(\mathbf{y}^k,\mathbf{x}^i)\ln[P(\mathbf{y}^k/\mathbf{x}^i)]$$

$$= -\sum_i \sum_k P(\mathbf{y}^k/\mathbf{x}^i)P(\mathbf{x}^i)\ln[P(\mathbf{y}^k/\mathbf{x}^i)] \qquad \text{(A.32)}$$

Damit lässt sich die Transinformation I(X;Y) schreiben als

$$I(X;Y) = H(X) + H(Y) - H(X,Y)$$

$$= H(X) - H(X|Y) = H(Y) - H(Y|X)$$

$$= -\sum_k P(\mathbf{y}^k)\ln[P(\mathbf{y}^k)] + \sum_i P(\mathbf{x}^i)\sum_k P(\mathbf{y}^k/\mathbf{x}^i)\ln[P(\mathbf{y}^k/\mathbf{x}^i)] \qquad \text{(A.33)}$$

Die erwartete Transinformation H_{trans} ist maximal, wenn

$$H(Y) \overset{!}{=} \max \qquad \text{(A.34)}$$

$$H(Y|X) \overset{!}{=} \min \tag{A.35}$$

Bestimmen wir zunächst die Parameter, mit denen die erste Forderung erfüllt werden kann. Angenommen, es sind keine weiteren Anforderungen an $P_k := P(\mathbf{y}^k)$ bekannt als die allgemeine Nebenbedingung für Wahrscheinlichkeiten $\Sigma_k P_k = 1$. Wie findet man das Maximum einer Funktion $H(P_1,..,P_M)$ unter einer Nebenbedingung $g(P_1,..,P_M) := \Sigma_k P_k - 1 = 0$? Am bekanntesten ist die Methode der Lagrangschen Parameter. Dazu bildet man die Hilfsfunktion

$$L(P_1,..,P_M,\mu) := H(P_1,..,P_M) + \mu g(P_1,..,P_M) \tag{A.36}$$

und bestimmt die Ableitungen nach allen Parametern und setzt sie null. Lagrange fand heraus, dass dies die notwendigen Bedingungen für ein multidimensionales Extremum $\mathbf{P}^*=(P_1^*,..,P_M^*)$ der Funktion H sind. Die Ableitung $\partial L/\partial\mu = 0$ gibt uns dabei die gewünschte Nebenbedingung g(.)=0 auch für alle Extrememalpunkte $\mathbf{P}^*$. Die Entscheidung, ob es sich um ein Minimum, Maximum oder Wendepunkt handelt muß allerdings auf anderem Wege durch Kenntnis der Problematik entschieden werden.

Die Ableitung nach P_k ergibt

$$\frac{\partial L}{\partial P_k} = \frac{-\partial}{\partial P_k}(P_k \ln P_k) + \mu = -\ln P_k - 1 + \mu = 0 \qquad \forall k \tag{A.37}$$

für alle Ereignisse $\mathbf{y}^k$. Also sind alle Wahrscheinlichkeiten gleich und die Nebenbedingung gibt uns mit $\Sigma P_k^* = M P_k^* = 1$ die Größe von P_k^* an.
Es ergibt sich also für das Maximum von H(y) bei der Nebenbedingung $\Sigma_k P(\mathbf{y}^k) = 1$, dass dies für *M* Ereignisse zutrifft wenn

$$P(\mathbf{y}^k) = P(\mathbf{y}^l) = 1/M \qquad \forall k, l \tag{A.38}$$

erfüllt ist. Identifiziert man die Ereignisse mit dem Auftreten einer Klasse (z.B. $\mathbf{y}^k$ ist ein Klassenprototyp), so müssen bei optimaler Verarbeitung alle Klassen mit gleicher Wahrscheinlichkeit auftreten. Sind dagegen noch andere, systematische (Neben) Bedingungen für $P(\mathbf{y}^k)$ bekannt, so ist (A.38) sicher nicht das Optimum.

Für die zweite Forderung (A.28) sind verschiedene Fälle denkbar. Für zwei mögliche Formen von $P(\mathbf{y}^k/\mathbf{x}^i)$ soll dies nun beispielhaft errechnet werden.

A.5.2 Feste Klassentrennung

Für die Forderung (A.28) wissen wir, dass $P(\mathbf{y}^k/\mathbf{x})$ bei verschiedenen k sehr ungleich sein muss, um ein Minimum zu erreichen. Im Fall $n < m$ ist also eine diskrete Zuordnung jedes $\mathbf{x}^k$ zu einem $\mathbf{y}^k$ optimal. Ist $n > m$, so wird die Optimalität beispielsweise durch eine Aufteilung des Eingaberaums $\Omega = \{\mathbf{x}\}$ in Teilmengen Ω_k erreicht, wobei es keine Rolle spielt, wie dies erreicht wurde. In jedem Fall wird ein Muster $\mathbf{x}$ nur zu einer Klasse Ω_k

(einem $\mathbf{y}^k$) deterministisch zugeordnet und damit der Einfluss des Rauschens minimisiert. Dann gilt für den Produktterm in Gl. (A.33)

$$\forall\, \mathbf{x} \in \Omega_k \quad P(\mathbf{y}^k/\mathbf{x})\, \ln[P(\mathbf{y}^k/\mathbf{x})] = 1 \cdot \ln[1] = 0$$

sonst

$$P(\mathbf{y}^k/\mathbf{x})\, \ln[P(\mathbf{y}^k/\mathbf{x})] = \lim_{P\to 0} P \cdot \ln[P] =^{1)} \lim_{P\to 0} \frac{(\ln[P])'}{(1/P)'} = \lim_{P\to 0} -P = 0 \qquad \text{(A.39)}$$

und damit

$$H(y|x) = 0 \qquad \text{(A.40)}$$

1) l'Hopital-Regel

bei nicht-überlappenden Klassen. Für eine maximale Informationstransmission ist also die Bedingung (A.34) auch hinreichend.

Damit haben wir Bedingungen für die Eingabemuster und Ausgabemuster gewonnen, mit denen wir die Leistungsfähigkeit der Algorithmen neuronaler Schichten an unserem Maßstab der "Optimalität" messen können.

A.5.3 Gestörte lineare Schichten

Als zweites Beispiel wollen wir die oben entwickelten Kriterien auf die vorher eingeführten linearen Schichten anwenden.

Betrachten wir nun eine Variable $\mathbf{x}$, die in der linearen Schicht "verrauscht" wird. Die Ausgabe ist dann

$$\mathbf{y} = \mathbf{W}\,\mathbf{x} + \tilde{\mathbf{y}} \qquad \text{oder } \tilde{\mathbf{y}} = \mathbf{y} - \mathbf{W}\,\mathbf{x} \qquad \text{(A.41)}$$

mit der Zufallsvariablen $\tilde{\mathbf{y}}$. Wie sollte man nun $\mathbf{W}$ wählen, um die Transinformation besonders groß zu machen ? Die Transinformation ist

$$I(Y;X) = H(\mathbf{y}) - H(\mathbf{y}|\mathbf{x}) = H(\mathbf{W}\,\mathbf{x} + \tilde{\mathbf{y}}) - H(\mathbf{W}\,\mathbf{x} + \tilde{\mathbf{y}}\,|\mathbf{x})$$

$$= H(\mathbf{W}\,\mathbf{x}) + H(\tilde{\mathbf{y}}) - H(\mathbf{W}\,\mathbf{x}|\mathbf{x}) - H(\tilde{\mathbf{y}}\,|\mathbf{x})$$

$$= H(\mathbf{x}) + \ln \det(\mathbf{W}) + H(\tilde{\mathbf{y}}) - H(\mathbf{x}|\mathbf{x}) - \ln \det(\mathbf{W}) - H(\tilde{\mathbf{y}}\,|\mathbf{x})$$

$$= H(\mathbf{x}) + H(\tilde{\mathbf{y}}) - H(\tilde{\mathbf{y}}\,|\mathbf{x})$$

und wird bei einer linearen Transformation mit mit vom Rauschen $\tilde{\mathbf{y}}$ unabhängigen Eingaben $\mathbf{x}$ genau dann *maximal* bezüglich der Parameter $\mathbf{W}$, wenn $H(\tilde{\mathbf{y}}\,|\mathbf{x})$ *minimal* wird. Dies bedeutet für Gaußverteilte, dekorrelierte Zufallsvariable $\tilde{\mathbf{y}}$

$$p(\tilde{\mathbf{y}}) = p(\tilde{\mathbf{y}}\,|\mathbf{x}) = A \exp(-\tilde{\mathbf{y}}^T\,\tilde{\mathbf{y}}/2\sigma_y^2)$$

und somit

$$H(\tilde{\mathbf{y}}\,|\mathbf{x}) = \langle -\ln p(\tilde{\mathbf{y}}\,|\mathbf{x})\rangle = \langle -\ln A + (\tilde{\mathbf{y}}^T\tilde{\mathbf{y}}\,/2\sigma_y^2)\rangle$$

$$= \langle -\ln A\rangle + \langle(\mathbf{y}-\mathbf{W}\,\mathbf{x})^2/2\sigma_y^2\rangle \overset{!}{=} \min$$

was genau dann minimal wird, wenn der erwartete quadratische Fehler

$$\langle(\mathbf{y}-\mathbf{W}\,\mathbf{x})^2\rangle \overset{!}{=} \min \qquad \text{(A.42)}$$

minimal wird. Bei Gaußverteilungen und linearen Schichten sind die beiden Leistungskriterien "kleinste erwartete quadratische Fehler (LMSE)" und "größte Transinformation" gleichwertig und führen zu gleichen Ergebnissen.

Anders ist dies dagegen, wenn die Musterdaten *nicht* Gauß-verteilt sind, wie dies beispielsweise häufig bei Klassifikationsdaten der Fall ist. Hier erbringt das Kriterium der maximalen Information die besseren Ergebnisse, s. Kapitel 2.

Aufgaben

1) Zeigen Sie: Die Information zweier unabhängiger Zufallsvariablen ist die Summe der Informationen der Einzelereignisse.

2) Zeigen Sie die Beziehung: Die Entropie zweier abhängiger Ereignisse ist

$$H(x,y) = H(x) + H_x(y)$$

Gehen Sie dazu auf die Definition der Entropie zurück und bestimmen Sie die Verbundwahrscheinlichkeit zweier Zufallsvariablen.

3) Warum ist die Information zweier Zufallsvariablen maximal, wenn sie unabhängig sind? Warum ist dann die Transinformation null?

4) Man zeige, dass aus der Unabhängigkeit zweier Zufallsvariablen folgt, dass sie auch dekorreliert sind:

$$P(x_1,x_2)=P(x_1)P(x_2) \;\Rightarrow\; \langle(x_1-\langle x_1\rangle)(x_2-\langle x_2\rangle)\rangle = 0$$

5) Eine Dekorrelation ist damit eine notwendige Voraussetzung für Unabhängigkeit und damit für maximalen Informationsgehalt von Variablen. Ist sie auch hinreichend?

B Minimierung des quadratischen Fehlers

Angenommen, wir wollen mit Hilfe einer linearen Abbildung N Eingaben $\mathbf{x}^1 ... \mathbf{x}^N$ auf N Ausgaben $\mathbf{y}^1 ... \mathbf{y}^N$ abbilden

$$\mathbf{y}^k = \mathbf{W}\mathbf{x}^k \qquad \text{bzw.} \quad y_i^k = \mathbf{w}_i^T \mathbf{x}^k \tag{B.1}$$

Wie muss dafür die Matrix $\mathbf{W}$ aussehen?
Bilden wir aus den Spaltenvektoren der Eingaben und Ausgaben die Matrizen

$$\mathbf{X} = (\mathbf{x}^1,...,\mathbf{x}^N) \quad \text{und} \quad \mathbf{Y} = (\mathbf{y}^1,...,\mathbf{y}^N) \tag{B.2}$$

so muss $\mathbf{W}$ die Gleichung erfüllen

$$\mathbf{Y} = \mathbf{W}\,\mathbf{X} \tag{B.3}$$

Bei N linear unabhängigen Spaltenvektoren $(\mathbf{y}^1,...,\mathbf{y}^N)$ und $\dim(\mathbf{x}) = N = \dim(\mathbf{y})$ ist (s. [KOH84])

$$\mathbf{W} = \mathbf{Y}\mathbf{X}^{-1} \text{ mit } \mathbf{X}^{-1} = \mathbf{X}^T (\mathbf{X}\mathbf{X}^T)^{-1} \tag{B.4}$$

Sind die Spaltenvektoren dagegen nicht linear unabhängig oder nur $\dim(\mathbf{x}) = m < N$ Dimensionen vorhanden, also $\text{rang}(\mathbf{Y}) < N$, so ist auch $\text{rang}(\mathbf{W}) < N$ und wir bilden die Muster $\{\mathbf{x}\}$ auf einen Unterraum mit geringerer Dimension $m < N$ ab. Das Muster $\mathbf{x}$ ist nur mit m linear unabhängigen Basisvektoren $\mathbf{b}_i = \mathbf{w}_i$ dargestellt und damit komprimiert gegenüber der vollständigen, fehlerfreien Entwicklung mit N linear unabhängigen Basisvektoren: wir führen eine Datenkompression durch. Die Rückabbildung lässt sich damit nur mit einem Fehler durchführen. Man kann dann nur noch diejenige Matrix $\mathbf{X}^+$ suchen, die dabei den quadratischen Fehler minimiert, wobei die Gewichtsmatrix mit

$$\mathbf{W} = \mathbf{Y}\mathbf{X}^+ \tag{B.5}$$

ermittelt wird. Die Matrix $\mathbf{X}^+$ wird dabei als Moore-Penrose *Pseudo-Inverse* bezeichnet [ALB72]. Damit schießen wir den Fall von N Eingaben der Dimension m ab.

Gehen wir nun auf den "normalen" Fall von sehr vielen Eingaben über. Angenommen, wir möchten den Fehler der Rekonstruktion $\hat{\mathbf{x}}$ zum originalen $\mathbf{x}$ nicht über einige Eingaben, sondern für die Gesamtheit aller zu erwartenden Eingaben im Erwartungswert minimieren. Unsere Zielfunktion des quadratischen Fehlers ist dann

$$\min_{\mathbf{w}} R(\mathbf{W}) = \min \langle (\mathbf{x} - \hat{\mathbf{x}})^2 \rangle \quad \textit{least mean squared error} \quad \text{(LMSE)} \tag{B.6}$$

Das ursprüngliche Muster $\mathbf{x}$ lässt sich aus $\mathbf{y}$ nur dann wieder fehlerfrei rekonstruieren, wenn die Zahl der linear unabhängigen Basisvektoren in beiden Räumen gleich groß ist. Mit $n = \dim(\mathbf{x}) = \dim(\mathbf{y})$ ist dann

$$\mathbf{x} = \mathbf{W}^{-1}\mathbf{y} \quad \text{oder} \quad \mathbf{x} = \sum_{i=1}^{n} y_i \mathbf{b}_i \quad \text{mit} \quad (\mathbf{b}_i^T) = \mathbf{B} = \mathbf{W}^{-1} \tag{B.7}$$

Bei orthonormalen Basisvektoren $\mathbf{w}_i$

mit
$$\mathbf{w}_i^T \mathbf{w}_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{B.8}$$

ist die Matrix $\mathbf{W}$ orthogonal und es gilt

$$\mathbf{B} = \mathbf{W}^{-1} = \mathbf{W}^T. \tag{B.9}$$

Damit lassen sich das Muster $\mathbf{x}$ und seine Rekonstruktion $\hat{\mathbf{x}}$ mit Hilfe der Basisvektoren $\mathbf{w}_i$ (Spaltenvektoren) ausdrücken

$$\mathbf{x} = \sum_{i=1}^{n} y_i \mathbf{w}_i = \sum_{i=1}^{m} y_i \mathbf{w}_i + \sum_{i=m+1}^{n} y_i \mathbf{w}_i \quad \text{und} \quad \hat{\mathbf{x}} = \sum_{i=1}^{m} y_i \mathbf{w}_i + \sum_{i=m+1}^{n} c_i \mathbf{w}_i \tag{B.10}$$

wobei das rekonstruierte Signal $\hat{\mathbf{x}}$ mit Hilfe der gespeicherten bzw. übermittelten Koeffizienten y_i und der konstanten, für alle Muster gleichen Koeffizienten c_i beschrieben wird.

Der erwartete Fehler der Rekonstruktion hängt dabei nicht nur von der Zahl m der Koeffizienten ab, sondern auch von der Wahl der Basis $\{\mathbf{w}_i\}$ und der Konstanten $\{c_i\}$. Beide wollen wir so bestimmen, dass unsere Zielfunktion in Gl. (B.6) möglichst klein wird. Beginnen wir mit den Konstanten. Das Minimum bezüglich der Wahl der Konstanten ist

$$\begin{aligned}
&\min_c R(\mathbf{c}) = \min_c \langle (\mathbf{x} - \hat{\mathbf{x}})^2 \rangle \\
&= \min_c \Big\langle \Big(\sum_{i=1}^{m} y_i \mathbf{w}_i + \sum_{i=m+1}^{n} y_i \mathbf{w}_i - \sum_{i=1}^{m} y_i \mathbf{w}_i - \sum_{i=m+1}^{n} c_i \mathbf{w}_i \Big)^2 \Big\rangle \\
&= \min_c \Big\langle \Big(\sum_{i=m+1}^{n} y_i \mathbf{w}_i - \sum_{i=m+1}^{n} c_i \mathbf{w}_i \Big)^2 \Big\rangle = \min_c \Big\langle \Big(\sum_{i=m+1}^{n} (y_i - c_i) \mathbf{w}_i \Big)^2 \Big\rangle \\
&= \min_c \Big\langle \sum_{i=m+1}^{n} \sum_{j=m+1}^{n} (y_i - c_i)(y_j - c_j) \mathbf{w}_i^T \mathbf{w}_j \Big\rangle = \min_c \Big\langle \sum_{i=m+1}^{n} (y_i - c_i)^2 \Big\rangle
\end{aligned} \tag{B.11}$$

bedingt durch die Orthogonalität Gl. (B.8) der Basis. Da die c_i nicht voneinander abhängen sollen und die Terme der Summe alle positiv sind, muss für ein Minimum jeder einzelne Term der Summe minimal werden. Dies bedeutet, das Minimum

$$\min_{c_i} \left\langle (y_i - c_i)^2 \right\rangle \tag{B.12}$$

zu finden. Das Minimum der quadratischen Funktion liegt vor, wenn die Ableitung null wird

$$\frac{\partial}{\partial c_i} \left\langle (y_i - c_i)^2 \right\rangle = -2\left\langle (y_i - c_i) \right\rangle = -2\left(\langle y_i \rangle - c_i \right) = 0$$

bzw. $\langle y_i \rangle = c_i$ oder $\mathbf{w}_i^T \langle \mathbf{x} \rangle = c_i$ (B.13)

gilt. Die besten Konstanten c_i sind also die Erwartungswerte der Koeffizienten y_i.

Wie sollten wir uns die Basisvektoren $\{\mathbf{w}_i\}$ wählen? Es ist

$$\min_{\mathbf{w}} R(\mathbf{w}) = \min_{\mathbf{w}} \left\langle (\mathbf{x} - \hat{\mathbf{x}})^2 \right\rangle$$

Damit entspricht diese Forderung analog der Anforderung Gl.(B.12) an jeden Basisvektor $\mathbf{w}_i$

$$\min_{\mathbf{w}_i} \left\langle (y_i - c_i)^2 \right\rangle = \min_{\mathbf{w}_i} \left\langle \left(y_i - \langle y_i \rangle \right)^2 \right\rangle = \min_{\mathbf{w}_i} \left\langle \left(\mathbf{w}_i^T \mathbf{x} - \mathbf{w}_i^T \langle \mathbf{x} \rangle \right)^2 \right\rangle$$

$$= \min_{\mathbf{w}_i} \left\langle \mathbf{w}_i^T \left(\mathbf{x} - \langle \mathbf{x} \rangle \right) \left(\mathbf{x} - \langle \mathbf{x} \rangle \right)^T \mathbf{w}_i \right\rangle = \min_{\mathbf{w}_i} \mathbf{w}_i^T \left\langle \left(\mathbf{x} - \langle \mathbf{x} \rangle \right) \left(\mathbf{x} - \langle \mathbf{x} \rangle \right)^T \right\rangle \mathbf{w}_i$$

so dass wir mit der Definition der Kovarianzmatrix der Eingabemuster

$$\mathbf{C}_{xx} = \left\langle \left(\mathbf{x} - \langle \mathbf{x} \rangle \right) \left(\mathbf{x} - \langle \mathbf{x} \rangle \right)^T \right\rangle \tag{B.14}$$

ein Minimum des erwarteten quadratischen Fehlers genau dann erreichen, wenn die Basisvektoren die Gleichung

$$\min_{\mathbf{w}_i} R(\mathbf{w}) = \min_{\mathbf{w}_i} \mathbf{w}_i^T \mathbf{C}_{xx} \mathbf{w}_i \qquad |\mathbf{w}_i| = 1 \tag{B.15}$$

minimieren. Dabei schließen wir die triviale Lösung $\mathbf{w}_i = 0$ aus, so dass wir die Lösung der Gleichung suchen unter der Nebenbedingung konstanter Länge der Basisvektoren. Die optimalen Basisvektoren zeichnen sich also durch ihre Richtungen aus, nicht durch ihre Länge.

Das Problem, den Extremwert einer Funktion unter Nebenbedingungen zu finden, wird durch die Methode der „Langrang'schen Parameter“ erreicht, siehe Anhang E.1. Dazu gehen wir folgendermaßen vor: Zuerst bilden wir eine Hilfsfunktion L, die aus der zu maximierenden Funktion und der mit einem Hilfsparameter μ gewichteten Nebenbedingung besteht, Die Nebenbedingung ist in einer Form aufgeschrieben, die null ergibt, hier: $\mathbf{w}^2 - 1 = 0$.

$$L(\mathbf{w}, \mu) = R(\mathbf{w}) + \mu (\mathbf{w}^2 - 1) \qquad \textit{Lagrange-Funktion} \tag{B.16}$$

Die für ein Extremum notwendigen Bedingungen sind die Ableitungen dieser Hilfsfunktion

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \quad \text{und} \quad \frac{\partial L}{\partial \mu} = 0 \tag{B.17}$$

In unserem Fall ist dies

$$\frac{\partial L}{\partial \mathbf{w}}(\mathbf{w},\mu) = \frac{\partial R(\mathbf{w})}{\partial \mathbf{w}} + \mu 2\mathbf{w} = \mathbf{C}_{xx}\mathbf{w} + \mu 2\mathbf{w} = 0$$

oder $\mathbf{C}_{xx}\mathbf{w} = \lambda\mathbf{w}$ mit $\lambda := -2\mu$ (B.18)

Dies ist eine *Eigenwertgleichung* für $\mathbf{C}_{xx}$; die Lösungen $\mathbf{e}_i$ sind die Eigenvektoren der Kovarianzmatrix $\mathbf{C}_{xx}$. Die Zielfunktion (B.6) wird bei dem Eigenvektor $\mathbf{e}_i$ mit dem größten Eigenwert maximal und beim Eigenvektor mit dem kleinsten Eigenwert minimal. In diesem Fall ist der Fehler mit Gl.(B.11)

$$R(\mathbf{e}_1,..,\mathbf{e}_n) = \left\langle \sum_{i=m+1}^{n} \left(y_i - \langle y_i \rangle\right)^2 \right\rangle$$

und mit Gl. (B.15) und Gl.(B.22)

$$= \sum_{i=m+1}^{n} \mathbf{e}_i^T \mathbf{C}_{xx} \mathbf{e}_i = \sum_{i=m+1}^{n} \lambda_i \tag{B.19}$$

Um diesen Fehler zu minimieren, ist es deshalb sinnvoll, für die vernachlässigten *(n-m)* Komponenten als Basisvektoren diejenigen Eigenvektoren zu verwenden, die die *kleinsten* Eigenwerte besitzen. Die tatsächlich zur linearen Transformation von **x** verwendeten *m* Basisvektoren $\mathbf{w}_i$ sollen also denjenigen Unterraum *(subspace)* aufspannen, der dem Raum der *m* Eigenvektoren mit den *größten* Eigenwerten entspricht.

Dabei können die neuen Basisvektoren

(a) sowohl selbst die Eigenvektoren darstellen $\mathbf{w}_i = \mathbf{e}_i$

(b) oder auch nur eine Linearkombination sein $\mathbf{w}_i = \sum_{k=1}^{n} a_{ik}\mathbf{e}_k$

Im ersten Fall (a) erfüllen die Koeffizienten y_i eine wichtige Voraussetzung für die Unabhängigkeit: sie sind *unkorreliert* bzw. *dekorreliert*

$$\langle (y_i - \langle y_i \rangle)(y_j - \langle y_j \rangle) \rangle = \mathbf{e}_i^T \left\langle \left(\mathbf{x} - \langle \mathbf{x} \rangle\right)\left(\mathbf{x} - \langle \mathbf{x} \rangle\right)^T \right\rangle \mathbf{e}_j$$

$$= \mathbf{e}_i^T \mathbf{C}_{xx} \mathbf{e}_j = \lambda_j \mathbf{e}_i^T \mathbf{e}_j = 0 \text{ für } i \neq j \tag{B.20}$$

Im zweiten Fall (b) lässt sich dies aber nicht allgemein folgern. Wenn die neuen Basisvektoren nur Linearkombinationen der Eigenvektoren sind, folgt umgekehrt aus einer Dekorrelation der Datenausgabe,

$$0 = \langle (y_i - \langle y_i \rangle)(y_j - \langle y_j \rangle) \rangle = \mathbf{w}_i^T \mathbf{C}_{xx} \mathbf{w}_j = \left(\sum_{k=1}^{n} a_{ik} \mathbf{e}_k \right)^T \mathbf{C}_{xx} \left(\sum_{s=1}^{n} a_{js} \mathbf{e}_s \right)$$

$$= \sum_k \sum_s a_{ik} \mathbf{e}_k^T a_{js} \lambda_j \mathbf{e}_s = \sum_k a_{ik} a_{jk} \lambda_k |\mathbf{e}_k|^2 = 0 \qquad \text{für } i \neq j \tag{B.21}$$

dass die neuen Basisvektoren nur im Raum der mit

$$|\mathbf{e}_k|^2 = \lambda_k^{-1} \tag{B.22}$$

normierten Eigenvektoren die Bedingung

$$\mathbf{w}_i^T \mathbf{w}_j = \left(\sum_{k=1}^{n} a_{ik} \mathbf{e}_k \right)^T \left(\sum_{s=1}^{n} a_{js} \mathbf{e}_s \right) = \sum_{k=1}^{n} a_{ik} a_{jk} = 0 \quad \text{für } i \neq j \tag{B.23}$$

erfüllen und damit orthogonal sind.
Die Proportion (B.22) ist sicher nicht für alle möglichen Basen $\{\mathbf{w}_i\}$ im Eigenvektorraum gegeben! Möchte man also eine Dekorrelation der transformierten Merkmale erreichen, so muss man dies unabhängig vom kleinsten quadratischen Fehler als zusätzliche, einschränkende Forderung stellen.
Bei der Dekorrelation ist die Basis $\{\mathbf{b}_i\}$ für die Rücktransformation nach Gl.(B.7) durch $\mathbf{W}^{-1}\mathbf{W} = \mathbf{B}\mathbf{W} = \mathbf{I}$, also $\mathbf{b}_i^T \mathbf{w}_j = 0$, $i \neq j$, und damit durch Gl.(B.21) mit $\mathbf{b}_i^T := \mathbf{w}_i^T \mathbf{C}$ gegeben.

C Transform coding

Die zwei Schichten, die für eine beliebig genaue Approximation einer unbekannten Schicht ausreichen, müssen nicht gleichartig aufgebaut, sondern können sehr unterschiedliche Aktivitäts- und Lernregeln enthalten. Ein Beispiel dafür ist das Verfahren des "transform coding", um Bilder und Sprache effizient zu komprimieren.

C.1 Das Konzept des "transform coding"

Eine Folge von zwei Operationen, ein lineare Transformation und anschließende Kompression des Signals durch Vektorquantisierung, ist für die Kodierung (bzw. Umkehroperationen Dequantisierung und lineare Rücktransformation für die Dekodierung) von Bildern oder Sprachsignalen sehr interessant und wird als *transform coding* bezeichnet [WINTZ72]. In Abb. C.1 ist diese Situation veranschaulicht.

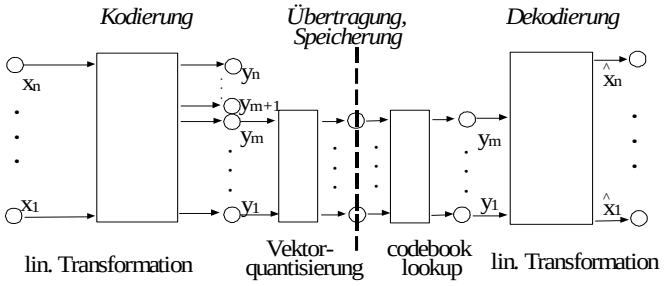


Abb. C.1 *Transform Coding* und Dekodierung

Fehlerlose Verfahren zur Kodierung, etwa die Lauflängenkodierung RLL und das wahrscheinlichkeitsgestützte Huffman-coding, erlauben maximal ein Kompressionsverhältnis von Code zu Originaldaten von ca. 1:2. Diesen ist das *transform coding* Verfahren trotz eines auftretenden (geringen) Fehlers bei der Kodierung überlegen, da es bei vorgegebenen, tolerierbaren Fehlern noch ein Kompressionsverhältnis von ca. 1:20 gestattet (z.B.

JPEG für Stillvideo). Benutzen wir zusätzlich noch die Tatsache, dass zwei Bilder einer Folge (z.B. beim Fernsehen) sehr ähnlich sind und eliminieren auch die Redundanz in der Zeit, so erhalten wir sogar Kompressionsverhältnisse von bis zu 1:100 (z.B. MPEG für Bewegtbilder). Das *transform coding* Verfahren ist deshalb ziemlich wichtig in allen modernen Informationstechnologien, die Signalübertragung und -Speicherung benötigen, wie zum Beispiel

- Telekommunikation (z.B. Bildtelefon)
- Satellitenübertragung (z.B. Wettersatelliten etc.)
- Bilddatenbanken (z.B. Umweltdaten, Medizindaten, Industrieteile, Teleshopping etc.)
- Digitale Musik
- Hochauflösendes Fernsehen (HDTV)
- Multi-Media Daten

Beim *transform coding* wird die Zahl der Signalwege in der linearen Transformation reduziert. Haben wir mit $m<n$ weniger Signale bei der Ausgabe als bei der Eingabe, so bedeutet dies eine Datenreduktion. Obwohl anschließend die reellen Koeffizienten noch quantisiert werden, um damit für eine digitale Nachrichtenübertragung oder Speicherung zur Verfügung stehen, betrachten wir hier nur ausführlich die Bedingungen für die erste, lineare Transformationsschicht. Sie können wir direkt mit neuronalen Netzen zur Signalverarbeitung in Realzeit implementieren.

C.1.1 Die lineare Bildtransformation

Beginnen wir dazu mit der Darstellung eines Bildes aus einzelnen Bildpunkten. Ordnen wir die Bildpunkte (x_{kl}) eines Bildes so an, dass die N_2 Bildzeilen aus jeweils N_1 Bildpunkten hintereinander gefügt werden, die $N_1 \times N_2$ Bildpunkten also alle in einer Reihe $(x_{11},x_{12},..,x_{1N1},x_{21},...,x_{N1N2}) = \mathbf{x}$ vorliegen, so ist das Bild als Vektor statt als Matrix beschrieben und ist mit einem Punkt im $N_1 \times N_2 = n$-dimensionalen Raum identisch. Das Gesamtbild lässt sich als endliche Überlagerung endlicher Basisvektoren (*Basisbilder*) darstellen und ist durch die Wahl der Basis charakterisiert.

$$x_i = \sum_{j=1}^{n} y_j \, \varphi_{ij} \qquad \text{bzw. mit } \mathbf{W} = (\varphi_{ij}) \text{ ist dies } \mathbf{x} = \mathbf{W}\,\mathbf{y} \tag{C.1}$$

Wählt man als Basisvektoren die Eigenvektoren der Kovarianzmatrix, so werden die Basisbilder auch als *Eigenbilder* bezeichnet.

Lässt man einige Komponenten weg bzw. setzt sie auf einen konstanten Wert, so ergibt sich bei der Reproduktion $\hat{\mathbf{x}}$ des Bildes aus dem veränderten Code eine Differenz $(\mathbf{x}-\hat{\mathbf{x}})$, die als erwarteter, mittlerer Fehler $R = \langle(\mathbf{x}-\hat{\mathbf{x}})^2\rangle$ gemessen wird. Der Erfolg des Verfahrens beruht im Wesentlichen darauf, dass die Information, die in den n Signalen

enthalten ist, durch eine Transformation auf *m* bestimmte Kanäle konzentriert werden kann, so dass ein Wegfallen der anderen Kanäle wenig ausmacht. In Abb. C.2 ist dies für die Kodierung und Reproduktion eines zufällig erzeugten Testbildes der Kantenlänge N = 256 mit Hilfe verschiedener Verfahren gezeigt.

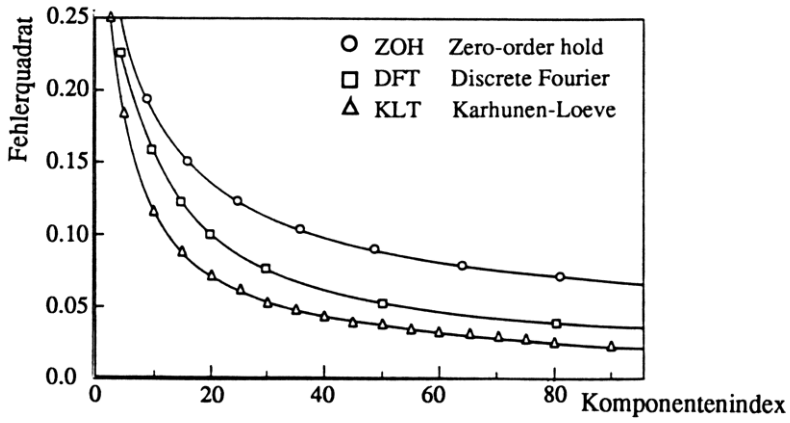


Abb. C.2 *Der Fehler beim Vernachlässigen höherer Komponenten (nach [HAB71])*

Dabei wurden als Basisvektoren die diskretisierten Versionen einer lokalen Mittelwertbildung (des Zero-Order-Hold (ZOH) Algorithmus), der Fouriertransformation (FT) und der Karhunen-Loéve Transformation (KLT), die identisch zur Hauptachsentransformation PCA ist, verwendet. Man sieht, dass anstelle der $n = 256 \times 256 = 65536$ Komponenten auch ca. 80-100 Komponenten (ca. n/3 bei einer guten Bildkodierung [HAB71]) für einen moderaten Fehler ausreichen – eine wesentliche Reduktion in der Anzahl der Kanäle. Dabei ist die KLT mit ihrer Transformation auf Eigenvektoren den anderen Transformationen deutlich überlegen – kein Wunder, da sie ja gerade dadurch definiert ist, dass sie den erwarteten quadratischen Fehler R minimiert.

Allerdings ist die Grundlage dieses Verfahrens durch die Tatsache bestimmt, dass sowohl Bild- als auch Sprachsignale im Kurzzeitbereich einer Gaußverteilung unterliegen und deshalb durch eine lineare, dekorrelierende Transformation, wie sie beispielsweise von den PCA-Netzen durchgeführt wird, effizient mit maximaler Information in wenigen Kanälen kodiert werden können.

C.1.2 Unterteilung in Unterbilder

Ein wesentlicher Fortschritt bei der Bildbearbeitung ergibt sich, wenn man das Gesamtbild in Unterbilder zerteilt und jedes Unterbild für sich, beispielsweise in sequentieller Reihenfolge, abarbeitet. Da die Laufzeitkomplexität der Tranformationsalgorithmen ziemlich groß ist (z.B. O(n logn) bei der schnellen FT und $O(n^3)$ bei der KLT) bedeutet eine sequentielle Bearbeitung eine Linearisierung der Ausführungszeit. Zusätzlich wird die Transformation fehlertoleranter: ein Übertragungsfehler der y_i wirkt sich nur im jeweiligen Unterbild aus, nicht im Gesamtbild. In der Abb. C.3 ist eine solche Unterteilung illustriert, wobei das Unterbild jeweils einem neuronalen Netz zur Bearbeitung als Eingabemuster übergeben wird.

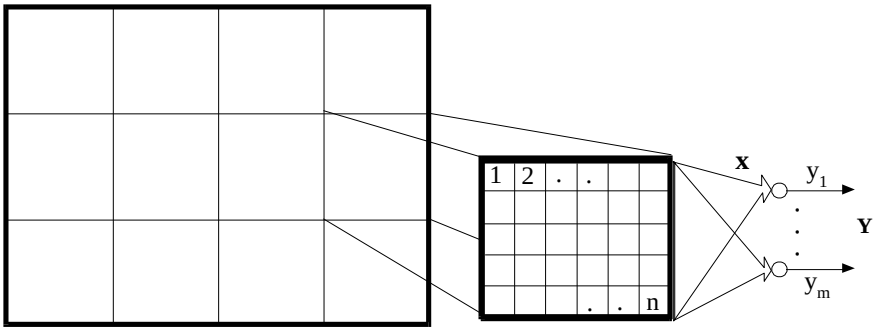


Abb. C.3 D*ie Bearbeitung von Unterbildern*

C.1.3 Das Bildmodell

Allerdings machen wir bei der Unterteilung einen Fehler, da im Mittel die langreichweitigen Korrelationen vernachlässigt werden. Dies bedeutet, dass zwar die Ausgabekanäle innerhalb eines Unterbildes untereinander dekorreliert sind, aber nicht notwendigerweise zwischen benachbarten Pixeln verschiedener Unterbilder. Es lässt sich das Originalbild aus den (unquantisierten) Codewerten bei *n* Ausgabekanälen wieder fehlerfrei herstellen, aber die Kanäle sind dabei nicht optimal kodiert: Mit den Korrelationen enthalten sie Redundanz. Eine Bildunterteilung ist also nur dann sinnvoll, wenn im Bild die Korrelationen zwischen weit entfernten Bildpunkten gering sind. Betrachten wir dazu mit Abb. C.4 eine typische Korrelationsstatistik, wie sie bei einem Bild einer Menschenmenge gemessen wurde. In der Abbildung ist die Zahl jeweils zweier Pixel **x** und **x`** mit gleichem Grauwert als Funktion ihres Abstands aufgetragen. Als Modell für diese

exponentiell abfallende Korrelationsfunktion wurde von Habibi und Wintz (1971) die Funktion

$$C_{xx}(\mathbf{x}\text{-}\mathbf{x}`) = e^{-\alpha|x_1 - x'_1| - \beta|x_2 - x'_2|} \qquad \text{(C.2)}$$

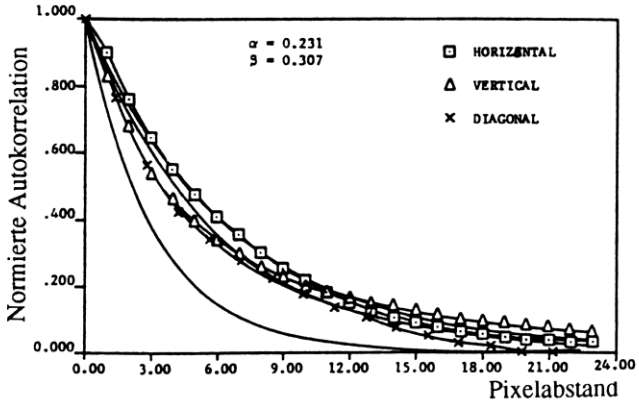


Abb. C.4 *Pixel-Pixel Korrelation als Funktion des Abstands (aus [HAB71])*

vorgeschlagen, wobei der Parameter α die horizontale und β die vertikale Abhängigkeit beschreibt. Die durchgezogenen Linien in Abb. C.4 ist die Funktion in Gl. (C.2) bei α = 0,231 und β = 0,307. Man sieht, dass zwar die horizontale und vertikale Richtung gut modelliert wird, aber nicht die Diagonale.

Die schnell exponentiell abfallende Korrelationsstatistik gestattet relativ kleine Unterbildgrößen. Nach Wintz [WINTZ72] ist die Mindestgröße ein 4x4 Unterbild; für die bekannten JPEG und MPEG Verfahren werden üblicherweise 8×8 Unterbilder (*Blöcke*) für die Schwarz-Weiß Information und 16×16 für die Farbinformation verwendet.

C.1.4 Anwendung: Bildkodierung

Das *transform coding* Verfahren und damit eine Bildkodierung lässt sich leicht mit neuronalen Netzen implementieren. Für die lineare Transformation benutzt man eine lineare Netzschicht, etwa ein PCA-Netz aus Kapitel 3, und für die quantisierende Schicht kann man ein Vektorquantisierungsnetzwerk aus Kapitel 4 eingesetzt werden. Interessant wird das Paradigma der neuronalen Netze mit ihren parallel und lokal arbeitenden Prozessoren aber erst dadurch, dass diese Netze die geforderten Transformationen selbst lernen können, ohne durch zeitaufwendige Optimierungsverfahren vorher errechnet

werden zu müssen. Im Folgenden betrachten wir die Ergebnisse, die wir von einem solchen Netz zur Bildkodierung erwarten dürfen.

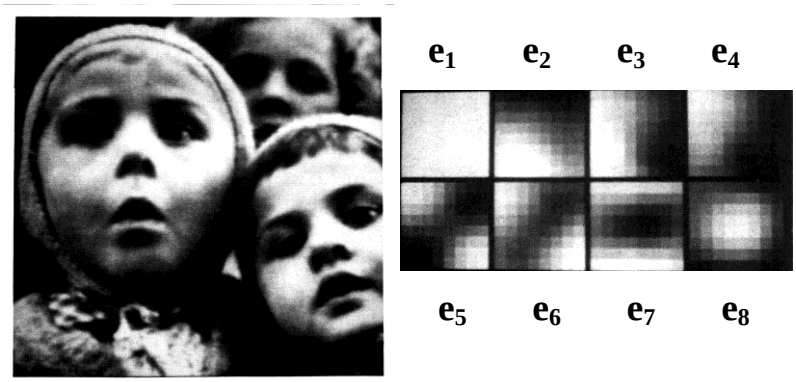


Abb. C.5 *Das Bildmaterial und seine Eigenvektoren [SAN88]*

Betrachten wir zur Illustration eine Datenkompression in der Bildverarbeitung. In der Abb. C.5 ist links ein Bild (256×256 Pixel, 8 Bit) gezeigt, das in Unterbilder der Größe 8×8 = 64 Pixel zerlegt wurde. Dazu wurde es mit einer 8×8 Matrix rasterförmig Unterbild für Unterbild sequentiell "abgescannt" wurde. Bei jedem Schritt erhielt die Abtastmatrix 64 neue Eingabewerte, die von 8 Neuronen mit je 64 Gewichten mit einem PCA-Netz ausgewertet wurden, siehe Abb. C.3. Die Gewichte nach der Iteration sind die gesuchten Eigenvektoren für die Korrelationsmatrix, die sich aus dem Ensemble aller Unterbilder ergibt. Die acht 8×8 Eigenbilder sind also nur typisch für ein einziges Bild, nicht etwa für ein Ensemble aller möglichen Hauptbilder, und sind rechts in der Abbildung dargestellt.

Nimmt man umgekehrt die für jede Position der Abtast-Matrix gewonnenen, quantisierten Darstellung des 8-dim Vektors **y** und errechnet mit dem Satz von Eigenvektoren daraus das Originalbild, so erhält man mit der auf 0,36 Bits/Pixel komprimierten Darstellung (insgesamt 23 Bit bei acht y-Komponenten gegenüber 64 Pixel × 8 Bit = 512 Bit jeder Abtastmatrix **x**) das Bild in Abb. C.6. Interessanterweise bringt eine Bildrestauration mittels der Codewerte eines völlig anderen Bildes, das mit den selben Eigenvektoren auf 0,55 Bits/Pixel komprimiert wurde, ähnlich gute Resultate [SAN88].

Abb. C.6 *Wiederhergestelltes, kodiertes Bild*

Dies deutet darauf hin, dass die Eigenvektoren durch die stark wechselnde Helligkeitsverteilung im Rasterfenster wenig bildspezifisch und sehr allgemein ausgebildet wurden. Die Abweichungen der Bilder durch das schwach erkennbare Streifenmuster in Abb. C.6. sind auf die viereckige, nichtüberlappende Form der 8×8 Matrix zurückzuführen, die eine lokale Fehlerminimierung ohne Nachbarschaftseinfluß (nicht-stetige Approximation, s. [WAT83]) verursacht.

Wie lassen sich die gezeigten Eigenbilder in Abb. C.5 interpretieren; wieso haben sie gerade solche eine unregelmäßige Form ?

C.1.5 Die Form der Eigenbilder

Für die Beantwortung dieser Frage verallgemeinern wir die Darstellung des Bildes. Oft ist es günstiger, anstelle der diskreten Formulierung für Eingaben und Neuronen einen kontinuierlichen Verarbeitungsprozeß zu formulieren, um die analytischen Lösungen leichter mathematisch zu errechnen. Dies führen wir auch hier durch.

Dazu interpretieren wir die Basisvektoren (φ_{i1}, ...,φ_{in}) in Gl.(C.1) als sind die diskreten Abtastwerte von kontinuierlichen Basisfunktionen φ_i, siehe Abb.C.7.

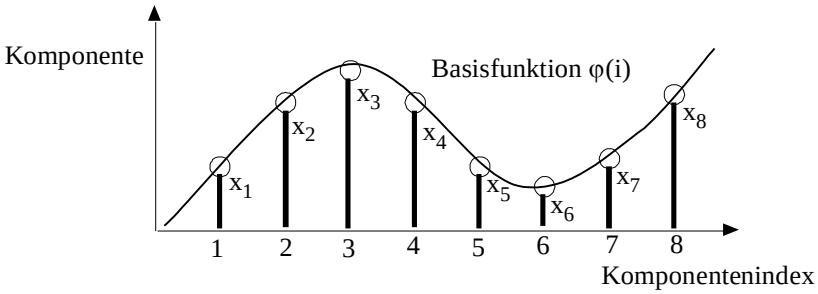


Abb. C.7 *Basisvektor als Abtastung einer Basisfunktion*

Im kontinuierlichen Fall wird bei der KLT das Signal **x**(t) nach orthogonalen Funktionen, den *Eigenfunktione*n, entwickelt. Diese sind wie die Eigenvektoren durch eine Eigenwertgleichung festgelegt, die im kontinuierlichen Fall für 2-dim. Bilder lautet

$$\int_0^N \int_0^N C(x_1,x_2,x_1',x_2')\, \varphi(x_1',x_2')\, dx_1'\, dx_2' = \lambda_{12}\varphi(x_1,x_2) \qquad \text{(C.3)}$$

Eigenwertgleichung

wobei die Korrelationsfunktion aus Gl.(C.2) separierbar ist

$$C(x_1,x_2,x_1',x_2') = C(x_1,x_1')\, C(x_2,x_2') \qquad \text{(C.4)}$$

und damit auch die Basisfunktionen separierbar sind

$$\varphi(x_1,x_2) = \varphi(x_1)\varphi(x_2), \qquad \lambda_{12} = \lambda_1\lambda_2 \qquad \text{(C.5)}$$

Es reicht also aus, Lösungen für die entsprechende ein-dimensionale Eigenwertgleichung zu suchen; die Gesamtlösung ergibt sich nach Gl.(C.4).

Mit den Definitionen B = N/2 und C(x,x') = exp(-γ |x-x'|) kann man die eindimensionale Eigenwertgleichung

$$\lambda\varphi(x) = \int_{-B}^{x} C(x,x')\, \varphi(x')\, dx' + \int_{x}^{B} C(x,x')\, \varphi(x')\, dx' \qquad \text{(C.6)}$$

zweimal ableiten, und mit der Umformung

$$\frac{\partial}{\partial x}\int_B^x f\,(x,t)\, dt = f\,(x,t=x) + \int_B^x \frac{\partial}{\partial x} f(x,t)\, dt \qquad \text{(C.7)}$$

ergibt sich

$$\lambda\varphi'(x) = \gamma\left(-\int_{-B}^{x} C(x,x')\,\varphi(x')\,dx' + \int_{x}^{B} C(x,x')\,\varphi(x')\,dx'\right) \qquad \text{(C.8)}$$

$$\lambda\varphi''(x) = -(2\gamma - \lambda\gamma^2)\,\varphi(x) \qquad \text{bzw.} \qquad \lambda\varphi''(x) + (2\gamma - \lambda\gamma^2)\varphi(x) = 0$$

eine gewöhnliche, homogene Differenzialgleichung. Unterscheiden wir die beiden Fälle

$$(2\gamma - \lambda\gamma^2) > 0 \quad \text{und} \quad (2\gamma - \lambda\gamma^2) < 0,$$

so erhalten wir nach einiger Rechnung als Lösungen

$$\varphi_i(x) = a\cos\left(b_i\gamma(x - N/2)\right) \quad \text{mit } \lambda_i = \frac{2}{\gamma\left(b_i^2 + 1\right)} \quad \text{wobei} \quad b_i \tan(b_i\gamma N/2) = 1 \qquad \text{(C.9)}$$

und

$$\varphi_j(x) = a'\sin\left(b_j'\gamma(x - N/2)\right) \quad \text{mit } \lambda_j = \frac{2}{\gamma\left(b_j'^2 + 1\right)} \quad \text{wobei} \quad b_j'\cot(b_j'\gamma N/2) = 1$$

Die Gesamtlösung nach Gl.(C.4). ist also ein Produkt aus zwei transzendenten Funktionen. Die entstehende Funktionslandschaft $f(x_1,x_2)$ aus Sinus-Hügeln und Tälern ähnelt einem Eierkarton.

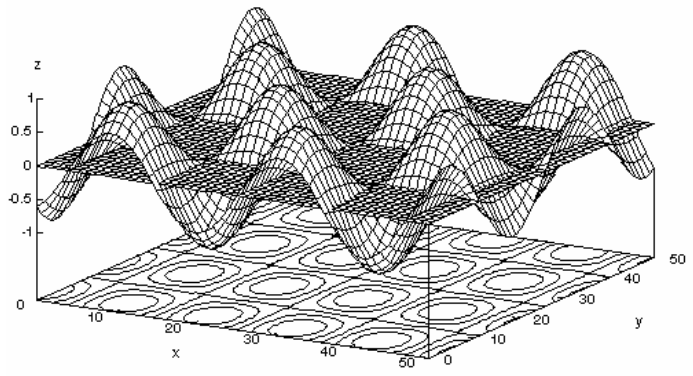


Abb. C.8 *Die erste 2D-Eigenfunktion*

Für die unterschiedlichen Eigenfrequenzen $\nu_i = b_i\gamma/2\pi$ ändert sich die qualitative Gestalt der Landschaft nicht; nur die Anzahl der Hügel und Täler pro Flächeneinheit wird mit zunehmenden Frequenzen größer. Zur Visualisierung sind in Abb. C.9 die Eigenfunktionen für ein 8×8 Unterbild mit der Bildstatistik nach Gl.(C.2) und $\alpha = 0.125$ und $\beta = 0.249$ (ein Bild eines Kameramannes [HAB71]) gezeigt. Sie bilden N×N Ausschnitte aus dem "Eierkarton", die je nach Frequenzen und Phasenlagen aus unterschiedlichen Stellen der Hügellandschaft stammen.

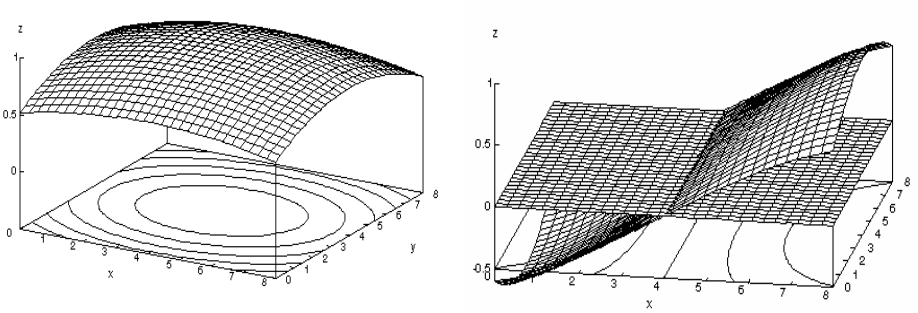


Abb. C.9 *Die ersten beiden Eigenfunktionen für N = 8, $\alpha = 0.125$, $\beta = 0.249$*

Für jede der Einzelfunktionen φ_i ist die Lösung aus Gl. (C.9) gegeben, wobei einmal $\gamma = \alpha$ und einmal $\gamma = \beta$ gilt. Sie entsprechen den Basisfunktionen der Fouriertransformation, allerdings mit einer wichtigen Einschränkung: Bei der Fouriertransformation waren die Frequenzen bzw. die Frequenzabstände der Sinus- bzw. Kosinusfunktionen willkürlich (z.B. alle 100 Hz), hier dagegen sind sie bei fester Unterbildgröße N genau durch die Statistikparameter α und β festgelegt. Obwohl dies die Eigenfunktionen eines anderen Bildes sind als das in Abb. C.5 gezeigte, stimmen doch analytisch und experimentell gefundene Eigenbilder – bedingt durch ähnliche Statistiken – gut überein.

Aufgaben

1) Vernachlässigt man Komponenten höherer Ordnung, so spricht man von "zonal sampling" im Unterschied zur Vernachlässigung aktuell kleiner Komponenten (threshold sampling). Welche Strategie davon verfolgt das *transform coding* Konzept? Das JPEG Verfahren verwendet zwar mit der Kosinus-Transformation ein effizientes Verfahren, aber behält alle Komponenten. Wie kann hier trotzdem (im Mittel) ein *zonal sampling* durchgeführt werden?

2) Man bestimme explizit mit Gl.(C.9) die Parameter und Form der ersten drei Eigenfunktionen für N = 8, $\alpha = 0{,}125$, $\beta = 0.249$.

D Konvergenz von Iterationsverfahren

Mit dem Begriff „Adaptives Lernen“ wird meist der Prozeß verstanden, mit dem „günstige“ Parameter gesucht werden, um das Optimum einer von den Parametern abhängigen Zielfunktion zu erreichen. Dies wird durch eine iterative Verbesserung der Parameter erreicht. Ziel der Iteration ist es, ein Extremum (Maximum oder Minimum) der Zielfunktion zu finden, möglichst das globale Extremum.

Ein solches Extremum F(w*) ist mit der Ableitung $f(w):=\frac{\partial F}{\partial w}$ durch die Bedingung

$$f(w^*) = 0 \tag{D.1}$$

charakterisiert.
Wie erreichen wir das Ziel, den Wert w* für die Nullstelle der Funktion f zu finden ?
Eine gern genutzte Möglichkeit ist das *Gradientenverfahren*. Für eine Minimumssuche wird die Iteration mit

$$w_{t+1} = w_t - \frac{\partial}{\partial w} f(w_t) \equiv g(w_t) \tag{D.2}$$

durchgeführt.
Eine andere Möglichkeit besteht darin, die Nullstelle der Funktion f(w) zu schätzen. In Abb. D.1 ist dies verdeutlicht.

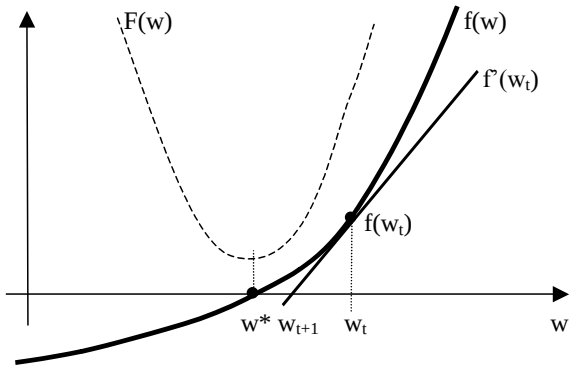


Abb. D.1 *Die Abschätzung der Nullstelle durch die Ableitung*

Die Ableitung der Funktion f(w) in $w = w_t$ schneidet als Tangente die w-Achse dichter am zu findenden Punkt w*. Ihr Wert ist gerade die Steigung der Tangente in diesem Punkt, also

$$f'(w_t) = \frac{f(w_t) - f(w_{t+1})}{w_t - w_{t+1}} = \frac{f(w_t)}{w_t - w_{t+1}}$$

oder

$$w_{t+1} = w_t - \frac{f(w_t)}{f'(w_t)} \qquad \textit{Newton-Verfahren} \tag{D.3}$$

Kennen wir die Ableitung $f' = \partial f/\partial w$ nicht, sondern nur die empirisch ermittelten Werte für f(w), so können wir die Ableitung $f' := (\Delta f/\Delta w)$ auch mit den empirisch ermittelten Differenzen $\Delta f = f(w_t) - f(w_{t-1})$ bei $\Delta w = w_t - w_{t-1}$ approximieren. Dies wird als *regula falsi* bezeichnet.

D.1 Die Konvergenzbedingung

Allgemein hat eine Iteration mit einer Funktion $g(w_t)$

$$w_{t+1} = g(w_t) \tag{D.4}$$

immer dann Erfolg (sie „konvergiert"), wenn bei $t \to \infty$ auch $w_t \to w^*$ gilt, also

$$w^* = g(w^*) \tag{D.5}$$

ist. Da ein solcher Punkt w* mit g(w*) auf sich selbst abgebildet wird, heißt er auch *Fixpunkt*.

Wann konvergiert eine solche Fixpunktgleichung? Betrachten wir dazu die Fixpunktiteration in Abb. D.2. In dieser zweidimensionalen Visualisierung lässt sich der Fortgang der Iteration grafisch verfolgen. Dazu beginnen wir mit dem Wert w_1 und ermitteln $g(w_1)$. Dies ist w_2. Von $g(w_1)$ ziehen wir eine Linie horizontal zur Diagonalen mit $g(w_) = w$ bis zum Schnittpunkt. Der Schnittpunkt hat die Koordinaten $(w_2, g(w_1))$. Ziehen wir von dort eine senkrechte Linie bis zur Funktionslinie, so schneidet sie bei $g(w_2) = w_3$. Auf diese Weise erhalten wir schrittweise $w_1, w_2, w_3, \ldots$ und beobachten die Konvergenz der Folge zum Punkt w*.

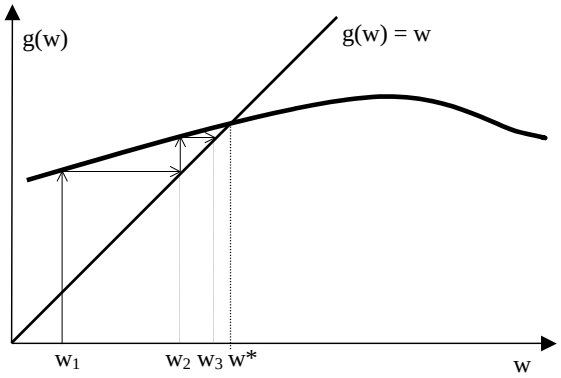


Abb. D.2 *Geometrische Iteration zum Fixpunkt w**

Hat die Funktion g(w) nur eine geringe positive Steigung kleiner als die Diagonale, so schneidet g(w) die Diagonale g(w) = w, auf der der Fixpunkt liegen muß. Die Bedingung „geringe Steigung“ lässt sich auch genauer mit der Ableitung $g'(w) = \partial g(w)/\partial w$ ausdrücken: $0 < g'(w) < 1$. Wie man leicht nachprüfen kann, gilt dies auch für die spiegelsymmetrische Situation mit negativer Ableitung, also für $-1 < g'(w) < 0$, so dass die Fixpunktbedingung allgemein lautet

$$|\square\ g'(w)| < 1 \qquad \text{(D.6)}$$

Bei einem Iterationssystem analog zu (D.4) mit einem Gewichtsvektor **w** statt einer skalaren Variablen lautet die Bedingung (D.6)

$$\left|\det\left(\frac{\partial g_i}{\partial w_j}\right)\right| < 1 \qquad \text{(D.7)}$$

mit der Funktionaldeterminanten (Jacobi-Determinanten) der Funktion g(**w**).

Beispiele

Für das *Newton-Verfahren* ist $g(w) = w - f(w)/f'(w)$ und damit $g'(w) = -f \cdot f'' / (f')^2 \approx 0$ bei $f(w) \approx 0$ um w* herum.

Dies gilt auch für das *regula falsi-Verfahren*.

D.2 Allgemeiner Satz

Der entsprechende mathematische Satz lautet dann also:

Satz Ist g´(w) stetig in einer Umgebung von w* und gilt (D.6), so konvergiert das Iterationsverfahren (D.4) gegen den Fixpunkt w*, wenn der Startpunkt nahe genug an w* liegt.

Beweis:
Wegen der Stetigkeit von g´(w) um w* und (D.6) gibt es ein Intervall U(w*) mit w* als Mittelpunkt derart, dass

$$|g'(w)| \le k < 1 \quad \text{gilt,} \quad \text{wobei} \quad k = \max_{w \in U} |g'(w)| \tag{D.8}$$

Dann gilt mit dem Mittelwertsatz

$$g(w_t) - g(w^*) = g'(\alpha_t)(w_t - w^*), \qquad w_t \le \alpha_t \le w^*$$

und mit (D.4) und (D.5)

$$w_{t+1} - w^* = g'(\alpha)(w_t - w^*)$$

Angenommen, w_0 liegt in U(w*). Für t = 0 bedeutet dies mit Gl.(D.8)

$$w_1 - w^* = g'(\alpha_0)(w_0 - w^*) \quad \Leftrightarrow \quad |w_1 - w^*| = |g'(\alpha_0)|\,|(w_0 - w^*)| \le k\,|w_0 - w^*|$$

Da k < 1 liegt also auch w_1 in U(w*) und somit auch α_1, so dass die obige Relation auch für t = 1 gilt

$$|w_2 - w^*| \le k\,|w_1 - w^*| \le k^2\,|w_0 - w^*|$$

Dies lässt sich weiter fortsetzen, so dass für alle t der Punkt w_t in der Umgebung von w* liegt und es gilt

$$|w_t - w^*| \le k^t\,|w_0 - w^*|$$

Bei festem $|w_0 - w^*|$ ist bei k < 1

$$\lim_{t \to \infty} k^t = 0 \quad \text{und somit} \quad \lim_{t \to \infty} |w_t - w^*| = 0 \quad \text{oder} \quad \lim_{t \to \infty} w_t = w^*$$

q.e.d.

Anmerkung: Enthält die Funktion f(w) eine andere, stark wachsende Funktion h(w), deren Funktionswerte tabelliert sind oder deren Umkehrfunktion wir kennen, so können wir das Verfahren (D.4) auch verallgemeinern. Anstelle des Schnittpunkts mit der linearen Funktion benutzen wir den Schnittpunkt mit der stark wachsenden Funktion, so dass die Iterationsvorschrift als

$$h(w_{t+1}) = g(w_t) \tag{D.9}$$

geschrieben werden kann. Hierbei muß auch die folgende Verallgemeinerung von (D.6) gelten

$$|h'(w)| > |g'(w)| \qquad \text{(D.10)}$$

D.3 Konvergenzarten

Die Konvergenz zu dem gewünschten Fixpunkt kann „langsamer“ oder auch „schneller“ vor sich gehen.

Definition

Eine konvergente Folge $w_1, w_2, \ldots$ mit dem Grenzwert w^* heißt *linear konvergent*, wenn ein *Konvergenzfaktor* q existiert mit

$$q = \lim_{t\to\infty} \frac{w_{t+1} - w^*}{w_t - w^*} \qquad \text{und } |q|<1,\ q\neq 0$$

In unserem Fall gilt mit dem Mittelwertsatz

$$g(w_t) - g(w^*) = g'(\alpha)(w_t - w^*), \qquad w_t \le \alpha \le w^* \qquad \text{(D.11)}$$

und mit (D.2) und (D.5)

$$w_{t+1} - w^* = g'(\alpha)(w_t - w^*) \qquad \text{(D.12)}$$

Im Konvergenzfall ist $\alpha \to w^*$ und somit $q = g'(w^*)$. Also ist unsere Iteration genau dann linear konvergent mit dem Konvergenzfaktor $q = g'(w^*)$, wenn die Bedingungen

$$|g'(w^*)| < 1, \quad g'(w^*) \neq 0$$

erfüllt sind.

Mit der Notation für den Fehler $\varepsilon_t := w_t - w^*$ lässt sich (D.12) auch schreiben als

$$\frac{\varepsilon_{t+1}}{\varepsilon_t} \to g'(w^*) = q$$

Man sieht, dass die Folge ε_t der Fehler mit dem Quotienten q kleiner wird und eine geometrische Reihe bildet. Betrachten wir nun den Fall, wenn die Funktion g(x) im Fixpunkt ein Extremum hat und so $g'(w^*)= 0$ gilt. In diesem Fall ist es sinnvoll, die Formel (D.11) als ersten Term einer Taylorentwicklung nach w_t um w^* zu betrachten und den quadratischen Term der Taylorentwicklung mit hinzuzunehmen unter Vernachlässigung höherer Ordnungen:

$$g(w_t) = g(w^*) + g'(w^*)(w_t - w^*) + \tfrac{1}{2} g''(\alpha)(w_t - w^*)^2, \qquad w_t \le \alpha \le w^*$$

Dies bedeutet für die Fehlerfolge mit $\varepsilon_{t+1} = g(w_t) - g(w^*)$

$$\varepsilon_{t+1} = g'(w^*)\,\varepsilon_t + \tfrac{1}{2} g''(\alpha)\,\varepsilon_t^2$$

Bei Funktionen g(w) mit g´(w*)= 0 ist also

$$\frac{\varepsilon_{t+1}}{\varepsilon_t^2} = \tfrac{1}{2} g''(\alpha) \to \tfrac{1}{2} g''(w^*) = p\,, \qquad g''(w^*) \neq 0$$

Die Proportionalität (konstante Konvergenzfaktor p) zwischen dem Quadrat des Fehlers und dem nächsten Glied bedeutet eine sehr schnelle (*quadratische*) Konvergenz im Unterschied zur obigen linearen Konvergenz.
Ist auch g´(w*)=g´´(w*) = 0, aber g´´´(w*) ≠ 0, so kann man analog verfahren und erhält *kubische* Konvergenz, usw.

Beispiel *Newton-Verfahren bei einem möglichen Fixpunkt*
Beim obig charakterisierten Newton-Verfahren ist die Ableitung g`(w*) = –f·f´´/f´² = 0 weil f(w*) = 0. Die zweite Ableitung ist g´´(w*) = –f´´/f´ ≠ 0, so dass das Newton-Verfahren bei einem Fixpunkt quadratisch konvergiert. Ist z.B. $\varepsilon_t = 10^{-t}$ so wird $\varepsilon_{\tau+1} \sim 10^{-2t}$, die Anzahl richtiger Stellen verdoppelt sich beinahe bei jedem Schritt.

D.4 Zeitabhängigkeit des Konvergenzverlaufs

Angenommen, wir wollen eine Zielfunktion R(w(t)) optimieren. Wie hängt R(w) von der Zeit ab, wenn der Parameter w(t) zeitlich iterativ ermittelt wird? Auch wenn wir die Abhängigkeit R(w) kennen, so ist doch die Entwicklung von w(t) unklar und hängt mit der Konvergenz R(w) → R(w*) von der Konvergenz w → w* ab. Betrachten wir nun den Verlauf von R(w) bzw. die Konvergenz von *w* anhand eines typischen Beispiels. Angenommen, wir schätzen die Zufallsvariable *x* durch den Parameter *w* ab und verwenden zur Verbesserung des Parameters *w* die Gradientenmethode

$$w(t) = w(t-1) - \gamma \frac{\partial R}{\partial w} \tag{D.13}$$

Dann gilt bei Verwendung des mittleren quadratischen Fehlers (MSE) als Zielfunktion

$$R(w) := \langle (x-w)^2 \rangle \tag{D.14}$$

für die Verbesserung von *w* bei der stochastischen Approximation

$$w(t) = w(t-1) - \gamma(w(t-1) - x(t)) \tag{D.15}$$

Bei der Wahl von γ(t) = 1/t lässt sich zeigen, dass bei obiger Iteration zu jedem Zeitpunkt der Parameter den Mittelwert der bisherigen Beobachtungen x(t) = x_t repräsentiert

$$w(t) = \frac{1}{t} \sum_{i=1}^{t} x_i \tag{D.16}$$

Also konvergiert *w* zum Erwartungswert der beobachteten Größe *x*

$$w \to w^* = \langle x \rangle_x \tag{D.17}$$

Soweit unsere allgemeinen Ergebnisse. Wie ist damit der Verlauf von w(t) einzuschätzen? Durch die Verwendung einer Zufallsvariablen *x* kann es viele Verläufe w(t) geben. Dabei können die einzelnen stochastischen Verläufe sehr unterschiedlich ausfallen, siehe Abb. D.3.

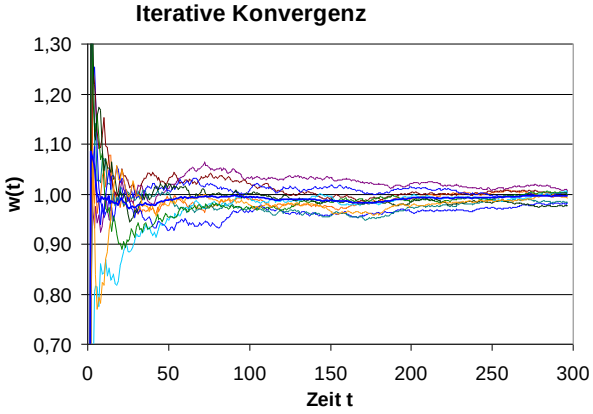


Abb. D.3 *Zehn stochastische Verläufe von w(t) bei w*=1,σ_x=0,288*

Alle Verläufe konvergieren zum selben Ziel, dem Erwartungswert $\langle x \rangle_x$. Angenommen, wir interessieren uns für den erwarteten Verlauf, einer Mittlung über alle Verläufe. Beschränken wir uns auf den Verlauf der Abweichung $|w^* - w(t)|$, so reicht es, für den mittleren Verlauf w*(t) die Größe

$$w^*(t) = w^* - \langle |w^* - w(t)| \rangle \tag{D.18}$$

zu betrachten.
Es gilt mit Gl. (D.16) für die mittlere quadratische Abweichung $\langle |w^* - w|^2 \rangle$ vom Konvergenzziel

$$\sigma_t^2 = \langle (w^* - w(t))^2 \rangle = \left\langle (w^* - \frac{1}{t}\sum_{i=1}^{t} x_i)^2 \right\rangle = \left\langle \left(\frac{1}{t}(t \cdot w^* - \sum_{i=1}^{t} x_i)\right)^2 \right\rangle \tag{D.19}$$

und damit der Erwartungswert über alle möglichen Verläufe

$$\sigma_t^2 = \left\langle \left(\frac{1}{t}\sum_{i=1}^{t} (w^* - x_i) \right)^2 \right\rangle \tag{D.20}$$

Mit der Abkürzung

$$a_i \equiv w^* - x_i \qquad \langle a_i \rangle = \langle w^* - x_i \rangle = \langle w^* \rangle - \langle x_i \rangle = w^* - w^* = 0 \tag{D.21}$$

haben wir eine neue Zufallsvariable mit Mittelwert null definiert, die bei unkorrelierten Zufallsvariablen $\{x_i\}$ ebenfalls unkorreliert ist (Beweis: Übung!). Damit gilt

$$\langle a_i \cdot a_j \rangle = \langle a_i \rangle \cdot \langle a_j \rangle = \begin{cases} 0 & i \neq j \\ \left\langle a_i^2 \right\rangle & i = j \end{cases} \quad \text{mit } \langle a_i^2 \rangle = \sigma_x^2 = \langle a_k^2 \rangle \text{ Varianz} \tag{D.22}$$

und für die Varianz der Konvergenzverläufe zur jeweils *t*-ten Iteration gilt somit

$$\sigma_t^2 = \left\langle \left(\frac{1}{t} \sum_{i=1}^{t} a_i \right)^2 \right\rangle = \left\langle \frac{1}{t^2} (a_1 + a_2 + \ldots + a_t)(a_1 + a_2 + \ldots + a_t) \right\rangle$$

$$= \frac{1}{t^2} \left\langle (a_1^2 + a_1 a_2 + \ldots + a_1 a_t) + (a_2 a_1 + a_2^2 + \ldots + a_2 a_t) + (a_t a_1 + a_t a_2 + \ldots + a_t^2) \right\rangle$$

$$= \frac{1}{t^2} \left\langle (a_1^2 + a_2^2 + \ldots + a_t^2) + \sum_{i \neq j} a_i a_j \right\rangle = \frac{1}{t^2} \sum_{i=1}^{t} \left\langle a_i^2 \right\rangle + 0$$

Nach der Definition Gl.(D.21) und nach Gl.(D.22) ist das erwartete Quadrat von a_i identisch mit der Varianz der Zufallsvariablen *x*, und dies für alle Indizes. Also gilt

$$\sigma_t^2 = \frac{1}{t^2} \sum_{i=1}^{t} \sigma_x^2 = \frac{1}{t^2} t\sigma_x^2 = \frac{1}{t} \sigma_x^2 \quad \text{bzw. } \sigma_t = \langle |w^* - w(t)| \rangle = \frac{1}{\sqrt{t}} \sigma_x \tag{D.23}$$

und somit für die gesuchte Abhängigkeit w*(t)

$$w^*(t) = w^* - \frac{1}{\sqrt{t}} \sigma_x \tag{D.24}$$

Der nach Gl.(D.24) erwartete Verlauf ist in Abb. D.4 dargestellt, zusammen mit einem Probelauf und dem mittleren Verlauf aus den obigen 10 stochastischen Verläufen.

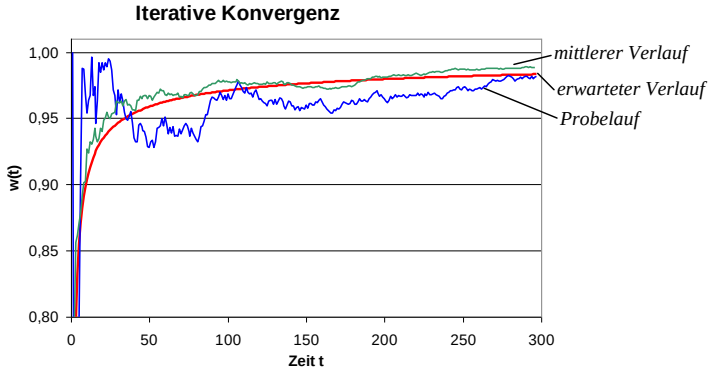


Abb. D.4 *Probelauf von w(t), gemittelter Verlauf und erwarteter Verlauf*

E Optimierung mit Nebenbedingungen

Dieser Abschnitt möchte Ihnen kurz die Systematik der Optimierung mit Nebenbedingungen vorstellen, soweit sie in diesem Buch verwendet wird. Wir setzen voraus, dass Sie aus der elementaren Schulmathematik wissen, wie Sie eine Kurvendiskussion (Nullstellen, Extremwerte, Wendepunkte, Sattelpunkte usw.) durchführen.

Das Minimum oder Maximum einer Zielfunktion lässt sich leicht dadurch errechnen, indem man die Extremwerte sucht: Man sucht die Stelle, an der die Ableitung der Zielfunktion null ist. Was tut man aber, wenn gleichzeitig noch eine oder mehrere Nebenbedingungen gegeben sind?

> **Beispiel**: Unter allen Rechtecken mit der Fläche *F* suche man dasjenige, das den kleinsten Umfang hat. Übersetzt bedeutet das, das Minimum der Funktion R(a,b) = 2a+2b zu suchen bei der Nebenbedingung ab = F. Wie findet man eine Lösung?

Für die Lösung derartiger Probleme benutzt man gern den Ansatz von Lagrange.

E.1 Der Lagrange-Formalismus

Angenommen, die Nebenbedingungen lassen sich in der Art formulieren

$$g(x) = 0 \qquad \text{(E. 1)}$$

was für unser Beispiel $g(a,b) = ab - F = 0$ bedeutet. Dann besagt der Satz von Lagrange, dass das Extremum der stetigen Funktion R(a,b) genau dann angenommen wird, wenn die Lagrangefunktion L

$$L(a,b,\lambda) := R(a,b) + \lambda g(a,b) \qquad \text{(E.2)}$$

ihre Extremwerte annimmt, also

$$\frac{\partial L}{\partial a} = 0,\ \frac{\partial L}{\partial b} = 0,\ \frac{\partial L}{\partial \lambda} = 0 \qquad \text{(E.3)}$$

gilt. Dabei ist λ ein neu eingeführter Parameter, der sog. *Lagrange-Multiplikator*, dessen Wert ermittelt werden muß. Die dritte Bedingung ist klar, dies entspricht gerade der

Definition der Nebenbedingung; die erste und zweite Bedingung ist aber neu. Für unser Beispiel liefert uns dies die zwei Gleichungen

$$\frac{\partial L}{\partial a} = \frac{\partial R}{\partial a} + \lambda \frac{\partial g}{\partial a} = 0 \quad \text{bzw.} \quad \frac{\partial L}{\partial b} = \frac{\partial R}{\partial b} + \lambda \frac{\partial g}{\partial b} = 0 \tag{E.4}$$

Für unser Beispiel bedeutet dies die Gleichungen $2 + \lambda b = 0$ sowie $2 + \lambda a = 0$. Lösen wir beide Gleichungen nach λ auf und setzen sie gleich, so erhalten wir $a = b$: Das optimale Rechteck mit kleinstem Umfang ist ein Quadrat. Man beachte, dass die Ausdehnung des einfachen Ansatzes für einen Extremwert auf die zusammengesetzte Lagrangefunktion nicht selbstverständlich ist, da die obige Funktion L kein multidimensionales Extremum hat, insbesondere bezüglich der Variablen λ: Für λ ist sie nur eine lineare Funktion.

Die obige Methode kann man leicht verallgemeinern: Hat man *n* Parameter der Zielfunktion R sowie *m* Nebenbedingungen, so müssen wir einen Lagrangefunktion aufstellen der Form

$$L(x_1, ..., x_n, \lambda_1, \dots, \lambda_m) = R(x_1, ..., x_n) + \sum_{i=1}^{m} \lambda_i \, g(x_1, ..., x_n) \tag{E.5}$$

und erhalten $n+m$ Ableitungen für die $n+m$ Variablen $x_1, ..., x_n, \lambda_1, \dots, \lambda_m$.

Allerdings gibt es ein Problem: Die Entscheidung, ob überhaupt ein Extremum existiert und ob es sich um ein Maximum oder Minimum handelt, muß man aus dem Kontext der Aufgabe erschließen.