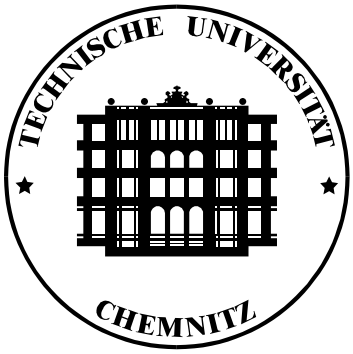
Neuronale Netze und deren Eignung zur Klassifikation im Data Mining

Diplomarbeit



Technische Universität Chemnitz
Fakultät für Informatik

eingereicht von Andrei Krell
geboren am 24. November 1975 in Meerane
Matrikelnummer: 16451

Betreuer: Prof. Dr. Werner Dilger

Chemnitz, den 26.09.2001

Inhaltsverzeichnis

Vorwort

Die folgende Arbeit ist das Ergebnis einer etwa sechs Monate andauernden Studie über neuronale Netz und deren Einsatz im Data Mining. Hiermit möchte ich mich bei allen bedanken, die mich bei meiner Diplomarbeit unterstützt haben und ohne deren Hilfe es nicht möglich gewesen wäre, diese Arbeit zu erstellen.

Ein besonders großer Dank gilt meinem Betreuer Prof. Dr. Werner Dilger, der mir nicht nur bei dieser Arbeit unterstützend zur Seite stand, sondern auch entscheidend meinen studentischen Werdegang mit beeinflusste. Er war es auch, der mich durch die Vorlesung „Einführung in die Künstliche Intelligenz“ dazu inspirierte, das Fachgebiet Künstliche Intelligenz als Vertiefungsgebiet zu wählen. Er hat somit den wichtigsten Teil meines Studiums entscheidend geprägt und stand mir stets hilfreich zur Seite.

Weiterhin möchte ich den Mitarbeitern der Firma Prudential Systems Software GmbH danken. Den Geschäftsführern danke ich für die finanzielle Unterstützung, auch während der letzten zwei Jahre in denen ich hier als Praktikant tätig war. Ich habe mir in dieser Firma sehr viel Wissen im Bereich Data Mining angeeignet. So entstand letztendlich auch die Idee für diese Arbeit in Abstimmung mit der Geschäftsführung und Prof. Dr. Werner Dilger.

Rolf Rossius und Sascha Trautzsch danke ich für ihre Unterstützung bei algorithmischen Problemen und speziellen Fragen zum Data Mining. Torsten Graf und Michael Theß unterstützten mich bei Fragestellungen in Zusammenhang mit der Programmiersprache Java. Weiterhin stand mir Dieter Linstädt mit seinen Englisch-Kenntnissen bei schwierigen Übersetzungen hilfreich zur Seite.

Schließlich möchte ich mich bei allen meinen Freunden für ihre Hilfe bedanken. Ist sie auch noch so klein gewesen, hat sie dennoch zum Gelingen dieser Arbeit beigetragen.

Abschließend möchte ich sagen, dass mir die Arbeit mit dieser Thematik sehr viel Spaß bereitete. Ich habe dadurch Einblick in eine Welt gehabt, die ich vorher nur oberflächlich kannte. Ich kann dies nur jedem weiter empfehlen, da die neuronalen Netze wirklich ein sehr interessantes Gebiet der Informatik darstellen.

Chemnitz, im September 2001 Andrei Krell

1 Einleitung

1.1 Motivation

Das Studium der neuronalen Netze ist stark inspiriert durch den Versuch, das menschliche Gehirn möglichst perfekt nachzuahmen. Deshalb zählen sie heute mit zu den wichtigsten Forschungsgebieten der Künstlichen Intelligenz, wenn nicht sogar der Informatik überhaupt. Sie stellen eine Art Verbindungsglied zwischen der Informatik, der Mathematik, der Biologie und der Medizin her.

Durch die Forschung auf diesem Gebiet wird u.a. versucht, die Fähigkeiten des menschlichen Nervensystems auf den Computer zu übertragen. Als wichtigste Eigenschaften seien hier die Fähigkeit zum selbstständigen Denken und die enorme „Rechenleistung“ des menschlichen Gehirns genannt. Letztere ergibt sich vor allem durch die massiv parallelisierte Verarbeitung, zu der selbst heutige Großrechner noch nicht in der Lage sind. Die Forschungen helfen ebenfalls, bisher ungeklärte Fragen über das „menschliche Denken“ zu beantworten.

Ein Ziel der Wissenschaftler ist es, die positiven Fähigkeiten der Systeme Mensch und Computer zu vereinen. Bei Computern sei hierbei vor allem die große Rechengeschwindigkeit genannt, die sich aus den extrem geringen Schaltzeiten der Transistoren ergibt. Künstliche neuronale Netze besitzen zudem eine Fähigkeit, die neben ihnen nur wenige andere Systeme der Informatik besitzen:

> Sie können selbständig aus gegebenen Daten lernen, ohne dafür explizit programmiert werden zu müssen.

Gerade diese Leistung, vorhandene Muster oder Daten und deren Zusammenhänge selbstständig zu erlernen, macht sie so interessant für den Einsatz im Data Mining.

Data Mining ist eine sehr junge Disziplin, die als Verbindungsglied zwischen der Informatik, der Mathematik und der Wirtschaft gesehen wird. Ziel ist es dabei, durch Methoden der künstlichen Intelligenz und der Statistik, aus einer großen Anzahl vorhandener Daten verborgenes Wissen zu extrahieren. Dieses sollte im Anschluss an die Untersuchung marktwirtschaftlich eingesetzt werden, um z.B. die erzielten Gewinne zu erhöhen. Bei der Verbindung zwischen Data Mining und der Erforschung von künstlichen neuronalen Netzen setzt diese Arbeit an.

Ziel dieser Arbeit ist es somit, dem Leser einen Einblick über die verschiedenen Möglichkeiten für den Einsatz neuronaler Netze bei Klassifikationsaufgaben im Data Mining zu geben und aus eigenem Ermessen eine sinnvolle Auswahl zu treffen.

1.2 Vor- und Nachteile neuronaler Netze

Um die Frage zu beantworten, **warum** der Einsatz neuronaler Netze sinnvoll erscheint (oder warum nicht), sollen zunächst einige wichtige Vorteile (und dementsprechend auch Nachteile) neuronaler Netze gegenüber herkömmlichen Algorithmen aufgeführt werden. Diese Aufzählungen sind natürlich keinesfalls vollständig oder bindend, da die Einschätzungen immer im Ermessen des Betrachters liegen.

1.2.1 Vorteile

Lernfähigkeit — Neuronale Netze besitzen die Fähigkeit, aus gegebenen Daten selbstständig zu lernen ohne dafür explizit programmiert werden zu müssen. Sie passen sich somit selbstständig den Eingaben an.

Fehlertoleranz — Das Erkennen von gespeicherten Mustern ist auch möglich, wenn das Eingabemuster unvollständig oder ein Teil davon fehlerhaft ist. Sie besitzen somit eine hohe Generalisierungsfähigkeit und Robustheit gegen Störungen von außen.

Parallelisierung — Die Berechnung der einzelnen Komponenten ist durch die Struktur der Netze massiv parallelisierbar. Sie eignen sich daher sehr gut für den Einsatz auf Parallelrechnern oder verteilten Systemen.

Nachtrainierbarkeit Neuronale Netze können relativ einfach mit neuen Daten nachtrainiert werden, ohne die Struktur des Netzes stark zu verändern oder vorhandenes Wissen neu erlernen zu müssen.

Datenkenntnisse Für das Training eines neuronalen Netzes muss nicht die gesamte Datenmenge als Ganzes zur Verfügung stehen. Es reicht aus, dem Netz die Datensätze einzeln nacheinander zu präsentieren, was wiederum Speicher spart oder vorteilhaft ist, wenn die Vorverarbeitung der Daten sehr viel Zeit in Anspruch nimmt.

1.2.2 Nachteile

Genauigkeit Neuronale Netze liefern keine genauen Ergebnisse. Für Aufgaben, bei denen exakte Lösungen verlangt werden (z.B. Rechenaufgaben) sind sie somit ungeeignet.

Geschwindigkeit Die Berechnungen neuronaler Netze laufen in der Regel sehr langsam ab. Für die Analyse sehr großer Datenmengen sollten sie somit nicht die erste Wahl sein, wenn effektivere Algorithmen zur Verfügung stehen.

Herleitung Die Analyse des Wissens bzw. die Herleitung des Problemlösungsweges ist sehr schwierig. Diese ist z.B. ein Nachteil für den Einsatz in Expertensystemen, bei denen der Herleitungsvorgang eine wichtige Rolle spielt.

1.3 Aufbau dieser Arbeit

Zunächst wird in Kapitel 2 ein Überblick über den grundsätzlichen Aufbau und die Funktionsweise künstlicher neuronaler Netze gegeben. Dadurch soll es Lesern ohne entsprechende Vorkenntnisse möglich sein, sich relativ schnell in die Thematik einzuarbeiten.

In Kapitel 3 wird eine Reihe von verschiedenen Netztypen und deren jeweilige Arbeitsweise vorgestellt. Die Reihenfolge orientiert sich im Groben an der strukturellen und algorithmischen Komplexität der Netze. Durch diese Strukturierung soll deutlich werden, dass viele kompliziert erscheinende Netztypen auf einfachen Strukturen aufbauen. Das Wissen über die Funktionsweise der „Grundtypen“ ist somit Vorraussetzung für tieferes Verständnis der gesamten Thematik. Ohne die Kenntnis verschiedener neuronaler Netze ist es auch nicht möglich, für die Aufgabenstellung ein geeignetes auszuwählen.

Mit der Auswahl eines Netztyps beschäftigt sich Kapitel 4. Am Anfang dieses Kapitels wird noch einmal genauer auf die Problemstellung, die „Klassifikation im Data Mining mit Hilfe neuronaler Netze“, eingegangen. Die im Kapitel 3 vorgestellten Netzen werden somit auf ihre Eignung für die Klassifikation hin untersucht und schließlich wird die Wahl zugunsten eines Netztyps getroffen. Das eine Auswahl hierbei nicht so einfach ist wird ebenfalls schnell deutlich.

Auf Kapitel 3 und vor allem Kapitel 4 liegt somit auch der Schwerpunkt der Arbeit.

Im Kapitel 5 folgt eine Zusammenfassung zur Softwareimplementierung und den sich daraus resultierenden Erkenntnissen und Schwierigkeiten. Schließlich wird in Kapitel 6 ein kurzer Ausblick auf mögliche weitere Ziele gegeben.

Parallel zum theoretischen Teil der Arbeit bestand die Aufgabe auch darin, das in Kapitel 4 ausgewählte neuronale Netz in Java zu implementieren. Dieses soll in naher Zukunft als ein weiteres Klassifikationsmodul in ein bereits bestehendes Data-Mining-System der Firma Prudential Systems Software GmbH integriert werden.

Die in dieser Arbeit verwendeten Formelzeichen sind Anhang A zu entnehmen.

2 Aufbau Neuronaler Netze

Beim Begriff des neuronalen Netzes muss zwischen zwei verschiedenen Arten unterschieden werden. Zum einen ist dies das biologische Vorbild, also das Gehirn. Zum anderen ist dies das informationsverarbeitende System, das lediglich versucht das biologische Vorbild im Computer zu simulieren.

In den meisten Fällen ist mit dem Begriff *Neuronales Netz* (*NN*) das *künstliche neuronale Netz* (*KNN*) gemeint. Der Aufbau solch eines Netzes ist durch den Aufbau des menschlichen Gehirns inspiriert, bei dem Informationsverarbeitung durch viele einfache Nervenzellen stattfindet. Das künstliche neuronale Netz besteht somit ebenfalls aus einer großen Menge einfacher Einheiten (*Zellen*, *Neuronen*), die sich Informationen über gerichtete, gewichtete Verbindungen (*connections*, *links*) zusenden.

Auf den Aufbau und die Funktionsweise des biologischen Vorbilds, also des menschlichen Gehirns möchte hier nicht eingehen, da es nicht sehr viel mit dem eigentlichen Thema dieser Arbeit zu tun hat. Eine Beschreibung findet sich in vielen Büchern wieder, eine sehr gute z.B. bei [Zell].

2.1 Aufbau einzelner Neuronen

2.1.1 Bestandteile eines Neurons

Die Neuronen eines künstlichen neuronalen Netzes entsprechen stark vereinfachten Nervenzellen. Ein Neuron besteht, wie sein Vorbild die Nervenzelle, aus einem Zellkörper, mehreren Eingängen (in der Natur die *Dendriten*) sowie einem Ausgang (in der Natur das *Axon*). Der Ausgang gibt die Information an die Eingänge mehrerer Nachfolgezellen ab. Die Stärke der Verbindungen zwischen den Neuronen wird durch ein Verbindungsgewicht angegeben.

Die Werte am Ausgang des Neurons sind eine direkte Folge der Werte, die am Eingang anliegen. Es sind jedoch zwischen Eingang und Ausgang mehrere Zwischenstufen geschaltet, die die Berechnung einerseits komplizierter machen, aber andererseits den Netzen mehr Flexibilität ermöglichen. Dies sind im Einzelnen die *Eingangsfunktion* und der damit berechnete *effektive Eingang*, die *Aktivierungsfunktion* und die damit verbundene *Aktivität* und schließlich die *Ausgangsfunktion*, die den *Ausgang* berechnet. Der Ausgang des Neurons verzweigt sich im Allgemeinen und stellt somit einen Eingang für ein oder mehrere andere Neuronen dar.

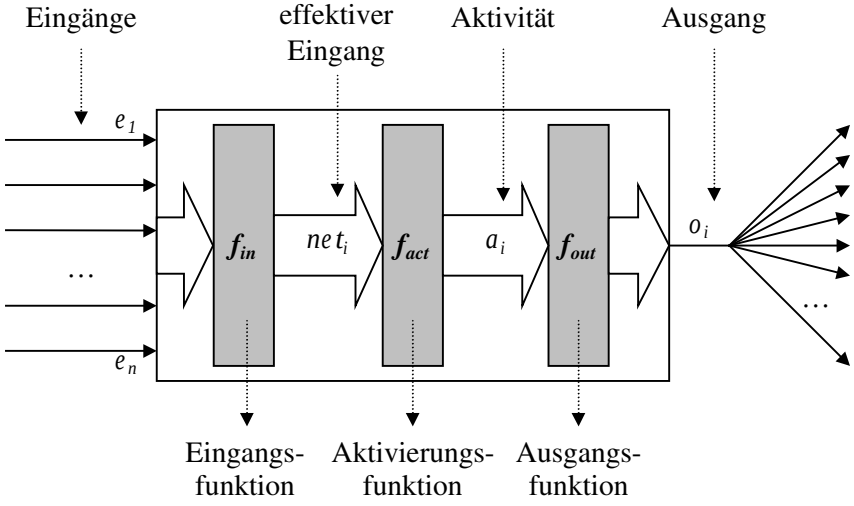


Abbildung 2-1: Schematischer Aufbau eines Neurons

2.1.2 Eingangsfunktion f_{in}

(auch *Propagierungsfunktion*)

Die Eingabefunktion legt fest, wie der effektive Eingang net_i (auch als *Eingabewert* bezeichnet) eines Neurons aus den Werten an den Eingängen (e_1, ..., e_n) und den Verbindungsgewichten berechnet wird. In der Regel werden dazu die Eingabewerte (multipliziert mit dem jeweiligen Verbindungsgewicht) aufsummiert:

$$net_i = \sum_{j=1}^{n} w_j e_j \ .$$

Da die Verbindungen direkt von Zelle zu Zelle verlaufen, entsprechen die Eingabewerte des aktuellen Neurons *i* den Ausgabewerten der jeweiligen Vorgängerneuronen. Alternativ können auch folgende Methoden zur Berechnung verwendet werden:

- Maximalwert der gewichteten Eingabewerte,
- Minimalwert der gewichteten Eingabewerte,
- Produkt der gewichteten Eingabewerte.

2.1.3 Aktivierungsfunktion f_{act} und Aktivität a

(auch *Aktivierungszustand*, *Aktivierung*)

Die Aktivität eines Neurons ergibt sich aus dem effektiven Eingang mittels einer Aktivierungsfunktion. Diese gibt an, wie sich eine neue Aktivität $a_i(t + 1)$ des Neurons aus der alten Aktivierung und dem effektiven Eingang berechnet:

$$a_i(t+1) = f_{act}(a_i(t), net_i(t), \Theta_i),$$

Θ_i gibt dabei den Schwellenwert des Neurons an.

Es gibt verschiedene Aktivierungsfunktionen, die am häufigsten verwendete ist jedoch die *lineare Aktivierungsfunktion*, bzw. als Spezialfall dieser die *Identität*. Die lineare Aktivierung drückt eine Proportionalität zwischen der Aktivität und dem effektiven Eingang aus:

$$a_i = s \cdot net_i \ ,$$

mit *s* als Skalierungsfaktor. Da dieser meist zum Ausgang dazugeschlagen wird, ergibt sich als Funktion die Identität:

$$a_i = net_i \ .$$

Bei [Hoffmann] (S.18ff.) werden noch weitere Aktivierungsfunktionen beschrieben, die ich hier aber nur auszugsweise erwähnen möchte:

- BSB-Aktivierungsfunktion,
- DMA-Aktivierungsfunktion,
- Hopfield-Aktivierungsfunktion.

Die Aktivierungsfunktion wird oft als Teil der Ausgabefunktion gesehen und bei der Berechnung vernachlässigt. Dies stellt bei der linearen Funktion und besonders bei der Identität auch kein Problem dar, da der berechnete Wert ja nur „weitergeleitet“ wird.

2.1.4 Ausgangsfunktion f_{out}

Die Ausgabefunktion bestimmt aus der Aktivität der Zelle deren Ausgabe. Der Ausgangswert ist also eine direkte Folge der Aktivität eines Neurons. Die einzige Forderung, die an eine Ausgangsfunktion gestellt wird, ist die Monotonie. Sie muss monoton wachsend sein, da man erwartet, dass bei steigender Aktivität der Ausgang nicht abnimmt. Die Berechnung der Ausgabe erfolgt nach der allgemeinen Formel:

$$o_i(t) = f_{out}(a_i(t)).$$

Die Ausgabewerte können im Prinzip beliebige reelle Werte sein. Die meisten Modelle schränken den Wertebereich jedoch auf ein Intervall ein. Sehr häufig wird der Ausgang sogar bis auf wenige (meist zwei oder drei) Werte eingegrenzt, z.B. 0 und 1 als binäres Modell.

Bei einigen Modellen hängt der Ausgang eines Neurons auch direkt von den Ausgängen anderer Neuronen ab. Ein Beispiel ist eine *winner-takes-all-Schicht*, bei der nur das Neuron einer Schicht „feuert", dessen Aktivität am höchsten ist. Bei den anderen Neuronen dieser Schicht wird das Feuern unterdrückt.

Die Ausgangsfunktionen kann man in folgende Arten unterteilen:

Schwellenwertfunktionen

(siehe Abbildung 2-2a)

Diese sind den Verhältnissen bei den biologischen Nervenzellen nachempfunden. Das Neuron feuert, wenn die Aktivität ein bestimmtes Potential, den Schwellenwert, überschreitet. Für den Ausgang sind nur zwei Werte zugelassen, im Allgemeinen {0, 1} oder {-1, 1}. Im ersten Fall spricht man von einem binären Neuron.

Folgende Kennwerte bestimmen das Aussehen einer Schwellenwertfunktion:

- *Minimum*,
- *Maximum*,
- *Schwelle* Θ.

An der Stelle, an der die Aktivität gleich dem Schwellenwert Θ ist (also $a = \Theta$), hat die Funktion einen Sprung. Welchen Wert (ob Minimum oder Maximum) man dem Ausgang an dieser Stelle zuordnet ist willkürlich.

Die Schwellenwertfunktion hat den Nachteil, dass sie nicht stetig und nicht differenzierbar ist, was sie für einige Anwendungen unbrauchbar macht.

Sigma-Funktionen

(siehe Abbildung 2-2e,f)

Eine Sigma-Funktion ist im Gegensatz zur Schwellenwertfunktion stetig und differenzierbar. Da an eine Ausgangsfunktion die Forderung gestellt wird, dass diese monoton wachsend sein muss, sieht der Graph dieser Funktionen wie ein verzerrtes „S" aus (daher auch in einigen Büchern *Quetschfunktion* genannt).

Einige Beispiele für Sigma-Funktionen sind:

- die *Fermi-Funktion* (einfache oder allgemeine),
- *Tangens hyperbolicus* (*tanh*),
- *Sinus-Funktion*.

Eine Sigma-Funktion hat vier Kennwerte, mit denen sie (aber nicht eindeutig) beschrieben wird:

- das *Minimum*, als der Wert dem die Funktion für negative Werte zustrebt,
- das *Maximum*, als der Wert dem die Funktion für große positive Werte zustrebt,
- die *Schwelle* Θ entspricht der Schwelle bei der Schwellenwertfunktion, kann aber nicht willkürlich definiert werden und ist somit meist der Mittelpunkt der Funktion,
- die *Steigung* σ, welche ein Maß für die Stärke des Anstiegs ist.

Wie bereits erwähnt, ist die Sigma-Funktion durch diese vier Werte keinesfalls eindeutig bestimmt. Ein Beispiel dafür sind die Sinus- und Tangens hyperbolicus-Funktionen. Diese haben die gleichen Kennwerte, unterscheiden sich aber im Detail (z.B. ist der Funktionswert an der Stelle π bei der Sinus-Funktion 1 und bei der tanh-Funktion 0,99627).

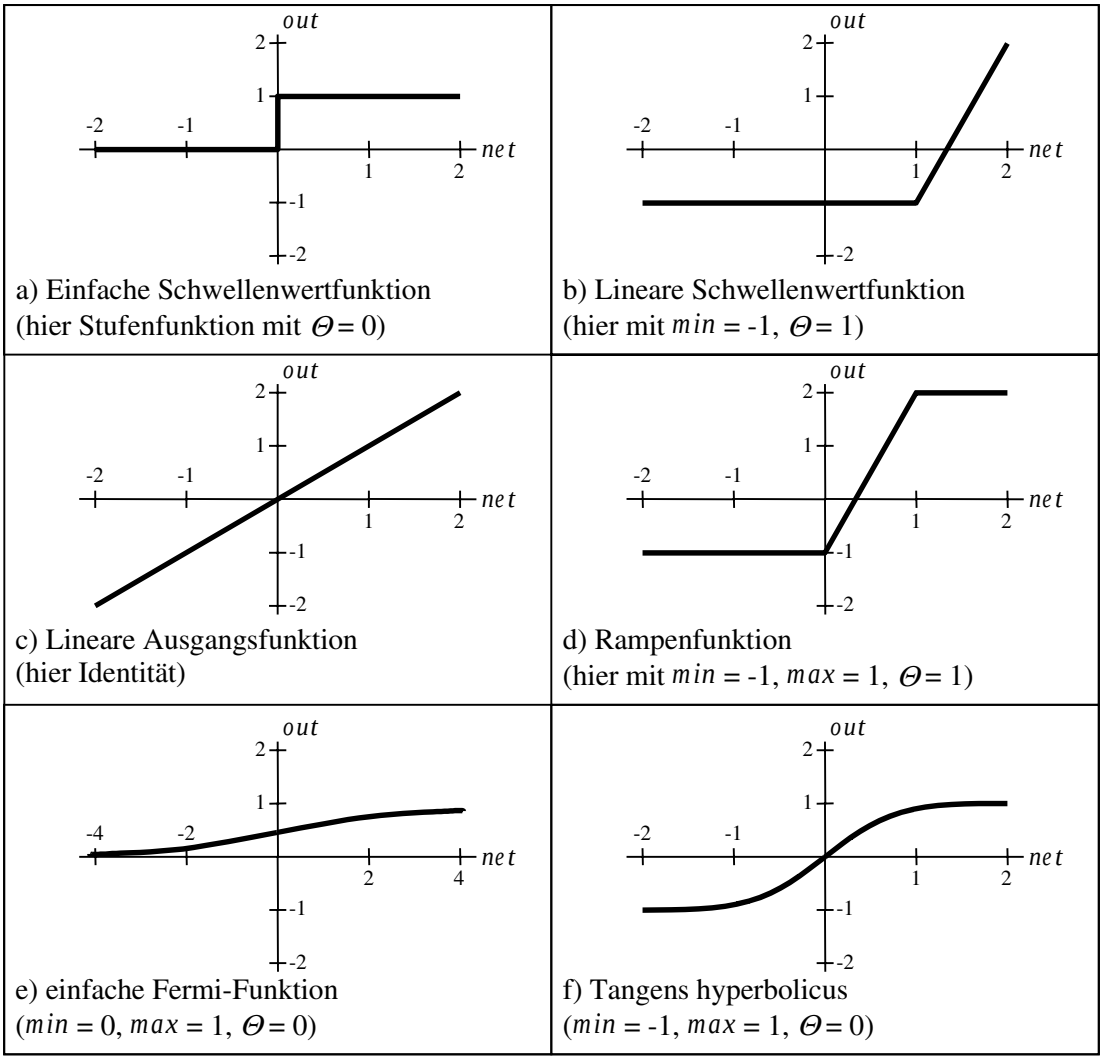


Abbildung 2-2: Beispiele häufig verwendeter Aktivierungs- bzw. Ausgangsfunktionen

Abbildung 2-2 gibt einen kurzen Überblick über das Aussehen häufig verwendeter Ausgangsfunktionen bzw. Aktivierungsfunktionen.

Weitere Funktionen

Es gibt noch weitere Arten von Ausgangsfunktionen, die hier noch kurz erwähnt werden sollen, ohne näher darauf einzugehen:

- Rampenfunktion:

  (siehe Abbildung 2-2d)

  Diese ist ähnlich der Sigma-Funktion aufgebaut, jedoch nicht differenzierbar. Sie ist wie eine Schwellenwertfunktion aus drei Geradenstücken zusammengesetzt, wobei das mittlere linear ansteigt und es somit keinen Sprung zwischen Minimum und Maximum gibt.
- Lineare Schwellenwertfunktion:

  (siehe Abbildung 2-2b)

  Diese ist eine spezielle Variante der Rampenfunktion ohne obere Begrenzung durch das Maximum.

- Lineare Ausgangsfunktion:

  (siehe Abbildung 2-2c)

  Diese Funktion ist nur durch die Steigung und den Schwellenwert charakterisiert. Eine obere bzw. untere Schranke gibt es bei dieser Funktion nicht. Ein Spezialfall dieser Funktion ist die *Identität*. Dabei wird der Ausgabewert gleich dem Eingabewert der Funktion gesetzt.
- Stochastische Ausgangsfunktion:

  Diese Funktion ist, im Gegensatz zu den bisher erwähnten, nicht deterministisch. Die Modelle die diese Funktion benutzen ermitteln den Ausgang (üblicherweise 0 oder 1) nach einer stochastischen Regel. Die Funktion gibt dann lediglich an, mit welcher Wahrscheinlichkeit der Ausgang einen bestimmten Wert annimmt. Ein Beispiel dafür ist die Boltzmann-Maschine, die im Abschnitt 3.9 behandelt wird.
- Wettbewerbs-Ausgangsfunktion:

  Bei dieser Funktion handelt es sich um das bereits beschriebene *winner-takes-all-Prinzip*. Es gibt nur ein Neuron, das aktiv wird (Ausgang wird z.B. 1). Dies ist das Neuron, das die höchste Aktivität innerhalb der Schicht besitzt.

2.1.5 Typen von Neuronen

In [Hoffmann] (S.33) ist eine Tabelle der verschiedenen Arten von Neuronen abgebildet, die eine besonders weite Verbreitung gefunden haben. Diese Neuronentypen unterscheiden sich im Wertebereich und in den Funktionen zur Berechnung. Die Formeln wurden entsprechend der in dieser Arbeit verwendeten Nummerierung umformuliert:

Typ	Ausgangs-werte-bereich	Ausgangs-funktion	Formel	Eingangs-funktion	Aktivie-rungs-funktion
McCulloch-Pitts	{0, 1}	Stufen-funktion	$o(a)=\begin{cases}0 & \text{für } a<\Theta\\ 1 & \text{für } a\geq\Theta\end{cases}$	Skalarprodukt	Identität $a = net$
ADALINE	{-1, 1}	Signum-funktion	$o(a)=\begin{cases}-1 & \text{für } a<\Theta\\ 1 & \text{für } a\geq\Theta\end{cases}$	Skalarprodukt	Identität
Linear	(-∞, +∞)	Linear	$o(a)=\sigma(a-\Theta)$	Skalarprodukt	Identität
Fermi	(0, 1)	Einfache Fermi-funktion	$o(a)=\frac{1}{1+e^{-a}}$	Skalarprodukt	Identität
Tanh	(-1, 1)	Tanh	$o(a)=\frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$	Skalarprodukt	Identität
Hopfield	{0, 1} oder {-1, 1}	Identität	$o(t+1)=\begin{cases}0 & \text{für } net<\Theta\\ o(t) & \text{für } net=\Theta\\ 1 & \text{für } net>\Theta\end{cases}$	Skalarprodukt	Hopfield
Boltzmann	{0, 1}	Boltzmann	$P(o=1)=\frac{1}{1+e^{-(a-\Theta)/T}}$	Skalarprodukt	Identität

Tabelle 2-1: Tabelle verschiedener Standard-Neuronentypen

Diese erhebt allerdings keinen Anspruch auf Vollständigkeit, sie zeigt lediglich die laut [Hoffmann] am häufigsten benutzten Typen (in gekürzter Form).

on-Neuron

(auch *Bias-Neuron*)

In den meisten Netzwerkmodellen besitzen die verwendeten Neuronen einen Schwellenwert. Dieser gibt an, wann ein Neuron „feuert", und zwar wenn die Aktivität des Neurons über die Schwelle ansteigt. Im biologischen Vorbild ist der Schwellenwert durch die Reizschwelle gegeben, die überschritten werden muss damit die Nervenzelle feuert.

Im künstlichen neuronalen Netz ist der Schwellenwert entweder Bestandteil des Neurons selber (als Parameter Θ in der Aktivierungs- oder Ausgabefunktion), oder er wird durch ein sogenanntes zusätzliches *on-Neuron* realisiert. Dieses *on-Neuron* hat stets den konstanten Ausgang 1 und wird mit jedem Neuron der entsprechenden Schicht verbunden. So erhält jedes Neuron einen zusätzlichen Eingang, an dem immer ein konstanter Wert anliegt, der mit in die Berechnung des effektiven Eingangs eingeht.

Durch eine geeignete Wahl der Gewichte dieser Verbindungen kann in den Gleichungen zur Berechnung der Aktivität die mit einbezogene Schwelle zum Verschwinden gebracht werden. Das negative Gewicht $-w_{ij}$ vom *on-Neuron* zum Neuron i entspricht dann genau dem Schwellenwert Θ.

Die Netzeingabe enthält dann also schon den Schwellenwert und die Aktivierungsfunktion kann ohne diesen rechnen. Aus

$$a_i(t+1) = f_{act}(a_i(t), net_i(t), \Theta_i)$$

wird dann:

$$a_i(t+1) = f_{act}(a_i(t), net_i(t)),$$

da der Schwellenwert mit in die Berechnung des Eingangs eingeht:

$$net = \left(\sum_{j=1}^{n} w_j e_j \right) - \Theta_i .$$

2.2 Aufbau eines Netzes

2.2.1 Arten von Netzwerken

Die Verbindungen der einzelnen Neuronen werden üblicherweise zu einem *Verbindungsnetzwerk* zusammengefasst, das in Matrixschreibweise angegeben wird. Diese sogenannte *Gewichtsmatrix* enthält als Elemente die Gewichte der Verbindung zwischen den entsprechenden Neuronen.

Für das Gewicht zwischen zwei einzelnen Neuronen gibt es unterschiedliche Bezeichnungen. Die Schreibweise w_{ij} kann zum einen das Gewicht vom Neuron i zum Neuron j bedeuten, in anderer Literatur ist die Bedeutung genau umgekehrt. In dieser Arbeit wird die Bezeichnung aus [Hoffmann] verwendet, w_{ij} meint somit das Gewicht des Neurons i am Eingang j (der Verlauf ist also von j nach i). Das aktuell betrachtete Neuron wird mit dem Index i gekennzeichnet.

In der Matrixschreibweise hat ein Eintrag w_{ij} in der Gewichtsmatrix W folgende Bedeutung:

- $w_{ij} = 0$ es existiert keine Verbindung zwischen Neuron j und i,
- $w_{ij} < 0$ Neuron j hemmt seinen Nachfolger i durch ein Gewicht mit angegebener Stärke,
- $w_{ij} > 0$ Neuron j regt seinen Nachfolger i durch ein Gewicht mit angegebener Stärke an.

Die einzelnen Neuronen werden zu bestimmten *Ebenen* (auch als *Schichten* bezeichnet) zusammengefasst. Dies dient einerseits einem besseren Überblick, andererseits kann dadurch die Funktion des Netzes besser gegliedert werden. Jede Ebene von Neuronen kann eine unterschiedliche Aufgabe realisieren, z.B. dient eine Eingabeschicht nur zur Weiterleitung der Netzeingabe an die erste verarbeitende Neuronenschicht.

Im Groben unterteilt man die neuronalen Netze in *Netze ohne Rückkopplung* und *Netze mit Rückkopplung*. Diese zwei Gruppen lassen sich dann noch genauer unterteilen. Ein Überblick mit verschiedenen Netztopologie-Beispielen ist in Abbildung 2-3 zu finden.

Netze ohne Rückkopplung

Bei dieser Art von Netzwerken (auch als *feedforward-Netze* bezeichnet) existieren nur vorwärtsgerichtete Pfade, zu Neuronen in nachfolgenden Ebenen. Es gibt keinen Pfad, der von einem Neuron direkt oder über zwischengeschaltete Neuronen wieder zurückführt. Es handelt sich mathematisch gesehen also um einen azyklischen Graph. Daher ist in der Gewichtsmatrix nur die obere oder untere (je nach Definition) Dreiecksmatrix mit Werten ungleich 0 belegt.

- Ebenenweise verbundene feedforward-Netze:

  Es gibt nur Verbindungen von einer Schicht zur nächsten, Verbindungen die Ebenen überspringen gibt es nicht (siehe Abbildung 2-3a).

- Allgemeine feedforward-Netze:

  (mit *shortcut connections*)

  Neben den Verbindungen von einer Schicht zur nächsten gibt es hier auch Verbindungen die Ebenen überspringen. Diese Verbindungen, die von einem Neuron der Ebene k zu einem der Ebene $k + i$ mit $i > 1$ verlaufen, werden als *shortcut connections* bezeichnet (siehe Abbildung 2-3b).

Netze mit Rückkopplungen

Neben den vorwärtsgerichteten Verbindungen existieren bei diesen Netzen auch Verbindungen zu Neuronen vorgeschalteter Schichten oder der selben Schicht. In der Gewichtsmatrix existieren dadurch Einträge ungleich 0 in beiden Matrixhälften und in einigen Fällen in der Hauptdiagonale.

- Netze mit direkten Rückkopplungen:

  (*direct feedback*)

  Bei diesen Netzen besitzen Neuronen Rückkopplungen zu sich selbst, d.h. der Ausgang eines Neurons wird zu einem Eingang des selben Neurons direkt zurückgeführt (siehe Abbildung 2-3c). Dadurch können sich die Neuronen selbst anregen bzw. hemmen und haben somit die Möglichkeit Grenzzustände ihrer Aktivierung anzunehmen.

- Netze mit indirekten Rückkopplungen:

  (*indirect feedback*)

  Bei diesen Netzen gibt es Rückkopplungen zu Neuronen „niederer“, also vorgeschalteter Ebenen (siehe Abbildung 2-3d). Die rückgeführten Verbindungen können dabei auch einzelne Schichten überspringen, ähnlich wie bei den *shortcut connections* der vorwärtsgekoppelten Netze. Diese Art der Rückkopplung ist z.B. hilfreich, will man eine Aufmerksamkeitssteuerung auf bestimmte Bereiche der Eingabeneuronen lenken.

- Netze mit Rückkopplungen innerhalb einer Schicht:

  (*lateral feedback*)

  Die Neuronen einer Schicht sind bei diesen Netzen untereinander vollständig verbunden (siehe Abbildung 2-3e). Eine solche Schicht von Neuronen kann z.B. als *winner-takes-all-Schicht* fungieren. Dazu erhält jedes Neuron hemmende Verbindungen zu den anderen Neuronen der selben Schicht (und manchmal noch eine aktivierende Verbindung zu sich selbst). Das Neuron mit der stärksten Aktivierung hemmt dann die anderen Neuronen und wird somit der „Gewinner“.

- Vollständig verbundene Netze:

  Bei diesen Netzen gibt es Verbindungen zwischen allen Neuronen des Netzes. Bei einigen Netzen (z.B. dem Hopfield-Netz aus Abschnitt 3.6) gibt es dabei noch spezielle Anforderungen an die Gewichtsmatrix: sie muss symmetrisch sein und die Hauptdiagonale darf nur Nullen enthalten. Ein solches Netzes ist in Abbildung 2-3f dargestellt.

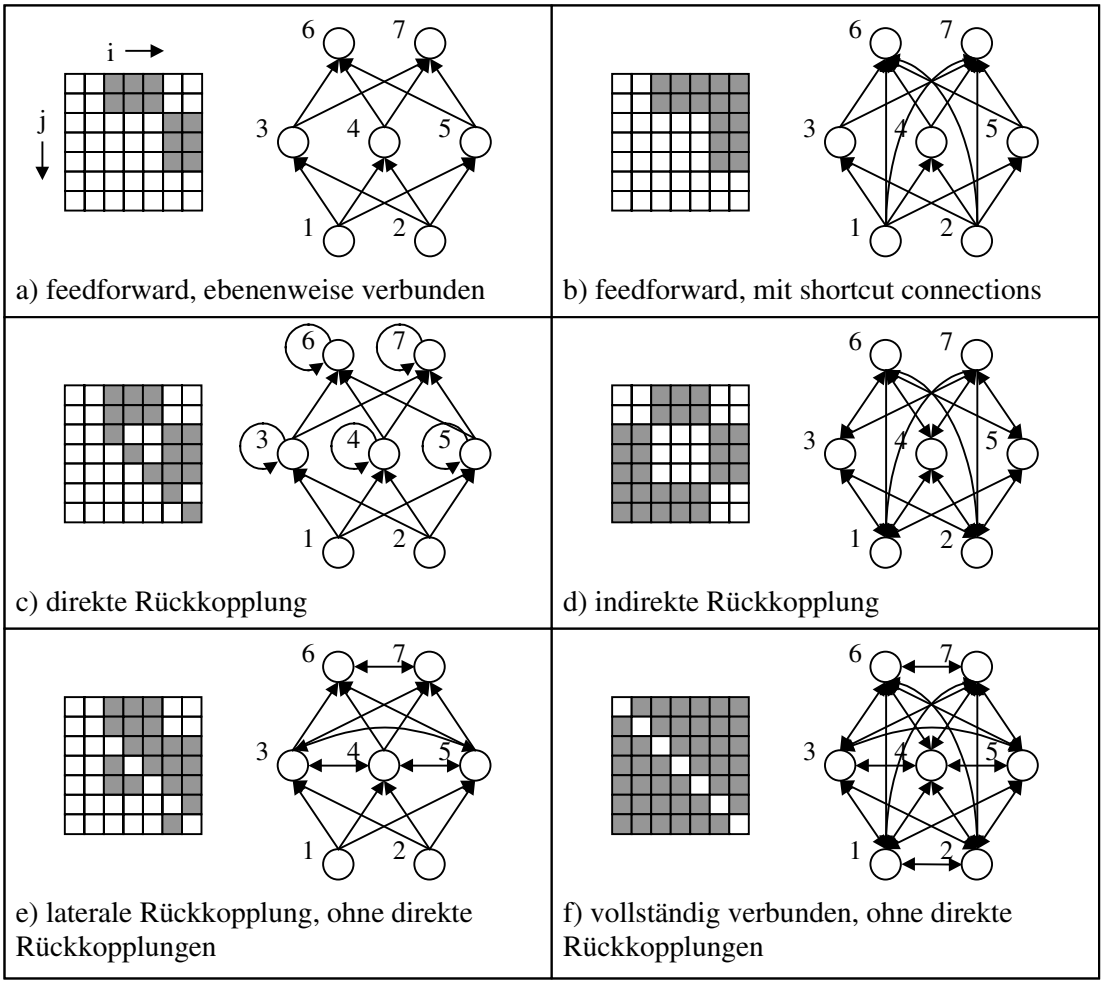


Abbildung 2-3: Beispiele von Netzwerktopologien und ihren Gewichtsmatrizen

Aufgrund der unterschiedlichen Bezeichnung von w_{ij} sind bei den Matrizen die Zeilen und die Spalten vertauscht zu interpretieren (im Gegensatz zur Originalabbildung in [Zell] S.79). Im ersten Segment wird die korrekte Bezeichnung zusätzlich mit angegeben.

2.3 Reproduktion

Bei der Reproduktion werden an die Netzeingänge Eingabevektoren angelegt und daraus anhand der Kennwerte des Netzes der Ausgabevektor berechnet. Die Kennwerte des Netzes (Gewichte der Verbindungen, Anzahl und Verteilung der Neuronen, Funktionen der Neuronen usw.) ändern sich dabei im Gegensatz zum Lernvorgang nicht. Der Lernvorgang sollte schon vor der Reproduktion stattgefunden haben, da diese sonst wenig Sinn ergibt.

Vorwärtsgekoppelte Netze

Besonders einfach ist die Reproduktion bei vorwärtsgekoppelten Netzen. Die Netzeingänge werden angelegt und daraus die Werte der Eingangsneuronen berechnet. Die Ausgänge dieser werden zur Berechnung der Neuronen der nächsten Schicht herangezogen. Das setzt sich fort, bis man am Ausgang des Netzes den Ausgangsvektor erhält. Berechnet man beliebige Neuronen noch einmal, so ändert sich nichts mehr an ihrer Ausgabe und damit an der Netzausgabe. Natürlich müssen die Eingangswerte während des gesamten Vorganges einer Reproduktion festgehalten werden, eine Änderung der Eingänge führt zu einer Änderung des Netzzustandes.

Rückgekoppelte Netze

Ein rückgekoppeltes Netzes verhält sich bei der Reproduktion komplizierter. Der wesentliche Unterschied zu vorwärtsgekoppelten Netzen liegt darin, dass sie in der Regel keinen stabilen Endzustand einnehmen. Um den Ausgang eines solchen Netzes zu berechnen, braucht man einen Startzustand, man muss also Eingangs- und Ausgangwerte der Neuronen vorgeben, da einige Eingänge eines Neurons Ausgänge nachfolgender Neuronen sind. Weiterhin hängt das Ergebnis noch von der Reihenfolge ab, in der man die Werte der einzelnen Neuronen berechnet.

Nähere Erläuterungen zur Reproduktion und eine Vorgehensweise zur Berechnung von rückgekoppelten Netzen findet man sehr gut beschrieben in [Hoffmann] S.43ff.

2.4 Lernregeln

Die Lernregel ist die interessanteste Komponente neuronaler Netze. Sie erlaubt es, dass sich ein Netz an eine bestimmte Aufgabe möglichst gut und vor allem selbständig anpasst. Dazu gibt es verschiedene Lernregeln (auch *Lernverfahren* genannt). Durch diese Regel lernt das neuronale Netz für eine vorgegebene Eingabe die gewünschte Ausgabe zu produzieren. Es gibt viele Arten, wie ein neuronales Netz lernen kann:

- Entwicklung neuer Verbindungen,
- Löschen bereits existierender Verbindungen,
- Modifikation der Stärke von Verbindungen,
- Modifikation des Schwellenwertes der Neuronen,
- Modifikation der Funktionen (Eingabe-, Aktivierungs-, Ausgabefunktion) der Neuronen,
- Entwicklung neuer Neuronen, oder
- Löschen bereits existierender Zellen.

Die Modifikation der Stärke der Verbindungen w_{ij} (somit also das Ändern des Wertes in der Gewichtsmatrix) ist die am häufigsten benutzte Methode. Natürlich sind auch Kombinationen aus einzelnen Verfahren möglich. Die Lernverfahren kann man in drei Gebiete gliedern, die sich darin unterscheiden, wie der Lernalgorithmus die Eingaben verarbeitet:

- Überwachtes Lernen:

  (*supervised learning*)

  Das Wesentliche beim Überwachten Lernen ist, dass man das gewünschte bzw. geforderte Verhalten des Netzes genau kennt. Zu jedem Eingabemuster wird das korrekte bzw. beste Ausgabemuster mit angegeben. Zum Muster am Netzeingang soll am Ausgang das zugehörige Ausgangsmuster (*Sollmuster*) erscheinen. Durch die Kenntnis des Paares Eingangsmuster/Ausgangsmuster lässt sich ein Fehler oder ein Fehlverhalten relativ leicht abschätzen. Nach dem Lernen soll das Netz in der Lage sein, selbständig ein korrektes oder ähnliches Ausgabemuster zu einem vorgegebenen Eingabemuster zu assoziieren.

- Bestärkendes Lernen:

  (*reinforcement learning*)

  Beim Bestärkenden Lernen wird nicht die korrekte bzw. erwünschte Ausgabe mit angegeben, sondern nur, ob die vom Netz gelieferte Ausgabe richtig oder falsch ist. Eventuell kann auch noch der Grad des Fehlers oder der Korrektheit mit angegeben werden. Das Lernverfahren muss also selbst die richtige Ausgabe zur Eingabe finden.

- Unüberwachtes Lernen:

  (*unsupervised learning*, *self-organized learning*)

  Beim Unüberwachten Lernen besteht die Lernmenge nur aus Eingabemustern, es existieren keine Informationen über die Korrektheit oder Falschheit der erzeugten Ausgangsmuster und somit auch kein „externer Lehrer“. Der Algorithmus versucht stattdessen, die Eingabevektoren in Gruppen ähnlicher Vektoren zu unterteilen (zu *clustern*).

Zwei grundlegende Lernregeln werden im Folgenden kurz vorgestellt. Später wird an geeigneter Stellen noch einmal genauer darauf eingegangen:

2.4.1 Hebbsche Lernregel

Dies ist eine besonders einfache überwachte Lernregel. Sie wurde in ihrer ersten Form bereits 1949 von Donald O. Hebb formuliert und bildet die Grundlage für die meisten komplizierteren Lernregeln. Die Regel basiert auf einer Vermutung von Hebb, dass die Synapse (Verbindung zweier Zellen im Gehirn) zweier benachbarter Nervenzellen stärker wird, wenn beide Zellen gleichzeitig feuern.

Die einfache Hebbsche Lernregel lautet:

> Wenn Neuron *i* eine Eingabe von Neuron *j* erhält und beide gleichzeitig stark aktiviert sind, dann verstärke das Gewicht w_{ij} (also die Stärke der Verbindung von Neuron *j* zu Neuron *i*).

Die Größe der Verstärkung ist eine vorgegebene reelle Zahl und heißt *Lernrate* η. In der mathematischen Form sieht dies folgendermaßen aus:

$$\delta w_{ij} = \eta o_i e_j .$$

Dabei ist δw_{ij} die Änderung des Gewichts, η die Lernrate, o_i die Ausgabe des Neurons und e_j der Eingang vom Vorgängerneuron. Bei einschichtigen Netzen stimmen die Neuroneneingänge mit den Netzeingängen und die Neuronenausgänge mit den Netzausgängen und damit mit dem Sollwert überein:

$$\delta w_{ij} = \eta S_i E_j .$$

Diese Lernregel kann nur auf Ausgangsneuronen oder einschichtige Netze angewendet werden, für Netze mit verborgen Neuronen ist sie nicht brauchbar.

2.4.2 Delta-Lernregel

Die Delta-Lernregel (auch *Widrow-Hoff-Regel*) berücksichtigt die Abweichung zwischen Soll- und Istwert. Der Ausgang eines Neurons wird umso größer, je größer net_i ist. Die genaue Form der Aktivierungs- und Ausgangsfunktion ist dabei ohne Bedeutung.

Ist der berechnete Ausgang (wir betrachten Ausgangsneuronen, also ist der Ausgang des Neurons gleich dem Netzausgang) zu klein ($O < S$), so muss net_i vergrößert werden. Die Vergrößerung muss umso stärker ausfallen, je größer der Unterschied zwischen O und S ist, bei $O = S$ brauchen die Gewichte natürlich nicht geändert werden. Liefert ein Eingang E_j keinen Beitrag zum Neuronenausgang ($E_j = 0$), so bringt es nichts, wenn das zugehörige Gewicht w_j vergrößert wird, im Gegensatz zu $E_j > 0$. Daher geht E_j als Faktor in die Gleichung ein. Weiterhin wird wieder eine Lernrate η eingeführt, und man erhält als Delta-Lernregel:

$$\delta w_{ij} = \eta(S_i - O_i)E_j .$$

Diese Lernregel kann ebenfalls nur für einschichtige Netze verwendet werden. Viele andere Lernregeln bauen aber auf ihr auf, wie z.B. die Backpropagation-Lernregel. Sie ist auch auf Netze mit mehr als einer Schicht trainierbarer Gewichte anwendbar (siehe Abschnitt 3.7).

Eine nähere Erläuterung zur Herleitung dieser beiden grundlegenden Lernregeln findet der Leser bei [Zell] oder [Hoffmann].

2.4.3 Stabilitäts-Plastizitäts-Dilemma

Hat ein neuronales Netz schon Muster gelernt, so verlangt man gewöhnlich von ihm, dass es später ohne Probleme noch weitere Muster hinzulernen kann, es soll also *plastisch* sein. Beim Hinzulernen neuer Muster dürfen aber keine Konflikte auftreten, so dass bereits bekannte Muster wieder „verlernt“ werden, das Netz soll also *stabil* sein.

Das Problem beide Eigenschaften so gut wie möglich zu vereinen, bezeichnet man als *Stabilitäts-Plastizitäts-Dilemma*. Eine Ausführung zu diesem Thema findet man in der Arbeit von [Protzel].

3 Typen überwacht-lernender Netze

3.1 Allgemeiner Muster-Assoziator

3.1.1 Einleitung

Ein Muster-Assoziator gehört zur Gruppe der einfach überwacht-lernenden Netze. Von der allgemeinen Struktur wurden für verschiedene Anwendungsgebiete speziellere Formen von Muster-Assoziatoren abgeleitet.

3.1.2 Aufbau

Ein einfacher Muster-Assoziator ist ein einschichtiges Netz mit N_E Eingängen und N_O Ausgängen. Jedes Neuron ist mit jedem Netzeingang verbunden und jedes Neuron gibt seinen Ausgang lediglich an den Netzausgang (Neuronenausgang ist gleich dem Netzausgang) weiter.

Somit hat jedes Neuron die selbe Anzahl von Eingängen (nämlich gleich N_E) und die Anzahl der Netzausgänge N_O ist gleich der Anzahl der Neuronen. Da die Eingänge der Neuronen mit den Netzeingängen verbunden sind, braucht man zwischen den Netzeingängen E_j und den einzelnen Neuroneneingängen e_i nicht zu unterscheiden.

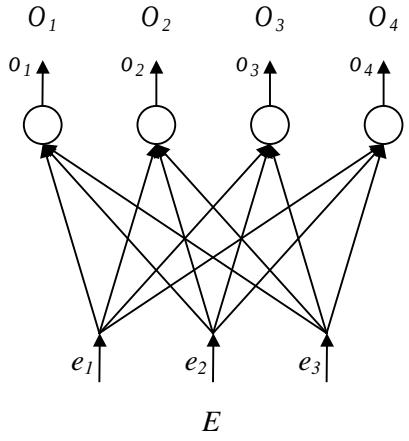


Abbildung 3-1: Aufbau eines allgemeinen Muster-Assoziators

Der Muster-Assoziator gehört zur Klasse der vorwärtsgekoppelten Netze (feedforward-Netze).

3.1.3 Funktionen zur Berechnung

Der effektive Eingang eines Neurons *i* ist somit gegeben durch:

$$net_i = \sum_{j=1}^{n} w_{ij} E_j, \quad i = 1 \ldots N .$$

Die Aktivität der Neuronen spielt beim Muster-Assoziator keine direkte Rolle. So kann man die Identität als Aktivierungsfunktion benutzen, also gilt:

$$a_i = net_i .$$

Die Ausgabefunktion ist beliebig. Dabei kann jedes Neuron eine andere Ausgabefunktion haben, was in der nachfolgenden Gleichung durch den Index *i* ausgedrückt wird. Es gilt die folgende Beziehung:

$$O_i = o_i\left(\sum_{j=1}^{n} w_{ij} E_j \right), \quad i = 1 \ldots N .$$

3.1.4 Lernverfahren

Ein Muster-Assoziator ergibt für jedes angelegte Eingangsmuster ein zugehöriges Ausgangsmuster, er stellt also eine Assoziation zwischen Musterpaaren her (*heteroassoziatives Netz* (siehe [Hoffmann] S.142f.)). Gibt man einen Satz von Musterpaaren (E_j^μ, S_i^μ) vor, besteht die Aufgabe darin, die Gewichte so zu bestimmen, dass für jedes μ das Eingangsmuster E_j^μ zum Ausgangsmuster S_i^μ (entspricht also dem Sollwert) führt.

Da keine Rückkopplung vorhanden ist, sind die einzelnen Neuronen voneinander unabhängig. Ein Muster-Assoziator aus einem Neuron verhält sich daher im Prinzip nicht anders als einer aus mehreren.

Als Lernverfahren wird die Hebbsche Lernregel bzw. die Delta-Lernregel angewandt.

Hebbsche Lernregel

Die Schwellenwerte der Neuronen werden zur Vereinfachung auf 0 gesetzt, also $\Theta_i = 0$. Die Anwendung der Hebbschen Lernregel auf die Lernaufgabe ergibt folgende Gleichung zur Anpassung der Gewichte:

$$w_{ij} = \eta \sum_{\mu=1}^{p} S_i^\mu E_j^\mu \ .$$

Beim Muster-Assoziator ist die Hebbsche Lernregel nur dann sinnvoll, wenn das Eingangsmuster ein Orthonormalsystem bildet.

Die Gewichte können alle in einem einzigen Schritt berechnet werden.

Delta-Lernregel

Die Delta-Lernregel überwindet die Einschränkungen der Hebbschen Lernregel (Eingangsmuster muss Orthonormalsystem sein). Die Lernregel lautet hier:

$$\delta w_{ij} = \eta (S_i - O_i) E_j \ .$$

Die Gewichte werden anfangs üblicherweise auf 0 gesetzt, danach werden die Eingabemuster E_j^μ der Reihe nach einzeln angelegt und das dazugehörige Ausgabemuster O_i berechnet und die Gewichte gemäß der Lernregel verändert. Nach Abarbeitung aller zu lernenden Musterpaare ist ein Lernschritt beendet.

Die Berechnung der Gewichte erfolgt bei der Delta-Lernregel iterativ, dabei ist meist eine größere Anzahl von Lernschritten erforderlich.

3.1.5 Anwendung, Grenzen und Probleme

Ein Muster-Assoziator wird als heteroassoziativer Speicher eingesetzt (siehe [Hoffmann] S.140). Wenn jeder Ausgang nur zwei verschiedene Werte annehmen kann (was bei einer nichtlinearen Ausgangsfunktion der Fall ist), so teilt ein Neuron das Eingangsmuster in zwei Klassen ein, also wirkt es als Klassifikator. Jedes Neuron klassifiziert dabei unter anderen Gesichtspunkten.

Ein Muster-Assoziator kann nur solche Musterpaare lernen, deren Eingangsmuster linear unabhängig sind.

3.2 Perzeptron

3.2.1 Einleitung

Das Perzeptron ist vom Typ her ein spezieller Muster-Assoziator. Es handelt sich beim Perzeptron jedoch nicht nur um eine einzige Art von Netz, sondern um eine ganze Familie verwandter Modelle. Entwickelt wurden diese Ende der 50iger Jahre von Frank Rosenblatt.

3.2.2 Aufbau

In [Hoffmann] wird das Perzeptron als ein dreischichtiges, in [Zell] als ein einschichtiges vorwärtsgekoppeltes Netz angegeben. In anderer Literatur (z.B. bei [Sauer]) wird das ursprüngliche Perzeptron sogar nur als ein einzelnes Neuron beschrieben. Ich möchte in diesem Abschnitt die erste Variante (dreischichtiges Netz) beschreiben.

Die Neuronen der ersten Schicht (*Retina*) heißen *S-Zellen* (*Stimulus-Zellen*). Sie wird auch als Eingabeschicht bezeichnet. Ihre Aufgabe besteht darin, das Eingangsmuster *E* aufzunehmen und an die zweite Schicht weiterzuleiten. Demnach besteht diese erste Schicht aus Verteilungsneuronen. Sie haben auf die eigentliche Funktion des Netzes keinerlei Einfluss.

Die zweite Schicht wird als *Assoziationsschicht* bezeichnet, die Zellen als *A-Zellen* (*Assoziationszellen*). Die A-Zellen sind McCulloch-Pitts-Neuronen (siehe Tabelle 2-1).

Die dritte Schicht heißt *Perzeptron-Schicht* oder *Verarbeitungsschicht*. Sie repräsentiert das eigentliche Perzeptron. Diese Schicht kann als Muster-Assoziator (eventuell mit on-Neuron) angesehen werden. Die Neuronen dieser dritten Schicht heißen *R-Zellen* und sind ebenfalls McCulloch-Pitts-Neuronen. Sie stellen gleichzeitig die Ausgabeneuronen dar.

Die Verbindungen zwischen den S- und den A-Zellen sind feste Verbindungen. Sie werden am Anfang nach Zufallskriterien ausgewählt. Die Verbindungen von den A-Zellen zu den R-Zellen sind trainierbar, d.h. variabel. Dadurch handelt es sich beim Perzeptron eigentlich um ein einstufiges Netzwerk, weil es nur eine Ebene trainierbarer Gewichte gibt.

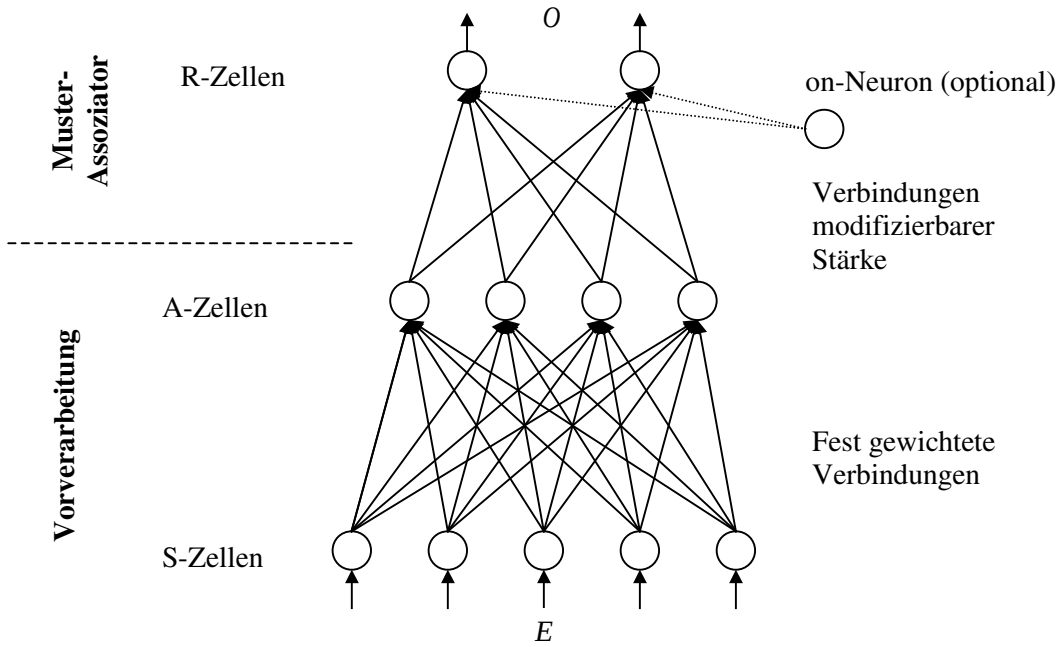


Abbildung 3-2: Aufbau eines Perzeptrons

Ein Perzeptron kann beliebig viele Ausgabeneuronen besitzen, ohne dass sich an dem Prinzip Grundlegendes ändert. Die Anzahl der Eingänge muss auch nicht der Anzahl der S-Zellen entsprechen.

3.2.3 Funktionen zur Berechnung

Es genügt beim Perzeptron, die Neuronen der Eingabeschicht als binäre Eingaben zu betrachten. Die A-Zellen und die R-Zellen, also die McCulloch-Pitts-Neuronen, werden dagegen wie folgt berechnet:

$$net_i = \sum_{j=0}^{n} w_{ij} o_j \ ,$$

$$o_i = a_i = \begin{cases} 0 & \textit{für } net_i < \Theta_i \\ 1 & \textit{für } net_i \geq \Theta_i \end{cases} .$$

Das Neuron summiert seine Eingaben auf und wendet auf diese Netzeingabe net_i eine binäre Schwellenwertfunktion (auch *Stufenfunktion* genannt, siehe Abbildung 2-2a) an. Die Ausgabe o_i ist 1, falls die Netzeingabe größer oder gleich dem Schwellenwert Θ_i ist, andernfalls 0.

3.2.4 Reproduktion

Durch die feste Verbindung der S-Zellen mit den A-Zellen wird das Eingabemuster auf eine Art „Zwischenmuster“ abgebildet. Dadurch wird neben der einfachen Weitergabe des Musters eine Art von „Vorverarbeitung“ erreicht.

Der Eingangsvektor *E* wird durch die dritte Schicht verarbeitet, die damit das eigentliche Perzeptron repräsentiert. Durch die Ausgabefunktion kann der Ausgang der Neuronen nur einen von zwei Zuständen annehmen. Da dies auch auf die Ausgabeschicht (repräsentiert durch die R-Zellen) zutrifft, berechnet ein Perzeptron folglich ein logisches Ergebnis seiner Eingabewerte, im Sinne von *wahr* oder *falsch*.

3.2.5 Lernverfahren

Der Lernvorgang erfolgt nur bei den R-Zellen, da nur die Verbindungen zwischen den A-Zellen und den R-Zellen trainierbar sind. Das Lernverfahren basiert laut [Zell] auf der Hebbschen Lernregel:

- Wenn Ausgabe und erwartete Ausgabe übereinstimmen, führt das Netz keine Gewichtsänderung bei einem Ausgabeneuron durch.
- Sind sie unterschiedlich und hat die Ausgabezelle *i* den Wert 0, dann wird das Gewicht zwischen *j* und *i* um die Ausgabe von *j* (0 oder 1) erhöht.
- Ist die Ausgabe falsch und hat den Wert 1, dann wird dann wird das Gewicht zwischen *j* und *i* um die Ausgabe von *j* (0 oder 1) verringert.

Eine Gewichtsänderung erfolgt also nur, wenn die Ausgabe o_i einen falschen Wert hat und die Ausgabe o_j des Vorgängerneurons den Wert 1 hat (hat o_j den Ausgabewert 0 wird um 0 erhöht bzw. verringert, also passiert nichts).

Durch das on-Neuron werden die Schwellenwerte in den Lernvorgang mit einbezogen (Erläuterungen dazu findet man Abschnitt 2.1.5). Bei [Hoffmann] wird als Lernregel für das Perzeptron die Delta-Lernregel angegeben, aufgrund des Wertebereichs der Ausgaben $\{0,1\}$ ist diese aber äquivalent zur der hier beschriebenen Notation.

3.2.6 Anwendung, Grenzen und Probleme

Das Perzeptron kann für Klassifikationsaufgaben und als heteroassoziativer Speicher eingesetzt werden. Es lernt im Prinzip wie jeder Muster-Assoziator Musterpaare, bestehend aus Eingangsmuster und Ausgangsmuster.

Das einstufige Perzeptron ist im nur für sehr einfache Aufgaben mit einer geringen Zahl von Eingaben pro Zelle geeignet (siehe [Zell] S.101f.). Als Erweiterung dazu gibt es noch zwei- und dreistufige Perzeptrons, die um einiges mächtiger sind. Diese können dann auch konvexe Polygone berechnen. Dreistufige Perzeptrons können durch Überlagerung konvexer Polygone Mengen beliebiger Form repräsentieren (z.B. Verknüpfungen der Form *A and not B*).

3.3 ADALINE

3.3.1 Einleitung

ADALINE bedeutet *adaptive linear element* oder *adaptive linear neuron*. Das ADALINE ist ein spezieller Muster-Assoziator mit Bias-Neuron.

3.3.2 Aufbau

Das ADALINE ist ein Netz mit nur einer Schicht von Neuronen und einem zusätzlichen Bias-Neuron. Es gehört zur Klasse der vorwärtsgekoppelten Netze.

Die Neuronen der Schicht sind ADALINE-Neuronen (siehe Tabelle 2-1). Der Ausgang jedes Neurons ist zugleich Netzausgang. Also gibt es genauso viele Ausgänge des Netzes wie Neuronen.

Die Anzahl der Eingänge ist beliebig. Jeder Netzeingang wird dabei mit jedem Neuron verbunden. Der Ausgang des on-Neurons wird ebenfalls mit jedem Neuron verbunden. Diese Verbindungen haben den konstanten Wert 1. Dadurch erhält jedes Neuron neben dem Netzeingang einen zusätzlichen Eingang vom on-Neuron.

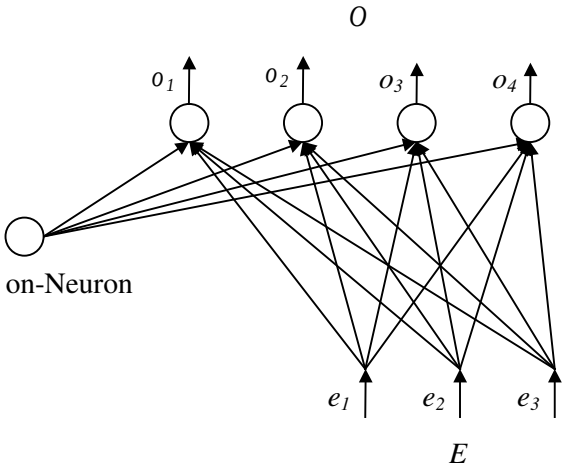


Abbildung 3-3: Aufbau eines ADALINE-Netzes

3.3.3 Funktionen zur Berechnung

Die Ein- und Ausgangswerte der ADALINE-Neuronen liegen in der Menge {-1, +1}. Die Berechnungsformel eines Neurons *i* lautet demnach:

$$net_i = \sum_{j=0}^{n} w_{ij} e_j ,$$

$$o_i = \begin{cases} -1 & \text{für } net_i < 0 \\ +1 & \text{für } net_i \geq 0 \end{cases}.$$

Manchmal wird auch $o(0) = -1$ gesetzt. Die Aktivität a_i eines Neurons wird mit der Eingabe net_i (ist in diesem Fall gleich der Netzeingabe) gleichgesetzt.

Die Ausgangsfunktion eines ADALINE-Neurons ist eine Signum-Funktion (siehe Tabelle 2-1). Diese entspricht der normalen Schwellenwertfunktion, mit dem Unterschied, dass der untere der beiden Werte nicht 0, sondern -1 ist.

3.3.4 Lernverfahren

Als Lernregel dient eine Variante der Delta-Lernregel. Statt des Ausgangs wird die Aktivität zur Bestimmung des Fehlers benutzt:

$$\delta w_{ij} = \eta(S_i - a_i)E_j .$$

Das ist günstig für die Stabilität des Netzes. Ein Neuron, dessen Aktivität in unmittelbarer Nähe der Schwelle liegt, kann noch lernen, auch wenn der Ausgang wegen der Signum-Funktion bereits korrekt ist.

Die Variablen haben dabei folgende Bedeutung:

- Lernrate η
- Aktivität a_i
- Sollwert S_i
- Eingänge E_j

Für die Lernregel gibt es noch eine Reihe anderer Varianten (siehe dazu [Hoffmann] S.80), die aber hier nicht weiter erläutert werden.

3.3.5 Anwendung, Grenzen und Probleme

Das ADALINE-Netz kann als heteroassoziativer Speicher eingesetzt werden (siehe [Hoffmann] S.142f.). Es lernt im Prinzip wie jeder Muster-Assoziator Musterpaare. Wird ein gelerntes Eingangsmuster angelegt, so soll am Ausgang das zugehörige (assoziierte) Ausgangsmuster erscheinen. Wird ein nicht gelerntes Eingangsmuster angelegt, so wird irgendein Muster am Ausgang angezeigt.

Ein ADALINE-Netz kann nur linear teilen (siehe dazu [Hoffmann] S.76).

3.4 MADALINE

3.4.1 Einleitung

MADALINE bedeutet *multiple ADALINE*. Das MADALINE wurde entwickelt, um das Problem der linearen Teilbarkeit beim ADALINE zu überwinden (ein ADALINE-Netz kann nur linear teilen).

3.4.2 Aufbau

Das MADALINE ist ein zweischichtiges, vorwärtsgekoppeltes Netz.

Die erste Schicht (*ADALINE-Schicht*) stimmt mit einem ADALINE-Netz aus Abschnitt 3.3 überein. Daran schließt sich eine weitere Schicht, die *MADALINE-Schicht*, an. Die Neuronen beider Schichten sind ebenfalls ADALINE-Neuronen.

Jeder Ausgang eines Neurons der ADALINE-Schicht führt genau zu einem Neuron der MADALINE-Schicht. Verzweigungen der Ausgänge der Neuronen der ersten Schicht sind also nicht zugelassen. Die Gewichte dieser Verbindungen haben alle den festen Wert 1.

Die Ausgänge der Neuronen der MADALINE-Schicht sind zugleich die Netzausgänge. Die Anzahl der Netzeingänge ist beliebig. Jeder Netzeingang wird dabei mit jedem Neuron der ersten Schicht verbunden.

Der Ausgang des on-Neurons wird ebenfalls mit jedem Neuron der ersten Schicht verbunden. Es hat den konstanten Ausgang 1.

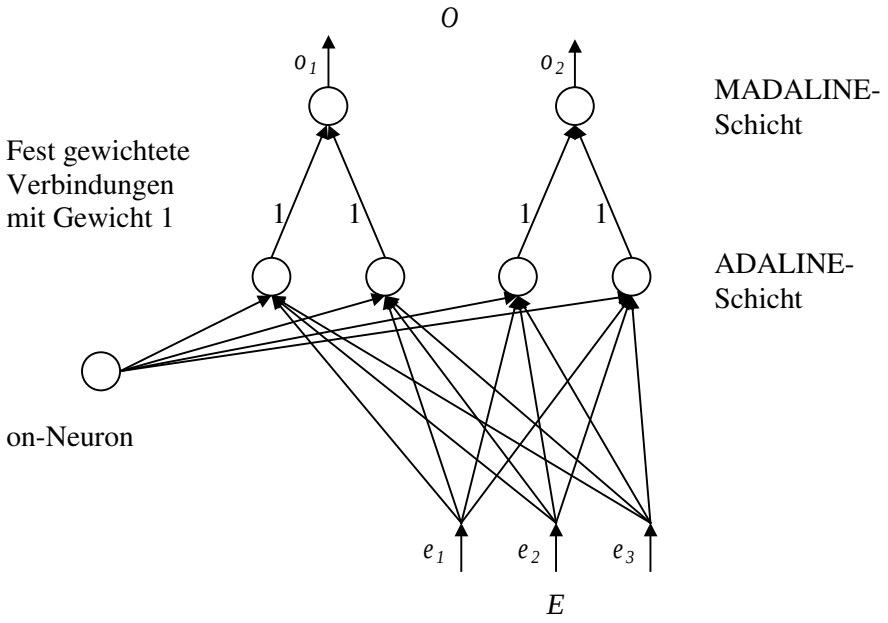


Abbildung 3-4: Aufbau eines MADALINE-Netzes

3.4.3 Funktionen zur Berechnung

Die Ein- und Ausgangswerte der Neuronen der ADALINE-Schicht und die Ausgänge der MADALINE-Schicht liegen in der Menge {-1, +1}. Die Verbindungen zwischen der ADALINE und der MADALINE-Schicht haben alle das feste Gewicht 1. Ein Eingang eines Neurons wird als *aktiv* bezeichnet, wenn sein Gewicht +1 ist, als *inaktiv*, wenn sein Gewicht -1 ist.

Die Neuronen der ersten Schicht (ADALINE-Schicht) berechnen ihren Ausgang wie beim ADALINE-Netz beschrieben (siehe Abschnitt 3.3.3). Für die Neuronen der MADALINE-

Schicht gibt es drei verschiedene Transferfunktionen, die unter dem Begriff *Verdichtungsverfahren* zusammengefasst werden:

Mehrheitsverfahren

Der Ausgang o_i wird +1 gesetzt, wenn die Anzahl der aktiven Eingänge (+1) größer als die Zahl der inaktiven Eingänge (-1) ist. Da die Gewichte zwischen der ersten und zweiten Schicht alle fest den Wert 1 haben, berechnen sich die Ausgänge der Neuronen der MADALINE-Schicht durch das Vorzeichen der Ausgabe der ersten Schicht:

$$o_i = \operatorname{sgn}\left(\sum_j e_j\right) = \operatorname{sgn}\left(\sum_j w_{ij} e_j\right) = \operatorname{sgn}(net_i).$$

Einstimmigkeitsverfahren

Der Ausgang wird +1 gesetzt, wenn alle Eingänge aktiv sind. Dies entspricht einem logischen UND:

$$o_i = \min_j(e_j).$$

Singulärverfahren

Der Ausgang wird +1 gesetzt, wenn mindestens ein Eingang aktiv ist. Dies entspricht einem logischen ODER:

$$o_i = \max_j(e_j).$$

3.4.4 Lernverfahren

Gelernt werden die Gewichte der ADALINE-Schicht, dabei ist die Formel dieselbe wie beim ADALINE-Netz (siehe Abschnitt 3.3.4). Das wesentliche bei diesem Netz ist jedoch, **welche** Gewichte angepasst werden.

Wenn ein Ausgangsneuron die falsche Ausgabe liefert, so betrachtet man zunächst alle Neuronen der ADALINE-Schicht, die mit diesem Neuron verbunden sind. Unter denen wähle man diejenigen aus, die das falsche Vorzeichen haben. Nur bei dem Neuron, dessen Aktivität am nächsten bei 0 liegt, werden die Gewichte angepasst.

3.4.5 Anwendung, Grenzen und Probleme

Das MADALINE-Netz kann als heteroassoziativer Speicher eingesetzt werden. Es lernt im Prinzip wie jeder Muster-Assoziator Musterpaare. Wird ein gelerntes Eingangsmuster angelegt, so soll am Ausgang das zugehörige (assoziierte) Ausgangsmuster erscheinen. Wird ein nicht gelerntes Eingangsmuster angelegt, so wird irgendein Muster am Ausgang angezeigt.

Das Problem der linearen Teilbarkeit beim ADALINE-Netz wird bei diesem Netz überwunden.

3.5 Allgemeiner Auto-Assoziator

3.5.1 Einleitung

Ein Auto-Assoziator gehört zur Gruppe der überwacht lernenden, rückgekoppelten Netze. Im Gegensatz zum Muster-Assoziator speichert er einzelne Muster. Seine Struktur kann durch Angabe einer einzigen Zahl, etwa der Neuronenanzahl *N*, charakterisiert werden.

Neben dem allgemeinen Auto-Assoziator gibt es noch zwei spezielle Varianten, das BSB-Modell (*brain state in the box*) und das DMA-Modell (*distributed memory and amnesia*). Eine Beschreibung dieser findet man bei [Hoffmann] S.84f..

3.5.2 Aufbau

Ein einfacher Auto-Assoziator ist ein einschichtiges, vollständig verbundenes Netz. Es gehört zur Klasse der rückgekoppelten Netze. Die Anzahl der Netzeingänge ist gleich der Anzahl der Netzausgänge ($N_E = N_O = N$). Jedes Neuron ist mit genau einem Netzeingang verbunden und jedes Neuron gibt seinen Ausgang an einen Netzausgang und an die anderen Neuronen weiter (Rückkopplung).

Es ist hier zweckmäßig zwischen externen und internen Eingängen der Neuronen zu unterscheiden, mit den internen Eingängen sind dabei die Rückkopplungen gemeint. Gewöhnlich (vor allem bei kleinen Netzen) schließt man Selbstrückkopplung aus, also ist $w_{ii} = 0$. Der Neuronentyp bei diesen Netzen ist beliebig. Daher lassen sich auch keine bestimmten Berechnungsformeln wie bei anderen Netzen angeben.

Das Bild zeigt einen Auto-Assoziator mit drei Neuronen. Der Netzeingang E_j führt führt zum externen Eingang e_0 des Neurons *j* und der Ausgang des Neurons *i* führt zum Netzausgang O_i und den internen Eingängen e_i aller Neuronen.

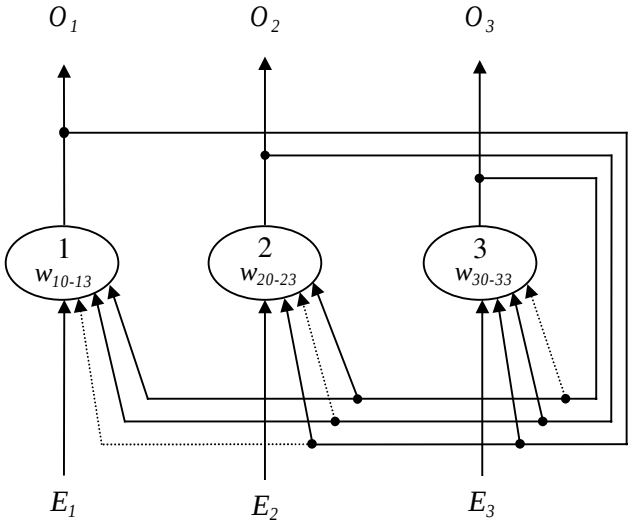


Abbildung 3-5: Aubau eines allgemeinen Auto-Assoziators

3.5.3 Reproduktion

Wegen der Rückkopplung ändert sich der Ausgang bei jedem Reproduktionsschritt. In der Regel kann man erwarten, daß asymptotisch (bei binären Neuronen exakt) ein Endzustand erreicht wird. Die Reproduktion muss dabei in einem definierten Startzustand beginnen.

Ist die Transferfunktion unbegrenzt (z.B. bei linearen Auto-Assoziatoren mit linearer Ausgangsfunktion (siehe Abbildung 2-2c)), besteht die Gefahr, dass sich die Ausgänge beliebig aufsummieren. In diesem Fall sollte eine andere Aktivierungsfunktion angewandt werden.

3.5.4 Lernverfahren

Zum Lernen gibt man dem Auto-Assoziator einen Satz von Mustern vor:

$$S_j^{\mu}, \quad \mu = 1 \ldots p .$$

Es wird dabei entweder die Hebbsche Lernregel oder die Delta-Regel verwendet:

Hebbsche Lernregel

Da die Sollwerte mit den Eingängen übereinstimmen, nimmt die Hebbsche Lernregel folgende Form an:

$$\delta w_{ij} = \eta E_i E_j, \quad i, j = 1 \ldots N .$$

Die externen Gewichte werden nicht gelernt, sondern auf einen festen Wert (gewöhnlich 1) gesetzt. Waren die Gewichte am Anfang gleich 0, so ist die Gewichtsmatrix symmetrisch.

Delta-Lernregel

Die Standardform der Delta-Lernregel lautet:

$$\delta w_{ij} = \eta (S_i - O_i) E_j .$$

Der Faktor (S_i - O_i) bezeichnet dabei den Lernfehler.

Beim Auto-Assoziator ist es günstig, als Lernfehler die Differenz zwischen dem externen und dem internen Anteil des effektiven Eingangs anzusetzen:

$$\delta w_{ij} = \eta \left(w_{i0} E_i - \sum_{k=1}^{N} w_{ik} O_k \right) E_j, \quad i, j = 1 \ldots N .$$

3.5.5 Anwendung, Grenzen und Probleme

Wird dem Netz eines der gelernten Muster angeboten, so soll am Ausgang dasselbe Muster erscheinen. Der Auto-Assoziator assoziiert also die Muster mit sich selbst. Es handelt sich also um einen autoassoziativen Speicher (siehe [Hoffmann] S.141f.), dessen Aufgabe es ist, einzelne Muster zu speichern.

Wird ein Teil eines gespeicherten Musters oder ein ähnliches Muster angelegt, so liefert es am Ausgang im Idealfall das vollständige gespeicherte Muster oder dasjenige, das dem Eingangsmuster am ähnlichsten ist. Legt man ein beliebiges Muster an, so ergibt die Reproduktion im Idealfall eines der gelernten Muster. Der Auto-Assoziator eignet sich daher für folgende Aufgaben:

Ergänzung

Ein Teilmuster eines gelernten Musters (der andere Teil ist z.B. 0 gesetzt) wird angelegt. Am Ausgang erscheint das vollständige Muster.

Rekonstruktion

Eine „verstümmelte“ Variante wird angelegt. Am Ausgang erscheint das vollständige Muster.

Generalisierung

Ein nicht gelerntes Muster (ist einem gelernten Muster ähnlich) wird angelegt. Am Ausgang erscheint das Muster, das dem angelegten am ähnlichsten ist.

3.6 Hopfield-Netz

3.6.1 Einleitung

Hopfield-Netze gehen auf den amerikanischen Physiker John Hopfield zurück. Sie sind im Wesentlichen aufgebaut wie ein Auto-Assoziator, weisen jedoch im Detail einige Unterschiede auf.

3.6.2 Aufbau

Das Hopfield-Netz gehört zur Klasse der vollständig verbundenen Netze. Es besteht aus einer einzigen Schicht von Neuronen, die sowohl als Eingang als auch als Ausgang des Netzes fungieren. Seine Struktur kann also auch durch die Angabe einer einzigen Zahl (der Neuronenzahl $N_E = N_O = N$) charakterisiert werden. Die Neuronen sind miteinander vollständig verbunden, also gibt jedes Neuron seinen Ausgang sowohl an den Netzausgang als auch an die anderen Neuronen weiter (Rückkopplung).

Gegenüber dem Auto-Assoziator besitzt ein Hopfield-Netz folgende Unterschiede im Aufbau:

- Selbstrückkopplungen ist grundsätzlich ausgeschlossen:

$$w_{ii} = 0 .$$

- Die Gewichte müssen symmetrisch sein:

$$w_{ij} = w_{ji} .$$

Als Neuronen werden Hopfield-Neuronen (siehe Tabelle 2-1) verwendet. Die Ein- und Ausgänge liegen also in der Menge {0, 1} oder {-1, 1}, je nach Definition.

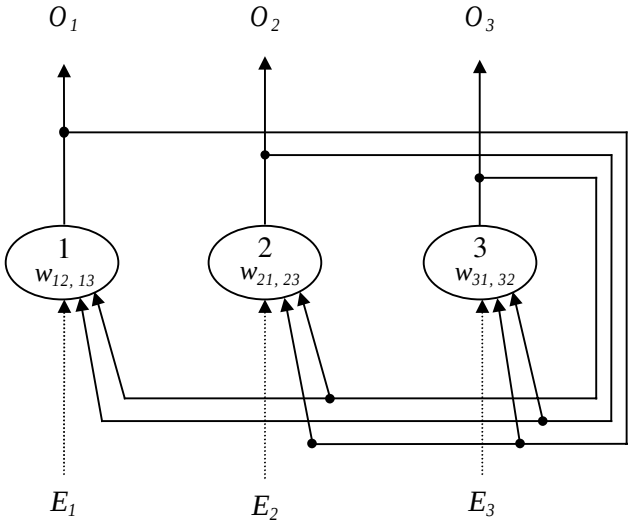


Abbildung 3-6: Aufbau eines Hopfield-Netzes

Die Verbindungen der Netzeingänge zu den Neuronen sind gestrichelt gezeichnet, da sie bei der Reproduktion nicht benötigt werden.

3.6.3 Reproduktion

Wie bei allen rückgekoppelten Netzen muss zunächst ein Startzustand erreicht werden. Dazu setzt man die Neuronenausgänge gleich den Netzeingängen. Das ist die einzige Phase in der Reproduktion, bei der die Netzeingänge eine direkte Rolle spielen.

Nun können die effektiven Eingänge berechnet werden:

$$net_i = \sum_{j=1}^{N} w_{ij} O_j \, .$$

Damit ist der Startzustand eindeutig definiert.

Ein einzelner Reproduktionsschritt geht von einem gegebenen Zustand aus und berechnet den Folgezustand mit folgender Funktion:

$$o_i(t+1) = \begin{cases} -1 & \text{für } net_i < \Theta_i \\ o_i(t) & \text{für } net_i = \Theta_i \\ +1 & \text{für } net_i > \Theta_i \end{cases} .$$

Der erste Reproduktionsschritt verwendet den Startzustand als Grundlage, alle Weiteren bauen auf den jeweils gerade berechneten Zustand auf. Dadurch, dass die Gewichtsmatrix symmetrisch ist, ist dieses Netz „stabil". Es wird also ein stabiler Endzustand bei der Berechnung erreicht (genauere Erläuterungen dazu findet man in [Zell] S.199f.).

Für die Reihenfolge der Berechnungen sind folgende drei Varianten üblich:

Stochastisch

Ein Neuron wird nach einem Zufallskriterium ausgewählt und berechnet. Alle anderen bleiben unberührt.

Der Reihe nach

Auswahlkriterium ist die natürliche Reihenfolge der Neuronen. Das aktuelle Neuron wird berechnet, während die anderen unberührt bleiben.

Synchron

Man berechnet die Ausgänge aller Neuronen. Bei dieser Variante werden von den Neuronen die Ausgangswerte des vorhergehenden Zustands benötigt.

3.6.4 Lernverfahren

Der Lernvorgang erfolgt mit Hilfe der Hebbschen Lernregel. Die Lernregel setzt hier Neuronen mit Werten aus der Menge {-1, 1} voraus. Die hier angegebene Lernregel funktioniert aber nur dann zufriedenstellend, wenn in jedem Muster die Werte -1 und +1 etwa gleich oft vorkommen. Andernfalls müssen auch die Schwellen der Neuronen gelernt werden. Die Lernaufgabe ist die selbe wie beim Auto-Assoziator, das Hopfield-Netz bekommt also einen Satz von Mustern:

$$S_j^{\mu}, \quad \mu = 1 \ldots p \, .$$

zum Lernen vorgegeben. Die Lernregeln lauten dann wie folgt:

$$w_{ii} = 0 \, ,$$

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{p} S_i^{\mu} S_j^{\mu}, \quad i \neq j \, ,$$

$$\Theta_i = 0 \, .$$

Haben beide Neuronen dieselbe Aktivität, so wird die Verbindung verstärkt, andernfalls gehemmt. Der Faktor $1/N$ dient lediglich dazu, die Gewichte nicht allzu groß werden zu lassen (kann im Prinzip auch wegfallen).

3.6.5 Varianten

Neben dem hier beschriebenen Hopfield-Netz findet man in der Literatur noch weitere Varianten, die hier nur kurz angeführt werden (siehe dazu [Hoffmann] S.96f.):

Schwellenlernen

Kommen die Werte -1 und +1 in den Mustern mit sehr verschiedener Häufigkeit vor, so müssen auch die Schwellen Θ_i gelernt werden. Dazu führt man ein Bias-Neuron ein, welches mit der Schwelle durch $\Theta_i = - w_{i0}$ zusammenhängt.

Festgehaltene Eingänge

Manche Modelle verwenden bei jedem Schritt zur Berechnung des effektiven Eingangs auch den Netzeingang mit.

Signum-Funktion

Als Ausgabefunktion wird eine Signum-Funktion, anstatt der oben beschriebenen verwendet. Dies ist eine spezielle Schwellenwertfunktion (siehe Abbildung 2-2a) mit dem Wertebereich {-1, +1}. Die Ausgangsfunktion sieht dann wie folgt aus:

$$o_i = \begin{cases} -1 & \textit{für } net_i < \Theta_i \\ +1 & \textit{für } net_i \geq \Theta_i \end{cases}.$$

Hopfield-Netze $\in$ {0, 1}

Der Wertebereich der Mustervektoren (also auch der Neuronen) ist {0, 1}. Die Funktionen werden dementsprechend angepasst. In [Zell] S.197ff. wird das gesamte Hopfield-Netz mit diesem Wertebereich behandelt.

3.6.6 Anwendung, Grenzen und Probleme

Hopfield-Netze eignen sich zur Mustererkennung, indem sie Eingabemuster mit vorher gespeicherten Mustern vergleichen. Das dem Angelegten am ähnlichsten gespeicherte Muster wird dabei ausgegeben.

Diese Netze eignen sich auch hervorragend zur Suche von optimalen Lösungen aus mehreren Möglichkeiten (Bsp.: *Traveling Salesman Problem*) und beim Umgang mit unvollständigen Daten, indem sie die fehlenden Daten aus ihren Strukturen ermitteln.

Ein Hopfield-Netz kann nur eine begrenzte Anzahl von Mustern speichern. Abschätzungen zeigen, dass man den Wert $p_{max} = 0{,}138 \cdot N$ nicht überschreiten sollte. Eine Begründung dazu findet man z.B. bei [Hertz].

Das Problem von Hopfield-Netzen ist, dass sie sich häufig in einem lokalen Minimum stabilisieren, statt im angestrebten globalen Minimum. Abhilfe davon schaffen sogenannte statistische Methoden, wie sie z.B. in einer Boltzmann-Maschine verwendet werden. Die Neuronen verändern ihren Zustand nicht mehr deterministisch, sondern zufällig nach einer Wahrscheinlichkeitsverteilung (siehe dazu Abschnitt 3.9).

3.7 Backpropagation-Netz

3.7.1 Einleitung

Das Backpropagation-Netz findet man in der Literatur auch unter dem Namen *Fehlerrückführungs-Netz*. Von anderen Netzen unterscheidet es sich weniger durch seine Struktur als vielmehr durch die verwendete Lernregel, die *Backpropagation-Regel*.

Diese ist eine Verallgemeinerung der Delta-Regel für Netze mit mehr als einer Schicht trainierbarer Gewichte und für Neuronen mit nichtlinearer Aktivierungsfunktion. Sie ist also auch auf verborgene Neuronen anwendbar.

3.7.2 Aufbau

Ein Backpropagation-Netz ist ein mehrschichtiges, vorwärtsgekoppeltes Netz. Neben der Ein- und Ausgabeschicht enthält es zudem noch eine oder mehrere versteckte Neuronenschichten.

Die Aktivierungsfunktion der Neuronen ist monoton, differenzierbar und nichtlinear (diese drei Eigenschaften werden unter dem Begriff *semilineare Funktionen* zusammengefasst). Jedes Neuron einer Schicht ist mit jedem Neuron der nachfolgenden Schicht verbunden.

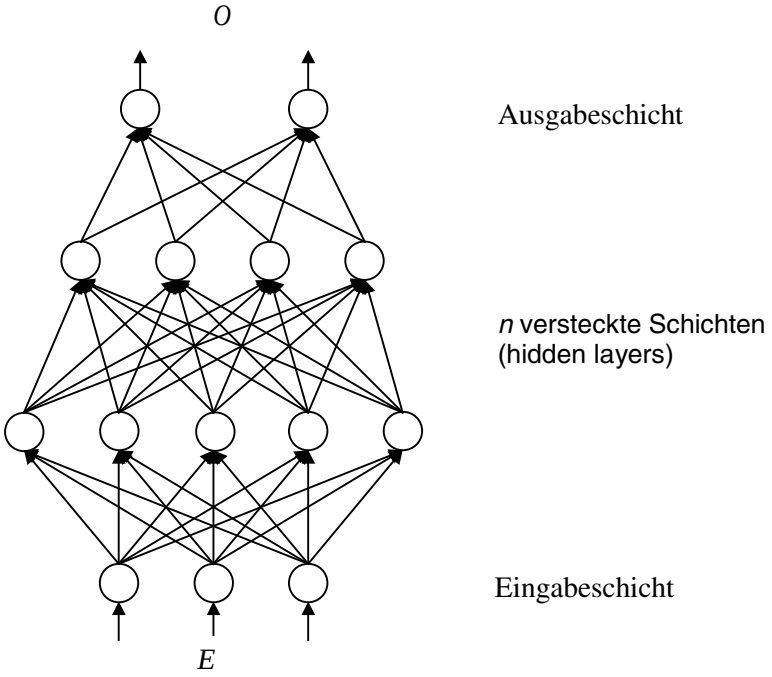


Abbildung 3-7: Aufbau eines Backpropagation-Netzes

3.7.3 Lernverfahren

Der Backpropagation-Algorithmus ist ein *Gradientenabstiegsverfahren*. Alle Gradientenabstiegsverfahren berechnen den Gradienten einer Zielfunktion, hier der Fehlerfunktion. Es wird dabei versucht durch Änderung der Gewichte den Fehler zu minimieren, indem eine Änderung aller Gewichte um einen Bruchteil des negativen Gradienten der Fehlerfunktion vorgenommen wird.

Die Backpropagation-Lernregel wird auf Netze mit folgenden Eigenschaften angewendet:

- das Netz hat mehrere Ebenen (also auch versteckte Neuronenschichten),
- die Aktivierungsfunktion muss nichtlinear, monoton und differenzierbar sein.

Lernvorgang

Gewichte auf Zufallswerte setzen	
	Lernschritt
Wiederholen bis zum Lernerfolg	

Abbildung 3-8: Schema des Backpropagation-Lernvorgangs

Für die einzelnen Lernschritte sind zwei Schemata üblich, die im Folgenden und in [Hoffmann] S.88ff. beschrieben werden.

Kumulatives Lernen

Zunächst reproduziert man das Netz für alle zu lernenden Muster, die Reihenfolge ist dabei gleichgültig. Die dabei erhaltenen Aktivitäten aller Neuronen sowie die Ausgangswerte der Ausgangsneuronen muss man zwischenspeichern, da sie für die Fehlermaße benötigt werden.

Als nächstes berechnet man die Fehler der Ausgangsneuronen. Anschließend sind, ausgehend von der letzten verborgenen Schicht, die Fehlermaße der weiteren Schichten an der Reihe. Zum Schluss werden die Gewichte angepasst. Dieses Lernverfahren hat den Nachteil, dass man viele Zwischenergebnisse abspeichern muss.

Netz für alle Muster reproduzieren und Zwischenergebnisse abpeichern	
Fehlermaße der Ausgangsneuronen berechnen	
Letzte verborgene Schicht wählen	
	Fehlermaße der gewählten Schicht berechnen
	Vorhergehende Schicht wählen
Wiederholen, bis alle Fehlermaße berechnet sind	
Alle Gewichte ändern	

Abbildung 3-9: Schema des Kumulativen Lernens

Vereinfachtes Lernen

Bei diesem Lernschritt werden die Muster einzeln berechnet. Man wählt ein Musterpaar aus, reproduziert das Netz, berechnet (ausgehend von der Ausgangsschicht) die Fehlermaße und ändert die Gewichte. Diese Prozedur führt man für alle Musterpaare durch. Damit ist ein Lernschritt beendet.

Wiederhole für alle Muster	
	Netz reproduzieren
	Fehlermaße der Ausgangsneuronen berechnen
	Fehlermaße der verborgenen Neuronen berechnen
	Alle Gewichte ändern

Abbildung 3-10: Schema des Vereinfachten Lernens

Bei der Berechnung der Fehlermaße fällt dabei die Summation über alle Musterpaare weg (Summenzeichen). Die Reihenfolge in der die Muster gelernt werden ist beliebig, jedoch ist eine zufällige Verteilung günstiger.

Auf die Initialisierung der Gewichte ist besonderes Augenmerk zu richten. Im Gegensatz zur Delta-Lernregel (hier konnten alle Gewichte mit 0 initialisiert sein) müssen alle Gewichte ungleich 0 und untereinander verschieden sein. Dafür setzt man die Gewichte auf Zufallswerte, die gewöhnlich merklich kleiner als 1 sind.

Es wird folgende Backpropagation-Regel für den Lernprozess verwendet:

$$\Delta^{\mu} w_{ij} = \eta o_i^{\mu} \delta_j^{\mu} .$$

Die Formel für das Fehlersignal δ_{pj} lautet dabei:

$$\delta_j^{\mu} = \begin{cases} o_j^{\mu}\left(1-o_j^{\mu}\right)\left(t_j^{\mu}-o_j^{\mu}\right), & \textit{falls } j \textit{ Ausgabezelle} \\ o_j^{\mu}\left(1-o_j^{\mu}\right)\sum_k \delta_k^{\mu} w_{jk}, & \textit{falls } j \textit{ verdeckte Zelle} \end{cases}.$$

In diesem Fall (im Unterschied zur bisherigen Nummerierung) bezeichnet der Wert δ_j^{μ} nicht die Gewichtsänderung, sondern das Fehlersignal der Zelle j bei Muster μ. Die Gewichtsänderung bei Muster μ wird durch den Wert $\Delta^{\mu} w_{ij}$ ausgedrückt (siehe dazu [Zell] S.110 oder [Hoffmann] S.86f.)

3.7.4 Anwendung, Grenzen und Probleme

Eine Gradientenabstiegsmethode findet im ungünstigsten Fall nur ein lokales Minimum der Fehlerfunktion. Ob ein globales Minimum gefunden wird hängt hauptsächlich von der günstigen Wahl der Startgewichte ab. Leider ist keine systematische Methode bekannt, die solche Gewichte bestimmt.

3.8 BAM

3.8.1 Einleitung

BAM bedeutet *bidirektionaler Assoziativspeicher* bzw. in der Originalbezeichnung *bidirectional associative memory*. Das BAM ist mit dem Hopfield-Netz verwandt, es handelt sich jedoch um ein heteroassoziatives Netzwerk. Die Verwandtschaft zum Hopfield-Netz wird weniger beim Aufbau als mehr beim Lernen und bei der Reproduktion deutlich.

3.8.2 Aufbau

Das BAM-Netz besteht aus zwei Schichten. Die Ausgänge der Eingangsneuronen (Anzahl N_E) sind mit allen Ausgangsneuronen (Anzahl N_O) verbunden. Die Ausgangsneuronen sind auf die gleiche Weise wieder mit den Eingängen der Eingangsneuronen verbunden (Rückkopplung). Die direkte Rückkopplung innerhalb einer Schicht wie sie beim Hopfield-Netz (siehe Abschnitt 3.6) zu finden ist, wird durch eine indirekte Rückkopplung ersetzt.

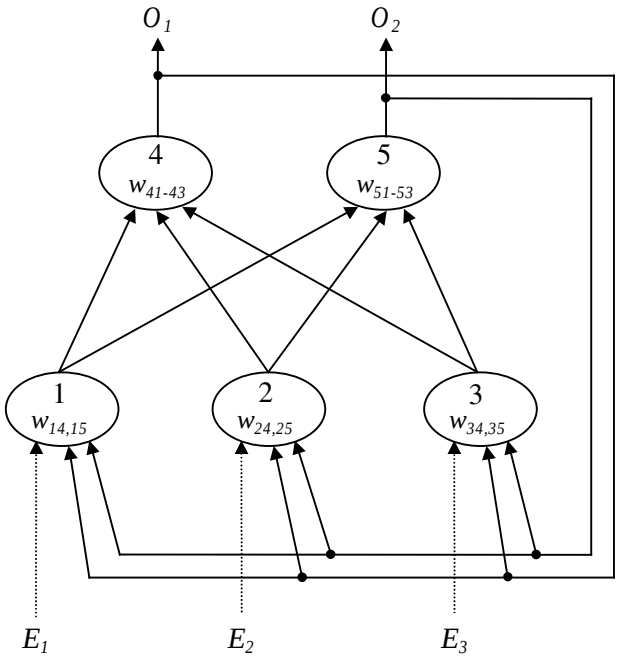


Abbildung 3-11: Aufbau eines BAM-Netzes

3.8.3 Funktionen zur Berechnung

Die Neuronen des BAM sind alles Hopfield-Neuronen. Als Gesamtanzahl der Neuronen ergibt sich dabei:

$$N = N_E + N_O .$$

Die Gewichtsmatrix ist eine N x N – Matrix, wobei die Gewichte nicht angeschlossener Eingänge gleich 0 gesetzt werden. Für das Bild sieht sie z.B. wie folgt aus:

$$W = \begin{pmatrix} 0 & 0 & 0 & w_{14} & w_{15} \\ 0 & 0 & 0 & w_{24} & w_{25} \\ 0 & 0 & 0 & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & 0 & 0 \\ w_{51} & w_{52} & w_{53} & 0 & 0 \end{pmatrix}.$$

Die Eingänge der Ausgangsneuronen sind nur für j = 1, ..., N definiert. Damit die Eingangsvektoren dieser Neuronen zur Gewichtsmatrix passen, werden sie wie folgt erweitert:

$$e_j = 0 \quad für\ j = N_1 + 1 \ldots N .$$

Der Eingangsvektor eines Ausgangsneurons des obigen Beispiels sieht dann wie folgt aus:

$$e = (e_1, e_2, e_3, 0, 0).$$

Mit den Eingangsneuronen wird analog verfahren:

$$e_j = 0 \quad für\ j = 1 \ldots N .$$

Die Ein- und Ausgänge des Netzes benötigen keine Erweiterung (Informationen zur Indizierung findet man bei [Hoffmann] S.99).

Die Ein- und Ausgänge aller Neuronen liegen in der Menge {-1, +1} (In anderer Literatur findet man auch häufig {0, +1} als Wertebereich, diese Möglichkeit wird etwas später noch beschrieben). Der effektive Eingang der Eingangsneuronen lautet daher:

$$net_i = \sum_{j=N_1+1}^{N} w_{ij} e_j, \quad i = 1 \ldots N_E .$$

Für die Ausgangsneuronen ergibt sich in etwa dieselbe Formel:

$$net_i = \sum_{j=1}^{N} w_{ij} e_j, \quad i = 1 \ldots N_O .$$

Die Transferfunktion ist dieselbe wie beim Hopfield-Netz:

$$o_i(t+1) = \begin{cases} -1 & für\ net_i(t+1) < \Theta_i \\ o_i(t) & für\ net_i(t+1) = \Theta_i \\ +1 & für\ net_i(t+1) > \Theta_i \end{cases} .$$

3.8.4 Reproduktion

Zu Beginn der Reproduktion wird ein definierter Startzustand hergestellt, indem die Ausgänge der Eingangsneuronen gleich den Netzeingängen gesetzt werden. So können die effektiven Eingänge der Ausgangsneuronen berechnet werden.

Dann werden deren Ausgangswerte berechnet, die wiederum die Eingänge der Eingangsneuronen darstellen. Es wird jetzt abwechselnd die Ausgangs- und die Eingangsschicht berechnet, bis das Abbruchkriterium erfüllt ist.

Im Gegensatz zum Hopfield-Netz müssen hier die Neuronen einer Schicht jeweils synchron arbeiten.

3.8.5 Lernverfahren

Das BAM arbeitet heteroassoziativ, die Lernaufgabe ist daher dieselbe wie beim Muster-Assoziator (siehe Abschnitt 3.1):

$$(E_j^\mu, S_i^\mu), \quad \mu = 1 \ldots p,\ j = 1 \ldots N_E, i = N_E + 1 \ldots N .$$

Die Lernregel ist eine Variante der Hopfield-Lernregel:

$$w_{ij} = \begin{cases} \sum_{\mu=1}^{p} E_j^\mu S_i^\mu & für\ j = 1 \ldots N_E, i = N_E + 1 \ldots N \\ w_{ji} & für\ i = 1 \ldots N_E,\ j = N_E + 1 \ldots N . \\ 0 & sonst \end{cases}$$

Die zweite Zeile bedeutet wieder Symmetrie der Gewichtsmatrix, die dritte verhindert Rückkopplungen innerhalb einer Schicht (sowie Selbstrückkopplungen).

3.8.6 Varianten

Neben dem hier beschriebenen Aufbau gibt es noch weitere Varianten, die sich allerdings nur geringfügig unterscheiden.

BAM $\in$ {0, +1}

Hier ist der Wertebereich der Elemente gleich $\{0, +1\}$. Die erste Zeile der Lernregel lautet demnach:

$$\sum_{\mu=1}^{p} \left(2E_j^{\mu} - 1\right)\left(2S_i^{\mu} - 1\right).$$

Nummerierung

Jede Neuronenschicht wird für sich nummeriert. Die Symmetrie der Gewichtsmatrix wird dann durch die Bedingung ersetzt, dass die Gewichtsmatrix der Ausgangsschicht gleich der transponierten Gewichtsmatrix der Eingangsschicht ist:

$$W^O = \left(W^E\right)^T.$$

3.8.7 Anwendung, Grenzen und Probleme

Wie bereits erwähnt, eignet sich das BAM als heteroassoziativer Speicher. Es akzeptiert einen Eingabevektor in einer Schicht von Neuronen und produziert einen dazu passenden, verschiedenen Ausgabevektor in einer anderen Schicht von Neuronen.

Ein BAM hat damit die Eigenschaft zur Generalisierung und kann somit auch mit unvollständigen Eingaben ein entsprechend assoziiertes Muster ausgeben.

3.9 Boltzmann-Maschine

3.9.1 Einleitung

Die Boltzmann-Maschine ist eine Erweiterung des Hopfield-Netzes aus Abschnitt 3.6, verwendet auch in etwa dieselbe Netzstruktur. Der wesentliche Unterschied liegt in der außerordentlich komplizierten Lernregel.

Ein Problem der Hopfield-Netze ist, dass sie sich häufig in einem lokalen statt globalen Minimum stabilisieren. Abhilfe davon schaffen stochastische Methoden, die unter folgenden Begriffen bekannt sind:

- Statistische Methoden,
- **Boltzmann-Maschine**,
- Simulated Annealing (simuliertes Abkühlen, ähnlich wie in der Stahlproduktion).

3.9.2 Aufbau

Die Boltzmann-Maschine ist aus dem Hopfield-Netz entstanden. Es besteht aus N Neuronen die vollständig miteinander verbunden sind. Die Neuronen werden in drei Klassen eingeteilt: die Eingangsschicht, die verdeckte Schicht und die Ausgangsschicht. Es gelten dabei folgende Einschränkungen:

- Eingangs- und Ausgangsneuronen haben keine direkte Verbindung, die Gewichte zwischen diesen Schichten sind also gleich 0,
- Selbstrückkopplung ist ausgeschlossen, d.h. $w_{ii} = 0$,
- die Gewichte sind symmetrisch, d.h. $w_{ij} = w_{ji}$.

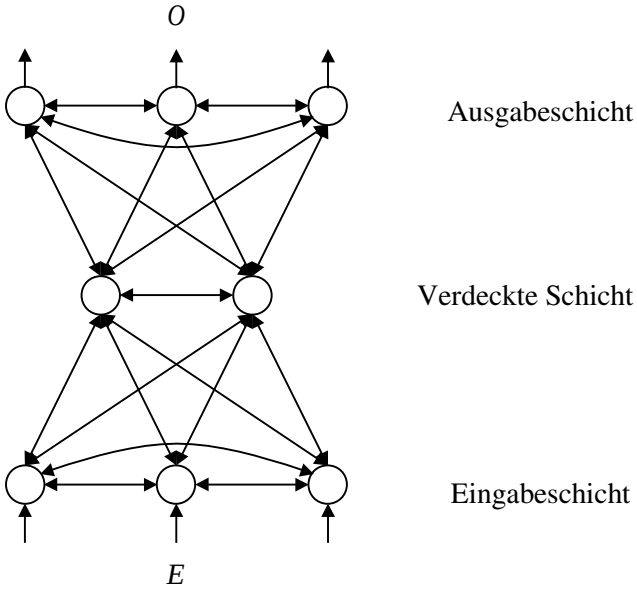


Abbildung 3-12: Aufbau einer Boltzmann-Maschine

Die Neuronen der Boltzmann-Maschine sind Boltzmann-Neuronen (siehe Tabelle 2-1).

3.9.3 Funktionen zur Berechnung

Die verdeckten Zellen werden nicht durch die Umgebung beeinflusst, sie sollen Strukturen im Eingangsvektor entdecken, die nicht durch paarweise Beziehungen seiner Komponenten ausgedrückt werden können.

Die Aktivierungen der Neuronen (Boltzmann-Neuronen) liegen in der Menge {0, 1}, die Gewichte der Verbindungen können beliebige reelle Zahlen sein. Die Gesamtanzahl der Neuronen ist *N*. Der effektive Eingang der Zellen lautet:

$$net_i = \sum_{j=1}^{N} w_{ij} o_j \,.$$

Die Transferfunktion der Neuronen ist eine stochastische Funktion. Eine zufällig ausgewählte Zelle wird mit folgender Wahrscheinlichkeit aktiv:

$$P(o_i = 1) = \frac{1}{1 + e^{-(net_i - \Theta_i)/T}} \,.$$

3.9.4 Reproduktion

Bei der Reproduktion setzt man die Ausgänge der Eingangsneuronen gleich dem vorgegebenen Eingangsmuster und hält diese Werte während der gesamten Reproduktion konstant. Nun führt man simuliertes Kühlen (wird bei der Beschreibung des Lernverfahren in Abschnitt 3.9.5 erläutert) durch, bis sich die Netzausgänge nicht mehr ändern. Dabei ist ohne Bedeutung, ob die verborgenen Neuronen einen stationären Zustand erreicht haben oder nicht.

3.9.5 Lernverfahren

Die Boltzmann-Maschine arbeitet heteroassoziativ, sie lernt also Musterpaare der Art:

$$\left(E_j^{\mu}, S_i^{\mu}\right), \quad \mu = 1 \ldots p \,.$$

In der Lernphase liegt der wesentliche Unterschied des Boltzmann-Netzes zum Hopfield-Netz. Man führt „simuliertes Kühlen“ (*simulated annealing*) durch.

Zunächst setzt man die Gewichte auf Zufallswerte, wobei allerdings die im Abschnitt 3.9.2 genannten Einschränkungen noch gelten müssen. Dann folgen die einzelnen Lernschritte, die wiederum aus drei Teilen bestehen. Folgendes Bild veranschaulicht das Schema:

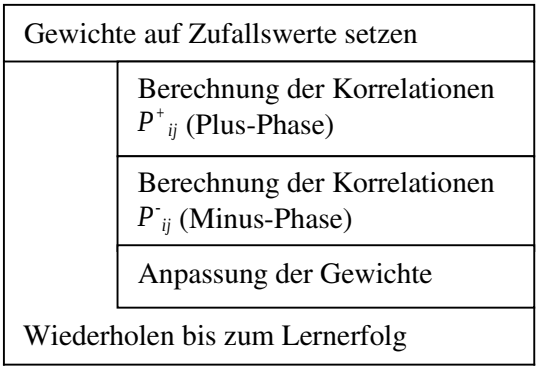


Abbildung 3-13: Schema des Lernvorgangs der Boltzmann-Maschine

Ziel des Lernalgorithmus ist es, die Gewichte so einzustellen, dass zwei Zellen im (thermischen) Gleichgewicht immer mit derselben Wahrscheinlichkeit aktiv sind, unabhängig davon, ob das Netz in der positiven oder in der negativen Phase läuft.

Der dreiteilige Lernschritt (Berechnung der Korrelationen und Anpassung der Gewichte) wird so oft wiederholt, bis das Lernziel erreicht ist.

Plus-Phase

Man legt ein Musterpaar μ an das Netz an. Das bedeutet, dass die Ausgänge der Eingangsneuronen gleich dem Eingangsvektor *E* und die Ausgänge der Ausgangsneuronen gleich dem Ausgangsvektor *A* gesetzt werden. Diese Ausgänge werden während der gesamten Plus-Phase festgehalten, es dürfen also alle sichtbaren Zellen ihren Zustand nicht ändern.

Nun führt man *simuliertes Kühlen* durch, bis sich bei einer niedrigen „Temperatur“ T_0 ein „thermisches Gleichgewicht“ einstellt. Das simulierte Kühlen läuft wie folgt ab:

Man beginnt mit einem hohen Wert der „Temperatur“ T und „kühlt“ das Netz allmählich ab, verringert T also schrittweise bis auf den Wert 0 (das Verringern geschieht sehr langsam – oft wird dazu die Funktion $T = 1 / i$ verwendet, mit i als Index der Zeitschritte).

Dabei verändern sich nur die Ausgänge der verdeckten Neuronen. Für jedes Neuronenpaar (i, j) kann man nun die *Korrelation* $P^{\mu+}_{ij}$ definieren. Es handelt sich hierbei um die Wahrscheinlichkeit (relative Häufigkeit), dass die beiden Neuronen i und j im thermischen Gleichgewicht gleichzeitig aktiv sind. Diese Größe wird dann für die spätere Auswertung noch benötigt.

Diesen Vorgang (Musterpaar anlegen, simuliertes Kühlen, Bestimmung der Korrelation) führt man r mal durch. Im Allgemeinen ist r ein Vielfaches der Musterzahl p. Zum Schluss berechnet man die Korrelationsmittelwerte mit folgender Formel (die Summe erstreckt sich über alle r Korrelationen):

$$P_{ij}^{+} = \frac{1}{r} \sum P_{ij}^{\mu+} .$$

Abbildung 3-14 zeigt noch einmal das Schema zur Berechnung der Korrelationen. Es gilt sowohl für die Plus-, als auch für die Minus-Phase.

Wiederhole r mal	
	Anlegen eines zu lernenden Musterpaares
	Simuliertes Kühlen bis zur Einstellung des thermischen Gleichgewichts
	Bestimmung der Korrelationen P^{μ}_{ij} für alle (i, j)
Berechnen der Korrelationsmittelwerte	

Abbildung 3-14: Schema der Plus- und Minus-Phase

Minus-Phase

Die Vorgangsweise ist dieselbe wie in der Plus-Phase. Hier bringt man jedoch das Netz in einen Startzustand, indem man die Eingangsneuronen gleich dem Eingangsmuster E^{μ} setzt. Anschließend unterwirft man alle Neuronen des Netzes der simulierten Kühlung. Es wird also auch den sichtbaren Neuronen erlaubt, ihren Zustand zu ändern. Das Netz läuft sozusagen „frei“. Man erhält Korrelationen $P^{\mu-}_{ij}$, die man wieder zum Mittelwert zusammenfasst:

$$P_{ij}^{-} = \frac{1}{r} \sum P_{ij}^{\mu-} .$$

Gewichtsänderung

Nach diesen beiden Lernschritten werden nach folgender Formel die Gewichte geändert:

$$\delta w_{ij} = \eta \left(P_{ij}^{+} - P_{ij}^{-} \right).$$

3.9.6 Anwendung, Grenzen und Probleme

Ein Problem, das die Boltzmann-Maschinen für viele Anwendungen ungeeignet erscheinen lässt, ist der große Rechenaufwand. Besonders aufwendig ist dabei die Bestimmung der beiden Werte $P^{\mu+}_{ij}$ und $P^{\mu-}_{ij}$. Darüber hinaus arbeitet das Netz nur dann zufriedenstellend, wenn die Parameter r und η günstig gewählt werden. Dafür gibt es aber keine allgemeinen Regeln.

3.10 LVQ

3.10.1 Einleitung

LVQ bedeutet *lernende Vektorquantisierung* bzw. auf Englisch *learning vector quantization*. Lernende Vektorquantisierung und selbstorganisierende Karten (*Kohonen maps*) sind zwei Gruppen von Lernverfahren, die eng miteinander verwandt sind. Dabei lassen sich die selbstorganisierenden Karten als eine Weiterentwicklung der LVQ-Netzwerke ansehen.

LVQ ist ein überwachtes Lernverfahren, die selbstorganisierenden Karten sind unüberwachte Lernverfahren.

3.10.2 Aufbau

Beim LVQ handelt es sich um ein einschichtiges Netz, d.h. es besitzt eine Schicht aktiver Neuronen. Diese werden als *Kohonen-Neuronen* bezeichnet. Davor werden meist noch Eingabeneuronen geschaltet, die aber keine Informationsverarbeitung realisieren, sondern nur den Eingabevektor weiterreichen.

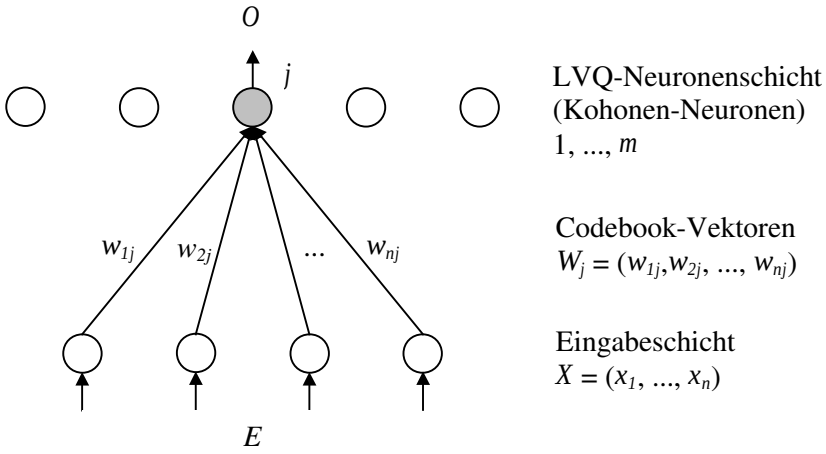


Abbildung 3-15: Aufbau eines LVQ-Netzes

Jedes Neuron der Eingangsschicht ist vollständig mit jedem Neuron der LVQ-Schicht verbunden. Es gibt im LVQ-Netz eine Reihe von sogenannten *Codebookvektoren* (Gewichtsvektoren der Neuronen), die alle einer Zielklasse zugeordnet sind. Es gibt üblicherweise für jede Klasse von Eingabevektoren mehrere Codebookvektoren.

3.10.3 Funktionen zur Berechnung

Im Gegensatz zur bisherigen Ordnung, wird das aktuell betrachtete Neuron hier mit dem Index *j* gekennzeichnet (dies wird in [Zell] S.171ff. so gehandhabt, während bei [Hoffmann] diese Art von Netz nicht beschrieben wird). Der Eingabevektor wird mit *X* (anstelle von *E*) bezeichnet.

Ein Codebookvektor (in [Zell] als m_j bezeichnet) ist im Grunde genommen nichts anderes als der Gewichtsvektor $W_j = (w_{1j}, ..., w_{nj})$ des Kohonen-Neurons *j*, das von den *n* Eingabeneuronen seine Eingabe erhält.

3.10.4 Reproduktion

Ein neuer bzw. bereits trainierter Eingabevektor wird an die Eingabeschicht angelegt und parallel mit allen Referenzvektoren verglichen. Derjenige Vektor der dem Eingabevektor am ähnlichsten ist, gibt seine Klasse an. Die Kohonen-Schicht arbeitet also nach dem *winner-takes-all-Prinzip*.

3.10.5 Lernverfahren

LVQ ist ein überwachtes Lernverfahren, welches sich hervorragend zur Klassifikation eignet. Zu jedem Eingabevektor muss zusätzlich bekannt sein, zu welcher Klasse (oder Kategorie) er gehört. In [Zell] S.172ff. werden vier verschiedene Varianten des LVQ-Lernverfahrens beschrieben:

LVQ 1

Zunächst wird der Eingabevektor X parallel mit den Codebookvektoren $W_1, ..., W_m$ verglichen. Das Neuron c, dessen Gewichtsvektor W_c zum Eingabevektor am ähnlichsten ist, ist der Gewinner (also wie bei der Reproduktion). Bei LVQ 1 handelt es sich im Prinzip also um einen „Nächster-Nachbar-Klassifikator". Mathematisch wird diese Ähnlichkeit durch eine spezielle Norm ||.|| definiert:

$$\|X - W_c\| = \min_j \left(\|X - W_j\| \right),$$

oder unter Verwendung der *arg*-Funktion (liefert den Index der Minimumsuche):

$$c = \arg\min_j \left(\|X - W_j\| \right).$$

Die Werte für die Gewichtsvektoren, die den Klassifikationsfehler minimieren, können wie folgt gefunden werden:

Der Gewichtsvektor W_c des Gewinnerneurons c das zum Eingabevektor am ähnlichsten ist, wird noch ähnlicher zum Eingabevektor gemacht, wenn c der gleichen Klasse wie der Eingabevektor angehört, sonst unähnlicher. Die Gewichtsvektoren der übrigen Neuronen bleiben unverändert. Dies geschieht durch Addition bzw. Subtraktion eines variablen Bruchteils α des Differenzvektors $X(t) - W_c(t)$:

$$W_c(t+1) = \begin{cases} W_c(t) + \alpha(t)\left[X(t) - W_c(t)\right] & falls\ Klasse(W_c) = Klasse(X) \\ W_c(t) - \alpha(t)\left[X(t) - W_c(t)\right] & falls\ Klasse(W_c) \neq Klasse(X) \end{cases}.$$

Der Wert α ist in der Regel konstant, oder er kann mit der Zeit t monoton fallen. Es gilt dabei

$$0 < \alpha(t) < 1.$$

Die folgende Grafik veranschaulicht noch einmal das Verfahren:

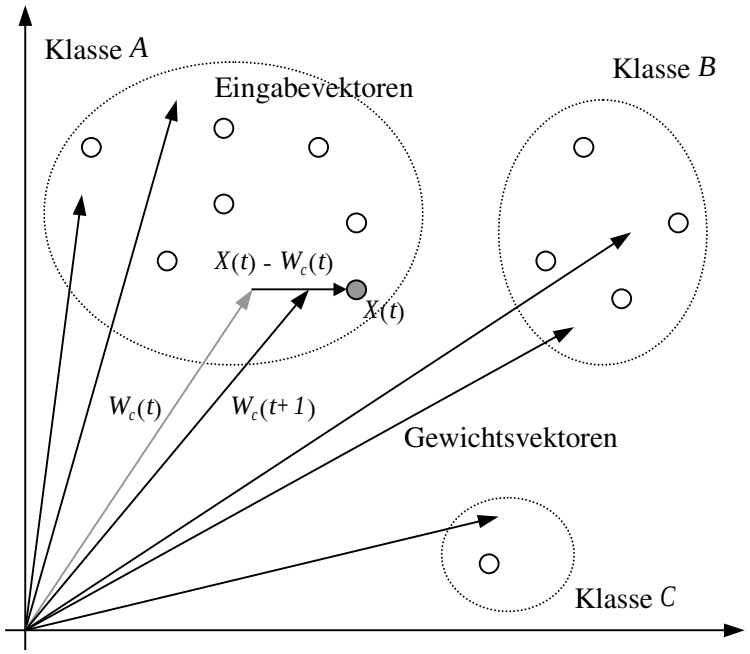


Abbildung 3-16: Grafische Darstellung des LVQ 1-Lernverfahrens

LVQ 2.1

Diese Version unterscheidet sich von LVQ 1 dadurch, dass sie nicht nur den nächsten Gewichtsvektor W_i (also der dem Eingabevektor am ähnlichsten ist) sucht sondern auch den übernächsten Gewichtsvektor W_j. Eine Anpassung der Gewichte findet aber nur statt, wenn die folgenden Voraussetzungen erfüllt sind:

- die Klassen der beiden Gewichtsvektoren W_i und W_j sind unterschiedlich,
- der Eingabevektor X gehört einer der beiden Klassen von W_i oder W_j an, und
- X liegt in einem „Fenster“ entlang der Mittelsenkrechte zwischen beiden Klassen.

Das Fenster ist in diesem Fall kein Rechteck. Es wird mit der folgenden Formel bestimmt:

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s, \quad mit\ s = \frac{1-v}{1+v}.$$

Mit d_i und d_j sind hier die euklidischen Abstände des Eingabevektors X von den Vektoren W_i und W_j gemeint. Die Breite v des Fensters sollte laut [Zell] zwischen 0.2 und 0.3 liegen. Abbildung 3-17 veranschaulicht dies noch einmal:

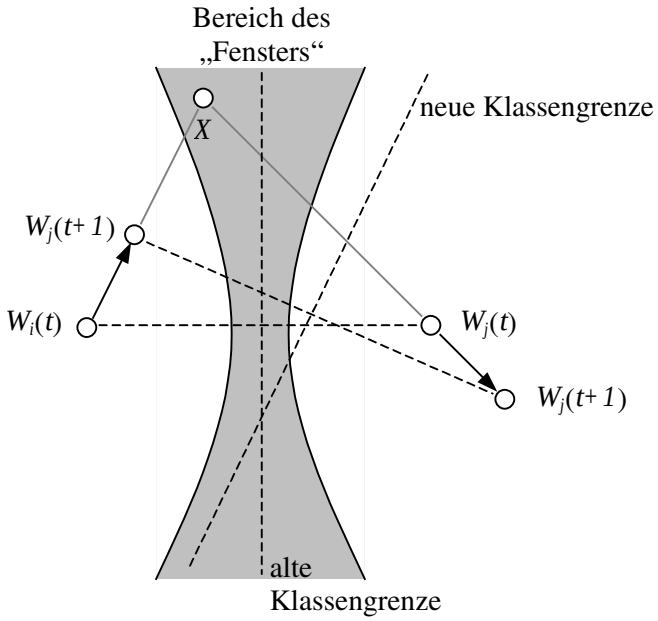


Abbildung 3-17: Grafische Darstellung des LVQ 2.1-Lernverfahrens

Der Vektor $W_i(t)$, dessen Klasse mit der von X übereinstimmt, wird in Richtung des Eingabevektors X verschoben, der nächste Vektor $W_j(t)$, dessen Klasse nicht mit der von X übereinstimmt, wird von X weggeschoben. Dadurch dreht sich die durch die Mittelsenkrechte der Verbindungslinie gebildete Klassengrenze zwischen den beiden Codebookvektoren.

Die Formeln für die Gewichtsänderung bei LVQ 2.1 lauten:

$$W_i(t+1) = W_i(t) + \alpha(t)\left[X(t) - W_i(t)\right],$$
$$W_j(t+1) = W_j(t) + \alpha(t)\left[X(t) - W_j(t)\right].$$

W_i und W_j sind hierbei die beiden nächsten Codebookvektoren zu X, wobei W_i zur gleichen Klasse wie X gehört und W_j zu einer anderen.

Dieses Verfahren modifiziert die Klassengrenze durch Verschiebung der Vektoren. Es garantiert jedoch nicht, dass sich die Dichte der Vektoren der Verteilungsdichte der Eingabevektoren annähert. Um diesen Mangel zu beheben, wurde LVQ 3 entwickelt.

LVQ 3

Dies ist eine Weiterentwicklung des LVQ 2.1 - Verfahrens. Es werden zusätzlich zur Änderung der Gewichtsvektoren an einer Klassengrenze die Vektoren W_i und W_j verändert, wenn beide der selben Klasse wie X angehören.

Die Formeln für das LVQ 3 Verfahren lauten

- für den Fall, dass die beiden nächsten Vektoren W_i und W_j verschiedenen Klassen angehören und W_i der selben Klasse wie X (also wie bei LVQ 2.1):

$$W_i(t+1) = W_i(t) + \alpha(t)[X(t) - W_i(t)],$$
$$W_j(t+1) = W_j(t) + \alpha(t)[X(t) - W_j(t)],$$

- für den Fall, dass W_i und W_j der selben Klasse wie X angehören (bei LVQ 2.1 wurde in diesem Fall keine Gewichtsänderung vorgenommen):

$$W_i(t+1) = W_i(t) + e\alpha(t)[X(t) - W_i(t)],$$
$$W_j(t+1) = W_j(t) + e\alpha(t)[X(t) - W_j(t)].$$

Als gute Richtwerte wurden bei [Zell] Werte von e zwischen 0.1 und 0.5 angegeben, wobei der optimale Wert von der Größe des Fensters abhängt und für „engere“ Fenster besser kleiner zu wählen ist.

Dieser Algorithmus ist stabiler als LVQ 2.1. Ein fortlaufendes Lernverfahren verändert optimal eingestellte Gewichtsvektoren nicht mehr.

Die beiden Verfahren LVQ 1 und LVQ 3 passen die Lage der Codebookvektoren der Verteilung der Eingabevektoren an, während LVQ 2.1 nur die relativen Entfernungen der Codebookvektoren von der Grenzlinie zwischen den Klassen optimiert. Dies garantiert allerdings nicht, dass die Vektoren optimal platziert sind.

LVQ 2.1 sollte daher in erster Linie zur schärferen Klassentrennung mit einer kleinen Lernrate und einer relativ kleinen Zahl von Lernschritten verwendet werden.

OLVQ 1

Für LVQ 1 ist es möglich, die Lernrate für schnelle Konvergenz des Verfahrens zu optimieren. Dazu wurde das Verfahren OLVQ 1, das *optimierte LVQ 1* (*optimized learning vector quantization 1*), entwickelt. Im Unterschied zu LVQ 1 wird bei diesem Verfahren jedem Gewichtsvektor eine eigene Lernrate $\alpha_c(t)$ zugewiesen. Es werden die folgenden Formeln verwendet:

$$W_c(t+1) = \begin{cases} W_c(t) + \alpha_c(t)[X(t) - W_c(t)] & falls\ Klasse(W_c) = Klasse(X) \\ W_c(t) - \alpha_c(t)[X(t) - W_c(t)] & falls\ Klasse(W_c) \neq Klasse(X) \end{cases},$$
$$W_j(t+1) = W_j(t) \quad falls\ j \neq c\ .$$

Für die optimale Konvergenz wird für die Änderung der Lernfaktoren $\alpha_c(t)$ folgende Regel verwendet:

$$\alpha_c(t) = \frac{\alpha_c(t-1)}{1 + s(t)\alpha_c(t-1)}.$$

Es muss dabei sichergestellt werden, dass $\alpha_c(t)$ nicht über 1 anwächst.

3.10.6 Anwendung, Grenzen und Probleme

Die LVQ-Verfahren werden für Klassifikationsaufgaben genutzt. Zu jedem Eingabe- und Codebookvektor muss daher die Klassenzugehörigkeit bekannt sein. Die Genauigkeit, mit der eine Klassifikationsaufgabe gelöst werden kann, hängt von einer Reihe von Faktoren ab:

- der Zahl der verfügbaren Codebookvektoren für jede Klasse,

- der Initialisierung der Codebookvektoren,
- dem verwendeten Lernalgorithmus,
- der richtigen Lernrate für die einzelnen Schritte, und
- einem guten Abbruchkriterium zur Terminierung des Lernens.

Eine obere Grenze für die Gesamtzahl der Codebookvektoren ist im allgemeinen nur durch die begrenzte Rechenzeit gegeben.

Für die Anfangsinitialisierung der Codebookvektoren wählt man oft Vektoren aus den Trainingsdaten aus. Besitzen Codebookvektoren Initialwerte, die weit entfernt von den Werten der Trainingsmuster liegen, würden diese als „tote Neuronen“ (*dead neurons*, *dead codebook-vectors*) enden, da sie nie der nächste Nachbar eines Eingabevektors sein würden.

Zum Lernen wird laut [Zell] S.177 empfohlen, immer mit OLVQ 1 zu beginnen, wegen dessen schneller (und optimierter) Konvergenz. Die Grenzen der Erkennungsgenauigkeit werden dabei nach einer Zahl von Lernschritten erreicht, die der 30- bis 50-fachen Zahl der Codebook-vektoren entspricht.

Lässt man die LVQ-Verfahren zu lange lernen, tritt sehr schnell der Effekt des *Overfittings* auf, d.h. die Codebookvektoren haben sich zu sehr an die Trainingsmenge angepasst und es verringert sich die Erkennungsrate bei einer entsprechenden Testmenge. Der Lernvorgang sollte daher nach einer Schrittzahl, die der 50- bis 200-fachen Zahl der Codebookvektoren entspricht (abhängig von Algorithmus und Lernrate), abgebrochen werden.

3.11 Counterpropagation-Netz

3.11.1 Einleitung

Ein Counterpropagation-Netz wird in der Literatur auch häufig unter der Bezeichnung *Gegenstrom-Netz* erwähnt. Entwickelt wurde es von Robert Hecht-Nielsen. Trotz seiner Namensähnlichkeit und den Ähnlichkeiten im Aufbau hat es keine Verwandtschaft mit dem Backpropagation-Netz, sondern ist eher ein Verfahren, dass die Ansätze eines Kohonen-Netzes (wird u.a. in [Zell] S.179ff. und [Hoffmann] S.116ff. beschrieben) und eines Grossberg-Netzes vereint (Informationen hierzu findet man bei [Grosb69], [Grosb71] und [Grosb78]). Der Unterschied zum Backpropagation-Netz liegt im Lern- und Reproduktionsverhalten.

Ein Counterpropagation-Netz hat folgende Eigenschaften:

- Gegenüber Backpropagation ist in vielen Fällen die Trainingszeit stark reduziert, es ist dafür aber nicht so allgemein anwendbar wie ein Backpropagation-Netz.
- Es ist eine Kombination aus zwei Netztypen (Kohonen-Schicht und Grossberg-Schicht). Es liefert somit die Eigenschaften eines kombinierten Netzes, die keine der beiden Komponenten für sich alleine besitzt.
- Es verkörpert die Idee des „Baukasten-Systems“ bei neuronalen Netzen. Somit dient es als Vorbild zur Konstruktion komplexer neuronaler Netze aus einfachen modularen Komponenten.

3.11.2 Aufbau

Das Counterpropagation-Netz ist ein zweischichtiges vorwärtsgekoppeltes Netz (*feedforward-Netz*). Die erste Schicht wird als *Kohonen-Schicht*, die zweite als *Grossberg-Schicht* bezeichnet.

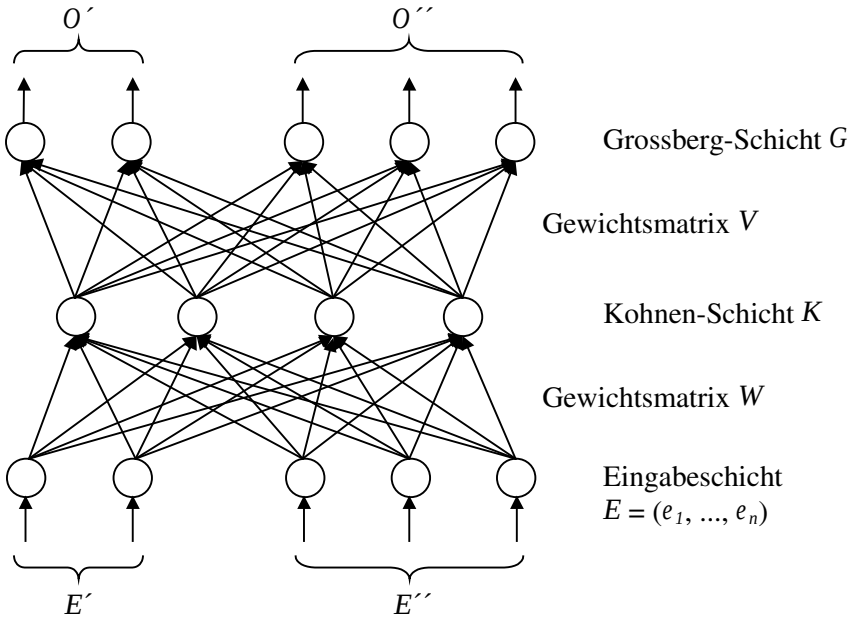


Abbildung 3-18: Aufbau eines Counterpropagation-Netzes

Um die Berechnungsformeln zu vereinfachen und um die Unterschiede in der Funktionsweise der beiden Schichten deutlicher zu machen, wird jede Neuronenschicht separat nummeriert. In der Kohonen-Schicht sind die Neuronen K_1 *bis* K_m (im Beispiel ist $m = 4$), in der Grossberg-Schicht die Neuronen G_1 bis G_p (im Beispiel ist $p = 5$).

Jedes Eingabeneuron ist mit jedem Neuron der Kohonen-Schicht verbunden. Diese Gewichte w_{ij} bilden die Gewichtsmatrix W. Jedes Kohonen-Neuron ist wiederum mit jedem Neuron der Grossberg-Schicht verbunden, diese Gewichte v_{ij} bilden die Gewichtsmatrix V.

3.11.3 Funktionen zur Berechnung/Reproduktion

Eingabeschicht

Die Eingabeschicht dient lediglich zur Verteilung der Eingabe, sie führt also keine Berechnungen aus. Die Aktivierungsfunktion dieser Neuronen ist die Identität. Für die Eingangsmuster sind Werte aus dem abgeschlossenen Intervall [0, 1] zulässig, das Counterpropagation-Netz ist also sowohl für binäre als auch für kontinuierliche Eingaben geeignet. Die Normalisierung der Eingabevektoren auf die Länge 1 (Intervall [0, 1]) ist nicht unbedingt notwendig, aber laut [Zell] empfehlenswert.

Reproduktion der Kohonen-Schicht

Die Kohonen-Schicht *K* bildet ein *winner-takes-all-Netz*, d.h. auf den Eingabevektor *X* reagiert nur ein einziges Neuron mit der Ausgabe 1, alle anderen mit der Ausgabe 0. Die Reproduktion erfolgt also durch Wettbewerb, das Kohonen-Neuron mit der größten Netzeingabe ist der Gewinner. Die Netzeingabe der Kohonen-Neuronen ist einfach die gewichtete Summe der Eingaben.

$$net_i = \sum_{j=1}^{N_E} w_{ij} E_j, \quad i = 1,\ldots,N_K \; .$$

Der Ausgang der Neuronen ergibt sich aufgrund des Wettbewerbs durch:

$$o_i^K = \begin{cases} 1 & \text{für } net_i^K = \max\limits_{k=1\ldots N_E} \left(net_k^K\right) \\ 0 & \text{für } net_i^K < \max\limits_{k=1\ldots N_E} \left(net_k^K\right) \end{cases}, \quad i = 1,\ldots,N_K \; .$$

Der obere Index *k* kennzeichnet in beiden Formeln die Kohonen-Schicht. Im Falle gleicher maximaler Netzeingaben ist es üblich, unter diesen Neuronen einen Gewinner zu bestimmen (z.B. durch Zufallsprinzip).

Reproduktion der Grossberg-Schicht

Die Neuronen der Grossberg-Schicht besitzen die gleiche Aktivierungsfunktion, wie die der Kohonen-Schicht. Sie arbeiten aber nicht nach dem *winner-takes-all-Prinzip*. Da hier die Gewichtsmatrix *V* verwendet wird, berechnet sich der Ausgang der Neuronen mit:

$$O_i = o_i = \sum_{j=1}^{N_K} v_{ij} o_j^k, \quad i = 1,\ldots,N_G \; .$$

Aufgrund der Festlegung, dass in der Kohonen-Schicht nur ein Neuron aktiv ist, lässt sich die Formel noch wie folgt vereinfachen:

$$O_i = v_{ik} \; .$$

Der Indes *k* bezeichnet dabei dasjenige Kohonen-Neuron mit dem Ausgang 1.

3.11.4 Lernverfahren

Das Counterpropagation-Netz lernt heteroassoziativ, also Musterpaare der Form:

$$\left(E^\mu, S^\mu\right), \quad \mu = 1,\ldots,p \; .$$

Die Kohonen-Schicht (also die verborgenen Neuronen) werden unüberwacht angelernt. Dies schränkt die Möglichkeiten dieses Netzes gegenüber eines Backpropagation-Netzes ein. Man benötigt aber weniger Rechenaufwand.

Die Kohonen-Schicht klassifiziert die Eingaben in Gruppen ähnlicher Vektoren. Die Gewichte werden dabei so modifiziert, dass ähnliche Eingabevektoren das gleiche Kohonen-Neuron aktivieren. Die Grossberg-Schicht muss daraus die gewünschte Ausgabe erzeugen.

Lernen in der Kohonen-Schicht

Die Kohonen-Schicht klassifiziert die Eingaben zu Klassen von ähnlichen Vektoren (siehe dazu [Zell] S.192). Das Lernen erfolgt in Analogie zur Reproduktion nach dem *winner-takes-all-Prinzip*. Die Eingangsvektoren und die Gewichtsvektoren sind auf die Länge 1 normiert. Ist dies nicht der Fall, kann das nach folgender Formel geschehen:

$$e_i^{new} = \frac{e_i}{\sqrt{e_1^2 + \ldots + e_n^2}}.$$

Bei einem Lernschritt verändert nur das Gewinner-Neuron seine Gewichte. Wie bei der Reproduktion ist der Gewinner das Neuron, mit der größten Netzeingabe.

Je Kleiner der Winkel zwischen Gewichts- und Eingabevektor ist, d.h. je dichter die beiden Vektoren beieinander liegen, um so größer ist der effektive Netzeingang eines Neurons. Also erfolgt die Bestimmung des Gewinner-Neurons mit Hilfe des Skalarproduktes, da dies ein Maß für die Ähnlichkeit zwischen dem Gewichtsvektor und dem Eingabevektor darstellt. Zeigen die beiden Vektoren in die gleiche Richtung, dann ist $net_i = 1$. Das Neuron, dessen Gewichtsvektor die größte Ähnlichkeit zum Eingabevektor zeigt, hat die größte Aktivität.

Durch die Änderung der Gewichte W_k, wird das Neuron k, dessen Gewichtsvektor dem Eingabevektor am ähnlichsten ist diesem noch ähnlicher gemacht. Für das Gewinner-Neuron k lautet die Lernregel:

$$\delta w_{kj}^K = \eta^K \left(E_j - w_{kj}^K\right), \quad j = 1, \ldots, N_E,$$

oder in anderer Form (äquivalente Formulierungen):

$$w_{kj}(t+1) = w_{kj}(t) + \eta^K \left(E_j(t) - w_{kj}^K(t)\right),$$
$$W_j(t+1) = W_j(t) + \eta^K \left(E(t) - W_j^K(t)\right).$$

Die letztere Formel gibt die Vektorschreibweise an, die zwei anderen beziehen sich auf einzelne Gewichte. Der obere Index K bezeichnet auch hier die Kohonen-Schicht, demnach ist η^K die Lernrate der Kohonen-Schicht. Sie wird laut [Zell] üblicherweise auf einen Wert bei ca. 0.7 gesetzt und wird dann allmählich verringert.

Der Effekt des Trainings ist eine Rotation des Gewichtsvektors in Richtung des Eingabevektors, ohne wesentliche Längenänderung. Im zweidimensionalen Fall lässt sich das gut veranschaulichen:

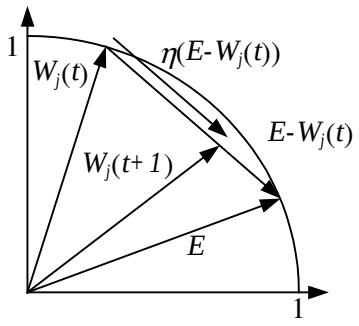


Abbildung 3-19: Änderung des Gewichtsvektors in der Kohonen-Schicht

Lernen in der Grossberg-Schicht

Im Gegensatz zur Kohonen-Schicht erfolgt das Lernen hier durch ein überwachtes Lernverfahren, nach der Delta-Lernregel. Es wird ein Eingabevektor angelegt und es erfolgt eine Vorwärtspropagierung und Bestimmung der Ausgabe in der Kohonen-Schicht und der Grossberg-Schicht.

Nur diejenigen Gewichte der Grossberg-Schicht werden geändert, deren Kohonen-Neuron eine Ausgabe ungleich 0 hat (also nur die Gewichte des Gewinner-Neurons). Die Lernregel sieht wie folgt aus:

$$v_{ij}(t+1) = v_{ij}(t) + o_j(t)\eta^G\left(S_i^\mu - v_{ij}(t)\right),$$

oder in anderer Schreibweise:

$$\delta v_{ij} = \eta^G\left(S_i^\mu - O_i\right)o_i^G, \quad i = 1,\ldots,N_G,\ j = 1,\ldots,N_K\,.$$

Hier ist η^G die Lernrate der Grossberg-Schicht. Ist *k* das aktive Neuron der Kohonen-Schicht, so gilt:

$$A_i = v_{ik}\text{, und}$$

$$o_k^K = 1\,.$$

Die Formel lässt sich daher wie folgt zusammenfassen (da für alle anderen Kohonen-Neuronen *j* der Ausgang o_j^K gleich 0 ist):

$$\delta v_{ij} = \eta^G\left(S_i^\mu - v_{ij}\right)o_j^K, \quad i = 1,\ldots,N_G, j = 1,\ldots,N_K\,.$$

Die Ähnlichkeit mit der Lernregel für die Kohonen-Schicht ist unverkennbar. Die Lernrate η^G wird anfangs meist auf 0.1 gesetzt und während des Trainings langsam reduziert. Dadurch konvergieren die Gewichte der Grossberg-Schicht langsam zu den mittleren Werten der gewünschten Ausgabe.

Die Grossberg-Schicht erzeugt zu jeder Klasse (d.h. zu jedem Kohonen-Element) eine sinnvolle Ausgabe.

Initialisierung der Gewichtsvektoren

Die Gewichtsvektoren werden auf zufällige Werte gesetzt. Für das Training der Kohonen-Schicht sollten diese Gewichtsvektoren zusätzlich normalisiert werden.

Der Grund dafür ist, dass nach dem Training die Gewichtsvektoren den normalisierten Eingabevektoren ähnlich sein sollen. Durch eine Normalisierung am Anfang sind sie daher näher an ihrem Zielzustand (große Ähnlichkeit mit den Eingabevektoren), was wiederum eine verkürzte Trainingszeit bedeutet.

Die zufällige Auswahl der Gewichtsvektoren kann aber wieder zu großen Problemen führen. Es wird unter Umständen in höher-dimensionalen Räumen keine Gleichverteilung erreicht, sondern die Vektoren gruppieren sich in einem kleinen Bereich des Hyperraums. Die Vektoren dieser Gruppen sind unter Umständen so weit vom Eingabevektor entfernt, dass sie nie die beste Näherung darstellen (ein ähnliches Problem trat schon bei den LVQ-Netzen aus Abschnitt 3.10 auf). Techniken zur Behandlung dieses Problems werden in [Zell] S.149f beschrieben.

3.11.5 Vollständiges Counterpropagation-Netz

Wie in Abbildung 3-18 ersichtlich, zerfallen die Netzeingänge und die Netzausgänge in zwei Gruppen mit den Anzahlen $N_{E'}$ und $N_{E''}$ bzw. $N_{O'}$ und $N_{O''}$. Dabei ist wichtig, dass die folgenden Beziehungen gelten:

$$N_{E'} = N_{O'},$$

$$N_{E''} = N_{O''}.$$

Dadurch gilt auch die Einschränkung, dass die Gesamtanzahl der Netzeingänge N_E gleich der Netzausgänge N_O ist.

Wird ein Paar von Vektoren (*E´*, *E´´*) angelegt, so erscheint am Ausgang das Ausgangsmuster *O* (bzw. (*O´*, *O´´*)). Der Ausgangsvektor ist dabei eine identische Kopie des Eingangsvektors, das Netz arbeitet in diesem Fall also autoassoziativ.

Das interessante daran ist aber, dass durch Anlegen nur eines der beiden Eingabevektoren (also E' oder E'' gleich 0 gesetzt) am Ausgang trotzdem der vollständige Ausgangsvektor (entspricht ja dem vollständigen Eingabevektor) erscheint, da das gleiche Neuron aktiviert wird.

Durch Anlegen von (E', E'') wird also das gleiche Kohonen-Neuron aktiviert, wie durch (E', 0) oder (0, E''). In allen Fällen wird dabei die Ausgabe (O', O'') = (E', E'') produziert. Das bedeutet, dass das Counterpropagation-Netz sowohl eine Funktion f, als auch gleichzeitig ihre inverse f^{-1} approximieren kann.

3.11.6 Anwendung, Grenzen und Probleme

Das Netz kann sowohl als autoassoziativer Speicher, als auch als heteroassoziativer Speicher eingesetzt werden.

Wenn der Eingabevektor E dem Trainingssatz angehört, dann ist der Ausgangsvektor O gleich dem Eingangsvektor E. In diesem Fall arbeitet das Netz autoassoziativ.

Legt man einen Eingangsvektor E = (E', E'') an, setzt dabei aber E'' = 0, so erhält man ebenfalls $O = E$ = (E', E'') als Ausgangsvektor. Der Ausgang O' entspricht dann einer autoassoziativen (der Vektor E' wird durchgereicht), der Ausgang O'' einer heteroassoziativen Arbeitsweise.

Das Netz arbeitet sowohl mit binären, als auch mit reellen Eingabewerten. Gegenüber einem Backpropagation-Netz ist die Trainingszeit in den meisten Fällen stark reduziert. Es ist dafür aber nicht so allgemein anwendbar wie ein Backpropagation-Netz.

Mögliche Anwendungen sind Probleme der Mustererkennung und –klassifikation sowie der Mustervervollständigung. Es sind also Anwendungen, in denen es in Konkurrenz zum Backpropagation-Netz steht.

4 Betrachtung der Netztypen

4.1 Die Klassifikationsaufgabe

Zunächst einmal muss genau geklärt werden, was unter dem Begriff *Klassifikation* zu verstehen ist und wie ein klassifizierendes Netz arbeiten soll, denn in der Literatur gibt es verschiedene Auffassungen davon, wie ein Klassifizierer arbeitet.

Unter dem allgemeinen Begriff der Klassifikation von Daten versteht man im Groben die Einteilung von Mustern (dargestellt als Vektoren) in Gruppen (auch *Klassen*, *Kategorien*, *Cluster*). Die Reproduktionsaufgabe (siehe dazu Abschnitt 2.3) eines Klassifizierers besteht darin, zu einem gegebenen Eingangsmuster die richtige Klasse anzugeben.

4.1.1 Speichermodelle

Jedes neuronale Netz arbeitet grundsätzlich nach dem selben Muster:

> An den Eingang wird eine Folge von Eingangsmustern angelegt, woraufhin das Netz am Ausgang als Ergebnis eine Folge von Ausgangsmustern produziert.

Meistens ist dieser Vorgang statisch, d.h. das Eingangsmuster bleibt so lange unverändert am Eingang bis der gewünschte Ausgangszustand eingenommen wird oder wenigstens approximiert wird (bei rückgekoppelten Netzen).

Im Wesentlichen unterscheidet man zwei Arten von Speichermodellen, die autoassoziativen Speicher und die heteroassoziativen Speicher.

Autoassoziative Speicher

Ein autoassoziativer Speicher liegt vor, wenn ein gelerntes Muster, welches an den Eingang angelegt wird, sich selbst reproduziert. Es erscheint bei der Reproduktion am Netzausgang im idealen Fall das angelegte Eingangsmuster, das Netz speichert also einzelne Muster in seinen Strukturen ab. Wird einem solchen Netz nur ein Teil eines gespeicherten Musters oder ein ähnliches (z.B. verrauschtes Muster) angeboten, so sollte es das korrekte gespeicherte Muster darstellen.

Ein solches Netz eignet sich also ideal zur *Musterergänzung*. Dabei ist es natürlich notwendig, dass die Anzahl der Ausgangsneuronen gleich der Anzahl der Eingangsneuronen sein muss. Für die Klassifikation ist diese Tatsache nicht von Vorteil, da es am Netzausgang unnötig viele Neuronen erfordern würde, obwohl als Ausgabe die zugehörige Klasse ausreicht. Dies ist wiederum verbunden mit einem Mehraufwand an Berechnungen.

Heteroassoziative Speicher

Ein heteroassoziativer Speicher liegt vor, wenn das Eingangsmuster im Prinzip mit beliebigen Ausgangsmustern assoziiert wird. Legt man eines der gelernten Eingangsmuster an, so soll das Netz das zugehörige gespeicherte Ausgangsmuster reproduzieren, das Netz lernt also Musterpaare der Form (Eingangsmuster, Ausgangsmuster). Bietet man dem Netz ein nichtgelerntes Muster an, so erhält man irgendein Muster am Ausgang, das natürlich in einem vernünftigen Bezug zum Eingangsmuster stehen soll. Es erscheint im Idealfall ein Ausgangsmuster das zu einem Eingangsmuster gehört, das dem angelegten am ähnlichsten ist.

Daher eignet sich ein solches Netz sehr gut zur *Generalisierung*. Die Aufgabe besteht also im wesentlichen darin, das Eingangsmuster auf ein möglichst ähnliches, allgemeineres Muster abzubilden.

Die Klassifikation, wie sie in dieser Arbeit von Interesse ist, ist ein Spezialfall der heteroassoziativen Speicher (siehe auch Abschnitt 4.1.4), bei dem das Ausgangsmuster die zugehörige Klasse ist.

4.1.2 Umsetzung

Wie bereits erwähnt, versteht man unter dem Begriff der Klassifikation von Daten im Groben die Einteilung von Mustern in Klassen. Nun stellt sich die Frage, wie diese Klassen durch ein neuronales Netz dargestellt werden sollen und nach welcher Methode das Netz die Aufgabe erlernen soll.

Am einfachsten erreicht man das Ziel, wenn für die einzelnen Netzausgänge nur zwei Werte zugelassen werden, mit denen die Neuronen angeben, ob der Eingangsvektor zu dieser Klasse gehört oder nicht. Dabei kann z.B. jedes Ausgangsneuron eine bestimmte Klasse repräsentieren. Gilt zusätzlich die Einschränkung, dass höchstens ein Neuron aktiv wird, so gibt dieses Neuron schließlich die Klasse an, der das Eingangsmuster zugeordnet wird.

Eine typische Lösung für dieses Problem erhält man, wenn die Ausgangsschicht des Netzes das aktive Neuron durch Wettbewerb ermittelt, es sich also um eine *winner-takes-all-Schicht* (siehe dazu auch Abschnitt 2.1.4 oder Abschnitt 2.2.1) handelt.

Ein Klassifizierer kann dabei überwacht oder unüberwacht lernen, was für die eigentliche Aufgabenstellung von wesentlicher Bedeutung ist.

4.1.3 Unüberwachtes Lernen

In vielen Büchern wird die Klassifikation als die „Einteilung von vorgegebenen Mustern in Klassen“ beschrieben, wobei diese Klassen vorher noch nicht bekannt sind. Diese Gruppen werden meist unter dem Begriff *Cluster* geführt, daher wird für diese Methoden der Begriff *Clustering* verwendet. Das neuronale Netz wird bei dieser Aufgabenstellung mit einem unüberwachten Lernalgorithmus trainiert.

Die Aufgabe des Algorithmus besteht darin, selbst eine gute Klasseneinteilung zu finden. Er muss dabei in der Menge der Eingabevektoren Ähnlichkeiten oder häufig auftretende Regelmäßigkeiten erkennen und daraufhin Gruppen von Objekten bilden, die innerhalb ihrer Gruppe eine möglichst große Ähnlichkeit aufweisen und zu Objekten anderer Klassen jedoch möglichst unähnlich sind ([Polifke] S.6).

Diese Art der Klassifizierung ist für diese Arbeit aber nicht von Interesse.

4.1.4 Überwachtes Lernen

In meiner Arbeit und in vielen Bereichen des Data Mining ist mit dem Begriff Klassifikation nicht die in Abschnitt 4.1.3 beschriebene Clusterung von Datensätzen gemeint, sondern eine Klassifikation, bei der in der vorgegebenen Menge der Eingabevektoren die zugehörige Klasse mit bekannt ist. Der Algorithmus soll nun anhand dieser Trainingsmenge eine möglichst gute Beschreibung der einzelnen Klassen finden. Diese wird dann auf Vektoren angewendet, bei denen die Zielklasse noch unbekannt ist (Klassifikationsmenge), um diese den vorhandenen Klassen möglichst fehlerfrei zuzuordnen.

Das neuronale Netz wird bei dieser Art mit einem Muster der Gestalt (Eingangsmuster, Klasse) angelernt. Ein Klassifizierer kann somit auch als heteroassoziativer Speicher aufgefasst werden, dessen Ausgangsmuster höchstens eine aktive Komponente enthalten. Andersherum gesagt, kann man einen beliebigen heteroassoziativen Speicher als Klassifizierer verwenden, wenn als Ausgangsmuster die Klasse des Eingangsmusters angegeben wird.

Für diese Aufgabenstellung benötigt man überwachte Lernverfahren. Da in dieser Arbeit nur diese Art der Klassifikation von Interesse ist, wurden auch im Kapitel 3 nur überwacht-lernende Netze behandelt.

4.2 Anforderungen

Zunächst stellt sich die folgende „große Frage“:

„Wie geht man bei der Auswahl eines geeigneten Netztyps vor?“

Zu dieser einfachen aber entscheidenden Frage findet man leider keine konkrete Antwort. Einen allgemeingültigen Ansatz zur Auswahl eines Netztyps der zum Problem passt, scheint es nicht zu geben. Laut [Zell] (S.416f.) ist es derzeit auch unmöglich, eine genaue Vorhersage über die Leistungsfähigkeit verschiedener Lernverfahren für eine gegebene Anwendung zu geben. Daher ist es für eine Untersuchung sinnvoller, die ungeeigneten Netze zu identifizieren und aus den übrig gebliebenen sich ein möglichst „gut geeignetes“ auszuwählen.

4.2.1 Allgemeine Anforderungen

Die Ausgangssituation eines Klassifikationsproblems kann nach folgendem allgemeinem Schema dargestellt werden:

> Es gibt eine Menge von N Objekten ($N = \{1, 2, ..., n\}$), die einzelnen Datensätze. Diese besitzen jeweils m Merkmale von i.A. verschiedener Art, deren Ausprägungen bekannt sind ($M = \{k_1, k_2, ..., k_m\}$). Fasst man diese Merkmalsausprägungen zusammen, so ergibt sich folglich die Datenmatrix A (hochdimensionaler Eingabevektor). Eines dieser Merkmale (meist als *Zielmerkmal* bezeichnet) gibt i.d.R. die Zielklasse an, der ein Datensatz angehört.

Welche Anforderungen werden nun an das neuronale Netz gestellt, um diesen Eingabevektor optimal zu verarbeiten und brauchbare Ergebnisse zu erhalten? Im Folgenden wird versucht, die wichtigsten dieser Anforderungen kurz vorzustellen.

Schnelligkeit

Die Geschwindigkeit eines Programms ist in der Praxis mit einer der wichtigsten Faktoren, die es zu beachten gilt. Frei nach dem Motto „Zeit ist Geld“ muss man bei der Entwicklung eines Produkts einen Kompromiss finden, der Schnelligkeit, Qualität (Generalisierungsfähigkeit) und Preis optimal verbindet.

Die Schnelligkeit, mit der ein Algorithmus arbeitet, hängt im Wesentlichen von zwei Punkten ab. Zum einen ist es der Rechner auf dem er ausgeführt wird, zum anderen ist dies die Anzahl der Rechenschritte und die Dauer eines Rechenschritts im Algorithmus selber. Will man unter verschiedenen Verfahren das Geeignetste auswählen sollte man zweckmäßig die erste Komponente (die Leistung des Rechners) als konstant ansehen.

Die Geschwindigkeit eines Algorithmus selber lässt sich dann durch die folgenden Kriterien bestimmen:

- Trainingsdauer bis zum Erreichen des Abbruchkriteriums

  Die Dauer zum Lernen aller Eingabemuster lässt sich durch zwei Maße beschreiben (siehe [Zell] S.414f.):

  Ein *Zyklus* besteht aus dem Anlegen eines einzelnen Trainingsmusters (Eingangsvektor und zugehöriger Ausgangsvektor bei überwachten Lernverfahren) an das neuronale Netz, der Vorwärtspropagierung (Reproduktion) durch das Netz und der Durchführung der nötigen Gewichtsänderungen, welche durch das Lernverfahren bestimmt werden (manche Algorithmen, wie z.B. Backpropagation, benötigen zusätzlich noch eine Rückwärts-propagierung des Fehlers).

  Eine *Epoche* ist die nächst höhere Stufe. Hierunter versteht man das einmalige Anlegen aller Trainingsmuster, die zum Anlernen des Netzes zur Verfügung stehen. Die Gewichtsänderungen werden (je nach verwendeten Algorithmus) entweder in jedem Zyklus vorgenommen oder in einem Schritt am Ende der Epoche.

  Die gesamte Trainingsdauer ergibt sich somit aus der Anzahl der benötigten *Epochen* mit der entsprechenden Anzahl *Zyklen*.

- Aufwand für einen einzelnen Zyklus

  Der Aufwand eines einzelnen Zyklus kann z.B. durch die Anzahl der elementaren Rechenschritte, wie Multiplikationen, Additionen usw., angegeben werden. Dabei ist noch zu beachten, dass die Dauer eines Rechenschritts nicht immer gleich ist, so braucht etwa eine Division mehr Aufwand als eine Addition. Genauso unterschiedlich können die Größen der zu berechnenden Werte sein, was sich ebenfalls auf die Zeitdauer auswirkt.
- Skalierungseigenschaften des Algorithmus

  Dieses Kriterium kann man mit folgender Frage erklären:

  „Wie wächst der Zeitbedarf des Lernverfahrens mit wachsender Netzgröße, wie mit wachsender Menge der Trainingsmuster?“ ([Zell] S.415). Bei genauerer Überlegung stellt man jedoch fest, das dieser Punkt mit den bereits beschriebenen (Anzahl der Zyklen und Epochen, Aufwand für einen Zyklus) abgedeckt wird.

Umgang mit großen Datenmengen

In der Industrie sind oft riesige Datenmengen in Datenbanken gespeichert. Ein gutes Beispiel ist ein Mobilfunkunternehmen mit teilweise Millionen von Kunden. Für die Bearbeitung solch riesiger Datenmengen braucht man entweder große Rechenleistung oder effiziente Algorithmen.

Eine andere Möglichkeit besteht in der Komprimierung (Verdichtung) der Daten. Der hochdimensionale Eingangsvektor wird dabei in einen niederdimensionalen Vektor „umgewandelt“. [Hoffmann] nennt hier als typisches Beispiel das LVQ-Verfahren (siehe Abschnitt 3.10). Die ursprüngliche Datenmenge wird auf wenige Prototypen zusammengefasst und zu jedem Prototyp eine bestimmte Klasse mit angegeben. Das Netz braucht dadurch nicht mehr die Gesamtmenge der Datensätze abzuspeichern sondern es passt lediglich die Prototypen der gegebenen Datenmenge an und speichert sie ab.

Damit diese Art der Datenverdichtung zufriedenstellend funktioniert, muss die Ausgangsmenge Redundanzen aufweisen, d.h. das zwischen einzelnen Datensätzen große Ähnlichkeiten existieren. Bei praktischen Anwendungen ist dies in der Regel der Fall. Z.B. gibt es in den Datenbanken der Mobilfunkunternehmen sehr viele 14-17 jährige männliche Schüler.

Generalisierungsfähigkeit

Dies ist ebenfalls ein wichtigstes Kriterium bei der Bewertung eines neuronalen Netzes für eine praktische Anwendung. Die Fähigkeit zur Generalisierung gibt an, wie gut ein Netz bisher unbekannte Eingaben verarbeitet, also wie gut die gefundene Lösung ist. Im speziellen Fall der Klassifikation ist dies das Maß, wie gut ein Netz neue, unbekannte Eingabevektoren den bekannten Klassen richtig zuordnet.

Dieser Punkt ist sehr stark abhängig von der konkreten Anwendung und damit von der Struktur der Daten (verschiedenartige Eingangsdaten) und der Verteilung und der Menge der Trainings- und Testdaten. Hier tritt sehr schnell das Problem des *Overfittings* auf. Passt sich ein Algorithmus zu sehr an die Trainingmenge an, so ist er unter Umständen für zufällige Testmengen unbrauchbar, da er große Teile dieser Daten dann falsch klassifiziert.

Einen großen Einfluss hat dabei die Menge der Daten, die das Netz lernt und die Größe des Netzes (ist ein Zeichen für die Anpassungsfähigkeit des Netzes). Bei wenigen Musterpaaren wird das Netz die Muster exakt abspeichern, bei sehr vielen wird es lediglich Gemeinsamkeiten der Datensätze erkennen und dazugehörige „Prototypen“ abspeichern. Große, komplexe Netze können wiederum mehr Muster abspeichern als kleine einfache Netze. „Daher lernen große Netze gut und generalisieren schlecht (*overtraining*); bei kleinen Netzen ist es umgekehrt. Die Größe des Netzes ist dabei in Relation zum Umfang des Problems zu sehen.“ ([Hoffmann] S.143; [Bellido])

Übliche Messdaten für die Fähigkeit unbekannte Daten zu generalisieren, erhält man z.B. mit der Anzahl der korrekt bzw. falsch klassifizierten Datensätze auf einer zur Lernmenge disjunkten zufälligen Testmenge (also einer Überprüfung des Ergebnisses auf bekannten Testdaten).

Nachtrainierbarkeit

Ein Vorteil neuronaler Netze gegenüber herkömmlichen Klassifikationsverfahren, wie z.B. Entscheidungsbäumen, ist die Fähigkeit zum Nachtrainieren des Netzes mit neuen Datensätzen. Hat ein neuronales Netz einen Lernvorgang abgeschlossen, so sollte es möglich sein ohne größere Probleme noch einige weitere Daten zu lernen.

Hier kommt ein Problem zu Geltung, dass unter den Namen *Stabilitäts-Plastizitäst-Dilemma* bekannt ist. Eine kurze Ausführung dazu findet man im Abschnitt 2.4.3 eine genauere bei [Protzel].

Umgang mit verschiedenartigen Eingangsdaten

Jeder dem Netz präsentierte Eingangsvektor stellt in der praktischen Anwendung einen Datensatz dar. Die einzelnen Elemente sind die Merkmale des Datensatzes. Diese Merkmale können sowohl kategorischer (also mit einer bestimmten Zahl von Ausprägungen) als auch numerischer Art sein. Binäre Merkmale sind im Grunde genommen nur ein Spezialfall der kategorischen Merkmale, mit nur zwei Ausprägungen, die in der Regel 0 und 1 sind.

In der Praxis stellen solche Datensätze meist Personen, Messdaten oder andere Objekte dar und die Merkmale beschreiben diese. Aus diesem Grund ist es wichtig, dass der Algorithmus entweder mit verschiedenartigen Eingabewerten umgehen kann oder eine Anpassung der Eingabedaten auf den Algorithmus vorgenommen wird.

In einigen Netzen wird vorrausgesetzt, dass die Eingangsdaten in normierter Form vorliegen. Die verschiedenartigen Eingangsdaten auf ein einheitliches Maß zu normieren verbessert in den meisten Fällen die Funktionalität des Netzes und damit entscheidend das Ergebnis. Ein Beispiel ist die 0/1 – Normierung, bei der alle Merkmale auf den Wertebereich [0, 1] abgebildet werden.

Umgang mit verrauschten Daten

Nicht alle in der Industrie vorkommenden Daten sind vollständig oder 100%ig korrekt abgespeichert. Bei Umfragen in der Bevölkerung machen z.B. auch viele Leute ungenaue oder absurde Angaben oder lassen Angaben aus persönlichen Gründen ganz weg. Es besteht jetzt zum einen die Möglichkeit, die Eingabewerte vorher zu bereinigen (was auch meist gemacht wird) oder dies dem Netz zu überlassen.

Fehlende Werte sollten im Vorfeld schon als solche markiert werden. Man sollte fehlenden Angaben am zweckmäßigsten sinnvolle Werte zuordnen, beim Alter etwa den Mittelwert aller vorhandenen Datensätze. So wird das Lernergebnis des neuronalen Netzes am wenigsten beeinflusst.

Anders verhält es sich bei – extremen – Ausreißern. Diese sind im Vorfeld nicht so einfach zu erkennen, da sie ja gültige Werte darstellen, aber doch keinen Sinn ergeben. Ein gutes Beispiel sind „600jährige Rentner“, bei denen versehentlich eine Null zuviel angegeben wurde. Solche Werte innerhalb der Eingabemerkmale sollten sich nicht gleich auf die Struktur des ganzen Netzes niederschlagen, sondern vom Lernalgorithmus so gut wie möglich „ignoriert“ werden. Auch hier eignet sich z.B. ein Netz, dass mit normierten Werten arbeitet. Ausreißer werden dabei einfach auf die Grenzen des Wertebereichs abgebildet.

Bei unbekannten Daten, die es zu klassifizieren gilt, ist der Umgang mit verrauschten Eingabedaten ebenfalls von besonderer Wichtigkeit. Hier kann man dies mit dem Punkt Generalisierungsfähigkeit erklären. Ungenaue oder unvollständige Eingaben sollten vom Netz dennoch richtig erkannt und behandelt werden. Diese Fähigkeit ist ein großer Vorteil neuronaler Netze gegenüber anderen Methoden (siehe Abschnitt 1.2.1).

Geringer Speicherverbrauch

Bei heutigen Rechnern spielt diese Problematik nicht so eine bedeutende Rolle, jedoch sollte sie auch nicht ganz unbeachtet bleiben. Man kann hier zwischen Festplattenspeicher und benötigtem Hauptspeicher unterscheiden. Der Festplattenspeicher ist nur für die Speicherung/Installation des Programms von Bedeutung und kann in Zeiten der Festplatten mit GB-Kapazitäten vernachlässigt werden.

Der Hauptspeicherverbrauch eines laufenden Programms ist schon von größerem Interesse. Herkömmliche Personalcomputer stoßen auch heute noch schnell an ihre Grenzen, was hauptsächlich am großen Umfang der zu verarbeitenden Daten liegt. Der Speicherverbrauch hängt neben dem Netztyp auch wesentlich von der Implementierung des jeweiligen Netzes und Lernverfahrens ab. So ist es z.B. in den meisten Fällen sinnvoller, die einzelnen Neuronen nur durch die Gewichtsmatrix darzustellen, anstatt jedes Neuron einzeln abzuspeichern und in ihnen die Berechnungen durchzuführen.

Der Speicherbrauch wirkt sich auch direkt auf die Schnelligkeit des Algorithmus aus. Ist die Speicherkapazität eines Computers erschöpft, so ist das Programm oder der Computer gezwungen Speicher wieder freizugeben oder auszulagern, was wiederum Zeit in Anspruch nimmt und somit die Ausführung verlangsamt.

Im Allgemeinen kann man sagen, dass ein größeres und komplexeres Netz (also mit mehreren Ebenen und sehr vielen Verbindungen) mehr Speicher und damit mehr Zeit zum Lernen benötigt. Negativ auf den Speicherverbrauch wirkt sich auch aus, wenn der Algorithmus fordert, die Gewichtsänderungen oder Eingangsvektoren zwischenzuspeichern. Dies ist z.B. der Fall, wenn die Änderung aller Gewichtsvektoren am Ende einer Epoche geschehen soll, statt nach jedem Zyklus.

Komplexität

Zusammenhänge zwischen den einzelnen Merkmalen bzw. Datensätzen sind in der Regel nicht sofort zu erkennen. Oftmals sind es komplexe mathematische Funktionen, die Beziehungen in den einzelnen Klassen beschreiben. Ein sehr einfaches Netz stößt beim Anlernen dabei schnell an seine Grenzen, da es solche Beziehungen nicht erkennt. Das beeinflusst in erheblichem Maße die Qualität des Ergebnisses. Abschnitt 4.3 zeigt dafür ein einfaches Beispiel.

Das neuronale Netz sollte in der Lage sein, auch komplexe mathematische Zusammenhänge zu erkennen, dabei aber aufgrund von Geschwindigkeitsanforderungen möglichst einfach aufgebaut sein. Je komplexer ein Netz ist, desto besser passt es sich an die Daten an, desto länger dauert in der Regel aber auch die Berechnung. Auch hier gilt es einen guten Kompromiss zu finden.

Genauigkeit der Ausgabe

Die Genauigkeit kann man nicht gerade als Stärke neuronaler Netze nennen (siehe Abschnitt 1.2.2), sie steht im Gegensatz zur Generalisierungsfähigkeit. Gibt man einem neuronalen Netz eine einfache Rechenaufgabe der Art „4.1 + 5.4“ vor, so wird es mit „ungefähr 10“ antworten ([Haun] S.19).

Es gilt also ein Netz zu finden, das Genauigkeit der Ausgabe und Generalisierungsfähigkeit vereint. Auch wenn die Fähigkeit, unbekannte und z.T. unvollständige Daten zu erkennen und zuzuordnen (Generalisierung) die wichtigere ist, kann man den Punkt Genauigkeit doch nicht unbeachtet lassen.

Würde ein Netz nicht genau genug arbeiten, so könnte es z.B. vorkommen das ein vorher gelernter Datensatz beim späteren Anlegen falsch erkannt wird (Fehlklassifikation). So sollten auch Eingabevektoren die einander ähnlich sind, jedoch einer unterschiedliche Zielklasse angehören nicht dem gleichen „Prototyp“ zugeordnet werden. Also muss die Trennung der Klasse auch an solchen Stellen des Eingaberaumes „scharf genug“ sein.

Qualität der Ergebnisse

Die Qualität des Ergebnisses lässt sich bei neuronalen Netzen durch die Fähigkeit zur Generalisierung und die Erfolgswahrscheinlichkeit erklären.

Das Netz kann sich z.B. so gut der Eingabemenge anpassen, dass diese fast exakt wieder gegeben werden kann (das zeugt von einer guten Lernqualität) andererseits aber unbekannte Daten nicht mehr gut zugeordnet werden können, was mit dem Begriff *Overfitting* bereits beschrieben wurde. Für die praktische Anwendung ist jedoch die Generalisierungsfähigkeit von entscheidender Bedeutung, so dass die Qualität eines Netzes daran gemessen werden sollte. Somit sollte sich das Netz nicht zu gut an die Eingabemenge anpassen.

Ebenso ist es zwar positiv, wenn der Algorithmus eine Lösung findet, wenn diese aber nur ein lokales Optimum (Minimum) darstellt ist die Qualität schlechter als bei einem Algorithmus, der das globale Optimum als Lösung ausgibt.

Erfolgswahrscheinlichkeit des Netzes

Die Erfolgswahrscheinlichkeit gibt an, ob ein Lernalgorithmus auch wirklich zu einem Ergebnis gelangt, der Algorithmus muss also konvergieren. Dies stellt vor allem bei rückgekoppelten Netzen ein Problem dar, da bei diesen häufig die Gefahr besteht, dass sie sich „aufschaukeln“ .

Weiterhin sollte der Algorithmus nicht in einem lokalen Optimum stecken bleiben sondern das globale Optimum der Lösung finden. Dies ist ein Problem, was z.B. bei der Fehlerfunktion von Backpropagation-Netzen (siehe Abschnitt 3.7) häufig auftritt.

4.2.2 Auswahlmethodik

Wie bereits erwähnt gibt es kein einheitliches Schema, das dem Anwender genau sagt, was für ein neuronales Netz er für welchen Problemtyp verwenden soll. Es gibt viele verschiedene Probleme die mit neuronalen Netzen gelöst werden können. Es kommen auch für solche Lösungen mehrere verschiedene Typen von Netzen in Frage. [Hoffmann] hat versucht, typische reale Anwendungen zu nennen und für jede einen geeigneten Netztyp mit anzugeben. Allerdings ist dieses „Vorgehen nach Schema“ nicht so einfach, da gewisse Probleme sich meist nur im Detail unterscheiden. Ein gutes Beispiel hierfür ist gerade die einfache Aufgabe der Klassifikation, die wiederum nur ein Teilproblem der Mustererkennung (laut [Hoffmann]) ist. Wie man aus Abschnitt 4.1 erkennt kann diese Aufgabe verschieden aufgefasst werden und somit mit komplett verschiedenen Methoden (überwacht oder unüberwacht lernend) gelöst werden.

Ein LVQ-Netz wird z.B. von [Zell] als typischer Klassifizierer gesehen (somit als eine Art heteroassoziativer Speicher), während [Hoffmann] es eher zur Datenverdichtung und Datenübertragung vorschlägt, bei der im Idealfall der Eingangsvektor und Ausgangsvektor übereinstimmen (also eine Art autoassoziativer Speicher).

Eines steht jedoch fest:

> Das hier behandelte Problem erfordert ein überwachtes Lernverfahren und ein Netz, das als heteroassoziativer Speicher funktioniert.

Dabei ist aber nicht ausgeschlossen, dass ein anderer Netztyp (auch einer der nicht betrachtet wurde), die gegebene Aufgabe nicht auch lösen kann. Die Eingabe bzw. die Struktur kann man unter Umständen auch auf das gegebene Problem anpassen, auch wenn dies wenig sinnvoll erscheint, da man ja ein Netz auswählen sollte, das zu dem gegebenen Problem am besten passt, ohne die Struktur oder Eingabedaten großartig zu modifizieren, was wiederum Zeitverlust bedeutet.

So kann z.B. ein autoassoziativer Speicher unter Umständen auch als Klassifizierer dienen. Man muss nur den Eingabevektor und seine Klasse zusammenfassen und dann dem Netz als Eingabe vorlegen. Die zugehörige Klasse wird dabei mit abgespeichert und am Ausgang erscheint neben der Klasse dann auch der Vektor mit. Ob das Sinn ergibt, weil ein heteroassoziativer Speicher die Aufgabe einfacher löst, ist dabei eine andere Frage.

Es bleibt also nichts weiter übrig, als Netze die diese Anforderung nicht erfüllen von vornherein aus der Betrachtung auszuschließen und die übrig gebliebenen anhand einiger im Abschnitt 4.2.1 dargestellten Kriterien zu bewerten.

Diese Bewertung ist allerdings auch nicht so einfach wie es scheint. Bei vielen Netzen kennt man von vornherein nicht die benötigte Struktur. Ein Backpropagation-Netz hat z.B. eine bestimmte Anzahl versteckter Schichten. Aber wie viele sind das und wie viele Neuronen pro Schicht sind optimal für die gestellte Aufgabe. Die Anzahl der Schichten und Neuronen bestimmt im Wesentlichen die Dauer eines Trainingszyklus. Backpropagation-Netze haben z.B. das Problem, dass sie bei einer tiefen Netzstruktur (schon bei 2 bis 3 verdeckten Schichten) nur noch sehr geringe Gewichtsänderungen in den ersten Schichten vornehmen, was zu einem sehr langsamen Trainingsprozess führen kann ([Zell] S.418). Andere Netztypen können ihre Topologie während des Lernvorganges sogar ändern, man kann sie also mit Netzen, die eine statische Struktur haben, schlecht oder gar nicht vergleichen.

Hat man sich auf eine Struktur eines Netzes geeinigt kommt ein weiterer wichtiger Punkt zum Tragen, die geeignete Wahl der Parameter. Während bei einigen Netzen nur ein Parameter vorgegeben werden muss (z.B. die Lernrate η) sind bei anderen Typen mehrere Parameter zu bedenken (z.B. bei der Boltzmann-Maschine noch die „Temperatur“ T).

Ein nächster kritischer Punkt ist die Anfangsinitialisierung der Gewichte. Man kann nicht genau sagen, welche die beste Methode ist, es lässt sich lediglich bei einigen Anwendungen eine Einschränkung angeben. Beim Backpropagation-Netzwerk sollten diese zufällig, voneinander verschieden und kleiner 1 sein. Beim LVQ-Netz sollten die Codebook-Vektoren am Anfang ebenfalls möglichst zufällig im Raum verstreut sein, jedoch so, dass es möglichst keine „dead neurons“ gibt (siehe dazu 3.10.6). Ist die Auswahl der Gewichte aber zufällig, kann es ebenso passieren, dass man eine sehr ungünstige Belegung erhält.

4.2.3 Erste Einschränkung der Auswahl

Die vorangegangenen Überlegungen zeigen, dass ein Vergleich der Netze nicht ohne weiteres möglich ist. Im Endeffekt ist es dem Anwender selbst überlassen, für welches Netz er sich entscheidet. Dabei ist es nicht ausgeschlossen, dass andere Untersuchungen eine andere Vorgehensweise benutzen. Die weiteren Betrachtungen erfolgen also nach meinem persönlichen Ermessen.

Zunächst treffen wir eine Vorauswahl nach der Arbeitsweise der Netze (siehe Abschnitt 4.1.1):

Netztyp	**Beschreibung**	**Arbeitsweise (Speichermodell)**	**Kommt für das Problem in Frage?**
Allgemeiner Muster-Assoziator	Abschnitt 3.1	Heteroassoziativ	Ja
Perzeptron	Abschnitt 3.2	Heteroassoziativ	Ja
ADALINE	Abschnitt 3.3	Heteroassoziativ	Ja
MADALINE	Abschnitt 3.4	Heteroassoziativ	Ja
Allgemeiner Auto-Assoziator	Abschnitt 3.5	Autoassoziativ	Nein
Hopfield-Netz	Abschnitt 3.6	Autoassoziativ	Nein
Backpropagation-Netz	Abschnitt 3.7	Heteroassoziativ	Ja
BAM	Abschnitt 3.8	Heteroassoziativ	Ja
Boltzmann-Maschine	Abschnitt 3.9	Heteroassoziativ	Ja
LVQ	Abschnitt 3.10	Heteroassoziativ	Ja
Counterpropagation-Netz	Abschnitt 3.11	Heteroassoziativ/ autoassoziativ	Ja

Tabelle 4-1: Erste Aussonderung von Netzen anhand des Speichermodels

4.3 Grenzen einfacher Netze

Im Folgenden sollen anhand eines Beispiels die Grenzen von einfachen, einschichtigen Netzen aufgezeigt werden. In [Hoffmann] werden Beispiele dieser Art treffenderweise als *Schulprobleme* bezeichnet, da sie von großem inhaltlichem Lehrwert sind. Sie werden in der Literatur über neuronale Netze immer wieder behandelt, spielen aber bei praktischen Anwendungen keine große Rolle.

4.3.1 Das XOR-Problem

Das XOR-Problem ist ein sehr einfaches bekanntes Beispiel. Es ist unter den Autoren so beliebt, dass es in sehr vielen Veröffentlichungen als einführendes Problem verwendet wird. Es ist klein genug, um im Kopf durchgerechnet werden zu können, aber gleichzeitig mächtig genug, um zu zeigen, dass ein einfaches, einschichtiges Netz diese Aufgabe nicht lösen kann.

Beschreibung

Die Aufgabe besteht darin, die logische Funktion XOR (Exclusiv-ODER-Verknüpfung) durch ein neuronales Netz darzustellen. Die folgende Tabelle zeigt die Musterpaare und die zugehörigen Eingangs- und Ausgangswerte:

Musterpaar	E_1	E_2	O
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Tabelle 4-2: Variablenbelegung der XOR-Funktion

Das Netzwerk benötigt für die Aufgabe nur die binären Werte {0, 1} für die Neuronen. Als Aktivierungsfunktion reicht also eine einfache Schwellenwertfunktion (siehe Abbildung 2-2) mit einem geeigneten Schwellenwert Θ aus.

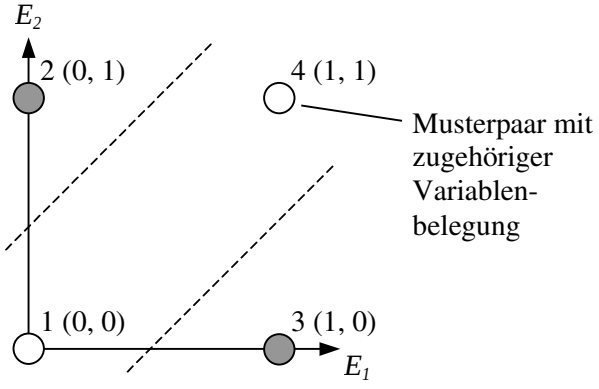


Abbildung 4-1: Grafische Darstellung des XOR-Problems

Die Eingangsmuster besitzen nur Dimensionen, die linear voneinander abhängig sind. Das Problem ist also nicht linear teilbar, was auch aus Abbildung 4-1 hervorgeht. Die einzelnen Klassen (dargestellt durch verschiedenfarbige Punkte) sind nicht zusammenhängend. Im linearen Raum würde man eine Gerade benötigen, die die dargestellte Ebene so trennt, dass auf der einen Seite nur die Einsen und auf der anderen Seite nur die Nullen liegen. Man kann leicht erkennen, dass dies nicht möglich ist und man mindestens zwei verschiedene Geraden benötigt.

Lösungen

Zur Lösung dieses Problems mit einfachen Neuronen benötigt ein vorwärtsgekoppeltes Netz mindestens eine Schicht verdeckter Neuronen. In Abbildung 4-2 wird dafür ein feedforward-Netz mit shortcut connections verwendet. Die Schwellenwerte sind in der Abbildung mit in die Neuronen eingetragen. Durch die Schwellenwertfunktion lassen sich die Ausgangswerte der einzelnen Neuronen leicht bestimmen, indem einfach die Eingangswerte addiert und mit dem Schwellenwert verglichen werden.

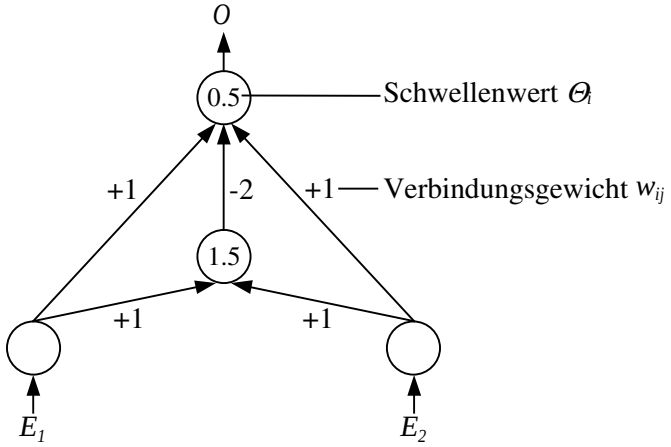


Abbildung 4-2: Beispiel eines minimalen Netzes für das XOR-Problem

Man kann relativ einfach nachrechnen, dass das dargestellte Netz die Aufgabe erfüllt.

Eine Lösung des Problems ohne *shortcut connections* (also ein einfaches feedforward-Netz) spaltet die XOR-Funktion in die einfachen boolschen Funktionen AND und OR auf:

$$XOR(E_1, E_2) = \overline{E_1} E_2 + E_1 \overline{E_2}\,.$$

Das zugehörige neuronale Netz sieht dann wie folgt aus:

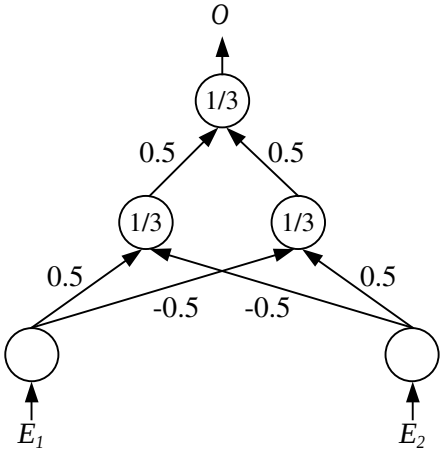


Abbildung 4-3: Weiteres Beispiel eines minimalen Netzes für das XOR-Problem

Die verdeckten Neuronen realisieren dabei die beiden AND-Funktionen

$$o_1 = E_1 \; AND \; \overline{E_2} \text{ und } o_2 = \overline{E_1} \; AND \; E_2$$

und das Ausgangsneuron die OR-Verknüpfung der beiden Teilergebnisse:

$$O = o_1 \; OR \; o_2\,.$$

Man kann feststellen, dass die beiden verdeckten Neuronen die Unterscheidung zwischen den Klassenpunkten {(0,1)} / {(0,0), (1,0), (1,1)} bzw. {(1,0)} / {(0,0), (0,1), (1,1)} realisieren und

das Ausgangsneuron die Zustände dieser Neuronen mit OR verknüpft und somit auf die Klasse schließt.

Erkenntnis

Man sieht in beiden Fällen, dass ein Netz aus einfachen Neuronen mindestens eine Schicht verdeckter Neuronen benötigt. Weiterhin kann man erkennen, dass eine komplexe mathematische Funktion mit Hilfe mehrschichtiger feedforward-Netze durch überlagerte Einzelfunktionen darstellbar ist. Die Anzahl der Schichten und die Anzahl der Neuronen pro Schicht hängt stark vom betrachteten Problem ab, eine eindeute Lösungsmethodik gibt es aber nicht. Als Lernverfahren für dieses Problem wird das Backpropagation-Verfahren verwendet.

Hinweis

Eine alternative Möglichkeit bieten neuronale Netz mit sogenannten *Sigma-Pi-Neuronen*, von denen für die Lösung des Problems sogar nur ein einziges Neuron benötigt wird (in dieser Arbeit habe ich diese Neuronen nicht behandelt, da man aus Gründen der Komplexität Abstriche an bestimmten Stellen vornehmen muss). Das sind Neuronen „höherer Ordnung“, die als Eingangsfunktion eine Summe aus Produkten der Eingangswerte verwenden. Die Eingangsfunktionen werden dabei nach dem Prinzip der Taylorreihe entwickelt. Weitere Informationen dazu findet man z.B. bei [Hoffmann] S.17 und S110ff. Die hier vorgestellten Neuronen sind im Grunde genommen ein Spezialfall 1.Ordnung dieses Neuronentyps.

4.3.2 Weitere Schulprobleme

Es gibt noch eine Reihe weiterer solcher Beispielprobleme, die hier aber nur kurz angeführt werden sollen. Genauere Beschreibungen und Untersuchungen zu diesen Aufgaben findet man z.B. bei [Hoffmann] S.146ff.

Paritätsproblem

Dieses Problem ist eine Verallgemeinerung des XOR-Problems. Ein gegebener Eingangsvektor aus Nullen und Einsen soll daraufhin untersucht werden, ob er eine gerade oder ungerade Anzahl von Einsen enthält.

Man kann sich leicht überlegen, dass ein einfaches einschichtiges Netz (bestehend aus einfachen Neuronentypen) diese Aufgabe nicht lösen kann, ein zweischichtiges Netz jedoch löst dieses Problem leicht. Die Anzahl der Neuronen der ersten Schicht ist natürlich gleichgesetzt mit der Dimension des Eingangsvektors und der Netzausgang besteht nur aus einem einzigen Neuron. Als Lernalgorithmus kann der Backpropagation-Algorithmus verwendet werden, da es im Grunde genommen ein heteroassoziatives Problem ist (genauer betrachtet ist es sogar ein Klassifikationsproblem).

Interessant an diesem Problem ist, dass die erforderliche Anzahl von Lernschritten sehr groß werden kann. Die Parität ist ein Problem, dass ein neuronales Netz sehr genau lösen muss. Es wurde jedoch bereits in Abschnitt 4.2.1 darauf hingewiesen, dass die Angabe von exakten Werten keine Stärke neuronaler Netze ist. Eingangswerte die sich nur in einer einzigen Komponente unterscheiden und somit einander sehr ähnlich sind, müssen hier zu unterschiedlichen Ausgangswerten führen.

Dieses Beispiel ist u.a. auch bei [Rumel] beschrieben. [MinPap] wies nach, dass dieses Problem „mit einem linearen Perzeptron (einem feedforward-Netz mit nur einer Ebene trainierbarer Gewichte) nicht lösbar ist“ ([Zell] S.421).

Encoder-Decoder-Problem

Beim Encoder-Decoder-Problem besteht die Aufgabe darin, einen Eingangsvektor unverändert zum Netzausgang durchzuschalten. Es gilt jedoch die Bedingung, dass der Signalfluss durch eine Schicht verborgener Neuronen läuft, deren Anzahl kleiner ist als die der Netzeingänge. Das

zugehörige Netz soll ein feedforward-Netz sein, ohne shortcut connections. Also ist die Ausgangsschicht nur mit den verdeckten Neuronen verbunden.

Als Lösung für dieses Problem eignet sich somit wegen der einfachen feedforward-Struktur ein Backpropagation-Netz. Das ist aber gleichzeitig der interessante Aspekt an diesem Problem. Da das Eingangsmuster gleich dem Ausgangsmuster sein soll, arbeitet das Netz eigentlich autoassoziativ. Da es die Muster jedoch nicht speichern sondern lediglich weiterleiten soll, zeigt es nicht die typische Fähigkeit eines autoassoziativen Netzes, die Musterergänzung. Weiterhin arbeitet das Netz vorwärtsgekoppelt und nach dem Backpropagation-Algorithmus, was eigentlich auf ein heteroassoziatives Problem schließen lässt.

Eine Beschreibung dieses Problems findet man z.B. bei [Hertz] und [Rumel].

4.3.3 Einschränkung durch diese Betrachtung

Nach dieser Betrachtung eines relativ einfachen Problems lässt sich die Liste der geeigneten Netztypen aus Tabelle 4-1 weiter einschränken. Da in realen Datenbanken die Beziehungen der Daten untereinander oft nur mit komplexen mathematischen Funktionen zu beschreiben sind, können wir bei der Auswahl einfache, einschichtig-vorwärtsgekoppelte Netze wegfallen lassen.

Schränken wir die Liste also weiter ein:

Netztyp	**Struktur**	**Bemerkung**	**Ist geeignet?**
Allgemeiner Muster-Assoziator	Einschichtig	Kann nur linear trennen	Nein
Perzeptron	Einschichtig bis Dreischichtig	Auch komplexere Funktionen beschreibbar, wenn mehr-schichtig	Ja
ADALINE	Einschichtig	Kann nur linear trennen	Nein
MADALINE	Zweischichtig	Erweiterung des ADALINE, kann auch nichtlinear trennen	Ja
Backpropagation	Mehrschichtig	Beliebig viele Schichten möglich	Ja
BAM	Zweischichtig, rückgekoppelt	Durch Rückkopplung Fähigkeit zur Musterspeicherung	Ja
Boltzmann-Maschine	Mehrschichtig, rückgekoppelt	Durch Rückkopplung und speziellen Lernalgorithmus gut geeignet	Ja
LVQ	Einschichtig	Durch speziellen Lernalgorith-mus wird ein Prototyp zu den einzelnen Klassen erzeugt	Ja
Counterpropagation-Netz	Mehrschichtig	Kombination zweier verschiedener Netze	Ja

Tabelle 4-3: Einschränkung der Auswahl durch nichtlineare Teilbarkeit

Eine Frage der Komplexität?

Aus den bisherigen Betrachtungen geht hervor, dass für eine praktische Anwendung die einfach aufgebauten Netze nicht in Frage kommen. Mit „einfach aufgebaut“ sind die Grundtypen (Allgemeiner Muster-Assoziator, Allgemeiner Auto-Assoziator, ADALINE) der Netze gemeint, auf denen andere Typen aufbauen. Am Beispiel des LVQ-Netzes sieht man dagegen, dass mit einen geeigneten Lernverfahren (das LVQ-Netz ist ein vektorbasiertes Netz und erlaubt somit eine komplexere Trennung der Klassen) das gewünschte Netz nicht sonderlich kompliziert aufgebaut sein muss. Deshalb folgt eine Betrachtung der einzelnen Lernverfahren.

4.4 Betrachtung der Lernverfahren

4.4.1 Hebbsche Lernregel und Delta-Regel

Wie aus Abschnitt 4.3 zu erkennen war, kommen bei einfachen feedforward-Netzen nur mehrschichtige Netze in Frage. Ein Beispiel hierfür ist das sogenannte „2-Spiralen-Problem“ (siehe [Zell] S.422), ein relativ einfach zu generierendes, aber für neuronale Netze recht schwierig zu lösendes Problem. Je 97 Messpunkte sind auf zwei ineinander verschränkte Spiralen verteilt (drei Umdrehungen mit je 32 Punkten plus Endpunkte), wobei jeder Punkt durch seine (x, y)-Koordinate beschrieben wird. Die Ausgabe ist ein typischer Fall für eine Klassifikation mit zwei Zielklassen (Zugehörigkeit zu einer der zwei Spiralen). [LanWit] zeigten, dass diese Aufgabe für Backpropagation-Netze mit wenigen verdeckten Ebenen schwierig zu lösen ist. Ein 2-5-5-5-1 Backpropagation-Netz mit shortcut-connections benötigte ca. 20.000 Epochen, ein 2-20-10-1 Netz ohne shortcut-connections ca. 13.900 Epochen.

Wie bereits in Abschnitt 2.4.1 und Abschnitt 2.4.2 erwähnt wurde, sind die Hebbsche Lernregel und die Delta-Regel nur auf Ausgangsneuronen oder auf einschichtige neuronale Netze anwendbar. Da für diese Aufgabe bei einfachen, vorwärtsgekoppelten Netzen nur mehrschichtige Netze in Frage kommen (also mit verdeckten Neuronen), scheiden Netze mit diesen Lernverfahren oder einfachen Varianten davon aus (damit ist jedoch nicht die Backpropagation-Lernregel gemeint, da diese auch auf verdeckte Neuronen anwendbar ist, obwohl sie von der Idee her auf der Delta-Lernregel aufbaut).

4.4.2 Backpropagation

Der Backpropagation-Algorithmus ist ein Gradientenabstiegsverfahren, das die Gewichtsänderung anhand einer Zielfunktion berechnet. Die Zielfunktion ist in diesem Fall die Fehlerfunktion, an der sich der Gradient nach unten bewegt, bis ein Minimum erreicht ist. Eine ausführliche Herleitung dieses Algorithmus ist in [Zell] S108ff. zu finden.

Der große Vorteil von Backpropagation-Netzen ist sicherlich die Anpassungsfähigkeit. Da sich der Algorithmus auf beliebige Netzstrukturen anwenden lässt (durch Modifikationen ist er auch auf rückgekoppelte Netze anwendbar) ist es möglich, für nahezu jedes Problem ein geeignetes Netz zu entwickeln, indem man die Anzahl der Schichten und die Verbindungen variiert. Die Anzahl der Neuronen pro Schicht kann in fast jedem Netz verändert werden, die Anzahl der Schichten ist dagegen bei vielen Netztypen fest vorgegeben. Durch die beliebige Anzahl von verdeckten Schichten lassen sich somit komplexe mathematische Funktionen gut beschreiben.

Das Backpropagation-Verfahren ist zweifellos mit eines der am meisten verwendeten Lernverfahren für neuronale Netze, was sicherlich auf seine Anpassungsfähigkeit zurückzuführen ist. Allerdings besitzt es auch viele Nachteile, die -auszugsweise- kurz angeführt werden sollen. Eine ausführlichere Beschreibung dazu findet sich z.B. in [Zell] S.110ff. oder [Hoffmann] S.87ff.

Geschwindigkeit

Der Geschwindigkeitsnachteil des Backpropagation-Netzes entsteht hauptsächlich dadurch, dass der Lernalgorithmus aus zwei Teilen besteht. Zuerst wird das Eingangssignal in einem vorwärtsgerichteten Fluss zum Ausgang geführt (Reproduktion), danach wird der Fehler vom Ausgang zurück zum Eingang geführt und die Gewichte berechnet. Das Netz muss also für jedes angelegte Muster zweimal durchlaufen werden, was die Dauer eines einzelnen Zyklus im gegebenen Netz in etwa verdoppelt.

Die Tatsache, dass ein Backpropagation-Netz langsamer lernt als andere Netze (deren Algorithmus direkter auf die Netzstruktur angepasst ist) zeigen auch viele Benchmark-Tests. Für das in Abschnitt 4.4.1 erwähnte 2-Spiralen-Problem benötigt ein Cascade-Correlation-Netz 1.700 Lernepochen mit 19 M Multiplikations- und Additions-Operationen, ein Backpropagation-Netz dagegen 20.000 Lernepochen mit 1.100 M Multiplikations- und Additions-Operationen (siehe [Zell] S.166f.).

Ein Cascade-Correlation-Netz ist ein Netzmodell, dass neben den Gewichten auch die Topologie des Netzes mit ändert. Diese Netzstruktur wurde jedoch in dieser Arbeit nicht mit beschrieben. Dies liegt vor allem an der Art des Netzes, da der Lernalgorithmus auch die interne Struktur des Netzes mit ändert (diese Abstriche mussten gemacht werden, da der Inhalt sonst den Rahmen der Arbeit sprengen würde).

Symmetry Breaking

Unter diesem Begriff versteht man das Problem der Initialisierung der Startgewichte eines Backpropagation-Netzes. Der Algorithmus stellt die Forderung, dass die Startgewichte verschieden und ungleich 0 sein müssen. Werden die Gewichte alle auf 0 gesetzt (aus der einfachen Überlegung heraus, dass das Netz die korrekten Gewichte noch finden muss), kann der Lernalgorithmus keine unterschiedlichen Gewichte mehr ausbilden und schlägt somit fehl.

Hier sei aber erwähnt, dass das Problem der Anfangsinitialisierung auch bei anderen Netzen auftritt. Es sind meist nur Vorschläge oder gar nur Bedingungen für eine Anfangsinitialisierung gegeben, eine konkrete Entscheidung findet man eher selten.

Lokale Minima

In den Literaturbeschreibungen des Backpropagation-Verfahrens wird ein Thema immer wieder besonders hervorgehoben (z.B. bei [Zell] S.112). Gradientenverfahren haben das Problem, dass sie häufig in einem lokalen Minimum der Zielfunktion, also der Fehlerfunktion, hängen bleiben, anstatt weiter zum globalen Minimum zu tendieren.

Der Hauptgrund dafür ist, dass die Gewichtsänderung stets entlang des steilsten Abstiegs der Zielfunktion stattfindet. Befindet sich der aktuelle Zustand des Netzes erst einmal in einem „Tal“ (lokales Minimum), kann dieses somit auch nicht mehr in Richtung des globalen Minimums verlassen werden. Backpropagation ist also ein lokales Verfahren, das keine Informationen über die Fehlerfläche insgesamt hat.

Ob ein absolutes Minimum erreicht wird hängt hauptsächlich von der günstigen Wahl der Startgewichte ab, hierfür ist aber keine systematische Methode bekannt. Das Problem lässt sich lösen, wenn man von dem allgemeinen Schema der Fehlerrückführung abweicht, z.B. durch simuliertes Kühlen (simulated annealing).

Modifikationen

Zwar gibt es für ein Backpropagation-Netz viele Modifikationen der Lernregel (siehe [Zell] S.115ff.), diese lösen jedoch meist nur einige der Probleme des Backpropagation-Algorithmus. Die Rückpropagierung des Fehlers (Bestimmung der Fehlerfunktion wird bei Gradientenabstiegsverfahren immer benutzt) und damit der erhebliche Zeitaufwand zur Berechnung der Gewichte bleibt jedoch bestehen. Da bei großen Datenmengen die Berechnungszeit ein sehr wichtiges Kriterium ist, ist dieses Verfahren nicht unbedingt für die Aufgabe geeignet.

4.4.3 Hopfield-Regel im BAM

Beim BAM wird eine Variante der Hopfield-Regel zum Lernen verwendet, da es sich um eine Erweiterung des Hopfield-Netzes handelt. Der Nachteil der Erfolgswahrscheinlichkeit von rückgekoppelten Netzen tritt beim BAM nicht auf. Nach [Kosko] sind alle BAM’s für jedes Netzwerk stabil.

Ein entscheidender Nachteil des BAM ist die geringe Speicherkapazität. Dies wirk sich besonders negativ bei sehr vielen Eingangsvektoren aus, da das Netzwerk durch seine Struktur versucht die Eingangsmuster abzuspeichern. Nach [McElie] gilt in diesem Zusammenhang:

> Für die Berechnung der Speicherkapazität eines BAM wird vorausgesetzt, dass die Assoziationen in einer Kodierung vorliegen, welche die gleiche Anzahl von +1 bzw. –1 in jedem Vektor besitzt. Es seien *L* zufällig gewählte Assoziationen dieser Art der Kodierung vorhanden und die kleinere der beiden Schichten des BAM besitze *n* Neuronen. Wenn bis

auf einen kleinen Bruchteil (der Anteil der korrekten Assoziationen soll 98% betragen, also 2% Fehlklassifikationen) alle Assoziationen wiedergewonnen werden sollen, muss gelten:

$$L < \frac{n}{2\log n}.$$

Im Falle von 1024 Neuronen (also $n = 1024$) muss $L < 51$ sein, was natürlich extrem wenig ist (siehe [Zell] S.221).

Da das BAM-Netz vom Typ her immer noch ein rückgekoppeltes Netz ist (lehnt sich von der Struktur her an das Hopfield-Netz an), liegt ein weiterer Nachteil auf der Hand. Die Dauer eines Zyklus ist bei einem rückgekoppelten Netz deutlich höher als bei einem vorwärtsgekoppelten Netz.

Dies liegt daran, dass sich das Netz in einen stabilen Zustand einschwingen muss, da sich bei einem angelegten Eingangsmuster die Eingangswerte der Neuronen durch die interne Rückkopplung ständig ändern. Während eines Zyklus werden also die Gewichte mehrere male verändert, bis mit den vorgegebenen Parametern ein definiertes Abbruchkriterium erreicht wird.

4.4.4 Simulated Annealing

Dieses Verfahren wurde entwickelt, um das Problem der lokalen Minima bei Backpropagation zu umgehen. Durch Nutzung der Wahrscheinlichkeitsrechnung wird dem Netz die Möglichkeit gegeben die lokalen Minima zu verlassen und sich in einem globalen Minimum zu stabilisieren. Bei hohen „Temperaturen“ (große Werte des Parameters *T*) kann sich das System nahezu frei bewegen. Dabei werden alle möglichen Konfigurationen eingenommen. Durch langsame Abkühlung stabilisiert sich das Netz selbstständig, wobei die Wahrscheinlichkeit, dass sich das System dennoch in einem lokalen Minimum aufhält immer geringer wird. Somit erzielt dieses Netz zumindest gute Ergebnisse.

Wie bereits in Abschnitt 3.9.6 erwähnt, besteht der größte Nachteil einer Boltzmann-Maschine in der außerordentlich langen Rechenzeit. Dies ist vor allem auf die folgenden Faktoren zurückzuführen:

- Das angestrebte thermische Gleichgewicht stellt sich bei der Berechnung nur sehr langsam ein. Dies ist eine Folge der sehr aufwendigen Berechnung der beiden Werte $P^{\mu^+}_{ij}$ und $P^{\mu^-}_{ij}$.
- Die eigentliche Gewichtsänderung wird schließlich durch Differenz dieser Größen berechnet. Es handelt sich bei $P^{\mu^+}_{ij}$ und $P^{\mu^-}_{ij}$ jedoch um zwei fluktuierende Größen. Da diese wiederum in etwa die gleiche Größenordnung besitzen, ist es wichtig die Werte möglichst genau zu ermitteln.
- Weiterhin darf die „Abkühlung“ nicht zu schnell erfolgen, da sonst das System nicht das globale Minimum findet sondern in einem lokalen Minimum stecken bleibt. Die Änderung der Temperatur *T* muss also sehr langsam erfolgen (siehe [Brause] S.201f.).

4.4.5 LVQ

Der LVQ-Algorithmus ist ein von [Kohonen97] speziell für Klassifikationsaufgaben entwickeltes Verfahren. Das LVQ-Netz ist ein sehr einfach aufgebautes Netz, was dennoch durch die vektorbasierte Funktionsweise eine komplexe Trennung der einzelnen Klassen ermöglicht.

Für die Auswahl des LVQ-Algorithmus als Implementierung für Klassifikationsaufgaben spricht die Studie [ELENA]. Es wurden dabei Benchmarktests mit verschiedenen typischen Klassifikationsalgorithmen durchgeführt, wobei das LVQ-Verfahren recht gut abschnitt. Allein die Tatsache, dass andere -hier aufgeführte- Netztypen nicht mit in diesen Test einbezogen wurden, spricht für die Eignung des LVQ-Netzes als Klassifikator.

Weitere Punkte, die für die Eignung dieses Netzes zur Klassifikation sprechen, lassen sich schnell finden:

Schnelligkeit

Dieser Nachteil vieler bisher betrachteter Verfahren wird durch die einfache Struktur des Netzes und den einfachen Lernalgorithmus überwunden. Da es nur eine Schicht von Neuronen gibt, geschieht die Berechnung der Gewichte während eines Lernzyklus wesentlich schneller. Weiterhin müssen nur die Verbindungsgewichte des Gewinnerneurons und nicht aller Neuronen neu berechnet und angepasst werden, da die anderen Vektoren von der Berechnung unberührt bleiben.

Lediglich die Bestimmung des Gewinnerneurons nimmt einige Zeit in Anspruch, da jeder Codebookvektor mit dem Eingabevektor verglichen werden muss. Dies kann man evt. noch dadurch begrenzen, dass man nur die Menge der Vektoren betrachtet, die überhaupt für einen Vergleich in Frage kommen (etwa die Codebookvektoren, die der gleichen Klasse wie der Eingabevektor angehören).

Flexibilität

Wie in Abschnitt 4.2.1 angeführt, besteht ein Vorteil -und gleichzeitig eine Forderung- neuronaler Netze darin, dass diese später relativ leicht nachtrainiert werden können. Handelt es sich dabei um eine große Zahl neuer Vektoren oder liegen sie im Raum dort, wo sich keiner der bereits gelernten Vektoren befindet, gestaltet sich die Sache etwas schwieriger.

Für diesen Fall müsste man zum Netz wahrscheinlich einige neue Neuronen hinzufügen, also die interne Struktur des Netzes verändern. Das nachträgliche Anlernen erfordert somit in einem mehrschichtigen Netz unter Umständen auch eine Veränderung der bereits angepassten Gewichte, was wiederum zu dem im Abschnitt 2.4.3 beschriebenen Stabilitäts-Plastizitäts-Dilemma führt.

Beim LVQ gestaltet sich die Sache einfacher, in dem entsprechend neue Codebookvektoren zum Netz hinzugefügt werden. Da die Neuronen nicht dem Konzept einer Nachbarschaftsbeziehung unterliegen (und sich somit bei einem Lernschritt sich nur die Werte eines Neurons ändern und dieses andere Neuronen nicht beeinflusst) dürften auch keine Fehler auftreten. Theoretisch stellt es damit auch kein Problem dar, eine neue Zielklasse durch das Hinzufügen neuer Codebookvektoren einzuführen.

Eignung für große Datenmengen

Das LVQ-Verfahren zeigt eine weitere Stärke im Umgang mit großen Datenmengen. Anstatt zu versuchen, die Gesamtheit der Ein- bzw. Ausgabevektoren in der internen Struktur des Netzes abzuspeichern (wie z.B. aus Abschnitt 4.4.3 hervorgeht, ist die geringe Speicherkapazität ein Problem des BAM), bildet das Verfahren die Eingabemenge auf wenige Prototypen ab und passt diese den Eingaben an. Es wird also eine Art „Datenkompression“ durchgeführt.

Durch die, im Vergleich zur Menge der Eingabedaten, geringe Anzahl von Codebookvektoren folgt auch ein relativ geringer Speicherverbrauch, was sich wiederum positiv auf die Verarbeitungsgeschwindigkeit auswirkt.

4.4.6 Counterpropagation-Lernverfahren

Das Counterpropagtion-Netz besitzt u.a. folgende positive Eigenschaften (siehe Zell S.189):

- eine gegenüber Backpropagation in vielen Fällen stark reduzierte Trainingszeit,
- die Kombination von zwei Netzarten liefert Eigenschaften eines kombinierten Netzes, die keine der beiden Komponenten für sich alleine besitzt und
- es funktioniert als „look-up table“ mit guter Generalisierungsfähiglkeit indem es Eingabevektoren mit den zugehörigen Ausgabevektoren assoziiert.

Durch die kombinierten Lernverfahren erhalten wir mit dem Counterpropagation-Netz also sehr gute Ergebnisse, dafür ist dieser Netztyp nicht so allgemein anwendbar wie Backpropagation-Netze, was aber für diese spezielle Anwendung nicht so wichtig ist.

Die Kombination zweier verschiedener Netztypen gestaltet jedoch in erster Linie die Berechnung schwieriger und auch zeitaufwendiger, als bspw. bei einem LVQ-Netzwerk. Sicherlich stoßen wir bei diesem Netz wieder schnell an die Grenzen bei einer sehr großen Anzahl von Eingabevektoren, da ebenfalls versucht wird, die Eingabevektoren mit den entsprechenden Ausgabevektoren in der Struktur abzuspeichern.

Weiterhin liefert das Netz als Ausgabe stets einen kompletten Vektor in der Dimension des Eingabevektors, was natürlich bei einer Klassifikation (bei der die Klassenzugehörigkeit ausreicht) vollkommen unnötig ist. Dies geht natürlich zu Lasten der Performance.

4.5 Abschließende Auswahl

Aufgrund der Überlegungen aus Abschnitt 4.4 ergibt sich folgende abschließende (zum Teil intuitive) Auswahl eines Netztyps für die Eignung zur Klassifikation:

Netztyp	**Lernverfahren**	**Bemerkung**	**Geeignet?**
Perzeptron	Variante der Hebbschen Lernregel	Zu einfache Lernregel	Nein
MADALINE	Variante der Delta-Regel	Zu einfache Lernregel	Nein
Backpropagation	Backpropagation-Verfahren	Zu langsam, Problem der lokalen Minima	Weniger gut
BAM	Variante der Hopfield-Regel	Geringe Speicherkapazität und langsam	Weniger gut
Boltzmann-Maschine	Simulated Annealing	Liefert gute Ergebnisse, aber sehr langsam	Weniger gut
LVQ	LVQ-Algorithmen	Schnelles Verfahren, gute Ergebnisse, direkt für Klassifikation konzipiert	Sehr gut
Counterpropagation	Kombination zweier verschiedener Netztypen	Liefert gute Ergebnisse, aber nicht so schnell wie LVQ-Verfahren	Gut

Tabelle 4-4: Abschließende Auswahl eines Verfahrens

5 Implementierung und Erkenntnisse

In diesem Kapitel werden die wichtigsten Fragen und Probleme, die sich während der Softwareimplementierung ergaben, und deren Lösungsansätze vorgestellt. Über den realen Nutzen der LVQ-Implementierung (und damit des LVQ-Netztyps) im Vergleich zu anderen Klassifikationsmethoden lässt sich nur sehr schwer etwas sagen, da das übergeordnete Data-Mining-System (in das der Algorithmus eingebettet wird) noch nicht existiert. Somit ist auch eine Anwendung unter realen Bedingungen noch nicht möglich.

5.1 Initialisierung der Codebookvektoren

Auf die Initialisierung der Codebookvektoren muss beim LVQ-Netz (sowie bei allen anderen Netztypen) besonderes Augenmerk gelegt werden. Eine sinnvolle Auswahl der Startwerte des LVQ-Netzes beschleunigt die Berechnung und beeinflusst die Qualität des entstehenden Klassifikator erheblich.

5.1.1 Bedingungen für die Initialisierungswerte

Die Struktur bzw. Verteilung der Eingabedaten ist i.d.R. vor der Data-Mining-Analyse nicht bekannt. Für die korrekte Initialisierung eines neuronales Netzes muss die Eingabemenge deshalb hinsichtlich ihrer Zielklasse analysiert werden. Folgende Erkenntnisse sind dabei von Bedeutung:

- die Anzahl der Zielklassen,
- die Merkmalsausprägungen der Zielklassen und
- die Verteilung der Datensätze auf die einzelnen Zielklassen (Häufigkeiten).

Zur Bestimmung dieser Größen muss jeder Datensatz der Eingabedaten einmal eingelesen werden. Daraus resultieren folgende Bedingungen für die Anfangsinitialisierung der Codebookvektoren:

- jeder Zielklasse muss mindestens ein Codebookvektor zugeordnet werden,
- es sollte keine Codebookvektoren mit den gleichen Initialwerten geben und
- die Verteilung der Zielklassen der Codebookvektoren sollte der Verteilung in den Eingabedaten entsprechen.

In der Implementierung werden diese Forderungen bei der Initialisierung mit Werten aus Eingabedaten oder der zufälligen Initialisierung erfüllt (Initialisierungsmethoden werden in Abschnitt 5.1.2 beschrieben). Über den dritten Punkt, der Verteilung der Zielklassen, besteht jedoch noch Uneinigkeit. Unter Umständen ist hier eine andere Zielklassenverteilung sinnvoller (siehe auch Abschnitt 5.4 oder 6.4).

Für die Klassifikation oder Evaluierung von Daten muss bereits ein fertig initialisiertes und angelerntes Netz zur Verfügung stehen, die Netzstruktur muss mit der Struktur der Eingabedaten übereinstimmen. Für die Abspeicherung der Netzstruktur wurde im Programm das XML-Format gewählt. Die Auswahl dieser Sprache wurde vor allem in Hinblick auf PMML (*predictive model markup language*) getroffen, die eine allgemeine Beschreibungssprache für Data-Mining-Modelle darstellt. PMML basiert auf der Struktur von XML.

5.1.2 Initialisierungsmethoden

In der Implementierung stehen dem Nutzer für die Anfangsinitialisierung drei Methoden zur Verfügung:

- Initialwerte aus den Trainingsdaten:

  Bei dieser Methode werden aus der Eingabemenge zufällig Vektoren ausgewählt und deren Werte auf die Codebookvektoren übertragen. Dies mindert vor allem die Gefahr, dass viele Codebookvektoren so weit außerhalb der Gebiete der Eingabedaten liegen. Solche Vektoren die nie den kürzesten Abstand zum Eingabevektor besitzen, werden als *dead neurons* bezeichnet.

Bei der Initialisierung mit Werten aus den Trainingsdaten werden die unter Abschnitt 5.1.1 beschriebenen Bedingungen für die Codebookvektoren erfüllt. Diese Art der Initialisierung ist in den meisten Fällen die sinnvollste, vor allem wenn die einzelnen Zielklassen auf bestimmte Gebiete im Eingaberaum begrenzt sind und sich die Zahl der „Ausreißer“ in Grenzen hält.

- Zufällige Initialwerte:

  Den Werten der Codebookvektoren werden bei dieser Methode zufällige Zahlen im Bereich [0, 1] zugewiesen. Die in Abschnitt 5.1.1 aufgeführten Bedingungen werden auch bei dieser Methode erfüllt. Lediglich die Tatsache, dass zwei Codebookvektoren die gleichen Initialwerte besitzen lässt sich nicht sicherstellen, da es sich um Zufallszahlen handelt. Dieser Fall ist aber sehr unwahrscheinlich.

  Diese Methode könnte für Testzwecke sinnvoll sein, oder wenn die Eingabedaten wahllos verteilt im Raum liegen. Sinn macht die zufällige Initialisierung aber nur dann, wenn die Anzahl der Dimensionen (Merkmale) sehr gering ist. Je höher die Dimensionsanzahl ist, desto unwahrscheinlicher ist der Fall, dass sich ein Codebookvektor in der Nähe eines Eingabevektors befindet.

- Initialwerte eines früheren Klassifikators aus einer XML-Datei:

  Diese Methode wird verwendet, um einen bereits bestehenden Klassifikator einzulesen und auf eine neue Eingabemenge anzuwenden. Sinnvoll ist dieses Vorgehen z.B. zum Nachtrainieren des Klassifikators mit neuen Eingabedaten oder zum Evaluieren bzw. Klassifizieren von neuen Eingabedaten.

  Beim Einlesen der Werte aus einer Datei müssen die Anzahl der Merkmale, die Namen der Merkmale sowie das Zielmerkmal (bei der Klassifizierung unbekannter Daten ist das Zielmerkmal in den Eingabedaten natürlich nicht vorhanden) übereinstimmen. Ansonsten ist die Netzstruktur nicht auf die Eingabedaten anwendbar. Ob jeder Zielklasse mindestens ein Codebookvektor zugeordnet ist, wird nicht sichergestellt, da die Dateien vom Nutzer im Vorfeld gezielt verändert werden können.

5.1.3 Anzahl der Codebookvektoren

Die optimale Anzahl an Codebookvektoren zu bestimmen ist nicht sehr einfach, da sie für jede vorgegebene Datenmenge unterschiedlich sein wird. Klar ist jedoch, dass eine größere Anzahl von Codebookvektoren auch eine bessere Anpassung an die Eingabedaten zur Folge hat. Eine zu große Anzahl hat wiederum den Effekt des *Overfittings* zur Folge. So ergibt sich die erste obere Grenze der Anzahl der Codebookvektoren aus der Anzahl der Eingabedaten. Ansonsten ist eine Obergrenze für die Anzahl nur durch den begrenzten Speicher gegeben.

Liegen die Vektoren sehr verstreut im Eingaberaum, so ist es sinnvoller eine größere Anzahl von Codebookvektoren zu wählen. Sind die einzelnen Klassen jedoch auf bestimmte Gebiete begrenzt, so reichen wenige Codebookvektoren zur Repräsentation dieser Gebiete vollkommen aus.

In der Implementierung kann der Nutzer selbst die Anzahl der Codebookvektoren vorgeben. Ansonsten wird die Anzahl n aus der Anzahl i der Eingabedaten und deren Dimension d ermittelt. Die Dimension d der Eingabedaten fließt in diese Berechnung aus Gründen des Speicherplatzbedarfs und der Berechnungszeit mit ein. Die Anzahl n der Codebookvektoren wird mit der folgenden Formel bestimmt:

$$n = \begin{cases} i & \textit{für} \quad i \le 10 \\ 10 & \textit{für} \quad i \le 10 \cdot d \\ i/d & \textit{für} \quad i \le 10.000 \cdot d \\ 10.000 & \textit{sonst} \end{cases}$$

5.2 Abbruchkriterien

Dies ist ein sehr wichtiger Aspekt bei der Nutzung von Klassifikationsalgorithmen. Bei zu langem Training passt sich das Netz der gegebenen Eingabemenge zu sehr an und es tritt der Effekt des *Overfittings* auf. Der so entstandene Klassifikator weist dann eine relativ schlechte Güte auf, wenn er auf bisher unbekannte Daten angewendet wird.

Die Grundidee des Abbruchkriteriums beim LVQ-Netz ist die vorgegebene Anzahl von Lernepochen. Die Faustregel die hierbei verwendet wird besagt, dass die Anzahl der Lernepochen der 50- bis 200-fachen Zahl der Codebookvektoren entsprechen soll. Jeder Codebookvektor wird somit beim LVQ-Verfahren 50 bis 200 mal verändert.

Beim LVQ 2.1- und LVQ 3- Algorithmus sind mit den Bedingungen zur Veränderung der Codebookvektoren noch weitere Abbruchkriterien vorgegeben. Codebookvektoren werden z.B. nur verändert, wenn einer der beiden nächsten der Zielklasse des Eingabevektors angehört und wenn der Eingabevektor eine Mindestentfernung zum Codebookvektor besitzt. Diese Mindestentfernung wird durch das vorgegebene „Fenster“ realisiert. Werden während einer Lernepoche keine Codebookvektoren mehr verändert, so wird es auch in der nächsten Epoche keine Veränderung mehr geben. Das Training kann somit abgebrochen werden.

LVQ 2.1 verändert nur die Codebookvektoren an den Klassengrenzen. Die vorgegebenen Abbruchkriterien kommen bei diesem Verfahren auch sehr oft zum Zuge. Bei LVQ 3 werden auch die Vektoren innerhalb eines „Zielklassengebietes“ verändert. Der Abbruch des Lernvorgangs findet somit relativ selten statt.

Beim LVQ 1- und OLVQ 1-Verfahren gibt es keine Abbruchkriterien. Dort wird in jedem Falle der nächstliegende Codebookvektor verändert, egal ob er der selben oder einer anderen Klasse angehört. Die Möglichkeiten einer vorzeitigen Unterbrechung des Lernvorgangs sind jedoch beschränkt. Eine Variante wäre den Abstand des nächstliegenden Codebookvektors zum Eingabevektor zu untersuchen, wobei dies aber den LVQ 2.1- oder LVQ 3-Algorithmus wieder sehr nahe kommt. Eine andere Möglichkeit besteht in der ständigen Evaluierung auf einer zur Eingabemenge disjunkten Testmenge und dem Abbruch des Trainings wenn die Anzahl der korrekten Klassifikationen stetig abnimmt. Diese Methoden sind allerdings bis jetzt nur aus Überlegungen entstanden und noch nicht implementiert.

5.3 Lernvorgang

Ein Problem von neuronalen Netzen ist, dass die Trainingsqualität von der Reihenfolge der Präsentation der Eingabedaten abhängt. Das kommt daher, das ein neuronales Netz nicht das Wissen über die Gesamtstruktur der Eingabe kennt, sondern jeder Datensatz einzeln dem Netz präsentiert wird. Dies ist dann problematisch, wenn die Eingabemenge nach der Zielklasse sortiert vorliegt. Das Netz passt sich somit zuerst der Zielklasse an, die am Anfang der Eingabemenge steht.

Um dieses Problem zu umgehen und somit die Qualität des Klassifikators zu verbessern, wurde bei der Implementierung der „Zufall“ verwendet. Bei der anfänglichen Analyse der Eingangsdaten wird ein Feld mit den Indexnummern der Eingabedatensätze angelegt, das vor Beginn jeder Lernepoche zufällig gemischt wird. Während einer Lernepoche wird dieses Feld nacheinander ausgelesen und der Datensatz an der entsprechenden Stelle dem Netz präsentiert.

Für die schrittweise Verringerung der Lernrate *alpha* wird ein lineares Verfahren verwendet, um eine gleichmäßige Veränderung zu gewährleisten. Die Veränderung unterliegt folgender Formel:

$$\alpha(t) = \alpha(0) \cdot \left(1.0 - \frac{t}{\#epochen}\right).$$

Der Startwert $\alpha(0)$ kann vom Nutzer vorgegeben werden, [Kohonen97] schlägt dafür einen Wert von 0.1 vor.

5.4 Abschätzung der Güte des LVQ-Netzes

Eine Möglichkeit zur Abschätzung der Güte eines Klassifikators besteht in der sogenannten *Test and Train*–Methode. Dabei wird der Klassifikator auf eine zur Lernmenge disjunkten Testmenge angewendet und der Anteil der dadurch korrekt klassifizierten Datensätze ermittelt. Dieses Verfahren wurde auch zum Test des LVQ-Netzes an einer Datenmenge in Anhang B – Testdomäne verwendet. Als Ergebnis dieses Tests kann man feststellen, dass für eine „hinreichend gute“ Klassifikation nicht unbedingt sehr viele Codebookvektoren oder Lernepochen notwendig sind. Für die verwendete Domäne war es z.B. schon ausreichend, das Lernen schon auf wenige Lernepochen zu beschränken. Dies lässt auf eine recht gute Initialisierungsmethode schließen, wenn die Codebookvektoren mit Werten aus den Eingabevektoren initialisiert werden.

Weiterhin ist auch festzustellen, dass für die Qualität nicht unbedingt die „Masse“ an Codebookvektoren verantwortlich ist, sondern deren Verteilung im Raum. Dies trifft vor allem dann zu, wenn die Klassen der Eingabevektoren sich auf bestimmten Gebiete im Eingaberaum verteilen. Die vom System berechnete Anzahl von Codebookvektoren (siehe Abschnitt 5.1.3) schien jedenfalls den Anforderungen für die verwendete Domäne zu genügen, beim LVQ 2.1-Algorithmus lieferten diese Werte sogar mit die besten Ergebnisse. Eine geringe Anzahl von Codebookvektoren und relativ wenige Lernepochen beschleunigen den Lernprozess erheblich, und Geschwindigkeit ist eine wichtige Anforderung an einen Algorithmus.

Die zwei Lernverfahren LVQ 2.1 und LVQ 3, die als Abbruchkriterium den Abstand des Eingabevektors verwenden lieferten im Test bessere Ergebnisse. Dadurch lässt sich die Vermutung aus Abschnitt 5.2 bestätigen, dass mit diesen Methoden die Gefahr des Overfittings verringert werden kann.

[Trautzsch] stellte in seiner Arbeit fest, dass die LVQ-Algorithmen sich auf Klassen mit nur wenigen Ausprägungen schlechter anpassen als auf solche mit relativ hohem Anteil. Auch diese Erkenntnis lies sich durch verschiedene Versuche bestätigen. Abhilfe kann dabei eine andere Verteilung der Codebookvektoren auf die einzelnen Zielklassen schaffen, z.B. durch Gewichtung (siehe Abschnitt 6.4). Grund dafür ist, dass die Zielklassen mit hohem Anteil viel öfter zum Lernen verwendet werden als andere.

5.5 Erkenntnisse

Während der Untersuchung verschiedener Netztypen wurde relativ schnell klar, dass es ein allgemeingültiges Auswahlverfahren für einen bestimmten Netztyp nicht geben kann. Eine Möglichkeit diesen Nachteil zu überwinden besteht darin, die Aufgabenstellung genau zu definieren und aufgrund der sich ergebenden Bedingungen bestimmte Typen von neuronalen Netzen aus der Auswahl auszuschließen.

Der Nutzer muss zu diesem Zweck genau abwägen, welche Forderungen an den Algorithmus stellt und welche vernachlässigt werden können. So ist z.B. das in dieser Arbeit ausgewählte LVQ-Netz sehr gut für Klassifikationsaufgaben geeignet. Dies ergibt sich auch aus der Tatsache, dass es nach einem bekannten Klassifikationsprinzip, dem *Nächster-Nachbar-Prinzip*, arbeitet. So werden die Vorteile beider Verfahren miteinander vereint.

Steht aber z.B. die Flexibilität des Netzes im Bezug auf verschiedenartige Aufgabenstellungen im Vordergrund, so ist es durchaus sinnvoll auf ein Backpropagation-Netz zurückzugreifen. Dieses scheint durch seine äußerst flexible Struktur und eine große Zahl verschiedener Lernalgorithmen für die Lösung der vielfältigsten Aufgabenstellungen geeignet. Allerdings bezahlt der Nutzer einen hohen Preis bei der Verarbeitungsgeschwindigkeit.

Ebenso liegt es fern zu behaupten, neuronale Netze seien „besser“ oder „schlechter“ als andere bisher verwendete Verfahren. Vor- und Nachteile sind überall zu finden, die Einschätzung über den Einsatz liegt somit beim Anwender. Neuronale Netze werden sicherlich in der Zukunft noch einen sehr hohen Stellenwert besitzen, vor allem wenn es gelingt die Abarbeitung massiv zu parallelisieren. Die dazu notwendige Hardware wird sicherlich in naher Zukunft nicht mehr allzu teuer sein.

6 Ausblick

Es gibt sicherlich sehr viele Möglichkeiten zum Einsatz und zur Verbesserung des LVQ-Netzes bzw. der Implementierung. Einige wichtige werden in diesem Abschnitt kurz vorgestellt.

6.1 Einbettung in ein distributed Data-Mining-System

Da die Berechnungen des LVQ-Netzes im Gegensatz zu anderen neuronalen Netzen relativ schnell ablaufen, erscheint es sinnvoll diese gesparte Zeit anders zu nutzen. Die sinnvollste Möglichkeit besteht darin, mehrere Berechnungen verteilt mit verschiedenen Lernverfahren oder Parametern auf verschiedenen Computern parallel durchzuführen. Das LVQ-Netz bietet hierfür sehr gute Vorraussetzungen, da verschiedene Algorithmen auf die selbe Netzstruktur anwendbar sind. Dies macht einen direkten Vergleich der Ergebnisse viel einfacher. Die Einbettung des LVQ-Netzes in ein distributed Data Mining System stellt dafür eine ideale Grundlage zur Verfügung.

Das verteilte parallele Training eines Netztyps mit verschiedenen Lernverfahren wird auch als *Lernverfahren-Parallelität* bezeichnet, die verteilte parallele Berechnung mit verschiedenen Parametern als *Trainingsparameter-Parallelität* (siehe [Zell] S.433). Gerade die letztere Art der parallelen Berechnung ist für neuronale Netze besonders interessant, da die Qualität eines Lernprozesses sehr stark von der günstigen Wahl der Parameter abhängt, die je nach Eingabedaten unterschiedlich ist.

Eine weitere Möglichkeit des verteilten Rechnens besteht darin, die Eingabedaten aufzuteilen und auf verschiedenen Computern parallel abzuarbeiten. Dieses Vorgehen wird auch als *Trainingsmusterparallelität* bezeichnet. Die verschiedenen Gewichtsänderungen werden nach den Berechnungen aufsummiert und auf ein Netz übertragen.

Der wesentlich interessantere Aspekt, der sich durch die Aufteilung der Eingabedaten ergibt ist jedoch die Möglichkeit der Durchführung von *cross-validation*. Dies ist eine im Data Mining sehr häufig eingesetzte Methode zum Überprüfen der Güte eines Data Mining Modells. Die Eingabedaten werden bei diesem Verfahren in z.B. zehn Teile unterteilt (*ten-fold-cross-validation*). Von diesen werden dann neun zum Lernen und das letzte zum Evaluieren benutzt. Das Verfahren besteht nun darin, die Berechnungen zehn mal durchzuführen, so dass jede Teilmenge genau ein mal zum Evaluieren verwendet wird. Die Ergebnisse werden dann ggf. verglichen und daraus ein resultierendes Modell erstellt. Für diese Methode braucht man somit im Idealfall zehn verteilt rechnende Computer, was in einem distributed Data-Mining-System i.d.R. kein Problem darstellt.

6.2 Parallele Ausführung der Berechnung

Die Informationsverarbeitung bei einem LVQ-Netz ist gerade zu prädestiniert für massiv parallele Berechnungen. Dies ergibt sich aus der einfachen Tatsache, dass es sich um ein einschichtiges Netz handelt. Die einzelnen Neuronen sind somit nicht auf Zwischenergebnisse anderer Neuronen angewiesen, was die Gefahr von inkonsistenten Werten deutlich minimiert. Beim LVQ-Netz schlägt die netzinterne Berechnung der euklidischen Abstände zwischen den Codebookvektoren und dem Eingabevektor mit den größten Zeitaufwand zu Buche. Dazu ein kleines Rechenbeispiel:

> Für eine Menge von 1.000 Eingabevektoren und einer vorgegebenen Trainingsdauer von 20 Lernepochen ergibt sich eine Gesamtrechenzeit von 20.000 Lernzyklen. Wenn das LVQ-Netz beispielsweise 100 Codebookvektoren besitzt, ergeben sich somit insgesamt 2.000.000 Abstandberechnungen. Je höher die Dimension der Vektoren, desto mehr Zeit ist für jede einzelne Abstandberechnung natürlich aufzubringen. Die Zahlenbeispiele hierbei sind durchaus reale Werte.

Würden die Abstandsberechnungen parallel auf verschiedenen Prozessoren (unter Umständen auch verschiedene Computer) durchgeführt, ergäbe sich eine deutliche Zeitersparnis. Am besten geeignet dafür sind SIMD-Parallelrechner (*single instruction multiple data*). Aber auch auf einfachen Computern mit mehreren parallel arbeitenden Prozessoren ergeben sich große Geschwindigkeitsvorteile.

Die parallele Verarbeitung auf verschiedenen vernetzten Computern mit jeweils einem Prozessor macht wiederum nur dann Sinn, wenn die Dimension der Vektoren sehr hoch ist oder die Berechnungen für mehrere Codebookvektoren auf einem einzelnen Rechner gleichzeitig ablaufen. Ansonsten würde der Mehraufwand für die Verteilung der Daten auf verschiedene Rechner im Verhältnis zur eigentlichen Berechnung so hoch sein, das sich das Ganze dann nicht mehr rentiert.

Eine weitere Motivation, das LVQ-Netz für eine parallele Berechnung zu implementieren ergibt sich daraus, dass laut [Zell] S.432 der am häufigsten auf eine parallele Implementierung hin untersuchte Netztyp das Backpropagation-Netz ist. Dadurch ist ein Anreiz gegeben, sich von der „Masse“ der Untersuchungen abzuheben und somit über diesen weniger erforschten Netztyp bisher unbekannte Informationen zu sammeln.

6.3 Kombination mit selbstorganisierenden Karten (SOM)

Die LVQ-Verfahren und die selbstorganisierenden Karten (*self-organizing-maps*) sind zwei eng verwandte Netztypen. Auch die selbstorganisierenden Karten sind eine Entwicklung von [Kohonen97], sie können als Weiterentwicklung des LVQ-Netzwerks aufgefasst werden. Der Unterschied zwischen beiden Arten besteht in der Art des Lernverfahrens. Die selbstorganisierenden Karten werden unüberwacht trainiert (siehe Abschnitt 4.1.3) und eignen sich somit für das Clustering von Datenmengen.

Da beide Lernverfahren auf der selben Netzstruktur arbeiten, ist es gut möglich, beide Lernverfahren zu kombinieren. Denkbar ist z.B. eine Vorverarbeitung durch Clustering auf den Daten und eine anschließende Klassifikation einer der entstandenen Datenmengen.

6.4 Gewichtung der Zielklassen

Bei realen Klassifikationsaufgaben im Data Mining ist es häufig der Fall, dass der interessierende Teil der Datensätze nur einen geringen Teil der gesamten Datenmenge ausmacht. Besonders ausgeprägt ist diese Tatsache bei Direktmailings, wo der Reagiereranteil nur 2-3 % aller Personen ausmacht. Dieser relativ geringe Anteil der Kunden sorgt aber für den Umsatz eines Unternehmens.

Eine Möglichkeit das Training mit Hinsicht auf diese Tatsache durchzuführen, wäre eine Art *Gewichtung* der einzelnen Zielklassen. Die Initialisierung der Codebookvektoren kann z.B. so erfolgen, dass Zielklassen mit höherer Gewichtung ein entsprechend höherer prozentualer Anteil an Codebookvektoren zugewiesen wird. Solch eine Art Gewichtung muss das übergeordnete Data Mining System (welches dem Algorithmus die Eingabedaten zur Verfügung stellt) allerdings auch unterstützen.

6.5 Behandlung von dead neurons

Beim Training eines LVQ-Netzes kann es vorkommen, das einige der Codebookvektoren als *dead neurons* keiner Gewichtsänderung mehr unterliegen (siehe Abschnitt 3.10.6). Dies kann z.B. bei einer ungünstigen Wahl der Initialwerte auftreten oder wenn ein Codebookvektor aufgrund unterschiedlicher Klassenzugehörigkeit entsprechend weit weg von den Gebieten der Eingabedaten geschoben wurde.

Solche *dead neurons* können sehr leicht identifiziert werden (sie werden ja nie verändert). Eine Möglichkeit besteht nun darin, diese Werte zu entfernen und durch neue Codebookvektoren zu ersetzen, eine weitere durch gezieltes Verschieben dieser Codebookvektoren. Ebenso könnten solche Codebookvektoren verändert werden, die im Gegensatz zu anderen relativ selten verändert werden. Diese lassen z.B. auf Ausreißer in den Datenmengen schließen.

6.6 Abschlußbemerkung

Durch die ständig steigende Flut an Informationen werden auch in Zukunft immer wieder neue effektivere Algorithmen benötigt. Die neuronalen Netze stellen hierfür eine gute Grundlage zur Verfügung, die es allerdings noch zu erforschen gilt. Eine Auswahlmethodik, die zu einem gegebenen Problem das passende neuronale Netz angibt ist zwar schwer zu realisieren, wäre aber von großen Vorteil. Weitere Untersuchungen könnten an dieser Stelle ansetzen.

Anhang A – Symbolverzeichnis

Begründung der Auswahl

In der Literatur über neuronale Netze gibt es noch keine einheitliche Bezeichnung für die verschiedenen Bestandteile dieser Netze. Auch in den von mir verwendeten Büchern wiesen vor allem die verwendeten Variablen und Indizes erhebliche Unterschiede auf.

Ich habe daher eine für mich sinnvoll erscheinende Bezeichnung der Symbole gewählt, die sich aus den Angaben zweier Bücher zusammensetzt, die für diese Arbeit am meisten verwendet wurden, [Hoffmann] und [Zell].

Liste der Symbole

Die folgende Liste enthält eine Übersicht der Symbole, die in dieser Arbeit verwendet werden, in alphabetischer Reihenfolge:

- a Aktivierung bzw. Aktivierungszustand; demnach ist a_i die Aktivität des Neurons i,
- $E_1, E_2, ...$ Netzeingänge; werden zum Vektor E zusammengefasst,
- $e_1, e_2, ...$ Eingänge des Neurons i; der Index läuft von 1 bis n,
- η Lernrate,
- f_{act} Aktivierungsfunktion; berechnet den neuen Aktivierungszustand des Neurons,
- f_{in} Eingangsfunktion; berechnet den effektiven Eingang des Neurons,
- f_{out} Ausgabefunktion; berechnet die Ausgabe des Neurons,
- i (als Index) nummeriert die Neuronen des Netzes und bezeichnet das aktuell betrachtete Neuron,
- j (als Index) nummeriert die Neuroneneingänge und bezeichnet den Vorgänger des Neurons i,
- n Anzahl der Eingänge eines Neurons,
- N Anzahl der Neuronen eines Netzes,
- N_E Anzahl der Eingangsneuronen eines Netzes,
- N_O Anzahl der Ausgangsneuronen eines Netzes,
- net_i effektiver Eingang des Neurons i, der sich aus allen Einzeleingängen berechnet,
- $O_1, O_2, ...$ Netzausgänge; werden zum Ausgangsvektor O zusammengefasst,
- o Neuronenausgang; demnach ist o_i der Ausgang des Neurons i,
- p Anzahl der Muster in einem Trainingssatz,
- s Skalierungsfaktor,
- $S_1, S_2, ...$ Sollwerte bzw. Sollausgabewerte,
- w_{ij} Gewicht des Neurons i am Eingang j; der Verlauf ist also vom Neuron j zum Neuron i hin,
- W Gewichtsmatrix; in ihr sind alle Verbindungsgewichte zusammengefasst,
- δw_{ij} Änderung des Gewichts von w_{ij},
- η Lernrate,
- μ (als Index) nummeriert die Muster einer Menge von Trainingsbeispielen,
- Θ_i Schwellenwert des Neurons i,
- σ Anstieg einer Funktion.

Zur besseren Veranschaulichung sind die wichtigsten mathematischen Größen zweier Neuronen in Abbildung A-1 noch einmal dargestellt.

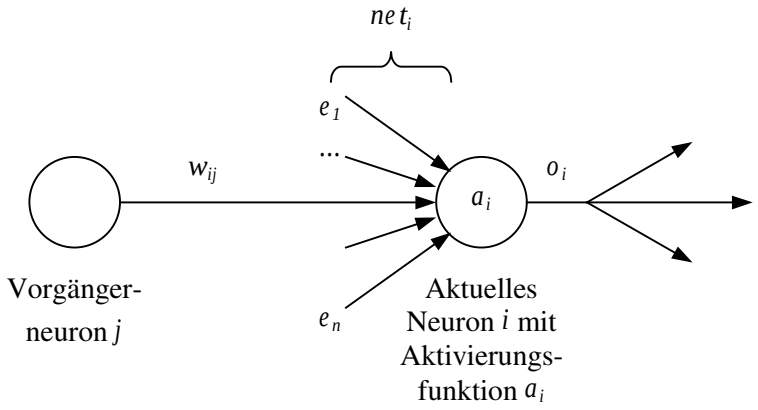


Abbildung A-1: Symbolübersicht am Beispiel zweier verbundener Neuronen

Sollten sich in einem Abschnitt die Symbole oder die Indizierung ändern, so wird an der entsprechenden Stelle darauf hingewiesen.

Anhang B – Testdomäne

Zum Testen der Güte des LVQ-Netzes wurde die *SPAM E-mail Database* verwendet. Es handelt sich um dabei Daten einer großen Firma, die Emails danach untersucht hat, ob es sich um Werbemails handelt oder nicht. Die Daten können im Internet unter der Adresse

http://www.ics.uci.edu/~mlearn/MLRepository.html

bezogen werden.

Struktur der Daten

Die Gesamtdatenmenge umfasst etwa 4.600 Datensätze. Aus der Datenmenge wurden zwei disjunkte gleichgroße Teilmengen erstellt, eine zum Trainieren und eine zum Evaluieren. Das Zielmerkmal besitzt genau zwei Ausprägungen, die sich wie folgt aufteilen:

- ca. 40 % Zielklasse 1 (Spam-mail)
- ca. 60 % Zielklasse 0 (keine Spam-mail)

Es handelt sich also um ein typisches „2-Klassen-Problem“. Zur Beschreibung der Datensätze stehen neben dem Zielmerkmal noch weitere 57 Merkmale zur Verfügung, auf deren Bedeutung hier aber nicht eingegangen werden soll. Alle Merkmale sind numerisch und wurden für die Analyse mit dem LVQ-Netz normiert (Z-Normierung).

Testablauf

Bei den Testreihen wurde mit den Lerndaten das Netz trainiert und anschließend auf den Evaluierungsdaten die Anzahl der richtig und falsch klassifizierten Datensätze bestimmt. Dieses Verfahren wird auch als *Test and Train* bezeichnet. Daraus lässt sich mit folgender Formel die *Accurcy* (der prozentuale Anteil richtig klassifizierter Datensätze) ermitteln:

$$Accuracy = \frac{Anzahl\ korrekter\ Klassifikationen \cdot 100}{Gesamtzahl\ der\ Datensätze}.$$

Die Tests wurden mit allen vier implementierten LVQ-Lernalgorithmen durchgeführt. Die Anzahl der Codebookvektoren und die Anzahl der Lernepochen stellten die veränderlichen Größen dar. Als Ergebnis der Untersuchung wurde jeweils der Anteil richtig klassifizierter Datensätze ermittelt. Die Lernparameter blieben bei allen Testreihen konstant, es wurden die von [Kohonen97] vorgeschlagenen Werte benutzt:

- *alpha* (alle Verfahren): 0.1
- *v* (LVQ 2.1, LVQ 3): 0.2
- *e* (nur LVQ 3): 0.3

Die Codebookvektoren wurden mit zufällig ausgewählten Eingabevektoren initialisiert.

Ergebnisse

Folgende Ergebnisse entstanden nach Anwendung des Klassifikators auf die Evaluierungsmenge.

LVQ 1		**Anzahl Codebookvektoren**				
		10	**39**	**100**	**200**	**500**
Anzahl Lernepochen	**10**	86%	90,5%	89,7%	90,6%	91%
	50	86,5%	89,4%	89,9%	90,5%	90,6%
	100	88,9%	89,2%	**90,9%**	90,8%	90,4%
	500	89,2%	89,2%	89,8%	90,7%	**90,9%**
	1.000	88,9%	89,3%	89,8%	91%	**90,9%**

Tabelle A 1: Erfolgsrate des LVQ 1-Algorithmus

LVQ 2.1		Anzahl Codebookvektoren				
		10	**39**	**100**	**200**	**500**
Anzahl Lernepochen	**10**	92,2%	**92,7%**	91,9%	91,6%	90,4%
	50	90,9%	**92,7%**	92,3%	91,7%	90,7%
	100	90,1%	91,9%	92%	91,9%	90,7%
	500	90,8%	91 %	**92,7%**	92%	91%
	1.000	90,4%	90,7%	90,3%	92%	91%

Tabelle A 2: Erfolgsrate des LVQ 2.1-Algorithmus

LVQ 3		Anzahl Codebookvektoren				
		10	**39**	**100**	**200**	**500**
Anzahl Lernepochen	**10**	88,8%	92%	91,3%	91,2%	90,5%
	50	91%	91,3%	91,5%	**92,2%**	91,7%
	100	91,2%	90,7%	91,3%	91,6%	91,6%
	500	90,7%	91,7%	92,1%	91,4%	91,9%
	1.000	90,7%	90,8%	92%	90,6%	91,2%

Tabelle A 3: Erfolgsrate des LVQ 3-Algorithmus

OLVQ 1		Anzahl Codebookvektoren				
		10	**39**	**100**	**200**	**500**
Anzahl Lernepochen	**10**	87,1%	89%	89,9%	90,4%	90,4%
	50	87,2%	89,8%	89,6%	90,8%	**91,2%**
	100	87,2%	89,4%	89,1%	90,9%	90,7%
	500	87%	90%	89,8%	89,9%	90,9%
	1.000	86,9%	90%	89,9%	90,5%	90,4%

Tabelle A 4: Erfolgsrate des OLVQ 1-Algorithmus

Die grau hinterlegte Spalte in den Tabellen bezeichnet die vom System selbstständig errechnete Anzahl Codebookvektoren (siehe Abschnitt 5.1.3). Die jeweils dick markierten Werte beschreiben das „beste“ Ergebnis.

Erkenntnisse

Die Testergebnisse der Algorithmen können bei dieser Domäne durchaus als zufriedenstellend angesehen werden. Eine entsprechende Schlussfolgerung über die Qualität des Verfahrens bei anderen Datenmengen kann natürlich nicht gezogen werden.

Bei Betrachtung der Ergebnisse kann man erkennen, dass nicht unbedingt die Anzahl der Lernepochen für die Güte des Klassifikators verantwortlich ist. Teilweise sind bei steigernder Anzahl sogar Verschlechterungen zu erkennen. Die Anzahl der Codebookvektoren scheint dagegen einen etwas größeren Einfluss auf die Qualität zu haben. Je mehr Codebookvektoren erzeugt werden, desto besser decken sie den Eingaberaum ab. Die Unterschiede der einzelnen Testreihen sind jedoch nicht sehr groß, so dass bei diesen Testdaten auch mit wenig Zeitaufwand (d.h. relativ geringe Anzahl von Codebookvektoren und Lernepochen) ein ausreichend guter Klassifikator erstellt werden kann.

Beim Training mit dem LVQ 2.1-Algorithmus setzte bei 500 Codebookvektoren jeweils nach einer bestimmten Anzahl Lernepochen das Abbruchkriterium ein. Die Verteilung der Codebookvektoren auf die Gebiete der Zielklassen schien dadurch „hinreichend gut“ zu sein.

Weiterhin war zu beobachten, dass die Verfahren LVQ 2.1 und LVQ 3, die Vektoren an den Klassengrenzen verändern, bessere Ergebnisse lieferten. Das lässt darauf schließen, das die Eingabevektoren auf bestimmte Gebiete im Eingaberaum begrenzt und nicht wahllos verstreut sind. Auch der Abbruch des Trainings beim LVQ 2.1-Verfahren (bei 500 Codebookvektoren) lässt diesen Schluss zu). Durch die Verwendung der LVQ 2.1- oder LVQ 3-Algorithmen scheint sich die Gefahr des Overfittings zu verringern. Dies liegt sicherlich an der Beschränkung der Berechnung durch das in den zwei Algorithmen verwendete „Fenster“.

Literaturverzeichnis

[Bellido] Bellido, E.; Fernández, G.: *Backpropagation Growing Networks: Towards Local Minima Elimination*, In: [Hoffmann] 1993

[Brause] Brause, R.: *Neuronale Netze*, 2. Auflage, B. G. Teubner, Stuttgart, 1995

[ELENA] Guérin-Dugué, A.: *ELENA – Enhanced Learning for Evolutive Neural Architecture*, 1995

http://www.dice.ucl.ac.be/neural-nets/Research/Projects/ELENA/elena.htm

[Görz] Görz, G.: *Einführung in die künstliche Intelligenz*, 2.Auflage, Addison-Wesley (Deutschland) GmbH, Bonn, Paris, 1995

[Grosb69] Grossberg, S.: *Some networks that can learn, remember and reproduce any number of complicated space-time patterns*, In: [Zell] 1994

[Grosb71] Grossberg, S.: *Embedding fields: Underlying philosophy, mathematics, and applications of psychology, physiology, and anatomy*, In: [Zell] 1994

[Grosb78] Grossberg, S.: *Studies in Mind and Brain*, In: [Zell] 1994

[Haun] Haun, M.: *Simulation Neuronaler Netze - Eine praxisorientierte Einführung*, expert-Verlag, Renningen-Malmsheim, 1998

[Hoffmann] Hoffmann, N.: *Kleines Handbuch Neuronale Netze*, Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig, 1993

[Hertz] Hertz, J.; Krogh, A.; Palmer RG.: *Introduction to the theory of neural computation*, In: [Hoffmann] 1993

[Kosko] Kosko, B.: *Bi-directional associative memories, IEEE Transactions on Systems, Man and Cybernetics 18(1)*, In: [Zell] 1994

[Kohonen97] Kohonen, T.: *Self-Organizing Maps*, 2nd ed. P. cm., Springer-Verlag, Berlin Heidelberg, 1997

[Kohonen95] Kohonen, T.; Hynninen, J.; Kangas, J.; Laaksonen, J.; Torkkola, K.: *LVQ_PAK: The Learning Vector Quantization Program Package Version 3.1*, Espoo, 1995

http://www.cis.tugraz.at/igi/pauer/LVAs/LVAs/MLAB-97/lvq_doc.ps

[LanWit] Lang, K.J.; Witbrock M.J.: *Learning to Tell Two Spirals Apart*, In: [Zell] 1994

[McElie] McEliece, R.J.;Posner, E.C.;Rodemich, E.R.; Venkatesh, S.S.: *The capacity of the Hopfield associative memory, IEEE Vehicular Technology Conference ´92*, In: [Zell] 1994

[MinPap] Minsky, M.; Papert, S.: *Perceptrons*, 2nd ed., In: [Zell] 1994

[Polifke] Polifke, A.: *Adaptive Neuronale Netze zur Lösung von Klassifikationsproblemen im Marketing*, Peter Lang GmbH, Europäischer Verlag der Wissenschaften, 1998

[Protzel] Protzel, P.; Tagscherer M.; Fazlija N.: *Stabilität und Plastizität Neuronaler Netze bei kontinuierlichem Lernen*, TU-Chemnitz, Universität Erlangen

http://bfws7e.informatik.uni-erlangen.de/aknn/publications.html

[Rumel] Rumelhart, D.E.; Hinton, G.E.; Williams, R.J.: *Learning internal representations by error propagation*, In: [Zell] 1994

[Sauer] Sauer, J.: *Neuronale Netze und Fuzzy Control-Systeme*, 1991-2000

http://rfhs8012.fh-regensburg.de/~saj39122/vhb/NN-Script/script/gen

[Trautzsch] Trautzsch, S.: *Entwicklung einer Methodik zur Durchführung von Induktionsexperimenten für die Erstellung von Entscheidungsbäumen und Regelsätzen*, Diplomarbeit, Leipzig, 1996

[Zell] Zell, A.: *Simulation Neuronaler Netze*, Addison-Wesley (Deutschland) GmbH, Bonn, Paris, 1994

Abbildungsverzeichnis

Tabellenverzeichnis