Was können neuronale Netze?

Raúl Rojas

1 Das Approximationsproblem

Neuronale Netze bilden ein alternatives Berechnungsmodell, das in den letzten Jahren zunehmende Beachtung gefunden hat. Die Motivation dieses neuen Paradigmas liegt, wie der Name schon andeutet, in der Biologie: man möchte Automaten bauen, die gewisse Aspekte der Informationsverarbeitung bei Mensch und Tier nachahmen. In der Sprache der Informatik ausgedrückt, können wir sagen, daß biologische Nervensysteme *massiv parallel* arbeiten, *fehlertolerant* sind und sich *adaptiv* verhalten. Gerade diese Eigenschaften sollten auch künstliche Automaten aufweisen. Bei gewissen Anwendungen sind sie sogar unerläßlich, wie z.B. massive Parallelität in der Hochenergiephysik [4], Fehlertoleranz in der Robotik [14] und adaptives Verhalten bei der Spracherkennung [10]. Neuronale Netze sollen sich außerdem als *lernende Systeme* verhalten, die ihre eigenen Parameter bestimmen und anpassen können.

Obwohl neuronale Netze erst in den letzten Jahren allgemeine Anerkennung fanden, ist ihre Geschichte fast so alt wie die der traditionellen Computersysteme selbst. Die ersten Modelle abstrakter Neuronen wurden von Warren McCulloch und Walter Pitts im Jahre 1943 vorgeschlagen, also zu einer Zeit, in der gerade die ersten Computeranlagen gebaut wurden. Nur wenige wissen, daß sich John von Neumann in seiner Arbeit, die die Architektur zukünftiger Computersysteme auf lange Zeit festschrieb, in seiner Argumentation des biologischen Modells bediente [17]. Dabei bildeten die Untersuchungen von McCulloch und Pitts, sowie die Kybernetik Norbert Wieners den Hintergrund seiner Ausführungen. Daß später über künstliche neuronale Netze eher marginal geforscht wurde, ist zum Teil das Ergebnis des überwältigenden Erfolges der traditionellen Computersysteme, die einen alternativen Zugang zur maschinellen Intelligenz bieten. Erst seit den siebziger aber vor allem seit den achtziger Jahren erleben wir eine Renaissance der Forschung auf dem Gebiet des *Konnektionismus*. Die subsymbolischen Fähigkeiten neuronaler Netze sollen jetzt jene Systeme unterstützen, die bis heute menschliche Intelligenz auf der rein symbolischen Ebene, d.h. auf der Ebene der Logik, modelliert haben.

In diesem Aufsatz wollen wir besprechen, was neuronale Netze sind und welche Art von Berechnungen sie durchführen können. Wir zeigen, wie solche Systeme trainiert werden, welches der Unterschied zum traditionellen algorithmischen Zugang ist und wie adaptives Verhalten implementiert werden kann. Es wird insbesondere gezeigt, daß künstliche neuronale Netze geeignete Systeme für statistische Modellierung bilden, und es wird anhand des Beispieles der automatischen Spracherkennung erläutert, wie künstliche neuronale Netze in traditionelle algorithmische Verfahren eingebettet werden können.

Der Aufsatz ist folgendermaßen gegliedert: In Abschnitt 1 zeigen wir, vom biologischen Modell ausgehend, daß künstliche neuronale Netze ein altes Problem der Approximationstheorie berühren, nämlich die Ermittlung der besten Approximation zu einer nur unvollständig definierten Funktion. Wir zeigen später, daß dies ein hartes Berechnungsproblem ist. In Abschnitt 2 geben wir ein Beispiel eines einfachen Modells, das Perzeptron, dessen Lernalgorithmus einfach zu visualisieren ist. Im dritten Abschnitt wird der Backpropagation-Algorithmus abgeleitet, der für das Training mehrschichtiger Netze häufig verwendet wird. Die letzten zwei Abschnitte behandeln dann die oben erwähnten statistischen Eigenschaften neuronaler Netze.

1.1 Das biologische Modell

Mit den künstlichen neuronalen Netzen werden biologische neuronale Netze als informationsverarbeitende Systeme nachgeahmt. Auch wenn die einzelnen biologischen Modelle des Gehirns und der Nervensysteme von Lebewesen sich in vielen Aspekten unterscheiden, herrscht allgemeine Übereinstimmung darüber, daß das Wesen der Funktion des Nervensystems in der Kontrolle durch Kommunikation besteht. Nervensysteme bestehen aus vielen Tausenden oder Millionen von miteinander vernetzten Nervenzellen — den Neuronen. Jede dieser Zellen ist sehr komplex in ihrem Aufbau und kann eintreffende Signale auf vielfältige Weise verarbeiten. Neuronen sind jedoch langsam im Vergleich zu elektronischen Bausteinen. Während die letzteren Schaltzeiten in der Größenordnung von Nanosekunden besitzen, schalten Neuronen innerhalb einiger Millisekunden. Und trotzdem: Das menschliche Gehirn kann Probleme lösen, die für jeden konventionellen Rechner noch in unerreichbarer Ferne liegen.

Nervenzellen empfangen Signale und geben neue, von ihnen erzeugte, Signale weiter. Die allgemeine Struktur eines generischen Neurons wird in Abb. 1 gezeigt. Die Fortsätze des Neurons, die die Eingangsinformation sammeln, werden Dendriten genannt. Sie übernehmen Signale aus anderen Nervenzellen an spezifischen Kontaktstellen, den Synapsen. Der Zellkörper reagiert auf diese Stimuli und f
ängt an, Signale zu übertragen. Die Ausgabesignale eines Neurons werden durch das Axon an andere Neuronen weitergereicht.

Diese vier Elemente (Dendriten, Synapsen, Nervenkörper und Axon) bilden die minimale Struktur, die wir aus biologischen Modellen für unsere Zwecke übernehmen, so daß auch künstliche Neuronen als informationsverarbeitende Elemente

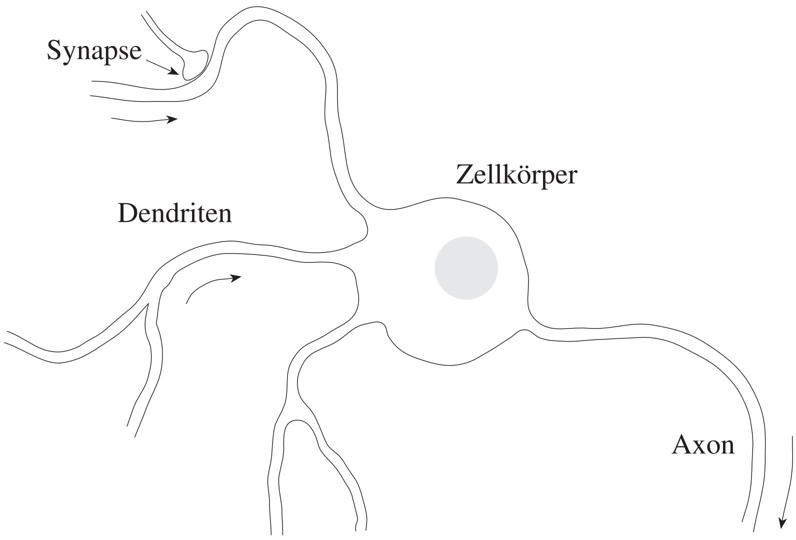

Abbildung 1: Struktur eines Neurons

- *gerichtete, gewichtete Eingabeleitungen,*
- *einen Berechnungskörper,*
- *eine Ausgabeleitung*

besitzen werden. Abb. 2 zeigt die Struktur eines abstrakten Neurons mit n Eingängen. Jede Eingabeleitung überträgt einen reellen Wert x_i. Die *"primitive"* Funktion f, die vom abstrakten Neuron berechnet wird, kann im Prinzip beliebig ausgewählt werden. Die Eingabeleitungen sind gewichtet, d.h. jeder Eingabewert

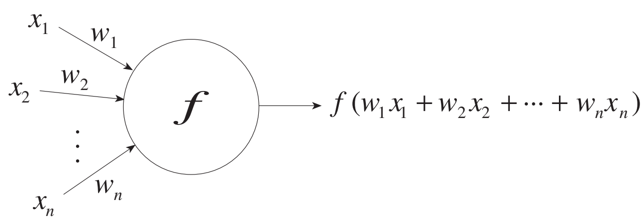

Abbildung 2: Struktur eines künstlichen Neurons

x_i wird mit dem entsprechenden Gewicht w_i multipliziert. Die im Berechnungselement ankommende Information wird addiert, und das Resultat wird als Argument für die primitive Funktion f verwendet. Ist die Struktur der einzelnen Neuronen spezifiziert, können diese Elemente in Netzen miteinander gekoppelt werden.
Ein wichtiger Spezialfall eines Berechnungselements ist das *Perzeptron*, dessen Diagramm in Abb. 3 gezeigt wird [9]. Ein Perzeptron besitzt n Eingabeleitungen, die jeweils mit den Gewichten $w_1, w_2, \ldots, w_n$ behaftet sind. Das Perzeptron kann

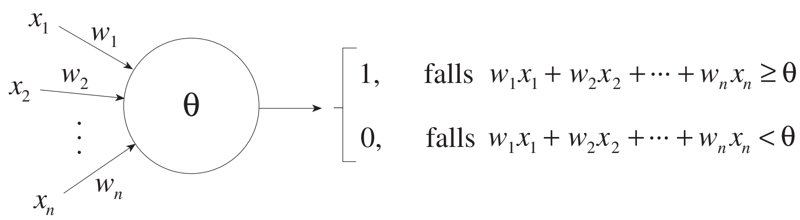

Abbildung 3: Struktur eines Perzeptrons

nur Schwellenwertentscheidungen treffen, d.h. die Ausgabe der Zelle ist 1, falls

$$x_1w_1 + x_2w_2 + \cdots + x_nw_n \geq \theta$$

gilt, wobei θ der sogenannte Schwellenwert der Zelle ist. Falls

$$x_1w_1 + x_2w_2 + \cdots + x_nw_n < \theta$$

gilt, wird eine Null ausgegeben.
Mit Perzeptronen können logische Funktionen ohne weiteres realisiert werden. Die AND-Verknüpfung zweier binärer Variablen x_1 und x_2 kann durch eine Zelle mit zwei Eingabeleitungen implementiert werden, bei der w_1 und w_2 beide gleich 1 sind und $\theta = 2$ gilt. Die Zelle wird nur dann eine 1 feuern, wenn $x_1 + x_2 \geq 2$ erfüllt ist, d.h. bei binären Eingaben genau dann, wenn $x_1 = x_2 = 1$. Abb. 4 zeigt die Implementierung der logischen Funktionen AND, OR und NOT mittels eines Perzeptrons. Da diese drei Funktionen eine Basis für die Implementierung aller möglichen booleschen Funktionen bilden, ist klar, daß mit einem Netz von Perzeptronen beliebige logische Funktionen berechnet werden können.

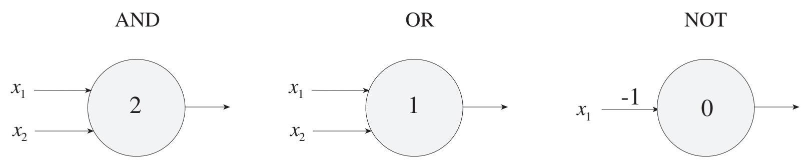

Abbildung 4: Logische Schaltungen

1.2 Neuronale Netze als Funktionennetze

Wird jedes Schaltelement in einem neuronalen Netz als eine primitive Funktion aufgefaßt, die ihre Eingabe in eine bestimmte Ausgabe verwandelt, dann sind neuronale Netze nichts anderes als Netze von Funktionen. Verschiedene abstrakte Modelle unterscheiden sich hinsichtlich der Annahmen über die einzelne primitiven Funktionen im Netz, die Topologie der Vernetzung und die Bedingungen für synchrone oder asynchrone Bearbeitung der Information. In der konventionellen

Datenverarbeitung ähnelt die Architektur der Datenflußrechner der Architektur neuronaler Netze am meisten [23].
Typische künstliche neuronale Netze haben die in Abb. 5 gezeigte Struktur. Das Netz kann als eine Funktion Φ interpretiert werden, die für die Eingabe (x, y, z) ausgewertet wird. An den Knoten werden elementare Funktionen f_1, f_2, f_3 und f_4 berechnet, deren Komposition Φ definiert. Durch eine Veränderung der Netzgewichte $\alpha_1, \alpha_2, \ldots, \alpha_5$ (–das Gewicht 1 wird in Abbildungen nicht vermerkt) kann Φ verändert und für unterschiedliche Aufgaben angepaßt werden. Die von einem neuronalen Netz implementierte Funktion Φ nennen wir die *Netzfunktion*.

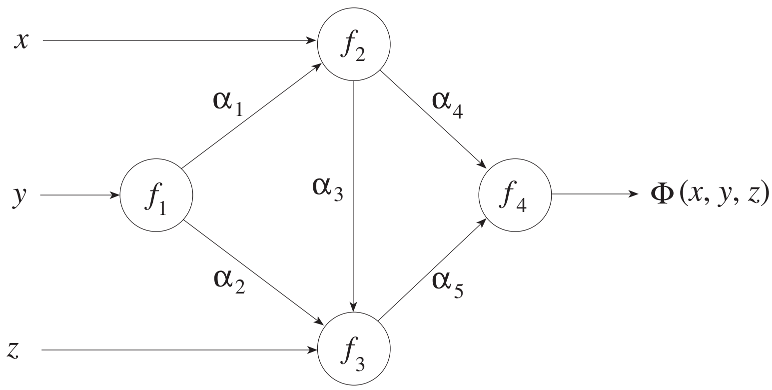

Abbildung 5: Funktionales Modell eines neuronalen Netzes

Neuronale Netze werden dann verwendet, wenn eine vorgegebene Menge von Eingabewerten auf eine vorgegebene Menge von Ausgabewerten abgebildet werden soll. Eine Menge von m Ein-Ausgabepaare $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ wird *Trainingsmenge* genannt. Das *Lernproblem* besteht darin, jene Funktion zu finden, die die Trainingsausgabewerte am genauesten den entsprechenden Trainingseingabewerten zuordnet. Optimale Gewichte sind diejenigen, welche die "beste" Anpassung der Netzausgabe an die vorgegebenen Ausgabewerte vornehmen. Jedoch spielen dabei zwei entgegengesetzte Motivationen eine Rolle. Einerseits soll das Netz nach der Lernphase die Trainingseingabewerte möglichst genau auf die Ausgabewerte abbilden. Andererseits wird aber erwartet, daß das Netz verallgemeinern kann, d.h. es muß für unbekannte Eingabewerte eine Ausgabe produzieren, die eine Art Interpolation der Ausgaben aller bekannten Eingaben ist.
Abb. 6 veranschaulicht, wie ein Netz den Eingaberaum in einen Ausgaberaum abbildet. Die schwarzen Punkte im Eingaberaum stellen die Trainingseingaben dar, die auf die Trainingsausgaben (schwarze Punkte im Ausgaberaum) abgebildet werden sollen. Es ist aber auch erwünscht, daß die Nachbarschaften der Trainingseingaben auf Nachbarschaften der Trainingsausgaben abgebildet werden. Eine für das Netz unbekannte Eingabe, die dicht an einer trainierten Eingabe liegt, soll in der Nähe der trainierten Ausgabe abgebildet werden. Offensichtlich soll das Netz eine *stetige* Abbildung des Eingaberaums in den Ausgaberaum berechnen.

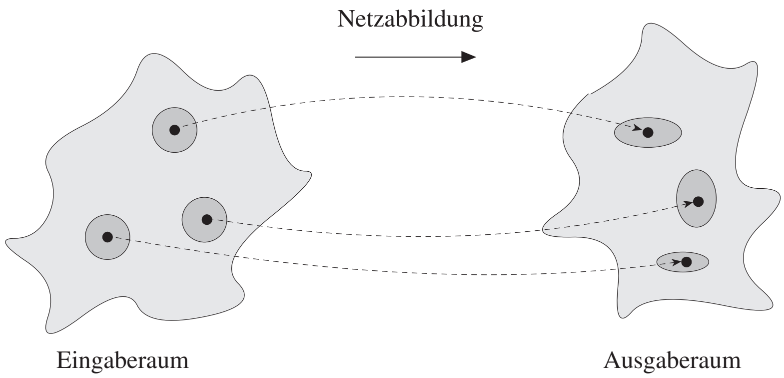

Abbildung 6: Abbildung von Nachbarschaften in Nachbarschaften

Wenn die primitiven Funktionen in den Knoten des Netzes stetig sind, ist diese Forderung automatisch erfüllt. In der Statistik werden ähnliche Probleme durch eine lineare Funktionsapproximation (lineare Regression) gelöst. Neuronale Netze finden in der Regel eine nichtlineare Approximation zu den experimentellen Punkten und stellen in diesem Sinne eine Erweiterung der traditionellen statistischen Methoden dar.
Künstliche neuronale Netze sind nur eines von mehreren möglichen Berechenbarkeitsparadigmen. Sie besitzen jedoch eine stärkere Faszination als andere Berechenbarkeitsmodelle, weil sie, obwohl nur abstrakte mathematische Modelle von vernetzten Funktionen darstellend, uns etwas über die organische Welt verraten können. Soll eine Definition für die angesprochenen Systeme gegeben werden, könnte gesagt werden, daß künstliche neuronale Netze sich mit den Eigenschaften von Netzen von primitiven Funktionen beschäftigen. Welche Funktionen können aber auf diese Weise modelliert werden? Die Antwort wird von der Approximationstheorie geliefert.

1.3 Funktionsapproximation mit neuronalen Netzen

Ein altes Problem der Approximationstheorie besteht darin, eine vorgegebene Funktion $F : I\!R \to I\!R$ durch die Komposition primitiver Funktionen exakt oder annähernd auszudrücken. Ein klassisches Beispiel ist die Approximation von eindimensionalen Funktionen mittels Polynomen oder Fourierreihen. Die Taylorsche Reihe für eine Funktion F, die am Punkt x_o approximiert wird, hat die Form

$$F(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots + a_n(x - x_0)^n + \cdots,$$

wobei die Konstanten $a_0, ..., a_n$ vom Wert der Funktion F und ihrer Ableitungen am Punkt x_0 abhängen. Abb. 7 zeigt, wie die polynomielle Approximation einer Funktion $F(x)$ als Funktionennetz dargestellt werden kann. An den Knoten des Netzes werden die Funktionen $z \mapsto 1, z \mapsto z^1, \ldots, z \mapsto z^n$ berechnet. Einzige freie

Parameter sind die Konstanten $a_0, a_1, \ldots, a_n$. Der Ausgabeknoten sammelt additiv die ankommende Information und gibt den Wert des ausgewerteten Polynoms aus. Die Netzgewichte können analytisch durch die Berechnung der ersten $n+1$ Glieder der zu F gehörenden Taylorschen Reihe ermittelt werden. Interessanter ist jedoch, sie durch einen Lernalgorithmus zu bestimmen.

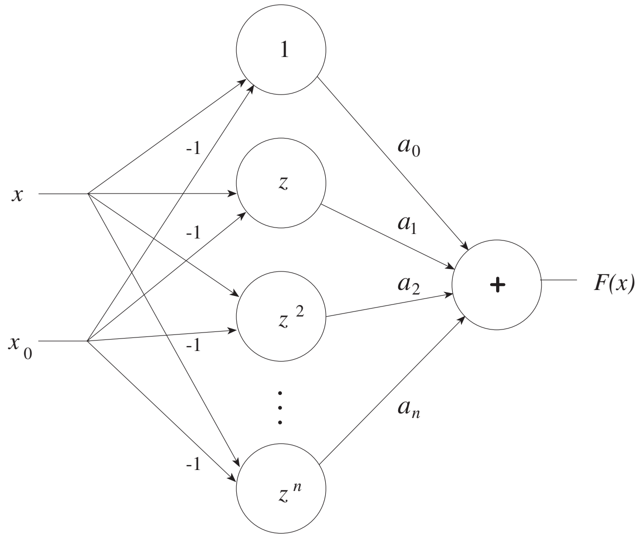

Abbildung 7: Taylorsche Reihe als Funktionennetz

Abb. 8 zeigt, wie eine Fourierreihe als neuronales Netz dargestellt werden kann. Soll die Funktion F als Fourierreihe entwickelt werden, dann hat diese die Form

$$F(x) = \sum_{i=0}^{\infty} (a_i \cos(ix) + b_i \sin(ix)).$$

Ein neuronales Netz mit der Sinus-Funktion als primititiver Knotenfunktion kann eine endliche Anzahl der Glieder der Reihe implementieren. In der Abb. 8 legen die Konstanten $k_0, \ldots, k_n$ die Frequenz der Sinus-Ausgabe eines jeden Knotens fest. Die Konstanten $d_0, \ldots, d_n$, die einfach in die Knoten eingegeben werden, spielen die Rolle von Phasenverschiebungen (so gilt z.B. mit $d_0 = \pi/2$ die Gleichung $\sin(x + d_0) = cos(x)$), und so braucht die Kosinus-Funktion nicht explizit als primitive Funktion in den Knoten aufzutauchen). Die Konstanten $w_0, \ldots, w_n$ spielen die Rolle der Amplituden der Fourierreihe. Wie man sieht, ist das Netz sogar allgemeiner als die konventionelle Formel, da nichtganzzahlige Wellenzahlen k_i erlaubt sind und auch Phasenverschiebungen, die kein mehrfaches von $\pi/2$ sind.

Im Unterschied zu Taylorschen oder Fourierreihen ist bei neuronalen Netzen die zu modellierende Funktion F nur indirekt, d.h. anhand einiger Ein-Ausgabepaare bekannt. Wir kennen die Abbildung F nur an einigen Stellen, möchten aber die beste mögliche Verallgemeinerung daraus ableiten. In unseren beiden Beispielen

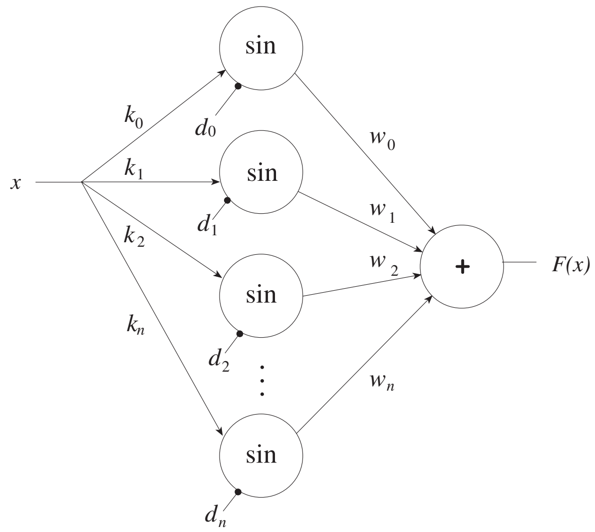

Abbildung 8: Fourierreihe als Funktionennetz

würden wir versuchen, die Netzgewichte optimal zu bestimmen. Im Fall des Netzes für die Taylorsche Reihe kann die klassische Methode der minimalen Quadrate verwendet werden. Im Fall der Fourierreihe kann ein *Lernalgorithmus* verwendet werden. Dafür wird zuerst ein zufälliger Satz von Gewichten ausgewählt und dann so lange iterativ angepasst, bis das Funktionennetz die gesuchte Funktion F optimal approximiert.
Abb. 9 zeigt, wie dies geschehen soll: Die unbekannte Funktion $F : \mathbb{R}^2 \to \mathbb{R}$ soll von einem Netz berechnet werden. Wir kennen nur einige Stützpunkte ihres Graphen. Ein Netz wird mit zufälligen Gewichten initialisiert und berechnet am Anfang nur eine schlechte Approximation der gewünschten Funktion. Im Laufe eines Lernverfahrens (d.h. durch iterative Korrektur der Gewichte) wird die vom Netz berechnete Approximation immer besser, bis letztendlich, im Idealfall, die Graphik der Netzfunktion und die der modellierten Funktion übereinstimmen.
Schon das Beispiel der Taylor- und Fourier-Netze deutet an, daß es möglich sein sollte, eine umfangreiche Klasse von Funktionen durch neuronale Netze zu approximieren. In beiden Fällen würden wir aber eine unendliche Anzahl von Knoten brauchen, um eine Funktion F *exakt* berechnen zu können. Überraschenderweise können jedoch viele nicht triviale Funktionen exakt bereits durch ein endliches Funktionennetz berechnet werden. Ein klassisches Resultat von Kolmogorov bildet dafür die Grundlage: Mehrdimensionale stetige Funktionen können als ein endliches Netz primitiver eindimensionaler Funktionen dargestellt werden.
Der Satz von Kolmogorov in einer moderneren Variante lautet:

Satz 1 *Gegeben sei eine n-dimensionale stetige Funktion $F : [0,1]^n \to [0,1]$. Dann existieren eindimensionale stetige Funktionen g und φ_q für $q = 1, \ldots, 2n+1$*

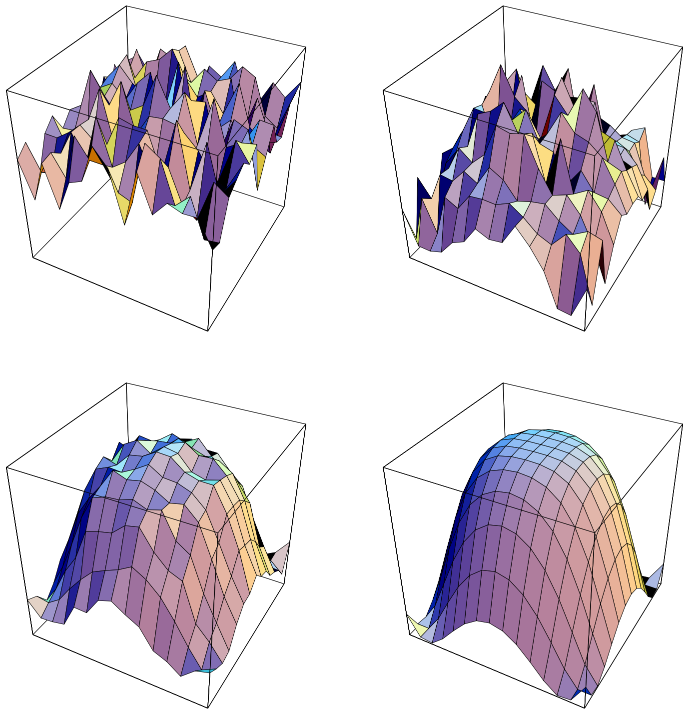

Abbildung 9: Schrittweise Annäherung an die gewünschte Approximation von F

sowie Konstanten λ_p, $p = 1, \ldots, n$, *so daß*

$$F(x_1, x_2, \ldots, x_n) = \sum_{q=1}^{2n+1} g\left(\sum_{p=1}^{n} \lambda_p \varphi_q(x_p)\right).$$

Der Beweis kann bei Sprecher [22] nachgelesen werden.

Abb. 10 zeigt die Verschaltung, die für die Berechnung von $F(x_1, x_2, \ldots, x_n)$ benutzt werden kann. Die Funktionen φ_q können im voraus festgelegt werden, so daß nur die Netzgewichte und die Funktion g zu finden sind. In neuerer Zeit haben andere Autoren einige der Bedingungen des Theorems von Kolmogorov abgeschwächt und die Darstellbarkeit von Lebesgue-integrierbaren Funktionen untersucht. Autoren wie Gallant und White haben Fourierreihen verwendet, um vergleichbare Resultate zu erzielen. Voraussetzung dafür ist nur, daß die Netzknoten irgendeine Form von *squashing function* verwenden, d.h. eine Sigmoide (vgl. 3.1) oder eine ähnliche Funktion. Hornik et al. [7] haben allgemeinere Resultate für eine größere Klasse von Funktionen in den Netzknoten erzielt. Werden außerdem Knoten im Netz zugelassen, die als Bausteine für die Berechnung von Polynomen der Eingabewerte verwendet werden können, garantiert bereits der klassische Satz von Weierstraß-Stone, daß jede reelle stetige Funktion mit

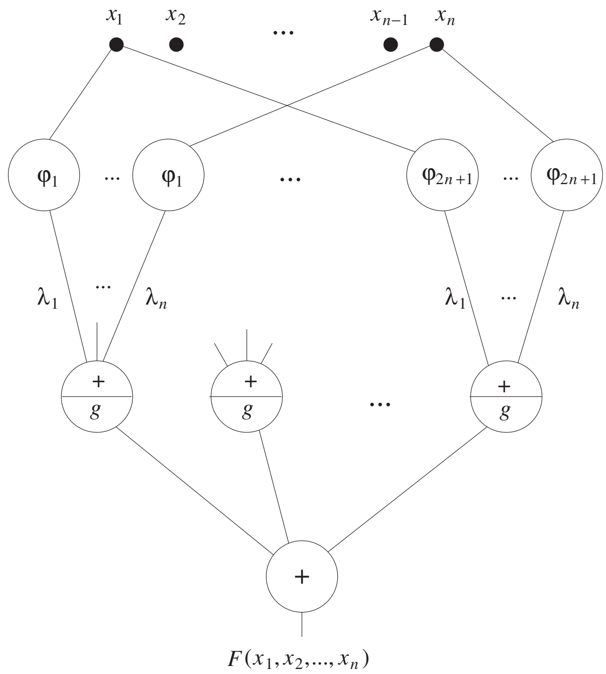

Abbildung 10: Netz für die Berechnung einer stetigen Funktion F

kompaktem Definitionsbereich durch endlich viele Berechnungselemente beliebig genau approximiert werden kann.

1.4 Das Lernproblem ist NP-vollständig

Die Bestimmung der Netzparameter in einem neuronalen Netz ist im allgemeinen ein NP-vollständiges Problem. Das bedeutet, daß kein Algorithmus bekannt ist, der alle Instanzen des Problems in polynomieller Zeit lösen kann und daß höchstwahrscheinlich kein solcher Algorithmus existiert (vgl. dazu Abschnitt 10 des Aufsatzes von S. Koppelberg in diesem Band). Leser, die wenig vertraut mit der Komplexitätstheorie sind, können diesen Unterabschnitt überspringen. Sie sollten aber im Kopf behalten, daß Lernalgorithmen extrem rechenintensiv sein können.
Es kann mit Hilfe des Erfüllbarkeitsproblems bewiesen werden, daß sich ein NP-vollständiges Problem in polynomieller Zeit auf ein Lernproblem für neuronale Netze reduzieren läßt. Das Erfüllbarkeitsproblem wird wie folgt definiert:

Sei V eine Menge von n logischen Variablen und F eine logische Formel in konjunktiver Normalform (Konjunktion von Disjunktionen)[1]*, die nur Variablen aus*

[1] Eine Konjunktion bzw. Disjunktion ist eine durch $\wedge$ (logische und) bzw. $\vee$ (logisches oder) verbundene Aussage.

V enthält. Das Erfüllbarkeitsproblem besteht darin, eine solche logische Belegung für die Variablen in V zu finden, daß F wahr ist.

Es wurde bereits 1971 von Cook bewiesen [5], daß das Erfüllbarkeitsproblem NP-vollständig ist. Dies gilt auch für jenes mit maximal drei Variablen in jeder Disjunktion (in der Literatur 3SAT genannt). Wir transformieren 3SAT in ein Lernproblem für neuronale Netze mit Hilfe des Netzes in Abb. 11. Dabei folgen wir Judd [8], vereinfachen den Beweis aber beträchtlich. Die einzelnen Neuronen sind Perzeptronen. Wir gehen aus von einer Formel F, die n Variablen $x_1, x_2, \ldots, x_n$ enthält und in konjunktiver Normalform, also einer Konjunktion von Disjunktionen ausgedrückt ist, wobei jede Disjunktion maximal drei Literale[2] enthalten soll.
Für jede der logischen Variablen x_i wird ein Neuron mit unbekanntem Gewicht w_i und Schwellenwert 0,5 bereitgestellt. Die Ausgabe des i-ten Neurons in der ersten Schicht wird als eine logische Belegung für die jeweilige logische Variable x_i interpretiert. Die Neuronen mit Schwellenwert $-0,5$ haben nur die Aufgabe, den Wahrheitswert der x_i-Werte umzukehren, so daß die Ausgabe dieser Neuronen als $\neg x_i$ interpretiert werden kann. Die vorletzte Schicht (Klausel-Schicht) von Neuronen implementiert die Disjunktion der angeschlossenen logischen Werte. Wird eine Leitung auf Eins gesetzt, dann feuert das entsprechende Klausel-Neuron. In der Abbildung wurden folgende zwei Klauseln exemplarisch verdrahtet: $x_1 \vee \neg x_2 \vee \neg x_3$ und $x_2 \vee x_3 \vee \neg x_n$. Das letzte Neuron implementiert die Konjunktion der angeschlossenen Disjunktionen. Es wird angenommen, daß die Formel F m Disjunktionen enthält, wobei m als Schwellenwert des letzten Neurons genommen wird. F ist nur dann wahr, wenn alle Disjunktionen wahr sind. Nach diesen Vorüberlegungen kann folgender Satz bewiesen werden.

Satz 2 *Das allgemeine Lernproblem für Netzen aus Perzeptronen ist NP-vollständig.*

Beweis. Eine logische Formel F in konjunktiver Normalform, die n logische Variablen enthält, kann in polynomieller Zeit in die Beschreibung eines Netzes nach dem Muster von Abb. 11 transformiert werden. Es muß dabei für jede Variable ein unbekanntes Gewicht definiert und die Verdrahtung der disjunktiven Klausel festgelegt werden. Offensichtlich erfordert dies, eine angemessene Codierung vorausgesetzt, keinen großen Rechenaufwand, da für die Anzahl m der Disjunktionen in 3SAT $m \leq (2n)^3$ gilt.
Dem Netz wird folgendes Lernproblem gestellt: Eine Eingabe $x = 1$ soll die Ausgabe $F = 1$ erzeugen. Dafür müssen die notwendigen Gewichte gefunden werden. Falls eine Belegung $\mathcal{A}$ der Variablen x_i für die logische Formel F existiert, die diese gültig macht, dann gibt es Gewichte $w_1, w_2, \ldots, w_n$, welche das Lernproblem

[2]Ein Literal ist eine logische Variable v oder seine Negation $\neg v$.

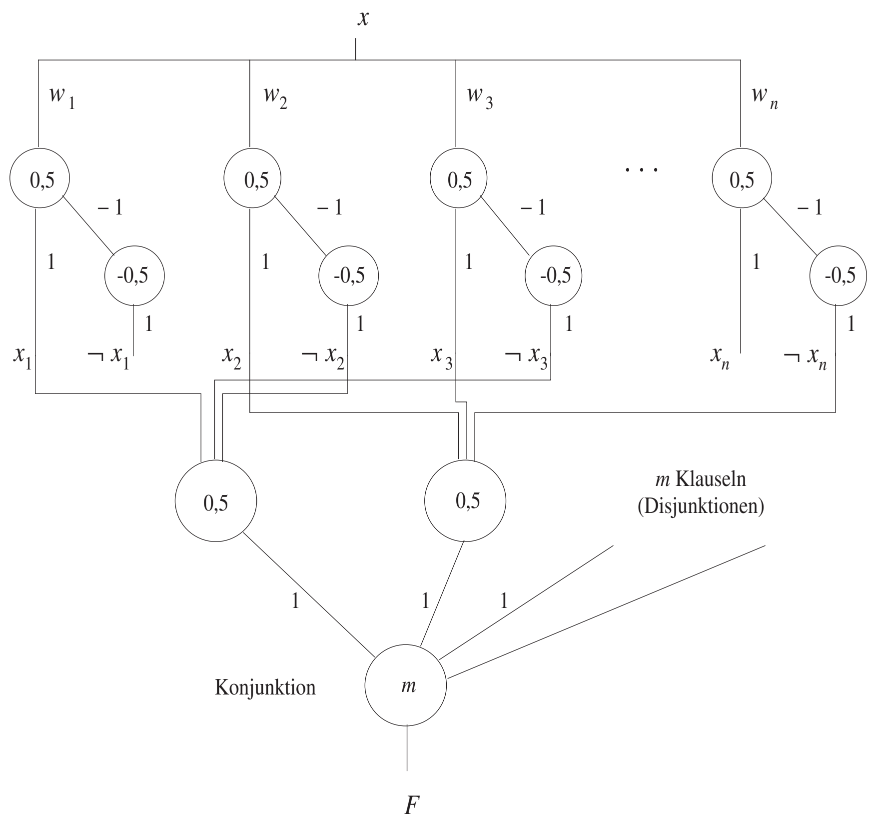

Abbildung 11: Netz für die Lösung des Erfüllbarkeitsproblems

lösbar machen. Dafür brauchen unter der Belegung $\mathcal{A}$ nur folgende Gewichte ausgewählt zu werden: $w_i = 1$, falls $x_i = 1$, oder $w_i = 0$, falls $x_i = 0$. Kurz: $w_i = x_i$
.
Falls andererseits Gewichte $w_1, w_2, \ldots, w_n$ existieren, so daß das Lernproblem lösbar ist, dann führt die Belegung $x_i = 1$, falls $w_i \geq 0,5$, und andernfalls $x_i = 0$ zur Lösung des Erfüllbarkeitsproblems. Damit wäre bewiesen, daß Erfüllbarkeit von logischen Formeln auf ein Lernproblem in einem neuronalen Netz reduzierbar ist. Es muß nur noch gezeigt werden, daß das Lernproblem zur Klasse *NP* gehört, eine Lösung also in polynomieller Zeit überprüft werden kann.
Werden Gewichte $w_1, w_2, \ldots, w_n$ vorgegeben, dann reicht zur Überprüfung, ob die Ausgabe F eine Eins ist, ein einziger Lauf des Netzes. Die dafür notwendigen Berechnungen sind direkt proportional zur Anzahl n der Variablen x_i und zur Anzahl m der disjunktiven Klauseln, die durch ein Polynom beschränkt ist. Die für die Überprüfung notwendige Zeit steigt also polynomiell in n. Das Lernproblem gehört somit zur Klasse *NP*. □

Es könnte uns vorgehalten werden, das oben gestellte Lernproblem sei schwieriger zu lösen als andere typische Fälle, da die meisten Gewichte des Netzes schon im voraus festgelegt wurden. Normalerweise werden alle Gewichte als unbekannte Variablen behandelt. Es könnte daher sein, daß unter diesen Umständen das Lernproblem nicht mehr NP-vollständig ist, weil irgendein anderer Algorithmus den erweiterten Bewegungsraum eventuell besser ausnutzen kann. Das ist aber nicht der Fall. Auch bei Netzen, in denen alle Gewichte unbekannt sind, bleibt der Satz gültig. Dasselbe gilt, wenn die Treppenfunktion in den Neuronen durch eine Sigmoide ersetzt wird [8], [16].

2 Das Perzeptron-Modell

Die Bedingung $x_1 w_1 + x_2 w_2 + \cdots + x_n w_n \geq \theta$, die bei Perzeptronen mit n Eingabeleitungen geprüft wird, ist äquivalent zu einer Teilung des n-dimensionalen Eingaberaumes in zwei Halbräume: Den *"positiven" Halbraum*, dessen Punkte die Bedingung erfüllen und den *"negativen" Halbraum*, für dessen Punkte $x_1 w_1 + x_2 w_2 + \cdots + x_n w_n < \theta$ gilt. Beide Halbräume werden durch die Hyperebene $x_1 w_1 + x_2 w_2 + \cdots + x_n w_n = \theta$ getrennt. Zwei Punktmengen, die von einer Ebene getrennt werden können, werden *linear trennbar* genannt. Das Lernproblem für Perzeptronen besteht darin, vorgegebene positive und negative Punktmengen linear zu trennen. Die Entwicklung der Halbraumtrennung eines Perzeptrons soll hier visualisiert werden, um damit Lernalgorithmen für neuronale Netze anschaulich darstellen zu können. Dafür wird eine Methode entwickelt, die geeigneter ist als die umständlichen herkömmlichen Visualisierungstechniken.

2.1 Visualisierung der Lösungsregionen

Das Problem, das mit einem einzelnen Perzeptron zu lösen ist, kann wie folgt dargestellt werden. Nehmen wir an, $N = \{(0,0),(1,0),(0,1)\}$ und $P = \{(1,1)\}$ seien zwei linear zu trennende zweidimensionale Punktmengen. P sei die Menge, die dem positiven Halbraum, N die Punktmenge, die dem negativen Halbraum zugeordnet werden soll. Dies ist die Trennung, die zur AND-Funktion gehört.
Zwei Gewichte w_1, w_2 und der Schwellenwert θ müssen gefunden werden, um die Einteilung mit einem Perzeptron verwirklichen zu können. Die einzelnen zu erfüllenden Bedingungen sind:

$$\begin{array}{rcl} 0 \cdot w_1 + 0 \cdot w_2 - \theta & < & 0 \\ 1 \cdot w_1 + 0 \cdot w_2 - \theta & < & 0 \\ 0 \cdot w_1 + 1 \cdot w_2 - \theta & < & 0 \\ 1 \cdot w_1 + 1 \cdot w_2 - \theta & > & 0 \end{array}$$

Die vier Bedingungen lauten als Matrix-Ungleichung:

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ -\theta \end{pmatrix} < \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Diese Matrix-Ungleichung beschreibt alle Punkte im Inneren eines konvexen Polytops. Die Seiten des Polytops sind durch die Flächen begrenzt, die jede der Zeilen-Ungleichungen definiert. Als Lösung der gestellten Aufgabe muß also ein Punkt im Inneren des in Abb. 12 gezeigten Polytops gefunden werden.

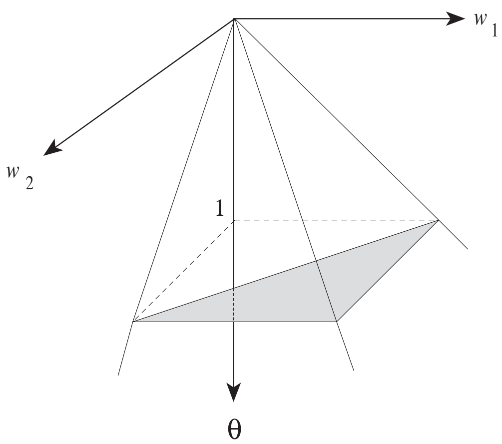

Abbildung 12: Lösungspolytop der AND-Funktion

Das Polytop ist in die θ-Richtung offen, d.h. der absolute Wert des Schwellenwerts kann beliebig groß sein. Es lassen sich immer passende Gewichte w_1 und w_2 finden, mit denen die AND-Funktion berechnet werden kann. Das schattierte Dreieck zeigt die Lösungspunkte für den Fall $\theta = 1$. Für jeden anderen positiven Wert θ gibt es eine ähnliche Lösungsregion in Form eines Dreiecks.

2.2 Die boolesche Kugel

Bei einem Perzeptron, das die OR-Funktion berechnen soll, sind die Werte der zwei Gewichte w_1, w_2 und des Schwellenwerts θ für folgende Ungleichungen gesucht:

$$\begin{array}{lll} \text{Für den Punkt (0,0):} & 0 \cdot w_1 + 0 \cdot w_2 - \theta < 0, & \text{Belegung} = 0 \\ \text{Für den Punkt (0,1):} & 0 \cdot w_1 + 1 \cdot w_2 - \theta > 0, & \text{Belegung} = 1 \\ \text{Für den Punkt (1,0):} & 1 \cdot w_1 + 0 \cdot w_2 - \theta > 0, & \text{Belegung} = 1 \\ \text{Für den Punkt (1,1):} & 1 \cdot w_1 + 1 \cdot w_2 - \theta > 0, & \text{Belegung} = 1 \end{array}$$

Der dreidimensionale Gewichteraum wird durch diese Ungleichungen durch vier unterschiedlichen Ebenen geschnitten, die den Koordinatenursprung enthalten. Die Gleichungen der vier Schnittebenen sind:

Ebene 1:	$\theta = 0$
Ebene 2:	$w_2 - \theta = 0$
Ebene 3:	$w_1 - \theta = 0$
Ebene 4:	$w_1 + w_2 - \theta = 0$

Durch diese vier Schnittebenen im dreidimensionalen Gewichteraum werden maximal 14 verschiedene Regionen erzeugt (s.u.). Jede Region enthält verschiedene Gewichtekombinationen mit denen jeweils eine logische Funktion berechnet werden kann. Von den insgesamt 16 booleschen Funktionen zweier Variablen können also nur 14 durch ein einzelnes Perzeptron berechnet werden. Für die Funktionen XOR und ¬XOR existieren keine Lösungsregionen [1].
Diese Tatsache kann auf einer dreidimensionalen Kugel veranschaulicht werden. Die vier Ungleichungen, die von der Belegung der vier Punkte (0,0), (0,1), (1,0) und (1,1) abhängig sind, definieren ein Lösungspolytop im Gewichteraum. Für einen normierten Gewichtsvektor (w_1, w_2, θ) können die Schnitte der Lösungspolytope mit Hilfe der Einheitskugel betrachtet werden. Die 14 Lösungspolytope erzeugen 14 unterschiedliche Regionen auf der Einheitskugel, wobei jede Region einer logischen Belegung , d.h. einer logischen Funktion entspricht. Abb. 13 zeigt die von den 14 Polytopen erzeugten Regionen auf der Einheitskugel. Die Bezeichnung der Regionen mit vier Bits entspricht der für jede Region erzeugten vier Funktionswerten für die Punkte (1,1), (1,0), (0,1) und (0,0). Die Region 0000 etwa ist die Lösungsregion für die Funktion $(x_1, x_2) \mapsto 0$. Die Region 1000 ist die Lösungsregion für die AND-Funktion.

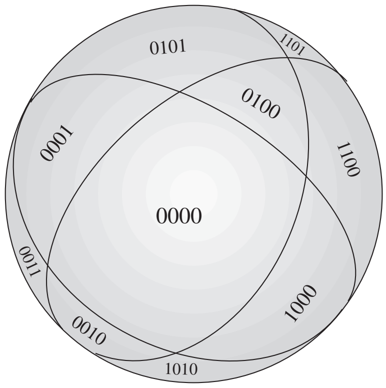

Abbildung 13: Die boolesche Kugel

In Abb. 13 ist deutlich zu erkennen, daß sich die Bezeichnung zweier benachbarter Regionen nur in einem Bit unterscheidet. Die Regionen 0000 und 1000 werden z.B. nur von einem Großkreis getrennt. Region 0000 hat vier Nachbarn, da jedesmal eines der vier Bits verschieden sein kann. Die AND-Region besitzt jedoch nur drei Nachbarn, deren Bezeichnung in einem Bit unterschiedlich ist. Die fehlende Nachbarregion ist die Region 1001, d.h. ¬XOR, für die es somit keine Lösung gibt. Alle Regionen der "booleschen Kugel" haben vier oder drei Kanten.

2.3 Projektion der Lösungsregionen auf die Ebene

Zur Untersuchung der 14 Lösungsregionen auf der Kugel kann deren Projektion auf die Ebene betrachtet werden. Dafür kann die stereographische Projektion benutzt werden, die jedem Punkt P der Einheitskugel einen Punkt der Ebene zuordnet, nämlich den Schnittpunkt der Ebene mit der Geraden durch P und den Nordpol der Kugel. Die stereographische Projektion hat den Vorteil, daß die Großkreise der Kugel, die den Nordpol nicht berühren, auf Ellipsen abgebildet werden. Die vier Schnitte durch vier Ebenen erzeugen vier Großkreise auf der Einheitskugeloberfläche, die auf vier Ellipsen projiziert werden. Für unsere Zwecke sind nur die "logischen" Regionen von Interesse. Deswegen werden die projizierten Großkreise durch Maßstabsveränderung so "normiert", daß die Projektion Kreise produziert. Abb. 14 zeigt das Resultat einer solchen Projektion, wenn das Zentrum der Region 1111 im "Nordpol" liegt.

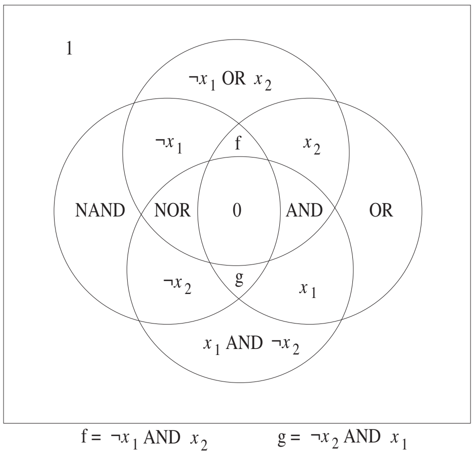

Abbildung 14: Projektion der Lösungsregionen der booleschen Kugel auf die Ebene

Statt die Bezeichnung der Regionen anzugeben, wird in Abb. 14 gezeigt, welche logische Funktion mit den Gewichten einer Region berechnet wird. Vierzehn Regionen für vierzehn boolesche Funktionen sind vorhanden. Offensichtlich kann die Anzahl der Regionen nicht mehr erhöht werden, da vier sich schneidende Kreise auf der Ebene maximal 14 unterschiedliche Regionen definieren können.
Wichtig sind nur die Nachbarschaftsbeziehungen zwischen Regionen. Auf der booleschen Kugel befinden sich Regionen wie AND und $\neg$AND (NAND) auf entgegengesetzten Seiten. Auf der Projektion ist dieser Sachverhalt teilweise zu erkennen. Die Anzahl der Nachbarn einer jeden Region ist vor allem wichtig, wenn ein iterativer Algorithmus verwendet wird, der, angefangen von einer beliebigen Kombination von Gewichten (einem Punkt auf der Kugeloberfläche) zu neuen Gewichten übergeht, wobei der vorhandene Berechnungsfehler minimiert werden soll. Der Perzeptron-Lernalgorithmus [11] geht auf diese Weise vor. Der Algo-

rithmus springt solange von einer Region zur nächsten, bis eine Lösung gefunden wird.
Der Perzeptron-Lernalgorithmus sucht im Gewichteraum die Kombination von w_1, w_2 und θ (im dreidimensionalen Fall), mit der ein Perzeptron eine vorgegebene boolesche Funktion f berechnen kann. Falls das Perzeptron die Funktion $g \neq f$ in Abhängigkeit von w_1, w_2 und θ berechnet, dann ist der Fehler der Approximation die Anzahl der falsch berechneten Funktionswerte. Die Fehlerfunktion kann somit als die Anzahl von falsch berechneten Funktionswerten für jede mögliche Kombination von w_1, w_2 und θ definiert werden. Die Fehlerfunktion für die Berechnung einer booleschen Funktion zweier Variablen kann mit dem Diagramm der Abb. 14 vollständig dargestellt werden. Falls die zu berechnende Funktion $(x_1, x_2) \mapsto 1$ ist, dann ergibt sich die in der Abb. 15 gezeigte Verteilung der Fehlerfunktionswerte.

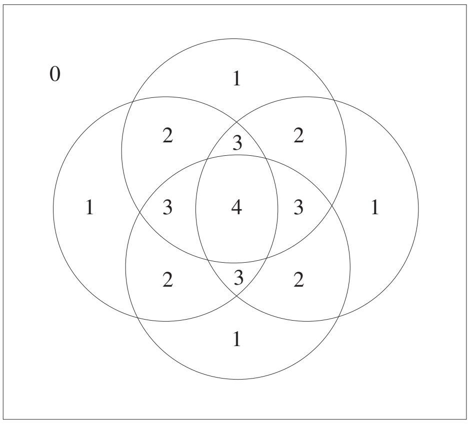

Abbildung 15: Fehlerfunktionswerte für die Berechnung von $(x_1, x_2) \mapsto 1$

Es gibt nur ein globales Maximum und ein globales Minimum. Das globale Maximum wird in der Lösungsregion für $(x_1, x_2) \mapsto 0$ erreicht. Das globale Minimum liegt in der Lösungsregion von $(x_1, x_2) \mapsto 1$. Die beste Strategie, ausgehend von einem zufälligen Gewichtsvektor, besteht darin, die Stufen der Fehlerfunktion herabzusteigen. Aus den Regionen mit Fehler 1 gibt es jeweils nur einen Pfad zu der Region mit Fehler 0. Aus den Regionen mit Fehler 2 sind es 2 unterschiedliche Pfade. Je höher der Fehler, desto mehr alternative Wege gibt es zu der Region mit Fehler 0. Die Abb. 16 zeigt die Fehlerfunktion für die Berechnung der Funktion $(x_1, x_2) \mapsto 1$. Die globalen Maxima und globalen Minima sind deutlich zu erkennen.

2.4 Perzeptron-Lernen

Wir können jetzt den Perzeptron-Lernalgorithmus formulieren. Startbedingung sind zwei endliche Punktmengen P und N in einem n-dimensionalen Raum. Es

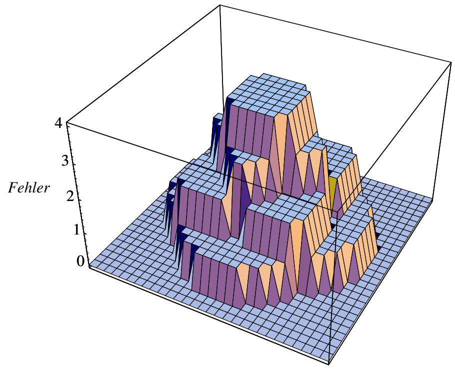

Abbildung 16: Fehlerfunktion für die Berechnung von $(x_1, x_2) \mapsto 1$

wird ein Gewichtsvektor $w = (w_0, w_1, \ldots, w_n)$ gesucht, der beide Mengen linear trennen kann, wobei P im offenen positiven und N im offenen negativen Halbraum liegen soll. Dabei bezeichnen $w_0, \ldots, w_n$ die Gewichte der Eingabeleitungen zum Perzeptron. Ohne Beschränkung der Allgemeinheit kann angenommen werden, daß der Schwellenwert Null ist. Das *Skalarprodukt* zweier Vektoren $a = (a_1, \ldots, a_m)$ und $b = (b_1, \ldots, b_m)$ wird mit $a \cdot b$ bezeichnet, wobei $a \cdot b = \sum_{i=1}^{m} a_i b_i$.

Perzeptron-Lernalgorithmus

Start:	Der Gewichtsvektor $w^{(0)}$ wird zufällig generiert. Setze $t = 0$.
Testen:	Ein Punkt x in $P \bigcup N$ wird zufällig gewählt. Falls $x \in P$ und $w^{(t)} \cdot x > 0$ gehe zu *Testen* Falls $x \in P$ und $w^{(t)} \cdot x \leq 0$ gehe zu *Addieren* Falls $x \in N$ und $w^{(t)} \cdot x < 0$ gehe zu *Testen* Falls $x \in N$ und $w^{(t)} \cdot x \geq 0$ gehe zu *Subtrahieren*
Addieren:	Setze $w^{(t+1)} = w^{(t)} + x$. Setze $t = t + 1$. Gehe zu *Testen*.
Subtrahieren:	Setze $w^{(t+1)} = w^{(t)} - x$. Setze $t = t + 1$. Gehe zu *Testen*.

Dieser Algorithmus [11] nimmt immer dann eine Korrektur am Gewichtsvektor vor, wenn irgendein Punkt in P oder N nicht richtig eingeteilt wurde. Ein Konvergenzsatz für den Perzeptron-Lernalgorithmus garantiert, daß in diesem Algorithmus der Anfangsvektor $w^{(0)}$ höchstens eine endliche Anzahl von Korrekturen

erfährt, wenn die Mengen P und N linear trennbar sind. Der Algorithmus kann in dem Moment gestoppt werden, in dem alle Punkte von P und N richtig klassifiziert worden sind. Dafür müßte ein entsprechender Test im obigen Pseudocode eingebaut werden (nur so ist diese Rechenvorschrift wirklich ein Algorithmus).
Da wir aber bereits gesehen haben, daß Perzeptron-Lernen die Ermittlung von inneren Punkten in n-dimensionalen Polytopen entspricht, kann jeder andere Algorithmus, der solche *inner point* Aufgaben löst, verwendet werden. Dazu zählen der Simplex-Algorithmus der linearen Programmierung und schnellere Varianten, wie der Algorithmus von Karmarkar [16]. Die Gewichte von einzelnen Perzeptronen können also in polynomieller Zeit bestimmt werden. Erst bei einer Vernetzung von Perzeptronen wird das Problem NP-vollständig. Andere Lernverfahren sind dann notwendig.

3 Der Backpropagation-Algorithmus

3.1 Gradientenabstiegsverfahren

Netze mit mehreren Schichten von Berechnungselementen können ein breiteres Anwendungsspektrum als solche mit einer einzigen Schicht abdecken, da die vom Netz implementierte Funktion komplexer sein kann. Entsprechend größer ist aber auch der Trainingsaufwand, da die Anzahl der unbekannten Gewichte eines Netzes mit jedem zusätzlichen Neuron steigt. Deswegen besprechen wir in diesem Abschnitt eine populäre Lernmethode — den Backpropagation-Algorithmus. Dieses numerische Verfahren wurde in den siebziger Jahren von verschiedenen Forschern unabhängig voneinander und für unterschiedliche Anwendungen entwickelt, geriet jedoch in Vergessenheit, bis 1985 Rumelhart und Mitarbeiter den Algorithmus einer breiteren Öffentlichkeit vorlegten [21]. Seitdem ist er zu einer der am weitesten verbreiteten Lernmethoden für neuronale Netze geworden. Backpropagation sucht das Minimum der Fehlerfunktion E eines bestimmten Lernproblems durch Abstieg in der negativen Gradientenrichtung. Die Gradientenrichtung ist die Richtung der stärksten Steigung; sie ist durch die Bildung der partiellen[3] Ableitungen $\partial E/\partial w_i$ in Richtung der Koordinate w_i bestimmbar (s.u.). Die Kombination derjenigen Gewichte eines Netzes, die den Berechnungsfehler minimiert, wird als Lösung des Lernproblems betrachtet. Der Gradient der Fehlerfunktion muß also für alle Punkte des Gewichteraums existieren, d.h. die partiellen Ableitungen der Fehlerfunktion nach den einzelnen Gewichten müssen überall definiert sein. Das bedeutet, daß die einfache Schwellenwertfunktion der Perzeptronen durch eine stetige und differenzierbare Funktion ersetzt werden muß. In der Regel wird dafür die *Sigmoide* s_c genommen, die folgendermaßen definiert

[3]Bei einer Funktion f in mehreren Variablen bezeichnet $\partial f/\partial x_i$ die Ableitung von f nach x_i, wobei die anderen Variablen x_j als Konstanten behandelt werden.

wird:

$$s_c(x) = \frac{1}{1+e^{-cx}}.$$

Für große Werte der Konstanten c ähnelt die Sigmoide der Treppenfunktion der Perzeptronen. Für kleinere Werte wird sie immer flacher.

3.2 Erweiterung des Netzes

Wir wissen bereits, daß ein neuronales Netz einem Funktionennetz entspricht, welches eine Kette von Funktionskompositionen berechnet. Die Parameter der Berechnung sind die Netzgewichte, die am Anfang zufällig ausgewählt werden. Die vom Netz implementierte Netzfunktion nennen wir F. Da wir an einer gewissen Trainingsmenge von Ein-Ausgabepaaren $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ mit $x_i \in \mathbb{R}^n$, $y_j \in \mathbb{R}^k$ interessiert sind, können wir den quadratischen Fehler E wie folgt berechnen:

$$E = (y_1 - F(x_1))^2 + (y_2 - F(x_2))^2 + \cdots + (y_m - F(x_m))^2.$$

Dabei muß beachtet werden, daß der quadratische Fehler E eine Funktion der ℓ Netzgewichte $w_1, w_2, \ldots, w_\ell$ ist. Für die iterative Korrektur der Gewichte muß der Gradient

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_\ell})$$

berechnet werden. Dies erfordert die mehrmalige Anwendung der Kettenregel der Differentialrechnung. Aus diesem Grund zeigen wir in den folgenden Abschnitten, wie die Kettenregel in Funktionennetzen berechnet werden kann.

Zuerst müssen wir anmerken, daß durch die Erweiterung des ursprünglichen Netzes die analytische Behandlung einfacher wird. Nehmen wir an, das erste Ein-Ausgabepaar sei (x_1, y_1). Wir können am Ausgang des Netzes zusätzliche Knoten hinzufügen, die für jede Komponente o_{1i} des Ausgabevektors des Netzes den quadratischen Abstand vom erwarteten Wert y_{1i} für $i = 1, \ldots, k$ berechnen (Abb. 17). Dabei bezeichnen wir mit x_{ij} und y_{ij} die j-te Komponente des i-ten Ein- bzw. Ausgabevektors. Das Ergebnis ist

$$E_1 = (o_{11} - y_{11})^2 + (o_{12} - y_{12})^2 + \cdots + (o_{1k} - y_{1k})^2.$$

Dasselbe kann für jedes Ein-Ausgabepaar gemacht werden, so daß das ursprüngliche Netz in m Netze verwandelt wird, die die quadratischen Fehler $E_1, E_2, \ldots, E_m$ berechnen. Durch ein letztes zusätzliches Berechnungselement, das alle diese Zwischenergebnisse addiert, kann ein Gesamtnetz aufgestellt werden, dessen Netzfunktion dem quadratischen Fehler E entspricht. Das bedeutet, daß wir das Problem der Berechnung des Gradienten von E auf sehr allgemeine Weise lösen können: Wir brauchen nur zu zeigen, daß es eine Methode gibt, um den Gradienten einer differenzierbaren Netzfunktion effizient zu berechnen. Der Algorithmus kann dann für das erweiterte Netz verwendet werden.

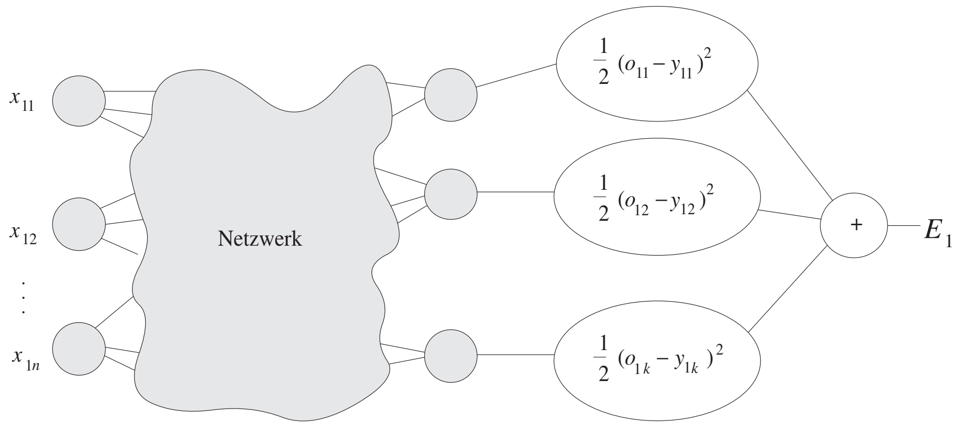

Abbildung 17: Erweiterte Netz für ein Ein-Ausgabepaar

Betrachten wir also das Problem des Differenzierens in allgemeinen Funktionennetzen.

3.2.1 Sequentielle Schaltungen

Nehmen wir an, daß f und g eindimensionale, reelle, differenzierbare Funktionen sind. Sei die Funktionskomposition $f \circ g$. Die partielle Ableitung von $f \circ g$ nach x ist gemäß der Kettenregel $f'(g(x))g'(x)$. Abb. 18 zeigt ein Netz mit zwei Neuronen, wobei deren Ausgabefunktion g bzw. f ist. Die Neuronen besitzen jetzt auch einen linken Teil, in dem die Ableitungen ihrer Aktivierungsfunktionen, also g' und f' berechnet werden. Die Leitungen sind nicht gewichtet. Ein Eingabewert x wird in die Ausgabe $f(g(x))$ verwandelt. Der sogenannte *Feedforward-Teil* des Verfahrens durchläuft das Netz von links nach rechts unter alleiniger Verwendung von Funktionskompositionen. Im linken Teil der Berechnungselemente wird die Ableitung der Aktivierungsfunktion für die eingetroffene Information berechnet und gespeichert. Abb. 18 zeigt den Zustand des Netzes nach der Bearbeitung der Eingabe x.

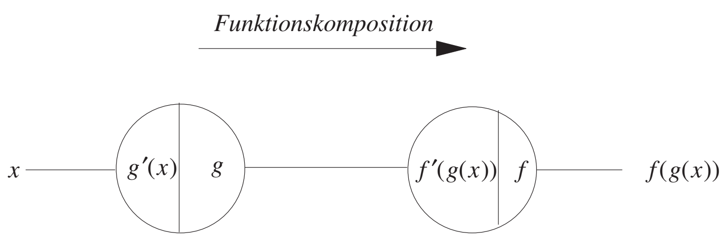

Abbildung 18: Ergebnis des Vorwärtsschritts

Die sogenannte *Backpropagation-Berechnung* verläuft nun von rechts nach links. Angefangen mit dem Traversierungswert 1 wird das Netz rückwärts durchlaufen.

An jedem Knoten wird der Traversierungswert mit dem gespeicherten Wert der Ableitung der jeweiligen Aktivierungsfunktion multipliziert.

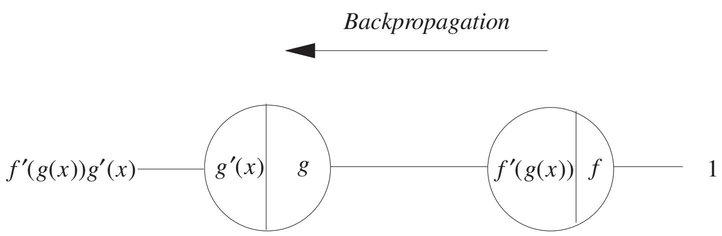

Abbildung 19: Ergebnis des Rückwärtsschritts

Der Traversierungswert ist nach dem Durchgang durch das rechte Neuron $f'(g(x))$. Bei der Traversierung des linken Neurons wird dieser Wert mit $g'(x)$ multipliziert. Das Endresultat ist $f'(g(x))g'(x)$, die gewünschte partielle Ableitung der Funktionskomposition nach x.
Auch die partielle Ableitung von drei oder mehr Funktionen in einer Kompositionskette kann durch diese Methode berechnet werden. Wir können uns vorstellen, daß das Netz rückwärts verwendet wird, wobei die Eingabe 1 ist und an jedem Knoten nicht eine Funktionskomposition, sondern eine Multiplikation mit dem jeweils in der linken Hälfte gespeicherten Wert durchgeführt wird.

3.2.2 Parallelschaltungen

In Backpropagation-Netzen wird nicht nur die Komposition, sondern auch die Addition von Funktionen in den Knoten verwendet. Dies kann als die Komposition der Additionsfunktion mit anderen Funktionen verstanden werden. Abb. 20 zeigt ein Netz, das die Addition der Funktionen f_1 und f_2 berechnet. Jedes Neuron hat wiederum einen linken Teil, in dem die Ableitung der Aktivierungsfunktion ausgewertet wird. Die partiellen Ableitungen der Additionsfunktion sind 1, da

$$\frac{\partial(x_1 + x_2)}{\partial x_1} = \frac{\partial(x_1 + x_2)}{\partial x_2} = 1.$$

Für die Addition der Funktionen f_1 und f_2 gilt $(f_1 + f_2)'(x) = f_1'(x) + f_2'(x)$. Dies kann ebenfalls durch Backpropagation berechnet werden. Im Feedforward-Schritt wird $f_1(x) + f_2(x)$ berechnet, und in den Knoten werden die ausgewerteten Ableitungen der Aktivierungsfunktionen gespeichert. Beim Backpropagation-Schritt wird wieder mit dem Traversierungswert 1 gestartet. Nach dem Plus-Knoten teilt sich die Traversierung in zwei Pfade, die sich später wieder treffen. Für jeden Pfad wird ein Traversierungswert berechnet. Beim Durchlaufen des oberen Pfades wird der Traversierungswert $f_1'(x)$, beim Durchlaufen des unteren wird $f_2'(x)$ berechnet. An der Stelle, an der sich beide Pfade wieder treffen, werden beide Werte addiert. Das Resultat ist die partielle Ableitung von $f_1 + f_2$ nach x, wie in

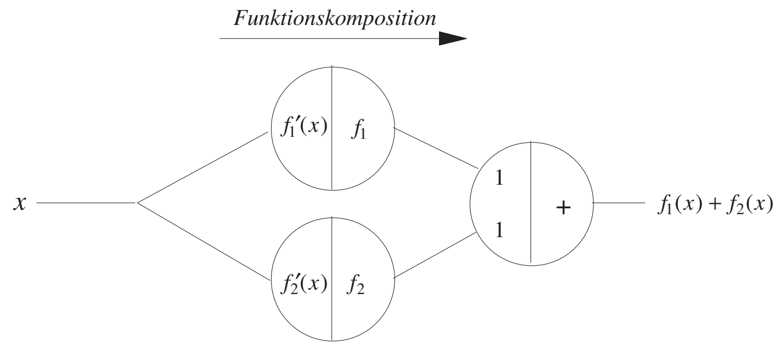

Abbildung 20: Addition von Funktionen

Abb.21 gezeigt wird. Die "parallele" Verknüpfung von Funktionen kann auf mehrere Pfade verallgemeinert werden. Durch Induktion kann gezeigt werden, daß die Berechnung der partiellen Ableitung der Addition von beliebig vielen Funktionen durch einen Backpropagation-Schritt geleistet werden kann.

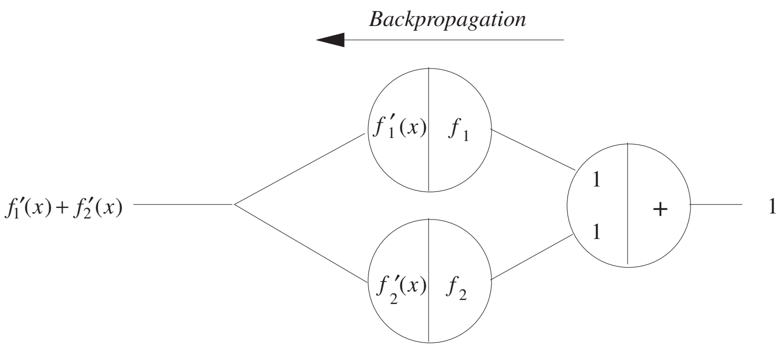

Abbildung 21: Backpropagation-Schritt

Auf diese Weise verfügen wir über eine Methode, mit der beliebige partielle Ableitungen einer Komposition von Funktionen berechnet werden können.

3.2.3 Gewichtete Kanten

Gewichtete Kanten können wie die Funktionskomposition behandelt werden (Multiplikation mit einer Konstanten). Sie lassen sich aber auch einfacher darstellen. Im Feedforward-Teil des Verfahrens wird eine Eingabe x über eine gewichtete Kante einfach mit dem Gewicht w der Kante multipliziert. Im Backpropagation-Teil wird genau dasselbe mit dem Traversierungswert gemacht. Das Resultat ist die partielle Ableitung von wx nach x, nämlich w (Abb. 22). Interessant ist auch, daß durch Vertauschen von x und w im Netz die Feedforward-Berechnung unverändert bleibt; im Backpropagation-Schritt wird jedoch die Ableitung von wx nach w, nämlich x, berechnet. In Backpropagation-Netzen interessiert uns die Ableitung der Fehlerfunktion nach den Gewichten, wobei die Eingabe x als Gewicht und w als die Eingabe behandelt werden muß (siehe [16]).

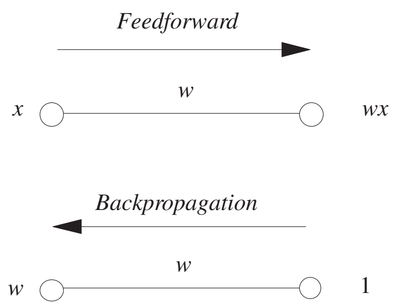

Abbildung 22: Behandlung der gewichteten Kanten

Die Regeln für graphbasierte Berechnung der drei oben behandelten Fälle sind hinreichend zum Verständnis des Backpropagation-Algorithmus. Das Verfahren kann auch auch für Netze, die n reelle Eingaben $x_1, x_2, \ldots, x_n$ verarbeiten verwendet werden. Die partiellen Ableitungen nach jeder Variable werden durch die Traversierung der n Subnetze gefunden, die von der Ausgabe bis zu den n Eingabestellen in die umgekehrte Richtung des Informationsflusses vorhanden sind. Diese Methode kann für jede beliebige vorwärtsgerichtete Netzarchitektur verwendet werden.

3.3 Effizienter Gradientenabstieg

Wir können den Backpropagation-Algorithmus folgendermaßen zusammenfassen:

- Im Vorwärtschritt wird ein Eingabevektor in das Netz eingespeist und das Netz vorwärts durchlaufen. An den Neuronen werden die Ableitungen der Knotenfunktionen ausgewertet und gespeichert. Der Fehler wird am Ausgang des erweiterten Netzes gemessen.

- Im Backpropagation-Schritt wird das Netz rückwärts durchlaufen. Die Eingabe am Ausgang ist eine 1. Die übertragene Information wird an den Knoten addiert und mit den gespeicherten Ableitungen multipliziert. Der Traversierungswert bis zu jedem Gewicht ist die Ableitung der Fehlerfunktion nach diesem Gewicht.

Es gibt eine Reihe von Details, die für die Implementierung des Algorithmus beachtet werden müssen, aber die Grundidee des Verfahrens haben wir offengelegt. Die Detailfragen können in [16], [6], [2] und [3] nachgelesen werden.
Der Backpropagation-Algorithmus dient nur dazu, den Gradienten der Fehlerfunktion nach den Netzgewichten zu bestimmen. Dies liefert die Richtung für den nächsten Iterationsschritt bei der Suche nach dem Minimum der Fehlerfunktion.

Die *Schrittlänge* der iterativen Korrektur kann aber beliebig ausgewählt werden, und dies führt zu einer Reihe von schwierigen Problemen. Ist die Schrittlänge zu groß, dann kann ein "Tal" der Fehlerfunktion übersprungen werden. Ist die Schrittlänge zu klein, dann konvergiert der Prozeß sehr langsam oder bleibt an einem suboptimalen lokalen Minimum der Fehlerfunktion stehen.
Unerwünschte Oszillationen des Suchverfahrens können durch eine adaptive Strategie vermieden werden. Eine ganze Reihe von Varianten des Backpropagation-Algorithmus sind bis heute vorgeschlagen worden, um den Gradientenabstieg zu beschleunigen. Dazu zählen Strategien wie der Delta-bar-Delta-Algorithmus, Quickprop oder RPROP. Diesen Algorithmen liegt im Wesentlichen folgende Überlegung zugrunde: Hat bei der letzten Iteration die Schrittlänge zu einer Verminderung des Fehlers geführt, so kann sie vergrößert werden. Der Iterationsvorgang wird somit in absteigender Richtung beschleunigt. Trifft man an eine Stelle, wo die Fehlerfunktion wieder ansteigt, so heißt dies, daß jetzt die Schrittlänge reduziert werden muß. Solche Algorithmen können mit einer globalen Schrittlänge oder mit einer unterschiedlichen für jedes Gewicht arbeiten. Einen Vergleich einiger der bis dato vorgeschlagenen Algorithmen findet man in [12].

4 Statistische Eigenschaften von neuronalen Netzen

Wir haben in den vorherigen Abschnitten gesehen, wie ein neuronales Netz verwendet werden kann, um eine gewisse unbekannte Funktion anhand von Beispielen zu approximieren. Solch eine Approximation ist immer nützlich, wenn das zu lösende Problem eine schwierige Struktur hat, die eine analytische Lösung unmöglich oder sehr komplex macht. Dies ist z.B. der Fall in der Robotik und in der Kontrolltheorie. Will man einen Roboterarm mit mehreren Freiheitsgraden steuern, muß dafür eine Kontrollfunktion gefunden und implementiert werden. Eine Alternative hierzu wäre, mit dem Roboterarm mehrere beispielhafte Aufgaben durch manuelle Steuerung zu lösen und die entsprechenden Kontrollparameter zu speichern. Ein neuronales Netz kann mit dieser Trainingsmenge die unbekannte Steuerungsfunktion rekonstruieren. So wird das Steuerungsproblem nicht analytisch sondern mit einem Lernverfahren gelöst.
Es gibt eine andere Klasse von Problemen, bei denen nicht eine Steuerungsfunktion, sondern eine Wahrscheinlichkeitsverteilung zu lernen ist. Im Fall der automatischen Spracherkennung kann in Abhängigkeit des Kontexts derselbe Vektor von akustischen Merkmalen zu zwei unterschiedlichen Phonemen gehören. Ein neuronales Netz kann die entsprechende Wahrscheinlichkeitsverteilung approximieren, wie in den nächsten zwei Abschnitte gezeigt wird.

4.1 Klassifizierungsnetze

Das Diagramm in Abb. 23 zeigt die allgemeine Struktur eines Klassifizierungsnetzes. Die Eingabe in das Netz ist ein n-dimensionaler Vektor $(x_1, x_2, \ldots, x_n)$ und die Ausgabe ist die dazugehörige Klassifikation. Soll das Netz z.B. zwischen M verschiedenen Klassen unterscheiden, dann besitzt es M Ausgaben. Wenn der Eingabevektor zur Klasse i gehört, dann soll $o_i = 1$ sein, und $o_j = 0$ für alle $j \neq i$. Diese *sparse* Darstellung der Ausgangswerte kann bei der Codierung der Trainingsmenge vereinbart werden. Ein Beispiel wären akustische Eingabevektoren, die als eines von z.B. 64 Phonemen zu identifizieren sind. Das Netz soll lernen, solche Vektoren den entsprechenden Phonemen zuzuordnen. Was passiert jedoch, falls eine unbekannte Eingabe in das Netz eingegeben wird? In einem solchen Fall antwortet das Netz in der Regel mit Ausgabewerten zwischen Null und Eins. Interessant ist dann aber, daß diese Ausgabewerte echte Bayes-Wahrscheinlichkeiten darstellen [15]. So ist o_i die Wahrscheinlichkeit, daß der Eingabevektor x zur Klasse i gehört.

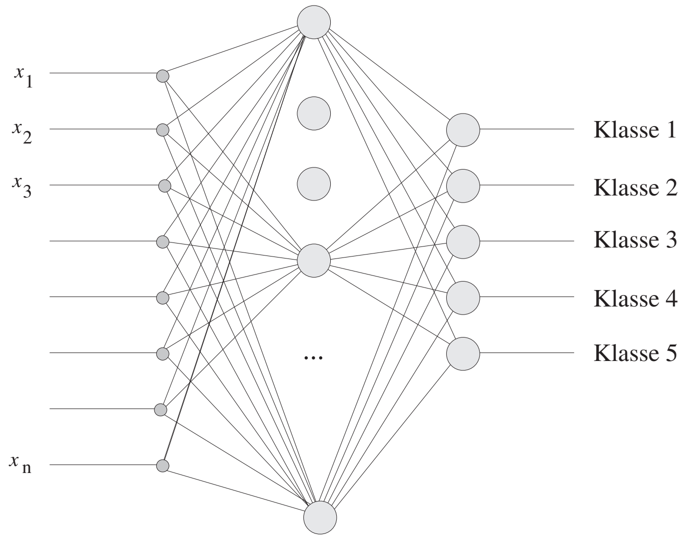

Abbildung 23: Allgemeine Struktur eines Klassifizierungsnetzes

Wir zeigen in diesem Abschnitt, daß erfolgreich trainierte Klassifizierungsnetze Bayes-Wahrscheinlichkeiten berechnen. Wenn das Netz dahingehend trainiert wird, eine vektorielle Eingabe einer von M Ausgabeklassen zuzuordnen, wird eine unbekannte Eingabe x in der Regel keine binären Ausgaben $o_1, o_2, \ldots, o_k$ erzeugen, sondern Zahlen im Intervall $(0, 1)$ produzieren. Wir beweisen folgenden Satz[4].

Satz 3 *Die Ausgabewerte* $o_1, o_2, \ldots, o_k$ *eines trainierten Klassifizierungsnetzes*

[4]Der Beweis kann von einem ungeübten Leser übersprungen werden.

mit M Ausgabeklassen entsprechen den Bayes-Wahrscheinlichkeiten

$$o_i = p(k_i|x)$$

wobei x die vektorielle Eingabe in das Netz ist und $p(k_i|x)$ die bedingte Wahrscheinlichkeit, daß x zur Klasse k_i gehört.

Beweis. Sei die Trainingsmenge T bestehend aus m Ein-Ausgabepaaren der Form (x, t), wobei $x \in \mathbb{R}^n$ und $t \in \mathbb{R}^M$. Die Netzgewichte werden so ausgewählt, daß für die Eingabe x die Differenz zwischen Netzausgabe $o_1, o_2, \ldots, o_M$ und gewünschter Ausgabe $t_1, t_2, \ldots, t_M$ minimiert wird. Dies soll für beliebige zufällig selektierte Trainingsmengen gelten, so daß

$$\Delta = E\{\sum_{i=1}^{M}(o_i(x) - t_i)^2\} \tag{4.1.1}$$

sein Minimum erreicht, wobei $E\{\cdot\}$ den Erwartungswert berechnet. Die gemeinsame Wahrscheinlichkeit einer Eingabe x und der dazugehörigen Klasse i wird mit $p(x, k_i)$ bezeichnet. Nach der Definition des Erwartungswerts kann (4.1.1) wie folgt geschrieben werden:

$$\Delta = \int \sum_{j=1}^{M} \left[\sum_{i=1}^{M}(o_i(x) - t_i)^2\right] p(x, k_j)\, dx \tag{4.1.2}$$

Durch die Ersetzung $p(x, k_j) = p(k_j|x)p(x)$ erhalten wir:

$$\Delta = \int \left[\sum_{j=1}^{M}\sum_{i=1}^{M}(o_i(x) - t_i)^2 p(k_j|x)\right] p(x)\, dx$$

was äquivalent ist zu

$$\begin{aligned} \Delta &= \int \sum_{\ell=1}^{M} \left[\sum_{j=1}^{M}\sum_{i=1}^{M}(o_i(x) - t_i)^2 p(k_j|x)\right] p(x, k_\ell)\, dx \\ &= E\left\{\sum_{j=1}^{M}\sum_{i=1}^{M}(o_i(x) - t_i)^2 p(k_j|x)\right\}. \end{aligned}$$

Durch die algebraische Entwicklung des quadratischen Terms erhalten wir:

$$\Delta = E\left[\sum_{j=1}^{M}\sum_{i=1}^{M}(o_i^2(x)p(k_j|x) - 2o_i(x)t_i p(k_j|x) + t_i^2 p(k_j|x))\right]$$

Da die t_i eine Funktion von x sind und da $\sum_{j=1}^{M} p(k_j|x) = 1$ ist, folgt

$$
\begin{aligned}
\Delta &= E\left\{\sum_{i=1}^{M}\left[o_i^2(x) - 2o_i(x)\sum_{j=1}^{M} t_i p(k_j|x) + \sum_{j=1}^{M} t_i^2 p(k_j|x)\right]\right\} \\
&= E\left\{\sum_{i=1}^{M}\left[o_i^2(x) - 2o_i(x)E\{t_i|x\} + E\{t_i^2|x\}\right]\right\}
\end{aligned}
$$

wobei $E\{t_i|x\}$ und $E\{t_i^2|x\}$ die bedingten Erwartungswerte von t_i und t_i^2 sind. Durch die Addition und Subtraktion von $\sum_{i=1}^{M} E^2\{t_i|x\}$ erhalten wir:

$$
\begin{aligned}
\Delta &= E\left\{\sum_{i=1}^{M}\left[o_i^2(x) - 2o_i(x)E\{t_i|x\} + E^2\{t_i|x\} + E\{t_i^2|x\} - E^2\{t_i|x\}\right]\right\} \\
&= E\left\{\sum_{i=1}^{M}[o_i(x) - E\{t_i|x\}]^2\right\} + E\left\{\sum_{i=1}^{M} var\{t_i|x\}\right\}
\end{aligned}
$$

wobei $var\{t_i|x\}$ die bedingte Varianz von t_i ist. Da der zweite Term unabhängig von der Netzausgabe ist, spielt er für die Minimierung von Δ keine Rolle. Das absolute Minimum von Δ wird also dann erreicht, wenn $o_i(x) = E\{t_i|x\}$. Da aber bei einer Klassifikationsausgabe mit M Klassen nur eine davon auf Eins schalten soll, wissen wir, daß

$$
E\{t_i|x\} = \sum_{j=1}^{M} t_i p(k_j|x) = p(k_i|x),
$$

denn wenn x zur Klasse t_i gehört, so ist $t_i = 1$ und $t_j = 0$, für alle $j \neq i$, und dies bedeutet $E\{t_i|x\} = p(k_i|x)$. Die Ausgabe des Netzes an der i-ten Ausgabeleitung hat bei der Eingabe x den Wert $p(k_i|x)$, d.h. $o_i = p(k_i|x)$, was zu beweisen war. $\square$

Dieses Resultat kann mit Hilfe von Abb. 24 visualisiert werden. In einem Eingaberaum (Merkmalsraum) gibt es verschiedene Cluster, die unterschiedliche Klassen definieren. Die Zugehörigkeit zu einer Klasse ist aber nur probabilistisch gegeben. Ein solcher Merkmalsraum könnte z.B. aus n-dimensionalen Vektoren bestehen, die n verschiedene Krankheitssymptome beschreiben. Die definierten Klassen sind dann bestimmte Krankheiten. Ein Vektor von Symptomen gehört dann mit einer gewissen Wahrscheinlichkeit zu einer gewissen Krankheit. Dies wird in der Abbildung mit stilisierten Gauß-Glocken dargestellt. Cluster von Symptomen können sich sogar überlappen, so daß ähnliche Symptome z.B. zu unterschiedlichen Krankheiten gehören. Solche Überlappungen wären nur durch zusätzliche Information aufzulösen. In einem noch höher dimensionalen Raum könnten die

Cluster damit entflochten werden. Diese zusätzliche Information steht aber normalerweise nicht zur Verfügung, so daß nur probabilistische Aussagen möglich sind. Dies ist aber gerade das, was das neuronale Netz anhand der vorklassifizierten Beispiele lernt.

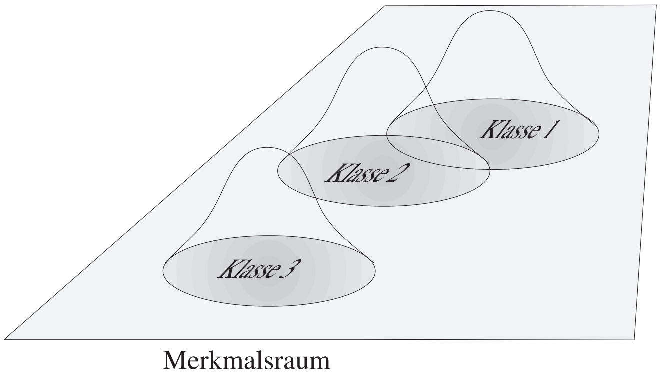

Abbildung 24: Wahrscheinlichkeitsverteilung der Klassenzugehörigkeit

Damit ist schon eine mögliche Anwendung solcher Klassifizierungsnetzen umrissen worden, nämlich der Einsatz in der Medizin. Existierende medizinische Datenbanken können als Trainingsmenge für ein Netz genommen werden. Das Netz erstellt dann eine erste statistische Prognose anhand der Symptome, und ein Arzt kann die Bayes-Wahrscheinlichkeiten in seine Überlegungen einbeziehen.

4.2 Ein Beispiel: NETtalk

Ein elegantes Beispiel für die Anwendung von Klassifizierungsnetzen finden wir bei der Sprachsynthese. Systeme für die Ausgabe von Sprache werden schon seit Jahren kommerziell angeboten. Sie transformieren Zeichenketten durch die Anwendung linguistischer Regeln in eine Reihe von sprachlichen Phonemen. Die dafür notwendige Regelmenge ist keineswegs trivial.
Mitte der achtziger Jahre wurde ein Backpropagation-Netz entwickelt, das ohne eingebaute linguistische Regeln in der Lage war, englische Texte mit einer ähnlichen Erfolgsquote wie die komplexeren Systeme auszusprechen. Das NETtalk-Netz besteht aus 7 Gruppen von 29 Eingabestellen, 80 Neuronen in der mittleren Schicht und 26 Ausgabeneuronen (Abb. 25, wobei die Verbindungsleitungen nicht gezeichnet wurden). Der vorzulesende Text wird durch ein Fenster gescannt. In jedem Schritt werden sieben Buchstaben abgetastet. Jeder davon setzt eine der 29 zugehörigen Eingabestellen auf 1 und den Rest auf 0. Die Aufgabe des Netzes besteht darin, die richtige Aussprache für die im Textfenster gezeigten Buchstaben zu produzieren. Insgesamt können 26 Phoneme selektiert werden. Das Netz besitzt rund 18000 Gewichte. Die große Anzahl von Freiheitsgraden erlaubt es, selbständig statistische Regelmäßigkeiten herauszufinden, die gewisse linguistische Regeln widerspiegeln.

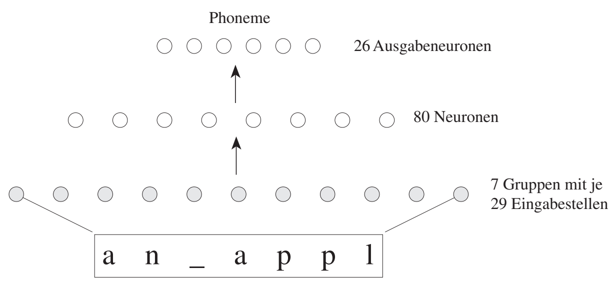

Abbildung 25: Die NETtalk-Architektur

Das Netz wird mit einem Text und dessen Codierung in Phoneme trainiert. Dafür müssen mehrere hundert Worte per Hand codiert werden. Nach dem Training wird dem Netz ein unbekannter Text mit neuen Worten vorgelegt. Die Aussprache des Netzes ist qualitativ ähnlich der von regelbasierten Systemen.
Ein interessantes Phänomen des NETtalk-Lernprozesses ist die Tatsache, daß das Netz am Anfang des Trainings ähnliche sprachliche Schwächen wie Vorschulkinder zeigt. Im Laufe des Lernens werden die Fehler allmählich korrigiert. Eine Analyse der Neuronengewichte zeigt, daß einige Neuronen der verborgenen Schicht sich auf spezielle linguistische Regeln spezialisieren. So scheint sich die Vermutung zu bestätigen, daß wir selber nichts anderes als statistische geometrische Lernsysteme sind.

5 Spracherkennung als Anwendungsbeispiel

Bei der automatischen Spracherkennung geht es darum, ein akustisches Signal in den entsprechenden Text zu transformieren. Es handelt sich also gerade um das zu NETtalk inverse Problem. Die Lösung zu finden, ist aber viel schwieriger als im Fall der Sprachsynthese. Der Grund dafür liegt in der höhen Sensitivität des Erkennungsprozesses in Abhängigkeit vom Kontext. Auch kognitive Faktoren spielen sicherlich eine Rolle, denn wir Menschen sind in der Lage, Sprache auch unter extremen Rauschbedingungen richtig zu erkennen. Dies alles deutet daraufhin, daß nur statistische Modellierung der Sprachsignale zum Ziel führen kann, wenn wir nicht gigantische Systeme bauen wollen, die auf unzählige symbolische Regeln zugreifen.
Bei der automatischen Spracherkennung sind viele unterschiedliche Ansätze getestet und implementiert worden. Eine populäre Methode besteht darin, die akustischen Signale für bestimmte Worte zu speichern und sie als *Schablone* zu benutzen. Soll ein unbekanntes Signal analysiert werden, wird es einfach mit den vorhandenen Schablonen verglichen und diejenige ausgewählt, die den kleinsten Abstand zu dem Signal hat. Diese Methode hat den Nachteil, daß nur eine begrenzte Anzahl von Schablonen verarbeitet werden kann. Darüber hinaus ist die

Methode sprecherabhängig und kann nur isolierte Wörter aus einem Wörterbuch erkennen und keine fließend gesprochenen Texte.
Bei dem Ansatz, den wir besprechen, werden neuronale Netze eingesetzt, um flüssig gesprochene Sprache sprecherunabhängig zu erkennen. Wir beschränken uns auf die schematische Darstellung der verwendeten Prinzipien. Eine umfassende Behandlung des Themas findet man in [10] und [13].

5.1 Vorverarbeitung der phonetischen Signale

Sollen aus dem Sprachsignal Phoneme (Laute) erkannt werden, so ist es naheliegend, ein Klassifizierungsnetz dafür zu verwenden. Wir möchten also eine Art Mustererkennung über dem Sprachsignal durchführen. Hierfür ist es nützlich, eine Vorverarbeitung des Signals durchzuführen. Eine häufig verwendete Methode besteht darin, das Signal in kleine Segmente (Fenster) zu teilen, die z.B. nur 10 Mikrosekunden lang sind. Es wird dann eine *diskrete Fouriertransformation* des Signals in jedem Fenster durchgeführt. Abb. 26 zeigt ein *Spektrogramm*, d.h. die graphische Darstellung der in dem Sprachsignal enthaltenen Frequenzen nach der Zeit. Frequenzen werden auf der senkrechten Achse dargestellt, die Zeit auf den horizontalen. Die Amplituden der Fouriertransformation für jede Frequenz werden als Grauwerte im Diagramm dargestellt.

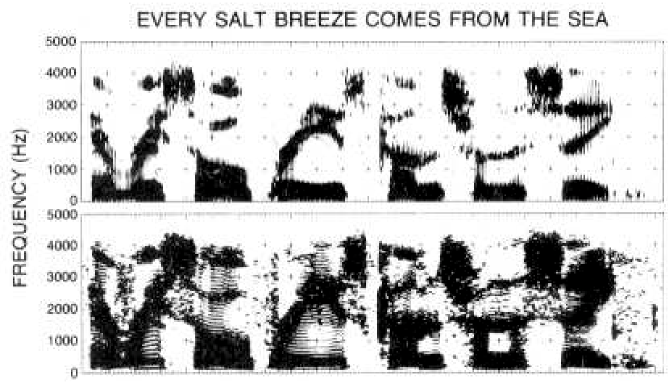

Abbildung 26: Spektrogramm des Satzes "Every salt breeze comes from the sea"

Abb. 26 zeigt auch einige typische Muster, die in Spektrogrammen zu finden sind. Stimmhafte Laute, d.h. jene bei denen die Stimmbänder schwingen, machen sich durch breite Bänder im Spektrogramm bemerkbar. Diese Bänder, die Resonanzfrequenzen des vokalen Traktes, werden Formanten genannt. So gesehen besteht das Problem der Erkennung von stimmhaften Lauten in der Ermittlung der vorhandenen Formanten-Muster. Diese Muster sind allerdings ziemlich variabel. Weiterhin kommen die nicht stimmhaften Laute, wie "sch" oder "t" dazu, die keine so sauberen Muster wie die stimmhaften Laute erzeugen. Wir begnügen uns also mit einer statistische Aussage. Sind zum Zeitpunkt t die Frequenzen $f_1, f_2, \ldots, f_n$ mit Amplituden $a_1, a_2, \ldots, a_n$ vorhanden, dann möchten wir wissen, welches die bedingte Wahrscheinlichkeit ist, daß Phonem k_i ausgesprochen wurde. Wir möchten also

$$p(k_i|(a_1, a_2, \ldots, a_n))$$

berechnen. Dies kann mit einem Klassifizierungsnetz geleistet werden.

5.2 Trainings des Klassifizierers

Abb. 27 zeigt die Architektur eines Netzes, das bei einem an der Universität von Kalifornien entwickelten Spracherkenner zum Einsatz gekommen ist [10]. Die Ausgabe des Netzes besteht aus 64 Ausgabewerten, die anzeigen sollen, welches von 64 Phonemen in das Netz eingegeben wurde. Idealerweise sollte nur eine der Ausgabeleitungen auf Eins schalten, während die anderen "stumm" (d.h. bei Null) bleiben. Die Sprachsignale werden segmentiert, und von jedem Fenster werden 18 Koeffizienten genommen. Dies können die Amplituden der diskrete Fouriertransformation sein, wobei nur 18 unterschiedliche Frequenzbereiche berücksichtigt werden. Es können aber auch zusätzliche Transformationen der Sprachsignale durchgeführt werden, die aber für unsere Zwecke unerheblich sind. Jedoch werden in das Netz nicht nur die 18 Koeffizienten eines Fensters eingegeben, sondern auch der "Kontext" des Signals. Dieser besteht aus den Koeffizienten der sechs Fenster vor und sechs Fenster nach dem aktuellen Fenster. Dies ergibt insgesamt $13 \times 18 = 234$ reelle Eingabewerte für das Netz. Die gezeigte Architektur kommt damit auf etwa 300000 Gewichte.

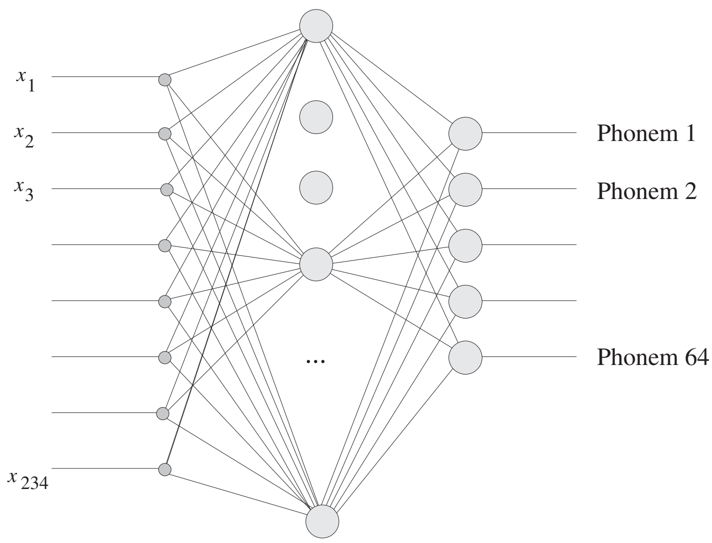

Abbildung 27: Bayes-Netz für die Klassifikation der Phoneme

Für das Training des Netzes werden viele Stunden sprachlicher Daten verwendet.

Da man für diese Daten die korrekte Transkription kennt, ist es relativ einfach, die Korrespondenz zwischen Text und segmentierten Daten herzustellen. Dies kann manuell, oder besser noch semiautomatisch gemacht werden. Für die englische Sprache existieren bereits umfangreiche Traniningsmengen, die auf CD-ROM geliefert werden. Damit können unterschiedliche Sprachsysteme anhand desselben Corpus getestet werden.

Das Netz wird nun mit den Ein-Ausgabepaaren trainiert, wofür der Backpropagation-Algorithmus verwendet werden kann. Wegen der hohen Anzahl der Gewichte und der großen Trainingsmenge nimmt der Lernvorgang viel Zeit in Anspruch. Durch schnelle parallele Hardware können aber die Lernzeiten auf ein vernünftiges Maß reduziert werden. Dies erlaubt ein schnelles Prototyping und die Einbettung des Netzes in traditionelle Spracherkenner, die z.B. mit Hidden-Markov-Modellen arbeiten.

5.3 Bestimmung des Pfades maximaler Wahrscheinlichkeit

Im normalen Betrieb kann das Klassifizierungsnetz verwendet werden, um die Wahrscheinlichkeit zu ermitteln, daß zum Zeitpunkt t das Phonem i ausgesprochen wurde. Wir erhalten damit für die Zeit $t = 1$ bis $t = n$ eine Reihe von Vektoren V_i mit 64 Komponenten, die die bedingten Wahrscheinlichkeiten jedes Phonems zum Zeitpunkt t_i ausdrücken. Man könnte natürlich für jeden Zeitpunkt t_i das Phonem mit der höchsten Wahrscheinlichkeit auswählen. Dieses einfache Kriterium scheitert aber in der Praxis, da sprecherunabhängige Daten zu viel Variabilität enthalten.

Ein besseres Auswahlkriterium besteht darin, den Pfad der maximalen Wahrscheinlichkeit von $t = 1$ bis $t = n$ zu ermitteln. Dafür kann zusätzliche Information verwendet werden, wie z.B. die Wahrscheinlichkeit des Übergangs von jedem Phonem zu jedem anderen. Ist die Wahrscheinlichkeit des Übergangs von Phonem q auf Phonem r Null, dann wäre es sicherlich ein Irrtum, zum Zeitpunkt t_1 das Phonem q und zum Zeitpunkt t_2 das Phonem r auszuwählen. Eine andere Kombination würde eine höhere Gesamtwahrscheinlichkeit aufweisen.

Abb. 28 zeigt die Art von Berechnung, die durchgeführt werden muß. Für jeden Zeitpunkt t haben wir 64 bedingte Phonem-Wahrscheinlichkeiten. Alle diese 64 Werte werden mit der nächsten Spalte von Phonem-Wahrscheinlichkeiten verbunden, wobei die Wahrscheinlichkeit eines Übergangs als Gewicht jeder Kante notiert wird (nicht alle Verbindungen wurden gezeichnet, um die Abbildung übersichtlich zu halten). Durch die Methoden der dynamischen Programmierung oder anderer passender Algorithmen kann dann der Pfad der maximalen Wahrscheinlichkeit bestimmt werden (schattierte Felder). Wichtig dafür ist es, die Pfade, die verfolgt werden, passend zu segmentieren. Dies kann z.B. durch "silence" oder "word spotting" gemacht werden. Wo Pausen erkannt werden, kann der aktuelle

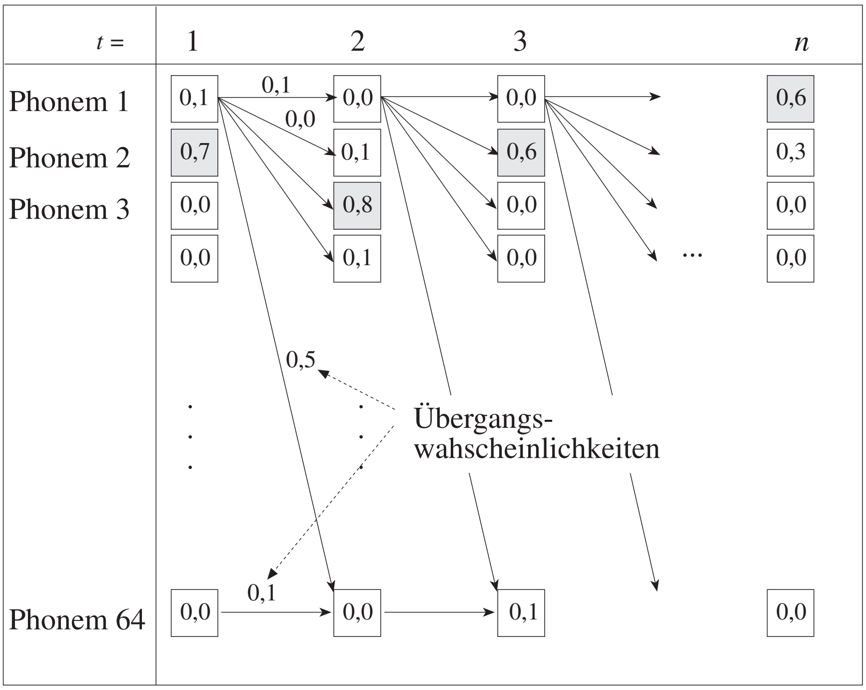

Abbildung 28: Bestimmung des Pfades maximaler Wahrscheinlichkeit

Pfad abgebrochen und ein neuer Pfad gestartet werden. Dies verhindert, daß die Pfadwahrscheinlichkeit letztendlich zu stark abnimmt, so daß Alternativen sich nur gering voneinander unterscheiden.
Selbstverständlich kann viel mehr Information in die Berechnung eingebracht werden. Mit Hilfe eines Wörterbuches können unsinnige Phonemkombinationen ausgeschlossen werden. Wird noch dazu die Grammatik der Sätze überprüpft, können ungrammatikalische Sätze ebenfalls ausgeschlossen werden. “State of the art” Systeme, die diese und mehr Information verwenden, erreichen bereits eine Erkennungsquote von 95% flüssig gesprochener Sprache. Wird dazu das System noch mit der Zeit an den Anwender angepaßt (durch die graduelle Veränderung der Gewichte des Klassifizierungnetzes), so ist es möglich, noch höhere sprecherabhängige Erkennungsraten zu erzielen.

6 Fazit

Wir haben in diesem Aufsatz gezeigt, was neuronale Netze sind, und einige exemplarische Anwendungen besprochen. Damit haben wir aber eigentlich nur an der Oberfläche eines sehr breiten Gebietes gekratzt. Außer den vorwärtsgerichteten Netzen, die hier besprochen wurden, gibt es eine große Anzahl alternativer Modelle. Dazu zählen u.a. Hopfield-Netze, zeitverzögerte Netze, das Kohonen-Modell und Fuzzy-Kontroller. Allen diesen Modellen ist aber eines gemeinsam:

Ihr Substrat ist ein gerichteter Graph von Berechnungselementen.
Durch die im Text behandelten Anwendungsbeispiele sollte klar sein, daß neuronale Netze kein exotisches Berechnungsinstrument sind. Sie sind aus einer biologischen Motivation hervorgegangen, haben sich aber bereits unabhängig gemacht und sind zu einem zusätzlichen Werkzeug im Werkzeugkasten der Informatiker und Mathematiker geworden. Vor allem ihre Fähigkeit, unbekannte Funktionen und Wahrscheinlichkeitsverteilungen approximieren zu können, kann für die statistische Modellierung verwendet werden. Dies befreit niemanden davon, weiterhin Statistik zu lernen, um umfangreiche Datensätze zu verarbeiten, liefert aber eine Systematik, um dies in bestimmten Anwendungen ohne großen Aufwand zu tun. Wir hoffen, daß dieser kurze Aufsatz dazu beiträgt, den Leser für dieses Gebiet zu interessieren, indem das Geheimnis, das hinter der Bezeichnung "neuronale Netze" steckt, gelüftet wird. Die Bezeichnung *Funktionennetze* ist sicherlich mathematisch präziser, aber die ursprüngliche Bezeichnung *neuronale Netze* ist zum einem schöner, zum anderen anspruchsvoller, weil sie uns ständig daran erinnert, daß das Verstehen menschlicher Intelligenz das Hauptanliegen des konnektionistischen Unternehmens war und ist.

Danksagung. Ich bedanke mich bei Prof. R.-H. Schulz, M. Pfister, G. Feuer und H. Dörr, deren Kommentare zur Verbesserung dieses Aufsatzes beitrugen.

Literatur

[1] Abu-Mostafa, Y. und J. St. Jacques (1985), "Information Capacity of the Hopfield Model", *IEEE Transactions on Information Theory*, Vol. IT-31, Nr. 4, S. 461-464.

[2] Brause, R. (1991), *Neuronale Netze: Eine Einführung in die Neuroinformatik*, B.G. Teubner, Stuttgart.

[3] DARPA (1988), *DARPA Neural Network Study*, AFCEA International Press, Fairfax.

[4] Denby, B. (1993), "The Use of Neural Networks in High-Energy Physics", *Neural Computation*, Vol. 5, S. 505-549.

[5] Garey, M. und D. Johnson (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York.

[6] Hertz, J., A. Krogh und R. Palmer (1991), *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City.

[7] Hornik, K., M. Stinchcombe und H. White (1989), "Multilayer feedforward networks are universal approximators", *Neural Networks* Vol. 2, S. 359-366.

[8] Judd, J.S. (1990), *Neural Network Design and the Complexity of Learning*, MIT Press, Cambridge, Massachusetts.

[9] Minsky, M. (1967), *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs.

[10] Bourlard, H., Morgan, N. (1993), *Connectionist Speech Recognition*, Kluwer.

[11] Minsky, M und S. Papert (1969), *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Massachusetts.

[12] Pfister, M., Rojas, R. (1993), "Speeding-up Backpropagation: A Comparison of Orthogonal Techniques", *International Joint Conference on Neural Networks*, Nagoya, Japan.

[13] Rabiner, L., Bing-Hwang, J. (1993), *Fundamentals of Speech Recognition*, Prentice-Hall International, London.

[14] Ritter, H., T. Martinetz und K. Schulten (1990), *Neuronale Netze: Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke*, Addison-Wesley, Bonn.

[15] Richard, M. D., Lippmann, R. P. (1991), "Neural Network Classifiers Estimate *a posteriori* Probabilities", *Neural Computation*, Vol. 3, N. 4, S. 461-483.

[16] Rojas, R. (1993), *Theorie der neuronalen Netze – Eine systematische Einführung*, Springer-Verlag, Berlin.

[17] Rojas, R. (1994), "Who invented the computer – The debate from the Viewpoint of Computer Architecture", in: W. Gautschi (Hrsg.), *Mathematics of Computation 1943-1993, Proceedings of Symposia on Applied Mathematics*, AMS, Vancouver, 1994.

[18] Rojas, R. (1994), "Oscillating Iteration Paths in Neural Networks Learning", erscheint in: *Computers & Graphics - International Journal*, Vol. 18, N. 4.

[19] Rojas, R. (1993), "A Graphical Proof of the Backpropagation Learning Algorithm", W. Malyshkin (Hrsg.), *Parallel Processing Technologies 1993*, Obninsk, Russia.

[20] Rojas, R., Pfister, M. (1994), "First and second-order gradient descent in SIMD computers", *IMACS 14th World Congress on Computation and Applied Mathematics*, Atlanta, July 11-15, 1994.

[21] Rumelhart, D. und J. McClelland (1986), *Parallel Distributed Processing*, MIT Press, Cambridge, Massachusetts.

[22] Sprecher, D. (1964), "On the structure of continuous functions of several variables", *Transactions of the American Mathematical Society*, Vol. 115, S. 340-355.

[23] Ungerer, T. (1993), *Datenflußrechner*, B.G. Teubner, Stuttgart.