Zusammenfassung

Diese Diplomarbeit bestand daraus, die Theorie der Ausdünnungsverfahren für neuronale Netze vorzustellen, mehrere davon im Stuttgarter Neuronale Netze Simulator (SNNS) zu implementieren, einige Testläufe durchzuführen und die Verfahren abschließend zu bewerten.

In dieser Arbeit implementierte ich Backpropagation mit Weight Decay, Magnitude Based Pruning, Optimal Brain Damage, Optimal Brain Surgeon und Skeletonization. Einige zusätzliche interessante Verfahren wurden vorgestellt.

Untersucht wurde das Verhalten der Ausdünnungsverfahren hinsichtlich der Leistung der Netze bei der Klassifizierung der Trainingsmuster und der Testmuster (Generalisierung). Ein weiterer wichtiger Punkt war die Merkmalsextraktion, bei der man versucht, aus einer großen Menge von Eingabewerten die relevanten Kriterien herauszufiltern.

Für die Testläufe verwendete ich ein kleines Netz zur Erkennung von Großbuchstaben, ein Netz zur Erkennung von 1000 handgeschriebenen Ziffern sowie drei Testbeispiele mit akustischen Daten aus der subjektiven Geräuschnachbeurteilung von Generatoren, Startern und Motoren, die mir die Firma Robert Bosch GmbH freundlicherweise zur Verfügung stellte.

Die Tests ergaben eine höchstens gleich gute, in manchen Fällen schlechtere Generalisierungsleistung, was in Einklang mit anderen Forschungsarbeiten steht. Die Klassifizierungsleistung der ausgedünnten Netze stand den ursprünglichen Netzen nicht nach.

Positiv war, daß die Verfahren gute Ergebnisse hinsichtlich der Merkmalsextraktion ergaben. Von einer sinnvollen Anfangsarchitektur konnten in den meisten Fällen 80 bis 90 Prozent der Verbindungen und Neuronen gelöscht werden.

Vorbemerkung

An dieser Stelle möchte ich ganz herzlich all denen danken, die mich in der Verwirklichung dieser Arbeit unterstützt haben. Besonderer Dank gilt meinem Betreuer, Andreas Zell, der mir mit Rat und Tat zur Seite stand und mir in mancher Situation durch ein paar ermunternde Worte die Richtung wies. Sehr geholfen haben mir auch die Mitarbeiter der SNNS-Gruppe an der Universität Stuttgart, die ich des öfteren mit meinen Problemen beschäftigt habe. Außerdem danke ich Heidi, Kai und Jürgen für ihre Korrekturen zu dieser Dokumentation und allen anderen, die mir im Laufe meiner Arbeit unter die Arme gegriffen haben. Ich erkannte im Laufe des vergangenen halben Jahres, daß wissenschaftliches Arbeiten ohne Feedback und ohne Austausch von Erfahrungen kaum möglich ist.

Nicht zuletzt möchte ich mich bei Herrn Dr. Attia von der Firma Robert Bosch GmbH bedanken, der mich mit Beispieldaten für Testläufe meiner Verfahren versorgte.

Tobias Schreiner, Juli 1994

Inhaltsverzeichnis

Notationen

Ich möchte vorab häufig verwendete mathematische Notationen und Abkürzungen in alphabetischer Reihenfolge auflisten, um dem Leser das Verständnis zu erleichtern.

C	Kostenterm bei Weight Decay
E	Netzfehler (auch Fehler- oder Energiefunktion)
F	Netzfunktion $O = F(W, In)$
G	Gradientenvektor $G = \frac{dE}{dW}$
H	Hesse-Matrix $H = \frac{d^2E}{dW^2}$
I	Einheitsmatrix
In, $In^{[p]}$.	Eingabevektor (bei Muster p)
O, $O^{[p]}$ ...	Ausgabevektor (bei Muster p)
T, $T^{[p]}$ ...	Vektor der erwarteten Ausgabe (bei Muster p)
W	Gewichtsvektor
δ_j	Delta für Backpropagation
η	Lernfaktor
θ_j	Schwellenwert des Neurons j
d	Weight-Decay-Faktor
$g_{(ij)}$	Komponente des Gradientenvektors
$h_{(ij),(kl)}$ ...	Element der Hesse-Matrix
in_{pj}, in_j ..	Aktivierung des Eingabeneurons j (bei Muster p)
n_e	Anzahl der Epochen
n_h	Anzahl der verdeckten Neuronen
n_i	Anzahl der Eingabeneuronen
n_n	Anzahl der Neuronen
n_o	Anzahl der Ausgabeneuronen
n_p	Anzahl der Muster
n_w	Anzahl der Verbindungen

o_{pj}, o_j	tatsächliche Ausgabe der Zelle j (im Muster p)
p	Laufvariable für Muster
t_{pj}, t_j	erwartete Ausgabe der Zelle j (im Muster p) (teaching input)
w_{ij}	Gewicht der Verbindung von der Zelle i zur Zelle j
w_{min}	Minimalgewicht bei Weight-Decay-Verfahren
$k \succ j$	Es gibt einen gerichteten Weg von Neuron j zu Neuron k
M^T	Transponierte einer Matrix M
M^{-1}	Inverse einer Matrix M

1. Einführung

1.1 Neuronale Netze – Grundlagen

1.1.1 Motivation

Ausgangspunkt der Überlegungen zu *neuronalen Netzen* oder *Konnektionismus* ist der Vergleich der Informationsverarbeitung von Mensch und Computer. Während der Computer dem Menschen in „einfachen“ Anwendungen wie Kopfrechnen, Suche von Zeichenketten in einem Text oder Auswertung großer Datenbestände in Geschwindigkeit und Präzision klar überlegen ist, reichen seine Fähigkeiten bei komplexen Aufgaben wie z. B. Erkennen eines Bildes, Analysieren von Sprache oder Geräuschen sowie Steuern von Bewegungen bei weitem nicht an die eines Menschen oder Tieres heran. Darüberhinaus kann ein Computer nur ihm aufgetragene, in kleine Schritte zerteilte Arbeiten erledigen, nicht aber einen Sachzusammenhang verstehen oder gar lernen. Es ist also naheliegend, die Struktur des menschlichen Gehirns einmal genauer unter die Lupe zu nehmen, um daraus Anregungen für eine Verbesserung der maschinellen Datenverarbeitung zu erhalten.

Ich möchte nur kurz auf die für die Informatik in diesem Zusammenhang wichtigen Erkenntnisse der Neurobiologie eingehen, umfassendere Darstellungen finden sich in [Kandel et al. 91], [Spektrum 84] oder [Zell 94].

Unser Gehirn ist aus Nervenzellen *(Neuronen)* aufgebaut, deren hervorstechende Eigenschaft ist, daß sie Information durch ein veränderbares elektrisches Potential speichern und mit Hilfe von Nervenimpulsen über Verdickungen am Ende ihrer Ausläufer *(Synapsen)* an andere Nervenzellen weitergeben können. Neuronen mit erhöhtem Potential bezeichnet man als *angeregt* oder *aktiviert*. Es wird geschätzt, daß der Mensch ca. 100 Milliarden Nervenzellen besitzt, wobei jede Zelle mit ca. 1000 bis 10.000 „Nachbarn“ verbunden ist.

In der Sprache der Informatik zeichnet sich ein Gehirn also durch seine hohe *Konnektivität* und seine *massive Parallelität* aus. Ein weiterer Vorteil ist die Möglichkeit des *Lernens*, technisch u. a. durch die Veränderung der Stärke einer

Synapse bewerkstelligt, wodurch die Stärke des Impulses reguliert wird, der über eine Verbindung zwischen zwei Neuronen geleitet wird. Jede Synapse wird nur in einer Richtung benutzt. Das „Wissen" ist über eine größere Gruppe von Neuronen verteilt und kann nur in seltenen Fällen lokalisiert oder gar von außen analysiert werden (es ist also i. a. keine Introspektion möglich).

1.1.2 Informationsverarbeitung mit neuronalen Netzen

Die Informationsverarbeitung läuft in biologischen neuronalen Netzen – in groben Zügen – wie folgt ab: Kommen an den eingehenden Verbindungen (genannt *Dendriten*) eines Neurons Impulse an und übersteigt die Summe ihrer Stärken einen bestimmten, dem Neuron eigenen Schwellenwert, so wird das Neuron für eine bestimmte Zeit aktiviert. Alle Neuronen, die aktiviert sind, geben ihr erhöhtes Potential als Impuls über die ausgehenden Verbindungen *(Axone)* an alle nachgeschalteten Neuronen weiter (sie „feuern"). So werden also ununterbrochen Signale ausgetauscht, die Neuronen können ständig zwischen einem angeregten und einem Ruhezustand wechseln. Die Verbindung mit der Umwelt findet z. B. dadurch statt, daß Nervenzellen von Sinneszellen angeregt werden oder daß sie einen Impuls an eine Muskelzelle weitergeben, sodaß sich ein Muskel bewegt.

In der Neuroinformatik verwendet man nur sehr vereinfachte Modelle eines Nervenzellengeflechts. Aus der Biologie übernommen werden i. a. die Neuronen mit ihren Schwellenwerten und ihre Verbindungen mit den Verbindungsstärken oder *Gewichten*. Dieses Modell kann sehr gut durch einen Graphen dargestellt werden, in dem die Neuronen durch Knoten und die Verbindungen durch gerichtete, gewichtete Kanten repräsentiert sind. Ein Ausschnitt aus einem solchen Graph ist in Abb. 1.1 gezeigt.

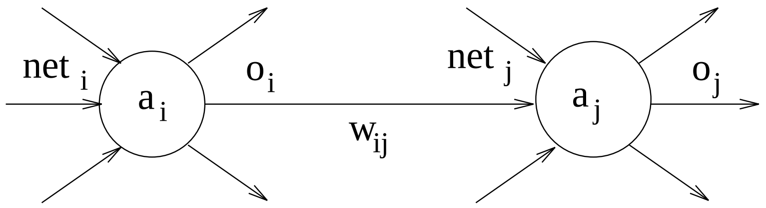

Abbildung 1.1: Schema der Informationsübertragung bei Neuronen

Neuronen werden i. a. charakterisiert durch

- den Aktivierungszustand $a_j(t)$, der den Grad der Anregung (das Potential) einer Zelle j zur Zeit t angibt.

- die Aktivierungsfunktion f_{act}, die angibt, wie sich der Aktivierungszustand eines Neurons j aus der Netzeingabe net_j und dem Schwellenwert θ_j eines Neurons berechnet. Oft gilt die Formel

$$a_j(t+1) = f_{act}(net_j(t) - \theta_j(t)) \tag{1.1}$$

wobei für f_{act} gerne die logistische Funktion $1/(1 + e^{-x})$ oder der tangens hyperbolicus $\tanh(x)$ verwendet wird. Die Netzeingabe net_j ist die Summe aus den gewichteten Ausgaben der Vorgängerneuronen:

$$net_j(t) = \sum_i o_i(t) \cdot w_{ij} \tag{1.2}$$

- die Ausgabefunktionf_{out}, die aus der Aktivierung der Zelle i die Ausgabe o_i berechnet. Zumeist ist dies die Identität, also

$$o_i(t) = f_{out}(a_i(t)) = a_i(t) \tag{1.3}$$

Verbindungen werden charakterisiert durch ihre Gewichte. Die Verbindung von Neuron i zu Neuron j hat das Gewicht w_{ij}. Alle Gewichte werden im Gewichtsvektor W zusammengefaßt.

1.1.3 Feed-Forward-Netze

Für meine Arbeit sind ausschließlich *Feed-Forward-Netze* von Belang. Das sind azyklische neuronale Netze, in denen die Informationsverarbeitung in einer vorbestimmten Richtung stattfindet. Dabei werden die Neuronen nach ihrer Topologie unterschieden (siehe Abb. 1.2). An die Schicht von sogenannten *Eingabeneuronen* wird ein Eingabemuster angelegt[1], das durch die *verdeckten Neuronen* zur Schicht von *Ausgabeneuronen* propagiert wird. Die verdeckten Neuronen können ebenfalls in Schichten unterteilt werden. Das Netz berechnet hier nichts anderes als eine mehrstellige Funktion F, wobei das Eingabemusters In auf einen Ausgabevektor O abgebildet wird, nämlich den Zustand der Ausgabeneuronen, nachdem das Eingabemuster angelegt und zur Ausgabeschicht propagiert wurde. Es gilt also

$$F : In \to O \quad \text{oder} \quad F : (in_1, \ldots, in_{n_i}) \mapsto (o_1, \ldots, o_{n_o}) \tag{1.4}$$

wobei n_i die Anzahl der Eingabeneuronen und somit die Größe des Eingabvektors und n_o die Anzahl der Ausgabeneuronen ist.

[1] Ihre Aktivierungen werden auf die Werte des Eingabemusters gesetzt. Die Eingabeneuronen tragen eigentlich nichts zur Informationsverarbeitung bei.

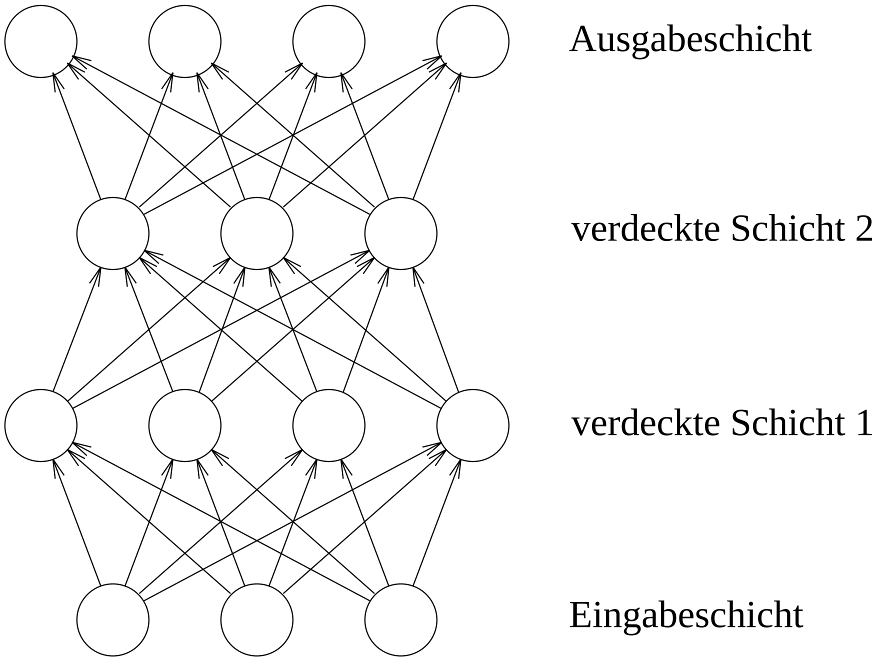

Abbildung 1.2: Topologie eines Feed-Forward-Netzwerks

1.2 Lernverfahren für neuronale Netze

Zu einem funktionierenden neuronalen Netz gehört auch eine Lernfunktion – ein Algorithmus, der bestimmt, wie das Netz verändert wird (meistens werden Gewichte und Schwellenwerte trainiert). Hierzu verwendet man eine Menge von *Trainingsmustern (Pattern)*, die jeweils aus einem Eingabevektor und dem dazugehörigen Ausgabevektor bestehen, als Beispiele für die von dem Netz zu realisierende Funktion. Die Eingabemuster werden nacheinander angelegt und durch das Netz propagiert, woraufhin die Aktivierung der Ausgabeneuronen mit dem Ausgabemuster verglichen wird. Das Netz wird jetzt im Sinne einer Angleichung der beiden Vektoren verändert, d. h. trainiert.

In den folgenden Abschnitten stelle ich die grundlegenden Lernverfahren für Feed-Forward-Netze kurz dar. Für eine genauere Betrachtung sei hier auf [Zell 94] verwiesen.

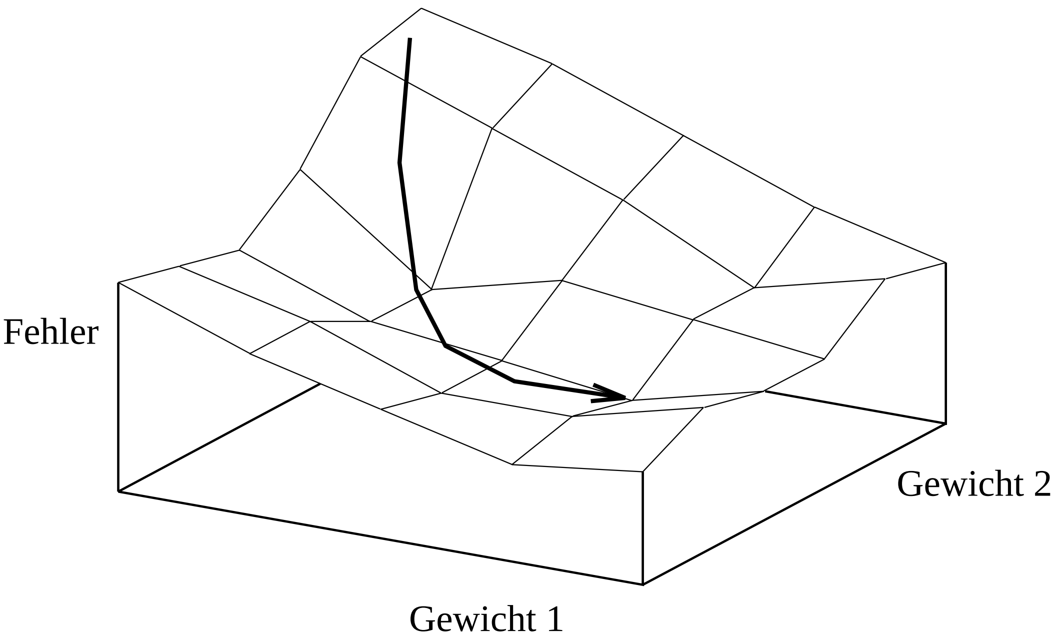

Abbildung 1.3: Fehlergebirge mit eingezeichnetem Weg des Gradientenabstiegsverfahrens

1.2.1 Backpropagation

Backpropagation ist das bekannteste Lernverfahren für Feed-Forward-Netze. Es geht auf Rumelhart und McClelland [Rumelhart et al. 86] zurück.

Backpropagation ist ein überwachtes Lernverfahren mit Gradientenabstieg, d. h. man versucht, in einem „Fehlergebirge“[2], wie es in Abb. 1.3 zu sehen ist, schrittweise durch kontrolliertes Modifizieren der Gewichte in ein Fehlerminimum („Tal“) zu kommen.

Die Fehlerfunktion E ist häufig als quadratische Summe der Differenzen zwischen tatsächlicher (o_{pj}) und erwarteter Ausgabe[3] (t_{pj}, teaching input) der Ausgabeneuronen definiert:

$$E = \frac{1}{2} \sum_{p=1}^{n_p} \sum_{j=1}^{n_o} (t_{pj} - o_{pj})^2, \tag{1.5}$$

[2] Die Veranschaulichung der Fehlerfunktion über den Gewichten sieht in einem dreidimensionalen Achsenkreuz, also Fehler über zwei Gewichten, wie eine Berglandschaft aus.

[3] Die Indizes bedeuten: p = Nummer des Musters und j = Nummer der Ausgabezelle

n_p ist dabei die Anzahl der Muster und n_o die Anzahl der Ausgabe-Neuronen im Netz. Der Faktor $\frac{1}{2}$ wurde eingeführt, um ihn nach dem Differenzieren kürzen zu können.

Der Gewichtsvektor W muß nun so modifiziert werden, daß der Fehler sich in Richtung des negativen Gradienten ändert. Das wird durch die Berechnung des neuen Gewichtsvektors aus

$$\Delta W = -\eta \cdot \nabla E(W) \tag{1.6}$$

erreicht. Hier führt man η als Lernfaktor ein, der die Größe der Schritte auf das lokale Minimum hin sinnvoll beschränken soll. Für ein einzelnes Gewicht w_{ij} (die Verbindung von der Zelle i zur Zelle j) gilt somit:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E(W)}{\partial w_{ij}} \tag{1.7}$$

Hier verwendet man die Kettenregel und erhält somit:

$$\frac{\partial E(W)}{\partial w_{ij}} = \frac{\partial E(W)}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}} = \delta_j \cdot \frac{\partial \sum_k o_k \cdot w_{kj}}{\partial w_{ij}} = \delta_j \cdot o_i \tag{1.8}$$

mit der rekursiv definierten Hilfsvariable δ_j, die für verdeckte Neuronen und Ausgabeneuronen auf unterschiedliche Weise berechnet werden muß. Zunächst wird allerdings für beide Fälle nochmals die Kettenregel angewandt

$$\delta_j = \frac{\partial E(W)}{\partial net_j} = \frac{\partial E(W)}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} = f'_{act}(net_j) \cdot \frac{\partial E(W)}{\partial o_j} \tag{1.9}$$

Für den Fall, daß j ein Ausgabeneuron ist, ergibt sich hierauf aus der Definition der Fehlerfunktion nach (1.5) – quasi als Rekursionsabbruch –

$$\frac{\partial E(W)}{\partial o_j} = -(t_j - o_j) \tag{1.10}$$

Ist j jedoch ein verdecktes Neuron, so erhält man die Rekursionsformel

$$\begin{aligned}\frac{\partial E(W)}{\partial o_j} &= \sum_k \frac{\partial E(W)}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_j} = \sum_k \delta_k \cdot \frac{\partial \sum_i o_i \cdot w_{ik}}{\partial o_j} \\ &= \sum_k \delta_k \cdot w_{jk} \end{aligned} \tag{1.11}$$

Zusammenfassend kann man also folgende Formeln aufstellen:

$$\Delta w_{ij} = \eta \cdot \delta_j \cdot o_i \tag{1.12}$$

mit

$$\delta_j = \begin{cases} f_j'(net_j) \cdot (o_j - t_j) & \text{falls } j \text{ Ausgabeneuron ist} \\ f_j'(net_j) \cdot \sum_k \delta_k w_{jk} & \text{falls } j \text{ verdecktes Neuron ist} \end{cases} \tag{1.13}$$

Diese Formel zur Gewichtsänderung kann auf zwei verschiedene Weisen angewandt werden. Zum einen können die Terme Δw_{ij} sofort nach ihrer Berechnung für jedes Muster zu den Gewichten addiert werden, zum anderen kann man sie zunächst aufsummieren und nach der ganzen Epoche verrechnen *(batch backpropagation)*.

Zu Backpropagation existieren viele Erweiterungen. Mit dem *Momentum Term* verhindert man, daß das Lernen in einem flachen Bereich des Fehlergebirges nur noch langsam vorangeht (dann ist nämlich der Gradient ziemlich klein, woraus eine kleine Schrittweite folgt). Man gibt dem Verfahren einen „Schwung" mit, indem man einen Bruchteil der letzten Gewichtsänderung nochmals addiert:

$$\Delta w_{ij}(t) = \eta \cdot \delta_j \cdot o_i \ + \ \alpha \cdot \Delta w_{ij}(t-1) \tag{1.14}$$

Als weiterer Effekt wird das Lernen in stark zerklüfteten Fehlerfächen abgebremst. Die Variable α liegt oft zwischen 0,6 und 0,9.

Neuronen, die schon im Sättigungsbereich ihrer Aktivierungsfunktion operieren, können ihre Gewichte nur schwer ändern. Der Grund dafür ist die Verwendung der Ableitung der Aktivierungsfunktion in der Formel (1.13) für die Gewichtsänderung, die im Sättigungsbereich einer sigmoiden Funktion immer sehr klein ist *(flat spots)*. Um diesen Effekt zu vermeiden, addiert die *flat spot elimination* eine Konstante – z. B. 0,1 – zur Ableitung der Aktivierungsfunktion.

1.2.2 Quickprop

Quickprop wurde von Scott Fahlman ([Fahlman 88]) als Verbesserung von Backpropagation vorgeschlagen. Er geht von der Annahme aus, daß die Fehlerfunktion lokal quadratisch ist und versucht, mit Hilfe der beiden letzten Berechnungen der Fehlerfunktion als Stützstellen den Scheitel der Parabel zu finden, der ja das (lokale) Minimum darstellen muß. (Man versucht, genauer gesagt, nach dem Newton-Verfahren die Nullstelle der Steigung zu finden.)

Die Steigungen seien für die Bestimmung des neuen Wertes für das Gewicht w_{ij} gegeben durch

$$S(t) = \frac{\partial E(t)}{\partial w_{ij}(t)} \tag{1.15}$$

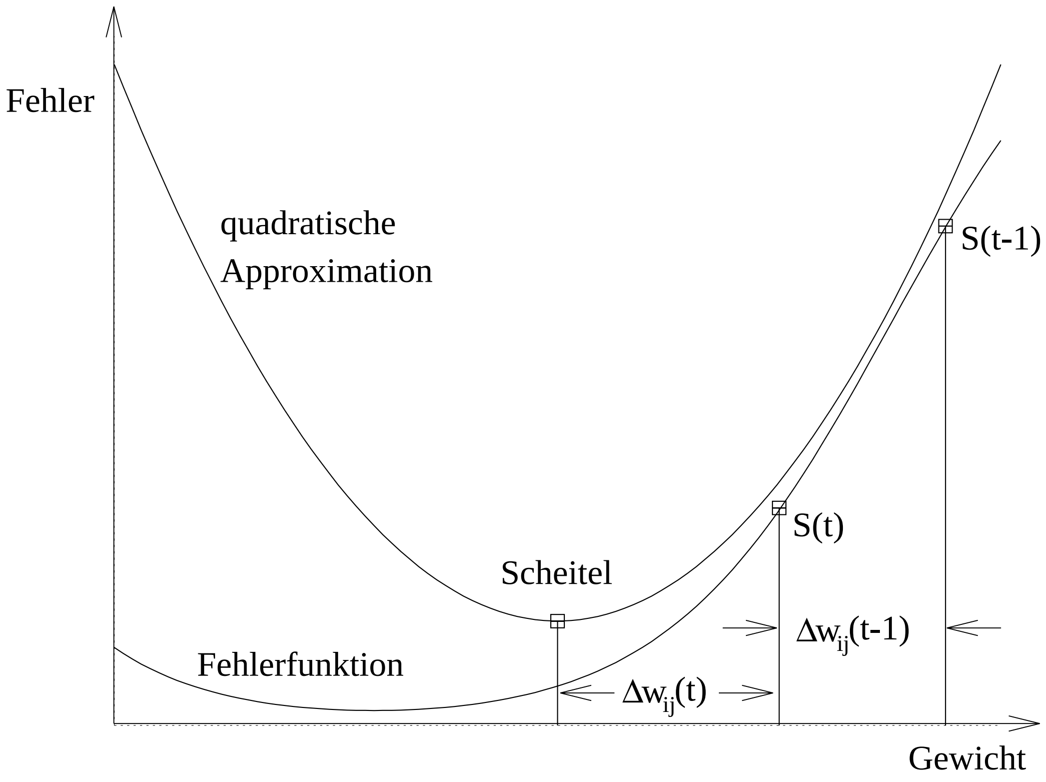

Abbildung 1.4: Quickprop approximiert die Fehlerfunktion mit Hilfe zweier Stützstellen durch eine Parabel

Damit wird die Veränderung für w_{ij} bestimmt durch

$$\Delta w_{ij} = \frac{S(t)}{S(t-1) - S(t)} \cdot \Delta w_{ij}(t-1) \tag{1.16}$$

Das Verfahren ist in Abb. 1.4 dargestellt (nach [Zell 94]). Quickprop ist für manche Probleme schneller als Backpropagation, konvergiert jedoch nicht so oft.

Quickprop wirft jedoch noch einige Probleme auf, die jeweils kleine Änderungen des reinen Verfahrens nach Formel (1.16) erfordern.

- Wird die Differenz zwischen den beiden Steigungen $S(t)$ und $S(t-1)$ sehr klein, so resultieren daraus sehr große Sprünge. Diese werden durch eine Schrittweitenbegrenzung aufgefangen.
- Zu Beginn existieren die Werte $S(t)$ und $S(t-1)$ noch nicht; ein Anfang wird daher grundsätzlich mit einem Gradientenverfahren wie Backpropagati-

on gemacht. Man kann auch grundsätzlich einen solchen Gradiententerm zur Gewichtsänderung hinzufügen, was allerdings bei Überspringen eines Minimums nicht mehr sinnvoll ist.

- Um die Gewichte nicht zu groß werden zu lassen, wird zusätzlich ein Weight-Decay-Term verwendet. Die Verwendung solcher Terme wird in Kapitel 3 genauer beschrieben.

1.2.3 Rprop

Rprop steht für „Resilient Propagation“ und wurde von M. Riedmiller und H. Braun ([Riedmiller et al. 92]) entwickelt. Es geht von der sogenannten „Manhattan-Regel“ aus, bei der die Ableitung des Fehlers nach einem Gewicht nur die Richtung, nicht aber die Schrittweite (wie z. B. bei Backpropagation) bestimmt. Diese wird nämlich durch die problemabhängige Konstante Δ_{ij} bestimmt. Es gilt also:

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t) & \text{falls } S(t-1) \cdot S(t) > 0 \ \wedge \ S(t) > 0 \\ +\Delta_{ij}(t) & \text{falls } S(t-1) \cdot S(t) > 0 \ \wedge \ S(t) < 0 \\ -\Delta w_{ij}(t-1) & \text{falls } S(t-1) \cdot S(t) < 0 \\ 0 & \text{sonst} \end{cases} \tag{1.17}$$

mit $S(t) = \frac{\partial E(t)}{\partial w_{ij}(t)}$. Um ein Hin- und Herspringen zu verweiden, muß im dritten Fall die Steigung $S(t)$ auf null gesetzt werden.

Rprop greift diese Idee auf, verfeinert die Methode allerdings dadurch, daß die Konstante Δ_{ij} im Laufe des Trainings geändert wird, und zwar für jedes Gewicht einzeln. Je nach Vorzeichen der letzten beiden Trainingsläufe wird Δ_{ij} erhöht oder verkleinert. Die Formel dafür ist:

$$\Delta_{ij}(t) = \begin{cases} \Delta_{ij}(t-1) \cdot \eta^{+} & \text{falls } S(t+1) \cdot S(t) > 0 \\ \Delta_{ij}(t-1) \cdot \eta^{-} & \text{falls } S(t+1) \cdot S(t) < 0 \\ \Delta_{ij}(t-1) & \text{sonst} \end{cases} \tag{1.18}$$

wobei $0 < \eta^{-} < 1 < \eta^{+}$ gilt.

Rprop ist in vielen Fällen stabiler als Quickprop und schneller als Backpropagation. Durch die Idee der eigenen Schrittweite pro Gewicht hat Rprop Ähnlichkeit mit dem Verfahren SuperSAB von Tollenaere ([Tollenaere 90]).

2. Grundlagen der Ausdünnungsverfahren

2.1 Motivation für Ausdünnungsverfahren

Diese Arbeit beschäftigt sich mit Ausdünnungsverfahren für neuronale Netze (speziell Feed-Forward-Netze). Feed-Forward-Netze (siehe Abschnitt 1.1.3) bestehen aus mehreren Ebenen, die der Einfachheit halber oft vollständig miteinander verbunden sind, d. h. jedes Neuron einer Schicht ist mit jedem Neuron der folgenden Schicht verknüpft. Daraus resultieren Netze mit sehr vielen Verbindungen.

Ein Problem der Anwendung neuronaler Netze ist es, die richtige Netzarchitektur festzulegen. Bei vollständig verbundenen Feed-Forward-Netzen reduziert sich das auf die Wahl von Anzahl und Größe der verdeckten Schichten. Als zusätzliche Möglichkeit kann man sogenannte *shortcut-Verbindungen* einbauen, die über mehrere Ebenen hinweg gehen.

Wählt man die Netzgröße falsch, so kann sich das in verschiedener Hinsicht negativ auswirken:

- Ist das Netz zu klein, so kann es die Datenmenge nicht repräsentieren, d. h. die gegebenen Beispiele nicht lernen.
- Ist das Netz zu groß, dann lernt das Netz zwar vermutlich die ihm vorgegebenen Beispiele, hat aber u. U. folgende Nachteile:
  - Es ist in der Anwendung langsamer als ein kleineres Netz. Die Komplexität von Feed-Forward-Algorithmen wie z. B. Backpropagation liegt in der Ordnung
    $$O\ (n_w \times n_p \times n_e).$$
    wobei n_w die Anzahl der Gewichte, n_p die Anzahl der Muster und n_e die Anzahl der Epochen bezeichnet.

– Es ist aufwendiger zu implementieren. Wird das neuronale Netz in Hardware implementiert, so entstehen hier i. a. größere Kosten für ein Netz mit mehr Neuronen und Verbindungen. Für eine Software-Simulation ergibt sich ein erhöhter Speicherplatzbedarf.

– Die Generalisierungsleistung ist mangelhaft (mehr dazu in 2.2).

– Seine Arbeitsweise ist schwieriger zu verstehen.[1]

Zum Finden der richtigen Architektur gibt es im wesentlichen drei Ansätze:

Trial and error: Man probiert systematisch verschiedene Architekturen aus (z. B. ausgehend von einer großen Architektur immer kleinere Netze), bis man ein Netz findet, das die Daten gerade noch repräsentieren kann.

Der konstruktive Ansatz: Man fügt in ein anfangs nur aus Ein- und Ausgabeneuronen bestehendes Netz sukzessive verdeckte Neuronen ein, bis man glaubt, eine zufriedenstellende Architektur gefunden zu haben. Das bekannteste Verfahren dieser Art ist Cascade Correlation von Scott Fahlman ([Fahlman 91]).

Die Ausdünnungsverfahren: Man wählt eine Architektur, die offenbar zu groß ist, und löscht aus dieser dann nach und nach die Verbindungen und/oder Neuronen, die unwichtig erscheinen, bis das Netz offensichtlich eine vernünftige Größe hat.

2.2 Die Generalisierung neuronaler Netze

Nachdem die Generalisierung ein wichtiger und darüberhinaus auch komplizierter Bewertungsfaktor für neuronale Netze ist, will ich diesen Begriff hier etwas näher erklären.

Üblicherweise trainiert man das Netz, indem man ein Lernverfahren mehrmals auf eine *Trainingsmenge* von Mustern anwendet. Unter der Generalisierungsleistung versteht man nun die Fähigkeit eines neuronalen Netzes, zu Daten aus einer *Testmenge* die richtigen Ausgaben zu erzeugen, wenn diese dem Netz nach dem Training zum ersten Mal präsentiert werden.

Die Testdaten sind dabei entweder vollkommen neue Muster – die aber natürlich der bisherigen Problemstellung entsprechen müssen – oder sie werden aus den Trainingsdaten durch „Verrauschen“ gewonnen.

[1] Dieser Nachteil kommt selten zum Tragen, da es in den meisten Fällen ohnehin unmöglich ist, ein neuronales Netz zu analysieren.

2.2.1 Probleme der Generalisierung

Es besteht ein Zusammenhang zwischen der Anzahl der freien Parameter im System (dies sind i. a. Gewichte und Schwellenwerte) und der Anzahl der Trainingsmuster, die man für ein gutes Generalisierungsverhalten braucht.

Rumelhart meinte dazu (zitiert nach [Hanson et al. 89]):

> „... the simplest most robust network which accounts for a data set will, on average, lead to the best generalization to the population from which the training set has been drawn.“

Beim Training eines für die Anzahl der Trainingsmuster zu großen Netzes besteht die Gefahr, daß das Netz übertrainiert wird *(overtraining, overfitting)* und sich ganze Muster merkt, anstatt einige spezielle Merkmale *(features, samples)* zu extrahieren, die das Muster charakterisieren. Dadurch kann das Netz dann verrauschte oder unvollständige Muster schlechter erkennen.

Man kann sich das auch ausgehend von einem Interpolationsproblem vorstellen: Wenn man eine gegebene Menge von Wertepaaren einer Funktion durch ein Polynom interpoliert, so kann es in ungünstigen Fällen dazu kommen, daß das Polynom zwischen den Stützstellen stark schwingt. Werte zwischen den Stützstellen kann man sich in der Analogie als Testdaten vorstellen, die dem neuronalen Netz beim Training nicht präsentiert wurden. Approximiert man jedoch die Funktion stattdessen durch ein Polynom kleineren Grades (bzw. ein kleineres Netz) mit weniger freien Parametern, so wird die Kurve i. a. glatter, trifft aber nicht mehr alle Stützstellen.

2.2.2 Analyse und Lösungsansätze

Nach Reed ([Reed 93]) genügt es jedoch nicht, einfach ein paar Netzarchitekturen auszuprobieren und daraufhin diejenige zu verwenden, welche die Daten gerade noch repräsentieren kann. Zum einen ist dies eine zeitaufwendige Methode, da einige Netze trainiert und evaluiert werden müssen, bis das offensichtlich optimale gefunden ist; zum anderen besteht die Gefahr, daß das „kleinstmögliche“ Netz zu sensibel für Initialisierung und Lernparameter ist und sich mit größerer Wahrscheinlichkeit in lokalen Minima verfängt.

Das Problem des Overtraining tritt aber nur auf, wenn das Netz zu lange trainiert wird (da dann die Vorstellung des Netzes von den trainierten Daten immer genauer wird). Dieser Effekt wird illustriert durch Abb. 2.1. Hier erkennt man sehr gut, daß bei fortschreitendem Training zwar der Netzfehler der Trainingsbeispiele stetig

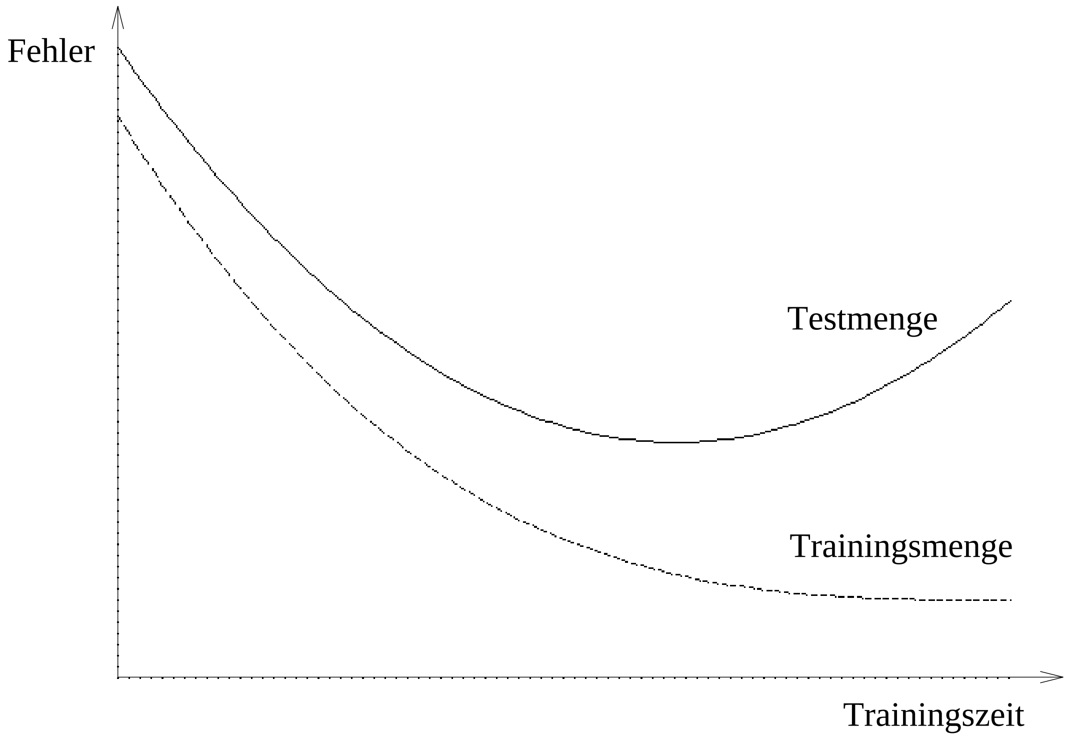

Abbildung 2.1: Typischer Verlauf der Fehlerkurven von Trainings- und Validierungsmenge

sinkt, der Fehler für eine Testmenge ihm jedoch nur ein Stück weit folgt, um dann nach einer bestimmten Anzahl von Zyklen wieder zuzunehmen.

Ein Ansatz, Overfitting zu vermeiden, ist daher die Aufteilung der Daten in eine Trainings- und eine Validierungsmenge (nicht zu verwechseln mit der Testmenge, die dem neuronalen Netz als dritte Menge erst nach Ende des Trainings präsentiert wird). Das Training des Netzes erfolgt wie gehabt mit der Trainingsmenge, mit der Validierungsmenge wird nach jedem Trainingszyklus die Generalisierungsleistung gemessen und dadurch der Zeitpunkt bestimmt, an dem die Generalisierungsleistung wieder abnimmt. Sobald dieser Effekt eintritt, wird das Training abgebrochen.

Kann man keine Validierungsmenge bilden (z. B. weil die Menge der Beispieldaten zu klein ist, um sie auch noch aufzuspalten), so kann man behelfsmäßig das Netz bis zum „Knick" trainieren, das ist die Stelle, an der der Verlauf der Fehlerkurve nach Abb. 2.1 merklich flacher wird. Dort ist dann die Generalisierungsleistung am größten. Dies stellt jedoch eine ungenaue Vereinfachung dar.

Eine weitere Möglichkeit wäre, die Trainingsdaten mit einem Rauschen zu überlagern, wie es von Sietsma und Dow ([Sietsma et al. 91]) beschrieben wird.

Die Ausdünnungsverfahren, die ich in meiner Diplomarbeit untersucht habe, verfolgen jedoch einen ganz anderen Ansatz. Sie reduzieren die Anzahl der freien Parameter, indem sie die Netzwerkarchitektur durch Löschen von Verbindungen oder gar Neuronen verändern.

2.3 Einteilung der Ausdünnungsverfahren

Man kann die Ausdünnungsverfahren ganz grob durch zwei Kriterien klassifizieren, nämlich durch das Objekt der Ausdünnung und die verwendete Methode. Dieses Schema soll vorgestellt und anschließend auf die Verfahren angewandt werden, die ich im Rahmen meiner Diplomarbeit untersucht habe.

2.3.1 *Was* wird eliminiert?

Verfahren unterscheiden sich zum einen dadurch, was sie zu eliminieren versuchen: Neuronen oder Verbindungen. Man spricht hier von

- *weight pruning* (Löschen von Verbindungen) und
- *node pruning* (Löschen von Neuronen), manchmal unterteilt in
  - *input pruning* (Löschen von Eingabeneuronen) und
  - *hidden unit pruning* (Löschen von verdeckten Neuronen)

wobei node-pruning-Verfahren üblicherweise sowohl input pruning als auch hidden unit pruning durchführen. Input pruning ist speziell dann interessant, wenn man die Zahl der Eingabewerte minimieren möchte *(Merkmalsextraktion)*, z. B. wenn diese teuer zu bestimmen (zu messen) sind oder wenn man eine konventionelle Verarbeitung bevorzugt und das neuronale Netz nur als Hilfsmittel zur Bestimmung der relevanten Eingabewerte dient.

2.3.2 *Wie* wird eliminiert?

Zum anderen kann man die Verfahren nach ihrer Vorgehensweise einteilen. Es gibt hier zwei große Klassen, in denen die Mehrzahl der bekannten Verfahren liegen:

- *Weight-Decay-Verfahren* (Verfahren mit *Kosten-* oder *penalty-Term*) und
- *Sensitivitäts-Verfahren (direktes Ausdünnen).*

Weight-Decay-Verfahren

Weight-Decay-Verfahren geben den freien Parametern im Netz (vor allem den Gewichten) eine Tendenz, sich zu verkleinern. Dies geschieht durch Einführung eines zusätzlichen Kostenterms in die Fehlerfunktion, der monoton mit der Größe der Parameter steigt. Konkret bedeutet dies, daß die Gewichte, die durch das Training nicht verstärkt werden und somit für das zu bearbeitende Problem nicht so wichtig sind, während des Trainings stetig gegen null verändert werden. Parameter, die kleiner als ein Minimalwert werden, können ganz entfernt (d. h. dauerhaft auf null gesetzt) werden.

Sensitivitäts-Verfahren

Sensitivitäts-Verfahren dagegen sind iterative Verfahren, in denen so lange Elemente aus dem Netz gelöscht werden, bis ein Abbruchkriterium eintritt (z. B. der Anstieg des Netzfehlers über ein bestimmtes Maß). Bei den meisten Verfahren muß das Netz nach jedem Löschen nachtrainiert werden. Diese Verfahren können durch den in Abb. 2.2 angegebenen Rahmenalgorithmus beschrieben werden.

2.3.3 Übersicht über die verwendeten Verfahren

In dieser Arbeit werden folgende Verfahren vorgestellt:

Backpropagation mit Weight Decay: Dies ist das klassische Kostenterm-Verfahren. Es wird vorgestellt in Kapitel 3.

Magnitude Based Pruning: Das Löschen der betragsmäßig kleinsten Gewichte ist das „naive“ Sensitivitäts-Verfahren (siehe Kapitel 4).

Optimal Brain Damage (OBD): Dieses Verfahren verwendet eine Approximation der Energiefunktion zum direkten Löschen von Verbindungen (siehe Kapitel 5).

Optimal Brain Surgeon (OBS): OBS stellt eine Weiterentwicklung von OBD dar, die Energiefunktion wird hier genauer bestimmt (siehe Kapitel 6).

Skeletonization (Skelettierung): Dies ist das einzige Verfahren zum direkten Löschen von Neuronen. Es wird in Kapitel 7 näher betrachtet.

Weitere Verfahren werden in Kapitel 8 kurz angerissen.

1. Wähle eine großzügige Anfangsarchitektur.
2. Trainiere das Netz mit einem herkömmlichen Lernverfahren (siehe 1.2), bis der Netzfehler minimal ist (bzw. bis der Fehler auf der Validierungsmenge wieder ansteigt).
3. Bestimme zu allen Elementen (Neuronen oder Verbindungen) deren Relevanz *(Saliency)* für das Netzverhalten.
4. Lösche die unwichtigsten Neuronen bzw. Verbindungen (diejenigen mit der kleinsten Saliency).
5. Trainiere das Netz erneut in ein Minimum oder bis zum Ansteigen des Testfehlers *(retraining)*.
6. Ist der Gesamtfehler nicht zu stark angewachsen (prozentual oder über eine gewisse feste Grenze), so wiederhole das Verfahren ab Punkt 3.
7. (fakultativ) Da der Fehler jetzt größer ist als eigentlich erlaubt, stelle den Zustand wieder her, der vor dem letzten Löschen bestanden hat.

Abbildung 2.2: Rahmenalgorithmus für Sensitivitätsverfahren

2.4 Implementierung der Verfahren

2.4.1 Der Stuttgarter Neuronale Netze Simulator (SNNS)

Der Stuttgarter Neuronale Netze Simulator (SNNS) entstand 1989 an der Universität Stuttgart und wird seitdem dort weiterentwickelt (siehe [Zell et al. 94]). Er bietet eine effiziente und flexible Entwicklungsumgebung für neuronale Netze und wird für Forschung und Anwendung eingesetzt.

Er besteht hauptsächlich aus zwei Komponenten, nämlich dem Simulatorkern und der graphischen Benutzeroberfläche XGUI[2]. Der Kern hält interne Datenstrukturen zur Beschreibung eines neuronalen Netzes vor sowie diverse Lernfunktionen und Analysemöglichkeiten, die alle von der Benutzeroberfläche aus kontrolliert werden können.

[2] X Graphical User Interface

Desweiteren existiert ein Programm SNNSBAT für Batchläufe des SNNS, das ebenfalls direkt auf die Kern-Routinen zugreift.

Der Simulator SNNS ist komplett in ANSI-C geschrieben und läuft auf Unix-Plattformen unter XWindows.

2.4.2 Funktionalität der Erweiterung für den SNNS

Im Rahmen dieser Diplomarbeit implementierte ich fünf Ausdünnungsverfahren mit einer dazugehörigen Basisfunktionalität, mit der man leicht weitere Verfahren in den SNNS einbinden kann.

Man kann alle Verfahren sowohl in der Oberfläche als auch als Batchläufe starten. Die Implementierungen unterscheiden sich nach Kostenterm- und Sensitivitäts-Verfahren. Als einziges Kostenterm-Verfahren wurde Backpropagation mit Weight Decay implementiert. Die Verwendung dieses Verfahrens ist in Abschnitt 3.2 beschrieben. Hier beschränke ich mich auf die vier Sensitivitätsverfahren, die alle nach dem Algorithmus in Abb. 2.2 ablaufen.

Die Sensitivitäts-Verfahren sind unter einer Lernfunktion zusammengefaßt, die „PruningFeedForward“ heißt. In der Online-Version wähle man im Hauptmenü die Option „PRUNING“ aus, woraufhin das in Abb. 2.3 gezeigte Pruning-Fenster auf dem Bildschirm erscheint. Hier kann man das Ausdünnungsverfahren und seine Parameter auswählen. In der Batchversion verwende man entsprechende Einträge in der Konfigurationsdatei (siehe Tabelle 2.1). Die angezeigten Parameter sind die default-Werte, die das System verwendet, wenn keine Änderung vorgenommen wird.

Der erste Block des Fensters steuert das Ausdünnen selbst, der zweite hält Parameter für das (logisch untergeordnete) Training vor, in zwei weitere Blöcke können spezielle Parameter für einzelne Ausdünnungsverfahren eingegeben werden. Der „DONE“-Button schließt das Fenster und übernimmt die Parameter in die entsprechenden Variablen im Simulatorkern.

Im Block „GENERAL PARAMETERS FOR PRUNING“ wird zunächst einmal die aktuelle Ausdünnungsfunktion angezeigt. Durch Mausklick auf den SELECT-Button erscheint eine Liste der vorhandenen Verfahren (siehe Abb. 2.4). Durch Auswahl der gewünschten Funktion und anschließende Betätigung des „DONE“-Buttons der Liste wird die aktuelle Funktion geändert und im Pruning-Fenster angezeigt.

Jetzt muß dem Simulator noch das Abbruchkriterium mitgeteilt werden. Es gibt hier zwei Faktoren: Zum einen kann im Feld „MAXIMUM ERROR INCREASE IN %:“ ein Prozentsatz eingegeben werden. Dieser Prozentsatz wird zum Fehler nach dem ersten Training addiert, dies ist dann die oberste Schranke des Netzfehlers nach dem Nachtrainieren. Da es aber möglich ist, daß beim ersten Training schon ein ziemlich niedriger Netzfehler erreicht wird, kann unter „ACCEPTED ERROR:“ angegeben werden, welcher absolute Netzfehler erreicht werden muß, damit das Netz tatsächlich stoppt. Es müssen also beide Schranken überschritten werden, damit der Simulator abbricht. Ob er jedoch zuvor noch den Zustand vor dem letzten Löschen wiederherstellt – wie dies im Algorithmus nach Abb. 2.2 als letzter, fakultativer Schritt vorgesehen ist –, hängt von der Stellung des Radio-Buttons „RECREATE LAST PRUNED ELEMENT ?“ ab.

Der zweite Block des Pruning-Fensters enthält „GENERAL PARAMETERS FOR TRAINING“, darunter einen Auswahlknopf für die untergeordneten Lernfunktionen, die analog zu den Ausdünnungsfunktionen angezeigt und eingegeben werden. Desweiteren kann hier die Anzahl der Lernzyklen eingegeben werden, und zwar separat für das erste Training und die folgenden Nachtrainings. Ins Feld „MINIMUM ERROR TO STOP“ kann der absolute Netzfehler eingegeben werden, dessen Unterschreitung das Training sofort beendet, ohne die restlichen Zyklen noch abzuarbeiten. Dadurch ist es möglich, ein Übertrainieren zu verhindern.

Der Parameter für OBS bietet die Möglichkeit, den Initialisierungsparameter für die Diagonale der inversen Hessematrix direkt einzugeben (näheres zu dessen Bedeutung siehe Kapitel 6). Für Skeletonization kann man angeben, ob nur Eingabeneuronen gelöscht oder alle Neuronen gleich betrachtet werden sollen.

Die Lernparameter der untergeordneten Lernfunktion müssen nach wie vor im Remote-Panel eingegeben werden. Von hier aus wird auch die Ausdünnung angestoßen, und zwar wie bei den Lernverfahren durch Betätigung des „ALL“- oder „SINGLE“-Buttons. Man beachte, daß die Angabe einer Zyklenzahl im Remote-Panel keine Wirkung hat.

Die entsprechenden Befehle für die Batchversion des SNNS sind in Tabelle 2.1 zusammengefaßt. Es gelten die gleichen default-Werte wie in der Oberfläche. Ausdünnungs- und Lernfunktion müssen in der Netzdatei richtig gesetzt sein.

Der Verlauf des Ausdünnens wird sowohl von der Online- als auch der Batch-Version protokolliert. Die ausgewählten Parameter werden ausgegeben. Bei jedem Training wird in regelmäßigen Abständen der Netzfehler protokolliert, wie bei SNNS üblich. Zusätzlich wird über das Löschen von Neuronen oder Verbindungen informiert.

Befehl	Typ	Bedeutung
`MaxLearnCycles`	<int>	Maximale Zyklenzahl für das Training
`PruningMaxRetrainCycles`	<int>	Maximale Zyklenzahl für das Nachtrainieren
`MaxErrorToStop`	<float>	Fehler, bei dem das Training vorzeitig beendet wird
`PruningMaxErrorIncrease`	<float>	Prozentsatz, der auf den ersten Netzfehler aufgeschlagen wird. Den entstehenden Wert darf der Netzfehler nach dem Nachtrainieren nicht überschreiten, es sei denn, er liegt noch unter dem akzeptierten Fehler.
`PruningAcceptedError`	<float>	Akzeptierter Fehler, unterhalb dessen das Ausdünnen nicht abgebrochen wird
`PruningRecreate`	[YES | NO]	Schalter zur Wiederherstellung des Zustandes nach dem letzten Löschen
`PruningOBSInitParam`	<float>	Initialisierungswert für OBS
`PruningSkelInputPruning`	[YES | NO]	Schalter für reines Input Pruning bei Skeletonization

Tabelle 2.1: Tabelle der Befehle für Batchläufe

2.4.3 Verteilung der Aufgaben auf Oberfläche und Kern

In der Oberfläche wird die Simulation – wie bereits erwähnt – nur kontrolliert, sie ist für die Konstruktion von Netzen, die Eingabe von Parametern und die Präsentation der Ergebnisse zuständig. Die eigentliche Rechenarbeit auf internen Datenstrukturen wird im Kern geleistet.

Hier finden sich auch die Funktionen für die Ausdünnungsverfahren. Backpropagation mit Weight Decay wird als eigene Lernfunktion betrachtet, die anderen Verfahren sind als eine Lernfunktion zusammengefaßt, aber einzeln als Ausdünnungsfunktionen in der Funktionentabelle des SNNS eingetragen. Aufgerufen werden sie über eine Schnittstellenfunktion, die nach einigen Konsistenzprüfungen die richtige Ausdünnungsfunktion aufruft und anschließend die für unwichtig gehaltenen Verbindungen oder Neuronen löscht und dabei auch „tote" Neuronen aus dem Netz nimmt – das sind solche, die nicht mehr auf einem Pfad von Eingabe- zu Ausgabeneuronen liegen und somit wirkungslos geworden sind.

In der Oberfläche wird zunächst einmal das oben beschriebene Fenster zur Parametereingabe verwaltet, außerdem aber auch der Ablauf der Ausdünnung kontrolliert und protokolliert. Von hier aus werden Lern- und Ausdünnungsfunktionen aufgerufen, die Anzahl der Lernzyklen kontrolliert und über die Fortsetzung des Ausdünnens entschieden sowie am Ende des Ausdünnens evtl. der Zustand vor dem letzten Löschen wiederhergestellt.

2.4.4 Einzelheiten der Implementierung

In diesem Kapitel möchte ich die technischen Einzelheiten der Implementierung der Sensitivitätsverfahren darstellen, um eine Erweiterung oder Änderung meiner Funktionen zu erleichtern.

Das Erscheinungsbild und die Funktionalität des Fensters nach Abb. 2.3 wird durch die Funktion

```
void pr_createWindow (void)
```

im Modul pr_main.c (Oberfläche) beschrieben. Zur Darstellung der Listen wurde die Funktion `ui_info_displayList` aus ui_infoP.c (Oberfläche) wiederverwendet, die eingegebenen Daten werden bei Betätigung des DONE-Buttons durch die Funktion

```
void pr_doneProc (void)
```

in die entsprechenden Variablen im Modul prun_f.h (Kern) abgespeichert. Die default-Werte für die Variablen sind in prun_f.ph zu sehen.

Gestartet werden die Ausdünnungsverfahren durch Aufruf der *X Work Procedure* (siehe [McCormack et al. 91])

```
Boolean ui_rem_performPrunInit (int learnType)
```

aus dem Modul ui_remoteP.c (Oberfläche). Der Parameter `learnType` gibt an, ob alle oder nur ein Muster betrachtet werden sollen. Diese Funktion führt das erste Training durch, berechnet den maximalen Fehler und ruft dann die Work Procedure

```
Boolean ui_rem_performPrun (int learnType)
```

auf, die das Netz speichert, das Ausdünnen und ein Nachtraining veranlaßt und anschließend den Netzfehler überprüft. Ist dieser zu hoch geworden, so wird das zuletzt gespeicherte Netz wieder geladen und das Ausdünnen abgebrochen. Die entsprechende Funktion für Batchläufe steht im Modul snnsbat.c (Tools) und heißt

```
void pruneNet ()
```

Zum Trainieren des Netzes wird die Funktion

```
float ui_rem_performFFLearn (int cycles, int learnType)
```

in der Oberfläche bzw.

```
float trainFFNet (int no_of_cycles_ff)
```

in der Batchversion verwendet, die die Anzahl der Lernzyklen erhält und den Netzfehler bei Ende des Trainings zurückgibt. Das Training wird schon vor dem Erreichen der maximalen Zyklenzahl abgebrochen, wenn der Netzfehler den minimalen Fehler unterschreitet.

Das Ausdünnen selbst wird von der Funktion

```
krui_err pr_callPrunFunc (int pattern)
```

im Modul prun_f.c (Kern) kontrolliert. Sie ruft die aktuelle Ausdünnungsfunktion auf, welcher der Lern-Typ übergeben wird und die nach Berechnung des überflüssigen Elements dieses in den Variablen `pr_candidateTargetUnit` und `pr_candidateLink` bzw. `pr_candidateUnit` ablegt.

Die Funktion `pr_callPrunFunc` löscht anschließend diese Verbindung bzw. alle Verbindungen dieses Neurons. Zum Schluß löscht die Funktion

```
void pr_checkDeadUnits (int delete_units)
```

alle toten Neuronen im Netz. Ein- und Ausgabeneuronen werden auf jeden Fall zu speziellen Neuronen *(special units)*, verdeckte Neuronen nur, wenn der Aufrufparameter `delete_units` auf FALSE steht, sonst werden sie gelöscht.

Die einzelnen Ausdünnungsfunktionen sind in den jeweiligen Kapiteln im Abschnitt „Implementierung“ beschrieben. Sie sind in die Funktionentabelle im Modul func_tbl.c (Kern) eingetragen. Außerdem mußten dort die zweiten Ableitungen der Aktivierungsfunktionen eingetragen werden. Die Feed-Forward-Lernfunktionen wurden mit einem entsprechenden Flag versehen.

Weitere wichtige allgemeine Funktionen sind (alle im Kern):

- `pr_calcMeanDeviation`
  Diese Funktion bestimmt den Netzfehler und trägt in jedes Ausgabeneuron seine mittlere Abweichung ein.
- `krui_checkPruning`
  Diese Funktion gibt genau dann TRUE zurück, wenn die Lernfunktion auf „PruningFeedForward“ gesetzt ist.
- `krui_learnAllPatternsFF` und `krui_learnSinglePatternFF`
  analog zu `krui_learnAllPatterns` und `krui_learnSinglePattern`, diese Funktionen verwenden jedoch die untergeordneten (Feed-Forward-)Lernfunktionen.
- `krui_getPrunFunc`, `krui_getFFLearnFunc`, `krui_setPrunFunc` und `krui_setFFLearnFunc`
  analog zu `krui_getLearnFunc` und `krui_setLearnFunc` für die Ausdünnungsfunktion und die untergeordnete Lernfunktion.

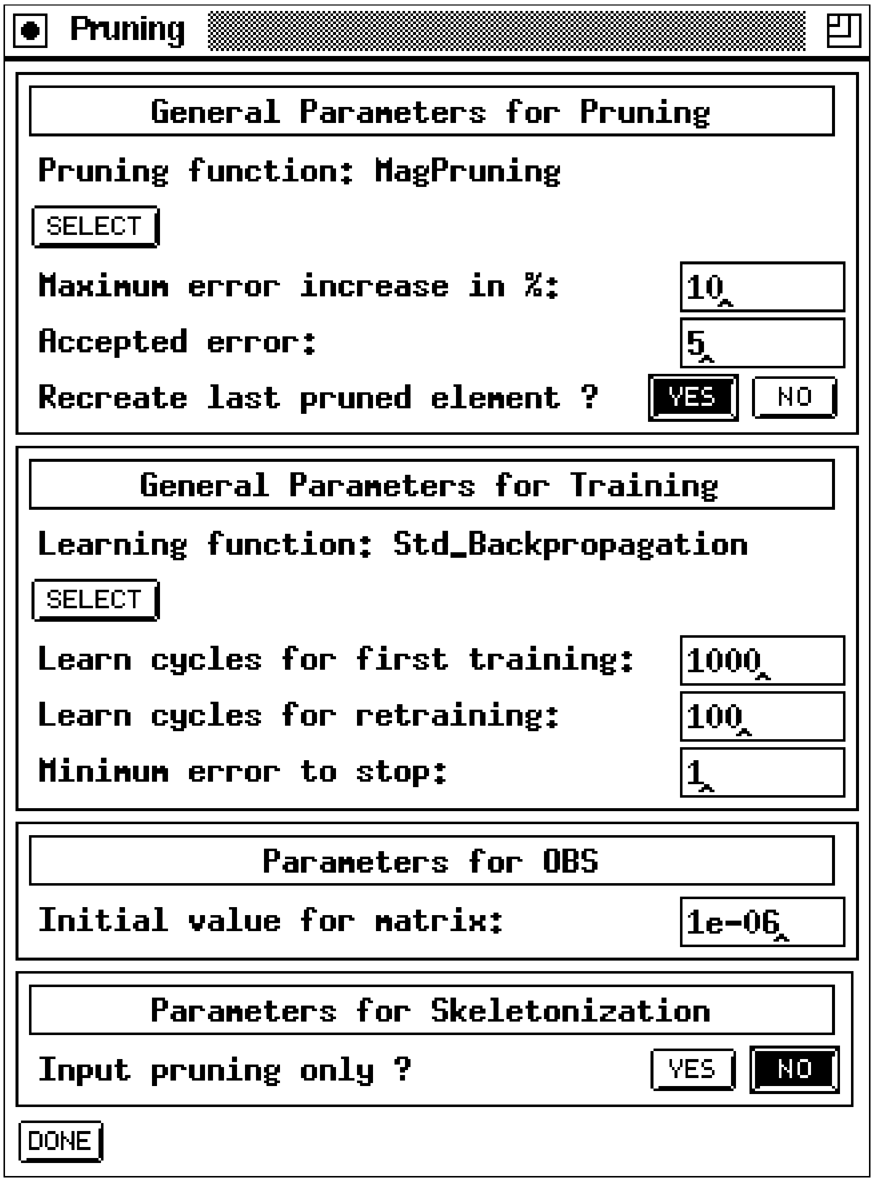

Abbildung 2.3: Parameter-Auswahlmenü für die Ausdünnungsverfahren

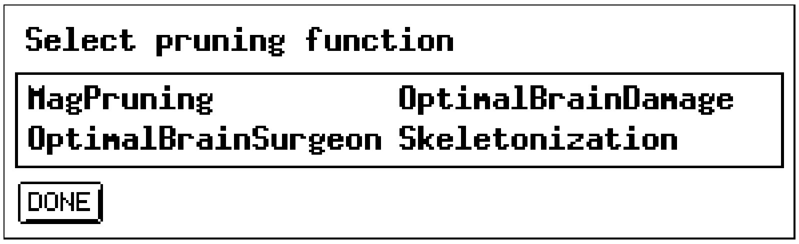

Abbildung 2.4: Listen zur Auswahl der Ausdünnungsfunktion

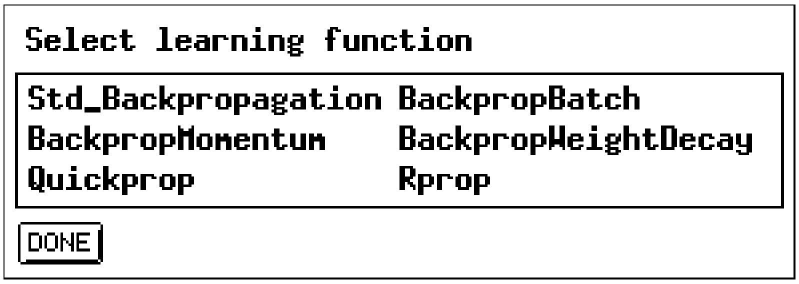

Abbildung 2.5: Listen zur Auswahl der Lernfunktion

3. Backpropagation mit Weight Decay

3.1 Beschreibung des Verfahrens

Backpropagation mit Weight Decay wurde zuerst von Paul Werbos in [Werbos 88] beschrieben. Ziel dieser Modifikation von Backpropagation ist es, den Gewichten im Netz eine Tendenz zur stetigen Verkleinerung zu geben. Dies erreicht man durch eine modifizierte Fehlerfunktion:

$$E = E_{Backpropagation} + C \tag{3.1}$$

Der Kostenterm C kann dabei verschiedene Ausprägungen haben, steigt aber immer monoton mit dem Gewicht. Ein einfacher Kostenterm C ist:

$$C = \frac{d}{2} \sum_{i,j=1}^{n_n} w_{ij}^2 \tag{3.2}$$

wobei d der Weight-Decay-Faktor ist, der oft zwischen 0,005 und 0,03 liegt; der Faktor $\frac{1}{2}$ ist wieder zum Kürzen bei der Ableitung eingefügt worden.

Bei zu großen Werten für den Weight-Decay-Faktor werden die Gewichte zu klein gehalten, was dazu führen kann, daß das Verfahren nicht konvergiert.

Durch partielle Ableitung nach w_{ij} entsteht eine neue Formel für die Gewichtsänderung:

$$\Delta w_{ij}(t+1) = \eta\, \delta_j\, o_i - d\, w_{ij}(t) \tag{3.3}$$

Werden zusätzlich Gewichte, die unter einem bestimmten Minimalgewicht w_{min} liegen, gelöscht, so können wir hier von einem Weight-Pruning-Verfahren sprechen.

3.2 Implementierung

Um dieses Verfahren im SNNS zu verwenden, wähle man als Lernfunktion im SNNS „BackpropWeightDecay“ aus und verwende insgesamt vier Lernparameter:

1. η als obligatorischer Lernparameter für Backpropagation.
2. d als Weight Decay Parameter, der die Stärke der Ausdünnung angibt. Will man kein Weight Decay durchführen, sondern nur kleine Gewichte mit Hilfe des dritten Paramters löschen, so setze man hier null ein.
3. w_{min} ist das Minimalgewicht, unterhalb dessen die Gewichte gelöscht werden. Will man diesen Effekt vermeiden, so kann man den Parameter auf null setzen.
4. d_{max} ist schließlich der maximale Abstand zwichen output o_j und teaching input t_j eines Ausgabeneurons, bei dem kein Fehler zurückpropagiert wird. Dieser Parameter ist aus dem herkömmlichen Backpropagation des SNNS entnommen.

Zu diesem Verfahren gehört die Funktion

```
krui_err LEARN_backpropWeightDecay
(int start_pattern, int end_pattern,
float *parameterInArray, int NoOfInParams,
float **parameterOutArray, int *NoOfOutParams)
```

im Modul learn_f.c (Kern). Ihr werden im Parameter `parameterInArray` die vier Parameter in der obengenannten Reihenfolge übergeben. Die Schnittstelle und das Verhalten der Funktion ist sonst Backpropagation ähnlich.

3.3 Bewertung

Weight-Decay-Verfahren sind nach Krogh und Hertz ([Krogh et al. 92]) gut geeignet, die Generalisierung zu verbessern, da sie einerseits die irrelevanten Komponenten eines Gewichtsvektors löschen, andererseits das statische Rauschen (den „unverläßlichen Lehrer“) in den Mustern ausgleichen können.

Auch wenn keine Verbindung gelöscht wird (d. h. der Parameter w_{min} ist null), hat Backpropagation mit Weight Decay nach Reed ([Reed 93]) einen positiven Effekt auf die Generalisierungsleistung.

Außer den üblichen Vorteilen der Ausdünnungsverfahren gibt es noch folgende Erwägungen, die für Weight Decay sprechen:

- Zu große Gewichte sind neurobiologisch unplausibel.
- Für große Gewichte wird die Fehlerfläche steiler und unregelmäßiger, sodaß die Suche nach einem Minimum schwieriger wird.
- Die Wahl der Initialisierung der Gewichte wird weniger wichtig.

Folgender Kostenterm wurde von Rumelhart vorgeschlagen:

$$C = \frac{d}{2} \sum_{i,j=1}^{n_n} \frac{w_{ij}^2}{(1 + w_{ij}^2)} \tag{3.4}$$

Er hat den Vorteil, daß er die kleinen Gewichte stärker gegen null konvergieren läßt als die großen, ist aber dafür nicht so schnell zu berechnen. Aus seiner Ableitung folgt folgende Gewichtsänderung:

$$\Delta w_{ij}(t+1) = \eta\, \delta_j\, o_i - d \frac{w_{ij}(t)}{(1 + w_{ij}^2)^2} \tag{3.5}$$

Hanson und Pratt ([Hanson et al. 89]) vergleichen Backpropagation mit Weight Decay mit den Methoden der Regressionsrechnung, wo mit dem Fehlerterm ein Maß für die Einfachheit minimiert wird. Für eine genauere Darstellung wird auf [Rawlings 88] verwiesen.

4. Löschen der betragsmäßig kleinsten Gewichte

4.1 Algorithmus

Das Ausdünnen der betragsmäßig kleinsten Gewichte *(Magnitude Based Pruning)* ist der einfachste Ansatz für ein Sensitivitätsverfahren. Nach jedem Training wird das kleinste Gewicht aus dem Netz gelöscht, in der Annahme, es trage zum Netzverhalten und damit auch zum Gesamtfehler am wenigsten bei. Die Relevanz bzw. Saliency der Verbindung ist hier also gleich dem Betrag seines Gewichts.

Der Algorithmus für dieses Verfahren ist in Abb. 4.1 zusammengefaßt.

---

1. Trainiere das Netz mit einem herkömmlichen Lernverfahren (z. B. Backpropagation) bis zum Punkt der maximalen Generalisierungsleistung.
2. Speichere das Netz.
3. Bestimme die Verbindung mit dem betragsmäßig kleinsten Gewicht.
4. Lösche dieses Gewicht
5. Führe ein Nachtraining durch, bis der Fehler auf der Validierungsmenge wieder ansteigt.
6. Ist der Gesamtfehler nicht zu stark angewachsen, so springe zu Punkt 2.
7. Da der Fehler jetzt zu groß ist, lade das zuletzt gespeicherte Netz wieder.

Abbildung 4.1: Löschen der betragsmäßig kleinsten Gewichte

4.2 Implementierung

Um dieses Verfahren zu verwenden, muß in der Oberfläche oder in der Konfigurationsdatei der Batchversion des SNNS als Ausdünnungsfunktion „MagPruning“ ausgewählt worden sein. Die Rahmenfunktion *pr_callPrunFunc* ruft dann die Funktion

```
krui_err PRUNE_Mag (int pattern)
```

auf, welche in zwei geschachtelten Schleifen die Verbindung mit dem kleinsten Gewicht sucht. Dabei wird die modulglobale Variable `pr_candidateSaliency` zur Speicherung des bisher kleinsten Gewichts verwendet. Zielneuron und Verbindung mit kleinstem Gewicht stehen nach Beendigung der Funktion in den Variablen `pr_candidateTargetUnit` bzw. `pr_candidateLink`.

4.3 Bewertung

Dieses Verfahren wird in der Literatur oft zum Vergleich verwendet (z. B. in [Le Cun et al. 90] oder [Hassibi et al. 93a]), ist jedoch nirgends eingehender untersucht worden.

Es scheint im ersten Moment einleuchtend, daß kleine Gewichte auch nur einen geringen Einfluß auf das Netzverhalten haben. Doch in der Praxis zeigt sich, daß dieser Ansatz in manchen Fällen zu grob ist. Auch kleine Gewichte können für das Verhalten eines neuronalen Netzes wichtig sein.

Nach [Zell 94] funktioniert das Verfahren für große Netze anfangs ganz gut, sobald jedoch eine Anzahl Gewichte gelöscht wurde (ca. ein Drittel der gesamten Verbindungen), steigt der Netzfehler wieder stark an. Dieses Verhalten wird auch durch die Experimente von Le Cun, Denker und Solla ([Le Cun et al. 90]) bestätigt.

5. Optimal Brain Damage

Optimal Brain Damage (OBD) wurde von Yann Le Cun, John Denker und Sara Solla entwickelt (siehe [Le Cun et al. 90]) und auf der Konferenz NIPS 1990 vorgestellt. Es ist ein direktes Ausdünnungsverfahren für Verbindungen.

Das Verfahren versucht, die Relevanz für Verbindungen nicht über ihre betragsmäßige Größe zu bestimmen, sondern die Veränderung der Fehler- bzw. Energiefunktion zu approximieren. Die Verbindung, die den Netzfehler am wenigsten zu beeinflussen scheint, hat dann wohl auch die geringste Saliency und kommt daher für eine Löschung in Frage. Es wäre aber zu aufwendig, jedes Gewicht einzeln aus dem Netz zu nehmen, das Netz zu trainieren und den resultierenden Netzfehler festzustellen. Daher begnügt man sich mit einer Approximation der Fehlerfunktion.

5.1 Mathematische Grundlagen

Der Anstieg der Fehlerfunktion beim Löschen einer Verbindung wird mit einer *Taylorreihen-Entwicklung* approximiert. Nach [Bronstein et al. 73] gilt für eine Funktion f in einer Umgebung $[x_0 - \alpha, x_0 + \alpha]$, wenn sie dort $(n+1)$-mal differenzierbar ist:

$$f(x) = \sum_{\nu=0}^{n} \frac{f^{\nu}(x_0)}{\nu!} \cdot (x - x_0)^{\nu} + R_n(x) \tag{5.1}$$

wobei $R_n(x)$ das Restglied von der Ordnung $n+1$ ist. Hierfür gibt es verschiedene Abschätzungen, die bekannteste ist das *Lagrangesche Restglied*:

$$R_n(x) = \frac{f^{(n+1)}(x_0 + \vartheta \cdot (x - x_0))}{(n+1)!} \cdot (x - x_0)^{n+1} \tag{5.2}$$

mit der Konstanten $\vartheta \in [0, 1]$.

Des weiteren werden in diesem Abschnitt die Kettenregeln für die erste und zweite Ableitung benötigt. Sie seien hier zur Erinnerung nochmals aufgeführt:

$$(f \circ g)' = g' \cdot f'(g) \tag{5.3}$$

$$(f \circ g)'' = g'' \cdot f'(g) + (g')^2 \cdot f''(g) \tag{5.4}$$

5.2 Approximation der Fehleränderung

Die Taylorreihen-Entwicklung nach Formel (5.1) wird nun angewandt, um die Veränderung der Fehlerfunktion $E(W)$ mit $n = 2$ zu approximieren. Dabei ergibt sich:

$$\begin{aligned} \delta E &= E(W + \delta W) - E(W) \\ &= \frac{\partial E(W)}{\partial W} \cdot \delta W + \frac{1}{2}\frac{\partial^2 E(W)}{\partial W^2} + R_2(W + \delta W) \end{aligned} \tag{5.5}$$

Die Saliency eines Gewichts w_{ij} berechnet sich nun aus der vorangehenden Formel, indem man den Gewichtsänderungsvektor auf $\delta W = (0, \ldots, 0, -w_{ij}, 0, \ldots, 0)$ setzt. Man benötigt die Formeln jedoch in der Komponentenschreibweise und mit einigen Umformungen:

$$\begin{aligned} \delta E = & \sum_{1 \leq i,j \leq n_n} g_{(ij)} \cdot \delta w_{ij} + \frac{1}{2} \sum_{1 \leq i,j \leq n_n} h_{(ij),(ij)} \cdot \delta w_{ij}^2 \\ & + \frac{1}{2} \sum_{\substack{1 \leq i,j,k,l \leq n_n \\ (i,j) \neq (k,l)}} h_{(ij),(kl)} \cdot \delta w_{ij} \cdot \delta w_{kl} + O(\delta W^3) \end{aligned} \tag{5.6}$$

Man beachte, daß nicht-existente Verbindungen hier ein Gewicht von null haben. Außerdem sind w_{ij} die Komponenten des Gewichtsvektors W, $g_{(ij)}$ die Komponenten des Gradientenvektors G von E nach W sowie $h_{(ij),(kl)}$ die Elemente der *Hesse-Matrix*[1] H von E nach W. Für $g_{(ij)}$ und $h_{(ij),(kl)}$ gelten die Formeln:

$$g_{(ij)} = \frac{\partial E}{\partial w_{ij}} \quad \text{und} \quad h_{(ij),(kl)} = \frac{\partial^2 E}{\partial w_{ij} \partial w_{kl}} \tag{5.7}$$

Der Index (ij) bezeichnet hier ein Gewicht, nämlich das von Neuron i zu Neuron j, und ist daher als eine Komponente zu betrachten.

Des weiteren wurden in (5.6) das Restglied durch einen kubischen Term abgeschätzt und der Term der zweiten Ableitung aufgespalten in die Elemente auf

[1]benannt nach Ludwig Otto Hesse (1811 – 1874)

und neben der Diagonale der Hesse-Matrix, um die folgende Argumentation anschaulicher zu gestalten. Gleichung 5.6 entspricht dem Ausgangspunkt der Darstellung in [Le Cun et al. 90].

Die Formel (5.6) wird jetzt noch durch die folgenden Approximationen vereinfacht:

- Die extremale Approximation: Wenn man nach einem Algorithmus vorgeht, der dem Schema von Abb. 2.2 auf Seite 24 folgt, so kann man annehmen, daß das Netz vor dem Ausdünnen in ein Minimum trainiert wurde, d. h. die erste Ableitung der Fehlerfunktion (der Gradientenvektor G) ist null, der erste Term der rechten Seite der Taylorentwicklung fällt somit weg.
- Die quadratische Approximation: Hier setzt man voraus, daß die Energiefunktion lokal annähernd quadratisch ist, wodurch der letzte Term der rechten Seite (das Restglied) vernachlässigt werden kann.
- Die diagonale Approximation: Durch diese Approximation unterscheidet sich OBD von der Weiterentwicklung Optimal Brain Surgeon (siehe Kapitel 6). Von der Hesse-Matrix werden nämlich nur die Diagonalelemente (also alle $h_{(ij),(ij)}$ für $1 \leq i,j \leq n_n$) verwendet. Man kann also auch den dritten Term der rechten Seite von Gleichung (5.6) fallen lassen.

Mit all diesen Vereinfachungen erhält man folgende reduzierte Gleichung:

$$\delta E = \frac{1}{2} \sum_{1 \leq i,j \leq n_n} h_{(ij),(ij)} \cdot \delta w_{ij}^2 \tag{5.8}$$

5.3 Berechnung der Diagonalen der Hesse-Matrix

Die Relevanz einer Verbindung wird hier also durch die angenäherte Veränderung des Netzfehlers nach Gleichung (5.8) bestimmt. Das nächste Problem ist die Bestimmung der Diagonalen der Hesse-Matrix, also der Werte $h_{(ij),(ij)}$.

Ich verwende hier wieder die üblichen Bezeichnungen, wie auch in den vorangegangenen Kapiteln. Zur Berechnung der Diagonalelemente der Hesse-Matrix bestimme ich im Sinne einer konsistenten Notation auch den Gradientenvektor. Die Anwendung der Kettenregel auf die Definition des Gradientenvektors in (5.7) liefert:

$$\begin{aligned} g_{(ij)} &= \frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} \cdot \frac{\partial (\sum_k o_k w_{kj})}{\partial w_{ij}} \\ &= \delta_j \cdot o_i \end{aligned} \tag{5.9}$$

mit

$$\delta_j = \frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} = f'(net_j) \cdot \frac{\partial E}{\partial o_j} \tag{5.10}$$

wobei der zweite Term für Ausgabeneuronen ergibt:

$$\frac{\partial E}{\partial o_j} = \frac{\partial(\frac{1}{2}\sum_j (t_j - o_j)^2)}{\partial o_j} = o_j - t_j \tag{5.11}$$

Für verdeckte Neuronen erhält man dagegen

$$\begin{aligned} \frac{\partial E}{\partial o_j} &= \sum_k \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_j} = \sum_k \delta_k \cdot \frac{\partial(\sum_l o_l w_{lk})}{\partial o_j} \\ &= \sum_k \delta_k \cdot w_{jk} \end{aligned} \tag{5.12}$$

Man kann jetzt die Berechnung des Gradientenvektors zusammenfassen als

$$g_{(ij)} = o_i \cdot \delta_j \tag{5.13}$$

$$\delta_j = f'(net_j) \cdot \begin{cases} (o_j - t_j) & \text{für Ausgabeneuronen} \\ \sum_k \delta_k \cdot w_{jk} & \text{für verdeckte Neuronen} \end{cases} \tag{5.14}$$

Aus der Gleichung (5.7) und der Kettenregel nach (5.4) erhält man analog zu den Umformungen zur Bestimmung des Gradientenvektors

$$\begin{aligned} h_{(ij),(ij)} &= \frac{\partial^2 E}{\partial w_{ij}^2} = \frac{\partial^2 E}{\partial net_j^2} \cdot \left(\frac{\partial net_j}{\partial w_{ij}}\right)^2 + \frac{\partial E}{\partial net_j} \cdot \frac{\partial^2 net_j}{\partial w_{ij}^2} \\ &= \frac{\partial^2 E}{\partial net_j^2} \cdot \left(\frac{\partial(\sum_k o_k w_{kj})}{\partial w_{ij}}\right)^2 + \frac{\partial E}{\partial net_j} \cdot \frac{\partial^2(\sum_k o_k w_{kj})}{\partial w_{ij}^2} \\ &= \frac{\partial^2 E}{\partial net_j^2} \cdot o_i^2 + \frac{\partial E}{\partial net_j} \cdot 0 \\ &= \delta_j^{(2)} \cdot o_i^2 \end{aligned} \tag{5.15}$$

Durch erneute Anwendung der Regel (5.4) ergibt sich für $\delta_j^{(2)}$:

$$\begin{aligned} \delta_j^{(2)} &= \frac{\partial^2 E}{\partial net_j^2} = \frac{\partial^2 E}{\partial o_j^2} \cdot \left(\frac{\partial o_j}{\partial net_j}\right)^2 + \frac{\partial E}{\partial o_j} \cdot \frac{\partial^2 o_j}{\partial net_j^2} \\ &= f'(net_j)^2 \cdot \frac{\partial^2 E}{\partial o_j^2} + f''(net_j) \cdot \frac{\partial E}{\partial o_j} \end{aligned} \tag{5.16}$$

Jetzt fehlt nur noch die Berechnung für $\frac{\partial^2 E}{\partial o_j^2}$. Dazu unterscheide ich die Neuronen wieder nach Netztopologie. Für Ausgabeneuronen ergibt sich

$$\frac{\partial^2 E}{\partial o_j^2} = \frac{\partial(o_j - t_j)}{\partial o_j} = 1 \tag{5.17}$$

Für verdeckte Neuronen hingegen (wieder unter Verwendung von (5.4))

$$\begin{aligned}\frac{\partial^2 E}{\partial o_j^2} &= \sum_k \left(\frac{\partial^2 E}{\partial net_k^2} \cdot \left(\frac{\partial net_k}{\partial o_j} \right)^2 + \frac{\partial E}{\partial net_k} \cdot \frac{\partial^2 net_k}{\partial o_j^2} \right) \\ &= \sum_k \left(\delta_k^{(2)} \cdot \left(\frac{\partial(\sum_l o_l w_{lk})}{\partial o_j} \right)^2 + \delta_k \cdot \frac{\partial^2(\sum_l o_l w_{lk})}{\partial o_j^2} \right) \\ &= \sum_k \delta_k^{(2)} \cdot w_{jk}^2 \end{aligned} \tag{5.18}$$

Somit kann man auch die Berechnung für die Hesse-Matrix zusammenfassen:

$$h_{(ij),(ij)} = o_i^2 \cdot \delta_j^{(2)} \tag{5.19}$$

wobei für Ausgabeneuronen gilt

$$\delta_j^{(2)} = f'(net_j)^2 + f''(net_j) \cdot (o_j - t_j) \tag{5.20}$$

und für verdeckte Neuronen

$$\delta_j^{(2)} = f'(net_j)^2 \cdot \sum_k \delta_k^{(2)} \cdot w_{jk}^2 + f''(net_j) \cdot \sum_k \delta_k \cdot w_{jk} \tag{5.21}$$

5.4 Algorithmus zur Bestimmung der Relevanz

Im Algorithmus verwende ich der Einfachheit halber die doppelte Saliency für jede Verbindung, d. h.

$$2s_{ij} = h_{(ij),(ij)} \cdot w_{ij}^2 = o_i^2 \cdot \delta_j^{(2)} \cdot w_{ij}^2 \tag{5.22}$$

Sinnvollerweise werden die Summen nach den Gleichungen (5.12) und (5.18) zur Berechnung der Saliency aller Verbindungen in den jeweiligen Neuronen aufsummiert, wie dies auch bei Backpropagation passiert.

Dafür sollen in jedem Neuron i die zur Berechnung von δ_i und $\delta_i^{(2)}$ nötigen Werte gespeichert werden. Sie entstehen für verdeckte Neuronen durch Aufsummieren von Teilsummen für die ausgehenden Verbindungen nach den Gleichungen

(5.14) und (5.21). Die Ausgabeneuronen werden nach (5.14) und (5.20) initialisiert. Die Verarbeitung muß nach absteigender topologischer Ordnung erfolgen, d. h. ein Neuron i wird erst dann verarbeitet, wenn sämtliche Nachfolgerneuronen (das sind alle Neuronen j, zu denen eine Verbindung w_{ij} vom Neuron i aus existiert) schon abgearbeitet sind.

Da einige Terme mehrmals in den obengenannten Formeln auftauchen, kann man durch geschicktes Zurückpropagieren von Werten den Aufwand für die Berechnung um einen Faktor senken. Am besten stellt man den Vorgängerneuronen i die Terme $w_{ij} \cdot \delta_j$ und $w_{ij}^2 \cdot \delta_j^{(2)}$ zur Verfügung, aus denen sich mit Hilfe der ersten und zweiten Ableitung der eigenen Aktivierungsfunktion leicht die Werte δ_i und $\delta_i^{(2)}$ berechnen lassen. Daraus ergibt sich dann schließlich der Algorithmus in Abb. 5.1.

Für alle Neuronen j

$$term_j^1 := 0$$
$$term_j^2 := 0$$

Für alle Ausgabeneuronen j

$$\delta_j := f'(net_j) \cdot (o_j - t_j)$$
$$\delta_j^{(2)} := f'(net_j)^2 \ + \ f''(net_j) \cdot (o_j - t_j)$$

Für alle Verbindungen (ij)

$$2 \cdot s_{ij} \ = \ o_i^2 \cdot \delta_j^{(2)} \cdot w_{ij}^2$$
$$term_i^1 := term_i^1 \ + \ w_{ij} \cdot \delta_j$$
$$term_i^2 := term_i^2 \ + \ \delta_j^{(2)} \cdot w_{ij}^2$$

Für alle verdeckten Neuronen j

** in topologisch absteigender Reihenfolge **

$$\delta_j := f'(net_j) \cdot term_j^1$$
$$\delta_j^{(2)} := f'(net_j)^2 \cdot \ term_j^2 \ + \ f''(net_j) \cdot term_j^1$$

Für alle Verbindungen (ij)

$$2 \cdot s_{ij} \ = \ o_i^2 \cdot \delta_j^{(2)} \cdot w_{ij}^2$$
$$term_i^1 := term_i^1 \ + \ w_{ij} \cdot \delta_j$$
$$term_i^2 := term_i^2 \ + \ \delta_j^{(2)} \cdot w_{ij}^2$$

Abbildung 5.1: Algorithmus zur Bestimmung der Saliency s_{ij} für alle Verbindungen (ij) nach Optimal Brain Damage

5.5 Implementierung

Zur Verwendung von OBD im SNNS ist die Ausdünnungsfunktion „OptimalBrain-Damage“ auszuwählen. Die zugehörige Funktion im Kern steht wieder im Modul prun_f.c und lautet

```
krui_err PRUNE_OBD (int pattern)
```

Es wird die Saliency für jedes Muster einzeln berechnet und aufsummiert. Anschließend wird die Verbindung mit dem kleinsten Gewicht in modulglobalen Variablen abgelegt, und zwar der Zeiger auf das Nachfolgerneuron in der Variablen `pr_candidateTargetUnit` und der Zeiger auf die Verbindung selbst in der Variablen `pr_candidateLink`.

6. Optimal Brain Surgeon

Optimal Brain Surgeon (OBS) wurde 1993 von Babak Hassibi, David Stork und Gregory Wolff ([Hassibi et al. 93a], [Hassibi et al. 93b]) als Weiterentwicklung und Verbesserung von Optimal Brain Damage (siehe Kapitel 5) vorgestellt. Weitere Darstellungen dazu finden sich auch in [Hassibi et al. 94].

Die Theorie der Verfahren OBD und OBS ist tatsächlich ähnlich. Der Unterschied besteht darin, daß OBS die Saliency der Gewichte genauer berechnet.

6.1 Mathematische Grundlagen

Zur Berechnung der Saliency und der Gewichtsänderung der nicht gelöschten Verbindungen in OBS muß man Extrema mit Nebenbedingungen berechnen. Ich will hier nur kurz auf dieses Problem eingehen, das Heuser in [Heuser 83] genauer beschreibt. Dort wird die Fragestellung folgendermaßen formuliert:

> Auf $X \subset R^p$ sei eine reellwertige Funktion f und eine R^q-wertige Funktion g definiert, wobei $q < p$ sein möge. Wir sagen, f besitze im Punkte $\xi \in X$ ein *lokales Maximum* bzw. *Minimum unter der Nebenbedingung* $g(x) = 0$, wenn ξ in der Menge $M := \{x \in X \mid g(x) = 0\}$ liegt und eine δ-Umgebung U von ξ vorhanden ist, sodaß stets
>
> $$\forall x \in U \cap M : f(x) \leq f(\xi) \text{ bzw. } f(x) \geq f(\xi)$$
>
> gilt. Gesucht sind die Stellen lokaler Extrema von f unter der angegebenen Nebenbedingung und die Werte, die f in ihnen annimmt.

In diesem Falle existieren nämlich unter einigen weiteren Rahmenbedingungen (siehe [Heuser 83]) sogenannte Lagrangesche Multiplikatoren $\lambda_1, \ldots, \lambda_q$, mit denen die Gleichungen

$$\frac{\partial f(\xi)}{\partial x_k} \; + \; \sum_{j=1}^{q} \lambda_j \cdot \frac{\partial g_j(\xi)}{\partial x_k} \; = \; 0 \tag{6.1}$$

für alle $1 \leq k \leq p$ gelten.

Diese p Lagrangegleichungen und die q Nebenbedingungen

$$g_j(x) \; = \; 0 \tag{6.2}$$

bilden insgesamt ein Gleichungssystem mit $p + q$ Gleichungen und den $p + q$ Unbekannten $x_1, \ldots, x_p, \lambda_1, \ldots, \lambda_q$. Seine Lösung ergibt einen Kandidaten für den gesuchten Extremwert. Es muß jetzt noch geprüft werden, ob dieser Kandidat tatsächlich ein Extremum ist (genaueres hierzu siehe [Heuser 83]).

Außerdem wird in diesem Kapitel eine Formel zur Matrixinversion benötigt, die [Kailath 80] liefert:

$$\begin{aligned} &(A + B \cdot C \cdot D)^{-1} \\ &\quad = \; A^{-1} - A^{-1} \cdot B \cdot (C^{-1} + D \cdot A^{-1} \cdot B)^{-1} \cdot D \cdot A^{-1} \end{aligned} \tag{6.3}$$

6.2 Approximation der Fehleränderung nach OBS

Hassibi und Stork verwenden wie Le Cun, Denker und Solla die Taylorreihen-Entwicklung nach (5.6). Ich fasse hier allerdings die Terme anders zusammen:

$$\begin{aligned} \delta E \; = \; & \sum_{1 \leq i,j \leq n_n} g_{(ij)} \cdot \delta w_{ij} \; + \; O(\delta W^3) \\ & + \; \frac{1}{2} \sum_{1 \leq i,j,k,l \leq n_n} h_{(ij),(kl)} \cdot \delta w_{ij} \cdot \delta w_{kl} \end{aligned} \tag{6.4}$$

Sie verwenden nur zwei der drei Approximationen von OBD (siehe Seite 40), nämlich die extremale und die quadratische Approximation, sodaß also auf der linken Seite der Gleichung der erste Term (durch die Annahme, das Netz sei in ein Fehlerminimum trainiert und die erste Ableitung sei daher null) und der zweite Term (durch die Annahme, die Fehlerfunktion sei lokal annähernd quadratisch) wegfallen. Dadurch entsteht also die Form

$$\delta E \; = \; \frac{1}{2} \sum_{1 \leq i,j,k,l \leq n_n} h_{(ij),(kl)} \cdot \delta w_{ij} \cdot \delta w_{kl} \tag{6.5}$$

oder in Vektorschreibweise

$$\delta E \;=\; \frac{1}{2}\,\delta W^T \cdot H \cdot \delta W \tag{6.6}$$

Nachdem ein Gewicht auf null gesetzt werden soll, muß also der Gewichtsänderungsvektor δW folgender Gleichung genügen:

$$E_{(ij)}^T \cdot \delta W \;+\; w_{(ij)} \;=\; \delta w_{(ij)} + w_{ij} \;=\; 0 \tag{6.7}$$

wobei $E_{(ij)}$ der Einheitsvektor für das Gewicht $w_{(ij)}$ ist, d. h. er ist überall null bis auf die Stelle (ij), dort steht eine Eins. Im Endeffekt muß also folgendes Minimum bestimmt werden:

$$\min_{(ij)} \left\{ \min_{\delta W} \left(\frac{1}{2}\,\delta W^T \cdot H \cdot \delta W \right) \;|\; E_{(ij)}^T \cdot \delta W \;+\; w_{ij} \;=\; 0 \right\} \tag{6.8}$$

Hieraus kann folgende Lagrangegleichung nach Formel (6.1) bestimmt werden:

$$L \;=\; \frac{1}{2}\,\delta W^T \cdot H \cdot \delta W \;+\; \lambda \cdot (E_{(ij)}^T \cdot \delta W \;+\; w_{(ij)}) \tag{6.9}$$

mit λ als Lagrangeschem Multiplikator.

Nach mehreren Umformungsschritten erhalten Hassibi und Stork

$$\delta W \;=\; -\frac{w_{ij}}{[H^{-1}]_{(ij),(ij)}} \cdot H^{-1} \cdot E_{(ij)} \tag{6.10}$$

$$L_{(ij)} \;=\; s_{ij} \;=\; \frac{1}{2}\frac{w_{(ij)}^2}{[H^{-1}]_{(ij),(ij)}} \tag{6.11}$$

Hier ist $L_{(ij)}$ aus Gleichung (6.11) die Saliency und δW aus Gleichung (6.10) der Vektor der Gewichtsänderung beim Löschen der Verbindung von i nach j. Hier wird also nicht nur einfach ein Gewicht gelöscht, auch alle anderen Gewichte werden mit Hilfe dieser Gleichung ebenfalls angepaßt[1]. Der Effekt des Verfahrens im Vergleich mit Optimal Brain Damage und Magnitude Based Pruning ist in Abb. 6.1 veranschaulicht.

Um diese beiden Formeln benutzen zu können, muß allerdings zuvor die inverse Hesse-Matrix berechnet werden.

[1] Daher heißt das Verfahren auch Optimal Brain SURGEON (engl. Chirurg).

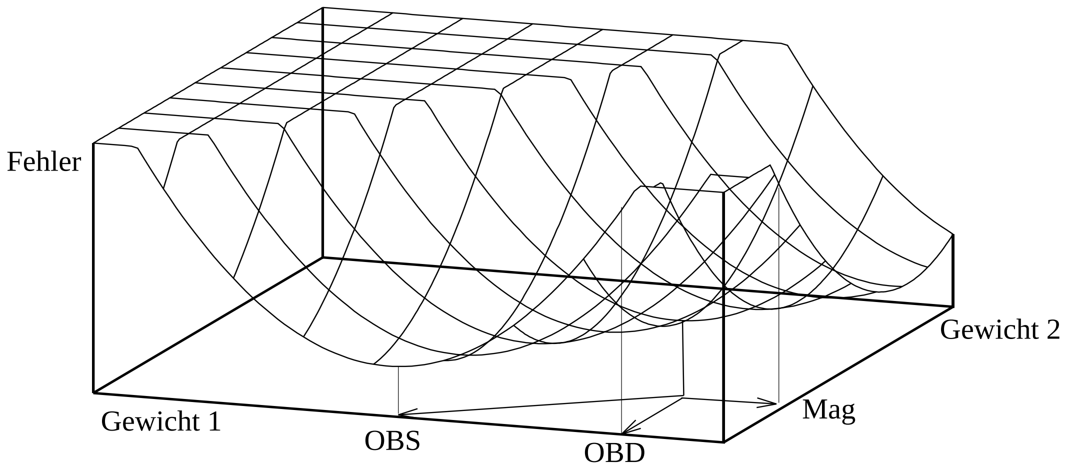

Abbildung 6.1: Vergleich der Wirkung von OBS, OBD und Magnitude Based Pruning nach Hassibi, Stork und Wolff

6.3 Berechnung der Inversen der Hesse-Matrix

Zu diesem Zweck soll die Energiefunktion zunächst zweimal abgeleitet werden, um daraus die Hesse-Matrix zu erhalten, welche dann invertiert werden muß.

Dazu sei das Netz als Funktion F beschrieben, das einen Eingabevektor In und die aktuellen Gewichte W in den Ausgabevektor O abbildet:

$$O \; = \; F(W, In) \tag{6.12}$$

für ein Ausgabeneuron oder

$$O \; = \; (F_1(W, In), \ldots, F_{n_o}(W, In)) \tag{6.13}$$

für mehrere Ausgabeneuronen. Die Energiefunktion E lautet in Vektorschreibweise wie folgt:

$$E \; = \; \frac{1}{2n_p} \sum_{k=1}^{n_p} (T^{[k]} - O^{[k]})^T \cdot (T^{[k]} - O^{[k]}) \tag{6.14}$$

Einmal abgeleitet ergibt sich:

$$\frac{\partial E}{\partial W} = -\frac{1}{n_p} \sum_{k=1}^{n_p} \frac{\partial F(W, In^{[k]})}{\partial W} \cdot (T^{[k]} - O^{[k]}) \tag{6.15}$$

Die zweite Ableitung hiervon ist:

$$H = \frac{\partial^2 E}{\partial W^2} = \frac{1}{n_p} \sum_{k=1}^{n_p} \left(\frac{\partial F(W, In^{[k]})}{\partial W} \cdot \frac{\partial F(W, In^{[k]})^T}{\partial W} \right. \tag{6.16}$$

$$\left. - \frac{\partial^2 F(W, In^{[k]})}{\partial W^2} \cdot (T^{[k]} - O^{[k]}) \right) \tag{6.17}$$

Da man davon ausgeht, daß das Netz in ein Minimum der Fehlerfunktion trainiert wurde, kann die tatsächliche und die gewünschte Netzausgabe nicht so weit voneinander entfernt liegen, sodaß der Term $(T^{[k]} - O^{[k]})$ hier vernachlässigt wird[2]. Dadurch reduziert sich die Formel für die Hesse-Matrix auf

$$H = \frac{1}{n_p} \sum_{k=1}^{n_p} \frac{\partial F(W, In^{[k]})}{\partial W} \cdot \frac{\partial F(W, In^{[k]})^T}{\partial W} \tag{6.18}$$

Jetzt definiert man der Einfachheit halber einen Ableitungsvektor $X^{[k]}$ als

$$X^{[k]} = \frac{\partial F(W, In^{[k]})}{\partial W} \tag{6.19}$$

sodaß man Gleichung (6.18) schreiben kann als

$$H = \frac{1}{n_p} \sum_{k=1}^{n_p} X^{[k]} \cdot X^{[k]T} \tag{6.20}$$

oder in einer rekursiven Form:

$$H_{m+1} = H_m + \frac{1}{n_p} X^{[m+1]} \cdot X^{[m+1]T} \tag{6.21}$$

mit der Vorbelegung $H_0 = \alpha I$, wobei α eine kleine Konstante[3] und I die Einheitsmatrix ist. Als Ergebnis nehmen wir $H_{n_p} = H$. Da aber nicht die Hesse-Matrix selbst, sondern ihre Inverse benötigt wird, setzen wir die obige Gleichung in die Formel (6.3) ein und erhalten:

$$H_{m+1}^{-1} = H_m^{-1} - \frac{H_m^{-1} \cdot X^{[m+1]} \cdot X^{[m+1]T} \cdot H_m^{-1}}{n_p + X^{[m+1]T} \cdot H_m^{-1} \cdot X^{[m+1]}} \tag{6.22}$$

[2] Dies wird in [Hassibi et al. 93b] (auch für größere Netzfehler) dadurch begründet, daß einerseits Singularitäten bei der Berechnung der Inversen für die möglicherweise degenerierte Hesse-Matrix vermieden werden können, andererseits der Unterschied zwischen $T^{[k]}$ und $O^{[k]}$ das Rauschen darstellt, das ja vom Netz nicht gelernt werden soll.

[3] α liegt zwischen 10^{-8} und 10^{-4}, die Wahl dieser Konstante hat laut Hassibi und Stork jedoch nur geringen Einfluß auf das Verfahren.

mit $H_0^{-1} = \alpha^{-1} I$ und $H_{n_p}^{-1} = H^{-1}$. Eigentlich berechnet diese Methode ja die Inverse von $(H+\alpha I)$, was der Einführung eines penalty-Terms $\alpha|\delta W|^2$ in Gleichung (6.9) entspricht, sodaß im Grunde genommen zusätzlich ein Weight Decay (siehe Seite 23) durchgeführt wird.

Die obige Berechnung funktioniert allerdings nur für ein Ausgabeneuron. Im Normalfall gibt es aber mehrere, sodaß $X^{[k]}$ kein Vektor, sondern eine Matrix der Größe $n_w \times n_o$ darstellt:

$$\begin{aligned} X^{[k]} &= \frac{\partial F(W, In^{[k]})}{\partial W} = \left(\frac{\partial F_1(W, In^{[k]})}{\partial W}, \ldots, \frac{\partial F_{n_o}(W, In^{[k]})}{\partial W} \right) \\ &= (X_1^{[k]}, \ldots, X_{n_o}^{[k]}) \end{aligned} \tag{6.23}$$

Für diesen Fall lautet die Gleichung (6.20):

$$H = \frac{1}{n_p} \sum_{k=1}^{n_p} \sum_{l=1}^{n_o} X_l^{[k]} \cdot X_l^{[k]T} \tag{6.24}$$

Auch hier ist die rekursive Form von Vorteil. Es entstehen zwei Gleichungen:

$$\begin{aligned} H_{m,l+1} &= H_{m,l} + \frac{1}{n_p} X_{l+1}^{[m]} \cdot X_{l+1}^{[m]T} \\ H_{m+1,1} &= H_{m,n_o} + \frac{1}{n_p} X_1^{[m+1]} \cdot X_1^{[m+1]T} \end{aligned} \tag{6.25}$$

Jetzt kann man mit Hilfe der Formel (6.9) wieder die Berechnungsvorschrift für die inverse Matrix angeben:

$$\begin{aligned} H_{m,l+1}^{-1} &= H_{m,l}^{-1} - \frac{H_{m,l}^{-1} \cdot X_{l+1}^{[m]} \cdot X_{l+1}^{[m]T} \cdot H_{m,l}^{-1}}{n_p + X_{l+1}^{[m]T} \cdot H_{m,l}^{-1} \cdot X_{l+1}^{[m]}} \\ H_{m+1,1}^{-1} &= H_{m,n_o}^{-1} - \frac{H_{m,n_o}^{-1} \cdot X_1^{[m+1]} \cdot X_1^{[m+1]T} \cdot H_{m,n_o}^{-1}}{n_p + X_1^{[m+1]T} \cdot H_{m,n_o}^{-1} \cdot X_1^{[m+1]}} \end{aligned} \tag{6.26}$$

Ergebnis der Berechnung ist $H_{n_p,n_o} = H$.

6.4 Bestimmung des Ableitungsvektors

Die Bestimmung des Ableitungsvektors X geschieht im Netz durch Rückwärtspropagierung von Teilsummen, ähnlich Backpropagation.

Allgemein gilt ja für X:

$$X_k^{[m]} = \left(\frac{\partial o_k}{\partial w_{ij}} \right)_{(ij) \in W} \tag{6.27}$$

bei Anlegen des Musters m und Betrachtung des Ausgabeneurons k. Die Komponenten des Vektors kann man hiervon wie folgt ableiten:

$$\begin{aligned}
x_{k,(ij)}^{[m]} &= \frac{\partial o_k}{\partial w_{ij}} = \frac{\partial f(net_k)}{\partial w_{ij}} = \frac{\partial f(net_k)}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{ij}} \\
&= f'(net_k) \cdot \frac{\partial(\sum_l o_l \cdot w_{lk})}{\partial w_{ij}} \qquad (6.28)
\end{aligned}$$

Hier muß man drei Fälle für die Neuronen j und k unterscheiden: Der einfachste Fall ist, daß die Neuronen identisch sind. In diesem Fall ergibt sich ganz einfach:

$$\frac{\partial(\sum_l o_l \cdot w_{lk})}{\partial w_{ij}} = o_i \qquad (6.29)$$

Im Falle $k \neq j$ muß unterschieden werden, ob Neuron k von Neuron j aus erreichbar ist, d. h. ob es einen gerichteten Weg von Neuron j zu Neuron k gibt (Notation: $k \succ j$). Dann muß $x_{k,(ij)}^{[m]}$ nämlich rekursiv berechnet werden:

$$\frac{\partial(\sum_l o_l \cdot w_{lk})}{\partial w_{ij}} = \sum_l w_{lk} \cdot \frac{\partial o_l}{\partial w_{ij}} = \sum_l w_{lk} \cdot x_{l,(ij)}^{[m]} \qquad (6.30)$$

Im dritten Fall jedoch, wenn es keinen gerichteten Weg im neuronalen Netz vom Neuron j zum Neuron k gibt, ist die Ableitung null.

Zusammengefaßt erhält man also die Formel

$$\frac{\partial o_k}{\partial w_{ij}} = \begin{cases} f'(net_j) \cdot o_i & \text{falls } k = j \\ f'(net_k) \cdot \sum_l w_{lk} \cdot x_{l,(ij)}^{[m]} & \text{falls } k \succ j \\ 0 & \text{sonst} \end{cases} \qquad (6.31)$$

Die Gleichung (6.31) habe ich in den Algorithmus nach Abb. 6.2 umgearbeitet. Dort wird der Ableitungsvektor $X_k^{[m]}$ für das Ausgabeneuron k im Muster m berechnet.

Die Idee des Algorithmus ist es, die Pfade von jeder Verbindung zu dem jeweils aktuellen Ausgabeneuron aufzuaddieren. Denn schließlich gilt für eine Verbindung der letzten Ebene (also zum Ausgabeneuron hin):

$$\frac{\partial o_j}{\partial w_{ij}} = f'(net_j) \cdot o_i \qquad (6.32)$$

und für eine Verbindung der vorletzten Ebene:

$$\begin{aligned}\frac{\partial o_k}{\partial w_{ij}} &= f'(net_k) \cdot \sum_l w_{lk} \cdot x^{[m]}_{l,(ij)} \\ &= f'(net_k) \cdot w_{jk} \cdot f'(net_j) \cdot o_i \end{aligned} \tag{6.33}$$

Wenn man noch eine Ebene weiter zurückgeht, erhält man:

$$\begin{aligned}\frac{\partial o_k}{\partial w_{ij}} &= f'(net_k) \cdot \sum_l w_{lk} \cdot x^{[m]}_{l,(ij)} \\ &= f'(net_k) \cdot \sum_l w_{lk} \cdot f'(net_l) \cdot w_{jl} \cdot f'(net_j) \cdot o_i \end{aligned} \tag{6.34}$$

Hieraus wird ersichtlich, daß man zur Berechnung von $x^{[m]}_{k,(ij)}$ eine Summe von Produkten berechnen muß, und zwar pro Pfad $(i, j, l_1, \ldots, l_n, k)$ von Verbindung (ij) zu Neuron k ein Produkt der folgenden Art:

$$o_i \cdot f'(net_j) \cdot w_{jl_1} \cdot f'(net_{l_1}) \cdot \ldots \cdot f'(net_{l_n}) \cdot w_{l_n k} \cdot f'(net_k) \tag{6.35}$$

Es ist also naheliegend, für jedes verdeckte Neuron die Teilprodukte aufzusummieren (dies bereitet keine Probleme, da Distributivität gilt) und an die Vorgängerneuronen weiterzugeben, wobei dann jeweils das bisherige Produkt mit der Verbindungsstärke und der Ableitung der Aktivierungsfunktion des Vorgängerneurons multipliziert wird. Eine Komponente des Ableitungsvektors X für eine bestimmte Verbindung berechnet sich dann als Produkt der Ausgabe des Vorgängerneurons und dem zwischengespeicherten Teilprodukt des nachfolgenden Neurons.

Also ergibt sich für ein Gewicht der letzten Schicht

$$x_{(ik)} = o_i \cdot f'(net_k) \tag{6.36}$$

falls das Nachfolgerneuron das ausgewiesene Ausgabeneuron k ist, und null sonst. Für das Vorgängerneuron als Hilfsterm zur Rückpropagierung erhält man

$$term_i = w_{ik} \cdot f'(net_k) \tag{6.37}$$

Für alle anderen Verbindungen erhält man:

$$x_{(ij)} := o_i \cdot term_j \tag{6.38}$$

und für deren Vorgängerneuronen berechnet man den Hilfsterm nach der Formel

$$term_i = \sum_j w_{ij} \cdot term_j \tag{6.39}$$

Daraus folgt dann die Vorgehensweise nach dem Algorithmus in Abb. 6.2.

Für alle Neuronen i
    $term_i := 0$

Für alle Verbindungen (ik)
** zum ausgezeichneten Ausgabeneuron* k ***
    $x_{(ik)} := o_i \cdot f'(net_k)$
    $term_i := w_{ik} \cdot f'(net_k)$

Für alle Ausgabeneuronen $l \neq k$
    Für alle Verbindungen (il)
        $x_{(il)} := 0$

Für alle verdeckten Neuronen j
** in topologisch absteigender Reihenfolge **
    $term_j := term_j \cdot f'(net_k)$
    Für alle Verbindungen (ij)
        $x_{(ij)} := o_i \cdot term_j$
        $term_i := term_i + w_{ij} \cdot term_j$

Abbildung 6.2: Algorithmus zur Bestimmung des Ableitungsvektors $X_k^{[m]}$ für das Muster m und das Ausgabeneuron k

6.5 Implementierung

Die zugehörige Ausdünnungsfunktion des SNNS heißt hier „OptimalBrainSurgeon". Bei deren Auswahl in der Oberfläche oder bei einem Batchlauf wird von der Rahmenfunktion die Funktion

```
krui_err PRUNE_OBS (int pattern)
```

im Modul prun_f.c (Kern) aufgerufen. Hier werden zunächst einige Datenstrukturen – z. B. für die inverse Hesse-Matrix – allokiert, danach die inverse Hesse-Matrix mit der Funktion

```
krui_err pr_obs_calculateInverseHessian (int pattern)
```

berechnet und die Saliency sämtlicher Verbindungen nach Formel (6.11) bestimmt, zuletzt werden noch alle Gewichte nach Gleichung (6.10) angepaßt. Die zum Löschen vorgesehene Verbindung wird – wie bei den anderen Verfahren auch – in den Variablen `pr_candidateTargetUnit` und `pr_candidateLink` abgelegt.

Die Funktion `pr_obs_calculateInverseHessian` initialisiert zuerst die außerhalb der Diagonalen liegenden Elemente der Matrix mit null und die Diagonalelemente mit einem kleinen Wert, der vom Benutzer im Pruning-Fenster (siehe 2.4.2) angegeben werden kann. Anschließend wird die inverse Hesse-Matrix schrittweise nach dem Verfahren in (6.26) berechnet. Dazu wird für jede Kombination aus Muster und Ausgabeneuron der Ableitungsvektor mit Hilfe der Funktion

```
void pr_obs_calculateDerivVector (struct Unit *output_unit)
```

nach dem Verfahren in Abb. 6.2 berechnet und anschließend ein Schritt der Berechnung nach (6.26) durch die Funktion

```
krui_err pr_obs_updateInverseHessian (int no_of_patterns)
```

durchgeführt.

Die Funktion

```
void pr_obs_setInitParameter (float param)
```

wird benötigt, um den Initialisierungsparameter für die inverse Hesse-Matrix (den Wert, den die Diagonalelemente der inversen Hesse-Matrix am Anfang des Verfahrens (6.26) erhalten) von der Oberfläche oder dem Batchprogramm aus zu verändern.

7. Skeletonization

Skeletonization oder Skelettierung ist ein Verfahren, mit dem sich irrelevante Neuronen aus dem Netzwerk entfernen lassen. Michael Mozer und Paul Smolensky stellten dieses Verfahren schon 1989 in [Mozer et al. 89] vor.

7.1 Verfahren

Mozer und Smolensky bestimmen die Relevanz ρ_i für ein Neuron i aus der Differenz der Trainingsfehlerwerte E mit und ohne dieses Neuron.

$$\rho_i \;=\; E_{\text{ohne Neuron } i} \;-\; E_{\text{mit Neuron } i} \tag{7.1}$$

Da auch hier wieder das Problem auftaucht, daß einfaches Ausprobieren zu aufwendig ist, schlagen die Autoren eine Approximation vor. Dazu führen sie zunächst den Begriff der *attentional strength* („Stärke der Aufmerksamkeit") α_i ein, ein konstanter Faktor zwischen 0 und 1, der mit der Ausgabe des Neurons i multipliziert wird. Ein Nachfolgeneuron erhält also folgende Netzeingabe:

$$net_j \;=\; \sum_i w_{ij} \cdot \alpha_i \cdot o_i \tag{7.2}$$

Ein Netz mit Aufmerksamkeitsfaktoren ist in Abb. 7.1 zu sehen.

Man kann sich den Faktor α_i wie einen Regler vorstellen, der die Ausgabe eines Neurons reduzieren, ja sogar ganz unterdrücken kann. Mit diesem Regler ist es also möglich, ein Neuron ganz aus dem Netz zu nehmen, indem man $\alpha_i = 0$ setzt. Für $\alpha_i = 1$ verhält sich das Netz wie gewohnt. Daraus folgt, daß man Gleichung (7.1) auch schreiben kann als

$$\rho_i \;=\; E_{\alpha_i=0} \;-\; E_{\alpha_i=1} \tag{7.3}$$

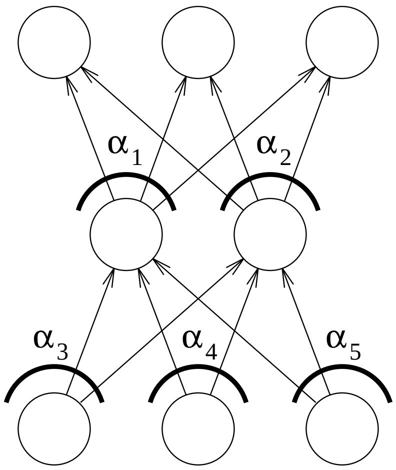

Abbildung 7.1: Neuronales Netz mit Aufmerksamkeitsfaktoren an den Eingabe- und verdeckten Neuronen

Bei dieser Berechnung sind natürlich die Stärken aller anderen Neuronen unvermindert. Gleichung (7.3) kann man als Differenzengleichung sehen, die durch eine Differentialgleichung approximiert werden kann. Es gilt nämlich:

$$\lim_{\gamma \to 1} \frac{E_{\alpha_i=\gamma} \; - \; E_{\alpha_i=1}}{\gamma - 1} \;=\; \left.\frac{\partial E}{\partial \alpha_i}\right|_{\alpha_i=1} \tag{7.4}$$

Für $\gamma = 0$ ergibt sich dann näherungsweise:

$$\frac{E_{\alpha_i=0} \; - \; E_{\alpha_i=1}}{-1} \;\approx\; \left.\frac{\partial E}{\partial \alpha_i}\right|_{\alpha_i=1} \tag{7.5}$$

und damit

$$\rho_i \;\approx\; \hat{\rho}_i \;:=\; -\left.\frac{\partial E}{\partial \alpha_i}\right|_{\alpha_i=1} \tag{7.6}$$

Abweichend von der sonstigen Praxis wird für die Berechnung der Relevanz bei Skeletonization die lineare Fehlerfunktion E^l verwendet, d. h.

$$E \;=\; E^l \;=\; \sum_j |t_{pj} - o_{pj}| \tag{7.7}$$

Hieraus konnte ich ableiten:

$$\hat{\rho}_i = -\frac{\partial E}{\partial \alpha_i} = -\frac{\partial \sum_j |t_j - o_j|}{\partial \alpha_i} = -\sum_j \frac{\partial |t_j - o_j|}{\partial \alpha_i} \tag{7.8}$$

Mit der Kettenregel nach (5.3) wird dies zu

$$\begin{aligned}
\hat{\rho}_i &= -\sum_j sgn(t_j - o_j) \cdot \frac{\partial (t_j - o_j)}{\partial \alpha_i} && (7.9)\\
&= \sum_j sgn(t_j - o_j) \cdot \frac{\partial o_j}{\partial \alpha_i} && (7.10)\\
&= \sum_j sgn(t_j - o_j) \cdot \delta_j^i && (7.11)
\end{aligned}$$

Die Hilfsvariable δ_j^i wird für $j = i$ null, da die Ausgabe nicht von der eigenen Stärke abhängt. Auch wenn Neuron j von Neuron i nicht mehr erreichbar ist, korrelieren o_j und α_i nicht mehr und die Ableitung verschwindet. Für $j \succ i$ gilt hingegen:

$$\begin{aligned}
\delta_j^i &= \frac{\partial o_j}{\partial \alpha_i} = \frac{\partial f(\sum_k w_{kj} \alpha_k o_k)}{\partial \alpha_i} && (7.12)\\
&= f'(net_j) \cdot \sum_k \frac{\partial w_{kj} \alpha_k o_k}{\partial \alpha_i} && (7.13)\\
&= f'(net_j) \cdot \left(\sum_k w_{kj} \alpha_k \frac{\partial o_k}{\partial \alpha_i} \; + \; w_{ij} o_i \right) && (7.14)\\
&= f'(net_j) \cdot \left(\sum_k w_{kj} \alpha_k \delta_k^i \; + \; w_{ij} o_i \right) && (7.15)
\end{aligned}$$

Unter der Annahme, daß sämtliche Stärken α_k außer der des untersuchten Neurons eins sind, erhält man insgesamt:

$$\begin{aligned}
\hat{\rho}_i &= \sum_j sgn(t_j - o_j) \cdot \delta_j^i \\
\delta_j^i &= \begin{cases} f'(net_j) \cdot (\sum_k w_{kj} \cdot \delta_k^i \; + \; w_{ij} \cdot o_i) & \text{falls } j \succ i \\ 0 & \text{sonst} \end{cases} && (7.16)
\end{aligned}$$

7.2 Berechnung der Relevanz nach Skeletonization

Formel (7.16) eignet sich wie bei Optimal Brain Surgeon in Kapitel 6 wieder als Grundlage zum Entwurf eines Algorithmus, der die Relevanz aller Neuronen entlang der Topologie des Netzwerks bestimmen kann. Da diese Formel jedoch nicht direkt zur Implementierung verwendet werden kann, muß die Rekursion zunächst „umgestülpt“ werden.

Löst man die Rekursion entlang eines Pfades $(i, k_1, k_2, \ldots, k_n, j)$ von einem Neuron i zum Ausgabeneuron j auf, so ergibt sich:

$$\begin{aligned}
\hat{\rho}'_i &= sgn(t_j - o_j) \cdot \delta^i_j \\
&= sgn(t_j - o_j) \cdot f'(net_j) \cdot w_{k_n j} \cdot \delta^i_{k_n} \\
&= \ldots \\
&= sgn(t_j - o_j) \cdot f'(net_j) \cdot w_{k_n j} \cdot \\
&\quad \ldots \cdot f'(net_{k_2}) \cdot w_{k_1 k_2} \cdot f'(net_{k_1}) \cdot w_{i k_1} \cdot o_i \qquad (7.17)
\end{aligned}$$

Diese Formel läßt sich wieder in eine Berechnungsvorschrift für die einzelnen Neuronen verwandeln.

In einer lokalen Variablen jedes Neurons werden Teilprodukte aufsummiert, und zwar von jedem Nachfolgerneuron. Ausgabeneuronen j gehen von dem Teilprodukt $sgn(t_j - o_j)$ aus. Jedes Neuron i multipliziert seine Teilsumme mit der Ableitung seiner Aktivierungsfunktion $f'(net_i)$ und addiert das Ergebnis zum Teilprodukt der Vorgängerzellen, nachdem es mit der Verbindungsstärke multipliziert wurde. Die approximierte Relevanz berechnet sich dann aus dem Teilprodukt und der eigenen Ausgabe. Für ein verdecktes Neuron der letzten Schicht ergibt sich also ein Hilfsterm von

$$term_i = \sum_j w_{ij} \cdot f'(net_j) \cdot sgn(t_j - o_j) \qquad (7.18)$$

Für alle anderen verwendet man

$$term_i = \sum_j w_{ij} \cdot f'(net_j) \cdot term_j \qquad (7.19)$$

Man beachte, daß die Hilfsterme $term$ und die Variablen δ aus (7.16) eine verschiedene Bedeutung haben. Die Saliency der verdeckten und Eingabeneuronen berechnet sich dann zu

$$\hat{\rho}_j := term_j \cdot o_j \qquad (7.20)$$

Der Algorithmus ist in Abb. 7.2 zusammengefaßt.

Für alle Neuronen i
$term_i := 0$

Für alle Ausgabeneuronen j
Für alle Vorgängerneuronen i
$term_i := term_i + w_{ij} \cdot f'(net_j) \cdot sgn(t_j - o_j)$

Für alle verdeckten Neuronen j
** in topologisch absteigender Reihenfolge **
$\hat{\rho}_j := term_j \cdot o_j$
Für alle Vorgängerneuronen i
$term_i := term_i + w_{ij} \cdot f'(net_k) \cdot term_j$

Für alle Eingabeneuronen j
$\hat{\rho}_j := term_j \cdot o_j$

Abbildung 7.2: Bestimmung der approximativen Relevanz $\hat{\rho}_i$ für alle Neuronen i

7.3 Implementierung

Will man Skeletonization verwenden, so wähle man in der Oberfläche oder in der Konfigurationsdatei der SNNS-Batchversion die gleichnamige Ausdünnungsfunktion aus. Die Rahmenfunktion `pr_callPrunFunc` ruft dann die spezielle Funktion

```
krui_err PRUNE_Skeletonization (int pattern)
```

aus dem Modul prun_f.c (Kern) auf. Dort wird das Verfahren nach Abb. 7.2 durchgeführt.

Die Funktion

```
void pr_skel_setInputPruning (int param)
```

ermöglicht es, aus der Oberfläche oder aus der Batchversion den Schalter für Input Pruning zu setzen.

8. Weitere Ausdünnungsverfahren

Dieses Kapitel gibt einen kurzen Überblick über die Verfahren, die außer den in den vorigen Kapiteln vorgestellten noch von Interesse sind.

8.1 Kostenfunktion für verdeckte Neuronen nach Hanson und Pratt

Stephen Hanson und Lorien Pratt gehen in [Hanson et al. 89] vom Weight-Decay-Verfahren für Backpropagation aus und entwickeln daraus ein ähnliches Verfahren, das allerdings alle Verbindungen, die in einem Neuron enden, zusammenfaßt und für den Mittelwert ihrer Gewichte

$$w_j = \sum_i |w_{ij}| \tag{8.1}$$

einen „bias-Term“ B (ähnlich dem C in Kapitel 3) festlegt.

Sie vergleichen hier drei mögliche Terme. Der erste ist der in Kapitel 3 beschriebene Term von Rumelhart (siehe Gleichung (3.4)), der zweite ein hyperbolischer und der dritte ein negativ exponentieller Term. Alle drei Kostenterme sind mit den zugehörigen Ableitungen (die ja für die Gewichtsänderungen wichtig sind) in der folgenden Tabelle 8.1 einander gegenübergestellt.

Hanson und Pratt konnten mit ihrem Verfahren einige Netze so ausdünnen, daß nur die minimale Anzahl der benötigten Neuronen übrig blieb. Mit sinkender Anzahl Neuronen sank allerdings auch die Konvergenzrate der Netze.

Kostenterm	Ableitung
$B = \dfrac{w_j^2}{(1+w_j)^2}$	$\dfrac{\partial B}{\partial w_{ij}} = \dfrac{2\,w_j(t)\,\mathrm{sgn}(w_{ij})}{(1+w_j^2)^2}$
$B = \dfrac{w_j}{(1+\lambda\,w_j)}$	$\dfrac{\partial B}{\partial w_{ij}} = \dfrac{\lambda\,\mathrm{sgn}(w_{ij})}{(1+w_j)^2}$
$B = 1 - e^{-\lambda\,w_j}$	$\dfrac{\partial B}{\partial w_{ij}} = \dfrac{\mathrm{sgn}(w_{ij})}{e^{-\lambda\,w_j)}}$

Tabelle 8.1: Kostenterme für das Verfahren von Hanson und Pratt

8.2 Minimierung der Aktivierungen verdeckter Neuronen nach Chauvin

Yves Chauvin schlug 1989 ein weiteres Kostentermverfahren für verdeckte Neuronen vor. Er betrachtet speziell deren Ausgabe. Seine Fehlerfunktion C berechnet sich aus zwei Summanden, wovon der erste der bekannte Fehlerterm von Backpropagation und der zweite ein Kostenterm ist:

$$C \;=\; \mu_{er} \cdot E_{er} \;+\; \mu_{en} \cdot E_{en} \;=\; \mu_{er} \cdot \sum_{i=1}^{n_o} (t_i - o_i) \;+\; \mu_{en} \cdot \sum_{i=1}^{n_h} e(o_i^2) \qquad (8.2)$$

mit einer positiven monotonen Funktion e. Die Ableitung des Kostenterms wird wieder durch ein rekursives Verfahren berechnet. Das Netz wird Schicht für Schicht untersucht. Die Gewichte vor der betrachteten Schicht werden nicht mit dem Kostenterm belegt. Für die restlichen Schichten gilt:

$$\frac{\partial E_{en}}{\partial w_{ij}} \;=\; \frac{\partial E_{en}}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{ij}} \;=\; \delta_i^{en} \cdot o_j \qquad (8.3)$$

Jetzt wird wieder unterteilt. Die Verbindungen, die auf Neuronen der betrachteten Schicht zeigen, erhalten folgenden Wert für δ_i^{en}:

$$\delta_i^{en} \;=\; \frac{\partial e(o_i^2)}{\partial net_i} \;=\; \frac{\partial e(o_i^2)}{\partial o_i^2} \cdot \frac{\partial o_i^2}{\partial o_i} \cdot \frac{\partial o_i}{\partial net_i} \;=\; 2 \cdot e' \cdot o_i \cdot f'(net_i) \qquad (8.4)$$

Für alle weiter unten liegenden Schichten ergibt sich:

$$\delta_k^{en} \;=\; f'(net_k) \cdot \sum_j \delta_j^{en} w_{jk} \qquad (8.5)$$

Dies ist das gleiche Verfahren wie Backpropagation, jedoch mit einem anderen zurückpropagierten Term, deshalb können die beiden Terme einfach addiert werden.

$$\begin{aligned}\Delta w_{kl} &= -\alpha\mu_{er}\delta_k^{er} o_l \; - \; \alpha\mu_{en}\delta_k^{en} o_l \\ &= -\;\alpha o_l(\mu_{er}\delta_k^{er} \; + \; \mu_{en}\delta_k^{en}) \\ &= \alpha o_l \delta_k^{ac} \qquad (8.6)\end{aligned}$$

mit einem δ_k^{ac} für beide Fehlerterme

$$\begin{aligned}\delta_k^{ac} &= f'(net_k)\cdot\sum_i(\mu_{er}\delta_i^{er} \; + \; \mu_{en}\delta_i^{en})\cdot w_{ik} \\ &= f'(net_k)\cdot\sum_i \delta_i^{ac}\cdot w_{ik} \qquad (8.7)\end{aligned}$$

Es bleibt noch, die Kostenfunktion e zu bestimmen. Hier schlug Chauvin drei verschiedene vor:

$$e(o^2) \; = \; o^2, \quad e(o^2) \; = \; log(1+o^2) \quad \text{und} \quad \frac{o^2}{1+o^2} \qquad (8.8)$$

Hierbei bestraft der erste Term alle Neuronen gleich stark, unabhängig von ihren Kosten. Die beiden anderen Terme bestrafen Neuronen mit niedrigen Kosten stärker. Als weitere Variante besteht die Möglichkeit, zusätzlich einen Weight-Decay-Term in die Fehlerfunktion einzubringen.

8.3 Das Löschen unnützer Neuronen nach Sietsma und Dow

Jocelyn Sietsma und Robert Dow ([Sietsma et al. 91]) konzentrierten sich in ihrer Arbeit vor allem auf eine Verbesserung der Generalisierungsleistung. In ihren Experimenten fügten sie einerseits Rauschen zu den Eingabedaten hinzu und versuchten andererseits überflüssige Neuronen zu löschen.

Die Netze wurden schichtenweise untersucht, wobei in einer ersten Phase des Ausdünnens Neuronen gesucht wurden, die entweder stark mit den Ausgaben anderer Neuronen korrelieren[1] oder fast konstante Ausgabe über die Menge der Muster haben[2].

[1] Sietsma und Dow untersuchten auch das Verfahren, ganz generell nach einer linearen Abhängigkeit der Ausgaben zu suchen, stellten aber fest, daß dies keine besseren Ergebnisse lieferte und darüberhinaus sehr sensibel gegenüber der Wahl einer Toleranzkonstante war.

[2] also mit dem on-Neuron korrelieren

In der zweiten Runde wurde versucht, Neuronen aufzuspüren, die zwar unabhängig von den anderen Neuronen ihrer Schicht sind, deren Ausgabe aber nicht relevant ist für die folgende Schicht. Man kann Mengen von Mustern nämlich in Klassen einteilen gemäß der Ausgabe einer verdeckten Schicht (d. h. wenn bei zwei Mustern die Ausgaben sämtlicher Neuronen einer Schicht gleich sind, gehören sie bezüglich dieser Schicht in eine Klasse). Um diese Klassen darzustellen, genügt in der vorhergehenden Schicht eine bestimmte Anzahl von Neuronen. Sind dort mehr Neuronen als erforderlich, so können die überflüssigen gelöscht werden.

Diese Vorgehensweise birgt eine Gefahr: Falls die entstehenden Klassen nicht linear separierbar sind, genügt eine Schicht trainierbarer Gewichte nicht zu deren Darstellung. Sietsma und Dow behelfen sich dadurch, daß sie in diesem Fall zusätzliche verdeckte Schichten einbauen, wodurch die Netze zwar schmaler, aber auch länger werden.

Die Untersuchungen der Autoren ergaben für das zu den Eingabedaten hinzugefügte Rauschen eine erstaunliche positive Wirkung auf das Generalisierungsverhalten der Testnetze. Sie fanden heraus, daß in diesen Netzen mehr verdeckte Neuronen aktiv an der Informationsverarbeitung beteiligt sind als in den ohne Rauschen trainierten Netzen. Diese Beobachtung widerspricht auch der These, größere Netze würden sich einzelne Muster merken, da ja gar nicht alle Neuronen tatsächlich arbeiten.

Die Ausdünnungsverfahren der ersten Phase hatten eine leicht verbesserte Generalisierungsleistung, allerdings nur, wenn das Netz ohne Rauschen trainiert worden war. Nach der zweiten Phase wurde die Leistung der Netze sogar eher schlechter.

Die Verfahren der ersten Stufe wurden von Jochen Biedermann implementiert und getestet. Diese Verfahren werden neben meinen Implementierungen in den SNNS eingebaut.

8.4 Weitere Vorschläge

8.4.1 Sensitivitätsverfahren

Auf die Methode der *Principal Component Analysis* nach Jolliffe ([Jolliffe 86] baut ein Verfahren von Levin, Leen und Moody ([Levin et al. 94]) auf: *Principal Component Pruning*. Analysiert wird die Aktivierung von Neuronen aufeinanderfolgender Schichten. Dazu wird für jede Schicht eine Korrelationsmatrix der Aktivierungen und Gewichte erstellt. Das Verfahren wird auf ein trainiertes Netz angesetzt, braucht aber kein Nachtraining.

Whiteley, Starkweather und Bogart stellten ein Verfahren zum Ausdünnen mit Hilfe genetischer Algorithmen vor ([Whiteley et al. 90]). Weitere Betrachtungen dazu siehe auch Hancock ([Hancock 92]). Heine und Neumann ([Heine et al. 94]) verwendeten einen evolutionären Algorithmus zur Optimierung von Prognosemodellen für Energieversorgungsunternehmen. Bei ihren Arbeiten gewannen sie die Erkenntnis, daß die Ergebnisse eines Ausdünnungsverfahrens sich verbessern, wenn man das Ende von Training und Nachtraining mit Hilfe des Fehlers auf einer Testmenge bestimmt.

Tsaptsinos, Mirzai und Leich ([Tsaptsinos et al. 92]) verwendeten die Korrelationsanalyse zur Bestimmung der Saliency von Verbindungen und Neuronen in einem Sensitivitätsverfahren.

Kurokawa, Ho und Mori ([Kurokawa et al. 93]) schlagen ein Sensitivitätsverfahren für Neuronen vor mit der Saliency

$$C_j = \frac{2}{1 - o_j^2} \tag{8.9}$$

Lim und Ho ([Lim et al. 94]) verbessern die Topologie neuronaler Netze durch abwechselndes Hinzufügen und Löschen verdeckter Neuronen. Das Kriterium (die Saliency) dabei ist die Summe der Gewichte der eingehenden Verbindungen eines Neurons.

8.4.2 Kostentermverfahren

Hier kann man auch noch Quickprop anführen, das ja standardmäßig einen Weight-Decay-Parameter eingebaut hat. Andreas Sieler minimierte die Topologie neuronaler Netze, die Verbrauchsprognosen für Energieversorgungsunternehmen treffen sollten ([Sieler 93]). Er konnte die Netze zwar ausdünnen, die Generalisierungsleistung blieb jedoch bestenfalls konstant und wurde in manchen Fällen sogar schlechter.

Einige Kostentermverfahren basieren auch auf Betrachtungen aus der Informationstheorie. Wichtig ist hier die *Minimum Description Length*, die minimale Anzahl Bits zur Beschreibung einer Information. Weigend, Rumelhart und Huberman ([Weigend et al. 91]) vergleichen zwei Ansätze: die Minimierung der Anzahl der Gewichte und die Minimierung der pro Gewicht nötigen Bits. Hinton und van Camp hingegen minimieren die Bits für die Beschreibung des neuronalen Netzes und des Abstands zwischen der von diesem Netz produzierten Ausgabe und den Lernbeispielen ([Hinton et al. 93]).

Deco, Finnoff und Zimmermann ([Deco et al. 93]) verwenden ebenfalls einen informationstheoretischen Ansatz für ein Weight-Decay-Verfahren. Mit der Berechnung der „mutual information“ konnte das Lernen von Rauschen verringert werden. Bei der Vorhersage der Zinsentwicklung in Deutschland mit einem unüberwacht trainierten Netz erzielten die Autoren gute Erfolge.

Santini stellt in [Santini 92] ein weiteres Weight-Decay-Verfahren vor. Zusätzlich zum üblichen Kostenterm gibt es einen *reduction term* $R(i)$ für jedes Neuron als Funktion aller seiner ausgehenden Verbindungen. Als ein Beispiel für einen solchen Term wird

$$R(i) \;=\; \phi(\sum_j w_{ij}) \quad \text{mit} \quad \phi(x) \;=\; \frac{1}{1 \;+\; e^{-x^2/\sigma}} \tag{8.10}$$

angegeben. Diese Funktion wirkt nur, wenn die Summe der Gewichte, die von einem Neuron ausgehen, klein ist. Der Faktor σ kontrolliert die Größe des Attraktor-Bassins. Ein Nachteil des Verfahrens ist, daß ein großes Gewicht den Ausdünnungseffekt auf seine kleinen „Nachbarn“ verhindern kann. Santini schlug daher als verbesserten Term

$$R(i) \;=\; \phi(\zeta_i) \text{ mit } \zeta_i \;=\; \frac{\sqrt{\sum_{j=1}^{M} w_{ij}^2}}{1 \;+\; \beta \cdot \left[\frac{\prod_j w_{ij}}{\left(\sum_{j=1}^{M} w_{ij}^2\right)^{\frac{n}{2}}} \right]} \tag{8.11}$$

vor, wobei ζ_i der *shape factor* für das Neuron i ist. Für $M = 2$ werden Verbindungen verstärkt gegen null trainiert, wenn Nachbarverbindungen schon sehr klein sind. Daher werden die großen Gewichte auf wenige Neuronen konzentriert.

9. Tests mit den Sensitivitätsverfahren

Meine Verfahren testete ich anhand von fünf Beispieldatensätzen. Als Lernverfahren verwendete ich grundsätzlich Standard Backpropagation. Der Lernparameter η wurde dabei nach vorhergehenden Experimenten mit Backpropagation geeicht. Auf das Verwenden verschiedener Werte wurde in den meisten Fällen zugunsten der Vergleichbarkeit verzichtet. Der maximale Anstieg des Netzfehlers war immer auf 25% gesetzt. Die Klassifizierungen werden nach der „Winner-Takes-All"-Methode ermittelt, d. h. nachdem immer nur ein Ausgabeneuron aktiviert sein muß, wird angenommen, das mit der größten Aktivierung zeige die Klasse an.

Die Tests wurden auf Sun Sparc-Stations gleicher Architektur durchgeführt. Dennoch sind die beiden verwendeten Maschinen unterschiedlich in ihrer Leistung. Daher wird zu den Angaben für die CPU-Zeit noch angegeben, ob es sich um die langsamere Maschine A oder die schnellere Maschine B handelt.

9.1 Tests mit dem Letters-Netz

Als erstes Beispiel diente ein vollständig verbundenes Netz zur Erkennung von Großbuchstaben mit den Ausmaße $35 \times 10 \times 26$[1], das als Beispiel mit dem SNNS mitgeliefert wird. Hierzu gehören 26 Muster (eines pro Klasse). Es gibt keine Testmenge.

Das Letters-Netz war als kleines Testnetz gedacht, mit dem die Wirkungsweise der Ausdünnung demonstriert werden kann. Da hier jedoch nur ein Muster pro Klasse existiert, können die Ergebnisse nicht verallgemeinert werden. Aufgrund der fehlenden Testmenge können auch keine Aussagen über die Generalisierung gemacht werden. Vor dem Ausdünnen hatte das Netz 610 Gewichte. Die Ergebnisse sind in Tabelle 9.1 zusammengefaßt.

[1] 35 Eingabe-, 10 verdeckte und 26 Ausgabeneuronen

η	Epochen		Fehler		gelöscht			richtig	CPU-Zeit	
	E	N	min.	akz.	L	I	H	Train	h:min:s	M
Backpropagation										
0,8	300	–	1	–	–	–	–	26	0:20	A
Löschen der kleinsten Gewichte										
0,8	300	100	1	5	509	17	–	26	1:47	B
0,8	400	150	0.5	3	506	17	–	26	1:53	A
Optimal Brain Damage										
0,8	400	150	2	5	442	14	2	25	2:23	B
0,8	400	150	7.5	10	483	13	2	26	1:42	B
0,8	400	150	0.5	3	283	10	1	26	1:27	B
Optimal Brain Surgeon										
0,2	400	150	0.5	3	517	19	–	25	27:17:36	B
0,8	400	150	0.5	3	503	16	–	26	43:58:14	A
0,8	1000	600	0.5	3	506	18	–	26	27:07:04	B
Skeletonization										
0,8	400	150	0.5	3	360	11	5	24	1:08	A
0,8	400	150	0.5	3	360	11	5	24	1:06	A

Tabelle 9.1: Ergebnisse der Tests mit dem Letters-Netz. Erläuterungen zu den Bezeichnungen finden sich im Text.

Ich möchte als kurze Einleitung in die Notation dieser und der folgenden Tabellen die erste Zeile unter „Löschen der kleinsten Gewichte“ erklären:

Die erste Spalte gibt den Lernparameter $\eta = 0,8$ für das untergeordnete Lernverfahren (in diesen Tests immer Backpropagation) an. Danach folgt die Information, daß das erste Training (E) maximal 300 und jedes Nachtraining (N) maximal 100 Epochen dauern sollte. Bei einem minimalen Netzfehler von 1 (absoluter Wert) soll das Netz das Training vor Erreichen der maximalen Zyklenzahl abbrechen. Ein Fehler von 5 ist nach einem Nachtraining auf jeden Fall noch klein genug, um das Ausdünnungsverfahren fortzuführen (akzeptierter Fehler).

In den folgenden Spalten kann das Ergebnis des Tests abgelesen werden. Es wurden 509 Verbindungen (L für Links), 17 Eingabeneuronen (I = Input Unit) und keine verdeckten Neuronen (H = Hidden Unit) gelöscht. Das ausgedünnte Netz erkannte 26 Trainingsmuster (Train) richtig (also in diesem Beispiel alle); über die richtige Klassifizierung von Testmustern (Test) kann keine Aussage gemacht werden, da hier eine Testmenge fehlt. Der Test dauerte 1 Minute und 47 Sekunden auf der Maschine B.

Bei Skeletonization sind die Verbindungen, die durch das Löschen der Neuronen zwangsweise wegfallen, der Vollständigkeit halber mit aufgeführt. Die gelöschten Neuronen bei den Weight-Decay-Verfahren sind „tote Neuronen", die durch entfallende Verbindungen nicht mehr auf einem Pfad zwischen Ein- und Ausgabeschicht lagen, somit zum Netzverhalten nichts mehr beitragen konnten und entfernt werden mußten.

Hier zeigten sich vor allem drei interessante Aspekte:

- Die Verfahren „Löschen der kleinsten Gewichte" und „Optimal Brain Damage" benötigten ungefähr dieselbe Zeit. OBD war aber trotz seines theoretisch fundierteren Algorithmus hier das schlechtere Verfahren.
- „Optimal Brain Surgeon" brauchte im krassen Gegensatz zu den beiden anderen Weight-Pruning-Verfahren jeweils über einen Tag, was auf die zwar sehr trickreiche, aber immer noch extrem zeitaufwendige Invertierung der Hesse-Matrix zurückzuführen ist.
- Skeletonization lieferte das schlechteste Klassifizierungsergebnis, es löschte aber jeweils elf Eingabeneuronen und fünf verdeckte Neuronen. Eine Merkmalsreduktion scheint mit diesem Verfahren also möglich. Bessere Ergebnisse erreichten in dieser Hinsicht jedoch die drei anderen Verfahren.

9.2 Verwendung der Kreßel-Ziffern

Als zweites Beispiel konnte ich eine Menge von 1000 handgeschriebenen Ziffern aus dem BITEX-Projekt ([Kreßel 92]) verwenden. Von jeder Ziffer wurden 100 Varianten in eine 16×16–Matrix von reellen Werten gefaßt. Diese Testreihe war die aufwendigste: hier wurden 1000 Muster in einem $256 \times 40 \times 10$-Netz mit 10.640 Verbindungen trainiert.

Ich verzichtete auf Optimal Brain Surgeon, da dieses die inverse Hesse-Matrix im Speicher verwalten muß, eine quadratische Matrix, bei der die Kantenlänge der Anzahl der Gewichte im Netz entspricht. Das allein hätte einen Speicherbereich von über 400 Megabyte erfordert. Zusätzlich war zu erwarten, daß ein Testlauf unangemessen lang gedauert hätte, nachdem schon das kleine Letters-Netz länger als einen Tag benötigt hatte. Bei Skeletonization wurde für einen Test nur Input Pruning durchgeführt. Die Ergebnisse sind in Tabelle 9.2 zu sehen.

Bei Tests im BITEX-Projekt ergaben sich für Backpropagation mit Lernparameter 0,2 (allerdings mit 10.000 Mustern – daher sind diese Werte nicht zum Vergleich geeignet) Werte von 9981 bzw. 9740 erkannten Mustern.

η	Epochen		Fehler		gelöscht			richtig		CPU-Zeit	
	E	N	min.	akz.	L	I	H	Train	Test	h:min:s	M
Backpropagation											
0,2	20	–	2	–	–	–	–	1000	930	12:16	A
Löschen der kleinsten Gewichte											
0,8	40	20	5	20	10416	179	6	997	897	39:18:15	B
Optimal Brain Damage											
0,8	40	20	5	20	9230	7	22	999	825	135:12:16	A
0,8	40	20	20	50	9616	15	26	984	831	55:47:46	B
Skeletonization											
0,2	40	20	20	50	9044	–	35	981	849	27:37	B
0,2	60	40	20	50	9282	72	33	989	834	14:47	B
0,8	100	50	20	50	9044	–	35	992	898	15:11	B
0,2	100	100	20	50	9400	18	36	986	849	36:40	B
0,2	1000	1000	5	30	9370	12	35	993	801	5:36:32	B
Skeletonization mit Input-Pruning											
0,2	20	20	2	4	8200	205	–	986	758	10:46:32	B

Tabelle 9.2: Ergebnisse der Tests mit den Kreßel-Ziffern

Folgende Beobachtungen sind hier interessant:

- Der Trainingsfehler erreicht in ausgedünnten Netzen in keinem Fall die Werte von purem Backpropagation.
- Die Generalisierung hat sich durchweg verschlechtert. Je mehr Elemente gelöscht wurden, desto schlechter waren die Ergebnisse. Die verwendeten Verfahren scheinen für eine Erhöhung der Generalisierungsleistung nicht geeignet zu sein.
- Bei einem Versuch mit Skeletonization wurden viele Input-Neuronen gelöscht (205 von 256), sodaß die Merkmale hier sinnvoll reduziert wurden. Förderlich war offensichtlich, daß das Netz bis zum „Knick" der Fehlerfunktion trainiert wurde, der nach zwanzig Epochen erreicht ist.
- Andererseits ist im obengenannten Beispiel die Generalisierungsleistung am schlechtesten, was der These widerspricht, kleinere Netze könnten einzelne Merkmale der Muster besser extrahieren und damit unbekannte Muster besser erkennen. Scheinbar wurden die Eingabemuster soweit reduziert, daß nur noch die Trainingsdaten erkannt werden können, die markanten Eingaben der Testmuster jedoch in bereits gelöschten Bereichen des Eingabevektors liegen.

- Weiterhin ist interessant, daß die Anzahl der gelöschten Eingabeneuronen bei ähnlichen Parametern ziemlich schwankt und nicht etwa monoton mit der Anzahl der Trainingszyklen steigt, wie man dies erwarten könnte.
- Die Testläufe mit Optimal Brain Damage waren sehr rechenzeitaufwendig. Dieses Verfahren ist für größere Netze vielleicht nur bedingt geeignet.

9.3 Training mit akustischen Daten

Herr Dr. Attia von der Firma Robert Bosch GmbH stellte mir freundlicherweise Beispieldaten aus der Subjektiven Geräusch-Nachbeurteilung (SGNB) von Generatoren, Startern und Motoren zur Verfügung (siehe [Attia et al. 93]). Dies sind Daten aus den Frequenzspektra der obengenannten Maschinen, die Aufschlüsse darüber geben sollen, ob die Maschine fehlerfrei gebaut wurde oder nicht.

9.3.1 Grundlagen der Geräuschbeurteilung

Da die Geräusche bisher subjektiv durch Menschen beurteilt werden, sind die Ergebnisse sehr inkonsistent. Die Prüfergebnisse stimmen im Mittel nur zu 60 bis 70 Prozent mit früheren Beurteilungen oder Beurteilungen anderer Prüfer überein. Wichtig wäre eine objektive und reproduzierbare Bewertung. Die Probleme der bisherigen Prüfstände liegen u. a. bei der *Merkmalsfindung* (welche Informationen aus dem Geräusch sind wichtig).

Künstliche neuronale Netze sollen dabei helfen. Sie können nach einer Vorverarbeitung auch andere Merkmale aus den Geräuschen verwenden, wie z. B. Körperschallanteile, die nicht als Luftschall abgegeben werden. Außer den rein physikalischen Größen aus Zeit- und Frequenzmessungen sind aber auch psychoakustische Größen wie Lautheit, Teilrauhigkeit, Schwankungsstärke und Schärfe zu verwenden, um eine gehörorientierte objektive Geräuschprüfung zu erreichen. Dazu wird die *Fourier Time Transformation (FTT)* verwendet, die ein Sonogramm mit den wichtigsten psychoakustischen Merkmalen enthält.

Mit der Subjektiven Geräusch-Nachbeurteilung (SGNB) können konsistente Meßdaten gewonnen werden: mehrere Prüfer können parallel Geräusche beurteilen und dazu Referenzsignale von guten und fehlerhaften Maschinen abrufen. Bewertet wird nach den Kategorien eins bis fünf (1 = gut, 5 = schlecht).

Für meine Untersuchungen wurden mir drei Beispielmengen zur Verfügung gestellt:

1. 112 Merkmale aus Schallsignalen von 139 Generatoren mit der Einteilung in fünf verschiedene Fehlerklassen, ohne Testmenge
2. 15 Merkmale für 183 Starter mit Bewertungen nach einer Notenskala von 1 (gut) bis 5 (schlecht), ohne Testmenge
3. 12 Merkmale von 89 E-Getriebemotoren mit einer Zuordnung zu einer aus vier Klassen. Aufteilung in eine Trainings- und eine Testmenge.

Für diese Beispiele ist eine Merkmalsreduktion besonders wichtig, die am besten mit Skeletonization erreicht werden kann. Die Beispielmengen waren teilweise in eine Trainings- und eine Testmenge aufgeteilt. Zwar kann eine verbesserte Generalisierung nur anhand von Testmengen gemessen werden, jedoch wollte ich die Beispiele bei fehlender Testmenge nicht aufteilen, einerseits um eine Vergleichbarkeit mit den bei der Firma Bosch erzielten Ergebnissen zu erhalten, andererseits weil mir die Struktur der Beispielmengen nicht bekannt war.

9.3.2 Akustische Fehlerdiagnose von Generatoren

Die Ergebnisse für die 139 Generatoren sind in Tabelle 9.3 zusammengefaßt. Hier wurde ein Netz mit den Ausmaßen $112 \times 60 \times 5$ verwendet. Das Netz war vollständig verbunden, hatte also vor dem Ausdünnen 7020 Verbindungen. Die Generatoren mußten anhand von 112 akustischen Merkmalen in fünf Fehlerklassen eingeteilt werden. Die Klassifizierung erfolgte in die Kategorien „fehlerfrei“, „magnetischer Fehler“, „Lagerfehler“, „Streifer“ und eine Restfehlerklasse. Es war keine Testmenge vorhanden.

Ein Vergleichstest mit Backpropagation klassifizierte nach 100 Trainingszyklen 131 der 139 Geräusche (94%) richtig. Es wurde jeweils mit Lernparameter $\eta = 0,2$ gearbeitet.

Zu den Ergebnissen läßt sich feststellen:

- Optimal Brain Damage schneidet hier nicht besonders gut ab. Es löscht weniger Verbindungen und Neuronen als andere Verfahren und erkennt trotzdem am wenigsten Muster.
- Bemerkenswert ist der zweite Lauf mit Magnitude Based Pruning, bei dem immerhin 127 Muster richtig erkannt wurden und mit 38 Verbindungen nur gut ein halbes Prozent der Gewichte im Netz übrig blieb.

η	Epochen		Fehler		gelöscht			richtig	CPU-Zeit	
	E	N	min.	akz.	L	I	H	Train	h:min:s	M
Backpropagation										
0,2	100	–	10	–	–	–	–	131	4:11	A
Löschen der kleinsten Gewichte										
0,2	100	80	20	25	6980	94	48	123	44:44	B
0,2	100	80	15	20	6982	93	47	127	46:49	B
Optimal Brain Damage										
0,2	100	80	20	25	6962	19	55	120	4:56:48	A
0,2	100	80	15	20	6789	3	55	90	5:20:29	A
Skeletonization mit Input Pruning										
0,2	100	80	20	25	5700	95	–	127	9:38	B
0,2	100	100	15	25	6060	101	–	126	17:08	B

Tabelle 9.3: Ergebnisse des Trainings mit akustischen Generatordaten

- Eine ebenso gute Leistung erreichte Skeletonization, das von den 112 Eingabeneuronen immerhin 101 löschen konnte.
- Optimal Brain Damage löschte – was in der Tabelle nicht zu erkennen ist – jeweils ein Ausgabeneuron, woraufhin eine Fehlerklasse nicht mehr erkannt werden kann.

9.3.3 Akustische Fehlerdiagnose von Startern

Weiterhin wurden 183 Starter analysiert, von denen jeweils 15 Merkmale aufgezeichnet waren. Die Bewertungen erfolgten nach der Subjektiven Geräusch-Nachbeurteilung auf einer Notenskala von 1 (gut) bis 5 (schlecht). Eine Testmenge war nicht gegeben. Das Netz hatte vor dem Ausdünnen zwei Schichten mit je zwanzig verdeckten Neuronen und insgesamt 800 Verbindungen. Die Ergebnisse sind in Tabelle 9.4 zu sehen.

Zu den Ergebnissen läßt sich folgendes sagen:

- Das Löschen der kleinsten Gewichte schneidet erstaunlich gut ab und überflügelt Optimal Brain Damage.
- Skeletonization konnte von den 15 Eingabewerten immerhin vier bzw. fünf löschen. Die verdeckten Neuronen konnte dagegen kein Verfahren nennenswert dezimieren. Das mag daran liegen, daß das Netz relativ viele Muster

η	Epochen		Fehler		gelöscht			richtig	CPU-Zeit	
	E	N	min.	akz.	L	I	H	Train	h:min:s	M
Backpropagation										
2,0	500	–	5	–	–	–	–	151	2:51	A
2,0	1000	–	5	–	–	–	–	165	8:45	A
Löschen der kleinsten Gewichte										
2,0	500	250	5	15	487	1	3	139	1:29:38	A
2,0	1000	400	5	10	477	1	3	154	1:00:08	B
Optimal Brain Damage										
2,0	500	250	5	15	384	1	4	156	1:24:59	A
2,0	1000	400	5	10	379	1	4	164	1:28:07	B
Skeletonization										
2,0	500	250	5	15	298	5	8	151	10:55	B
2,0	1000	400	5	10	130	4	2	170	9:26	B

Tabelle 9.4: Ergebnisse für die akustischen Daten für Starter

durch relativ wenige Eingaben klassifizieren muß. Die zwei Lagen verdeckter Neuronen erwiesen sich schon bei Vorversuchen mit Backpropagation als angemessene Architektur.

- Interessanterweise hat ein Lauf von Skeletonization fünf Muster mehr erkannt als Backpropagation. Auch sonst war die Klassifizierungsleistung von Optimal Brain Damage und Skeletonization genauso gut wie die des einfachen Backpropagation. Magnitude Based Pruning lieferte in dieser Hinsicht leider kein so gutes Ergebnis.

9.3.4 Akustische Fehlerdiagnose von E-Getriebemotoren

Die dritte Beispielmenge lieferte 12 Merkmale für 89 E-Getriebemotoren, die in vier Klassen eingeordnet wurden. Hier gab es eine Aufteilung in eine Trainingsmenge mit 45 und eine Testmenge mit 44 Beispielsätzen. Ich verwendete ein Netz mit der Größe $12 \times 20 \times 4$ mit 320 Verbindungen. Die Ergebnisse stehen in Tabelle 9.5.

Aus diesen Daten konnte man folgende Schlüsse ziehen:

- Wieder konnte Magnitude Based Pruning besser ausdünnen als Optimal Brain Damage. Skeletonization lieferte bei diesem Beispiel enttäuschende Ergebnisse; dieses Verfahren konnte nur bei der Anzahl der gelöschten Eingabeneuronen mit dem Löschen der kleinsten Gewichte mithalten.

η	Epochen		Fehler		gelöscht			richtig		CPU-Zeit	
	E	N	min.	akz.	L	I	H	Train	Test	h:min:s	M
Backpropagation											
1,0	100	–	3	–	–	–	–	43	34	0:04	A
Löschen der kleinsten Gewichte											
1,0	100	50	3	1	301	5	15	43	35	5:26	A
1,0	200	100	10	2	307	7	16	43	36	0:39	A
Optimal Brain Damage											
1,0	100	50	3	1	137	–	–	44	32	3:38	A
1,0	200	100	10	2	292	–	17	44	34	0:45	A
1,0	200	100	5	2	292	–	17	44	34	0:45	A
Skeletonization											
0,2	2000	1000	2	1	133	5	3	44	29	2:11	B
0,2	2000	1000	10	2	158	7	2	41	25	2:08	B

Tabelle 9.5: Ergebnisse für E-Getriebemotoren

- Erfreulich ist hier die gestiegene Generalisierungsleistung bei Magnitude Based Pruning. Optimal Brain Damage zog mit dem einfachen Backpropagation gleich.
- Die Leistung bei der Klassifizierung der Trainingsmuster war bei allen außer einem Testlauf gleich oder sogar besser als beim normalen Training ohne Ausdünnen.

10. Bewertung der Sensitivitätsverfahren

Aus meinen Testergebnissen und Aussagen in der zitierten Literatur ziehe ich folgende Schlüsse für die Ausdünnungsverfahren:

10.1 Bewertung der Verfahren im Einzelnen

Optimal Brain Surgeon

Dies ist ein extrem zeitaufwendiges Verfahren mit hohem Speicherplatzbedarf. Vor jedem Löschen einer Verbindung wird die inverse Hesse-Matrix berechnet, eine quadratische Matrix, deren Kantenlänge der Anzahl der Gewichte entspricht. Der Speicherplatz liegt also schon durch diese Datenstruktur in der Ordnung

$$O(n_w^2) \tag{10.1}$$

mit n_w als Anzahl der Gewichte. Kaum eine Workstation kann also Netze mit 10.000 Verbindungen berechnen.

Eine größere Einschränkung bringt aber vermutlich die Zeitkomplexität. Zur Berechnung der Hesse-Matrix werden zunächst Ableitungsvektoren berechnet, einer pro Muster und Ausgabeneuron. Die Dimension dieses Vektors entspricht der Anzahl der Gewichte im Netz, der Aufwand für seine Berechnung beläuft sich ebenfalls auf diesen Betrag. Alle diese Vektoren werden in einem iterativen Verfahren zur inversen Hesse-Matrix verarbeitet, ein Schritt dieses Algorithmus dauert $O(n_w^2)$ Schritte. Damit liegt die Zeitkomplexität der Berechnung der Hesse-Matrix insgesamt in der Ordnung

$$O(n_p \times n_o \times n_w^2) \tag{10.2}$$

wobei n_p die Anzahl der Muster und n_o die Anzahl der Ausgabeneuronen bezeichnet.

Bei dem einfachen Letters-Netz mit 26 Mustern, 26 Ausgabeneuronen und 610 Verbindungen dauerten die Trainings bei ca. 500 ausgedünnten Verbindungen zwischen 27 und 43 CPU-Stunden. Aufgrund dieser langen Laufzeit und des großen Speicherplatzbedarfs verzichtete ich bei den übrigen Testläufen auf dieses Verfahren.

Man muß allerdings anerkennen, daß OBS beim Letters-Netz das Verfahren mit der besten Ausdünnung war, wenn es auch nur ca. 1% mehr löschen konnte als andere Verfahren.

Optimal Brain Damage

Ich will Optimal Brain Damage vor allem mit Magnitude Based Pruning vergleichen, da dieses die einzigen durchgängig getesteten Verfahren zum Löschen von Gewichten sind.

Optimal Brain Damage hatte in manchen Testläufen ähnlichen Zeitaufwand wie das Löschen der betragsmäßig kleinsten Gewichte, war bei den komplexeren Beispielen jedoch bis zu siebenmal langsamer. Seine Zeitkomplexität liegt in der Ordnung

$$O(n_p \times n_w) \tag{10.3}$$

Optimal Brain Damage löschte in allen Fällen weniger Verbindungen als das Löschen der betragsmäßig kleinsten Gewichte. In der Klassifizierungsleistung ergibt sich im Vergleich der beiden Verfahren keine eindeutige Tendenz.

Löschen der betragsmäßig kleinsten Gewichte

Dieses Verfahren ist das einfachste Verfahren und liefert trotzdem erstaunlich gute Ergebnisse. Es liegt in der Zeitkomplexitätsklasse

$$O(n_w) \tag{10.4}$$

pro Ausdünnung und ist damit theoretisch der schnellste Algorithmus zum Ausdünnen von Gewichten.

Es klassifiziert außerdem in den meisten Fällen die Trainings- und Testmuster am besten.

Skeletonization

Das einzige Verfahren für Node Pruning benötigt pro Ausdünnen

$$O(n_w) \tag{10.5}$$

Schritte. Es verbrauchte oft weniger CPU-Zeit als das Löschen der betragsmäßig kleinsten Gewichte, da es die gleiche Anzahl Gewichte oder Neuronen in weniger Durchläufen löschen kann.

Bis auf ein Beispiel konnte es mehr oder gleich viele Neuronen löschen wie OBD oder Magnitude Based Pruning und hat damit durchaus seine Berechtigung als Verfahren zur Merkmalsextraktion. In der Klassifizierungsleistung lag es meist zwischen Magnitude Based Pruning und Optimal Brain Damage, seine Generalisierung war jedoch leider unbrauchbar.

Möglicherweise lag das in den vorliegenden Beispielen daran, daß Skeletonization zu viele Eingabeneuronen löschte, sodaß die entscheidenden Eingabewerte der Testmuster gar nicht mehr erkannt werden konnte. Beispielsweise wurden bei den Kreßel-Ziffern 255 Eingabewerte auf 50 reduziert, woraufhin das Netz nur noch etwa drei Viertel der Testmuster erkannte (vor der Ausdünnung waren dies 93%).

10.2 Gegenüberstellung der Verfahren

Optimal Brain Surgeon ist meiner Meinung nach zwar in seiner Wirkung ein gutes Ausdünnungsverfahren, aber wegen seines hohen Zeit- und Speicherplatzbedarfs für die meisten Anwendungen unbrauchbar. Es eignet sich vermutlich nur dann, wenn man aus dem Löschen weniger zusätzlicher Verbindungen einen so großen Vorteil ziehen kann, daß dies den hohen Aufwand rechtfertigt.

Die anderen Verfahren sind allesamt schneller als das Training mit Backpropagation – wenn es sich um größere Aufgaben handelt – und somit vom Zeitaufwand vernachlässigbar. Verantwortlich für den größten Teil der Rechenzeit ist also das Nachtraining. Hier kann es Probleme geben, wenn viele Elemente gelöscht werden und die Zeit für das Nachtraining zu lang ist.

Zu meiner Überraschung konnte sich Magnitude Based Pruning als einfachstes Verfahren mit den durch komplizierte mathematische Formeln begründeten Optimal Brain Damage und Skeletonization durchaus messen.

Jedoch ist das kein Grund, auf die anderen Verfahren zu verzichten. Die Erfahrungen mit den einzelnen Testbeispielen waren unterschiedlich. Jedes der Verfahren konnte gute Ergebnisse liefern. Bei einer Ausdünnung empfehle ich daher, immer mehrere Verfahren zu verwenden, damit man das beste Ergebnis auswählen kann.

Es ist möglicherweise angebracht, auf mehr Verfahren zurückzugreifen, als ich in dieser Diplomarbeit implementieren konnte. Einige interessante Verfahren habe ich in Kapitel 8 vorgestellt.

Auch für ein und dasselbe Verfahren erhielt ich beim gleichen Beispiel und ähnlichen Parametern manchmal recht unterschiedliche Ergebnisse. Daher ist es wichtig, für jedes Netz mehrere Trainingsläufe durchzuführen, möglichst unter wechselnden Parametern.

Offensichtliche Probleme der Sensitivitätsverfahren liegen zum einen bei der Berechnung der Saliency. Es wird nur die Wirkung der Entfernung *eines* Elements im Netz berechnet. Es ist unwahrscheinlich, daß eine schrittweise Löschung einzelner Elemente zu einer vergleichbaren Lösung führt wie eine (optimale, aber selbstverständlich viel zu aufwendige) Überprüfung jeder Teilmenge von Elementen eines gegebenen neuronalen Netzes. Problematisch ist außerdem, daß die Bestimmung der Relevanz nur vom Klassifizierungsverhalten der Trainingsmuster abhängt, nicht jedoch von einer Testmenge.

Zum anderen ist es nicht einfach, die richtigen Parameter für ein solches Verfahren zu finden. Soll tatsächlich in ein Minimum trainiert werden – wovon die Theorie der Verfahren ausgeht – oder schon am Punkt der maximalen Generalisierungsleistung gestoppt werden? Welches sind die richtigen Parameter, sodaß die gewünschte Trainingslänge auch nach mehreren Ausdünnungsläufen noch erreicht wird? Hier sind noch Heuristiken zu entwickeln.

Enttäuschenderweise konnte kein Verfahren die Generalisierungsleistung erheblich verbessern; Skeletonization verschlechterte sie sogar grundsätzlich. Sietsma und Dow ([Sietsma et al. 91]) mußten mit ihren Algorithmen ähnliche Erfahrungen machen. Sie stellten fest, daß ein Verrauschen der Eingabedaten in dieser Hinsicht viel bessere Erfolge erzielt. Grund für die gleichbleibende oder sogar sinkende Generalisierungsleistung war offensichtlich der Umstand, daß in einem zu großen Netz die überflüssigen Neuronen nur wenig zum Netzverhalten beitragen. In Netzen, die mit verrauschten Eingaben trainiert worden waren, wurden nach Angaben von Sietsma und Dow jedoch mehr verdeckte Neuronen gebraucht als in mit unveränderten Mustern trainierten Netzen. Das bedeutet, daß eine bessere Generalisierungsleistung in diesen Beispielen nicht durch eine kleinere Anzahl Neuronen erzeugt wurde, sondern offenbar sogar mit einer größeren Zahl benutzter Neuronen erreicht wird. Auch die Ergebnisse meiner Tests mit dem Ziffern-Netz – die Generalisierungsleistung brach ein, wenn besonders viele Elemente gelöscht wurden – läßt sich durch diese These erklären.

Den obigen Aussagen stehen natürlich die Forschungsergebnisse von Reed ([Reed 93]) und Krogh und Hertz ([Krogh et al. 92]) entgegen. Anscheinend läßt sich der Effekt mit dem bisherigen Kenntnisstand nicht eindeutig klären. Möglicherweise hat die Ausdünnung gleichzeitig verschiedene Einflüsse auf die Generalisierungsleistung, von denen sich je nach Beispiel die positiven oder die negativen durchsetzen.

Andere Erfahrungen machte ich mit der Klassifizierung der Trainingsmuster. Fast immer erkannten ausgedünnte Netze die Trainingsmuster gleich gut oder sogar besser als Netze ohne Ausdünnung.

Gute Ergebnisse konnten aber bei der Ausdünnung selbst und der Merkmalsextraktion erreicht werden. Oft genug blieben von einer anfangs für sinnvoll gehaltenen Architektur nur 10 bis 20 Prozent der Neuronen und Verbindungen übrig. Für diese Aufgabe scheinen sich Ausdünnungsverfahren besonders zu eignen.

Bei der Verwendung eines Ausdünnungsverfahrens sollte man daher auf zwei Gesichtspunkte achten. Einerseits müssen die Parameter so gut gewählt werden, daß ein beträchtlicher Anteil Verbindungen oder Neuronen gelöscht werden kann. Andererseits muß man darauf achten, daß die Generalisierungsleistung nicht allzusehr darunter leidet. Möglich wäre eine trainingsbegleitende Kontrolle des Generalisierungsfehlers, was sowohl zur Bestimmung des Trainingsendes bei jedem Nachtraining als auch zusätzlich zur Entscheidung über den Abbruch des ganzen Ausdünnungsverfahrens gute Dienste leisten könnte. Leider ist diese Möglichkeit im SNNS bisher nicht gegeben.

Literaturverzeichnis

[Attia et al. 93] F. Attia, M. Petry. Wie mit neuen Verfahren Geräusche besser zu beurteilen sind; in Bosch-Zünder 4/1993, Stuttgart 1993.

[Bronstein et al. 73] I. Bronstein, D. Semendjajew. Taschenbuch der Mathematik, 13. Auflage; Verlag Harri Deutsch, Zürich und Frankfurt am Main 1973.

[Chauvin 89] Y. Chauvin. A Backpropagation Algorithm with Optimal Use of Hidden Units; S. 519–526 in D. Touretzky (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 1, Morgan Kaufmann Publishers Inc., San Mateo 1989.

[Deco et al. 93] G. Deco, W. Finnoff und H.G. Zimmermann. Elimination of Overtraining by a Mutual Information Network; S. 744–749 in S. Gielen, B. Kappen (Hrsg.): Proceedings of the International Conference on Artificial Neural Networks (ICANN '93), Springer Verlag, London 1993.

[Fahlman 88] S. Fahlman. An empirical study of learning speed in back-propagation networks; in T. Sejnowski, G. Hinton, D. Touretzky (Hrsg.): Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann Publishers Inc., San Mateo 1988.

[Fahlman 91] S. Fahlman. The recurrent cascade-correlation architecture; S. 190–198 in R. Lippmann, J. Moody, D. Touretzky (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 3, Morgan Kaufmann Publishers Inc., San Mateo 1991.

[Hancock 92] J. Hancock. Pruning Neural Nets by Genetic Algorithm. S. 991–994 in I. Aleksander, J. Taylor (Hrsg.): Artificial Neural Networks 2, Bd. 2, Elsevier Science Publishers, Amsterdam 1992.

[Hanson et al. 89] S. Hanson, L. Pratt. Comparing Biases for Minimal Network Construction with Back-Propagation; S. 177–185 in D. Touretzky

(Hrsg.): Advances in Neural Information Processing Systems (NIPS) 1, Morgan Kaufmann Publishers Inc., San Mateo 1989.

[Hassibi et al. 93a] B. Hassibi, D. Stork. Second order derivatives for network pruning: Optimal Brain Surgeon; S. 164–171 in S. Hansen, J. Cowan, C. Giles (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 5, Morgan Kaufmann Publishers Inc., San Mateo 1993.

[Hassibi et al. 93b] B. Hassibi, D. Stork, G. Wolff. Optimal Brain Surgeon and General Network Pruning; S. 293–299 in Proceedings on the IEEE International Conference on Neural Networks, Vol. 1, IEEE Press, New York 1993.

[Hassibi et al. 94] B. Hassibi, D. Stork, G. Wolff, T. Watanabe. Optimal Brain Surgeon: Extensions and performance comparisons; erscheint in J. Cowan, G. Tesauro, J. Alspector (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 6, Morgan Kaufmann Publishers Inc., San Mateo 1994.

[Heine et al. 94] S. Heine, I. Neumann. Optimizing Load Forecast Models Using an Evolutionary Algorithm; erscheint im Tagungsband der Second European Congress on Intelligent Techniques and Soft Computing (EUFIT), Aachen 1994.

[Hertz et al. 91] J. Hertz, A. Krogh, R. Palmer. Introduction to the theory of neural computation; Addison-Wesley, Redwood City 1991.

[Heuser 83] H. Heuser. Lehrbuch der Analysis, Bd. 2, 2. Auflage; Teubner Verlag, Stuttgart 1983.

[Hinton et al. 93] G. Hinton, D. van Camp. Keeping Neural Networks Simple; S. 11–18 in S. Gielen, B. Kappen (Hrsg.): Proceeding of the International Conference on Artificial Neural Networks (ICANN '93), Springer Verlag, London 1993.

[Jolliffe 86] I. Jolliffe. Principal Component Analysis; Springer Verlag, New York 1986.

[Kailath 80] T. Kailath. Linear Systems; Prentice Hall, Englewood Cliffs 1980.

[Kandel et al. 91] E. Kandel, J. Schwartz, T. Jessell (Hrsg). Principles of neural science; Elsevier, New York 1991.

[Kreßel 92] U. Kreßel. BITEX–Bericht 1991; Daimler-Benz AG, Ulm 1992.

[Krogh et al. 92] A. Krogh und J. Hertz. A Simple Weight Decay Can Improve Generalisation; S. 950–958 in J. Moody, S. Hanson, R. Lippmann (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 4, Morgan Kaufmann Publishers Inc., San Mateo 1992.

[Kurokawa et al. 93] H. Kurokawa, C.-Y. Ho, S. Mori. A novel backpropagation algorithm with optimal number of hidden units; S. 783 in S. Gielen, B. Kappen (Hrsg.): Proceedings of the International Conference on Artificial Neural Networks (ICANN '93), Springer Verlag, London 1993.

[Le Cun et al. 90] Y. Le Cun, J. Denker, S. Solla. Optimal Brain Damage; S. 598–605 in D. Touretzky (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 2, Morgan Kaufmann Publishers Inc., San Mateo 1990.

[Levin et al. 94] A. Levin, T. Leen, J. Moody. Fast Pruning Using Principal Components; erscheint in J. Cowan, G. Tesauro, J. Alspector (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 6, Morgan Kaufmann Publishers Inc., San Mateo 1994.

[Lim et al. 94] S. Lim, S. Ho. Dynamic Creation of Hidden Units with selective Pruning in Backpropagation; S. 492–497 in Proceedings of the World Congress on Neural Networks 1994, San Diego 1994.

[McCormack et al. 91] J. McCormack, P. Asente, R. Swick. X Toolkit Intrinsics – C Language Interface; Massachusets Institute of Technology, Cambridge 1991.

[Mozer et al. 89] M. Mozer, P. Smolensky. Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment; S. 107–115 in D. Touretzky (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 1, Morgan Kaufmann Publishers Inc., San Mateo 1989.

[Rawlings 88] J. Rawlings. Applied Regression Analysis; Wadsworth & Brooks / Cole, Pacific Grove 1988.

[Reed 93] Russell Reed. Pruning Algorithms – A Survey; S. 740–747 in IEEE Transactions on Neural Networks, vol. 4, No. 5, New York 1993.

[Riedmiller et al. 92] M. Riedmiller, H. Braun. Rprop – A fast adaptive learning algorithm. Technical Report, University of Karlsruhe 1992.

[Rumelhart et al. 86] D. Rumelhart, J. McClelland. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations, The MIT Press, Boston 1986.

[Santini 92] S. Santini. The Bearable Lightness of Being: Reducing the Number of Weights in Backpropagation Networks; S. 139–142 in I. Aleksander, J. Taylor (Hrsg.): Artificial Neural Networks 2, Bd. 1, Elsevier Science Publishers, Amsterdam 1992.

[Sieler 93] A. Sieler. Topologieoptimierung neuronaler Netze für die Erhöhung der Genauigkeit der kurzfristigen Verbrauchsprognose eines Energieversorgungsunternehmens; Diplomarbeit am Fachbereich Elektrotechnik der Technischen Hochschule Leipzig 1993.

[Sietsma et al. 91] J. Sietsma, R. Dow. Creating Artificial Neural Networks That Generalize; S. 67–79 in Neural Networks, Vol. 4, No. 1, Pergamon Press, Oxford 1991.

[Spektrum 84] Spektrum der Wissenschaft: Gehirn und Nervensystem; Spektrum der Wissenschaft Verlag, Heidelberg 1984.

[Tollenaere 90] T. Tollenaere. SuperSAB: Fast Adaptive Backpropagation with Good Scaling Properties; S. 561–573 in Neural Networks, Vol. 3, Pergamon Press, Oxford 1990

[Tsaptsinos et al. 92] D. Tsaptsinos, A. Mirzai, J. Leigh. Matching the topology of a neural net to a particular problem: Preliminary results using correlation analysis as a pruning tool; S. 957–960 in I. Aleksander, J. Taylor (Hrsg.): Artificial Neural Networks 2, Bd. 2, Elsevier Science Publishers, Amsterdam 1992.

[Weigend et al. 91] A. Weigend, D. Rumelhart und B. Hubermann. Generalization by Weight-Elimination with Application to Forecasting; S. 875–882 in R. Lippmann, J. Moody, D. Touretzky (Hrsg.): Advances in Neural Information Processing Systems (NIPS) 3, Morgan Kaufmann Publishers Inc., San Mateo 1991.

[Werbos 88] P. Werbos. Backpropagation: Past and Future; S. 343–353 in Proceedings of the IEEE International Conference on Neural Networks, Vol. 1, IEEE Press, New York 1988.

[Whiteley et al. 90] D. Whiteley, T. Starkweather, C. Bogart. Genetic algorithms and neural networks: optimizing connections and connectivity; S. 347–361 in Parallel Computing Vol. 14, No. 3, 1990.

[Zell 94] A. Zell. Simulation Neuronaler Netze; Addison Wesley 1994.

[Zell et al. 94] A. Zell, G. Mamier, M. Vogt, N. Mache, R. Hübner, K.-U. Herrmann, T. Soyez, M. Schmalzl, T. Sommer, A. Hatzigeorgiou, S. Döring, D. Posselt. SNNS User Manual, Version 3.2; Report 6/94, Universität Stuttgart 1994.