Konstruktive neuronale Lernverfahren auf Parallelrechnern

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

vorgelegte

Dissertation

von

Lutz Prechelt

aus Bielefeld

Tag der mündlichen Prüfung:	15. Februar 1995
Erster Gutachter:	Prof. Walter F. Tichy
Zweiter Gutachter:	Prof. Alex Waibel

Danksagungen

Ich werde jedenfalls nicht auf der faulen Haut liegen.
Ich bin nämlich ein Sachensucher und da hat man
niemals eine freie Stunde. [...]
Die ganze Welt ist voll von Sachen,
und es ist wirklich nötig, daß jemand sie findet.
Und das gerade, das tun die Sachensucher.

Pippi Langstrumpf

Die Dissertation ist fertig. Uff! Zahlreiche Leute haben ihren Teil daran mitgeholfen. Zu allererst möchte ich mich bei meinen Eltern bedanken, die mir ermöglicht haben „Sachensucher" zu werden und mich nie in irgendeine Richtung drängen wollten. Mein Chef, Prof. Walter Tichy, hat mir beigebracht, wie man Wissenschaft macht. Er hat mich gelehrt, daß keine Panik angebracht ist, weil Forschung zunächst immer so unklar aussieht, und er hat für hervorragende Arbeitsbedingungen gesorgt. Die dritte *conditio sine qua non* war meine Freundin Sonja Gelhaus. Ohne ihre Nähe und Aufmunterung hätte ich vermutlich irgendwann aufgegeben oder einen Nervenzusammenbruch gehabt.

Dank geht auch an meine Kollegen vom Lehrstuhl: Rolf Adams, Ruth Ghafari, Stefan Hänßgen, Ernst Heinz, Heinz Herrmann, Christian Herter, James Hunt, Paul Lukowicz, Michael Philippsen und Thomas Warschko standen mir manchesmal mit Literatur, Information, Diskussion oder tatkräftiger Hilfe zur Seite und trugen wesentlich dazu bei, daß ich mich bei meiner Arbeit fast immer wohlgefühlt habe. Herauszuheben sind Heinz als Lebensgeist der Rechnerfunktion, Stefan als Softwareversorgungsautomat/X-Guru/subversiver Ablenker, Tom als TeX-Guru und MasPar-Superuser (nützlich!) und schließlich Michael als Hauptarbeitsfreund und Vorbild an Arbeitswut. Einige Kollegen von anderen Lehrstühlen waren auch immer hilfreiche Anlaufstellen, insbesondere Heinz Braun, Uwe Herzog, Tilo Sloboda, Monika Woszczyna und vor allem Michael Finke. Vielen Dank ferner an die Leute vom Eli-Team und von MasPar für stets sehr prompte Hilfe in Zeiten der Not, an Jordan Pollack für *neuroprose*, an Lothar Späth (stellvertretend) für das schöne neue Gebäude und an Prof. Dillmann und meinen Zweitbetreuer Prof. Alex Waibel für hilfreiche Gespräche.

Nicht zu vernachlässigen sind meine Freunde, die mir zwischendurch immer wieder gezeigt haben, daß es auch noch was anderes gibt als paraneuronetzdatenlokallastbalancierungsauswertungsvorbereitung: Dirk Fox, der mich zum FWS brachte, Olaf Löb, der Musikfreund, Susanne May, mein bester Freund, Daniela Müller, die Oberbetriebsnudel, Dörte Neundorf, der gute Geist $\forall X : X \in \{Abendessen, Gespräch, Fröhlichkeit\}$, Ute St. Clair, mein *pen pal*, Ursi Trexler, die Andere-Welt-Freundin, und noch ein paar, zu denen mir kein schöner Titel einfällt: Michael und Margit Fiegert, Bärbel Groß, Uwe Herzog, Gudula Kiehle, John Maraist, Holger Müller und Rose Sturm.

Für das kritische Lesen und Kommentieren meiner Dissertation danke ich Christian, Ernst, Paul, Sonja, meiner Schwester Uta, Walter und am allermeisten Michael. Ach ja: Dank auch an alle anderen, die ihn verdienen, die ich aber schändlicherweise hier vergessen habe!

Inhaltsverzeichnis

Zusammenfassung

Beim praktischen Einsatz neuronaler Lernverfahren zur Lösung statischer Diagnoseaufgaben mittels vorwärtsgerichteteter neuronaler Netze gibt es zwei Hauptprobleme: Erstens ist es schwierig, gute Netztopologien und gute Parameter für die Lernverfahren zu finden. Zweitens erfordert die Ausführung der Lernverfahren viel Rechenzeit. Zur Lösung dieser Probleme gibt es folgende Hauptstoßrichtungen:

- Zur Verringerung des Parametersuchaufwandes sollten *automatische Lernverfahren* entwickelt werden, d.h. solche, die auf Eingriffe des menschlichen Benutzers verzichten. Insbesondere können *konstruktive Lernverfahren* eine günstige Topologie für das Netz innerhalb abgesteckter Grenzen automatisch finden.
- Zur Verringerung des Zeitbedarfs bei der Benutzung solcher Verfahren sollten Parallelrechner eingesetzt werden. Diese werden idealerweise maschinenunabhängig in einer Hochsprache programmiert.

Die vorliegende zweiteilige Arbeit kombiniert beide Arbeitsrichtungen miteinander: Im ersten Teil werden einige Ansätze von automatischen Lernverfahren vorgestellt, untersucht und bewertet. Insbesondere werden konstruktive Algorithmen betrachtet, also solche, die die Topologie des neuronalen Netzes während des Trainings verändern. Eine umfangreiche Literaturübersicht belegt, warum dies eine sinnvolle Arbeitsrichtung ist. Ich untersuche zwei vielversprechende Klassen konstruktiver Verfahren, nämlich (subtraktive) Beschneidungsverfahren und (additive) Kandidatenlernverfahren. Aus jeder Klasse werden existierende Verfahren ausgewählt, Verbesserungen entwickelt und eine gründliche experimentelle Leistungsbewertung vorgenommen. Ich weise quantitativ nach, daß solche empirischen Auswertungen bislang vernachlässigt wurden.

Im zweiten Teil werden Optimierungen für Übersetzer vorgestellt, mit denen eine Programmiersprache zur Beschreibung neuronaler Lernverfahren effizient für massiv parallele Rechner verschiedener Bauart übersetzt werden kann. Im Gegensatz zu den bisher existierenden Implementationen von neuronalen Lernverfahren auf massiv parallelen Maschinen erlaubt der vorgestellte Ansatz, auch solche Verfahren effizient zu simulieren, bei denen die Netze durch dynamische Topologieveränderungen eine unregelmäßige Struktur erhalten. Der Übersetzer stellt eine Verteilung der Daten und der Berechnungen her, die zugleich annähernd optimale Prozeß- und Datenlokalität und annähernd optimale Lastbalance sichert. Damit zeige ich für ein eingeschränktes Anwendungsgebiet Techniken für die übersetzerbasierte Behandlung dynamischer unregelmäßiger Probleme auf Parallelrechnern, die bislang nicht befriedigend geleistet werden kann. Ein Übersetzer für die MasPar MP-1 wird vorgestellt; die durch die Optimierungen erzielten Verbesserungen werden gemessen und bewertet.

Die Auswertungen in beiden Teilen der Arbeit werden mit Hilfe einer eigens erstellten Sammlung von 12 Benchmark-Problemen vorgenommen. Alle diese Probleme sind Lernaufgaben mit realen Daten aus den Bereichen Medizin, Biologie, Bauingenieurwesen, Astronomie, Chemie, Landwirtschaft und Finanzwirtschaft. Diese Benchmarks füllen eine Lücke, denn bisher gibt es speziell für neuronale Verfahren keine einheitliche Sammlung von Benchmark-Problemen, die „echte“ Daten verwenden, dabei jedoch sowohl allgemein zugänglich als auch so genau definiert sind, daß sie exakt vergleichbare und reproduzierbare Ergebnisse zulassen.

Kapitel 1

Einführung

Should you understand this article,
please contact me.
I shall gladly explain it until you don't.
Ein Benutzer in Usenet NEWS

Writing is difficult.
[...] It is difficult
because it requires an assortment of activities
generally classified as thinking.
Ernst Jacobi

Die ersten vier Abschnitte dieses Kapitels stellen in knapper Form die Geschichte des Maschinenwesens, der Informatik, der Neuroinformatik und der Parallelrechnerei dar. Dabei lassen sich gewisse Entwicklungsphasen identifizieren, die in allen vier Gebieten wiederkehren. Zweck dieser Darstellung ist die Verdeutlichung der Tatsache, daß die in dieser Arbeit berührten Gebiete, nämlich Neuroinformatik und Parallelrechnerei, noch sehr unreif sind und wir deshalb bei der Forschung in diesen Gebieten noch an der Beherrschung recht grundlegender Probleme zu arbeiten haben. Der fünfte Abschnitt stellt daran anknüpfend den Aufbau und die Ziele der Arbeit vor.

1.1 Eine kurze Geschichte des Maschinenwesens

Es ist nicht genau bekannt, welches die erste von Menschen benutzte mehrteilige Maschine war. Der wahrscheinlichste Kandidat ist wohl die mechanische Tierfalle, die spätestens um 15000 v. Chr., also gegen Ende der Altsteinzeit, bekannt war. Hierbei werden Hebel so eingesetzt, daß ein Tier eine Reihe schräg aufgestellter Baumstämme zum Umfallen bringt, die es dann unter sich begraben [268, Seite 15]. Später, in der Jungsteinzeit (ca. 10000 bis 3000 v. Chr.), folgt dann der Bogen zum Verschießen von Pfeilen, der es erlaubt, ein Geschoß mit wesentlich höherer Geschwindigkeit auf den Weg zu bringen, als es dem Arm allein möglich wäre. Eine andere Erfindung dieser Epoche ist der Fidelbohrer, mit dessen Hilfe sich in tagelanger Arbeit Löcher in Steine bohren lassen, um bessere Werkzeuge herzustellen; dies ist zugleich die erste Werkzeugmaschine [268, Seite 19]. Eine Abart des Fidelbohrers dient zum bequemen und sicheren Entzünden von Feuer. Weitere Meilensteine der frühen Entwicklung von Maschinen sind der Räderkarren und die später daraus abgeleitete Töpferscheibe [268, Seite 18–19]. Man könnte diese Phase die *primitive Phase* des Maschinenwesens nennen.

Für mehrere Jahrtausende bleibt die Entwicklung von Maschinen nun in diesem Stadium: Maschinen sind im wesentlichen Werkzeuge wie andere auch; sie erlauben es, eine Arbeit zu erleichtern und die Kräfte des Körpers effektiver einzusetzen. Erst in den frühen Hochkulturen werden dann Maschinen entwickelt, die dem Menschen eine Arbeit *vollständig* abnehmen, anstatt sie nur zu erleichtern: das Wasserrad und das Windrad. Beides wird vermutlich etwa 300 bis 100 v. Chr. von den Griechen erfunden [147, Seiten 63 und 66] und zuerst nur als Spielzeug betrachtet. Erst allmählich nimmt im Laufe der Jahrhunderte die nutzbringende Anwendung größeren Umfang an [268, Seite 57–59], um im Mittelalter ihren Höhepunkt zu erreichen [268, Seite 74]. Dies ist die *Phase der Nutzbarmachung*.

Die dritte Epoche des Maschinenwesens beginnt in der Barockzeit. Ab dem Ende des 16. Jahrhunderts erfolgt ein starkes Aufblühen der Wissenschaften, zum Beispiel der Astronomie (Geräte von Tycho Brahe und Johannes Kepler, Beobachtungen und Modelle von Johannes Kepler und Galileo Galilei) und der Mathematik (Buchstabenrechnung von François Viète, Logarithmen von Jost Bürgi und John Napier). Aus den neuen Erkenntnissen und der Begeisterung für das gesetzmäßige Erfassen der Ordnung der Welt entwickelt sich ein mechanistisches Weltbild, das sich auch in der Ausrichtung des Maschinenwesens niederschlägt. Es entsteht der Traum von der intelligenten Maschine, ja gar der universellen Maschine. Als Versuch der Umsetzung dieses Traums werden verblüffende, wenngleich nutzlose Automaten gebaut. Diese *Androiden* machen fast natürlich wirkende Bewegungen für irgendeine sehr spezielle Tätigkeit und vermitteln die Illusion einer Nachbildung menschlicher Fähigkeiten. Beispiele sind die Musikerin von Pierre und Henri Droz [139, Seite 45–49], die, lebensecht bis hin zu den Augenbewegungen, ein Musikstück auf einer Orgel spielt (1790), oder, obgleich kein Android, die berühmte mechanische Ente von de Vaucanson [367, Seite 225], die aus mehr als 1000 Teilen besteht und angeblich schnattern, fressen, gehen und sogar verdauen kann (um 1738). Auch Betrug wird begangen beim Versuch, Androiden fortzuentwickeln; das wohl bekannteste und erfolgreichste Beispiel hierfür ist der Schachspieler des Wolfgang von Kempelen [147, Seite 147]. Der Apparat besteht aus einem großen Kasten mit dem Schachbrett auf der Oberseite, vor dem eine als Türke gekleidete Puppe sitzt. Die Puppe bewegt sich, raucht in den Spielpausen eine Pfeife und kann in mehreren Sprachen „Schach!“ bieten. Das wichtigste jedoch: die Figur spielt Schach — und zwar recht gut (1769). Die Zeitgenossen Kempelens an den Fürstenhöfen, denen dieser Automat vorgeführt wird, sind begeistert. Eine intelligente Maschine! Beim Öffnen des Kastens werden komplizierte Räder und Gestänge sichtbar, jedoch nicht der im Innern versteckte menschliche Schachspieler. Diese Zeit ist die *Phase der Verklärung*.

Die *industrielle Revolution* (ab ca. 1750) und die nachfolgende Elektrifizierung bilden die vierte Phase des Maschinenwesens. Die Idee von der universellen Maschine verschwindet. Sie wird ersetzt durch Maschinen, die sehr spezialisiert sind, dafür aber einen hohen praktischen Nutzen haben. Maschinen bekommen jetzt erstmals beherrschende Bedeutung für das Leben eines großen Teils der Menschen und bewirken durch ihren systematischen Einsatz eine atemberaubende Produktivitätserhöhung. Die kommerziellen Interessen bei der Nutzung von Maschinen treiben zugleich die wissenschaftliche Erforschung der Grundlagen der Technik voran, so daß die industrielle Revolution in ihrem Verlauf auch zu einer Ausweitung der Forschung und zu einer zunehmend schnelleren Vergrößerung des Wissens führt.

Die vorläufig letzte Phase des Maschinenwesens könnte man die *Mechanisierung geistiger Arbeiten* nennen, was in diesem Zusammenhang die elektronische Datenverarbeitung bezeichnet. Sie hat zu einer Erweiterung des Maschinenbegriffs geführt. Das Lexikon [100] definiert *Maschine* als „jede Vorrichtung zur Erzeugung oder Übertragung von Kräften, die nutzbare Arbeit leistet (Arbeitsmaschine) oder die eine Energieform in eine andere umwandelt (Kraftmaschine).“ Seit der Erfindung der digitalen elektronischen Computer (ab ca. 1940) wird der Maschinenbegriff auch auf die Verarbeitung von Informationen erweitert, bei der physikalische Arbeit nur als unerwünschter Nebeneffekt eine Rolle spielt.

Zusammenfassend können wir also fünf Hauptphasen in der Entwicklung des Maschinenwesens erkennen:

1. Die primitive Phase,
2. die Phase der Nutzbarmachung,
3. die Phase der Verklärung,

4. die industrielle Revolution und
5. die Mechanisierung geistiger Arbeiten.

Es sei bemerkt, daß diese Phasen natürlich nicht streng nacheinander ablaufen und für jede einzelne kein klarer Anfangs- oder Endzeitpunkt angegeben werden kann.

Es ist bemerkenswert, wie sich diese Entwicklungsphasen des Maschinenwesens in der Entwicklung der Computertechnik und Informatik wiederfinden lassen, wenngleich stärker überlappt und miteinander verwoben als in der Geschichte des Maschinenwesens. Der nächste Abschnitt zeichnet die Phasen in der Informatik (einschließlich der Computertechnik) nach.

1.2 Eine kurze Geschichte der Informatik

In den Vorphasen der Computertechnik ab dem 17. Jahrhundert gibt es zwar Rechenmaschinen zur Ausführung einzelner arithmetischer Operationen, wie die von Schickard [139, Seiten 120-125] etwa 1623, von Pascal [139, Seiten 125-128] etwa 1674, oder von Leibniz [139, Seiten 128-134] ab 1673, und auch Automaten, die von einer veränderlichen Kommandofolge gesteuert werden (Sequenzautomaten), wie die oben erwähnten Androiden oder die Webstühle von Vaucanson [367, Seite 223] um 1745 und Jacquard [147, Seite 174] ab etwa 1805. Die Rechenmaschinen sind aber aufgrund mechanischer Schwierigkeiten sehr unzuverlässig und weder Rechenmaschinen noch Sequenzautomaten haben eine Programmsteuerung im heutigen Sinne, also mit Programmverzweigung aufgrund von Fallunterscheidung und mit Iteration. Der erste Entwurf eines frei programmierbaren Computers ist die *analytical engine* von Charles Babbage [139, Seite 161-195] ab etwa 1838. Die Maschine ist ihrer Zeit weit genug voraus, um nie gebaut zu werden — der mechanische Aufwand ist zu hoch[1].

Aus diesem Grund beginnt die primitive Phase der Computertechnik erst im 20. Jahrhundert. Der erste funktionstüchtige frei programmierbare Rechner (elektromechanisch, mit externem Programm) ist Konrad Zuses Z3 im Jahre 1941 [139, Seiten 341-361]. Ein Jahr darauf folgt mit dem Atanasoff-Berry-Computer der erste elektronische (jedoch nicht programmierbare) Rechner [139, Seiten 473-475] (siehe auch [62]). Ein weiteres Jahr danach, 1943, folgt mit dem COLOSSUS der erste einigermaßen frei programmierbare elektronische Rechner [139, Seiten 454-465], der dem zwar viel berühmteren, aber auch nicht wesentlich vielseitigeren ENIAC [139, Seiten 467-472] (siehe auch [61]) von 1946 seinen Neuigkeitswert nimmt. Die ersten Computer im heutigen Sinne, nämlich *speicher*programmierte Rechner, sind 1948 mit Einschränkungen der SSEC [139, 493-497] und 1949 der EDVAC [139, Seiten 487-492][2]. Noch ein paar andere Gruppen entwerfen und bauen zu etwa derselben Zeit speicherprogrammierbare Rechner; da sich meist kein genauer Zeitpunkt angeben läßt, wann eine solche Maschine funktionstüchtig wird, ist die Reihenfolge der Fertigstellung strittig. Wie auch immer, alle diese frühen Rechner sind enorm teuer und unzuverlässig und außerordentlich schwierig zu handhaben.

Die Phase der Nutzbarmachung wird trotzdem schon in den 50er Jahren betreten. Computer übernehmen einfache menschliche Geistestätigkeiten, zum Beispiel Finanzbuchhaltung, Lagerverwaltung, Rechnungsstellung und ähnliches im wirtschaftlichen Bereich sowie numerische Berechnungen verschiedener Art im technisch-wissenschaftlichen Bereich. Die Bedeutung der Computer in diesen Bereichen steigt, von den zunächst (aufgrund der enormen Kosten der Hardware) bescheidenen Anfängen ausgehend, in den nächsten zwei Jahrzehnten steil an. Diese Entwicklung wird unterstützt von Fortschritten auf der Softwareseite: Zuerst machen die höheren Programmiersprachen FORTRAN und COBOL die aufwendige Programmierung in Maschinensprache überflüssig und erlauben es, Programme weitgehend unverändert auf die jeweils nächste der schnell wechselnden Hardwareplattformen zu

[1] Moderne Technik erlaubt die Fertigung dennoch: In einem Projekt am britischen Museum wird derzeit an einem Exemplar gebaut.

[2] Dem EDVAC verdanken wir übrigens die Bezeichnung „*von Neumann*-Computer“, weil von Neumann (fälschlicherweise alleine) der Autor des ersten Entwurfsberichts über den EDVAC ist [384], in dem die Speicherprogrammierbarkeit und die sequentielle Arbeitsweise besonders herausgestellt werden.

übernehmen. Dann machen die Einführung des Mehrprogrammbetriebs und des interaktiven Betriebs die Benutzung der Rechner einfacher und erhöhen die Zugänglichkeit.

Die Erfindung des Mikroprozessors durch Intel im Jahr 1974 legt die Basis für die industrielle Revolution der Informatik. Diese beginnt 1981 mit der Vorstellung des IBM PC. Damit beginnt ein explosionsartiges Wachstum des Computereinsatzes, das bis heute anhält. Ähnlich wie in der industriellen Revolution im Maschinenwesen erlaubt auch hier die Massenfertigung und die Zunahme der Leistungsfähigkeit sowohl eine große Steigerung der Produktivität als auch die Herstellung grundsätzlich neuer „Produkte“: Es werden dem Computer Anwendungen erschlossen, an die zuvor niemand ernsthaft gedacht hatte. Die dabei auftretenden Probleme, vor allem auf der Softwareseite, haben viele Ähnlichkeiten zur industriellen Revolution des Maschinenwesens; siehe dazu etwa den Aufsatz [215] von Nancy Leveson, ein Vergleich der Geschichte der Hochdruckdampfmaschinen mit der Softwaretechnik.

Die Phase der Verklärung und die Mechanisierung geistiger Arbeiten (hier nun: Schaffung künstlicher Intelligenz) sind in der Informatik bisher nicht klar voneinander zu trennen. Seit Beginn der Informatik gibt es Versuche, mit Computern Dinge zu tun, die ein außenstehender Betrachter als intelligente Leistungen (im Gegensatz zu bloßem Rechnen) einstuft. Die Frage, ob die Ergebnisse dieser Arbeiten tatsächlich als „künstliche Intelligenz“ (KI) zu bezeichnen seien, hängt von der Definition von Intelligenz ab — jedoch ist bislang keine Definition bekannt, die sich für die Informatik als brauchbar erweist; der berühmteste Ansatz, der sogenannte Turing-Test, hat sich als zu naiv herausgestellt, da schon simple Programme wie Weizenbaums ELIZA ihn zumindest beinahe bestehen können [238, Seiten 225-226 und 251-256]. Der lange um die KI geführte Grundsatzstreit hat sich in eine Art Waffenstillstand beruhigt, seit zumindest einige Dinge klar sind: Erstens hat die Disziplin KI durchaus eine Reihe von leistungsfähigen Anwendungsprogrammen hervorgebracht [108, 109]; zweitens sind jedoch bislang alle diese Programme ausgeprägte Fachidioten [138]; und drittens sträuben sich einige Bereiche der menschlichen Fähigkeiten, insbesondere die Wahrnehmung (Sehen, Hören) noch sehr gegen eine leistungsfähige Nachahmung auf dem Computer. Viele Forscher nehmen inzwischen den pragmatischen Standpunkt ein „Es ist mir egal, ob jemand es intelligent nennt oder nicht; Hauptsache, es ist nützlich.“ Mit dieser Haltung wird der geistes- und naturwissenschaftliche Grundsatzstreit zugunsten einer eher ingenieurmäßigen Herangehensweise zurückgestellt. Dieser Standpunkt erkennt an, daß die Fragen, was KI sei und ob sie möglich ist, noch nicht sinnvoll beantwortet werden können; ich werde den Standpunkt deshalb im weiteren übernehmen.

In den folgenden Abschnitten betrachten wir nun die beiden in dieser Arbeit relevanten Teilbereiche der Informatik: Die Lehre von den künstlichen neuronalen Netzen (*Neuroinformatik*) und die Informatik der parallelen Hard- und Software, hier der Kürze halber *Parallelrechnerei* genannt. Eine analoge geschichtliche Betrachtung anhand der oben eingeführten Phasen ergibt, daß sich beide Bereiche noch in ihren Anfängen befinden.

1.3 Eine kurze Geschichte der Neuroinformatik

Die Ursprünge der Neuroinformatik sind die Arbeiten von Warren McCulloch und Walter Pitts (1943) und von Donald Hebb (1949). McCulloch und Pitts beschreiben in ihrem Aufsatz [239] eine mathematisch formalisierte Version eines Neurons, das Schwellwertneuron: Die Ausgabe des Neurons ist 1, wenn die gewichtete Summe der Eingaben den Schwellwert des Neurons überschreitet, und 0 andernfalls. McCulloch und Pitts zeigen, daß Netze aus solchen Neuronen beliebige boolesche Funktionen berechnen können. Diesem diskreten Ansatz stehen andere gegenüber, die ein Kontinuum betrachten und auf Differentialgleichungen basieren, um Aussagen über globales Verhalten zu machen; am bekanntesten sind die Arbeiten von Wiener [397].

Auf diesen theoretischen Grundlagen baut die primitive Phase der Neuroinformatik auf, die etwa Mitte der 50er Jahre beginnt. Ausdrücklich als Versuch zur Modellierung des Gehirns wird eine Reihe von Varianten einer Klasse von Maschinen für optische Wahrnehmung untersucht, die *Perceptrons* (eingeführt in [303]). Ein Perceptron berechnet im einfachsten Fall aus einer festen Menge von lokalen Prädikaten über begrenzte Teilbereiche einer (gedachten) Netzhaut ein globales Prädikat über das auf der Netzhaut dargestellte Bild, indem eine gewichtete Summe der Einzelprädikate (mit Wert 0 oder 1) gegen einen Schwellwert geprüft wird. Zur Bestimmung der Gewichte dient ein einfacher Algorithmus namens *Perceptron-Lernregel*, der auf eigentlich schon früher gefundenen mathematischen Grundlagen aufbaut, z.B. [2], die jedoch unter den Neuroinformatikern erst einige Zeit später bekannt werden.

Mehrere Gruppen befassen sich in den folgenden Jahren mit Perceptrons (vor allem die Gruppe um Frank Rosenblatt) oder perceptron-ähnlichen Apparaten (recht bekannt wird hiervon z.B. [310]). Einen ersten Höhepunkt erreicht die Neuroinformatik mit dem 1962 veröffentlichten Buch von Rosenblatt [304]. Hierin wird neben einer ausführlichen experimentellen Analyse der Fähigkeiten von Perceptrons auch ein Beweis für die Konvergenz der Perceptron-Lernregel gegeben. Dieser Beweis besagt, daß die Lernregel für jedes von einem Perceptron darstellbare Prädikat auch eine Lösung findet. Der Beweis wurde vermutlich zuerst in dem wenig bekanntgewordenen Papier [273] geführt. Rosenblatt ist ein begeisterter und begeisternder Advokat des Perceptrons. Ein Kollege sagt später über ihn: „He was a press agent's dream, a real medicine man." [238, Seite 87]. Nicht zuletzt diese Ausstrahlung führt dazu, daß sich mit dem Erscheinen von [304] zahlreiche Forscher für das Perceptron begeistern und eine Welle neuer Arbeiten einsetzt, die allerdings kaum Erfolge hervorbringen. Insbesondere wird keine Lernregel für das mehrstufige Perceptron gefunden, von dem man weiß, daß es im Prinzip leistungsfähiger als das normale (einstufige) ist.

Die Begeisterung kommt zu einem abrupten Ende, als 1969 Minsky und Papert ihr berühmtes Buch „Perceptrons" veröffentlichen [248]. Sie liefern darin die Theorie, mit deren Hilfe sich unter anderem die bis dahin nur phänomenologisch bekannte Tatsache verstehen läßt, daß (einstufige) Perceptrons manche Probleme nicht lösen können. Die berühmteste Aussage dieser Theorie lautet, daß ein Perceptron der Ordnung 1 (das für jedes lokale Prädikat nur 1 Punkt der Netzhaut auswertet) die Paritätsfunktion nicht berechnen kann. Die Parität ist das Prädikat, welches angibt, ob die Anzahl der elementaren Punkte auf der Netzhaut, die schwarz sind, gerade ist oder nicht. Die einfachste Form dieses Problems ist der Fall von zwei Punkten; die Paritätsfunktion ist in diesem Fall das Exklusiv-Oder (XOR). Allgemeiner lautet die Aussage, daß alle Probleme, deren positive und negative Fälle nicht durch eine Hyperebene (bei n lokalen Prädikaten: im n-dimensionalen Raum) voneinander getrennt werden können, auch stets nicht mit einem Perceptron einer Ordnung kleiner n (also insbesondere: fester Ordnung) gelöst werden können; Perceptrons können nur die sogenannten *linear separierbaren* Probleme lösen.

Das Buch von Minsky und Papert wird (fälschlich) dahingehend verstanden, daß generell neuronale Netze keine aussichtsreichen Mechanismen für Lernapparate seien[3]. Dies hat zwei Konsequenzen: Erstens kommt die neuroinformatische Forschung für lange Zeit fast zum Erliegen und zweitens halten viele Forscher alle grundlegenden Probleme für überwunden, als mit dem Backpropagation-Verfahren später ein Lernalgorithmus gefunden wird, der auch mehrstufige Perceptrons zu trainieren erlaubt und mit dessen Hilfe folglich auch nicht linear trennbare Probleme gelöst werden können.

1969 beschreiben Bryson und Ho [60] Verfahren zur Lösung von Regelungsproblemen, die das heute unter dem Namen Backpropagation bekannte Lernverfahren als Spezialfall enthalten. Offenbar liest kein Neuroinformatiker die Arbeit, jedenfalls wird die Erfindung nicht beachtet. 1974 erfindet Werbos [394] das Backpropagationverfahren nochmals. Offenbar liest wieder kein einziger Neuroinformatiker die Arbeit, jedenfalls wird die Erfindung wieder nicht beachtet. Ein drittes Mal wird Backpropagation 1985 von Parker [274] (und in ähnlicher Form von Le Cun [84]) erfunden und diesmal endlich wahrgenommen, wenn auch hauptsächlich indirekt: 1986 schreibt eine Gruppe um David Rumelhart und James McClelland das Buch „Parallel Distributed Processing", das unter anderem das Backpropagationverfahren (unter Erwähnung von Parker und Le Cun) vorstellt und damit die zweite große Welle der Neuroinformatik einleitet [308].

[3] Die genaue Intention und Rolle des Buches ist strittig. Siehe beispielsweise [151, Seite 14ff] für eine andere Sichtweise, sowie den Epilog der Neuausgabe von „Perceptrons" [249].

Mit der Entdeckung des Backpropagation-Algorithmus beginnt endlich die Phase der Nutzbarmachung der Neuroinformatik. Waren fast alle Anwendungen des Perceptrons von hauptsächlich akademischem Interesse, so beginnt jetzt eine Phase hektischen Ausprobierens neuronaler Netze für so ziemlich jede erdenkliche Anwendung. Zahlreiche Verbesserungen des Backpropagation-Algorithmus, insbesondere zur Beschleunigung des Lernens werden vorgestellt. In ihrer 1988 erschienenen Neuausgabe von „Perceptrons“ beklagen Minsky und Papert, daß in all der Begeisterung über die Lösbarkeit nicht linear separierbarer Probleme, die wichtigen Fragen der Skalierbarkeit und Lernkomplexität kaum überhaupt gestellt, geschweige denn beantwortet werden [249, Seiten 247ff]. Später werden dann aber große Fortschritte in dieser Richtung erzielt, wenngleich die theoretischen Ergebnisse für praktische Fälle nur selten nützliche Aussagen liefern (siehe [18, 337] und Abschnitt 2.6 für einen Überblick). Die Nutzbarmachung steht also mittlerweile in voller Blüte.

Die industrielle Revolution der Neuroinformatik hat noch nicht begonnen, steht aber wohl bald bevor. Alle Requisiten sind vorhanden: Erfahrung mit kleineren Anwendungen, kommerzielles Interesse an größeren Anwendungen [82, 234], ernsthafte Ansätze der Theoriebildung und schließlich die nötige Basistechnologie (schnelle Rechner). Ein direkter oder indirekter Einfluß der Neuroinformatik auf das tägliche Leben breiter Bevölkerungsschichten ist aber bislang noch nicht auszumachen.

Im Gegensatz zur symbolischen künstlichen Intelligenz gibt es in der Neuroinformatik kaum Streit um die Frage, ob die bisher realisierten Modelle schon künstliche Intelligenz aufweisen: Nein. Die Mechanisierung geistiger Arbeiten ist also noch nicht gelungen. Eine Phase der Verklärung ist in der Neuroinformatik nicht zeitlich abzugrenzen. Zwar sind Ansätze entsprechender Vorstellungen vor allem in der Anfangszeit, später eine zeitlang nach Erfindung der Backpropagation und neuerdings in der *brain building*–Bewegung verbreitet, doch ist den meisten Beteiligten jederzeit klar, daß die Nachbildung des menschlichen Gehirns oder größerer Teile davon schon allein aufgrund der schieren Menge dort realisierter Bauelemente (etwa 10^{11} Neuronen mit über 10^{13} Verbindungen) noch für einige Zeit außerhalb der Reichweite verfügbarer technischer Realisierungen bleiben wird.

Nachbemerkung: In der obigen Darstellung wurden diverse Zweige der Neuroinformatik völlig außer acht gelassen, weil sie auch sonst im Rahmen dieser Arbeit nicht diskutiert werden. Die wichtigsten sind die aus der Physik — genauer: der statistischen Mechanik — motivierten Ansätze mit Energiefunktionen, beginnend mit den Arbeiten von Hopfield [165], die Assoziativspeicher (z.B. bei [341] und [400]), sowie verschiedene Arten von Arbeiten zum unüberwachten Lernen (z.B. [197] und [308]) und Lernen durch Bekräftigung (reïnforcement learning, z.B. [31] und [32]). Für einen Überblick siehe z.B. [156]. Allerdings sind alle diese Zweige zumindest derzeit im Hinblick auf industrielle Anwendung auch von geringerer Bedeutung.

1.4 Eine kurze Geschichte der Parallelrechnerei

Die Geschichte der Parallelrechnerei verläuft bei weitem nicht so bewegt wie die der Neuroinformatik. Statt dessen beginnt sie mit einigen Fehlschlägen und nähert sich dann allmählich einer Beherrschung dieser Technik.

Der erste funktionsfähige digitale elektronische Rechner ist bereits ein Parallelrechner: Der Atanasoff-Berry-Computer (ABC). Insofern beginnt die primitive Phase der Parallelrechnerei zugleich mit der primitiven Phase des digitalen elektronischen Rechnens überhaupt. Der ABC ist konzipiert zum Lösen von linearen Gleichungssystemen und kann in jedem Schritt 30 Variablen zugleich bearbeiten [62, Seite 266]. Allerdings ist der Rechner festprogrammiert und insofern kein Computer im heutigen Sinne. Die meisten anderen Rechner in der Frühzeit der Informatik arbeiten sequentiell, beziehungsweise beschränken ihre Parallelität auf die parallele Verarbeitung von Bits einzelner Operanden.

Der erste größere Versuch zum Bau eines Parallelrechners mit mehr als einer kleinen Zahl von Prozessoren ist der ILLIAC IV (ab 1966). Die Maschine ist ein sehr ehrgeiziges Projekt, das nur teilweise als erfolgreich bezeichnet werden kann. Zwar wird eine Maschine mit sehr hoher Spitzen- und Dauerrechenleistung gebaut, jedoch wegen Technologie-, Zeit- und Geldproblemen nur ein Viertel so groß wie ursprünglich geplant, nämlich mit 64 anstatt 256 Prozessoren (in einer 10 Meter langen Reihe von 16 Schränken); trotzdem ist die Maschine geradezu unglaublich teuer (etwa 31 Millionen Dollar, in damaligem Geldwert enorm viel). Der Rechner hat zuwenig Speicher (16 Kilobytes pro Prozessor), Zuverlässigkeitsprobleme aufgrund fehlender Redundanz, nur rudimentäre Betriebssystemunterstützung und eine komplizierte Programmierschnittstelle, die zuviele Details dem Programmierer überläßt [6, Seiten 320-324]. Trotz aller dieser Nachteile ist die Maschine von 1975 bis 1981 erfolgreich im Einsatz; aufgrund der enormen Kosten wird aber nie eine Serienfertigung aufgenommen.

Andere Vorstöße verwenden die im Prinzip sehr bestechende Datenflußtechnik, die ab 1971 von Jack Dennis am MIT entwickelt wird. Bei diesem Prinzip wird die Parallelität auf der Ebene voneinander unabhängiger (Teil)Ausdrücke formuliert. Eine Datenflußmaschine synchronisiert die Ausführung der parallelen Prozesse vollautomatisch über die Verfügbarkeit der (Zwischen)Ergebnisse, also den Datenfluß. Der erste funktionierende Datenflußrechner ist der DDM1 von Burroughs (1976). Eine Anzahl anderer Maschinen folgen, von denen jedoch nie eine über das Prototypstadium hinauskommt [6, Seiten 388-411]. Alle diese Versuche sind der primitiven Phase der Parallelrechnerei zuzuordnen; die Rechner gelangen mit Ausnahme des ILLIAC kaum je zu einer Nutzanwendung.

Die ersten tatsächlich nützlichen Maschinen, die man als Parallelrechner bezeichnen könnte, sind die Vektorrechner. Die erste solche Maschine, die Cray-1, wurde ab 1976 ausgeliefert und in hohen Stückzahlen verkauft. Bei Vektorrechnern wird nicht volle Parallelität von Befehlsausführungen mittels mehrerer Prozessoren verwendet, sondern die Überlappung verschiedener Ausführungs*phasen* eines einzelnen Befehls (Fließbandverarbeitung). Bei Abarbeitung mehrerer gleicher Befehle hintereinander, wie es bei Operatoren auf Vektoren und Matrizen vorkommt, läßt sich so mit einem moderaten Hardwareaufwand eine erhebliche Beschleunigung erzielen [6, Seiten 303-313]. Diese Architekturen haben bis heute den Vorteil, daß die Dauerleistung typischer Programme bei Vektorrechnern wesentlich näher an die Spitzenleistung herankommt als bei Parallelrechnern. Allerdings läßt sich Fließbandverarbeitung nicht zu beliebigen Parallelitätsgraden skalieren, weshalb die modernen Nachfolger der Cray-1 zusätzlich zur Fließbandverarbeitung eine kleine Anzahl echt parallel arbeitender Prozessoren benutzen. Da Vektorrechner keine skalierbaren Parallelrechner sind, insbesondere keinen verteilten Speicher aufweisen, werden sie hier ausgeklammert und nicht als Durchbruch zur Nutzbarmachung des parallelen Rechnens betrachtet.

Der Durchbruch gelingt 1985 mit der Connection Machine. Daniel Hillis entwirft am MIT diesen Rechner mit 65536 Prozessoren, die nur je 1 Bit breit sind, und baut einen Prototyp, die CM-1, bei der Firma Thinking Machines. Vor allem aber zeigt er in seiner Dissertation [155] auch, wie ein solcher Rechner programmiert werden kann. Die Programmierung im zugrundeliegenden datenparallelen Modell erlaubt erstmals, weitgehend von der internen Struktur der Maschine zu abstrahieren und parallele Programme in einer Hochsprache (CmLisp [154]) zu formulieren. Im Gegensatz zu fast allen früheren in Forschungslaboratorien entstandenen Parallelrechnern wird die Connection Machine als CM-2 und CM-200 auch in Serie gebaut und ist ein kommerzieller Erfolg. Ein zweiter wichtiger Durchbruch erfolgt ebenfalls 1985: Die Vorstellung des Transputers durch INMOS (IMS T414). Der Transputer ist nicht nur einer der leistungsfähigsten Mikroprozessoren seiner Zeit, er ist auch ausdrücklich zum Bau von Parallelrechnern mit verteiltem Speicher gedacht und verfügt deshalb neben einem normalen Bus zusätzlich über vier bidirektionale, bitserielle Kommunikationsanschlüsse (*links*). Mit diesem Prozessor als Basiselement können sehr einfach Parallelrechner verschiedener Bauart implementiert werden. Die geringen Kosten der Transputer-Hardware führen zu einer weiten Verbreitung. Transputer-basierte Rechnersysteme werden zu einer Art Volkswagen der Parallelrechnerei und werden für fast alle erdenklichen Aufgaben eingesetzt. Connection Machine und Transputer markieren den Beginn der Phase der Nutzbarmachung der Parallelrechnerei und werden gefolgt von einer beträchtlichen Zahl anderer mehr oder weniger erfolgreicher Parallelrechner.

Eine Phase der Verklärung ist nur in Ansätzen zu erkennen. So zieht Hillis ausdrücklich das menschliche Gehirn als Existenzbeweis für effiziente hochparallele Problemlösungen heran und versteht seinen Entwurf einer „Connection Machine“ als Versuch, wesentliche Struktureigenschaften des Gehirns nachzubilden. Der erste Satz seiner Arbeit lautet *„Someday, perhaps soon, we will build a machine that will be able to perform the functions of a human mind, a thinking machine.“* [155]. Einer der Schlüssel hierzu liegt seiner Ansicht nach in hoher Verarbeitungsgeschwindigkeit und als den Weg dorthin betrachtet er eine möglichst hohe, feinkörnige Parallelität. Mittlerweile ist klar geworden, daß Parallelverarbeitung *an sich* kein Allheilmittel ist. Vielmehr sind zu ihrer erfolgreichen Anwendung eine beträchtliche Anzahl schwieriger Hard- und Softwareprobleme zu lösen, deren Bewältigung noch viele Jahre in Anspruch nehmen wird.[4] Insofern ist die Phase der Verklärung, wenn es sie denn gegeben hat, wieder vorüber. Von der Mechanisierung geistiger Arbeiten spricht derzeit folglich auch niemand ernsthaft.

Die industrielle Revolution der Parallelrechnerei ist noch nicht im Gange, könnte aber in Kürze beginnen. Fortschritte in der Mikroprozessortechnik, insbesondere die Erhöhung der Packungsdichte und die Verringerung der Leistungsaufnahme, sowie Fortschritte bei der Programmierung von Parallelrechnern, vor allem im Hinblick auf die Portabilität von Programmen, haben die Grundlage für einen kommerziellen Einsatz von Parallelrechnern auf breiter Ebene inzwischen geschaffen. Nichtsdestoweniger steckt das ganze Gebiet noch sehr in den Anfängen, wie es mit der Maschinentechnik zu Beginn der industriellen Revolution ja auch der Fall war. Die Artikel [38] und [204] geben einen Überblick über den Stand und die Perspektiven der Entwicklung von Hardware und Software für paralleles Rechnen für die nächsten Jahre.

Abbildung 1.1 zeigt nochmals die Entwicklungsphasen und den gegenwärtigen Standort von Maschi-

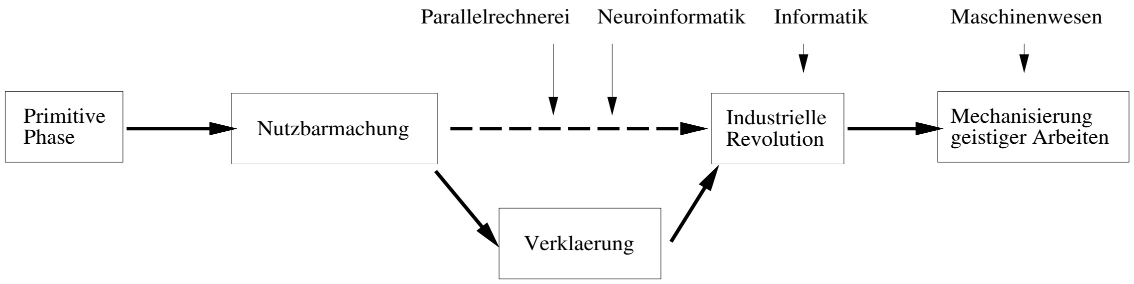


Abbildung 1.1: Entwicklungsphasen und Standort der vier Gebiete

nenwesen, Informatik, Neuroinformatik und Parallelrechnerei als Übersicht. Die wesentliche Erkenntnis aus dieser Betrachtung lautet, daß sowohl die Neuroinformatik als auch die Parallelrechnerei noch unreife Gebiete sind. Es gibt darin noch viele grundlegende Probleme, die aufgeklärt und gelöst werden müssen.

1.5 Ausgangspunkt, Aufbau und Ziele dieser Arbeit

Der Ausgangspunkt dieser Arbeit ist die Situation der Neuroinformatik und der Parallelrechnerei, die sich aus der obigen geschichtlichen Beschreibung ablesen läßt: Beide Gebiete haben sich zwar aus ihren Anfangsgründen gelöst, wir sind aber noch weit davon entfernt, zu wissen, wie ihre Basistechnologie auf einfache und zuverlässige Weise auf beliebige Probleme angewendet werden kann. Teilweise sind die im Wege stehenden Schwierigkeiten gut bekannt und verstanden und es fehlt „nur“ an den Lösungen (vor allem in der Parallelrechnerei); teilweise sind aber auch die diesen Schwierigkeiten zugrundeliegenden Mechanismen noch zu unklar, um direkt auf eine Lösung zuarbeiten zu können (vor allem in der Neuroinformatik). Noch fehlende Lösungen müssen erfunden und bewertet werden; noch fehlendes Verständnis der Mechanismen muß durch Beobachtung und Theoriebildung entwickelt werden.

[4] Es ist anzunehmen, daß auch Hillis der größere Teil dieser Probleme bewußt war.

Die vorliegende Arbeit liefert Beiträge in beiden Richtungen — sowohl Lösungen und ihre Auswertung, als auch Beobachtungen als Grundlage für Theoriebildung — auf zwei Gebieten:

- Lernverfahren für künstliche neuronale Netze
- Übersetzerbau für Parallelrechner mit verteiltem Speicher

Ziel der Arbeit ist die teilweise Beantwortung folgender Fragen:

1. Welche Ansätze für konstruktive Lernverfahren auf neuronalen Netzen eignen sich am besten für eine bestimmte Problemklasse?
2. Wie kann die Güte solcher Verfahren experimentell bewertet werden?
3. Können hochsprachliche Programme für konstruktive Verfahren trotz der dynamischen Unregelmäßigkeit ihrer Daten- und Berechnungsstrukturen effizient für Parallelrechner übersetzt werden?
4. Welche Übersetzungstechniken eignen sich dafür und wie groß ist ihr Nutzen verglichen mit einfacheren Übersetzungsmethoden?

Die Arbeit hat einen zweigeteilten Aufbau. Im ersten Teil untersuche ich konstruktive Lernverfahren aus der Neuroinformatik. Dazu wird zunächst der Stand des theoretischen Wissens über das Lernen in neuronalen Netzen (*NN*) begutachtet (Abschnitt 2.6) und ein Überblick über die bisherigen praktischen Ansätze und ihren Erfolg gegeben (Abschnitte 2.7 und 2.8). Beides zeigt, warum die Beschäftigung mit automatischen, insbesondere konstruktiven Lernverfahren lohnend ist. Anschließend untersuche ich drei Arten von Ansätzen zu automatischen Lernverfahren: Frühes Stoppen des Trainings (Kapitel 4), additives Lernen, also die allmähliche Vergrößerung des Netzes durch Hinzufügen von Ressourcen (Kapitel 5) und schließlich subtraktives Lernen (Beschneiden), also die allmähliche Verkleinerung des Netzes durch Entfernen von Ressourcen (Kapitel 6). Für jeden dieser Ansätze werden zunächst die bestehenden Verfahren diskutiert, daraus vielversprechende ausgewählt, mögliche Verbesserungen dafür hergeleitet und schließlich experimentell das Verhalten der bekannten und neuen Varianten untersucht. Die Organisation dieser Experimente liefert eine exemplarische Antwort auf Frage 2. Alle drei Lernverfahrensklassen werden außerdem verglichen (Kapitel 7), was eine Antwort auf Frage 1 liefert. Alle empirischen Untersuchungen wurden mit Programmen durchgeführt, die in der im zweiten Teil der Arbeit eingeführten Programmiersprache CuPit geschrieben sind und mit Hilfe des ebenfalls im zweiten Teil vorgestellten Übersetzers auf einem Parallelrechner ausgeführt werden können. Alle empirischen Untersuchungen basieren ferner auf einer Sammlung von Benchmark-Problemen, die mangels anderweitiger Verfügbarkeit für diese Arbeit erstellt wurde und nun öffentlich verfügbar ist (Kapitel 3); die meisten anderen Arbeiten auf dem Gebiet der NN-Lernverfahren vernachlässigen bisher eine gründliche experimentelle Auswertung ihrer Beiträge (Abschnitt 3.1). Im Hinblick auf die Auswertung der Versuchsreihen werden einige Untersuchungen zur Methodik gemacht. Dabei stellt sich heraus, daß parametrische statistische Methoden, namentlich der t-Test, bei ihren (wenigen) in der Literatur dokumentierten Anwendungen wohl meistens falsch eingesetzt wurden und die Ergebnisse deshalb unzuverlässig sind. Ich beschreibe den korrekten Einsatz von t-Tests zur Auswertung von Versuchsreihen mit neuronalen Lernverfahren (Abschnitt 3.4).

Der zweite Teil der Arbeit liefert Beiträge zum Übersetzerbau für Parallelrechner. Ich beschreibe ein Programmiermodell und dann konkret eine Programmiersprache, die speziell zur Beschreibung von konstruktiven neuronalen Lernverfahren entworfen sind — die dabei verwendeten Konstrukte lassen sich aber in allgemeine, objektorientierte, parallele Sprachen integrieren (Kapitel 9). Eine Betrachtung der Eigenschaften von konstruktiven Lernverfahren ergibt, daß man für diese Klasse von Programmen eine Daten- und Prozeßverteilung angeben kann, die zugleich das Datenlokalitäts- und das Lastbalancierungsproblem in guter Annäherung löst, ohne zu große zusätzliche Übersetzungs-, Laufzeit- oder Speicherkosten zu verursachen. Diese Verteilung und eine Reihe damit verbundener anderer Optimierungen werden in Kapitel 10 vorgestellt; Abschnitt 10.5 beschreibt einen Übersetzer, der die Optimierungen realisiert. Gemeinsam liefern diese Betrachtungen eine positive Antwort für Frage 3. Die Bewertung der Optimierungen des Übersetzers erfolgt anhand von Messungen der im ersten Teil betrachteten Lernverfahren und Lernprobleme und ist in Kapitel 11 beschrieben. Die Kapitel 10 und 11 beantworten zusammengenommen die Frage 4. Abschließend werden die Ergebnisse diskutiert (Kapitel 12).

Zur leichteren Erschließbarkeit enthält die Arbeit am Ende einen Schlagwortindex, in dem insbesondere diejenigen Textstellen aufgeführt sind, an denen Begriffe eingeführt werden. Phrasen werden im Index nicht umgestellt, das heißt zum Beispiel das Stichwort *einfaches Perceptron* wäre im Index unter E eingetragen und nicht unter P.

Kapitel 2

Lernen mit neuronalen Netzen

Neural Networks are the second best way
of doing just about anything.
John Denker

We would especially like to acknowledge
the effort and dedication of Peter Getting
who devoted 12 years to understanding
the organization of the Tritonia network
of 14 neurons.
Sylvie Ryckebusch et. al. (NIPS 88, p.393)

Dieses Kapitel definiert zunächst die formale Terminologie der neuronalen Netze (Abschnitt 2.1) und ordnet neuronale Lernverfahren in den Zusammenhang des maschinellen Lernens ein (Abschnitt 2.2). Danach wird der Zusammenhang von neuronalen Netzen und statistischen Modellen diskutiert (Abschnitt 2.3), insbesondere das gemeinsame Hauptproblem, das Bias/Varianz-Dilemma (Abschnitt 2.4), und eine Übersicht gegeben über die Ansatzpunkte, die eine Forschung über Lernverfahren für neuronale Netze nehmen kann (Abschnitt 2.5). Weitere Abschnitte stellen im Lichte dieser Einführung den aktuellen Stand der theoretischen (Abschnitt 2.6) und der praktischen Forschung (Abschnitt 2.7) über Lernverfahren für neuronale Netze vor, insbesondere bisherige Vorschläge für konstruktive Lernverfahren (Abschnitt 2.8). Der letzte Abschnitt faßt das Gesagte zusammen und stellt die Beiträge vor, die diese Arbeit für das Gebiet neuronaler Lernverfahren liefern wird (Abschnitt 2.9).

Die Abschnitte 2.2 bis 2.7 verfolgen dabei den Zweck, durch eine breite Übersicht über Forschungsfragen und bisherige -ergebnisse herauszuarbeiten, warum die in dieser Arbeit behandelten automatischen (insbesondere die konstruktiven) Lernverfahren ein besonders lohnender Forschungsgegenstand sind.

In diesem Kapitel werden wir von den vielen gebräuchlichen Modellen für neuronale Netze nur eine bestimmte Klasse diskutieren, die vorwärtsgerichteten Netze, und alle anderen Modelle ignorieren (siehe auch die Nachbemerkung im Abschnitt 1.3).

2.1 Definitionen

Dieser Abschnitt führt Terminologie und formale Schreibweisen ein, mit denen Aussagen über neuronale Netze und die zugehörigen Lernverfahren formuliert werden. Wir beschränken die Netzmodelle auf deterministische, vorwärtsgerichtete, musterabbildende Netze für überwachtes Lernen.

2.1.1 Neuronales Netz

Ein *neuronales Netz* (*NN*) ist eine Abbildung $N : \mathbb{R}^e \mapsto \mathbb{R}^a$. Das Netz hat e *Eingänge* und a *Ausgänge*. Der Begriff Abbildung wird im mathematischen Sinne gebraucht, das heißt ein NN hat keinen inneren Zustand (ist kein Zustandsautomat) und verhält sich deterministisch. Zur Veranschaulichung der nun folgenden Definitionen dient Abbildung 2.1. Die Struktur der durch das neuronale Netz definier-

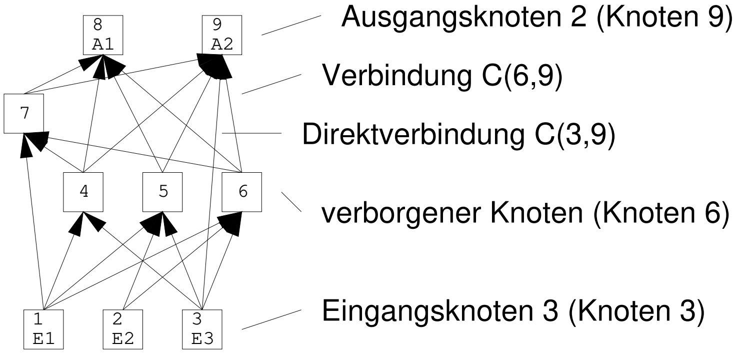


Abbildung 2.1: Unregelmäßig verbundenes Netz mit einigen Direktverbindungen: Eingangsschicht mit 3 Knoten, erste und zweite verborgene Schicht mit 3 bzw. 1 Knoten und Ausgangsschicht mit 2 Knoten.

ten Abbildung N wird in folgender Modellvorstellung beschrieben: N ist darstellbar durch ein Tupel $(\mathcal{N}, \mathcal{C}, A)$ bestehend aus einer geordneten Menge von *Knoten* (*nodes*) $\mathcal{N} = (\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n)$, *Verbindungen* (*connections*) $\mathcal{C}(i,j)$ zwischen diesen Knoten und Aktivierungsfunktionen $A(i)$ für die Knoten. Diese Modellvorstellung betrachtet das Netz als einen gerichteten Graphen mit einer zugehörigen Berechnungsvorschrift für den Fluß von Informationen durch diesen Graphen. Wir werden die Knoten oft mit ihren Nummern identifizieren. Die partielle Abbildung

$$\mathcal{C} : \{1, \ldots, n\} \times \{1, \ldots, n\} \mapsto \text{Verbindung}$$

ordnet einem Paar von Knoten $\mathcal{N}_i$ und $\mathcal{N}_j$ aus $\mathcal{N}$ (repräsentiert durch ihre Nummern i und j) eine gerichtete Verbindung von $\mathcal{N}_i$ nach $\mathcal{N}_j$ zu. 'Verbindung' ist ein abstrakter Typ, dessen Struktur nicht von Interesse ist. Jeder Verbindung wird ein *Gewicht* (*weight*) zugeordnet mit Hilfe einer Abbildung w. Zur Vereinfachung der Schreibweise benutzen wir $w_{ij} := w(\mathcal{C}(i,j))$. Wenn (und nur wenn) $\mathcal{C}(i,j)$ definiert ist, sagen wir $\mathcal{N}_i$ *hat eine Verbindung zu* $\mathcal{N}_j$ oder auch *die Verbindung* w_{ij} *existiert*. Oftmals werden die Verbindungen mit ihren Gewichten identifiziert. Wenn $\mathcal{C}$ nicht definiert ist, gelte $w_{ij} := 0$. Ein Netz heißt *vorwärtsgerichtet* (*feed-forward*), wenn $\mathcal{C}(i,j)$ undefiniert ist für alle $i \geq j$.

Die Abbildung $A : \mathcal{N} \mapsto (\mathbb{R} \mapsto \mathbb{R})$ ordnet jedem Knoten $\mathcal{N}_i$ eine reelle, reellwertige *Aktivierungsfunktion* $A(\mathcal{N}_i)$ zu. Eine auf der Heaviside-Funktion H basierende Aktivierungsfunktion der Form $a(x) = c_1 H(x-s) - c_2$ mit $c_1, c_2, s \in \mathbb{R}$ heißt *lineare Schwellwertfunktion* mit dem Schwellwert s; normalerweise ist $c_1 = 1$ und $c_2 = 0$, das heißt unterhalb von s ist die Funktion gleich 0 und ab s gleich 1. Eine nach oben und unten beschränkte, monoton steigende Aktivierungsfunktion heißt eine *Sigmoidfunktion*, oft auch *Quetschfunktion* (*squashing function*) genannt. Die Funktion $sig(x) = 1/(1+e^{-x})$ heißt die *Standard-Sigmoidfunktion*. Eine weitere oft benutzte Aktivierungsfunktion ist die Identität; sie ist der Regelfall für Eingangsknoten und eine häufige Wahl für Ausgangsknoten. Meist haben auch sigmoide Aktivierungsfunktionen einen Verschiebungs- oder Schwellwertparameter s wie oben beschrieben (*offset*, *threshold*, oder *bias*); dieser wird wie ein Gewicht behandelt und kann deshalb auch über eine gedachte Verbindung ausgehend von einem Knoten mit konstanter Ausgabe 1 (*Biasknoten*, *Verschiebeknoten*) ausgedrückt werden. Wenn nichts gegenteiliges gesagt wird, gehen wir stillschweigend von letzterer Konstruktion aus. Oft betrachten wir nur die auf ganz $\mathbb{R}$ differenzierbaren Sigmoidfunktionen.

Die Knoten $\mathcal{N}_i$ mit $i \leq e$ heißen *Eingangsknoten* (*input nodes*, alternativ auch *Eingabeknoten* genannt) $\mathcal{N}_{Ei}$; $\mathcal{C}(i,j)$ ist bei vorwärtsgerichteten Netzen undefiniert für alle $j \leq e$ und alle i. Die Knoten $\mathcal{N}_i$ mit

$i > n - a$ heißen *Ausgangsknoten* (*output nodes*, alternativ auch *Ausgabeknoten* genannt) $\mathcal{N}_{A(i-n+a)}$; in der Regel ist $\mathcal{C}(i,j)$ undefiniert für alle $i \geq n - a$. Alle anderen Knoten $\mathcal{N}_i$ heißen *verborgene Knoten* (*hidden nodes*) $\mathcal{N}_{V(i-e)}$; für jeden verborgenen Knoten $\mathcal{N}_v$ muß es mindestens ein i und ein j geben, so daß sowohl $\mathcal{C}(i,v)$ als auch $\mathcal{C}(v,j)$ definiert ist. Jede größtmögliche Gruppe $L = \{i | i \in L_{min} \dots L_{max}\}$ von verborgenen Knoten, zwischen denen keine Verbindungen existieren, d.h. für die gilt $\forall i,j \in L : (i,j) \notin dom(\mathcal{C})$, heißt eine verborgene *Schicht* oder *Lage* (*layer*). Ebenso bilden die Eingangsknoten und die Ausgangsknoten zusammen je eine Schicht. Die verborgenen Schichten werden in der durch die Knotennummern vorgegebenen Reihenfolge von 1 beginnend numeriert. Die Eingangsschicht erhält die Nummer 0, die Ausgangsschicht die Nummer k (bei $k-1$ verborgenen Schichten). Ein Netz heißt *vollverbunden* (*fully connected*), wenn jeder Knoten in Schicht i für $i < k$ eine Verbindung zu allen Knoten in Schicht $i+1$ hat. Ein Netz heißt *vollverbunden mit allen Direktverbindungen*, wenn jeder Knoten in Schicht i für $i < k$ eine Verbindung zu allen Knoten jeder Schicht mit höherer Nummer hat. Ein Netz heißt *vollverbunden mit Direktverbindungen von Eingängen zu Ausgängen*, wenn jeder Knoten in Schicht i für $i < k$ eine Verbindung zu allen Knoten in Schicht $i+1$ und in Schicht k hat. Die letzteren beiden Fälle unterscheiden sich nur, wenn das Netz mehr als eine verborgene Schicht hat.

Der durch die Knoten und die dazwischen bestehenden gerichteten Verbindungen definierte Graph beschreibt die *Topologie* eines neuronalen Netzes. Zur Topologie wird oft auch die Festlegung der Aktivierungsfunktionen gerechnet, nicht jedoch die Festlegung der Gewichte. Die Topologie wird oft auch *Architektur* genannt. Ist das Verbindungs*muster* einer Architektur klar, so kann sie einfach durch die Angabe der Knotenzahlen für jede Schicht beschrieben werden. Wir schreiben z.B. „10-16-8-6 Netz“ für ein Netz mit 10 Eingangsknoten, 6 Ausgangsknoten und zwei verborgenen Schichten mit 16 bzw. 8 Knoten. Analoge Schreibweisen funktionieren für andere Anzahlen von Knoten und von verborgenen Schichten, z.B. „10-32-6 Netz“ für ein Netz mit nur einer verborgenen Schicht. Ist die Zahl der Eingangs- und Ausgangsknoten ebenfalls bekannt, so schreiben wir verkürzt „Netz mit 16+8 verborgenen Knoten“, bzw. „Netz mit 32 verborgenen Knoten“ usw.

Bei jeder Berechnung des Netzes $N(\mathbf{x})$ mit gewünschter Ausgabe $\mathbf{y}$ für $\mathbf{x} = (x_1, \dots, x_e)$ und $\mathbf{y} = (y_1, \dots, y_a)$ gibt es für jeden Knoten $\mathcal{N}_i$ eine Eingangsgröße (*net input*) in_i und eine Ausgangsgröße (*output* oder *activation level*) $out_i := A(\mathcal{N}_i)(in_i)$, die auch als *Eingabe* bzw. *Ausgabe* des Knotens bezeichnet werden. Die Ausgangsgröße des Ausgangsknotens $\mathcal{N}_{Ai}$ bezeichnen wir auch als o_i, also $o_i := out_{i-n+a}$. Für die Eingangsknoten $\mathcal{N}_{Ei}$ gilt $in_i := x_i$. Die Berechnung innerhalb des Netzes erfolgt nun nach folgender Regel: Für alle verborgenen Knoten $\mathcal{N}_i$ und für alle Ausgangsknoten $\mathcal{N}_i$ gilt

$$in_i := \sum_{k=1}^{i-1} w_{ki} \, out_k$$

Diese Definition ist nur dann eindeutig, wenn das Netz vorwärtsgerichtet ist, andernfalls enthält das Netz und damit die Berechnungsdefinition Zyklen (*rekurrentes Netz*), und es muß eine Angabe gemacht werden z.B. nach wievielen Berechnungsschritten der Ausgabewert der Ausgangsknoten als Ergebnis betrachtet werden soll. Bei vorwärtsgerichteten Netzen sind zur Berechnung einer Netzausgabe aus einer Netzeingabe soviele nacheinanderfolgende Schritte nötig, wie das Netz Schichten hat. Innerhalb jedes dieser Schritte können alle Operationen parallel erfolgen. Es gibt neuronale Netze, die nach anderen Regeln als dieser linearen Summation rechnen, z.B. *Netze höherer Ordnung* (*higher order networks*) [104, 292], die (auch) multiplikative Terme verwenden, oder *stochastische Netze* [115, 144], die einen stochastischen Anteil in die Berechnung einfließen lassen. Solche Netztypen werden aber in dieser Arbeit nicht betrachtet.

Netze des oben beschriebenen Typs werden in der Literatur aus historischen Gründen oftmals *Perceptron* genannt [304, 248]. Dabei impliziert die Bezeichnung in der Form *(einfaches) Perceptron* meist ein Netz ohne verborgene Knoten und mit einer Schwellwert-Aktivierungsfunktion in den Ausgangsknoten, während die Bezeichnung *mehrschichtiges Perceptron* (*multi layer perceptron*, *MLP*) meist für ein Netz mit einer oder mehreren verborgenen Schichten steht, in deren Knoten sigmoide Aktivierungsfunktionen verwendet werden [308].

Für die Formulierung von Lernverfahren auf NNs ist es zweckmäßig, nicht die durch N definierte Abbildung abstrakt zu betrachten, sondern die Knoten und Verbindungen ähnlich wie in der obigen Beschreibung als Objekte anzusehen. Diese Objekte können dann auch zusätzliche, für die Abbildung nicht benötigte Information tragen. Diese Sicht werden wir im folgenden meist annehmen. Ein neuronales Netz, dessen Gewichte noch nicht bestimmt sind, definiert eine *Klasse* von Abbildungen. Oft werden wir das Netz mit dieser Klasse anstatt nur mit einer einzelnen Abbildung identifizieren.

Gelegentlich wird in dieser Arbeit der Begriff neuronales Netz in einem weiter gefaßten und nicht von dieser Definition abgedeckten Sinne gebraucht, um auch über andere Netztypen zu sprechen, wie es schon im Einleitungskapitel der Fall war. Dies ist jeweils aus dem Zusammenhang ersichtlich oder wird ausdrücklich vermerkt.

2.1.2 Beispiel und Fehler

Das Lernen in neuronalen Netzen basiert auf dem Paradigma „Lernen aus Beispielen", das heißt, dem Lernmechanismus werden einmal oder mehrmals verschiedene Beispiele des zu lernenden Zusammenhangs präsentiert, und der Lernmechanismus bildet daraus durch Induktion (mittels Interpolation) eine Hypothese über diesen Zusammenhang.

Ein *Beispiel* ist im Falle des neuronalen Netzes N, wie es oben definiert wurde, ein Paar von Vektoren $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^e \times \mathbb{R}^a$ wie ebenfalls oben schon eingeführt. Dabei repräsentiert $\mathbf{x}$ die Eingabe für das Netz und $\mathbf{y}$ die zugehörige *gewünschte* Ausgabe. Diese Ausgabe nennt man auch *Ziel* (*target*). Die Koeffizienten der Eingabe- und Ausgabevektoren kodieren die *Attribute* des Lernproblems, also diejenigen Merkmale, die allein als notwendig erachtet werden, das Lernproblem zu lösen. Die gewählte Kodierung bezeichnet man auch als *Repräsentation* des Problems. Die Attribute können *kardinal* sein, d.h. auf einer kontinuierlichen oder diskreten Skala mit Nullpunkt liegen, oder sie können *ordinal* sein, d.h. mit Ordnung aber ohne Nullpunkt und ohne Entfernungsmaß, oder sie können *nominal* sein, d.h. nur endlich viele verschiedene Werte ohne jede Ordnung aufweisen. Für ein neuronales Netz müssen alle Attribute durch reelle Zahlen kodiert werden, wobei für ein Attribut ein oder mehrere Koeffizienten verwendet werden können.

Ist im Vorhinein bekannt, daß es beim zu lernenden Problem nur genau n verschiedene Ziele geben kann und kennt man diese Ausgaben im Voraus, so spricht man von einem *Klassifikationsproblem* mit n Klassen, andernfalls von einem *Approximationsproblem*.

Das *Lernen* besteht darin, die durch das neuronale Netz definierte Abbildung dahingehend zu manipulieren, daß sie für die Beispieleingaben möglichst geringe Abweichungen zwischen tatsächlichen und gewünschten Ausgaben produziert. Die Abweichung, die die tatsächliche Ausgabe des Netzes von der gewünschten Ausgabe hat, wird durch eine *Fehlerfunktion* E, auch genannt *Kostenfunktion* oder *Zielfunktion* (*error function*, *cost function*, *loss function*, *objective function*) beschrieben, die dem Paar von erhaltener und gewünschter Ausgabe eine reelle Zahl zuordnet. Die am häufigsten verwendete Fehlerfunktion E ist die *quadratische Fehlerfunktion* (*squared error*): $E(N(\mathbf{x}), \mathbf{y}) := \sum_{i=1}^{a}(y_i - o_i)^2$. Soll das Netz eine Klassifikationsaufgabe lernen, so wird die Ausgabe des Netzes mit einer *Interpretationsfunktion* als Nummer einer erkannten Klasse interpretiert. In diesem Fall bezeichnet man den Prozentsatz falsch klassifizierter Beispiele einer Beispielmenge als den *Klassifikationsfehler*. Benutzt man zur Darstellung von k Klassen zum Beispiel k Ausgangsknoten mit einer 1-aus-k Kodierung (d.h. alle Ausgaben sollen Null sein, mit einer Ausnahme, die Eins sein soll; dies ist die üblichste Darstellung), so ist die in der Regel verwendete Interpretationsfunktion die *winner-takes-all*-Funktion, d.h. die erkannte Klassennummer ist die Nummer des Ausgangsknotens mit dem größten beobachteten Ausgabewert.

Die Menge aller Beispiele, die bei einem Lernvorgang für ein neuronales Netz verwendet werden, nennt man auch die *Trainingsdaten*. Diese Menge wird häufig in zwei disjunkte Teile gespalten, von denen einer, die *Trainingsmenge* (*training set*), tatsächlich zum Lernen verwendet wird, während der andere, die *Validationsmenge* oder *Validierungsmenge* (*validation set*) nur zur Steuerung des Lernens dient, wie später noch näher erläutert wird. Um bei der Erforschung von Lernverfahren nach Abschluß des Lernens zu beurteilen, wie gut das Netz die durch die Beispiele beschriebene Aufgabe gelernt hat, verwendet man weitere Beispiele, die während des Lernvorgangs überhaupt nicht betrachtet wurden. Die Menge dieser Beispiele heißt *Testmenge* (*test set*).

Der im Mittel über alle Beispiele der Trainingsmenge, Validationsmenge, bzw. Testmenge auftretende Fehler heißt auch kurz *Trainingsfehler*, *Validationsfehler* bzw. *Testfehler*. Der Testfehler wird auch *Generalisierungsfehler* genannt und charakterisiert die *Generalisierungsleistung* oder *Generalisierungsfähigkeit* des Netzes. Da der Testfehler mit Hilfe des Validationsfehlers geschätzt wird, werden beide Begriffe manchmal auch im Zusammenhang mit dem Validationsfehler gebraucht.

2.1.3 Lernregel und Lernverfahren

Lernen in neuronalen Netzen ist meist ein iterativer Prozeß. In jedem Schritt dieses Prozesses werden ein oder mehrere Beispiele vom Netz bearbeitet und anschließend aufgrund der dabei beobachteten Eingaben, Ausgaben und Fehler eine Veränderung an der das Netz definierenden Abbildung N vorgenommen. Diese Veränderung kann sowohl das Vorhandensein bestimmter Knoten betreffen, als auch die Aktivierungsfunktionen der Knoten, das Vorhandensein von Verbindungen und das Gewicht von Verbindungen. In manchen Netzen und Verfahren kommen noch weitere veränderliche Parameter vor.

Eine *Lernregel* ist eine Vorschrift, die angibt, wie in einem Veränderungsschritt die Gewichte der Verbindungen aufgrund der vorher beobachteten Eingaben, Ausgaben und Fehler und aufgrund des bisherigen Netzzustandes verändert werden sollen. Lernregeln haben im allgemeinen Parameter, deren Wert zur Verwendung der Lernregel festgelegt werden muß; insbesondere können die meisten Lernregeln mit unterschiedlichen Fehlerfunktionen benutzt werden. Die am häufigsten verwendete Lernregel ist die *verallgemeinerte Delta-Regel* (*generalized delta rule*), die anwendbar ist, wenn alle Aktivierungsfunktionen und die Fehlerfunktion differenzierbar sind. Sie besagt, daß jede Verbindung mit einem bisherigen Gewicht w_{ij} im nächsten Schritt das Gewicht $w_{ij} - \eta\,\delta_{ij}$ haben soll; dabei ist δ_{ij} die zu dem Gewicht gehörende Komponente des Gradienten des Fehlers in bezug auf die betrachteten Beispiele[1]. Der Parameter η heißt die *Lernrate*. Mit nicht zu großen Lernraten und gleichbleibenden Beispielen realisiert diese Lernregel einen Gradientenabstieg auf der Fehlerfunktion durch Anpassung der Gewichte. Die Berechnung des Gradienten für Gewichte, die nicht mit Ausgangsknoten verbunden sind, erfolgt durch Anwendung der Kettenregel der Differentiation. Es sind soviele Iterationen der Kettenregel notwendig, wie verborgene Schichten vorhanden sind. Dabei wird sozusagen das Entstehen der Fehler durch die verborgenen Knoten hindurch zurückverfolgt; deshalb trägt das Verfahren den Namen *Fehlerrückverfolgung* (*backward propagation of error* oder kurz *Backpropagation*).

Als *Lernverfahren* oder *Lernalgorithmus* bezeichnen wir die Gesamtheit aller Vorschriften zur Durchführung des Lernens: wann welche Beispiele aus den Trainingsdaten dem Netz präsentiert werden, wann welche Lernregel mit welchen Parametern angewendet wird, wann welche sonstige Veränderung vorgenommen wird (zum Beispiel das Ändern einer Aktivierungsfunktion, das Hinzufügen oder Entfernen von Knoten oder Verbindungen oder das Ändern eines Parameters) und schließlich wann das Lernen beendet wird. Das zugrundegelegte neuronale Netz ist einer der Parameter eines Lernverfahrens. Meistens bearbeiten Lernverfahren wiederholt hintereinander alle Beispiele der Trainingsmenge. In diesem Fall nennt man ein solches Durchlaufen aller Trainingsbeispiele eine *Epoche*.

[1] Der Fehler für eine feste Menge von Beispielen kann als Funktion der Gewichte im Netz verstanden werden. Sind jede Aktivierungsfunktion und die Fehlerfunktion differenzierbar, läßt sich die Ableitung des Fehlers in bezug auf die Gewichte berechnen.

Lernverfahren sind meist parametrisiert mit der (Anfangs-)Struktur des neuronalen Netzes, der verwendeten Menge von Beispielen und einer festen Zahl von weiteren Parametern, deren Wert zur Verwendung des Lernverfahrens festgelegt werden muß. Ein Grundprinzip neuronaler Lernverfahren ist die *Lokalität*: Die Lernregel sollte zur Berechnung von Änderungen an einem Element des Netzes (z.B. dem Gewicht einer Verbindung) nur Daten heranziehen, die lokal in diesem Element verfügbar sind. Dieses Prinzip ist aus der Arbeitsweise biologischer neuronaler Netze begründet, welche es immer einhalten. Die Einhaltung des Prinzips erbringt erhebliche Vorteile bei einer Hardware-Realisierung von Lernverfahren und bei der Software-Implementation auf Parallelrechnern: Sind irgendwelche Datenzugriffe nichtlokal, so müssen bei Hardware über die Verbindungen des neuronalen Netzes hinaus zusätzliche Signalwege eingebaut werden, bei paralleler Softwareimplementation fallen vermehrt Kommunikationsoperationen an. Lernverfahren mit nichtlokalen Zugriffen sind also nicht im engeren Sinne neuronale Verfahren.

Der Begriff Backpropagation wird üblicherweise als Synonym für ein Lernverfahren verwendet, das die verallgemeinerte Delta-Regel verwendet. Genaugenommen ist Backpropagation aber kein Lernverfahren und noch nicht einmal eine komplette Lernregel, sondern nur eine Rechenvorschrift, die einen Teil einer Lernregel bildet.

2.1.4 Modellauswahl und automatisches Lernen

In Anlehnung an die in der Statistik gebräuchliche Terminologie nennen wir jedes neuronale Netz auch ein *Modell*, nämlich ein Modell für den zu lernenden Zusammenhang. Somit beschreibt jedes iterative Lernverfahren eine *Folge von Modellen*, da jeder Zwischenzustand als Modell betrachtet werden kann.

Damit ergibt sich die Frage nach der *Modellauswahl*: Welche der darstellbaren Modelle steuert das Lernverfahren an, und welches der angesteuerten wählen wir als Resultat aus? Dabei gehen wir hier davon aus, daß die Fehlerfunktion, die Problemrepräsentation und die Beispielmenge bereits festliegen. Im einfachen Gradientenabstiegsfall sind die angesteuerten Modelle durch die bei den einzelnen Abstiegsschritten erreichten Punkte definiert; ausgewählt wird das letzte erreichte Modell. In komplizierteren Lernverfahren wählen wir möglicherweise ein Modell aus einem früheren Abstiegsschritt (*frühes Stoppen*, *early stopping*) oder wir verändern die zugrundeliegende Topologie des neuronalen Netzes während des Lernens (*konstruktive Lernverfahren*). Beides erweitert den Raum für die Modellauswahl.

Wir nennen ein Lernverfahren ein *automatisches Lernverfahren*, wenn es die Modellauswahl ohne menschlichen Eingriff vornimmt und die dafür zuvor zu wählenden freien Parameter fest sind, also nicht so sehr vom Lernproblem abhängen, daß sie jeweils vom Benutzer passend eingestellt werden müssen. Da eine günstige Auswahl von Parameterwerten aufgrund der kombinatorischen Explosion schon bei einer kleinen Parameteranzahl sehr aufwendig werden kann, haben automatische Lernverfahren große praktische Vorteile. Aus diesem Grund betrachte ich in meiner Arbeit ausschließlich automatische Lernverfahren. Eine Einschränkung des Begriffs wird bei der Auswahl einer geeigneten Netztopologie zugelassen.

2.2 Neuronale Netze und maschinelles Lernen

Dieser Abschnitt zeigt, wie sich das Lernen in neuronalen Netzen in den größeren Zusammenhang des maschinellen Lernens einordnen läßt und wo mögliche Vor- und Nachteile von neuronalen Netzen liegen. Wie wir sehen werden, bilden neuronale Netze einen sehr erfolgreichen Ansatz zum maschinellen Lernen und sind deshalb ein nutzbringender Forschungsgegenstand.

Die Neuroinformatik, zumindest soweit sie sich mit Lernverfahren zur Lösung von Anwendungsproblemen befaßt, ist nur ein Zweig des größeren Gebiets, das man als *maschinelles Lernen* bezeichnet[2]. Bis vor wenigen Jahren waren die Aktivitäten beider Gebiete fast gänzlich getrennt, jetzt wachsen sie allmählich zusammen. Der Grund für die Trennung war der Glaubensstreit um die Frage, ob es besser sei, das Lernen auf der Manipulation von Symbolen zu basieren, wie es für digitale Computer naheliegend ist, oder auf subsymbolischen Operationen, die mehr an der Arbeitsweise des Gehirns orientiert sind.

2.2.1 Symbolisches versus subsymbolisches Lernen

Der symbolische Ansatz gründet sich auf die formale Logik. Die elementaren Bestandteile symbolischer Lernansätze sind einerseits die *Symbole*, zum Beispiel Wahrheitswerte (wahr, falsch) oder Werte eines Aufzählungstyps (z.B. grün, gelb, blau, rot, schwarz) und andererseits diskrete Operationen auf diesen Symbolen, etwa logisches UND oder Wertvergleich ($A =$ gelb). Aufgrund dieser Darstellungsweise läßt sich das Verhalten symbolischer Systeme gut in Systemen formaler Logik darstellen, und es können leicht Schlüsse darüber gezogen und Eigenschaften bewiesen werden. Da jedes Symbol eine eigene Bedeutung hat und diese zu der Anwendung in erkennbarer Beziehung steht (so könnte „gelb“ etwa die Farbe eines gerade betrachteten und zu klassifizierenden Objekts sein), können symbolische Schlüsse leicht für den jeweiligen Anwendungsfall interpretiert werden.

Im Gegensatz dazu steht die subsymbolische Darstellungsweise. Hier sind die elementaren Bestandteile nicht mit eigener Bedeutung ausgestattet, sondern so primitiv, daß sie oft keine erkennbare Beziehung zum Ganzen mehr haben. Beim subsymbolischen System „Gehirn“ sind die Bestandteile z.B. einerseits chemische Botenstoffe oder elektrische Ladungen und andererseits Operationen wie der Transport der Stoffe und die Auslösung einer Entladung. Analog sind bei künstlichen neuronalen Netzen die elementaren Bestandteile einerseits Zahlen und andererseits Operationen wie Addition und Multiplikation. Solche Systeme sind in ihren Einzelteilen nicht zu verstehen. Erst das Zusammenwirken der Teile ergibt ein System, dessen Verhalten interpretierbar ist und dem ein eigener Sinn zugeordnet werden kann.

Wegen dieser Grundelemente eignen sich der symbolische und der subsymbolische Ansatz verschieden gut für bestimmte Anwendungsgebiete. Bei symbolischer Repräsentation ist die Darstellung diskret. Gut darstellen lassen sich alle Größen, die nur eine kleine feste Zahl verschiedener Werte haben können, wie zum Beispiel Wahrheitswerte oder gut unterscheidbare Farben. Die Darstellung kontinuierlicher Größen, beispielsweise einer Temperatur, erfordert hingegen eine künstliche Einteilung in Äquivalenzklassen (kalt, warm, heiß) [107]. Demgegenüber ist die Darstellung kontinuierlicher Werte für den subsymbolischen Ansatz problemlos, während die Darstellung echt symbolischer Werte aufwendig wird, da diese künstlich in Zahlen kodiert werden müssen.

Offensichtlich haben beide Ansätze ihre Existenzberechtigung, da es für beide jeweils Probleme gibt, die sich nicht adäquat darstellen lassen. Der Aufsatz [247] enthält eine schöne Darstellung des Streits und empfiehlt, den Gegensatz aufzulösen und die Ansätze miteinander zu verschmelzen.

2.2.2 Empirischer Vergleich

Man sollte nach der obigen Beschreibung erwarten, daß symbolische Lernverfahren bei Problemen mit ausschließlich diskreten Ein- und Ausgaben besser abschneiden als neuronale Netze. Zudem sind über viele Jahre hinweg raffinierte symbolische Lernverfahren entwickelt worden. Eine umfangreiche empirische Studie [350] hatte jedoch ein gegenteiliges Resultat (ohne Allgemeingültigkeit beanspruchen zu wollen).

[2] Wir wollen hier nur *induktives Lernen* betrachten, also das Lernen einer Hypothese über einen Zusammenhang zwischen Eingaben und Ausgaben anhand einer Reihe von Beispielen.

Dabei wurden insgesamt 25 Verfahren auf die gleichen drei einfachen Lernprobleme angesetzt. Jedes dieser Probleme hatte 6 Eingabeattribute mit je zwei bis vier verschiedenen Werten; zu lernen war eine binäre Klassifikation aufgrund von 122 bis 169 (von 432 möglichen) Beispielen. Jedes Verfahren wurde von einem Experten eingesetzt, der es befürwortete. Die betrachteten Verfahren waren AQ17-DCI, AQ17-HCI, AQ17-FCLS, AQ14-NT, AQ15-GA, Assistant Professional, mFOIL, ID5R, IDL, ID5R-hat, TDIDT, ID3, AQR, CN2, CLASS WEB, ECOWEB, PRISM, Backpropagation und Cascade Correlation, sowie zum Teil noch Varianten davon. Nur Backpropagation und Cascade Correlation sind Lernverfahren für neuronale Netze, alle anderen Verfahren arbeiten ganz oder überwiegend symbolisch.

Den Vergleich gewann Backpropagation mit Gewichtsverfall (weight decay) gleichauf mit Cascade Correlation und deutlich vor allen anderen Verfahren (mit 100%/100%/97,2% korrekter Klassifikation auf den untrainierten Beispielen bei den drei Problemen), nur ein Verfahren war besser als Backpropagation ohne Gewichtsverfall, nämlich AQ17-DCI mit 100%/100%/94,2%. (Für eine ähnliche, wenngleich kleine Studie, siehe [368].)

Man muß sich darüber klar sein, daß diese Studie natürlich keine allgemeingültigen Aussagen erlaubt. Beispielsweise ist die Anzahl von Attributen in den verwendeten Lernproblemen recht klein, und es ist nicht auszuschließen, daß manche Verfahren bei höherer Attributanzahl relativ besser abschneiden würden. Dennoch macht die Studie deutlich, daß neuronale Netze offenbar recht günstige Eigenschaften als Basis von Lernverfahren haben, sogar, wenn die Lernaufgaben eigentlich dem symbolischen Bereich zuzurechnen sind.

Mehrere andere (unterschiedlich gute) Vergleichsstudien fanden ebenfalls für neuronale Netze gleiche oder bessere Ergebnisse als für konventionelle Verfahren des Maschinellen Lernens oder konventionelle statistische Verfahren (z.B. CART und C4.5 [368], CART [21], polynomielle Regression oder Gaußsche Maximum-Likelihood-Schätzer [129]). Ein Vergleich von MLPs mit dem symbolischen ID3-Algorithmus ergab, daß das MLP geringfügig besser ist für sehr kleine Trainingsmengen, für verrauschte Daten und für Daten mit fehlenden Attributwerten; ferner ist es deutlich besser für numerische Daten. Oft ist aber die Qualität des gefundenen Schätzers für verschiedene Verfahren *bei optimaler Anwendung dieser Verfahren* so ähnlich, daß andere Merkmale den Ausschlag geben sollten [211, 212, 265]. Zu diesen gehören der Rechenaufwand zur Bestimmung des Schätzers, der Rechenaufwand für seine Ausführung in der späteren praktischen Anwendung, der Speicherbedarf, die Einfachheit der Implementierung und vor allem die Einfachheit der richtigen (d.h. annähernd optimalen) Anwendung.

2.2.3 Hybridverfahren

Trotz dieser Erfolge haben neuronale Verfahren Nachteile. So ist das Ergebnis eines Lernvorgangs aufgrund der subsymbolischen Struktur nicht für Menschen verständlich in dem Sinne, daß sich die beim Lernen konstruierte Hypothese verstehen ließe. Aus ähnlichem Grund ist es bei neuronalen Netzen schwieriger als bei symbolischen Verfahren, Vorwissen über das Problem in das Lernen einzubringen.

Es liegt daher nahe, eine Verbindung von symbolischen und neuronalen Ansätzen zu versuchen, um die Vorteile beider zu koppeln. Eine solche Kopplung von symbolischem und subsymbolischem Ansatz nennen wir ein *Hybridverfahren*. Tatsächlich gibt es zahlreiche Arbeiten auf diesem Gebiet. Eine Stoßrichtung besteht darin, direkt beim Lernvorgang symbolische und subsymbolische Teile miteinander zu verbinden [232, 311, 331] oder Lernmodule verschiedener Art zu einer Problemlösung zu kombinieren [53]. Eine andere Arbeitsrichtung versucht innerhalb des neuronalen Ansatzes symbolisches Wissen zu verwenden, beispielsweise zum Vorstrukturieren des Netzes vor Beginn des Lernens [320, 366] oder zum Extrahieren von Regeln aus einem gelernten Netz [131, 243] oder einer Kombination von beidem, bei der das Training des neuronalen Netzes die Regeln *verfeinert* [362].

Angesichts des erheblichen Potentials und der ebenso erheblichen Schwierigkeiten werden solche Versuche, das Beste aus beiden Welten zu bekommen, noch lange Zeit ein wichtiges Forschungsthema bleiben. In dieser Arbeit werden keine Hybridverfahren behandelt.

2.2.4 Genetische Algorithmen

Eine wichtige Klasse von Hybridverfahren sind die Anwendungen von genetischen Algorithmen auf neuronale Netze [317]. Ein *genetischer Algorithmus* [91] ist ein Optimierungsverfahren, das sich an die biologische Evolution anlehnt und für neuronale Netze beispielsweise folgendermaßen funktioniert (es gibt viele Varianten): Beginne mit einer *Population* von unterschiedlichen neuronalen Netzen (Individuen). Für eine Anzahl aufeinanderfolgender *Generationen* führe dann die Schritte Trainieren, Bewerten, Selektieren, Paaren, Mutieren aus. Selektieren, Paaren und Mutieren erzeugen zusammen die Population für die nächste Generation, wobei Individuen mit guter Bewertung probabilistisch bevorzugt werden. *Selektieren* wählt ein Mitglied der alten Population unverändert aus, *Mutieren* nimmt zusätzlich eine zufällige Veränderung daran vor und *Paaren* (*cross-over*) verbindet zwei Individuen zu einem neuen, das Teile der Eigenschaften seiner „Eltern" vereint. Zur Durchführung von Mutieren und Paaren ist jedes Individuum durch eine „Gen-Sequenz" beschrieben. Nach einer Zahl von Generationen wird das beste Individuum der letzten Generation als Ergebnis des Lernvorgangs betrachtet. Das Trainieren kann entfallen, wenn die Anpassung der Gewichte Teil des Mutierens und Paarens ist. Auch die Lernregel selbst könnte im Prinzip Gegenstand der Evolution sein [412].

Der Vorteil des genetischen Ansatzes ist, daß er eine *globale Suche* in einem sehr umfangreichen Hypothesenraum ermöglicht und dabei auch stochastische Elemente erlaubt, die den Rahmen von Gradientenabstiegsverfahren sprengen. Dieser Vorteil ist zugleich ein Nachteil, denn eine solche globale Suche bedingt eine große Menge von „unsinnigen" Berechnungen (d.h. Suche in nutzlose Richtungen), von denen ein Großteil bei der Auswahl aus Populationen wieder völlig verworfen wird. Deshalb ist der Rechenaufwand von genetischen Algorithmen für neuronale Netze meistens enorm hoch.

In der Literatur sind zahlreiche Anwendungen genetischer Algorithmen auf die Konstruktion neuronaler Netze publiziert; es gibt sogar eigene Konferenzen über das Thema, z.B. [3]. Angesichts des hohen Rechenaufwands ist es nicht verwunderlich, daß ein großer Teil dieser Publikationen nur winzige Spielzeugprobleme betrachtet, so daß z.B. [55, 318] positive Ausnahmen sind.

Wenn Methoden gefunden werden, die Suche schnell genug auf gute Bereiche des Suchraums einzuengen, so daß der Berechnungsaufwand beherrschbar wird, hat der genetische Ansatz ein großes Potential. In dieser Arbeit werden keine genetischen Algorithmen behandelt.

2.3 Neuronale Netze und Statistik

Lernen in neuronalen Netzen gemäß dem oben eingeführten Rahmen hat eine enge Verwandtschaft zu statistischen Methoden der Datenmodellierung. Zweck dieses Abschnittes ist es, zu zeigen, worin diese Verwandtschaft besteht, was neuronale Netze dennoch an neuen Anstößen liefern und warum sie als eigenständige Forschungsrichtung neben der Statistik verfolgt werden sollten.

Der überwiegende Teil der in der Neuroinformatik vorgestellten Techniken zur Benutzung in Lernverfahren ist nur eine neue Verkleidung für Techniken, die in der Statistik längst bekannt sind. Die Verkleidung besteht insbesondere darin, daß für das Lernen in neuronalen Netzen eine völlig andere Terminologie benutzt wird, als sie in der Statistik üblich ist. Hier ein kleines Wörterbuch: Was bei neuronalen Netzen Eingabe genannt wird, heißt in der Statistik *unabhängige Variable* (genauer: Eingang gleich unabhängige Variable und Eingabewert gleich Wert der unabhängigen Variablen), eine Ausgabe heißt *abhängige Variable*, ein Beispiel heißt *Beobachtung*, die Trainingsdaten heißen *Stichprobe*, Fehler heißen *Residuen*, Lernen heißt *Schätzen*, die Fehlerfunktion heißt *Schätzkriterium*, das (angenäherte) Lösen eines Approximationsproblems heißt *Regression*, das Lösen eines Klassifikationsproblems heißt *Diskriminantenanalyse*, die Gewichte der Verbindungen heißen *Parameter* und die Werte dieser Gewichte heißen *Parameterschätzungen*. Mit Ausnahme des Begriffs Parameter wird diese statistische Terminologie in der Literatur über neuronale Netze kaum verwendet.

Diese unterschiedliche Terminologie führt dazu, daß der Zusammenhang zur Statistik nicht augenfällig wird. Ein weiterer Grund dafür, daß so vieles aus der Statistik nacherfunden wird liegt im großen Umfang und der Komplexität der statistischen Literatur, aufgrund derer vielen Forschern in der Neuroinformatik die relevanten Teile nicht ausreichend bekannt sind.

Was zum Beispiel wurde denn nun nacherfunden? Nun, einfache Perceptrons mit linearer Aktivierungsfunktion am Ausgang entsprechen einer *linearen Regression*, mit Schwellwert-Aktivierungsfunktion wird daraus eine *lineare Diskriminantenanalyse*, bei mehreren Ausgängen entsprechend *multivariate lineare Regression* oder *multivariate lineare Diskriminatenanalyse*. Ein mehrschichtiges Perceptron (mit Sigmoid-Aktivierungsfunktionen) ist ein Spezialfall der *multivariaten nichtlinearen Regression*. Netze aus *radialen Basisfunktionen* entsprechen *Regression mit Kernfunktionen* (*kernel regression*), auch bekannt als *Parzen-Fenster*n. Die oben angeführten Begriffe bezeichnen jeweils *Modelle*, die Hypothesen- bzw. Lösungsräume definieren. Die für diese Modelle zu verwendenden Lernalgorithmen (statistisch: Schätzverfahren) sind damit noch nicht festgelegt. Diese Unterscheidung zwischen Modellen und Verfahren wird in der Neuroinformatik oft verwischt. Auch viele Lernverfahren für neuronale Netze sind in der Statistik altbekannt. Wie bereits erwähnt, ist zum Beispiel das als Backpropagation bekannte Vorgehen nichts anderes als ein Gradientenabstieg auf einer mehrdimensionalen Funktion. Selbst ein so innovativ aussehendes Verfahren wie *Cascade Correlation* erfindet großenteils die in der Statistik als *backfitting* bekannte Methode neu, wobei mit der Korrelation allerdings ein „falsches" Schätzkriterium verwendet wird. Zahlreiche andere Beispiele ließen sich anführen. Ein ähnliches Bild ergibt sich im Bereich des unüberwachten Lernens, der hier nicht besprochen wird.

Eine ausführlichere Diskussion des Zusammenhangs von Statistik und neuronalen Netzen findet sich zum Beispiel in [298, 312, 396]. Dort werden auch Literaturhinweise für die jeweiligen statistischen Modelle und Verfahren gegeben.

Hat denn nun die Neuroinformatik überhaupt etwas Neues zu bieten? Ja, sie hat. Es gibt eine Zahl von Beiträgen aus der Forschung über neuronale Netze, die man als Innovationen für die Statistik betrachten kann und andere, die zumindest Anstöße geben, in der Statistik bisher weniger beachtete Zweige genauer zu erforschen.

1. Die vermutlich wichtigste Innovation ist die Einführung *nichtkonvergenter Methoden*. Traditionell wird in der Statistik bei iterativen Regressionsmethoden stets bis zum minimalen Fehler auf der Menge der Beispiele „trainiert"; alle theoretischen Aussagen über Eigenschaften der entstehenden Lösungen gelten nur für dieses Minimum. Praktiker, die mit neuronalen Netzen arbeiteten, begannen jedoch bald, das Training vorzeitig, also vor der Konvergenz zum Fehlerminimum, abzubrechen, nämlich sobald auf einer unabhängigen Menge von zusätzlichen Beispielen (Validationsmenge) der beobachtete Fehler nicht mehr weiter abfällt, sondern wieder ansteigt, während zugleich der Fehler auf den eigentlichen Trainingsbeispielen sehr wohl noch kleiner wird. Dieses Verfahren, genannt *frühes Stoppen*, erwies sich als eine empirisch so gute Regularisierungsmethode (siehe auch Abschnitt 2.7.6), daß die Statistiker nach einiger Zeit begannen, ihre Abneigung gegen diese Herangehensweise aufzugeben und sich jetzt an die Gründung einer Theorie nichtkonvergenter Methoden machen.
2. Das mehrschichtige Perceptron ist eine nützliche Ergänzung zu den in der Statistik verwendeten nichtlinearen Modellen. Es hat als Regressionsmodell Eigenschaften, die erfahrungsgemäß auch in hochdimensionalen Räumen für viele praktische Anwendungen günstig sind. Der gute Erfolg beim praktischen Einsatz von mehrschichtigen Perceptrons hat auch allgemein in der Statistik den nichtlinearen Modellen verstärkte Aufmerksamkeit verschafft. Nichtlinear heißt hier *nichtlinear in den Parametern* (im Gegensatz zu *nichtlinear in den Eingaben*). So ist eine polynomielle Regressionsfunktion der Form $r(x) = \sum_{i=0}^{n} w_i x^i$ zwar eine nichtlineare Funktion der Eingabe x, verhält sich aber linear im Hinblick auf Änderungen der Parameter w_i. Die bei mehrschichtigen Perceptrons vorliegende Regressionsfunktion besteht im Gegensatz dazu beispielsweise aus Teilen der Form $r(x) = 1/(1+e^{-wx})$, die auch in den Parametern nichtlinear sind. Nichtlineare Modelle sind zwar bekanntermaßen mächtig, haben aber im allgemeinen einen hohen Berechnungsaufwand

und sind schwierig anzuwenden. Die Verfügbarkeit schneller Rechner und die relative Gutartigkeit von mehrschichtigen Perceptrons für viele Probleme sind die Grundlage dafür, daß sich das Interesse an nichtlinearen Modellen in der Statistik verstärkt.

3. Eine dritte Gruppe von Anstößen aus der Neuroinformatik entsteht dadurch, daß die hinter neuronalen Netzen stehende Modellvorstellung eine Reihe möglicher Vorgehensweisen nahelegt, die in der Statistik zuvor nicht betrachtet wurden. Beispiele hierfür sind manche Regularisierungsterme [117, 145] und Beschneidungsverfahren [85, 117, 216], die während des Trainings einzelne Parameter aus dem Modell entfernen.

Zum genaueren Verständnis dieser Punkte ist die im folgenden Abschnitt gegebene Beschreibung des Bias/Varianz-Dilemmas nützlich.

2.4 Das Bias/Varianz–Dilemma

Das Bias/Varianz–Dilemma charakterisiert die fundamentale Entwurfsabwägung zwischen Flexibilität und Kontrollierbarkeit, die für alle statistischen Schätzverfahren und für alle symbolischen wie subsymbolischen Lernverfahren gilt. Dieser Abschnitt stellt das Dilemma im Hinblick auf das Lernen in neuronalen Netzen dar. Der Abschnitt hat den Zweck, zu verdeutlichen, daß die Suche nach guten Lernverfahren gleichbedeutend ist mit der Suche nach der Realisierung eines gut zu den jeweiligen Lernproblemklassen passenden Bias.

In der folgenden Darstellung werden wir zur Vereinfachung der Schreibweise nur den univariaten Fall (d.h. Abbildung nach $\mathbb{R}$) betrachten, zudem mit nur einer Eingangsvariablen. Die Diskussion läßt sich aber auf den höherdimensionalen und den multivariaten Fall analog erweitern; das Dilemma ist dort sogar noch wesentlich schlimmer. Ferner legen wir die quadratische Fehlerfunktion zugrunde; auch dies geschieht ohne Beschränkung der Allgemeinheit — abgesehen von der Tatsache, daß die Varianz nur bei quadratischem Fehler so genannt wird. Die Diskussion ist gleichermaßen für alle statistischen Schätzverfahren gültig, neuronale Netze sind nur ein spezieller Anwendungsfall; deshalb werden wir teilweise statistische Terminologie gebrauchen. Eine ausführliche Abhandlung des Bias/Varianz-Dilemmas mit anschaulichen Beispielen aus dem Gebiet der neuronalen Netze findet sich in [125]; eine allgemeine Behandlung findet sich in vielen Statistikbüchern über Regressionstechniken.

2.4.1 Bias, Varianz und das Dilemma

Eine Stichprobe $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ aus einer Wahrscheinlichkeitsverteilung P sei als Trainingsmenge gegeben. P beschreibt das zu lösende Lernproblem. Sollte das Lernproblem keinen stochastischen Anteil enthalten, sondern deterministisch sein, ist die Verteilung entartet. Ein vollständig spezifiziertes Lernverfahren L liefert aufgrund dieser Stichprobe eine Funktion $L(\mathbf{x}, T)$, die die „richtige“ Funktion $f^*_P(\mathbf{x})$ schätzt. f^*_P ist dadurch definiert, daß das Integral über die $(\mathbf{x}, y)$ gemäß P von $(f^*_P(\mathbf{x}) - y)^2$ minimal ist. Im vorliegenden Fall der quadratischen Fehlerfunktion wird das erfüllt durch $f^*_P(\mathbf{x}) = E[y|\mathbf{x}]$, also dem Erwartungswert von y, gegeben $\mathbf{x}$. Dieser Erwartungswert ist der optimale Schätzer für y und folglich das, was unser Lernverfahren idealerweise liefern sollte.

Wir notieren den Erwartungswert eines Terms $f(T)$ über alle möglichen Stichproben T jeden Umfangs aus der Verteilung P als $E_P[f(T)]$. Dann läßt sich die erwartete Abweichung unserer gelernten Schätzung $L(\mathbf{x}, T)$ von der optimalen Schätzung $f^*_P(\mathbf{x})$ für irgendein $\mathbf{x}$ ausdrücken als $E_P[(L(\mathbf{x}, T) - f^*_P(\mathbf{x}))^2]$. Diese erwartete Abweichung läßt sich durch Umformungen in folgende zwei Teile zerlegen

$$\begin{aligned}
&E_P[(L(\mathbf{x}, T) - f^*_P(\mathbf{x}))^2] = \\
&\qquad (E_P[(L(\mathbf{x}, T)] - f^*_P(\mathbf{x}))^2 && \text{„Bias“} \\
&\qquad + E_P[(L(\mathbf{x}, T) - E_P[L(\mathbf{x}, T)])^2] && \text{„Varianz“}
\end{aligned}$$

Der *Bias* (*systematischer Fehler*) ist also die Abweichung des im Mittel über alle Trainingsmengen T zu erwartenden gelernten Schätzers $\bar{L}(\mathbf{x}, T)$ vom optimalen Schätzer $f_P^*(\mathbf{x})$ an der Stelle $\mathbf{x}$. Die *Varianz* (statistischer Fehler) ist die erwartete Abweichung des gelernten Schätzers $L(\mathbf{x}, T)$ von dem im Mittel zu erwartenden gelernten Schätzer $\bar{L}(\mathbf{x}, T)$.

Informell heißt dies, der Bias gibt an, wie stark die Vorliebe des Lernverfahrens L für gewisse Funktionen ist, gemessen daran, wie stark die aufgrund dieser Vorliebe für eine Stichprobe T ausgewählten Schätzfunktionen von der optimalen Schätzfunktion abweichen. Entsprechend beschreibt die Varianz, wie stark die Neigung von L ist, sich auf zufällige Eigenarten einer Stichprobe T einzulassen — anders ausgedrückt: wie flexibel L ist. Der Begriff Bias läßt sich auch in anderer, verwandter Weise charakterisieren, nämlich durch die Klasse von Funktionen, die L zu liefern imstande ist und die Wahrscheinlichkeitsverteilung auf dieser Klasse über die Menge aller möglichen Trainingsmengen. Die Stärke des Bias hängt in dieser Betrachtungsweise mit der Größe der Funktionenklasse zusammen. Kann diese Größe skaliert werden, etwa bei neuronalen Netzen durch Vergrößerung oder Verkleinerung des Netzes, so ist der Bias variabel, und es muß ein „passender" Bias ausgewählt werden. Ein Beispiel für einen Bias wäre die Einschränkung, daß L grundsätzlich nur lineare Funktionen liefert.

Offensichtlich kann man Lernverfahren angeben, die auf der Stichprobe T stets einen Bias von Null haben, falls die Stichprobe keine Widersprüche enthält; der erzeugte Schätzer $L(\mathbf{x}, T)$ muß lediglich zu jedem $\mathbf{x}$ aus T das zugehörige y liefern. Solche Lernverfahren heißen *biasfrei*. Sie können dennoch einen großen Fehler außerhalb von T verursachen, falls ihre Varianz zu groß ist. Auch die Varianz läßt sich immer zu Null machen, indem L alle angebotenen Daten in T völlig ignoriert und immer denselben Schätzer liefert. In diesem Fall ist aber natürlich der Bias und damit auch der gesamte Fehler sehr groß. Sinnvolle Lernverfahren müssen deshalb eine sinnvolle Balance zwischen Bias und Varianz finden. Dies geschieht, indem ein Bias verwendet wird, der gut zum Lernproblem P „paßt". Dieser Bias verringert dann automatisch die Varianz.

Dieses Bias-Findungsproblem ist bei gegebener Lernaufgabe P um so schwieriger, je weniger Beispiele vorliegen. Das liegt daran, daß mit steigender Stichprobengröße die Varianz automatisch abnimmt, denn die durch T gelieferte Beschreibung von P wird mit steigender Größe von T immer genauer. Jedes biasfreie Lernverfahren L liefert also Schätzer $L(\mathbf{x}, T)$, die sich mit steigender Größe von T allmählich für alle $\mathbf{x}$ beliebig weit dem optimalen Schätzer $f_P^*(\mathbf{x})$ nähern. Oft verwendet man Modelle, deren Flexibilität sich skalieren läßt, etwa durch die Wahl der Anzahl freier Parameter (bei neuronalen Netzen also die Wahl der Anzahl von Gewichten, z.B. durch Wahl der Zahl von verborgenen Knoten). Solche Modelle mit sehr vielen Parametern werden in der Statistik paradoxerweise *nichtparametrische Modelle* genannt. Der Bias eines solchen Modells nimmt mit steigender Größe (Parameteranzahl) ab, die Varianz zu. Die Art des Bias ist durch die Eigenarten der Modellklasse als Ganzes vorgegeben (z.B. nur Polynomfunktionen, nur Kombinationen radialer Basisfunktionen, nur MLPs, etc.).

Das *Bias/Varianz-Dilemma* besteht nun darin, daß sich die dringend erwünschte kleine Varianz nur auf Kosten eines unerwünscht hohen Bias erreichen läßt. Es soll also eine optimale Abwägung zwischen Bias und Varianz gefunden werden, um schon für kleine Stichproben T einen geringen Fehler *außerhalb* von T zu erreichen, jedoch ist ohne genaue Kenntnis von P nicht bekannt, wo diese optimale Abwägung liegt und wie Art und Stärke eines Bias aussehen, der sie erreicht.

2.4.2 Beispiel

Betrachten wir zur Veranschaulichung ein einfaches Beispiel. Gegeben sei der in Abbildung 2.2 dargestellte Datensatz aus 61 Punkten. Er ist eine Stichprobe der Überlagerung einer quadratischen Funktion mit einer standardnormalverteilten Zufallsvariablen. Angenommen, wir wollen eine Funktion lernen, die den Prozeß approximiert, der diese Daten erzeugt hat.

Die Extremvarianten mit Bias Null oder Varianz Null für Lernverfahren sind in Bild 2.3 dargestellt. Der Schätzer mit Varianz Null beruht auf dem Lernverfahren „Egal was für Daten Du mir zeigst,

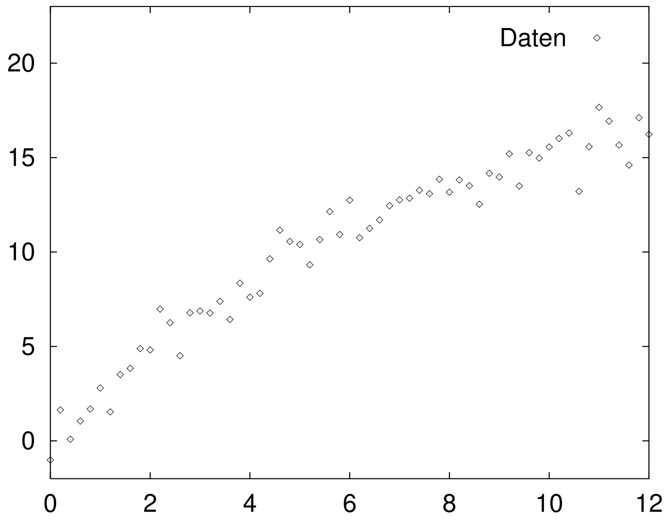


Abbildung 2.2: Punkte des Beispieldatensatzes. Die Daten sind nicht, wie man meinen könnte, mit einer verrauschten linearen Funktion erzeugt, sondern aus der Überlagerung von $2,5x + 0,1x^2$ mit standardnormalverteiltem Rauschen.

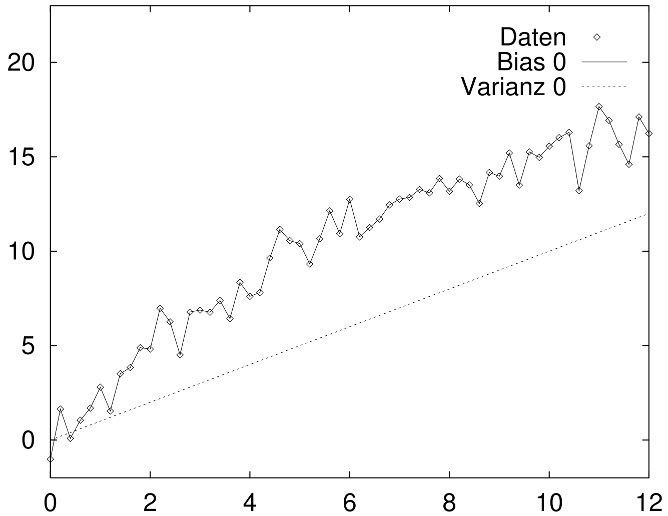


Abbildung 2.3: Beispieldatensatz und zwei zugehörige Schätzer für den zugrundeliegenden Prozeß; einer mit Bias Null (linearer Interpolator) und einer mit Varianz Null (Identitätsfunktion).

ich glaube immer, daß die beste Schätzfunktion die Identität ist“. Das ist in diesem Fall gar nicht so schlecht, im allgemeinen aber natürlich nicht sinnvoll. Der Schätzer mit Bias Null auf den Daten ist eine lineare Interpolation von Punkt zu Punkt. Die Varianz dieses Schätzers ist hoch, aber nicht maximal. Zum Beispiel hätte ein Polynom vom Grad 60, das die Datenpunkte interpoliert, ebenfalls Bias Null auf den Daten, jedoch eine viel höhere Varianz aufgrund der starken Überschwinger zwischen den interpolierten Punkten.

In Bild 2.4 sehen wir nun zwei sinnvolle Schätzer für das Verhalten der Datenmenge. Der erste beruht auf der Annahme, daß das zu lernende System linear sei, und erzeugt als Schätzer folglich eine lineare Funktion, die mit linearer Regression berechnet wird. Der Bias dieses Lernverfahrens ist im vorliegenden Fall sinnvoll und das Ergebnis dementsprechend brauchbar. Ein etwas schwächerer Bias besteht darin, auch quadratische Funktionen zuzulassen und eine Regression auf einem Polynom zweiten Grades auszuführen. Da dieser Bias hier exakt das dem Lernproblem zugrundeliegende System repräsentiert, weicht der resultierende Schätzer nur sehr wenig vom optimalen Schätzer ab. Die restliche Abweichung rührt daher, daß die 61 Datenpunkte aufgrund der Zufallsschwankungen auch mit quadratischer Modellannahme kein exaktes Abbild der Verteilung liefern.

Eine Veranschaulichung des Bias/Varianz-Dilemmas bei neuronalen Netzen geben die folgenden Grafiken. Abbildung 2.5 zeigt die Schätzung, die das Training eines kleinen neuronalen Netzes mit 4 verborgenen Knoten (10 Parameter) liefert. Das Netz ist nicht in der Lage, eine genaue Approxi-

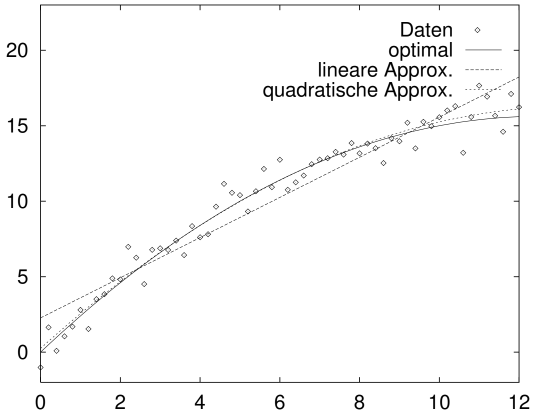


Abbildung 2.4: Beispieldatensatz mit optimalem Schätzer des Systems, linearer Regression der Daten und quadratischer Regression der Daten

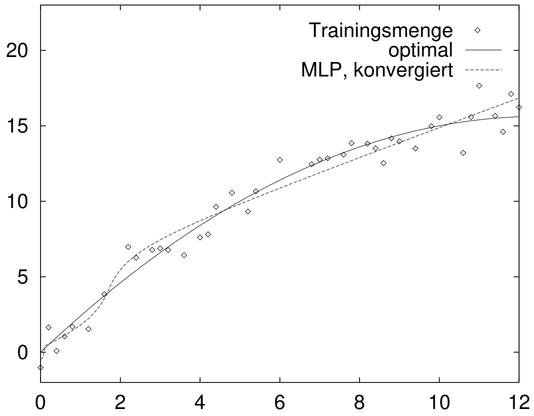


Abbildung 2.5: Trainingsdatensatz mit optimalem Schätzer und Schätzer aus neuronalem Netz (MLP). Das Netz hat 1-4-1 Architektur und folglich freie 10 Parameter. Aktivierungsfunktion der verborgenen Knoten ist die Sigmoidfunktion $y = x/(1 + |x|)$, der Ausgangsknoten hat lineare Aktivierung. (Trainiert mit dem RPROP-Verfahren, Initialisierungswerte der Gewichte gleichverteilt zufällig aus dem Intervall $[-0.1; 0.1]$. Trainingsbeispiele sind zufällig gewählte 40 Punkte des Datensatzes.)

mation zu liefern, es hat einen zu starken Bias.

Abbildung 2.6 zeigt als Gegenstück ein Netz mit zu starker Varianz. Die Bedingungen sind diesel-

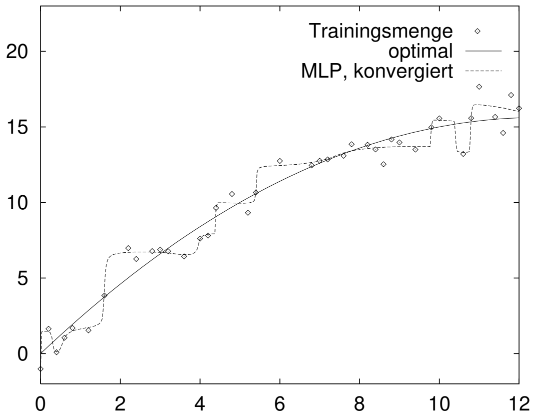


Abbildung 2.6: Trainingsdatensatz mit optimalem Schätzer und Schätzer aus 1-200-20-1 neuronalem Netz.

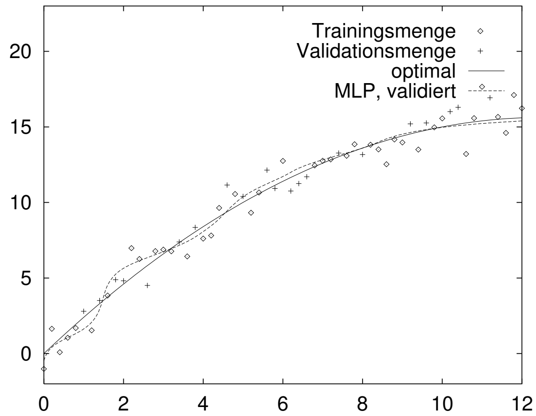


Abbildung 2.7: Wie nebenstehend, jedoch Training mit Hilfe einer Validationsmenge bestehend aus den restlichen 20 Punkten am optimalen Punkt abgebrochen.

ben wie in Abbildung 2.5, nur daß das hier verwendete Netz zwei verborgene Schichten mit 200 bzw. 20 Knoten und damit insgesamt 4220 Parameter besitzt. Die Trainingsbeispiele werden von diesem Netz besser angenähert als von dem in Abbildung 2.5 gezeigten, jedoch ist die Interpolation zwischen den Trainingsbeispielen miserabel. Dieses Eigenschaftspaar bezeichnet man als *Überanpassung*. Abbildung 2.7 zeigt das Ergebnis für das gleiche Netz, bei dem jedoch eine Regularisierungsmethode angewendet wurde, um die Varianz zu beschränken: das Training wird zu dem Zeitpunkt abgebrochen, an dem das Netz einen optimalen Zwischenzustand erreicht hat. Zur Schätzung dieses Zeitpunktes wird der Fehler auf einer zusätzlichen Menge von Beispielen beobachtet, die nicht zum Training herangezogen werden (Validationsmenge). Dieses Netz zeigt qualitativ ein ähnliches Verhalten wie das kleine Netz aus Abbildung 2.5, jedoch ist die Approximation des optimalen Schätzers vor allem bei größeren x wesentlich genauer. Es ist anzumerken, daß die Eigenschaften von Polynomfunktionen als Modelle im allgemeinen bei weitem nicht so günstig sind, wie es dieses Beispiel vermuten läßt. Das gilt insbesondere in höheren Dimensionen.

2.4.3 Folgerung

Die Konsequenz aus dem Bias/Varianz-Dilemma besteht in folgender Erkenntnis:

> Die wichtigste Aufgabe zur Entwicklung guter Lernmethoden besteht darin, einen zur gegebenen Problemklasse gut passenden Bias zu finden.

Ferner gibt es offenbar zwei grundsätzlich verschiedene Methoden, wie ein Lernverfahren ein schlechtes Ergebnis produzieren kann: durch zu wenig Flexibilität (zu starker Bias) oder durch zu viel Flexibilität (zu starke Varianz). Somit ist die zweite wichtige Aufgabe, die Stärke des gewählten Bias richtig an das Problem anzupassen.

2.5 Ansatzpunkte

Aus obiger Betrachtung ergeben sich als Ansatzpunkte für Verbesserungen von Problemlösungen mit neuronalen Lernverfahren die Wahl der Problemrepräsentation, der Fehlerfunktion, der Beispiele und der Kriterien zur Modellauswahl. Zum Stand der Forschung in diesen Punkten siehe den Abschnitt 2.7. Ich deute zur leichteren Einordnung der späteren Diskussion hier die Bedeutung jedes dieser Ansatzpunkte in einem separaten Unterabschnitt schon vorab an, um klarzumachen, daß die Frage der Modellauswahl eine besonders interessante ist. Im Rahmen dieser Arbeit werde ich mich folglich auf Modellauswahlmethoden beschränken und für alle anderen Aspekte Standardlösungen verwenden.

2.5.1 Problemrepräsentation

Offensichtlich ist die Art, in der das Lernproblem dem neuronalen Netz dargestellt wird, von großer Bedeutung dafür, wie leicht und wie gut es gelöst werden kann. Dies zeigt die folgende Überlegung: Klarerweise ist zur Herstellung einer bestimmten Repräsentation der Daten aus den wie auch immer gearteten Rohdaten eine *Vorverarbeitung* nötig (ggf. die Identität). Die Lösung jedes Problems ist leicht, wenn als Bestandteil dieser Vorverarbeitung bereits das gewünschte Ergebnis berechnet und als Teil der Eingabe dem Netz angeliefert wird. Andererseits könnte eine Vorverarbeitung alle Information, die zur Berechnung des Resultats benötigt wird, entfernen. Zwischen diesen Extremfällen gibt es Repräsentationen mit verschiedenstem Schwierigkeitsgrad der Lernaufgabe des Netzes. Eine günstige Problemrepräsentation stellt also eine besonders erfolgversprechende Form von Bias dar.

Insbesondere stellt sich nach gegebener Attributauswahl die Frage, wie ein nominales, ordinales oder kardinales Attribut in der numerischen Form, die als Eingabe für ein Netz geeignet ist, kodiert werden soll. Ein interessantes Einzelproblem ist dabei die Darstellung *fehlender* Attributwerte. Die meisten dieser Fragen bedingen eine Abwägung zwischen der Einfachheit der Repräsentation und der Anzahl von nötigen Parametern. Einfache Darstellungen, beispielsweise 1-aus-n für ein nominales Attribut mit n verschiedenen Werten, sind vom Netz leicht auszuwerten und erleichtern insofern das Lernen, bedingen aber meist eine große Anzahl von Parametern, was unerwünschterweise die Varianz erhöht.

2.5.2 Fehlerfunktion

Zur Diskussion der Bedeutung von Fehlerfunktionen müssen wir unterscheiden zwischen kontinuierlichen Approximationsaufgaben und diskreten Klassifikationsaufgaben.

Bei Approximationsaufgaben legt die Fehlerfunktion fest, wie Abweichungen von der zu lernenden Funktion gewichtet werden sollen. Die Fehlerfunktion ist zugleich das einzige unmittelbare quantitative Qualitätsmaß für die Güte eines Netzes. Aus der statistischen Theorie können einige Aussagen über die Auswirkungen verschiedener Fehlerfunktionen im Hinblick auf die resultierende Verteilung der Fehler gemacht werden; so führt beispielsweise die quadratische Fehlerfunktion zu einem Schätzer, der normalverteilte Fehler macht. Trotz dieser Aussagen liegt für praktische Anwendungen oftmals nicht klar auf der Hand, was die optimale Fehlerfunktion ist.

Bei Klassifikationsaufgaben ist die Situation noch komplizierter. Hier gibt es ein von der Fehlerfunktion unabhängiges Qualitätsmaß, nämlich die resultierende Klassifikationsleistung. Unterschiedliche Fehlerfunktionen haben im Hinblick auf dieses Maß unterschiedliche Nachteile. So sind viele Fehlerfunktionen nicht monoton in der Klassifikationsleistung, andere führen zu sehr langsamer Konvergenz des Verfahrens oder konvergieren für manche Probleme überhaupt nicht zu einer Lösung, auch wenn eine solche existiert. Man beachte, daß häufig eine Klassifikationsaufgabe als Approximationsaufgabe betrachtet werden sollte, da die gewünschte Ausgabe nicht nur die vermutete Klasse sein soll, sondern eine Schätzung der a-posteriori Wahrscheinlichkeit für jede Klasse [294]; diese Sicht nehmen wir auch in dieser Arbeit stets ein. Die Wahrscheinlichkeitsinterpretation der Ausgaben ist aber nicht bei allen für Klassifikationsprobleme sinnvollen Fehlerfunktionen gegeben. Es gibt bisher kein einfaches Verfahren, das angibt, wie man für ein gegebenes Problem ermittelt, welche Fehlerfunktion zu bevorzugen ist; die bisherigen Ergebnisse beschreiben solche Kriterien allenfalls für Familien von Fehlerfunktionen und für Probleme, deren Eigenschaften im Voraus bekannt sind.

2.5.3 Beispielauswahl

Offensichtlich läßt sich ein Netz besonders schnell trainieren, wenn die verwendeten Trainingsbeispiele so ausgesucht werden, daß sie ein Maximum an Information liefern. Eventuell kann sich solche Beispielauswahl auch positiv auf die Generalisierungsfähigkeit des Netzes auswirken, wenn, grob gesagt, die Berücksichtigung zusätzlicher Beispiele unglückliche „Beulen“ in der vom Netz realisierten Funktion verursachen würde. Dies gilt insbesondere (aber nicht nur) für Beispiele, die als Ausreißer zu betrachten sind. Wenn genügend Beispiele vorhanden sind oder beliebige Beispiele beschafft werden können (*query learning*), kann also die gezielte Beispielauswahl eine Verbesserung des Lernergebnisses oder zumindest eine Vereinfachung des Lernens bewirken.

2.5.4 Modellauswahl

Auch bei gegebener Problemrepräsentation, Fehlerfunktion und Beispielauswahl gibt es noch eine Vielzahl von Möglichkeiten zur Modellauswahl. Jedes Verfahren zur Modellauswahl bringt einen bestimmten Bias in das Lernen ein. Diese Möglichkeiten betreffen das Lernverfahren selbst und die darin verwendete Klasse von neuronalen Netzen.

Im Lernverfahren kann bei gegebener Topologie des Netzes auf verschiedene Weise eine Regularisierung vorgenommen werden. Dazu wird ein Bewertungsterm eingeführt, der bestimmte Konfigurationen von Gewichten bevorzugt und den Gradientenabstieg entsprechend beeinflußt. Dieser Term arbeitet in Konkurrenz zur Fehlerfunktion, deshalb gibt es bei Regularisierungsverfahren immer einen (schwierig einzustellenden) Parameter, der die relative Gewichtung zwischen Fehler und Regularisierung angibt. Die meisten Regularisierungsterme haben auf die eine oder andere Art das Ziel, die effektive Komplexität des Netzes zu minimieren. Ein Beispiel für einen solchen Regularisierungsterm ist die Bevorzugung von Gewichten mit kleinem Betrag, was die Aktivierungsfunktionen möglichst in ihrem annähernd linearen Bereich hält.

Ein noch weitaus größerer Raum zur Modellauswahl steht zur Verfügung, wenn dem Lernverfahren erlaubt wird, auch die Topologie des Netzes zu verändern. Solche Veränderungen können das Entfernen von Knoten oder Verbindungen sein (*subtraktive Verfahren*) oder auch deren Hinzufügen (*additive Verfahren*). Die Gesamtheit der additiven und subtraktiven Verfahren nennen wir *konstruktive Verfahren*. Natürlich können additive und subtraktive Verfahren auch kombiniert werden. In beiden Fällen sind die Verfahren bestrebt, eine Topologie zu finden, die dem Lernproblem besonders gut angemessen ist. Subtraktive Verfahren (*Beschneidungsverfahren*, *pruning*) beginnen mit einem „ausreichend großen" Netz und entfernen daraus Ressourcen, die als überflüssig angesehen werden. Additive Verfahren beginnen mit einem zu einfachen Netz und fügen schrittweise Knoten und Verbindungen hinzu, damit das Netz die Aufgabe lösen kann. Bei beiden Ansätzen und auch bei Kombinationen davon lauten die Fragen, die für die Topologiemodifikationen beantwortet werden müssen:

1. Wann soll modifiziert werden?
2. Wo und wie soll modifiziert werden?
3. Wie stark soll modifiziert werden?

Obwohl zweifellos in der richtigen Beantwortung dieser Fragen ein erhebliches Potential zur Verbesserung im Vergleich zu nichtkonstruktiven Verfahren steckt, sind die Antworten noch weitestgehend unbekannt, weil die Dynamik des Lernprozesses in Netzen mit verborgenen Knoten noch kaum verstanden wird.

2.5.5 Sonstiges

Neben diesen vier Hauptaspekten gibt es noch weitere Punkte, die mehr technischer Natur sind. Als prominentestes Beispiel sei hier die Vermeidung lokaler Minima genannt, die bei Gradientenabstiegsverfahren nicht-optimale Resultate bewirken. Möglichkeiten zur Bekämpfung lokaler Minima sind einerseits globale oder stochastische Suchverfahren und andererseits Transformationen des Fehlerraums, die diesen so „verbiegen", daß er weniger lokale Minima aufweist. Auch die Effizienz von Abstiegsverfahren ist von Bedeutung, da es die Möglichkeiten zum Experimentieren erheblich einschränkt, wenn jedes einzelne Training zu lange dauert. Neben der Effizienz des Verfahrens selbst kann natürlich die Geschwindigkeit seiner Ausführung verbessert werden, was im zweiten Teil dieser Arbeit durch den Einsatz von Parallelrechnern getan wird.

2.6 Stand der theoretischen Forschung

Dieser Abschnitt gibt einen Überblick über die fundamentalen theoretischen Ergebnisse, die über das Lernen in neuronalen Netzen bekannt sind. Einige dieser Ergebnisse gelten für Lernverfahren aller Art, nicht nur für neuronale. Da die Darstellung nur als Überblick gedacht ist, sind viele der Ergebnisse sehr informell und teilweise vereinfacht wiedergegeben. Für einen genaueren Überblick siehe zum Beispiel [18, 337]. Dieser Abschnitt soll verdeutlichen, daß die Suche nach guten Lernverfahren nur wenig auf theoretisch abgesicherten Grundlagen aufbauen kann: Wie wir sehen werden, sind die meisten der theoretischen Ergebnisse für die praxiswichtigen Fragestellungen trivial oder nicht relevant oder nicht gültig.

Ich beschränke mich in der nachfolgenden Darstellung vorwiegend auf Resultate, die die Grenzen des maschinellen Lernens beschreiben. Der erste Teilabschnitt (2.6.1) befaßt sich mit der Frage „Was kann ein neuronales Netz überhaupt *darstellen*“, wobei nur die Klasse der von den Netzen realisierten Funktionen betrachtet wird und die Frage des Lernens außen vor bleibt. Der zweite Teilabschnitt (2.6.2) beschreibt dann einige Eigenschaften — und vor allem Grenzen — von Lernverfahren.

2.6.1 Darstellungsmächtigkeit

Das wohl am häufigsten zitierte Resultat zur Darstellungsmächtigkeit von neuronalen Netzen besagt, daß jede beliebige stetige Funktion mit einem neuronalen Netz mit einer verborgenen Schicht beliebig genau approximiert werden kann, wenn das Netz genügend viele verborgene Knoten mit (beispielsweise) Standard-Sigmoid-Aktivierungsfunktion besitzt [168]. Tatsächlich muß es nicht unbedingt die Standardsigmoidfunktion sein; vielmehr ist fast jede beschränkte Funktion eine solche *universelle Aktivierungsfunktion*, wie die folgenden zwei hinreichenden Bedingungen angeben.

Sei f lokal Riemann-integrierbar, d.h. stetig bis auf eine Menge von Maß Null, und beschränkt auf jeder abgeschlossenen Teilmenge, dann ist f eine universelle Aktivierungsfunktion dann, und nur dann, wenn f kein Polynom ist [214].

Die folgende Bedingung ist einfacher, allerdings auch schärfer: Jede stetige, nichtkonstante, beschränkte Funktion ist eine universelle Aktivierungsfunktion [166]. Für weitere Resultate zur Approximierbarkeit siehe [167].

Allerdings besagt dieses Resultat über universelle Approximationsfähigkeit nichts darüber, wieviele verborgene Knoten im Einzelfall nötig sind, um die Approximation auf eine bestimmte Genauigkeit zu leisten. Solche Aussagen machen *Kapazitätsmaße* wie die folgenden.

Sei A eine Klasse von neuronalen Netzen und T eine Menge von Beispielen; ferner sei $E(a,T)$ der summierte Fehler des Netzes $a \in A$ über alle Beispiele aus T. Dann nennen wir T *ladbar* nach A, wenn gilt

$$\inf_{a \in A} E(a,T) = 0,$$

d.h. wenn sich der Fehler beliebig klein machen läßt. Wir definieren nun als die *Kapazität* $c(A)$ der Netzklasse A diejenige Zahl, für die jede Menge T mit höchstens $c(A)$ Beispielen nach A ladbar ist und mindestens eine Menge T mit $|T| = c(a) + 1$ nicht.

Für dieses Kapazitätsmaß haben wir die folgenden asymptotischen Aussagen [337]: Sei A die Klasse der vollverbundenen Netze mit einer verborgenen Schicht und der Anzahl $W(A)$ Gewichten. Dann gilt für Netze A mit Schwellwert-Aktivierungsfunktion in den verborgenen Knoten und lineare Aktivierungsfunktion in den Ausgangsknoten

$$\liminf_{n \to \infty} \frac{c(A)}{W(A)} = 1/3$$

und für A mit sigmoiden Aktivierungsfunktionen, die an mindestens einer Stelle differenzierbar sind mit Ableitung ungleich Null

$$\liminf_{n \to \infty} \frac{c(A)}{W(A)} \geq 2/3.$$

Für je 2 zusätzlich zu ladende Beispiele werden also asymptotisch 6 bzw. höchstens 3 weitere Parameter benötigt. Für Klassifikationsnetze mit Schwellwertfunktion am Ausgang gelten ähnliche Resultate, jedoch können hier asymptotisch mit geeigneten Aktivierungsfunktionen der verborgenen Knoten sogar unendlich viele Beispiele pro Parameter geladen werden.

Leider gilt dieses Resultat wie gesagt nur asymptotisch, so daß sein praktischer Wert wiederum gering ist. Genauere Abschätzungen gibt es für das folgende Kapazitätsmaß.

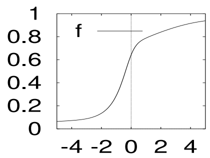
Wir betrachten hier nur Abbildungen in die Menge $\{0, 1\}$, also die binäre Klassifikation. Sei also eine Klasse A von Funktionen und eine Menge T von Beispielen $(\mathbf{x}_i, y_i)$ gegeben; sei X die Menge der Eingaben $\mathbf{x}_i$ der Beispiele aus T. Wir sagen A *erschüttert* (*shatters*) X, wenn zu jeder möglichen Zuweisung von Ausgaben y_i zu den Eingaben X eine Funktion $a \in A$ gibt, die diese Ausgaben erzeugt. Anders ausgedrückt beschreiben die y_i eine *Dichotomie* von X, und A erschüttert X, wenn die Funktionen in A jede mögliche Dichotomie darstellen können.

Gibt es eine größte Zahl d, für die es mindestens eine Menge von Eingaben X der Kardinalität d gibt, so daß A diese Menge X erschüttert, so nennen wir d die *Vapnik-Chervonenkis-Dimension* oder kurz *VC-Dimension* von A, geschrieben $VC(A)$. Gibt es keine solche Zahl, ist die VC-Dimension von X unendlich. Der Begriff der VC-Dimension wurde aufgrund von [379] geprägt. Beschrieb $c(A)$ eine Art Mindestkapazität für das Lernen von Funktionen, so beschreibt $VC(A)$ die untere Grenze für eine Art Höchstkapazität für das Lernen von Mengen oder, anders betrachtet, von *Konzepten*.

Für dieses Kapazitätsmaß gibt es eine Reihe recht genauer Aussagen. So hat ein einfaches Perceptron mit W Gewichten und einem Schwellwert eine VC-Dimension von genau $W + 1$ [97, 18]. Eine der meistzitierten Arbeiten [35] auf diesem Gebiet gibt eine Grenze für mehrschichtige Perceptrons A mit Schwellwertfunktionen an, die insgesamt W Gewichte und Schwellwerte und n Berechnungsknoten haben: $VC(A) < 2W \log_2(eN)$, wobei e die Basis des natürlichen Logarithmus ist. Eine etwas schwächere Fassung dieser Aussage lautet $VC(A) < 6W \log_2 W$. Beide Schranken sind asymptotisch optimal, wie in [229] nachgewiesen wird. Sie machen eine positive Aussage über die Leistungsfähigkeit von neuronalen Netzen: Daß die VC-Dimension superlinear in der Zahl der Parameter wächst, heißt sozusagen, daß das Ganze mehr ist als die Summe seiner Teile; das Zusammenwirken einfacher Elemente läßt eine höhere Stufe der Berechnungsmächtigkeit entstehen.

Die Mächtigkeit von Netzen mit anderen Aktivierungsfunktionen kann sogar noch wesentlich höher sein. Die Arbeit [336] beweist für ein spezielles Netz mit 2 Eingängen und 2 verborgenen Knoten mit der etwas exotischen Aktivierungsfunktion

$$f(x) = \frac{1}{\pi} \arctan x + \frac{cos x}{7 + 7x^2} + \frac{1}{2}$$



daß es die VC-Dimension unendlich hat! Für Netze mit Standardsigmoidfunktionen weist [231] nach, daß die VC-Dimension endlich ist, die sich aus dem Beweis ergebende und bisher beste bekannte Schranke ist jedoch doppelt exponentiell in der Anzahl W der Parameter [18].

Es gibt andere Kapazitätsmaße, die die VC-Dimension auf den Fall stochastischer Konzepte oder den Fall kontinuierlicher reeller Ausgaben erweitern, zum Beispiel die *Graphdimension* [264] oder die *Pseudodimension* (*Pollard-Dimension*, *kombinatorische Dimension*) [284] und für die ähnliche Resultate bekannt sind; [18] gibt einen Überblick.

Was aber haben wir denn nun eigentlich davon, wenn wir die VC-Dimension eines Netzes kennen? Nun, aus der VC-Dimension lassen sich Schranken berechnen, die angeben, wieviele Beispiele nötig sind, um eine gewisse Generalisierungsleistung garantieren zu können. Dies wird unter anderem im folgenden Abschnitt beschrieben.

2.6.2 Generalisierung und Lernkomplexität

In [376] wird ein konzeptueller Rahmen zur Beschreibung von Lernsituationen eingeführt, innerhalb dessen sich quantitativ beschreiben läßt, was es bedeutet, daß ein Lernverfahren eine Hypothese liefert, die eine gewisse Qualität hat. Ich diskutiere nun einige der Ergebnisse, die aus diesem Ansatz entwickelt wurden.

Wir betrachten wieder, wie bei der VC-Dimension, den Fall, daß es gilt, anhand einer Menge T von positiven und negativen Beispielen ein Konzept zu lernen. Die Beispiele sind also Paare $(\mathbf{x}_i, y_i)$ mit $y_i \in \{0, 1\}$. Das von T beschriebene Konzept c stammt aus einem Konzeptraum C, das betrachtete Lernverfahren L liefert eine Hypothese $L(T)$ über Konzepte aus C aus einem Hypothesenraum H. Wir nennen L auch ein (C, H)–Lernverfahren. Im Falle von neuronalen Netzen wäre der Hypothesenraum die Menge aller Abbildungen, die diejenigen neuronalen Netze realisieren, die vom Lernverfahren erzeugt werden können. Seien die Beispiele gemäß einer Wahrscheinlichkeitsverteilung P verteilt, dann bezeichnen wir als den Fehler $err_P(h, c)$ einer Hypothese h im Hinblick auf ein Konzept c die Wahrscheinlichkeit, daß ein gemäß P gezogenes Beispiel von h falsch klassifiziert wird, also

$$err_P(h, c) = P(\{x | h(x) \neq c(x)\})$$

Ein Lernverfahren heiße *konsistent*, wenn für jede Beispielmenge T die Hypothese $L(T)$ alle Beispiele aus T korrekt klassifiziert.

Wir nennen nun ein (C, H)–Lernverfahren L ein *PAC-Verfahren*, wenn es für alle positiven ϵ und δ eine Beispielmengengröße $m_L(\delta, \epsilon)$ gibt, mit der für alle $c \in C$ und alle Verteilungen P auf den Beispielmengen gilt, daß

$$|T| \geq m_L(\delta, \epsilon) \implies P(\{T \subseteq c : err_P(L(T), c) > \epsilon\}) < \delta$$

PAC steht dabei für *probably approximately correct*, also „wahrscheinlich ungefähr richtig“. Die Bezeichnung bezieht sich auf die doppelte Relativierung durch erstens den Genauigkeitsparameter ϵ und zweitens den Zuverlässigkeitsparameter δ. Die Definition besagt, daß ein Verfahren L genau dann PAC ist, wenn man für jede gewünschte Genauigkeit $1 - \epsilon$ und Zuverlässigkeit $1 - \delta$ der Hypothese $L(T)$ eine feste Schranke für die Anzahl von Beispielen angeben kann, die L benötigt, um eine Hypothese zu finden, die mindestens mit der angegebenen Zuverlässigkeit die gewünschte Klassifikationsgenauigkeit erreicht oder übertrifft. Das Interessante an der Definition ist, daß diese Schranke für die Beispielanzahl nicht vom Zielkonzept c und auch nicht von der Verteilung P abhängt. Wir nennen $m_L(\delta, \epsilon)$ auch die *Beispielkomplexität* von L.

Die gute Nachricht ist nun, daß die VC-Dimension eines Hypothesenraums H benutzt werden kann, um Schranken für m_L anzugeben; PAC-Lernen wird also von der VC-Dimension charakterisiert. Das globale Ergebnis lautet wie folgt: Wenn C in H enthalten ist und H eine endliche VC-Dimension hat, dann ist jedes konsistente Lernverfahren L auch PAC. Eine obere Schranke für m_L lautet

$$m_L(\delta, \epsilon) \leq \frac{8}{\epsilon} \left(\ln \frac{4}{\delta} + VC(H) \cdot \ln \frac{48}{\epsilon} \right)$$

wobei die Konstanten noch verbessert werden können. Eine untere Schranke lautet

$$m_L(\delta, \epsilon) > \max \left(\frac{VC(C) - 1}{32\epsilon}, \frac{1}{\epsilon} \ln \frac{1}{\delta} \right)$$

für alle $\epsilon \leq 1/8$ und $\delta \leq 1/100$ (zitiert nach [18]). In [35] wird für neuronale Netze mit Schwellwertneuronen eine ähnliche Aussage auch für nicht konsistente Lernalgorithmen gemacht; hier ist eine Genauigkeit von $1 - \epsilon^2$ auf den Lernbeispielen ausreichend. Wie bei der VC-Dimension gibt es ferner auch für die Beispielkomplexität Verallgemeinerungen auf stochastische Konzepte und auf Netze mit reellen Ausgaben, siehe z.B. [18].

Diverse schlechte Nachrichten folgen jedoch auf dem Fuße: Erstens lassen sich diese Ergebnisse wegen fehlender praktikabler Schranken für die VC-Dimension überhaupt nicht auf neuronale Netze mit Standardsigmoidfunktionen anwenden.

Zweitens ergibt eine Proberechnung für ein Netz mit Schwellwertneuronen, daß die Aussagen dieser Theorie recht deprimierend sind. Nehmen wir ein Netz mit VC-Dimension 10000, was einigen Hundert Parametern entspricht. Ferner nehmen wir $C = H$ an und setzen die gewünschte Genauigkeit auf 90% (also $\epsilon = 0,1$) und die Zuverlässigkeit auf 99% (also $\delta = 0,01$). Dann ergeben die obigen Formeln als Schranken für m_L

$$3124 < m_L \leq 4939508$$

Wir brauchen also garantiert mehr als 3000 und weniger als 5 Millionen Beispiele. Selbst die untere Schranke ist schon recht hoch, ein Tribut an die Tatsache, daß die Aussage frei von irgendwelchen Verteilungsannahmen ist.

Drittens sagen die Resultate nichts darüber, ob es überhaupt ein Lernverfahren L gibt, das die m_L Beispiele gut genug lernen kann, geschweige denn, wie ein solches Verfahren funktioniert; einfaches Abspeichern der Trainingsbeispiele ist kein solches Verfahren, weil die dafür nötige Funktionenklasse eine zu hohe VC-Dimension hat.

Viertens kann man meist nicht sicherstellen, daß $C \subseteq H$ ist. Ist dies nicht erfüllt, so ist aber die angegebene obere Schranke für m_L nicht gültig.

Fünftens ist bekannt, daß selbst in den Fällen, in denen ein Lernen der Beispiele möglich ist, der Berechnungsaufwand für das Lernverfahren exponentiell mit der Zahl der Beispiele wachsen kann [48].

All diese Einschränkungen zeigen, daß die Ergebnisse der PAC-Theorie, so beeindruckend sie in ihrer Allgemeinheit auch sind, nur geringen praktischen Nutzen haben. Sie deuten lediglich einige Grenzen der Leistungsfähigkeit an, die allgemeine Lernverfahren haben können.

Eine fundamentale Grenze gibt der folgende Satz an, der aus [96] folgt und explizit in [315] formuliert wurde. Angenommen, eine aus der Trainingsmenge T gelernte Hypothese $L(T)$ klassifiziere q% aller möglichen Beispiele, die *nicht* in der Trainingsmenge vorkommen, richtig. Da eine Zufallsklassifikation 50% richtig klassifiziert, kann man als Generalisierungsleistung von $L(T)$ den Wert $p - 50$ bezeichnen. Die Generalisierungsleistung kann also sehr wohl negativ sein — $L(T)$ hat etwas falsches gelernt. Nun gilt folgende Aussage: *Die Summe der Generalisierungsleistung über alle möglichen Konzepte ist für jedes mögliche Lernverfahren exakt gleich Null.* Es gibt also kein universelles Lernverfahren, das für jedes Konzept Hypothesen findet, die besser als Zufallsklassifikation sind. Ferner führt jede Änderung eines Lernverfahrens, die die Generalisierungsleistung auf einigen Konzepten verbessert, zugleich dazu, daß die Generalisierungsleistung auf anderen Konzepten sinkt. Aus diesem Grund heißt der Satz *Erhaltungssatz* (*conservation law*) des maschinellen Lernen oder auch *No-free-lunch-Theorem*. Als wesentlicher Unterschied zur PAC-Theorie wird hier zur Bestimmung der Generalisierungsleistung die vollständige Menge all derer Beispiele herangezogen, die nicht in der Trainingsmenge waren, während bei PAC die Beispiele zur Bestimmung des Fehlers aus einer Wahrscheinlichkeitsverteilung gezogen werden, so daß auch Beispiele vorkommen können, die bereits in der Trainingsmenge enthalten waren. Man beachte, daß auch unter den Bedingungen des Erhaltungssatzes die *zu erwartende* Generalisierungsleistung über alle *vorkommenden* Konzepte sehr wohl positiv sein kann, weil in der realen Welt nicht alle Konzepte gleich häufig vorkommen. Dies ist schon deshalb klar, weil das Universum, da es endlich ist, nur einen sehr kleinen Teil der denkbaren Konzepte überhaupt darstellen kann[3]. Dennoch macht der Satz klar, daß alle Lernverfahren notwendigerweise nur für spezielle Anwendungsgebiete geeignet sein können, wobei über die Breite dieser Gebiete nichts gesagt wird.

[3] Dieses schöne Argument stammt von Ralph Hartley (hartley@aic.nrl.navy.mil), vorgebracht in einer intensiven Diskussion über den Satz, die im Juli und August 1994 auf der „Machine Learning“ Mailingliste ml@ics.uci.edu stattfand.

Auch diese Sichtweise zeigt, ähnlich wie Abschnitt 2.4, daß das Ziel beim Entwurf von Lernverfahren nicht sein darf, den Stein der Weisen zu finden, sondern lediglich einen Bias in die Verfahren einzubauen, der gut zu den beabsichtigten Anwendungen paßt.

Neben diesen fundamentalen Resultaten über Möglichkeiten und Grenzen des Lernens, gibt es eine Fülle von Einzelresultaten zu spezielleren Fragen des Lernens in neuronalen Netzen. Ich werde im Verlauf der Arbeit auf einige dieser Ergebnisse, die für die Arbeit von Belang sind, noch eingehen.

2.7 Stand der praktischen Forschung

Weit mehr noch als bei den theoretischen Ergebnissen ist es für praktische Forschung über neuronale Netze kaum möglich, einen vollständigen Überblick zu geben. Ich werde daher in den meisten folgenden Unterabschnitten nur wenige repräsentative Ergebnisse erwähnen. Eine gründlichere Behandlung erfährt allein das Gebiet der konstruktiven Lernverfahren, die im separaten Abschnitt 2.8 behandelt werden.

Die folgenden Unterabschnitte behandeln zunächst Methoden zur Beschleunigung von Lernverfahren und zur Vermeidung lokaler Minima und dann die oben bereits angesprochenen Ansatzpunkte Problemrepräsentation, Fehlerfunktionen, Beispielauswahl und Modellauswahl. Wie wir sehen werden, sind einige dieser Punkte schon recht gut untersucht, andere haben nur begrenzte Bedeutung. Diese Beurteilung veranlaßt mich, die konstruktiven Lernverfahren als interessantesten Ansatzpunkt herauszugreifen und in dieser Arbeit zu erforschen. Die dafür relevante verwandte Literatur wird deshalb wie gesagt ausführlich separat vorgestellt.

2.7.1 Beschleunigung von Lernverfahren

Die Backpropagation-Lernregel für Gradientenabstieg benötigt in jedem einzelnen Schritt einen Rechenaufwand, der proportional zur Anzahl der Parameter im Netz und zur Anzahl der Trainingsbeispiele ist. Jede Elementaroperation ist dabei eine Multiplikation (Aktivierungswert mal Gewicht) und eine Addition (Summation der Produkte), ferner sind Berechnungen der Aktivierungsfunktionen und der Fehlerfunktion nötig. Da bis zur Konvergenz viele Schritte gebraucht werden, ist der Rechenaufwand erheblich. Typischerweise dauert ein Lernvorgang mit Backpropagation auf größeren Aufgaben im Bereich von Stunden, das entspricht heute einer Größenordnung von 10^{10} Rechenoperationen.

Angesichts dieser Situation ist es nicht verwunderlich, daß nach Wegen gesucht wird, durch andere Lernregeln das Lernen zu beschleunigen. Dazu können beispielsweise Techniken aus der numerischen Mathematik (unbeschränkte Funktionsoptimierung) herangezogen werden [128]. Einige solche Techniken oder Varianten davon wurden für neuronale Netze wiedererfunden, andere sind echte Innovationen.

Zu den Möglichkeiten der Verbesserung zählen etwa bei den echten Gradientenverfahren die heuristische Anpassung der Lernrate [71, 87, 88, 297] oder die Wahl der optimalen Schrittweite mittels Liniensuche [128]. Ferner können Verfahren verwendet werden, die Information über die Krümmung der Fehleroberfläche mitverarbeiten, anstatt nur ihre Steigung zu betrachten; hierzu gehören beispielsweise das *Moment* [308], Verfahren mit *konjugierten Gradienten* (*conjugate gradient*) [128, 251], sowie die *Quasi-Newton-Verfahren* und das *Newton-Verfahren* [128].

Andere heuristische Abstiegsverfahren weichen von der Gradientenrichtung ab, indem sie beispielsweise für jedes Gewicht eine lokale, veränderliche Lernrate vorsehen [101, 179, 327, 354] oder nur das Vorzeichen des Gradienten betrachten und eine adaptive Schrittweite verwenden [296]. [316] enthält einen Vergleich von 14 verschiedenen Lernregeln zur Verwendung mit MLPs.

Viele dieser Techniken sind für neuronale Netze jedoch in den meisten Fällen bei weitem zu aufwendig. Um das zu verstehen, muß man sich klarmachen, wie komplex die Funktionsterme sind, die beim

Lernen in einem neuronalen Netz minimiert werden müssen. Siehe dazu folgendes triviale Beispiel. Angenommen, die Aufgabe bestehe darin, die Exklusiv-Oder-Funktion (XOR) zu lernen. Wir wollen dazu das in Abbildung 2.8 dargestellte Netz und die quadratische Fehlerfunktion verwenden. Minimiert

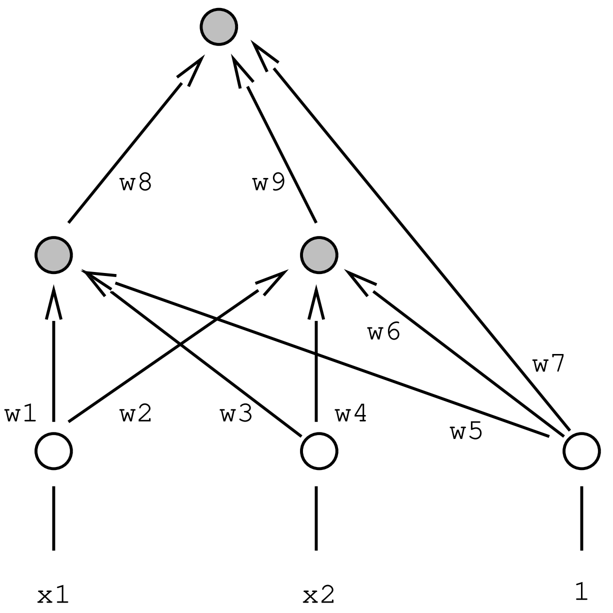


Abbildung 2.8: Neuronales Netz für Exklusiv-Oder-Funktion. Eingangsknoten und Verschiebungsknoten sind weiß, Rechenknoten (also verborgene Knoten und Ausgangsknoten) sind grau dargestellt. Das Netz hat 9 Parameter.

werden soll also die summierte quadratische Abweichung von erhaltener Ausgabe $A(x_1, x_2, w)$ gegenüber der korrekten Ausgabe für die 4 Eingabepaare $(x_1, x_2) \in \{(0,0),(0,1),(1,0),(1,1)\}$, nämlich 0, 1, 1, und 0. Die Optimierung erfolgt durch Anpassung des Gewichtsvektors $\mathbf{w} = (w_1, w_2, \ldots, w_9)$. Mit der Aktivierungsfunktion $\operatorname{sig}(x) = \frac{1}{1+e^{-x}}$ lautet die Ausgabe des Netzes

$$A(x_1, x_2, \mathbf{w}) = \operatorname{sig}(w_8 \cdot A_8(x_1, x_2, \mathbf{w}) + w_9 \cdot A_9(x_1, x_2, \mathbf{w}) + w7) \tag{2.1}$$

wobei

$$A_8(x_1, x_2, \mathbf{w}) = \operatorname{sig}(w_1 x_1 + w_3 x_2 + w_5) \tag{2.2}$$

$$A_9(x_1, x_2, \mathbf{w}) = \operatorname{sig}(w_2 x_1 + w_4 x_2 + w_6) \tag{2.3}$$

Minimiert werden soll nun die Funktion

$$\begin{aligned} F(\mathbf{w}) &= (A(0,0,\mathbf{w}) - 0)^2 + (A(0,1,\mathbf{w}) - 1)^2 + \\ &\quad (A(1,0,\mathbf{w}) - 1)^2 + (A(1,1,\mathbf{w}) - 0)^2 \end{aligned} \tag{2.4}$$

Ausgeschrieben und teilweise zusammengefaßt lautet sie

$$\begin{aligned} F(\mathbf{w}) = &\left(\frac{1}{1 + e^{-(w_8 \frac{1}{1+e^{-w_5}}) + (w_9 \frac{1}{1+e^{-(w_6)}}) + w_7}}\right)^2 + \\ &\left(\frac{1}{1 + e^{-(w_8 \frac{1}{1+e^{-(w_2+w_5)}}) + (w_9 \frac{1}{1+e^{-(w_2+w_6)}}) + w_7}} - 1\right)^2 + \\ &\left(\frac{1}{1 + e^{-(w_8 \frac{1}{1+e^{-(w_1+w_5)}}) + (w_9 \frac{1}{1+e^{-(w_2+w_6)}}) + w_7}} - 1\right)^2 + \\ &\left(\frac{1}{1 + e^{-(w_8 \frac{1}{1+e^{-(w_1+w_3+w_5)}}) + (w_9 \frac{1}{1+e^{-(w_2+w_4+w_6)}}) + w_7}}\right)^2 \end{aligned} \tag{2.5}$$

Diese Formel wäre noch erheblich komplizierter, wenn die einzelnen Ein- und Ausgabewerte nicht immer 0 oder 1 wären. Die Anzahl von Quadrattermen in der Formel entspricht der Anzahl der Beispiele, die Struktur jedes Terms korrespondiert zur Struktur des Netzes.

Dieses Beispiel deutet an, warum so viele Techniken der numerischen Optimierung bei neuronalen Netzen nicht sinnvollerweise verwendet werden können. So erfordert z.B. eine Liniensuche zahlreiche Auswertungen der sehr berechnungsaufwendigen Gesamtfehlerfunktion F für einen einzigen Gradientenabstiegsschritt. Die Newton-Methode erfordert die Inversion der Hessematrix (d.h. der zweiten Ableitung der Fehlerfunktion), deren Größe das Quadrat der Länge des Parametervektors $\mathbf{w}$ ist; für große Netze ist aber bereits das bloße Abspeichern der Hessematrix hoffnungslos zu aufwendig, von der Inversion ganz zu schweigen (siehe aber [250] für eine Vermeidung dieses Problems). Ähnliche Betrachtungen gelten für andere Verfahren.

Die besten Kompromisse scheinen das Verfahren des skalierten konjugierten Gradienten (SCG) [251], sowie das Quickprop-Verfahren [101] und das RPROP-Verfahren [296] zu sein. SCG ist zwar bei optimaler Anwendung oft das effizienteste Verfahren, erfordert jedoch große Sorgfalt, da das Verfahren eine positiv definite Hessematrix annimmt, andernfalls können die Ergebnisse katastrophal sein. Als anderes Extrem ist RPROP gleich effizient wie optimal angewendetes Quickprop, dabei aber sehr unkritisch in der Wahl der Parameter. Diese Eigenschaft macht RPROP zur robustesten und am einfachsten anzuwendenden bekannten Lernregel, weshalb ich es auch in dieser Arbeit einsetze.

Insbesondere wenn die Anzahl der Trainingsbeispiele sehr groß wird, sind aber alle der oben angesprochenen Verfahren unerträglich langsam. Eine Abhilfe schafft hier die Idee des *stochastischen Gradientenabstiegs*: Man berechne den Gradienten nicht im Hinblick auf alle Beispiele, sondern nur im Hinblick auf einige davon (einen *Block*), im nächsten Schritt dann für einige andere, u.s.w. [29]. Wenn die Beispiele einander ähneln, was sie in großen Trainingsmengen in der Regel tun, ist dies eine Annäherung an einen echten Gradientenabstieg; daß überhaupt ein Abstieg stattfindet, hat Wahrscheinlichkeit Eins [396]. Der Vorteil dieses Verfahrens ist, daß es bei Trainingsmengen mit hoher Redundanz sehr viel schneller (gemessen in Rechenzeit) konvergiert als Verfahren, die den totalen Gradienten berechnen [29]. Als weiterer Vorteil führt die stochastische Minimumsuche manchmal dazu, daß Einzugsbereiche lokaler Minima wieder verlassen werden. Diesen Vorteilen stehen jedoch auch Probleme gegenüber: Offensichtlich gibt es in diesem Szenario kein definiertes Fehlerminimum mehr, zu dem das Verfahren konvergieren kann. Diesem Problem muß man durch eine Erhöhung der Blockgröße gegen Ende des Trainings begegnen. Gute Verfahren zur ebenfalls nötigen automatischen Anpassung der Lernrate sind schwer zu finden, siehe [29, 87, 252] für erste Ansätze.

In dieser Arbeit werden wir nur Probleme betrachten, bei denen jedes Beispiel relativ teuer ist, also hohe Kosten für seine Herstellung oder Beschaffung anfallen, und dementsprechend die Zahl der vorhandenen Beispiele nicht sehr groß ist. Deshalb ist die Verwendung von stochastischen Verfahren hier nicht notwendig. Stattdessen besteht die Möglichkeit, auch aufwendigere Lernverfahren einzusetzen, die erheblich mehr Lernschritte benötigen, aber die in dem kleinen Datensatz enthaltene Information besonders gut ausnutzen.

2.7.2 Vermeidung lokaler Minima

Der Fehler eines neuronalen Netzes mit W Parametern auf einer Menge T von Beispielen definiert eine Funktion von W Variablen, die man sich als W-dimensionale Hyperfläche im $(W+1)$-dimensionalen Raum vorstellen kann — man kann es zumindest versuchen. Die Lernregel soll dazu dienen, den tiefsten Punkt dieser Fläche zu finden, also das globale Minimum. Aufgrund der Nichtlinearität der vom Netz berechneten Funktion kann diese Fläche jedoch mehrere lokale Minima haben, zu denen Abstiegsverfahren unbeabsichtigterweise konvergieren können. Welches lokale Minimum von einem Abstiegsverfahren erreicht wird, hängt in erster Linie vom Startpunkt und in zweiter Linie von der Weite und Richtung der einzelnen Schritte im Parameterraum ab; weite Schritte können enge lokale Minima überspringen. Ungünstige Startpunkte können zusätzlich zu sehr langsamer Konvergenz führen [198].

Globale Suchverfahren [128, 359] versuchen im Gegensatz zu Abstiegsverfahren nicht das nächstgelegene lokale Minimum zu finden, sondern das tiefste. Da der Suchraum unendlich groß ist, können aber auch diese Verfahren nicht *garantieren*, daß das globale Minimum gefunden wird. Sie finden aber bei korrekter Anwendung manchmal zumindest tiefere lokale Minima als Abstiegsverfahren. Bei neuronalen Netzen werden vor allem sogenannte *Annealing*-Verfahren (*Abkühlungsverfahren*) verwendet, zum Beispiel das *Simulated Annealing* [193] oder das effizientere *Mean-field Annealing* [42, 43]. Beispiele für die Anwendung dieser Techniken sind die *Boltzmann-Maschine* [308] und das *Alopex* Lernverfahren [375].

Gegen die Verwendung dieser Techniken sprechen die enorm hohen Rechenzeiten und die Schwierigkeiten beim Finden guter Werte für die vom Benutzer einzustellenden Parameter. Die oft nur unwesentlich verbesserten Ergebnisse gegenüber Abstiegsverfahren rechtfertigen den Aufwand meistens nicht.

Ein gänzlich anderer Ansatz versucht, lokale Minima in der Fehleroberfläche zu beseitigen, indem die Fehleroberfläche transformiert wird. Ein Beispiel hierfür ist die in [132] beschriebene Homotopiemethode. Leider lassen sich in der Regel keine theoretischen Aussagen darüber machen, wie wirksam diese Methoden tatsächlich sind — genau wie bei globalen Suchverfahren — da die Eigenschaften der Fehleroberfläche nicht genügend bekannt sind.

Ein nochmals anderer Ansatz geht davon aus, daß ein lokales Minimum beseitigt werden kann, indem dem Netz zusätzliche Freiheitsgrade gegeben werden. Solche Verfahren fügen z.B. einen zusätzlichen verborgenen Knoten ins Netz ein, wenn festgestellt wird, daß das Training vermutlich in einem lokalen Minimum steckt [158]. Es ist auch hier nicht geklärt, wie nützlich die Idee ist.

In der Regel behilft man sich damit, Abstiegsverfahren mehrfach mit unterschiedlichen Startpunkten durchzuführen und die beste gefundene Lösung auszuwählen. Teilweise werden auch innerhalb der Abstiegsverfahren Neustarts durchgeführt, wenn die gefundenen Lösungen den Ansprüchen nicht genügen [101, 285, 380].

2.7.3 Problemrepräsentation

Es ist bekannt, daß neuronale Netze gut mit verteilten Repräsentationen für Eingaben und Ausgaben funktionieren (ebenso wie sie verteilte Repräsentationen der Problemlösung in ihren verborgenen Knoten entwickeln). Siehe z.B. [323, 95]. Bei einer verteilten Repräsentation wird ein einzelnes, insbesondere numerisches Attribut durch mehrere Koeffizienten modelliert, die bei verschiedenen Werten des Attributs unterschiedlich ausfallen, zum Beispiel gemäß einer Glockenkurve, deren Mitte für jeden Koeffizienten woanders liegt. Eine solche Repräsentation ist an die biologische Implementierung der Reizleitung im menschlichen Nervensystem angelehnt [4]. Es gibt allerdings noch keine guten Regeln dafür, wie die Auflösung einer solchen verteilten Repräsentation zu wählen ist. Bei nominalen Attributen ergibt sich die Auflösung kanonisch aus der Anzahl von Werten. Allerdings könnte auch hier eine Verringerung sinnvoll sein, wenn manche der Werte nur selten vorkommen, oder eine Vergrößerung z.B. zur Verwendung von fehlerkorrigierenden Codes [95].

Zur Behandlung fehlender Attributwerte der Eingaben gibt es grundsätzlich zwei Möglichkeiten: Man kann entweder die fehlenden Werte schätzen oder fehlende Werte explizit als solche in der Netzeingabe kodieren. Das Schätzen der fehlenden Werte aus den vorhandenen desselben Beispiels kann mit Hilfe eines weiteren neuronalen Netzes vorgenommen werden, das beispielsweise mit dem EM-Algorithmus [185] oder mit einer Boltzmann-Maschine [308] trainiert wird [126]. Die dabei vorgenommene Maximum-Likelihood-Schätzung ist an ihrem globalen Extremum theoretisch optimal. Eine simplere Version besteht darin, stets den aus den vorhandenen Werten desselben Attributs geschätzten Mittelwert einzusetzen. Da die Auswirkung verschiedener Schätzungen auf den Fehler des Netzes an der Ausgabeseite jedoch nicht vorhergesagt werden kann, ist manchmal in der Praxis dennoch ein anderer Ansatz besser: Eine Methode, die der Betrachtung neuronaler Netze als Lernmaschinen nähersteht, besteht darin, durch einen zusätzlichen Koeffizienten in der Eingaberepräsentation ausdrücklich anzuzeigen, wann ein fehlender Wert vorliegt und das Netz die beste Reaktion selbst lernen zu lassen. Diese Idee kann entweder mit den obigen kombiniert werden, oder man setzt die Koeffizienten der fehlenden Attribute konstant auf Null. Obwohl einige empirische Daten und auch Beiträge aus dem Gebiet der Statistik [222] vorliegen, ist die Frage der Behandlung fehlender Attributwerte noch nicht für alle praktischen Fälle zufriedenstellend geklärt [338, 365, 377].

Besteht der Verdacht, daß eine gewählte Eingaberepräsentation zu viele Koeffizienten enthält, so daß die Varianz zu groß ist, so können Techniken zur Dimensionsreduktion wie *principal components analysis* oder nichtlineare Varianten davon eingesetzt werden [93]. Ein neuronales Netz mit der richtigen Architektur kann solche Reduktionen aber auch von alleine lernen.

2.7.4 Fehlerfunktionen

Die meistverwendete Fehlerfunktion ist der quadratische Fehler (genauer: der mittlere quadratische Fehler). Dafür gibt es zwei Gründe. Zum einen funktionieren manche Verfahren nur für die quadratische Fehlerfunktion, und zum anderen ist der quadratische Fehler optimal, wenn man auf den Daten einen stochastischen Anteil annimmt, der normalverteilt ist oder wenn man ein Netz erhalten möchte, dessen Fehler normalverteilt sind; quadratischer Fehler ist deshalb die normale Wahl für Approximationsaufgaben. Außerdem hat die quadratische Fehlerfunktion den Vorzug, daß sie mit einer 1-aus-n Kodierung bei Klassifikationsproblemen dazu führt, daß das Netz die a-posteriori Wahrscheinlichkeiten der Klassen schätzt [294]. Eine solche Interpretation der Netzausgaben als Wahrscheinlichkeiten ist vor allen dann nützlich, wenn die Ausgaben noch weiterverarbeitet werden müssen; diese Annahme werde ich auch in meiner Arbeit zugrundelegen.

Andererseits gibt es auch andere Verteilungsannahmen. So ist ein natürliches Fehlermaß für binomialverteilte Ausgaben die *Entropie-Fehlerfunktion* (*cross entropy*), die ebenfalls zur Schätzung von a-posteriori-Wahrscheinlichkeiten führt. Manchmal kann für Klassifikationsprobleme mit der Entropiefehlerfunktion eine Lösung gefunden werden, wo das mit quadratischem Fehler nicht gelingt [403].

Die Schätzung von a-posteriori-Wahrscheinlichkeiten wird durch die Verwendung einer *softmax*-Normalisierungsfunktion [58] verbessert, was vor allem bei kleinen Trainingsmengen Vorteile haben kann [115].

Für wieder andere Verteilungsannahmen sind auch wieder andere Fehlerfunktionen optimal. So gehört beispielsweise zu Laplace-verteilten Fehlern die mittlere lineare Abweichung als Fehlerfunktion und zu gleichverteilten Fehlern die maximale lineare Abweichung [117]. Da meist die tatsächliche Verteilung der Fehler auf den Daten nicht bekannt ist, bleibt die Auswahl einer Fehlerfunktion ein heuristisch zu lösendes Problem. Der quadratische Fehler ist aber fast immer ein guter Kompromiß.

Einen anderen Ansatz verfolgen spezielle Fehlerfunktionen für Klassifikationsprobleme. Diese versuchen, einen monotonen Zusammenhang zwischen dem Fehler und dem Klassifikationsfehler herzustellen, damit eine Verringerung des Fehlers, die ja im Training vorgenommen wird, keine Erhöhung des Klassifikationsfehlers nach sich ziehen kann, wie es bei anderen Fehlerfunktionen vorkommt. Dazu verzichten diese Funktionen auf eine gute Annäherung der a-posteriori-Wahrscheinlichkeiten. Eine solche Fehlerfunktion ist die *classification figure of merit* (*CFM*) [142], die jüngst verallgemeinert wurde zu einer Klasse namens *minimum misclassification error* (*MME*) [345]; die letztere Arbeit enthält auch einen Überblick über andere Ansätze in die gleiche Richtung. Der Klassifikationsfehler, der mit solchen Fehlerfunktionen erreicht wird, ist in der Regel kleiner als mit quadratischem Fehler oder mit Entropiefehler. MME-Funktionen können Klassifikationsprobleme mit wesentlich kleineren Netzen lösen als die quadratische Fehlerfunktion [345]. Die *exponentielle Fehlerfunktion* [250] hat eine abgeschwächte Monotonitätseigenschaft (*soft-monotone error*) und soll sowohl viele Beispiele richtig klassifizieren, als auch den restlichen Fehler gleichmäßig balancieren; leider hat diese Funktion zwei freie Parameter, deren Wahl schwierig ist. Hauptnachteil der am Klassifikationsfehler orientierten Funktionen ist neben dem Verlust der Wahrscheinlichkeitsinterpretation oft eine starke Verlangsamung des Trainings. Für genügend große Datenmengen liefern auch der quadratische Fehler und der Entropiefehler optimale Klassifikation, weil dann die Schätzung der a-posteriori-Wahrscheinlichkeiten genau wird [294].

2.7.5 Beispielauswahl

Die Beispielauswahl zur Unterdrückung von „Ausreißern" ist ein in der Statistik lange studiertes Problem. Sie entspricht dem (in der Regel unüberwachten) Lernen eines Zweiklassenproblems, Ausreißer und Nichtausreißer, und enthält insofern alle Schwierigkeiten des Lernens generell, einschließlich des Bias/Varianz-Dilemmas („Des Einen Ausreißer ist des Anderen Datenpunkt"). Es gibt jedoch Verfahren, die in vielen Fällen gut funktionieren, vorausgesetzt es sind genügend viele Beispiele vorhanden, z.B. [259, 307].

Erst seit kurzem wird die Frage erforscht, wie man aus einer großen Menge von Beispielen eine Teilmenge so auswählen kann, daß das Lernergebnis dennoch gut ist. Diese Frage ist vor allem von Bedeutung, um die Trainingszeit in erträglichen Grenzen zu halten in Anwendungsgebieten, in denen man von den verfügbaren Beispielmengen geradezu erschlagen wird (z.B. Maschinensehen oder Spracherkennung). Die *aktive Beispielauswahl* berücksichtigt dabei den Zwischenzustand des trainierten Netzes, um inkrementell möglichst nützliche Beispiele zu einer allmählich wachsenden Trainingsmenge hinzuzunehmen [282, 283] oder um die Beispiele mit hohem Fehler häufiger zu trainieren als andere [64]. Unter günstigen Umständen läßt sich mit diesen Verfahren der Trainingsaufwand um eine Größenordnung reduzieren. Das erstere Verfahren funktioniert schlecht auf Problemen, bei denen einzelnen Beispiele nur wenig Information liefern, das zweite Verfahren führt zu Verzerrungen der a-priori-Wahrscheinlichkeiten der Beispiele, was das Ergebnis verschlechtern kann. Beide Verfahren lassen sich außerdem nicht gut auf Probleme mit stochastischem Anteil anwenden. In [416] werden beide Ideen kombiniert, allerdings in recht naiver Weise.

2.7.6 Modellauswahl

Zur Verbesserung der Modellauswahl über einen einfachen Abstieg auf der Fehlerfunktion hinaus stehen für gegebene Problemrepräsentation, Fehlerfunktion und Trainingsmenge grundsätzlich folgende Möglichkeiten zur Verfügung:

1. Regularisierung
2. Quervalidation
3. Konstruktive Lernverfahren
4. Kombination von Netzen

Diese Möglichkeiten können im Prinzip zum großen Teil miteinander kombiniert werden, was bei der Erforschung allerdings einen erschreckend großen Suchraum aufspannt. Aus diesem Grund ist bislang meist nur die Quervalidation mit anderen Methoden gekoppelt worden. Da konstruktive Lernverfahren ein Thema dieser Arbeit sind, ist ihnen ein eigener Abschnitt gewidmet. Die übrigen Ansätze besprechen wir nun.

Der Grundgedanke bei Regularisierungsverfahren ist, daß nicht alle Lösungen gleichermaßen gewünscht sind, sondern solche mit geringer Komplexität bevorzugt werden sollten, weil sie besser generalisieren. In der Statistik wird dies *Glättung* (*smoothing*) genannt. Die Regularisierung besteht meist darin, daß das jeweilige Komplexitätsmaß als Term zur Fehlerfunktion hinzuaddiert wird und folglich das Netz darauf trainiert wird, einen Kompromiß zwischen Komplexität und Fehler zu finden. Ein solcher Regularisierungsterm reduziert in der Regel die Varianz. Offensichtlich liegt ein Problem des Ansatzes darin, daß die beste Abwägung zwischen Fehler und Komplexität nicht bekannt ist. Die Lernverfahren haben also einen zusätzlichen freien Parameter, der die Gewichtung des Regularisierungsterms angibt und dessen Wahl den Erfolg des Lernens kritisch beeinflußt; die richtige Wahl des Parameters hängt vom jeweiligen Problem ab. Reine Regularisierungsverfahren sind deshalb keine automatischen Lernverfahren. Das einfachste und meistverwendete Regularisierungsverfahren ist der exponentielle Gewichtsabfall (*weight decay*), bei dem der Betrag der Gewichte als Komplexitätsmaß herangezogen wird. Während des Trainings werden die Gewichte in jedem Schritt mit einem Faktor kleiner 1 (z.B. 0,99999) multipliziert; dies entspricht dem Hinzufügen des Terms $\sum_{i,j} w_{ij}^2$ zur Fehlerfunktion. Das Verfahren macht die implizite Annahme, daß die Beträge der Gewichte im angestrebten Netz normalverteilt sind. Diese Annahme ist oftmals falsch [39], so daß Gewichtsabfall nicht immer positive Auswirkungen hat. Dennoch gibt es zahlreiche Erfolgsmeldungen über den Einsatz des Verfahrens, z.B. [203, 117]; stets ist jedoch eine experimentelle Anpassung des Gewichtungsparameters nötig[4]. Aus anderen Annahmen lassen sich andere Regularisierungsterme herleiten, z.B. [59, 77, 117, 145, 390, 391, 392]. Das Hauptproblem von Regularisierungsverfahren ist stets die richtige Wahl des Gewichtungsparameters; [241, 242] weist nach, daß eigentlich sogar die Verbindungen zwischen jedem Paar von Schichten einen eigenen Gewichtungsparameter haben sollten. Das Überlagern der Beispiele mit Rauschen während des Trainings wirkt ebenfalls als Regularisierung [326]. [270] beschreibt ein Regularisierungsverfahren, das die Verbindungen während des Lernens zu Gruppen zusammenfaßt, die *soft weight sharing* betreiben, d.h. ähnliche Gewichte haben. Die Verteilung der Gewichte wird dabei als eine Mixtur von n (z.B. 8) Gaußverteilungen modelliert, deren Mittelwerte, Varianzen, und Gewichtungen gelernt werden; einziger freier Parameter ist n. Nachteil des Verfahrens ist sein relativ hoher Berechnungsaufwand, der etwa um den Faktor n über normalem Gradientenabstieg liegt. Außerdem arbeitet das Verfahren mit globaler Information: die Parameter aller n Gaußverteilungen müssen bei jedem Gewicht bekannt sein. Dadurch verletzt das Verfahren einen Grundgedanken des neuronalen Lernens.

Quervalidation (*cross validation*) ist ein Verfahren zur Beurteilung der Generalisierungsfähigkeit eines Netzes. Betrachtet wird der Fehler auf einer Menge von Beispielen, die nicht zum Training des Netzes verwendet wurden, der sogenannten *Validationsmenge*. In der Statistik, aus der die Idee stammt [342], wird die Quervalidation iterativ zur optimalen Einstellung von Gewichtungsfaktoren für Regularisierungsterme verwendet. Im Prinzip kann man es für neuronale Netze genauso machen, jedoch stehen hier die langen Trainingszeiten für jeden einzelnen Versuch im Wege. Eine der wenigen wirklich wichtigen Innovationen der Neuroinformatik ist jedoch die Art, wie Quervalidation hier eingesetzt wird: Man fahre mit dem Gradientenabstieg nur solange fort, bis der Fehler auf der Validationsmenge wieder zunimmt. An dieser Stelle wird das Training vorzeitig, nämlich vor der Konvergenz des Gradientenabstiegs, abgebrochen (*frühes Stoppen*, *early stopping*). [256] und [207] sind frühe Veröffentlichungen, die dieses Prinzip beschreiben. Das Verfahren wirkt effektiv wie ein Regularisierungsverfahren, das die Ausdifferenzierung der Gewichte begrenzt, kommt jedoch ohne expliziten Regularisierungsterm und damit auch ohne Gewichtungsfaktor aus. Inzwischen liegen überwältigende empirische Beweise der Wirksamkeit des frühen Stoppens vor, z.B. [117]. In den meisten Fällen führt das Trainieren eines „zu großen" Netzes mit frühem Stoppen zu besseren Ergebnissen, als das Trainieren eines Netzes der „richtigen" Größe bis zum Minimum; siehe auch das Beispiel in Abschnitt 2.4.2 auf Seite 24.

Konstruktive Lernverfahren verändern während des Lernens die Topologie des neuronalen Netzes, um eine optimal an das Lernproblem angepaßte Architektur zu finden. Diese Verfahren spannen einen sehr großen Raum von Möglichkeiten auf und werden unten im Abschnitt 2.8 behandelt; sie lassen sich mit Regularisierung und Quervalidation koppeln.

Noch weiter gehen Ansätze zur Kombination von Netzen, insbesondere Klassifikatoren. Hier werden mehrere Klassifikatoren so trainiert, daß sie möglichst unterschiedliche Fehler machen. Dies läßt sich beispielsweise durch sehr verschiedene Problemrepräsentationen oder den Einsatz unterschiedlicher (auch nicht-neuronaler) Lernverfahren erreichen. Die Ausgaben mehrerer solcher Klassifikatoren können dann mit Hilfe von Abstimmungsverfahren zu einer gemeinsamen Klassifikation kombiniert werden, wobei sich erstens sehr niedrige Gesamtfehlerraten erreichen lassen und zweitens eine günstige Abwägung zwischen Falschklassifikationsrate und Rückweisungsrate gefunden werden kann [34, 300, 349]. Dieser Gedanke kann auch im Innern eines einzelnen Lernverfahrens realisiert werden [185]. Eine Variante dieses Ansatzes schaltet Netze hintereinander anstatt nebeneinander und versucht, die Fehler jedes Netzes mit dem folgenden Netz gezielt zu korrigieren [407]. Der Kombinationsansatz ist abgewandelt auch für Approximationsaufgaben einsetzbar, seine Wirksamkeit hierfür ist aber weitgehend unerforscht.

[4] Hierbei wird häufig der methodische Fehler gemacht, diese Anpassung unter Betrachtung der jeweils auftretenden Fehler auf der Testmenge vorzunehmen, was die Resultate ungültig macht.

2.8 Konstruktive Lernverfahren

Es gibt eine beträchtliche Zahl von veröffentlichten Vorschlägen über konstruktive Lernverfahren. Einen großen Teil davon werden wir in diesem Abschnitt überblickshaft diskutieren. Der überwiegende Teil dieser Arbeiten hat allerdings erhebliche Mängel in der experimentellen Auswertung des vorgeschlagenen Verfahrens[5]. Um diese Mängel ohne eine lange Diskussion anzudeuten, werde ich bei der Beschreibung der Verfahren stichwortartige Informationen zur vorgenommenen Auswertung geben. Unter dem Stichwort „Beispielprobleme:“ gebe ich an, welche Lernprobleme in der Veröffentlichung über das Lernverfahren zur Auswertung herangezogen wurden. Sehr oft finden sich hier nur Spielzeugbeispiele, z.B. aus dem klassischen Backpropagationtext [308] entnommen, wie das n-bit Paritätsproblem (*Parity*, bei 2 bit: *XOR*), n-bit Bitmuster-Encoder/Decoder (*Encoder*), n-bit Bitmuster-Symmetrieproblem (*Symmetry*), Unterscheidung der Buchstaben T und C in diversen Lagen (*T/C*), Entdeckung von echten Sequenzen von 1-Bits (*two-or-more clumps*), etc. Unter dem Stichwort „Vergleich:“ gebe ich an, welche anderen Lernverfahren als Vergleichsmaßstab in die Auswertung eingehen. Häufig fehlt ein Vergleich gänzlich. Hat die Auswertung darüber hinaus methodische Mängel, wird das gesondert vermerkt. Insbesondere werden häufig die exakten Kriterien für Entscheidungen quantitativer Art im Lernverfahren nicht genannt, z.B. das Kriterium zur Wahl des Zeitpunkts, zu dem eine Ressourcenveränderung vorgenommen wird. Dies notiere ich als „Keine quantitativen Kriterien angegeben“. Alle diese Angaben sollen verdeutlichen, wie selten bisher gründliche empirische Auswertungen für die vorgestellten konstruktiven Lernverfahren publiziert wurden. Diese Situation nehme ich als Anlaß um in meiner Arbeit solche Auswertungen vorzunehmen (Kapitel 4 bis 7). Als Untersuchungsgegenstand wähle ich diejenigen Verfahren, die anhand ihrer Grundidee und der vorliegenden Literatur noch am plausibelsten als gute Verfahren vermutet werden können.

Die folgenden Unterabschnitte diskutieren der Reihe nach erst *additive* Verfahren, die sukzessive Knoten und Verbindungen zu einem anfänglich zu einfachen Netz hinzufügen, dann *subtraktive* Verfahren, die sukzessive Knoten und Verbindungen aus einem anfänglich zu großen Netz entfernen, dann Kombinationen beider Ideen und schließlich konstruktive Lernverfahren, die genaugenommen gar nicht als neuronale Verfahren einzustufen sind. Manche der Verfahren eignen sich nur für Klassifikationsprobleme, andere auch für Approximationsprobleme. Additive Verfahren werden später in meiner Arbeit in Kapitel 5 erforscht, subtraktive Verfahren in Kapitel 6.

Eine Reihe von konstruktiven Verfahren sind für einzelne Anwendungsgebiete spezialisiert, z.B. [163, 164, 393] für maschinelles Sehen, [180] zur syntaktischen und semantischen Analyse natürlichsprachlicher Sätze oder [49, 50] zur Erkennung gesprochener Sprache. Diese bereichsspezifischen Verfahren diskutiere ich nicht. Außerdem behandele ich nur Verfahren, die Perceptron-artige Netze verwenden und ignoriere z.B. Netze aus radialen Basisfunktionen (RBF) wie in [52, 280, 395], Netze mit Vektorquantisierungsmechanismen wie in [9] und rekurrente Netze wie in [16].

2.8.1 Additive Verfahren

Additive Verfahren beginnen mit einem Netz, von dem in der Regel anzunehmen ist, daß es zu klein sei, um das gegebene Problem zu lösen. Anhand einer Beobachtung des Lernfortschritts wird gelegentlich entschieden, das Netz um zusätzliche Knoten und zugehörige Verbindungen zu erweitern. Das Ziel dabei ist, eine (möglichst kleine) Netzarchitektur zu finden, die das Problem gut lösen kann.

Die ersten Vorschläge stammen dabei noch aus der Perceptron-Zeit und arbeiten nur für die binäre Klassifikation (teilweise auf N-Klassen-Probleme erweiterbar). Gallant schlägt 1986 in [124] drei Varianten vor: Man trainiere ein einfaches Perceptron dazu, möglichst viele Beispiele richtig zu klassifizieren, indem man während einer langen Trainingsperiode immer denjenigen Satz von Gewichten „in der

[5] Einige Beiträge erheben gar nicht erst den Anspruch, zu einer guten Generalisierung zu führen.

Tasche behält", der am wenigsten Fehler produziert (*Pocket-Algorithmus*). Dann wandle man den Ausgangsknoten in einen verborgenen Knoten um, indem man einen neuen Ausgangsknoten erzeugt, der außer den normalen Eingaben auch die Ausgabe des vorherigen Ausgangsknotens als Eingabe erhält; dies ist Variante 1, die *Turmkonstruktion* (*tower construction*). Bei Variante 2 werden als Eingaben für den neuen Ausgangsknoten auch die Ausgaben *aller* früheren Ausgangsknoten benutzt, anstatt nur die des letzten (*invertierte Pyramide*, *inverted pyramid construction*). Beide Varianten produzieren also viele verborgene Schichten mit je nur einem Knoten — eine Kaskade. Als Variante 3 schlägt er vor, eine große Zahl von Schwellwertknoten mit zufälligen Gewichten zu erzeugen und diese in einer verborgenen Schicht zusammengefaßt als Eingaben des Ausgangsknotens hinzuzunehmen (*verteilte Methode*, *distributed method*). Beispielprobleme: Keine.

Im *Tiling-Algorithmus* von Mèzard und Nadal [246] wird im Vergleich zur Turmkonstruktion vor dem Zufügen eines neuen Ausgangsknotens erst noch eine Anzahl zusätzlicher Knoten mit den gleichen Eingaben wie der bisherige Ausgangsknoten erzeugt, um auf der so entstehenden späteren verborgenen Schicht insgesamt eine treue Repräsentation (*faithful representation*) zu erzeugen, in der Beispiele mit verschiedener gewünschter Ausgabe des einzigen Ausgangsknotens auch stets unterschiedliche verborgene Repräsentation haben. Dann erst wird ein neuer Ausgangsknoten erzeugt, der alle Ausgaben dieser verborgenen Schicht plus alle normalen Eingaben erhält und damit trainiert wird. Kann dieser das Problem noch immer nicht lösen, beginnt der Zyklus von neuem. Es wird stets nur der zuletzt zugefügte Knoten trainiert, die Gewichte aller anderen sind fest. Beispielprobleme: Parity, zufällige boolesche Funktionen, number of clumps. Vergleich: Keiner.

Nadal stellt kurz darauf eine Vereinfachung des Tiling-Algorithmus vor, der eine Wiedererfindung von Gallants Turmkonstruktion ist, und stellt eine zu Tiling vergleichbare Leistung fest [263]. Das *Extentron* von Baffes und Zelle [23] ist eine nochmalige Neuerfindung der Turmkonstruktion, erstmalig mit Verallgemeinerung auf N Ausgaben und mit einer halbwegs brauchbaren Auswertung: Auf dem *soybean* Problem (Diagnose von 15 verschiedenen Krankheiten an Sojabohnenpflanzen) ist das Extentron signifikant schlechter als das symbolische Lernverfahren ID3, bei der Klassifikation von DNA Promotersequenzen signifikant besser als dieses und insignifikant schlechter als ein Backpropagation-Netz. Letzteres Problem hat nur 106 Beispiele, aber 228 Eingabekoeffizienten; es ist unklar, ob und wieweit eine Überanpassung vermieden wurde.

Eine Verbesserung des Tiling-Algorithmus bringt der *Upstart-Algorithmus* von Frean [123], ebenfalls für binäre Klassifikation mit Netzen aus Schwellwertknoten. Er erzeugt das Netz vom Ausgang aus, indem dem Ausgang zwei verborgene Knoten vorgeschaltet werden, wenn der Ausgang Fehler macht: je ein Knoten zur Korrektur von *fälschlich 1* und von *fälschlich 0* Fehlern. Weitere Korrekturknoten können nach demselben Prinzip rekursiv zur Korrektur der Korrekturknoten vorgeschaltet werden. Jeder Knoten erhält die normalen Eingaben und die Ausgaben seiner bis zu zwei Korrekturknoten als Eingabe; es entsteht ein unbalancierter Binärbaum. Beispielprobleme: 10-bit Parity und zufällige boolesche Funktionen. Vergleich: Erzeugt kleinere Netze mit besserer Generalisierung als Tiling.

Nochmals besser ist der *BINCOR-Algorithmus* von Simon [328], von dem es zwei Varianten gibt. Er verwendet Korrekturknoten wie Upstart, vermeidet aber das exponentielle Breitenwachstum der Upstart-Bäume, indem er Paare von Korrekturknoten entweder (Variante 1) einfach nebeneinander plaziert und nur mit Eingaben und dem Ausgangsknoten verbindet oder (Variante 2) jedes neue Paar von Korrekturknoten zusätzlich mit den Ausgängen des letzten vorhergehenden Paars von Korrekturknoten verbindet. Variante 1 führt zu einer einzigen breiten verborgenen Schicht, Variante 2 zu vielen verborgenen Schichten jeweils der Breite 2. BINCOR verwendet (ebenso wie Upstart) außerdem eine gegenüber dem Pocket-Algorithmus nochmals verbesserte Lernregel. Beispielprobleme: Zufällige boolesche Funktionen, MONKS-Probleme [350]. Vergleich: BINCOR ist besser als Tiling und Upstart, aber schlechter als Backpropagation.

Wie die BINCOR-Variante 1 arbeitet auch das Verfahren von Zollner et al. [418] mit nur einer verborgenen Schicht von Schwellwertknoten. Alle Eingabevektoren werden normalisiert, so daß sie alle auf einer Hyperkugel liegen. Die Trennhyperfläche eines Schwellwertknotens kann davon eine Hyperkugelkappe abschneiden, insbesondere für jedes Beispiel eine solche, die nur dieses eine Beispiel enthält. Das Verfahren trainiert nun mehrere Schwellwertknoten darauf, jeweils eine Kappe abzuschneiden, die ausschließlich positive Beispiele enthält, von diesen aber möglichst viele. Dazu wird die Trainingsmenge jedes Knotens durch schrittweises Herausnehmen positiver Beispiele solange manipuliert, bis der Knoten seine Aufgabe löst. Es werden soviele Knoten erzeugt, bis jedes positive Beispiel von mindestens einem Knoten als positiv klassifiziert wird. Die so entstandenen Schwellwertknoten werden durch einen weiteren Schwellwertknoten ODER-verknüpft, um die Gesamtausgabe zu berechnen. Beispielprobleme: Korrelierte Zufallsmuster. Vergleich: Erzeugt kleinere Netze als Tiling-Algorithmus.

Zum konstruktiven Lernen in MLPs mit Sigmoidknoten liegt das von Ash in [19] als *dynamische Knotenerzeugung* (*dynamic node creation*, *DNC*) vorgeschlagene und oft zitierte Verfahren nahe. Hier wird ein Netz mit genau einer verborgenen Schicht verwendet, die anfangs nur wenige oder gar keine Knoten enthält. Wenn die Kurve des Fehlers auf der Trainingsmenge über die Zeit zu sehr abflacht, wird ein weiterer Knoten in die verborgene Schicht eingefügt und das ganze Netz normal weitertrainiert. Dieses Vorgehen hilft angeblich insbesondere, lokale Minima zu vermeiden bzw. wieder zu verlassen. Beispielprobleme: Diverse Encoder, Symmetry und Parity Probleme und binäre Addition. Keine Generalisierungstests. Vergleich: Backpropagation lernt langsamer und konvergiert häufiger nicht. Keine quantitativen Kriterien für Algorithmus angegeben. Die gleiche Methode schlagen Bellido und Fernández vor [40], jedoch ohne jegliche Auswertung.

Das Verfahren von Wang, di Massimo, Tham und Morris [388] baut ein Netz mit zwei verborgenen Schichten auf. Zunächst wird ein Netz mit wenigen verborgenen Knoten in jeder der beiden Schichten weitmöglichst trainiert. Dann wird in jeder der beiden Schichten genau dann ein Knoten zugefügt und anschließend weitertrainiert, wenn die Knoten der Schicht einem Kriterium genügen, das die Autoren *δ-lineare-Unabhängigkeit* nennen. Dies ist ein Maß dafür, daß die Knotenanzahl bislang kleiner ist als die Dimension des von der Schicht zu modellierenden Raums. Zu seiner Berechnung ist die Determinante einer Korrelationsmatrix zu bestimmten, was strenggenommen den Rahmen eines neuronalen Verfahrens sprengt. Beispielprobleme: Diverse stetige Funktionen und Biomassen-Schätzung eines Penicillin-Fermentationsprozeßes. Vergleich: Keiner.

Waibel schlägt in [385] vor, mehrere kleine Netze mit je einer verborgenen Schicht auf Teilaufgaben einer größeren Aufgabe zu trainieren und die entstehenden Netze zusammenzufügen. Beispielsweise lernen die kleinen Netze je 3 Klassen einer Klassifikationsaufgabe mit 6 Klassen. Das Zuammenfügen erfolgt dadurch, daß ein neues Netz gebildet wird, das von den kleineren Netzen nur die Gewichte zu deren verborgener Schicht übernimmt und daraus eine größere verborgene Schicht zusammenbaut, die nicht mehr trainiert wird. Eine Ausgabeschicht und eventuell weitere verborgene Schichten oder zusätzliche „freie“ Knoten in der ersten verborgenen Schicht werden daran angeschlossen und auf die Gesamtaufgabe trainiert. Beispielprobleme: Klassifikation der Phoneme b,d,g und p,t,k. Vergleich: Geringfügig bessere Ergebnisse als beim Start mit einem Gesamtnetz für beide Teile bei zugleich stark verkleinertem Trainingsaufwand. Es ist kein exakter Algorithmus angegeben.

Hanson gibt in [144] eine Methode zum gezielten lokalen Verfeinern eines MLP an, die er *Meiosis* nennt. Er verwendet stochastische Verbindungen, die anstatt eines deterministischen Gewichtswertes durch eine Normalverteilung charakterisiert werden, deren Mittelwert dem herkömmlichen Gewicht entspricht und mit Gradientenabstieg trainiert wird. Die Varianz wird proportional zu den durch die Verbindung hindurchpropagierten Fehlern erhöht und fällt exponentiell über die Zeit wieder ab. Wird die summierte Varianz der Verbindungen an einem Knoten zu groß, so wird dieser Knoten in zwei Knoten aufgeteilt, dessen Verbindungen Verteilungen haben, die je eine Standardabweichung über bzw. unter den ursprünglichen liegen und die halbe Varianz haben. Beispielprobleme: Parity und Blutkrankheitsdiagnose. Vergleich: Generalisiert besser als Backpropagation-Lösung. Keine quantitativen Kriterien angegeben.

Das *Knotenteilen* (*node splitting*) von Wynne-Jones [409] ist eine Verbesserung von Meiosis. Da bei Meiosis die Varianzen nur addiert werden, kann die Art der Korrektur der Mittelwerte beim Teilen der Knoten unsinnig sein, nämlich dann, wenn es wenig Korrelation zwischen den Vorzeichen der einzelnen Gewichtsänderungen der verschiedenen Verbindungen gibt. Knotenteilen umgeht dieses Problem, indem es als Kriterium für das Teilen von Knoten und als Richtung der Gewichtskorrektur die größte Hauptkomponente des Gewichtsänderungs*vektors* für jeden Knoten benutzt. Diese wird mit einer Hauptkomponentenanalyse (*principal components analysis*) berechnet. Anders als bei Meiosis werden gewöhnliche Gewichte verwendet. Im Gegensatz zu Hanson berichtet Wynne-Jones, daß das Knotenteilungsverfahren für MLPs kaum Nutzen hat, weil meist durch die Teilung neue Fehler induziert werden, so daß die geteilten Knoten alsbald wieder die gleichen Gewichte entwickeln; für Netze mit lokalen Knotenwirkungen, z.B. mit radialen Basisfunktionen, sei es hingegen nützlich. Beispielprobleme: Unterscheidung eines Rings vom Rest der zweidimensionalen Ebene. Vergleich: Keiner. Keine quantitativen Kriterien angegeben.

Das bei weitem erfolgreichste additive Verfahren ist *Cascade Correlation* (*CasCor*) von Fahlman und Lebiere [102, 103]. Dies scheint das einzige Verfahren zu sein, das in einer größeren Zahl von Anwendungen tatsächlich eingesetzt wird (z.B. [220]); außerdem wird von den Autoren auch eine Softwarefassung öffentlich bereitgestellt[6]. CasCor beginnt mit einem Netz ohne verborgene Schicht und trainiert dieses, bis Stagnation eintritt. Dann wird wiederholt ein neuer verborgener Knoten A in folgender Weise ins Netz eingefügt: In der Phase „input training“ werden die zu A führenden Gewichte so trainiert, daß die Korrelation zwischen der Aktivierung von A und dem Restfehler des Netzes betragsmäßig möglichst groß wird. A wird also quasi dazu trainiert, ein starkes fehleranzeigendes Signal zu produzieren. Vom verwendeten Korrelationsmaß (eigentlich: Kovarianz) stammt der „Correlation“-Teil des Namens. Stagniert dieses Korrelationstraining, so werden in der Phase „output training“ die Eingangsgewichte von A und allen anderen verborgenen Knoten festgehalten und die mit den Netzausgängen verbundenen Gewichte zur Minimierung des Fehlers trainiert. Für die Gewichtsänderungen wird das Quickprop-Verfahren eingesetzt. Jeder verborgene Knoten realisiert somit eine Art Merkmalsdetektor der nach seiner Einführung nicht mehr geändert, sondern nur in variabler Weise benutzt wird (feste Eingangsgewichte, aber variable Ausgangsgewichte). Der „Cascade“-Teil des Namens rührt daher, daß jeder verborgene Knoten die Ausgaben aller zuvor installierten verborgenen Knoten zusätzlich zu den normalen Eingabekoeffizienten als Eingabe erhält. Dies entspricht der invertierten Pyramide von Gallant und wird von Fahlman/Lebiere eine Kaskade genannt. Jeder Merkmalsdetektor hat gewissermaßen eine höhere Ordnung als alle früheren, jeder verborgene Knoten bildet eine eigene verborgene Schicht. Damit kann das entstehende Netz sehr starke Nichtlinearitäten darstellen. Das Korrelationstraining der neuen verborgenen Knoten ermöglicht, mehrere *Kandidaten* gleichzeitig zu trainieren und am Ende den besten auszuwählen. Die Kandidaten unterscheiden sich in der Zufallsinitialisierung ihrer Eingangsgewichte. Das Verfahren hat Ähnlichkeit mit einer in der Statistik als *backfitting* bekannten Regressionsprozedur, bei der wiederholt eine einzelne Ressource (hier: verborgener Knoten) erst einzeln optimiert und dann das Ganze einschließlich der neuen Ressource feinabgestimmt wird. Die Verwendung einer Menge von Kandidaten für die neue Ressource ist dabei eine neue Idee. Die Korrelationsregel ist im Prinzip falsch, denn sie belohnt eine möglichst große Aktivierung von A auch für solche Beispiele, bei denen der zu korrigierende Netzfehler nur klein ist, so daß eine Tendenz besteht, geringe Fehler überzukompensieren. Beispielprobleme: Parity, Two Spirals. Vergleich: Lernt schneller als Backpropagation. In der Literatur sind eine Reihe von erfolgreichen Anwendungen von CasCor berichtet worden, z.B. [220]. Diese Anwendungen lassen darauf schließen, daß CasCor besser als die meisten anderen additiven Verfahren ist. Aus diesem Grund untersuche ich in Kapitel 5 eine Reihe von Varianten des CasCor-Verfahrens.

Sjøgaard argumentiert in [332, 333], daß die extrem große potentielle Nichtlinearität der bei CasCor entstehenden kaskadierten Netze sich negativ auf das Generalisierungsverhalten auswirkt. Er schlägt dementsprechend vor, alle verborgenen Knoten *nebeneinander* in einer verborgenen Schicht zu plazieren und weist am Beispiel eines einzigen, gutmütigen, zweidimensional geometrischen Klassifikationsbeispiels mit 8 binär kodierten Klassen (*Three Disks Problem*) nach, daß das so entstehende *2CCA-Verfahren* tatsächlich zu besserer Generalisierung führt. Leider bleibt in seiner Arbeit völlig

[6] ftp://ftp.cs.cmu.edu/afs/cs/project/connect/code/cascor-v1.0.4.shar. CasCor ist inzwischen auch in anderen Simulatoren implementiert.

unklar, ob sich dieses Ergebnis auf andere Aufgaben überträgt. Insbesondere trifft Sjøgaard keine Vorkehrungen um Überanpassung zu verhindern. Die gleiche Idee einer CasCor-Variante mit nur einer verborgenen Schicht hatte unabhängig davon Yeung [413]. Er stellt für mehrere Lernprobleme mit realen Datensätzen (thyroid, sonar, waveform) etwa gleiche Generalisierungsleistung wie bei CasCor fest, veröffentlicht aber leider die genauen Parameter seiner Versuche nicht.

Klagges und Soegtrop schlagen mit ihrem *limited fan-in random-wired CasCor* [195] eine Variante vor, bei der jeder Knoten nur mit den Ausgaben einer kleinen, festen Zahl anderer Knoten (z.B. 2) verbunden wird und zwar in zufälliger Weise. Auf Problemen mit wenigen Eingabekoeffizienten funktioniert das Verfahren gut, es skaliert aber in dieser Hinsicht sehr schlecht, da zur Erhaltung guter Leistung die Kandidatenanzahl exponentiell in der Zahl der Eingaben ansteigen müßte. Beispielprobleme: Two Spirals, Three Discs. Vergleich: CasCor. Keine quantitativen Kriterien angegeben.

Simon et al. [328, 329] schlagen vor, bei Problemen mit k Ausgangsknoten eine eigene Kandidatenmenge für jeden Ausgangsknoten vorzusehen, in denen die Kandidaten jeweils nur auf Korrelation mit dem Fehler des zugeordneten Ausgangsknotens (anstatt aller Ausgangsknoten zugleich) trainiert werden. Es werden also pro Schritt k verborgene Knoten in die Kaskade eingefügt, die jeweils nur mit der Ausgabe verbunden sind, für die sie trainiert wurden (Variante 1), oder die mit allen Ausgaben verbunden sind (Variante 2). Beispielprobleme: Inverse kinematische Transformation, Erzeugung eines Hamming-Codes, 3×3 bit rotationsinvariante Mustererkennung. Vergleich: Bessere Generalisierung als CasCor bei Kinematikproblem, sonst gleich. Keine quantitativen Kriterien angegeben. Sehr kleine Trainingsmengen verwendet, dabei keine Maßnahmen gegen Überanpassung getroffen.

Littmann und Ritter schlagen mit *CasEr* und *CasQEF* weitere Varianten von CasCor vor [223]. CasEr verwendet die Minimierung des quadratischen Fehlers anstatt der Korrelationsmaximierung als Lernregel für die Kandidatenknoten, CasQEF fügt außerdem höhere Potenzen (meist nur das Quadrat) der Ausgaben verborgener Knoten als zusätzliche Eingaben späterer verborgener Knoten hinzu. Beispielprobleme: XOR, zwei überlappende Gaußverteilungen, drei überlappende Dreiecksverteilungen, Mackey-Glass Zeitreihe. Vergleich: CasCor funktioniert besser als CasEr aber schlechter als CasQEF. Es ist unklar, warum die Autoren für die Gewichtsänderungen feste Schrittweiten verwenden anstatt Quickprop. Die genauen Kriterien für den Wechsel zwischen *input training* und *output training* sind nicht angegeben. Es werden keine Vorkehrungen gegen Überanpassung getroffen.

2.8.2 Subtraktive Verfahren

Subtraktive Verfahren (*Beschneidungsverfahren*) beginnen mit einem Netz, daß „übergroß" gewählt ist. Übergroß bedeutet, daß man erwarten kann, ein *Teilnetz* des Netzes sei gut geeignet, das gegebene Problem zu lösen. Diese Entscheidung erfordert eine Schätzung der benötigten Netzgröße, jedoch kann diese Schätzung ruhig grob ausfallen. Ziel der Verfahren ist, die Darstellungsmächtigkeit eines vorgegebenen Netzes in gezielter Weise soweit abzusenken, daß es für die gegebene Aufgabe optimal angepaßt ist und Überanpassung vermieden wird. In dieser Hinsicht ähneln Beschneidungsverfahren den Regularisierungsverfahren [293]. Anders als diese reduzieren sie jedoch nicht die *effektive* Zahl von Parametern, sondern die *tatsächliche* Zahl von Parametern. Dies hat den potentiellen Vorteil, daß den verbleibenden Parametern keine Beschränkungen auferlegt werden. Dieser Vorteil kommt vor allem dann zum Tragen, wenn aufgrund starker Nichtlinearitäten in der zu realisierenden Funktion große Gewichte benötigt werden, die die meisten Regularisierungsverfahren ja gerade verhindern [117]. Im Gegensatz zu zahlreichen additiven Verfahren gehen fast alle Beschneidungsverfahren von Netzen mit Sigmoidknoten aus und sind gleichermaßen für Approximations- wie für Klassifikationsaufgaben geeignet.

Sietsma und Dow [325, 326] verwenden zwei Arten der Beschneidung: verborgene Knoten werden erstens entfernt, wenn sie für alle Beispiele fast gleiche Ausgaben liefern, und von den verbleibenden werden dann zweitens jene entfernt, die zur Herstellung einer die Ausgabe eindeutig beschreibenden internen Repräsentation nicht nötig sind. Letzteres wird mit einem Suchprogramm geprüft, das zugleich alle Aktivierungen der verborgenen Knoten betrachtet, so daß das Verfahren eigentlich kein neuronales Verfahren ist. Die Autoren kommen in ihren Experimenten zu dem Ergebnis, daß das Beschneiden zwar die Generalisierung verbessert, jedoch Trainieren mit verrauschten Beispielen noch bessere Resultate liefert. Das ist nicht überraschend, da sie Netze mit 64 Eingabekoeffizienten mit nur 43 Beispielen zu trainieren versuchen und dabei zunächst keinerlei Vorkehrungen gegen Überanpassung treffen. Beispielprobleme: Klassifikation der Frequenz von Sinuswellen. Vergleich: Normale Backpropagation.

Es gibt mehrere Vorschläge, wie überflüssige verborgene Knoten automatisch und mit lokalen Verfahren identifiziert und eliminiert werden können. Diese besprechen wir in den folgenden Absätzen.

Mozer und Smolensky schlagen *skeletonization* vor [258]. Hierbei werden die verborgenen Knoten mit der niedrigsten *Relevanz* entfernt. Die Relevanz wird geschätzt, indem die Ableitung der Fehlerfunktion relativ zu einem gedachten *Durchlaßkoeffizienten* an jedem verborgenen Knoten berechnet wird. Dieser Koeffizient skaliert die Ausgabe des Knotens linear; er ist immer 1, das Entfernen des Knotens macht ihn zu 0. Knoten mit kleinster Ableitung des Fehlers gegen den Durchlaßkoeffizienten haben die geringste Relevanz und werden entfernt. Beispielprobleme: Fünf verschiedene Spielzeugprobleme. Vergleich: Lernt schneller als Backpropagation. Keine Generalisierungstests.

Karayiannis [186] verfolgt bei seinem *ALADIN* die gleiche Idee, verwendet jedoch ein anderes Relevanzmaß (bei ihm Sensitivität genannt), das ein Produkt aus zwei Faktoren ist: Erstens die Ableitung des Fehlers gegen die Aktivität des Knotens und zweitens der Quotient aus Aktivität und Fehler. Dieses Kriterium scheint mir nicht sinnvoll zu sein; eine Bestätigung oder Widerlegung dieses Eindrucks ist jedoch aus dem Artikel nicht zu entnehmen. Beispielprobleme: Zufällige Abbildungen. Vergleich: Keiner.

Ji, Snapp und Psaltis betrachten die Approximation einer reellwertigen Funktion einer Variablen [181]. Sie verwenden zunächst zwei Regularisierungsterme: einen, der die „Benutzung“ eines verborgenen Knotens im Sinne von gleichzeitig großem Eingangs- und Ausgangsgewicht bestraft und einen weiteren, der allgemein große Gewichte bestraft. Gewichte, die im Verlauf des Trainings sehr klein werden, werden dann entfernt. Beispielprobleme: Lernen der Funktion $\exp(-(x-1)^2) + \exp(-(x+1)^2)$ aus nur 9 oder 17 Beispielen. Vergleich: Normale Backpropagation ergibt katastrophale Überanpassung. Für eine solche eindimensionale Aufgabe sollte man kein neuronales Netz einsetzen, weil z.B. Spline-Funktionen eine bessere Anpassung ergeben.

Sperduti und Starita [339] erweitern das normale Backpropagation-Verfahren durch eine Optimierung des Steilheitsparameters a der Sigmoidfunktion $1/(1+e^{-ax})$. Die a der verborgenen Knoten werden mittels Gradientenabstieg optimiert und zugleich einem exponentiellen Abfall über die Zeit unterworfen. Alle Knoten, deren a dadurch fast zu Null abfällt, werden entfernt, da sie nur noch fast konstante Ausgaben liefern. Beispielprobleme: 4-bit Parity, 8×8 bit Ziffernlernen (mit nur 10 Beispielen!), 8-16(sic!)-8 encoder. Vergleich: Keiner. Keine Generalisierungstests.

Ein feinkörnigerer Ansatz für Beschneidungsverfahren besteht darin, nicht ganze Knoten zu entfernen, sondern einzelne Verbindungen. Mit dieser Möglichkeit haben Beschneidungsverfahren offensichtlich Vorteile gegenüber additiven Verfahren im Hinblick auf die Genauigkeit der Anpassung des Netzes an das Problem. Der einfachste Ansatz besteht darin, Gewichte mit sehr kleinem Betrag während des Trainings oder nach Abschluß des Trainings mit anschließendem Nachtrainieren zu entfernen. Im Vergleich zu den nachfolgend beschriebenen Verfahren ist diese Methode jedoch weniger gut [117], weil auch kleine Gewichte oft empfindliche Auswirkungen auf den Fehler haben können.

Eine stochastische Methode verwenden Omidvar und Wilson in [272]. Hier wird während des Trainings zufällig jedes Gewicht auf Null gesetzt mit einer Wahrscheinlichkeit, die exponentiell fällt mit steigendem Betrag des Gewichts und sinkender „Temperatur“. Die Temperatur ist ein Parameter des Lernverfahrens, der während des Lernens sukzessive abgesenkt wird. Da die zu Null gesetzten Gewichte anschließend normal weitertrainiert werden (also wieder wachsen können), wird das Verfahren erst dann ein echtes Beschneidungsverfahren, wenn die Gewichte bei einer geringen Temperatur stattdessen irgendwann endgültig entfernt werden. Beispielprobleme: Ein nicht näher beschriebenes Zeichenerkennungsproblem. Vergleich: Durch das Verfahren wird nur noch ein Zehntel der Gewichte benötigt; die Generalisierungsleistung ist gleich.

Karnin schlägt in [187] vor, die Sensitivität des Netzes gegenüber der Entfernung einer Verbindung dadurch abzuschätzen, daß während des Trainings die Abhängigkeit des Fehlers von den normalen Änderungen des Gewichts beobachtet und kumuliert wird. Die Gewichte mit geringstem Einfluß auf den Fehler werden dann entfernt. Beispielprobleme: Ein zweidimensionales Klassifikationsproblem mit unbeschränkter Beispielmenge für 2-2-1 Netz und ein boolesches „Regel plus Ausnahme"–Problem. Vergleich: Keiner.

Thodberg beschreibt *Ace of Bayes*, eine Trainings- und Beschneidungsmethode, die auf Bayes'schen a-priori-Annahmen aufbaut und mathematisch hergeleitet wird [349]. Das Trainingsverfahren verwendet Regularisierungsterme, die wechselseitig aus den Beispielen geschätzt und zur Modellierung der Beispielmenge verwendet werden. Im Anschluß an das Training wird eine *brute force*–Beschneidungsmethode angewendet: Irgendein Gewicht wird entfernt und das Netz für einige Epochen weitertrainiert. Hat sich das Qualitätsmaß für das Netz verschlechtert, wird der ursprüngliche Netzzustand wiederhergestellt, andernfalls wird das beschnittene Netz beibehalten. Diese Prozedur wird für jede Verbindung wiederholt, und die gesamte Beschneidung (für alle Verbindungen) wird ebenfalls mehrmals wiederholt. Der Autor behauptet, dieses sehr aufwendige Verfahren sei für Netze bis 1000 Verbindungen praktikabel, benutzt aber selber nur Netze mit maximal 90 Verbindungen für maximal 200 Beispiele. Beispielprobleme: Netz-Wiederentdeckung, Bestimmung des Fettgehalts von Fleisch aus infrarot-spektroskopischen Daten. Vergleich: Lineare und quadratische Regression, sowie Backpropagation mit frühem Stoppen (na endlich einmal!) werden alle in der Generalisierung übertroffen.

Das am häufigsten zitierte und dennoch fast nie eingesetzte Beschneidungsverfahren ist *optimal brain damage* (*OBD*) von Le Cun, Denker und Solla [85]. Es verwendet drei vereinfachende Annahmen, um den Einfluß der Entfernung einer Verbindung auf den Fehler abzuschätzen: Erstens, daß das Netz bis zu einem Fehlerminimum trainiert wurde, zweitens, daß die Fehleroberfläche dort annähernd eine polynomielle Fläche zweiten Grades ist und drittens, daß sich die Entfernung verschiedener Verbindungen nicht gegenseitig beeinflußt, also die Hessematrix eine Diagonalmatrix ist (Diagonalannahme). Unter diesen Annahmen ist die Relevanz oder *Auffälligkeit* (*saliency*) eines Gewichts proportional zur zweiten Ableitung des Fehlers gegen das Gewicht. Es müssen also die Diagonalelemente der Hessematrix der Fehlerfunktion berechnet werden, was mit dem Backpropagation-Verfahren in einem zusätzlichen Rückverfolgungsschritt erledigt werden kann. Die Verbindungen mit kleinster Auffälligkeit werden dann entfernt und das Netz wird nachtrainiert. Das Verfahren wird mehrfach wiederholt. Beispielprobleme: Weitere Beschneidung eines bereits von Hand optimierten Netzes für die Erkennung handgeschriebener Ziffern. Das Netz kann von anfänglich 2578 Parametern bis auf etwa 1000 Parameter verkleinert werden, ohne daß die Generalisierung schlechter wird. Sie wird allerdings auch nicht besser, da das Netz ja schon gut optimiert war. Vergleich: Keiner. Es wird kein genauer Algorithmus angegeben. Die Autoren benutzen das Verfahren interaktiv.

Verschiedene Verbesserungen von OBD sind vorgeschlagen worden. So berechnen Hassibi und Stork in ihrem *optimal brain surgeon* (*OBS*) [148] die vollständige (inverse) Hessematrix mit nur noch schwachen vereinfachenden Annahmen. Sie argumentieren, daß die bei OBD benutzte Diagonalapproximation für reale Fälle in hohem Maß unzutreffend sei. Mit Hilfe der Hessematrix kann die unwichtigste Verbindung gefunden und entfernt werden. Dabei wird eine Korrektur für alle verbleibenden Gewichte berechnet, die das Anwachsen des Fehlers minimiert. Das Verfahren ist sehr aufwendig, weil die Hessematrix in einem Netz mit n Gewichten n^2 Elemente enthält und die Berechnung für jedes zu entfernende Gewicht wiederholt werden muß, wenn die Korrektur der anderen Gewichte genau sein soll. Außerdem ist die Berechnung der Hessematrix keine lokale Operation im Netz. Beispielprobleme: Für die MONKS-Probleme konnten zwischen 62% (MONK2) und 90% (MONK3) der Gewichte entfernt werden, bei gleichbleibender Generalisierungsleistung. Für NETtalk [321] wurde ein Netz von

5546 auf 2438 Verbindungen verkleinert, bei gleichzeitiger Verbesserung der Generalisierung um zehn Prozent. Vergleich: Backpropagation mit Gewichtsabfall, OBD.

Levin, Leen und Moody [216] verwenden eine blockdiagonale Approximation an die Hessematrix als Kompromiß zwischen OBS und OBD. Die Größe der Blöcke entspricht der Zahl von Knoten jeder Schicht, und die Blöcke werden mit Hilfe einer Hauptkomponentenanalyse berechnet. Alle Hauptkomponenten, deren Entfernung die Generalisierung nicht verschlechtert (was mit Hilfe einer Validationsmenge geprüft wird), werden entfernt. Das Entfernen geschieht durch Projektion des Gewichtsvektors jedes Knotens auf den Unterraum, der durch Wegnahme der entfernten Komponente entsteht. Diese Projektion modifiziert zwar Gewichte, entfernt sie aber nicht; insofern ist dieses Verfahren eigentlich ein Regularisierungsverfahren — es wird nur die *effektive* Parameterzahl verkleinert. Beispielprobleme: Eindimensionale Funktionsapproximation, Zeitreihenvorhersage des Index der industriellen Produktion. Vergleich: Erheblich bessere Generalisierung als Backpropagation.

Eine ähnliche Methode der Projektion auf kleinere Räume verwendet Yang [411]. Er führt auf die Ausgaben der verborgenen Schichten und auf die Eingänge eine diskrete orthogonale Transformation aus und unterdrückt die kleineren Komponenten des Resultats. Dies ist jedoch eigentlich keine neuronale Methode. Beispielprobleme: XOR, Unterscheidung von zweimal drei Sektoren in einem Kreis. Vergleich: Backpropagation. Keine Generalisierungstests.

Finnoff, Hergert und Zimmermann [116, 117] verwenden eine Teststatistik für die Hypothese, daß ein Gewicht beim Training den Wert Null erreicht, um zu entscheiden, welche Gewichte entfernt werden sollten. Diese Statistik ist wie die *saliency* von OBD ein Spezialfall einer Wald-Teststatistik, hat OBD gegenüber aber den Vorteil, daß sie auch schon vor Erreichen eines Fehlerminimums ausgewertet werden kann. In ihrer umfangreichen Studie [117] vergleichen die Autoren mit sechs verschiedenen künstlichen, aber unter realistischen Annahmen erzeugten Problemen insgesamt zwölf verschiedene Trainingsverfahren. Eine Variante ihres Beschneidungsverfahrens, genannt *autoprune*, schneidet dabei insgesamt unter allen Verfahren am besten ab, zusammen mit Gewichtsabfall mit von Hand optimal eingestelltem Parameter. OBD ist bei den meisten Problemen deutlich schlechter. Damit scheint autoprune unter den Beschneidungsverfahren mit mäßigem Berechnungsaufwand das leistungsfähigste zu sein. Im Kapitel 6 werde ich deshalb autoprune und eine von mir als Verbesserung vorgeschlagene Variante untersuchen.

2.8.3 Additiv-subtraktive Verfahren

Offensichtlich lassen sich die Grundideen des additiven und subtraktiven konstruktiven Lernens miteinander verbinden. Es gibt in der Literatur auch einige Vorschläge dazu, von denen jedoch bisher keiner so recht überzeugen kann. Jede Kombination von additiven und subtraktiven Elementen in einem Lernverfahren wirft gegenüber seinen Einzelteilen das zusätzliche Problem der Balance zwischen diesen Teilen auf. Da wir noch nicht einmal rein additive und rein subtraktive Verfahren richtig beherrschen, halte ich deshalb den Versuch von additiv-subtraktiven Verfahren für zu sehr mit Schwierigkeiten befrachtet, um derzeit sinnvoll zu sein. Aus diesem Grund untersuche ich in meiner Arbeit keine solchen Verfahren. Ich stelle im folgenden dennoch die bisher publizierten Ideen vor, um zu belegen, daß es bisher keine überzeugenden Ideen gibt. Es ist kennzeichnend für die Schwierigkeiten bei der Kombination der Verfahren, daß die meisten additiv-subtraktiven Verfahren auch bei der Beschneidung nur auf Ebene der Knoten arbeiten, anstatt die leistungsfähigere feinkörnige Beschneidung einzelner Verbindungen zu verwenden.

Hirose, Yamashita und Hijiya [158] trainieren ein Netz mit einer verborgenen Schicht solange mit anfangs nur einem Knoten, bis die Verkleinerung des Fehlers stagniert, fügen dann einen verborgenen Knoten hinzu, trainieren weiter bis zur nächsten Stagnation und so fort bis zu einem nicht spezifizierten Abbruchkriterium. Dann entfernen sie den jüngsten Knoten wieder und trainieren weiter bis das Netz wieder ein zufriedenstellendes Fehlerniveau erreicht hat. Weitere Knoten werden in der gleichen Weise entfernt, bis das gewünschte Niveau nicht mehr erreicht wird. Das vorletzte der bei dieser Prozedur erreichten Netze ist das Resultat. Beispielprobleme: XOR, Erlernen eines 8×8 bit alphanumerischen Fonts (mit nur 36 Beispielen). Vergleich: Konvergiert oftmals, wo normale Backpropagation dies nicht tut. Keine Generalisierungstests. Nicht alle quantitativen Kriterien angegeben.

Noch wilder geht es bei Hagiwara zu [140]. Er definiert als *badness factor* eines Knotens die Summe der quadrierten Fehlersignale, die durch den Knoten propagiert werden. Man beachte, daß dieses Maß hoch ist, wenn ein Knoten gerade besonders stark lernt (im Gegensatz zu anderen Knoten, deren Gewichte gewissermaßen nur faul in der Gegend herumliegen — ob sie nun nützlich sind oder nicht). Wenn der Rückgang des globalen Fehlers stagniert, setzt das Verfahren die Gewichte des nach obigem Maß schlechtesten Knotens auf neue Zufallswerte. Führen mehrere solche Rücksetzversuche nicht zum weiteren Rückgang des Fehlers, so wird ein neuer Knoten eingefügt. Nach Konvergenz des obigen Verfahrens zu einem ausreichend kleinen Fehler werden nach demselben Maß die schlechtesten Knoten einer nach dem anderen aus dem Netz entfernt und das Netz dazwischen jeweils nachtrainiert. Beispielprobleme: XOR und Lernen eines 3×5 bit Fonts mit 20 Beispielen[7]. Vergleich: Konvergiert schneller als normale Backpropagation. Keine Generalisierungstests.

Das *dynamic node architecture learning* (*DNAL*) von Bartlett [30] verwendet grundsätzlich das gleiche Vorgehen des Knotenzufügens bei Stagnation. Zum Entfernen von Knoten wird jedoch ein informationstheoretisch begründetes und mathematisch hergeleitetes Maß herangezogen, das mit Hilfe von stochastischen Gewichtsänderungen bestimmt wird. Es sind keine exakten Kriterien dafür angegeben, wann ein Zufügen oder Löschen eines Knotens erfolgen sollte. Beispielprobleme: XOR, 3-bit *De*coder, Zeitreihenvorhersage der logistischen Verhulst-Funktion. Vergleich: Keiner.

Moody [253] empfiehlt eine Kombination von *sequential network construction* (*SNC*), mit *sensitivity based pruning* (*SBP*) und optimal brain damage (OBD). Bei SNC wird wie bei obigen Verfahren ein zusätzlicher Knoten (oder auch mehrere) in die verborgene Schicht eingefügt, wenn das Training stagniert. Allerdings wird hier nun zunächst nur der neue Knoten trainiert und die Gewichte aller anderen werden festgehalten. Erst wenn dieses Training stagniert, werden wieder alle Gewichte trainiert, und der nächste Zyklus beginnt. SBP ermittelt den Einfluß des Entfernens von Eingangsknoten auf den Gesamtfehler, z.B. durch direktes Ausprobieren, und entfernt die unwichtigsten Eingaben. Entfernen bedeutet hierbei das Ersetzen durch den Mittelwert über die Beispiele. Wechselwirkungen zwischen den Eingängen werden dabei ignoriert. Optimal brain damage ist schließlich das von oben bekannte Verfahren zur Elimination einzelner Verbindungen und wird als dritter und letzter Schritt ausgeführt. Exakte Kriterien für den Wechsel zwischen den Phasen und für den Zeitpunkt und die Stärke der einzelnen Veränderungen sind nicht angegeben. Beispielprobleme: Keine. Vergleich: Keiner.

Fahner und Eckmiller [104] beschreiben zwei Verfahren zur Erzeugung von „frugalen Netzen höherer Ordnung“ (*parsimonious higher order networks*, *parsiHONs*). Betrachtet wird nur die binäre Klassifikation von Bitvektoren. Das sogenannte *ultimative HON* ist ein Schwellwertneuron, dem als Eingaben alle 2^n Produktterme der n Eingabebits zur Verfügung stehen und das folglich 2^n Gewichte hat, wohingegen ein parsiHON nur sehr wenige dieser Terme benutzt. Das erste Verfahren beschneidet ein ultimatives HON zu einem parsiHON, indem während des Trainings alle Gewichte entfernt werden, deren Betrag unter einem bestimmten (nicht genannten) Wert liegt. Das zweite Verfahren, genannt *STRADA*, startet gleich mit einem parsiHON der gewünschten Gewichtsanzahl und entfernt während des Trainings wiederholt Terme, die durch andere ersetzt werden. Das Entfernen erfolgt stochastisch, wobei Terme mit kleinem Gewicht bevorzugt werden. Das Einfügen wählt aus einer großen Menge von zufälligen Kandidatentermen diejenigen aus, die die höchste Korrelation zur gewünschten Ausgabe der bislang noch falsch klassifizierten Beispiele aufweisen. Beispielprobleme: Two Spirals (mit 7-bit-Kodierung jeder der beiden Koordinaten), Unterscheidung linksverschobener von rechtsverschobenen Bitstrings. Vergleich: Keiner. Nicht alle quantitativen Kriterien angegeben.

Das *Teilen-und-Herrschen-Verfahren* (*divide and conquer*), von Romaniuk und Hall [301], das eigentlich Herrschen-und-Teilen heißen müßte, erzeugt Netze mit mehreren verborgenen Schichten für

[7] genannt „relatively difficult problem“!

Klassifikationsprobleme. Im „Herrschen"-Schritt, mit dem das Verfahren beginnt, wird ein neuer Ausgangsknoten erzeugt, der als Eingaben die Ausgaben aller bisher vorhandenen Knoten erhält. Nur die Gewichte dieses Knotens werden trainiert. Der Knoten soll die Klasse i von allen anderen Klassen unterscheiden; jede Klasse wird einzeln trainiert. Kann der Knoten das Problem nicht lösen, so wird der „Teilen"-Schritt ausgeführt. Dabei wird wiederholt ein weiterer Knoten in die mit dem „Herrschen"-Knoten begonnene Schicht gesetzt, der nur mit den Beispielen trainiert wird, die bisher noch nicht korrekt gelernt wurden. Um eine Überanpassung zu vermeiden, wird zu jedem solchen Beispiel allerdings noch sein nächster Nachbar aus der Trainingsmenge hinzugefügt. Wenn der Gesamtfehler so nicht weiter reduziert werden kann, findet der nächste „Herrschen"-Schritt statt, der nun alle neuen „Teilen"-Knoten mit ihren speziellen Fähigkeiten zusätzlich zur Verfügung hat. Nach Ende des Trainings werden mit einem einfachen Beschneidungsverfahren einzelne Verbindungen wieder entfernt. Beispielprobleme: Parity, Two Spirals, Iris Artenunterscheidung, Parteizugehörigkeitsvorhersage und kontinuierliches XOR. Vergleich: Etwa gleich gute Ergebnisse wie CasCor, Backpropagation und diverse symbolische Lernverfahren (C4, NTGrowth, PROTO-TO).

Nabhan und Zomaya stellen in [262] einen Algorithmus vor, der nach einer Erzeugen-und-Prüfen-Methode (*generate and test*) arbeitet. Ausgehend von einem Netz mit einer vorgegebenen Topologie trainiert der Algorithmus jeweils soweit es geht und führt dann in jedem Schritt der Reihe nach folgende heuristisch gesteuerten Modifikationen aus: entferne Knoten, entferne Schicht, füge Knoten ein, füge Schicht ein. Die erste erfolgreiche Modifikation wird angenommen und der nächste Schritt gestartet. Eine Modifikation ist erfolgreich, wenn das entstehende Netz nach weiterem Training kleineren Fehler hat. Modifikationen, die zu bestimmten ungünstigen Topologien führen würden (z.B. „Taillen" im Netz (hourglass shapes)), werden gar nicht erst versucht. Alle verworfenen Topologien werden in einer Liste von „Sackgassen" geführt. Bei Erreichen einer solchen, wird auf die letzte Struktur mit nur einer verborgenen Schicht zurückgesetzt. Beispielprobleme: Polynome der Form $x^a + y^a$ für $a \in 1 \dots 7$, $\sin(ax)$ für $a \in 1 \dots 3$, Wagen-balanciert-Stab-System und inverse Roboter-Kinematik. In allen Fällen treten nur deshalb keine Probleme mit Überanpassung auf, weil die Trainingsmengen 1000 Beispiele enthalten, während die größten auftretenden Netze unter 100 Gewichte haben. Vergleich: Keiner.

2.8.4 Andere Verfahren

Es gibt noch eine Reihe anderer konstruktiver Lernverfahren, die Ähnlichkeiten zu neuronalen Verfahren aufweisen, jedoch selber keine sind und deshalb hier nur sehr kurz beschrieben werden. Ein Verfahren ist nicht-neuronal, wenn es z.B. eine globale Optimierungsprozedur benutzt oder wenn eine Struktur zugrundeliegt, durch die die Daten nicht im Sinne eines neuronalen Netzes hindurchfließen. Letzteres ist z.B. bei Entscheidungsbäumen der Fall, die keinen Datenfluß beschreiben, sondern eine Regel zur Steuerung eines Datenflusses.

Solche Entscheidungsbäume werden zum Beispiel in [331] aufgebaut. An jedem Knoten eines nach und nach erzeugten Baumes sitzt dabei ein einfaches Perceptron, das seine Eingaben direkt von den Eingangsknoten bezieht und dessen Ausgaben bestimmen, welches Netz als nächstes für eine Teilentscheidung zuständig ist. Die Netze an den Blättern des Baumes erzeugen die endgültigen Ausgaben. Eine ähnliche Konstruktion liegt bei [92] und bei [311] zugrunde, wobei in letzterem Fall zusätzlich eine Beschneidungsprozedur für solche Perceptron-Bäume angegeben wird. Ein anderes Verfahren mit Entscheidungsbaum ist [57].

Diverse Verfahren benutzen externe Optimierungsprozeduren, so in [173] zur Bestimmung einer optimal angepaßten Aktivierungsfunktion für ein *projection-pursuit*-Verfahren und in [27] zur Bestimmung einer glatten Interpolationsfunktion als Teil einer Bereichszerlegung. Eine Prozedur zur globalen Optimierung wird in [80] benutzt, um die dort verwendete unfreundliche (d.h. mit vielen flachen Bereichen und lokalen Minima versehene) Kostenfunktion zu minimieren.

Noch weiter gehen [259, 292, 307], die lineare Programmierung einsetzen, um Optimierungsprobleme zu lösen, die als Teil ihrer konstruktiven Algorithmen auftreten. [307] hat erwähnenswerterweise eine

Auswertung mit mehreren realen Problemen zu bieten; die Ähnlichkeit des Verfahrens mit einem neuronalen Lernverfahren beschränkt sich allerdings darauf, daß man das Resultat als ein mehrschichtiges Perceptron mit Schwellwertknoten interpretieren kann — das Lernen erfolgt aber unabhängig von dieser Repräsentation.

Weitere nicht-neuronale konstruktive Verfahren sind [114], das mit erschöpfender Aufzählung aller möglichen Architekturen zum Finden eines optimalen Netzes höherer Ordnung arbeitet, [346], das ein Simulated Annealing verwendet, um Knoten mit zufälligen Verbindungen und unterschiedlichen Aktivierungsfunktionen zu erzeugen, sowie [36], das von einem allwissenden Orakel anstelle einer beschränkten Trainingsmenge ausgeht, um gezielt gute Trennflächen für Klassifikationsnetze bauen zu können.

2.9 Aufbau und Beiträge dieser Arbeit

Aus obiger Darstellung ergibt sich folgendes Bild der Situation der Forschung über neuronale Lernverfahren:

1. Die Kunst beim Entwurf guter Lernverfahren liegt darin, einen Bias im Verfahren zu verankern, der gut an die vorgesehenen Anwendungsprobleme angepaßt ist.
2. Es gibt eine große Zahl von Vorschlägen für Lernverfahren oder Aspekte davon, wie Fehlerfunktionen, Problemrepräsentation oder Modellauswahlkriterien.
3. Die meisten dieser Vorschläge sind aber hinsichtlich ihrer Eignung nur sehr mangelhaft untersucht. Es mangelt an gesicherten und reproduzierbaren empirischen Daten.
4. Insbesondere für die konstruktiven Lernverfahren gilt, daß ihr Ansatz im Prinzip sehr vielversprechend ist, die vorliegenden Auswertungen jedoch nichtssagend oder gar widersprüchlich ausfallen.
5. Ein spezieller Mangel sehr vieler Beiträge über Lernverfahren besteht darin, daß keine quantitativen Kriterien dafür genannt werden, wann und in welchem Umfang gewisse vom Verfahren vorgesehene Operationen auszuführen sind. Die Verfahren haben dadurch freie Parameter, die oft sehr kritisch für ihren Erfolg sind, aber vom Benutzer gewählt werden müssen. Im Gegensatz dazu wäre es nötig, *automatische Lernverfahren* zu haben, deren Parameter für eine große Klasse von Problemen *fest* gewählt werden können (und bekannt sind!), um eine erfolgreiche Anwendung ohne menschlichen Eingriff zu erlauben.

Mit der vorliegenden Arbeit leiste ich deshalb die folgenden Beiträge zum Stand des Wissens und zur Unterstützung der laufenden Forschung:

1. Ich weise quantitativ nach, wie schlecht Lernverfahren ausgewertet werden (Abschnitt 3.1). Die Ergebnisse dieser Studie sollten ein Anlaß sein, der experimentellen Auswertung von Lernverfahren erheblich mehr Aufmerksamkeit zu widmen als bisher.
2. Ich definiere eine Sammlung von Benchmark-Problemen, die zur Auswertung von Lernverfahren verwendet werden können. Der Anwendungsbereich, aus dem diese Probleme stammen, läßt sich als *Diagnoseprobleme* charakterisieren (Abschnitt 3.2).
3. Ich definiere eine Menge von Regeln für die Durchführung und Publikation von Auswertungen. Mit Einhaltung dieser Regeln werden erstens viele häufig gemachte methodische Fehler vermieden, zweitens die Reproduzierbarkeit von Experimenten erreicht, die heute nur selten gegeben ist, und drittens eine häufigere Übereinstimmung von Auswertungsbedingungen hergestellt, was das heute nur selten mögliche direkte Vergleichen verschiedener Experimente ermöglicht (Abschnitte 3.3 und 3.4).
4. Ich untersuche für das Verfahren des frühen Stoppens quantitativ den Einfluß verschiedener Stoppkriterien auf Trainingsdauer und resultierenden Testfehler (Kapitel 4).

5. Ich leite eine Reihe von Varianten bekannter konstruktiver Lernverfahren her, die auf manchen Problemen eine Verbesserung derselben darstellen. Diese Verfahren werden nicht nur vorgestellt, sondern auch ausführlich im Vergleich zu den bekannten Verfahren auf ihre Leistungsfähigkeit untersucht (Kapitel 5 und 6). Für jedes neue und jedes bekannte Verfahren wird eine Formulierung als automatisches Lernverfahren gegeben, d.h. ein Verfahren, bei dem keine freien Parameter mehr vom Benutzer eingestellt werden müssen (teilweise mit Ausnahme der Wahl einer anfänglichen Netztopologie). Dies ist ein wichtiger Fortschritt gegenüber den meisten in der Literatur bisher zu findenden Beschreibungen von Lernverfahren, die in der Regel mehrere vom Benutzer einzustellenden Parameter aufweisen (manchmal zehn oder mehr), deren Wahl zum Teil sehr heikel ist.
6. Ich liefere damit eine große Menge empirischer Daten über das Verhalten von Lernverfahren mit frühem Stoppen, additiven Lernverfahren und subtraktiven Lernverfahren (Beschneidungsverfahren). Es besteht ein großer unbefriedigter Bedarf an solchen Ergebnissen, da nur sie die Beurteilung der Leistungsfähigkeit plausibler Lernverfahrensideen gestatten (Kapitel 4 bis 7). Da aufgrund der öffentlichen Verfügbarkeit der Benchmark-Sammlung die Meßanordnung exakt definiert ist, können meine Daten von anderen Forschern als Basis für weitere Vergleiche herangezogen werden. Die Daten sind elektronisch verfügbar (siehe Anhang A).

Kapitel 3

Empirische Auswertung neuronaler Lernverfahren

Arbeite lieber für die Beseitigung konkreter Mißstände
als für die Verwirklichung abstrakter Ideale.
Karl Popper

Gleicht Euch nicht
der Denkweise dieser Welt an.
Römer 12,2

I don't think one is any more justified in regarding
the two spirals problem as a model for real tasks
than one is in doing the same thing for XOR.
Michael Witbrock

In diesem Kapitel weise ich quantitativ nach, daß die Auswertung von Lernverfahren in der heutigen Forschungspraxis schwere Defizite hat (Abschnitt 3.1). Alsdann beschreibe ich eine Sammlung von Benchmark-Problemen, die dazu beitragen soll, diesem Zustand abzuhelfen (Abschnitt 3.2). Zu der Benchmark-Sammlung gehören auch Regeln für die Durchführung und Publikation von Auswertungen zur Verringerung der Häufigkeit methodischer Mängel und zur Verbesserung der Reproduzierbarkeit der Experimente (Abschnitt 3.3). Da bei statistischen Prozeduren zum Vergleich von Lernergebnissen häufig eine Normalverteilung angenommen wird, prüfe ich diese Annahme genau (Abschnitt 3.4). Ziel des Kapitels ist es, eine solide Basis für die empirische Auswertung von allgemeinen (d.h. nicht auf eine bestimmte Anwendung hin gestalteten) neuronalen Lernverfahren zu schaffen, welche bislang gefehlt hat.

3.1 Auswertung von Lernverfahren: Heutige Forschungspraxis

Dieser Abschnitt enthält eine empirische Studie, welche für eine umfangreiche Stichprobe von Zeitschriftenartikeln über neuronale Lernverfahren nachweist, daß über 80% der Beiträge schon einer moderaten Anforderung an die Qualität der experimentellen Auswertung des Lernverfahrens nicht entsprechen.

3.1.1 Ansatz der Studie

Wie wir im Abschnitt 2.8 über konstruktive Lernverfahren gesehen haben, wurde in der Vergangenheit die experimentelle Auswertung von vorgeschlagenen Lernverfahren nur unzureichend durchgeführt. Da viele der dort angeführten Artikel schon einige Jahre alt sind, stellt sich die Frage, ob diese Situation immer noch anhält. Ebenso ist fraglich, ob das gleiche Problem auch für Beiträge gilt, die nicht konstruktive Algorithmen, sondern andere Lernverfahren zum Gegenstand haben.

Deshalb habe ich eine empirische Studie durchgeführt, in der ich alle Artikel über Lernverfahren untersucht habe, die 1993 und in der ersten Hälfte von 1994 in den angesehenen Zeitschriften „Neural Networks" (Band 6 und Nummern 1 bis 5 von Band 7) und „Neural Computation" (Band 5 und Nummern 1 bis 4 von Band 6) erschienen sind. Dies sind die beiden ältesten Zeitschriften über neuronale Netze auf dem Markt. Die Studie ist einschließlich der dabei erfaßten Rohdaten als [15] veröffentlicht.

Zur Beurteilung der Qualität der Auswertung werden ausschließlich zwei objektive Kriterien verwendet:

1. Die Zahl verwendeter Beispielprobleme und
2. die Zahl bereits bekannter Lernverfahren, die zum Vergleich herangezogen werden.

Beides wird unten noch genauer beschrieben. Diese Kennzahlen sind unzureichend, um eine hohe Qualität einer Auswertung begründen zu können: Auch wenn die Zahlen hoch sind, kann die Auswertung schlecht sein — zum Beispiel aufgrund methodischer Fehler. Es kann aber aus niedrigen Kennzahlen auf eine niedrige Qualität geschlossen werden. Da es um die Untersuchung der Hypothese geht, daß die Qualität typischerweise niedrig ist, bewirken diese Kennzahlen also einen konservativen Ansatz der Studie.

Aufgrund der Einfachheit und Objektivität der verwendeten Kennzahlen, kann man bei der Erfassung der Daten der Studie von einer geringen Fehlerquote ausgehen — eine Messung dieses Fehlers durch Wiederholung der Zählung mit mehreren Zählpersonen war leider aus Aufwandsgründen nicht möglich. Siehe jedoch [227] für eine Abschätzung des Fehlers in einer ähnlichen Studie. Wie wir unten sehen werden, ändern sich aber die Resultate der Studie auch dann kaum, wenn ein großer Fehler angenommen wird.

3.1.2 Klassifikation von Artikeln

Beide untersuchten Zeitschriften decken einen recht weiten Bereich ab; bei weitem nicht alle ihrer Beiträge befassen sich mit Lernverfahren zum Lösen praktischer Probleme. Wir teilen deshalb die Artikel zunächst in folgende drei einander ausschließende Kategorien ein:

Theorie. Ein Artikel gehört zur Kategorie „Theorie" genau dann, wenn seine wesentlichen Beiträge in formal bewiesenen Sätzen bestehen.

Modellierung. Artikel, die sich hauptsächlich mit der formalen Modellierung von Aspekten biologischer Nervensysteme oder der Diskussion der Eigenschaften solcher Modelle oder mit anderen Aspekten von biologischer Plausibilität befassen, gehören in die Kategorie „Modellierung".

Lernverfahren. Artikel, deren Hauptbeitrag in einem neuen Lernverfahren (oder Teilen davon) zur Anwendung auf praktische Probleme besteht, gehören in die Kategorie „Lernverfahren". Empirische Studien, die mehrere Verfahren vergleichen ohne selbst neue vorzustellen, wurden hier eingeschlossen, da sie recht selten sind — sie bilden nur 5% der Kategorie. Das gleiche gilt für Artikel über Techniken zur Anwendung bekannter Lernverfahren auf bestimmte Problemfelder.

Andere. Alle Artikel, die in keine der obigen Kategorien passen, gelangen in die „Andere"–Kategorie. Dies sind beispielsweise Übersichtsartikel und Beiträge über Hardware.

Der Gegenstand der Studie sind im weiteren nur die Artikel der Kategorie „Lernverfahren“. Dies sind 113 von insgesamt 271 Artikeln; die genaue Klassifikation aller Artikel ist in [15] dokumentiert. Im Zweifelsfall wurde ein Artikel *nicht* in „Lernverfahren“ klassifiziert, um eine negative Beeinflussung der Resultate durch solche Artikel zu vermeiden, die gar nicht vorrangig ein Lernverfahren präsentieren wollen, und folglich auch keine gute Auswertung enthalten. Ferner sind alle Artikel aus „Neural Computation“, die dort als „Note“ erschienen, aufgrund ihrer Kürze ausgeschlossen worden.

3.1.3 Ermittlung der Kennzahlen

Um die Zählung der Beispielprobleme aussagekräftiger zu gestalten, wurde jedes Beispielproblem einer von drei Klassen zugeordnet.

Künstliche Probleme sind solche deren Daten künstlich erzeugt wurden unter Verwendung einer einfachen logischen oder arithmetischen Formel. Hierzu zählen die alten „Klassiker“ wie XOR und Parity, Encoder, Symmetry, two-or-more clumps und so weiter. Künstliche Probleme sollten allenfalls zur Illustration von Verfahren dienen, können aber nicht zu deren Bewertung herangezogen werden, weil sie keine Aussagen über das Verhalten des Lernverfahrens für reale Probleme zulassen.

Realistische Probleme bestehen auch aus künstlichen Daten, die jedoch mit einem Modell erzeugt wurden, das Eigenschaften hat, wie wir sie auch in realen Problemen erwarten können. Es gibt genau drei Klassen von Datenerzeugungsprozessen, deren Resultate ich als realistische Probleme betrachtet habe: Erstens die Datenerzeugung unter Verwendung eines komplexen und realistischen mathematischen Modells eines physikalischen Systems, beispielsweise ein Wagen-balanciert-Stab (cart/pole) System oder Roboterkinematik. Zweitens die Datenerzeugung mit chaotischen mathematischen Prozessen, beispielsweise der Mackey-Glass Differentialgleichung. Drittens Datenerzeugung unter Verwendung stochastischer Prozesse, beispielsweise Mixturen von Gaußverteilungen. Realistische Probleme sind nützlich zur Charakterisierung des Verhaltens von Lernverfahren unter bekannten Bedingungen.

Reale Probleme bestehen aus Daten, die tatsächliche Beobachtungen eines Phänomens in der physikalischen Welt repräsentieren. Solche Daten enthalten meist eine gewisse Menge an Fehlern und Rauschen. Im Gegensatz zu realistischen Problemen haben reale Probleme eine nicht genau bekannte Charakteristik (überraschende Merkmale). Wir wollen Lernverfahren finden, die gut mit solchen Problemen mit nicht genau bekannter Charakteristik zurechtkommen. Wie gut ein bestimmtes Lernverfahren dies schafft, läßt sich am besten mit realen Problemen prüfen.

Bei der Zählung der Beispielprobleme zählen Variationen desselben Problems genau dann als einzelne Probleme, wenn es plausibel ist, daß der Vergleich zweier Verfahren auf der Variation wesentlich anders ausfällt als auf dem Originalproblem. Häufig treten z.B. zwei Varianten auf, eine mit und eine ohne Rauschen in den Daten. Eine stark unterschiedliche Problemrepräsentation führt ebenfalls zu einer separat gezählten Variante — wobei „stark unterschiedlich“ nicht formal definiert werden kann, sondern nach meinem subjektiven Ermessen beurteilt wird.

Ein Beispielproblem gilt als *verwendet* dann, wenn irgendeine Art von quantitativen Daten über das Lernen dieses Problems im Artikel berichtet wird, sei es Lerngeschwindigkeit, Konvergenzwahrscheinlichkeit, Fehler auf der Trainingsmenge oder Fehler auf einer Testmenge (Generalisierung). Auch dieses Kriterium ist konservativ.

Die Zählung von zum Vergleich herangezogenen Verfahren umfaßt alle Verfahren, die nicht im Artikel neu eingeführt wurden; die Verfahren, die Gegenstand des Artikels sind, werden nicht gezählt. Eine Ausnahme sind empirische Studien, bei denen alle Verfahren gezählt werden. Ein Vergleichsverfahren wird gezählt, sobald es irgendwo im Artikel zu einem Vergleich herangezogen wird, es braucht nicht für alle Beispielprobleme oder für alle eingeführten neuen Verfahren der Vergleich angegeben zu sein.

Ein möglicher Einwand gegen diese Studie betrifft Lernverfahren für spezialisierte Anwendungsgebiete, beispielsweise Erkennung handgeschriebener Buchstaben. Man könnte argumentieren, daß es hierbei von vornherein nur ein Problem gibt und deshalb Arbeiten über solche Themen in der Studie grundsätzlich schlecht abschneiden müssen. Dies ist nicht ganz zutreffend. Auch wenn ein Verfahren nur für ein sehr enges Problemfeld gemacht wurde, lassen sich mehrere *Instanzen* des Problems finden, die sich zum Beispiel durch die unterschiedliche Herkunft ihrer Trainingsmengen unterscheiden. Die Verwendung mehrerer solcher Instanzen ist sinnvoll, um zu belegen, daß das vorgeschlagene Verfahren das betrachtete Problem tatsächlich allgemein gut löst — nicht nur für eine ganz bestimmte Menge von Beispieldaten.

3.1.4 Ergebnisse und Diskussion

Die während der Studie erfaßten Rohdaten finden sich in [15]. Wir werden hier nur einige summarische Schaubilder diskutieren. Da die Unterschiede zwischen „Neural Networks" und „Neural Computation" nicht groß sind, diskutieren wir nur das Gesamtbild; die Einzelergebnisse sind jedoch in den Schaubildern eingetragen.

Betrachten wir als erstes die Gesamtzahl von Problemen, die zur Auswertung benutzt werden, wie in Abbildung 3.1 dargestellt. Das Diagramm ist folgendermaßen zu lesen: Auf der Abszisse (x-Achse),

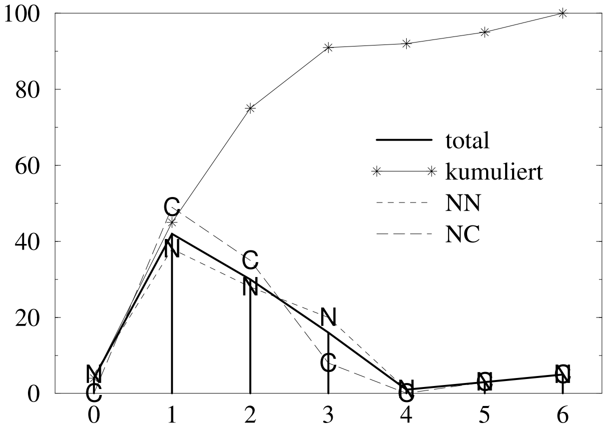


Abbildung 3.1: Prozentsatz y von Artikeln, die insgesamt x verschiedene Beispielprobleme benutzen.

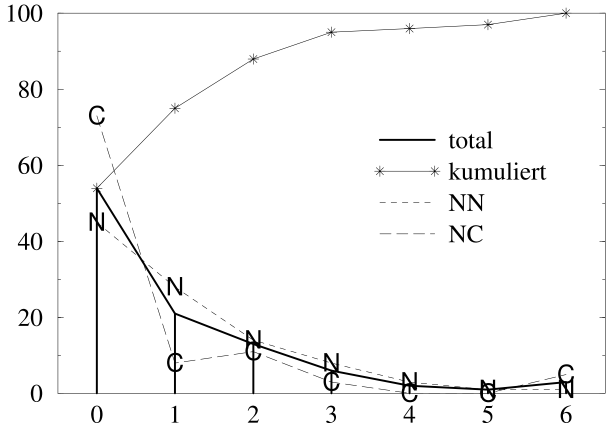


Abbildung 3.2: Prozentsatz y von Artikeln, die x verschiedene *künstliche* Beispielprobleme benutzen.

finden wir die Artikelklassen von „0 Beispielprobleme benutzt" bis hin zu „5 Beispielprobleme benutzt". Der letzte Punkt, $x = 6$, steht für „6 oder mehr Beispielprobleme benutzt". Die Ordinate (y-Achse) gibt zu jeder dieser Klassen den Prozentsatz von Artikeln an, die ihr angehören. Die mit einer dicken Linie dargestellte Kurve gibt diese Werte für die Summe aller Lernverfahrens-Artikel an; die gestrichelten Linien zeigen die Ergebnisse für jede der beiden Zeitschriften Neural Networks (NN) bzw. Neural Computation (NC) einzeln. Die mit Sternchen markierte Linie zeigt die Akkumulation der Werte der dicken Linie und kann zum Ablesen von Quantilen benutzt werden. Alle späteren Diagramme haben dieselbe Struktur.

Wie wir sehen, haben 4% aller Artikel überhaupt keine Auswertung und nur 25% aller Artikel verwenden mehr als zwei Beispielprobleme. Es ist überraschend, daß man offenbar gelegentlich in angesehenen Zeitschriften Lernverfahren veröffentlichen kann, ohne sie experimentell auszuwerten, aber noch verblüffender ist, wie selten in Artikeln eine breite Menge von Problemen verwendet wird. Nur 9% aller Artikel verwenden mehr als 3 Probleme.

Abbildung 3.2 zeigt die Anzahl benutzter künstlicher Probleme. Hierzu ist nichts weiter zu sagen. Eine große Zahl künstlicher Probleme ist weder gut noch schlecht, sie vergeudet allenfalls Platz. Eine interessante Randbeobachtung während der Studie war, daß immerhin 18% aller Artikel den Urvater aller Benchmarkprobleme benutzen, nämlich XOR bzw. n-bit Parity.

Abbildung 3.3 zeigt die Anzahl benutzter realistischer Probleme. Wie erwähnt, sind solche Probleme

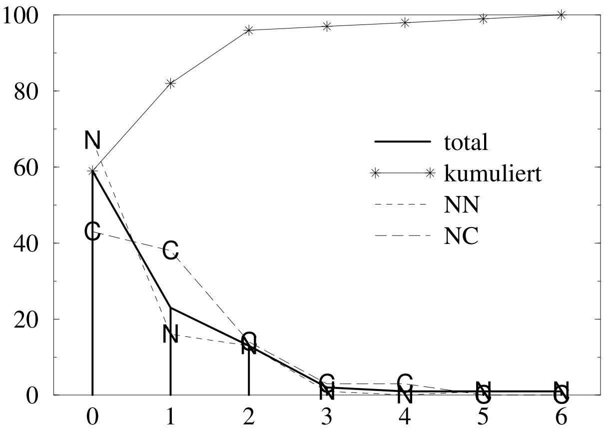


Abbildung 3.3: Prozentsatz y von Artikeln, die x verschiedene *realistische* Beispielprobleme benutzen.

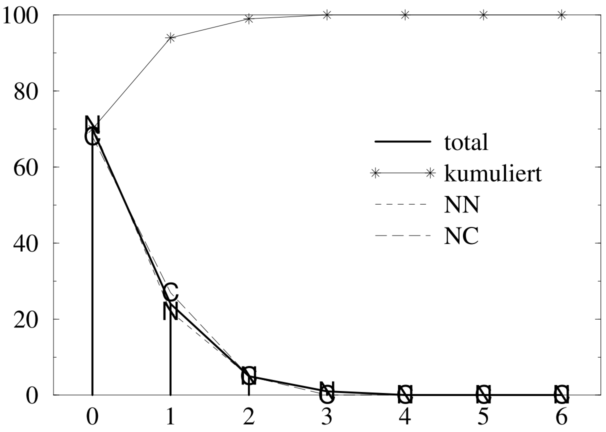


Abbildung 3.4: Prozentsatz y von Artikeln, die x verschiedene *reale* Beispielprobleme benutzen.

nützlich, um die Leistung eines Verfahrens unter halbwegs realistischen aber genau bekannten Bedingungen zu charakterisieren. Trotz dieser Nützlichkeit werden sie kaum eingesetzt: 59% aller Artikel benutzen überhaupt kein realistisches Problem, nur 4% benutzen mehr als zwei und 3% mehr als drei. Eine experimentelle Erforschung der Frage „Für welche Arten von Problemen ist dieser Algorithmus geeignet?“ wird offenbar kaum betrieben.

In Abbildung 3.4 sehen wir die Anzahl verwendeter realer Probleme.[1] Natürlich läßt sich nicht sagen, wieweit sich die Ergebnisse für ein reales Problem auf andere reale Probleme übertragen lassen, aber ebensowenig läßt sich von Ergebnissen für realistische Probleme auf die Ergebnisse für reale Probleme schließen. Folglich sollte ein Verfahren zumindest mit *einigen* realen Problemen ausprobiert werden, insbesondere wegen der darin enthaltenen unerwarteten Merkmale. Ganz im Gegensatz dazu ist der tatsächliche Gebrauch von realen Problemen erschütternd selten. 70% aller Artikel verwenden überhaupt keines, nur ein einziger (1%) benutzt mehr als zwei und keiner mehr als drei.

Faßt man die Zahlen für realistische und reale Probleme zusammen, wie in Bild 3.5 dargestellt, sieht die Situation immer noch sehr traurig aus. 34% aller Artikel benutzen weder ein realistisches noch ein reales Problem, nur 6% benutzen mehr als zwei und lediglich 3% mehr als drei. Ein Drittel aller Artikel läßt also eine wenigstens ansatzweise sinnvolle Auswertung gänzlich vermissen.

Abbildung 3.6 zeigt schließlich die Anzahl von zum Vergleich herangezogenen bekannten Verfahren. Auch hier bietet sich ein wüstes Bild. 34% aller Beiträge vergleichen das vorgeschlagene Verfahren überhaupt nicht mit anderen (nach dem Motto „Hier habt ihr's; nun seht selber, was ihr damit anfangt.“). Nur 19% vergleichen mit mehr als zwei alternativen Verfahren. Dies wäre kein großes Problem, wenn die meisten Artikel standardisierte Probleme in standardisierten Versuchsaufbauten verwenden würden, aber das ist nicht der Fall. Vielmehr ist eine direkte Vergleichbarkeit zweier Lernverfahrensauswertungen eine seltene Ausnahme. Ohne solche Vergleichbarkeit müßte aber die sogenannte Auswertung in einem Drittel der Artikel eigentlich eine Nabelschau genannt werden.

Was sagen uns diese Zahlen nun über die Qualität der Auswertungen insgesamt? Nehmen wir an, wir definieren den folgenden sehr gemäßigten Standard: *Die Auswertung eines Lernverfahrens heißt*

[1] *Der Zen-Meister sagt: Der einzige Weg, um herauszufinden, wie gut ein Verfahren für ein reales Problem funktioniert, ist zu testen, wie gut es für ein reales Problem funktioniert.*

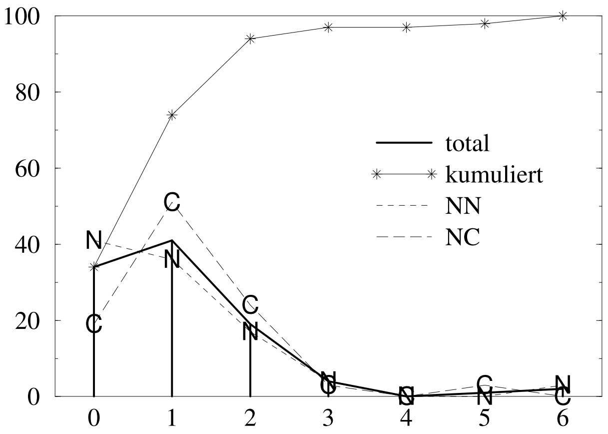


Abbildung 3.5: Prozentsatz y von Artikeln, die x verschiedene *realistische oder reale* Beispielprobleme benutzen.

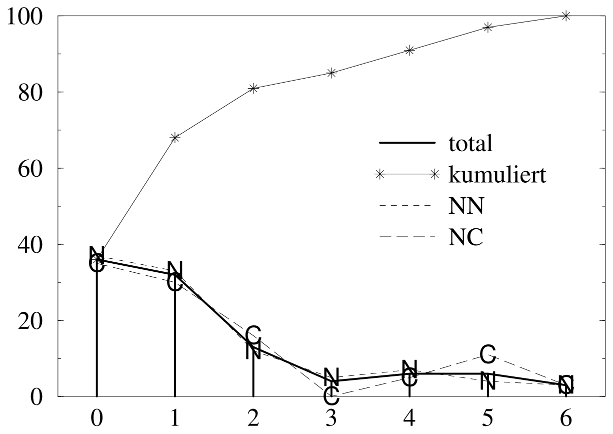


Abbildung 3.6: Prozentsatz y von Artikeln, die x verschiedene bekannte Lernverfahren zum Vergleich heranziehen.

akzeptabel, wenn sie mindestens zwei reale oder realistische Probleme heranzieht und die Resultate mit mindestens einem bereits bekannten Verfahren vergleicht.

> Diesem Standard genügen von den im untersuchten Zeitraum von Neural Networks und Neural Computation veröffentlichten Beiträgen über Lernverfahren 82% *nicht*!

Anders ausgedrückt: Nur jeder fünfte Artikel enthält eine wenigstens möglicherweise aussagekräftige empirische Bewertung seines (analytisch nicht bewertbaren) Lernverfahrensvorschlags.

3.1.5 Folgerungen

Da erst die experimentelle Auswertung die Vorstellung eines neuen Lernverfahrens zu einem wissenschaftlichen Fortschritt macht, ergeben sich aus obiger Untersuchung die folgenden Forderungen.

1. Editoren und Gutachter sollten höhere Maßstäbe für die Akzeptanz eines Zeitschriftenartikels anlegen. Beiträge über Lernverfahren ohne ausreichende experimentelle Auswertung sind abzulehnen.
2. Forscher müssen genügend Ressourcen für eine experimentelle Auswertung ihrer Arbeiten vorsehen.
3. Die Neuroinformatik-Gemeinde sollte öffentlich zugängliche Sammlungen von Benchmark-Problemen erstellen und benutzen, mit denen die Durchführung einer Ausführung erleichtert wird und die Vergleichbarkeit der Ergebnisse zunimmt. Nur in wenigen Teilfeldern, wie Spracherkennung und Schrifterkennung ist dies bislang in ausreichendem Maß geschehen.
4. Diese Problemsammlungen sollten von standardisierten Experimentaufbauten und standardisierten Präsentationsformaten für die Aufbauten und Ergebnisse begleitet werden und die Reproduzierbarkeit und Vergleichbarkeit der Auswertungen zu verbessern.

Eine solche Sammlung von Beispielproblemen und zugehörigen Verwendungsregeln wird in den folgenden Abschnitten vorgestellt.

3.2 Die Proben1 Benchmark-Sammlung

Um nicht nur zu meckern, sondern auch etwas zur „Beseitigung konkreter Mißstände“ beizutragen, habe ich eine öffentlich verfügbare Sammlung von Benchmark-Problemen, genannt PROBEN1, erstellt.

Eine solche Sammlung gab es bisher nicht — vermutlich, weil ihre sorgfältige Zusammenstellung und Aufbereitung verblüffend viel Arbeit macht. Dieser Abschnitt steckt zunächst den Bereich ab, aus dem die PROBEN1 Benchmark-Probleme stammen und beschreibt dann jedes Problem kurz in einem eigenen Abschnitt. Die Probleme sind der Reihe nach zuerst die Klassifikationsprobleme cancer, card, diabetes, gene, glass, heart, horse, mushroom, soybean und thyroid und dann die Approximationsprobleme building, flare und hearta. Abschließend folgt ein Abschnitt der die wichtigsten formalen Daten der Probleme zusammenfaßt und Ergebnisse für das Lernen mit linearen Netzen präsentiert.

Als Handbuch zur Verwendung der Benchmark-Sammlung dient der Report [14]. Dort ist auch die Herkunft der Datensätze dokumentiert. PROBEN1 ist öffentlich verfügbar per anonymem FTP von der Maschine ftp.cs.cmu.edu im Verzeichnis /afs/cs/project/connect/bench/contrib/prechelt. Dieser Rechner verwaltet das sogenannte „Neural Bench“ Archiv, eine Benchmark-Sammlung für neuronale Lernverfahren, die schon vor vielen Jahren angelegt wurde und als allmählich wachsende, offene Sammlung gedacht war. Dennoch befinden sich darin bisher nur 4 reale Probleme (zuzüglich drei künstliche: XOR, Parity und Two Spirals).

3.2.1 Bereich der Proben1 Benchmarks

Die Forschung nach Lernverfahren für neuronale Netze ist ein Feld, auf dem für sehr unterschiedliche Anwendungen geforscht wird. Teilfelder, wie etwa Schrifterkennung, Maschinensehen oder Sprachverstehen sind hochspezialisiert und benötigen folglich auch spezialisierte Benchmarks. Auch manche allgemeinen Teile der Lernverfahrensforschung verlangen nach bestimmten Eigenschaften in den Lernproblemen; so etwa für rekurrente Netze, daß die Zeit oder der Raum als Dimensionen in der Problemstellung auftauchen. Eine einzelne Sammlung von Beispielproblemen kann also schwerlich das gesamte Feld abdecken. PROBEN1 beschränkt sich im Sinne der in dieser Arbeit betrachteten Lernverfahren auf Klassfikations- und Approximationsprobleme für überwachtes Lernen mit vorwärtsgerichteten Netzen.

Den Bereich, aus dem die PROBEN1 Probleme stammen, bezeichne ich als *Diagnoseprobleme*. Diese lassen sich folgendermaßen charakterisieren:

1. Die verwendeten Eingabeattribute sind ähnlich denen, die ein Mensch zum Lösen derselben Aufgabe verwenden würde.
2. Die Ausgaben repräsentieren entweder eine Klassifikation in eine kleine Zahl verständlicher Klassen oder die Vorhersage einer kleinen Zahl verständlicher, kontinuierlicher Größen.
3. Die gleichen Probleme werden tatsächlich oft von Menschen gelöst.
4. Lernbeispiele sind teuer. Deshalb sind die Datenmengen nicht sehr groß.
5. Häufig fehlen einige Attributwerte.

Eine besonders wichtige Eigenschaft ist dabei die geringe Anzahl von verfügbaren Trainingsbeispielen. Dies macht die Anwendung von Lernverfahren besonders kritisch, weil schon die geringste Verschwendung von Information zu empfindlichen Verschlechterungen des Ergebnisses führen kann, zugleich aber stets eine heftige Überanpassung vermieden werden muß. Andererseits erleichtert aber die geringe Datenmenge die Anwendung aufwendiger Lernverfahren, zu denen die in dieser Arbeit behandelten überwiegend gerechnet werden müssen.

PROBEN1 enthält 15 Lernprobleme, die aus 12 verschiedenen Anwendungsbereichen stammen. Darunter sind Anwendungen aus der Medizin, der Botanik, der Mikrobiologie, der Astronomie, dem Bauingenieurwesen, dem Finanzwesen, der Gerichtsmedizin und der Landwirtschaft. Alle haben den Charakter von Diagnoseproblemen wie oben beschrieben.

3.2.2 Allgemeiner Aufbau

Die beiden wesentlichen Eigenschaften der Benchmarks wie sie in PROBEN1 präsentiert werden, sind

1. exakt definierte Eingabe- und Ausgaberepräsentation und
2. exakt definierte Unterteilung in Trainings-, Validations- und Testmenge.

Beides markiert einen wesentlichen Fortschritt gegenüber den meisten bisher verfügbaren Benchmarks. Da der überwiegende Teil der Datensätze, die bei Arbeiten zu neuronalen Lernverfahren eingesetzt werden (insofern überhaupt mit realen Problemen gearbeitet wird), ursprünglich von Forschern aus dem Gebiet des maschinellen Lernens beschafft, formatiert und zur Verfügung gestellt wurden, sind diese Daten meistens symbolisch kodiert. Das bedeutet, daß zur Verwendung mit neuronalen Netzen zunächst Entscheidungen getroffen werden müssen über die Repräsentation, insbesondere der ordinalen und nominalen Attribute mit mehr als zwei Werten und über die Darstellung fehlender Attributwerte (nur durch Standardwert oder zusätzlich per Extrakoeffizient). Da unterschiedliche Lösungen möglich sind, lassen sich die früher publizierten Ergebnisse meist nicht miteinander vergleichen, denn die Forscher veröffentlichen die von ihnen gewählte Kodierung aus Platzgründen in der Regel nicht. In PROBEN1 ist die Repräsentation aller Datensätze unmittelbar für neuronale Netze geeignet. Bei der Festlegung dieser Repräsentation habe ich sorgfältig versucht, eine Darstellung zu wählen, die die Information in für neuronale Netze gut verarbeitbarer Form enthält, dabei aber die übertriebene Aufblähung der Darstellung hinsichtlich der Anzahl nötiger Koeffizienten vermeidet. Insbesondere bei den Attributen mit fehlenden Werten, die es in den PROBEN1-Problemen in reicher Zahl gibt, erfordert die Darstellungsentscheidung eine sorgfältige Abwägung zwischen Aufwand und Nutzen.

Fehlende Vergleichbarkeit herrscht in bisherigen Publikationen auch für die Partitionierung der Daten in Trainingsdaten (Trainings- und ggf. Validationsmenge) und Testdaten (Testmenge). Die gewählte Unterteilung kann einen erheblichen Einfluß auf das Ergebnis haben (siehe beispielsweise Tabelle 3.9 auf Seite 64), wird aber in der Regel nicht veröffentlicht.

In PROBEN1 besteht jeder Datensatz aus einer Datei mit einem sehr einfachen textuellen Format. In mehreren Kopfzeilen ist die Anzahl von Ein- und Ausgabekoeffizienten beschrieben sowie die Unterteilung des Datensatzes in Trainings-, Validations- und Testmenge. Dahinter folgt ein Beispiel pro Zeile, das umittelbar die komplett numerisch kodierten und normalisierten Eingabe- und Ausgabekoeffizienten angibt, mit denen ein neuronales Netz trainiert werden kann. Beispiel (aus glass1.dt):

```
bool_in=0
real_in=9
bool_out=6
real_out=0
training_examples=107
validation_examples=54
test_examples=53
0.281387 0.36391 0.804009 0.23676 0.643527 0.0917874 0.261152 0 0 1 0 0 0 0 0
0.260755 0.341353 0.772829 0.46729 0.545966 0.10628 0.255576 0 0 0 1 0 0 0 0
```

Für jedes der 15 Probleme stehen drei verschiedene Datensätze zur Verfügung; für das Problem 'glass' beispielsweise glass1, glass2 und glass3. Diese sind identisch bis auf die Reihenfolge der Beispiele. Jede Reihenfolge ist eine Zufallspermutation. Diese unterschiedliche Anordnung führt zu drei verschiedenen Partitionierungen der Daten in Trainings-, Validations-, und Testmenge. Durch Verwendung dreier verschiedener Partitionierungen lassen sich die recht erheblichen Effekte verringern, die eine zufällige Wahl von Trainings-, Validations- und Testmenge bei Datensätzen mit nicht sehr vielen Beispielen nach sich zieht: Oft unterscheiden sich die erzielbare Generalisierungsleistung je nach zufälliger Aufteilung erheblich; siehe z.B. die Tabelle 3.9 auf Seite 64. Dies veranschaulicht, wie wichtig es für die Vergleichbarkeit von publizierten Ergebnissen ist, eine exakt definierte Aufteilung der Datenmenge vorzunehmen. Da diese Effekte reine Artefakte eines stochastischen Prozesses mit hoher Varianz (aufgrund zu kleiner Stichprobengröße) sind und möglicherweise Eigenschaften der Lernaufgabe vorspiegeln, die es gar nicht gibt, ist ihre Unterdrückung erwünscht.

Die nun folgenden Unterabschnitte beschreiben kurz je eines der PROBEN1 Benchmark-Probleme.

3.2.3 Cancer

Diagnose von Brustkrebs. Aufgabe: Klassifiziere einen Tumor als gutartig oder bösartig aufgrund der Daten aus einer mikroskopischen Untersuchung von Zellgewebe. Eingabeattribute sind beispielsweise die Klumpendicke, die Gleichmäßigkeit von Zellgröße und Zellform, die Stärke des Randzusammenhangs und die Häufigkeit nackter Zellkerne.

9 Eingaben, 2 Ausgaben, 699 Beispiele. Alle Eingaben sind kontinuierlich. Zwei Drittel der Beispiele sind aus der Klasse „gutartig". Der Datensatz stammt von den University of Wisconsin Hospitals von William H. Wolberg [404].

3.2.4 Card

Kreditkartenausgabe. Aufgabe: Sage die Ausgabe oder Verweigerung einer Kreditkarte an einen Bankkunden voraus. Jedes Beispiel repräsentiert einen Kreditkartenantrag und die Angabe, ob die Bank dem Antrag stattgegeben hat oder nicht. Aus Gründen der Vertraulichkeit ist die Bedeutung der einzelnen Attribute nicht veröffentlicht.

51 Eingaben, 2 Ausgaben, 690 Beispiele. Dieser Datensatz hat eine gute Mischung aus kontinuierlichen Attributen, nominalen Attributen mit wenigen Werten und nominalen Attributen mit vielen Werten. 5% der Beispiele haben außerdem einen oder mehrere fehlende Attributwerte. 44% der Beispiele sind positiv.

3.2.5 Diabetes

Diagnose von Diabetes. Aufgabe: Gegeben eine Beschreibung einer Person mit Angaben wie Alter, Anzahl von Schwangerschaften, Blutdruck, Körpermassenindex, Resultat eines Glukose-Toleranztests und anderen, diagnostiziere das Vorliegen oder Nichtvorliegen von Diabetes.

8 Eingaben, 2 Ausgaben, 768 Beispiele. Alle Eingaben sind kontinuierlich. Zwei Drittel der Beispiele sind Diabetes negativ. Obwohl der Datensatz laut seiner Originaldokumentation keine fehlenden Attributwerte aufweist, gibt es darin einige sinnlose Nullwerte. Da diese wie normale Werte behandelt werden, wahrscheinlich aber eigentlich fehlende Werte darstellen, weist der Datensatz also eine Anzahl von Fehlern (Rauschen) auf.

3.2.6 Gene

Donor/Akzeptor-Detektion. Aufgabe: Gegeben ein Abschnitt von 60 DNS Sequenzelementen (Nukleotiden), entscheide, ob die Mitte des Abschnitts ein sogenannter „Donor" (Spender) ist oder ein „Akzeptor" oder keins von beiden.

120 Eingaben, 3 Ausgaben, 3175 Beispiele. Jedes Nukleotid, es gibt vier verschiedene, ist binärkodiert mittels zweier zweiwertiger Eingabekoeffizienten dargestellt. Der Datensatz enthält 25% Spender und 25% Akzeptoren.

3.2.7 Glass

Glasartenklassifikation. Aufgabe: Gegeben die Resultate einer chemischen Analyse (Prozentgehalt für 8 verschiedene chemische Elemente) und den Brechungsindex eines Glassplitters, entscheide, welcher von 6 verschiedenen Glasarten der Splitter zugehört. Es gibt zwei Arten von Fensterglas für Häuser, Autofensterglas, Autoschweinwerferglas, Flaschenglas und Glas für Trinkgefäße. Das Problem stammt aus der Gerichtsmedizin.

9 Eingaben, 6 Ausgaben, 214 Beispiele. Alle Eingaben sind kontinuierlich, zwei von ihnen haben nur sehr geringe Korrelation mit dem Ergebnis. Da die Anzahl von Beispielen in diesem Datensatz recht klein ist, reagiert er empfindlich auf Lernverfahren, die Information verschwenden. Die sechs Klassengrößen sind 70, 76, 17, 13, 9 bzw. 29 Beispiele.

3.2.8 Heart

Diagnose von Herzgefäßverengung. Aufgabe: Gegeben persönliche Daten wie Alter, Geschlecht, Rauchgewohnheiten und subjektive Schmerzbeschreibungen, sowie die Resultate medizinischer Untersuchungen wie Blutdruck und Elektrokardiogramm-Messungen, sage voraus ob mindestens eines von vier Hauptgefäßen zu mehr als 50% verengt ist oder nicht.

35 Eingaben, 2 Ausgaben, 920 Beispiele. Die meisten der Attribute haben fehlende Werte, manche sogar recht viele: Für die Attribute 10, 12 und 11 fehlen 309, 486 bzw. 611 der Werte. Die meisten anderen Attribute haben etwa 60 fehlende Werte. Zusätzliche binäre Eingabekoeffizienten werden in der Repräsentation für das neuronale Netz verwendet, um das Fehlen dieser Werte anzuzeigen. Der Datensatz ist die Summe von vier kleineren Datensätzen, die von vier verschiedenen Kliniken stammen: Cleveland Clinic Foundation (Robert Detrano), Hungarian Institute of Cardiology (Andras Janosi), V.A. Medical Center Long Beach (Robert Detrano) und Universitätskrankenhaus Zürich (William Steinbrunn). Es gibt eine zweite Version des Datensatzes heart, genannt heartc, der nur die Cleveland-Daten enthält (303 Beispiele). Dieser Datensatz ist der vollständigste der vier; es fehlen nur insgesamt zwei Attributwerte. Ferner gibt es noch zwei weitere Versionen dieses Problems, genannt hearta und heartac (das „a“ steht für analog), korrespondierend zu heart und heartc. In hearta und heartac wird mit Hilfe einer einzelnen Ausgabe, die kontinuierlich kodiert ist, nicht nur vorhergesagt, ob überhaupt ein Gefäß verengt ist, sondern wieviele Gefäße verengt sind (0 bis 4). Diese Versionen sind also Approximationsprobleme anstatt Klassifikationsprobleme. Bei heart und hearta sind 45% der Patienten ohne ein verengtes Gefäß, bei heartc und heartac sind es 54%.

3.2.9 Horse

Ausgang von Pferdekoliken. Aufgabe: Gegeben die Resultate einer umfangreichen tierärztlichen Untersuchung eines Pferdes, das eine Kolik hat, sage voraus, ob das Tier überlebt, stirbt oder eingeschläfert wird.

58 Eingaben, 3 Ausgaben, 364 Beispiele. In 62% der Beispiele überlebt das Pferd, in 24% stirbt es und in 14% wird es eingeschläfert. Dieser Datensatz hat außerordentlich viele fehlende Attributwerte: 30% aller Attributwerte fehlen. Wie bei heart werden auch hier zusätzliche Eingabekoeffizienten eingesetzt, um das Fehlen von Attributwerten anzuzeigen.

3.2.10 Mushroom

Eßbarkeit von Pilzen. Aufgabe: Gegeben eine Beschreibung eines Pilzes nach Form, Farbe, Geruch und Lebensraum, entscheide, ob der Pilz eßbar ist oder giftig.

125 Eingaben, 2 Ausgaben, 8124 Beispiele. Nur ein Attribut hat fehlende Werte; dort fehlen etwa 30%. Dieses Problem ist ein besonderes in mehrerlei Hinsicht: Erstens ist es dasjenige mit den meisten Eingabekoeffizienten, zweitens dasjenige mit den meisten Beispielen, drittens das als Lernaufgabe einfachste und viertens das einzige, dessen Daten keine „realen“ Daten sind in dem Sinne, daß sie als Messungen in der physikalischen Welt aufgenommen wurden; stattdessen sind sie Beschreibungen aus einem Bestimmungsbuch („The Audubon Society Field Guide to North American Mushrooms“). Die Beispiele gehören zu 23 Arten von Lamellenpilzen der Familien Agaricus und Leopiota. 52% der Beispiele sind eßbar, äh, gehören zur Klasse „eßbar“.

3.2.11 Soybean

Diagnose von Sojapflanzenkrankheiten. Aufgabe: Gegeben ist eine Beschreibung einer Sojabohne (z.B. ob die Größe und Farbe normal sind) und der zugehörigen Pflanze (z.B. die Größe von Punkten auf den Blättern, ob diese Punkte ein Halo haben, ob die Pflanze normalstark wächst, ob die Wurzeln faulig sind) sowie Information über die Vergangenheit des Pflanzenlebens (z.B. ob sich die Ernte in den letzten ein oder zwei Jahren verändert hat, ob die Samen schutzbehandelt wurden, die Umgebungstemperatur). Diagnostiziere, welche von 19 verschiedenen Sojabohnen-Krankheiten vorliegt.

35 Eingaben, 19 Ausgaben, 683 Beispiele. Dies ist der Datensatz mit der höchsten Zahl von Klassen. Das Sojabohnenproblem ist schon oft in der Literatur zu neuronalen Netzen und maschinellem Lernen verwendet worden, aber leider gibt es mehrere verschiedene Versionen (mindestens drei, aber niemand weiß es genau) dieser Daten, so daß Vergleiche nahezu unmöglich sind. Viele frühere Publikationen verwendeten nur 15 Klassen, weil die übrigen Klassen als zu klein galten. Diese „kleinen“ Klassen enthalten 8, 14, 15 bzw. 16 Beispiele, die meisten anderen Klassen 20 Beispiele.

3.2.12 Thyroid

Diagnose der Schilddrüsenfunktion. Aufgabe: Gegeben die Daten aus einer Patientenbefragung und einer Patientenuntersuchung, stelle fest, ob die Schilddrüse Überfunktion, Normalfunktion oder Unterfunktion hat.

21 Eingaben, 3 Ausgaben, 7200 Beispiele. Für manche Attribute fehlen gelegentlich Werte, was mit zusätzlichen binären Eingabekoeffizienten angezeigt wird. Die Klassengrößen sind 5,1%, 92,6% bzw. 2,3%.

3.2.13 Building

Energieverbrauchsprognose. Aufgabe: Gegeben Datum und Tageszeit, sowie Außentemperatur, Außenluftfeuchte, Sonneneinstrahlung und Windgeschwindigkeit am Standort eines bestimmten Gebäudes, sage für dieses Gebäude den Verbrauch von kaltem Wasser, warmem Wasser und elektrischer Energie für die folgende Stunde voraus.

14 Eingaben, 3 Ausgaben, 4208 Beispiele. In seiner originalen Formulierung war diese Aufgabe eine Extrapolationsaufgabe: Gegeben stündliche Beispiele für vier Monate, sage die Ausgabedaten der zwei Folgemonate voraus. Der Datensatz building1 gibt diese Formulierung wieder, seine Daten sind in chronologischer Reihenfolge, so daß Trainings-, Validations- und Testmenge disjunkte Zeitabschnitte abdecken. Die Datensätze building2 und building3 sind hingegen zufällige Permutationen der Daten, so daß das Problem sich zu einer Interpolationsaufgabe vereinfacht.

3.2.14 Flare

Protuberanzvorhersage. Aufgabe: Gegeben Information über Sonnenprotuberanzen in der nahen Vergangenheit und über Typ und Historie einer aktiven Region der Sonnenoberfläche, sage für die folgende 24-Stunden-Periode die Anzahl von kleinen, mittleren und großen Protuberanzen vorher, die von dieser aktiven Region ausgehen werden.

24 Eingaben, 3 Ausgaben, 1066 Beispiele. 81% der Beispiele sind Null in allen drei Ausgabevariablen.

3.2.15 Hearta

Diagnose von Herzgefäßverengung. Dies ist die analoge (kontinuierliche) Version des heart Problems. Beschreibung siehe oben. In hearta hat die Anzahl verengter Gefäße für 44,7%, 28,8%, 11,8%, 11,6%, 3,0% der Beispiele den Wert 0, 1, 2, 3 bzw. 4. Für heartac lauten die entsprechenden Werte 54,1%, 18,2%, 11,9%, 11,6%, und 4,3%.

3.2.16 Übersicht

Die folgenden zwei Tabellen geben eine Kurzübersicht über das Format der oben vorgestellten Probleme. Die Tabelle 3.7 faßt die externen Eigenschaften der oben beschriebenen Klassifikationsprobleme

Tabelle 3.7: Übersicht der Klassifikationsprobleme

Problem	Problemattribute				Eingabekoeffizienten				Klassen	Beispiele
	b	k	n	Σ	b	k	f	Σ	b	
cancer	0	9	0	9	0	9	0	9	2	699
card	4	6	5	15	40	6	5	51	2	690
diabetes	0	8	0	8	0	8	0	8	2	768
gene	0	0	60	60	120	0	0	120	3	3175
glass	0	9	0	9	0	9	0	9	6	214
heart	1	6	6	13	18	6	11	35	2	920
heartc	1	6	6	13	18	6	11	35	2	303
horse	2	13	5	20	25	14	19	58	3	364
mushroom	0	0	22	22	125	0	0	125	2	8124
soybean	16	6	13	35	46	9	27	82	19	683
thyroid	9	6	0	21	9	6	6	21	3	7200

Die Klassifikationsprobleme mit zugehöriger Zahl von binären (b), kontinuierlichen (k) und nominalen (n) Attributen in der ursprünglichen Formulierung des Problems, mit Anzahl von binären (b) und kontinuierlichen (k) Eingabekoeffizienten und Anzahl von zusätzlichen Eingabekoeffizienten zur Darstellung des Fehlens von Attributwerten (f), mit Anzahl der Klassen und mit Anzahl der Beispiele.

zusammen. Sie unterscheidet zusätzlich zu den oben gemachten Angaben zwischen Eingaben mit nur zwei verschiedenen Werten (binäre Eingaben), Eingaben, die mehr als zwei verschiedene Werte annehmen können (kontinuierliche Eingaben) und Eingaben, die nur anzeigen, ob die korrekten Werte anderer Eingaben fehlen. Außerdem gibt die Tabelle die Anzahl von Attributen (im Unterschied zu Eingabekoeffizienten) des Problems an und unterscheidet diese in binäre Attribute, kontinuierliche Attribute (oder ordinale Attribute mit mehr als zwei Werten) und nominale Attribute mit mehr als zwei Werten.

Die Tabelle 3.8 enthält die entsprechenden Daten für die Approximationsprobleme.

3.2.17 Lernergebnisse mit linearen Netzen

Dieser Unterabschnitt präsentiert die Ergebnisse des Lernens der Proben1-Benchmark-Probleme mit linearen Netzen unter Anwendung der quadratischen Fehlerfunktion und der Methode des frühen Stoppens.

Ein lineares Netz hat keine verborgenen Knoten, sondern nur direkte Verbindungen von den Eingabeknoten zu den Ausgabeknoten, und benutzt die lineare Aktivierungsfunktion für die Ausgaben. Es werden also als Ausgaben nur Linearkombinationen der Eingaben berechnet. Ein solches Netz bis zum

Tabelle 3.8: Übersicht der Approximationsprobleme

Problem	Problemattribute				Eingabekoeffizienten				Ausgaben	Beispiele
	b	k	n	Σ	b	k	f	Σ	k	
building	0	6	0	6	8	6	0	14	3	4208
flare	5	2	3	10	22	2	0	24	3	1066
hearta	1	6	6	13	18	6	11	35	1	920
heartac	1	6	6	13	18	6	11	35	1	303

Die Tabelle ist analog zur Tabelle 3.7, jedoch mit der Anzahl kontinuierlicher Ausgabevariablen anstelle der Anzahl von Klassen.

Minimum des quadratischen Fehlers auf der Trainingsmenge zu trainieren ist gleichwertig mit der Berechnung einer unmodifizierten, mehrdimensionalen, multivariaten linearen Regression. Im Gegensatz dazu verwenden wir hier aber die Methode des frühen Stoppens mit dem Kriterium GL_5 wie in Abschnitt 4.2 eingeführt, um eine Überanpassung zu vermeiden, die auch schon bei linearen Netzen eintreten kann. Das Kriterium GL_5 bedeutet im wesentlichen, daß trainiert wird, bis der Fehler auf der Validationsmenge auf mehr als 5% über sein bisheriges Minimum ansteigt, und dann der Netzzustand mit dem bisher kleinsten Fehler auf der Validationsmenge als Ergebnis betrachtet wird.

Das Training fand mit dem RPROP-Verfahren [296] statt. Die verwendeten Parameter waren $\eta^+ = 1,2$, $\eta^- = 0,5$, $\Delta_0 \in 0,05 \ldots 0,2$ zufällig für jede Verbindung, $\Delta_{max} = 50$, $\Delta_{min} = 0$, anfängliches Gewicht aus dem Bereich $-0,1 \ldots 0,1$ zufällig für jede Verbindung. Diese Parameter werden in Abschnitt 4.3.1 genauer erläutert; die Ergebnisse sind recht robust im Bezug auf Änderungen dieser Werte. Für 14 der Lernprobleme wurden je 10 Läufe mit jedem der drei verschieden permutierten Datensätze gemacht. Für das mushroom-Problem nur ein Lauf (siehe unten).

Die Ergebnisse dieser Läufe sind in den Tabellen 3.9 (Klassifikationsprobleme) und 3.10 (Approximationsprobleme) dargestellt. Die Resultate dieser Läufe geben einen ersten Eindruck vom jeweiligen Schwierigkeitsgrad jedes Problems. Wir können ein paar interessante Beobachtungen machen:

1. Manche der Probleme sind sehr anfällig für Überanpassung, sogar mit einem einfachen linearen Netz (cancer2, glass2, heartac2, heartac3, heartc3 und alle horse). Dies legt nahe, daß die Benutzung einer Quervalidierungstechnik wie dem frühen Stoppen für die PROBEN1-Probleme sehr nützlich ist.
2. Bei manchen Problemen gibt es große Unterschiede in den Fehlern zwischen den drei Versionen des Datensatzes (cancer, card, flare, heart, heartac, heartc, horse). Dies illustriert, wie gefährlich es ist, Resultate zu vergleichen, für die die Aufteilung der Datenmenge in Trainings- und Testdaten nicht identisch war.
3. Manche der Probleme können recht gut bereits mit einem linearen Netz gelöst werden.
4. Das mushroom-Problem ist langweilig. Der eine hier gezeigte Lauf (mit mushroom1) hatte bereits nach 80 Epochen Klassifikationsfehler Null auf der Testmenge und nach 1550 Epochen auch auf der Validationsmenge. Dennoch stoppte das Training erst an der festgelegten Obergrenze von 3000 Epochen, weil die quadratischen Fehler fielen und fielen und fielen. Aufgrund dieser Resultate wird das mushroom-Problem in den übrigen Experimenten, die in dieser Arbeit berichtet werden, nicht weiter eingesetzt. Es kann dennoch ein interessantes Problem sein, um zum Beispiel das Skalierungsverhalten eines Lernverfahrens im Hinblick auf die Anzahl verfügbarer Trainingsbeispiele zu untersuchen.

Die extrem hohe Standardabweichung bei den Epochenzahlen für cancer2 und building3 rührt daher, daß dort mehrere der Läufe schon sehr früh beendet wurden. Dies ist bei linearen Netzen aus reinem Zufall durch entsprechende Initialisierungsbedingungen möglich, weil das Training von linearen Netzen

Tabelle 3.9: Lineare Lösbarkeit der Klassifikationsprobleme

Problem	Trainings-menge		Validations-menge		Test-menge		Testmengen-klassifikation		Über-anpassung		Epochen total		Epochen relevant	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
cancer1	4,25	0,00	2,91	0,01	3,52	0,04	2,93	0,18	0,55	0,59	129	13	104	31
cancer2	3,95	0,52	3,77	0,47	4,77	0,39	5,00	0,61	5,36	10,21	87	51	79	51
cancer3	3,30	0,00	4,23	0,04	4,11	0,03	5,17	0,00	0,35	0,64	115	18	92	29
card1	9,82	0,01	8,89	0,11	10,61	0,11	13,37	0,67	4,57	1,05	62	9	26	3
card2	8,24	0,01	10,80	0,16	14,91	0,55	19,24	0,43	4,22	1,08	65	10	23	5
card3	9,47	0,00	8,39	0,07	12,67	0,17	14,42	0,46	1,52	0,69	102	9	44	12
diabetes1	15,39	0,01	16,30	0,04	17,22	0,06	25,83	0,56	0,05	0,07	209	50	203	47
diabetes2	14,93	0,01	17,47	0,02	17,69	0,04	24,69	0,61	0,02	0,02	209	32	204	34
diabetes3	14,78	0,02	18,21	0,04	16,50	0,05	22,92	0,35	0,12	0,17	214	22	185	46
gene1	8,42	0,00	9,58	0,01	9,92	0,01	13,64	0,10	0,03	0,07	47	6	43	10
gene2	8,39	0,00	9,90	0,00	9,51	0,00	12,30	0,14	0,02	0,03	46	4	40	6
gene3	8,21	0,00	9,36	0,01	10,61	0,01	15,41	0,13	0,03	0,06	42	4	39	6
glass1	8,83	0,01	9,70	0,04	9,98	0,10	46,04	2,21	3,81	0,42	129	13	23	5
glass2	8,71	0,09	10,28	0,19	10,34	0,15	55,28	1,27	5,74	0,67	34	6	14	2
glass3	8,71	0,02	9,37	0,06	11,07	0,15	60,57	3,82	1,76	0,57	135	30	27	11
heart1	11,19	0,01	13,28	0,06	14,29	0,05	20,65	0,31	1,14	0,45	134	15	41	5
heart2	11,66	0,01	12,22	0,02	13,52	0,06	16,43	0,40	0,13	0,09	184	14	146	48
heart3	11,11	0,01	10,77	0,02	16,39	0,18	22,65	0,69	0,14	0,23	142	15	113	53
heartc1	10,17	0,01	9,65	0,03	16,12	0,04	19,73	0,56	0,15	0,11	128	10	114	23
heartc2	11,23	0,03	16,51	0,08	6,34	0,25	3,20	1,56	3,98	0,56	136	22	25	10
heartc3	10,48	0,31	13,88	0,33	12,53	0,44	14,27	1,67	6,23	1,15	26	9	12	3
horse1	11,31	0,16	15,53	0,29	12,93	0,38	26,70	1,87	6,22	0,57	27	7	9	2
horse2	8,62	0,28	15,99	0,21	17,43	0,45	34,84	1,38	5,54	0,47	42	16	13	3
horse3	10,43	0,27	15,59	0,30	15,50	0,45	32,42	2,65	6,34	1,07	26	6	8	3
mushroom	0,014	—	0,014	—	0,011	—	0,00	—	0,00	—	3000	—	3000	—
soybean1	0,65	0,00	0,98	0,00	1,16	0,00	9,47	0,51	0,28	0,18	553	11	418	41
soybean2	0,80	0,00	0,81	0,00	1,05	0,00	4,24	0,25	0,02	0,02	509	19	504	18
soybean3	0,78	0,00	0,96	0,00	1,03	0,00	7,00	0,19	0,03	0,04	533	27	522	28
thyroid1	3,76	0,00	3,78	0,01	3,84	0,01	6,56	0,00	0,01	0,03	104	16	99	22
thyroid2	3,93	0,00	3,55	0,01	3,71	0,01	6,56	0,00	0,01	0,02	98	16	96	16
thyroid3	3,85	0,00	3,39	0,00	4,02	0,00	7,23	0,02	0,02	0,02	114	22	109	21

Trainingsmenge: Mittelwert (μ) und Standardabweichung (σ) des minimalen während jedes Laufes auf der Validationsmenge erreichten quadratischen Fehlers (in Prozent).
Validationsmenge: dito, für Validationsmenge.
Testmenge: Mittelwert und Standardabweichung des quadratischen Fehlers auf der Testmenge, gemessen im Zustand des Netzes mit minimalem Fehler auf der Validationsmenge.
Testmengenklassifikation: Mittelwert und Standardabweichung des zugehörigen Klassifikationsfehlers.
Überanpassung: Mittelwert und Standardabweichung des GL-Werts am Ende des Trainings.
Epochen total: Mittelwert und Standardabweichung der Anzahl tatsächlich durchgeführter Epochen.
Epochen relevant: Mittelwert und Standardabweichung der Anzahl Epochen bis zum Erreichen des minimalen Validationsfehlers.

Tabelle 3.10: Lineare Lösbarkeit der Approximationsprobleme

Problem	Trainings-menge		Validations-menge		Test-menge		Überanpassung		Epochen total		Epochen relevant	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
building1	0,21	0,01	0,92	0,06	0,78	0,02	2,15	4,64	407	138	401	142
building2	0,34	0,00	0,37	0,00	0,35	0,00	0,00	0,01	298	23	297	23
building3	0,37	0,04	0,38	0,07	0,38	0,08	1,99	4,45	229	107	217	102
flare1	0,37	0,00	0,34	0,01	0,52	0,01	2,17	1,61	41	5	12	4
flare2	0,42	0,00	0,46	0,00	0,31	0,02	0,72	0,90	37	3	16	10
flare3	0,39	0,00	0,46	0,00	0,35	0,00	0,57	0,73	35	6	18	12
hearta1	3,82	0,00	4,42	0,03	4,47	0,06	1,68	0,68	118	12	27	10
hearta2	4,17	0,00	4,28	0,02	4,19	0,01	0,06	0,13	112	10	107	15
hearta3	4,06	0,00	4,14	0,02	4,54	0,01	0,05	0,05	116	8	110	10
heartac1	4,05	0,00	4,70	0,02	2,69	0,02	0,01	0,02	98	10	96	11
heartac2	3,37	0,11	5,21	0,21	3,87	0,16	6,99	2,27	19	4	13	4
heartac3	2,85	0,09	5,66	0,16	5,43	0,23	6,06	0,99	29	9	14	3

(Es gilt die gleiche Erklärung wie für Tabelle 3.9, nur daß hier kein Klassifikationsfehler angegeben ist.)

im Prinzip sehr einfach ist: die Fehleroberfläche ist eine quadratische Fläche ohne lokale Minima. Bei Netzen mit verborgenen Knoten tritt der Effekt in der Regel nicht auf.

3.3 Die Proben1 Benchmark-Regeln

Zusätzlich zu den Beschreibungen der Benchmark-Probleme selbst enthält der PROBEN1-Report [14] eine Reihe von Regeln und Konventionen für die Durchführung und Publikation von Benchmarks mit neuronalen Lernverfahren. Entsprechend der Auswahl der Datensätze sind diese Regeln auf überwachte Lernverfahren für vorwärtsgerichtete Netze zugeschnitten.

Ich verzichte hier auf eine detaillierte Darstellung der Regeln und gebe nur kurz die ihnen zugrundeliegende Idee an. Die Regeln verfolgen drei Ziele für die Auswertung von Lernverfahren:

Gültigkeit: Manche publizierten Auswertungen enthalten methodische Fehler, die die publizierten Daten schlechthin ungültig machen. Diese Fehler beruhen meist auf einer unzulässigen Verwendung der Testmenge zur Modellauswahl, zum Beispiel bei der Suche nach guten Werten für die vom Benutzer zu wählenden Parameter oder bei der Auswahl des besten Netzes aus einer Reihe von Wiederholungen von Lernverfahrensläufen. Die PROBEN1-Regeln beschreiben dieses Problem und geben falsche und korrekte Verfahrensweisen an.

Reproduzierbarkeit: Sehr häufig ist ein Lernverfahren in heutigen Publikationen nicht so genau spezifiziert, daß die damit durchgeführten Versuche sich exakt reproduzieren ließen. Während dies im Hinblick auf die Zufallsinitialisierung der Netze akzeptabel ist (hier muß nur die Verteilung der Zufallszahlen angegeben werden), kann eine ungenaue oder fehlende Angabe von benutzerwählbaren Parametern, insbesondere Phasenübergangs- und Stoppkriterien, nicht hingenommen werden; Ergebnisse, die sich nicht reproduzieren lassen, sind keine wissenschaftlichen Resultate. Die Regeln geben deshalb eine Liste von Angaben, die in einer Publikation gemacht werden müssen, um Reproduzierbarkeit zu gewährleisten und schlagen teilweise konkrete Formulierungen dafür vor, um Mehrdeutigkeiten zu verringern.

Vergleichbarkeit: Es wäre offensichtlich nützlich, wenn man die Ergebnisse verschiedener Autoren direkt miteinander vergleichen könnte. Leider ist solche Vergleichbarkeit heute nur sehr selten gegeben, da nur in wenigen Fällen gleiche Versuchsaufbauten zugrundegelegt werden. Die PROBEN1-Benchmarkprobleme stellen, wenn sie in Zukunft genügend häufig benutzt werden, in dieser Hinsicht bereits eine erhebliche Verbesserung dar. Die PROBEN1-Regeln ergänzen die Vereinheitlichung noch dadurch, daß sie für einige Aspekte von Versuchsaufbauten (z.B. Skalierung der Fehlerfunktion und Anzahl von Wiederholungsläufen) Standardwerte vorschlagen, deren Einhaltung die Vielfalt von Versuchsaufbauten reduzieren würde, ohne dabei die Flexibilität der Experimentatoren sehr zu beschränken.

Im Hinblick auf diese Ziele wird in den Regeln in je einem eigenen Abschnitt die Behandlung und Publikation folgender Aspekte von Lernverfahren diskutiert: Definition und öffentliche Verfügbarkeit des Lernproblems; Festlegung und Benutzung von Trainings-, Validations- und Testmenge; Topologie und Aktivierungsfunktionen des Netzes; Lernverfahren, einschließlich benutzerwählbarer Parameter und ggf. deren Anpassungsregeln, Netzinitialisierung, Neustart-, Phasenübergangs- und Stoppkriterien; Fehlerfunktion, einschließlich ihrer Skalierung im Lernverfahren und der Normalisierung und Präsentation der Gesamtfehlerwerte; Wiederholungen von Läufen und die Berechnung der Gesamtergebnisse aus den Einzelergebnissen mehrere Läufe.

3.4 Die Normalverteilungsannahme

Für die Vergleiche zwischen verschiedenen Methoden sollen parametrische statistische Methoden eingesetzt werden, um zu prüfen, ob beobachtete Unterschiede statistisch signifikant sind. Da diese Methoden eine Normalverteilung der betrachteten Daten zugrundelegen, müssen wir zur Absicherung des Vorgehens diese Normalverteilungsannahme prüfen. Eine solche Prüfung wird in diesem Abschnitt beschrieben. Die Größe, für die uns die Annahme in erster Linie interessiert, ist der Fehler auf der Testmenge nach Beendigung des Trainings, weil diese Größe die Qualität des erzeugten Netzes beschreibt und daher die übliche Zielgröße beim Vergleich von Verfahren ist.

Der zentrale Grenzwertsatz der Wahrscheinlichkeitsrechnung besagt, daß jede Größe x, die sich als Summe aus n Größen x_i ergibt, näherungsweise normalverteilt ist und zwar um so besser, je größer n ist. Dies gilt auch dann, wenn die x_i lauter unterschiedliche Verteilungen haben (insbesondere keine Normalverteilung), sofern sie etwa gleich groß sind und es gilt auch dann, wenn die x_i nicht stochastisch unabhängig sind, sofern ihre Abhängigkeit nicht zu stark ist.

Diese Situation liegt beim Testfehler eines neuronalen Netzes offenbar vor. Der Fehler ist die Summe der Fehler auf den einzelnen Beispielen, die überwiegend die gleiche Größenordnung haben. Jeder einzelne Fehler wird zudem vom Zusammenwirken einer Vielzahl von Gewichten hervorgerufen. Zwischen den einzelnen Fehlern besteht ein mäßiger Zusammenhang dadurch, daß sie zwar alle vom gleichen Netz, aber aufgrund verschiedener (wiederum nur mäßig zusammenhängender) Beispiele produziert wurden. Die Gewichte hängen durch den Trainingsprozeß zusammen, sind aber bei der Initialisierung zunächst vollkommen unabhängig. Aus diesen Gründen können wir erwarten, beim Testfehler annähernd eine Normalverteilung vorzufinden und zwar weitgehend unabhängig vom Datensatz und vom Lernverfahren.

Ich habe diese Annahme an zwei Beispielen geprüft: Erstens dem glass1 Datensatz mit einem Netz mit einer verborgenen Schicht mit 8 Knoten und zweitens den cancer2 Datensatz mit seiner Pivot-Architektur (siehe Abschnitt 4.4.1) von 8+4 Knoten in zwei verborgenen Schichten. Beide Netze sind im Vergleich zu den ansonsten in den Experimenten benutzten Netzen recht klein (siehe die Tabelle 3.7 der Problemgrößen auf Seite 62 und die Tabelle 4.7 der Pivot-Architekturen auf Seite 82). Der glass1 Datensatz hat zudem nur sehr wenige Beispiele. Beides verschlechtert die Chancen, eine gute Annäherung an eine Normalverteilung vorzufinden. Der cancer2 Datensatz ist außerdem offenbar besonders bösartig in den Zufallsschwankungen der mit ihm erzielten Ergebnisse, was sich daran erkennen läßt, daß der zweitbeste Lauf im Stoppkriterien-Experiment von Kapitel 4 einen mehr als 10% höheren Fehler hatte als der beste. Dies geschah sonst nur noch bei 3 weiteren der 45 Datensätze. Diese Eigenschaften legen nahe, daß die Untersuchung dieser beiden Probleme vermutlich eine Untersuchung des schlechtesten Falls darstellt.

Um die Verteilung der Testfehler genau mit einer Normalverteilung vergleichen zu können, wurden je 500 Läufe mit jedem der beiden Probleme gemacht. Dabei wurde trainiert wie in Abschnitt 3.2.17 angegeben.

Die Abbildungen 3.11 und 3.12 zeigen die Ergebnisse dieser Läufe. Im linken Teil der Abbildung befin-

```
Testfehler         Histogramm                  Anzahl                    Normalquantil-Plot
6,1+**                                             2 +                                                  *
   .*                                              1 |                                                  *
   .*                                              1 |                                                 *
   .                                                 |
   .*                                              3 |                                              ***
   .                                                 |
   .*                                              3 |                                             **
   .*                                              1 |                                            *
   .*                                              2 |                                            *   ++
   .***                                           10 |                                         ******+
   .**                                             6 |                                      ++**+
   .********                                      30 |                                  +++++++
   .**********                                    40 |                             ++++****
   .*********************                         83 |                        +++++****
   .******************************************   168 |                   *********
   .*****************************                113 |          **********
   .*********                                     35 |***************+
2,7+**                                             2 +*      ++++
    ----+----+----+----+----+----+----+----+--        +----+----+----+----+----+----+----+----+----+----+
    * bedeutet bis zu 4 Laeufe                            -2        -1         0        +1        +2
```

Abbildung 3.11: Histogramm und Normalquantil-Plot der Testfehler aus 500 Läufen mit cancer2.

det sich jeweils ein Histogramm der aufgetretenen Fehlerwerte, wobei die Klassen von unten nach oben aufeinanderfolgen und die Klassengröße als Balken aus Sternchen von links nach rechts aufgetragen ist. Dieses Histogramm sollte im Idealfall die vertraute Glockenkurvenform der Normalverteilung zeigen. Im Abbildung 3.11 erkennen wir deutliche Abweichungen von dieser Form: der lange Schwanz bei kleinen Fehlerwerten fehlt, statt dessen gibt es beim Schwanz der großen Fehlerwerte einige unerwartet große.

Die Abweichungen von der Normalverteilung kann man noch besser erkennen, wenn man einen Normalquantil-Plot betrachtet, wie im rechten Teil der Abbildung angegeben. Darin wird die Anzahl von Werten der Größe bis zu y gegen diejenige Anzahl solcher Werte aufgetragen, die man gemäß Mittelwert und Standardabweichung der Stichprobe erwarten müßte, wenn die Stichprobe einer Normalverteilung entstammt. Ein Normalquantil-Plot einer ideal normalverteilten Stichprobe ist also eine gerade Linie; Abweichungen von einer Geraden zeigen Abweichungen von der Normalverteilung an. Ein Band geringer Abweichungen ist durch Pluszeichen eingetragen, die vorgefundenen Werte durch Sternchen. Dies ist das Werkzeug, das Statistiker verwenden, um die Normalverteilung einer Stichprobe zu prüfen — statistische Tests können nur verwendet werden, um die Normalverteiltheit zu widerlegen, nicht um sie zu bestätigen. Wie wir sehen, stellen sowohl der fehlende Schwanz kleiner Werte als auch die Ausreißer nach oben erhebliche Abweichungen von der Normalverteilung dar. Die Fehler von cancer2 in diesem Versuchsaufbau als normalverteilt zu betrachten, kann die Aussagen statistischer Inferenzen ungültig machen.

Etwas günstiger ist die Situation bei glass1, wie in Abbildung 3.12 dargestellt. Hier sind die Abweichungen von der Normalverteilung akzeptabel bis auf die beiden Ausreißer am oberen Ende. Allerdings ist auch hier der untere Schwanz der Verteilung etwas schwach ausgeprägt.

Was sollen wir nun tun, um mit parametrischen statistischen Prozeduren gesicherte Ergebnisse erhal-

```
Testfehler          Histogramm                        Anzahl                     Normalquantil-Plot
13,5+*                                                 1 +                                                  *
    .                                                    |
    .                                                    |
    .*                                                 1 |                                                  *
    .                                                    |
    .                                                    |
    .                                                    |
    .                                                    |
    .                                                    |
    .                                                    |
    .**                                                3 |                                                **
    .*                                                 1 |                                               *
    .*                                                 1 |                                              *  +
10,9+*                                                 2 +                                             **++
    .****                                              8 |                                          ****
    .*******                                          14 |                                      +****
    .***********                                      21 |                                  ++****
    .*****************************                    57 |                               +*****
    .***************************************          77 |                            ******
    .************************************************ 95 |                       ******
    .*******************************************      85 |                   *****
    .**************************                       52 |               *****+
    .*****************                                34 |            ****+
    .**********                                       19 |         ****
    .***********                                      22 |   *******
    .**                                                4 | ****+
 8,3+**                                                3 +*+
    ----+----+----+----+----+----+----+----+----+---     +----+----+----+----+----+----+----+----+----+----+
    * bedeutet bis zu 2 Laeufe                               -2        -1         0        +1        +2
```

Abbildung 3.12: Histogramm und Normalquantil-Plot der Testfehler aus 500 Läufen mit glass1.

ten zu können? Es ergibt sich eine bessere Annäherung an die Normalverteilung, wenn wir anstatt der Fehlerwerte den Logarithmus derselben betrachten; Verteilungen mit dieser Eigenschaft werden *log-normal* genannt und kommen häufig vor, z.B. bei Körpergrößen, Pulsfrequenzen, Monatseinkommen und vielen anderen Größen, die eine untere Schranke haben [309, Seite 173]. Das Logarithmieren staucht den langen oberen Schwanz zusammen und zieht zugleich den unteren Bereich weiter auseinander, so daß dort ein etwas längerer Schwanz entsteht. Wenn wir prüfen wollen, welches Verfahren den kleineren Mittelwert produziert (sprich: besser ist), so verändert die Logarithmierung das Ergebnis nicht, da der Logarithmus streng monoton steigend ist. Deshalb wird diese Technik von Statistikern häufig eingesetzt [255, Seiten 512f]. Die Ausreißer nach oben werden jedoch auch hiermit nicht beseitigt. Wie sollten wir mit ihnen umgehen? Allgemein gibt es in der Statistik drei Herangehensweisen zur Behandlung von Ausreißern, zwischen denen mit gesundem Urteilsvermögen ausgewählt werden muß:

1. Man kann die Ausreißer in der Stichprobe belassen. Dies ist für robuste Statistiken wie den Median akzeptabel. Bei nichtrobusten Statistiken wie dem Mittelwert stellen sich dadurch jedoch massive und irreführende Verzerrungen des Ergebnisses ein. Deshalb ist in unserem Fall, in dem Mittelwerte betrachtet werden sollen, nicht empfehlenswert.
2. Man kann die Stichprobe *trimmen* oder *winsorieren*. Dabei werden die k größten und die k kleinsten Werte jeweils gemeinsam entfernt (beim trimmen) bzw. auf die maximalen oder minimalen akzeptablen Randwerte geändert (beim winsorieren) [309, Seiten 365 und 371]. Dieses Vorgehen ist empfehlenswert, wenn sich nach einer solchen Beschneidung eine gute Annäherung an die erwartete Verteilung ergibt. Bei den später betrachteten kleineren Stichproben führt aber in der Regel ein Wegnehmen von Werten an dem Ende, das den Ausreißern entgegengesetzt ist, zu einer ungewollten Verkürzung der Verteilung an diesem Ende. Die Symmetrie wird dabei also verletzt, weil die Ausreißer meist sozusagen „zusätzliche“ Werte darstellen und der Rest der Stichprobe die erwartete Verteilung am besten annähert, wenn nur die Ausreißer entfernt werden.
3. Man kann die Ausreißer (und nur die Ausreißer) entfernen. Dieses Vorgehen ist das *einzig* richtige, wenn die Ausreißer eindeutig auf Meßfehlern oder ähnlichem Versagen in der Daten*sammlung*

beruhen [309, Seite 364]. Dies ist im vorliegenden Fall allerdings nicht gegeben. Ein Entfernen ist jedoch auch zu rechtfertigen, wenn die Ausreißer als seltene pathologische Fälle einzustufen sind, von denen angenommen werden kann, daß sie einer anderen Grundgesamtheit entstammen. In unserem Fall werden solche Fälle durch pathologische Zufallsinitialisierung des Netzes verursacht. Bilden dann die übrigen Werte eine gut normalverteilte Stichprobe, so sollten die Ausreißer entfernt werden; bei der Interpretation der Ergebnisse ist dann Vorhandensein und Häufigkeit von Ausreißern separat zu berücksichtigen [309, Seite 365]. Dies ist im vorliegenden Fall die angemessene Methode. In der Beurteilung danach ist ein Verfahren, das nur wenige Ausreißer (mit hohen Fehlerwerten) produziert, als etwas robuster anzusehen, als ein anderes, bei dem solche Ausreißer häufiger auftreten.

Die Ergebnisse nach Logarithmierung und Wegnahme der Ausreißer sind in den Abbildungen 3.13 und 3.14 dargestellt. Für cancer2 sind auch nach Logarithmierung noch deutliche Abweichungen von

```
log(Testfehler)        Histogramm           Anzahl                       Normalquantil-Plot
1,475+**                                        7 +                                               ****
     .***                                      10 |                                           *******+
     .*****                                    19 |                                        *******+
     .*********                                36 |                                    *****+
     .**************                           54 |                              +*****
1,225+**************************              104 +                         +******
     .*************************************   145 |                  *********
     .*******************                      75 |           ********
     .********                                 31 |   *********+
     .**                                        5 |******+
0,975+*                                         2 +**
      ----+----+----+----+----+----+----+--        +----+----+----+----+----+----+----+----+----+----+
      * bedeutet bis zu 4 Laeufe                       -2        -1         0        +1        +2
```

Abbildung 3.13: Histogramm und Normalquantil-Plot für log(Testfehler) aus 488 Läufen mit cancer2 (nach Entfernen von 12 Ausreißern).

der Normalverteilung zu beobachten. Wenn man, wie im Abbildung 3.13 dargestellt, die 12 Ausreißer oben entfernt (2,4% aller Läufe), so wird die Annäherung jedoch gut. Für glass1 entsteht eine sehr

```
log(Testfehler)        Histogramm                Anzahl                       Normalquantil-Plot
2,45+*                                               2 +                                                  *
    .*                                               2 |                                                 **
    .*                                               1 |                                                 *
    .*                                               2 |                                               ***+
    .****                                            8 |                                            ****
    .*******                                        14 |                                        +****
    .**************                                 28 |                                    +*****
    .******************************                 60 |                                ******
2,29+***************************************        77 +                            *****
    .*********************************************  90 |                       ******
    .******************************************     84 |                   *****
    .**********************                         44 |                ****+
    .*****************                              33 |             ****
    .*********                                      17 |         +***
    .***********                                    21 |     +*****
    .*****                                          10 | +*****
2,13+***                                             5 +**
     ----+----+----+----+----+----+----+----+----+      +----+----+----+----+----+----+----+----+----+----+
     * bedeutet bis zu 2 Laeufe                             -2        -1         0        +1        +2
```

Abbildung 3.14: Histogramm und Normalquantil-Plot für log(Testfehler) aus 498 Läufen mit glass1 (nach Entfernung von 2 Ausreißern).

gute Anpassung schon nach Wegnahme von nur 2 Ausreißern (0,4% der Läufe).

Die Entfernung der Ausreißer ist deshalb wichtig, weil der Mittelwert gegen Ausreißer nicht robust ist und deshalb eine durch Mittelwert und Standardabweichung aus Daten mit Ausreißern geschätzte Verteilung falsch positioniert würde.

Eine analoge Prüfung wurde auch für einige andere Kenngrößen der Trainingsergebnisse durchgeführt. Der minimal erreichte Validationsfehler ist in guter Näherung normalverteilt, wobei im Gegensatz zum Testfehler eher der obere Schwanz fehlt. Für den Validationsfehler wäre eine log-Transformation also verkehrt. Bei cancer2 gibt es zudem 7 sehr weite Ausreißer nach oben (1,4%). Der Trainingsfehler ist für cancer2 nach Entfernen von 7 Ausreißern gut normalverteilt, für glass1 ist die Verteilung zweigipflig mit einem Nebenmaximum im oberen Bereich, aber noch als normalverteilt akzeptabel. Dieses Nebenmaximum findet sich im Klassifikationsfehler auf der Testmenge von glass1 wieder, wo es jedoch als langer oberer Schwanz interpretiert werden kann, so daß der log-transformierte Klassifikationsfehler wieder einigermaßen normalverteilt ist. Bei cancer2 hat der Klassifikationsfehler nach Wegnahme der Ausreißer nur 7 verschiedene Stufen, kann aber dennoch als log-normalverteilt angesehen werden. Die Anzahl trainierter Epochen ist ungefähr log-normal verteilt, wobei bei cancer2 vier Ausreißer nach oben und sechs nach unten auftreten und bei glass1 drei Ausreißer nach unten. Wie wir sehen, ist die Situation bei den verschiedenen Kenngrößen also keineswegs einheitlich.

Um zu zeigen, welche Ergebnisse eine unbedachte Anwendung statistischer Inferenz haben kann, betrachten wir als Beispiel den Vergleich der beiden Verfahren aus Kapitel 6, nennen wir sie A und L. Es wurden für 42 verschiedene Datensätze (14 Lernprobleme in je 3 Varianten) jeweils 30 Läufe mit beiden Verfahren durchgeführt und die entstehenden Verteilungen von Testfehlern in 42 Anwendungen eines t-Tests auf signifikante Unterschiede im Mittelwert geprüft. Der t-Test prüft die Hypothese, daß zwei vorgelegte Stichproben denselben Mittelwert haben; er macht dabei die Voraussetzungen, daß die Grundgesamtheiten, aus denen die Stichproben entnommen wurden, normalverteilt sind und daß beide Normalverteilungen dieselbe Varianz aufweisen. Bei den meisten Anwendungen des t-Tests in der Literatur über Lernverfahren wird der t-Test ohne jegliche Korrektur direkt auf die Fehlerwerte angewendet. Korrekterweise müßte jedoch zunächst logarithmiert werden, dann einige Ausreißer entfernt (in diesem Fall 39 von 2520 Werten, also 1,5%) und schließlich bei der Berechnung der t-Test–Statistik eine Korrektur vorgenommen (die Cochran/Cox–Approximation), weil die Varianzen der Fehlerverteilungen für A und L oft sehr verschieden sind, der t-Test aber gleiche Varianzen annimmt. In der Tabelle 3.15 sind links die Ergebnisse mit allen drei Korrekturen angegeben, rechts die Ergebnisse ohne Korrektur. Auf dem hier zur Veranschaulichung verwendeten Signifikanzniveau von 0,02, d.h. bei 98% Konfidenz, ändern sich bei 9 signifikanten Unterschieden 3 Ergebnisse, wenn man die Korrektur entfallen läßt. Ein mit hoher Konfidenz signifikanter Unterschied bei diabetes2 verschwindet und zwei neue bei glass2 und bei heartc3 tauchen auf; diese Änderungen sind in der Tabelle fett dargestellt. In diesem zugegebenermaßen schweren Fall sind also ein Drittel der erzielten Ergebnisse falsch, wenn man die oben angegebenen (und oft gemachten) methodischen Fehler nicht vermeidet!

Als Konsequenz verwende ich in Zukunft folgende Vorgehensweise zur Vorbereitung von Testfehlerwerten für die statistische Inferenz:

1. Logarithmiere alle Testfehlerwerte und betrachte nur noch die Logarithmen. Validationsfehlerwerte werden ohne Logarithmierung benutzt.
2. Prüfe mit Hilfe eines Normalquantil-Plots die Normalverteilungsannahme; identifiziere dabei eventuelle Ausreißer und entferne sie aus der Stichprobe. Entferne jedoch höchstens 10% aller Werte.
3. Nimm von den restlichen Daten an, daß sie normalverteilt sind und markiere alle Resultate, für die diese Annahme unzulässig ist..
4. Prüfe den Unterschied in der Varianz der zu vergleichenden Stichproben. Wenn er zu groß ist, wende die Cochran/Cox–Approximation zur Korrektur der t-Test-Statistik an. Wenn er nicht zu groß ist, wende den normalen t-Test oder ANOVA (Analysis of Variance) an.

Die in diesem Abschnitt vorgenommene Betrachtung zeigt, daß selbst solche Arbeiten, die zur Prüfung von Aussagen über neuronale Lernverfahren statistische Methoden heranziehen (was selten geschieht), ungültige Ergebnisse produzieren können, wenn die nötigen Voraussetzungen für das statistische Schließen nicht sichergestellt werden (was häufig versäumt wird).

Tabelle 3.15: Ergebnisse von t-Tests mit und ohne Korrektur

Problem	korrigiert			unkorrigiert		
building	L 0,0	—	—	L 0,0	—	—
cancer	—	—	—	—	—	—
card	—	—	—	—	—	—
diabetes	—	—	**L 0,9**	—	—	—
flare	—	—	—	—	—	—
gene	A 0,0	A 0,0	A 0,0	A 0,0	A 0,0	A 0,0
glass	—	—	L 1,8	—	**L 1,1**	L 1,7
heart	—	A 0,4	—	—	A 0,8	—
hearta	—	A 0,1	—	—	A 0,2	—
heartac	—	—	—	—	—	—
heartc	—	—	—	—	—	**A 1,9**
horse	—	—	—	—	—	—
soybean	—	—	—	—	—	—
thyroid	—	—	L 0,3	—	—	L 0,2

Spalte 2 bis 4: Ergebnisse t-Test mit Korrekturen für Versionen 1 bis 3 der Probleme.
Spalte 5 bis 7: Dito, ohne die nötigen Korrekturen.
Ein Querstrich bedeutet „kein signifikanter Unterschied zwischen A und L auf Niveau 2%“, andere Einträge geben an, welches Verfahren signifikant besser ist und den zugehörigen P-Wert (Signifikanzniveau) in Prozent.

3.5 Zusammenfassung und Beiträge dieser Arbeit

Wie wir gesehen haben, hat die heutige Forschungspraxis auf dem Gebiet der Lernverfahren für neuronale Netze schwere Mängel, was die Aussagekraft der für neu vorgeschlagene Verfahren vorgenommenen experimentellen Auswertung anbelangt. Ein Beitrag dieser Arbeit besteht darin, diese Mängel konkretisiert und quantitativ nachgewiesen zu haben. Um dem mangelhaften Zustand zu entkommen, brauchen wir standardisierte Sammlungen von Benchmark-Problemen, die dem einzelnen Forscher die Arbeit erleichtern und darüberhinaus die Vergleichbarkeit von Ergebnissen ermöglichen.

PROBEN1 ist ein erster Schritt in diese Richtung und ein weiterer Beitrag dieser Arbeit. Für eine als Diagnoseaufgaben bezeichnete Klasse von Lernproblemen stellt diese Sammlung 45 Datensätze zu 15 verschiedenen Lernproblemen zur Verfügung. Für deren Benutzung werden Konventionen vorgegeben, die erreichen sollen, daß publizierte Resultate stets gültig (d.h. ohne methodische Fehler produziert) und reproduzierbar sind und möglichst oft den direkten Vergleich mit Resultaten anderer Forscher erlauben.

Ein dritter Beitrag dieser Arbeit liegt in der Untersuchung der Normalverteilungsannahme für die Anwendung parametrischer statistischer Tests zum Vergleich von Lernverfahren und der Beschreibung einer Methode zur korrekten Aufbereitung der Daten für solche Tests. Ich habe gezeigt, daß die gemessenen Daten einer genaueren Untersuchung bedürfen, weil eine kritiklose Anwendung der Normalverteilungsannahme im Falle neuronaler Lernverfahren zu erheblichen Verfälschungen der Ergebnisse führen kann — in einer als Beispiel betrachteten Situation werden ein Drittel aller Ergebnisse falsch!

Natürlich ist damit das Problem der Auswertung von Lernverfahren noch lange nicht gelöst. Mindestens zwei wichtige Schwierigkeiten verbleiben. Erstens nützt eine Sammlung von Problemen und Regeln wie PROBEN1 nur dann etwas, wenn sie auch tatsächlich eingesetzt wird. Darauf hinzuwirken ist Sache der einzelnen Forscher, sowie Sache von Gutachtern und Editoren. Zweitens nutzt sich eine Benchmark-Sammlung allmählich ab: Je länger sie benutzt wird, desto mehr passen sich die gemessenen Systeme unmerklich an die speziellen Eigenheiten der Benchmarks an. Deshalb müssen weitere

Benchmark-Sammlungen erstellt werden, die Proben1 ergänzen und schließlich ablösen. Die Ziffer 1 am Ende des Namens ist in diesem Sinne als hoffnungsvoller Vorschlag zu verstehen.

Kapitel 4

Automatisches Lernen I: Frühes Stoppen

The reasonable man adapts himself to the world;
the unreasonable man persists in trying
to adapt the world to himself.
Therefore all progress depends on the unreasonable man.
George Bernard Shaw

If we do not succeed
then we face the risk of failure.
Danforth Quayle, ehemaliger Vize-Präsident der USA

In diesem Kapitel untersuche ich empirisch die Effizienz und Effektivität verschiedener Stoppkriterien für die Methode des frühen Stoppens. Dazu wird zunächst dargelegt, warum dies überhaupt eine interessante Frage ist und was man bisher darüber weiß (Abschnitt 4.1). Dann führe ich drei Klassen von Stoppkriterien ein, die in dieser Arbeit untersucht werden (Abschnitt 4.2), und beschreibe den Versuchsaufbau und die Ergebnisse der Untersuchung (Abschnitt 4.3). Außerdem gebe ich als Basis für einen Vergleich mit konstruktiven Lernverfahren Ergebnisse des Lernens mit der Methode des frühen Stoppens an (Abschnitt 4.4). Der Hauptbeitrag dieses Kapitels besteht in der Quantifikation der Verbesserungen, die sich durch längeres Training bei der Methode des frühen Stoppens erzielen lassen.

4.1 Einführung und verwandte Arbeiten

Wie oben bereits erwähnt, ist die Methode des frühen Stoppens eine der wenigen echten Innovationen, die die Neuroinformatik hervorgebracht hat. Die Erfindung geschah rein heuristisch aus der Beobachtung heraus, daß bei Fortschreiten des Trainings der Fehler auf der Testmenge zunächst wie erwartet absank, irgendwann aber wieder größer wurde. Dies ist nichts weiter als das aus der Statistik seit langem bekannte Phänomen der Überanpassung eines Modells an eine Datenmenge, das immer dann auftritt, wenn ein Modell eine zu hohe Varianz (sprich: Flexibilität) aufweist. Der neue Gedanke beim frühen Stoppen besteht darin, die iterative Anpassung des Modells an die Daten beim Beginn der Überanpassung vorzeitig abzubrechen. In der Statistik war ein solches Vorgehen bislang undenkbar, weil fast alle theoretischen Aussagen über Eigenschaften der Modelle nur galten, wenn sie optimal an die Daten angepaßt waren. Dort wurde deshalb immer die Varianz des Modells explizit verringert und eine Anpassung des geänderten Modells vorgenommen. Das frühe Stoppen bedeutet demgegenüber, jeden

Zwischenzustand, den das Netz während des Trainings einnimmt als ein mögliches (fertig angepaßtes) Modell zu betrachten und eines von diesen auszuwählen.

Die algorithmische Vorschrift für dieses Vorgehen lautet etwa folgendermaßen:

> **Frühes Stoppen**: Trainiere das Netz auf einer Trainingsmenge. Überwache während des Trainings den Generalisierungsfehler anhand einer separaten Validationsmenge. Wenn das Netz in die Überanpassung eingetreten ist (Wiederansteigen des Generalisierungsfehlers), breche das Training ab und betrachte denjenigen Zustand des Netzes als Ergebnis, in dem das Netz den geringsten Generalisierungsfehler hatte.

Die ursprüngliche Annahme war, daß die Kurve des Generalisierungsfehlers über die Zeit eine konvexe Form hat, wie in Abbildung 4.1 gezeigt. Inzwischen hat sich aber gezeigt, daß das wirkliche zeitliche

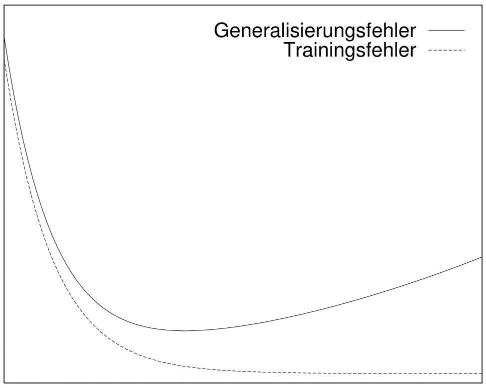


Abbildung 4.1: Idealisierte Vorstellung des zeitlichen Verlaufs von Fehler auf der Trainingsmenge (Trainingsfehler) und auf einer davon unabhängigen Testmenge (Generalisierungsfehler).

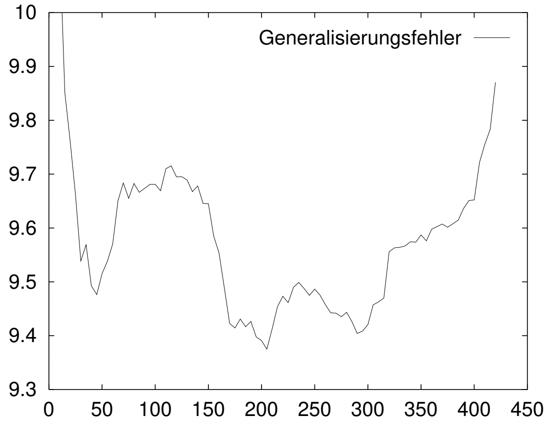


Abbildung 4.2: Beispiel eines realen zeitlichen Verlaufs des Generalisierungsfehlers (Trainingsepochen in x-Richtung, Fehler in y-Richtung, Datensatz glass1, Netz mit 4+4 verborgenen Knoten und allen Direktverbindungen).

Verhalten des Generalisierungsfehlers bei weitem komplizierter ist. Ein Beispiel ist in Abbildung 4.2 gezeigt; dieser Verlauf des Generalisierungsfehlers hat 16 lokale Minima bevor etwa in Epoche 400 die endgültige Überanpassung einsetzt.

Theoretische Aussagen über diesen Verlauf gibt es bislang nur für lineare Netze. Baldi und Chauvin zeigen analytisch in [26], daß lineare Netze mit n Eingängen und n Ausgängen bis zu n lokale Minima des Generalisierungsfehlers haben können. Wang und Venkatesh [387] zeigen analytisch für lineare Netze mit einem Ausgang, daß die Entwicklung des Generalisierungsfehlers drei Phasen durchläuft, zu deren Beschreibung sie ihn in einen Teil aufspalten, der dem Fehler auf der Trainingsmenge zugehört (Approximationsfehler) und einen anderen Teil, der aus der Komplexität des Netzes erwächst (entspricht etwa der Varianz). In Phase 1 wird der Generalisierungsfehler vom Approximationsfehler dominiert, in Phase 2 wetteifern beide Komponenten und in Phase 3 ist der Komplexitätsfehler dominierend. Der optimale Stopzeitpunkt liegt deshalb irgendwann in Phase 2, während derer zahlreiche lokale Minima des Generalisierungsfehlers auftreten können. Das frühe Stoppen leistet eine Begrenzung des Komplexitätsfehlers, indem es die *effektive Komplexität* des Netzes beschränkt. Das liegt vereinfacht gesagt daran, daß die Gewichte während einer begrenzten Trainingsperiode nicht unbegrenzt wachsen können und deshalb der von den k Gewichten aufgespannte k-dimensionale Parameterraum in jeder seiner Dimensionen nur eine endliche Ausdehnung hat, die vom Stopzeitpunkt limitiert wird. Beide Analysen sagen nichts über das Verhalten von Netzen mit verborgenen Knoten aus.

Aus dem Vorhandensein mehrerer lokaler Minima in der Generalisierungsfehlerkurve folgt, daß ein *optimales* Kriterium für das frühe Stoppen nicht so einfach sein kann, wie in der Regel oben angedeutet.

Auch wenn der Generalisierungsfehler ansteigt, kann man nicht sicher sein, daß er nicht später einmal auf einen noch kleineren Wert als bisher abfallen wird. Leider gibt es anscheinend keine typische Form von Generalisierungsfehlerkurven. Weder für die Anzahl noch für Breite, Tiefe und Form der Täler und Hügel in der Kurve lassen sich Regeln angeben.

Daraus ergeben sich zwei interessante Fragen zum Thema frühes Stoppen, die ich in diesem Kapitel untersuchen werde:

- Wie skaliert sich das gefundene Minimum des Generalisierungsfehlers beim Steigen der Trainingszeit?
- Wie skaliert sich die Trainingszeit bei unterschiedlichen Stoppkriterien?
- Zusammengefaßt: Welches Stoppkriterium sollte man verwenden, um eine gute Abwägung zwischen Generalisierungsfehler und benötigter Trainingszeit zu bekommen?

Die Beantwortung dieser Fragen wird noch durch einen weiteren Umstand kompliziert. Da der Generalisierungsfehler ja nicht direkt gemessen werden kann, sondern nur mit Hilfe einer Validationsmenge geschätzt wird, kann es passieren, daß das frühe Stoppen in die Irre führt: Der gewählte Stopzeitpunkt ist gar nicht wirklich günstig, sondern sieht nur aufgrund einer zufälligen Eigenschaft der endlichen Validationsmenge günstig aus. Dieser Effekt ist umso stärker, je kleiner die benutzte Validationsmenge ist; man kann sie aber nicht beliebig groß machen, da sie ja von den wertvollen Trainingsdaten abgezweigt werden muß. Ein solcher Fall ist in Abbildung 4.3 dargestellt.

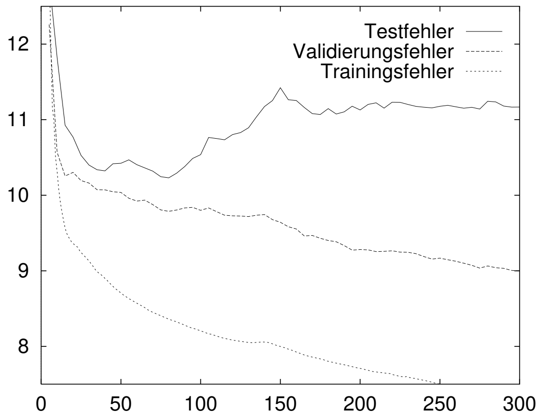


Abbildung 4.3: Beispiel für eine irreführende Schätzung des Generalisierungsfehlers. Das scheinbare Sinken des Generalisierungsfehlers, geschätzt durch den Validationsfehler (mittlere Kurve), findet im später zur Beurteilung des Netzes benutzten Testfehler (obere Kurve) keinen Niederschlag (Trainingsepochen in x-Richtung, Fehler in y-Richtung, Datensatz glass1, Netz mit 8+8 verborgenen Knoten und allen Direktverbindungen).

Hierzu noch zwei Anmerkungen. Erstens: Man kann dieses Problem verkleinern, indem man eine sogenannte n-fache Quervalidation ausführt. Hierfür werden n verschiedene Partitionierungen der Trainingsdaten in Trainings- und Validationsmenge vorgenommen und für jede ein anscheinend günstiger Stopzeitpunkt bestimmt. Aus diesen Stopzeitpunkten schätzt man dann einen, der für die Gesamtdatenmenge geeignet ist. Dieses Verfahren ist aber sehr berechnungsaufwendig. Zweitens: Das Irreführen durch die Ungenauigkeit der Schätzung kann bei der experimentellen Untersuchung von Lernverfahren genausogut bei der endgültigen Bewertung der Generalisierungsleistung mit Hilfe der Testmenge geschehen. Da sich der Schätzfehler hierbei jedoch mit gleicher Wahrscheinlichkeit positiv oder negativ niederschlagen kann, dürfen wir dieses Problem ignorieren.

4.2 Drei Familien von Stoppkriterien

Interessanterweise wird in der Literatur kaum je eine exakte Formulierung von Stoppkriterien für frühes Stoppen angegeben. Finnoff, Hergert und Zimmermann [117] sagen beispielsweise in ihrer ansonsten sehr sorgfältig dokumentierten Studie nicht mehr als „...training was stopped whenever overtraining was observed, defined by a repeated increase in the error on the validation set". Eine Klasse von Kriterien ist also mehrfacher Anstieg des Generalisierungsfehlers. Eine andere Klasse, die in der Literatur verwendet wird, ist das Ansteigen um einen gewissen Prozentsatz über das bisherige Minimum.

In diesem Abschnitt werde ich diese beiden Klassen von Stoppkriterien exakt definieren und eine dritte Klasse neu einführen, die plausiblerweise eine Verbesserung darstellen könnte.

4.2.1 GL

Sei E die Fehlerfunktion des Lernverfahrens, zum Beispiel der quadratische Fehler. Ferner sei $E_{tr}(t)$ der mittlere Fehler pro Beispiel über alle Beispiele der Trainingsmenge, gemessen in Epoche t. $E_{va}(t)$ ist analog dazu der mittlere Fehler auf der Validationsmenge nach Epoche t und $E_{te}(t)$ ist der mittlere Fehler auf der Testmenge.

$E_{opt}(t)$ wird definiert als der kleinste bis Epoche t je erreichte Validationsfehler:

$$E_{opt}(t) := \min_{t' \leq t} E_{va}(t')$$

Jetzt definieren wir den *Generalisierungsverlust* (*generalization loss*) in Epoche t als die relative Mehrgröße (in Prozent) des aktuellen Validationsfehlers, verglichen mit seinem bisherigen Minimum:

$$GL(t) := 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$

Ein hoher Generalisierungsverlust ist ein naheliegendes Kriterium, um das Training abzubrechen. Diese Überlegung definiert die erste Klasse von Stoppkriterien wie folgt: Wir definieren die Klasse GL_α als

$$GL_\alpha : \text{ stoppe nach der ersten Epoche } t \text{ für die } GL(t) > \alpha$$

4.2.2 UP

Die zweite obenerwähnte Klasse von Stoppkriterien betrachtet nur das Vorzeichen aufeinanderfolgender Änderungen des Generalisierungsfehlers; es wird gestoppt, wenn dieser mehrmals hintereinander ansteigt. Hier stellt sich die Frage, wie häufig der Generalisierungsfehler überhaupt gemessen wird. Die Messung nach jeder einzelnen Epoche stellt einen beträchtlichen zusätzlichen Rechenaufwand dar. Außerdem ist hierbei ein sehr erratisches Verhalten zu befürchten, für das sich schwer zuverlässige Stoppkriterien angeben lassen. Beide Probleme lassen sich auf einmal lösen, indem der Generalisierungsfehler nur gelegentlich gemessen wird. Wir definieren einen *Trainingsstreifen* (*training strip*) der Länge k als eine Sequenz von Epochen mit den Nummern $n+1, \ldots, n+k$, wobei n durch k teilbar ist. Messen wir nun den Generalisierungsfehler nur einmal nach jedem Trainingsstreifen, so haben wir $(k-1)/k$ des Aufwands gespart und erreichen zugleich eine erwünschte Glättung der Entwicklung des Fehlers, indem wir effektiv immer über k Änderungen integrieren. Durch einige Experimente wurde $k = 5$ als ein sinnvoller Wert gefunden.

Nun ergibt sich die zweite Klasse von Stoppkriterien (genannt UP, für „upwards") als „stoppe, wenn der Generalisierungsfehler in s aufeinanderfolgenden Trainingsstreifen größer wird":

$$\begin{aligned} UP_{s,k} &: \text{ stoppe nach Epoche } t \text{ genau dann, wenn } UP_{s-1,k} \text{ nach Epoche } t-k \text{ stoppt} \\ &\quad\; \text{und } \; E_{va}(t) > E_{va}(t-k) \; \text{ gilt} \\ UP_{1,k} &: \text{ stoppe nach erster durch } k \text{ teilbarer Epoche } t \text{ mit } E_{va}(t) > E_{va}(t-k) \end{aligned} \tag{4.1}$$

Wo im folgenden k nicht angegeben wird, ist jeweils der Wert 5 anzunehmen.

4.2.3 PQ

Eine bei aller Uneinheitlichkeit von Generalisierungsfehlerkurven wiederkehrende Beobachtung ist diese: Relativ früh im Verlauf des Trainings hat der Generalisierungsfehler fast immer einen „Buckel“ und erreicht danach ein niedrigeres Niveau als davor. Die weit nach dem Buckel folgenden weiteren lokalen Minima bringen, wenn überhaupt, meist nur noch kleine Verbesserungen. Zeitpunkt, Höhe, Breite und Form des Buckels können dabei variieren. Eine wünschenswerte Eigenschaft eines effizienten Stoppkriteriums wäre, diesen Buckel noch zu überspringen, danach aber relativ schnell anzuhalten. Der Schlüssel zur Realisierung dieses Wunsches liegt darin, daß der Buckel in einer recht frühen Phase des Trainings auftritt, wenn das Training noch schnell fortschreitet. Die Idee ist, diesen Trainingsfortschritt zu messen und ein Stoppkriterium zu benutzen, das bei langsamem Trainingsfortschritt eher zutrifft als bei schnellem.

Der Trainingsfortschritt kann daran gemessen werden, wie stark der Fehler auf der Trainingsmenge in jeder Epoche abnimmt. Dieses Maß ist jedoch erstens sehr instabil und wird zweitens unsinnig, wenn der Fehler einmal zunimmt, was gelegentlich vorkommt. Wir modifizieren es deshalb durch Glättung über einen Trainingsstreifen der Länge k und Gleichbehandlung von Zunahme und Abnahme des Fehlers („Hauptsache, es tut sich ordentlich was“). Wir definieren deshalb den *Trainingsfortschritt* (*training progress*) oder kurz *Fortschritt* (in Promille) als

$$P_k(t) = 1000 \cdot \left(\frac{\sum_{t' \in t-k+1 \ldots t} E_{tr}(t')}{k \cdot \min_{t' \in t-k+1 \ldots t} E_{tr}(t')} - 1 \right)$$

Diese Formel beschreibt, um wieviel der mittlere Trainingsfehler während des Streifens höher war als der minimale. Ähnliche Maße werden auch in der Literatur angegeben, allerdings meistens weniger robust, z.B. nur der Vergleich des Fehlers am Anfang und am Ende des Streifens. $P_k(t)$ ist hoch während Trainingsphasen mit schneller Abnahme des Fehlers, zum Beispiel zu Beginn des Trainings, und in Phasen vorübergehender Instabilität, es geht jedoch langfristig immer gegen Null, es sein denn, das Training ist global instabil (z.B. oszillierend), beispielsweise aufgrund zu hoher Lernraten.

Nun formulieren wir eine Klasse von Stoppkriterien, die nicht den Generalisierungsverlust allein, sondern das Verhältnis von Generalisierungsverlust und Fortschritt betrachten und deshalb zu Beginn des Trainings unempfindlich, später dann empfindlich reagieren. Die Klasse heißt *Generalisierungs/Fortschritts-Quotient* (*generalization/progress quotient* oder einfach *progress quotient*, *PQ*):

$$PQ_{\beta,k}: \text{ stoppe nach der ersten durch } k \text{ teilbaren Epoche } t \text{ mit } \frac{GL(t)}{P_k(t)} > \beta$$

Ein solches Kriterium ist meines Wissens bisher in der Literatur nicht vorgeschlagen worden.

Die Arbeitsweise dieses Kriteriums ist am Beispiel in Abbildung 4.4 zu erkennen. Die untere Kurve gibt den Generalisierungsverlust an. Ist dieser Null, so wird ein neues, bis dahin globales Optimum des Generalisierungsfehlers erreicht. Ein Training mit Stoppkriterium GL_α würde für $\alpha < 3,5$ immer das Netz aus Epoche 25 liefern. Mit $\alpha = 4$ wird zwar das optimale Netz aus Epoche 155 gefunden, dafür jedoch über 400 Epochen lang trainiert (nicht in der Abbildung erkennbar). $PQ_{2,5}$ überspringt den Buckel ebenfalls und findet das optimale Netz in Epoche 155, trainiert jedoch nur bis Epoche 200.

Im folgenden benutzen wir immer die Streifenlänge 5 und geben den Parameter k deshalb nicht extra an. Alle Experimente messen den Generalisierungsfehler nur am Ende jedes Trainingstreifens. Alle drei Klassen von Kriterien sind nicht hinreichend, um das Training abzubrechen, wenn keine oder nur sehr wenig Überanpassung auftritt. Deshalb ergänzen wir alle Kriterien um die Vorgabe, daß das Training auch dann abbricht, wenn der Fortschritt unter 0,1 sinkt oder wenn eine Maximalzahl von Epochen erreicht wird, die wir auf 3000 festsetzen.

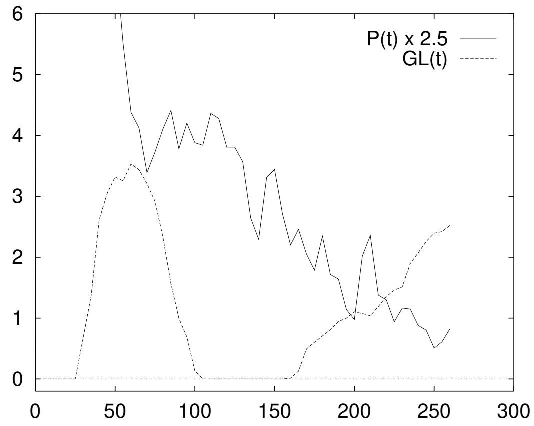


Abbildung 4.4: Entwicklung von Trainingsfortschritt und Generalisierungsverlust über die Zeit, mit „Generalisierungsbuckel". Zur leichteren Lesbarkeit ist der Fortschritt mit Faktor 2,5 überhöht dargestellt. (Trainingsepochen in x-Richtung, Datensatz heart1, Netz mit 6 verborgenen Knoten und Direktverbindungen.)

Welches der Kriterien die günstigste Abwägung darstellt, kann nur mit einer empirischen Untersuchung festgestellt werden, die im nächsten Abschnitt beschrieben ist.

4.3 Versuchsaufbau und Ergebnisse

4.3.1 Versuchsaufbau

Mit 12 der PROBEN1-Probleme aus dem vorherigen Kapitel wurde eine Serie von Testläufen mit 15 verschiedenen Stoppkriterien durchgeführt. Die Probleme waren building, cancer, card, diabetes, flare, gene, glass, heart, hearta, horse, und soybean; um eine zu starke Beeinflussung der Resultate durch die Varianten des heart-Problems zu vermeiden, wurden heartc und heartac nicht benutzt. Die untersuchten Stoppkriterien sind GL_1, $GL_{1,5}$, GL_2, GL_3, GL_5, $PQ_{0,5}$, $PQ_{0,75}$, PQ_1, PQ_2, PQ_3, UP_2, UP_3, UP_4, UP_6 und UP_8. (Vorversuche ergaben, daß Kriterien, die noch wesentlich früher anhalten als GL_1 oder UP_2 keine guten Ergebnisse bringen.)

Für jedes Problem wurden Läufe mit 12 verschiedenen Netztopologien gemacht. Diese waren Netze mit einer verborgenen Schicht mit 2, 4, 8, 16, 24 oder 32 Knoten und Netze mit zwei verborgenen Schichten mit 2+2, 4+2, 4+4, 8+4, 8+8 oder 16+8 Knoten in der ersten+zweiten Schicht. Die Netze hatten jeweils sämtliche möglichen Vorwärtsverbindungen. Für jede dieser Topologien und jedes Problem wurden drei Läufe ausgeführt; je einer mit sigmoider und zwei mit linearer Aktivierungsfunktion in den Ausgabeknoten. Alle Läufe verwendeten die quadratische Fehlerfunktion und die Aktivierungsfunktion $y = x/(1 + |x|)$ in den verborgenen Knoten und ggf. in den Ausgabeknoten. Dies gilt, soweit nichts anderes gesagt wird, auch für alle weiteren Versuchsreihen in dieser Arbeit.

Für jedes Kriterium wurde jeweils erfaßt nach welcher Epoche es stoppte, welches der Validations- und der Testfehler im gefundenen Validationsfehlerminimum waren und in welcher Epoche dieses Minimum zuvor erreicht worden war. Auch für die Läufe mit Klassifikationsproblemen betrachten wir immer den quadratischen Fehler und nicht den Klassifikationsfehler, weil wir in dieser Arbeit stets an einer Approximation der a-posteriori Wahrscheinlichkeiten der Klassen interessiert sind. Jeder der Läufe erfaßte die Ergebnisse für jedes der 15 Stoppkriterien zugleich. Dieses Vorgehen verringert die Varianz in den Ergebnissen, indem den Kriterien exakt gleiche Ausgangsbedingungen zur Verfügung gestellt werden [240]. Alle Läufe verwendeten als Lernregel das RPROP-Verfahren mit den Parametern $\eta^+ = 1,1$, $\eta^- = 0,5$, $\Delta_0 \in 0,05 \ldots 0,2$ zufällig pro Verbindung, $\Delta_{max} = 50$, $\Delta_{min} = 0$, anfängliche Gewichte zufällig aus dem Bereich $-0,5 \ldots 0,5$ pro Verbindung. Insgesamt wurden also 1296 Trainingsläufe durchgeführt, die zusammen 19440 Datenpunkte für die 15 Stoppkriterien lieferten. Das Epochenlimit betraf dabei 1,5% der Stops aus insgesamt 9,6% der Läufe. Die Läufe wurden vom 26. März bis zum 7. Mai 1994 auf der MasPar MP-1 mit 16384 Prozessoren des Rechenzentrums der Universität Karlsruhe durchgeführt. Das dabei verwendete Programm war in der im zweiten Teil dieser Arbeit vorgestellten

Programmiersprache CuPit geschrieben und wurde mit Hilfe des ebenfalls im zweiten Teil vorgestellten Übersetzers für den Parallelrechner übersetzt.

4.3.2 Bewertungskriterien

Zur Beschreibung der Eigenschaften der Stoppkriterien führen wir einige Größen ein: Für jeden Lauf definieren wir $E_{va}(C)$ als den kleinsten gefundenen Fehler auf der Validationsmenge bis zum von Kriterium C angezeigten Stopzeitpunkt; dieser Fehler wird nach Epoche $t_m(C)$ erreicht, das Anhalten erfolgt nach Epoche $t_s(C)$. Aus den beiden letzteren Werten ergibt sich der *Wirkungsgrad* von C als der Anteil der verwendeten Epochen, die zum Erreichen des Punktes mit kleinstem Validationsfehler tatsächlich nötig waren, zu $\eta_C := t_m(C)/t_s(C)$. Dieses Maß unterdrückt bei Durchschnittsbildung den Einfluß von Läufen mit extrem großem Unterschied von t_m und t_s, weshalb wir den mittleren Wirkungsgrad als Kehrwert aus dem Mittel der Kehrwerte berechnen. $E_{te}(C)$ ist der Fehler auf der Testmenge in Epoche $t_m(C)$.

Das *beste* Kriterium $\hat{C}$ eines Laufes ist dasjenige mit dem kleinsten t_s unter allen mit kleinstem E_{te}; es kann gelegentlich mehrere beste Kriterien in einem Lauf geben. Daß wir das beste Kriterium über den Testfehler anstatt über den Validationsfehler definieren, bedeutet, daß der oben erwähnte Irreführungseffekt aufgrund ungenauer Fehlerschätzungen durch zu kleine Validationsmenge mit in die Bewertung einfließt. Ein Kriterium C heißt *„gut"* in einen Lauf, wenn $E_{te}(C) = E_{te}(\hat{C})$. $P_g(C)$ ist die Wahrscheinlichkeit dafür, daß C in einem Lauf „gut" ist, gemessen über die 1296 Läufe.

Ein Kriterium heißt *schnell* wenn sein t_s typischerweise klein im Vergleich zu anderen Kriterien ist; entsprechend heißen manche anderen Kriterien *langsam*. Der relative *Aufwand* eines Kriteriums C gegenüber einem anderen Kriterium x ist $A_x(C) := t_s(C)/t_s(x)$, also die relative Anzahl benötigter Epochen. Wir werden unten $A_{\hat{C}}$ betrachten, also den Aufwand relativ zum besten Kriterium jedes Laufs. Als *Verlust* V des Kriteriums C bezeichnen wir die Verschlechterung des mit C gegenüber $\hat{C}$ erzielten Testfehlers, also $V(C) := E_{te}(C)/E_{te}(\hat{C})-1$. Der *scheinbare Verlust* v ist das analoge Maß für den Validationsfehler $v(C) := E_{va}(C)/E_{va}(\hat{C}) - 1$. Zur gemeinsamen Beurteilung der Effektiviät und Effizienz eines Kriteriums bilden wir als Kostenmaß K das Produkt aus Aufwand und Verlust, also $K(C) = A(C) \cdot V(C)$. Alle „guten" Kriterien in einem Lauf haben also Kosten Null. Leider ist dieses Maß sehr instabil; bei direkter Summation über die Werte ergeben sich Standardabweichungen, die ein Vielfaches der Mittelwerte betragen. So sind beispielsweise die mittleren Kosten $\overline{K(GL_3)} = 16,6$ und $\overline{K(GL_{1,5})} = 8,9$. Der scheinbar große Unterschied ist jedoch eine Illusion, weil die Standardabweichung für GL_3 88 beträgt. Wir betrachten deshalb unten für einen Vergleich zwei geglättete Fassungen: Zum einen die sich aus mittlerem Aufwand und mittlerem Verlust ergebenden Kosten und zum anderen logarithmisch transformierte Kosten nur der nicht-„guten" Kriterien jedes Laufs.

4.3.3 Ergebnisse: Mittelwertsbetrachtung

Die mittleren Resultate aller Stoppkriterien über die 1296 Läufe sind in der Tabelle 4.5 dargestellt. Bei der Interpretation dieser Aufstellung ist stets zu bedenken, daß ihre Einträge mit Ausnahme des Wirkungsgrads allesamt von der konkreten Auswahl der zugrundeliegenden Kriterienmenge abhängen. Aus dieser Aufstellung ergibt sich, daß UP_3, UP_4 und UP_6 besonders günstige Kriterien sind, wenn man die Abwägung des mittleren Aufwands gegen den mittleren Verlust betrachtet. Für die Abwägung des Aufwands gegen die Wahrscheinlichkeit P_g des „Gut"-seins liefern $GL_{1,5}$, GL_2, GL_3 und GL_5 besonders gute Ergebnisse. Diese beiden Zusammenhänge sind graphisch nochmals in Abbildung 4.6 dargestellt. Diese Ergebnisse sind jedoch mit Vorsicht zu interpretieren. Durch das getrennte Mitteln über Aufwand und Verlust ist nicht auszuschließen, daß die diesbezüglichen Beobachtungen Artefakte sind, es müßte direkt über die Kosten pro Kriterium und Lauf gemittelt werden, was aber die oben diskutierten Probleme aufwirft.

C	$A_{\hat{c}}(C)$	$v(C)$	$V(C)$	η_C	$P_g(C)$
UP_2	792	58	55	783	587
GL_1	951	50	44	766	678
$GL_{1,5}$	956	50	44	765	*680
UP_3	1010	22	*26	705	631
GL_2	1237	36	34	661	*723
UP_4	1243	13	*20	625	666
$PQ_{0,5}$	1253	21	27	601	658
$PQ_{0,75}$	1466	14	21	537	682
GL_3	1550	24	25	584	*748
PQ_1	1635	10	18	491	704
UP_6	1786	4	*12	471	737
GL_5	2014	20	21	463	*772
PQ_2	2184	4	12	379	768
UP_8	2485	1	10	354	759
PQ_3	2614	1	9	318	800

Tabelle 4.5: Stoppkriterien und ihr mittlerer Aufwand $A_{\hat{c}}(C)$, mittlerer scheinbarer Verlust $v(C)$, mittlerer Verlust $V(C)$, mittlerer Wirkungsgrad η_C (als Kehrwert des Mittels der Kehrwerte) und die Wahrscheinlichkeit $P_g(C)$ des „Gut“-seins gemessen über alle 1296 Läufe. Alle Einträge sind zur leichteren Lesbarkeit in Promille angegeben, d.h. mit dem Faktor 1000 multipliziert und sind geordnet nach mittlerem Aufwand. Die Sterne markieren die besonders günstigen Fälle, die dadurch definiert sind, daß sie den monotonen Zusammenhang zwischen $A_{\hat{c}}(C)$ und $V(C)$ bzw. $P_g(C)$ durchbrechen.

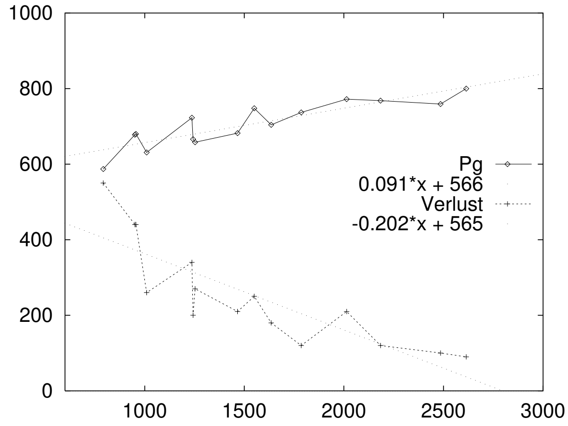


Abbildung 4.6: x-Achse: mittlerer Aufwand $A_{\hat{c}}(C)$. y-Achse: Wahrscheinlichkeit $P_g(C)$ des „Gut“-seins (obere Kurve) und mittlerer Verlust $V(C)$ (untere Kurve). Jeder Datenpunkt steht für eines der 15 Kriterien. Zu beiden Kurven ist jeweils die zugehörige Regressionsgerade angegeben.

4.3.4 Versuch der statistischen Prüfung

Durch die obenerwähnte Instabilität ist eine solche direkte Betrachtung der mittleren Kosten jedoch unsinnig und würde irreführen. Durch einen Kunstgriff können wir allerdings trotzdem eine statistische Inferenz durchführen: Der mehrfache Vergleich von Mittelwerten wird mit einer statistischen Prozedur namens *Varianzanalyse* (*analysis of variance*, *ANOVA*) durchgeführt. Diese setzt normalverteilte Stichproben mit annähernd gleicher Varianz voraus, die Standardabweichungen dürfen sich höchstens um Faktor 2 unterscheiden. Unsere Kostenwerte erfüllen keine dieser beiden Voraussetzungen auch nur im entferntesten. Schließt man jedoch für jeden Lauf die „guten“ Kriterien, die ja Kosten Null haben, aus (es verbleiben 5544 Datenpunkte), so sind die verbleibenden Kostenwerte annähernd log-normal verteilt, d.h. der Logarithmus der Kostenwerte hat ungefähr eine Normalverteilung. Die Standardabweichungen dieser Verteilungen liegen für die 15 Kriterien zwischen 1,0 und 1,6. Somit kann auf den Logarithmen der Kostenwerte für die nicht-„guten“ Kriterien jedes Laufs eine ANOVA ausgeführt werden. Da der Logarithmus monoton ist, sind die Mittelwerte der Logarithmen genau dann statistisch signifikant verschieden, wenn dies auch für die originalen Kostenwerte gilt. Der Unterschied liegt darin, daß wir für die Logarithmen eine Prozedur kennen, eben die ANOVA, die diese Signifikanz zu prüfen erlaubt. Solche Transformationen der Daten zur Erfüllung der Anwendungsbedingungen sind ein normales Vorgehen in der Statistik.

Ich habe die Prozedur `glm` (general linear models) aus dem SAS-Statistikpaket verwendet [313], um einen mehrfachen Vergleich von Mittelwerten (je einen pro Stoppkriterium) auszuführen. Dabei wurde eine simultane Konfidenz von 95% nach der Bonferroni-Methode eingestellt (Anzahl der Freiheitsgrade $df = 5529$). Dieses Vorgehen ist äquivalent zur Durchführung von 105 separaten Tests der Hypothese, daß zwei Mittelwerte gleich sind (ein Test für jedes mögliche Paar von Stoppkriterien), mit einem Konfidenzniveau, welches so hoch gewählt ist, daß die Wahrscheinlichkeit für einen Irrtum über alle 105 Tests zugleich bei 5% liegt. Auf diesem Konfidenzniveau bilden die Kriterien fünf einander überlappende Klassen von Kriterien mit nicht signifikant verschiedenem Mittelwert für die Kosten innerhalb jeder Klasse. In jeder dieser Klassen liegen Kriterien aller drei Arten (GL, UP, PQ). Dies deutet darauf hin, daß keine der Klassen signifikant besser ist als eine andere. An diesem Ergebnis ändert sich selbst dann nichts, wenn man das Konfidenzniveau auf 50% senkt; dies ist eine wichtige Feststellung, weil die Trennschärfe (*power*) des Tests aufgrund der recht hohen Zahl von 15 Gegenständen nicht sehr hoch ist. Eine Betrachtung der simultanen Bonferroni-Konfidenzintervalle auf Konfidenzniveau 95% liefert das Ergebnis, daß einige Kriterien signifikant besser oder schlechter als das Mittel *aller* Kriterien sind. Dies ist daran zu erkennen, daß ihre Konfidenzintervalle vollständig unter bzw. über dem Mittel aller Kriterien liegen. Die in diesem Sinne signifikant schlechteren Kriterien sind (in dieser Reihenfolge) PQ_3, GL_5, UP_8 und PQ_2. Die signifikant besseren sind GL_1, $GL_{1,5}$ und UP_3. Stärker als die Unterschiede zwischen den Klassen ist also der Effekt, daß langsame Kriterien weniger kosteneffizient sind als schnelle; zusätzlicher Aufwand in der Laufzeit läßt sich offenbar nicht voll durch Verringerung des Fehlers ausgleichen.

Ebenso wie die summarischen Resultate oben müssen auch diese Ergebnisse mit Vorsicht interpretiert werden, denn die Beurteilungsgrundlage war durch das Ausscheiden derjenigen Kriterien jedes Laufes mit Kosten Null erheblich beschnitten.

4.3.5 Zusammenfassung

Die Resultate stellen sich zusammengefaßt so dar:

1. Die Laufzeitkosten für eine Verbesserung des Testfehlers sind hoch. Auf den PROBEN1-Problemen kann eine Verbesserung um etwa 4% bei Verdreifachung des Zeitaufwands erreicht werden. Innerhalb dieses Bereichs kann durch Skalierung des geschwindigkeitsbestimmenden Parameters des Stoppkriteriums aus einer Reihe von Kompromissen ausgewählt werden.

2. Statistisch signifikante systematische Unterschiede zwischen den drei Kriterienklassen GL, UP und PQ gibt es nicht. Die UP-Kriterien sind tendenziell besonders gut in der Abwägung zwischen mittlerer Laufzeit und mittlerem Fehler, die GL-Kriterien in der Abwägung zwischen mittlerer Laufzeit und der Wahrscheinlichkeit dafür, ein „gutes" Ergebnis zu finden. Die neu vorgeschlagene Klasse PQ scheint trotz ihrer Plausibilität eher schlechter denn besser als die herkömmlichen zu sein. Vermutlich wären PQ-Kriterien erfolgreicher, wenn sie einen quadratischen anstatt linearen Zusammenhang zwischen Fortschritt und Generalisierungsverlust benutzen würden. Zur Überprüfung müßte jedoch die gesamte Versuchsreihe wiederholt werden, worauf ich hier verzichte.

3. Gegeben sei ein Kostenmaß, das als Kosten das Produkt aus Laufzeit und zusätzlichem Fehler (beides gegenüber dem jeweils besten Kriterium) verwendet, ist der einzige statistisch signifikante Trend der, daß die zusätzlich aufgewendete Laufzeit bei langsamen Stoppkriterien sich nicht voll durch Fehlerersparnis ausgleicht. Um dieses Ergebnis zu erhalten, mußten allerdings alle Datenpunkte ignoriert werden, die zu „guten" Ergebnissen gehören.

Da die PQ-Kriterien und die UP-Kriterien naturgemäß empfindlich von der gewählten Streifenlänge und möglicherweise auch von der verwendeten Lernregel abhängen, werde ich in Zukunft vorwiegend die robusten GL-Kriterien verwenden.

4.4 Lernresultate als Vergleichsbasis

Zur Beurteilung der in den nächsten beiden Kapiteln untersuchten konstruktiven Lernverfahren wollen wir Lernergebnisse, die mit der Methode des frühen Stoppens erzielt wurden, als Vergleichsmaßstab verwenden. Deshalb wählen wir anhand der Ergebnisse der oben beschriebenen Trainingsläufe für jedes Problem eine günstige Netztopologie aus, die ich Pivot-Architektur nenne (Abschnitt 4.4.1), und präsentieren die Ergebnisse des Lernens mit frühem Stoppen auf diesen Netzen (Abschnitt 4.4.2).

4.4.1 Pivot-Architekturen

Im obigen Experiment wurden für jede Netztopologie mit jedem Datensatz nur ein oder zwei Läufe ausgeführt. Aufgrund dieser kleinen Zahl hat auch eine suboptimale Topologie eine gute Chance, rein aus Zufall das beste Ergebnis aller Läufe für einen Datensatz hervorzubringen, da die Ergebnisse ja wegen der Zufallsinitialisierung des Netzes schwanken. Es stellt sich also die Frage, welche der Topologien man für jeden Datensatz auswählen und als gute Topologie empfehlen sollte. Die Erfahrungen mit der Methode des frühen Stoppens zeigen, daß oftmals die Verwendung von solchen Netzen die besten Ergebnisse bringt, die erheblich größer sind als die minimale Architektur, mit der brauchbare Resultate auf dem gleichen Datensatz erzielt werden können. Diese Erfahrung setzen wir in eine Regel um, mit der aus den Resultaten der obigen Experimente für jedes Problem eine empfohlene Architektur ausgewählt wird, die sogenannte *Pivot-Architektur*. Die Regel lautet: Sortiere alle Läufe für einen Datensatz nach dem dabei erzielten minimalen Validationsfehler und betrachte den besten Lauf, sowie alle Läufe die höchstens 5% schlechter sind. Wähle unter allen darunter vorkommenden Netzen das größte. Dies ist die Pivot-Architektur. Hierbei können noch Kollisionen zwischen den gleichgroßen Netzen mit sigmoider versus linearer Ausgangsaktivierungsfunktion auftreten, die zuerst nach Mehrheit und dann nach kleinerem Validationsfehler entschieden werden.

Die Ergebnisse dieser Auswahl sind in der Tabelle 4.7 dargestellt. Die Tabelle gibt die Pivot-

Problem	Pivot-Arch.		Problem	Pivot-Arch.		Problem	Pivot-Arch.	
building1	16+0	l	building2	16+8	l	building3	16+8	s
cancer1	4+2	l	cancer2	8+4	l	cancer3	16+8	l
card1	32+0	l	card2	24+0	l	card3	16+8	l
diabetes1	32+0	l	diabetes2	16+8	l	diabetes3	32+0	l
flare1	32+0	s	flare2	32+0	s	flare3	24+0	s
gene1	4+2	l	gene2	4+2	s	gene3	4+2	s
glass1	16+8	l	glass2	16+8	l	glass3	16+8	l
heart1	32+0	l	heart2	32+0	l	heart3	32+0	l
hearta1	32+0	l	hearta2	16+0	l	hearta3	32+0	l
heartac1	2+0	l	heartac2	8+4	l	heartac3	16+8	s
heartc1	16+8	l	heartc2	8+8	l	heartc3	32+0	l
horse1	16+8	l	horse2	16+8	l	horse3	32+0	l
soybean1	16+8	l	soybean2	32+0	l	soybean3	16+0	l
thyroid1	16+8	l	thyroid2	8+4	l	thyroid3	16+8	l

Tabelle 4.7: Die Pivot-Architekturen für die einzelnen Datensätze. Angegeben sind die Anzahl von Knoten in der ersten und ggf. zweiten verborgenen Schicht sowie die Aktivierungsfunktion der Ausgabeknoten als „l“ für linear oder „s“ für sigmoid.

Architektur jedes Datensatzes an, beschrieben durch die Anzahl von verborgenen Knoten in der ersten und ggf. zweiten verborgenen Schicht sowie die Aktivierungsfunktion der Ausgabeknoten als „l“ für linear oder „s“ für sigmoid.

Es ist wichtig festzustellen, daß diese Pivot-Architekturen natürlich nicht notwendigerweise optimal sind, auch nicht innerhalb der zugrundegelegten Klasse von Architekturen. Es ist jedoch davon auszugehen, daß die Pivot-Architekturen zumindest eine brauchbare Auswahl darstellen. Eine mögliche Verbesserung könnte in vielen Fällen darin bestehen, Netze ohne Direktverbindungen zu verwenden, da diese eine erheblich kleinere Zahl von freien Parametern haben. Beispielsweise machen die Direktverbindungen von den Eingängen zu den Ausgängen beim glass-Problem 60 Gewichte aus; etwa ebensoviel wie ein komplettes Netz mit 4 verborgenen Knoten ohne Direktverbindungen hat. Da die Trainingsmenge dieses Problems nur 107 Beispiele aufweist, kann das Weglassen dieser 60 Verbindungen möglicherweise die Lernergebnisse verbessern, weil die Neigung zur Überanpassung abgeschwächt wird. Ähnliche Überlegungen gelten auch für mehrere der anderen PROBEN1-Probleme. Aus diesem Grund verwende ich in den unten beschriebenen Versuchsreihen neben den Pivot-Architekturen alternativ auch die gleichen Architekturen ohne Direktverbindungen, also mit Netzen, die ausschließlich Verbindungen zur jeweils nächsten Schicht aufweisen, jedoch keine Verbindungen direkt von Eingängen zu Ausgängen und bei Netzen mit zwei verborgenen Schichten auch keine Verbindungen, die eine verborgene Schicht überspringen. Ich nenne die so erhaltenen Architekturen *Reinschicht-Pivotarchitekturen*.

4.4.2 Ergebnisse

Für jeden Datensatz wurden 60 Läufe durchgeführt. Es wurden dieselben Parameter für das Lernverfahren verwendet, wie für die Läufe mit linearen Netzen in Abschnitt 3.2.17. Die Tabellen 4.8 (Klassifikationsprobleme) und 4.9 (Approximationsprobleme) zeigen die Ergebnisse der Trainingsläufe mit den Pivot-Architekturen. Es können eine Reihe interessanter Beobachtungen gemacht werden (siehe auch Abschnitt 3.2.17 zum Vergleich):

1. Für manche der Probleme sind die erzielten Resultate *schlechter* als mit linearen Netzen; am stärksten bei gene, aber auch bei horse und vielen der heart-Probleme.
2. Die Standardabweichungen der Validations- und Testfehler und die Neigung zur Überanpassung sind in den meisten Fällen wesentlich höher als bei linearen Netzen. Dies ist wenig überraschend, da die höhere Komplexität der mehrschichtigen Netze ein solches Verhalten nahelegt.
3. Die Korrelation der Validationsfehler mit den Testfehlern ist bei einigen Problemen recht klein: Weniger als 0,5 bei cancer3, card3, flare3, glass1, heartac1, heartc1, horse1, horse2, soybean3; in zwei Fällen sogar knapp negativ (card3, heartac1).
4. Die Korrelation ist bei einigen Problemen dramatisch unterschiedlich über die drei Varianten hinweg (card, flare, heartac, heartc, horse).
5. Allerdings bedeutet eine geringe Korrelation trotzdem keine insgesamt schlechten Resultate für den Testfehler; siehe cancer, card, flare, heartac, horse.
6. In einigen Fällen treten dramatische Variationen der Trainingsdauer auf (building1, gene2, gene3, thyroid3 und schwächer auch cancer1, cancer2, diabetes3, glass1, glass3, thyroid1, thyroid2).
7. Ansonsten sind die Trainingsdauern (gemessen in Anzahl Epochen) meist von derselben Größenordnung wie bei den linearen Netzen, mit ein paar Ausnahmen, die wesentlich niedriger liegen (die meisten der heart-Probleme) oder wesentlich höher (thyroid, building2, building3).

Die Tabellen 4.11 (Klassifikationsprobleme) und 4.10 (Approximationsprobleme) stellen die Ergebnisse der Läufe mit den Reinschicht-Pivotarchitekturen dar. Auch hier wurden für jeden Datensatz 60 Läufe durchgeführt und dieselben Parameter für das Lernverfahren verwendet. Wiederum können einige interessante Beobachtungen gemacht werden:

1. Das Weglassen der Direktverbindungen scheint häufiger angemessen zu sein als erwartet (siehe nachfolgende Untersuchung).
2. Die Testfehler bei den gene-Problemen sind nun besser als bei linearen Netzen (bei den Pivot-Architekturen sind sie schlechter). Die Klassifikationsfehler haben diese Verbesserung allerdings nicht nachvollzogen: Sie sind sogar noch schlechter als bei den Pivot-Architekturen. Dies ist ein

Tabelle 4.8: Resultate für Klassifikationsprobleme mit Pivot-Architekturen

Problem	Trainings-menge		Validations-menge		Test-menge		ρ	Testmengen-klassifikation		Über-anpassung		Epochen total		Epochen relevant	
	μ	σ	μ	σ	μ	σ		μ	σ	μ	σ	μ	σ	μ	σ
cancer1	2,87	0,27	1,96	0,25	1,60	0,41	0,81	1,47	0,60	4,48	4,87	152	111	133	97
cancer2	2,08	0,35	1,77	0,32	3,40	0,33	0,51	4,52	0,70	5,76	6,70	93	75	81	72
cancer3	1,73	0,19	2,86	0,11	2,57	0,24	0,28	3,37	0,71	3,37	1,32	66	20	51	16
card1	8,92	0,54	8,89	0,59	10,53	0,57	0,92	13,64	0,85	3,77	4,47	33	7	25	5
card2	7,12	0,55	11,11	0,32	15,47	0,75	0,53	19,23	0,80	3,32	1,03	32	8	22	6
card3	7,58	0,87	8,42	0,37	13,03	0,50	-0,03	17,36	1,61	3,52	1,46	37	10	28	9
diabetes1	14,74	2,03	16,36	2,14	17,30	1,91	0,99	24,57	3,53	2,31	0,67	196	98	118	72
diabetes2	13,12	1,35	17,10	0,91	18,20	1,08	0,77	25,91	2,50	2,75	2,54	119	42	85	31
diabetes3	13,34	1,11	17,98	0,62	16,68	0,67	0,55	23,06	1,91	2,34	0,65	307	193	200	132
gene1	6,45	0,42	10,27	0,31	10,72	0,31	0,76	15,05	0,89	2,67	0,49	46	9	29	6
gene2	7,56	1,81	11,80	1,19	11,39	1,28	0,97	15,59	1,83	2,12	0,44	321	698	222	595
gene3	6,88	1,76	11,18	1,06	12,14	0,95	0,95	17,79	1,73	2,06	0,50	435	637	289	508
glass1	7,68	0,79	9,48	0,24	9,75	0,41	0,33	39,03	8,14	2,76	0,71	67	44	45	39
glass2	8,43	0,53	10,44	0,48	10,27	0,40	0,72	55,60	2,83	4,27	1,75	29	9	20	7
glass3	7,56	0,98	9,23	0,25	10,91	0,48	0,54	59,25	7,83	2,68	0,47	66	46	45	41
heart1	9,25	1,07	13,22	1,32	14,33	1,26	0,97	19,89	2,27	2,83	1,89	65	16	43	12
heart2	9,85	1,68	13,06	3,29	14,43	3,29	0,98	17,88	1,57	3,27	2,34	57	19	38	13
heart3	9,43	0,64	10,71	0,78	16,58	0,39	0,67	23,43	1,29	3,35	3,72	51	10	37	9
heartc1	6,82	1,20	8,75	0,71	17,18	0,79	0,10	21,13	1,49	4,04	2,98	45	12	36	11
heartc2	10,41	1,76	17,02	1,12	6,47	2,86	0,83	5,07	3,37	4,05	1,89	29	14	21	11
heartc3	10,30	1,79	15,17	1,83	14,57	2,82	0,85	15,93	2,93	8,22	18,67	24	13	17	11
horse1	9,91	1,06	16,52	0,67	13,95	0,60	0,30	26,65	2,52	4,66	2,28	28	5	20	4
horse2	7,32	1,52	16,76	0,64	18,99	1,21	0,30	36,89	2,12	3,87	1,49	31	8	22	8
horse3	9,25	2,36	17,25	2,41	17,79	2,45	0,92	34,60	2,84	3,48	1,26	30	10	21	7
soybean1	0,32	0,08	0,85	0,07	1,03	0,05	0,54	9,06	0,80	2,55	1,37	665	259	551	218
soybean2	0,42	0,06	0,67	0,06	0,90	0,08	0,77	5,84	0,87	2,17	0,16	792	281	675	243
soybean3	0,40	0,07	0,82	0,06	1,05	0,09	0,33	7,27	1,16	2,16	0,13	759	233	639	205
thyroid1	0,60	0,53	1,04	0,61	1,31	0,55	0,99	2,32	0,67	3,06	3,16	491	319	432	266
thyroid2	0,59	0,24	0,88	0,19	1,02	0,18	0,85	1,86	0,41	2,58	1,07	660	460	598	417
thyroid3	0,69	0,20	0,97	0,13	1,16	0,16	0,91	2,09	0,31	2,39	0,43	598	624	531	564

Trainingsmenge: Mittelwert μ und Standardabweichung σ des minimalen, normalisierten quadratischen Fehlers auf der Trainingsmenge („Trainingsfehler"), der irgendwann während des Trainings erreicht wurde.
Validationsmenge: dito, auf der Validationsmenge („Validationsfehler").
Testmenge: Mittelwert und Standardabweichung des normalisierten quadratischen Fehlers auf der Testmenge („Testfehler") am Punkt des minimalen Fehlers auf der Validationsmenge.
ρ: Korrelation zwischen Validationsfehler und Testfehler.
Testmengenklassifikation: Mittelwert und Standardabweichung des zum Testfehler gehörigen Klassifikationsfehlers.
Überanpassung: Mittelwert und Standardabweichung des GL-Werts am Ende des Trainings.
Epochen total: Mittelwert und Standardabweichung der Anzahl tatsächlich durchgeführter Epochen.
Epochen relevant: Mittelwert und Standardabweichung der Anzahl Epochen bis zum Erreichen des minimalen Validationsfehlers.

Tabelle 4.9: Resultate für Approximationsprobleme mit Pivot-Architekturen

Problem	Trainings-menge		Validations-menge		Test-menge		ρ	Über-anpassung		Epochen total		Epochen relevant	
	μ	σ	μ	σ	μ	σ		μ	σ	μ	σ	μ	σ
building1	0,63	0,50	2,43	1,50	1,70	1,01	0,96	31,93	44,07	394	602	329	529
building2	0,23	0,02	0,28	0,02	0,26	0,02	0,98	0,11	0,70	1183	302	1175	303
building3	0,22	0,02	0,26	0,01	0,26	0,01	0,93	0,42	1,09	1540	466	1408	505
flare1	0,39	0,26	0,55	0,81	0,74	0,80	1,00	3,13	2,48	71	28	52	21
flare2	0,42	0,16	0,55	0,43	0,41	0,47	1,00	3,20	3,73	60	15	42	10
flare3	0,36	0,01	0,49	0,01	0,37	0,01	0,32	2,58	0,58	76	28	51	18
hearta1	3,75	0,76	4,58	0,81	4,76	1,14	0,95	4,98	7,85	46	16	34	13
hearta2	3,69	0,87	4,47	1,00	4,52	1,10	0,97	7,18	24,23	59	21	45	19
hearta3	3,84	0,66	4,29	0,73	4,81	0,87	0,97	5,34	14,19	45	13	35	13
heartac1	3,86	0,32	4,87	0,23	2,82	0,22	-0,06	3,98	2,25	44	23	34	21
heartac2	3,41	0,42	5,51	0,65	4,54	0,87	0,79	7,53	5,27	22	9	16	7
heartac3	2,23	0,57	5,38	0,37	5,37	0,56	0,80	4,64	2,96	38	10	30	10

(Es gilt die Erläuterung der Tabelle 4.8, außer daß hier keine Klassifikationsfehler angegeben sind.)

Tabelle 4.10: Resultate für Approximationsprobleme mit Reinschicht-Pivotarchitekturen

Problem	Trainings-menge		Validations-menge		Test-menge		ρ	Über-anpassung		Epochen total		Epochen relevant	
	μ	σ	μ	σ	μ	σ		μ	σ	μ	σ	μ	σ
building1	0,47	0,28	2,07	1,04	1,36	0,63	0,88	33,93	49,93	307	544	248	457
building2	0,24	0,15	0,30	0,19	0,28	0,20	1,00	0,14	0,78	1074	338	1044	330
building3	0,22	0,01	0,26	0,01	0,26	0,01	0,74	0,25	0,58	1380	350	1304	360
flare1	0,35	0,02	0,35	0,01	0,54	0,01	0,10	3,02	0,90	48	20	35	16
flare2	0,40	0,01	0,47	0,01	0,32	0,01	0,43	2,93	0,99	47	11	32	8
flare3	0,37	0,01	0,47	0,01	0,36	0,01	0,34	2,53	0,47	57	21	32	11
hearta1	3,55	0,53	4,48	0,35	4,55	0,41	0,93	4,17	7,53	47	18	35	16
hearta2	3,45	0,56	4,41	0,21	4,33	0,15	0,55	2,91	0,75	54	22	41	20
hearta3	3,74	0,72	4,46	1,01	4,89	0,91	0,99	5,35	9,90	46	17	34	15
heartac1	3,59	0,24	4,77	0,32	2,47	0,38	0,21	3,78	1,85	42	22	32	18
heartac2	2,58	0,42	5,16	0,32	4,41	0,56	-0,15	6,43	4,43	24	7	18	7
heartac3	2,45	0,46	5,74	0,36	5,55	0,52	0,84	5,52	4,02	31	12	23	10

(Es gilt die Erläuterung der Tabelle 4.8, außer daß hier keine Klassifikationsfehler angegeben sind.)

gutes Beispiel für die fehlende Monotonitätseigenschaft des Klassifikationsfehlers in Bezug auf Veränderungen des quadratischen Fehlers.

3. Die Testfehler für die horse-Probleme haben sich ebenfalls verbessert, sind aber immer noch schlechter als bei linearen Netzen.

4. Die Korrelationen zwischen Validations- und Testfehler sind teilweise sehr anders als bei den Pivot-Architekturen (siehe zum Beispiel card, flare, glass, heartac). Dies weist daraufhin, daß diese Korrelationen nicht allein von der Partitionierung der Daten bestimmt werden, sondern maßgeblich auch vom Lernverfahren bzw. der Netzarchitektur geändert werden können. Da eine gute Korrelation die Qualität eines Lernverfahrens tendenziell verbessert, ist dies also ein anzu-

Tabelle 4.11: Resultate für Klassifikationsprobleme mit Reinschicht-Pivotarchitekturen

Problem	Trainingsmenge		Validationsmenge		Testmenge		ρ	Testmengenklassifikation		Überanpassung		Epochen total		Epochen relevant	
	μ	σ	μ	σ	μ	σ		μ	σ	μ	σ	μ	σ	μ	σ
cancer1	2,83	0,15	1,89	0,12	1,32	0,13	0,64	1,38	0,49	3,10	2,54	116	123	95	115
cancer2	2,14	0,23	1,76	0,14	3,47	0,28	0,14	4,77	0,94	3,82	1,90	54	31	44	28
cancer3	1,83	0,26	2,83	0,13	2,60	0,22	0,59	3,70	0,52	3,33	1,64	54	20	41	17
card1	8,86	0,41	8,69	0,26	10,35	0,29	0,25	14,05	1,03	3,54	1,25	30	7	22	5
card2	7,18	0,51	10,87	0,27	14,94	0,64	0,44	18,91	0,86	3,99	1,52	26	7	17	5
card3	7,13	0,62	8,62	0,46	13,47	0,51	0,41	18,84	1,19	4,81	3,24	29	7	22	6
diabetes1	14,36	1,14	15,93	1,04	16,99	0,91	0,95	24,10	1,91	2,23	0,53	201	119	117	83
diabetes2	13,04	1,27	16,94	0,91	18,43	1,00	0,76	26,42	2,26	2,50	0,50	102	46	70	26
diabetes3	13,52	1,46	17,89	0,90	16,48	1,16	0,91	22,59	2,23	2,32	0,59	251	132	164	85
gene1	2,70	1,52	8,19	1,33	8,66	1,28	0,91	16,67	3,75	2,46	0,53	124	58	101	53
gene2	4,55	2,60	9,46	1,95	9,54	1,91	0,97	18,41	6,93	2,29	0,28	321	284	250	255
gene3	4,99	2,79	9,45	2,17	10,84	1,93	0,97	21,82	7,53	2,33	0,39	262	183	199	163
glass1	7,16	0,65	9,15	0,21	9,24	0,32	0,13	32,70	5,34	2,69	0,64	71	31	52	27
glass2	8,42	0,66	10,03	0,27	10,09	0,28	0,37	55,57	3,70	4,00	1,80	30	9	22	8
glass3	7,54	1,06	9,14	0,24	10,74	0,52	0,73	58,40	7,82	2,97	1,17	60	30	46	26
heart1	9,24	0,82	13,10	0,65	14,19	0,64	0,89	19,72	0,96	3,16	2,38	57	15	38	12
heart2	9,73	1,24	12,32	1,09	13,61	0,89	0,88	17,52	1,14	3,56	3,47	51	15	36	12
heart3	9,46	0,88	10,85	1,39	16,79	0,77	0,93	24,08	1,12	3,91	4,42	46	13	32	10
heartc1	5,98	1,33	8,08	0,49	16,99	0,77	0,22	20,82	1,47	5,08	2,64	38	10	30	9
heartc2	9,85	1,16	16,86	0,70	5,05	1,36	0,40	5,13	1,63	4,83	2,34	25	10	18	9
heartc3	10,35	1,07	14,30	1,21	13,79	2,62	0,75	15,40	3,20	9,73	10,48	17	6	11	5
horse1	10,43	1,23	15,47	0,37	13,32	0,48	0,24	29,19	2,62	6,09	2,53	19	3	13	3
horse2	6,68	1,85	16,07	0,79	17,68	1,41	-0,19	35,86	2,46	4,28	1,67	25	7	18	6
horse3	10,54	1,68	15,91	1,19	15,86	1,17	0,88	34,16	2,32	5,51	3,89	20	5	14	5
soybean1	1,53	0,09	1,94	0,06	2,10	0,07	0,58	29,40	2,50	3,14	1,99	219	112	159	79
soybean2	0,46	0,19	0,59	0,13	0,79	0,22	0,96	5,14	1,05	5,06	6,49	417	222	362	202
soybean3	0,61	0,21	0,93	0,21	1,25	0,15	0,76	11,54	2,32	6,12	7,99	450	273	382	228
thyroid1	0,59	0,20	1,01	0,16	1,28	0,12	0,84	2,38	0,35	3,99	7,14	377	308	341	280
thyroid2	0,60	0,13	0,89	0,11	1,02	0,11	0,59	1,91	0,24	4,71	6,86	421	269	388	246
thyroid3	0,74	0,18	0,98	0,13	1,26	0,14	0,92	2,27	0,32	3,91	9,18	324	234	298	223

(Es gilt die Erläuterung der Tabelle 4.8.)

strebendes Merkmal bei der Entwicklung von Lernverfahren.

5. Doch Merkwürdigkeiten allüberall: Bei flare2 und flare3 sind die Standardabweichungen der Testfehler erheblich kleiner als bei den Pivot-Architekturen, obwohl die Korrelationen viel geringer sind als bei jenen.

4.4.3 Statistischer Vergleich

Es stellt sich die Frage, ob und wo die Verbesserungen durch das Weglassen der Direktverbindungen statistisch signifikant sind. Tabelle 4.12 zeigt die Ergebnisse entsprechender Tests. Der Vergleich wurde mit der `ttest` Prozedur des Softwarepakets SAS [313] durchgeführt. Es wurden für jeden Datensatz die logarithmierten Testfehler der Läufe mit Pivot-Architekturen mit denen mit Reinschicht-

Problem	1	2	3
building	—	S 2,1	P 7,8
cancer	S 0,0	—	—
card	S 0,1	S 0,0	P 0,0
diabetes	—	P 2,9	S 2,1
flare	S 0,0	S 0,0	S 0,0
gene	S 0,0	(S 0,0)	(S 0,0)
glass	S 0,0	S 0,1	S 3,2
heart	S 1,6	S 0,6	—
hearta	—	—	P 0,0
heartac	S 0,0	—	(P 0,0)
heartc	—	S 0,0	S 2,6
horse	S 0,0	S 0,0	S 0,0
soybean	P 0,0	(S 0,0)	P 0,0
thyroid	P 6,9	—	P 0,1

Tabelle 4.12: Architekturvergleich mittels t-Test. Resultate statistischer Signifikanztests für den Vergleich der Mittelwerte der logarithmierten Testfehler der Läufe mit Pivot-Architekturen (P) gegenüber Reinschicht-Pivotarchitekturen (S) für die Varianten 1, 2, und 3 jedes Problems. Die Einträge zeigen Unterschiede, die auf einem 10%-Niveau signifikant sind und den zugehörigen p-Wert (in Prozent). Der Buchstabe gibt jeweils an, welche Architektur signifikant besser ist; ein Querstrich bedeutet, daß keine signifikanten Unterschiede vorliegen. Eingeklammerte Resultate sind unzuverlässig (dubios), weil mindestens eine der beteiligten Stichproben nicht normalverteilt ist. Es wurde ein t-Test mit Cochran/Cox-Approximation für ungleiche Varianzen verwendet. Ergebnis: 10 mal kein signifikanter Unterschied, 23 mal Reinschicht-Pivotarchitektur besser (davon 3 dubios), 9 mal Pivot-Architektur besser (davon 1 dubios).

Pivotarchitekturen verglichen. Der in der Tabelle eingetragene p-Wert gibt jeweils an, wie groß die Wahrscheinlichkeit dafür ist, daß der beobachtete Unterschied in den Mittelwerten der beiden Stichproben auf bloßem Zufall beruht und in Wirklichkeit die Mittelwerte der zugrundeliegenden Verteilungen identisch sind. Ein kleiner p-Wert deutet also auf einen erheblichen Unterschied hin, ein größerer p-Wert zeigt nur geringe Unterschiede an. Liegt der p-Wert über 10%, so wird (auf dem hier verwendeten Signifikanzniveau von 0,1) der Unterschied als nicht statistisch signifikant angesehen und in der Tabelle ein Strich eingetragen. Man beachte, daß keine simultane Konfidenz über alle 42 Tests hergestellt wird; jeder p-Wert gibt separat für den zugehörigen Test die gefundene Irrtumswahrscheinlichkeit für eine fälschlich als vorhanden erkannte Signifikanz an.

Beim t-Test wurde die Cochran/Cox-Approximation verwendet, weil zumindest bei einigen der Stichproben (cancer1, gene1, hearta1) die Standardabweichungen um mehr als Faktor 2 unterschiedlich sind. Außerdem waren einige Ausreißer aus den Stichproben zu entfernen, die die Normalverteilung ihrer Stichprobe erheblich verformt und damit die Testergebnisse verfälscht hätten: In den 2520 Läufen mit Pivot-Architekturen gab es 4 Ausreißer mit zu kleinen Fehlerwerten und 61 mit zu großen. Bei den Reinschicht-Pivotarchitekturen gab es keine zu kleinen Werte und 66 zu große. Insgesamt ergibt dies einen Wert von 2,6% entfernter Ausreißer. Maximal 10% Ausreißer, also 6 von 60 Werten, wurden aus einer einzelnen Stichprobe entfernt; dieser Fall trat nur bei heartc2 und heartc3 für die Pivot-Architekturen und bei horse3 für die Reinschicht-Pivotarchitekturen auf.

Einige der Stichproben wichen so stark von einer Normalverteilung ab, daß die Resultate des t-Tests als unsicher angesehen werden müssen. Wo diese Ergebnisse einen signifikanten Unterschied behaupteten, ist der entsprechende Eintrag in der obigen Tabelle eingeklammert. Das Resultat des Tests muß deshalb nicht unbedingt unzutreffend sein, aber zumindest ist das angegebene Signifikanzniveau ungenau. Für die Pivot-Architekturen traten nicht-normale Verteilungen auf bei building1, gene2 und gene3, für die Reinschicht-Pivotarchitekturen bei building1, gene2, heartac3 und soybean2. Aus nicht-normalen Stichproben wurden keine Ausreißer entfernt.

Diese Diskussion der Testmethodik verdeutlicht nochmals, wie wichtig es ist, bei der Anwendung statistischer Methoden auf Lernergebnisse neuronaler Netze sehr sorgfältig zu verfahren. Andernfalls können statistische Methoden Ergebnisse produzieren, die zwar sehr beeindruckend und wissenschaftlich aussehen, in Wirklichkeit aber unsicher oder sogar falsch sind, wie in Abschnitt 3.4 vorgeführt wurde.

Doch zurück zum eigentlichen Vergleich: In 10 der Fälle werden auf dem 90%-Konfidenzniveau (also Signifikanzniveau 0,1) keine signifikanten Unterschiede zwischen Pivot-Architekturen und Reinschicht-Pivotarchitekturen gefunden. In 9 Fällen ist die Pivotarchitektur signifikant besser, in 23 Fällen ist die Reinschicht-Pivotarchitektur besser. Diese Ergebnisse deuten an, daß die Suche nach noch besseren Netzarchitekturen für die betrachteten Probleme lohnend sein könnte, weil die verwendeten Architekturen alle durch Untersuchung von Netzen mit Direktverbindungen gewonnen wurden und das einfache Weglassen dieser Direktverbindungen vermutlich nicht der beste Weg ist, diese Architekturen zu verbessern. Wir bleiben dennoch auch im folgenden bei der Betrachtung der angegebenen Architekturen, weil die Methode, wie wir sie gefunden haben, plausibel ist und deshalb die unter diesen Voraussetzungen erzielten Resultate als praktisch relevant angesehen werden sollten.

Die obigen Resultate bieten eine Basis für Erforschung und Vergleiche von Lernverfahren auf den PROBEN1-Datensätzen. Es muß betont werden, daß zur Vereinfachung der Methodik bei keinem der obigen Resultate die Validationsmenge mit zum Training herangezogen wurde, wodurch sich sicherlich Verbesserungen erzielen ließen.

4.5 Zusammenfassung und Beiträge dieser Arbeit

In dieser Arbeit wurden erstmalig mehrere Klassen von Stoppkriterien für die Methode des frühen Stoppens genau definiert und zugleich einer empirischen Untersuchung ihrer Wirksamkeit unterworfen. Das Ergebnis der Untersuchung lautet, daß zwischen den verschiedenen Klassen von Kriterien nur moderate Unterschiede bestehen, andererseits aber die „Empfindlichkeit“ des Kriteriums, welche über einen Parameter eingestellt wird, spürbaren Einfluß auf seine Wirksamkeit hat: Ein weniger empfindliches Kriterium ermöglicht (in Maßen) verbesserte Lernergebnisse auf Kosten einer erheblich erhöhten Trainingsdauer. Es kann durch Verdreifachung der Trainingsdauer die Wahrscheinlichkeit dafür, das Minimum des Validationsfehlers zu finden von etwa 60% auf etwa 80% erhöht werden. Damit einher geht eine Absenkung des Testfehlers um im Mittel etwa 3%.

Ein weiterer Beitrag dieser Arbeit liegt in der Bereitstellung detaillierter Daten über das Verhalten von Lernverfahren mit frühem Stoppen auf einer Kollektion von 14 verschiedenen Lernproblemen; die Ergebnisse sind in [14] veröffentlicht. Diese Daten wurden unter genau definierten und reproduzierbaren Umständen produziert und können deshalb als Vergleichsbasis für Versuche mit anderen Lernverfahren dienen. Eine solche Basis fehlte bislang.

Kapitel 5

Automatisches Lernen II: Additive Verfahren

Wenn man bedenkt, daß das menschliche Gehirn
zu 80% aus Wasser besteht
— dafür geht's doch!
Erwin Grosche

One thing that connectionist networks
have in common with brains
is that if you open them up and peer inside
all you can see is a big pile of goo.
Michael Mozer und Paul Smolensky

In diesem Kapitel nehme ich eine Untersuchung einer vielversprechenden Klasse von additiven Lernverfahren vor. Die sechs untersuchten Verfahren basieren alle auf der Kandidatentrainingsidee von Fahlman (Abschnitt 5.1), die in zweifacher Hinsicht weiterentwickelt wird (Abschnitt 5.2). Eine umfangreiche empirische Untersuchung dient zum Vergleich der drei bekannten und drei neuen Verfahren (Abschnitt 5.3). Ziele des Kapitels sind die Beschreibung und Bewertung einer erweiterten Nutzung der Quervalidation in Kandidatenlernverfahren, der Vergleich konkurrierender bekannter Varianten von Kandidatenlernverfahren und die Bereitstellung empirischer Daten über solche Verfahren als Basis für einen Vergleich mit anderen Ansätzen.

5.1 Einleitung und verwandte Arbeiten

Wie wir im Abschnitt 2.8 gesehen haben, gibt es eine Reihe von Vorschlägen zu additiven oder additiv-subtraktiven Lernverfahren. Wir beschränken uns in dieser Arbeit auf universelle Verfahren, d.h. solche, die für mehrdimensionale Approximationsaufgaben geeignet sind — bei Klassifikationsaufgaben sind wir an einer guten Approximation der a-posteriori-Wahrscheinlichkeiten interessiert.

Die hinsichtlich ihrer Akzeptanz bei weitem erfolgreichste Arbeit auf diesem Gebiet ist das *Cascade Correlation* (*Cascor*) Lernverfahren von Fahlman und Lebiere [102]. Wir konzentrieren uns deshalb auf Varianten dieses Lernverfahrens. Cascade Correlation funktioniert folgendermaßen: Anfänglich trainieren wir mit Gradientenabstieg ein Netz ohne verborgene Knoten. Dann fügen wir in aufeinanderfolgenden Trainingsphasen je einen neuen verborgenen Knoten in das Netz ein. Dies geschieht in zwei Teilschritten: Erstens wird eine Gruppe von *Kandidatenknoten* erzeugt. Die Knoten werden verschieden initialisiert und unabhängig voneinander trainiert. Der beste Knoten wird ausgewählt und fest in das Netz eingefügt. Zweitens wird der Rest des Netzes an das Zusammenspiel mit dem neuen Knoten angepaßt. Hier ist eine genauere Präsentation in Pseudocode:

Cascade Correlation:
  Erzeuge Netz ohne verborgene Knoten;
  `REPEAT`
    `IF NOT` Ist erster Durchlauf
    `THEN` *Erzeuge und trainiere Kandidaten*;
      Füge besten Kandidaten ins Netz ein;
    `END`;
    Trainiere Ausgabeverbindungen;
  `UNTIL` *Ende*;

Unter *Ausgabeverbindungen* verstehen wir dabei genau diejenigen Verbindungen, die zu Ausgabeknoten führen. Bei Cascade Correlation werden also die zu einem verborgenen Knoten führenden Verbindungen nur einmal vor dem Einfügen des Knotens trainiert und danach unverändert gelassen. Ein Netz ohne verborgene Knoten, wie es zu Beginn eines CasCor-Laufes benutzt wird, ist in Abbildung 5.1 dargestellt.

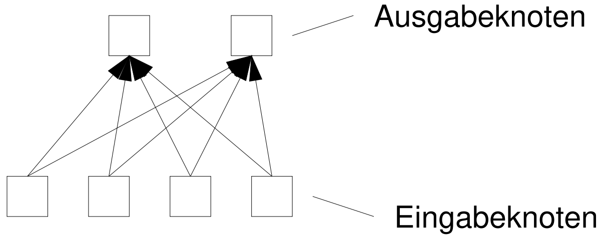


Abbildung 5.1: Anfängliches CasCor-Netz. Die Verbindungen mit dicken Spitzen werden jeweils trainiert.

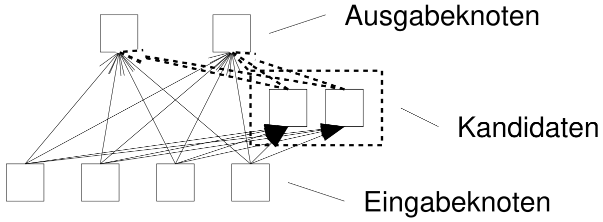


Abbildung 5.2: CasCor-Netz während der ersten Kandidatentrainingsphase. Die Verbindungen mit offenen Spitzen werden nicht trainiert.

Erzeuge und trainiere Kandidaten:
  Erzeuge k neue Knoten K_1 bis K_k;
  Verbinde ihre Eingänge mit den Ausgängen der Eingabeknoten
  und der schon existierenden verborgenen Knoten;
  `REPEAT`
    Mache Gradientenaufstiegsschritt für Kovarianz der Aktivierung jedes
      Kandidaten mit Ausgabeabweichung;
  `UNTIL` *Ende Kandidatentraining*;

Die Form des Netzes während der ersten Kandidatentrainingsphase ist in Abbildung 5.2 dargestellt; ein Netz während der dritten Kandidatentrainingsphase, also mit schon zwei verborgenen Knoten, in Abbildung 5.4.

Dem Gradientenaufstieg auf dem Kovarianzmaß liegt der Gedanke zugrunde, die Kandidaten darauf zu trainieren, immer gerade dann eine hohe Aktivierung zu haben, wenn das bisherige Netz an den Ausgabeknoten eine große Abweichung vom Sollwert produziert (mit linearer Fehlerfunktion). Die Kovarianz S für einen Kandidaten mit Aktivierung K_p und Ausgabeabweichung E_o am Ausgabeknoten o über die Menge der Trainingsbeispiele p in der von Fahlman und Lebiere gegebenen Formulierung [103] ist

$$S = \sum_o \left| \sum_p (K_p - \overline{K})(E_{p,o} - \overline{E_o}) \right|$$

Wobei $\overline{K}$ die mittlere Aktivierung des Kandidaten und $\overline{E_o}$ die mittlere Abweichung am Ausgabeknoten o sind. Die partielle Ableitung dieser Kovarianz bezüglich der Gewichte w_i, die vom Knoten i in den Kandidaten führen, lautet

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E_o}) f'_p \; out_{i,p}$$

wobei f'_p die partielle Ableitung der Aktivierungsfunktion des Kandidaten im Bezug auf die Summe seiner Eingaben ist und $out_{i,p}$ die Ausgabe von Knoten i auf Beispiel p. σ_o ist das Vorzeichen der Kovarianz mit Ausgabeknoten o, also die Ableitung der Betragsfunktion im obigen Kovarianzausdruck.

Das Terminierungskriterium *Ende Kandidatentraining* hat im originalen CasCor drei vom Benutzer einzustellende Parameter und lautet: *Maximalanzahl Kandidatentrainingsepochen erreicht oder Kandidatentraining stagniert*, wobei Stagnation bedeutet, daß sich die maximal von irgendeinem Kandidaten erreichte Kovarianz in den letzten k Epochen um weniger als einen bestimmten Prozentsatz erhöht hat. Dieses Kriterium ist jedoch zu simpel; ich werde deshalb in Abschnitt 5.2.1 ein besser funktionierendes herleiten.

Die Kovarianz, die ein Kandidat im Training erreicht, hängt auf unbekannte Weise von der Zufallsinitialisierung seiner Eingangsgewichte ab. Deshalb wird nicht nur ein Kandidat verwendet, sondern eine ganze Gruppe (typischerweise etwa 8 bis 16 Stück). Dies ist nur deshalb möglich, weil die Kandidaten während des Kovarianztrainings die Ausgaben des Netzes noch nicht beeinflussen, also auch voneinander unabhängig sind. Diese Möglichkeit ist einer der großen Vorzüge von CasCor. Nach Ende des Kandidatentrainings wird der beste Kandidat ins Netz eingefügt, die übrigen werden eliminiert:

Füge besten Kandidaten ins Netz ein:
  Bestimme Kandidaten mit höchster Kovarianz;
  Entferne alle anderen Kandidaten;
  Verbinde den verbleibenden Kandidaten mit den Ausgabeknoten;

Der beste Kandidat wird damit zu einem neuen verborgenen Knoten. Die resultierende Situation ist in Abbildung 5.3 dargestellt. Die Gewichte der neuen Verbindungen werden mit kleinen Werten

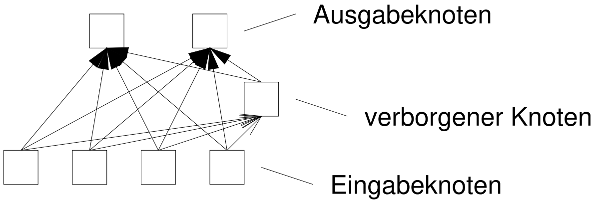


Abbildung 5.3: CasCor-Netz nach Ende der ersten Kandidatentrainingsphase.

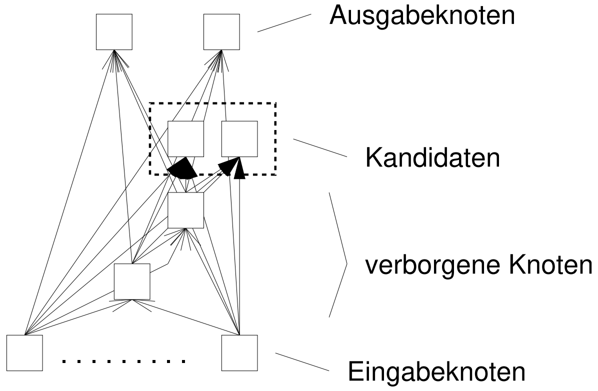


Abbildung 5.4: CasCor-Netz während der dritten Kandidatentrainingsphase. Zwei der Eingabeknoten sind nicht dargestellt.

initialisiert, deren Vorzeichen das umgekehrte Vorzeichen der Kovarianz bezüglich des jeweiligen Ausgabeknotens ist. Die Gewichte, die in den neuen verborgenen Knoten hineinführen, bleiben, anders als beispielsweise beim üblichen Backpropagation-Training, für den Rest des Trainings unverändert. Ein solcher verborgener Knoten ist quasi ein Merkmalsdetektor für ein Merkmal der Eingabedaten, das vor Einfügen des Knotens für eine Komponente des Restfehlers verantwortlich war. Diese Komponente des Restfehlers kann mit dem neuen verborgenen Knoten durch eine Anpassung der Ausgangsgewichte verkleinert werden.

Nach Einfügen jedes verborgenen Knotens (und vor Einfügen des ersten) werden die Ausgabeverbindungen trainiert. Als Ausgabeverbindungen bezeichnen wir genau diejenigen Verbindungen, die zu Ausgabeknoten führen.

Trainiere Ausgabeverbindungen:
`REPEAT`
  Mache Gradientenabstiegsschritt für Ausgabefehler bezüglich der Ausgabegewichte;
`UNTIL` *Ende Ausgabetraining*;

Diese Phase des Trainings funktioniert also genau wie das Training eines Netzes ohne verborgene Knoten; die verborgenen Knoten werden hierbei wie zusätzliche Eingabeknoten behandelt. Ein Rückverfolgen von Fehlern durch verborgene Knoten wie bei Backpropagation ist nicht notwendig.

Das Terminierungskriterium *Ende Ausgabetraining* lautet im originalen CasCor *Maximalanzahl Ausgabetrainingsepochen erreicht oder Ausgabetraining stagniert*, analog zum obenerwähnten Kriterium für das Kandidatentraining. Das Terminierungskriterium *Ende* für das gesamte Training lautet *Maximalanzahl von verborgenen Knoten erreicht oder letzter verborgener Knoten brachte zuwenig Fehlerverringerung oder Fehler klein genug*. In der originalen Formulierung von CasCor (und seiner verfügbaren Softwarefassung) sind die Parameter für diese Kriterien ausdrücklich für jedes Lernproblem passend vom Benutzer zu wählen, was für ein automatisches Lernverfahren nicht akzeptabel ist. Wir diskutieren die Abbruchkriterien deshalb im einem späteren Abschnitt getrennt.

Cascade Correlation hat zwei Hauptschwächen:

1. Die Kovarianz ist als Zielfunktion für die Kandidaten eigentlich das falsche Kriterium. Die Kandidaten werden bei der Maximierung der Kovarianz darauf trainiert, immer eine möglichst große Aktivierung (genauer: Abweichung von ihrer mittleren Aktivierung) zu haben, wenn der Fehler am Ausgang ungleich Null ist (genauer: ungleich dem mittleren Fehler). Das heißt, große Aktivierungen werden von der Zielfunktion zwar bei großen Fehlern stärker *belohnt* als bei kleinen, aber auch bei kleinen Fehlern führt eine möglichst große Aktivierung zu maximalem Kovarianzmaß und wird mithin antrainiert. Diese schlechte Unterscheidung zwischen großen und kleinen Restfehlern bewirkt, daß Cascade Correlation dazu neigt, kleine Fehler überzukompensieren. Aus diesem Grund ist das Verfahren für Approximationsaufgaben nicht gut geeignet; es funktioniert nur gut, wenn eine reine Klassifikationsentscheidung verlangt wird.
2. Durch die Kaskadierung entsteht ein Netz, das sehr starke Nichtlinearitäten realisieren kann. Obwohl diese Eigenschaft im Prinzip vorteilhaft ist, kann sie sich zum Nachteil verkehren, wenn starke Nichtlinearität zum Lösen des Problems gar nicht nötig ist und die vorhandene nicht durch genügend viele Trainingsbeispiele im Zaume gehalten wird [332].

Zur Beseitigung des ersten Problems kann man die Lernregel ändern und die Kandidaten anstatt auf hohe Kovarianz direkt auf Minimierung des Ausgabefehlers trainieren. Den Fehlerminimierungsansatz für das Kandidatentraining benutzt das noch nicht publizierte *Cascade2*-Verfahren von Fahlman und in etwas anderer Form das *CasEr*-Verfahren von Littmann und Ritter [223]. Die folgende Form des Verfahrens entspricht ungefähr Cascade2 (Abweichung siehe unten).

Zur direkten Fehlerminimierung müssen *virtuelle Ausgabeverbindungen* für die Kandidaten eingerichtet werden. Diese Verbindungen transportieren zwar auf dem Vorwärtsdurchlauf keine Aktivierung zu den Ausgabeknoten, bekommen aber im Rückwärtsdurchlauf ein Fehlersignal. Dieses wird um die gedachte Beteiligung der Verbindung korrigiert und dann wie in normaler Backpropagation verwendet, wobei aber nur die Kandidatenknoten trainiert werden und der Rest des Netzes fixiert bleibt. Eine virtuelle Verbindung von einem Kandidaten i zu einem Ausgabeknoten j berechnet also ihren Gradientenbeitrag

Δw_{ij} nicht wie eine normale Verbindung als

$$\Delta w_{ij} = \underbrace{\frac{\partial E(y_j - f(in_j))}{\partial f(in_j)} \frac{\partial f(in_j)}{\partial in_j}}_{=:\delta_j} out_i$$

(wobei in_j die summierte Eingabe in den Knoten sei, E die Fehlerfunktion und f die Aktivierungsfunktion von j), sondern als

$$\Delta w_{ij} = \frac{\partial E(y_j - f(in_j + w_{ij}out_i))}{\partial f(in_j + w_{ij}out_i)} \frac{\partial f(in_j + w_{ij}out_i)}{\partial(in_j + w_{ij}out_i)} out_i$$

Diese Berechnung vereinfacht sich, falls f die Identität ist (lineare Ausgabefunktion) und E die quadratische Fehlerfunktion. Dann kann die Korrektur lokal in der Verbindung vorgenommen werden, weil der Beitrag der Verbindung linear in den zurückzupropagierenden Deltaterm eingeht, welcher vor der Korrektur als δ_j routinemäßig zur Verfügung steht. In diesem Fall lautet die Berechnung

$$\Delta w_{ij} = \underbrace{(\delta_j - w_{ij}\, out_i)}_{=:\hat{\delta_j}} out_i$$

Nach der analogen Korrektur des durch die Verbindung weiterpropagierten Fehlersignals wird der Fehler für die in den Kandidatenknoten hineinführenden Verbindungen wie bei ganz normaler Backpropagation behandelt. Ist die Vereinfachung nicht möglich, müssen der Verbindung sowohl E und f als auch der Wert von in_j bekannt sein, was dem Lokalitätsgedanken von neuronalen Netzen widerspricht. Wir gehen im folgenden immer von der quadratischen Fehlerfunktion aus.

Die *Güte eines Kandidaten* kann ausgedrückt werden durch das Verhältnis von Gesamtfehler des Netzes zum Fehler nach Einbeziehung des Kandidaten, wobei für wirkungslose Kandidaten die Güte auf Null normiert und ansonsten in Prozent gemessen wird, also

$$G = 100\left(\frac{E_{netz}}{E_{kand}} - 1\right) = 100\left(\frac{\sum_{j,p}(y_j(p) - o_j(p))^2}{\sum_{j,p}\hat{\delta_j}(p)} - 1\right)$$

wobei die Güte G sich ergibt aus der Summe E_{netz} der quadratischen Fehler des Netzes ohne den Kandidaten über alle Ausgaben j und Beispiele p und der Summe E_{kand} der Quadrate der korrigierten Fehlersignale $\hat{\delta_j}$. Dieser Gütewert kann negativ sein, wenn nämlich der Kandidat zusätzlichen Fehler verursacht. Dies geschieht oft zu Beginn des Kandidatentrainings und auch für die Fehler auf der Validationsmenge. Ein solcher Knoten mit negativer Güte auf der Validationsmenge kann trotzdem eine Verbesserung des Netzes sein, weil sich oft beim Ausgabetraining doch noch ein positiver Gesamteffekt einstellt.

Im Unterschied zur obigen Darstellung benutzt Cascade2 bei Klassifikationsaufgaben in der Regel nichtlineare Ausgabeaktivierungsfunktionen und arbeitet trotzdem mit der vereinfachten Korrektur. Das gleiche geschieht im CasEr-Verfahren ($tanh$-Ausgangsaktivierungsfunktion), wobei dort als zusätzliche Einschränkung die Ausgabegewichte konstant als 1 angenommen, also nicht mittrainiert werden. Die CasEr-Publikation [223] berichtet mit Ausnahme eines fehlgeschlagenen Versuchs, die Mackey-Glass Zeitreihe zu lernen, nur über Klassifikationsprobleme und kommt zu dem Ergebnis, daß CasCor besser funktioniert als CasEr. Ob dieses Ergebnis auch bei einer vollständigen Korrektur des Fehlersignals gültig bleibt, ist unklar. Für Approximationsprobleme (auch zur bei uns gewünschten Approximation der a-posteriori-Wahrscheinlichkeiten bei Klassifikationsaufgaben) ist aber die Kovarianzlernregel offensichtlich falsch und sollte durch direkte Fehlerminimierung ersetzt werden. Dies wird auch in der Statistik so gemacht, bei der schon oben erwähnten, wesensverwandten *backfitting*-Prozedur.

Bezüglich des zweiten Problems weist Sjøgaard [332] (an leider nur einem einzigen künstlichen Beispiel) nach, daß mit CasCor erzeugte Netze schlechter generalisieren, als wenn man bei ansonsten gleichem Vorgehen die verborgenen Knoten nicht zu einer Kaskade stapelt, sondern in einer einzigen verborgenen Schicht nebeneinandersetzt. Yeung [413] kommt unabhängig von Sjøgaard auf die gleiche Idee und stellt an mehreren Problemen fest, daß sich fast kein Unterschied zu CasCor ergibt. Fahlman selber meint, daß die Kaskade für manche Probleme wichtig ist und für andere in der Regel zumindest keinen Schaden anrichtet. Offenbar sind zusätzliche empirische Daten nötig, um diese Frage entscheiden zu können.

Ich werde in diesem Kapitel eine Untersuchung vorstellen, die sechs Varianten des Cascade Correlation Verfahrens vergleicht. Da wir in dieser Arbeit immer an Approximationsproblemen oder der Approximation der a-posteriori Wahrscheinlichkeiten bei Klassifikationsproblemen interessiert sind, wird die Fehlerminimierungslernregel anstelle der Kovarianzlernregel verwendet.

5.2 Sechs Lernverfahren mit Kandidatentraining

5.2.1 Abbruchkriterien

Alle Kandidatentrainingsverfahren reagieren sensibel auf Änderungen der Abbruchbedingungen für die Kandidaten- und Ausgabetrainingsphase. Ein zu kurzes Training läßt ein Netz mit unzureichend aufeinander abgestimmten Einzelteilen entstehen, das schlechte Generalisierung hat. Zu langes Training kostet erstens viel Zeit und birgt zweitens die Gefahr der Überanpassung. In den bisher publizierten Kandidatentrainingsverfahren sind nirgendwo Maßnahmen zur Verhinderung der Überanpassung vorgesehen. Deshalb müssen die Abbruchkriterien entsprechend erweitert werden, wir verwenden dabei unter anderem wieder die Technik des frühen Stoppens, die hier nur auf eine Trainings*phase* angewendet wird. Zur Diskussion der Abbruchbedingungen benötigen wir zunächst einige Definitionen.

Wir betrachten für die Abbruchkriterien immer den Kandidaten mit der in Epoche t gerade jeweils besten Güte $\hat{G}(t)$ und definieren den *Fortschritt des Kandidatentrainings* für einen Trainingsstreifen der Länge k, gemessen in Promille, als

$$P_k(t) = 10 \left(\max_{t' \in t-k+1 \ldots t} (\hat{G}(t)) - \frac{1}{k} \sum_{t' \in t-k+1 \ldots t} \hat{G}(t) \right)$$

in Anlehnung an die Definition des Trainingsfortschritts aus Abschnitt 4.2. Die Definition muß ein subtraktives Maß anstatt eines Verhältnismaßes verwenden, weil ein Nulldurchgang von $\hat{G}(t)$ möglich ist. Die Güte ist hierbei immer die Güte auf der Trainingsmenge, analog gibt es die Güte $\hat{G_{va}}(t)$ auf der Validationsmenge. In Anlehnung an den Generalisierungsverlust GL definieren wir den *Güteverlust* VL_{va} auf der Validationsmenge als

$$VL_{va}(t) = 100 \frac{\max_{t' \le t}(\hat{G}_{va}(t')) - \hat{G}_{va}(t)}{\max\left(\left| \max_{t' \le t} \hat{G}_{va}(t') \right|, 1 \right)}$$

Dieses Maß normalisiert die Differenz von Gütewerten mit deren Betrag, außer wenn sie im Bereich $-1 \ldots 1$ liegen, denn bei 0 gäbe es sonst eine Singularität. Im Gegensatz zum Generalisierungsverlust interessiert uns der Güteverlust auch auf der Trainingsmenge, wir notieren dies als VL_{tr} mit Definition analog zu VL_{va} oben.

Mit Hilfe dieser Definitionen können wir nun die Überlegungen zu Abbruchkriterien darstellen. Da das Ausgabetraining weniger kritisch ist, beginnen wir mit dem Kandidatentraining.

Erstens: Die Stagnationsbedingung beim Kandidatentraining ist sehr instabil. Schon bei CasCor kann es vorkommen, daß ein Kandidat über 10 Epochen kaum eine Erhöhung der Kovarianz zeigt und dann wieder kräftig weiter ansteigt. Dieses Verhalten ist beim Kandidatentraining mit Fehlerminimierung noch verstärkt. Dabei ist überhaupt die Entwicklung der Fehler sehr turbulent; auch kräftige Verschlechterungen kommen zwischendurch immer wieder vor. Deshalb darf ein Abbruchkriterium nicht zu schnell auf schwache Fortschritte reagieren. Wir definieren die *letzte Fortschrittsepoche* als die Epoche t, für die $P_5(t)$ zuletzt über 0,5 Promille lag, und brechen erst dann wegen Stagnation ab, wenn t volle 40 Epochen zurückliegt. Fahlman verwendet ein ähnliches Kriterium, geht aber nur 12 Epochen zurück. Meine Versuche ergaben, daß dieser Wert für reale Datensätze häufig zu riskant ist.

Zweitens: Auch beim Kandidatentraining tritt Überanpassung auf wie an einem dramatischen Beispiel in Abbildung 5.5 dargestellt ist. Fahlman erzielte in seinen bisherigen Versuchen mit Cascade2

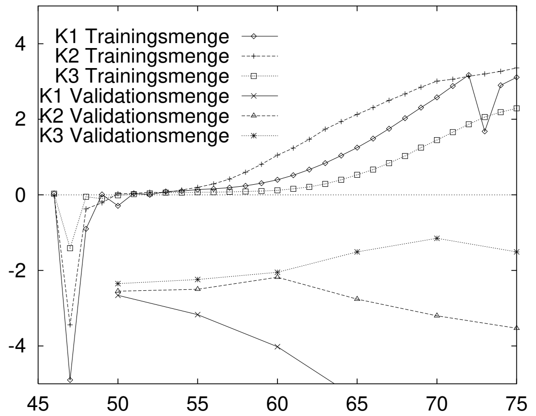


Abbildung 5.5: Gütewerte von drei Kandidaten K1, K2, K3 auf der Trainingsmenge und der Validationsmenge während des ersten Kandidatentrainings von glass1 mit dem Cascade-Lernverfahren. Es tritt eine starke Überanpassung auf.

teilweise schlechtere Ergebnisse als mit CasCor und schiebt dies darauf, daß er nie Maßnahmen gegen Überanpassung ergriffen hat; CasCor ist im Vergleich wenig empfindlich für Überanpassung. Dieser Erklärung stimme ich zu. Im Gegensatz zum Training mit frühem Stoppen für komplette Netze, darf man wegen der sehr turbulenten Fehlerentwicklung beim Kandidatentraining allerdings kein zu sensibles Kriterium für das Stoppen wegen Überanpassung verwenden. Wir stoppen, wenn $VL_{va}(t) > 25$, allerdings frühestens nach 25 Epochen und nur, wenn zugleich $VL_{tr}(t) = 0$, d.h. wenn sich die Güte auf der Trainingsmenge nicht verschlechtert hat. Nach dem Stoppen werden die Gewichte für jeden Kandidaten K zum Punkt $\hat{t}_K$ seiner höchsten Validationsgüte $G_{va}(\hat{t}_K)$ zurückgesetzt. Der zur Auswahl des besten Kandidaten herangezogene Gütewert ist dann logischerweise die diesem Punkt zugehörige Trainingsgüte $G_{tr}(\hat{t}_K)$.

Drittens: Das Kandidatentraining dauert bisweilen auch mit den beiden obengenannten Abbruchkriterien sehr lange. Da mehrere lange Kandidatentrainingsphasen insgesamt zu einer unerträglich langen Trainingszeit führen würden, geben wir eine Maximalzahl von Epochen für jede Kandidatentrainingsphase vor. Der Wert darf jedoch nicht zu klein sein, weil das Netz sonst mangelhaft abgestimmte verborgene Knoten bekommt, die der Gesamtleistung Schaden zufügen. Ein Wert von 150 Epochen scheint ein brauchbarer Kompromiß zu sein; möglicherweise lassen sich mit höheren Werten manchmal bessere Ergebnisse erzielen.

Diese drei Teile ergeben insgesamt folgendes Abbruchkriterium für das Kandidatentraining:

Ende Kandidatentraining:
Letzte Fortschrittsepoche ist 40 Epochen zurück `OR`
($VL_{va}(t) > 25$ nach mindestens 25 Epochen `AND` $VL_{tr}(t) = 0$) `OR`
Kandidatentraining läuft schon 150 Epochen lang.

Für die Diskussion des Abbruchkriteriums beim Ausgabetraining ist vor allem die Frage interessant, wie man es gegen den Abbruch des gesamten Trainings abgrenzt. In jedem Fall sollte das Ausgabetraining ebenso wie das Kandidatentraining eine gewisse Mindestdauer haben; ich wähle hierfür wiederum 25 Epochen. Außerdem kommen für beide Kriterien dieselben Merkmale in Frage, wie beim frühen Stoppen für komplette Netze: zu hoher Generalisierungsverlust, zu langsamer Trainingsfortschritt und Erreichen einer Maximalzahl von Epochen. Meine Versuche ergaben folgendes als eine brauchbare Wahl:

Ende Ausgabetraining:
Mindestens 25 Epochen in diesem Ausgabetraining trainiert `AND`
(Insgesamt mehr als 5000 Epochen trainiert `OR`
Generalisierungsverlust $GL(t) > 2$ `OR` Trainingsfortschritt $P_5(t) < 0,4$)

Das gesamte Training wird dann beendet, wenn entweder der Generalisierungsverlust erheblich ist, oder das Einfügen des letzten verborgenen Knotens keinen deutlichen Fortschritt mehr erbracht hat:

Ende:
Insgesamt mehr als 5000 Epochen trainiert `OR`
Generalisierungsverlust $GL(t) > 5$ `OR`
($P_k(t) < 0,1$ `AND` *Wenig Fortschritt durch letzten neuen Knoten*)

Wenig Fortschritt durch letzten neuen Knoten:
Verringerung des Trainingsfehlers durch letzten neuen Knoten um weniger als 0,1% `AND`
Verschlechterung des Validationsfehlers durch letzten neuen Knoten

Das Unbefriedigende an all diesen Kriterien ist, daß sie nicht hergeleitet werden können, sondern heuristisch gewählt werden. Es ist aber nicht zu sehen, wie diesem Umstand abgeholfen werden soll. Zur Herleitung aller Kriterien wurden nur die Verläufe der Trainings- und Validationsfehler herangezogen, nicht jedoch die Testfehler.

5.2.2 Lernverfahren

In diesem Abschnitt stelle ich die in der nachfolgend beschriebenen Untersuchung verglichenen Lernverfahren dar. Allen Verfahren ist gemeinsam, daß sie die lineare Aktivierungsfunktion in den Ausgabeknoten benutzen, mit der quadratischen Fehlerfunktion trainiert werden, als Lernregel RPROP eingesetzt wird und die oben beschriebenen Abbruchkriterien zur Anwendung kommen.

Die ersten drei Verfahren sind bereits von oben bekannt:

Cascor ist die Mutter der ganzen Klasse.

Cascade entspricht ungefähr dem momentan von Fahlman untersuchten Cascade2. Es ist also CasCor mit der Fehlerminimierungslernregel anstelle der Kovarianzlernregel.

Cand ist das gleiche ohne die Kaskadierung. Die verborgenen Knoten werden also, wie von Sjøgaard und Yeung vorgeschlagen, nebeneinander in eine verborgene Schicht gesetzt und empfangen nur Verbindungen von den Eingabeknoten.

Die übrigen drei Verfahren schlage ich aus folgender Überlegung heraus vor: Wenn wir schon eine Reihe von Kandidaten mit unterschiedlicher Güte auf der Trainingsmenge und der Validationsmenge produziert haben, warum sollen wir dann zur Auswahl eines Kandidaten ausschließlich die Güte auf der Trainingsmenge heranziehen? Benutzt man zusätzlich die Validationsgüte, so kann man vermutlich oft einen Kandidaten mit besserer Generalisierungsleistung auswählen. Definieren wir also den Begriff des besten Kandidaten neu und verwenden als Gesamtgütemaß eine Kombination aus G_{tr} und G_{va}.

Diese Idee hat zwei mögliche Schwächen:

1. Wird die Trainingsgüte zuwenig berücksichtigt, so verwirrt sich das Training gewissermaßen selbst, weil der ausgewählte Kandidat zuwenig an die Trainingsmenge angepaßt ist und somit im weiteren Verlauf des Trainings störend wirkt.

2. Die Verwendung der Validationsgüte zur Kandidatenauswahl erzeugt ein „Leck", durch das in kleinen Mengen Information aus der Validationsmenge in das Trainingsverfahren einfließt. Dies kann die Abbruchkriterien verwirren, weil sie davon ausgehen, daß die Ergebnisse auf der Validationsmenge nur mittelbar vom Training beeinflußt werden.

Wegen dieser zwei Probleme ist nicht im Vorhinein zu sagen, ob der grundsätzliche Vorteil der Berücksichtigung der Validationsgüte bei der Kandidatenauswahl sich tatsächlich in besseren Netzen niederschlägt. Um dies herauszufinden, verwende ich in der Untersuchung drei Lernverfahren aus einer Klasse von Kandidatenlernverfahren mit Benutzung der Validationsgüte. Die Klasse heißt *Kogi* („Kombination guter Ideen"), die drei gewählten Vertreter sind *kogi2*, *kogi3* und *kogi9*. Die Bezeichnungen haben historische Gründe und werden aus Sentimentalität beibehalten.

Kogi2 entspricht cascade, bis auf die Verwendung der Validationsgüte zur Kandidatenauswahl. Der anstelle der Trainingsgüte G_{tr} verwendete Gütewert G_{kogi} berechnet sich zu $1/3(G_{va} + 2G_{tr})$. Dieser Wert wird auch in den anderen Kogi-Verfahren benutzt. Die Kombination der Gütewerte im Verhältnis eins zu zwei wurde in Vorversuchen als brauchbarer Kompromiß zwischen Robustheit durch Betonung der Trainingsgüte und möglicher Verbesserung durch Berücksichtigung der Validationsgüte herausgefunden.

Kogi3 entspricht cand bis auf zwei Änderungen: Erstens die Verwendung der Validationsgüte zur Kandidatenauswahl und zweitens die Verwendung unterschiedlicher Aktivierungsfunktionen in den Kandidaten. Je gleichviele Kandidaten verwenden die Aktivierungsfunktion $x/(1+|x|)$ oder $1/(1+e^{-x})$ (Standard-Sigmoid) oder $e^{-x^2/2}$ (Gaußkurve). Diese Idee wurde schon von Fahlman in der Präsentation von CasCor vorgeschlagen, aber nicht ausprobiert [103]. Sjøgaard [332] findet für sein Anwendungsbeispiel, daß die Idee für das flache Netz mit nur einer verborgenen Schicht ein schnelleres Training mit gleich guten Ergebnissen bewirkt, bei CasCor jedoch eine Verschlechterung der Generalisierung nach sich zieht. Deshalb setze ich unterschiedliche Aktivierungsfunktionen bei Kaskadennetzen nicht ein.

Kogi9 ist ein Versuch, die möglichen Vorteile von kogi2 (mächtiges Netz wegen Kaskade) und kogi3 (weniger Neigung zur Überanpassung) miteinander zu verbinden. Der Versuch basiert auf der Beobachtung, daß in Veröffentlichungen zu statischen Netzen nur selten mehr als zwei verborgene Schichten eingesetzt werden. Bei kogi9 wird ein solches Netz erzeugt, indem mit zwei Gruppen von Kandidaten gearbeitet wird: Die erste Gruppe versorgt die erste verborgene Schicht; ihre Kandidaten erhalten Eingaben nur von den Eingabeknoten. Die zweite Gruppe versorgt die zweite verborgene Schicht; ihre Kandidaten erhalten Eingaben von den Eingabeknoten und allen zuvor schon eingefügten Knoten der ersten verborgenen Schicht. Es konkurrieren alle Kandidaten beider Gruppen miteinander darum, ausgewählt zu werden. Würde nur die Trainingsgüte zur Auswahl herangezogen, wäre zu erwarten, daß die Kandidaten der zweiten Gruppe fast immer den ausgewählten Knoten stellen, weil sie mächtigere Eingabeinformationen zur Verfügung haben. Dieser Vorteil kann jedoch auch zur Überanpassung führen, weshalb bei Heranziehen des gemischten Gütekriteriums, wie es in den Kogi-Verfahren benutzt wird, die Auswahl automatisch nach den tatsächlichen Gegebenheiten des Lernproblems angepaßt wird. Damit wird wenigstens ein Teil der Mächtigkeit einer Kaskadierung ausgenutzt und zugleich ihr potentieller Nachteil vermieden. Je nach Wahl der Größe der Kandidatengruppen verursacht ein Kandidatentraining bei kogi9 einen höheren Aufwand als bei den anderen Verfahren. Kogi9 verwendet wie kogi3 unterschiedliche Aktivierungsfunktionen.

5.3 Versuchsaufbau und Ergebnisse

Zum Vergleich der oben beschriebenen Kandidatenlernverfahren wurden umfangreiche Versuchsreihen durchgeführt, wiederum unter Verwendung der Probleme aus der PROBEN1-Benchmarksammlung. Diese Versuchsreihen fanden in der Zeit vom 7. Juli bis zum 20. November 1994 auf der MasPar MP-1216A des Rechenzentrums der Universität Karlsruhe, der KSR1 (32 Prozessoren) des Rechenzentrums der Universität Mannheim und 6 Sun Workstations der Typen SparcStation 10 und SparcClassic am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe statt. Die verwendeten Programme sind in der im zweiten Teil dieser Arbeit vorgestellten Programmiersprache CuPit geschrieben und wurden mit Hilfe der ebenfalls im zweiten Teil vorgestellten Übersetzer in parallelen (für MasPar) bzw. sequentiellen Code (für KSR und Sun) übersetzt.

Die Implementierung der Verfahren spart viele der Netzdurchläufe durch eine *Pufferung* teilweise ein: Während des Trainings der Ausgabeverbindungen hat jeder verborgene Knoten K für jedes Beispiel i immer den gleichen Ausgabewert $A_{K,i}$. Statt diese Ausgabewerte auf normale Weise durch Verfolgen der Verbindungen aus den Eingabedaten zu berechnen, werden sie nach ihrer ersten Berechnung in einem Puffer abgespeichert und bei jeder folgenden Benutzung direkt wieder in die Knoten geladen. Das gleiche geschieht während des Trainings der Kandidatenknoten mit den Ausgabewerten der Ausgangsknoten. Erst diese Pufferung erlaubt eine effiziente parallele Implementation der Kaskade, deren Durchlaufen ansonsten nur einen geringen Parallelitätsgrad aufweist.

Aufgrund historischer Umstände und diverser technischer Schwierigkeiten sind die Versuchsreihen nicht ganz balanciert. So wurden Programmläufe, die z.B. aufgrund von Rechnerneustarts oder -ausfällen vorzeitig abgebrochen wurden, nicht in allen Fällen wiederholt, und die Probleme heartc und heartac waren nicht in allen Teilserien enthalten. Pro Lernverfahren und Datensatz liegen daher die Daten von zwischen 12 und 47 Läufen vor. Insgesamt stehen die Daten von 8524 Läufen zur Verfügung; im Mittel also 34 Läufe pro Datensatz und Verfahren.

Dabei wurden dieselben Parameter für das zugrundeliegende RPROP-Verfahren benutzt wie in Abschnitt 3.2.17 und in Abschnitt 4.4.2. Die Parameter für die Steuerung des Kandidatenlernens waren wie oben angegeben.

Die in den Versuchsreihen gesammelten Daten wurden mittels statistischer Signifikanzprüfung auf Unterschiede im Mittelwert der erzielten quadratischen Fehler auf der Testmenge untersucht. Damit sollen die folgenden Fragen beantwortet werden:

1. Ist cand besser als cascade, wie es Sjøgaard behauptet, oder verhält es sich eher umgekehrt, wie es Fahlman behauptet?
2. Ist cascade besser als cascor, wie die Theorie erwarten läßt?
3. Ist kogi3 besser als cand?
4. Ist kogi2 besser als cascade?
5. Wie verhält sich kogi9 im Vergleich zu kogi2 und kogi3?

Ich vergleiche hier zunächst nur die Kandidatenlernverfahren miteinander. Ein Vergleich mit Beschneidungsverfahren und mit dem Lernen mit frühem Stoppen erfolgt im Kapitel 7.

Um den Leser nicht mit Zahlenfriedhöfen zu ermüden, verzichte ich auf die Wiedergabe der Tabellen mit den Einzelergebnissen[1] und gehe direkt zur Darstellung der Ergebnisse des statistischen Vergleichs über. Wie in Abschnitt 4.4.2 wurden die logarithmierten Testfehler zugrundegelegt, zum Vergleich ein t-Test benutzt, dabei die Cochran/Cox-Approximation für ungleiche Varianzen verwendet und Ausreißer aus den Stichproben entfernt. Auch in diesen Vergleichen gibt es ferner einige nicht normalverteilte Stichproben. Tabelle 5.6 gibt die Gesamtanzahl von Ausreißern in den Stichproben für die einzelnen Lernverfahren an, sowie die Anzahl von Stichproben, die nicht normalverteilt waren. Nun zu

[1] Die Rohdaten sämtlicher einzelner Programmläufe sind jedoch elektronisch verfügbar; siehe Anhang A.

Tabelle 5.6: Ausreißer und nicht-normalverteilte Stichproben

Verfahren	Ausreißer nach unten	oben	Nicht normalverteilte Stichproben
cand	7 (0,5%)	28 (2,0%)	card3, soybean2, thyroid2 building1, building3
cascade	11 (0,8%)	28 (2,0%)	cancer2, gene3, thyroid2, thyroid3 building1, flare1
cascor	8 (0,5%)	41 (2,5%)	diabetes2, gene1, gene2, gene3, heart2, thyroid2 heartac1
kogi2	9 (0,6%)	25 (1,7%)	cancer2, soybean2, thyroid2, thyroid3 building1, building3, hearta2
kogi3	7 (0,5%)	48 (3,7%)	cancer1, card2, soybean2, thyroid2 building1, building3, flare2, hearta2, hearta3
kogi9	4 (0,3%)	52 (3,8%)	cancer1, cancer3, card3, soybean2 building3, flare1, hearta2

den Ergebnissen. Tabelle 5.7 zeigt den Vergleich von cand mit cascade und den von cascade mit cascor. Die erste Frage kann anhand dieser Ergebnisse sofort beantwortet werden: Für die hier betrachteten Probleme macht es meist keinen Unterschied, ob man cand oder cascade verwendet. Einzig auf dem building- und dem thyroid-Problem scheint jeweils cand besser zu sein, in allen anderen Fällen sind die Unterschiede, wenn überhaupt vorhanden, gering.

Die zweite Frage muß differenziert werden: cascade ist tatsächlich besser als cascor, wenn es um die Approximationsaufgaben geht (building, flare, hearta, heartac). Bei den Klassifikationsaufgaben hingegen schneidet cascor geringfügig besser ab als cascade — und das, obwohl wir nicht die Klassifikationsfehler sondern die quadratischen Fehler betrachten. Die Erklärung für dieses Phänomen liegt wahrscheinlich in der schnelleren Konvergenz der Kandidatenknoten beim Kovarianztraining gegenüber dem Training mit quadratischem Fehler. Denn obwohl im Versuchsaufbau bis zu 150 Epochen zum Training der Kandidaten vorgesehen waren, wird das Kandidatentraining oft früher abgebrochen als es angemessen wäre. Da die Kandidaten sich bei cascor schneller entwickeln, wird cascade von dieser Beschränkung stärker getroffen. Vermutlich würden die Unterschiede also bei längeren Kandidatentrainingsphasen dahinschmelzen. Eine zweite Vermutung lautet, daß das Kovarianztraining weniger Neigung hat, eine Überanpassung zu entwickeln und deshalb auch dort länger durchgeführt werden kann, wo die Obergrenze von 150 Epochen noch nicht relevant ist; diese Frage wurde jedoch nicht weiter verfolgt. In Tabelle 5.8 sind als weiterer Vergleich cand und cascor gegenübergestellt.Wie wir sehen, ist der Vorsprung, den cascor vor cascade hat, nicht größer als der geringe Vorsprung von cand vor cascade. Dennoch ist es eine interessante Erkenntnis, daß das theoretisch ungünstige Trainingsverfahren von cascor offenbar für zahlreiche Anwendungen tatsächlich (kleine) Vorteile gegenüber dem „richtigen“ aufweist.

Es ist interessant, die Netzarchitekturen zu betrachten, die von cand und cascade aufgebaut werden. Dazu sind in den Tabellen 5.9 (cand) und 5.10 (cascade) die Anzahlen von verborgenen Knoten angegeben, die diese Lernverfahren in den Versuchsserien im Mittel, sowie minimal und maximal in die Netze eingebaut haben. Mehrere Beobachtungen drängen sich auf: Erstens baut cascade nicht immer weniger Knoten ein als cand, obwohl dies aufgrund der höheren Abbildungsmächtigkeit der kaskadierten Knoten und aufgrund der höheren Anzahl von Verbindungen bei gleicher Knotenanzahl zu erwarten wäre. Dies deutet darauf hin, daß cascade nicht in der Lage ist, seine Ressourcen optimal einzusetzen. Zweitens tritt bei einigen Problemen eine enorme Varianz der Knotenzahlen über verschiedene Läufe auf. Auch dies ist ein Anzeichen für mangelnde Fähigkeit der Verfahren, gezielt das „richtige“ Netz zusammenzubauen. Drittens erzeugen die Verfahren in manchen Fällen z.B. bei building, cancer,

Tabelle 5.7: Vergleich von cand und cascade sowie cascade und cascor mittels t-Test

cand vs. cascade			
Problem	1	2	3
building	—	c 0,0	(c 0,0)
cancer	—	(c 0,2)	—
card	—	—	(c 9,8)
diabetes	—	C 2,6	C 2,2
flare	—	—	—
gene	—	—	(C 2,5)
glass	c 0,1	—	c 0,0
heart	—	C 2,1	—
hearta	—	—	—
heartac	—	—	—
heartc	—	—	—
horse	—	—	C 1,3
soybean	c 0,0	(C 8,2)	—
thyroid	c 0,0	(c 8,3)	(c 0,0)

cascade vs. cascor			
Problem	1	2	3
building	—	R 1,9	C 9,7
cancer	R 0,0	(R 3,5)	R 0,0
card	—	—	—
diabetes	C 0,0	(C 0,0)	—
flare	—	C 0,0	C 0,7
gene	—	—	—
glass	R 0,0	—	R 0,0
heart	—	—	—
hearta	R 4,2	C 1,4	—
heartac	—	—	—
heartc	—	—	—
horse	—	—	—
soybean	R 0,9	R 0,0	—
thyroid	R 0,9	(R 0,0)	(R 0,7)

Der Aufbau der Tabellen ist analog zu Tabelle 4.12 von Seite 87. Verglichen wurden die Mittelwerte der logarithmierten Testfehler der Läufe von cand (c) und cascade (C) in der linken Tabelle, sowie von cascade (C) und cascor (R) in der rechten Tabelle für die Varianten 1, 2, und 3 jedes Problems. Die Einträge zeigen Unterschiede, die auf einem 10%-Niveau signifikant sind und den zugehörigen p-Wert (in Prozent). Der Buchstabe gibt jeweils an, welche Architektur signifikant besser ist; ein Querstrich bedeutet, daß keine signifikanten Unterschiede vorliegen. Eingeklammerte Resultate sind dubios, weil mindestens eine der beteiligten Stichproben von logarithmierten Testfehlern nicht normalverteilt ist. Es wurde ein t-Test mit der Cochran/Cox-Approximation für den Fall ungleicher Varianzen verwendet.
Links (cand/cascade): 26 mal kein signifikanter Unterschied, 10 mal cand besser („c“, davon 4 dubios), 6 mal cascade besser („C“, davon 2 dubios).
Rechts (cascade/cascor): 24 mal kein signifikanter Unterschied, 12 mal cascor besser („R“, davon 3 dubios), 6 mal cascade besser („C“, davon 1 dubios). Bei Approximationsproblemen: 6 mal kein signifikanter Unterschied, 4 mal cascade besser, 2 mal cascor besser.

cand vs. cascade			
Problem	1	2	3
building	R 0,7	c 0,0	c 0,0
cancer	R 0,0	c 7,5	R 0,0
card	c 4,8	—	—
diabetes	c 0,0	(c 0,2)	R 2,0
flare	—	c 0,0	c 9,9
gene	—	—	(R 0,3)
glass	R 3,6	—	—
heart	R 9,5	(R 0,1)	c 1,1
hearta	R 7,0	c 6,3	—
heartac	—	—	—
heartc	—	R 1,8	—
horse	—	—	—
soybean	—	(R 2,1)	—
thyroid	c 0,0	—	c 0,0

Tabelle 5.8: Vergleich von cand und cascor mittels t-Test. 19 mal kein signifikanter Unterschied, 12 mal cand besser („c“, davon 1 dubios), 11 mal cascor besser („R“, davon 3 dubios). Bei Approximationsproblemen: 5 mal kein signifikanter Unterschied, 5 mal cand besser, 2 mal cascor besser.

Tabelle 5.9: Anzahl von cand erzeugter verborgener Knoten

	Knoten f. Version 1			Knoten f. Version 2			Knoten f. Version 3		
Problem	μ	σ	min-max	μ	σ	min-max	μ	σ	min-max
building	19	32	0–92	58	15	0–65	61	9	22–65
cancer	16	9	3–43	24	13	3–59	18	9	7–42
card	2	1	1–6	2	1	0–7	3	1	2–7
diabetes	12	8	3–34	8	6	0–32	23	19	0–71
flare	3	2	1–8	3	1	2–5	4	2	2–7
gene	5	2	2–13	5	2	1–12	5	2	3–9
glass	9	3	4–21	2	3	0–20	11	5	2–25
heart	5	3	2–14	5	3	2–15	4	2	1–8
hearta	5	3	0–13	4	2	2–11	4	1	2–9
heartac	3	2	2–8	1	1	0–4	1	1	0–3
heartc	3	1	2–6	4	4	0–19	2	1	0–4
horse	1	1	0–2	2	1	0–5	2	1	1–4
soybean	25	6	11–39	25	10	1–44	19	6	10–33
thyroid	48	12	15–59	37	16	2–59	48	12	21–59

Für jede Version (1, 2 oder 3) jedes Problems: Anzahl verborgener Knoten, die das cand-Verfahren in den verschiedenen Läufen dafür erzeugt hat, beschrieben durch den Mittelwert μ der Knotenanzahl, die Standardabweichung σ und das Minimum und Maximum.

Tabelle 5.10: Anzahl von cascade erzeugter verborgener Knoten

	Knoten f. Version 1			Knoten f. Version 2			Knoten f. Version 3		
Problem	μ	σ	min-max	μ	σ	min-max	μ	σ	min-max
building	4	5	0–16	9	6	0–25	16	7	0–35
cancer	15	14	3–54	15	15	0–60	14	9	2–39
card	1	1	1–3	2	1	1–6	2	1	1–4
diabetes	8	8	0–33	9	7	0–34	12	9	0–37
flare	3	2	1–8	3	1	2–7	6	4	2–17
gene	7	4	2–19	6	3	2–13	6	2	2–15
glass	12	9	2–38	3	5	0–29	14	11	2–42
heart	5	4	2–24	6	7	2–42	4	2	2–12
hearta	6	3	0–14	5	4	2–18	6	4	0–18
heartac	2	1	1–4	0	0	0–1	1	1	0–3
heartc	3	1	1–5	5	3	1–15	1	1	0–4
horse	1	1	0–2	1	1	0–3	1	2	0–9
soybean	32	9	13–43	25	12	2–43	27	10	12–40
thyroid	23	19	2–56	15	17	2–55	7	7	2–27

(Analog zu Tabelle 5.9)

soybean, thyroid erheblich größere Netze, als bei der Suche der Pivot-Architekturen in Kapitel 4 als Vorauswahl zugrundegelegt wurden. Dies könnte den Verfahren einen Vorsprung im Vergleich zu den statischen Verfahren und den Beschneidungsverfahren verschaffen.

Die Vergleiche von cand mit seinem Pendant kogi3 und von cascade mit seinem Pendant kogi2 sind in Tabelle 5.11 dargestellt. Anhand dieser Daten können die Fragen 3 und 4 beantwortet werden:

Tabelle 5.11: Vergleiche cand und kogi3 sowie cascade und kogi2 mittels t-Test

	cand vs. kogi3		
Problem	1	2	3
building	—	—	(c 1,1)
cancer	(c 0,0)	c 0,0	c 0,5
card	—	—	—
diabetes	c 0,0	—	c 0,0
flare	—	(c 0,6)	c 0,1
gene	k 0,0	k 0,1	k 0,0
glass	c 0,0	k 0,3	—
heart	c 0,0	—	c 0,0
hearta	c 0,3	(c 0,1)	(c 1,7)
heartac	c 1,8	—	—
heartc	c 2,4	—	—
horse	—	—	—
soybean	c 0,0	(c 3,1)	c 0,0
thyroid	c 0,0	—	—

	cascade vs. kogi2		
Problem	1	2	3
building	—	—	—
cancer	C 7,8	—	—
card	C 2,3	C 9,4	—
diabetes	—	—	—
flare	—	K 0,1	—
gene	—	—	(C 4,0)
glass	—	K 5,1	—
heart	—	—	—
hearta	—	(C 2,2)	—
heartac	—	—	—
heartc	—	—	—
horse	—	—	—
soybean	—	(C 0,9)	—
thyroid	—	—	—

Der Aufbau der Tabellen ist analog zu Tabelle 5.7.
Links (cand/kogi3): 18 mal kein signifikanter Unterschied, 20 mal cand besser („c", davon 6 dubios), 4 mal kogi3 besser („k").
Rechts (cascade/kogi2): 34 mal kein signifikanter Unterschied, 6 mal cascade besser („C", davon 3 dubios), 2 mal kogi2 besser („K").

Im wesentlichen sind kogi3 und kogi2 nicht besser als ihre herkömmlichen Entsprechungen cand und cascade. Bei cand und kogi3 ergibt sich eine klare Unterlegenheit von kogi3, mit einer Ausnahme: Auf dem gene-Problem liefert kogi3 die deutlich besseren Ergebnisse gegenüber cand. Eine Erklärung für dieses Phänomen könnte die hohe Anzahl von 120 Eingängen bei diesem Problem sein, der die Netze besonders anfällig für die Überanpassung eines Kandidaten in Bezug auf einzelne Eingänge macht. Solche Überanpassung wird mit kogi3 eher vermieden. Eine korrespondierende Unterlegenheit von kogi2 gegenüber cascade ist kaum ersichtlich; offenbar ist die Nützlichkeit der Validation der Kandidaten hier relativ höher, weil sich durch die große Mächtigkeit kaskadierter Knoten auch stärkere Überanpassungen realisieren lassen.

Diese Vermutung wird noch unterstützt von den Ergebnissen in Tabelle 5.12: Im direkten Vergleich schneidet das kaskadierende kogi2 besser ab, als kogi3, welches flache Netze baut. Insgesamt scheint die kogi-Idee also vor allem für kaskadierende Netze nützlich zu sein. Die in den vorliegenden Versuchsreihen benutzten Varianten sind aber wohl noch nicht optimal abgestimmt.

Es bleibt uns zum Abschluß die Beurteilung des „Zwitters" kogi9, der ein Netz mit genau zwei verborgenen Schichten baut. Die Vergleiche dieses Verfahrens mit kogi2 und kogi3 sind in der Tabelle 5.13 dargestellt. Offensichtlich ist die Kreuzung der beiden Verfahren technisch gelungen: kogi9 ist manchmal besser als kogi3 und manchmal besser als kogi2. Es scheint dabei recht nah an kogi3 zu liegen, zu dem es nur in wenigen Fällen überhaupt signifikante Unterschiede aufweist. Insgesamt ist kogi9 jedoch nicht besser als eines der beiden anderen Verfahren, es sei denn, es ist schon im Vorhinein bekannt, daß ein Netz mit zwei verborgenen Schichten eine besonders gute Wahl ist — das gene-Problem ist

Tabelle 5.12: Vergleich von kogi3 und kogi2 mittels t-Test

Problem	kogi3 vs. kogi2 1	2	3
building	—	k 0,0	—
cancer	(K 0,4)	—	—
card	—	—	—
diabetes	K 0,0	—	K 0,0
flare	—	(K 0,0)	K 0,1
gene	k 0,0	k 1,9	k 0,0
glass	K 0,8	k 4,1	k 4,2
heart	K 0,0	K 0,3	K 0,1
hearta	K 8,6	(K 2,7)	(K 5,5)
heartac	K 3,3	—	k 7,1
heartc	K 1,0	—	—
horse	K 8,7	K 4,9	—
soybean	K 0,0	(K 4,7)	K 0,0
thyroid	k 1,6	—	(k 0,0)

Der Aufbau ist analog zu Tabelle 5.7. 14 mal kein signifikanter Unterschied, 19 mal kogi2 besser („K", davon 5 dubios), 9 mal kogi3 besser („k", davon 1 dubios).

Tabelle 5.13: Vergleich von kogi2 und kogi3 mit kogi9 mittels t-Test

Problem	kogi3 vs. kogi9 1	2	3
building	—	k 0,0	—
cancer	—	—	—
card	—	(Z 1,0)	k 3,6
diabetes	—	—	—
flare	—	—	—
gene	—	—	k 9,0
glass	—	k 1,5	—
heart	—	—	—
hearta	—	—	—
heartac	—	—	—
heartc	—	Z 8,8	—
horse	—	—	—
soybean	—	—	—
thyroid	—	—	k 0,0

Problem	kogi2 vs. kogi9 1	2	3
building	—	Z 0,0	—
cancer	—	—	—
card	—	Z 0,1	—
diabetes	K 0,0	—	K 0,0
flare	—	K 0,0	K 0,1
gene	Z 0,0	Z 3,7	Z 0,0
glass	K 0,8	—	Z 2,0
heart	K 0,0	K 0,0	K 0,0
hearta	—	(K 1,1)	K 9,2
heartac	K 4,9	—	—
heartc	—	—	—
horse	—	—	—
soybean	K 0,0	(K 2,9)	K 0,0
thyroid	—	(Z 1,4)	(Z 0,0)

Der Aufbau der Tabellen ist analog zu Tabelle 5.7.
Links: 35 mal kein signifikanter Unterschied, 5 mal kogi3 besser („k"), 2 mal kogi9 besser („Z", davon 1 dubios).
Rechts: 20 mal kein signifikanter Unterschied, 14 mal kogi2 besser („K", davon 2 dubios), 8 mal kogi9 besser („Z", davon 2 dubios).

ein solcher Fall.

5.4 Zusammenfassung und Beiträge dieser Arbeit

In diesem Kapitel habe ich sechs Vertreter der Klasse von additiven Lernverfahren vorgestellt, die man als Kandidatenlernverfahren bezeichnen kann. Drei dieser Verfahren waren schon vorher bekannt, die anderen drei, genannt kogi, sind neue Verfahren, die auf folgender Idee basieren: Benutze zur Auswahl des besten aus einer Menge trainierter Kandidatenknoten nicht nur den Fehler dieser Kandidaten auf der Trainingsmenge, sondern zusätzlich den Fehler auf der Validationsmenge. Diese Herangehensweise kann den Vorteil haben, die Überanpassung zu verringern, kann aber zugleich durch das gewissermaßen in die Validationsprüfung geschlagene „Leck“, das in der Auswahlmethode steckt, dazu führen, daß dennoch stattfindende Überanpassung nicht entdeckt wird. Außerdem erschwert die Benutzung eines Kandidaten mit nicht optimaler Leistung auf der Trainingsmenge das weitere Training.

Die empirische Auswertung hat gezeigt, daß sich positive und negative Effekte ungefähr die Waage halten. Die kogi-Verfahren sind auf einigen der betrachteten Probleme besser als ihre Pendants, auf diversen anderen schlechter. Möglicherweise läßt sich durch eine bessere Abstimmung der Kandidatenauswahl ein durchgängiger Vorteil gegenüber den normalen Verfahren erreichen; in der vorgestellten Fassung ist das aber jedenfalls noch nicht gelungen.

Der zweite Beitrag dieser Arbeit besteht in der Bereitstellung einer großen Menge empirischer Daten über das Verhalten von Kandidatenlernverfahren bei Diagnoseaufgaben. Die in den Versuchsreihen ermittelten Daten stehen anderen Forschern zur Verfügung (siehe Anhang A) und können aufgrund der Standardisierung der PROBEN1-Datensammlung direkt für weitere Vergleiche genutzt werden. In den Versuchsreihen wurden insbesondere zwei Fragen geklärt. Erstens: Ob man Verfahren verwendet, die Netze mit nur einer verborgenen Schicht erzeugen, oder solche, die die verborgenen Knoten kaskadieren, macht in den meisten Fällen (für Diagnoseprobleme!) kaum einen Unterschied. Zweitens: Trotz seines theoretisch ungünstigen Trainingskriteriums funktioniert auch das Cascade-Correlation-Verfahren gut und hat bei Klassifikationsaufgaben sogar einen leichten Vorsprung gegenüber dem theoretisch „richtigeren“ Training der Kandidaten mit dem quadratischen Fehler. Nur bei Lernaufgaben mit kontinuierlichen Zielwerten der Ausgabeknoten kann letzteres seine theoretischen Vorzüge auch in bessere Lernergebnisse umsetzen.

Kapitel 6

Automatisches Lernen III: Subtraktive Verfahren

Fools ignore complexity.
Pragmatists suffer it.
Some can avoid it.
Geniuses remove it.
Alan Perlis' Programming Proverb #58

Perfection is achieved, not when
there is nothing left to add,
but when there is nothing left to take away.
Antoine de St. Exupery

In diesem Kapitel untersuche ich zwei subtraktive Verfahren zur Entfernung einzelner Verbindungen. Zunächst diskutiere ich die möglichen Vor- und Nachteile solcher Beschneidungsverfahren und die relevanten bisherigen Arbeiten (Abschnitt 6.1). Danach beschreibe ich ein Verfahren im Detail und schlage eine Verbesserung vor, die eine Teillösung für das bislang ungelöste Problem der automatischen Wahl der Beschneidungsstärke liefert (Abschnitt 6.2). Schließlich beschreibe ich eine empirische Untersuchung dieser beiden Verfahren und die dabei erzielten Ergebnisse (Abschnitt 6.3). Dieses Kapitel liefert den dritten Block von Daten für einen Vergleich verschiedener Ansätze für automatische Lernverfahren.

6.1 Einführung und verwandte Arbeiten

Ein Nachteil der im letzten Kapitel besprochenen additiven Lernverfahren ist die grobe Granularität ihrer Topologieänderungen. Es wird immer ein Knoten eingefügt, der mindestens mit allen Eingangsknoten und allen Ausgangsknoten verbunden ist. Für eine genau an das Problem angepaßte Topologie ist es aber offensichtlich wünschenswert, auch die Existenz einzelner Verbindungen dynamisch anzupassen. Leider ist alles andere als offensichtlich, wie vor dem Vorhandensein einer Verbindung entschieden werden sollte, ob ihre Existenz nützlich wäre. Selbst wenn man ein Kriterium hierfür angeben könnte, wäre dieses offenbar nicht von lokaler Natur und würde insofern einem Grundgedanken neuronaler Lernverfahren widersprechen.

Topologieveränderungen auf der Ebene einzelner Verbindungen werden aber sinnvoll möglich, wenn man den umgekehrten Weg wählt: Es kann von einer vorhandenen Verbindung während des Trainings entschieden werden, ob sie wirklich benötigt wird (und deshalb erhalten bleibt) oder ob sie überflüssig ist (und deshalb entfernt werden sollte). Der Grundgedanke solcher Verfahren besteht darin, für jede

Verbindung einen Koeffizienten zu berechnen, der die *Wichtigkeit* der Verbindung beschreibt. Die am wenigsten wichtigen Verbindungen werden dann entfernt. Ein solches Vorgehen kann während des Trainings mehrmals wiederholt werden. Eventuell müssen zur Berechnung solcher Wichtigkeitskoeffizienten während des Trainings kontinuierlich zusätzliche Daten gesammelt werden. Ein subtraktives Verfahren nach diesem Grundgedanken hat etwa folgende allgemeine Form:

Subtraktives Lernverfahren:
```
Initialisiere Netz mit fester Topologie;
REPEAT
  IF NOT erster Durchlauf THEN
    Bestimme Wichtigkeit jeder Verbindung;
    Entferne die k unwichtigsten Verbindungen END;
  REPEAT
    Mache Gradientenabstiegsschritt und sammle Wichtigkeitsinformation;
  UNTIL Beschneidungsschritt fällig;
UNTIL Ende;
```

Offenbar läßt sich dieser Beschneidungsgedanke mit additiven Verfahren koppeln: Ressourcen, die dem Netz während des Trainings zugefügt wurden, werden im weiteren Verlauf des Trainings daraufhin untersucht, ob sie tatsächlich benötigt werden, und werden entfernt, falls das nicht der Fall ist (additiv-subtraktive Verfahren). Bisher ist jedoch nicht bekannt, wie eine solche Kopplung gestaltet werden muß, um gut zu funktionieren, die Dynamik des Lernens bei additiven Verfahren steht nämlich in Wechselwirkung mit den Kriterien zum Beschneiden, weil eine „junge", d.h. erst vor kurzem dem Netz zugefügte Ressource sich qualitativ anders verhält, als eine „alte". Aus diesem Grund begnügen sich die meisten bisherigen Beschneidungsmethoden damit, ein Netz zu beschneiden, daß im Ganzen trainiert wird oder wurde, wie im obigen Pseudocode angegeben. Ich werde mich ebenfalls auf solche Verfahren beschränken.

Das einfachste Verbindungsbeschneidungsverfahren benutzt den Betrag des Gewichts einer Verbindung als Maß für deren Wichtigkeit und entfernt die betragskleinsten Verbindungen. Obwohl das Verfahren bisweilen eine Verbesserung bewirken kann [117], ist seine Nützlichkeit beschränkt — zu häufig werden Verbindungen entfernt, die trotz ihres kleinen Gewichts wichtig sind. Eine Verbesserung der Idee stellen die schon in Abschnitt 2.8.2 auf Seite 43ff besprochenen Verfahren *optimal brain damage* [85] und *autoprune* [117] dar.

Bei Optimal Brain Damage (OBD) wird im Maß für die Wichtigkeit T (dort genannt „Auffälligkeit" (*saliency*)) einer Verbindung w_i eine Annäherung an die zweite Ableitung der Fehlerfunktion E in Bezug auf das Gewicht dieser Verbindung benutzt.

Bestimme Wichtigkeit jeder Verbindung (OBD):

$$T(w_i) = w_i^2 \frac{\partial^2 E}{\partial w_i^2}$$

Diese Annäherung verwendet nur die Diagonalterme der Hessematrix (also der zweiten Ableitung) zur Berechnung der Wichtigkeit. Diesem Vorgehen liegen drei Annahmen zugrunde. Erstens die Annahme, daß bereits zu einem Minimum der Fehlerfunktion trainiert wurde, so daß der Gradient Null ist; das Kriterium *Beschneidungsschritt fällig* ist also erfüllt, wenn der Trainingsfehler nicht weiter absinkt. Der Nachteil dieser Annahme liegt darin, daß man die Beschneidung ja eigentlich durchführen will, um eine Überanpassung zu vermeiden, Training bis zum Minimum aber höchstmögliche Überanpassung nach sich zieht, die also erst nachträglich wieder bereinigt anstatt von vornherein vermieden wird. Ein Nebeneffekt dieses Problems ist, daß das Verfahren sehr hohen Rechenaufwand verursacht, weil vor jedem Beschneidungsschritt bis zum Minimum trainiert werden muß. Die zweite, durchaus plausible Annahme besagt, daß die Fehleroberfläche in der Nähe von Minima annähernd eine Fläche zweiten Grades ist, so daß Terme dritten und höheren Grades vernachlässigt werden können. Die dritte Annahme lautet, daß sich Gewichte nicht gegenseitig beeinflussen, so daß alle Terme $\partial^2 E/(\partial w_i \partial w_j)$, die in der Hessematrix abseits der Diagonalen liegen, Null sind. Diese letzte Annahme ist offensichtlich falsch, denn die Gewichte sind ja über die gemeinsame Beeinflussung der Ausgabeknoten miteinander gekoppelt; man hofft aber, daß die gegenseitige Beeinflussung der Gewichte nicht stark ist. Der obige Term für $T(w_i)$ kann auch als Teststatistik interpretiert werden für einen Test der Hypothese, daß w_i den Fehler beeinflußt [118]. Zur Berechnung von T müssen während des Trainings zusätzlich zu den Fehlerwerten aus der ersten Ableitung des Fehlers auch noch die aus der zweiten Ableitung des Fehlers (zweimal nach dem gleichen Gewicht abgeleitet) für jedes Beispiel durch das Netz zurückverfolgt und summiert werden.

Beim autoprune-Verfahren wird ebenfalls eine Teststatistik verwendet. Genau wie die von OBD gehört sie zur Familie der Wald-Statistiken und beschreibt die Signifikanz für die Ablehnung der Nullhypothese, daß das betrachtete Gewicht während des Trainings den Wert Null erreicht.

Bestimme Wichtigkeit jeder Verbindung (autoprune):

$$T(w_i) = \log\left(\frac{\left|\sum_p w_i - \eta(\partial E/\partial w_i)_p\right|}{\eta\sqrt{\sum_p((\partial E/\partial w_i)_p - \overline{(\partial E/\partial w_i)})^2}}\right)$$

Dabei gehen die Summen über alle Beispiele p der Trainingsmenge, η ist die Lernrate für den Gradientenabstieg und der Querstrich bedeutet den Mittelwert über die Beispiele. Auch diese Teststatistik macht die Annahme der Unabhängigkeit der Gewichte. Sie vermeidet jedoch eine rein quadratische Approximation der Fehleroberfläche, sondern entnimmt ihre Hinweise aus der Verteilung der Fehleranteile über die Beispielmenge. Der wichtigste Vorzug gegenüber OBD besteht jedoch darin, daß die autoprune-Statistik auch während des Trainings, d.h. vor Erreichen eines Fehlerminimums angewendet werden kann, so daß gar nicht erst eine starke Überanpassung entsteht. Die Berechnung der autoprune-Statistik ist weniger aufwendig als die Berechnung der OBD-Statistik, da keine zusätzlichen Fehlerwerte durch das Netz verfolgt werden müssen; es genügt, zu den Gradientenbeiträgen jedes Beispiels in jedem Gewicht auch deren Quadrate aufzusummieren. Gegenüber der originalen Formulierung der Statistik ist obige Form logarithmiert, weil dies für die weitere Behandlung bequemer ist — für die Benutzung oder Wirkung der Statistik macht es keinen Unterschied, da die Statistik nur zum Ordnen der Verbindungen nach Qualität benutzt wird und der Logarithmus streng monoton ist.

Finnoff, Hergert und Zimmermann finden in ihrer hervorragenden Studie [117], daß autoprune in vielen Fällen signifikant bessere Netze produziert als OBD und niemals schlechtere. Deshalb benutze ich für meine Untersuchungen zu Beschneidungsverfahren autoprune als Grundlage.

6.2 Zwei Beschneidungsverfahren

Ähnlich wie bei den im vorigen Kapitel besprochenen additiven Verfahren liegt auch bei den Beschneidungsverfahren einiges im Argen, was die Angabe präziser quantitativer Kriterien für die Durchführung der Verfahren anbelangt. Für Beschneidungsverfahren sind hier vor allem zwei Dinge interessant, nämlich

1. Wann wird ein Beschneidungsschritt ausgeführt? (Kriterium *Beschneidungsschritt fällig*)
2. Wieviele Verbindungen werden jeweils entfernt? (Wahl von k in Schritt *Entferne die k unwichtigsten Verbindungen*).

Nach Beantwortung dieser Fragen ist das Verfahren klar, wie im obigen Pseudocode angegeben. Die Beschneidung wird solange fortgesetzt, bis das Netz offensichtlich „ruiniert“ ist, also zuviel beschnitten wurde; dann wird das beste unterwegs gefundene Netz, gemessen am Validationsfehler, als Ergebnis verwendet.

Die Antwort auf die erste Frage ist noch relativ naheliegend: Bei OBD wird jeweils beschnitten, sobald ein Fehlerminimum erreicht ist. Auch wenn diese Formulierung natürlich kein unmittelbar einsetzbares Kriterium ist, da ein Minimum ja nie exakt erreicht wird, genügt sie doch für eine praktische Anwendung. Für autoprune möchte man einen Beschneidungsschritt immer dann ausführen, wenn eine Überanpassung des Netzes beginnt; als Kriterium geben die Autoren an „a repeated increase in the error on the validation set". Diese reichlich wolkige Beschreibung meint offenbar ein Kriterium aus der Klasse UP. Ich verwende im Folgenden UP_2 mit der üblichen Streifenlänge 5. Außerdem wird nach jedem Beschneidungsschritt mindestens 10 Epochen lang trainiert, bevor die nächste Beschneidung erfolgen kann.

Beschneidungsschritt fällig:
Seit letztem Beschneidungsschritt mindestens 10 Epochen trainiert `AND`
UP_2 Stoppkriterium ist erfüllt

Vor dem ersten Beschneidungsschritt wird außerdem ein Optimum des Validationsfehlers mit dem Stoppkriterium GL_5 gesucht, d.h. nach Zutreffen des Stoppkriteriums wird zum Optimalzustand des Netzes zurückgesetzt, dann weitertrainiert bis ein zweimaliger Anstieg des Validationsfehlers erfolgt und dann erst wird beschnitten. Dieses Vorgehen vermeidet eine voreilige Beschneidung vor Überschreiten des in Abschnitt 4.2.3 erwähnten „Generalisierungsbuckels".

6.2.1 autoprune

Problematischer ist die zweite Frage, wieviele Verbindungen in einem Beschneidungsschritt entfernt werden sollten. Der Artikel über OBD gibt darauf die vollkommen unzureichende Antwort „delete some low-saliency parameters" [85]. Für autoprune ist eine konkretere Angabe vorhanden, nämlich „im ersten Schritt 15% oder 35% aller Verbindungen, in allen folgenden Schritten 10%". Die Wahl zwischen 15% und 35% für den ersten Schritt wird mit Versuch und Irrtum in mehreren getrennten Trainingsläufen getroffen, was für ein automatisches Lernverfahren nicht in Frage kommt. Wir verwenden immer 35% im ersten Schritt, weil die zugrundeliegenden Netze als deutlich überdimensioniert angenommen werden können.

6.2.2 lprune

Der starre Beschneidungsplan von autoprune kann natürlich nicht für alle Fälle passend sein. Nötig wäre eine automatische Bestimmung der optimalen Beschneidungsstärke. Leider ist bisher vollkommen unbekannt, welche Kriterien man für diese Bestimmung heranziehen könnte. Ich beschreibe im folgenden ein Verfahren, das die Beschneidungsstärke automatisch an das jeweilige Trainingsstadium anpaßt und damit ein Teilproblem dieser wichtigen Frage löst.

Dem Verfahren liegt eine Betrachtung der Verteilung der Testkoeffizienten $T(w_i)$ zugrunde. Diese Koeffizienten sind in grober Annäherung normalverteilt. Zu Beginn des Trainings sind die Koeffizienten überwiegend noch durch den bei der Zufallsinitialisierung zugeteilten Gewichtsbetrag bestimmt. Die resultierende Verteilung ist schmal und bei kleinen Koeffizientenbeträgen angesiedelt, wie in Abbildung 6.1 als Histogramm dargestellt (alle Histogrammklassen haben Breite 1). In dieser Phase wird nicht beschnitten. Im Verlauf des Trainings wird diese Verteilung breiter, da sich die Gewichte ausdifferenzieren. Insbesondere entwickeln sich Verbindungen mit hoher Wichtigkeit, die dementsprechend einen großen T-Koeffizienten haben, so daß auch der Mittelwert der Verteilung ansteigt. Abbildung 6.2 zeigt die Verteilung in dem Zustand des Netzes, den ein Training mit frühem Stoppen als Resultat auswählen würde (kleinster Validationsfehler). Man beachte den geänderten Maßstab in y-Richtung.

Der erste Beschneidungsschritt findet nach Epoche 40 statt, bei einer Verteilung, wie sie in Abbildung 6.3 gezeigt ist. Diese Verteilung ist noch weiter ausdifferenziert als die aus Abbildung 6.2, insbesondere gibt es jetzt eine größere Zahl von Gewichten mit kleinen T-Koeffizienten und trotzdem

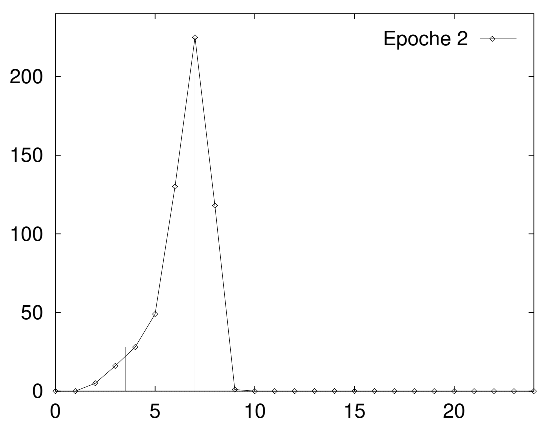


Abbildung 6.1: Häufigkeitsverteilung der Testkoeffizienten T der Gewichtsbeschneidung. x-Achse: T, y-Achse: Anzahl (Klassenbreite=1). Training von glass1 mit Pivot-Architektur (16+8 verborgene Knoten), Zustand in Epoche 2.

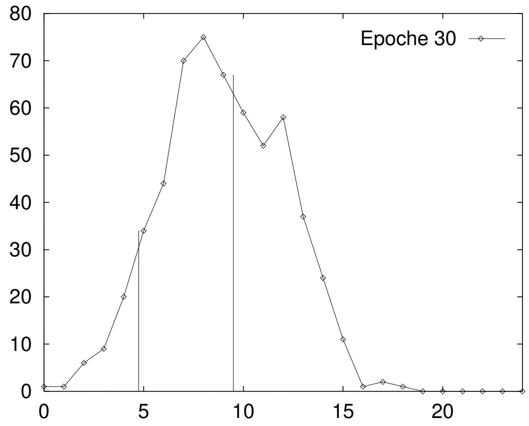


Abbildung 6.2: Dito, jedoch Zustand am Validationsminimum in Epoche 30. Rechter senkrechter Strich ist beim Mittelwert von T, linker senkrechter Strich ist beim halben Mittelwert (ebenso in den anderen Abbildungen).

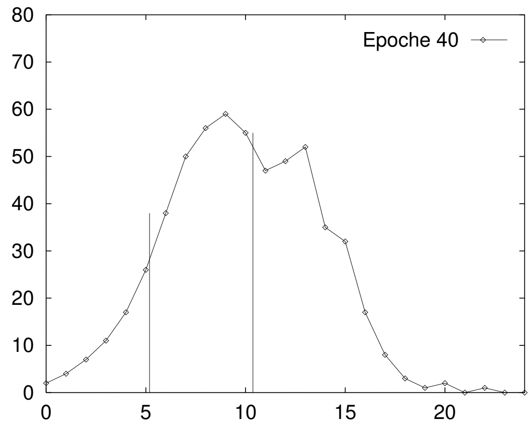


Abbildung 6.3: Verteilung der T-Koeffizienten in Epoche 40, unmittelbar vor erster Beschneidung.

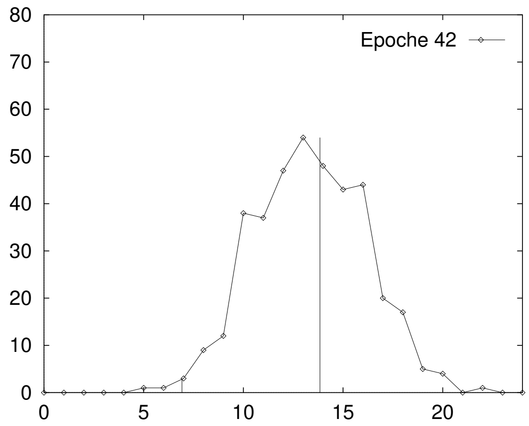


Abbildung 6.4: Verteilung der T-Koeffizienten in Epoche 42, kurz nach erster Beschneidung mit Entfernen von 35% der Verbindungen.

ist der Mittelwert der Verteilung ein wenig weiter angestiegen. Beschneiden wir das Netz in diesem Zustand, so steigt der Mittelwert der Verteilung aus zwei Gründen sprunghaft an (siehe Abbildung 6.4): Erstens weil viele Verbindungen mit kleinem Koeffizienten nun fehlen und zweitens weil einige der weiterhin existierenden Verbindungen plötzlich eine stark erhöhte Wichtigkeit bekommen.

Im weiteren Verlauf des Trainings differenziert sich dann auch diese durch die Beschneidung schmaler gewordene Verteilung wieder aus (Abbildung 6.5), der nächste Beschneidungsschritt erfolgt usw. Wie wir sehen, bewirkt die normale Entwicklung der Gewichte und auch das Beschneiden jeweils eine Änderung der Varianz und des Mittelwerts der Verteilung der T-Koeffizienten. Die Varianz steigt durch das normale Training an und wird durch das Beschneiden reduziert; der Mittelwert steigt durch das normale Training langsam und durch Beschneidung sprunghaft an; nach einer Beschneidung geht der Mittelwert dann zunächst zurück. Der Grundgedanke des vorgeschlagenen Verfahrens besteht darin, die Veränderungen von Mittelwert und Varianz zu verwenden, um abzuschätzen, wieviele Verbindungen in einem Beschneidungsschritt entfernt werden sollten.

Die einfachste Art, diese Idee umzusetzen, besteht darin, in jedem Beschneidungsschritt alle diejenigen Verbindungen zu entfernen, deren Koeffizienten kleiner sind als ein bestimmter Bruchteil $\lambda \in 0 \ldots 1$

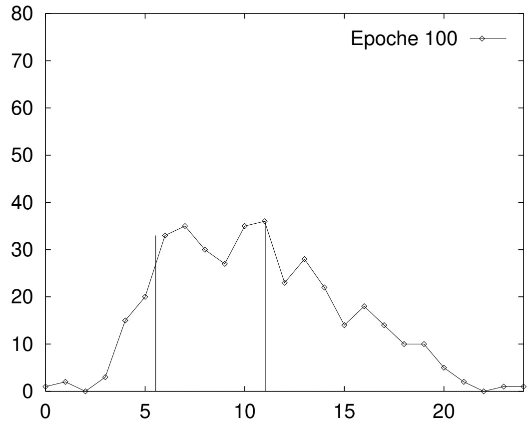


Abbildung 6.5: Verteilung der T-Koeffizienten in Epoche 100, kurz vor der zweiten Beschneidung.

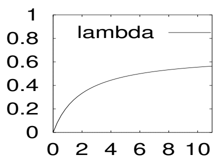
des Mittelwertes, also

$$\text{Entferne } w_i \quad \Longleftrightarrow \quad T(w_i) < \lambda \overline{T}$$

Für $\lambda = 1/2$ ist die Konsequenz dieser Regel anhand der linken senkrechten Linie in obigen Abbildungen abzulesen; es würde jeweils derjenige Teil der Verbindungen entfernt, der im Histogramm links dieser Linie erscheint. Solange die Gewichte nicht gut ausdifferenziert sind, ist dieser Anteil klein. Der Anteil steigt mit dem Maß der Ausdifferenzierung an. Insbesondere ist unmittelbar nach einer Beschneidung der Anteil sehr klein, so daß eine früh folgende weitere Beschneidung nicht zu viele Verbindungen entfernen würde.

Leider ist diese Regel nicht für alle Probleme gleichermaßen brauchbar. Ein zu großes λ beschneidet offensichtlich zu stark; für $\lambda \approx 1$ würden in jedem Schritt etwa die Hälfte der Gewichte entfernt. Ein zu kleines λ kann hingegen bei Problemen mit heftiger Überanpassung nicht schnell genug für eine Verkleinerung des Netzes sorgen. Da zwischen zwei Beschneidungsschritten mit seltenen Ausnahmen mindestens 15 Epochen lang trainiert wird, muß bei starker Überanpassung stärker beschnitten werden, als es sonst sinnvoll wäre. Glücklicherweise haben wir mit dem Generalisierungsverlust GL ein Maß für die Überanpassung, welches wir verwenden können, um das Kriterium entsprechend zu modifizieren. Experimente ergaben, daß eine Anpassung von λ im Bereich $0 \ldots \frac{2}{3}$ mit hyperbolisch von GL abhängender Charakteristik eine brauchbare Anpassungsmethode zu sein scheint. Wir formulieren damit eine Regel zur automatischen Anpassung von λ wie folgt

$$\lambda := \lambda_{max}\left(1 - \frac{1}{1 + \frac{GL}{\alpha}}\right)$$



Dabei ist α ein Parameter, der angibt, bei welchem Wert von GL das λ den Wert $\lambda_{max}/2$ erreicht und λ_{max} ist, wie oben erwähnt, 2/3. Für größere Werte von λ_{max} wird die Beschneidung instabil, d.h. es kommen Beschneidungsschritte vor, die zu stark sind und die das Netz vollkommen ruinieren. Für $GL < \alpha$ steigt λ steil mit GL an, darüber nur noch langsam, mit asymptotischer Annäherung an λ_{max} für große GL.

Da zum Zeitpunkt der Beschneidung zwei Verschlechterungen des Validationsfehlers unmittelbar vorausgegangen sind, finden wir stets Werte von GL vor, die größer als Null sind. Es empfiehlt sich, für α einen Wert zu wählen, der etwa so groß ist, wie die GL Werte, die wir beobachten, wenn unmittelbar vor den zwei Validationsfehleranstiegen $GL = 0$ galt. In diesem Sinne wähle ich $\alpha = 2$, also $\lambda = 1/3$ bei 2% Validationsfehlerverschlechterung. Im Falle heftiger Überanpassung ist GL schon *vor* den zwei Anstiegen größer als Null und λ würde entsprechend höher gewählt.

Das Beschneidungsverfahren mit dieser Anpassungsmethode für die Beschneidungsstärke nenne ich *lprune* für *lambda-autoprune*.

In Abbildung 6.6 ist die Fehlerentwicklung bei einem Lauf von autoprune zu sehen. Jeder Beschneidungsschritt führt zu einem ruckartigen Anstieg und dann schnellem Wiederrückgang sowohl des Trainings- als auch des Validationsfehlers. Im ersten Beschneidungsschritt werden etwa 35% der Verbindungen entfernt, danach jeweils etwa 10% der verbliebenen. Gegen Ende des Trainings wird das Netz überbeschnitten, weil ich ein sehr konservatives Stoppkriterium verwende:

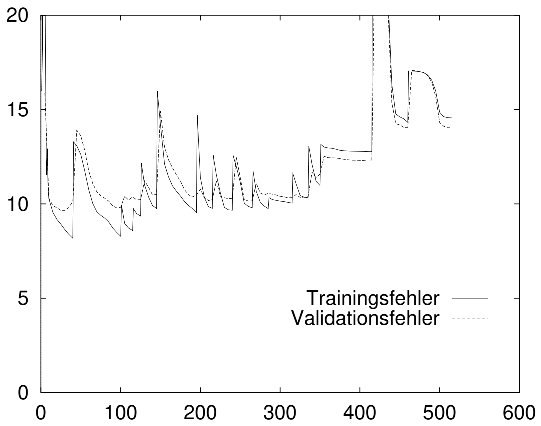


Abbildung 6.6: Zeitlicher Verlauf von Trainings- und Validationsfehler bei autoprune für den Lauf, aus dem die obigen Abbildungen stammen. x-Wert: Epoche, y-Wert: Fehler. Es erfolgen 15 Beschneidungsschritte, am Ende verbleiben 15% der ursprünglichen Verbindungen. In diesem Fall ist die Beschneidung nicht erfolgreich: Es wird kein kleinerer Validationsfehler erreicht als vor dem ersten Beschneidungsschritt. Dieser Fall ist nicht ungewöhnlich.

Ende:
Insgesamt mehr als 5000 Epochen trainiert `OR` $P_5(t) < 0,1$ `OR`
(Mindestens 25 Epochen seit letzter Beschneidung trainiert `AND`
Generalisierungsverlust $GL(t) > 100$ `AND` Trainingsfortschritt $P_5(t) < 0,4$)

Es wird also erst gestoppt, wenn der Validationsfehler nach Verlangsamung des Fehlerrückgangs im Anschluß an eine Beschneidung doppelt so groß wie sein bisheriges Minimum ist; *Ende* ist also ein modifiziertes GL_{100} Stoppkriterium. Zum Vergleich: Beim frühen Stoppen war das langsamste Kriterium dieser Klasse GL_5. Die hohe Schwelle ist deshalb nötig, weil das Beschneiden auch nach Phasen stark erhöhten Validationsfehlers noch zu neuen Minima führen kann. Das gleiche Kriterium verwende ich auch für lprune.

Bei lprune verlaufen die ersten Beschneidungsschritte im Gegensatz zu autoprune sehr vorsichtig, die Beschneidung ist nur gering. Das zu Abbildung 6.6 analoge Beispiel ist in Abbildung 6.7 gezeigt: Die

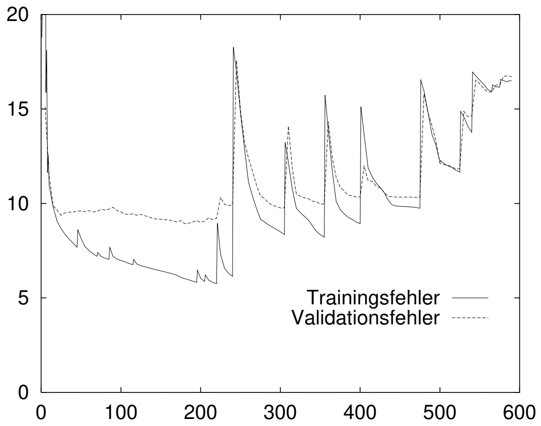


Abbildung 6.7: Verlauf des Trainings- und des Validationsfehlers bei lprune für das gleiche Beispiel wie in obigen Abbildungen. Es erfolgen 21 Beschneidungsschritte, anfangs schwach, später stark, am Ende verbleiben 0,5% der ursprünglichen Verbindungen. In diesem Fall ist die Beschneidung scheinbar erfolgreich: Nach 7 Beschneidungsschritten wird in Epoche 180 ein Validationsfehlerminimum erreicht, welches niedriger liegt, als das von frühem Stoppen erreichte in Epoche 25; allerdings ist der (hier nicht gezeigte) Testfehler in Epoche 180 schlechter als in Epoche 25.

ersten vier Beschneidungsschritte liegen alle vor Epoche 100 und entfernen 2%, 3%, 3% und 5% der (jeweils verbliebenen) Verbindungen. Spätere Schritte (z.B. in den Epochen 240 bis 400) entfernen dann zwischen 30% und 37%. Bei anderen Lernproblemen kommen aber auch andere Verläufe zustande.

Vermutlich sind die bei lprune gewählten Werte für λ_{max} und α nicht optimal. Möglicherweise ließen sich z.B. mit kleineren Werten wie $\lambda_{max} = 0,6$, $\alpha = 1,5$ bessere Ergebnisse erzielen. Es ist wiederum zu betonen, daß zur Herleitung aller Kriterien nur die Trainings- und Validationsfehler herangezogen wurden, nicht jedoch die Testfehler.

6.3 Versuchsaufbau und Ergebnisse

Zum Vergleich der Verfahren autoprune und lprune wurden mit jedem von beiden 30 Läufe für jedes PROBEN1-Problem mit den Pivot-Architekturen als Startnetz durchgeführt und weitere 30 mit den Reinschicht-Pivotarchitekturen. Insgesamt liegen also 4 mal 14 Stichproben zu je 30 Läufen vor, insgesamt 2560 Läufe. Diese Läufe wurden kontinuierlich in der Zeit vom 31. August bis 2. Oktober 1994 auf der KSR1 (32 Prozessoren) des Rechenzentrums der Universität Mannheim durchgeführt. Die verwendeten Programme sind wiederum in CuPit geschrieben. Es wurden dieselben Parameter für das zugrundeliegende RPROP-Verfahren benutzt wie in Abschnitt 3.2.17 und in Abschnitt 4.4.2. Die Parameter für die Beschneidung waren wie oben angegeben.

Die in den Versuchen gesammelten Daten wurden mittels statistischer Signifikanzprüfung auf Unterschiede im Mittelwert der erzielten quadratischen Fehler auf der Testmenge untersucht, um die folgenden Fragen zu beantworten:

1. Ist lprune besser als autoprune, wenn man die Pivot-Architekturen zugrundelegt?
2. Ist lprune besser als autoprune, wenn man die Reinschicht-Pivotarchitekturen zugrundelegt? Ändert sich an diesem Vergleich durch den Wechsel der Startnetze wesentliches?
3. Ist es bei autoprune sinnvoll, Netze mit mehr Verbindungen (nämlich den Direktverbindungen) zugrundezulegen?
4. Ist es bei lprune sinnvoll, Netze mit mehr Verbindungen (nämlich den Direktverbindungen) zugrundezulegen?
5. Sind in dieser Hinsicht systematische Unterschiede zwischen autoprune und lprune zu erkennen? Welche?

Ich vergleiche hier zunächst nur die Beschneidungsverfahren miteinander. Ein Vergleich der Beschneidungsverfahren mit Kandidatenlernverfahren und mit dem Lernen mit frühem Stoppen folgt im Kapitel 7.

Ich verzichte auf die Wiedergabe der Tabellen mit den Einzelergebnissen[1] und gehe direkt zum statistischen Vergleich über. Wie in Abschnitt 4.4.2 wurden die logarithmierten Testfehler zugrundegelegt, zum Vergleich ein t-Test benutzt, dabei die Cochran/Cox-Approximation für ungleiche Varianzen verwendet und Ausreißer aus den Stichproben entfernt. Auch in diesen Vergleichen gibt es einige nicht normalverteilte Stichproben.

Für autoprune gab es bei den Pivot-Architekturen 10 Ausreißer nach oben, also mit abnorm großen Werten des Testfehlers, und 4 nach unten (insgesamt also 1,1%), sowie ebenfalls eine nicht normalverteilte Stichprobe (hearta1); bei den Reinschicht-Pivotarchitekturen waren es 15 Ausreißer nach oben und 0 nach unten (insgesamt 1,2%), sowie eine nicht normalverteilte Stichprobe (cancer3).

Für lprune gab es bei den Pivot-Architekturen 20 Ausreißer nach oben und 5 nach unten (insgesamt also 2,0%), sowie ausschließlich normalverteilte Stichproben; bei den Reinschicht-Pivotarchitekturen

[1] Die Rohdaten sämtlicher einzelner Programmläufe sind jedoch elektronisch verfügbar; siehe Anhang A.

waren es 21 Ausreißer nach oben, 7 nach unten (insgesamt 2,2%), sowie ebenfalls ausschließlich normalverteilte Stichproben.

Nun zu den Ergebnissen. In Tabelle 6.8 ist der Vergleich von autoprune und lprune angegeben. Angesichts dieser Ergebnisse müssen die ersten beiden Fragen mit einem klaren „Jein“ beantworten werden: lprune ist auf manchen Problemen besser als autoprune, auf anderen schlechter. Dies gilt gleichermaßen für die Pivot-Architekturen wie für die Reinschicht-Pivotarchitekturen; es stellen sich auch in beiden Fällen sehr ähnliche Einzelergebnisse im Hinblick auf die Überlegenheit eines Verfahrens über das andere ein. Die Ergebnisse deuten darauf hin, daß lprune tendenziell bei den Problemen mit kleineren Eingangsschichten besser funktioniert, autoprune eher bei jenen mit großen. Dies könnte darauf zurückzuführen sein, daß bei Netzen mit einer großen Anzahl von Verbindungen, die in jeden verborgenen Knoten führen, eine rasche Beschneidung relativ früh während des Lernens nützlich ist, um eine Überanpassung durch zahlreiche wenig relevante Eingaben zu vermeiden. Prinzipbedingt führt autoprune eine solche aus, da der erste Beschneidungsschritt ja 35% aller Verbindungen entfernt, während lprune gerade zu Beginn der Beschneidung fast immer „vorsichtig“ vorgeht und nur wenige Verbindungen entfernt. Insgesamt scheint zumindest für die hier betrachteten Beispiele autoprune doch das etwas bessere Verfahren zu sein.

Tabelle 6.8: Vergleich von autoprune und lprune mittels t-Test

Pivot-Architekturen			
Problem	1	2	3
building	L 0,0	—	—
cancer	—	—	—
card	—	—	—
diabetes	—	A 2,5	L 0,9
flare	—	A 7,3	—
gene	A 0,0	A 0,0	A 0,0
glass	—	L 2,3	L 1,8
heart	A 5,5	A 0,4	—
hearta	—	A 0,1	—
heartac	—	—	—
heartc	—	—	A 2,5
horse	—	—	A 3,4
soybean	—	—	—
thyroid	—	L 6,2	L 0,3

Reinschicht-Pivotarchitekturen			
Problem	1	2	3
building	l 0,3	—	l 3,7
cancer	—	a 0,9	—
card	—	—	—
diabetes	—	a 4,3	—
flare	a 0,8	a 1,0	a 0,0
gene	a 5,4	l 6,6	a 1,7
glass	a 7,6	—	—
heart	—	a 3,3	—
hearta	—	—	—
heartac	—	—	—
heartc	—	—	—
horse	—	—	a 1,5
soybean	l 7,0	—	—
thyroid	—	—	l 0,0

Der Aufbau der Tabellen ist analog zu Tabelle 4.12 von Seite 87: Verglichen wurden die Mittelwerte der logarithmierten Testfehler der Läufe von autoprune (A) und lprune (L) mit Pivot-Architekturen (linke Tabelle), sowie von autoprune (a) und lprune (l) mit Reinschicht-Pivotarchitekturen (rechte Tabelle) für die Varianten 1, 2, und 3 jedes Problems. Die Einträge zeigen Unterschiede, die auf 10%-Niveau signifikant sind und den zugehörigen p-Wert (in Prozent). Der Buchstabe gibt jeweils an, welcher Algorithmus signifikant besser ist; ein Querstrich bedeutet, daß keine signifikanten Unterschiede vorliegen. Eingeklammerte Resultate (siehe nachfolgende Tabellen) sind dubios, weil mindestens eine der beteiligten Stichproben nicht normalverteilt ist. Es wurde ein t-Test mit der Cochran/Cox-Approximation für den Fall ungleicher Varianzen verwendet.
Links (Pivot-Architektur): 26 mal kein signifikanter Unterschied, 10 mal autoprune besser („A“), 6 mal lprune besser („L“).
Rechts (Reinschicht-Pivotarchitektur): 27 mal kein signifikanter Unterschied, 10 mal autoprune besser („a“), 5 mal lprune besser („l“).

In Tabelle 6.9 sind die unterschiedlichen Netzarchitekturen (bei konstantem Lernverfahren) einander gegenübergestellt. Die Ergebnisse liefern die Antworten auf die Fragen 3 bis 5: Sowohl bei autoprune als auch bei lprune macht es keinen sehr großen Unterschied, ob man die Pivot-Architektur oder die

Tabelle 6.9: Vergleich von Netzarchitekturen für autoprune und lprune mittels t-Test

autoprune

Problem	1	2	3
building	a 7,2	a 0,6	A 5,0
cancer	a 7,8	—	(A 5,0)
card	a 7,4	—	A 4,2
diabetes	—	—	—
flare	a 6,4	—	a 0,0
gene	a 0,0	—	—
glass	a 0,0	—	—
heart	—	—	—
hearta	—	—	—
heartac	a 0,2	A 2,6	A 2,2
heartc	—	—	A 1,2
horse	—	—	a 0,0
soybean	A 0,0	a 0,0	A 0,0
thyroid	—	a 1,8	—

lprune

Problem	1	2	3
building	—	—	—
cancer	l 7,6	L 1,3	—
card	l 2,8	—	L 0,7
diabetes	—	—	—
flare	—	—	—
gene	l 0,0	l 0,0	l 5,2
glass	—	—	—
heart	—	—	—
hearta	—	—	—
heartac	l 0,0	L 1,7	—
heartc	—	l 2,2	—
horse	—	—	l 0,5
soybean	L 0,0	l 0,0	L 0,0
thyroid	—	—	—

Der Aufbau der Tabellen ist analog zu Tabelle 6.8. Hier werden jeweils für das gleiche Lernverfahren die Architekturvarianten Pivot-Architektur einerseits und Reinschicht-Pivotarchitektur andererseits verglichen.
Links (autoprune): 22 mal kein signifikanter Unterschied, 12 mal Reinschicht-Pivotarchitektur besser („a“), 8 mal Pivot-Architektur besser („A“, davon 1 mal dubios).
Rechts (lprune): 28 mal kein signifikanter Unterschied, 9 mal Reinschicht-Pivotarchitektur besser („l“), 5 mal Pivot-Architektur besser („L“).

Reinschicht-Pivotarchitektur verwendet. Zwar hat die Reinschicht-Pivotarchitektur insgesamt leichte Vorteile, jedoch kommen die Verfahren für die Mehrzahl der Probleme mit beiden Architekturen zu etwa gleichen Ergebnissen. Die Beschneidungsverfahren können also Mängel in der Wahl der Ausgangsarchitektur tatsächlich in einem gewissen Umfang ausgleichen, wie sich durch Vergleich mit Tabelle 4.12 erkennen läßt: Bei der Methode des frühen Stoppens hat die Wahl der Architektur einen spürbar größeren Einfluß auf die Ergebnisse. Systematische Unterschiede zwischen autoprune und lprune in der Fähigkeit, mit unterschiedlichen Ausgangsarchitekturen zurechtzukommen, lassen sich nicht erkennen.

6.4 Zusammenfassung und Beiträge dieser Arbeit

In diesem Kapitel habe ich eine neue Methode *lprune* vorgestellt, die erstmals in einem gewissen Grad die automatische Wahl der Beschneidungsstärke für Beschneidungsverfahren unterstützt. Die Beschneidungsstärke wird dabei aufgrund einer Betrachtung der Verteilung der Beschneidungskoeffizienten an den momentanen Trainingsfortschritt angepaßt. In der empirischen Auswertung hat dieses Verfahren gezeigt, daß es tatsächlich in zahlreichen Fällen signifikant bessere Ergebnisse liefert, als das aus der Literatur bekannte autoprune-Verfahren, aus dem es hergeleitet wurde. Allerdings mangelt es lprune an der Fähigkeit, auch absolut eine Beschneidungsstärke zu wählen, anstatt nur relativ aus dem Trainingsfortschritt. Dies ist vermutlich der Grund, weshalb lprune auf zahlreichen anderen Problemen autoprune unterlegen ist — offenbar immer dann, wenn eine drastische Beschneidung in frühen Trainingsphasen nötig wäre, die lprune jedoch prinzipbedingt nicht vornimmt.

Ein weiterer Beitrag dieser Arbeit ist die Sammlung einer beträchtlichen Menge von empirischen Daten über das Verhalten von Beschneidungsverfahren auf realen Datensätzen. Diese Daten werden in Kapitel 7 Grundlage eines Vergleichs von statischen, additiven und subtraktiven Lernverfahren sein; sie sind öffentlich verfügbar (siehe Anhang A) und können aufgrund der Standardisierung der PROBEN1-Datensammlung für weitere Vergleiche anderer Forscher genutzt werden.

Kapitel 7

Konklusion: Automatisches Lernen

The fundamental principle of science,
the definition almost, is this:
the sole test of the validity of any idea
is experiment.
Richard P. Feynman

Vergleichende Studien ergeben
wertvolle „harte“ experimentelle Fakten
zur Verfeinerung unserer Theorien.
Wolfgang Maas (Theoretiker für maschinelles Lernen)

In diesem Kapitel vergleiche ich zunächst quantitativ die drei in den vorherigen Kapiteln untersuchten Ansätze zum automatischen Lernen, also frühes Stoppen, additive Verfahren und subtraktive Verfahren. Im Anschluß fasse ich die Ergebnisse des ersten Teils dieser Arbeit zusammen und bewerte sie. Zuletzt folgt ein Ausblick auf die Zukunft automatischer Lernverfahren mit dynamischer Netztopologie.

7.1 Vergleich der Ansätze

In den folgenden Abschnitten vergleiche ich die Ergebnisse der Lernverfahren über die Grenzen der Verfahrensklassen hinweg. Zur Vereinfachung wird dabei jeweils nur das im Mittel beste Verfahren aus jeder Klasse herangezogen.

7.1.1 Additiv versus frühes Stoppen

Tabelle 7.1 zeigt den Vergleich des cand-Verfahrens mit der Methode des frühen Stoppens bei statischer Netztopologie. Offenbar haben diese beiden Verfahren ihre Stärken auf verschiedenen Gebieten, wie der recht kleine Anteil von Einträgen ohne signifikanten Unterschied anzeigt. Möglicherweise rühren allerdings viele der Erfolge von cand daher, daß nicht alle der Architekturen der statischen Netze tatsächlich nahe an einer optimalen vollverbundenen Architektur liegen. Eventuell waren die Kriterien, die in Abschnitt 4.4.1 zur Bestimmung der Pivot-Architekturen verwendet wurden, für einige Fälle nicht geschickt, oder wir haben bei den zugrundegelegten Programmläufen für die Auswahl der Architektur einfach Pech gehabt. Nichtsdestoweniger ist ja gerade dies ein Vorteil der additiven Verfahren: Daß eben eine solche sorgfältige Auswahl der Netzarchitektur nicht notwendig ist, um ein brauchbares Lernergebnis zu erhalten. Insofern haben die untersuchten Kandidatenlernverfahren also sicherlich ihren Platz; von einer generellen Überlegenheit über das Lernen mit herkömmlichen statischen Netzen kann jedoch nicht gesprochen werden.

Tabelle 7.1: Vergleich von cand und frühem Stoppen mittels t-Test

Pivot-Architekturen			
Problem	1	2	3
building	(c 0,0)	c 8,1	—
cancer	P 0,0	P 0,1	P 0,0
card	P 2,1	—	(c 0,0)
diabetes	P 0,0	—	—
flare	c 0,0	c 0,0	c 0,0
gene	c 0,0	(c 0,0)	(c 0,0)
glass	c 0,0	—	c 0,0
heart	P 0,0	P 0,0	c 1,8
hearta	P 0,2	—	—
heartac	—	c 3,1	P 6,2
heartc	c 0,0	P 0,0	—
horse	c 0,2	c 0,1	c 0,1
soybean	—	(P 5,4)	c 0,0
thyroid	P 0,0	(P 0,0)	P 0,0

Reinschicht-Pivotarchitekturen			
Problem	1	2	3
building	(c 0,1)	—	(c 0,9)
cancer	S 0,0	S 1,3	S 0,0
card	S 0,0	S 0,3	(c 0,0)
diabetes	S 1,0	c 1,1	S 0,1
flare	c 0,0	c 0,2	c 0,3
gene	S 0,0	—	—
glass	—	S 0,3	c 0,0
heart	S 0,0	S 0,0	c 0,0
hearta	S 0,0	c 0,8	c 0,2
heartac	S 0,1	c 3,2	(c 9,7)
heartc	c 0,0	S 0,0	—
horse	S 3,5	S 0,7	S 0,0
soybean	c 0,0	(S 0,1)	c 0,0
thyroid	S 0,0	(S 0,0)	S 0,0

Der Aufbau der Tabellen ist analog zu Tabelle 6.8. Hier wird das additive Verfahren cand mit dem frühen Stoppen auf Pivot-Architekturen und auf Reinschicht-Pivotarchitekturen verglichen.
Links (Pivot-Architektur): 10 mal kein signifikanter Unterschied, 18 mal cand besser („c", davon 4 dubios), 14 mal frühes Stoppen mit Pivot-Architektur besser („P", davon 2 dubios).
Rechts (Reinschicht-Pivotarchitektur): 5 mal kein signifikanter Unterschied, 21 mal frühes Stoppen mit Reinschicht-Pivotarchitektur besser („S", davon 2 dubios), 16 mal cand besser („c", davon 4 dubios).

7.1.2 Subtraktiv versus frühes Stoppen

Tabelle 7.2 zeigt den Vergleich des autoprune Verfahrens mit der Methode des frühen Stoppens bei statischer Netztopologie. Die Varianz dieses Vergleichs wurde reduziert, indem gepaarte Läufe verwendet wurden: Jeder Lauf eines Beschneidungsverfahrens trainiert vor dem ersten Beschneidungsschritt zunächst bis zur Erfüllung des GL_5 Stoppkriteriums. Das an diesem Punkt vorliegende Zwischenergebnis wurde als Ergebnis eines Laufs mit frühem Stoppen registriert. Der Lauf des Beschneidungsverfahrens kann dasselbe Ergebnis erreichen (wenn er später keinen niedrigeren Validationsfehler mehr produziert) oder ein besseres (später niedrigerer Validationsfehler und niedrigerer Testfehler) oder schlechteres (später niedrigerer Validationsfehler aber höherer Testfehler). Durch diese Varianzreduktion sind die Ergebnisse der Versuchsreihe so trennscharf, wie es sonst nur mit erheblich längeren Versuchsreihen möglich gewesen wäre.

Im Mittel schneidet autoprune spürbar besser ab als das Lernen mit statischer Netztopologie. Offensichtlich ist die Beschneidung von Verbindungen eine Technik, die nicht nur zu kleineren Netzen führt, sondern bei richtiger Anwendung auch die Qualität der erzielten Lösungen verbessert. Die Verbesserungen durch das Beschneiden fallen deutlicher aus, wenn man eine weniger günstige Netztopologie zugrundelegt, wie der höhere Vorsprung von autoprune auf den (ungünstigeren) Pivot-Architekturen verglichen mit den Reinschicht-Pivotarchitekturen anzeigt.

7.1.3 Additiv versus subtraktiv

Tabelle 7.3 zeigt den Vergleich von autoprune auf Reinschicht-Pivotarchitekturen mit dem cand-Verfahren und dem cascade-Verfahren. Dieser Vergleich macht deutlich, daß die grobkörnige Anpassung der Netzarchitektur auf der Ebene von Knoten, wie sie bei den additiven Verfahren eingesetzt wird, der feinkörnigen Anpassung durch Beschneidung einzelner Verbindungen unterlegen ist. Es gibt

Tabelle 7.2: Vergleich von autoprune und frühem Stoppen mittels t-Test

Pivot-Architekturen			
Problem	1	2	3
building	(A 0,0)	—	—
cancer	—	P 3,1	P 9,9
card	—	A 0,0	A 2,2
diabetes	—	A 4,0	—
flare	A 0,0	A 0,0	A 0,0
gene	A 0,0	(A 0,0)	(A 0,0)
glass	A 8,6	A 2,3	A 0,1
heart	—	—	—
hearta	—	A 2,0	—
heartac	—	—	—
heartc	—	—	A 0,0
horse	—	A 0,4	A 0,1
soybean	—	—	—
thyroid	A 0,4	—	—

Reinschicht-Pivotarchitekturen			
Problem	1	2	3
building	(a 0,0)	—	S 7,9
cancer	—	—	S 0,1
card	—	a 0,0	a 7,1
diabetes	—	a 6,1	—
flare	—	a 0,0	a 0,3
gene	a 1,1	—	(a 0,4)
glass	—	—	—
heart	S 0,2	—	—
hearta	S 2,4	a 3,4	a 0,5
heartac	—	S 9,2	(a 5,4)
heartc	—	S 2,4	a 1,3
horse	S 4,7	—	S 8,6
soybean	—	(a 1,7)	—
thyroid	a 0,0	a 0,1	a 2,2

Der Aufbau der Tabellen ist analog zu Tabelle 6.8. Hier wird das autoprune-Beschneidungsverfahren mit dem frühen Stoppen für Pivot-Architektur (linke Tabelle) und Reinschicht-Pivotarchitektur (rechte Tabelle) verglichen.
Links (Pivot-Architektur): 22 mal kein signifikanter Unterschied, 18 mal autoprune besser („A", davon 3 dubios), 2 mal frühes Stoppen besser („P").
Rechts (Reinschicht-Pivotarchitektur): 18 mal kein signifikanter Unterschied, 16 mal autoprune besser („a", davon 4 dubios), 8 mal frühes Stoppen besser („S").

Tabelle 7.3: Vergleich von autoprune mit cand und cascade mittels t-Test

autoprune vs. cand			
Problem	1	2	3
building	(a 0,4)	—	(c 0,1)
cancer	a 0,0	—	(a 9,3)
card	a 0,1	a 0,0	(c 0,4)
diabetes	a 1,1	—	a 1,1
flare	c 0,0	a 0,0	—
gene	a 0,0	—	a 0,3
glass	—	a 7,1	c 0,0
heart	a 0,5	a 0,0	c 0,1
hearta	—	—	—
heartac	a 3,2	c 0,7	—
heartc	c 0,3	a 0,0	a 0,1
horse	—	—	a 0,0
soybean	c 0,0	(a 0,0)	c 0,0
thyroid	a 0,0	(a 0,0)	a 0,0

autoprune vs. cascade			
Problem	1	2	3
building	a 5,9	a 0,0	a 0,0
cancer	a 0,0	(a 0,0)	—
card	a 0,0	a 0,0	C 3,3
diabetes	a 2,6	C 0,1	—
flare	C 0,4	a 0,0	—
gene	a 0,0	—	(a 0,8)
glass	a 0,0	a 1,7	C 8,4
heart	a 5,1	a 0,3	C 1,6
hearta	a 8,6	—	—
heartac	a 0,8	C 2,8	—
heartc	C 0,2	a 0,0	a 0,1
horse	—	—	a 0,4
soybean	C 0,0	a 0,0	C 0,0
thyroid	a 0,0	(a 0,0)	(a 0,0)

Der Aufbau der Tabellen ist analog zu Tabelle 6.8. Hier wird das autoprune-Verfahren auf Reinschicht-Pivotarchitekturen mit den Verfahren cand und cascade verglichen.
Links (autoprune/cand): 12 mal kein signifikanter Unterschied, 21 mal autoprune besser („a", davon 4 dubios), 9 mal cand besser („c").
Rechts (autoprune/cascade): 9 mal kein signifikanter Unterschied, 24 mal autoprune besser („a", davon 4 dubios), 9 mal cascade besser („C").

in der betrachteten Problemsammlung kein Problem, bei dem ein Kandidatenlernverfahren für alle drei Problemvarianten mindestens gleich gut wie das autoprune-Verfahren ist. Damit scheint die Beschneidung das beste Verfahren zu sein, um (nach grober Vorauswahl einer geeigneten Netzarchitektur) automatisch zu einer guten Lösung von Lernproblemen des hier betrachteten Diagnosetyps zu kommen.

7.1.4 Gesamtvergleich

Die bisherigen Vergleiche von Verfahren wurden immer paarweise Verfahren mittels statistischer Tests verglichen, um die statistische Signifikanz der beobachteten Unterschiede zu finden. Für einen schnellen Gesamtüberblick über verschiedene Verfahren gebe ich in diesem Abschnitt eine stark vereinfachte Darstellung. Darin werden die mittleren Testfehlerwerte für mehrere Verfahren in Relation zu einem festen Verfahren (und zwar frühem Stoppen mit den Reinschicht-Pivotarchitekturen) angegeben. Zu jedem Verfahren ist außerdem ein Fehlerbalken angegeben, der die Standardabweichung der Testfehler zeigt. Eine Schar von solchen Angaben für 6 Verfahren ist für jedes der 14 betrachteten Probleme in den Abbildungen 7.4 und 7.5 gezeigt. Bei den Einträgen für autoprune wurde die Pivot-Architektur

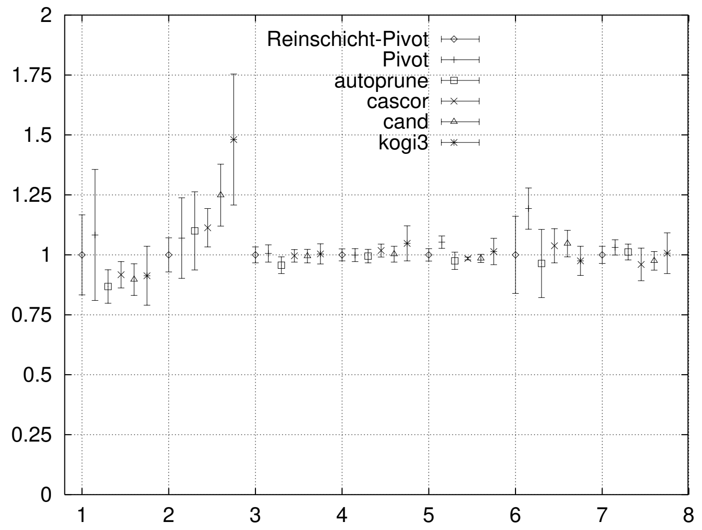


Abbildung 7.4: x-Achse zeigt Lernproblem (1: building, 2: cancer, 3: card, 4: diabetes, 5: flare, 6: gene, 7: glass), y-Achse zeigt Mittelwert des Testfehlers verschiedener Verfahren relativ zum Verfahren „frühes Stoppen mit Reinschicht-Pivotarchitektur“, sowie die jeweils zugehörige Standardabweichung als Fehlerbalken.

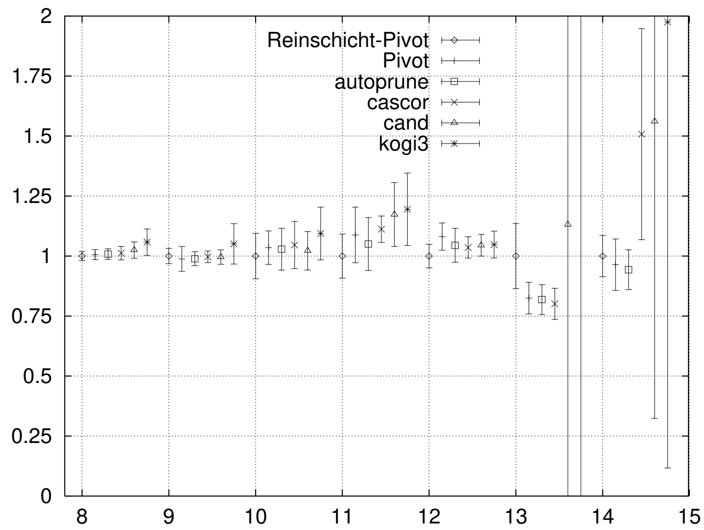


Abbildung 7.5: x-Achse zeigt Lernproblem (8: heart, 9: hearta, 10: heartac, 11: heartc, 12: horse, 13: soybe-
an, 14: thyroid), y-Achse zeigt Mittelwert des Testfehlers verschiedener Verfahren relativ zum Verfahren „frühes Stoppen mit Reinschicht-Pivotarchitektur“, sowie die jeweils zugehörige Standardabweichung als Fehlerbalken.

zugrundegelegt. Es wurde für jedes Problem über alle drei Varianten des Problems gemittelt. Wie wir sehen, sind die Unterschiede zwischen den Verfahren in den meisten Fällen nur klein. Jedoch empfiehlt sich auch in dieser Darstellung das autoprune-Verfahren als das robusteste: Es hat nicht nur niedrige Fehler, sondern auch eine geringe Variation derselben. Dies steht im Gegensatz zu den Kandidatenverfahren, die zumindest auf den Problemen 13 (soybean) und 14 (thyroid) starke Schwankungen in der Qualität ihrer Lernergebnisse aufweisen. Dennoch ist auch in dieser Darstellung zu erkennen, daß keines der Verfahren gleichmäßig überall einem anderen überlegen ist. Wir müssen also weiterhin erstens nach besseren Verfahren suchen und zweitens lernen zu charakterisieren, für welche Probleme sich welche Verfahren besonders gut eignen.

7.2 Zusammenfassung und Beiträge dieser Arbeit

In Abschnitt 3.1 habe ich nachgewiesen, daß bislang in den allermeisten Fällen keine brauchbare empirische Auswertung neu vorgeschlagener neuronaler Lernverfahren vorgenommen wird. Da dieser Zustand der wissenschaftlichen Methode Hohn spricht, war er der Ausgangspunkt für die folgenden Beiträge dieser Arbeit:

1. Als Basis für zukünftige bessere empirische Auswertungen habe ich die erste umfangreiche Benchmark-Sammlung zusammengestellt, die speziell für neuronale Lernverfahren gemacht ist. Darin sind nicht nur die Daten enthalten, sondern es werden auch Konventionen vorgegeben, die für methodische Korrektheit und exakte Reproduzierbarkeit der Experimente sorgen und außerdem eine direkte Vergleichbarkeit von Ergebnissen verschiedener Forscher herbeiführen sollen.
2. Als Ergänzung hierzu habe ich untersucht, wie die Anwendung des t-Tests beim Vergleich von Lernergebnissen erfolgen sollte, damit die Testergebnisse zuverlässig sind.
3. Auf Basis dieser Benchmarkdaten und -methodik habe ich vier Serien von Versuchen durchgeführt: Eine zur Bewertung der Effizienz und Effektivität verschiedener Stoppkriterien für Lernen mit frühem Stoppen, eine zweite zur Untersuchung des frühen Stoppens als Lernverfahren, eine dritte zur Untersuchung additiver Kandidatenlernverfahren und eine vierte zur Untersuchung von Verfahren zur Beschneidung von Verbindungen. Die Daten aus diesen Versuchsreihen stehen als Vergleichsbasis öffentlich zur Verfügung (siehe Anhang A). Solche Vergleichsdaten fehlten bislang.
4. In den Bereichen Stoppkriterien, additive Verfahren und subtraktive Verfahren habe ich jeweils eine oder mehrere Varianten bekannter Verfahren als mögliche Verbesserungen hergeleitet.
5. Jedes neue und auch jedes bekannte Verfahren wurde als automatisches Lernverfahren (d.h. ohne freie Parameter, insbesondere für Kriterien zum Abbruch oder Phasenwechsel des Trainings) formuliert. Dies wird in der Literatur bisher selten getan, ist aber eine Voraussetzung dafür, daß der erfolgreiche Einsatz eines Verfahrens ohne schwarze Magie und ohne exzessives Probieren nach Versuch-und-Irrtum möglich wird.
6. Innerhalb der Versuchsserien wurden sodann die bekannten Verfahren und meine neuen Varianten verglichen. Obwohl keine der Modifikationen eine durchgängige Verbesserung gegenüber den bekannten Verfahren darstellt, sind die meisten zumindest für einige Fälle ein Fortschritt. Ich habe ungefähre Kriterien dafür angegeben, welches diese Fälle sind. Offenbar realisieren also die verschiedenen Verfahren je eine unterschiedliche Art von Bias, die bei unterschiedlichen Problemen angemessen ist.
7. Die Versuchsserien ermöglichten in ihrer Gesamtheit erstmalig einen großangelegten Vergleich von frühem Stoppen, additiven und subtraktiven Lernverfahren, bei dem sich die subtraktiven Verfahren als diejenigen herausgestellt haben, die im Mittel die besten Lösungen produzieren. In der Literatur wird bei der Behandlung subtraktiver Verfahren häufig vor allem der Aspekt der Verkleinerung der Netze herausgestellt und eine Verbesserung der Qualität nur vage und ohne eigene Überzeugung der Autoren als Möglichkeit angedeutet. Demgegenüber hat die vorliegende Arbeit nachgewiesen, daß Beschneidungsverfahren erheblich mehr Aufmerksamkeit verdienen, als ihnen gegenwärtig zuteil wird.

7.3 Ausblick

Die in dieser Arbeit untersuchten konstruktiven Lernverfahren sind noch recht primitiv. Bei den subtraktiven Verfahren gibt es vor allem drei Gebiete, auf denen noch Verbesserungen nötig sind:

1. Es sollten Beschneidungskoeffizienten gefunden werden, die wenigstens zum Teil auch die Wechselwirkungen zwischen verschiedenen Verbindungen berücksichtigen, ohne dabei aber einen so hohen Berechnungsaufwand zu erfordern, wie das OBS-Verfahren.
2. Die automatische Wahl der Beschneidungsstärke ist auch nach der Einführung von lprune ein nur zum Teil gelöstes Problem. Insbesondere für den ersten Beschneidungsschritt scheint diese Wahl sehr kritisch zu sein, ohne daß lprune hierzu einen wesentlichen Beitrag liefern kann. Vermutlich können Verfahren mit guter automatischer Steuerung der Beschneidungsstärke noch wesentlich bessere Ergebnisse erzielen als die in dieser Arbeit vorgestellten.
3. Schließlich wäre es gut, wenn Beschneidungsverfahren die Wahl der Ausgangstopologie noch mehr vereinfachen würden. Es ist unklar, ob dies bereits durch eine gute automatische Wahl der Beschneidungsstärken geleistet würde oder ob dazu die Kombination mit additiven Verfahren nötig ist.

Die additiven Verfahren haben mit der Idee des Kandidatentrainings eine gute Grundlage für weitere Verbesserungen, die vor allem auf zwei Gebieten nötig wären:

1. Es wäre wünschenswert, wenn additive Verfahren eine allgemeinere Klasse von Netztopologien erzeugen könnten als nur Netze mit genau einer verborgenen Schicht oder kaskadierende Netze. Ein Ansatz in diese Richtung ist kogi9, das aber noch der Verbesserung bedarf. Eine Hauptschwierigkeit besteht in der kombinatorischen Explosion der Anzahl möglicher Folgenetze bei allgemeineren Architekturen.
2. Für eine feinkörnigere Auswahl der Netztopologie sollten additive Verfahren mit Beschneidungsverfahren verbunden werden. Dabei ergeben sich allerdings heikle neue Fragen bei den Abbruch- und Übergangskriterien.

Möglicherweise sind diese Probleme im Rahmen monolithischer Lernverfahren nicht zufriedenstellend lösbar. In diesem Fall bieten nur genetische Algorithmen und andere Verfahren des evolutionären Programmierens eine hinreichend allgemeine Basis für das automatische Lernen mit dynamischen Netzarchitekturen. Vermutlich gibt es aber noch reichlich Spielraum zur Verbesserung von Lernverfahren, die nur ein einzelnes Netz verwenden.

Die übergeordnete Frage, welche Art von Bias für welches Anwendungsproblem gut geeignet ist, und wie man diesen Bias in einem neuronalen Algorithmus realisiert, bleibt vorläufig ebenfalls offen.

Wenn diese Fragen beantwortet und die entsprechenden Verfahren bekannt sind, kann die Neuroinformatik, zumindest was die Lernverfahren für praktische Problemlösungen angeht, die in Kapitel 1 beschriebenen frühen Phasen ihrer Entwicklung hinter sich lassen.

Kapitel 8

Parallelrechnerei

Die wahre Besonderheit der Moderne
ist ein enormer Beschleunigungsvorgang des Veraltens.
Hans-Georg Gadamer

While SIMD machines are easy to program
they are notoriously difficult to program
for high performance.
On the other hand, it is often difficult
to produce correct code for MIMD machines,
although the incorrect code sometimes achieves
very high performance.
David E. Schimmel

Dieses Kapitel gibt die Einführung in den zweiten Teil der Arbeit, der sich mit effizienter automatischer Übersetzung konstruktiver neuronaler Algorithmen für Parallelrechner befaßt. Ich gebe zunächst einige Begriffsdefinitionen aus dem Bereich der Parallelrechnerei (Abschnitt 8.1) und beschreibe die beiden Hauptprobleme, die für effizientes paralleles Rechnen zu lösen sind (Abschnitt 8.2). Nach einem Überblick über die Entwicklungstendenzen bei Parallelrechnern (Abschnitt 8.3) gebe ich einen Überblick über den Stand der praktischen Forschung auf den für diese Arbeit relevanten Gebieten (Abschnitt 8.4). Zuletzt beschreibt Abschnitt 8.5 den Aufbau und die Beiträge der Arbeit. Das Kapitel zeigt, daß die Hauptprobleme der Parallelrechnerei, nämlich Datenlokalität und Lastbalance, für unregelmäßig strukturierte Anwendungsprobleme noch kaum beherrscht werden. Deshalb sind Beiträge zu ihrer Lösung, wie diese Arbeit sie liefert, auch dann willkommen, wenn sie sich nur auf ein eingeschränktes Anwendungsgebiet beziehen.

8.1 Einführung und Definitionen

Unter einem *Parallelrechner* verstehen wir einen Rechner mit P gleichen Prozessoren, die gemeinsam ein Programm abarbeiten können. Wir betrachten dabei vor allem *massiv parallele* Rechner, deren Prozessoranzahl P in der Größenordnung von Hunderten oder Tausenden liegt. Im Gegensatz zu verteilten Systemen wird bei der Programmierung von Parallelrechnern die Prozessoranzahl als bekannt und konstant und jeder Prozessor als zuverlässig und jederzeit dienstbereit angenommen. Außerdem sind die vorhandenen Kommunikationsverbindungen der Prozessoren untereinander gleichmäßig angeordnet und von gleicher Leistung.

Parallelrechner sollten skalierbar, d.h. so konstruiert sein, daß sich durch Erhöhung der Prozessoranzahl ihre tatsächliche Leistungsfähigkeit vergrößern läßt. Um dies zu erreichen, muß ein in irgendeiner

Form *verteilter Speicher* verwendet werden, weil auf heutige Halbleiterspeicher im wesentlichen immer nur ein Zugriff gleichzeitig abgewickelt werden kann, so daß ein solcher Speicher bei Erhöhung der Prozessorzahl zum Leistungsengpaß wird. Verteilter Speicher heißt, der Hauptspeicher des Rechners ist in Teile aufgeteilt, auf die unabhängig voneinander echt gleichzeitig zugegriffen werden kann und die in der Regel je einem Prozessor beigeordnet sind. Jeder Prozessor hat die Möglichkeit, in allen Speicherteilen zu lesen und zu schreiben. Diese Speicherteile sind jeweils einem Prozessor zugeordnet und die Prozessoren werden über ein *Kommunikationsnetz* miteinander verbunden. Nichtlokale Speicherzugriffe erfolgen dann per *Nachrichtenaustausch* (*message passing*). In der Regel ist an jedem Prozessor ein zusätzlicher Cache-Speicher vorhanden, was zu einer mindestens vierstufigen Speicherhierarchie führt: Register, Cache, lokaler Speicher, entfernter Speicher. Wegen ihrer mangelhaften Skalierbarkeit werden Architekturen, die auf physikalisch gemeinsamem Speicher basieren, in dieser Arbeit weitgehend außer acht gelassen. Ein Speicherzugriff heißt *lokal* wenn er aus dem Cache oder dem lokalen Speicher erfolgt, andernfalls heißt der Zugriff *entfernt*. Registerzugriffe zählen nicht als Speicherzugriffe. Die Unterscheidung zwischen Cache-Zugriffen und Zugriffen auf den restlichen lokalen Speicher werden wir nicht separat betrachten. Die *Datenlokalität* eines Programmablaufs gibt an, wie hoch der Anteil lokaler Speicherzugriffe an der Menge aller Speicherzugriffe ist.

Ein Kommunikationsnetz verbindet jeden Prozessor nur mit einem kleinen Teil der übrigen Prozessoren direkt, weil andernfalls die Anzahl der Leitungen des Kommunikationsnetzes quadratisch mit der Anzahl der Prozessoren wachsen würde. Zur Kommunikation nicht direkt verbundener Prozessoren müssen die Nachrichten über Zwischenstationen *geroutet* werden. Das Verbindungsmuster des Netzes heißt *Kommunikationstopologie*. Die gebräuchlichsten sind 1- bis 3-dimensionale Gitter (Kette, Flachgitter, Raumgitter) und Tori (Ring, Torus, Hypertorus), Hyperwürfel, *perfect shuffle*, Schmetterlingsnetz (*butterfly network*) und Kombinationen davon, wie Würfel von Ringen (*cube-connected cycles*), Würfel von Gittern (*mesh of meshes*) oder Ring von Ringen. Alle diese Netztopologien unterscheiden sich in Parametern wie Anschlußzahl pro Prozessor, *Durchmesser* (maximale Anzahl von einer Nachricht zu passierender Leitungen zwischen zwei Prozessoren), *Bisektionsbreite* (Mindestanzahl von Leitungen, die durchschnitten werden müssen, um das Netz in zwei Teile zu zerlegen), mittlere und maximale *Verkehrsdichte* (Anzahl Nachrichten pro Leitung bei Kommunikation jedes Prozessors mit einem anderen), mittlere und maximale *Leitungslänge* (bei vorgegebener physikalischer Anordnung) und Varianz der Leitungslänge, *Planarisierbarkeit* und *Kubisierbarkeit*, *Fehlertoleranz* (mittlere Anzahl zu durchtrennender Leitungen, bis irgendein Paar von Prozessoren unverbunden ist), *Verklemmungsfreiheit*, *Routingfunktion* (Regel für die Auswahl einer Ausgangsleitung zur Vermittlung einer Nachricht ans Ziel) und anderen. Diese Parameter werden hier nicht im einzelnen diskutiert, sie bestimmen aber gemeinsam mit der hard- und softwaremäßigen Realisierung der eigentlichen Kommunikation die aus Programmierungssicht entscheidenden Netzparameter *Bandbreite* und *Latenzzeit* für verschiedene Muster von Kommunikationsoperationen. Die Bandbreite gibt die Datenmenge an, die ein Prozessor im Mittel pro Zeiteinheit einem anderen zusenden kann, wenn unendlich viele Daten transportiert werden müssen. Die Latenzzeit beschreibt die „Einschwingzeit", d.h. die Dauer der Übertragung der kleinstmöglichen Datenmenge (1 Byte). Diese Werte hängen nicht nur vom Kommunikationsnetz allein, sondern auch davon ab, wieviele und welche Prozessoren zugleich miteinander wieviel Daten austauschen. Wenn nichts anderes angegeben ist, beziehen sich die Werte auf den Fall einer zufälligen *Permutation*, d.h. daß jeder Prozessor genau eine Nachricht sendet und eine empfängt, wobei die Prozessorpaare gleichverteilt zufällig sind und alle Nachrichten gleich groß. Wird keine Nachrichtengröße angegeben, so ist ein Wort (32-bit) anzunehmen. Die Latenzzeit von Lese- und Schreibzugriffen ist meist unterschiedlich, ihre Bandbreite meist gleich.

Es gibt zwei grundsätzlich verschiedene Arten von Parallelrechnern: SIMD und MIMD [119]. Bei SIMD-Rechnern (*single instruction multiple data*) arbeiten die parallelen Prozessoren nur als Rechenwerke, die von einem einzigen Steuerwerk (Kontrollprozessor, Steuerprozessor) dirigiert werden. Zu jedem Zeitpunkt führt jeder Prozessor entweder die aktuell vom Steuerprozessor zentral angebotene Instruktion aus, oder hat sich abgeschaltet. Die Prozessoren arbeiten also befehlsweise synchron (*Lockstep-Modus*). Die Art der von diesen Rechnern bereitgestellten Parallelität nennt man auch *Da-*

tenparallelität, weil immer derselbe Befehl parallel für mehrere Sätze von Operanden ausgeführt wird. Im Gegensatz dazu ist bei einem MIMD-Rechner (*multiple instruction multiple data*) jeder Prozessor ein kompletter Rechner, der völlig autonom ein Programm abarbeiten kann, weshalb die Art der von MIMD-Rechnern realisierten Parallelität auch als *Prozeßparallelität* bezeichnet wird.

Die Begriffe Daten- und Prozeßparallelität sind auch auf parallele Programmiermodelle anwendbar. Ein Programmiermodell, das die parallele Anwendung derselben Operation auf zahlreiche Datenobjekte zur Formulierung von Parallelität heranzieht, heißt datenparallel. Wird das Aufrufen verschiedener, unabhängiger, paralleler Prozesse zur Formulierung der Parallelität verwendet, heißt das Modell prozeßparallel. Ein Programmiermodell kann zugleich datenparallel und prozeßparallel sein, indem es beide Arten von parallelen Operationen anbietet. Datenparallelität kann in einem Programmiermodell weiter gefaßt sein, als sie in SIMD-Rechnern realisiert ist: Die parallele Operation muß nicht elementar sein (*reine Datenparallelität*, *feinkörnige Datenparallelität*), sondern kann größere Programmabschnitte umfassen (*grobkörnige Datenparallelität*, *locker synchrone Parallelität*), z.B. komplette Prozeduraufrufe. Der Unterschied zur Prozeßparallelität liegt darin, daß datenparallele Arbeitsabschnitte nicht miteinander interagieren und nur an ihrem Ende eine (implizite) Synchronisation auftritt, während prozeßparallele Arbeitsabschnitte miteinander über Kommunikationsoperationen in beliebige, komplizierte Wechselwirkungen treten können. Das Begriffspaar kann analog auf Algorithmen (genauer: bestimmte Formulierungen von Algorithmen) angewendet werden.

Die wichtigste zusammengesetzte parallele Operation ist die *Reduktion*. Darunter versteht man die wiederholte parallele Anwendung eines assoziativen Operators auf Operandenpaare aus einer Reihung von n Operanden, um in $\lceil \log(n) \rceil$ parallelen Schritten daraus ein Gesamtergebnis zu berechnen, z.B. die Summe oder das Maximum eines Vektors von Zahlen.

Die Instruktionsfolge, die zwischen zwei Synchronisationspunkten parallel von einem gedachten oder realen parallelen Prozeß bearbeitet wird, heißt eine *parallele Arbeitseinheit*. Die Summe dieser Folgen über alle Prozesse heißt *paralleler Abschnitt*. Wenn die Arbeitseinheiten eines parallelen Abschnitts unterschiedliche Teile derselben zusammengehörenden Datenstruktur bearbeiten, so nennen wir letztere eine *parallele Datenstruktur*.

Als *Granularität* (*Korngröße*) eines parallelen Algorithmus bezeichnet man die mittlere Größe einer parallelen Arbeitseinheit, gemessen in Instruktionen. Algorithmen und Programme heißen *feinkörnig parallel*, wenn ihre Granularität unter 10^2 Instruktionen liegt und *grobkörnig parallel* bei über 10^3 Instruktionen. Diese Grenzziehung wird von der verfügbaren Technologie bestimmt: Manche Maschinen eignen sich nicht zur effizienten Ausführung feinkörnig paralleler Programme, andere tun es. Ein feinkörnig paralleler Algorithmus mit verschachtelten Ebenen von Parallelität kann oft als grobkörnig paralleles Programm formuliert werden, indem die innerste Ebene der Parallelität nicht genutzt wird.

Wir nennen ein Problem *regelmäßig*, wenn die Größen der parallelen Arbeitseinheiten im selben parallelen Abschnitt sich in fast allen Fällen um höchstens Faktor 2 unterscheiden; andernfalls heißt das Problem *unregelmäßig*. Wir nennen ein Problem ebenfalls *unregelmäßig in den Zugriffsmustern*, wenn sich die Zugriffsmuster auf die Daten in den parallelen Abschnitten in vielen Fällen nicht durch affine Ausdrücke beschreiben lassen. Unter einem *Problem* wird dabei eine Kombination aus einem Programm und seinen Eingabedaten verstanden. Regelmäßige Probleme sind dadurch charakterisiert, daß die *Lastbalance*, d.h. das gleichmäßige Verteilen der Arbeit auf die vorhandenen Prozessoren dadurch erledigt werden kann, daß jeder Prozessor dieselbe Anzahl von Teilaufgaben zur Bearbeitung erhält. Bei unregelmäßigen Problemen sind zur Erzielung von Lastbalance Maßnahmen zur *Lastbalancierung* notwendig, d.h. das Verteilen der Arbeitseinheiten gemäß ihrer Größe anstatt nur ihrer Zahl. Die *Lastbalance* als quantitatives Maß gibt die von einem Programm erzielte Ausführungsgeschwindigkeit als Anteil der Geschwindigkeit an, die sich bei optimaler Verteilung der Arbeit auf die Prozessoren unter ansonsten gleichen Bedingungen eingestellt hätte.

Ein unregelmäßiges Problem heißt *dynamisch*, wenn die zur Lastbalancierung sinnvolle Arbeitsverteilung sich bei nacheinanderfolgenden Wiederholungen des textuell gleichen parallelen Abschnitts ändern kann und dies auch tatsächlich mindestens einmal während des Programmablaufs tut.

Wichtige Kenngrößen zur Charakterisierung der Qualität einer parallelen Problemlösung sind *Beschleunigung* (*speedup*) und *Effizienz*. Dabei wird die Laufzeit T_P eines Problems in einer parallelen Implementierung mit P Prozessoren verglichen mit der Laufzeit T_1 einer sequentiellen Implementierung. Es gilt: Beschleunigung $B = T_1/T_P$ und Effizienz $E = T_1/(P \cdot T_P)$. Damit der Vergleich aussagekräftig ist, müssen mehrere Bedingungen erfüllt sein: Es sollte ein kompletter Programmlauf gemessen werden, nicht nur ein algorithmischer Kern; für T_1 muß der gleiche Prozessor zugrundegelegt werden wie für T_P; jedoch ist für T_1 nicht das gleiche Programm zu verwenden wie für T_P, sondern eines, wie man es für einen sequentiellen Rechner als optimal erachtet, möglicherweise mit einem völlig anderen Algorithmus [24]. Im Prinzip kann die Effizienz einer parallelen Implementierung größer als 1 sein (*superlinear speedup*), wenn nämlich Effekte wie z.B. erhöhte Gesamtgröße von Cache-Speichern im parallelen Fall eine höhere mittlere Leistung der einzelnen Prozesoren gegenüber dem Einprozessor-Fall bewirken.

Die konkrete Messung der Leistung paralleler Implementierungen ist unter anderem deshalb nötig, weil die bislang zur theoretischen Analyse von Algorithmen verwendeten Modelle für parallele Maschinen die für die tatsächliche Leistung entscheidenden Aspekte realer Maschinen (z.B. Bandbreite und Latenzzeit der Kommunikation) meist völlig ignorieren. So nimmt das bisher populärste Modell, die P-RAM Maschine [121], beispielsweise eine gleichförmige Zugriffszeit für jeden Prozessor auf jede beliebige Speicherstelle an. Erst jüngst werden verfeinerte Modelle verwendet, z.B. das logP-Modell [83] für gegenwärtige Maschinen oder *spatial machines* [113] für zukünftige, bei denen die Leitungslänge der limitierende Faktor ist. Eine theoretische Beurteilung verschiedener Implementierungsformen des gleichen Algorithmus ist jedoch auch mit diesen Modellen nur in trivialen Fällen möglich; die Berechnungen sind zu kompliziert, um eine geschlossene Lösung erhalten zu können.

Als Zugänge zu Literatur über die oben eingeführten Begriffe und für eine genauere Beschreibung siehe [7, 275].

8.2 Hauptprobleme der Parallelrechnerei

Aus einer geeigneten Perspektive betrachtet, gibt es nur einen einzigen Grund, weshalb ein paralleles Programm eine schlechte Effizienz E aufweisen kann: Die Prozessoren verbringen einen zu kleinen Teil ihrer Zeit damit, die eigentlich produktiven Programmbefehle zu bearbeiten. Dafür kann es zwei Gründe geben: Entweder die Prozessoren „drehen Däumchen“ (d.h. warten auf irgendetwas) oder sie führen redundante oder sonstwie im Sinne der eigentlichen Problemlösung nicht unmittelbar produktive Befehle aus. In dieser Arbeit nehmen wir an, der Anteil unproduktiver Befehle sei durch entsprechend optimierte Software des Betriebssystems und einen guten Algorithmus bereits minimiert. Dann bleibt als Grund schlechter Effizienz nur die Wartezeit. Diese kann wiederum in zwei Formen auftreten: Warten auf Speicherzugriffe[1] und Warten auf Synchronisation. Den Fall, daß ohnehin nicht genügend Parallelität im Problem vorhanden ist, um alle Prozessoren zu beschäftigen, lassen wir außer acht. Das Warten auf Speicherzugriffe tritt umso stärker in Erscheinung, je höher die Ebene in der Speicherhierarchie ist, über die der Zugriff erfolgt. Es muß also vor allem der Zugriff auf entfernten Speicher vermieden werden. Das Warten auf Synchronisation ist vor allem dann sehr effizienzschädigend, wenn einzelne Prozessoren für den Programmabschnitt vor dem Synchronisationspunkt wesentlich länger brauchen als die meisten anderen, weil dann auf vielen Prozessoren eine lange Wartezeit anfällt.

Aus dieser Betrachtungsweise ergeben sich, wenn ein guter paralleler Algorithmus gegeben ist, die folgenden zwei technischen Hauptprobleme der Parallelrechnerei[2]:

[1] Hierunter kann im weiteren Sinne auch Ein- und Ausgabe gefaßt werden; diese lasse ich jedoch in meiner Arbeit außer Acht.

[2] Diese Probleme beschreiben Eigenschaften sowohl des Algorithmus wie auch seiner technischen Realisierung. Wir betrachten hier nur den letzteren Aspekt.

- **Datenlokalität.** Daten und Prozesse müssen so über die Prozessoren verteilt werden, daß möglichst selten Zugriffe auf Datenelemente erfolgen, die nicht im lokalen Speicher liegen. Dies ist vor allem dann schwierig, wenn die Speicherzugriffsmuster der parallelen Aktivitäten unregelmäßig sind. Quellen und Ziele unvermeidlicher nichtlokaler Zugriffe müssen sich ungefähr gleichmäßig über die Prozessoren verteilen, um weitere Vergrößerung der Zugriffszeiten durch Zugriffskonflikte zu vermeiden.

- **Lastbalance.** Die Arbeitseinheiten der parallelen Abschnitte müssen so auf die Prozessoren verteilt werden, daß jeder Prozessor ungefähr gleich lange zu tun hat. Dies muß für jeden parallelen Abschnitt, an dessen Ende synchronisiert werden muß, einzeln erreicht werden. Das ist vor allem dann schwierig, wenn die einzelnen Arbeitseinheiten verschieden lang dauern. Im schlimmsten Fall ist die Dauer zu Beginn des Abschnitts nicht bekannt.

Die Arbeiten im Bereich paralleler Hardware verfolgen den Ansatz, die Konsequenzen dieser Probleme abzumildern, indem schnelle Kommunikations- und Synchronisationsoperationen angeboten werden; zum Stand der Forschung siehe den nächsten Abschnitt. Die Arbeiten bei der Software versuchen, das Auftreten der Probleme von vornherein zu verringern, indem Programme so auf den Parallelrechner übersetzt werden, daß Datenlokalität und Lastbalance möglichst hoch ausfallen. Das Lösen der beiden Probleme steht dabei teilweise im Widerspruch, weil ein Umverteilen von Arbeit von einem Prozessor auf einen anderen eine hohe Datenlokalität beeinträchtigt. Zum Stand der Forschung siehe den übernächsten Abschnitt.

8.3 Parallele Hardware

Dieser Abschnitt beschreibt die für diese Arbeit wichtigsten Eigenschaften bisheriger Parallelrechner und die Tendenzen zukünftiger Entwicklung; dabei werden jeweils die Konsequenzen für den Übersetzerbau auf parallelen Maschinen diskutiert. Einige Arten von Rechnern werden ausgelassen, weil sie von geringer praktischer Bedeutung sind (siehe aber den Abschnitt 1.4). Ich beschränke mich weitgehend auf kommerziell angebotene Rechner. Ein eigener Unterabschnitt ist spezieller Hardware für neuronale Netze gewidmet.

8.3.1 Frühe MIMD-Rechner

Das wichtigste Merkmal älterer MIMD-Rechner im Hinblick auf ihre Programmierbarkeit in einer maschinenunabhängigen, höheren Programmiersprache, ist die sehr hohe Latenzzeit ihrer Kommunikationsoperationen. Hier einige Beispiele:

Der Intel iPSC/2 [7, Seite 488f] und der Ncube/ten [150] waren etwa 1986 die ersten kommerziell angebotenen Rechner mit einem Hyperkubus-Verbindungsnetzwerk. Sie wurden beide später durch schnellere Nachfolgemodelle mit ähnlicher Architektur ersetzt. Der iPSC/860, Nachfolger des iPSC/2, verwendet den schnellen i860 RISC Prozessor; die Zeit nur allein zum Absenden einer Nachricht ins Netzwerk (*Startzeit*) beträgt 1500 Taktzyklen [275, Seite 632], in denen der Prozessor im Prinzip 1500 Gleitkomma-Multiplikationen ausführen kann, das Durchreichen einer ganzen Marge von Nachrichten durch mehrere Zwischenprozessoren, wie es bei der Kommunikation einer zufälligen Permutation auftritt, verlangt noch mehr Zeit. Beim Ncube/2 wird ein speziell entwickelter Prozessor eingesetzt; die Latenzzeit beträgt etwa 420 Gleitkomma-Multiplikationen [352].

Eine Reihe von Rechnern wurde ab etwa 1986 basierend auf dem INMOS T414 Transputer (später auf dem T800) gebaut. Ein Beispiel ist der Parsytec Supercluster SC-1024 FT von 1990 auf T800-Basis mit gitterförmiger Struktur des Kommunikationsnetzes und einer Latenzzeit von etwa 2000 Gleitkomma-Multiplikationen [352].

Trotz hoher theoretischer Rechenleistung kann die effektive Rechenleistung dieser Maschinen selbst mit handoptimierten Programmen nur bei manchen Anwendungen befriedigen [25]. Ein feinkörnig paralleles Programm in einer maschinenunabhängigen Sprache kann auf solchen Maschinen nicht in effizienten Code übersetzt werden.

8.3.2 SIMD-Rechner

SIMD-Rechner haben prinzipiell einige bestechende Vorteile: Sie sparen im Vergleich zu MIMD-Rechnern die hohen Kosten für die wiederholte Speicherung des Programms auf jedem Prozessor und für die Chipfläche zur Realisierung der zahlreichen Steuerwerke. Die dadurch vereinfachte Prozessorarchitektur ermöglicht sehr hohe Prozessoranzahlen und durch die inhärent synchrone Arbeitsweise der Maschine läßt sich für Programme, die eine häufige Synchronisation erfordern, viel Synchronisationszeit einsparen. Die Programmierung von SIMD-Maschinen ist aufgrund der synchronen Arbeitsweise viel einfacher, weil der Maschinenzustand leichter beobachtbar ist und keine laufzeitabhängigen Fehler (*race conditions*) auftreten.

Ein SIMD-Rechner war es dann auch, der den Durchbruch zum massiv parallelen Rechnen geschafft hat: Hillis' Connection Machine [155] von 1985. Ihr Nachfolger, die CM-2 [347] der Firma Thinking Machines, ist eine der wenigen kommerziell erfolgreichen SIMD-Architekturen. Im Gegensatz zu den früheren SIMD-Rechnern ICL DAP [161] und Goodyear MPP [33], die sich eher als Feld-Prozessoren denn als Universalrechner verstanden, kann bei der Connection Machine über ein allgemeines Kommunikationsnetz jeder Prozessor auf den Speicher eines beliebigen anderen Prozessors zugreifen, anstatt nur auf die Nachbarn. Die CM-2 hat nur noch eine Einschränkung der universellen Programmierbarkeit: die Prozessoren können nicht selber Zeiger dereferenzieren, d.h. parallele Speicherzugriffe erfolgen immer auf dieselbe Speicherstelle in allen Prozessoren. Die Connection Maschine war der erste parallele Rechner, der in einer maschinenunabhängigen Weise in der höheren Programmiersprache CmLisp [155] feinkörnig parallel programmiert werden konnte. Die CM-2 hielt längere Zeit den Titel des schnellsten Supercomputers, wobei die dafür vorgewiesene Rechenleistung aber nur bei wenigen Anwendungen erreicht oder angenähert wurde [275, Seite 645].

Die modernste Familie von SIMD-Maschinen, genannt *autonomes SIMD* (*ASIMD*) [45] und hergestellt von der Firma MasPar [44, 266] seit 1990, erlaubt auch die prozessorlokale Zeigerdereferenzierung. Die MasPar MP-1 und die neuere MP-2, die eine höhere Prozessorleistung bei ansonsten gleicher Architektur, insbesondere gleichem Kommunikationsnetz aufweist, sind neben der CM-2 die einzigen kommerziell erfolgreichen SIMD-Maschinen. Die MP-1 ist außerdem trotz ihres Alters der immer noch am besten zwischen Prozessorleistung und Kommunikationsleistung balancierte Rechner für feinkörnig parallele Programme: Im allgemeinen Fall benötigt die Kommunikation einer zufälligen Permutation von je einem 32-bit-Wert pro Prozessor etwa solange wie 30 Gleitkomma-Multiplikationen mit einfacher Genauigkeit. Mit dieser Latenzzeit liegt die MP-1 um ein mehrfaches besser als selbst die kommunikationsschnellsten heute kommerziell verfügbaren MIMD-Rechner und ist etwa dreimal so schnell[3] wie die CM-2. [106] vergleicht die Gesamtleistung von CM-2, MPP, DAP und MP-1 anhand von zwei numerischen Algorithmen.

Es gibt noch eine ganze Reihe anderer SIMD-Rechner. Diese sind aber eher als Spezialarchitekturen aufzufassen, z.B. der GF11 von IBM [37], der für Berechnungen zur Quantenchromodynamik gebaut wurde, der Warp [17] von der Carnegie Mellon Universität und sein Nachfolger iWarp von Intel, die als systolische Arrays [206] für Berechnungen z.B. im Bereich Maschinensehen und Spracherkennung konzipiert sind, oder die CNAPS von Adaptive Solutions [141], die speziell für neuronale Netze konstruiert ist (siehe Abschnitt 8.3.5).

Es gibt drei Gründe, weshalb trotz der obengenannten Vorzüge der SIMD-Maschinen die Zukunft des parallelen Rechnens eindeutig den MIMD-Maschinen gehört:

[3] Setzt man für die MP-1 die Adressierungsbeschränkung an, die für die CM-2 unvermeidlich ist, so steigt der Unterschied auf etwa den Faktor 5.

1. Da eine SIMD-Maschine immer nur ein Programm zu jedem Zeitpunkt bearbeiten kann, eignet sie sich nur schlecht für einen Rechenzentrumsbetrieb mit vielen Benutzern. Insbesondere interaktive Programme können nur mit Einschränkungen benutzt werden.
2. Ebenfalls aufgrund des Einprogramm-Betriebs ist es schwierig, eine SIMD-Maschine immer voll auszulasten; es ist nicht möglich zur Ausführung eines Programms auf einem kleinen Problem ein Teilsegment der Maschine abzuspalten und den Rest für andere Programme einzusetzen.
3. Die Kostenvorteile durch Ersparnis an Chipfläche und Hauptspeicher werden mehr als aufgewogen durch die hohen Kosten für die Entwicklung eines speziellen SIMD-Prozessors. Durch die kleinen Fertigungsstückzahlen paralleler Rechner lastet auf jedem Prozessor ein enorm hoher Deckungsbeitrag für den Chipentwurf. MIMD-Rechner können hingegen in großen Stückzahlen gefertigte Standard-Mikroprozessoren einsetzen, die sehr leistungsfähig und zugleich preiswert sind.

Dennoch sind die SIMD-Maschinen den MIMD-Rechnern bis heute voraus, was die Balance zwischen Rechen- und Kommunikationsleistung angeht. Insbesondere die MasPar MP-1 hat hierin noch immer eine Generation Vorsprung vor den aktuellen Parallelrechnern (siehe unten). Deshalb eignet sie sich als Grundlage von Forschungsarbeiten, die ein Verhältnis von Kommunikationsgeschwindigkeit zu Rechengeschwindigkeit voraussetzen, wie sie bei zukünftigen Parallelrechnern zu erwarten sein wird.

8.3.3 Neuere MIMD-Rechner

Bei neueren Parallelrechnern zeichnen sich mehrere Trends ab: Es werden sehr leistungsfähige Standard-Mikroprozessoren eingesetzt, die Kommunikationsleistung wird erhöht und es wird spezielle Hardware zur Unterstützung der Kommunikation eingesetzt anstatt die Kommunikation hauptsächlich durch Software zu realisieren.

Der erste Rechner dieser neuen Generation war die CM-5 von Thinking Machines [7, Seiten 509-512]. In dieser Maschine, die 1991 auf den Markt kam, werden Sparc Mikroprozessoren der Firma Sun eingesetzt und die Kommunikation im *fat tree* Netzwerk des Rechners wird ergänzt durch ein eigenes Kontrollnetzwerk, das beispielsweise schnelle Synchronisationsoperationen unterstützt. Jeder Sparc-Prozessor kann von vier Vektorprozessoren ergänzt werden, um höchste Gleitkommaleistung zu erzielen. Ohne Vektorprozessoren ist die Gleitkommaleistung der Maschine relativ niedrig und die Balance mit der Kommunikationsleistung recht gut: Die Zeit zum Starten einer Nachricht entspricht mit sorgfältigst handoptimierter Kommunikationssoftware nur 14 Gleitkommamultiplikationen. Die von Thinking Machines normalerweise ausgelieferte Software benötigt allerdings 360! Dies zeigt den beträchtlichen Einfluß von Softwarezeiten auf die Kommunikationsleistung bei MIMD-Rechnern. Die CM-5 wird wegen wirtschaftlicher Schwierigkeiten der Firma inzwischen nicht mehr gebaut.

Die KSR-1 von Kendall Square Research bot als erster kommerzieller Rechner einen in Hardware realisierten, virtuell gemeinsamen Speicher an, der mit einer als ALLCACHE bezeichneten Technologie realisiert wird. Dabei wird der gesamte verteilte Speicher als konsistenter Cache verwaltet. Die Prozessoren sind dazu über ein Ring-Netzwerk oder bei großen Maschinen über einen Ring von Ringen miteinander verbunden. Der Rechner und sein Nachfolger KSR-2 werden wegen wirtschaftlicher Schwierigkeiten der Firma inzwischen nicht mehr gebaut.

Die Intel Paragon [54] wurde aus dem Prototypen Touchstone Delta entwickelt, der der Nachfolger des iPSC/860 war. Bei dieser Maschine wurde besonders großer Wert auf schnelle Kommunikation (sowohl Bandbreite als auch Latenzzeit) gelegt und deshalb zur Vereinfachung des Routens ein 2-D Gitternetzwerk verwendet und ein schneller Kommunikationsprozessor eingebaut. Die in [160] berichteten Werte für die Kommunikationsleistung, die im Mai 1993 gemessen wurden, liegen jedoch noch weit unter den vom Hersteller angegebenen Werten, die laut Auskunft der Firma Intel etwa Ende 1994 erreicht werden sollen. Gemessen wurde eine Nachrichtenstartzeit von 172μs, angestrebt werden 25μs. Diese Werte entsprechen bei 50Mhz Taktfrequenz und einer Spitzenleistung von einer Gleitkommaoperation pro Takt [7] Latenzzeiten von 8600 bzw. 1250 Gleitkommaoperationen. Diese Unterschiede kommen von noch nicht abgeschlossener Optimierung der Software, einschließlich der richtigen Einbindung des Kommunikationsprozessors.

Meiko CS-2 [306] verwendet wie die CM-5 ein *fat tree* Netzwerk und Sparc-Prozessoren (allerdings neuere Varianten). Unter der Annahme eines 50 Mhz Prozessors, der zwei Takte pro Gleitkommaoperation benötigt, ergibt sich aus den Angaben in [160] eine Latenzzeit von 2200 Gleitkommaoperationen mit der PARMACS Nachrichtensoftware oder 200 mit DMA-Zugriffen. Bei Zufügen der optionalen zwei Vektorprozessoren je Prozessor steigt letzterer Wert auf 800.

Der Prototyp S3.mp von Sun und DEC [269] verfolgt ein anderes Konzept. S3.mp ist ein Baustein aus zwei CPUs und einem Kommunikationsprozessor, der über einen marktüblichen Bus mit herkömmlichen Rechnern verbunden wird. Die Bausteine selbst lassen sich in beliebiger Topologie zu Parallelrechnern verbinden, die bis zu 200 Meter physikalischen Durchmesser haben können. S3.mp unterstützt dynamische Rekonfigurierung des Rechners und stellt die Semantik von virtuell gemeinsamem Speicher zur Verfügung.

Andere moderne MIMD-Rechner sind der Cray T3D [271] mit 3-D Torusnetzwerk und die IBM SP1 [334] und SP2 mit Omega-Netzwerk. Auch diese Rechner verwenden Standard-Mikroprozessoren.

Tabelle 8.1: Ungefähre Latenzzeiten für verschiedene Parallelrechner

Rechner	t_{Start}	t_{Latenz}	Verbergung
Intel iPSC/860	1500		nein
NCUBE nCube/2		420	
Parsytec SC-1024 FT		2000	nein
Thinking Machines CM-2		100	nein
MasPar MP-1		30	nein
MasPar MP-2		100	nein
Intel Paragon, Mai 1993		8600	ja
Intel Paragon, 1995		1250	ja
Meiko CS-2		200	
CS-2 + Vektorproz.		800	
Thinking Machines CM-5	360		
CM-5, optimierte SW	14		
dito + Vektorproz.	500		
Cray Y-MP/C90		(<100)	—

Ungefähre Nachrichtenstartzeiten t_{Start} oder Latenzzeiten t_{Latenz} für mehrere Parallelrechner, ausgedrückt in Anzahl Gleitkommaoperationen pro Prozessor bei Spitzenleistung. Die letzte Spalte gibt an, ob die Hardware eine (teilweise) Verbergung der Latenzzeit unterstützt. Die Zahlenwerte sind möglicherweise sehr ungenau, weil die Interpretation der in der Literatur zugänglichen Daten oft schwierig ist. Es ist zu beachten, daß die Spitzengleitkommaleistung bei den modernen Maschinen auch ohne Kommunikation bei weitem nicht realistisch zu erreichen ist und daß die Latenzzeit bei der Y-MP/C90 praktisch nicht auftritt, da meistens darüber vektorisiert wird.

Die ungefähren Latenzzeiten der Kommunikation im Verhältnis zur maximalen Gleitkommarechenleistung sind für einige Rechner in der Tabelle 8.1 dargestellt. Wie man an der zum Vergleich aufgenommenen Cray Y-MP/C90 sieht, bei der die Speicherzugriffszeiten aus einer „Skalar mal Vektor plus Vektor"–Operation in Latenzzeiten umgerechnet wurden [160, 7], sind auch die besten Parallelrechner einem modernen Vektorrechner in der Latenzzeit hoffnungslos unterlegen: Beim Vergleich der Speicherzugriffszeit des Vektorrechners mit der Latenzzeit einer parallelen Maschine mit 512 Prozessoren ist der angegebene Basiswert bereits auf 512fache Vektorisierung zu beziehen, was die *effektive* Zugriffszeit des Vektorrechners von 100 auf etwa 1 absenkt. Dies erklärt den erheblichen Rückstand, den Parallelrechner trotz ihrer hohen Spitzenrechenleistung (*peak performance*[4]) bei der tatsächlich zu beobachtenden Rechenleistung für reale Anwendungen meistens gegenüber Vektorrechnern aufweisen. Hierzu ein Zahlenbeispiel aus [7, Seite 379]: Eine Cray Y-MP/C90 mit 16 Prozessoren hat eine theoretische Spitzenleistung von etwa 15,2 GFLOPS, eine Intel Touchstone Delta mit 512 Prozessoren liegt bei 20 GFLOPS. Beim Lösen linearer Gleichungssysteme mit 1000 Variablen erreicht die Cray 9,7 GFLOP (64% der Spitzenleistung), die Delta hingegen nur 0,45 GFLOP (2% der Spitzenleistung).

Trotz einiger Fortschritte ist es aufgrund dieser Bedingungen immer noch sehr schwierig, für moderne MIMD-Maschinen feinkörnig parallele maschinenunabhängige Programme in effizienten Code zu übersetzen.

8.3.4 Entwicklungstrends

Derzeit erlangen busgekoppelte Multiprozessorrechner mit kohärenten Caches und bis zu einigen Dutzend Prozessoren eine erhebliche Popularität. Eine beträchtliche Auswahl solcher Maschinen wird bereits angeboten und die Preise sinken ständig. Skalierung in den massiv parallelen Bereich scheidet aber prinzipbedingt für diese Maschinen aus, und sie haben auch schon bei moderaten Prozessoranzahlen schwerwiegende Kommunikationsengpässe für alle Programme, die nicht eine extrem hohe Datenlokalität aufweisen.

Bei den massiv parallelen Rechnern werden vermutlich hardwareunterstützte Maßnahmen zur Verbergung der Latenzzeit die künftige Entwicklung bestimmen. Angesichts der hohen Arbeitsgeschwindigkeit der Prozessoren ist es nur damit möglich, eine akzeptable Balance zwischen Rechenleistung und effektiver Kommunikationsleistung für feinkörnig parallele Programme zu erreichen [389]. Die Ausnutzung der Latenzzeitverbergungsmechanismen durch Vorladebefehle (*prefetch*) kann und muß dann von Übersetzern geleistet werden. . Es müssen Kommunikationsnetzwerke gewählt werden, die dem hohen Nachrichtenaufkommen bei mehreren zugleich ausstehenden Datenanforderungen pro Prozessor gewachsen sind, z.B. mehrstufige Netzwerke.

Die Entwicklung läßt erhoffen, daß das Problem hoher Latenzzeiten, welches bisher die Erzielung hoher Leistung auf Parallelrechnern in den meisten praktischen Fällen verhindert hat, erheblich abgeschwächt wird.

8.3.5 Spezialhardware für neuronale Netze

Es gibt eine Reihe verschiedener Architekturformen von Spezialhardware für die Ausführung von Lernverfahren für neuronale Netze [174]. Die erste Kategorie bilden frei programmierbare Neuro-Parallelrechner, die zweite sind digitale Neuro-VLSI-Chips, die dritte sind analoge Neuro-VLSI-Chips. Analoge Chips [244] eignen sich nur schlecht für die Implementation von Lernverfahren, erst recht für so komplexe wie die in dieser Arbeit besprochenen konstruktiven Verfahren, und werden deshalb hier nicht diskutiert. Digitale Chips sind zumindest sehr schwierig zu programmieren, bilden aber zugleich die Grundlage vollständiger Neuro-Parallelrechner mit besserer Unterstützung der Programmierung und werden deshalb nicht getrennt diskutiert.

In der Kategorie frei programmierbarer Neuro-Rechner gibt es schon eine ganze Reihe von Geräten, z.B. den SNAP-32 von Hecht-Nielsen, den MY-NEUPOWER von Hitachi, die CNAPS von Adaptive Solutions und die Synapse-1 von Siemens. Die letzteren drei stelle ich im folgenden etwas genauer vor, um ein Bild von Leistungsfähigkeit und Grenzen solcher Maschinen zu geben. Weitere Maschinen befinden sich in der Entwicklung, wie z.B. die CNS-1 des ICSI [260, 261].

[4] Zitat eines anonymen Zynikers: „The peak performance is the performance the manufacturer guarantees the machine will never exceed.“

Hitachis MY-NEUPOWER [314] realisiert 512 Prozessoren für je ein „Neuron“ auf 64 firmeneigenen Chips ($1,0\mu$ CMOS, 25 MHz), die auf zwei Platinen untergebracht sind. Jeder Prozessor hat 1 Kilobyte Speicher für die Gewichte der Verbindungen, die mit 8 oder 16 bit Genauigkeit dargestellt werden und kann wahlweise bis zu 8 virtuelle „Neuronen“ emulieren. Gewichtsänderungen werden mit 8 bit Genauigkeit dargestellt, die Ausgaben eines „Neurons“ wahlweise mit 10 oder 16 bit. Multiplikationen können mit 16×10 bit oder mit 32×16 bit ausgeführt werden, Additionen mit zwei mal 32 oder 64 bit. Die Programmierung erfolgt in Mikrocode, der in einem Programmspeicher von 64K Worten zu je 63 bit abgelegt wird. Die Trainingsdaten liegen in 4M Worten zu je 10 bit. Die Maschine erreicht eine theoretische Spitzenleistung von 12,8 Milliarden Verbindungspassagen pro Sekunde (12,8 GCUPS, giga connection updates per second) und angeblich tatsächliche 1,26 GCUPS für Backpropagation-Lernen, also trotz Spezialarchitektur nur 10% der Spitzenleistung.

Die CNAPS von Adaptive Solutions [141] hat eine ähnliche Architektur. Hier werden 64 „Neuronen“ auf einem Chip ($0,7\mu$ CMOS, 25 MHz, Systemtakt 15 MHz) integriert und ebenfalls insgesamt bis zu 512 „Neuronen“ in ein System eingebaut. Jedes „Neuron“ hat 4 Kilobyte Gewichtsspeicher für Gewichte mit 1, 8 oder 16 bit Genauigkeit, 9×16 oder 16×16 bit Genauigkeit für Multiplikationen, Akkumulation mit 32 bit. Programmierung in Mikrocode, Assembler oder CNAPS-C, 512 Kilobyte Programmspeicher, bis zu 32 Mbyte Trainingsdatenspeicher. 7,7 GCUPS theoretische Spitzenleistung, etwa 2 GCUPS Spitzenleistung für Backpropagation-Lernen[5].

Sowohl MY-NEUPOWER als auch CNAPS haben erhebliche Einschränkungen in ihrer Flexibilität und sind deshalb für das in dieser Arbeit verfolgte Programmiermodell kaum geeignet. Zum Beispiel können die Maschinen keine Division ausführen und die Verwendung unregelmäßig verbundener Netze ist mit großen Schwierigkeiten verbunden. Die Größe der bearbeitbaren Netze ist vom mäßig großen Gewichtespeicher beschränkt; das betrifft insbesondere Lernverfahren, die eine größere Anzahl von Parametern pro Verbindung speichern müssen als reines Backpropagation-Lernen. Von der CNAPS ist außerdem bekannt, daß sie noch nicht ganz ausgereift ist; zahlreiche Fehler in den mitgelieferten Mikrocode-Bibliotheken, die erst allmählich beseitigt werden, erschweren den praktischen Einsatz erheblich[6].

Eine erheblich komplexere und flexiblere Architektur hat die Synapse-1 von Siemens [291]. Dieser Rechner besteht aus vier Baugruppen. Erstens der Prozessorkarte mit 4 Spalten zu je 2 Stück MA16 Signalprozesoren mit einer Gesamtgeschwindigkeit von 5 GCUPS bei 16×16 bit Multiplikationen und 48 bit Additionen; die Prozessorkarte verfügt insgesamt über 32 Megabyte lokalen Speicher (genannt Z-Speicher). Zweitens der Gewichtsspeicherkarte, mit 16 bis 512 Megabyte Speicher für die Parameter der Verbindungen (W-Speicher). Drittens der Dateneinheit mit einem MC68040 als Ganzzahleinheit (für „non-compute-intensive operations“), 16 bis 64 Megabyte Speicher für die Trainings- und sonstige Daten (C-Speicher) und 8 bis 32 Megabyte Speicher für die Kommunikation mit der Prozessorkarte (Y-Speicher). Viertens der Steuereinheit mit einem weiteren MC68040 mit 8 bis 32 Megabyte Speicher und dem Sequenzierer für die Signalprozessoren mit 0,25 bis 2 Megabyte statischem RAM als Mikroprogrammspeicher. Für die Programmierung stehen oberhalb der Mikrocode-Ebene drei Schichten von Bibliotheken zur Verfügung: Die nAPL Nachrichtenschicht realisiert synchrone und asynchrone Aufrufe von Rechen- und Speicheroperationen auf den Prozessoren; die nAPL Matrixschicht realisiert Operationen auf sogenannten Block-Gleitkommamatrizen, die 16 bit Mantisse für jedes Element sowie einen gemeinsamen Exponenten für alle Elemente haben, sowie auf Wertetabellen (*look-up tables*); die SENN++ Bibliothek stellt höhere Konzepte zur Verfügung wie „Neuron“, „Synapse“, Haufen von „Neuronen“, Konnektoren zwischen Haufen, Visualisierungsmethoden und Lernverfahren. Alle drei Schichten sind als C++ Klassenbibliotheken realisiert, als vierte Schicht gibt es einen fertigen Neuro-Simulator. Die Synapse-1 ist flexibel genug, um auch komplexe Lernverfahren realisieren zu können, obwohl auch hier die Programmierung schwierig ist; zur flexiblen Programmierung muß die nAPL Matrixschicht verwendet werden, in der schon viel Wissen über die komplizierte Architektur sichtbar

[5] Sven Wahle berichtet von einer realen Anwendung eine Leistung von 126 MCUPS für Backpropagation mit einem 112-500-147-Netz (persönliche Kommunikation, November 1994).

[6] Tilo Sloboda, Sven Wahle (persönliche Kommunikation, Oktober 1994).

wird, z.B. die Unterscheidung von Z-, W-, C- und Y-Speicher sowie die Unterscheidung sämtlicher Prozessorarten und Einzelprozessoren.

Die in dieser Arbeit vorgestellten Übersetzertechniken sollen für eine große Klasse von Parallelrechnern anwendbar sein und auch Anstöße für den Übersetzerbau für Parallelrechner auf anderen Gebieten als den neuronalen Algorithmen geben. Aus diesen beiden Gründen wird keine hochspezialisierte Maschine wie die Synapse-1 zugrundegelegt.

8.3.6 Optische und biologische Rechner

Optische und optoelektronische Rechnertechnik wird derzeit für verschiedene Anwendungsfelder erforscht: Im Mittelpunkt des Interesses stehen optische Kommunikationsverbindungen und optische Rechner für Spezialzwecke (unter anderem neuronale Netze). Eine im Prinzip interessante, aber bislang nur wenig bearbeitete Anwendung sind optische RAM-Speicher und noch exotischer sind universelle optoelektronische Prozessoren.

Optische Kommunikationsverbindungen erlauben im Prinzip eine sehr hohe Integrationsdichte [111] durch die Nichtbeeinflussung sich kreuzender Lichtstrahlen und geringen Platzbedarf optoelektronischer Sende- und Empfangselemente, einen geringen Bauteileaufwand durch die Nutzung des Freiraums und den Verzicht auf Leitungen, sowie eine starke Erhöhung des möglichen Eingangs- und Ausgangsgrades von Kommunikationsanschlüssen. Wegen des Fehlens von Leitungskapazitäten und Induktivitäten ist es bei längeren Verbindungsstrecken ferner wesentlich einfacher, eine hohe Übertragungsgeschwindigkeit einzuhalten, was wiederum auch die Verlustleistung verringert [112]. Dieser Zweig optischer Rechnertechnik ist schon am weitesten fortgeschritten und wird auch praktisch eingesetzt, stellt aber eigentlich kein optisches Rechnen dar. Allerdings sind optische Verbindungen eine entscheidende Grundlage beim Bau hybrider optisch-optoelektronisch-elektronischer Rechner [79].

Optische Kommunikationsverbindungen sind auch das Schlüsselelement zum Bau optischer neuronaler Netze. Die Ausnutzung von Freiraumverbindungen ermöglicht die hohe Verbindungsdichte, die hierfür nötig, aber mit den Leitungsbeschränkungen elektronischer Schaltungen schlecht zu erreichen ist [70]. Optoelektronische Realisierungen neuronaler Netze gibt es bereits [194, 174], sie haben jedoch bei weitem nicht genügend Flexibilität, um zur Implementation konstruktiver Lernverfahren geeignet zu sein. [290, 371] geben einen Überblick.

Optische Direktzugriffsspeicher haben im Vergleich zu ihren elektronischen Gegenstücken, den statischen und dynamischen RAMs, den Vorteil, daß sie einen echt parallelen Zugriff beliebig vieler lesender und schreibender Prozessoren auf dieselbe Speicherzelle erlauben (*CRCW-Speicher*). Ein solcher Speicher würde viele der Effizienzprobleme heutiger Parallelrechner beseitigen, selbst wenn er nur mit relativ geringer Kapazität als Ergänzung eines ansonsten elektronischen Speichers zur Verfügung stünde. Die bislang gebauten Systeme sind aber relativ langsam, haben eine zu geringe Lebensdauer oder eine zu kleine Kapazität. [228] gibt einen Überblick über diese Problematik und die aktuellen Forschungsansätze.

Der Vorteil eines optoelektronischen Prozessors gegenüber der herkömmlichen Halbleitertechnik liegt darin, daß er eine enorm viel kleinere Verlustleistung hat [343]. Die Schaltgeschwindigkeit ist mit elektronischen Prozessoren vergleichbar. Die geringere Verlustleistung erlaubt bei ausreichender Miniaturisierung viel höhere Parallelitätsgrade als sie mit Halbleitertechnik realisierbar sind, was irgendwann zu einer neuen Dimension der Leistungsfähigkeit von Parallelrechnern führen könnte. Optoelektronik anstatt reiner Optik ist nötig, weil sich manche Elemente der Schaltlogik, insbesondere die Negation, mit reiner Optik nicht gut realisieren lassen. Das Problem bei optoelektronischen Prozessoren ist, daß ihre Flexibilität mit abnehmendem Anteil der Elektronik ebenfalls sinkt. Deshalb ist die Forschung nach universellen optoelektronischen Prozessoren bislang exotisch.

Für alle diese Ansätze mit Ausnahme der optischen Verbindungen gilt gleichermaßen, daß sie für Anwendung im alltäglichen Einsatz noch lange nicht ausgereift sind. Forschungen, die nur auf den

Randbedinungen elektronischer Rechner aufbauen, werden noch für lange Zeit gerechtfertigt bleiben.

Noch weiter in die Zukunft weisen Forschungen zu biologischen Rechnern. Obwohl bereits erste Experimente zum Einsatz biologischer Hardware für kontrollierte Informationsverarbeitung gemacht werden, wird es noch sehr lange dauern, bis entsprechende Systeme eine Größe erreicht haben, in der sie nützliche Arbeiten verrichten können. Der prinzipielle Vorteil von biologischen Rechnern über elektronische und auch optische Rechner ist ihre Kombination von niedrigster Verlustleistung pro Schaltvorgang mit höchster Packungsdichte [343]. Als Nachteile stehen geringe Schaltgeschwindigkeit und eventuell Ermüdung gegenüber. Es ist noch vollkommen unklar, ob und inwieweit es je möglich sein wird, biologische Rechner zu bauen, deren Informationsverarbeitung an Leistungsfähigkeit das menschliche Gehirn erreicht oder übertrifft ohne dessen Nachteile mitzuübernehmen.

8.4 Stand der Forschung

Vom Stand der Forschung auf dem Gebiet der Parallelrechnerei sind für diese Arbeit drei Teilgebiete relevant, die in je einem eigenen Unterabschnitt besprochen werden: Erstens der Übersetzerbau für Parallelrechner, einschließlich der Entwicklung geeigneter Programmiersprachen. Zweitens die Behandlung unregelmäßiger Probleme mit statischer oder dynamisch veränderlicher Struktur. Drittens die Implementation neuronaler Verfahren. Da die Menge der Forschung vor allem auf den ersten beiden Gebieten enorm ist, machen die angegebenen Literaturhinweise nicht den Versuch von Vollständigkeit, sondern sind als Beispiele zu verstehen.

8.4.1 Programmiersprachen und Übersetzerbau

Auf bisherigen MIMD-Rechnern läßt sich die realisierbare Rechenleistung bislang nur erreichen, wenn man sie von Hand im Nachrichtenaustausch-Stil (*message passing*) [159] programmiert. Dies erfolgt durch das Schreiben von Knotenprogrammen, also sequentiellen Programmen, die nur auf einem Prozessor ablaufen, unter Verwendung einer Bibliothek von Kommunikationsoperationen (send, receive etc.). Die verwendete Programmiersprache ist meistens C oder Fortran, auf Transputer-basierten Rechnern Occam [175]; in der Regel läuft auf allen Prozessoren das gleiche Programm ab, das nur mit einer unterschiedlichen Konstante für die Prozessornummer geladen wird. Diese Programmierweise ist sehr mühsam und fehleranfällig. Der Programmierer ist für alle Aspekte von Datenverteilung, Prozeßverteilung, Synchronisation und Nachrichtenaustausch selbst verantwortlich; allenfalls das Routen der Nachrichten wird von der Bibliothek übernommen. Da solche Programme meist mit zahlreichen Eigenschaften der Maschine, z.B. Prozessoranzahl, Verbindungstopologie und Eigenarten der Kommunikationsbibliothek, eng verwoben sind, ist ihre Portabilität sehr schlecht. Das ist vor allem deshalb schlimm, weil der rapide Fortschritt der Parallelrechnertechnik damit nach kurzer Zeit ein Neuschreiben des Programms erforderlich macht.

Die Portabilität von *message-passing*-Programmen verbessert sich in jüngster Zeit durch die Standardisierung von Kommunikationsbibliotheken wie PARMACS [65], p4 [63] oder PVM [344], deren transparentes Routing auch eine bessere Verbergung der Rechnertopologie erlaubt. Erhalten bleiben die Probleme des vollständig asynchronen Programmierstils, der zu laufzeitabhängigen Programmfehlern (*race conditions*) führen kann. Trotz ihrer Mängel wird die Programmierung mit explizitem Nachrichtenaustausch wohl noch lange ein wichtiger Zweig bleiben, weil wie gesagt mit ihr am ehesten hohe Rechenleistungen zu erzielen sind.

Die zweite Wurzel der parallelen Programmierung stammt von den Vektorrechnern. Hier lautete der ursprüngliche Ansatz, sequentielle Fortran 77 Programme automatisch zu vektorisieren (s. unten); es liegt eine datenparallele Denkweise zugrunde. Die Fortschreibung des Fortran-Standards führte dann zu Fortran 90 [245], das explizite Vektoroperationen in die Sprache aufnimmt, um den Übersetzern die Vektorisierung zu erleichtern; Fortran 90 eignet sich auch gut für SIMD-Rechner. Der effizienten

Übersetzung auf MIMD-Rechner stehen jedoch einige Schwierigkeiten entgegen: Die Granularität von Vektoroperationen ist zu fein und die Art ihrer Parallelität ist nicht allgemein genug. Ferner ist die Berechnung einer geeigneten Prozeß- und Datenverteilung ohne Hinweise des Programmierers schwierig.

Aus diesem Grund wurde eine Vielzahl von Fortran-Dialekten entwickelt, die Erweiterungen zum explizit parallelen Programmieren und/oder zur Beschreibung der Datenverteilung enthalten, z.B. Kali [196], Vienna Fortran [73, 74], Fortran D [122, 157] und HPF [170]. In diesen Sprachen gibt der Programmierer Hinweise (Annotationen) für eine geschickte Verteilung von Daten und Prozessen an und formuliert seine Programme explizit parallel in einem überwiegend als datenparallel zu bezeichnenden Stil. Manche Annotationen hängen nur von der Problemstruktur ab (*decomposition*), andere stellen wiederum einen direkten Bezug zur Zielmaschine her (*mapping*) und verschlechtern somit die Portabilität. In manchen Systemen wird auch eine automatische Parallelisierung angeboten. Der Übersetzer erzeugt automatisch Code, der eine geeignete Daten- und Prozeßverteilung herstellt und alle nötigen Kommunikations- und Synchronisationsoperationen ausführt, um die Semantik des Programms mit einer parallelen Implementierung herzustellen. Gegenüber dem Programmieren mit explizitem Nachrichtenaustausch bedeutet dies eine enorme Erleichterung. HPF (High Performance Fortran) ist ein gemeinsamer halbformeller Standard mehrerer Arbeitsgruppen, der gute Aussichten auf erhebliche Verbreitung hat; seine wesentliche Grundlage ist Fortran D. Das Ziel dieser Sprachentwicklung besteht darin, alle die Sprachmittel bereitzustellen, die nötig sind, um auf realen Parallelrechnern eine hohe Rechenleistung zu erhalten, ohne die Abstraktionen (halb-)automatischer Daten- und Prozeßverteilung aufzugeben.

Ursprünglich für SIMD-Maschinen entwickelt, gibt es ferner datenparallele Varianten von C, nämlich C* [348] von Thinking Machines (siehe auch [353]) und MPL [235] von MasPar. Diese Sprachen verwenden eine kanonische, gleichmäßige Verteilung von Prozessen und Daten und arbeiten gemäß dem reinen datenparallelen Programmiermodell. Sie benötigen deshalb keinerlei zusätzliche Deklarationen vom Programmierer. Allerdings ist in MPL die Maschinenarchitektur sichtbar, so daß es eher auf eine niedrige Sprachebene einzustufen ist. Dataparallel C [149] ist eine Variante von C* für MIMD-Maschinen. C* ist auch auf der CM-5 verfügbar.

Noch konsequenter zu höheren Sprachniveaus gehen Modula-2* [351] und Modula-3* [152], die jeweils auf entsprechenden sequentiellen Sprachen basieren, diese nur um eine parallele Schleife (FORALL) und (im Falle von Modula-2*) minimale Datenverteilungsdeklaratoren erweitern, und alles andere dem Übersetzer überlassen. Noch ist nicht klar, ob (oder wie) ein Übersetzer immer gut genug selbst eine Daten- und Prozeßverteilung bestimmen kann, um zu ähnlich effizienten Realisierungen zu kommen, wie sie mit Fortran D möglich sind. Dies gilt insbesondere für mehrdimensionale Felder und für Fälle, in denen eine dynamische Umverteilung der Daten in verschiedenen Phasen des Programms sinnvoll wäre.

Den bisher genannten Sprachen ist gemeinsam, daß ihre Parallelität hauptsächlich auf Schleifen über Feldelemente ein- oder mehrdimensionaler Felder basiert. Die verwendeten Optimierungstechniken, zu denen ich unten noch komme, machen die bisherigen Übersetzer ausschließlich für regelmäßige Probleme geeignet. Bei unregelmäßigen Problemen kann weder Datenlokalität noch Lastbalance optimiert werden. Die bisherigen Optimierungen befassen sich deshalb nur mit einer Prozeßverteilung für den Fall gleichmäßiger Last und vor allen mit der Herstellung von Datenlokalität und der Vermeidung von Kommunikation. Dies wird unten noch genauer diskutiert.

Das höchste Sprachniveau erreichen funktionale Sprachen, in denen sämtliche Parallelität, nicht nur solche von Schleifen, leicht aus dem Programm abgelesen werden kann. Eine Vielzahl von Arbeiten befaßt sich mit der parallelen Ausführung funktionaler Programme; die Bibliographie [319] führt 350 Titel an. Da Programme, die die Mächtigkeit funktionaler Sprachen tatsächlich ausnutzen selbst auf sequentiellen Rechnern kaum effizient ausgeführt werden können, ist es nicht verwunderlich, daß effiziente parallele Realisierungen bisher nur für eingeschränkte Fälle erzeugt wurden: Für SISAL gibt es einen Übersetzer, der bei Programmen, die auf Feldzugriffen basieren, mit Fortran konkurrieren kann — zumindest auf Vektorrechnern [66]. NESL [46] ist eine Sprache, die auf verschachtelten Listen basiert („*nested sequence language*“). Dieses Paradigma erlaubt die Darstellung von verschachteltem Datenparallelismus, der selbst bei geradliniger Übersetzung meist zu voller Lastbalance führt. Leider ist aber die Herstellung von Datenlokalität schwierig, weshalb NESL bislang auch nur auf Vektorrechnern und SIMD-Maschinen eine hohe Effizienz erlaubt [46]. Die Eigenschaften funktionaler Sprachen erlauben zum Teil völlig andersartige Ansätze zur parallelen Realisierung als prozedurale Sprachen, z.B. beschreibt [171] die dynamische Parallelisierung von Programmen in der Sprache ML, bei der Parallelität ausgenutzt werden kann, die statisch sogar in einem explizit parallelen Programm nicht feststellbar ist.

Einen weiteren Zweig paralleler Sprachen bilden objektorientierte Sprachen. Diese bieten insbesondere einen guten Zugang zu grobkörnigem Datenparallelismus über Mengendatentypen. Beispiele für diesen Ansatz sind PC++ [210] und C** [209]. Grobkörnige Datenparallelität ist ein Ansatz, der die Entschärfung asynchroner Parallelität erlaubt [340]. Die Entschärfung beruht darauf, daß die Möglichkeit von laufzeitabhängigen Fehlern in der Sprache ausgeschlossen wird, indem den parallelen Arbeitseinheiten eines parallelen Abschnitts verboten wird, miteinander zu interagieren; für jede Interaktion ist deshalb automatisch eine Synchronisation vorhanden. Allgemeine objektorientierte Programmierung macht allerdings wegen des dynamischen Erzeugens und Vernichtens von Objekten und der Zeiger-basierten beliebigen Verknüpfung erhebliche Schwierigkeiten beim Finden guter Strategien für gleichmäßige Daten- und Prozeßverteilungen bei zugleich hoher Datenlokalität. Aus diesem Grund ist das parallele objektorientierte Programmieren wesentlich besser bei verteilten Systemen etabliert [1, 69], wo es grobkörniger eingesetzt und von speziellen Deklarationen und Anweisungen des Programmierers unterstützt wird; [408] gibt einen Überblick.

Ein Teil der bei Übersetzern für parallele Sprachen eingesetzten Optimierungen hat seine Wurzeln schon in sequentiellen Übersetzern, ein weiterer Teil bei den vektorisierenden Übersetzern: Datenflußanalyse (*data flow analysis*) stellt Zusammenhänge (oder ihr Fehlen) zwischen einzelnen Lese- und Schreibzugrifen auf Variablen fest. Einzelne Elemente von Feldern werden herkömmlicherweise bei der Datenflußanalyse nicht unterschieden. Die Ergebnisse einer Datenflußanalyse ermöglichen beispielsweise Optimierungen wie Elimination unerreichbaren Codes, Lade-, Speicher- und Registeroptimierung und Verzweigungsvorhersage, die durch den Einsatz moderner superskalarer Prozessoren gleichermaßen für Parallelrechner wie für sequentielle Rechner zunehmend an Bedeutung gewinnen. Zur schnellen Übersetzung großer Programme kann die Datenflußanalyse ihrerseits parallelisiert werden [201]. Demgegenüber betrachtet die Abhängigkeitsanalyse (*dependence analysis*) Wechselwirkungen (oder ihr Fehlen) zwischen den Feldzugriffen in verschiedenen Iterationen einer Schleife. Dabei werden die verwendeten Indexausdrücke analysiert, was den Verfahren auch den Namen Indexanalyse eingetragen hat. Die Abhängigkeitsanalyse ist nötig, um zu bestimmen, ob und inwieweit eine Schleife parallelisiert oder vektorisiert werden kann — nur eine Schleife ohne jegliche Abhängigkeit der Iterationen voneinander kann vollständig parallelisiert werden. Abhängigkeitsanalyse ist deshalb ein Schlüsselverfahren bei der automatischen Parallelisierung. Einige Arbeiten zur Abhängigkeitsanalyse sind der „klassische“ Text von Banerjee [28] und zahlreiche Verbesserungen und Erweiterungen [200, 217, 237, 372, 406]. Abhängigkeitsanalyse wird in allen diesen Arbeiten nur für solche Indexausdrücke durchgeführt, die lineare Funktionen der Laufvariablen (in einer oder mehreren Dimensionen) sind und nur von den Laufvariablen und Konstanten abhängen; [172] erweitert die Analyse auf dynamisch erzeugte anonyme Objekte, die über Zeiger angesprochen werden. Die bei der Abhängigkeitsanalyse gewonnenen Informationen können nicht nur zur Parallelisierung verwendet werden, sondern auch, um eine günstige *Ausrichtung* (*alignment*) von Feldern für hohe Datenlokalität — also eine gute Datenverteilung — zu finden, weshalb Abhängigkeitsanalyse auch für Übersetzer explizit paralleler Sprachen von Bedeutung ist. Die wichtigsten Arbeiten auf dem Gebiet der Datenausrichtung sind [68, 76, 78, 236, 277, 278, 279, 299]. Noch nicht gut beherrscht wird bislang das automatische Umverteilen oder Replizieren von Daten für verschiedene Phasen eines Programms, um unterschiedlichen Zugriffsmustern gerecht zu werden.

Weder die Datenflußanalyse noch die Abhängigkeitsanalyse sind im allgemeinen Fall berechenbar, d.h.

es kann nicht immer mit Sicherheit entschieden werden, ob oder wo eine Abhängigkeit vorliegt. Die Genauigkeit der Approximation kann zunehmen — was oftmals bessere Optimierung zuläßt — indem man die Verfahren miteinander verbindet, also die Datenflußanalyse um die Unterscheidung einzelner Feldelemente erweitert, anstatt komplette Felder als atomare Werte zu behandeln [10, 98, 236].

Eine weitere wichtige Optimierung, vor allem auf MIMD-Maschinen, ist das Bündeln von Kommunikationsoperationen mit gleichem Quell- und Zielprozessor, um ein mehrfaches Auftreten der Latenzzeit einzusparen und um redundante Kommunikationen zu vermeiden [10, 157, 196, 299]. Auch hierfür werden die Resultate von Datenfluß- und Abhängigkeitsanalyse eingesetzt und in schwierigen Fällen Laufzeitprüfungen vorgenommen, wie z.B. bei [157].

Verschiedene Arten von Schleifentransformationen werden eingesetzt, um beispielsweise die Granularität von Berechnungen zu erhöhen oder zu vereinheitlichen, Kommunikation einzusparen, Kommunikationsmuster zu vereinfachen oder statisch die Last zu balancieren [14, 75, 89, 90, 133, 372, 383, 405, 410].

Fast alle bisher vorhandenen Techniken dienen jedoch nur zur Übersetzung regelmäßiger Probleme. Die bestehenden Herausforderungen sind die effiziente Übersetzung unregelmäßiger Probleme und die Verwendung von Laufzeitinformationen zur statischen und dynamischen Optimierung [146]. Hierzu gibt es erst wenige Ansätze: [257] verwendet ein Laufzeitprofil als Hilfsmittel bei der Codeerzeugung und [386] verdeutlicht die Grenzen solcher statischen Ansätze, die darin begründet liegen, daß ein Laufzeitprofil nicht vollständig repräsentativ für zukünftige Programmläufe ist. [133, 157] sind Beispiele für die direkte Nutzung von Laufzeitinformationen während der Laufzeit eines Programms. Die Behandlung unregelmäßiger Probleme wird im folgenden Abschnitt besprochen.

8.4.2 Unregelmäßige Probleme

Unregelmäßige Probleme unterscheiden sich in der Art ihrer Behandlung vor allem danach, wann die konkrete Ausprägung der Unregelmäßigkeit feststeht:

1. Steht sie schon zur Übersetzungszeit fest, so kann eine *statische Lastbalancierung* vom Übersetzer vorgenommen werden. Diese spiegelt sich direkt in der Struktur des ereugten Codes wider und kann festkodiert oder durch Datenobjekte (Deskriptoren) beschrieben sein.
2. Steht sie nach einer Initialisierungsphase zu Beginn des Programms für den Rest des Programmlaufs fest, so kann eine statische Lastbalancierung zu diesem Zeitpunkt vorgenommen werden. Ihre Ergebnisse werden wiederum in Deskriptoren abgelegt, die die Verteilung von Daten und Prozessen beschreiben.
3. Steht sie zwar jeweils vor Beginn eines parallelen Abschnitts für diesen Abschnitt fest, ändert sich aber eventuell von Abschnitt zu Abschnitt, so muß eine Lastbalancierung während des Programmlaufs mehrfach durchgeführt werden. Oftmals ist jedoch keine Neubalancierung vor jedem parallelen Arbeitsabschnitt notwendig, sondern nur von Zeit zu Zeit. Es muß dann Merkmale geben, an denen sich ablesen läßt, wann eine Neubalancierung nötig ist. Eine solche Lastbalancierung nenne ich *semidynamische Lastbalancierung*.
4. Ergibt sich die Ausprägung der Unregelmäßigkeit erst während des Ablaufs eines parallelen Abschnitts, ist also überhaupt nicht vorhersagbar, sondern nur beobachtbar, so muß eine *dynamische Lastbalancierung* vorgenommen werden.

Zu allen diesen Fällen gibt es eine Vielzahl von Arbeiten. Lastbalancierung zur Übersetzungszeit wird normalerweise nur zum Ausgleich von lokalen Unregelmäßigkeiten verwendet, die unmittelbar aus der statischen Anweisungsstruktur des Programms (im Gegensatz zur Laufzeitwirkung der eingelesenen Daten) herrühren. Dieser Ansatz steckt als Bestandteil in einigen der Schleifentransformationen, die im vorigen Abschnitt erwähnt wurden, z.B. [133]. Für HPF sollen jedoch in Zukunft spezielle *map* Datenverteilungsanweisungen eingeführt werden, die auch eine an den Daten orientierte Lastbalancierung zur Übersetzungszeit zu spezifizieren gestatten [188].

Die statische Lastbalancierung zur Laufzeit kommt in zwei Varianten vor: Erstens als Planungsproblem für die Suche einer guten Ablaufreihenfolge einer Menge unabhängiger Arbeitseinheiten (*tasks*) mit im Voraus bekannter Laufzeit und zweitens als Partitionierungs- und Abbildungsproblem für ein paralleles Programm, in dem die parallelen Arbeitseinheiten miteinander in Wechselwirkung stehen. Der erstere Fall kommt für vollkommen unabhängige Aufträge vor [51, 370] oder für Aufträge, die durch eine Vorgängerrelation (*task graph*) geordnet sind [169, 191]. Beide Varianten sind hier von geringem Interesse. Das Problem der Partitionierung und Abbildung eines parallelen Programms kommt in folgender Situation vor: Gegeben ein Programm, dessen parallele Operationen auf einer strukturell unveränderlichen Datenstruktur ablaufen. Für jedes Teilobjekt der Datenstruktur wird in jedem parallelen Abschnitt eine parallele Arbeitseinheit ausgeführt. Die Unregelmäßigkeit des Problems rührt daher, daß erstens die Menge jeweils zu leistender Rechenarbeit für die Objekte verschieden ist und zweitens nicht jedes Objekt gleichviel Kommunikation mit anderen Objekten benötigt. Sind sowohl Rechenarbeit als auch Kommunikationsbedarf fest und bekannt, kann statische Lastbalancierung angewendet werden. Die Aufgabe lautet, die Datenstruktur so in Mengen M_i von Teilobjekten zu partitionieren, daß

1. eine Menge M_i pro Prozessor gebildet wird,
2. die Summe der Rechenarbeit für jede Menge M_i etwa gleich ist, und
3. die Summe der Kommunikation zwischen denjenigen Teilobjekten, die nicht in derselben Menge liegen minimal ist.

Eine solche Partitionierung muß also Mengen von Teilobjekten bestimmen, die eine stärkere Nachbarschaft aufweisen als andere. Das Problem kann als Graphpartitionierung formuliert werden, indem man als Knoten die Teilobjekte bzw. die parallelen Arbeitseinheiten benutzt, diese mit ihrem Rechenaufwand gewichtet und als gewichtete Verbindungen die Kommunikationsbeziehungen einträgt. Der Graph soll dann so partitioniert werden, daß die Knotengewichte für jeden Teilgraphen gleiche Summen haben und die Verbindungsgewichte der durchschnittenen Verbindungen minimale Summen ergeben; die Abwägung zwischen den Zielen kann durch eine Kommunikationskostenfunktion angegeben werden [190, 302]. Die Teilgraphen sollen ferner so auf die Prozessoren verteilt werden, daß die Distanzen der Kommunikationsbeziehungen im Mittel minimiert werden, um die Verkehrsdichte im Kommunikationsnetz klein zu halten.

Lastbalancierungsaufgaben dieses Typs kommen häufig in numerischen Programmen vor, die Methoden implementieren, welche auf einer *Bereichszerlegung* basieren, z.B. Finite-Elemente-Methoden oder *domain decomposition*-Methoden zur Lösung von Differentialgleichungen. In solchen Fällen, die zum Beispiel in der Strömungsmechanik auftreten [302], wird die Datenstruktur in der Regel als ein Gitter in zwei oder drei Dimensionen formuliert, wobei zu jedem Gitterpunkt oder jedem Gitterfeld ein Teilobjekt gehört. Die Gitter stellen eine Diskretisierung eines kontinuierlichen Problems dar und sind unregelmäßig, weil in manchen Bereichen mit höherer Genauigkeit gearbeitet werden soll als in anderen, also die Dichte von Gitterpunkten nicht überall gleich ist und auch die Änderung der Gitterpunktdichte nicht kontinuierlich verläuft. Die optimale Paritionierung solcher Gitter (oder von Graphen allgemein) erfordert einen exponentiellen Aufwand. Deshalb werden in der Praxis verschiedene heuristische Verfahren eingesetzt, die sich in ihrem Laufzeitaufwand und der Qualität des Resultats erheblich unterscheiden. Die meisten dieser Verfahren können parallelisiert werden. [153, 233] beschreiben insgesamt 11 solcher Methoden.

Ein neuronales Netz läßt sich als ein solches Gitter interpretieren, das allerdings mehr als zwei oder drei Dimensionen hat: Die Gitterpunkte sind die Knoten, die Nachbarschaftsbeziehungen sind durch die Verbindungen beschrieben. Die algorithmische Behandlung ist allerdings für geschichtete Netze anders als oben angegeben, weil nicht auf allen Gitterpunkten zugleich operiert wird. Für jedes Paar von Schichten, zwischen denen Verbindungen existieren, liegt quasi ein eigenes Gitter vor und diese Gitter sind gekoppelt. Deshalb ist die unveränderte Anwendung der Paritionierungsverfahren für Gitter nur begrenzt sinnvoll; für eine gute Aufteilung müßten erweiterte Verfahren gefunden werden (siehe auch Abschnitt 11.5.2).

Die gleichen Techniken wie für die statische Lastbalancierung zur Laufzeit können im Prinzip auch für semidynamische Lastbalancierung verwendet werden. Jedoch spielt hier die Laufzeit des Lastbalancierungsalgorithmus eine größere Rolle, weil sich die aus einer erneuten oder besseren Lastbalancierung ergebenden Vorteile nur über eine kürzere Programmlaufzeit, nämlich nur bis zur nächsten Balancierung, amortisieren können, anstatt über den gesamten Programmlauf hinweg [302, 382]. Dies schließt den Einsatz guter aber aufwendiger Methoden eventuell aus. Wie wir in Kapitel 11 sehen werden, spielt diese Überlegung bei konstruktiven Lernverfahren für neuronale Netze eine große Rolle.

Die allgemeinste Form der Lastbalancierung ist die dynamische. Das Problem hierbei ist, daß eine dynamische Umverteilung der Prozesse während eines parallelen Abschnitts in der Regel den Verlust der Datenlokalität nach sich zieht. Erfordert der Abschnitt keinerlei Kommunikation, so können die betroffenen Daten zusammen mit dem Prozeß verlagert werden [226]; andernfalls wird die Adressierung jedoch zu kompliziert. Außerdem amortisiert sich eventuell eine solche Datenverlagerung nicht. In den meisten Fällen wird jedoch Kommunikation nötig sein, was auf Maschinen mit echt verteiltem Speicher zu erheblich schlechterer Effizienz führt. Die einfachsten Methoden zur dynamischen Lastverteilung verwenden eine zentrale Schlange von parallelen Arbeitseinheiten, von der sich ein zum Zeitpunkt t freiwerdender Prozessor p jeweils eine gewisse Anzahl $A_{p(t)}$ von Einheiten abholt. Durch den Engpaß einer zentralen Datenstruktur skaliert dieser Ansatz nur beschränkt; er wird meist nur auf Rechnern mit physikalisch gemeinsamem Speicher eingesetzt. Da trotzdem jeder Zugriff auf die zentrale Schlange Kosten verursacht, liegen die Schwerpunkte der Forschungsarbeiten zu dieser Klasse von Lastbalancierungsverfahren darin, möglichst gute Regeln für die Berechnung von $A_{p(t)}$ zu finden, die die Gesamtanzahl von Zugriffen auf die Schlange minimieren, ohne zugleich die Ungleichverteilung der Last in der letzten Phase der Abarbeitung des parallelen Abschnitts zu groß werden zu lassen. Beispiele für solche Arbeiten sind [120, 226, 373]. Für massiv parallele Rechner eignen sich jedoch nur dezentrale Methoden. [399] vergleicht mehrere davon und kommt zu dem Schluß, daß die *empfänger-initiierte Diffusion* den besten Kompromiß zwischen Effizienz und Effektivität darstellt. Dieses recht einfache Verfahren verwendet nur Nachbarkommunikation und skaliert deshalb optimal. [335] betrachtet die Lastbalancierung für den Fall, daß die Arbeitseinheiten nicht auf einmal anfallen, sondern von den einzelnen Prozessoren sukzessive *produziert* werden. [322] behandelt Mischungen von sequentiellen und parallelen Aufträgen mit unterschiedlichem Resourcenbedarf, mit und ohne gegenseitige Abhängigkeiten; die Arbeit berücksichtigt auch den Einfluß unterschiedlicher Verbindungsnetze auf die Güte der Problemlösung.

Eigentlich sollte Lastverteilung in Zusammenarbeit mit dem Betriebssystem erfolgen. Da unterschiedliche Phasen eines Programms unterschiedlich hohe Parallelität haben können, wäre es günstig, wenn das Betriebssystem die Größe einer Prozessorpartition für ein Programm während dessen Ablaufs ändern könnte, um ungenutzte Prozessoren anderen Programmen zur Verfügung zu stellen. [72] diskutiert Strategien hierfür. Automatische Unterstützung solcher Mechanismen in Übersetzern gibt es bisher aber kaum.

Spärlich besetzte Matrizen sind eine spezielle Klasse von unregelmäßigen Problemen, die im Zusammenhang mit unregelmäßigen neuronalen Netzen relevant ist. Das Durchlaufen eines geschichteten neuronalen Netzes kann nämlich als Matrix-Vektor-Multiplikation interpretiert und realisiert werden: Die Aktivierungen der Anfangsknoten sind ein Vektor, die Gewichte der traversierten Verbindungen bilden eine Matrix, die für nicht vorhandene Verbindungen Nullen enthält. Das Resultat der Multiplikation sind die Eingabewerte der Zielknoten. Es gibt einige Softwarepakete für die allgemeine Behandlung spärlich besetzter Matrizen, z.B. [12, 192]. Ihre Parallelisierung wird aber zur Verringerung der Datenlokalitäts- und Lastbalanceschwierigkeiten oftmals nur auf Vektorrechnern [12, 47] oder auf Mehrprozessormaschinen mit gemeinsamem Speicher betrachtet [305]. Es gibt einige Implementationen, die sich eng an der Architektur einer bestimmten Maschine orientieren, z.B. Suprenum [56] oder DAP [11]. Häufig wird die Behandlung der Matrizen nur im Kontext einer bestimmten numerischen Anwendung betrachtet, was das Problem vereinfacht, weil dadurch die Art der Unregelmäßigkeit und der Gebrauch der Matrizen eingeschränkt werden [56, 184, 205, 305]. In manchen Arbeiten wird die Lastbalance als gegeben *angenommen*, anstatt daß Maßnahmen zu ihrer Herstellung ergriffen werden

[184]. [267] beschreibt Datenstrukturen zur Darstellung unregelmäßiger Graphen und empfiehlt für spärlich verbundene Graphen eine kantenorientierte Darstellung; eine solche Darstellung ist auch für neuronale Netze geeignet.

Alle diese Arbeiten geben keine oder nur eine sehr vage Hilfe für Techniken zur Übersetzung konstruktiver neuronaler Lernverfahren.

8.4.3 Neuronale Netze auf Parallelrechnern

Die Simulation Neuronaler Netze auf hochparallelen Rechnern wird derzeit noch kaum durch geeignete Werkzeuge unterstützt. Zwar gibt es eine beträchtliche Zahl von interaktiven Simulationssystemen für Neuronale Netze (z.B. [130, 213, 281, 378, 415] und auch zahlreiche kommerzielle Systeme von NeuralWare, California Scientific, SAIC und anderen ([219] enthält Literaturhinweise auf 17 Systeme), die meisten laufen jedoch nur auf sequentiellen Rechnern. Nichtsdestoweniger gibt es eine Reihe von Arbeiten mit parallelen oder zumindest parallelisierbaren Implementierungen:

Eine Reihe von massiv parallelen Simulationen von Neuronalen Netzen basieren auf handkodierten Programmen, die ein einzelnes Lernverfahren (zumeist Backpropagation) implementieren. Sie sind meist auf eine relativ kleine Klasse von Problemen zugeschnitten und in einer maschinennahen Sprache geschrieben, z.B. für die CM-2 [94, 321, 330, 417], die MP-1 [5, 134, 136, 224], den DAP [22], GF11 [402], die CM-5 [225] oder für Transputersysteme (z.B. [295], Bibliographie in [355]). Diese Programme sind nicht portabel. Es gibt zweifellos viele weitere Implementationen, die nicht publiziert wurden.

Mir sind derzeit folgende parallele Implementierungen von allgemeinen Neuro-Simulatoren bekannt: Für den Stuttgarter Neuronale Netze Simulator SNNS [415], der in seiner sequentiellen Version zahlreiche verschiedene Lernverfahren implementiert, gibt es eine feinkörnig parallele Implementierung (parallel über die Knoten und die Beispiele) für das Backpropagation-Verfahren auf der MasPar MP-1/MP-2 [230]. Später sollten Time Delay Netzwerke (TDNN) hinzukommen; parallele Implementierungen weiterer Lernverfahren sind nicht angekündigt. Dynamisch veränderliche Topologien werden nicht unterstützt. Der KNNS [199] arbeitet ebenfalls auf der MasPar MP-1/MP-2 und funktioniert für diverse Lernverfahren auf vorwärtsgerichteten, geschichteten Netzen. Er unterstützt durch einen Vorverarbeitungsschritt auch die effiziente Simulation von Netzen mit unregelmäßiger Struktur (durch die Einfügung von Pseudoknoten), erlaubt aber ebenfalls keine dynamische Veränderung der Topologie während des Lernens und macht einige Einschränkungen an die Berechnungen, die ein Lernverfahren durchführen kann. Der Rochester Connectionist Simulator (RCS) auf dem eher grobkörnig parallelen BBN Butterfly nutzt nur die Parallelität auf der Ebene der Knoten [110]. Neurograph [178] kann auf Rechnern mit gemeinsamem Speicher die Berechnung der Knoten über mehrere Prozessoren verteilen und bei Rechnern mit verteiltem Speicher die Berechnung der verschiedenen Beispiele durch Replikation des Netzes parallelisieren [177, 398]. Es ist jedoch eher für Workstation-Netze als für massiv parallele Rechner vorgesehen. Dynamisch veränderliche Topologien werden nicht unterstützt. Alle diese Simulatoren erfordern großen Implementierungsaufwand in maschinennaher Form für die Programmierung eines neuen Lernverfahrens.

Einige Ansätze versuchen einen schichtartigen Aufbau eines Simulationssystems. Hierbei wird zum Beispiel die Definition eines Netzes (Schicht 1) und die Abbildung des Netzes auf den Parallelrechner (Schicht 2) vom eigentlichen Lernverfahren (Schicht 3) getrennt und vor dem Programmablauf ausgeführt (z.B. [356] oder ANNE vom Oregon Graduate Center). Dadurch kann zumindest ein Teil der Arbeit bei der Implementierung eines neuen Verfahrens eingespart werden, weil die übrigen Module (Schicht 1 und 2) wiederbenutzt werden können; die bestehenden Systeme bieten jedoch keine Unterstützung für dynamische Netztopologien. Im Mittelpunkt dieser meist auf Transputersysteme ausgerichteten Arbeiten steht die Zerlegung des Netzes und Abbildung auf den Rechner [357, 358, 127]. Portabilität auf unterschiedliche Parallelrechnerarchitekturen ist auch hier nicht gegeben. Der Ansatz geht von modularen neuronalen Netzen aus, d.h. solchen, die aus mehreren in sich jeweils stark zusammenhängenden Teilen bestehen, welche untereinander aber nur wenige Verbindungen aufweisen.

Ohne diese Annahme bringt die Partitionierung fast keine Verbesserung gegenüber einer zufälligen Verteilung.

Einen Schritt hin zu größerer Flexibilität tun verschiedene Systeme, die spezielle Programmiersprachen einführen, um die Simulationen zu beschreiben. [363] enthält allgemeine Überlegungen zu Eigenschaften, die Programmiersprachen für Neuronale Algorithmen haben sollten, bleibt aber oberflächlich und stellt kaum Bezug zu Parallelrechnern her.

Im Esprit-Projekt Pygmalion [364] wurde eine Programmierumgebung für Neuronale Algorithmen geschaffen, die aus einer höheren Sprache namens „N", einer in N geschriebenen Algorithmenbibliothek, einer Zwischensprache namens „nC" und einer graphischen Bedien- und Beobachtungsoberfläche besteht. N ist ähnlich zu C++ und wird nach nC übersetzt. nC ist eine Untermenge von C, die um ein PAR–Konstrukt (ähnlich dem von Occam) für die parallele Ausführung einzelner Anweisungen und eine parallele FOR-Schleife erweitert wurde [381]. nC soll schließlich von verschiedenen Übersetzern auf unterschiedliche Zielsysteme abgebildet werden. Tatsächlich implementiert wurde nC nur für sequentielle Rechner und für Transputersysteme, die aber wegen ihrer hohen Latenzzeit bei Kommunikationsoperationen für feinkörning parallele Algorithmen kaum geeignet sind. Dies ist vielleicht auch der Grund, weshalb keine Veröffentlichungen über Leistungsmessungen dieser parallelen Implementierungen existieren. Dynamisch veränderliche Netztopologien werden nicht unterstützt. Der Ansatz verspricht keine optimale Effizienz auf parallelen Maschinen: Die Struktur von nC erlaubt es nicht, zahlreiche Informationen, die von einem optimierenden Übersetzer zur Optimierung des Zielprogramms herangezogen werden könnten, darzustellen. Eine besondere Ausnutzung der strukturellen Eigenarten eines Neuronalen Algorithmus durch den Übersetzer ist somit kaum möglich.

Im Esprit-Projekt NERVES wurde ein ähnlicher Ansatz verwendet [41]: Die höhere Programmiersprache MENTAL bietet ein explizites Zeitkonzept zur Modellierung von synchronen und asynchronen Abläufen und enthält Konstruktoren zur statischen Beschreibung von Netztopologien. Die auszuführenden Teile der Sprache werden direkt in der Sprache der darunterliegenden sogenannten virtuellen Maschine geschrieben und können zum Beispiel direkt die Modifikation einer ganzen Gewichtsmatrix umfassen. Die Sprache der virtuellen Maschine ist, ähnlich wie nC, eine parallel erweiterte Untermenge von C.
Der Entwurf zielt direkt auf zwei bestimmte Zielrechner: Den im gleichen Projekt zu entwickelnden Spezialprozessor für Matrixoperationen SMART und den ebenfalls im gleichen Projekt zu entwickelnden transputerbasierten Rechner SuperNode. Die Eigenschaften dieser Maschinen spiegeln sich im Sprachentwurf direkt wider; MENTAL ist deshalb unportabel. Es werden keine dynamisch veränderlichen Netztopologien unterstützt. Die Beurteilung dieses Ansatzes fällt auch ansonsten ähnlich aus wie bei Pygmalion.

Die Sprache AXON [151] von HNC wurde für die von der gleichen Firma angebotene spezielle Hardware entwickelt. Sie unterstützt keine dynamischen Topologien und erlaubt die Formulierung von Parallelität nur auf der Ebene der Knoten. Eine Optimierung findet nicht statt.

CONDELA („Connection Definition Language", [8]), unterstützt auch dynamische Netzstrukturen, erlaubt aber die Darstellung von Parallelität nur unvollständig. Die Sprache umfaßt nur die Erzeugung von Netzstrukturen, nicht jedoch die Beschreibung von Lernverfahren; diese müssen als externe Prozeduren in C formuliert werden. Es existiert keine parallele Implementation und nur eine rudimentäre sequentielle.

In [105] wird ebenfalls eine (allerdings nicht näher spezifizierte) Beschreibungssprache verwendet, mit der textuell oder graphisch erstellte Topologiebescheibungen einer Art Übersetzer vorgelegt werden, welcher daraus eine Datenverteilung für Parallelrechner ableitet, die hauptsächlich auf einer Parallelisierung über die Beispiele basiert. Die vom Übersetzer erzeugten Programmteile für den Zugriff auf diese Datenverteilung werden dann mit handgeschriebenen Teilen, welche das eigentliche Lernverfahren implementieren, zusammengebunden. Es existiert eine parallele Implementierung für die Connection Machine CM-2, die äußerst effizient ist, wenn man eine (naive) Parallelisierung über zehntausende von

Beispielen verwendet [330]. Eine dynamische Veränderung der Netztopologie ist nicht möglich.

MUME („an environment for multi-net and multi-architectures neural simulation“, [176]) ist eine Umgebung, die ein Zusammenbauen von Netzen aus Netzmodulen mit vordefinierter Funktionalität unterstützt. Die Teile werden über festgelegte Parameter instanziiert und mittels parametrisierter Schnittstellen miteinander verbunden. Die Architektur dieses Systems liegt zwischen einer Klassenbibliothek und einem kompletten Simulator. Das Modulkonzept erlaubt im Prinzip das Unterstützen von Parallelität im Innern der einzelnen Module. Eine Beschreibung von möglicher Parallelität über Module hinweg ist jedoch nicht vorgesehen. Das System ist als Bibliothek von Modulen in C implementiert. Eine parallele Implementierung existiert nicht.

SESAME („Software Environment for the Simulation of Adaptive Modular systEms“, [218, 219]) verfolgt einen ähnlichen Ansatz. Der Schwerpunkt liegt auf dem einfachen Bau komplexer Anwendungen aus vorgefertigten Modulen. Solche Anwendungen können zahlreiche verschiedene Lernverfahren enthalten, die zudem über Vererbungsmechanismen erweitert werden können. Das System wurde als eine Menge von C++ Klassen implementiert, die außer den eigentlichen Netzklassen und Lernverfahren auch Klassen für die Interaktion über eine graphische Schnittstelle umfaßt. Zu den Interaktionsmöglichkeiten gehört auch die dynamische Veränderung von Netztopologien; anstelle von Interaktion können dafür auch Programme in einer Kommandosprache benutzt werden. Wie bei MUME wäre auch bei SESAME eine parallele Implementation der Modulrümpfe möglich; es ist jedoch keine Beschreibung von parallelen Vorgängen durch den Benutzer vorgesehen. Eine tatsächlich parallele Implementierung existiert nicht.

Andere Arbeiten betrachten spezifische Aspekte der Parallelisierung von neuronalen Netzen, z.B. rekurrente Netze [86, 94], heterogene Netze, d.h. Kombinationen von konnektionistischen, lokalistischen und markenpropagiernden Netzen [208], oder die Laufzeitoptimierung der Netzreplikatanzahl bei Ausnutzung der Parallelität über die Beispiele [276].

8.5 Aufbau und Beiträge dieser Arbeit

Der Beitrag dieser Arbeit besteht darin, für eine Klasse von dynamisch unregelmäßigen Problemen, nämlich konstruktiven Lernverfahren für neuronale Netze, zu zeigen, wie hochsprachliche Programme automatisch in effizienten Code übersetzt werden können.

Die bislang für die hochsprachliche Programmierung von Parallelrechnern durchgeführten Forschungsarbeiten beschäftigen sich fast ausschließlich mit der Optimierung für Programme mit einer regelmäßigen Struktur des zu lösenden Problems. Die Behandlung einer Klasse von unregelmäßigen und zugleich dynamisch veränderlichen Problemen stellt einen neuen Beitrag dar. Der Schlüssel zu diesem Beitrag ist die Ausnutzung spezifischer Eigenarten des Programmverhaltens, das für konstruktive neuronale Lernverfahren charakteristisch ist.

Das Ziel wird in zwei Schritten erreicht:

1. Zunächst wird ein Programmiermodell für konstruktive neuronale Algorithmen eingeführt und eine Programmiersprache definiert, die dieses Modell realisiert (Kapitel 9).
2. Dann wird eine Reihe von Optimierungen beschrieben, die ein Übersetzer aufgrund des im Programmiermodell und in der Sprache enthaltenen Wissens vornehmen kann. Die Architektur eines konkreten, für die MasPar MP-1/2 implementierten Übersetzers wird ebenfalls vorgestellt (Kapitel 10).

In einem weiteren Kapitel wird die Leistungsfähigkeit des Übersetzers (insbesondere der Optimierungen) durch Messungen untersucht und bewertet (Kapitel 11).

Kapitel 9

Ein Programmiermodell für konstruktive Algorithmen

Don't write any parallel programs unless you have done it for at least five years.
Andreas Reuter

Algol 60 was a great improvement on many of its successors.
Edger Dijkstra

Dieses Kapitel führt das Programmiermodell ein, das in dieser Arbeit zur parallelen Implementation konstruktiver Lernverfahren verwendet wird. Zuerst wird die Grundidee des Modells vorgestellt und in Beziehung zu anderen Arbeiten auf dem Gebiet paralleler Programmiersprachen gesetzt (Abschnitt 9.1). Dann beschreibe ich die dem Modell zugrundeliegenden Datenstrukturen, mit denen ein neuronales Netz beschrieben wird (Abschnitt 9.2), und die parallelen Operationen auf diesen Datenstrukturen, die den besonderen Charakter des Modells ausmachen (Abschnitt 9.3). Weitere Abschnitte beschreiben kurz die konkrete Programmiersprache, die zur Realisierung des Modells vorgeschlagen wird und in dieser Arbeit implementiert wurde (Abschnitt 9.4), sowie die Möglichkeit der Einbettung des Modells in andere Programmiersprachen (Abschnitt 9.5).

9.1 Einführung und verwandte Arbeiten

Die erste Frage, die sich immer bei einer Überschrift der Form „Ein Programmiermodell für…“ aufdrängt, lautet „Wieso überhaupt? Kann man das nicht mit herkömmlichen Programmiersprachen machen?“. Die Antwort lautet, daß man natürlich kann, aber in diesem Fall nicht will: Wie wir im vorigen Kapitel gesehen haben, ist bei paralleler Programmierung bislang noch ein starker Widerspruch zwischen Effizienz und Sprachniveau vorhanden. Es ist noch nicht bekannt, wie dieser aufgelöst werden kann. Deshalb sind Vorschläge für parallele Programmiersprachen willkommen, die zeigen, wie sich ein hohes Sprachniveau mit einer Übersetzung in effizienten Code vereinbaren läßt. Auf der Ebene allgemeiner Programmiersprachen, die weder auf eine Maschinenarchitektur noch auf ein Anwendungsgebiet spezialisiert sind, ist, wie die aktuellen Forschungsarbeiten zeigen, dieses Effizienzproblem sehr schwierig zu lösen (siehe Abschnitt 8.4). An dieser Stelle setzt die Grundidee meines Programmiermodells an, das auf konstruktive Lernverfahren für neuronale Netze zielt und deshalb KNA-Programmiermodell (KNA: konstruktive neuronale Algorithmen) genannt wird.

> **Grundidee des KNA-Programmiermodells:** Formuliere ein Programmiermodell speziell für konstruktive neuronale Algorithmen so, daß es die spezifischen Eigenarten dieser Algorithmen in einer Form widerspiegelt, die sich leicht von einem Übersetzer für Optimierungen ausschöpfen läßt. Es sind dann auf einfache Weise Optimierungen möglich, die sonst gar nicht oder nur sehr schwer vorgenommen werden könnten.

Die Idee basiert also darauf, das Universum auszudrückender Programme einzuschränken und diese Einschränkungen für Optimierungen auszunutzen. Im vorliegenden Fall muß also zunächst eine Definition dafür gegeben werden, was unter einem neuronalen Algorithmus verstanden werden soll. Der Begriff soll konstruktive und nicht-konstruktive Lernverfahren für neuronale Netze im Sinne von Abschnitt 2.1 (Seite 11ff) abdecken. Die wichtigsten Eigenschaften sind dabei die zugrundeliegende Graphenstruktur des Netzes als Basis aller Berechnungen und die Lokalität der Berechnungen auf dieser Struktur. Die folgende ungefähre Definition steckt den Begriff ab, sie wird durch die später folgende Beschreibung des Programmiermodells präzisiert:

> Ein **neuronaler Algorithmus** ist ein Verfahren, dessen parallele Berechnungen auf einer graphenförmigen Datenstruktur aus Knoten und Verbindungen (dem neuronalen Netz) stattfinden. Dabei gibt es im wesentlichen fünf Arten von Operationen: lokale Operationen, Reduktionen und Rundrufe, sowie Erzeugen und Zerstören von Knoten oder Verbindungen. Ein solcher neuronaler Algorithmus realisiert Lernen aus Beispielen mit folgender Berechnungsstruktur: Er liest wiederholt ein Beispiel und führt darauf Berechnungen A aus, die viele oder alle Komponenten des Netzes betreffen. Nach einer gewissen Zahl von Beispielen, erfolgt jeweils eine andere Berechnung B und eventuell eine Veränderung C der Graphenstruktur.

In neuronalen Algorithmen werden nicht beliebige (z.B. über Indexausdrücke ausgewählte) Paare von Datenobjekten miteinander verknüpft und auch nicht beliebige parallele Datenstrukturen oder beliebige parallele Prozesse erzeugt. Statt dessen sind die Verknüpfungen von Objekten und die Parallelität durch die Graphstruktur vorgezeichnet und an diese gekoppelt.

Die Hauptidee des Programmiermodells besteht darin, durch diese Kopplung die Klasse der möglichen Datenzugriffe einzuschränken und auf diese Weise viel Information über das Programmverhalten dem Übersetzer zugänglich zu machen. Dies ähnelt einem Hauptvorzug des parallelen funktionalen Programmierens [319]: Dort ist viel Information deshalb zugänglich, weil es keinen Programmzustand und deshalb keine Seiteneffekte gibt. Deshalb können keine unvorhersehbaren Interaktionen zwischen Objekten stattfinden, wie sie in Sprachen vorkommen, die globale Speicherbereiche direkt zu adressieren erlauben. Die Lokalität der Zugriffe im Netzgraphen bei KNA-Programmen hat eine ähnliche Wirkung. Im Vergleich zum deklarativen, zustandslosen Stil des funktionalen Programmierens ist das KNA-Programmiermodell jedoch prozedural und damit mehr den herkömmlichen prozeduralen Programmiersprachen ähnlich. Eine prozedurale Formulierung ist für neuronale Algorithmen angemessener, weil ihnen die schrittweise Veränderung eines Datenobjekts, eben des neuronalen Netzes, zugrundeliegt.

Alle anderen mir bekannten Programmiermodelle für neuronale Algorithmen beziehen sich entweder nur auf die Modellierung der Netzstruktur und lassen den eigentlichen Berechnungsteil außer acht, wie Condela [8] und Nessus [414], oder sie sind als Bibliothek in sequentielle Sprachen eingebettet und für eine parallele Realisierung schlecht zugänglich, weil bei ihrem Entwurf nicht an eine Parallelisierung gedacht wurde, wie bei Sesame [218, 219] und MUME [176]. Eine Klasse für sich bildet [324]. Diese Arbeit betrachtet die Umsetzung eines beliebigen Algorithmus in ein äquivalentes rekurrentes neuronales Netz. Dabei wird beispielsweise zur Repräsentation einer Liste eine einzelne reelle Zahl mit unbeschränkter Genauigkeit benutzt; der praktische Wert dieses Ansatzes ist entsprechend gering.

Die beiden folgenden Abschnitte 9.2 und 9.3 stellen das Programmiermodell konkreter vor, der darauf folgende Abschnitt 9.4 beschreibt seine Umsetzung in eine Programmiersprache.

Programmiermodell und Programmiersprache erreichen die folgenden Ziele:

1. **Adäquatheit.** Das Modell enthält gerade die Konstrukte, die zur Beschreibung typischer neuronaler Operationen angemessen sind.
2. **Effizienz.** Modell und Sprache bewirken, daß Programme genügend leicht extrahierbare Information über das Programmverhalten enthalten, damit ein Übersetzer effizienten Code für verschiedenste Arten von parallelen Rechnern erzeugen kann. Dies gilt insbesondere für solche Algorithmen, die die Struktur des neuronalen Netzes während des Programmablaufs mehrfach verändern (konstruktive Algorithmen).
3. **Flexibilität.** Während das Programmiermodell als solches eine Einschränkung der ausdrückbaren parallelen Programme darstellt, macht die Sprache innerhalb dieser Grenzen keine weiteren Annahmen über die verwendeten Datenstrukturen und den Kontrollfluß.
4. **Abstraktion.** Die Sprache ist eine echte Hochsprache und abstrahiert vollständig von der zugrundeliegenden parallelen (oder seriellen) Maschine. Der Programmierer muß insbesondere keinerlei Entscheidungen über die Verteilung der Daten und der Prozesse treffen. Durch die Abstraktion wird zugleich Portabilität erreicht.

Um die Implementation der Sprache nicht zu schwierig zu gestalten, wurde auf einige andere mögliche Ziele verzichtet. So gibt es keine Objektorientierung mit Vererbung und keine spezielle Unterstützung für Fehlersuche, für genetische Algorithmen und für hybride Programme aus neuronalen und nichtneuronalen Teilen.

9.2 Parallele Datenstrukturen

Die einzige parallele Datenstruktur, die es im KNA-Programmiermodell gibt, ist das neuronale Netz. Wir betrachten das Netz auf vier Ebenen, die aufeinander aufbauen: Verbindung, Knoten, Knotengruppe und Netz. Jede dieser Ebenen stellt im Modell eine Kategorie von Datentypen dar, ähnlich den Kategorien Feld und Verbund, wie sie aus herkömmlichen Programmiersprachen bekannt sind. Ich beschreibe die Kategorien der Reihe nach.

Eine **Verbindung** ist strukturell äquivalent zu einem Verbund benutzerdefinierter Komponenten. Die Unterscheidung zu normalen Verbunden erfolgt nur, weil Verbindungen in einer Weise verwendet werden, die mit Verbunden nicht vorgesehen ist. Insbesondere werden Verbindungen stets erst beim Programmablauf erzeugt und sind fest mit je zwei Knoten verbunden.

Ein **Knoten** ist ein Verbund benutzerdefinierter Komponenten plus eine feste Anzahl von benutzerdefinierten **Schnittstellen**, an die Verbindungen angeschlossen werden können. Jede Schnittstelle ist gekennzeichnet als Anschluß entweder für hereinkommende Verbindungen (`IN`) oder für herausführende Verbindungen (`OUT`)[1]. Eine solche Schnittstelle beschreibt eine Menge von Verweisen auf Verbindungen. Zu jeder Verbindung gehören stets genau zwei Verweise, sie ist also mit exakt zwei Schnittstellen verbunden (besser gesagt: sie verbindet exakt zwei Schnittstellen miteinander).

Eine **Knotengruppe** ist eine eindimensionale Reihung von Knoten mit laufender Nummer. Die Anzahl von Knoten in einer Knotengruppe kann sich zur Laufzeit ändern. Eine Knotengruppe definiert eine Menge von Knoten, auf denen Operationen parallel ausgeführt werden, und entspricht meist einer Schicht in einen neuronalen Netz.

Ein **Netz** ist ein Verbund benutzerdefinierter Komponenten plus eine feste Anzahl von benutzerdefinierten Knotengruppen. Alle Operationen auf dem neuronalen Netz werden auf der Netzebene

[1] Eine Verbindung ist immer gerichtet, d.h. aus dem einer ihrer Schnittstellen führt sie hinaus, in die andere hinein. Diese Gerichtetheit beschränkt aber nicht den Datenfluß durch die Verbindung, der in beide Richtungen erfolgen kann; sie ist vielmehr nur notwendig, um in einer Implementierung eine direkte Zuordnung der Verbindungen zu genau einem ihrer Knoten vornehmen zu können, was aus Effizienzgründen sinnvoll ist.

aufgerufen und von dort aus an die Knoten und von diesen an die Verbindungen weitergereicht. Zur parallelen Bearbeitung mehrerer Beispiele ist es möglich, dynamisch **Replikate** eines Netzes zu erzeugen und wieder zu vereinigen. Solche Replikate sind strukturell identisch, speichern und bearbeiten aber verschiedene Werte.

Diese Art der Datenmodellierung hat folgende Vorteile. Erstens ist sie sehr flexibel. Die Modellierung des Netzes durch frei definierbare Datenstrukturen erlaubt die Anwendung des Modells auch in solchen Fällen, wo völlig neuartige Netzstrukturen verwendet werden müssen. Im Gegensatz dazu beschränken viele heutige NN-Simulatoren die erlaubten Komponenten von Knoten und Verbindungen auf einen festen Satz und erlauben beispielsweise nur die Behandlung von genau einer `IN`- und einer `OUT`-Schnittstelle pro Knoten. Zweitens geschieht die Datenmodellierung in den Termini des zugrundeliegenden Netzmodells, ist also problemorientiert, anstatt sich nach technischen Eigenarten üblicher Programmiersprachen auszurichten und beispielsweise explizit Felder oder Listen zu verwenden. Drittens führt die strenge Trennung der Ebenen Verbindung, Knoten und Netz zu einer modularen Beschreibung, die das Verständnis der Programme erleichtert. Diese Modularität wird dadurch noch erhöht, daß auch die Beschreibung der Operationen an die Datentypen gekoppelt ist.

9.3 Parallele Operationen

Das KNA-Programmiermodell verwendet eine objekt-zentrierte Darstellung der parallelen Operationen:

> Zu jedem Verbindungsdatentyp, Knotendatentyp und Netzdatentyp gibt es eine Anzahl von nur diesem Typ zugehörigen Verbindungsprozeduren, Knotenprozeduren bzw. Netzprozeduren (zu Knotengruppentypen gibt es keine eigenen Prozeduren). Parallelität wird dadurch erreicht, daß diese Prozeduren jeweils für eine Menge von Objekten parallel aufgerufen werden. Die Parallelität ist verschachtelbar: Sequentielle Prozeduren rufen Netzprozeduren (parallel über die Netzreplikate); diese rufen Knotenprozeduren (parallel über die Knoten einer Knotengruppe); diese rufen Verbindungsprozeduren (parallel über die Verbindungen einer Schnittstelle).

Eine *Objektprozedur* zu einem Objekttyp X hat Lese- und Schreibzugriff auf alle Komponenten des X-Objekts, für das sie aufgerufen wird und Lesezugriff auf alle ihre Parameter und auf alle globalen Variablen des Programms. Sie kann ferner andere Objektprozeduren von X sowie von Komponenten von X aufrufen; letzteres realisiert (verschachtelte) Parallelität. Zusätzlich gibt es sogenannte *zentrale Prozedur*en, die als sequentieller Hauptteil des Programms die Ein-/Ausgabe erledigen und die Netzprozeduren aufrufen sowie sogenannte *freie Prozeduren*, die von überall her aufgerufen werden können. Diese Aufrufbeziehungen sind in Abbildung 9.1 dargestellt. Eine globale (d.h. nicht-Objekt-)

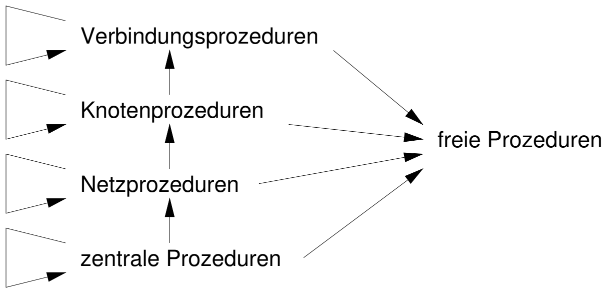


Abbildung 9.1: Mögliche Aufrufbeziehungen zwischen den Objektprozeduren der verschiedenen Datentypkategorien und normalen Prozeduren in KNA-Programmen. Ein Pfeil bedeutet jeweils „können aufrufen". Die möglichen Aufrufe von Objektprozeduren aus Objektprozeduren sind auf den eigenen Objekttyp und die Objekttypen der eigenen Komponenten beschränkt.

Prozedur ist genau dann eine freie Prozedur, wenn sie keinen Aufruf einer Netzprozedur vornimmt (weder direkt noch indirekt); andernfalls ist sie eine zentrale Prozedur. Unter Prozeduren sind in obiger Beschreibung auch Funktionen zu verstehen, wobei das Ergebnis eines Funktionsaufrufes für mehrere Objekte stets einer Reduktion unterworfen werden muß, z.B. der Aufruf einer Verbindungsfunktion aus einer Knotenprozedur.

Wie gesagt werden Objektprozeduren jeweils für eine Menge von Objekten parallel aufgerufen, was drei ineinander verschachtelte Ebenen von Parallelität ergibt. Diese Art von Parallelität ist ihrem Charakter nach Datenparallelität, hat jedoch eine variablere Granularität als reine Datenparallelität, weil die parallele Arbeitseinheit ein kompletter Prozeduraufruf anstatt nur einer einzigen elementaren Operation ist; der Rumpf der aufgerufenen Prozedur kann wegen der Lokalität von neuronalen Algorithmen asynchron über alle betroffenen Objekte abgearbeitet werden. Die Granularität der parallelen Operationen kann an die algorithmischen Bedürfnisse angepaßt werden, was die Effizienz verbessern kann. Außerdem sind auf diese Weise die technisch realisierten parallelen Arbeitseinheiten identisch mit den konzeptuell parallelen Arbeitseinheiten; diese Übereinstimmung führt zu einem einfachen und verständlichen Synchronisationsmodell: Eine Synchronisation paralleler Prozesse erfolgt immer nur genau dort, wo ihr jeweiliger Aufruf beendet wird. Diese Synchronisation erfolgt getrennt für jede der drei Ebenen der Parallelität. Mit anderen Worten: Ein paralleler Prozeduraufruf bedeutet genau das, was man davon erwartet, nämlich daß die aufrufende Prozedur genau dann fortgesetzt wird, wenn alle parallelen Aufrufe beendet wurden; die parallel aufgerufenen Prozedurinkarnationen selber können in beliebiger Reihenfolge und Verschränkung bearbeitet werden, weil wegen der Lokalität der Datenzugriffe keine Wechselwirkungen zwischen ihnen auftreten — mit einer Ausnahme wie unten beschrieben. Im einzelnen können im KNA-Programmiermodell folgende Operationen auftreten.

Verbindungsprozeduren realisieren die innerste Ebene der verschachtelten Parallelität, indem sie für zahlreiche Verbindungen zugleich aufgerufen werden. In der Prozedur selbst sieht ein solcher paralleler Aufruf wie ein Aufruf für eine einzige Verbindung aus, d.h. die Aufrufe für verschiedene Verbindungen im selben parallelen Aufruf sind vollkommen unabhängig voneinander. Der Rumpf einer Verbindungsprozedur ist rein sequentiell; es besteht Lese- und Schreibzugriff auf alle Komponenten der Verbindung, für die die Prozedur aufgerufen wurde, und Lesezugriff auf die Parameter des Aufrufs und auf alle globalen Variablen und Konstanten des Programms. Diese Regeln führen zu einer Semantik mit asynchroner Parallelität, in der keine Hazards (race conditions) möglich sind. Topologieveränderungen können aus Verbindungsprozeduren erfolgen, indem sich eine Verbindung selbst löscht.

Knotenprozeduren realisieren die zweite Ebene der Parallelität. Sie werden für zahlreiche Knoten einer Knotengruppe zugleich aufgerufen und können ihrerseits eine Verbindungsprozedur für alle Verbindungen einer ihrer Schnittstellen parallel aufrufen. Die dabei übergebenen Parameter werden per Rundruf an alle diese Verbindungen verteilt. Ist die Verbindungsprozedur eine Funktion, so werden die Funktionsresultate mit Hilfe eines Reduktionsoperators zu einem einzigen Wert verschmolzen. Hierzu können beliebige Reduktionsoperatoren für beliebige Datentypen vereinbart werden. Wie Verbindungsprozeduren haben auch Knotenprozeduren Lese- und Schreibzugriff auf die Komponenten ihres Knotens, sowie Lesezugriff auf alle ihre Parameter und alle globalen Variablen und Konstanten. Als Komponenten in diesem Sinne gelten nicht die Verbindungsschnittstellen; für diese ist nur der Prozeduraufruf möglich. Topologieveränderungen können aus Knotenprozeduren erfolgen, indem sich ein Knoten samt aller seiner Verbindungen selbst löscht oder indem ein Knoten sich in mehrere gleiche Exemplare „klont"; hierbei werden dann auch alle Verbindungen in entsprechender Zahl mitvervielfältigt; alle Datenkomponenten der entstehenden Knoten und Verbindungen sind anschließend identisch, die Knoten unterscheiden sich nur durch ihre implizite Nummer.

Netzprozeduren realisieren die dritte Ebene der Parallelität. Sie können für mehrere Replikate desselben Netzes zugleich aufgerufen werden und können ihrerseits eine Knotenprozedur für alle oder einige der Knoten einer Knotengruppe parallel aufrufen. Auch hier werden Parameter per Rundruf verteilt und ggf. Ergebnisse per Reduktion in einen einzelnen Wert zusammengefaßt. Wie Knoten- und Verbindungsprozeduren haben auch Netzprozeduren Lese- und Schreibzugriff auf die Komponenten ihres Netzes, sowie Lesezugriff auf alle ihre Parameter und alle globalen Variablen und Konstanten. Als Komponenten in diesem Sinne gelten nicht die Knotengruppen; für diese ist nur der Prozeduraufruf möglich. Topologieveränderungen können aus Netzprozeduren erfolgen, indem Verbindungen zwischen angegebenen Schnittstellen von Paaren von Knoten erzeugt oder gelöscht werden und indem neue Knoten (zunächst ohne Verbindungen) einer Knotengruppe zugefügt werden. Es können auch die Knoten mit größter Nummer aus einer Knotengruppe entfernt werden. Alle diese *Topologieänderungen* (auch jene in den Knoten- und Verbindungsprozeduren) können nur durchgeführt werden, während das Netz nicht repliziert ist, weil andernfalls die Replikate ihre Strukturgleichheit verlieren könnten und ihre Zusammenführung nicht mehr definiert wäre.

Eine **zentrale Prozedur** ist eine Prozedur, die keine Objektprozedur ist, aber eine Netzprozedur aufruft. Zentrale Prozeduren sind also sequentielle Prozeduren, aus denen heraus die parallelen Arbeitsabschnitte angestoßen werden. Solche Prozeduren können nicht aus Objektprozeduren heraus aufgerufen werden. Topologieveränderungen im eigentlichen Sinne können aus zentralen Prozeduren heraus nicht vorgenommen werden, jedoch können hier Replikate des Netzes erzeugt und wieder gelöscht werden. Beim „Löschen" von Replikaten werden die Daten der verschiedenen Replikate in dem einen Exemplar, das erhalten bleibt, zusammengeführt. Dazu werden spezielle Reduktionsprozeduren verwendet von denen je eine für jeden Verbindungstyp, Knotentyp und Netztyp vereinbart werden kann. Der typische Fall ist die Summierung einiger weniger Parameter jedes Teils (z.B. der Gradientenbeiträge der Verbindungen). Das Vorhandensein mehrerer Replikate führt die dritte Ebene von Parallelität zusätzlich zur Parallelität auf Knoten und auf Verbindungen in das Programmiermodell ein. Diese Ebene der Parallelität ist innerhalb der Objektprozeduren vollkommen transparent. Unterschiedliche Operationen werden von den Replikaten deshalb ausgeführt, weil unterschiedliche Daten in sie eingegeben werden können. Für die Dateneingabe und -ausgabe in das Netz und aus dem Netz sind ebenfalls die zentralen Prozeduren zuständig. Sie verfügen hierfür über spezielle Operatoren, mit denen Daten zwischen einem globalen Speicherbereich mit wohldefinierter Speicherverteilung und Knoten eines Netzes hin und her übertragen werden können. Bei diesen Operationen ist die Zahl von Replikaten bekannt und es werden entsprechend Daten für jedes Replikat bereitgestellt. Durch diese Aufteilung werden alle Programmteile, die unmittelbar mit dem Lernverfahren selbst zu tun haben, in die Objektprozeduren konzentriert, während Programmteile, die mehr verwaltungstechnischer Natur sind, insbesondere die Ein- und Ausgabe, in den zentralen Prozeduren stehen und deshalb gut vom eigentlichen algorithmischen Kern getrennt sind.

Freie Prozeduren sind alle Prozeduren, die weder eine Objektprozeduren noch zentrale Prozeduren sind. Eine *freie Prozedur* kann aus jeder anderen Prozedur heraus aufgerufen werden. Sie hat Lesezugriff auf ihre konstanten Parameter und auf globale Konstanten, sowie Lese- und Schreibzugriff auf ihre variablen Parameter und auf globale Variablen. Freie Prozeduren, die direkt oder indirekt aus Objektprozeduren aufgerufen werden, dürfen jedoch keine Schreibzugriffe auf globale Variablen enthalten.

In Abbildung 9.2 ist nochmals als Übersicht dargestellt, welche Arten von Prozeduren welche Änderungen der Topologie vornehmen können. Im folgenden Abschnitt ist beschrieben, wie das hier skizzierte

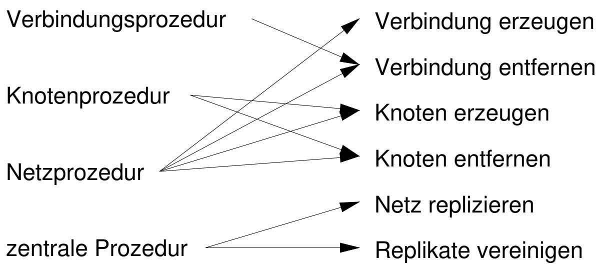


Abbildung 9.2: Prozedurarten und von ihnen aufrufbare Topologieänderungen in KNA-Programmen. Ein Pfeil bedeutet jeweils „kann ausführen".

Programmiermodell in eine Programmiersprache umgesetzt werden kann.

9.4 CuPit

Die Programmiersprache CuPit ist eine konkrete Realisierung des KNA-Programmiermodells. Das Referenzhandbuch zur Sprache [12] umfaßt 75 Seiten, einschließlich der Diskussion eines kompletten Programmbeispiels. Da die Details der Sprache für uns überwiegend nicht von Bedeutung sind, werden hier nur die wichtigsten Eigenschaften besprochen und anhand von Beispielen verdeutlicht. Die Syntax der Sprache ist an existierende prozedurale Sprachen angelehnt, insbesondere an Ada [374], C [189], ELAN [162] und Modula-2 [401]. An einigen Stellen mußten jedoch mangels Vorbild eigene Wege gegangen werden. CuPit ist benannt nach Warren McCulloch und Walter Pitts, die 1943 die erste Beschreibung eines formalen Neurons publizierten [239].

9.4.1 Verbindungstypen

Ein CuPit-Programm kann beliebig viele verschiedene Verbindungsdatentypen deklarieren. Ein Verbindungsdatentyp ist ein Datentyp der Kategorie `CONNECTION` und stellt strukturell einen Verbund fester Größe dar. Beispiel:

```
TYPE Weight IS CONNECTION
  Real in, out, weight, delta;
  (* hier können die Objektprozeduren des Typs deklariert werden *)
END;
```

deklariert einen Verbindungstyp *Weight* mit vier Komponenten *in, out, weight* und *delta* vom Typ *Real*. Diese Komponenten sind innerhalb jeder Objektprozedur des Weight-Typs als `ME`.weight und `ME`.delta ansprechbar. In jeder Objektprozedur bezeichnet `ME` das Objekt, für das die Prozedur aufgerufen wurde. Außerhalb der Objektprozeduren sind die Komponenten nur für Reduktionen verfügbar (siehe Abschnitt 9.4.5).

Es gibt in CuPit auch normale Verbunddatentypen. Beispiel:

```
TYPE Mytype IS RECORD
  Real a; Int b;
END;
```

Mit einem Verbindungstyp können einige Operationen vorgenommen werden, die für Verbunde nicht möglich sind, nämlich das dynamische Erzeugen und Ankoppeln an Knoten, das dynamische Zerstören und die Berechnung von Reduktionen über Verbindungsmengen. Deshalb werden die Datentypkategorien unterschieden.

9.4.2 Knotentypen und Knotengruppentypen

Ein CuPit-Programm kann beliebig viele verschiedene Knotendatentypen deklarieren. Ein Knotendatentyp ist ein Datentyp der Kategorie `NODE` und stellt strukturell einen Verbund dar, der neben normalen Datenkomponenten auch sogenannte *Schnittstellen* (*interfaces*) zu Verbindungen enthält. Beispiel:

```
TYPE Mynode IS NODE
  IN Weight in;
  OUT Weight out;
  Real inData, outData, errorSum;
  Int nrOfLargeErrors;
  (* hier können die Objektprozeduren des Typs deklariert werden *)
END;
```

deklariert einen Knotentyp *Mynode* mit vier Komponenten *inData, outData, errorSum* und *nrOfLargeErrors* vom Typ *Real* bzw. *Int*. Dies ist analog zur Definition von Verbindungstypen wie oben. Der Unterschied besteht in den Verbindungsschnittstellen *in* und *out*. Die Schnittstelle *in* erlaubt für jedes Objekt vom Typ *Mynode* den Anschluß von beliebig vielen Verbindungen des Typs *Weight*, die in den Knoten hineinführen. Die Gesamtheit dieser Verbindungen kann in Objektprozeduren des Typs Mynode als `ME`.in angesprochen werden. Das individuelle Ansprechen einzelner Verbindungen ist dabei nicht möglich und wäre auch nicht sinnvoll. Analog erlaubt die Schnittstelle *out* den Anschluß von Verbindungen, die aus dem Knoten herausführen. Jedes Verbindungsobjekt ist mit genau einer `IN`-Schnittstelle eines Knotens und einer `OUT`-Schnittstelle desselben Knotens oder eines anderen Knotens im selben Netz verbunden. Das Erzeugen und Anschließen von Verbindungen erfolgt mit der `CONNECT`-Anweisung, siehe Abschnitt 9.4.6. Die Verbindungen sind nicht Bestandteil der Knoten, sie sind ihnen nur zugeordnet. Ein Knotentyp kann, falls nötig, beliebig viele Schnittstellen mit beliebigen Namen und Typen deklarieren; meist genügen zwei.

Knoten werden in CuPit immer gruppenweise deklariert, erzeugt und verwendet. Dazu werden zu jedem Knotentyp ein oder mehrere Knotengruppentypen deklariert. Beispiel:

```
TYPE Dynamiclayer IS GROUP OF Mynode END;
TYPE Staticlayer IS ARRAY [12] OF Mynode END;
```

deklariert *Dynamiclayer* als eine dynamische Gruppe von *Mynode*-Knoten und *Staticlayer* als eine statische Gruppe von genau 12 *Mynode*-Knoten. Dynamische Gruppen sind anfangs leer, zugehörige Knoten werden dynamisch nach Bedarf erzeugt und wieder gelöscht. Operationen auf Knotengruppen können auf alle Knoten der Gruppe oder nur auf Teile davon angewendet werden.

Natürlich gibt es in CuPit auch normale Felder. Beispiel:

```
TYPE AnArray IS ARRAY [100] OF Int END;
```

Solche gewöhnlichen Felder können von Knotengruppen durch ihren Elementtyp unterschieden werden. Als Elementtyp eines gewöhnlichen Feldes ist nur ein elementarer Datentyp oder ein `RECORD`-Typ zulässig, nicht jedoch ein Verbindungstyp oder Netztyp. Auf gewöhnlichen Feldern können keine parallelen Operationen vorgenommen werden. Sie können jedoch als Komponenten in Verbindungs-, Knoten- oder Netztypen benutzt werden.

9.4.3 Netztypen

Ein CuPit-Programm kann beliebig viele verschiedene Netztypen deklarieren. Ein Netzdatentyp ist ein Datentyp der Kategorie `NETWORK` und ist eine Kombination eines Verbundes und einer festen Anzahl von Knotengruppen. Beispiel:

```
TYPE Mynet IS NETWORK
  Dynamiclayer in, hid, out;
  Real errorSum;
  (* hier können die Objektprozeduren des Typs deklariert werden *)
END;
```

deklariert einen Netztyp *Mynet* mit einer Datenkomponente *errorSum* vom Typ *Real* und drei dynamischen Knotengruppen *in, hid* und *out*. Diese Komponenten sind innerhalb jeder Netzprozedur des *Mynet*-Typs als `ME`.errorSum bzw. `ME`.in`[]`, `ME`.hid`[]`, und `ME`.out`[]` verfügbar. Aus den Knotengruppen können sogenannte Scheiben (*slices*) gemäß der linearen Numerierung der Knoten ausgewählt werden durch Angabe eines ersten und letzten Knotens, z.B. `ME`.hid`[4...8]` wählt nur die Knoten Nummer 4 bis 8 aus der *hid* Gruppe. Außerhalb der Netzprozeduren sind die Komponenten nur für Reduktionen verfügbar (siehe Abschnitt 9.4.5).

Ein CuPit-Programm kann eine oder mehrere Variablen deklarieren, die einen Netztyp haben; meist genügt eine. Es können keine Variablen mit einem Verbindungs-, Knoten- oder Knotengruppentyp deklariert werden; diese Datentypen kommen ausschließlich für Bestandteile einer Netzvariablen vor.

9.4.4 Prozeduren, Prozeduraufrufe, Parallelität

Die Deklaration normaler Prozeduren und Funktionen unterscheidet sich nur syntaktisch von entsprechenden Deklarationen in (beispielsweise) Modula-2.

Objektprozeduren sind syntaktisch mit normalen Prozeduren identisch. Sie werden ihren Objekttypen dadurch zugeordnet, daß ihre Deklaration im Innern einer Typdefinition steht, wie im den vorangehenden Abschnitten bereits angedeutet. In Objektprozeduren ist das Objekt, für das die Prozedur aufgerufen wurde, in der impliziten Variablen ME verfügbar. Eine Prozedur zur Durchführung des Vorwärtsdurchlaufschritts an einer Verbindung könnte beispielsweise so aussehen:

```
PROCEDURE transport (Real CONST val) IS
  ME.in := val;
  ME.out := val*ME.weight;
END PROCEDURE;
```

wobei diese Deklaration in der Deklaration des Typs *Weight* stehen muß. Der in die Verbindung einzugebende Wert wird als Argument übergeben. Die Prozedur errechnet daraus den Ausgabewert der Verbindung durch Multiplikation mit dem Gewicht.

Die zugehörige Knotenprozedur für den Vorwärtsdurchlauf durch einen verborgenen Knoten steht in der Deklaration des Typs *Mynode* und hat etwa folgende Form:

```
PROCEDURE forward (Real CONST steepness) IS
  REDUCTION ME.in[].out:rsum INTO ME.inData;
  ME.outData := sigmoid(steepness*ME.inData);
  ME.out[].transport (ME.outData);
END PROCEDURE;
```

Die erste Anweisung ist eine Reduktion der Ausgabewerte aller hereinkommenden Verbindungen und wird im folgenden Abschnitt erläutert. Die zweite Anweisung ist eine lokale Zuweisung wie in einer normalen Prozedur, dabei wird die freie Prozedur *sigmoid* aufgerufen. Die dritte Anweisung ist ein paralleler Aufruf der Verbindungsprozedur *transport* für alle an der Schnittstelle *out* an den Knoten angeschlossenen Verbindungen, denen der errechnete Ausgabewert *outData* des Knotens übergeben wird. Der Operator [] spezifiziert den Zugriff auf alle an einer Schnittstelle angeschlossenen Verbindungen, er erzeugt aus der Schnittstelle eine gedachte *parallele Variable*, auf der dann der Verbindungsprozeduraufruf ausgeführt wird.

Der Aufruf dieser Knotenprozedur in einer Netzprozedur hat beispielsweise die folgende Form:

```
ME.hid[].forward (1.0);
```

Auch hier erzeugt der []–Operator eine gedachte parallele Variable, nämlich bestehend aus allen Knoten der Gruppe, auf der dann der Aufruf der Knotenprozedur ausgeführt wird. Anders als die Verbindungen an einer Schnittstelle sind die Knoten einer Gruppe geordnet und indiziert. Deshalb kann hier der Operator eine Qualifizierung vornehmen, um nur einen bestimmten Teil der Knoten auszuwählen. Dies geschieht durch Angabe einer *Scheibe* (*slice*) von Knoten, z.B. wählt

```
ME.hid[1...MAXINDEX(ME.hid)].forward (1.0);
```

in einer Netzprozedur für den Aufruf von *forward* alle Knoten außer dem ersten, welcher die Nummer 0 hat. Solche Aufrufe werden in den Kandidatentrainingsverfahren verwendet, um Operationen nur auf den Kandidaten oder nur auf den schon „alteingesessenen“ Knoten einer Gruppe verborgener Knoten auszuführen.

Der Aufruf von Netzprozeduren in zentralen Prozeduren hat beispielsweise folgende Form: Gegeben eine Netzvariable, die mit

Mynet `VAR` net;

vereinbart wurde, kann ein Aufruf einer Netzprozedur namens *example* in einer zentralen Prozedur mittels

net`[]`.example();

erfolgen, wobei wiederum das `[]` eine parallele Variable andeutet, die in diesem Fall alle Replikate des Netzes umfaßt.

9.4.5 Reduktionen und winner-takes-all

Zur Vereinfachung der Implementation werden in CuPit Reduktionen nicht wie im KNA-Programmiermodell vorgesehen auf Funktionsaufrufe angewandt, sondern auf Datenobjekte. In vielen Fällen ist dies aber ohnehin die gewünschte Form, denn oft würde eine Funktion, deren Resultate reduziert werden sollen, lediglich einen bereits im Objekt gespeicherten Wert zurückgeben.

Ein Reduktionsaufruf hat wie im letzten Abschnitt angegeben beispielsweise folgende Form:

`REDUCTION ME`.out`[]`.in:rsum `INTO` ME.inData;

Dieser Aufruf bedeutet folgendes: Reduziere alle Werte, die in der Komponente *in* der Verbindungen von Schnittstelle *out* des Knotens `ME` gespeichert sind, unter Verwendung des Reduktionsoperators *rsum*, und speichere das Resultat in die Komponente `ME`.inData. Genau analog kann man auch Reduktionen auf Datenkomponenten von Knoten und von Netzen ausführen. Im vorliegenden Fall ist der Reduktionsoperator sehr simpel und wurde in folgender Weise vereinbart:

```
Real REDUCTION rsum IS
  RETURN (ME + YOU);
END REDUCTION;
```

Ein solcher Reduktionsoperator ist eine Funktion, die genau zwei Parameter hat. Die Parameter sind implizit deklariert; sie tragen die Namen `ME` und `YOU` und haben denselben Typ wie das Resultat der Reduktion. Reduktionsoperatoren sind stets global sichtbar und können für beliebige elementare oder Verbunddatentypen vereinbart werden, was den Reduktionsmechanismus von CuPit sehr flexibel macht. In den meisten Fällen kommt man allerdings mit der Summations-, Minimum- und Maximumreduktion auf den elementaren Datentypen *Int* und *Real* aus. Reduktionsoperatoren müssen assoziativ sein, weil die Abarbeitungsreihenfolge in einer Reduktionsoperation undefiniert ist.

Zusätzlich zur Reduktion von Daten gibt es eine Reduktion auf parallele Prozeduraufrufe, die sogenannte *winner-takes-all-Operation*, die in manchen Lernverfahren benötigt wird. Eine winner-takes-all Operation hat beispielsweise folgende Aufrufform:

`WTA ME`.out`[]`:in.wmin:update(a,b);

Diese Anweisung ruft die Verbindungsprozedur *update* mit den Argumenten *a* und *b* für genau eine Verbindung an der Schnittstelle *out* des Knotens `ME` auf. Die gewählte Verbindung ist diejenige, die die „Siegerin“ aus der Anwendung des Auswahloperators *wmin* auf die Komponente *in* von jeder Verbindung ist. Ein solcher Auswahloperator bekommt genau wie ein Reduktionsoperator zwei Werte `ME` und `YOU`. Er gibt jedoch nicht eine Reduktion dieser Werte als Ergebnis zurück, sondern einen Wahrheitswert, der angibt, ob `ME` der Sieger über `YOU` ist oder nicht; in letzterem Fall ist also `YOU` der Sieger. Der Sieger wird dann weiter mit anderen Werten verglichen, bis der Gesamtsieger feststeht. Nur für diesen wird der angegebene Prozeduraufruf ausgeführt. Der Auswahloperation für den Minimumvergleich des vorliegenden Beispiels sieht so aus:

```
Real WTA wmin IS
  RETURN (ME < YOU);
END WTA;
```

Genau wie bei den Reduktionsoperatoren können auch hier beliebige Funktionen beliebiger Datentypen als Operator eingeführt werden. Es sind auch winner-takes-all Operationen für Knoten und für Netze möglich.

Der Vorteil der Integration von Reduktion und winner-takes-all in die Programmiersprache liegt darin, daß ein Übersetzer aufgrund seines Wissens über die vorliegende physikalische Datenverteilung effiziente Realisierungen dieser Operationen erzeugen kann.

9.4.6 Topologieändernde Operationen

Zu Beginn des Programmablaufs hat ein Netz noch keinerlei Verbindungen. Von den Knoten existieren nur solche aus statischen Knotengruppen. Für unser Netz *net* aus obigen Beispielen könnten wir also beispielsweise 10 Eingabe- und 16 verborgene Knoten mit folgenden Anweisungen erzeugen:

```
EXTEND ME.in BY 10;
EXTEND ME.hid BY 16;
```

wobei diese Anweisungen in einer Netzprozedur stehen müssen. Mit analogen Anweisungen lassen sich während des Programmablaufs weitere Knoten zu den Gruppen hinzufügen, was beispielsweise in den Kandidatentrainingsverfahren zur Erzeugung der Kandidatengruppen verwendet wird, da die Kandidaten ja Bestandteil einer normalen Knotengruppe sind. Verbindungen zwischen den Knoten erzeugt die `CONNECT`-Anweisung. Soll beispielsweise je eine Verbindung von der *out* Schnittstelle jedes Knotens der *in* Gruppe zur *in* Schnittstelle jedes Knotens der *hid* Gruppe erzeugt werden, so kann dafür die Anweisung

```
CONNECT ME.in[].out WITH ME.hid[].in;
```

verwendet werden. Die Typen der Schnittstellen müssen übereinstimmen und legen den Typ der Verbindung fest; in diesem Fall den Typ *Weight*. Wie bei Knotenprozeduraufrufen können auch hier Scheiben angegeben werden, um nur einen Teil der Knoten einer oder beider beteiligter Knotengruppen auszuwählen. Zum Beispiel erzeugt

```
CONNECT ME.in[5...5].out WITH ME.hid[2...6].in;
```

nur Verbindungen von Knoten 5 der *in* Gruppe zu den Knoten 2 bis 6 der *hid* Gruppe. Analog zu `CONNECT` können mit `DISCONNECT` nicht mehr benötigte Verbindungen wieder entfernt werden.

Während des Programmablaufs können sich Knoten innerhalb einer Knotenprozedur selbst „klonen“, d.h. mehrere identische Kopien von sich und allen an sie angeschlossenen Verbindungen erzeugen. Dies erfolgt beispielsweise mit der Anweisung

```
REPLICATE ME INTO 2;
```

welche den betreffenden Knoten verdoppelt. Die entstandenen Knoten sind an der Knotennummer zu unterscheiden, die in Knotenprozeduren als `INDEX` verfügbar ist. Durch Angabe des Wertes Null für die Zahl von Exemplaren kann sich ein Knoten selbst löschen, also

```
REPLICATE ME INTO 0;
```

Diese Technik wird beispielsweise in den Kandidatentrainingsverfahren zum Löschen aller nicht erfolgreichen Kandidatenknoten verwendet; der erfolgreiche Kandidat identifiziert sich selbst als bester dadurch, daß er den besten aufgetretenen Gütewert hat und unterdrückt das Löschen. Die Anweisung ist auch in Verbindungsprozeduren zum Löschen einer Verbindung möglich und wird in den Beschneidungsverfahren angewendet.

Die Herstellung von Netzreplikaten erfolgt in zentralen Prozeduren mit Anweisungen der Form

```
REPLICATE net INTO 16;
REPLICATE net INTO 1...128;
REPLICATE net INTO 1;
```

wobei die erste Form genau 16 Replikate erzeugt, die zweite Form läßt dem Übersetzer die Entscheidung welche Anzahl von Replikaten zwischen 1 und 128 er für am effizientesten hält und demzufolge erzeugen will, denn das Benutzerprogramm kann meist mit verschiedenen Replikatanzahlen gleich gut zurechtkommen. Die Netzreplikate können sich in Netzprozeduren über ihre Nummer `INDEX` selbst identifizieren; die Gesamtzahl von Replikaten ist in zentralen Prozeduren als `MAXINDEX(net)+1` verfügbar. Die dritte Form der Anweisung führt die existierenden Replikate wieder zu einem einzigen Netz zusammen. Das Zusammenführen geschieht für alle Netzkomponenten einzeln und wird gesteuert von speziellen Reduktionsprozeduren. Diese haben beispielsweise die Form

```
MERGE IS (* im Typ Weight *)
  ME.delta += YOU.delta;
END MERGE;
```

von denen eine in jedem Verbindungs-, Knoten- und Netztyp deklariert werden kann und die angeben, welche Komponenten in welcher Weise zusammengeführt werden sollen. Dabei wird das Objekt `YOU` von der Reduktionsprozedur in das schreibbare Objekt `ME` „hineingemischt". Alle hiervon nicht berührten Komponenten werden lediglich von dem Netzreplikat mit Index 0 in das resultierende Netz übernommen.

9.4.7 Ein-/Ausgabe, externe Programmteile

Bliebe noch zu klären, wie die Daten in das Netz hinein und die Ergebnisse wieder heraus kommen. Da eine einheitliche Semantik für parallele Ein- und Ausgabe auf verschiedenen parallelen Rechnerarchitekturen noch nicht in Sicht ist, definiert CuPit diese Operationen nur auf der Ebene von Datentransfers im Hauptspeicher. Zu diesem Zweck gibt es eine dritte Art von Zugriffsrecht auf Datenobjekte (außer `CONST` für Konstanten und `VAR` für Variablen), nämlich `IO`. Ein `IO`-Objekt bezeichnet die Adresse eines Speicherbereiches, über den in definierter Weise die Eingaben in Netzknoten und die Ausgaben aus Netzknoten abgewickelt werden. Ein solches *Ein-/Ausgabebereich* genanntes Objekt wird beispielsweise global deklariert mit

```
Real IO x;
```

Die Speicheraufteilung eines dazugehörigen Speicherbereichs ist für jeden CuPit-Übersetzer und jede Zielmaschine maschinenabhängig definiert. Der Speicherbereich selbst muß von einer Prozedur außerhalb des CuPit-Programms allokiert werden. Die Größe hängt von der gewünschten Zahl von Knoten und Netzreplikaten ab, für die gleichzeitig Daten über den Ein-/Ausgabebereich transportiert werden sollen. Das Beschreiben des Ein-/Ausgabebereichs mit Eingabedaten für das Netz sowie das Auslesen von Ausgabedaten müssen ebenfalls in maschinenabhängigen externen Prozeduren vorgenommen werden. In CuPit selbst ist definiert, wie die Daten aus einem Ein-/Ausgabebereich in die Netzknoten oder aus Netzknoten in den Ein-/Ausgabebereich gelangen. Dies erfolgt mit Hilfe von speziellen Operatoren wie folgt:

```
net.in[].inData <-- x;
net.out[].inData --> x;
net.out[4...6].inData --> x;
```

Die erste Anweisung transportiert Daten aus dem Ein-/Ausgabebereich x in die *inData* Komponenten aller Knoten der *in* Gruppe aller Replikate des Netzes *net*. Die zweite Anweisung transportiert analog umgekehrt Daten aus allen *out* Knoten in den Ein-/Ausgabebereich. Die dritte Anweisung benutzt aus allen Replikaten jeweils nur die Knoten 4 bis 6 der *out* Gruppe. Ein- und Ausgabeanweisungen müssen in zentralen Prozeduren stehen.

Diese Art der Definition von Eingabe und Ausgabe hat den Vorteil, daß die Datenverteilung, die der Übersetzer für die Knoten benutzt in keiner Weise offengelegt werden muß. Insbesondere können Knotendaten für einen Knoten eines Netzreplikates beliebig über mehrere Prozessoren der parallelen Maschine repliziert werden, ohne daß die externen Programmteile davon etwas wissen müssen. Der Nachteil der Lösung ist, daß Eingaben und Ausgaben mindestens einmal kopiert werden.

9.4.8 Globale Programm- und Ausführungsstruktur

Global betrachtet besteht ein CuPit-Programm aus folgenden Teilen:

1. Deklarationen globaler Variablen und Konstanten,
2. Deklarationen von `IO`-Objekten,
3. Definitionen von Typen, insbesondere Verbindungs-, Knoten-, Knotengruppen- und Netztypen,
4. Definitionen von Reduktionsoperatoren,
5. Definitionen von winner-takes-all Auswahloperatoren,
6. Definitionen von Prozeduren,
7. Deklarationen externer Prozeduren.

Die Definitionen der Objektprozeduren sind dabei Bestandteil der Typdefinitionen. Deshalb sind alle global definierten Prozeduren entweder freie Prozeduren oder zentrale Prozeduren. Eine besondere zentrale Prozedur trägt den Namen *program*; sie ist der Einstiegspunkt in das Programm. Externe Prozeduren können deklariert werden und von beliebiger Stelle aufgerufen werden, sie sind stets freie Prozeduren. Insbesondere gibt es externe Prozeduren zum Lesen und Schreiben der Daten in Ein-/Ausgabebereichen. Externe Prozeduren können außerdem zur effizienten Implementation von Basisoperationen verwendet werden, wenn z.B. eine Aktivierungsfunktion tabellenbasiert implementiert werden soll und dafür die Tabelle einmal pro Prozessor der Zielmaschine vorhanden sein sollte.

Die Ausführung eines CuPit-Programms ist fast so einfach zu verstehen wie die eines sequentiellen Programms: Es gibt nur einen sequentiellen Programmablaufstrang, der sich unterwegs in mehrere Stränge aufspaltet. Diese parallelen Stränge haben jedoch keine Wechselwirkung miteinander und werden garantiert genau an der Stelle ihres Entstehens wieder zusammengeführt. Die parallelen Stränge lassen sich ihrerseits jeder einzeln als sequentieller Strang verstehen, wenn man ihre Bindung an ein Datenobjekt betrachtet. Dies kann auf jeder Ebene, auf der neue parallele Stränge entstehen, einzeln geschehen, also beim Aufruf von Netzprozeduren, Knotenprozeduren und Verbindungsprozeduren.

9.4.9 Sonstiges

Abschließend ein kurzer Überblick über die wichtigsten bisher nicht erklärten Teile von CuPit:

Elementare Datentypen sind die skalaren Typen Int, Int2, Int1 und Real (also Ganzahlen von 32 bit (4 Byte), 16 bit (2 byte) bzw. 8 bit (1 Byte) und Gleitkommazahlen), ferner die Typen Bool für Wahrheitswerte und String für Zeichenketten in der Art der Sprache C, sowie zu jedem der Zahldatentypen ein Intervalldatentyp (Interval, Interval2, Interval1, Realerval). Die Intervalldatentypen werden zur Darstellung der Scheiben gebraucht. Als benutzerdefinierte Typen gibt es die oben ausführlich beschriebenen Verbindungstypen, Knotentypen, Knotengruppentypen und Netztypen, ferner die kurz erwähnten Verbundtypen und Feldtypen, sowie Aufzählungstypen.

An Operatoren gibt es die üblichen arithmetischen Operatoren für Addition, Subtraktion, Multiplikation, Division, Modulo und Exponentiation, die üblichen Vergleichsoperatoren und logischen Operatoren, ferner Bitoperatoren wie in C, den ternären Fragezeichenoperator a?b:c von C, sowie den Intervallkonstruktionsoperator „..." und den Intervallabfrageoperator `IN`. Neben dem normalen Zuweisungsoperator `:=` gibt es die arithmetischen Zuweisungsoperatoren `+=, -=, *=, /=` und `%=`, die ebenfalls aus C entlehnt sind. Typumwandlungen zwischen Zahldatentypen sind mit den Typoperatoren möglich, der Subskriptionsoperator `[]` dient zusätzlich zur Konstruktion paralleler Variablen, wie oben beschrieben, einschließlich der Bildung von Scheiben.

Die Kontrollstrukturen sind ebenfalls im üblichen Rahmen: `IF THEN ELSIF THEN ELSE`, `WHILE`, `REPEAT UNTIL` und `FOR`, sowie `BREAK` zum Verlassen von Schleifen und `RETURN` zum Verlassen von Prozeduren. Als Erweiterung gibt es die kombinierten Schleifentypen `WHILE UNTIL` und `FOR UNTIL`. Parallelität wird ausschließlich über Objektprozeduraufrufe erzeugt, alle anderen Kontrollstrukturen haben die gewöhnliche Semantik; dies erleichtert das Verständnis von CuPit-Programmen.

Die Parameterübergabe erfolgt mit *call-by-value*-Semantik für konstante Parameter und mit *call-by-reference*-Semantik für variable Parameter.

Variablendeklarationen sind außerhalb von Prozeduren (globale Variablen) und zu Beginn jeder Anweisungsfolge (lokale Variablen mit Lebensdauer gleich Dauer der Anweisungsfolge) möglich, also beispielsweise zu Beginn des `THEN`-Teils einer `IF`-Anweisung. Variablen können initialisiert werden.

Operationen auf nichtexistenten Objekten sind Null-Operationen: Reduktionen über leere Objektmengen lassen die Zielvariable unverändert, winner-takes-all Anweisungen und Prozeduraufrufe mit leeren Objektmengen tun gar nichts, nichtexistente Knoten oder Netze in Scheiben werden ignoriert.

9.5 Alternative Realisierungen

Man muß nicht unbedingt eine Spezialsprache definieren, um eine Realisierung des KNA-Programmiermodells zu erhalten. Eine alternative Herangehensweise ist die Realisierung als Spracherweiterung. Die eleganteste Ausprägung man dann, wenn als Basis eine parallele objektorientierte Programmiersprache gewählt wird. Im folgenden skizziere ich eine Einbettung in Modula-3* [152], eine parallele Erweiterung der objektorientierten Sprache Modula-3 [67]. Eigenschaften der Zielmaschine wie Prozessoranzahl oder Kommunikations- und Synchronisationsmechanismen werden in Modula-3* genau wie in CuPit vollständig vor dem Programmierer verborgen. Parallelität wird in Modula-3* durch eine `FORALL`-Schleife realisiert: Die Anweisung

```
FORALL i : [1..n] NOSYNC DO
  A[i] := B[i] + C[i];
END;
```

realisiert beispielsweise eine asynchron parallele Addition der Komponenten 1 bis n der Vektoren B und C in den Vektor A. Synchrone Parallelität ist ebenfalls möglich. Modula-3* verfügt über die Möglichkeit der objektorientierten Typerweiterung, einschließlich Zufügung neuer Methoden (Objektprozeduren). Typerweiterung kann zur Realisierung des KNA-Programmiermodells eingesetzt werden, indem zugleich mit der Einführung bestimmter Basisklassen für dieselben eine gegenüber der Kernsprache erweiterte Semantik festgelegt wird; eine syntaktische Spracherweiterung ist dann nicht nötig. Die Erweiterung geschieht folgendermaßen.

Die Datentypkategorien `Connection`, `Node` und `Network` werden als Basisklassen realisiert. Diese Basisklassen enthalten keine Nutzdaten, sondern nur interne Verwaltungsinformation, die vor dem Benutzer jedoch verborgen bleibt. Als Methoden stellen sie nur die Operationen zur Topologieänderung, also zum Erzeugen und Löschen von Knoten und Verbindungen, zur Verfügung. Für jeden gewünschten Verbindungs-, Knoten- oder Netztyp erzeugt der Programmierer eine abgeleitete Klasse einer dieser

Basisklassen. Die abgeleitete Klasse spezifiziert alle benötigten Datenelemente und Objektprozeduren als zusätzliche Datenfelder und Methoden.

Mit jedem Verbindungstyp X werden implizit zwei weitere erweiterte Typen InX und $OutX$ definiert, die aus den Basisklassen `InInterface` und `OutInterface` abgeleitet werden und zur Definition von Schnittstellen in Knotentypen dienen. Diese zwei Typen erhalten implizit alle Methoden des Verbindungstyps. Analog wird zu jedem Knotentyp X implizit ein Knotengruppentyp gX definiert, der ebenfalls alle Methoden von X erhält. Diese impliziten Typdefinitionen stellen den einen Teil der Semantikerweiterung dar. Der andere Teil besteht in einer besonderen Semantik für Aufrufe von Methoden eines Verbindungs-, Knoten- oder Netztyps. Hinter dem einzelnen syntaktischen Objekt, das im Aufruf auftritt verbirgt sich ja faktisch jeweils eine Objektmenge, nämlich Replikate eines Netzes, Knoten einer Knotengruppe oder Verbindungen an einer Schnittstelle. Die erweiterte Aufrufsemantik besteht deshalb in einem asynchron parallelen Aufruf der Prozedur für alle betroffenen Objekte der Menge, genau wie in CuPit. Durch diese Semantik bleiben die genauen Mechanismen von Daten- und Prozeßverteilung komplett im Übersetzer verborgen.

Die Knoten und Verbindungen werden mit Pseudo-Indizes versehen, über die mit Hilfe einer in den Basisklassen bereitgestellten Dereferenzierungsprozedur auch ein direkter Zugriff auf einzelne Netzreplikate, Knoten oder Verbindungen möglich wird. Dadurch können die neuronalen Datenobjekte bei Bedarf auch in normale `FORALL`-Parallelität eingebunden werden.

Offensichtlich sind die Basisklassen `Connection`, `Node` und `Network` und die zugehörigen Operationen nicht als normale Bibliothek zu gestalten, sondern es muß Wissen über ihre Semantik in den Übersetzer eingebaut werden. Dieses Wissen umfaßt ungefähr alles, was auch der im nächsten Kapitel beschriebene CuPit-Übersetzer an Informationen über die Programme besitzt und ausnutzt. Damit dieses Wissen gültig ist, muß der Programmierer einige Konventionen einhalten, die die Verwendungsweise der neuronalen Datenelemente einschränken, insbesondere dürfen keine Verweise (Zeiger) auf solche Datenelemente erzeugt werden; der Übersetzer kann diese Konventionen aber statisch prüfen und ihre Einhaltung erzwingen. Unter diesen Voraussetzungen kann ein spezieller KNA-Modula-3*-Übersetzer die gleichen Optimierungen ausführen, wie ein CuPit-Übersetzer.

Man kann darüber streiten, ob man unter diesen Voraussetzungen aus Programmierersicht die eingebettete Version einer Spezialsprache wie CuPit vorziehen sollte. Einerseits erspart die Einbettung das Erlernen einer vollständig neuen Sprache, andererseits ist die Spezialsprache stellenweise doch eleganter, einfacher zu implementieren und eventuell effizienter: Bei einer Spezialsprache lassen sich einige komplizierte Einzelheiten vermeiden, die bei einer so mächtigen Sprache wie Modula-3* dazu führen, daß die Implementation sehr kompliziert wird. Neben dem reinen Sprachumfang mit solch komplexen Konstrukten wie insbesondere der Vererbung sind dies vor allem Komplikationen, die beim Mischen der spezialisierten parallelen Elemente aus dem KNA-Modell mit den allgemeinen parallelen Zugriffen auf verteilte Felder entstehen. Hierzu ein Beispiel: Eine unregelmäßige Netzstruktur sollte auch eine unregelmäßige Datenverteilung nach sich ziehen. Damit wird bei einer effizienten Realisierung die Anzahl tatsächlicher Prozesse verschieden von der Anzahl konzeptueller Prozesse, weil je nach Größe eines konzeptuellen Datenelements (z.B. Anzahl Verbindungen an einem Knoten, siehe nächstes Kapitel) mehr als ein oder weniger als ein tatsächlicher Prozeß für einen konzeptuellen Prozeß vorhanden ist. Diese Ungleichheit erschwert den geordneten Zugriff auf „normale“ parallele Felder, weil nicht auf der Hand liegt, welcher tatsächliche Prozeß für die Datenelemente jedes konzeptuellen Prozesses zuständig sein soll. Zahlreiche andere Detailschwierigkeiten treten außerdem auf.

9.6 Zusammenfassung und Beiträge dieser Arbeit

In diesem Kapitel wurde das KNA-Programmiermodell für explizit parallele Beschreibungen konstruktiver neuronaler Algorithmen und eine dieses Modell realisierende Programmiersprache namens CuPit vorgestellt.

Das Programmiermodell benutzt als Grundlage die explizite Modellierung eines Graphen, der das neuronale Netz beschreibt. Für Knoten und Verbindungen können beliebige Datentypen definiert werden. Das Modell begrenzt die möglichen parallelen Operationen auf lokale Operationen in Knoten und Verbindungen (durch Objektprozeduraufrufe), Rundrufe (durch Parameterübergabe beim Prozeduraufruf), Reduktionen (explizit, mit beliebigen Reduktionsoperatoren) und Operationen zum lokalen oder nicht-lokalen Erzeugen und Löschen von Knoten und Verbindungen.

Obwohl das Modell bzw. die Sprache CuPit aus programmiertechnischer Sicht für konstruktive neuronale Algorithmen möglicherweise einige Vorteile gegenüber allgemeinen parallelen Programmiersprachen haben, werden solche Vorteile nicht behauptet oder nachgewiesen. Vielmehr ist der angestrebte Nutzen lediglich die Erzeugung effizienten Codes auf parallelen Maschinen. Das in CuPit Programmen vorhandene Wissen über das Programmverhalten soll für Optimierungen ausgenutzt werden, die mit herkömmlichen parallelen Programmiersprachen nicht oder nur sehr schwierig möglich wären. Für den Nachweis dieser Möglichkeit wird im nächsten Kapitel ein paralleler Übersetzer für CuPit vorgestellt.

Abschnitt 9.5 beschreibt, wie man die Verwendung von Spezialprogrammiersprachen durch Einbettung des KNA-Programmiermodells in eine parallele objektorientierte Sprache vermeiden kann, falls man dies für sinnvoll hält. Der Ansatz einer solchen Einbettung besteht darin, den Übersetzer spezialisierte Implementationen für bestimmte Klassen und deren Operationen erzeugen zu lassen; dazu muß Wissen über die Verwendungskonventionen dieser Klassen und über das typische Verhalten ihrer Operationen in den Übersetzer eingebaut werden. Der Programmierer muß die Konventionen einhalten, braucht aber sonst von der Spezialisierung des Übersetzers nichts zu wissen, sondern erhält automatisch effizienten Code. Angesichts der Schwierigkeiten beim Erzeugen effizienten Codes für parallele Maschinen ist dieses Prinzip eine wichtige Ergänzung zu den Optimierungen, die für universelle parallele Sprachen wie HPF oder Modula-3* möglich sind. Vermutlich läßt sich dasselbe Prinzip nicht nur für neuronale Algorithmen sondern auch in einigen weiteren Anwendungsbereichen anwenden, in denen ausgeprägte typische Programmverhaltensweisen auftreten, beispielsweise für numerische Berechnungen auf unregelmäßigen Gittern.

Dies ist jedoch Zukunftsmusik. Der Beitrag dieser Arbeit besteht zunächst darin, eine Beschreibungsform für konstruktive neuronale Algorithmen entwickelt zu haben, die, wie in den nächsten zwei Kapiteln gezeigt werden wird, diverse Optimierungen bei der Erzeugung parallelen Codes erleichtert oder überhaupt erst ermöglicht.

Kapitel 10

Übersetzerarchitektur

Don't bite my finger,
look where I am pointing.
Warren McCulloch

Every group has a couple of experts.
And every group has at least one idiot.
Thus are balance and harmony (and discord) maintained.
Chuq Von Rospach

Dieses Kapitel beschreibt Codeerzeugungsstrategien und Optimierungstechniken für Übersetzer, die eine Sprache übersetzen, welche nach dem KNA-Programmiermodell gestaltet ist, und die einen Parallelrechner als Zielmaschine haben. Wir nennen die Klasse dieser Übersetzer *KNA-Übersetzer*. Zur Verdeutlichung wird dabei als Beispiel die Sprache CuPit herangezogen, aber alle Techniken sind auch auf andere KNA-Sprachen anwendbar. Als konkrete Zielmaschine für den realisierten Prototyp wurde MasPar MP-1/MP-2 benutzt, jedoch sind die beschriebenen Techniken ebenso für MIMD-Rechner zu verwenden, wie jeweils im Einzelfall begründet werden wird.

Im Einführungsabschnitt 10.1 grenze ich zunächst ab, welche Optimierungsziele in dieser Arbeit betrachtet werden, welche Annahmen zugrundeliegen, wie die Realisierung ungefähr aussieht und welchen Einfluß die Wahl der Zielmaschine hat; Abschnitt 10.2 gibt Definitionen. Die folgenden Abschnitte beschreiben erstens, wie die Datenverteilung und Prozeßverteilung funktioniert, mit der sich die Optimierungsziele erreichen lassen (Abschnitt 10.3), und zweitens, welche Optimierungen an Parametern dieser Verteilung vorgenommen werden können (Abschnitt 10.4).

10.1 Einführung und Überblick

10.1.1 Optimierungsziele

Die hier betrachteten Optimierungstechniken decken keinen der Aspekte ab, die üblicherweise von modernen Übersetzern für sequentielle Maschinen optimiert werden, z.B. Gucklochoptimierungen (*peephole optimizations*), Registerallokation, automatisches Ausrollen von Prozeduren (*inlining*), Instruktionsfolgenplanung (*instruction scheduling*), etc. Stattdessen betrachten wir einen Übersetzer, der in eine Sprache auf mittlerem Sprachniveau übersetzt (z.B. C). Dieser Zwischencode wird dann von einem weiteren Übersetzer, der die obengenannten Optimierungen leistet, in Maschinencode umgewandelt. Manche der Optimierungen könnten natürlich von einem KNA-Übersetzer etwas besser geleistet werden, weil mehr Information zur Verfügung steht, aber der Unterschied dürfte nicht sehr groß sein, und im Rahmen dieser Arbeit schien demgegenüber der nötige Mehraufwand zu hoch.

Ebenso bleiben auch viele Optimierungsaspekte außer acht, die bei heutigen Übersetzern für parallele Maschinen im Mittelpunkt des Interesses stehen, insbesondere automatische Parallelisierung, automatische Vektorisierung und die Elimination von Synchronisationspunkten, welche für einen CuPit Übersetzer auf der MasPar MP-1/MP-2 sämtlich nicht relevant sind.

> *Die hauptsächlich betrachteten Optimierungsziele sind*
>
> 1. *Datenlokalität und*
> 2. *semidynamische Lastbalancierung*
>
> *Hinzu kommt die Minimierung von Kommunikationsoperationen für diejenigen Fälle, in denen Kommunikation unvermeidlich ist.*

Datenlokalität (Objektlokalität) bedeutet, daß für eine Operation A, die auf einem einzelnen Objekt X arbeitet, zum Beispiel einem Knoten oder einer Verbindung, die benötigten Daten von vornherein, also ohne Kommunikation, auf dem Prozessor verfügbar sind, der A ausführt.

Semidynamische Lastbalancierung bedeutet, daß die Arbeitseinheiten eines jeden parallelen Abschnitts so auf die Prozessoren verteilt werden, daß die Laufzeit des Abschnitts auf jedem Prozessor ungefähr gleich ist und die dafür nötigen Entscheidungen komplett vor oder zu Beginn des Abschnitts getroffen werden.

Diese beiden Ziele liegen im Widerstreit, denn Maßnahmen zur Balancierung der Last führen normalerweise dazu, daß eine gegebene Prozeßverteilung mit hoher Datenlokalität so auseinandergerissen werden muß, daß die Lokalität weitgehend verloren geht. Dieser Effekt macht das gleichzeitige Erreichen beider Ziele für allgemeine Programme und Programmiersprachen enorm schwierig. Wie wir sehen werden, erlauben die speziellen Eigenschaften neuronaler Algorithmen jedoch eine gute Datenlokalität bei zugleich guter Lastbalancierung — und das mit relativ einfachen Methoden. Denn die Berechnung und Herstellung einer Daten- und Prozeßverteilung, die die beiden Ziele gut annähert, darf nicht zu aufwendig sein, weil sonst durch diesen Verwaltungsaufwand die erzielte Beschleunigung wieder aufgezehrt wird.

Der Realisierung dieser Optimierungsziele liegen einige Annahmen zugrunde, die im nächsten Unterabschnitt beschrieben sind.

10.1.2 Annahmen über das Programmverhalten

Die Techniken, mit denen die oben beschriebenen Optimierungsziele erreicht werden, basieren auf einer Reihe von Annahmen, die mit den bereichsspezifischen Eigenarten von neuronalen Algorithmen gerechtfertigt werden können. Zu jeder Annahme gebe ich ihren Nutzen bei der Realisierung von Optimierungen und ihre Rechtfertigung an. Die Annahmen sind nur qualitativ formuliert, denn der Zusammenhang zwischen dem konkreten Maß ihres Zutreffens und der Effizienz einer auf den Annahmen basierenden Implementierung ist stetig, so daß es keine zwingenden Grenzwerte gibt, die für eine quantitative Formulierung der Annahmen herangezogen werden könnten. Folgende Annahmen werden benutzt:

Annahme 1: Für jeden einzelnen Aufruf einer Verbindungsoperation ist die zu leistende Arbeit für jede betroffene Verbindung ungefähr gleich.
Nutzen: Unter dieser Annahme braucht man keine Lastbalancierungsmaßnahmen innerhalb von Verbindungsoperationen vorzunehmen; sie erlaubt die Benutzung der Anzahl von Verbindungen an einem Knoten als relatives Maß für die Arbeit in einem Aufruf einer Verbindungsoperation und bildet damit die Basis für die Erzielung der Lastbalance.
Rechtfertigung: In neuronalen Algorithmen sind die Verbindungen in der Regel primitive Objekte. Die auf ihnen durchgeführten Operationen enthalten normalerweise keine Schleifen; deshalb unterscheiden sich verschiedene Aufrufe derselben Operation in ihrer Laufzeit nur geringfügig.

Annahme 2: Für jeden einzelnen Aufruf einer Knotenoperation ist die zu leistende Arbeit für jeden Knoten der betroffenen Gruppe ungefähr gleich, wenn man von der auf den Verbindungen zu leistenden Arbeit absieht, die wegen unterschiedlicher Verbindungsanzahlen verschieden sein kann.
Nutzen: Mit dieser Annahme kann man die Lastbalancierung auf der Ebene von Knoten auf die Lastverteilung der Verbindungsoperationen beschränken.
Rechtfertigung: Sieht man von der Arbeit auf den Verbindungen ab, so gilt dieselbe Argumentation wie für Annahme 1. Die Tatsache, daß die Arbeit auf den Verbindungen in die Knotenoperation eingebettet ist, macht die Situation allerdings in bestimmten Fällen komplizierter: Sollte die Knotenoperation bestimmte Verbindungsoperationen nicht in allen Knoten durchführen, so gilt die Annahme nicht mehr. Ein solches Verhalten tritt allerdings in keinem der üblichen Lernverfahren auf. Das Zusammenbrechen der Lastbalancierung für diesen Fall nehmen wir deshalb ohne Gegenmaßnahmen hin. In den Kandidatenlernverfahren aus Kapitel 5, werden komplette Knotenoperationen nur auf Teilen der ganzen Knotengruppe ausgeführt, was ebenfalls die Annahme verletzt; dieses Problem kann jedoch gelöst werden, wie in Abschnitt 10.4.3 beschrieben wird.

Annahme 3: Die Datenverteilung muß nicht sehr häufig geändert werden, um Datenlokalität und Lastbalance zu erhalten.
Nutzen: Seltene Änderungen der Datenverteilung erlauben größere Investitionen in das Finden und Herstellen einer *guten* Datenverteilung, denn die zur Berechnung und Herstellung einer neuen Datenverteilung nötige Arbeit kann sich über viele Berechnungsschritte des eigentlichen Algorithmus amortisieren.
Rechtfertigung: In neuronalen Algorithmen treten Topologieveränderungen normalerweise nur nach einer oder mehreren Trainigsepochen auf. Typischerweise werden über die Hunderttausende oder Millionen von Durchläufen von Beispielen nur höchstens wenige Dutzend Topologieänderungen durchgeführt. Aufgrund der Annahme 4 sind aber Änderungen der Datenverteilung in der Regel nur bei Änderungen der Netztopologie nötig.

Annahme 4: Das Verhalten eines neuronalen Algorithmus im Hinblick auf die Arbeit an den einzelnen Teilobjekten des Netzes ändert sich nur allmählich, wenn überhaupt.
Nutzen: Langsame Verhaltensänderungen bedeuten, daß die Qualität einer Entscheidung über Datenverteilung, die auf dem Verhalten des Programms in früheren Phasen seiner Ausführung basiert, nur langsam abnehmen kann und meist gar nicht abnehmen wird. Insbesondere kann die Lastbalancierung, die von einer bestimmten Datenverteilung gewährleistet wird, nicht schlagartig schlecht werden, es sei denn, die Topologie des Netzes wird verändert.
Rechtfertigung: Die Struktur neuronaler Algorithmen ist derart, daß immer wieder dieselben einfachen Operationen auf den Teilobjekten des Netzes ausgeführt werden. Unterschiedliche Phasen im Trainingsverlauf mit völlig verschiedenem Verhalten treten nicht auf. Eine Ausnahme sind Verfahren von der Art der Kandidatentrainingsverfahren, wie sie in Kapitel 5 beschrieben wurden, bei denen solche Phasenübergänge aber stets an Topologieänderungen gebunden sind und deshalb beherrscht werden können.

Annahme 5: Der Versuch zahlt sich nicht aus, zur Erhöhung der Lokalität Knoten, zwischen denen eine Verbindung besteht, möglichst häufig auf demselben Prozessor anzuordnen.
Nutzen: Diese Annahme vereinfacht die Berechnung einer guten Datenverteilung dramatisch, weil keine Graphpartitionierungsverfahren verwendet werden müssen — welche sehr rechenaufwendig sind. Außerdem können bei Verzicht auf eine allgemeine Form von Graphpartitionierung andere effizienzsteigernde Eigenschaften in die Datenverteilung integriert werden, z.B. eine einfachere Adreßberechnung.
Rechtfertigung: Die Gültigkeit dieser Annahme hängt ab von der Varianz des Zusammenhangs von Teilgraphen mit dem Restgraphen in der Topologie der beim Lernen entstehenden neuronalen Netze. Gehen die Lernverfahren von in einem gewissen Sinne vollverbundenen Netzen (im Gegensatz zu modularen Netzen) aus, wie es alle im Rahmen dieser Arbeit betrachteten Verfahren tun, so ist die Annahme plausibel, daß die Varianz des Zusammenhangs von Teilgraphen mit dem Restgraphen relativ gering ist, also keine Partitionierungen existieren, die viel mehr Verbindungslokalität aufweisen als zufällige Partitionierungen.

Von der Gültigkeit der Annahmen 1 bis 4 kann man sich durch eine Betrachtung der existierenden Lernverfahren, wie sie in den Kapiteln 2 bis 6 beschrieben sind, überzeugen. Die Ergebnisse einer solchen Betrachtung für die in dieser Arbeit untersuchten Verfahren sind in Tabelle 10.1 zusammengestellt. Die Annahme 5 wird in Abschnitt 11.5.2 einer experimentellen Überprüfung unterzogen. Die

	Annahme				
Verfahren	1	2	3	4	5
fr. Stoppen	+	++	++	++	++
autoprune	+	+	+	++	∘
lprune	+	+	∘	++	∘
cascor	+	−	+	++	+
cand	+	−	+	++	++
cascade	+	−	+	++	+
kogi2	+	−	+	++	+
kogi3	+	−	+	++	++
kogi9	+	−	+	++	+

Tabelle 10.1: Gültigkeit der Annahmen 1 bis 5 am Beispiel der Lernverfahren des ersten Teils. ++: trifft perfekt zu; +: trifft gut zu; ∘: trifft mit Einschränkungen zu; −: trifft nur in Ansätzen zu; −−: trifft nicht zu.

Ergebnisse dieser Untersuchung sind in der Tabelle vorweggenommen. Zwar gibt es in manchen Verfahren Abweichungen von den Annahmen, diese sind jedoch (mit Ausnahme der gemäß Abschnitt 10.4.3 beherrschbaren Trennung von Kandidatenknoten und alten Knoten in den Kandidatenlernverfahren) stets nur schwach ausgeprägt.

10.1.3 Optimierungstechniken

Der Fluß eines Datums durch ein neuronales Netz ist zumeist nicht auf einen kleinen Teil dieses Netzes beschränkt, sondern betrifft sehr viele seiner Komponenten. Insofern weisen aus algorithmischer Sicht neuronale Netze keine hohe Datenlokalität auf. Ziel eines Übersetzers muß es sein, die restliche Datenlokalität möglichst vollständig auch zu realisieren. Innerhalb dieser Grenzen wird Datenlokalität vom Übersetzer durch die Benutzung einer Klasse von Daten- und Prozeßverteilungen erreicht, die folgendes garantiert:

1. Lokalität von Knotenobjekten bei Knotenoperationen und von Netzobjekten in Netzoperationen in allen Fällen und,
2. Lokalität von Verbindungsobjekten bei Verbindungsoperationen in etwa der Hälfte der Fälle.

Die Hauptidee hinter der Verteilung besteht darin, je nach Anzahl von Verbindungen an einem Knoten nicht nur einen Prozessor pro Knoten zu verwenden, sondern einen ganzen Block von Prozessoren (ggf. virtuellen Prozessoren). Die Knotendaten werden über diese Prozessoren repliziert, und die Verbindungen werden über diese Prozessoren verteilt (siehe Abschnitt 10.3). Dies erlaubt die Datenlokalität für Knoten. Der Prozeß für eine Knotenoperation läuft stets auf allen Prozessoren des Knotens ab. Die Daten von Verbindungsobjekten werden entweder am Ausgangsknoten einer Verbindung gespeichert oder am Zielknoten und werden durch einen Verweis auf das *Gegenende*, also das andere Ende der Verbindung, ergänzt; das Gegenende selbst erhält nur einen Verweis. Dadurch ergibt sich in etwa der Hälfte der Fälle Objektlokalität für Verbindungsoperationen, weil jene stets im Prozessor des aufrufenden Knotens ablaufen. Da Lernverfahren hinsichtlich der Richtung der Benutzung von Verbindungen nie ganz symmetrisch sind, kann durch die „richtige“ Wahl des Speicherungsortes für die Verbindungsdaten (Ausgangsknoten/Zielknoten) stets erreicht werden, daß *mehr* als die Hälfte der Operationen Lokalität aufweist (siehe Abschnitt 10.4.2). Diese Wahl wird stets für alle Verbindungen einer Schnittstelle einer Knotengruppe gleichsinnig getroffen, weil die Knoten gemäß Annahme 2 auch gleichsinnig verwendet werden.

Lastbalancierung wird erreicht, indem die Parameter der Daten- und Prozeßverteilung geschickt gewählt werden. Der wichtigste Aspekt hierbei ist, daß die Summe der Arbeit, die für die Verbindungen der auf jedem Prozessor untergebrachten (Teil-)Knoten anfällt, für jeden Prozessor etwa gleich groß gemacht wird (siehe Abschnitt 10.3).

Die Minimierung unvermeidlicher Kommunikationsoperationen erfolgt über eine einfache Benutzungsanalyse und die Bündelung der Kommunikationsoperationen für eine Verbindungsoperation (siehe Abschnitt 10.4.1).

Für Lernverfahren, die auch Parallelität über Trainingsbeispiele erlauben, ergibt sich eine weitere Optimierung bei der Wahl der richtigen Granularität durch geschickte Wahl der verwendeten Anzahl von Netzreplikaten, die in Abschnitt 10.4.4 beschrieben wird.

10.1.4 Zielmaschinen

Als Zielmaschine für den im Rahmen dieser Arbeit implementierten Übersetzer dient die MasPar MP-1 oder MP-2. Diese beiden Maschinen haben eine identische Architektur und unterscheiden sich nur in der Leistung der einzelnen Prozessoren, die bei der MP-2 etwa 3 mal so hoch ist.

Die MasPar MP-1 und MP-2 sind SIMD Maschinen mit 1024 bis 16384 Prozessoren (genannt Prozessorelemente, PEs), die von einem Kontrollprozessor, der sogenannten ACU, mit Befehlen versorgt werden. Die PEs sind logisch 32-bit Prozessoren und haben physikalisch eine Wortbreite von 4 Bit. Die nachfolgenden technischen Angaben gelten immer für eine MP-1 mit 16384 Prozessoren, da dies die Maschine ist, auf der die im nächsten Kapitel beschriebenen Messungen gemacht wurden. Die theoretische Gesamtleistung beträgt 1200 MFLOPs bzw. 26000 MIPS. Legt man die Dauer einer 32-bit Gleitkommamultiplikation als Zeiteinheit *gkm* zugrunde, so dauert das Laden eines Operanden aus dem Speicher etwa 0,3 gkm und einfache Ganzzahloperationen (d.h. außer Multiplikation etc.) zwischen 0,1 gkm und 0,3 gkm. Die Balance ist also anders als bei heutigen RISC-Mikroprozessoren, die eine relativ schnellere Gleitkommaarithmetik aufweisen. Wenn wir annehmen, daß in neuronalen Algorithmen für die Darstellung von Gewichten und Aktivierungen Gleitkommazahlen benutzt werden, ist jedoch das Verhältnis der Anzahlen von Gleitkomma- zu Lade- zu Ganzzahloperationen recht einheitlich, so daß uns die konkrete Balance der Einzelteile nicht interessieren muß, wenn die Effektivität gewisser Optimierungen im Übersetzer beurteilt werden soll. Diese Situation ist in der vorliegenden Arbeit gegeben.

Die Prozessoranzahl einer MP-1/MP-2 ist stets eine Potenz von 2 und die Prozessoren sind in Form eines zweidimensionalen Gitters angeordnet, dessen Seitenlängen folglich ebenfalls Zweierpotenzen sind. Die Enden des Gitters sind miteinander verbunden, so daß ein Torus entsteht. Auf dieses Gitter ist ein Kommunikationsnetz gelegt, das jeden Prozessor mit seinen 8 Nachbarn verbindet. Auf diesem Netz, genannt *xnet* kann jedoch ausschließlich eine Kommunikation betrieben werden, bei dem alle Prozessoren mit Partnern in derselben Richtung und Entfernung kommunizieren. Diese Kommunikation ist extrem schnell: Senden oder Holen von 32-bit zu oder von einem direkten Nachbarn dauern nur etwa 0,2 gkm, also kürzer als ein Ladebefehl! Diese schnelle Nachbarkommunikation ist eine typische Eigenart von SIMD Maschinen und wird durch deren synchrone Betriebsweise ermöglicht. Allgemeine Kommunikationsoperationen werden durch ein *global router* genanntes zweites Kommunikationsnetzwerk ermöglicht, das für eine beliebige Permutation von 32-bit-Paketen etwa 30 gkm benötigt (siehe Abschnitt 8.3.2).

Im Unterschied zu den meisten speziellen Implementierungen von neuronalen Netzen auf SIMD-Maschinen sind die in meinem Übersetzer verwendeten Techniken *nicht* darauf ausgelegt, ausschließlich die schnelle Nachbarkommunikation zu benutzen. Eine solche Auslegung, wie sie z.B. bei [199] und bei [230] vorgenommen wird, hätte zur Folge, daß die Techniken sich nur mit dramatischen Einbußen auf MIMD-Maschinen übertragen ließen, die bekanntlich eine solche extrem schnelle Nachbarkommunikation wegen fehlender Synchronität der Prozessoren nicht aufweisen. Ferner gibt erst die Nutzung allgemeiner Kommunikationsmuster die volle Flexibilität zur Realisierung beliebiger und dynamisch veränderlicher Topologien von neuronalen Netzen. Auch die übrigen spezifischen Eigenschaften von SIMD-Maschinen werden in den vorgestellten Techniken entweder nicht ausgenutzt oder die Übertragung auf MIMD-Maschinen ist einfach und wird beschrieben. Die weiter unten behandelten Techniken lassen sich also stets auch auf MIMD-Maschinen anwenden. Der Hauptunterschied in der Implementierung im Vergleich zu SIMD-Maschinen liegt dann darin, daß explizite Synchronisation ergänzt werden muß, weil SIMD-Maschinen jederzeit implizit synchron arbeiten. Ich gehe von Maschinen aus, bei denen die Anordnung der Daten im Speicher voll unter Kontrolle des Programms steht und nicht durch komplexe Cache-Mechanismen selbständig verändert wird, wie es z.B. bei der ALLCACHE-Architektur der KSR-Rechner der Fall ist.

Wo im folgenden Algorithmen in Pseudocode ausformuliert werden, wird dies aus Gründen der Visualisierbarkeit und Konkretheit für ein zweidimensionales Gitter von Prozessoren getan. Für andere Netztopologien, die ebenfalls *Nachbarschaften* (siehe unten) aufweisen, wie z.B. 3-D Gitter oder Hyperwürfel, ist die Verallgemeinerung jeweils einfach. In anderen Fällen, z.B. für perfect-shuffle-Netze, muß man die physikalische Verbindungsstruktur auf eine beliebige gewünschte logische Struktur uminterpretieren.

10.2 Definitionen

Um eine exaktere Ausdrucksweise zu ermöglichen, führen wir einige Definitionen zur Beschreibung von Daten- und Prozeßverteilungen und Lastverteilungssituationen ein. Für die Implementation eines Netzes betrachten wir die Unterteilung seiner Knoten in Gruppen. Deshalb benötigen wir eine etwas andere Schreibweise als sie in Abschnitt 2.1 eingeführt wurde.

Es seien G eine Knotengruppe, $|G|$ ihre Knotenanzahl und G_i ihr i-ter Knoten (der erste Knoten hat die Nummer 0). Analoge Schreibweisen gelten für andere Knotengruppen, z.B. H. Es seien ferner $in_{name1}(G)$ und $out_{name2}(G)$ Schnittstellen der Gruppe G für hereinkommende bzw. herausführende Verbindungen und $in_{name1,j}(G_i)$ etc. die j-te Verbindung des i-ten Knotens von G, die an einer solchen Schnittstelle angeschlossen ist. Die Indizes der Verbindungen sind für das Benutzerprogramm unsichtbar und werden nur zur eindeutigen Unterscheidung von Verbindungen verwendet. Normalerweise nehmen wir zur Vereinfachung der Schreibweise an, es gebe nur zwei Schnittstellen und schreiben kürzer $in_j(G_i)$ und $out_j(G_i)$. Dies ist für alle hier betrachteten Verfahren ausreichend. Außerdem schreiben wir z.B. „die Schnittstelle $s(G)$", wenn uns die Kategorie (hereinkommend/herausführend) der Schnittstelle nicht interessiert und $S(G)$ für die Menge aller Schnittstellen von G. Wir betrachten die Schnittstelle als äquivalent zur Menge der daran angeschlossenen Verbindungen. Die Anzahl von Verbindungen an einer Schnittstelle eines bestimmten Knotens notieren wir als $|s(G_i)|$. Das andere Ende einer Verbindung beschreibt die selbstinverse Funktion E. Wenn also $out_k(H_l)$ und $in_j(G_i)$ die beiden Enden derselben Verbindung bezeichnen, dann gelten die Invarianten

$$\begin{aligned}
E(out_k(H_l)) &= in_j(G_i) \\
out_k(H_l) &= E(in_j(G_i)) \\
E(E(out_k(H_l))) &= out_k(H_l) \\
E(E(in_j(G_i))) &= in_j(G_i)
\end{aligned}$$

Dabei gibt es zwei verschiedene Betrachtungsweisen: Logisch, d.h. aus Sicht des Benutzerprogramms, sind beide Enden äquivalent und bezeichnen dasselbe Verbindungsobjekt. Aus Sicht der Implementation hingegen (physische Datenverteilung) stellt nur ein Ende, z.B. $in_j(G_i)$, tatsächlich ein vollständiges Verbindungsobjekt dar (genannt *lokales Verbindungsobjekt*), das andere Ende $out_k(H_l)$ ist lediglich ein Verweis auf dieses Objekt (genannt *entferntes Verbindungsobjekt*). Die Verweise (in beide Richtungen) und damit die Funktion E werden technisch durch ein Paar von Zeigern realisiert, die sogenannten *Gegenenden-Zeiger*. Diese Anordnung des eigentlichen Verbindungsobjekts gilt immer gleichermaßen für alle Verbindungen derselben Schnittstelle $s(G)$. Die Anordnung wird ausgedrückt durch das Prädikat *lokal*, das angibt, ob eine logisches Verbindungsobjekt auch ein lokales Verbindungsobjekt bezeichnet oder nicht. Es gilt invariant

$$lokal(s_j(G_i)) \quad \Longleftrightarrow \quad \neg\, lokal(E(s_j(G_i)))$$

lokal kann analog auch auf Schnittstellen im Ganzen angewendet werden.

Die Verteilung der Teilobjekte des Netzes auf die Prozessoren wird mit folgenden Schreibweisen beschrieben: Die p Prozessoren der Zielmaschine werden durch die natürlichen Zahlen $0 \ldots p-1$ bezeichnet. Die Abbildung der Objekte auf Prozessoren wird durch die Funktion P beschrieben. Für ein Verbindungsobjekt liefert P die Nummer des Prozessors, auf dem es angeordnet ist, z.B. $P(in_j(G_i)) = \{17\}$; für Knoten liefert P die Menge der Nummern aller Prozessoren, über die der Knoten samt seinen Verbindungen verteilt ist, z.B. $P(G_i) = \{16, 17, 18, 19\}$ oder gegebenenfalls auch nur $P(G_i) = \{17\}$. Eine solche Menge nennen wir einen *Knotenblock*.

Um die Lastverteilungssituation beschreiben zu können, benötigen wir eine Schreibweise für den Laufzeitaufwand t, den Operationen verursachen. Wir bezeichnen die Operationen (also Netz-, Knoten- oder Verbindungsprozeduren) mit op_j, wobei der Index die verschiedenen Prozeduren über alle Typen hinweg eindeutig bezeichne. Für die Lastverteilung sind nur die Verbindungsoperationen interessant. Nach Annahmen 1 und 4 ist die Rechenarbeit t_R für jede Verbindungsoperation bei jeder Anwendung ungefähr gleich. Für die Gesamtausführungszeit der Operation kommt jedoch eventuell noch die Kommunikationszeit t_K hinzu: Sind die Daten der Verbindung nicht lokal vorhanden, so müssen die gelesenen Teile der Verbindung vor der Operation geholt und die veränderten nach der Operation zurückgeschrieben werden. Wir schreiben in diesem Fall $t = t_R + t_K$, während ansonsten, also bei lokal vorhandenem Verbindungsobjekt, $t = t_R$, da $t_K = 0$.

Die Replikation von Netzen beschreiben wir durch eine Transformation der Prozessornummern: Anstatt einer Maschine mit p Prozessoren denken wir uns bei n Replikaten n Maschinen mit je $\lfloor p/n \rfloor$ Prozessoren, die jeweils von 0 bis $\lfloor p/n \rfloor - 1$ numeriert sind; eine Replikation von Netzen über die Zahl physikalischer Prozessoren hinaus ist sinnlos. Die Daten- und Prozeßverteilung ist dann für jede dieser gedachten Maschinen exakt gleich; die Unterschiede ergeben sich ausschließlich über den Betrieb der Maschinen mit verschiedenen Daten. Die Prozessorteilmenge der echten Maschine für jede solche gedachte Maschine nennen wir ein *Segment*. Alle Segmente müssen gleiche Größe und Form (Topologie) haben, um in jedem genau die gleiche Daten- und Prozeßverteilung verwenden zu können.

Die meisten gebräuchlichen Topologien für Kommunikationsnetze bei Parallelrechnern enthalten *Nachbarschaften*. Eine Nachbarschaft ist eine Teilmenge der Prozessoren mit folgenden zwei Eigenschaften: Erstens verlaufen alle kürzesten Wege zwischen zwei Prozessoren der Nachbarschaft vollständig innerhalb der Nachbarschaft und zweitens ist der Durchmesser einer Nachbarschaft minimal für die Anzahl enthaltener Prozessoren. Beispielsweise sind Nachbarschaften in 2-D Gittern immer Rechtecke, in Baumnetzen immer Teilbäume. Nicht zu jeder beliebigen Prozessoranzahl existiert in einem gegebenen Netz eine Nachbarschaft. Segmente (siehe oben) wählen wir in Größe und Form stets als Nachbarschaften. Manche Netze enthalten keine Nachbarschaften mit mehr als einer bestimmten Zahl von Prozessoren (z.B. enthalten Perfect-Shuffle-Netze nur Nachbarschaften bis Größe 2). Wir werden Nachbarschaften verwenden, um die Verkehrsdichte im Kommunikationsnetz zu verringern, indem wir Datenobjekte, von denen wir im voraus wissen, daß sie miteinander verknüpft werden, immer über Nachbarschaften anstatt über beliebige Prozessorteilmengen verteilen.

Unter *Virtualisierung* verstehen wir die softwaremäßige Bereitstellung von v logischen Prozessoren (genannt *virtuelle Prozessoren*) auf jedem physikalischen Prozessor. Zur Realisierung einer Virtualisierung arbeitet für die parallel auf dem verteilten Objekt x auszuführende Operation *op* jeder physikalische Prozessor in einer Schleife der Reihe nach die Operation *op* für v verschiedene Teilobjekte von x ab, anstatt nur für ein Teilobjekt zuständig zu sein, wie es die logische Sicht vorspiegelt. Dabei heißt v der *Virtualisierungsgrad*.

10.3 Datenlokalität und Lastbalance

Die in diesem Abschnitt beschriebene Klasse von Daten- und Prozeßverteilungen stellt das Herzstück der Architektur der Codeerzeugung für KNA-Übersetzer dar. Ihre Anwendung ist die wichtigste Optimierung bei der Übersetzung von KNA-Programmen, die weiteren in dieser Arbeit vorgestellten Optimierungen dienen hauptsächlich zur möglichst guten konkreten Wahl der *Parameter der Daten- und Prozeßverteilung*, sind jenen also untergeordnet.

10.3.1 Prinzip der Daten- und Prozeßverteilung

Der Daten- und Prozeßverteilung liegen zwei Prinzipien zugrunde, die zu den beiden Hauptoptimierungszielen von Datenlokalität und Lastbalancierung korrespondieren:

1. Die Netzobjekte, Knotenobjekte und Verbindungsobjekte, auf denen eine Netzoperation, Knotenoperation bzw. Verbindungsoperation arbeitet, sollen gemäß Datenverteilung von vornherein jeweils lokal auf dem Prozessor vorliegen, auf dem die Operation ausgeführt wird (*Datenlokalität*). Der Kontrollfluß soll ebenfalls lokal sein, d.h. bei einem Prozeduraufruf soll der die gerufene Prozedur ausführende Prozeß auf dem gleichen Prozessor wie der Aufrufer bleiben (*Prozeßlokalität*); insbesondere auch bei Prozeduraufrufen, die den Parallelitätsgrad erhöhen. Bei Verbindungsobjekten kann diese Prozeßlokalität nur in einem Teil der Fälle mit der Datenlokalität vereinbart werden, weil Operationen auf Verbindungsobjekten von zwei verschiedenen Stellen aus aufgerufen werden können (Prozessor des Ausgangsknotens, Prozessor des Zielknotens). Wir geben der Prozeßlokalität den Vorzug, was insbesondere dann von Vorteil ist, wenn die Zielmaschine das asynchrone Anstoßen mehrerer Kommunikationsoperationen desselben Prozessors zur Verbergung der Latenzzeit erlaubt, denn mit Prozeßlokalität kann eine Lastbalancierung explizit an den aufrufenden Knoten vorgenommen werden (siehe unten), während sie bei Fernprozeduraufrufen von der Gleichmäßigkeit der Verteilung der *Ziel*knoten über die Maschine abhinge.
2. Die Menge von Prozessorresourcen, die einem Knoten zugeteilt werden, ist proportional zur im Mittel für die Operationen auf den Verbindungen dieses Knotens zu leistenden Arbeit. Durch solche Zuteilung wird gemäß der Annahmen 1 und 2 aus Abschnitt 10.1.2 eine Lastbalancierung erreicht. „Im Mittel" bezieht sich dabei immer auf den Programmabschnitt zwischen zwei Änderungen der Netztopologie. Diese lange Mittelwertbildung verschlechtert die Lastbalancierung nicht gegenüber der Mittelwertbildung über kürzere Abschnitte, wie man aus Annahme 3 entnehmen kann. Ferner ziehen wir zur Berechnung des Mittels, welches sich ja bei der Herstellung der Datenverteilung auf die Zukunft bezieht, den im jeweils letzten Programmabschnitt (oder in allen vorhergehenden Abschnitten) gemessenen Wert heran, was aufgrund von Annahme 4 keine Einschränkung ist. Eine dynamische Lastbalancierung für *einzelne* Verbindungsoperationen ist nicht mit Datenlokalität vereinbar und wird deshalb verworfen.

Zur Illustration der Techniken, die zur Umsetzung dieser Prinzipien dienen, werden wir im folgenden mehrfach Abbildungen heranziehen, die verschiedene Aspekte der Datenverteilung für das in der Abbildung 10.2 dargestellte kleine Netz veranschaulichen.

Folgende Techniken werden bei der Umsetzung dieser Prinzipien angewandt:

Technik 1: Datenreplikation. Um die Datenlokalität beim Aufruf von Knotenoperationen durch Netzoperationen und soweit möglich beim Aufruf von Verbindungsoperationen durch Knotenoperationen herzustellen, werden Daten repliziert. Die lokalen Daten eines Netzobjekts (abgesehen von Knoten und Verbindungen) werden über alle Prozessoren repliziert, über die die Knoten und Verbindungen des Netzes verteilt sind. Die lokalen Daten eines Knotenobjekts (abgesehen von den Verbindungen) werden ggf. über alle Prozessoren repliziert, über die die Verbindungen des Knotens verteilt sind. Die Konsistenzerhaltung dieser Replikate funktioniert folgendermaßen: Soweit die Operationen diese replizierten Daten nur *miteinander* verknüpfen, werden die Operationen ebenfalls repliziert und dadurch

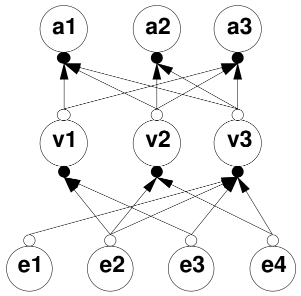


Abbildung 10.2: Beispielnetz: Vorwärtsgerichtetes 4:3:3 Netz mit Eingangsschicht e, verborgener Schicht v und Ausgangsschicht a. Die Verbindungsstruktur ist unregelmäßig. Jede Schicht entspricht einer Knotengruppe. Die kleinen Kreise symbolisieren Schnittstellen. Die Verbindungsobjekte werden wir später bei den ungefüllten Kreisen allokieren.

die Daten im Einklang gehalten. Treten Verknüpfungen mit Daten auf, die *extern* zum replizierten Objekt sind, so werden diese zuvor ebenfalls repliziert und dann wie andere replizierte Daten behandelt. Solche externen Daten treten in mehreren Formen auf:

1. Ergebnisse von Reduktionen über Verbindungen als Daten in Knotenoperationen. Hier entsteht das Reduktionsergebnis im ersten Prozessor des Knotenblocks und wird dann sofort auf alle Prozessoren des Knotenblocks verteilt. Dieses Verteilen verursacht zusätzlichen Laufzeitaufwand, der dem Kommunikationsaufwand (ohne Rechenaufwand) der Reduktion entspricht, d.h. die Laufzeit der Reduktion wird höchstens verdoppelt.
2. Ergebnisse von Reduktionen über Knoten als Daten in Netzoperationen. Dieser Fall ist analog zum obigen.
3. Eingabe von Knotendaten. Für jeden Knoten muß das eingelesene Datum auf alle Prozessoren des Knotenblocks verteilt werden. Die Laufzeit einer solchen Operation liegt also unter der einer Reduktion über Verbindungen.
4. Argumente für Aufrufe von Netzoperationen aus dem sequentiellen Programmteil. Diese Argumente werden mit einem vollständigen Rundruf (*Broadcast*) auf alle Prozessoren repliziert.

Die folgende Überlegung läßt uns die Alternative „keine Datenreplikation durchführen“ verwerfen: Jedes sinnvolle Programm berechnet solche oben angeführten externen Daten nur dann, wenn sie später auch benötigt werden[1]. Folglich müssen die zur Replikation notwendigen Kommunikationsoperationen irgendwann sowieso für dieselben oder durch Weiterverarbeitung entstandene neue Daten durchgeführt werden. Da die Weiterverarbeitung in neuronalen Algorithmen die Datenmenge nicht mehr oder nur geringfügig weiter reduziert, kann ein Verzicht auf Datenreplikation fast keine Ersparnis bringen. Andererseits können aber sehr wohl dieselben Daten mehrmals verwendet oder ein Datum in mehrere Argumente für einen Prozeduraufruf umgesetzt werden; in diesem Falle spart die Datenreplikation Kommunikation ein.

Technik 2: Knotenlastproportionale Prozessorzuweisung. Die Last L für einen Knoten G_i ergibt sich aus der Anzahl von Verbindungen $|s(G_i)|$ an jeder seiner Schnittstellen s, der Anzahl $|op_j(s)|$ von Aufrufen jeder Verbindungsoperation op_j für jede Schnittstelle und der Laufzeit $t(op_j(s))$ für jeden solchen Aufruf in Abhängigkeit von der Schnittstelle, über die er erfolgt:

$$L(G_i) = \sum_s \left(|s(G_i)| \sum_{op_j} |op_j(s)| \; t(op_j(s)) \right)$$

Da die meisten dieser Parameter nicht im Voraus berechnet werden können, verwende ich zur Bestimmung von L eine Messung (siehe Abschnitt 10.3.5). Damit ist sichergestellt, daß alle Einflußgrößen in ihrem tatsächlichen Ausmaß berücksichtigt werden. Ist die Last für jeden Knoten bekannt, so ergibt

[1] Da der Kontrollfluß in neuronalen Algorithmen simpel ist, tritt der Fall nicht auf, daß bei der Berechnung eines Datums noch nicht entschieden werden kann, ob es später benötigt wird oder nicht.

sein Anteil an der Gesamtlast der Knotengruppe seinen Anteil an den verfügbaren Prozessoren. Diese Betrachtung wird für jede Knotengruppe einzeln durchgeführt und ist in Abschnitt 10.3.5 genauer beschrieben.

Technik 3: Implizite Adressierung mit Deskriptoren. Da die Zuweisung der Prozessoren an die Knoten nach Last erfolgt, ist die Größe von Knoten, gemessen in Prozessoren, unterschiedlich. Ferner sollte die Reihenfolge (Anordnung) der Knoten so gewählt werden, daß möglichst geringer Verschnitt von Prozessorresourcen entsteht. Dies ist wichtig sowohl bei Verwendung von nicht-ganzzahligen Knotenblockgrößen als auch bei ganzzahligen, wenn günstig geformte Knotenblöcke benutzt werden sollen (siehe Abschnitt 10.3.6). Die unregelmäßige Datenverteilung, die sich hieraus ergibt, hat zur Folge, daß keine einfache Adressierungsfunktion mehr zur direkten Berechnung der Adressen von Objekten existiert. Deshalb werden den Objekten Deskriptoren zugeordnet, anhand derer sich lokal bestimmen läßt, welches die für eine Operation relevanten Objekte sind. Ferner enthalten die Deskriptoren alle dynamisch veränderlichen Angaben, die zur Beschreibung der aktuellen Netztopologie nötig sind (siehe Abschnitt 10.3.2). Da die *Herstellung* bzw. Reorganisation der Datenverteilung mit ausschließlich impliziter Adressierung sehr kompliziert wird, werden zwei verschiedene Datenverteilungen benutzt: Für die Rechenphasen des Programms wird eine als *Form A* bezeichnete Datenverteilung hergestellt, die nach den hier beschriebenen Prinzipien Lokalität und Lastbalance herstellt und eine unregelmäßige Verteilung der Knotenobjekte über die Prozessoren bewirkt. Für die Phasen, in denen Topologieveränderungen stattfinden, wird stattdessen eine regelmäßige Datenverteilung benutzt, genannt *Form 0*, mit der sich teilweise auch direkte (explizite) Adressierung durchführen läßt, was zu effizienteren Operationen bei der Herstellung und Reorganisation der Datenverteilung führt. Im folgenden wird immer nur die Form A beschrieben. Die bestimmenden Eigenschaften von Form 0 sind, daß erstens alle Knotenblöcke dieselbe Größe haben und in der Reihenfolge der Knotenindizes angeordnet sind und daß zweitens nur ein Exemplar des Netzes existiert, also Replikate nicht möglich sind. Letzteres wird von der CuPit Sprachdefinition ohnehin gefordert, weil Topologieveränderungen auf replizierten Netzen zu uneindeutigen Topologien führen könnten, die sich nicht mehr zusammenführen lassen.

Technik 4: Lokale Prozeduraufrufe. Gegeben alle diese Voraussetzungen wird die Prozeßverteilung einfach: Alle parallelen Prozeduraufrufe, insbesondere solche, die den Parallelitätsgrad erhöhen (Netz ruft Knoten, Knoten ruft Verbindungen), benötigen keinerlei Übertragung von Kontrolle auf andere Prozessoren. Stattdessen wird jeweils lokal auf jedem Prozessor ein Prozeduraufruf durchgeführt und dann anhand der Deskriptoren entschieden, welche Operationen der Prozessor auszuführen hat.

Anmerkung: Die Behandlung von Netzreplikaten zur Ausnutzung der Beispielparallelität funktioniert, wie schon in Abschnitt 10.2 erwähnt, immer über die Herstellung exakt kongruenter Kopien der Datenverteilung (mit Transformation der Prozessornummern). Aus diesem Grund betrachten wir Netzreplikate im folgenden meist nicht mehr separat.

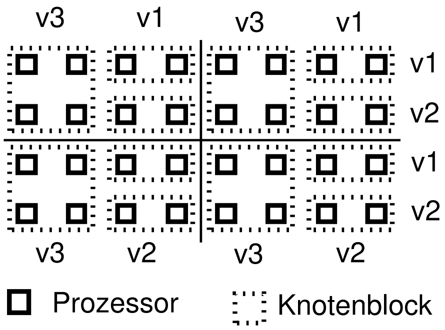


Abbildung 10.3: Knotendatenreplikation, knotenlastproportionale Prozessorzuweisung und Netzreplikation für verborgene Schicht (Knoten v1, v2, v3) des Beispielnetzes auf 2-D Prozessorfeld von 8 mal 4 Prozessoren, wenn 4 Netzreplikate verwendet werden. Netzreplikatgrenzen sind durch die durchgezogenen geraden Linien angedeutet.

Abbildung 10.3 gibt die Umsetzung der Techniken 1 und 2 sowie die Behandlung von Netzreplikaten für die Gruppe v des Beispielnetzes aus Abbildung 10.2 für ein 2-D Prozessorfeld an, wobei 4 Netzreplikate verwendet werden. Der Knotenblock für den Knoten v3 ist am größten, weil dieser an seinem Ein- und Ausgang die meisten Verbindungen und damit die höchste Last aufweist. Für die

Details der Entscheidung über die Knotenblockgrößen und die Anordnung der Knotenblöcke siehe Abschnitte 10.3.5 und 10.3.6.

10.3.2 Deskriptoren

Den einzelnen Objektkategorien sind jeweils Deskriptoren zugeordnet, die die unten aufgeführten Informationen enthalten. Deskriptoren werden wie Inhaltsdaten auf alle beteiligten Prozessoren repliziert. Je nach konkreter Implementierung müssen die Deskriptoren eventuell zusätzliche Informationen außer den aufgeführten aufnehmen.

Netzdeskriptor: Der Netzdeskriptor ist jeweils identisch auf allen Prozessoren, die zum selben Netzreplikat gehören. Jedes Netz hat einen eigenen Netzdeskriptor. Er enthält einen *Existenzanzeiger*, aufgrund dessen sich ggf. unbenutzte Prozessoren abschalten können, einen *Netzreplikatindex*, mit dem sich ein Netzreplikat selbst identifizieren kann, und die *Anzahl existierender Netzreplikate*, die es im Verbund mit dem statischen Wissen über Prozessoranzahl und -numerierung erlaubt, die zum eigenen Netzreplikat gehörende Prozessormenge und die zu einem Prozessor korrespondierenden Prozessoren der anderen Replikate zu berechnen.

Knotengruppendeskriptor: Der Knotengruppendeskriptor ist identisch auf allen Prozessoren. Jede Knotengruppe hat einen eigenen Knotengruppendeskriptor; er enthält die momentane Anzahl von Knoten der Gruppe und den *Knotenvirtualisierungsgrad*, d.h. die Anzahl auf jedem Prozessor (höchstens) vorhandener Knoten der Gruppe.

Knotendeskriptor: Der Knotendeskriptor ist identisch auf allen Prozessoren, die zum selben Knoten gehören. Jeder Knoten hat einen eigenen Knotendeskriptor; er enthält die *Knotennummer* des Knotens, d.h. seinen Index innerhalb seiner Knotengruppe, sowie einen *Existenzanzeiger* zur Erkennung unbenutzter physikalischer Knotendatenobjekte.

Schnittstellendeskriptor: Der Schnittstellendeskriptor ist identisch auf allen Prozessoren desselben Knotenblocks. Jeder Knoten in jeder Knotengruppe hat einen eigenen Schnittstellendeskriptor für jede seiner Schnittstellen; er enthält die *Verbindungsanzahl*, d.h. die Anzahl momentan tatsächlich existierender Verbindungen, und den *Verbindungsvirtualisierungsgrad*, d.h. die Größe des zur Speicherung der angeschlossenen Verbindungen dieser Schnittstelle derzeit allokierten Feldes auf diesem Prozessor. Ferner enthält er ein Zeitfeld, in dem die Meßwerte zur Bestimmung der Knotenlast im Hinblick auf diese Schnittstelle summiert werden.

Verbindungsdeskriptor: Jede Verbindung hat einen eigenen Verbindungsdeskriptor. Replikation findet nicht statt. Der Verbindungsdeskriptor enthält einen *Existenzanzeiger*, mit dem unbenutzte Verbindungsobjekte als solche identifiziert werden können, einen *Verbindungsindex* (bezogen auf die Verbindungen an derselben Schnittstelle desselben Knotens), der zur Adreßberechnung bei der Herstellung der Datenverteilung benötigt wird, und einen *Verweis auf das Gegenende* der Verbindung.

Zusätzlich enthält jeder Deskriptor einen Verweis auf den zugehörigen Deskriptor der nächst höheren Ebene (Verbindung → Schnittstelle → Knoten → Knotengruppe → Netz), so daß z.B. für eine Verbindung festgestellt werden kann, zu welchem Netzreplikat sie gehört. Siehe Abschnitt 11.6 für eine Diskussion des Speicherbedarfs der Deskriptoren.

10.3.3 Prinzip der Codeerzeugung

Das Prinzip der Codeerzeugung ist einfach. Es werden keine tiefgreifenden Transformationen des Programmtextes durchgeführt, sondern überwiegend wird jeder Teil des KNA-Quellprogramms in einen unmittelbar entsprechenden Teil des Zielprogramms übersetzt, wobei lediglich für viele Teile die Zieldarstellung relativ komplex ist. Zur Beschreibung dokumentiere ich für die wichtigsten Arten von

Elementen des Quellprogramms jeweils kurz die Methode der Umsetzung. Alle maschinenspezifischen Details können in [16] nachgelesen werden.

Aufrufe von Netzoperationen erfordern einen Rundruf von dem Prozessor, der den sequentiellen Programmteil ausführt, auf alle Prozessoren. In diesem Rundruf werden die Parameter übergeben und allen teilnehmenden Prozessoren die auszuführende Prozedur benannt. Die Prozessoren beginnen dann asynchron mit der Ausführung dieser Prozedur. Nach deren Ende werden die Prozessoren wieder synchronisiert; auf die dafür zu verwendenden Mechanismen gehe ich nicht weiter ein, da sie maschinenabhängig sind. Der Aufruf von Eingabe- und Ausgabeoperationen funktioniert ähnlich wie der Aufruf von Netzoperationen.

Netzoperationen werden eins zu eins übersetzt in parallel auf jedem Prozessor lokal auszuführende Prozeduren.

Aufrufe von Knotenoperationen werden übersetzt in Aufrufe von Knotenoperations-Virtualisierungsprozeduren (siehe unten). Dabei wird neben den Aufrufparametern der vom Aufruf betroffene Teilbereich (slice) der Knotengruppe durch Angabe eines Minimal- und eines Maximalindex mit übergeben. Aufgrund der Datenreplikation auf Ebene der Netzdaten ist hierbei kein Rundruf notwendig.

Knotenoperationen werden übersetzt in ein Paar aus Knotenoperations-Virtualisierungsprozedur und eigentlicher Knotenoperationsprozedur. Die Virtualisierungsprozedur enthält lediglich eine Schleife, die über alle lokal vorhandenen Knotenobjekte der jeweiligen Gruppe iteriert und für jedes Objekt, das an der Operation beteiligt ist, die eigentliche Knotenoperation aufruft; am Ende synchronisieren sich die Prozessoren. Die eigentliche Knotenoperationsprozedur ist eine direkte Umsetzung der Knotenoperation aus dem KNA-Quellprogramm, der neben den dort angegebenen Aufrufparametern jeweils ein Verweis auf das Knotenobjekt übergeben wird, auf dem sie arbeiten soll. Die Trennung in zwei Prozeduren ist notwendig, weil sich Knotenprozeduren gegenseitig aufrufen können, wobei *keine* erneute Virtualisierung fällig wird.

Aufrufe von Verbindungsoperationen werden übersetzt in Aufrufe von Verbindungsoperations-Virtualisierungsprozeduren (siehe unten). Dabei werden die Aufrufparameter und ein Verweis auf die betroffene Verbindungsschnittstelle übergeben. Aufgrund der Datenreplikation auf der Ebene der Knotendaten ist hierbei kein Rundruf notwendig.

Verbindungsoperationen werden übersetzt in ein Tripel aus zwei Verbindungsoperations-Virtualisierungsprozeduren und einer eigentlichen Verbindungsoperationsprozedur. Die Virtualisierungsprozeduren enthalten lediglich eine Schleife, die über alle lokal vorhandenen Verbindungsobjekte der jeweiligen Gruppe iteriert und für jedes Objekt, dessen Deskriptor es als existent bezeichnet, die eigentliche Verbindungsoperation aufruft; am Ende synchronisieren sich die Prozessoren des Knotenblocks. Der Unterschied zwischen den beiden Virtualisierungsprozeduren liegt in der Art des Zugriffs auf das Verbindungsobjekt: Eine Virtualisierungsprozedur wird für die Aufrufe der Verbindungsoperation auf die lokal liegenden Verbindungsobjekte benutzt und funktioniert durch Übergeben der Adresse an die eigentliche Verbindungsprozedur. Die andere Virtualisierungsprozedur wird für die Aufrufe der Verbindungsoperation auf die entfernt liegenden Verbindungsobjekte benutzt und muß vor Aufruf der eigentlichen Verbindungsprozedur ein virtuell lokales Verbindungsobjekt konstruieren. Dazu werden alle von benötigten Komponenten des Verbindungstyps aus dem entfernt liegenden Verbindungsobjekt geholt und in eine lokale Variable vom Verbindungstyp geschrieben. Die veränderten Komponenten werden nach der Operation in das entfernte Objekt zurückgeschrieben. Zur Optimierung dieser Kommunikation siehe Abschnitt 10.4.1. Offensichtlich kann diese Virtualisierungsprozedur auf sehr einfache Weise *prefetch*-Mechanismen zur *Latenzzeitverbergung* ausnutzen, wenn die Zielmaschine dies unterstützt, denn die Identität aller benötigten Objekte ist beim Betreten der Virtualisierungsprozedur bereits bekannt. Die eigentliche Verbindungsoperationsprozedur ist eine direkte Umsetzung der Verbindungsoperation aus dem KNA-Quellprogramm, der neben den dort angegebenen Aufrufparametern jeweils ein Verweis auf das Verbindungsobjekt übergeben wird, auf dem sie arbeiten soll.

Aufrufe von Reduktionsoperationen werden übersetzt in Aufrufe geeignet erzeugter spezieller Reduktionsprozeduren. Dabei wird der verwendete Reduktionsoperator mittels eines Prozedurzeigers als Argument übergeben.

Reduktionsoperationen: Es gibt im realisierten Übersetzer für jeden Werttyp, auf dem eine Reduktion durchgeführt wird, maximal eine solche Prozedur für alle Knotentypen und maximal eine für alle Verbindungstypen. Eine Reduktionsprozedur führt aufgrund der Angaben in den Deskriptoren die benötigten Schleifen und Kommunikationsoperationen durch, um die verteilten Datenobjekte zu reduzieren und das Resultat wieder an alle beteiligten Prozessoren zu verteilen. Die jeweils verwendete konkrete Reduktionsoperation wird als Argument übergeben und mit einer separaten Prozedur realisiert, die durch direkte Übersetzung der entsprechenden KNA-Reduktionsoperation entsteht.

Topologieverändernde Operationen auf der Ebene von Netzen, Knoten oder Verbindungen werden größtenteils in Aufrufe von komplexen Prozeduren umgesetzt, wie im folgenden Abschnitt angegeben.

10.3.4 Topologieverändernde Operationen

In diesem Abschnitt beschreibe ich knapp, wie die einzelnen topologieverändernden Operationen realisiert werden. Die maschinenspezifischen Details sind wiederum in [16] nachzulesen.

Selbstzerstörung von Verbindungen. Vernichtet sich ein Verbindungsobjekt in einer Verbindungsoperation selbst, so wird dies einfach im Existenzanzeiger im Deskriptor des Objektes vermerkt. Diese Änderung muß von der Verbindungsoperations-Virtualisierungsprozedur auch im Deskriptor des entgegengesetzten Endes der Verbindung nachgetragen werden. Das physikalische Löschen der Verbindung erfolgt erst beim Reorganisieren der Netzdatenstruktur beim nächsten Übergang in die Form A Datenverteilung.

Entfernen von Verbindungen zentral aus Netzprozeduren heraus funktioniert analog, nur daß es keine spezielle Unterscheidung von lokalem Ende und Gegenende gibt, weil beide Enden direkt bekannt sind.

Zufügen von Verbindungen. Die neuen Verbindungen werden erzeugt und initialisiert, wobei zuvor auf allen beteiligten Prozessoren, auf denen nicht genügend freie Einträge in den lokalen Feldern zur Speicherung der neuen Verbindungen vorhanden sind, diese lokalen Felder vergrößert neu allokiert werden müssen und wobei bestehende Verbindungen in das jeweils neue Feld verlagert und auf ihrer Gegenseite entsprechend korrigiert werden. Da das Zufügen von Verbindungen auf der Ebene einer Netzoperation geschieht, sind keine besonderen Auswirkungen auf die Ausführung aktueller Knotenoperationen zu berücksichtigen.

Selbstzerstörung von Knoten bewirkt zunächst eine Markierung des Knotens und aller seiner Verbindungen (einschließlich derer Gegenenden) als nicht existent. Danach wird die Knotengruppe des Knotens reorganisiert, um die korrekten neuen Knotennummern zu berechnen und wieder eine Verteilung der Knoten herzustellen, in der die von der Form 0 Datenverteilung vorgesehene direkte Adressierung der Knotenobjekte möglich ist. Dieser Prozeß funktioniert mit kleinen Änderungen wie die Herstellung der Datenverteilung, die in den Abschnitten 10.3.5 bis 10.3.7 beschrieben ist.

Zentrales Zufügen/Entfernen von Knoten wird ebenfalls über eine Reorganisation der Datenverteilung der Knotengruppe durchgeführt. Beim zentralen Entfernen von Knoten könnte im Prinzip auf eine Reorganisation verzichtet werden, indem nur die gelöschten Objekte als nicht existent markiert werden. Da dies zu einer mangelhaften Auslastung der Maschine führen kann, wird jedoch stets reorganisiert.

Replikation von Netzen. Bei der Replikation von Netzen gibt es zwei Richtungen. Erstens das Erzeugen von Replikaten. Dies entspricht zugleich der Überführung der Datenverteilung von Form 0 (regelmäßige Datenverteilung mit der Möglichkeit direkter Adreßberechnung und der Erlaubnis zu topologieverändernden Operationen) nach Form A (unregelmäßige Datenverteilung mit Lastbalancierung, Adreßberechnung nur implizit über Deskriptoren möglich, topologieverändernde Operationen verboten). Wegen der unterschiedlichen Datenverteilungen gibt es auch den Fall, daß ein Netz zwar nur einmal vorhanden ist, aber dennoch den Zustand „repliziert" hat (d.h. Form A). Zweitens gibt es die Richtung des Zusammenführens von Replikaten, mit dem Ergebnis eines nicht replizierten Netzes in Form 0. Überraschenderweise sind die Operationen, die für diese beiden Richtungen nötig sind, zu einem beträchtlichen Teil identisch; der allgemeinere und kompliziertere Fall ist die Überführung von Form 0 nach Form A. Dies ist zugleich der Schritt, der die Datenverteilung herstellt, mit der sich Datenlokalität und Lastbalancierung optimieren lassen.

Aus diesen Gründen beschreibe ich in den folgenden Abschnitten die Teilschritte dieser Überführung, die in folgenden Phasen abläuft:

1. Festlegung der Anzahl zu erzeugender Replikate und entsprechende Unterteilung der Prozessormenge in Segmente (Abschnitt 10.4.4).
2. Für jede Knotengruppe: Berechnung der Blockgröße für jeden Knoten (Abschnitt 10.3.5).
3. Berechnung der Verteilung der Knotenblöcke über die Prozessoren (Abschnitt 10.3.6).
4. Kopieren der Knotenobjekte und Verbindungsobjekte aus der alten Darstellung in das Segment des ersten Replikats der neuen Darstellung (Abschnitt 10.3.7).
5. Wiederherstellung der korrekten Zeiger der Verbindungen auf ihr jeweiliges Gegenende (Abschnitt 10.3.7).
6. Replizieren des ersten Replikats auf die Segmente der übrigen Replikate mit identischer Datenverteilung und Korrektur der Prozessornummern in den Adressen der Replikate (Abschnitt 10.3.7).

Nehmen wir die grundsätzlichen Ideen der Datenverteilung wie bereits beschrieben schon als gegeben an, dann sind die Schritte „Berechnung der Blockgröße" und „Berechnung der Blockverteilung" die Schlüssel zum Erreichen von Lastbalance.

Die Prozedur zur Herstellung der Datenverteilung Form A aus der Form 0 liest sich in Pseudocode wie folgt (kursiv gesetzte Teile werden später noch genauer erklärt):

```
Erzeuge Form A Datenverteilung:
  Bestimme Anzahl r von Replikaten;
  Bestimme Segmente und betrachte nur Segment 0 der neuen Verteilung;
  Kopiere globale Daten des alten Netzes auf die Prozessoren von Segment 0;
  FORALL Knotengruppen G DO
    Berechne Knotenblockgrößen(G);
    Berechne Knotenblockverteilung(G);
    Stelle neue Datenverteilung her(G);
  END;
  Mache neue Datenverteilung betriebsbereit;
  FORALL Knotengruppen G DO
    FORALL Schnittstellen s(G) DO
      Gib Speicher der alten Verbindungsobjekte frei;
    END;
    Gib Speicher der alten Knotenobjekte frei;
  END;
  Gib Speicher des alten Netzobjekts frei;
  Kopiere neue Datenverteilung von Segment 0 auf alle Segmente;
  Transformiere Prozessornummern in allen Segmenten außer 0;
```

Die in obigem Pseudocode kursiv gedruckten Teile werden in den folgenden Abschnitten diskutiert.

10.3.5 Berechnung der Knotenblockgröße

Die Berechnung der Knotenblockgröße wird für jede Knotengruppe getrennt durchgeführt. Gemäß Abschnitt 10.3.1 ist die Idee der Lastbalancierung in der vorgeschlagenen Daten- und Prozeßverteilung, daß die Größe jedes Knotenblocks proportional zur Last am betreffenden Knoten sein soll. Die Last L für einen Knoten G_i ist wie bereits erwähnt

$$L(G_i) = \sum_s \left(|s(G_i)| \sum_{op_j} |op_j(s)| \; t(op_j(s)) \right)$$

wobei die Kommunikationszeiten $t_K(op_j(s))$ und die Rechenzeiten $t_R(op_j(s))$, welche zusammen $t(op_j(s))$ bilden, für jede einzelne Operation auf jedem Prozessor explizit gemessen werden; diese Messung wird von der Verbindungsoperations-Virtualisierungsprozedur durchgeführt. Dabei wird über die einzelnen Aufrufe für jede Verbindungsschnittstelle auf jedem Prozessor summiert und für jede Entscheidung über die für eine Schnittstelle zusammengehörenden Prozessoren gemittelt. Die Meßwerte werden im Schnittstellendeskriptor gespeichert. Für den Zusammenhang mit der Datenlokalität der Verbindungsobjekte, siehe Abschnitt 10.4.2. Die Gesamtlast $L(G)$ der Knotengruppe G ist

$$L(G) = \sum_{G_i \in G} L(G_i)$$

Geradlinig ergibt sich damit die Menge von Prozessorresourcen (Knotenblockgröße KB) für einen Knoten G_i bei p verfügbaren Prozessoren zu

$$KB_i = p \, \frac{L(G_i)}{L(G)}$$

was im allgemeinen jedem Knotenblock eine gebrochene „Anzahl von Prozessoren" zuordnet und die Frage nach seiner konkreten technischen Realisierung aufwirft. Dazu gibt es mehrere Möglichkeiten:

1. Betrachte obige Zahl KB_i als Ergebnis der Knotenblockgrößenberechnung und verteile die Knotenblöcke der Reihe nach auf die Prozessoren. Man beachte, daß KB_i keine ganze Zahl ist und insbesondere kleiner als 1 sein kann.
2. Runde die KB_i durch geschickte Wahl eines Schwellwerts so auf ganze Zahlen auf und ab, daß deren Summe möglichst genau die Anzahl verfügbarer Prozessoren trifft. Verteile die so bestimmten Knotenblöcke auf die triviale Weise der Reihe nach auf die physikalischen Prozessoren.
3. Skaliere die Anzahlen und runde sie durch geschickte Wahl eines Schwellwerts so auf ganze Zahlen auf und ab, daß deren Summe möglichst genau eine Anzahl verfügbarer virtueller Prozessoren trifft und auch noch weitere wünschenswerte Eigenschaften hat (siehe unten). Verteile die Knotenblöcke so auf die Prozessoren, daß diese Eigenschaften genutzt werden können.

Der Nachteil von Möglichkeit 1 ist die unnötige Fragmentierung der Knotenblöcke; Blöcke werden häufig auf zwei Prozessoren verteilt, obwohl sie ihrer Größe nach auf einem Prozessor Platz hätten. Außerdem ist die Adreßberechnung mit gebrochenen Zahlen unschön und bei Verwendung von Gleitkommazahlen wegen möglicher Rundungsfehler zudem unsicher.

Möglichkeit 2 funktioniert offensichtlich überhaupt nicht, wenn die Zahl der Knoten größer ist als die Zahl der Prozessoren, was zumindest auf MIMD-Rechnern die Regel ist. In diesem Fall muß also anstatt mit physikalischen Prozessoren mit virtuellen Prozessoren gearbeitet werden, wodurch es wiederum wie bei Möglichkeit 1 passieren kann, daß ein Knotenblock, der aus mehreren virtuellen Prozessoren besteht, auf zwei physikalische Prozessoren verteilt werden kann, obwohl er auf einem einzelnen Platz hätte.

Für Möglichkeit 3 definieren wir deshalb folgendes als *wünschenswerte Eigenschaften*: Knotenblöcke, die weniger als einen physikalischen Prozessor groß sind, sollen auch stets auf nur einem physikalischen Prozessor plaziert werden. Ferner möchten wir soweit vorhanden gerne *Nachbarschaften* des Kommunikationsnetzes der Zielmaschine ausnutzen, um die Kommunikation der Prozessoren innerhalb des Knotenblocks zu optimieren. Drittens soll schließlich auch die Adreßberechnung für die Knotenblöcke möglichst einfach sein. Alle drei Eigenschaften können, wie unten diskutiert wird, erreicht werden, wenn wir als Knotenblockgrößen stets eine Anzahl von virtuellen Prozessoren verwenden, die eine Zweierpotenz ist.

Der Nachteil der Restriktion von Blockgrößen auf Zweierpotenzen besteht darin, daß die Lastbalance damit nicht mehr auf der Granularität hergestellt werden kann, die von der „Größe" der virtuellen Prozessoren definiert wird, sondern nur noch bis auf eine verbleibende Unbalanciertheit von (höchstens) Faktor 2. Der durch diese Unbalanciertheit entstehende Verlust von Rechenleistung wird aber durch Einsparungen ausgeglichen, die sich aus den Vereinfachungen aufgrund der „wünschenswerten Eigenschaften" ergeben. Diese Einsparungen fallen einmalig an bei der Berechnung der Knotenblockverteilung und Herstellung der Datenverteilung, sowie wiederholt bei jeder Reduktions- und Rundrufoperation auf einem Knotenblock. Ob die Einsparungen den Verlust zu mehr oder zu weniger als 100% ausgleichen, hängt von der konkreten Zielmaschine und vom Verhalten des jeweiligen Lernverfahrens ab (Stärke der restlichen Unbalanciertheit, Anzahl von Reduktions- und Rundrufoperationen pro Neuberechnung der Datenverteilung). Wir nehmen im folgenden an, daß die Beschränkung auf Zweierpotenzblockgrößen insgesamt eine Verbesserung oder zumindest nur eine geringfügige Verschlechterung bewirkt und bevorzugen sie aufgrund ihrer einfacheren Umsetzung (siehe unten).

Eine geeignete Anzahl von virtuellen Prozessoren ergibt sich aus folgender Überlegung: Eine vollständige feinkörnige Lastbalancierung könnte im Prinzip erreicht werden, wenn für jede Verbindung ein virtueller Prozessor zur Verfügung stünde. Diese Zahl ist jedoch in Wirklichkeit zu hoch, weil es mangels Parallelität keinen Sinn hat, die Verbindungen verschiedener Schnittstellen desselben Knotens auf unterschiedliche Prozessoren zu verteilen. Eine Oberschranke für die sinnvolle Anzahl virtueller Prozessoren für die Knotengruppe G ist demnach die Höchstzahl von Verbindungen an *einer* Schnittstelle, also

$$P_v = \max_s \sum_{G_i \in G} |s(G_i)|$$

Da für eine möglichst einfache Berechnung der Knotenblockverteilung auch der Virtualisierungsgrad eine Zweierpotenz sein sollte, wird man in der Praxis höchstens

$$P'_v = 2^{\lfloor \log_2(P_v) \rfloor}$$

benutzen. Kleinere Werte können sinnvoll sein, falls die benötigte Lastbalancierung keine so feine Unterteilung braucht, weil in diesem Fall die feine Unterteilung unnötigen Aufwand für zusätzliche Virtualisierungsschleifendurchläufe verursacht. Eine untere Schranke für die Zahl virtueller Prozessoren bildet die Zahl von Knoten der Gruppe.

Die Adreßberechnung wird durch die Wahl der Zweierpotenzen vereinfacht, indem viele Adreßberechnungen mit Bit-Schiebe- und Bit-Maskieroperationen anstatt Divisionen und Multiplikationen durchgeführt werden können und außerdem jeder Prozessor aus der Größe des Knotenblocks und seiner eigenen absoluten Nummer seine relative Lage im Knotenblock berechnen kann, was die Ausführung von Reduktionen vereinfacht.

Ist die Zahl p von virtuellen Prozessoren bekannt und ebenso die Last $L(G_i)$ an jedem Knoten und die Gesamtlast $L(G)$ der Gruppe, dann funktioniert die Berechnung der Knotenblockgrößen b_i folgendermaßen. Das entscheidende Problem ist die Wahl einer geeigneten Schwelle für den Übergang von Aufrunden zu Abrunden der Blockgröße, so daß einerseits p Prozessoren ausreichen, andererseits nicht viele davon leer bleiben. Die Suche dieser Schwelle erfolgt iterativ.

Berechne Knotenblockgrößen:
```
FORALL i ∈ 0...|G| − 1 DO
    soll_i := max(p L(G_i)/L(G), 1);
    auf_i := 2^⌈log2(soll_i)⌉;
    ab_i := 2^⌊log2(soll_i)⌋;
    wähle mittels binärer Suche eine möglichst große Schwelle q so, daß für
        b_i := ab_i  falls  auf_i/soll_i ≥ q
        b_i := auf_i  sonst
    gilt: Σ_i b_i ≤ p;
END;
```

Das Verfahren weist stets Knotenblockgrößen zu, die sich um weniger als Faktor 2 vom gewünschten Wert KB_i unterscheiden, sofern die Zahl virtueller Prozessoren nicht zu klein gewählt wird. Es benötigt maximal $\lceil \log_2(\max_{i\in 0\ldots|G|-1}(soll_i))\rceil$ parallele Arbeitsschritte. Die Anwendung dieses Verfahrens auf das Beispielnetz aus Abbildung 10.2 auf Seite 166 könnte je nach relativer Last pro Eingangsverbindung und Ausgangsverbindung zum Beispiel die in Abbildung 10.3 auf Seite 167 dargestellten Blockgrößen für die Knoten der Gruppe v ergeben.

Der tatsächliche Algorithmus ist noch etwas komplizierter als oben dargestellt: Die Komplikation ergibt sich aus der Tatsache, daß eine Knotenblockgröße nicht auf 0 abgerundet werden darf, so daß eventuell für manche Knoten sogar eine noch kleinere Knotenblockgröße gewählt werden muß, als durch Abrunden der ersten Schätzung $soll_i$ auf die nächstkleinere Zweierpotenz angegeben wird, um für die nicht abrundbaren kleinen Knotenblöcke genug Platz zu schaffen. Außerdem kann man die durch obigen Algorithmus gefundene Lösung oftmals noch etwas verbessern, indem man einige der von der Abrundung betroffenen Knotenblöcke wieder aufrundet, weil auch bei beliebig feiner Anpassung des q eventuell mehrere Blöcke zugleich abgerundet werden (weil sie dieselbe Last hatten), obwohl das gar nicht nötig wäre, weil dann Prozessoren frei bleiben. Im implementierten Übersetzer sind beide Verbesserungen realisiert [16].

10.3.6 Berechnung der Knotenblockverteilung

Gegeben die unregelmäßige Folge von Knotenblockgrößen stellt sich nun die Frage, wie diese so über die physikalischen Prozessoren verteilt werden können, daß möglichst wenig Verschnitt entsteht, d.h. tatsächlich jeder Prozessor ungefähr gleich viel Last erhält.

Dieses Problem ist verwandt mit dem Behälterpacken (*bin packing*), über das eine reiche Literatur existiert; siehe z.B. [15, 183, 202] und Verweise dort. Das Problem tritt in vielen Zusammenhängen auf, insbesondere eben auch bei der Lastverteilung auf Parallelrechnern, siehe z.B. [182, 369].

Auf der MasPar hat durch die Einschränkung auf Zweierpotenzen für die Blockgrößen das Packungsproblem angenehme Eigenschaften. Da die Blöcke auf Nachbarschaften abgebildet werden sollen, machen wir sie stets entweder quadratisch oder doppelt so breit wie hoch. Dies hat für das Packungsproblem zwei Vorteile: Erstens gibt es einen schnellen Algorithmus, der immer ein optimales Packungsergebnis garantiert, und zweitens hat das optimale Packungsergebnis niemals einen Verschnitt, sondern stets eine volle Ausnutzung der Prozessoren (wenn überhaupt entsprechend viele Prozessoren benutzt werden). Der Algorithmus basiert auf der Beobachtung, daß ein 2-D Feld, dessen Kantenlängen Zweierpotenzen sind, mit rechteckigen Blöcken, deren Kantenlängen ebenfalls Zweierpotenzen sind, optimal ausgefüllt werden kann, indem man die Blöcke ihrer Größe nach geordnet von links nach rechts und dann von oben nach unten in das Feld setzt.

Gegeben eine Folge von n Blöcken der Größen $2^{g_{min}}$ bis $2^{g_{max}}$, die auf ein 2-D Feld der Größe 2^k zu verteilen sind, macht das Verfahren in seiner sequentiellen Realisierung also einen Durchlauf durch die Folge für jede Blockgröße 2^g. Dabei werden alle Blöcke dieser Größe mit ihrer linken oberen Ecke auf den jeweils aktuell obersten linkesten noch freien Prozessor des Prozessorfelds plaziert. Jeder Durchlauf benötigt n Schritte, um jeden Block zu prüfen (oder weniger, wenn die bereits plazierten Blöcke aus der Folge entfernt werden). Der oberste linke freie Prozessor kann mit Hilfe eines Stapels als Hilfsdatenstruktur in konstanter Zeit bestimmt werden. Es gibt $g_{max} - g_{min}$ solche Durchläufe, im schlimmsten Fall also k Stück. Somit hat eine sequentielle Fassung des Verfahrens zum Bestimmen der Blockverteilung eine Laufzeit von höchstens $k \cdot n$ Schritten. Die Beweisidee für die Korrektheit und Optimalität dieses Verfahrens ist eine Induktion über die Nummer des plazierten Blockes. Im Induktionsschritt wird jeweils die Position des als nächstes zu betrachtenden obersten linkesten freien Prozessors sowie die Größe des nächsten einzufügenden Blockes betrachtet. Es gibt zwei Fälle: Entweder der Block füllt den verbleibenden Platz nach rechts exakt oder er läßt auch noch Platz für mindestens einen weiteren Block derselben oder einer kleineren Größe, denn für $p < q$ gibt es stets ein ganzzahliges k, so daß $2^q - 2^p = k \cdot 2^p$. Dieselbe Argumentation wie für die Breite gilt für die Höhe. Der einzufügende Block kann wegen der Sortierung nach Blockgröße nie zu groß für den Restplatz sein, es sei denn die gesamte Blockmenge ist zu groß für das gesamte Feld.

Sind für Zielfeld und/oder Blöcke nicht nur Zweierpotenzgrößen gegeben, so kann fast dasselbe Verfahren eingesetzt werden; es muß dann aber anstatt des Stapels von linkesten obersten freien Prozessoren eine Liste benutzt werden, weil die Reihenfolge, in der die Einträge benutzt werden von der Größe der nachfolgenden Blöcke abhängt. Das solchermaßen entstehende Verfahren ist eine *greedy*-Heuristik, die keine optimalen Packungen mehr garantieren kann, in der Regel jedoch ebenfalls zufriedenstellende Ergebnisse liefert. Der Laufzeitaufwand ist etwas höher, weil bei jedem Plazierungsschritt (also insgesamt n mal) die Liste linkester oberster freier Prozessoren durchsucht werden muß. Die Länge dieser Liste hängt von den konkreten Blockgrößen ab und ist schwierig abzuschätzen.

Im implementierten Übersetzer für die MasPar MP-1/MP-2 werden jedoch aus den schon diskutierten Gründen zweierpotenzgroße Blöcke und Prozessorfelder verwendet. Ich benutze einen parallelen Algorithmus, der nach demselben Prinzip funktioniert, wie der oben beschriebene sequentielle, jedoch eine andere Kostenrechnung zugrundelegt: Da nach dem Feststellen der Lage eines Blocks ohnehin in jeden seiner Prozessoren die Knotendaten kopiert werden müssen, ist die insgesamt zu leistende Arbeit mindestens 2^k Operationen. Wir erhalten deshalb für das Verteilen und Füllen der Blöcke einen parallelen Algorithmus mit einer Effizienz von mindestens $1/k$, indem wir in jedem der höchstens k parallelen Schritte jeden der 2^k Prozessoren mithelfen lassen, gemäß folgender Anweisungen.

Berechne Knotenblockverteilung:

```
Initialisiere alle Prozessoren p_i als frei;
FOR Blockgröße b := g_max DOWNTO g_min DO
   Schalte alle Prozessoren ab, die nicht frei sind;
   Schalte alle Prozessoren ab, die nicht linke obere Ecke eines Blocks der Größe b sind;
   Enumeriere die aktiven Prozessoren als L_i;
   Schalte alle Prozessoren an;
   Enumeriere die Knoten der Größe b als K_i;
   FORALL L_i DO
      hole die Daten deines Knotens K_i
      und verteile diese Daten nach rechts und unten über deinen Block;
   END;
END;
```

Da die Anzahl verschiedener Blockgrößen ($g_{max} - g_{min}$) aufgrund des exponentiellen Wachstums der Blockgrößen stets recht klein ist, ist dieser Algorithmus in praktischen Fällen sehr schnell. Der Schritt des Kopierens und Verteilens von Knotendaten kann noch beschleunigt werden, wenn schon in der bisherigen Darstellung des Knotens die Daten auf mehreren Prozessoren vorhanden waren, indem die Daten von allen diesen parallel geholt und dann nur ggf. über verbleibende *zusätzliche* Prozessoren des neuen Blocks verteilt werden.

Ein Beispiel für die aus diesem Verfahren resultierende Blockverteilung ist in Abbildung 10.4 zu sehen. Blöcke gleicher Größe werden entsprechend ihrer Reihenfolge in der vorgegebenen Blockfolge

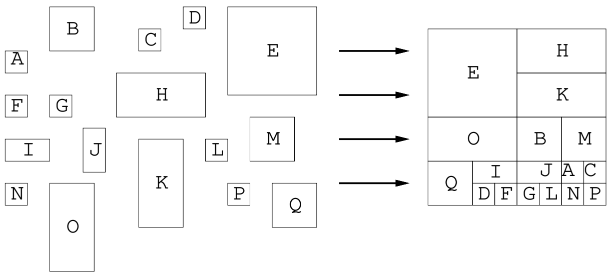


Abbildung 10.4: Beispiel für die Blockverteilung einer Folge von 17 Blöcken in 5 verschiedenen Größen auf ein 8x8 Prozessorfeld.

angeordnet (von links nach rechts, von oben nach unten), Blöcke verschiedener Größe in der Ordnung „groß vor klein".

10.3.7 Herstellung der Datenverteilung

Mit obigen Schritten sind die Parameter der neuen Datenverteilung zwar berechnet, zu ihrer eigentlichen Herstellung ist aber noch folgendes zu tun:

1. Alle Knoten- und Verbindungsobjekte werden an die richtigen Stellen kopiert,
2. die Zeiger der Verbindungen auf ihre neuen Gegenenden werden bestimmt,
3. der Speicher der Objekte aus der alten Datenverteilung wird freigegeben, und
4. ggf. werden Netzreplikate hergestellt, indem die neue Datenverteilung aus dem Segment 0 in die übrigen Segmente repliziert wird.

Die beiden letzteren Schritte waren bereits im Pseudocode des Abschnitts 10.3.4 angegeben, die anderen beiden folgen hier. Zunächst das Kopieren der Objekte:

```
Stelle neue Datenverteilung her:
  Kopiere Knoten von G aus alten in neue Knotenblöcke;
  Repliziere Knoten von G innerhalb jedes neuen Knotenblocks;
  FORALL Schnittstellen s(G) DO
    Kopiere und verteile Verbindungen von s(G) aus alten in neue Knotenblöcke;
    Modifiziere Gegenenden-Zeiger der alten Verbindungen;
  END;
```

Die Verbindungen eines Knotens werden gleichmäßig über den jeweiligen Knotenblock verteilt und zwar getrennt für jede Schnittstelle. Schließlich die Wiederherstellung der korrekten Semantik der Gegenenden-Zeiger aller Verbindungen:

```
Mache neue Datenverteilung betriebsbereit:
  FORALL Knotengruppen G DO
    FORALL Schnittstellen s(G) DO
      Korrigiere Gegenenden-Zeiger der Verbindungen;
    END;
  END;
```

Die Korrektur der Zeiger der Verbindungen auf ihre Gegenenden ist deshalb nötig, weil nach dem Umkopieren der Verbindungsobjekte diese Zeiger natürlich nach wie vor auf die Gegenenden der alten Verbindungsobjekte zeigen. Diese Situation ist auf der linken Seite von Abbildung 10.5 dargestellt. Um dies korrigieren zu können, müssen wir in den alten Verbindungsobjekten a den Gegenenden-Zeiger $E(a)$ so modifizieren, daß er jetzt anstelle des alten Gegenendes $E(a)$ der Verbindung angibt, wohin die Verbindung a in der neuen Datenverteilung „umgezogen" ist, also die Adresse $addr(a')$

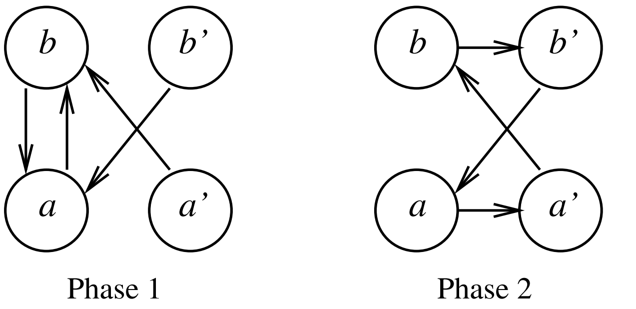


Abbildung 10.5: Gegenenden-Zeiger zwischen einem alten Paar (a, b) von lokalem und entfernten Verbindungsobjekt und einem dazu korrespondierenden neuen Paar (a', b') unmittelbar nach dem Kopieren der Verbindungsobjekte (Phase 1) und nach der Modifikation der alten Zeiger zu „Wohin ich umgezogen bin"–Zeigern (Phase 2).

der neuen Verbindung a'. Wir betrachten also Paare (a, a') von je einer alten Verbindung a und der korrespondierenden neuen Verbindung a'.

Modifiziere Gegenenden-Zeiger der alten Verbindungen:
  FORALL (a, a') DO
    $E(a) := addr(a')$; *(* Phase 2 *)*
  END;

Die Situation nach dieser Modifikation ist auf der rechten Seite von Abbildung 10.5 dargestellt. Auf dieser Basis können jetzt die korrekten neuen Gegenenden-Zeiger $E(a')$ bestimmt werden: Ein a' hat einen Zeiger auf ein altes Gegenende, das wiederum einen Zeiger auf seine neue Entsprechung hat. Es muß also der Gegenenden-Zeiger eine Indirektionsstufe weit verfolgt werden, um den neuen Gegenenden-Zeiger zu bestimmen. Notieren wir die Dereferenzierung eines Zeigers mit einem Stern, so ergibt sich:

Korrigiere Gegenenden-Zeiger der Verbindungen:
  FORALL a' DO
    $E(a') := E(*E(a'))$; *(* Phase 3 bzw. 4 *)*
  END;

Der Zustand nach Ausführung dieser Operation für ein Ende einer Verbindung und nach Ausführung für beide Enden ist in Abbildung 10.6 dargestellt. Nach Abschluß dieser Operation ist die neue Da-

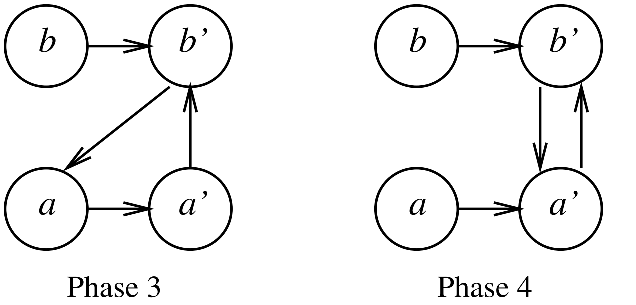


Abbildung 10.6: Gegenenden-Zeiger zwischen einem alten Paar (a, b) von lokalem und entfernten Verbindungsobjekt und einem dazu korrespondierenden neuen Paar (a', b') nach Korrektur für a' (Phase 3) und nach Korrektur für a und b (Phase 4).

tenverteilung funktionstüchtig und der Speicher der alten kann freigegeben werden. Die entstehende Verteilung von Knoten- und Verbindungsobjekten für die Gruppe v des Beispielnetzes aus Abbildung 10.2 von Seite 166 ist in Abbildung 10.7 zu sehen.

Da die verschiedenen Knotengruppen unabhängig voneinander nach demselben Verfahren auf die Prozessoren verteilt werden, existieren auf jedem Prozessor Knoten und Verbindungen aus verschiedenen Knotengruppen. Abbildung 10.8 stellt die resultierende Datenverteilung der Knoten- und Verbindungsobjekte für die Gruppen v und a des Beispielnetzes dar, wobei die Eingangsverbindungen von v nicht gezeigt sind. Es gilt die Legende von Abbildung 10.7. Die Pfeile geben die durch die Gegenende-Zeiger realisierten Verweise an und korrespondieren zu den Verbindungen.

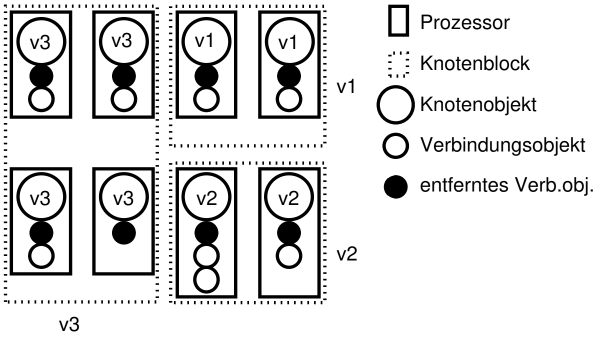


Abbildung 10.7: Verteilung der Knotenobjekte, lokalen Verbindungsobjekte und entfernten Verbindungsobjekte für die verborgene Schicht des Beispielnetzes.

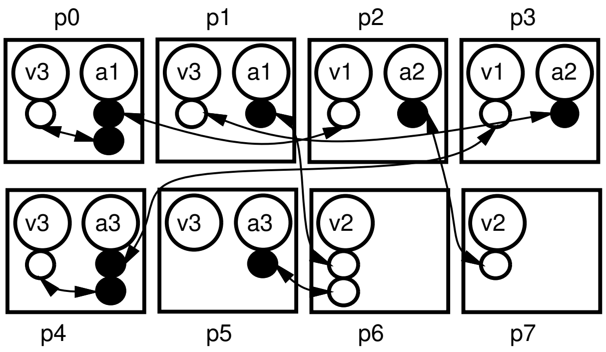


Abbildung 10.8: Datenverteilung der Knotenobjekte, lokalen Verbindungsobjekte und entfernten Verbindungsobjekte für die Knoten der Gruppen v und a des Beispielnetzes und die Verbindungen zwischen diesen Gruppen.

Es kann zufällig passieren, daß ein entferntes Verbindungsobjekt auf demselben Prozessor liegt, wie sein zugehöriges lokales Verbindungsobjekt; im Beispiel der Abbildung ist das für die Verbindung von v3 nach a3 der Fall. Die Häufigkeit solcher zufällig lokalen Verbindungen ist jedoch nur für kleine Segmentgrößen nennenswert; bereits bei Segmenten aus 16 Prozessoren ist sie vernachlässigbar.

10.4 Weitere Optimierungen

10.4.1 Kommunikationsbündelung

Für die Zugriffe auf ein entferntes Verbindungsobjekt c gibt es grundsätzlich verschiedene Realisierungsmöglichkeiten:

M1: Realisierung jedes einzelnen Lesezugriffs auf eine Komponente von c durch eine Hol-Operation (*fetch*) und jedes einzelnen Schreibzugriffs durch eine Sende-Operation (*send*).

M2: Dito, jedoch wird nur die erste Hol-Operation und die letzte Sende-Operation tatsächlich ausgeführt; die übrigen werden implizit durch lokale Zugriffe ersetzt (Cache-Mechanismus).

M3: Holen jeder möglicherweise gelesenen Komponente von c vor Beginn der Verbindungsoperation und Senden jeder tatsächlich geschriebenen (d.h. veränderten) Komponente nach ihrem Ende.

M4: Holen jeder möglicherweise gelesenen oder geschriebenen Komponente von c vor Beginn der Verbindungsoperation und Senden jeder möglicherweise geschriebenen nach ihrem Ende.

M5: Holen des kompletten Verbindungsobjekts vor Beginn der Verbindungsoperation und Senden des kompletten Verbindungsobjekts nach ihrem Ende.

Offensichtlich ist M1 in der Regel ungeschickt; es werden zu viele Kommunikationsoperationen ausgeführt. M2 kann dann günstig sein, wenn der Cache-Mechanismus von der Hardware unterstützt wird; zumindest für das Schreiben ist dies schwierig. Eine softwaremäßige Realisierung ist aber bei weitem zu laufzeitaufwendig.

M3 und M4 verlangen nach einer Bestimmung der von der Operation betroffenen Komponenten, die zur Übersetzungszeit oder zur Laufzeit erfolgen kann und auf die ich unten noch eingehe. Bei M3 ist es nötig, eine Kopie des Zustands des virtuell lokalen Verbindungsobjekts vor Beginn der Verbindungsoperation oder einen Satz von Änderungsmarkern zu speichern, damit nach Ende der Operation bestimmt werden kann, welche Komponenten tatsächlich verändert wurden. Dies ist nötig, weil das Senden einer nicht veränderten Komponente, die nicht in der Menge der gelesenen Komponenten war, das Verbindungsobjekt korrumpieren würde. Der Aufwand für diese Prüfung ist der geringen Größe der Verbindungsobjekte unangemessen.

Es bleiben damit als sinnvolle Lösungen M4 und M5. Lösung M5 besticht durch ihre Einfachheit und vermeidet den Wasserkopf mehrerer getrennter Lese- und Schreiboperationen für einzelne Komponenten. Auf der anderen Seite kann M4 eventuell die Gesamtmenge bewegter Daten verringern. Anzustreben ist deshalb eine Synthese dieser beiden Möglichkeiten durch eine geschickte Bündelung der Kommunikationsoperationen, welche im folgenden beschrieben wird. Grundsätzlich ist eine Kommunikationsbündelung keine neuartige Optimierung, sondern ist seit langem bekannt und wird in Übersetzern realisiert. Neu ist im vorliegenden Kontext die Einfachheit und Eleganz der Realisierung, die durch das spezifische Wissen aus dem KNA-Programmiermodell ermöglicht wird. Dies steht im Gegensatz zur Kommunikationsbündelung für allgemeine feldbasierte Sprachen, bei denen eine aufwendige Indexanalyse vorgenommen werden muß und oftmals nur Laufzeitprüfungen die nötige Information beschaffen können.

Der Ausgangspunkt ist in jedem Fall die Bestimmung der betroffenen Komponenten. Während dieses Zugriffsanalyseproblem für allgemeine Programme sehr kompliziert zu lösen ist und eine exakte Lösung nicht berechenbar ist, kann für neuronale Algorithmen, die im KNA-Programmiermodell formuliert sind, leicht eine gute Annäherung an die exakte Lösung bestimmt werden. Folgende Gründe sind dafür ausschlaggebend: Auf die Komponenten von Verbindungen kann nur in und während des Ablaufs der Verbindungsprozedur zugegriffen werden; Verbindungsprozeduren sind kurz und haben einen einfachen Kontrollfluß; Aufrufverschachtelungen von Verbindungsprozeduren kommen selten vor; verschachtelt aufgerufene Verbindungsprozeduren sind stets statisch benennbar, weil sie nur eindeutig namentlich aufgerufen werden können (keine Vererbung oder dynamische Typisierung oder Prozedurzeiger); Zugriffe auf Komponenten erfolgen stets namentlich (nicht über Zeiger); Komponenten sind in der Regel atomar (keine Felder); Jeder Lese- und Schreibzugriff ist textuell als solcher erkennbar (keine verborgenen Seiteneffekte).

Wir können daher zur Bestimmung der Menge der von einer Verbindungsoperation *op* gelesenen (K_l) bzw. geschriebenen (K_s) Verbindungskomponenten folgende triviale textuelle Analyse verwenden:

1. Bestimme für jede Verbindungsoperation op_i die Menge $M_l(i)$ der potentiell lokal in op_i gelesenen und die Menge $M_s(i)$ der potentiell lokal in op_i geschriebenen Komponenten, sowie die Menge $M_R(i)$ der potentiell von op_i aufgerufenen Verbindungsoperationen. Eine Komponente ist potentiell lokal gelesen, wenn ihr Wert textuell innerhalb von op_i in einem Ausdruck verwendet wird. Eine Komponente ist potentiell lokal geschrieben, wenn sie textuell innerhalb von op_i durch eine Zuweisung oder einen Prozeduraufruf mit Variablenargument verändert wird. Ob die betreffenden Anweisungen auch tatsächlich ausgeführt werden, wird nicht untersucht.
2. Bestimme die transitive Hülle $H(i)$ der durch die $M_R(i)$ definierten Aufrufrelation.
3. Setze nun
$$K_l(i) := M_l(i) \cup \bigcup_{j \in H(i)} M_l(j) \quad \text{und} \quad K_s(i) := M_s(i) \cup \bigcup_{j \in H(i)} M_s(j)$$

Dieses Verfahren ist mit sehr wenig Aufwand in einem Übersetzer zu realisieren. Seine Grenzen erreicht das Verfahren bei der Verwendung von großen Feldern als Komponenten in Verbindungen, wo die einzelnen Feldelemente nicht unterschieden werden können, sowie bei einem Programmierstil, der nur wenige umfangreiche Verbindungsprozeduren benutzt, die jeweils zahlreiche verschiedene Operationen realisieren, zwischen denen durch zusätzliche „Befehlscode"-Parameter ausgewählt wird; in diesem Fall wird die Bestimmung der betroffenen Komponenten sehr unscharf, weil meistens alle Komponenten betroffen zu sein scheinen, auch wenn das gar nicht stimmt. Beide Problemfälle sind jedoch ungewöhnlich.

Es müssen nun vor Beginn der Ausführung einer Verbindungsoperation op_i mindestens alle Komponenten $K_l(i) \cup K_s(i)$ geholt werden. Mit diesen Daten wird ein künstliches lokales Verbindungsobjekt initialisiert, auf dem op_i anschließend wie auf einem echten lokalen Verbindungsobjekt arbeitet. Nach Ende der Ausführung von op_i müssen mindestens alle Komponenten $K_s(i)$ zurück an das eigentliche Verbindungsobjekt gesendet werden; es dürfen nur jene Komponenten gesendet werden, die zuvor auch geholt worden waren oder garantiert geschrieben wurden, da sie andernfalls undefiniert sind. Deshalb war $K_s(i)$ zuvor Teil der Menge der geholten Komponenten.

Es bieten sich mehrere Möglichkeiten an, diese Komponenten zu transportieren:

1. Hole und sende jede Komponente einzeln. Dies ist die naive Realisierung von M4.
2. Hole und sende jeden gemäß der Speicherdarstellung der Verbindung zusammenhängenden Block von Komponenten einzeln. Dies ist eine naheliegende Verbesserung der ersten Möglichkeit, die die Zahl von Kommunikationsoperation verringert, ohne zusätzlichen Aufwand zu verursachen. Zu ihrer Realisierung muß lediglich der Übersetzer die Speicherdarstellung des Verbindungsobjekts kennen und berücksichtigen.
3. Packe alle benötigten Komponenten in ein Kommunikationspaket zusammen und am Zielprozessor wieder auseinander. Dies verursacht zusätzlichen Aufwand zur Umwandlung der normalen in die gepackte Darstellung und zurück. Dieser Aufwand ist inakzeptabel, falls die Zielmaschine hardwaremäßige Unterstützung für normale Zugriffe auf entfernten Speicher bietet, weil diese dann nicht gut genutzt würde. Diese Situation ist der Normalfall auf modernen Parallelrechnern. In jedem Fall lohnt sich das Packen nur bei hohen Latenzzeiten und zugleich niedriger Bandbreite. Dies würde sich ändern, wenn in Zukunft die Hardwareunterstützung auf das Packen und Entpacken ausgedehnt würde.
4. Hole und sende die konvexe Hülle über alle benötigten Komponenten. Dies ist quasi die naheliegende Verbesserung von M5, die zwar ebenfalls nur eine Kommunikationsoperation ausführt, die Größe des dabei versendeten Pakets jedoch minimiert.
5. Hole und sende die Komponenten in kostenminimalen Blöcken. Dies ist die obenerwähnte Synthese von M4 und M5 und die im CuPit-Übersetzer realisierte Methode; sie wird im folgenden erläutert.

Das Zusammenfügen (Verschmelzen) mehrerer nichtzusammenhängender Blöcke von betroffenen Komponenten in einer Nachricht bedeutet das Übertragen von nicht benötigten Daten, die in der Speicherdarstellung des Verbindungsobjektes zwischen benachbarten Blöcken benötigter Daten liegen. Sei für zwei solche Blöcke die Menge dieser nicht benötigten Daten gleich d Bytes, dann sollte man die beiden Blöcke genau dann samt der Lücke von d Bytes in einer Kommunikationsoperation übertragen, wenn die mittlere Latenzzeit größer ist als die Zeit zum Übertragen von d zusätzlichen Bytes in einem Kommunikationspaket. Der Schwellwert $\hat{d}$, an dem diese Entscheidung umschlägt, muß für eine gegebene Zielmaschine experimentell bestimmt werden und liegt auf der MasPar MP-1 bei 11,5 bis 12 Bytes. Je nach Verbindungsnetzwerk kann der Wert mit der Segmentgröße (bei Netzreplikation) etwas schwanken, so daß der Übersetzer für eine optimale Entscheidung Laufzeitinformation über die typische Segmentgröße für das Programm zur Verfügung haben sollte; auf der MasPar MP-1/MP-2 ist dies nicht der Fall [16].

Wir fügen also zwei Blöcke von zusammenhängenden benötigten Verbindungskomponenten für eine Verbindungsoperation genau dann in eine Kommunikationsoperation zusammen, wenn die Lücke in der Speicherdarstellung zwischen ihnen höchstens $\hat{d}$ Bytes beträgt und übertragen andernfalls die Blöcke getrennt in zwei Kommunikationsoperationen.

Dieses Vorgehen erlaubt im Prinzip eine weitere Optimierung durch die Suche nach der günstigsten Speicherdarstellung für ein Verbindungsobjekt. Die günstigste Darstellung ist die, bei der die Gesamtkosten der Kommunikationsoperationen über alle dynamisch auftretenden Operationen auf entfernten Verbindungen des Verbindungstyps minimiert werden, indem die Gesamtgröße mitübertragener Lücken minimiert wird. Ist die Häufigkeit jeder Verbindungsoperation aufgrund eines Laufzeitprofils bekannt, so kann die Kostenfunktion für Speicherdarstellungen aufgestellt werden, mit deren Hilfe sich die optimale Darstellung durch globale Optimierungsmethoden wie Branch-and-Bound oder Simulated Annealing bestimmen läßt. Diese Optimierung der Speicherdarstellung habe ich jedoch nicht weiter untersucht.

Auf Maschinen mit geringer Kommunikationsbandbreite oder bei Programmen, für die viele Lücken mitübertragen werden müßten, kann sich stellenweise auch die obenerwähnte Packungsmethode lohnen. In diesem Fall muß der Übersetzer über ein Kostenmodell verfügen, mit dessen Hilfe er für jede Verbindungsprozedur entscheiden kann, ob das Packen oder die Lückenübertragung vorzuziehen ist. Diese Möglichkeit wurde ebenfalls nicht weiter untersucht, zumal auf der MasPar die Amortisationsschwelle für das Packverfahren so hoch liegt, daß sie in den in dieser Arbeit betrachteten Programmen von keiner einzigen Prozedur je erreicht wird.

10.4.2 Verbindungsallokation

Bislang haben wir die Entscheidung, an welchem Ende einer Verbindung das lokale Verbindungsobjekt liegen soll, immer als gegeben angenommen. Offensichtlich ist aber diese Entscheidung nicht gleichgültig: Gemäß unserer Zerlegung der Laufzeit einer Verbindungsoperation in Rechenzeit und Kommunikationszeit $t(op_i) = t_R(op_i) + t_K(op_i)$ ergeben sich unterschiedliche Summen von Kommunikationszeiten, je nach Lokalisierung der eigentlichen Verbindungsobjekte. Sei t_K^* der Wert für t_K, der anfällt, falls das Objekt nichtlokal ist (also $t_K = 0 \vee t_K = t_K^*$), wobei wir gutverteilte Last voraussetzen. Angenommen wir haben Knoten mit zwei Schnittstellen, *in* und *out*. Dann ist die Last $L(G_i)$ an einem Knoten G_i

$$\begin{aligned} L(G_i) = \quad & \left(|in(G_i)| \sum_{op_j} |op_j(in)| \ (t_R(op_j(in)) + \lambda t_K^*(op_j(in))) \right) \\ + \quad & \left(|out(G_i)| \sum_{op_j} |op_j(out)| \ (t_R(op_j(out)) + (1-\lambda) t_K^*(op_j(out))) \right) \end{aligned}$$

wobei $\lambda = 0 \Leftrightarrow lokal(in(G))$ und $\lambda = 1 \Leftrightarrow lokal(out(G))$. Die beiden Möglichkeiten schließen sich gegenseitig aus, weil es Verbindungen zwischen *in* und *out* geben kann; aus demselben Grund ist die Entscheidung zwangsläufig für alle Knoten in allen Gruppen gleich. Es ist also folgendes Optimierungsproblem zu lösen:

$$\text{Wähle } \lambda \in \{0,1\} \text{ so, daß } \sum_G \sum_i L(G_i) = minimum.$$

Entscheiden wir uns zum Beispiel dafür, die eigentlichen Verbindungsobjekte an der Schnittstelle *in* anzuordnen, also $lokal(in(G)) \wedge \lambda = 0$ dann sind alle Kommunikationszeiten der an dieser Schnittstelle ausgeführten Operationen Null, während die Kommunikationszeiten der Operationen an der Schnittstelle *out* größer als Null sind. Die insgesamt anfallenden Kommunikationszeiten hängen also von der Größe der einzelnen t_K^* ab, sowie von der Anzahl der jeweiligen Aufrufe $|op_j(out)|$, der Anzahl von betroffenen Verbindungen $|out(G_i)|$ und den Blockgrößen b_i der einzelnen Knoten in allen Knotengruppen G.

Bei einer Änderung dieser als *Verbindungsallokation* bezeichneten Entscheidung zu $lokal(out(G)) \wedge \lambda = 1$ gehen geänderte summierte Kommunikationszeiten in die Last $L(G_i)$ und damit in die Wahl der Blockgröße ein (siehe Seite 172); deshalb hängen die Wahl der Verbindungsallokation und die Wahl der Blockgrößen gegenseitig voneinander ab. Aufgrund dieser zyklischen Abhängigkeit ist es nicht ohne weiteres möglich, aufgrund von Messungen während der Laufzeit die richtige Entscheidung über die Allokation der Verbindungen zu finden. Dazu wäre nämlich ein analytisches Modell der Kommunikationszeiten notwendig, das wegen der komplexen Abhängigkeit derselben von der Topologie des neuronalen Netzes praktisch nicht aufgestellt werden kann. Da je nach gewählter Verbindungsallokation unterschiedlicher Code für den Aufruf jeder Verbindungsoperation benötigt wird, sollte die Entscheidung ohnehin besser zur Übersetzungszeit vorliegen, um Laufzeitaufwand für die Auswahl bei jedem Aufruf zu vermeiden.

Glücklicherweise ist die Entscheidung in praktischen Fällen nicht allzu schwierig: Bei den normalerweise verwendeten Netzen mit nur einem Verbindungstyp und nur zwei Schnittstellen pro Knoten ergeben sich bedeutsame Unterschiede in der Laufzeit zwischen den zwei möglichen Verbindungsallokationen meist nur dann, wenn der Programmierer, wo immer er die Wahl hat, systematisch eine Schnittstelle bevorzugt verwendet. In diesem Fall liegt aber die korrekte Entscheidung auf der Hand: Die Verbindungen sollten an dieser bevorzugten Schnittstelle allokiert werden. In anderen Fällen ist der Einfluß der Entscheidung normalerweise nur gering. Die richtige Entscheidung kann ggf. durch Ausprobieren gefunden werden, da es ja nur zwei Möglichkeiten gibt. Dieses Ausprobieren kann der Übersetzer selbsttätig vornehmen, wenn man ihm vorgibt, wie ein deterministischer, vollautomatischer Programmablauf gestartet werden kann: Der Übersetzer erzeugt beide Versionen, läßt beide ablaufen und vergleicht die Laufzeiten, um die beste Verbindungsallokation auszuwählen. Ein solcher Probelauf mit nur einem Datensatz ist ausreichend, weil bei typischen neuronalen Algorithmen unterschiedliche Daten zwar verschieden *starke* Unterschiede hervorrufen (siehe Abschnitt 11.8), aber stets dieselbe Allokation besser sein wird.

Alternativ könnte man beide Allokationen zugleich in einem Programm realisieren und zwischen ihnen zur Laufzeit umschalten, automatisch gesteuert mit einem Verfahren ähnlich dem unten beschriebenen zur Wahl der Anzahl von Netzreplikaten. Dieses Vorgehen hat jedoch zwei Nachteile: Erstens führt es in seiner effizienten Form zu einer erheblichen Aufblähung der Menge erzeugten Codes, da nun auch alle Netz- und Knotenprozeduren doppelt vorhanden sein müssen[2]; zweitens stört das Verfahren dann wiederum die Berechnung der Blockgrößen und außerdem die Wahl der Zahl von Netzreplikaten, so daß die Allokationsauswahl effektiv trotz ihrer dynamischen Natur nur einmal zu Beginn des Programmablaufs durchgeführt werden könnte. Das ganze Problem kompliziert sich erheblich, wenn Programme mehr als zwei Schnittstellen an einem Knoten verwenden; dies tritt jedoch selten auf — in den in dieser Arbeit betrachteten konkreten Lernverfahren gar nicht.

Meinem implementierten Übersetzer kann man bei Bedarf die Allokation der Verbindungen zur Übersetzungszeit statisch durch eine Option vorschreiben; eine automatische Optimierung der besten Allokation ist nicht vorgesehen.

10.4.3 Knotenverteilung

Gelegentlich verletzen Lernverfahren die Annahme 2, welche besagt, daß die Arbeit für jede Operation an jedem Knoten der Gruppe gleich groß ist, wenn man von der Arbeit auf den Verbindungen absieht. Solche Verfahren führen eine Operation nicht auf der kompletten Gruppe aus, sondern nur auf Teilen einer Gruppe. Dies bedroht die Lastbalance, wenn keine Maßnahmen zu ihrer Erhaltung getroffen werden. Es gibt zwei Hauptfälle:

1. Manche Verfahren für rekurrente Netze gehen von einer asynchronen Betriebsweise der Knoten aus und verlangen, daß jeder Knoten einzeln berechnet wird, z.B. weil nur dann sich das Nichtauftreten von Oszillationen garantieren läßt. Solche Verfahren haben ohnehin einen zu geringen Parallelitätsgrad, um für eine parallele Implementierung gut geeignet zu sein, deshalb machen wir

[2] Dieser Ansatz wird bei gleichzeitiger Optimierung der Allokation mehrerer verschiedener Verbindungstypen aufgrund der kombinatorischen Explosion unrealistisch. Die Suche durch Ausprobieren wird hingegen dann immer noch funktionieren, vorausgesetzt, daß die Entscheidungen für die einzelnen Verbindungstypen nicht allzu starke Wechselwirkungen haben. Realisiert man alternativ den dynamischen Wechsel der Allokation durch Abfragen in den Verbindungsoperations-Virtualisierungsprozeduren, so entsteht ein Effizienzverlust durch diese zusätzlichen Abfragen.

uns um sie keine Gedanken. Für parallele Implementierungen werden gelegentlich Approximationen der asynchronen Berechnungsweise verwendet, die dynamisch zufällig ausgewählte Teilmengen der Knoten neu berechnen. Für diese Fälle gibt es keine Maßnahmen, die die Lastbalancierung einer einzelnen solchen Operation garantieren könnten, ohne dabei die Datenlokalität zu verlieren, weil erst unmittelbar vor Beginn der Operation die beteiligten Prozessoren überhaupt festgelegt werden. Allerdings ist zumindest bei genügend hohen Virtualisierungsgraden eine Balancierung auch nicht nötig, weil die beteiligten Knoten zufällig — und damit ungefähr gleichmäßig — über die Prozessoren verteilt sind.

2. Manche Verfahren, die dynamisch Knoten zufügen, behandeln die jeweils neu zugefügten Knoten zumindest eine zeitlang anders, als die bereits länger existierenden; beispielsweise während des Kandidatentrainings der in Kapitel 5 untersuchten Verfahren. In diesem Fall kann eine Lastbalancierung dadurch erreicht werden, daß man die Berechnung der Knotenblockgrößen und die Knotenverteilung für alte und für neue Knoten getrennt durchführt und beide Teilgruppen getrennt über alle Prozessoren verteilt.

Die Behandlung von Fall 2 erfolgt dadurch, daß die jeweils neuen Knoten als solche markiert werden, und der Übersetzer in das Program Code einfügt, welcher prüft, ob häufig Knotenoperationen aufgerufen werden, in denen nur die neuen Knoten oder nur die alten Knoten aktiv sind. In diesem Fall wird die erwähnte getrennte Verteilung von alten und neuen Knoten verwendet, andernfalls die normale Verteilung der gesamten Gruppe. Da dem Programmierer bekannt ist, welcher der beiden Fälle vorliegt, ist eine automatische Optimierung eigentlich sogar unnötig, sondern der Übersetzer kann die Information durch eine Option beim Aufruf erhalten und direkt passenden Code erzeugen. Im implementierten Übersetzer ist eine automatische Optimierung implementiert, die durch eine Option „überstimmt“ werden kann.

10.4.4 Wahl der Replikatanzahl

In CuPit kann der Programmierer dem Übersetzer die Wahl der Anzahl zu benutzender Netzreplikate durch Angabe eines Replikatanzahl-Intervalls freistellen. Der Übersetzer sollte dann einen laufzeitminimierenden Wert wählen. Sowohl eine zu hohe als auch eine zu niedrige Replikatanzahl kann die Laufzeit verschlechtern: Bei zu wenigen Replikaten kann der vom Netz her verfügbare Parallelitätsgrad zu gering sein, um eine gute Auslastung der Maschine zu erreichen; außerdem sind die Distanzen für Kommunikationsoperationen höher. Bei zu vielen Replikaten wächst der Verwaltungsaufwand für die Herstellung und Zusammenführung der Replikate stärker als die zusätzlichen Vorteile durch hohe Parallelität und kleine Segmente; außerdem kann je nach Lernverfahren die hohe Parallelität zum Bumerang werden, indem manchmal Replikate stillgelegt werden müssen, weil kein Beispiel für sie zur Verfügung steht.

Leider kann eine analytische Behandlung dieser Abwägung nicht vom Übersetzer geleistet werden, da die richtige Replikatanzahl auf zu komplexe Weise von den Eigenschaften der Zielmaschine, der gegenwärtigen Knoten- und Verbindungsstruktur, dem Lernverfahren und der Menge an Trainingsdaten abhängt. Dennoch kann eine automatische Wahl der Replikatanzahl erreicht werden.

Grundidee der automatischen Wahl ist das Verfahren von Versuch und Irrtum, aufbauend auf der Überlegung, daß auch eine ungünstige Replikatanzahl keine großen Auswirkungen hat, wenn sie nur über einen kurzen Abschnitt des gesamten Programmlaufs beibehalten wird. Das vorgeschlagene Verfahren funktioniert so:

1. Wähle zu Beginn die höchste erlaubte Replikatanzahl.
2. Nach jeweils einem Trainingsabschnitt verkleinere die Replikatanzahl; wiederhole dies solange, wie die Verkleinerung eine Beschleunigung des Programms bewirkt. Benutze dann die optimale Anzahl (also die vorletzte probierte).
3. Für den Rest des Programmlaufs wiederhole die folgenden Schritte 4 und 5:

4. Behalte die so gefundene Replikatanzahl solange bei, bis sich entweder die Laufzeit eines Trainingsabschnitts verändert (durch Verhaltensänderung des Lernverfahrens) oder eine Topologieänderung am Netz vorgenommen wird.
5. Untersuche dann wieder schrittweise zuerst größere Replikatanzahlen (sofern erlaubt) und, falls diese keine Beschleunigung bringen, kleinere Replikatanzahlen (sofern erlaubt), bis wieder die optimale Anzahl gefunden ist.

Da die Effizienzverluste bei nur mäßig ungünstigen Replikatanzahlen gering sind, kann dieses Verfahren allenfalls am Anfang eine wesentliche Abweichung von der optimalen Effizienz (insoweit diese durch die Replikatanzahl bestimmt wird) verursachen.

Ein „Trainingsabschnitt“ ist eine Phase des Programmablaufs, die ihrer Struktur nach regelmäßig wiederkehrt, z.B. eine Epoche oder ein konstant großer Teil einer Epoche. Da der Übersetzer aus dem Programmtext nicht ersehen kann, an welchen Stellen des Programms jeweils ein sinnvoller Trainingsabschnitt zu Ende ist und in CuPit auch keine Annotation vorgesehen ist, mit der der Programmierer dem Übersetzer diese Information zukommen lassen könnte, wurde die oben beschriebene Optimierung im Benutzerprogramm implementiert; ihre Basis ist die Messung der Gesamt-CPU-Zeit für den vergangenen Trainingsabschnitt.

Aufgrund der verwendeten Segmentierung der Maschine verursacht die Verkleinerung der Replikatanzahl stets zusätzlichen Verschnitt (unbenutzte Segmente), wenn nicht die resultierende Anzahl von Replikaten eine Zweierpotenz ist. Deshalb sind die Vergrößerungs- und Verkleinerungsschritte für die Replikatanzahlen stets Verdopplungen und Halbierungen, was zugleich die Höchstanzahl von Suchschritten klein hält.

10.5 Implementation des Übersetzers

Zur Bewertung der oben beschriebenen Übersetzungstechniken wurde ein CuPit Übersetzer für die MasPar MP-1/MP-2 implementiert. Auf die Eigenschaften dieser Maschine wird bei der Auswertung im Kapitel 11 noch näher eingegangen. Der Übersetzer erzeugt aus CuPit Quellprogrammen Zielprogramme in der Sprache MPL, dem maschinenabhängigen, datenparallelen C der MasPar-Rechner.

Die Implementation des Übersetzers wurde mit Hilfe des Übersetzerbausystems Eli [137] erstellt. Dabei wurden folgende Werkzeuge benutzt:

1. Sprache `gla` und Lesergenerator *GLA* zur Beschreibung und Erzeugung der lexikalischen Analyse,
2. Sprache `con` und Zerteilergenerator *PGS* zur Beschreibung der Syntax und Erzeugung des Zerteilers,
3. Sprache `sym` und Grammatiktransformator *CAGT* zur Beschreibung und Durchführung der Umwandlung von konkreter in abstrakte Syntax,
4. Sprache `LIDO` zur Beschreibung der Attributierung,
5. Sprache `oil` und Operatoridentifikator-Generator *OIL* zur Beschreibung und Durchführung der Operatorunterscheidung,
6. Attributevaluator-Generator *LIGA* zur Erzeugung des Attributberechnungsprogramms,
7. Sprache `ptg` und Codebaumerzeuger-Generator *PTG* zur Beschreibung und Realisierung kleiner Codeerzeugungsschablonen,
8. Präprozessor *cpp* zur Beschreibung und Realisierung großer Codeerzeugungsschablonen,
9. Sprache `C` und Übersetzer *gcc* zur Realisierung komplexer Zusatzfunktionen, die nicht unmittelbar von Eli abgedeckt werden,
10. Sprache `MPL` und Übersetzer *ampl* zur Beschreibung und Fertigübersetzung des Codes,
11. Präprozessor *C-Refine* zur Erweiterung des Sprachumfangs von C und MPL um Refinements[3],

[3] ftp://ftp.ira.uka.de/pub/gnu/crefine.3.0.tar.Z

12. Sprache `fw` und literate-programming-Werkzeug *FunnelWeb* zur Integration des Übersetzers und seiner Dokumentation, sowie LaTeX zur Formatierung der Dokumentation,
13. Expertensystem *Odin* und Hilfsprogramm *make* zur Integration aller dieser Werkzeuge,
14. Revisionsverwaltungssystem *RCS* zur Versionshaltung aller Quelltexte.

Im Gegensatz zu den meisten Forschungsprototypen ist der Übersetzer als strukturiert dokumentierter Quelltext veröffentlicht. Er ist ein im Stile des literarischen Programmierens (*literate programming*) geschriebenes Dokument, das alle Teile des Übersetzers und des Laufzeitsystems enthält und dabei jeweils logisch zusammengehörende Teile auch zusammen präsentiert und beschreibt, selbst wenn diese Teile technisch in verschiedene Dateien gehören; z.B. stehen bei Attributierungsregeln (Sprache LIDO) jeweils unmittelbar die Definitionen der darin benutzten Codeerzeugungsschablonen (Sprache `ptg`). Diese Präsentationsform wird durch das Werkzeug *FunnelWeb* möglich, das solche zerstückelten Teile zu Dateien zusammensetzt, wie sie von den jeweiligen Werkzeugen der späteren Verarbeitungsstufen erwartet werden. Die Gesamtübersicht im Dokument wird durch ein Inhaltsverzeichnis und einen Schlagwortindex hergestellt. Das Dokument ist als technischer Report veröffentlicht [16] und hat einen Umfang von 350 Seiten.

Der Übersetzer ist voll funktionstüchtig und stabil. Er realisiert den Sprachumfang von CuPit bis auf zwei Konstrukte, die aber beide im Rahmen dieser Arbeit keine Rolle spielen: Erstens direkte Verbindungsadressierung zum individuellen Erzeugen und Initialisieren einzelner Verbindungen und zweitens die Selbstvervielfachung eines Knotens samt seiner Verbindungen, deren sinnvoller Einsatz von CuPit aber ohnehin nicht ausreichend unterstützt wird, weil in der Sprache keine genügend einfache Möglichkeit vorgesehen ist, die Kopien voneinander zu unterscheiden. Darüber hinaus hat der Übersetzer nur geringe Einschränkungen und Mängel, welche sämtlich genau dokumentiert sind (siehe Anhang B von [16]).

Zusätzlich gibt es einen zweiten Übersetzer, der Code für sequentielle Zielmaschinen erzeugt. Auch dieser Übersetzer funktioniert stabil; ich habe ihn für einen großen Teil der Experimente eingesetzt, die im ersten Teil der Arbeit berichtet wurden, um nicht allein auf unsere stark frequentierte MasPar angewiesen zu sein. Der sequentielle Übersetzer wurde für 32-bit-Rechner vom Typ Sun-4 und für 64-bit-Rechner der Typen KSR-1 und KSR-2 als Zielmaschinen erprobt und eingesetzt; er erzeugt reinen C-Code, der jeweils mit irgendeinem ANSI-C Übersetzer auf der Zielmaschine fertigübersetzt wird. Der sequentielle Übersetzer wurde aus dem MasPar-Übersetzer hergeleitet und unterscheidet sich von diesem ausschließlich in der Codeerzeugung.

Kapitel 11

Auswertung

This must be wrong by a factor
that ought not to be too different from unity.
Ein unbekannter Dozent

It is impossible to travel faster than light,
and certainly not desirable,
as one's hat keeps blowing off.
Woody Allen

11.1 Übersicht

In diesem Kapitel werden die im vorigen Kapitel vorgestellten Optimierungen des CuPit-Übersetzers bewertet. Dazu wird eine Reihe von Experimenten durchgeführt, bei denen zumeist jeweils eine der Optimierungen abgeschaltet wird, um einen Vergleich der Programmlaufzeiten mit und ohne die Optimierung zu ermöglichen.

Abschnitt 11.2 enthält eine Fehlerbetrachtung, in der abgeschätzt wird, wie sehr systematische und statistische Fehler die später präsentierten Ergebnisse beeinflußt haben. Danach folgen der Reihe nach folgende Auswertungen:

1. Messung und Bewertung der absoluten Gesamtleistung des erzeugten Codes und Vergleich mit anderen Implementationen (Abschnitt 11.3).
2. Messung und Bewertung der Verbesserungen aufgrund der Lastbalancierung (Abschnitt 11.4).
3. Messung der Verbesserungen aufgrund der Datenlokalität (Abschnitt 11.5.1).
4. Prüfung der Annahme, daß weitere Optimierung der Datenlokalität nicht lohnend ist (Abschnitt 11.5.2).
5. Messung und Bewertung der Laufzeit zur Berechnung und Herstellung der Datenverteilung, sowie Diskussion des Speicherbedarfs für die Verwaltung der Datenverteilung (Abschnitt 11.6).
6. Messung und Bewertung der Verbesserungen durch optimale Wahl der Verbindungsallokation (Abschnitt 11.8).
7. Prüfung der Skalierbarkeit des erzeugten Codes über verschiedene Anzahlen von Prozessoren (Abschnitt 11.9).
8. Messung und Bewertung der Verbesserungen durch die automatische Wahl der besten Anzahl von Netzreplikaten (Abschnitt 11.10).

11.2 Fehlerbetrachtung

11.2.1 Systematischer Fehler

Ein systematischer Fehler entsteht in den unten beschriebenen Auswertungen auf dreierlei Weise, nämlich durch

1. nicht allgemeingültige Auswahl der Lernverfahren,
2. nicht allgemeingültigen Stil der Implementierung und
3. nicht allgemeingültige Auswahl von Datensätzen.

Die „richtige" Auswahl der Lernverfahren ist ein unlösbares Problem, weil der CuPit-Übersetzer ja vor allem bei der Entwicklung neuer Verfahren benutzt werden soll, da die Erforschung konstruktiver Lernverfahren für neuronale Netze noch in ihren Anfängen steckt. Die relevanten Verfahren gibt es also überwiegend noch gar nicht. Ich betrachte deshalb in den Auswertungen jeweils nur exemplarisch ein Verfahren bei jeder Optimierung. Für die endgültige Beurteilung der Ergebnisse ist also Augenmaß vonnöten, um abzuschätzen, wie sie sich auf andere Lernverfahren übertragen. Der vermutliche Einfluß des gewählten Lernverfahrens auf die Ergebnisse wird bei jeder Optimierung nochmals kurz diskutiert. Soweit im Text nichts anderes angegeben ist, liegt den Messungen als Programmbeispiel jeweils das autoprune-Verfahren aus Kapitel 6 zugrunde, wobei für die einzelnen Datensätze die in Abschnitt 4.4.1 angegebene Pivot-Architektur für das neuronale Netz mit Problemvariante 1 verwendet wird.

Der Stil der Implementierung eines Verfahrens ist insofern von Bedeutung, daß eine ungeschickte Programmierweise die Programme sehr ineffizient machen kann und damit die relative Wirksamkeit der Optimierungen teilweise verringert und teilweise erhöht. Bei den Messungen wurden Implementierungen zugrundegelegt, die nicht feinoptimiert sind, aber den folgenden Programmierstil einhalten, der ineffiziente Programme vermeidet:

1. Verbindungsprozeduren werden klein gehalten, d.h. es werden nur solche Operationen in einer Prozedur vereint, die auch im gleichen Aufruf benötigt werden. Große Verbindungsprozeduren verwirren die Kommunikationsbündelung.
2. Verbindungskomponenten werden ggf. für mehrere Zwecke verwendet, um die Verbindungsobjekte klein zu halten und Kommunikationsaufwand einzusparen.
3. Wo eine Verbindungsoperation sowohl von der Eingangsseite eines Knotens als auch von der Ausgangsseite des Gegenknotens aufgerufen werden kann, wird stets die Eingangsseite gewählt. Dadurch bekommt die Lastbalancierung und die Optimierung der Verbindungsallokation ein größeres Potential.

Dieser Programmierstil ist leicht einzuhalten und wird deshalb als repräsentativ angenommen. Der eventuelle Einfluß des Implementierungsstils auf die Meßergebnisse wird dennoch, sofern relevant, bei jeder Optimierung nochmals kurz diskutiert.

Eine Auswahl von Datensätzen, die man als repräsentativ bezeichnen kann, ist nur möglich, wenn man sich auf ein Anwendungsfeld einschränkt. In diesem Auswertungskapitel verwende ich deshalb wiederum die Datensätze aus der PROBEN1 Benchmarksammlung, deren Herkunftsbereich in Abschnitt 3.2.1 charakterisiert wurde. Für diesen Bereich erhebt die hier getroffene Auswahl einen gewissen Anspruch darauf, ein allgemeingültiges Bild zu zeichnen, weil eine beträchtliche Anzahl von Problemen verwendet wird, deren Eigenschaften sich erheblich unterscheiden. Eine Quantifizierung dieser Allgemeingültigkeit (oder ihres Fehlens) ist aber naturgemäß nicht möglich.

Insgesamt kann also der systematische Fehler nicht quantitativ abgeschätzt werden. Jedoch ist von seinen Komponenten nur die Auswahl der Lernverfahren kritisch; die übrigen Komponenten werden beherrscht (Datensätze) oder dürfen vernachlässigt werden (Programmierstil).

11.2.2 Statistischer Fehler

Ein statistischer Fehler entsteht in den unten beschriebenen Auswertungen auf dreierlei Weise, nämlich durch

1. Meßfehler bei der Bestimmung der tatsächlich vorliegenden Laufzeiten,
2. Schwankungen in den tatsächlichen Laufzeiten aufgrund der Maschineneigenschaften und
3. Schwankungen in den tatsächlichen Laufzeiten aufgrund nichtdeterministischer Algorithmen.

Der Meßfehler bei Zeitmessungen auf der MasPar ist außerordentlich gering. Die auf der Maschine verwendete Uhr hat eine Auflösung von 0,02 Sekunden und ist innerhalb dieser Auflösung über die für die Messungen relevanten Zeitintervalle als exakt zu betrachten. Die resultierenden Meßfehler liegen deshalb weit unter einem Promille und werden dementsprechend ignoriert. Aus Laufzeitprofilen, die für autoprune-Läufe mit den Problemen gene1, glass1, heart1 und soybean1 angefertigt wurden, kann man ablesen, daß die Zeit, die für das Zeitmessen aufgewendet wird, ebenfalls bei etwa einem Promille oder sogar darunter liegt. Die daraus entstehende Verfälschung kann also ebenfalls vernachlässigt werden.

Die Hardware der MasPar ist im Hinblick auf die Laufzeiten von Programmen nicht vollkommen deterministisch. Die Schwankungen rühren zum Beispiel daher, daß das Kommunikationsnetzwerk gelegentlich aufgrund von Datenfehlern eine Wiederholung der Übertragung eines Wortes vornehmen muß. Auch diese Schwankungen sind jedoch über die Gesamtlaufzeit eines Programms hinweg so gering, daß wir sie ignorieren.

In den meisten hier betrachteten Lernverfahren ist eine Zufallsinitialisierung der Gewichte nötig, um die Symmetrie der verborgenen Knoten zu brechen. Diese Zufallsinitialisierung führt zu unterschiedlichen Gesamtlaufzeiten bei Wiederholung des Programmlaufs, weil das Programm nach unterschiedlich vielen Schritten terminiert. Bei Lernverfahren, die unregelmäßige Netze erzeugen, kann die unterschiedliche Zufallsinitialisierung aber sogar dazu führen, daß ein einzelner Programmschritt (z.B. die Berechnung eines Beispiels) unterschiedlich lange dauert, weil das Netz von Mal zu Mal eine andere Struktur bekommt. Die daraus entstehenden Laufzeitschwankungen sind nicht sehr groß, müssen aber als Fehler beachtet werden. Zur Abschätzung dieses Fehlers wurde er an Beispielen konkret gemessen: Für das autoprune-Verfahren wurde in je 10 Läufen mit den Problemen cancer1 (mit nur 4+2 verborgenen Knoten in zwei verborgenen Schichten) und flare1 (mit 32 verborgenen Knoten in einer verborgenen Schicht) für jeden Lauf die mittlere Laufzeit pro Epoche gemessen. Dabei wurde über die je 10 Läufe eine Schwankung von bis zu 3,7% (cancer1) bzw. 8,2% (flare1) unter den Mittelwert und 4,8% bzw. 8,7% über den Mittelwert beobachtet. Die Standardabweichung lag bei 2,3% bzw. 4,7% des Mittelwerts. Wir nehmen den schlechteren der beiden Fälle als repräsentativ an und gehen für die folgenden Messungen von einer Unschärfe in der Größenordnung von 5% aus, wenn unterschiedliches Verhalten aufgrund verschiedener Zufallsinitialisierung vorliegt. In vielen Fällen ist es allerdings möglich, diese Unschärfe vollständig auszuschalten, indem für die verschiedenen Programmvarianten identische Startbedingungen und damit identisches logisches Programmverhalten erzwungen wird. In diesen Fällen wird also der größte Anteil des statistischen Fehlers eliminiert. Ein solches Vorgehen zählt zur Klasse der Techniken zur Varianzreduktion und ist allgemein empfehlenswert, wenn Varianten stochastischer Algorithmen verglichen werden sollen [240].

Insgesamt ist der statistische Fehler also erfreulich gut kontrollierbar und nicht sehr groß: Ist eine unterschiedliche Zufallsinitialisierung der verglichenen Programme unvermeidbar, so liegt er im Mittel bei 5% oder darunter. Kann die unterschiedliche Zufallsinitialisierung vermieden werden, liegt er im Bereich von einem Promille oder darunter.

11.3 Gesamtleistungsvergleich

Zur Rechtfertigung aller anderen Auswertungen muß zunächst sichergestellt sein, daß der Übersetzer Code produziert, der insgesamt eine gute Effizienz aufweist. Andernfalls könnten starke Verbesserungen durch die Optimierungen möglicherweise daher rühren, daß lediglich grobe Fehler der Codeerzeugung ausgebügelt werden. Für diese Prüfung vergleichen wir die Laufzeit eines CuPit-Programms mit der eines bekanntermaßen recht effizienten Programms auf der gleichen Maschine (Abschnitt 11.3.2). Zuvor ordnen wir die absolute Rechenleistung der erzeugten Programme ein und vergleichen sie mit der Spitzenleistung der Maschine und mit der Leistung spezialisierter Programme zur Simulation neuronaler Netze (Abschnitt 11.3.1).

11.3.1 Absolute Leistung

Abhängig von der Größe der gewählten Netze und Datensätze und von der Komplexität des Lernverfahrens erreichen CuPit-Programme mit dem vorgestellten Übersetzer auf der MasPar MP-1 eine absolute Rechenleistung in der Größenordnung von bis zu etwa 1500 KCUPS (connection updates per second, ein CU ist ein Vorwärts- plus ein Rückwärtsdurchlauf durch eine Verbindung) für das RPROP-Verfahren mit statischer Netztopologie (z.B. 1135 KCUPS für RPROP mit 32 verborgenen Knoten auf dem soybean-Problem; bei größeren Netzen und Datenmengen wird noch höhere Leistung erreicht). Ich verzichte auf eine genauere Aufstellung, weil diese wegen der Abhängigkeit von den obenerwähnten Parametern durch ihre subjektive Auswahl eher irreführend wäre. Stattdessen diskutieren wir den Vergleich des obigen Leistungswertes mit anderen Implementierungen.

11.3.1.1 Vergleich mit sequentiellen Rechnern

Die oben angegebene Rechenleistung ist absolut gesehen nicht hoch, wenn man sie mit der Leistung üblicher Implementationen von neuronalen Netzen auf sequentiellen Maschinen vergleicht. So erreicht zum Beispiel der Simulator SNNS [415] auf einer SUN SparcStation 1+, die derselben Rechnergeneration angehört, wie die MasPar MP-1, bis zu ca. 200 KCUPS ([199] Seite 60 und eigene Messungen; alle Angaben sind für 32-bit Gleitkommazahlen). Aufgrund ihrer wesentlich allgemeineren Implementierungsform, einem völligen Verzicht auf zahlreiche Feinoptimierungen, der Berechnung zahlreicher Fehlerstatistiken und dem etwas aufwendigeren RPROP-Verfahren, sind die zugrundegelegten CuPit-Programme allerdings unabhängig vom Übersetzer deutlich langsamer und erreichen auf einer SparcStation 1+ nur etwa 80 KCUPS. Dennoch ist die Beschleunigung von etwa Faktor 18 beim Übergang auf die MasPar nicht überwältigend.

11.3.1.2 Vergleich mit Spitzenleistung der MP-1

Da ein CU etwas mehr als 6 Gleitkommaoperationen repräsentiert, werden also von der auf der MasPar MP-1 höchstens erreichbaren Spitzenleistung von 490 MFLOPS[1] also ungefähr 1,8% erreicht. Dieser Wert ist für einen Parallelrechner aber nicht schlecht: [275, Seite 639] berichtet über 12 Benchmarks auf drei verschiedenen *Vektor*rechnern ein harmonisches Mittel von etwa 1% der Spitzenleistung — man bedenke, daß Vektorrechner eine sehr viel bessere Ausnutzung ihrer Rechenleistung bieten als echte Parallelrechner mit verteiltem Speicher. Wie wir auf Seite 131 am Beispiel des Neurorechners MY-NEUPOWER gesehen haben, der etwa 10% seiner Spitzenleistung erreichen kann, ist es bei Parallelrechnern selbst mit Spezialarchitekturen schwierig, mehr als einen kleinen Teil der Spitzenleistung zu realisieren.

11.3.1.3 Vergleich mit schnellen SIMD-Implementierungen

Die höchste für neuronale Netze auf der MP-1 realisierte Rechenleistung schafft der Simulator KNNS [199]. Im Gegensatz zu früheren Implementierungen neuronaler Netze auf der MasPar

[1] Berechnet aus der Dauer einer Gleitkommaoperation $t_{flop} = 2t_{load} + 1/2(t_{add} + t_{mult}) + t_{store}$.

[5, 134, 135, 136, 224, 230] kann dieser Simulator auch unregelmäßig strukturierte Netze bearbeiten. Er erreicht laut [199, Seite 70f] bei ausreichend großen Lernproblemen mit vollverbundenen Netzen eine Leistung von etwa 35 MCUPS, also mehr als 40% der Spitzenleistung — ein unerhört guter Wert. Für kleinere Probleme und nicht vollverbundene Netze sinkt diese Leistung ab, bleibt aber für ein recht weites Spektrum von Problemen über 10 MCUPS. Diese Leistung wird erreicht, indem für eine gegebene Netztopologie in einem separaten Vorverarbeitungsschritt eine Datenverteilung berechnet wird, die es erlaubt, ausschließlich die schnelle Nachbarkommunikation zu nutzen. Die verwendete Herangehensweise hat drei Nachteile:

1. Die Übertragung des Ansatzes auf MIMD-Maschinen ist weitgehend nutzlos, da dort die Nachbarkommunikation meist nur mäßig viel schneller ist als allgemeine Kommunikation, wohingegen auf der MasPar der Geschwindigkeitsunterschied zwischen den KNNS-Nachbarkommunikationen und den im CuPit-Übersetzer verwendeten allgemeinen Kommunikationsoperationen mehr als Faktor 200 beträgt!
2. Die Vorverarbeitung benutzt eine Transformation der Netzstruktur, die Knoten aufspaltet und Pseudoknoten einfügt, um eine ungefähr regelmäßige Netzstruktur zu erhalten. Diese Transformation benötigt Kenntnis der Semantik der in den Knoten und Verbindungen vorkommenden Datenelemente und Operationen, um die Umformung des Netzes ohne Veränderung von dessen Funktionalität durchführen zu können. Aus diesem Grund kann der Ansatz nur in einem geschlossenen Simulator verwendet werden, in dem diese Semantik für das jeweilige Lernverfahren genau bekannt ist. Eine Übertragung auf die Übersetzung einer Programmiersprache wie CuPit, in der eine beliebige unbekannte Semantik der Netzelemente vorkommen kann, ist nicht möglich.
3. Aufgrund des separaten Vorverarbeitungsschritts unterstützt das Verfahren keine dynamische Veränderung der Netztopologie. Damit geht die Anwendbarkeit gerade für die Fälle verloren, für die CuPit konzipiert ist, nämlich konstruktive Lernverfahren.

11.3.1.4 Fazit

Wie sind diese Ergebnisse zu interpretieren?

1. Auf SIMD-Maschinen wie der MasPar MP-1 können durch geeignete Programmarchitekturen, die ausschließlich die schnelle Nachbarkommunikation benutzen, sehr viel höhere Leistungen erzielt werden. Diese Ansätze erlauben jedoch nicht die Realisierung der Flexibilität des KNA-Programmiermodells in einem Übersetzer. Außerdem sind die Vorzüge auf SIMD-Maschinen beschränkt; die Leistungsvorteile gehen auf MIMD-Maschinen verloren.
2. Die mit dem CuPit-Übersetzer erzielten Leistungen sind, absolut betrachtet, nicht hoch. Die Ergebnisse sind aber nicht schlechter als für viele andere kommunikationsintensive Probleme auf Parallelrechnern.
3. Zusammenfassend kann man sagen, daß die gegenwärtigen Parallelrechner sich nicht gut für die effiziente allgemeine Simulation von neuronalen Netzen eignen. Das gilt selbst für die hier betrachtete, relativ am besten balancierte Maschine MasPar MP-1. Eine Verbesserung der effektiven Kommunikationsleistung (z.B. mit Hilfe von Latenzzeitverbergung) ist nötig, bevor zufriedenstellende Leistungen erreicht werden können.

Wir wenden uns deshalb von absoluten Leistungsbetrachtungen ab und untersuchen, was für relative *Verbesserungen* bei der Implementation dynamischer unregelmäßiger neuronaler Netze im Vergleich verschiedener Implementierungsmethoden zu erreichen sind.

11.3.2 Vergleich mit Modula-2*

Wie oben schon gesagt, muß zur Rechtfertigung aller anderen Auswertungen sichergestellt sein, daß der Übersetzer effizienten Code produziert. In der nunmehr eingenommenen relativen Betrachtungsweise bedeutet „effizient", daß der Code ungefähr so schnell ist, wie er unter den Vorgaben sein kann, wobei zu den Vorgaben gehören

1. ein gegebenes CuPit-Programm mit nicht näher bekannter Semantik,
2. die Erzeugung der Implementation durch einen Übersetzer,
3. eine gegebene Maschine, nämlich die MasPar MP-1, wobei keine Maschineneigenschaften ausgenutzt werden sollen, die die Effizienz stark verbessern, aber nicht auf andere Rechner (insbesondere MIMD) übertragbar sind.

Eine gute Vergleichsbasis wäre also ein anderer Übersetzer für eine Hochsprache auf der MP-1, wenn von ihm bekannt wäre, daß er effizienten Code erzeugt. Glücklicherweise ist ein solcher Übersetzer vorhanden, nämlich der Modula-2*-Übersetzer von Philippsen et al. [278]. Der von ihm erzeugte Code wurde auf einer Reihe von Beispielprogrammen für Algorithmen aus verschiedenen Anwendungsbereichen mit handgeschriebenen, handoptimierten Implementationen verglichen und erreichte im Mittel 80% von deren Leistung.

Wir vergleichen deshalb im folgenden die Laufzeit eines CuPit-Programms mit einem funktionsgleichen Modula-2*-Programm. Wir verwenden ein Programm für ein regelmäßiges Problem, das den Modula-2*-Übersetzer nicht dadurch benachteiligt, daß dieser nicht zur Behandlung unregelmäßiger Probleme konzipiert ist. Wenn der CuPit-Übersetzer ein gleichermaßen effizientes Programm erzeugt wie der Modula-2*-Übersetzer, dürfen wir im folgenden von einer ausreichend guten Qualität des Codes ausgehen.

11.3.2.1 Versuchsaufbau

Wir verwenden für den Vergleich ein Programm, welches das RPROP-Lernverfahren für völlig regelmäßig verbundene, dreischichtige Netze realisiert. Wie das CuPit-Programm realisiert auch das Modula-2*-Programm Parallelität auf der Ebene von Knoten, Verbindungen und Beispielen (letzteres durch Replikation des Netzes). Um Verfälschungen durch unterschiedlich effiziente Ein-/Ausgabeprozeduren auszuschließen, verwenden wir als Lernproblem das Encoder-Decoder-Problem, dessen triviale Trainingsdaten (nämlich die Worte eines 1-aus-n Bitcodes) vom Programm im Fluge selbst erzeugt werden können; das konkret verwendete Lernproblem ist ja für die Laufzeit eines nicht-konstruktiven Verfahrens völlig irrelevant.

Es wird alles getan, um sicherzustellen, daß der vom Modula-2*-Übersetzer erzeugte Code so effizient wie möglich ist:

1. Es wird ein regelmäßiges Netz benutzt, weil dies eine hinreichende und notwendige Bedingung dafür ist, daß der Modula-2*-Übersetzer Datenlokalität herstellen kann.
2. Aufrufe von Knoten- und Verbindungsprozeduren werden textuell expandiert, also der Prozedurrumpf an der Aufrufstelle eingefügt, um die Laufzeitkosten zu vermeiden, die andernfalls aus der *copy-in/copy-out-Semantik* der Argumentübergabe bei Modula-2*-Prozeduraufrufen entstehen würden.
3. Alle drei Parallelitätsebenen wurden in eine einzige FORALL-Anweisung „ausgerollt", um die Startkosten für das Betreten paralleler Abschnitte zu minimieren.
4. Datenelemente nichtlokaler Verbindungen, die in derselben Verbindungsoperation mehrfach gelesen oder geschrieben werden, werden in lokalen Variablen gepuffert.

Damit verbleiben zwei Nachteile für das Modula-2*-Programm gegenüber dem CuPit-Programm: Erstens ist der Aufruf einer FORALL-Anweisung allgemeiner und damit aufwendiger als ein paralleler Objektprozeduraufruf in CuPit. Zweitens kann der Modula-2*-Übersetzer nicht mehrere Kommunikationsoperationen zum Zugriff auf Elemente derselben nichtlokalen Verbindung zusammenfassen, wie es der CuPit-Übersetzer tut.

Auf der anderen Seite hat das Modula-2*-Programm aber auch zwei Vorteile gegenüber dem CuPit-Programm: Erstens holt und sendet es in einer Verbindungsoperation nur diejenigen Komponenten einer nichtlokalen Verbindung, die *tatsächlich* verwendet werden, während das CuPit-Programm alle Komponenten holen und senden muß, die gemäß der simplen statischen Analyse *möglicherweise* verwendet werden. Zweitens vermeidet das Modula-2*-Programm nach dem Zusammenführen der Daten von Replikaten die überflüssige Wiederverteilung der nicht betroffenen Komponenten. Diese Vermeidung ist im CuPit-Übersetzer bisher nicht implementiert. Im gegebenen Programm muß nur etwa ein Fünftel aller Daten zusammengeführt und wieder verteilt werden. Im Modula-2*-Programm treten auch nur diese Datenbewegungen auf, das CuPit-Programm bewegt hingegen jeweils sämtliche Daten.

Es werden die Laufzeiten folgender Probleme gemessen: ein 128-13-127 Netz mit 127 Beispielen[2], ausgeführt mit 64 oder 16 Netzreplikaten; ein 129-13-128 Netz mit 128 Beispielen, ausgeführt mit 16 Replikaten und ein 501-13-500 Netz mit 500 Beispielen, ausgeführt mit 16 oder 4 Replikaten. Diese Problemgrößen sind so ausgewählt, daß das CuPit-Programm tendenziell benachteiligt ist, denn 13 Knoten lassen sich nur schlecht auf ein zweierpotenzgroßes Segment verteilen, und die relativ kleine Zahl von Trainingsbeispielen führt dazu, daß der Wasserkopf der Wiederverteilung der Daten nach dem Zusammenführen der Replikate stark betont wird.

11.3.2.2 Ergebnisse und Fazit

Die Ergebnisse sind in Tabelle 11.1 dargestellt. Das Modula-2*-Programm ist schneller als das CuPit-

Netz	R	t_{rel}	t'_{rel}
127-13-127	64	90%	69%
127-13-127	16	182%	146%
128-13-128	16	140%	140%
500-13-500	16	111%	111%
500-13-500	4	187%	187%
(Mittel)		142%	130%

Tabelle 11.1: Relative Laufzeiten t_{rel} des Modula-2*-Programms im Vergleich zum äquivalenten CuPit-Programm für verschiedene Netzgrössen bei R Replikaten. t'_{rel} ist die relative Laufzeit bei abgeschalteter Kommunikationsbündelung im CuPit-Programm.

Programm, wenn zu viele Replikate benutzt werden. Im Beispiel mit 64 Replikaten beträgt die Laufzeit des Modula-2*-Programms relativ zum CuPit-Programm nur 90%. Dieses Resultat kommt durch die Ersparnis des Modula-2*-Programms beim Wiederverteilen der Daten zustande. Für kleinere Replikatanzahlen ist das CuPit-Programm stets schneller. Das Mittel über die angegebenen relativen Laufzeiten des Modula-2*-Programms ist 142%. Wird die Kommunikationsbündelung im CuPit-Übersetzer abgeschaltet, beträgt der Wert immer noch 130%.

Der vom CuPit-Übersetzer erzeugte Code ist also etwa ein Drittel schneller als ein äquivalenter Modula-2*-Code. Gemäß der erwähnten Ergebnisse aus [278] (Modula-2* Programme haben ca. 80% der Leistung handimplementierter Programme) ist CuPit-Code in der Leistung folglich Handimplementierungen vergleichbar, wenn wir annehmen, daß Modula-2* für das betrachtete Backpropagation-Programm nicht wesentlich schlechter ist als für andere Programme und wenn die Handimplementierungen bezüglich der Netzgröße etc. ebenso allgemein sind wie das CuPit-Programm. Die Verbesserung gegenüber Modula-2* beruht vor allem darauf, daß in CuPit-Programmen das Starten eines parallelen Abschnitts wesentlich effizienter erledigt werden kann, weil nicht die volle Allgemeinheit der FORALL-Semantik von Modula-2* benötigt wird.

[2] Die Werte kommen aufgrund der Lernaufgabe zustande: Es wurde ein Encoder-Problem benutzt, weil dafür die Trainingsdaten trivial vom Programm selbst erzeugt werden können (nämlich die Worte eines 1-aus-n Codes) und somit die von Modula-2* nicht unterstützte parallele Ein-/Ausgabe vermieden werden kann. Ein Encoder-Problem hat n Beispiele für ein n-k-n Netz. Der zusätzliche Eingabeknoten in obigen Netzen ist der Verschiebeknoten.

Es ist zu betonen, daß dieser Vergleich auf einem statischen und regelmäßigen Problem stattfand, also auf einem Bereich, in dem der CuPit-Übersetzer seine eigentlichen Stärken überhaupt noch nicht ausspielen kann. Ein fairer Vergleich für dynamische, unregelmäßige Probleme ist jedoch nicht möglich, weil auf der MasPar kein anderer Übersetzer zur Verfügung steht, der für solche Probleme geeignet ist.

11.4 Lastbalance

Laufzeitverbesserungen durch eine Lastbalancierung können naturgemäß fast beliebig hoch ausfallen (im Prinzip bis Faktor P bei P Prozessoren), wenn man nur entsprechend extreme Beispiele für die Bewertung benutzt. Wir sind hier jedoch an einer möglichst realistischen Einschätzung für praktisch auftretende Fälle interessiert und wählen deshalb zur Bewertung ein moderates Beispiel.

Die Klasse von Verfahren (unter den in dieser Arbeit betrachteten), bei denen sich nennenswerte Unregelmäßigkeiten in den Netzen ergeben, die eine Lastbalancierung sinnvoll machen, sind die Beschneidungsverfahren. Bei den Verfahren autoprune und lprune, wie sie im Kapitel 6 beschrieben sind, werden die Netze typischerweise bis auf eine Restgröße von 2% bis 15% der ursprünglich vorhandenen Verbindungen beschnitten, bevor das Verfahren terminiert. Um eine besser greifbare Vergleichsbasis herzustellen, betrachten wir hier die Laufzeit des autoprune-Verfahrens jeweils für die erste Epoche, die nach der Entfernung von (etwa) zwei Drittel der vorhandenen Verbindungen durchgeführt wird.

Wir vergleichen die Laufzeit für drei Programmvarianten des autoprune-Verfahrens: Erstens die normale Version (genannt *bal*) mit voller Lastbalancierung, deren Laufzeit wir zu 100% definieren. Zweitens eine Version, die zwar eine Lastbalancierung durchführt, die Last jedoch nicht wie die normale Version tatsächlich mißt, sondern aus der Anzahl von Verbindungen schätzt. Dabei wird im Unterschied zur messenden Version für jede Verbindung die gleiche Last veranschlagt. Diese Version nennen wir „dumme" Lastbalancierung (*dbal*). Drittens eine Version, die gar keine Lastbalancierung durchführt, sondern stets jedem Knoten einen gleichgroßen Prozessorblock zuweist. Diese Version nennen wir „nichtbalancierend" (*nbal*). Die sich aus diesem Versuchsaufbau ergebenden Werte für die Wirksamkeit der Lastbalancierung sind als eher konservativ einzustufen, denn es kommen ja durchaus stärkere Unregelmäßigkeiten als die betrachteten vor. Da die automatische Wahl der Replikatanzahl die Vermessung der Lastbalancierung stark stört, wurde sie in diesem Versuch abgeschaltet.

Die Ergebnisse dieser Messungen sind in der Tabelle 11.2 dargestellt. Wie wir sehen, ist die Lastba-

Problem	$\frac{t_{dbal}}{t_{bal}}$	$\frac{t_{nbal}}{t_{bal}}$
building	102%	123%
flare	105%	120%
hearta	102%	114%
cancer	110%	150%
card	102%	139%
diabetes	108%	129%
gene	102%	115%
glass	114%	130%
heart	105%	130%
horse	109%	137%
soybean	105%	132%
thyroid	100%	120%
(Mittel)	105%	128%

Tabelle 11.2: Relative Laufzeiten einer Version von autoprune ohne Lastbalancierung (t_{nbal}) bzw. mit „dummer" Lastbalancierung (t_{dbal}) im Vergleich zur optimierten Version mit voller Lastbalancierung. Das optimierte Programm entspricht 100%. Gemessen wurde nach Entfernung von etwa zwei Drittel aller Verbindungen.

lancierung für solche Netze zwar nicht unverzichtbar, ermöglicht aber doch schon eine deutliche Beschleunigung. Im Mittel über die betrachteten Probleme stellt sich eine Leistungsverbesserung durch die Lastbalancierung in Höhe von 28% ein. Die Verwendung echter Lastmessungen anstelle einer Gleichbehandlung aller Verbindungen verbessert die Leistung um 5% und ist folglich aufgrund der sehr geringen Kosten für die Messungen als sinnvoll einzustufen. Es ist interessant, wie unterschiedlich die Werte bei Änderung des Lernproblems ausfallen. Dies liegt am überaus komplexen Zusammenwirken der Einzelfaktoren Anzahl und Größe der Knotengruppen, Verhältnis der Knotengruppengrößen, Anzahl der Beispiele, Anzahl der Netzreplikate und natürlich Veränderung der Verbindungsstruktur durch das Beschneiden (abhängig von den Daten). Ähnliche Unterschiede werden sich auch bei den übrigen Messungen in diesem Kapitel einstellen. Die analytische Erklärung dieser Werte ist hoffnungslos schwierig, weil das Zusammenwirken der Einzelfaktoren mit den Eigenschaften der Zielmaschine zu kompliziert ist.

Für Verfahren, die stärkere Unregelmäßigkeiten in den Netztopologien erzeugen, zum Beispiel weil sie mit sehr viel größeren Netzen beginnen und schnell eine radikale Beschneidung daran vornehmen, stellen sich stärkere Verbesserungen durch die Lastbalancierung ein.

Bei der Übertragung der vorgestellten Übersetzungstechniken auf MIMD-Rechner ergeben sich zu der hier auf der MasPar MP-1 vermessenen Situation vor allem zwei eklatante Unterschiede: Erstens eine kleinere Prozessoranzahl und zweitens andere Latenzzeiten der Kommunikation im Verhältnis zur Rechenleistung eines Prozessors.

Während es schwierig ist, das Verhalten der Realisierung auf sehr viel kleineren Prozessorzahlen direkt aus der MasPar-Implementation des Übersetzers abzuschätzen, kann der Einfluß unterschiedlicher Latenzzeiten auf die Wirksamkeit der Lastbalancierung simuliert werden. Es gibt zwei Fälle: Erstens eine dramatisch höhere Latenzzeit, wie sie bei den heutigen MIMD-Rechnern überwiegend vorliegt, und zweitens eine sehr geringe Latenzzeit, die durch Latenzzeitverbergung auf zukünftigen Rechnern erreicht werden wird.

Simulationen dieser Fälle können auf folgende Weise durchgeführt werden: Zur Vortäuschung einer höheren Latenzzeit wird der Übersetzer so instrumentiert, daß er Code erzeugt, der für jede Kommunikationsoperation tote Wartezeit zur Simulation einer höheren Latenzzeit einfügt. Das entstehende Programm wird auf normale Weise vermessen. Zur Vortäuschung eines Rechners mit Latenzzeit Null werden die beim Zugriff auf nichtlokale Verbindungsobjekte tatsächlich auftretenden Latenzzeiten gemessen, summiert und von der tatsächlichen Laufzeit abgezogen. Diese reduzierte Laufzeit wird nicht nur zur Feststellung des Ergebnisses verwendet, sondern auch innerhalb des Programms bei der Berechnung der richtigen Datenverteilung für die Lastbalancierung. Beide Mechanismen sind im CuPit-Übersetzer implementiert und können in [16] nachgelesen werden; sie werden durch Optionen beim Aufruf des Übersetzers eingeschaltet. Mit dem solchermaßen erzeugten Code wurden einige informelle Versuche durchgeführt, die ergaben, daß die Wirksamkeit der Lastbalancierung sich nur wenig aufgrund anderer Latenzzeiten ändert. Die Simulation von Latenzzeit Null ergab zum Beispiel auf dem Datensatz gene1 für t_{nbal}/t_{bal} einen Wert von 113%, also eine geringfügig kleinere Effektivität der Lastbalancierung als bei der normalen Latenzzeit.

11.5 Datenlokalität

Für die Prüfung der Datenlokalität stellt sich die Frage, welches der angemessene Vergleichsfall ist. Ich betrachte hier die beiden Extremfälle:

1. Implementiert man neuronale Netze in feldbasierten (array-basierten) Sprachen, so geht bei unregelmäßigen Netzen fast sämtliche Datenlokalität verloren. Immerhin erlaubt ein solcher Ansatz, der im Prinzip auch von einer Sprache mit verschachtelter Parallelität wie NESL verfolgt wird, die Balancierung der Last für unregelmäßige Probleme. Auch wenn wegen des Verlusts der Datenlokalität niemand wirklich solche Implementationen verwendet, sondern lieber ganz auf unregelmäßige

Netze verzichtet, vergleiche ich hier diesen Fall (näherungsweise) quantitativ mit dem von mir implementierten Übersetzer (Abschnitt 11.5.1).

2. Eine allgemeinere Methode zur Herstellung von Datenlokalität als die von mir benutzte ist die Graphpartitionierung. Dabei werden kompliziertere Techniken zur Berechnung der Datenverteilung nötig, aber es wird auch eine höhere Datenlokalität erreicht. Diese Kosten-Nutzen-Abwägung untersuche ich in Abschnitt 11.5.2.

11.5.1 Simulierte Nichtlokalität

Möchte man unregelmäßige neuronale Netze in feldbasierten parallelen Sprachen wie Modula-2* oder Fortran D implementieren, so muß eine Indirektionsstufe eingefügt werden: Die Verbindungen einer Schnittstelle s aller Knoten K einer Gruppe G werden dicht hintereinander in einem verteilten Feld F abgelegt und enthalten Verweise auf andere solche Felder, in denen die Gegenenden der Verbindungen abgelegt sind. Jeder Knoten K erhält ein Paar (i, j) von Indizes, das Anfang und Ende desjenigen Bereichs in F angibt, in dem die Verbindungen von $s(K)$ abgelegt sind. Bei dieser Zugriffsmethode tritt Datenlokalität nur noch zufällig auf; im Normalfall müssen alle Argumente von Verbindungsprozeduraufrufen und diese Aufrufe selbst per Rundruf an die jeweils betroffenen Prozessoren verteilt werden, die die Verbindungen enthalten. Außerdem gibt es genau wie in meinem Übersetzer lokale und nichtlokale Verbindungsobjekte; deshalb ist in etwa der Hälfte der Fälle außerdem ein Fernzugriff auf das Verbindungsobjekt nötig.

Die direkte Herstellung dieser Verhältnisse im CuPit-Übersetzer für einen direkten Vergleich ist nicht möglich. Als Annäherung benutzen wir die folgende Modifikation der Codeerzeugung des CuPit-Übersetzers: Der Übersetzer erzeugt seinen Code wie gewohnt, nur werden auch bei den lokalen Verbindungsoperationen Kommunikationsoperationen eingefügt, so als wären die Verbindungen nichtlokal. Wir messen also nicht mit Kommunikation von Prozeduraufrufen und Prozedurparametern für *alle* Aufrufe von Verbindungsprozeduren, sondern stattdessen mit Kommunikation der betroffenen Verbindungsdaten für die *eigentlich lokalen* Verbindungsobjekte, also bei etwa der Hälfte der Aufrufe. Die dabei bewegten Datenmengen entsprechen sich ungefähr, so daß wir effektiv eine Annäherung der Verhältnisse von feldbasierten Sprachen messen. Da bei diesem Aufbau die Routinen zur Herstellung der Datenverteilung und zum Zusammenführen der Replikate nicht mitmodifiziert sind, nehmen wir ihre Ausführungszeiten für die originale und die modifizierte Version aus der Messung heraus. Die

Problem	relative Laufzeit
building	244%
flare	289%
hearta	325%
cancer	305%
card	333%
diabetes	294%
gene	221%
glass	309%
heart	320%
horse	359%
soybean	288%
thyroid	247%
(Mittel)	295%

Tabelle 11.3: Relative Laufzeit bei simuliertem Fehlen von Datenlokalität für verschiedene Probleme (optimiertes Programm entspricht 100%). Die Werte wurden für statische vollverbundene Netze gemessen. Sie ändern sich für Beschneidungsverfahren geringfügig nach oben oder unten.

erzielten Ergebnisse für RPROP-Lernen auf statischen Netzen sind in Tabelle 11.3 dargestellt. Sie ändern sich für die in dieser Arbeit betrachteten Beschneidungsverfahren nur wenig. Bei komplexen

Lernverfahren, die mehr Parameter in ihren Prozeduraufrufen benötigen, liegen die Werte höher, und bei Verfahren, die gleichmäßig viel ihre Eingangs- wie ihre Ausgangsschnittstellen beanspruchen, liegen sie niedriger. Die vorgestellten Kandidatenlernverfahren aus Kapitel 5 haben beide Eigenschaften und dürften deshalb ebenfalls etwa im angegebenen Bereich liegen; Messungen wurden jedoch nicht durchgeführt.

Wie wir sehen, wären feldbasierte Implementationen unregelmäßiger Netze aufgrund ihrer fehlenden Datenlokalität um etwa Faktor 2 bis 3 langsamer als die Implementationen des CuPit-Übersetzers.

11.5.2 Prüfung der Annahme 5

Eine der Annahmen, die der Datenverteilungsstrategie des Übersetzers zugrundeliegen, bedarf noch der Prüfung. Dies ist die Annahme 5 aus Abschnitt 10.1.2. Sie lautet:

> Der Versuch zahlt sich nicht aus, zur Erhöhung der Lokalität Knoten, zwischen denen eine Verbindung besteht, möglichst häufig auf demselben Prozessor anzuordnen.

Zur Prüfung dieser Annahme müssen eine Anzahl von möglichst repräsentativen unregelmäßigen Netzstrukturen einer Graphpartitionierung unterworfen werden. Die Anzahl in der Partitionierung durchschnittener Verbindungen ist ein Maß für die Kommunikationskosten und ist zu vergleichen mit der Anzahl durchschnittener Verbindungen einer zufälligen Partitionierung. Optimale Graphpartitionierung ist ein NP-hartes Problem; deshalb können nur heuristische Methoden verwendet werden, um die Zahl der durchschnittenen Verbindungen möglichst klein zu machen. Als Beurteilungsmaßstab für den Erfolg der schnittminimierenden Partitionierung dient neben der Anzahl durchschnittener Verbindungen die zur Berechnung erforderliche Zeit im Vergleich zur Rechenzeit für eine zufällige Partitionierung, wie sie der gegenwärtig im CuPit-Übersetzer verwendeten Datenverteilung faktisch zugrundeliegt.

Einem solchen Versuchsaufbau stellen sich mehrere Schwierigkeiten in den Weg: Erstens müßten zur genauen Beurteilung der Laufzeiten die Graphpartitionierungsverfahren tatsächlich im Laufzeitsystem für die MasPar MP-1 implementiert werden. Da die Verfahren aber zum Teil algorithmisch sehr aufwendig sind (siehe unten), kommt dies nicht in Frage. Als Ersatzlösung verwende ich ein sequentielles Graphpartitionierungsprogramm namens Chaco [153], das diverse einfache und komplexe Verfahren bereitstellt, insbesondere auch eine Zufallspartitionierung. Die an diesem sequentiellen Programm gemessenen Laufzeiten stellen nur eine Annäherung an die Verhältnisse dar, die sich bei einer parallelen Implementierung ergäben, weil nicht klar ist, ob sich die Zufallspartitionierung und die komplexeren Partitionierungsmethoden gleichermaßen gut parallelisieren lassen.

Zweitens müßte die Partitionierung genaugenommen für jedes Paar von Knotengruppen einzeln vorgenommen werden, wobei die einzelnen Partitionierungsprobleme gekoppelt sind, weil die verborgenen Schichten jeweils in zwei der Partitionierungen auftreten. Für solche Fälle wären also noch kompliziertere Algorithmen nötig als für die normale Partitionierung. Entsprechende Programme stehen aber nicht zur Verfügung. Deshalb machen wir die plausible Annahme, daß die Qualität der Lösungen, die sich für die gekoppelten Probleme finden lassen, höchstens so gut ist, wie die derjenigen, die wir bei der Partitionierung des gesamten Netzes finden. Die Laufzeit zur Lösung mehrerer gekoppelter Systeme nehmen wir als mindestens so groß an, wie die Laufzeit der Partitionierung in einem Stück. Wir berechnen nun Partitionierungen des Gesamtnetzes und verwenden die gemessenen Ergebnisse als Näherungen der gewünschten Ergebnisse für gruppenweises Partitionieren.

Drittens erhebt sich die Frage, was „repräsentative“ Netze sein könnten. Dies hängt natürlich erheblich von den verwendeten Lernverfahren ab. Insbesondere sind vollverbundene Netze einerseits ein häufiger Fall, andererseits einer Graphpartitionierung überhaupt nicht zugänglich (jede Partitionierung ist so gut wie jede andere), so daß die Häufigkeit, die man für sie annimmt, erheblichen Einfluß auf das Ergebnis der Betrachtung hat. Wir machen hier einen konservativen Versuchsaufbau, in dem überhaupt keine vollverbundenen Netze vorkommen.

11.5.2.1 Versuchsaufbau

Vollverbundene Netze haben keinerlei Lokalität in ihrem Verbindungsmuster, die ein Graphpartitionierungsverfahren ausnutzen könnte. Auch bei den Kandidatenlernverfahren aus Kapitel 5 entstehen aufgrund der Pufferung effektiv nur geringe Unregelmäßigkeiten in den Verbindungsstrukturen, so daß auch hierbei kaum Vorteile durch Graphpartitionierung zu erhalten sind. Wir betrachten im Sinne einer konservativen Abschätzung deshalb nur den im Hinblick auf die Gültigkeit der Annahme 5 schlimmsten Fall, nämlich die Beschneidungsverfahren. Konkret untersucht wird nur das autoprune-Verfahren, da beim lprune-Verfahren dieselben Beschneidungskriterien zugrunde liegen und sich deshalb ähnliche Verbindungsmuster einstellen. Die Beschneidungsstärke wird so gewählt, daß sie etwa repräsentativ für die Programmläufe ist, die im Kapitel 6 untersucht wurden: Wir betrachten die Netze, die nach dem ersten und nach dem sechsten Beschneidungsschritt vorliegen. Dies entspricht einer Beschneidung von etwa einem Drittel bzw. zwei Dritteln der ursprünglichen Verbindungen. Da einerseits typische Läufe von autoprune zwischen 9 und 18 Beschneidungsschritten ausführen, bevor sie terminieren, und dabei andererseits die auf die stärker beschnittenen Netze verwendete Rechenzeit relativ geringer ist, stellt diese Auswahl ein einigermaßen repräsentatives Bild her. Je ein solches Paar von Netzen für jedes der verschiedenen Benchmarkprobleme wird untersucht, also 24 Netze insgesamt.

Jedes Netz wird 4 verschiedenen Partitionierungsstrategien unterworfen:

1. Zufallspartitionierung („Zufall").
2. Lokale heuristische Partitionierung nach Kernighan-Lin („KL"). Dieses Verfahren verbessert eine (zufällige) Ausgangspartitionierung mit Hilfe eines *greedy*-Algorithmus schrittweise lokal.
3. Spektrale Bisektion („Spektral 2"). Das Verfahren berechnet den zweitkleinsten Eigenwert der Laplace-Matrix des Graphen und verwendet den zugehörigen Eigenvektor, um einen günstigen Schnitt des Graphen in zwei Teile zu berechnen; dabei wird zur Eigenwertberechnung die Lanczos-Iteration mit voller Orthogonalisierung benutzt. Dieses Verfahren wird iterativ wiederholt.
4. Spektrale Oktasektion mit lokaler Verfeinerung („Spektral 8+"). Das Verfahren basiert auf der spektralen Bisektion, hat jedoch zwei Erweiterungen: Erstens wird zusätzlich zum zweitkleinsten auch noch der dritt- und der viertkleinste Eigenvektor benutzt und der Graph in einem Schritt in 8 Teile zerlegt, was manchmal die Herstellung von für die weitere Partitionierung ungünstigen Teilgraphen vermeidet. Zweitens wird nach einem Zerteilungsschritt eine lokale Verfeinerung der Partitionierung mit dem Kernighan-Lin-Verfahren vorgenommen.
5. Mehrstufiges Vorgehen („Multi"). Hierbei wird der Graph durch Zusammenfassen von Knoten in einen kleineren Graphen (hier: 32 Knoten) umgewandelt, dieser dann partitioniert und die Partitionierung zurück auf den ursprünglichen Graphen projeziert. Die Partitionierung des kleineren Graphen wird mit spektraler Bisektion vorgenommen.

Jeder einzelne Schritt einer spektralen Zerteilung ist seinerseits ein iteratives und numerisch kompliziertes Verfahren. Für eine genauere Beschreibung dieser Techniken siehe [153] und die dort angegebene Literatur. Alle diese Verfahren erzeugen stets eine zweierpotenzgroße Anzahl von Teilgraphen und zwar in unseren Versuchen jeweils 16. Das Partitionierungsprogramm Chaco hat zahlreiche Parameter, mit denen das Verhalten der Verfahren gesteuert werden kann; es werden überall die voreingestellen Standardwerte verwendet, die von den Autoren des Programms zur Erzielung möglichst robust guter Ergebnisse vorgesehen sind.

11.5.2.2 Ergebnisse und Fazit

Tabelle 11.4 enthält die Laufzeiten und die Anzahl durchschnittener Verbindungen der von den verschiedenen Verfahren produzierten Partitionierungen für die 24 untersuchten Netze. Wie man sieht, hat die Graphpartitionierung ihre Tücken: Obwohl Chaco ein brauchbares Programm ist, mit dem sich

Tabelle 11.4: Ergebnisse der Graphpartitionierung

		Zufall		KL		Multi		Spektral 2		Spektral 8+	
Problem	n	t	S	T	ΔS	T	ΔS	T	ΔS	T	ΔS
building	1	10	224	6	-7%	20	-4%	36	-6%	179	-6%
building	6	10	130	6	-8%	21	-8%	33	-8%	106	-10%
cancer	1	20	69	3	-3%	6	-3%	7	-3%	X	N
cancer	6	20	44	2	-5%	8	-5%	X	N	X	N
card	1	20	1140	10	-4%	25	-3%	X	F	X	F
card	6	20	638	7	0%	X	F	X	F	X	F
diabetes	1	10	197	8	0%	23	0%	28	0%	85	0%
diabetes	6	X	X	X	X	X	X	X	X	X	
flare	1	10	633	13	-4%	36	-4%	X	F	X	F
flare	6	10	377	10	-6%	X	F	X	F	X	F
gene	1	20	692	8	-4%	X	F	X	F	X	F
gene	6	20	428	9	-10%	X	F	X	F	X	F
glass	1	20	344	5	-4%	12	-2%	15	-4%	37	-3%
glass	6	30	158	2	-9%	6	-8%	13	-10%	101	-9%
heartac	1	20	70	3	-3%	X	F	X	F	X	F
heartac	6	20	45	3	-7%	X	F	X	F	X	E
heartc	1	20	723	7	-3%	X	F	X	F	X	E
heartc	6	10	443	12	-7%	X	F	X	F	X	E
horse	1	20	1075	10	-4%	36	-4%	78	-4%	88	-4%
horse	6	10	560	17	-8%	X	F	X	F	X	F
soybean	1	30	2623	13	-3%	X	F	X	F	X	F
soybean	6	20	1592	17	-7%	X	F	X	F	X	F
thyroid	1	10	517	11	-5%	40	-4%	43	-5%	X	F
thyroid	6	10	315	11	-9%	X	F	X	F	X	F

Gemessen für je zwei Netze (nach $n = 1$ und $n = 6$ autoprune-Beschneidungsschritten) pro Problem. t: Laufzeit in Millisekunden; S: Anzahl durchschnittener Verbindungen; T: Laufzeit als Vielfaches von t; ΔS: Veränderung der Schnittanzahl gegenüber S; X: „kein Eintrag“; N: „Programmabbruch wegen numerischer Probleme“; F: „Programmabsturz wegen Datenzugriffsfehler“; E: „Programmabsturz (Endlosschleife)“.

Partitionierungsaufgaben in einer Größe von 100 000 Knoten erfolgreich lösen lassen[3], schlagen viele der Partitionierungsversuche für die vorliegenden neuronalen Netze fehl. Eine mögliche Erklärung hierfür liegt darin, daß Chaco hauptsächlich für Partitionierungen zu Finite-Elemente-Methoden konzipiert ist. Die dort höchstens auftretenden Verbindungszahlen pro Knoten (Knotengrade) sind typischerweise erheblich kleiner (ca. bis 20) als bei den neuronalen Netzen (bis 127). Es ist die Tendenz zu beobachten, daß gerade diejenigen Programmläufe mit hohen Knotengraden fehlschlagen. In einigen Fällen kann das Programm die Schwierigkeiten erkennen und sich vorzeitig ordentlich beenden, in den meisten Fällen tritt aber ein fehlerhafter Datenzugriff (Segmentation Fault) und folglich Programmabsturz auf.

Die Trends lassen sich jedoch aus den verbleibenden Daten hinreichend gut ablesen: Der Gewinn durch Graphpartitionierungsmethoden liegt bei weniger als 10% Verringerung der Schnittanzahl, während die Kosten ein Vielfaches der Kosten für die Zufallsverteilung betragen; interessanterweise sind die komplexen Partitionierungsmethoden zwar teurer aber keineswegs besser als die simple KL-Heuristik. Dieses Ergebnis gilt wie bereits im Abschnitt 10.1.2 erwähnt nicht für modulare Netze, die wir in dieser Arbeit nicht betrachten, da solche Netze von den betrachteten konstruktiven Lernverfahren nicht

[3] Auskunft von Paul Lukowicz (persönliche Kommunikation, November 1994)

erzeugt werden. Unter dieser Einschränkung scheint sich die Anwendung von Graphpartitionierungsverfahren bei neuronalen Netzen nicht zu lohnen, insbesondere in Anbetracht der Tatsache, daß in obigem Experiment nur relativ gut partitionierbare Netze verwendet wurden, während im wirklichen Programmbetrieb viele der vorgelegten Netze vollverbunden oder beinahe vollverbunden sind und deshalb sowieso kaum irgendwelche Verbesserungen gegenüber einer Zufallspartitionierung zulassen.

Nimmt man optimistisch an, daß im Mittel eine Verbesserung der Datenlokalität bei Zugriffen auf „nichtlokale" Verbindungen von 5% durch Graphpartitionierung möglich wäre, daß solche Zugriffe die Hälfte der Programmlaufzeit ausmachen und daß die Berechnung einer zufälligen Datenverteilung nur 0,5% der Programmlaufzeit kostet, so ergeben sich nur noch geringe Verbesserungsaussichten durch die Graphpartitionierung: Sei die Berechnung der Datenverteilung mit Graphpartitionierung als nur fünfmal so aufwendig angenommen, wie die zufällige Datenverteilung, dann steht einem Verbesserungspotential von 2,5% (5% von 50%) eine Erhöhung des Verwaltungsaufwands von 2% (Aufwand 2,5% anstatt 0,5%) gegenüber. Es verbleibt also allenfalls eine minimale Beschleunigung. Die in dieser Rechnung angenommenen Werte sind aber eher als optimistisch für die Graphpartitionierung zu betrachten, wie im nächsten Abschnitt zu sehen sein wird.

11.6 Speicher-, Verteilungs- und Kommunikationskosten

Um eine quantitative Schätzung für das Verhalten anderer als der hier betrachteten Lernprobleme und vor allem Lernverfahren berechnen zu können, wäre es interessant zu wissen, welcher Anteil der beobachteten Laufzeit auf das Konto der Verwaltungsoperationen geht, die für die Herstellung der Datenverteilung zuständig sind. Für die Abschätzung der Übertragung auf andere Rechner ist der Anteil von Kommunikationszeiten an der Gesamtlaufzeit von Bedeutung. Diese Werte wurden beide bereits im vorangegangenen Abschnitt in einer Modellrechnung eingesetzt.

Zur Erhebung dieser Werte wurden Laufzeitprofile für das autoprune-Verfahren (bis zum sechsten Beschneidungsschritt) erzeugt für die Probleme gene1, glass1, heart1 und soybean1. Die Resultate sind in Tabelle 11.5 dargestellt. Wie wir sehen, leidet der derzeitige Übersetzer erheblich daran, daß die

Tabelle 11.5: Anteile bestimmter Operationen an Gesamtlaufzeit

Prozedur	gene	glass	heart	soybean
MERGE	20,5%	49,6%	50,3%	29,4%
REPLICATE	3,4%	11,0%	5,4%	2,2%
compute layout	0,2%	1,1%	0,3%	0,1%
(Kommunikation)	40,7%	32,7%	45,0%	49,6%
(Verbindungen)	25,5%	4,0%	14,3%	29,3%

Gemessen bei autoprune-Läufen für gene1, glass1, heart1, soybean1. „MERGE": Vereinigen und Zurückverteilen der Daten der Netzreplikate nach jeder Epoche. „REPLICATE": Herstellen oder Zerstören von Netzreplikaten (Berechnung und Herstellung der Datenverteilung). „compute layout": Kern der Berechnung der Datenverteilung. „(Kommunikation)": Summe aller Kommunikationsoperationen. „(Verbindungen)": Summe der Kommunikationsoperationen zum Zugriff auf nichtlokale Verbindungen in Aufrufen von Verbindungsoperationen.

Prozedur MERGE, welche die Zusammenführung von Replikaten durchführt, nicht optimiert ist: Weder wird die Verteilung von Daten unterdrückt, die sich beim Zusammenführen ohnehin nicht ändern, noch werden die Kommunikationsoperationen für mehrere Objekte des gleichen Prozessors zusammengefaßt. Aus diesem Grund weisen vor allem die kleinen Beispielprobleme einen enormen Wasserkopf für diese Operation auf. Dies ist auch der Grund, weshalb die Kosten der MERGE-Operation in manchen der anderen Messungen in diesem Kapitel herausgerechnet werden: Ihre übersteigerten Kosten würden sonst das Ergebnis verdecken. Für die Interpretation der übrigen Werte ist zu bedenken, daß der Aufwand für MERGE auf weniger als die Hälfte reduziert werden könnte.

Die eigentliche Datenverteilungsoperation REPLICATE ist recht effizient. Sie benötigt nur etwa 5% der Laufzeit, obwohl sie bei einem Beschneidungsverfahren mindestens zweimal pro Beschneidungsschritt aufgerufen wird und zusätzlich diverse Male von der automatischen Optimierung der Replikatanzahl. Die Werte zeigen, daß die Verwendung einer dynamischen Datenstruktur, deren Verteilung bei jeder Topologieänderung neu berechnet wird, für neuronale Algorithmen ein sinnvoller Ansatz ist. Der Kern der Berechnung der Datenverteilung hat einen Aufwand von etwa 0,5% der Gesamtlaufzeit. Dieser Wert wurde im vorangegangenen Abschnitt in der Modellrechnung über Graphpartitionierung verwendet.

Insgesamt verbringt das autoprune-Programm weniger als die Hälfte seiner Zeit mit Kommunikation. Denkt man sich die MERGE-Prozedur effizienter, so würde etwa ein Drittel der Zeit für den Zugriff auf nichtlokale Verbindungsobjekte verwendet (ein Teil dieser Zeit fällt in Reduktionsoperationen über Verbindungen an und ist in der Tabelle nicht aufgeführt). Dieser relativ niedrige Wert würde sich erhöhen, wenn man das stark mit Protokollierungs-, Verwaltungs- und Überwachungsoperationen gespickte Benutzerprogramm mehr auf seinen algorithmischen Kern reduzierte, um es effizienter zu machen.

Der Speicherbedarf für die Verwaltung der Datenverteilung ist moderat. Es müssen zusätzlich zu den Nutzdaten jeweils die Deskriptoren gespeichert werden. Von allen Deskriptoren ist dabei nur der Verbindungsdeskriptor von Bedeutung, weil dieser als einziger in großen Anzahlen auftritt. In der MasPar-Implementierung ist ein Verbindungsdeskriptor einschließlich der Felder für die dynamische Lastmessung 13 Bytes groß, wovon ein prozessorlokaler Zeiger auf den Schnittstellendeskriptor 4 Bytes einnimmt und ein maschinenglobaler Zeiger auf das Gegenende der Verbindung 6 Bytes. Die Deskriptorgröße liegt somit in einem angemessenen Verhältnis zur Größe der eigentlichen Verbindungsobjekte: Letztere liegt für die Beschneidungsverfahren bei 36 Bytes, für die Kandidatenlernverfahren bei 32 Bytes. Bei komplexeren Lernverfahren sind noch höhere Werte zu erwarten. Somit wird in unserer Datenverteilung ungefähr ein Viertel des Speichers für die Verwaltungsinformation benötigt. Dieser Wert ist zwar höher, als er bei anderen Ansätzen auftreten würde, liegt aber angesichts der durch diese Datenverteilung ermöglichten Optimierungen in einem vernünftigen Bereich.

11.7 Kommunikationsbündelung

Die Bündelung der Kommunikationsoperationen, die vom CuPit-Übersetzer vorgenommen wird (siehe Abschnitt 10.4.1), läßt sich mit zwei einfacheren Alternativen vergleichen: Einerseits der Kommunikation immer des kompletten Verbindungsobjekts (M5) und andererseits der individuellen Kommunikation jeder einzelnen benötigten Komponente (M4).

Der CuPit-Übersetzer wurde entsprechend so instrumentiert, daß er automatisch diese beiden Varianten des eigentlichen Programms erzeugen kann. Wir untersuchen wiederum das Verhalten der verschiedenen Probleme für das autoprune-Verfahren. Es wurden je drei Läufe pro Problem durchgeführt und die Laufzeit der beiden unoptimierten Varianten mit der unmodifizierten, optimierenden Programmversion verglichen. Die Varianz aus der Zufallsinitialisierung wurde dabei durch reproduzierbar gleiche Initialisierungen unterdrückt.

Die Resultate sind in Tabelle 11.6 dargestellt. Wie wir sehen, ergeben sich gegenüber beiden vereinfachten Implementierungsvarianten Vorteile durch die Kommunikationsbündelung, die zwar nicht dramatisch, aber deutlich sind. Im Mittel gewinnt die Bündelung gegenüber dem Transfer kompletter Verbindungsobjekte etwa 28% Geschwindigkeit und gegenüber dem einzelnen Transfer aller benötigten Verbindungskomponenten etwa 10%. Daß der letztere Wert nicht höher ausfällt, liegt daran, daß im betrachteten autoprune-Lernverfahren stets nur wenige Komponenten in derselben Verbindungsoperation benutzt werden. Komplexere Lernverfahren, die mehr verschiedene Verbindungskomponenten benötigen, würden zu stärkeren Verbesserungen führen.

Problem	$\frac{t_{kompl}}{t_b}$	$\frac{t_{indiv}}{t_b}$
building	127%	110%
flare	138%	112%
hearta	138%	111%
cancer	105%	97%
card	128%	108%
diabetes	129%	110%
gene	100%	119%
glass	104%	101%
heart	128%	111%
soybean	197%	115%
thyroid	157%	121%
(Mittel)	128%	110%

Tabelle 11.6: Vergleich der Laufzeiten von autoprune mit Kommunikation kompletter nichtlokaler Verbindungsobjekte (t_{kompl}) oder mit individueller Kommunikation benötigter Komponenten (t_{indiv}) zu der Laufzeit bei annähernd optimaler Kommunikationsbündelung (t_b).

Für wesentlich andere Zielmaschinen würden sich die obigen Ergebnisse ändern: Auf Maschinen mit zunehmend höherer Latenzzeit nähert sich t_b immer mehr an t_{kompl} an (natürlich auf einem anderen absoluten Niveau als oben angegeben), weil zunehmend größere Lücken mitübertragen werden und die nichtübertragenen Reste des Verbindungsobjekts immer weniger ins Gewicht fallen. Es bleibt jedoch immer eine Ersparnis durch Bündelung übrig, wenn in irgendeiner Verbindungsoperation mindestens eine Komponente vom Anfang oder Ende des Verbindungsobjekts nicht mit übertragen werden muß. Mit zunehmend geringerer Latenzzeit hingegen nähert sich t_b immer mehr an t_{indiv} an, weil sich das Mitübertragen von Lücken immer weniger lohnt. Es bleibt jedoch immer eine Ersparnis durch Bündelung übrig, wenn in irgendeiner Verbindungsoperation mindestens zwei Komponenten des Verbindungsobjekts benötigt werden, die in der Speicherdarstellung direkt benachbart sind.

11.8 Verbindungsallokation

Wenn ein Programm die Verbindungen an Eingangs- und Ausgangsschnittstellen seiner Knoten nicht gleich stark beansprucht, macht es, wie in Abschnitt 10.4.2 dargelegt, für die Laufzeit einen Unterschied, an welche der Schnittstellen man die lokalen Verbindungsobjekte plaziert. Das Programm autoprune hat moderat eine solche ungleichmäßige Beanspruchung, da der Rückwärtsdurchlauf eines Beispiels beim Backpropagation-Verfahren mehr Verbindungskomponenten erfordert als der Vorwärtsdurchlauf und außerdem auch weitere Operationen (Gewichtsveränderungsschritt, Beschneidungsschritt) nur von der Eingangsschnittstelle aus aufgerufen werden. Wir messen daher zur Beurteilung des Nutzens aus richtiger Wahl der Verbindungsallokation die relativen Laufzeiten von autoprune-Läufen mit der „falschen" Entscheidung gegenüber Läufen mit der „richtigen".

Die Ergebnisse sind in Tabelle 11.7 dargestellt. Wie wir sehen, hat die richtige Wahl der Verbindungsallokation bei allen Problemen einen erheblichen Einfluß auf die Laufzeit. Im Mittel kann die Leistung durch die richtige Wahl im Vergleich zur falschen um 50% gesteigert werden. Dieser Wert wäre kleiner bei Lernverfahren, die nur wenig Unterschiede zwischen der Benutzung der Eingangs- und der Ausgangsschnittstellen der Knoten haben und größer bei Verfahren, bei denen dieser Unterschied stärker ist als bei den Beschneidungsverfahren. Für die meisten Verfahren ist wohl davon auszugehen, daß der Unterschied (und damit der beobachtete Effekt) etwas schwächer ist als bei dem hier betrachteten autoprune.

Problem	$\frac{t_{falsch}}{t_{richtig}}$
building	141%
flare	160%
hearta	178%
cancer	111%
card	144%
diabetes	180%
gene	140%
glass	115%
heart	149%
soybean	164%
thyroid	163%
(Mittel)	150%

Tabelle 11.7: Relative Laufzeiten der Versionen von autoprune mit falscher (d.h. ungünstiger) Verbindungsallokation im Vergleich zu denen mit richtiger (d.h. günstiger) Verbindungsallokation. Die günstige Entscheidung ist, die Verbindungen an die Eingangsschnittstellen zu plazieren.

11.9 Skalierung

Eine wichtige Frage bei parallelen Implementierungen lautet stets, wie gut die Rechenleistung mit der Zahl der Prozessoren skaliert (wohlgemerkt: bei konstanter Problemgröße [24]). Bei genügend feinkörniger Parallelität im Programm ist im Prinzip eine gute Skalierung (fast linear über einen weiten Bereich von Prozessoranzahlen) denkbar, jedoch stehen dem in der Regel einige Schwierigkeiten entgegen: erhöhter Verwaltungsaufwand, vermehrte Kommunikation und schwierigere Lastbalancierung. Im Falle eines CuPit-Übersetzers treffen alle drei Punkte zu, wobei sich der erhöhte Verwaltungsaufwand aus einer größeren Zahl von Netzreplikaten ergibt, falls nicht das betrachtete Netz sehr groß ist — was in keinem der untersuchten Probleme der Fall ist.

Leider kann auf einer SIMD-Maschine prinzipiell keine vollständige Skalierungskurve angegeben werden, d.h. eine, die bei $P = 1$ Prozessoren beginnt und alle Prozessoranzahlen abdeckt. Dies liegt daran, daß erstens SIMD-Prozessoren zu klein sind, als daß einer allein das ganze Problem bearbeiten könnte (Speichermangel) und zweitens der zentrale Steuerprozessor, der ja nicht mit herunterskaliert werden kann, das Ergebnis in diesem Fall zu stark beeinflussen würde.

Die auf der MasPar für eine Messung verfügbaren Prozessoranzahlen sind 1k, 2k, 4k, 8k und 16k Prozessoren; nur für diese Werte kann eine Skalierungskurve angegeben werden. Der davon abgedeckte Bereich ist jedoch groß genug, um die Skalierung der Übersetzungstechniken abzuschätzen.

Die Skalierung wird prinzipbedingt umso schlechter, je kleiner das bearbeitete Problem ist. Ein Skalieren ist prinzipiell nur soweit sinnvoll, wie das betrachtete Programm auch Parallelität enthält, das heißt, eine Skalierung auf P Prozessoren kann nur dann eine Beschleunigung des Programms erbringen, wenn es mindestens *eine* Menge von Verbindungen gibt, auf welcher im Programmablauf parallel gearbeitet wird und welche inklusive der Netzreplikate mindestens P Verbindungen enthält, d.h. $\max_G(\sum_{i \in G} |S(G_i)| \cdot n) \geq P$. Ich gebe deshalb Skalierungskurven für das kleinste Problem unter den in dieser Auswertung betrachteten an, das diese Bedingung erfüllt (glass), sowie für einige der größeren (gene, soybean, thyroid). Es handelt sich bei den dargestellten Werten um die Laufzeit pro Epoche im Mittel über einen Lauf des autoprune-Verfahrens bis zum sechsten Beschneidungsschritt. Schon bei diesem Maß der Beschneidung wird der Parallelitätsgrad bereits wieder erheblich reduziert, da nur noch etwa ein Drittel der Verbindungen vorhanden ist, was sich in einer schlechten Skalierung der Laufzeit beim glass-Problem niederschlägt.

Die Abbildungen 11.8 und 11.9 zeigen die Skalierung für die Probleme glass1 und gene1, jeweils gemessen für die Pivot-Architektur mit dem autoprune-Verfahren. Die mit „ideal" bezeichnete Gerade ist in den Bildern jeweils die lineare Extrapolation der gemessenen Leistung für 1k Prozessoren auf höhere

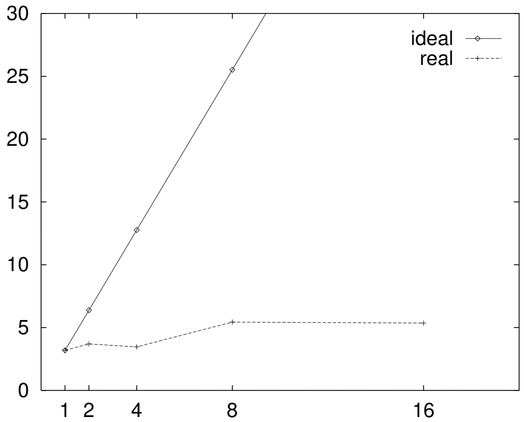


Abbildung 11.8: Ideales und reales Skalierungsverhalten von autoprune mit Pivot-Architektur für glass1 zwischen 1k und 16k Prozessoren (x-Achse), gemessen in Epochen pro Sekunde (y-Achse).

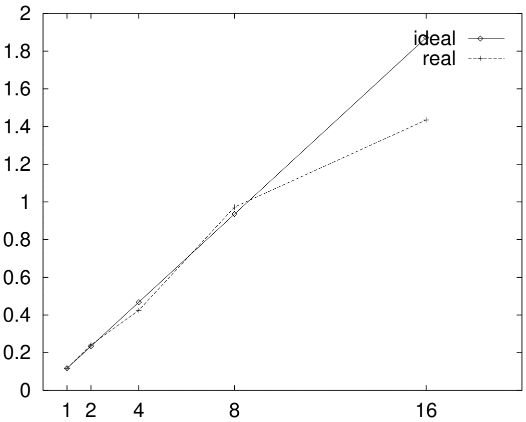


Abbildung 11.9: Dito, für gene1.

Prozessorzahlen; die mit „real" bezeichnete stellt die tatsächliche gemessenen Werte dar. Wie wir sehen, ist die Skalierung beim glass-Beispiel schlecht; die Effizienz beträgt bei 16k Prozessoren nur 11%, bei 8k Prozessoren 21%. Der schlechte Wert kommt zustande, weil das glass-Problem einfach zu klein ist: Der überhaupt maximal vorkommende Parallelitätsgrad in seinem 10-16-8-6 Netz ist theoretisch 32100 (bei Verwendung von einem Netzreplikat für jedes Trainingsbeispiel) für die Vorwärtspropagierung aus den Eingangsknoten bei noch vollverbundenem Netz. Die theoretischen Parallelitätsgrade für die übrigen Verbindungsoperationen betragen der Reihe nach 23968 und 5156 bei der Vorwärtspropagierung und 21828, 22256, 17120 beim Rückwärtsdurchlauf. Diese Werte sind jedoch aus zweierlei Gründen weitaus zu optimistisch: Erstens wird durch die Beschneidung die Parallelität während des Programmablaufs nach und nach auf ungefähr ein Drittel dieser Werte abgesenkt und zweitens sind die Werte für eine Replikatanzahl von 107 gültig. In Wirklichkeit wird das Netz jedoch entweder mit 64 Replikaten implementiert, was einen geringeren Parallelitätsgrad zur Folge hat, oder mit 128 Replikaten, was eine unvollständige Auslastung der Replikate zur Folge hat. Des weiteren kommen ja zwischen den Verbindungsoperationen Phasen mit wesentlich kleinerem Parallelitätsgrad vor, z.B. die Reduktionen und die Knotenoperationen. In Anbetracht dieser geringen Problemgröße ist die mangelhafte Skalierung nicht verwunderlich und kann nicht dem CuPit-Übersetzer angelastet werden. Das gene-Beispiel skaliert hingegen sehr gut. Bis 8k Prozessoren ist die Skalierung perfekt, die Effizienz bei 16k Prozessoren beträgt 77%.

In den Abbildungen 11.10 und 11.11 sehen wir die Skalierung für die Probleme soybean1 und thyroid1, ebenfalls gemessen für die Pivot-Architektur mit dem autoprune-Verfahren. Auch für diese Probleme ist die Skalierung so weit gut, wie die verfügbare Parallelität des Problems reicht. Bei thyroid wird ein eher kleines Netz mit zahlreichen Beispielen verwendet, was zu hohen Replikatanzahlen führt, während bei soybean ein relativ großes Netz mit nur wenigen Replikaten benutzt wird (siehe Tabelle 11.15 auf Seite 206). In beiden Fällen können die vorgestellten Übersetzungstechniken also offenbar die Datenverteilung so auslegen, daß eine gute Ausnutzung der Maschine hergestellt wird. Die erreichte Effizienz bei 16k Prozessoren beträgt für soybean 36% und für thyroid 44%. Für beide Probleme (vor allem für thyroid) wird eine bessere Skalierung vor allem von der ineffizienten MERGE-Prozedur verhindert (siehe Tabelle 11.5 auf Seite 199), deren Einfluß durch die erhöhten Replikatanzahlen größer wird; bei soybean liegt durch die nur 8 Knoten kleine zweite verborgene Schicht zusätzlich ein probleminhärenter Engpaß im Parallelitätsgrad vor, der eine noch bessere Skalierung verhindert.

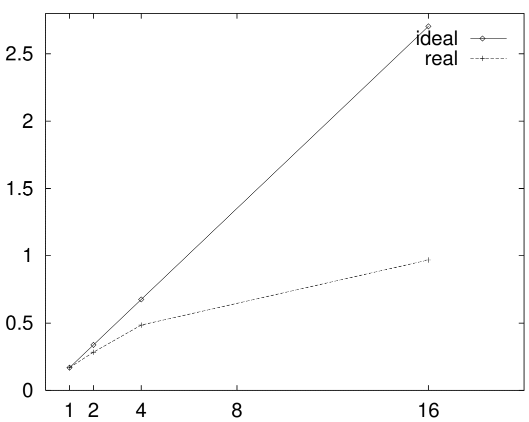


Abbildung 11.10: Dito, für soybean1.

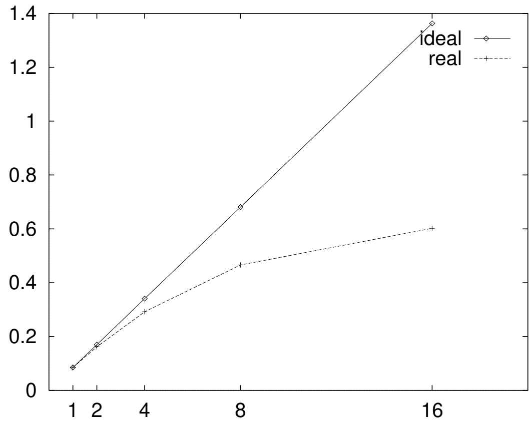


Abbildung 11.11: Dito, für thyroid1.

11.10 Automatische Wahl der Replikatanzahl

Zur Bewertung der Verbesserungen durch die automatische Wahl der Replikatanzahl ziehe ich als Vergleichsmaßstab die Laufzeiten heran, die sich ergeben, wenn eine feste Replikatanzahl verwendet wird, die ein mit Programmen, Datensätzen und Zielmaschine gut vertrauter Benutzer als optimal eingeschätzt hat. Dieser Benutzer bin aus naheliegenden Gründen ich selbst. Der Leser darf meiner Objektivität in dieser Sache durchaus vertrauen: Ich habe die Optimalität der verwendeten Replikatanzahlen zunächst so wenig in Zweifel gezogen, daß ich über mehrere Serien von Versuchen hinweg mit ihnen erheblich Rechenzeit verschwendet habe, bevor mir auffiel, daß andere Replikatanzahlen bessere Laufzeiten ergeben und ich daraufhin die automatische Optimierung entworfen und implementiert habe.

Die vermeintlich optimalen Replikatanzahlen ergeben sich aus der (unzutreffenden) Überlegung, daß höhere Replikatanzahlen, soweit sie überhaupt möglich sind, stets eine Laufzeitverbesserung bewirken, weil die Erhöhung des Parallelitätsgrades aufgrund kleinerer Maschinensegmente zu einer so starken Verringerung der Kommunikation führt, daß dies durch die gleichzeitige Erhöhung des Verwaltungsaufwands nicht wieder ausgeglichen wird.

Die höchstmöglichen Replikatanzahlen ergeben sich aus zwei Einschränkungen. Erstens ist es sinnlos, wesentlich mehr Replikate zu benutzen als Trainingsbeispiele vorhanden sind. Als Replikatanzahl kommt daher maximal die nächstgrößere Zweierpotenz in Frage. Zweitens erlaubt der Übersetzer nur maximal einen Knoten pro Prozessor als Größe einer Knotengruppe, und deshalb muß das Produkt aus maximaler Gruppengröße und Replikatanzahl kleiner als 16384 sein. Aus diesen Regeln ergeben sich für die verschiedenen Probleme Replikatanzahlen zwischen 128 und 512, die als Vergleichsmaßstab herangezogen werden. Die Replikatanzahlen sollten für alle Versuche gleichermaßen gültig sein. Deshalb müssen sie in jedem Fall 32 verborgene Knoten erlauben, und es kommen auch für die Probleme building und thyroid keine größeren Zahlen als 512 vor.

Abbildung 11.12 veranschaulicht die Arbeitsweise der automatischen Wahl der Replikatanzahl. Wie wir sehen, stellt das Verfahren nach kurzen Suchphasen jeweils eine geeignete Replikatanzahl ein, die dann längere Zeit erhalten bleibt, es sei denn, es ergeben sich neue Verhältnisse aufgrund einer Beschneidung. In Abständen von 250 Epochen werden außerdem auch dann versuchsweise Änderungen der Replikatanzahl vorgenommen, wenn keine Beschneidung oder sonstige Verhaltensänderung vorliegt. In Abbildung 11.13 ist der Einschwingvorgang der Replikatanzahlwahl dargestellt. Deutlich sieht man, wie ausgehend von der vom Benutzer vorgeschlagenen zu hohen Replikatanzahl (2^7) kleinere Anzahlen untersucht werden, bis sich eine Verschlechterung einstellt. An diesem Punkt wird die zuvor benutzte Anzahl wiederhergestellt und dann beibehalten. In Abbildung 11.14 sehen wir die Umgebung des Zeitpunktes, an dem die erste Beschneidung erfolgt. Vor der Beschneidung ist die optimale

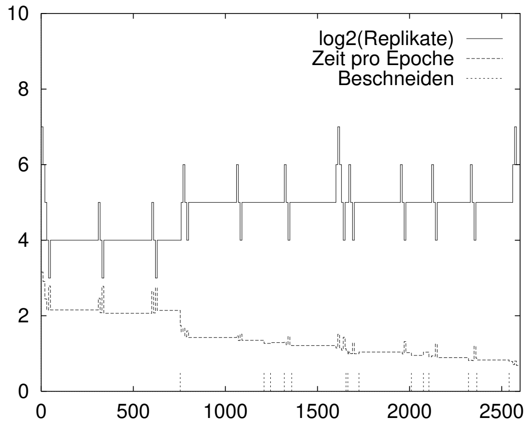


Abbildung 11.12: Automatische Wahl der Replikatanzahl beim Lernverfahren autoprune auf dem soybean1-Problem. Obere Kurve: Zweierlogarithmus der Replikatanzahl. Untere Kurve: Resultierende Laufzeit pro Epoche (in einer für günstigste Darstellung skalierten Zeiteinheit). Die senkrechten Strichlein geben die Zeitpunkte an, zu denen eine Beschneidung stattfand.

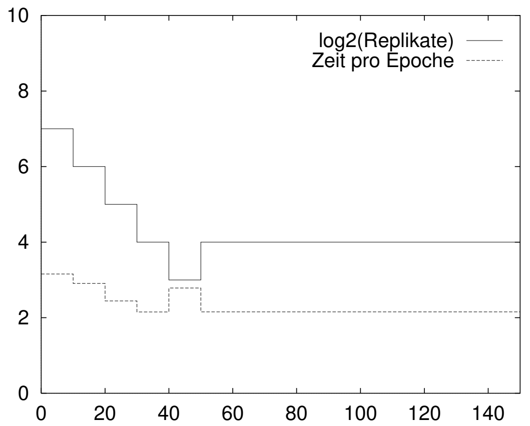


Abbildung 11.13: Ausschnittvergrößerung aus Abbildung 11.12 für die Epochen 0 bis 150. Dies ist der Einschwingvorgang der Replikatanzahlwahl.

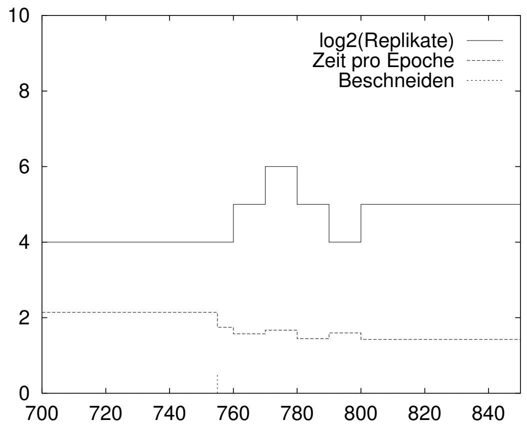


Abbildung 11.14: Dito, für die Epochen 700 bis 850. Dies ist der Übergang von 16 zu 32 Replikaten als bestem Wert, ausgelöst durch die erste Beschneidung (Epoche 755).

Replikatanzahl 16, nach der Beschneidung ist sie 32. Das Verfahren findet die neue optimale Anzahl in vier Suchschritten heraus und stellt sie als neuen Standardwert ein.

Doch nun zu einer systematischen Bewertung der automatischen Wahl der Replikatanzahl: Ich vergleiche für jedes Problem die Laufzeit von autoprune in der Version mit automatisch optimierter und mit fester Replikatanzahl. Die Ergebnisse sind in Tabelle 11.15 dargestellt. Wie wir sehen, ist der Effekt automatischer Replikatanzahlwahl von Problem zu Problem recht verschieden. Während sich bei manchen Problemen fast gar kein Effekt einstellt (building, cancer, glass, heartac), sind bei anderen moderate Verbesserungen zu beobachten (heart, hearta, heartc, thyroid) und bei wieder anderen erhebliche Verbesserungen von 20% und mehr (card, diabetes, flare, soybean). Bei gene ergibt sich eine Verschlechterung der Laufzeit, weil bei diesem Problem die bei der automatischen Anpassung auftretenden Phasen mit ungünstigen Replikatanzahlen so hohe Zeitverluste verursachen, daß diese von den Gewinnen in den übrigen Phasen nicht mehr ausgeglichen werden können. Dies liegt an der extrem ungleichförmigen 121-4-2-3 Netzstruktur dieses Problems. Das Anpassungsverfahren müßte (und könnte) für solche Fälle noch verfeinert werden, indem man es sehr ungünstige Phasen frühzeitig abbrechen läßt.

Man beachte, daß sich auch bei *optimaler* statischer Wahl der Replikatanzahl noch eine Verbesserung durch automatische Anpassung ergibt. Da sich bei konstruktiven Lernverfahren die Laufzeitverhältnisse während des Programmablaufs verändern, ist nämlich eine *dynamische* Anpassung der Replikatanzahl nötig. Dies tritt noch stärker als bei Beschneidungsverfahren bei den Kandidatentrainingsverfahren aus Kapitel 5 in Erscheinung, bei denen während einer Kandidatentrainingsphase eine

Problem	$R_{statisch}$	$R_{dynamisch}$	t_{rel}
building	512	128-512	99%
cancer	512	256-512	98%
card	256	16-256	120%
diabetes	512	64-128	161%
flare	512	16-256	165%
gene	128	32-128	77%
glass	128	32-128	102%
heart	256	32-256	110%
hearta	256	32-256	110%
heartac	256	128-256	97%
heartc	256	16-256	115%
soybean	128	8-64	130%
thyroid	512	128-256	114%
(Mittel)			115%

Tabelle 11.15: Relative Laufzeit t_{rel} von autoprune mit statischer Replikatanzahl $R_{statisch}$ im Vergleich zu autoprune mit dynamisch optimierter Wahl der Replikatanzahl (mit resultierenden Replikatanzahlen im Bereich $R_{dynamisch}$ nach dem anfänglichen Einschwingvorgang).

andere Replikatanzahl sinnvoll sein kann als während der Ausgabetrainingsphase und dies für jede Kandidatentrainingsphase unterschiedlich. Dieser Effekt tritt je nach Problemgröße und Parametern des Lernverfahrens in unterschiedlicher Art und Stärke in Erscheinung; er wurde aber nicht quantitativ untersucht.

Kapitel 12

Konklusion: Parallele Übersetzung

The whole problem with the world is
that fools and fanatics are always
so certain of themselves,
but wiser people so full of doubts.
Bertrand Russell

12.1 Zusammenfassung und Beiträge dieser Arbeit

Diese Arbeit ist eine der ersten, die sich mit der Optimierung von parallelen Hochsprachenprogrammen für dynamisch veränderliche unregelmäßige Probleme befaßt. Es wird eine spezifische Problemklasse behandelt, nämlich konstruktive neuronale Lernverfahren.

Das Ziel, solche Programme auf Parallelrechnern effizient auszuführen, wird in zwei Schritten erreicht: Zunächst habe ich ein Programmiermodell definiert, das die Eigenschaften, welche für konstruktive neuronale Algorithmen typisch sind, direkt widerspiegelt (Kapitel 9). Alsdann habe ich auf Basis dieses Programmiermodells Programmoptimierungen beschrieben, mit deren Hilfe effizienter Code erzeugt werden kann (Kapitel 10).

Diese Schritte wurden nicht nur theoretisch vollzogen: Das Programmiermodell wurde als konkrete Programmiersprache CuPit ausformuliert und es wurde ein CuPit-Übersetzer für die MasPar MP-1/MP-2 implementiert, der die beschriebenen Optimierungen realisiert. Die Laufzeitverbesserungen, die sich aus diesen Optimierungen ergeben, wurden in Meßreihen für eine Anzahl verschiedener Probleme ermittelt (Kapitel 11). Wie ich im Text jeweils grob dargelegt habe, lassen sich die Sprachkonstrukte, die in CuPit verwendet werden auch in andere Programmiersprachen einbetten (nämlich in parallele objektorientierte Sprachen), und die Optimierungstechniken können auf andere parallele Rechnerarchitekturen übertragen werden.

Die Beiträge dieser Arbeit liegen zum ersten in der Identifikation und Beschreibung des Programmiermodells und seiner Umsetzung in eine konkrete Form, zum zweiten in der Beschreibung der Optimierungstechniken und zum dritten im Nachweis von deren Realisierbarkeit und Nützlichkeit.

Kernstück der Optimierungen ist eine Klasse von Daten- und Prozeßverteilungen, die zugleich Lastbalancierung als auch die fast optimale Wahrung von Datenlokalität erlaubt. Diese Daten- und Prozeßverteilung ist die Grundlage weiterer Optimierungen, zum Beispiel der Kommunikationsbündelung beim Zugriff auf nichtlokale Verbindungsobjekte, der Wahl der Verbindungsallokation und der dynamischen Wahl der Anzahl verwendeter Netzreplikate.

Die Auswertung der einzelnen Optimierungsaspekte ergab für die im ersten Teil der Arbeit vorgestellten Anwendungsprobleme im Mittel folgende Verbesserungen der Rechenleistung (Kapitel 11):

1. Die Lastbalancierung beschleunigt den Programmablauf für nur moderat unregelmäßige Netze um 28%.

2. Die Datenlokalität beschleunigt im Vergleich zu einer auf Feldzugriffen basierenden Implementation den Ablauf um etwa 195%.

3. Die Kommunikationsbündelung beschleunigt gegenüber der einzelnen Übertragung benötigter Komponenten um 10% und gegenüber der Übertragung stets ganzer Verbindungsobjekte um 28%.

4. Die richtige Wahl der Verbindungsallokation beschleunigt gegenüber der falschen Wahl im Mittel um 50%.

5. Die dynamische, automatische Wahl der Anzahl von Netzreplikaten beschleunigt den Programmablauf gegenüber einer plausiblen, aber nicht optimalen statischen Wahl um 15%, obwohl das implementierte Verfahren noch Schwächen aufweist.

Diese konkreten Werte sind natürlich maschinenabhängig und würden sicherlich auf einer anderen Rechnerarchitektur abweichen. Selbst für den Fall, daß die Ergebnisse durchgehend etwas schlechter ausfallen würden, machen jedoch die angegebenen Resultate die allgemeine Nützlichkeit der Optimierungen deutlich. Eine sehr grobe Zusammenfassung dieser Resultate und einen Vergleich mit alternativen Implementationsmöglichkeiten zeigt die Abbildung 12.1: Der oberste Balken zeigt die Leistung des

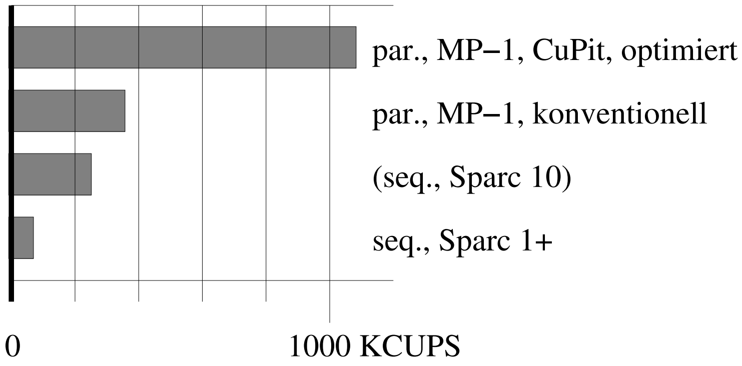


Abbildung 12.1: Stark vergröberter Leistungsvergleich

CuPit-Übersetzers samt aller Optimierungen auf dem soybean-Problem für die MasPar MP-1 (gemäß Abschnitt 11.3.1). Der zweite Balken zeigt die entsprechende Leistung einer hypothetischen parallelen Implementierung auf derselben Maschine bei Verwendung herkömmlicher, feldbasierter Sprachen und Übersetzer, die insbesondere bei konstruktiven Algorithmen keine Datenlokalität mehr herstellen können (gemäß Abschnitt 11.5.1). Der dritte Balken zeigt, gewissermaßen außer Konkurrenz, die Leistung desselben Programms, das mit dem in dieser Arbeit nicht behandelten sequentiellen CuPit-Übersetzer übersetzt wurde, für eine moderne Workstation vom Typ Sun SparcStation 10. Der unterste Balken zeigt die Leistung auf einer anderen Workstation, die der gleichen Hardwaregeneration wie die MP-1 angehört, nämlich einer Sun SparcStation 1+ (gemäß Abschnitt 11.3.1).

Die vorliegende Arbeit hat nachgewiesen, daß die Verwendung anwendungsbereichsspezifischer Optimierungstechniken in einem Übersetzer die effiziente Behandlung dynamischer unregelmäßiger Probleme ermöglichen kann, und stellt erstmals eine Sammlung solcher Techniken vor. Trotz des Erfolgs der Optimierungen ist allerdings die im Vergleich zur Spitzenleistung erzielte Rechenleistung noch so wenig überzeugend, daß sich auf den heutigen Parallelrechnern die Behandlung unregelmäßiger Probleme kaum empfiehlt.

12.2 Ausblick

12.2.1 Lokal

Es sind noch eine Reihe von Verbesserungen der in dieser Arbeit dargestellten Techniken möglich, um für spezielle Fälle eine bessere Leistung zu erhalten. Die wichtigsten spreche ich kurz an.

Berücksichtigung von Phasen. Bei Lernverfahren, die mehrere unterschiedlich strukturierte Phasen haben, wie beispielsweise die Kandidaten- und die Ausgangstrainingsphase bei den Kandidatenlernverfahren, kann es unter Umständen sinnvoll sein, Optimierungsentscheidungen für jede Phase getrennt zu treffen. Im vorgestellten Übersetzer geschieht dies automatisch bei der Wahl der Replikatanzahl, es wäre aber ebenfalls sinnvoll für die Messung der Lastbalancefaktoren und die Wahl der Verbindungsallokation. Dazu müßte der Übersetzer in der Lage sein, die Phasen als solche zu identifizieren und dann für jede Phase dynamisch die richtigen Entscheidungen zu verwenden. Für die Verbindungsallokation bedeutet dies, daß sie dynamisch veränderlich sein muß, was eine kompliziertere Form des Codes nach sich zieht und zusätzliche Laufzeitkosten verursacht, die gegen die Gewinne abgewägt werden müssen. Die Identifikation der Phasen muß entweder mittels Annotationen vom Programmierer geleistet werden oder sie muß durch heuristische Methoden im eigentlichen Programmlauf oder in vorgelagerten „Probeläufen" erfolgen, deren Resultate dann in eine erneute Übersetzung einfließen.

Verschiebung von Synchronisationspunkten. Die Semantik von CuPit verlangt eine Synchronisation nach jedem parallelitätserhöhenden Aufruf einer Objektprozedur. Eventuell kann die tatsächliche Synchronisation jedoch auf einen späteren Zeitpunkt verschoben werden, ohne die Semantik zu verletzen, weil nach dem parallelen Aufruf Befehle bearbeitet werden müssen, die keine Wechselwirkung mit diesem Aufruf haben. Ein Teil dieser Fälle kann statisch ermittelt werden, andere sind jedoch nur zur Laufzeit festzustellen, weil je nach der der konkret vorhandenen Verbindungsstruktur eine Wechselwirkung bestehen oder fehlen kann. In manchen Fällen kann auf die Synchronisation ganz verzichtet werden, weil sie bis zur nächsten Synchronisation verschoben werden kann. Eine Verschiebung erlaubt es, die Kosten für die Synchronisation zu senken, indem Synchronisation und Rechenarbeit überlappt werden und somit kürzere Wartezeiten anfallen. Da auf SIMD-Maschinen ohnehin keine explizite Synchronisation durchgeführt werden muß, wurde in dieser Arbeit keine Synchronisationsverschiebung realisiert.

Volle Nutzung der Speicherhierarchie. Bei der Betrachtung der Datenlokalität haben wir uns in dieser Arbeit auf die Unterscheidung von lokalem und entferntem Speicher beschränkt. Auf modernen Maschinen verfügt aber jeder Prozessor zusätzlich über einen *Cache*-Speicher, auf den erheblich schneller als auf den normalen lokalen Speicher zugegriffen werden kann. Die effiziente Nutzung des Caches verlangt nach einem nochmals verschärften Maß von Lokalität der Zugriffe, das eventuell mit einer geschickten Umstrukturierung der Ausführungsreihenfolge hergestellt werden kann. Es ist unklar, ob eine entsprechende Transformation eines KNA-Programms von einem Übersetzer geleistet werden kann. Mittels *prefetch*-Mechanismen können ggf. Zugriffe auf den lokalen Speicher in ähnlicher Weise durch Zugriffe auf den Cache ersetzt werden, wie Zugriffe auf entfernten Speicher in lokale Zugriffe umgewandelt werden, nämlich durch eine entsprechende Strukturierung der Verbindungsoperations-Virtualisierungsprozeduren.
Am anderen Ende der Speicherhierarchie kann noch der Hintergrundspeicher angefügt werden, was Fragen nach effizienter paralleler Ein-/Ausgabe und Zusammenarbeit von Übersetzer und Betriebssystem aufwirft, die erheblich über den bisherigen Stand der Forschung hinausgehen. Da die MasPar MP-1 weder Cache noch Hintergrundspeicher aufweist, wurden in dieser Arbeit keine entsprechenden Betrachtungen angestellt.

Wahl der Prozessoranzahl. Bei Problemen, die nicht genügend groß sind, um die parallele Maschine voll auszulasten kann es sinnvoll sein, gar nicht alle Prozessoren zu ihrer Bearbeitung einzusetzen: Das Programm läuft mit weniger Prozessoren schneller ab, weil die Verwaltungsverluste durch Hinzunahme weiterer Prozessoren größer sind, als der Gewinn durch die zusätzliche Parallelität. Ein solcher

Fall ist in Abbildung 11.8 zu sehen. Der Übersetzer sollte solche Fälle feststellen können und automatisch weniger Prozessoren einsetzen. Dies gilt insbesondere, wenn das Betriebssystem erlaubt, die unbenutzen Prozessorressourcen anderen zugleich ablaufenden Programmen verfügbar zu machen. Um diese Optimierung durchführen zu können ist ein gutes und dementsprechend komplexes Kostenmodell für die Ausführungszeit des Programms in Abhängigkeit von der Prozessorzahl notwendig, dessen Auswertung nur wenig Laufzeit in Anspruch nehmen darf, weil andernfalls die Verbesserungen wieder eliminiert würden. Die Realisierung dieser Optimierung ist deshalb schwierig.

12.2.2 Global

In der vorliegenden Arbeit wurden Techniken vorgestellt zur Erzeugung effizienten parallelen Codes für dynamisch veränderliche unregelmäßige Probleme eines bestimmten Anwendungsgebietes, nämlich neuronaler Lernverfahren. Die Techniken basieren darauf, die Ausnutzung von Eigenschaften, die für solche Programme charakteristisch sind, dadurch zu ermöglichen, daß eine spezielle Programmiersprache zugrundegelegt wird, die diese Eigenschaften gut widerspiegelt.

Weitere Arbeiten mit demselben Grundgedanken könnten zwei Ziele verfolgen: Zunächst kann das verwendete Programmiermodell in eine allgemeine parallele Programmiersprache eingebettet werden, um den Programmierern das Erlernen einer speziellen Sprache zu ersparen; hierfür eignen sich objektorientierte Sprachen (siehe Abschnitt 9.5). Ergänzend müssen auch die hier vorgestellten Optimierungstechniken in Übersetzer für diese allgemeine Sprache integriert werden.

Des weiteren sollte versucht werden, auch für andere Anwendungsgebiete mit unregelmäßigen Problemen Optimierungstechniken zu finden, die sich in einem Übersetzer realisieren lassen. In vielen Fällen wird dabei genau wie für neuronale Algorithmen die Technik nützlich sein, Teile der charakteristischen und für die Optimierungen relevanten Semantik der Probleme und Algorithmen in der Programmiersprache explizit zu modellieren. Anzustreben ist eine Integration vieler solcher Sammlungen von Optimierungstechniken für verschiedene Anwendungsbereiche mit unregelmäßigen Problemen im gleichen Übersetzer. Damit können eines Tages vielleicht Hochsprachenprogramme für Parallelrechner auf ähnlich unproblematische Weise in effizienten Code umgesetzt werden, wie es heute für sequentielle Rechner der Fall ist. Erst wenn dieses Ziel weitgehend erreicht ist, wird man davon sprechen können, daß die Parallelrechnerei ihre in Kapitel 1 dargestellten frühen Phasen verlassen hat.

Anhang A

Verfügbarkeit der Daten und Programme

Der überwiegende Teil der im Rahmen dieser Arbeit hergestellten Daten und Programme ist auf elektronischem Wege frei verfügbar:

- Die PROBEN1-Benchmarksammlung ist per anonymem FTP verfügbar vom Neural Bench Archiv der Carnegie Mellon University (Rechner `ftp.cs.cmu.edu`, Verzeichnis `/afs/cs/project/connect/bench/contrib/prechelt`) und von der Maschine `ftp.ira.uka.de` im Verzeichnis `/pub/neuron`. In beiden Fällen lautet der Dateiname `proben1.tar.gz`. Diese Datei enthält die komplette Benchmark-Sammlung einschließlich des technischen Berichts [14]. Die Datei ist etwa 1,8 Megabytes groß und benötigt ausgepackt ungefähr 20 Megabytes. Der technische Bericht ist auch einzeln erhältlich per FTP von `ftp.ira.uka.de` in Verzeichnis `/pub/papers/techreports/1994` als `1994-21.ps.Z`.
- Die Rohdaten der einzelnen Programmläufe aus den Versuchsreihen über Lernverfahren sind im Internet erhältlich per FTP von `ftp.ira.uka.de` in Verzeichnis `/pub/neuron` als `nndata.tar.gz`.
- Das Literate-Programming-Dokument, das den CuPit-Übersetzer für die MasPar MP-1/MP-2 enthält, ist über FTP verfügbar von `ftp.ira.uka.de` in Verzeichnis `/pub/papers/techreports/1995` als `1995-1.ps.Z`.
- Der Quelltext des Übersetzers für die MasPar MP-1/MP-2 zur Verwendung mit dem Eli Übersetzerbausystem Version 3.5 ist per FTP verfügbar von `i41s10.ira.uka.de` im Verzeichnis `/pub/cupit` als `cupit_maspar.tar.gz`.
- Ein Quelltext des Übersetzers für sequentielle Rechner, der C-Quellen enthält und deshalb auch ohne das Eli-System zur Erzeugung eines ablauffähigen Übersetzers (inklusive Bibliothek und Laufzeitsystem) verwendet werden kann, ist per FTP verfügbar von `i41s10.ira.uka.de` im Verzeichnis `/pub/cupit` als `cupit.tar.gz`.

Literaturverzeichnis

[1] Bruno Achauer. The Dowl distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, September 1993.

[2] S. Agmon. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6(3):382–392, 1954.

[3] R.F. Albrecht, C.R. Reeves, and N.C. Steele, editors. *Artificial Neural Nets and Genetic Algorithms*, Innsbruck, Austria, February 1993. Springer Verlag.

[4] J.S. Albus. *Brains, Behavior, and Robotics*. Byte Books, Peterborough, NH, 1981.

[5] James D. Allen and Dave E. Schimmel. Efficient neural networks on SIMD machines. Preprint. School of Electrical Engineering, Georgia Institute of Technology, Atlanta, 1993.

[6] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Series in Computer Science and Engineering. Benjamin/Cummings, Redwood City, CA, 1989.

[7] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Series in Computer Science and Engineering. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[8] Nikolaus Almássy. Ein Compiler für CONDELA-III. Master's thesis, Institut für praktische Informatik, TU Wien, February 1990.

[9] Ethem Alpaydin. Grow-and-learn: An incremental method for category learning. In *Proc. Int. Neural Network Conf., vol. 2*, pages 761–764, Paris, 1990.

[10] Saman Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory maschines. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 126–140, Albuquerque, NM, June 1993.

[11] J. Andersen, G. Mitra, and D. Parkinson. The scheduling of sparse matrix-vector multiplication on a massively parallel DAP computer. *Parallel Computing*, 18:675–697, 1992.

[12] Edward Anderson and Youcef Saad. Solving sparse triangular systems on parallel computers. *Int. Journal of High-Speed Computing*, 1(1):73–95, 1989.

[13] J.A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, 1988.

[14] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 112–125, Albuquerque, NM, June 1993.

[15] Richard Anderson, Ernst W. Mayr, and Manfred Warmuth. Parallel approximation algorithms for bin packing. Technical Report STAN-CS-88-1200, Department of Computer Science, Stanford University, March 1988.

[16] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1994.

[17] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H.T. Kung, Monica S. Lam, Ornat Mezilcioglu, and Jon A. Webb. Warp architecture and implementation. In *Proc. 13th Annual Int. Symposium on Computer Architecture*, pages 346–356. Computer Science Press, 1986.

[18] Martin Anthony. Probabilistic analysis of learning in artificial neural networks: The PAC model and its variants. Technical Report NC-TR-94-3, Department of Mathematics, London School of Economics and Political Science, London, UK, June 1994.

[19] Timur Ash. Dynamic node creation in backpropagation networks. *Connection Science*, 1(4):365–375, 1989.

[20] William F. Aspray and Arthur W. Burks, editors. *Papers of John von Neumann on Computing and Computer Theory.* MIT Press, Cambridge, MA, 1987.

[21] Les Atlas, Ronald Cole, Jerome Connor, Mohamed El-Sharkawi, Robert J. Marks, Veshwant Muthusamy, and Etienne Barnard. Performance comparisons between backpropagation networks and classification trees on three real-world applications. In *[361]*, pages 622–629, 1990.

[22] M.E. Azema-Barac and A.N. Refenes. Neural network implementations and speedup on massively parallel machines. In *EuroMicro*, Paris, September 1992.

[23] Paul T. Baffes and John M. Zelle. Growing layers of perceptrons: Introducing the Extentron algorithm. In *Proc. Int. Joint Conf. on Neural Networks 1992, vol. 2*, pages 392–397, Baltimore, 1992.

[24] David Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, 4(8):54–55, August 1991. Also as Technical Report RNR-91-020, Nasa Ames Research Center.

[25] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS parallel benchmarks — summary and preliminary results. In *Supercomputing*, pages 158–165, Albuquerque, NM, November 1991. IEEE Computer Society Press.

[26] Pierre Baldi and Yves Chauvin. Temporal evolution of generalization during learning in linear networks. *Neural Computation*, 3:589–603, 1991.

[27] M. R. Banan and K. D. Hjelmstad. Self-organization of architecture by simulated hierarchical adaptive random partitioning. In *Proc. Int. Joint Conf. on Neural Networks 1992, vol. 3*, pages 823–828, Baltimore, 1992.

[28] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[29] Etienne Barnard. Optimization for training neural nets. *IEEE Transactions on Neural Networks*, 3(2):232–240, March 1992.

[30] Eric B. Bartlett. Dynamic node architecture learning: An information theoretic approach. *Neural Networks*, 7(1):129–140, 1994.

[31] Andrew G. Barto and P. Anandan. Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(3):360–375, May 1985.

[32] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):834–846, September 1983.

[33] K. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, 29(9):836–840, 1980.

[34] Roberto Battiti and Anna Maria Colla. Democracy in neural nets: Voting schemes for classification. *Neural Networks*, 7(4):691–707, 1994.

[35] Eric B. Baum and David Haussler. What size net gives valid generalization. *Neural Computation*, 1:151–160, 1989.

[36] Eric B. Baum and Kevin J. Lang. Constructing hidden units using examples and queries. In *[221]*, pages 904–910, 1991.

[37] J. Beetem, M. Denneau, and D. Weingarten. GF11 — a supercomputer for scientific applications. In *Proc. 12th Int. Symposium on Computer Architecture*. IEEE Computer Society, 1986.

[38] Gordon Bell. Scalable parallel computers: Alternatives, issues, and challenges. *Int. J. of Parallel Programming*, 22(1):3–46, 1994.

[39] I. Bellido and E. Fiesler. Do backpropagation trained neural networks have normal weight distributions? In *Int. Conf. on Artificial Neural Networks (ICANN)*. European Neural Network Society, 1993. (ftp://maya.idiap.ch/pub/papers/neural/bellido.w-distribution.ps.Z).

[40] Ignacio Bellido and Gregorio Fernández. Backpropagation growing networks: Towards local minima elimination. In A. Prieto, editor, *Artificial neural networks, Proc. IWANN*, pages 130–135, 1991.

[41] Pierre Bessière, Ali Chams, and Philippe Chol. MENTAL: A virtual machine approach to artificial neural networks programming. Final Report ESPRIT B.R.A Project 3049, LGI–IMAG, Grenoble, June 1991.

[42] Griff Bilbro, Reinhold Mann, Thomas K. Miller, Wesley E. Snyder, David E. Van den Bout, and Mark White. Optimization by mean field annealing. In *[360]*, pages 91–98, 1989.

[43] Griff L. Bilbro and Wesley E. Snyder. Range image restoration using Mean Field Annealing. In *[360]*, pages 594–601, 1989.

[44] Tom Blank. The MasPar MP-1 architecture. In *Proc. of the COMPCON Spring 1990 — The 35th IEEE Computer Society Int. Conf.*, pages 20–24, San Francisco, CA, 1990.

[45] Tom Blank and John R. Nickolls. A Grimm collection of MIMD fairy tales. In H.J. Siegel, editor, *Frontiers of Massively Parallel Computation*, pages 448–457, McLean, Virginia, October 1992. IEEE Computer Society Press.

[46] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a nested data-parallel language. *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM SIGPLAN Notices*, 28(7):102–111, July 1993.

[47] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. Segmented operations for sparse matrix computations on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1993.

[48] Avrim Blum and Ronald L. Rivest. Training a 3–node neural network is NP–complete. In *[360]*, pages 494–501, 1989.

[49] Ulrich Bodenhausen and Alex Waibel. Application oriented automatic structuring of time-delay neural networks for high performance character and speech recognition. In *Proc. Int. Joint Conf. on Neural Networks*, San Francisco, 1993.

[50] Ulrich Bodenhausen and Alex Waibel. Tuning by doing: Flexibility through automatic structure optimization. In *Eurospeech*, September 1993.

[51] Yuri P. Boglaev. Exact dynamic load balancing of MIMD architectures with linear programming algorithms. *Parallel Computing*, 18:615–623, 1992.

[52] Brian V. Bonnlander and Michael C. Mozer. Metamorphosis networks: An alternative to constructive methods. In *[143]*, pages 131–138, 1992.

[53] Léon Bottou and Patrick Gallinari. A framework for the cooperation of learning algorithms. In *[221]*, pages 781–788, 1991.

[54] Joe Brandenburg. Technology advances in the Intel Paragon system. In *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, page 182, Velen, Germany, June 1993. ACM SIGACT/SIGARCH, EATCS, ACM Press.

[55] Heinz Braun and Joachim Weisbrod. Evolving neural feedforward networks. In *[3]*, pages 25–32, 1993.

[56] J. Brehm, A. Böhm, and Jens Volkert. Sparse matrix algorithms for SUPRENUM. In *CONPAR 90 — VAPP IV, Joint Int. Conf. on Vector and Parallel Processing*, pages 120–130, Zürich, Switzerland, September 1990.

[57] Richard P. Brent. Fast training algorithms for multilayer neural nets. *IEEE Transactions on Neural Networks*, 2(3):346–354, 1991.

[58] J. Bridle. Probabilistic interpretation of feedforward classification network outputs with relationships to statistical pattern recognition. In F. Fogelman-Soulie and J. Hérault, editors, *Neurocomputing: Algorithms, Architectures, and Applications*. Springer Verlag, New York, 1989.

[59] Garnett W. Bryant. Training self-configuring backpropagation networks. In *Proc. Int. Joint Conf. on Neural Networks 1992, vol. 1*, pages 365–370, Baltimore, 1992.

[60] A.E. Bryson and Y.C. Ho. *Applied Optimal Control*. Blaisdell, New York, 1969.

[61] Alice R. Burks and Arthur W. Burks. The ENIAC: The first general purpose electronic computer. *Annals of the History of Computing*, 3(4):310–399, October 1981.

[62] Alice R. Burks and Arthur W. Burks. *The First Electronic Computer: The Atanasoff Story*. University of Michgan Press, Ann Arbor, 1988.

[63] Ralph Butler and Ewing Lusk. *User's Guide to the p4 Parallel Programming System*. Argonne National Laboratory, 1992.

[64] Christian Cachin. Pedagogical pattern selection strategies. *Neural Networks*, 7(1):175–181, 1994.

[65] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, March 1994.

[66] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[67] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Technical report, Digital Systems Research Center, Lytton Ave., Palo Alto, CA, August 1988.

[68] Alan Carle, Ken Kennedy, Ulrich Kremer, and John Mellor-Crummey. Automatic data layout for distributed-memory machines in the Fortran D programming environment. In *AP '93, Int. Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution, and Automatic Parallel Performance Prediction*, pages 108–123, Saarbrücken, Germany, March 1993.

[69] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[70] H.J. Caulfield. Parallel n^4 weighted optical interconnections. *Applied Optics*, 26:4039, 1987.

[71] L.W. Chan and F. Fallside. An adaptive training algorithm for backpropagation networks. *Computer, Speech, and Language*, 2:205–218, 1987.

[72] Yao-Jen Chang, Jean-Lien C. Wu, and Jingshown Wu. Scheduling parallel programs with non-uniform parallelism profiles. In *Supercomputing*, pages 502–511, Albuquerque, NM, November 1991. IEEE Computer Society Press.

[73] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Vienna Fortran — a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. Elsevier Science Publishers, North-Holland, 1990.

[74] Barbara Chapman, Piyush Mehrotra, and Hans Zima. User-defined mappings in Vienna Fortran. In *Proc. Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, pages 72–75, Boulder, CO, September 1992. (ACM SIGPLAN Notices 28(1), January 1993).

[75] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and access inference for data-parallel programs. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages

130–144, Toronto, Canada, June 1991. ACM Press.

[76] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data parallel programs. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 16–28, Charleston, SC, January 1993.

[77] Yves Chauvin. A back-propagation algorithm with optimal use of hidden units. In *[360]*, pages 519–526, 1989.

[78] Tzung-Shi Chen and Jang-Ping Sheu. Communication-free data allocation techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):924ff, September 1994.

[79] P. Churoux, J.P. Bouzinac, M. Fraces, S. Kocon, D. Comte, N. Hifidi, and J.Y. Rousselot. Massively parallel optoelectronic computer: The Oedipe project. In A.M. Goncharenko, F.V. Karpushko, G.V. Sinitsyn, and S.P. Apanasevich, editors, *ICO Topical Meeting on Optical Computing*, pages 470–475, Minsk, Belarus, June 1992. SPIE Volume 1806.

[80] Pierre Courrieu. A convergent generator of neural networks. *Neural Networks*, 6:835–844, 1993.

[81] Jack D. Cowan, Gerald Tesauro, and J. Alspector, editors. *Advances in Neural Information Processing Systems 6*, San Mateo, CA, 1994. Morgan Kaufman Publishers Inc.

[82] Ian F. Croall and John P. Mason, editors. *Industrial Applications of Neural Networks*, volume Project 2092 ANNIE, Vol.1 of *Research Reports ESPRIT*. Springer Verlag, 1992.

[83] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Eric Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, San Diego, CA, May 1993. ACM Sigplan Notices 28(7).

[84] Yann Le Cun. Une procédure d'apprentissage pour réseau à seuil assymmétrique. In *Cognitiva: A la Frontière de l'Intelligence Artificielle des Sciences de la Connaissance des Neurosciences*, pages 599–604, Paris, France, 1985. CESTA.

[85] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *[361]*, pages 598–605, 1990.

[86] Antonio d'Acierno and Roberto Vaccaro. On parallelizing recursive neural networks on coarse-grained parallel computers: A general algorithm. *Parallel Computing*, 20:245–256, 1994.

[87] Christian Darken, Joseph Chang, and John Moody. Learning rate schedules for faster stochastic gradient search. In *Proc. Neural Networks for Signal Processing 2*. IEEE Press, August 1992. Shorter version in [254].

[88] Christian Darken and John Moody. Note on learning rate schedules for stochastic optimization. In *[221]*, pages 832–838, 1991.

[89] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):814ff, August 1994.

[90] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 1994.

[91] L. Davis, editor. *Handbook of Genetic Algorithms*. van Nostrand Reinhold, New York, 1991.

[92] Guillaume Deffuant. Neural units recruitment algorithm for generation of decision trees. In *Proc. Int. Joint Conf. on Neural Networks, vol. 1*, pages 637–642, 1990.

[93] David DeMers and Garrison Cottrell. Nonlinear dimensionality reduction. In *[143]*, pages 580–587, 1993.

[94] Etienne Deprit. Implementing recurrent back-propagation on the Connection Machine. *Neural Networks*, 2:295–314, 1989.

[95] T.G. Dietterich and G. Bakiri. Error-correcting output codes: A general method for improving multiclass inductive learning programs. In *Proc. of the 9th National Conf. of Artificial Intelligence (AAAI)*, pages 572–577, Anaheim, CA, 1991. AAAI Press.

[96] Thomas G. Dietterich. Limitations on induction. In *Proc. of the 6th Int. Workshop on Machine Learning*, pages 124–128, Ithaca, NY, June 1989. Morgan Kaufmann, San Mateo, CA.

[97] R.M. Dudley. Central limit theorems for empirical measures. *Annals of Probability*, 6(6):899–929, 1978.

[98] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 68–82, Albuquerque, NM, June 1993.

[99] *European Seminar on Neural Computing*, London, February 1988. IBC Technical Services Ltd.

[100] F.A. Brockhaus Lexikon Redaktion, editor. *dtv Brockhaus Lexikon*. F.A. Brockhaus und Deutscher Taschenbuch Verlag, Mannheim und München, 1982, 1989.

[101] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1988.

[102] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation learning architecture. Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1990.

[103] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation learning architecture. In *[361]*, pages 524–532, 1990.

[104] Gerald Fahner and Rolf Eckmiller. Structural adaption of parsimonious higher-order neural classifiers. *Neural Networks*, 7(2):279–289, 1994.

[105] R.M. Farber. Efficiently modeling neural networks on massively parallel computers. In *Proc. of the NASA Workshop on Parallel Computing*. Los Alamos Technical Report LA-UR-92-3568, November 1991.

[106] Rod A. Fatoohi. Performance comparisons of several SIMD machines. In *Proc. 5th SIAM Conf. on Parallel Processing for Scientific Computing*, pages 419–424, Houston, TX, March 1991.

[107] Usama M. Fayyad and Keki B. Irani. On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, 8:87–102, 1992.

[108] Edward A. Feigenbaum and Pamela McCorduck. *The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World*. Addison-Wesley, Reading, MA, 1983.

[109] Edward A. Feigenbaum, Pamela McCorduck, and H. Penny Nii. *The Rise of the Expert Company*. Times Books, Random House, New York, 1988.

[110] Jerome A. Feldman, Mark A. Fanty, Nigel H. Goddard, and Kenton J. Lynne. Computing with structured connectionist networks. In *[419]*, chapter 23, pages 434–454. 1990.

[111] M.R. Feldman, C.C. Guest T.J. Drabnik, and S.C. Esener. Comparison between electrical and free-space optical interconnects based on interconnect density capabilities. *Applied Optics*, 28(18):3820, September 1989.

[112] M.R. Feldman, S.C. Esener, C.C. Guest, and S.H. Lee. Comparison between optical and electrical interconnects based on power and speed considerations. *Applied Optics*, 27(9):1742–1751, May 1988.

[113] Yosee Feldman and Ehud Shapiro. Spatial machines: A more realistic approach to parallel computation. *Communications of the ACM*, 35(10):61–73, October 1992.

[114] E. Fiesler. Minimal and high order neural network topologies. In *Proc. of the 1993 Int. Simulation Technology Conf., San Diego (SimTec 93)*. Society for Computer Simulation (SCS), 1993.

[115] Michael Finke and Klaus-Robert Müller. Estimating a-posteriori probabilities using stochastic network models. In M.C. Mozer, P. Smolensky, D.S. Touretzky, J.L. Elman, and A.S Weigend, editors, *Proc. of the 1993 Connectionist Models Summer School*, Hillsdale, NJ, 1994. Erlbaum Associates.

[116] William Finnoff, Ferdinand Hergert, and Hans Georg Zimmermann. A comparison of weight elimination methods for reducing complexity in neural networks. In *Proc. Int. Joint Conf. on Neural Networks, vol. 3*, pages 980–987, Baltimore, 1992.

[117] William Finnoff, Ferdinand Hergert, and Hans Georg Zimmermann. Improving model selection by nonconvergent methods. *Neural Networks*, 6:771–783, 1993.

[118] William Finnoff and Hans Georg Zimmermann. Detecting structure in small datasets by network fitting under complexity constraints. In *Proc. Second Annual Workshop on Computational Learning Theory and Natural Learning Systems*, Berkeley, CA, 1991.

[119] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[120] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring — a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[121] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proc. Tenth ACM Symposium on the Theory of Computing*, pages 114–118, 1978.

[122] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[123] Marcus Frean. The Upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209, 1990.

[124] Stephen I. Gallant. Three constructive algorithms for network learning. In *Proc. of the 8th Annual Conf. of the Cognitive Science Society*, pages 652–660, 1986.

[125] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58, 1992.

[126] Z. Gharamani and Michael I. Jordan. Supervised learning from incomplete data via an EM approach. In *[81]*, 1994.

[127] Joydeep Ghosh and Kai Hwang. Mapping neural networks onto message-passing multiprocessors. *Journal of Parallel and Distributed Computing*, 6(2):291–330, April 1989.

[128] Philip E. Gill, Walter Murray, and Magaret H. Wright. *Practical Optimization.* Academic Press, London, 4th printing 1984 edition, 1981.

[129] Sheri L. Gish and W.E. Blanz. Comparing the performance of connectionist and statistical classifiers on an image segmentation problem. In *[361]*, pages 614–621, 1990.

[130] Nigel H. Goddard, Kenton J. Lynne, Toby Mintz, and Liudvikas Bukys. Rochester connectionist simulator. Technical Report TR 233, University of Rochester, Computer Science Department, Rochester, NY, October 1989.

[131] Rodney M. Goodman, John W. Miller, and Padhraic Smyth. An information theoretic approach to rule-based connectionist expert systems. In *[360]*, pages 256–263, 1989.

[132] Denise Gorse, A. Shepherd, and J. G. Taylor. A classical algorithm for avoiding local minima. In *Proc. of WCNN '94*, pages III–364—III–369, San Diego, CA, June 1994.

[133] Susan L. Graham, Steven Lucco, and Oliver Sharp. Orchestrating interactions among parallel computations. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 100–111, Albuquerque, NM, June 1993.

[134] Kamil A. Grajski. Neurocomputing using the MasPar MP-1. Technical Report 90-010, MasPar Computers, Sunnyvale, CA, 1990.

[135] Kamil A. Grajski. Neurocomputing using the MasPar MP-1. In *Parallel Digital Implementations of Neural Networks*. Prentice Hall, 1992?

[136] Kamil A. Grajski, G. Chinn, C. Chen, C. Kuszmaul, and S. Tomboulian. Neural network simulation on the MasPar MP-1 massively parallel processor. Technical report, MasPar Computers, Sunnyvale, CA, 1990.

[137] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[138] R.V. Guha and Douglas B. Lenat. Cyc: A mid-term report. *AI Magazine*, 11(3):32–59, 1990.

[139] F.R. Güntsch. Geschichte der Informationstechnik. Skriptum, Universität Karlsruhe, 1991.

[140] Masafumi Hagiwara. Novel back propagation algorithm for reduction of hidden units and acceleration of convergence using artificial selection. In *Proc. Int. Joint Conf. on Neural Networks, vol. 1*, pages 625–630, 1990.

[141] Dan Hammerstrom. The CNAPS architecture. Adaptive Solutions Inc., Beaverton, OR, January 1993.

[142] John B. Hampshire II and Alex Waibel. A novel objective function for improved phoneme recognition using time-delay neural networks. *IEEE Transactions on Neural Networks*, 1(2):216–228, 1990.

[143] Stephen J. Hanson, Jack D. Cowan, and C. Lee Giles, editors. *Advances in Neural Information Processing Systems 5*, San Mateo, CA, 1993. Morgan Kaufman Publishers Inc.

[144] Stephen José Hanson. Meiosis networks. In *[361]*, pages 533–541, 1990.

[145] Stephen José Hanson and Lorien Y. Pratt. Comparing biases for minimal network construction with back-propagation. In *[360]*, pages 177–185, 1989.

[146] Stefan Hänßgen. Compilation and efficient parallel execution of recursive programs. In *Proc. Int. Workshop on Parallel Processing*, Lessach, Austria, September 1994. (Technical Report University of Clausthal, Germany).

[147] Bodo Harenberg, editor. *Chronik der Technik*. Harenberg Kommunikation Verlags- und Mediengesellschaft, Dortmund, 2nd edition, 1988.

[148] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *[143]*, pages 164–171, 1993.

[149] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):377–383, July 1991.

[150] John P. Hayes, Trevor N. Mudge, Quentin F. Stout, Stephen Colley, and John Palmer. Architecture of a hypercube supercomputer. In *Proc. Int. Conf. on Parallel Processing*, pages 653–660, 1986.

[151] Robert Hecht-Nielsen. *Neurocomputing*. Addison Wesley, Reading, MA, 1990.

[152] Ernst A. Heinz. Modula-3*: An efficiently compilable extension of Modula-3 for problem-oriented explicitly parallel programming. In *Proc. of Joint Symposium on Parallel Processing*, pages 269–276, Waseda University, Tokyo, May 1993.

[153] Bruce Hendrickson and Robert Leland. The Chaco user's guide, version 1.0. UC-405 SAND93-2339, Sandia National Laboratories, Albuquerque, NM 87185, October 1993.

[154] Daniel W. Hillis and Guy L. Steele. *The Connection Machine Lisp Manual*. Thinking Machines Corp., Cambridge, MA, 1985.

[155] W. Daniel Hillis. *The Connection Machine*. ACM Distinguished Dissertations. MIT Press, 1985.

[156] Geoffrey E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40:185–234, 1989.

[157] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[158] Yoshito Hirose, Koichi Yamashita, and Shimpei Hijiya. Back-propagation algorithm which varies the number of hidden units. *Neural Networks*, 4(1):61–66, 1991.

[159] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[160] Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20:389–398, 1994.

[161] R.W. Hockney and C.R. Jesshope. *Parallel Computers.* Adam Hilger Ltd., Bristol, England, 1981.

[162] Günther Hommel, Joachim Jäckel, Stefan Jähnichen, Karl Kleine, Wilfried Koch, and Kees Koster. *ELAN Sprachbeschreibung.* Akademische Verlagsgesellschaft, Wiesbaden, Deutschland, 1979.

[163] Vasant Honavar and Leonard Uhr. A network of neuron-like units that learns to perceive by generation as well as reweighting of its links. In D. Touretzky, editor, *Proc. of the 1988 Connectionist models summer school*, pages 472–484, 1988.

[164] Vasant Honavar and Leonard Uhr. Brainsstructured connectionist networks that perceive and learn. *Connection Science*, 1(2):139–159, 1989.

[165] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Science, USA*, 79:2554–2558, 1982.

[166] Kurt Hornik. Approximation capabilities of feedforward neural networks. *Neural Networks*, 4:251–257, 1991.

[167] Kurt Hornik. Some new results on neural network approximation. *Neural Networks*, 6:1069–1072, 1993.

[168] Kurt Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[169] Edwin S.H. Hou, Nirwan Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113ff, February 1994.

[170] High Performance Fortran (HPF): Language specification. Technical report, Center for Research on Parallel Computation, Rice University, 1992.

[171] Lorenz F. Huelsbergen. Dynamic language parallelization. Ph.D. thesis TR 1178, University of Wisconsin Madison, September 1993.

[172] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic pointer-based data structures. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 218–228, Orlando, FL, June 1994.

[173] Jenq-Neng Hwang, Shih-Shien You, Shyh-Rong Lay, and I-Chang Jou. What's wrong with a cascaded correlation learning network: A projection pursuit learning perspective. In *Int. Symposium on Artificial Neural Networks*, pages E11–E20, 1993.

[174] IEEE Computer Society. *IEEE Transactions on Neural Networks: Special Issue on Hardware Implementations*, March 1992.

[175] INMOS. *Occam Programming Manual.* Prentice Hall, Eaglewood Cliffs, NJ, 1984.

[176] Marwan Jabri, Edward Tinker, and Laurens Leerink. MUME: An environment for multinet and multi-architectures neural simulation. Technical report, System Engineering and Design Automation Laboratory, University of Sydney, NSW 2006, Australia, 1993.

[177] Christian Jacob and Peter Wilke. A distributed network simulation environment for multi-processing systems. In *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*, pages 1178–1183, Singapore, 1991.

[178] Christian Jacob and Peter Wilke. The NeuroGraph neural network simulator. In *Proc. MASCOTS*, San Diego, CA, 1993.

[179] Robert A. Jacobs. Increased rates of convergence through local learning rate adaption. *Neural Networks*, 1:295–307, 1988.

[180] Ajay N. Jain. PARSEC: A connectionist learning architecture for parsing spoken language. Technical Report CMU-CS-91-208, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[181] Chuanyi Ji, Robert S. Snapp, and Demetri Psaltis. Generalizing smoothness constraints from discrete samples. *Neural Computation*, 2(2):188–197, 1990.

[182] Jiahuang Ji and Menkae Jeng. Bin-packing adjustable rectangles and applications to task scheduling on partitionable parallel computers. In *Proc. of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 312–315, Dallas, TX, December 1990. IEEE Computer Society, IEEE Computer Society Press.

[183] David S. Johnson. Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8:272–314, 1974.

[184] Mark T. Jones and Paul E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20:753–773, 1994.

[185] Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214, 1994.

[186] Nicolaos B. Karayiannis. Aladin: Algorithms for learning and architecture determination. In *Proc. Int. Joint Conf. on Neural Networks, vol. 1*, pages 601–606, Baltimore, 1992.

[187] Ehud D. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, 1990.

[188] Ken Kennedy. Compiler technology for machine-independent parallel programming. *Int. J. of Parallel Programming*, 22(1):79–97, 1994.

[189] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1977.

[190] Johan De Keyser and Dirk Roose. Load balancing data parallel programs on distributed memory computers. *Parallel Computing*, 19:1199–1219, 1993.

[191] Dongseung Kim and Byung-Guoen Yi. A two-pass scheduling algorithm for parallel programs. *Parallel Computing*, 20:869–885, 1994.

[192] David A. Kincaid, Thomas C. Oppe, John R. Respess, and David M. Young. ITPACKV 2C user's guide. Technical Report CNA-191, Center for Numerical Analysis, University of Texas, Austin, TX, November 1984.

[193] S. Kirkpatrick, C.D. Gellatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983. Reprinted in [13].

[194] Boris Kiselyov, Sergei Kleymjonov, and Nick Kulakow. Optoelectronic implementation of the Hopfield model using PDA as optical signal accumulator. In A.M. Goncharenko, F.V. Karpushko, G.V. Sinitsyn, and S.P. Apanasevich, editors, *ICO Topical Meeting on Optical Computing*, pages 187–188, Minsk, Belarus, June 1992. SPIE Volume 1806.

[195] Henrik Klagges and Michael Soegtrop. Limited fan-in random wired Cascade-Correlation. ftp from archive.cis.ohio-state.edu in /pub/neuroprose, 1992.

[196] Charles Koelbel and Piyush Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Int. Conf. on Supercomputing*, pages 414–423, Cologne, Germany, June 1991.

[197] Teuvo Kohonen. *Self-Organization and Associative Memory.* Springer Series in Information Sciences. Springer Verlag, 2nd edition, 1988.

[198] John F. Kolen and Jordan B. Pollack. Back propagation is sensitive to initial conditions. In *[221]*, pages 860–867, 1991.

[199] Detlef Koll. Untersuchung effizienter Methoden der Parallelisierung neuronaler Netze auf SIMD-Rechnern. Master's thesis, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, February 1994.

[200] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I-test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, July 1991.

[201] Robert Kramer, Rajiv Gupta, and Mary Lou Soffa. The combining DAG: A technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):805ff, August 1994.

[202] Berthold Kröger and Oliver Vornberger. Enumerative vs. genetic optimization—Two parallel algorithms for the bin packing problem. In B. Monien and Th. Ottmann, editors, *Data structures and efficient algorithms. Final Report on the DFG Special Joint Initiative*, LNCS 594, pages 330–362. Springer-Verlag, 1992.

[203] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *[254]*, pages 950–957, 1992.

[204] David J. Kuck. What do users of parallel computer systems really need? *Int. J. of Parallel Programming*, 22(1):99–127, 1994.

[205] P. Sreenivasa Kumar, M. Kishore Kumar, and A. Basu. Parallel algorithms for sparse triangular system solution. *Parallel Computing*, 19:187–196, 1993.

[206] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[207] K.J. Lang, A.H. Waibel, and G.E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3(1):33–43, 1990.

[208] Trent E. Lange. Simulation of heterogeneous neural networks on serial and parallel machines. *Parallel Computing*, 14:287–303, 1990.

[209] James R. Larus, Brad Richards, and Guhan Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical Report UW 1126, Computer Science Department, University of Wisconsin-Madison, Madison, WI, November 1992.

[210] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming experiments and results. In *Supercomputing*, pages 273–282, Albuquerque, NM, November 1991. IEEE Computer Society Press.

[211] Yuchun Lee. Handwritten digit recognition using k nearest-neighbor, radial basis function, and backpropagation neural networks. *Neural Computation*, 3:440–449, 1991.

[212] Yuchun Lee and Richard P. Lippmann. Practical characteristics of neural network and conventional pattern classifiers on artificial and speech problems. In *[361]*, pages 168–177, 1990.

[213] R. Leighton and A. Wieland. The Aspirin/MIGRAINES software tools, user's manual, release v4.0. Technical Report MTR90W00044, MITRE Washington Neural Network Group, 7525 Colshire Drive, McLean, VA, January 1991.

[214] Moshe Leshno, Vladimir Y. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a non-polynomial activation funktion can approximate any function. *Neural Networks*, 6:861–867, 1993.

[215] Nancy G. Leveson. High-pressure steam engines and computer software. In *Proc. of the 14th Int. Conf. on Software Engineering*, pages 2–14, Melbourne, Australia, 1992. IEEE Computer Society.

[216] Asriel U. Levin, Todd K. Leen, and John E. Moody. Fast pruning using principal components. In *[81]*, 1994.

[217] Zhyuan Li, Pen-Chung Yew, and Chuan-Qui Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[218] A. Linden, Th. Sudbrak, Ch. Tietz, and F. Weber. An object-oriented framework for

the simulation of neural nets. In C.L. Giles, S.H. Ilanson, and J.D. Cowan, editors, *Advances in Neural Information Processing Systems 5*, San Mateo, CA, 1992. Morgan Kaufman Publishers.

[219] Alexander Linden and Christoph Tietz. Combining multiple neural network paradigms and applications using SESAME. In *Proc. of the Int. Joint Conf. on Neural Networks*, Baltimore, June 1992. IEEE.

[220] W.-M. Lippe, Th. Feuring, and A. Jankrift. Klassifizierung neuromagnetischer Feldmuster unter Verwendung eines Cascade-Correlation Netzes. Technical Report 3/94–I, Institut für numerische und instrumentelle Mathematik/Informatik, Westfälische Wilhelms-Universität Münster, Münster, Germany, April 1994. Shorter version in Proc. World Congress on Computational Medicine, Austin, TX, April 1994.

[221] Richard P. Lippmann, John E. Moody, and David S. Touretzky, editors. *Advances in Neural Information Processing Systems 3*, San Mateo, CA, 1991. Morgan Kaufman Publishers Inc.

[222] R.J.A. Little. Regression with missing X's: A review. *Journal of the American Statistical Association*, 87(420):1227–1237, 1992.

[223] Enno Littmann and Helge Ritter. Cascade network architectures. In *Proc. Int. Joint Conf. on Neural Networks, vol. 2*, pages 398–404, Baltimore, 1992.

[224] Bin Liu, Matthew J. Cassaro, Dar-Jen Chang, and Rammohan K. Ragade. Data mapping for neural network error backpropagation training on MasPar. Preprint, March 1993.

[225] Xiao Liu and George L. Wilcox. Benchmarking of the CM-5 and the Cray machines with a very large backpropagation neural network. Technical Report 93/38, University of Minnesota Supercomputer Institute, Minneapolis, April 1993.

[226] Steven Lucco. A dynamic scheduling method for irregular parallel programs. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 200–211, San Francisco, CA, June 1992. ACM Press.

[227] Paul Lukowicz, Ernst A. Heinz, Lutz Prechelt, and Walter F. Tichy. Experimental evaluation in computer science: A quantitative study. Technical Report 17/94, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, August 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-17.ps.Z on ftp.ira.uka.de.

[228] Paul Lukowicz and Walter F. Tichy. A scalable opto-electronic CRCW shared memory. Technical Report 25/93, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, November 1993.

[229] Wolfgang Maas. Neural nets with superlinear VC dimension. *Neural Computation*, 6(5), 1994.

[230] Niels Mache. Entwicklung eines massiv parallelen Simulatorkerns für neuronale Netze auf der MasPar MP-1. Master's thesis, Universität Stuttgart, Fakultät Informatik, February 1992.

[231] A. Macintryre and E.D. Sontag. Finiteness results for sigmoidal "neural" networks. In *Proc. of the 25th Annual ACM Symposium on the Theory of Computing*, pages 325–334. ACM Press, 1993.

[232] J. Jeffrey Mahoney and Raymond J. Mooney. Combining neural and symbolic learning to revise probabilistic rule bases. In *[143]*, pages 107–114, 1993.

[233] Nashat Mansour and Geoffrey C. Fox. Parallel physical optimization algorithms for data mapping. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *CONPAR 92–VAPP IV, Second Joint Int. Conf. on Vector and Parallel Processing*, pages 91–96, Lyon, France, September 1992. Springer Verlag.

[234] Alianna J. Maren, Craig T. Harston, and Robert M. Pap, editors. *Handbook of Neural Networking Applications*. Academic Press, San Diego, CA, 1990.

[235] MasPar Computers, Sunnyvale, Calif. *MPL Language Reference Manual*.

[236] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Proc. 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 2–15, Charleston, SC, January 1993. ACM Press.

[237] Dror E. Maydan, John L. Hennesy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–14, Toronto, Canada, June 1991. ACM Press.

[238] Pamela McCorduck. *Machines Who Think.* W.H. Freeman, San Francisco, 1979.

[239] Warren McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943. Reprinted in [13].

[240] Catherine McGeoch. Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 24(2), June 1992.

[241] D.J.C. McKay. Bayesian interpolation. *Neural Computation*, 4:415–447, 1992.

[242] D.J.C. McKay. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4:448–472, 1992.

[243] Clayton McMillan, Michael C. Mozer, and Paul Smolensky. Rule induction through integrated symbolic and subsymbolic processing. In *[143]*, 1993.

[244] Carver Mead. *Analog VLSI and Neural Systems.* VLSI systems series, Computation and neural systems series. Addison Wesley, Reading, MA, 1989.

[245] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford Science Publications, 1990.

[246] Marc Mézard and Jean-Pierre Nadal. Learning in feedforward layered networks: The Tiling algorithm. *Journal of Physics A: Math. Gen.*, 22(12):2191–2203, 1989.

[247] Marvin Minsky. Logical vs. analogical or symbolic vs. connectionist or neat vs. scruffy. *AI Magazine*, 12(2):34–51, 1991.

[248] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1969.

[249] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, Cambridge, MA, expanded edition, 1988.

[250] Martin Møller. *Efficient Training of Feed-Forward Neural Networks.* PhD thesis, Computer Science Department, Aarhus University, Aarhus, Danmark, December 1993.

[251] Martin Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6(4):525–533, June 1993.

[252] Martin Møller. Supervised learning on large redundant training sets. *Int. Journal on Neural Systems*, 4(1):15–25, 1993.

[253] John Moody. Prediction risk and architecture selection for neural networks. In V. Cherkassky, J.H. Friedman, and H. Wechsler, editors, *From Statistics to Neural Networks: Theory and Pattern Recognition Applications.* Springer, NATO ASI Series F, 1994.

[254] John E. Moody, Stephen J. Hanson, and Richard P. Lippmann, editors. *Advances in Neural Information Processing Systems 4*, San Mateo, CA, 1992. Morgan Kaufman Publishers Inc.

[255] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics.* W.H. Freeman and Company, New York, 1993.

[256] N. Morgan and H. Bourlard. Generalization and parameter estimation in feedforward nets: Some experiments. In *[361]*, pages 630–637, 1990.

[257] W.G. Morris. CCG: A prototype coagulating code generator. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 45–58, Toronto, Canada, June 1991.

[258] Michael C. Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *[360]*, pages 107–115, 1989.

[259] Somnath Mukhopadhyay, Asim Roy, Lark Sang Kim, and Saneep Govil. A polynomial time algorithm for generating neural networks for pattern classification: Its stability properties and some test results. *Neural Computation*, 5:317–330, 1993.

[260] Silvia Müller. A performance analysis of the cns-1 on large dense backpropagation networks. Technical Report TR-93-046, International Computer Science Institute, Berkeley, CA, 1993.

[261] Silvia Müller and Benedict Gomes. A performance analysis of CNS-1 on sparse connectionist networks. Technical Report TR-94-009, International Computer Science Institute, Berkeley, CA, February 1994.

[262] Tarek M. Nabhan and Albert Y. Zomaya. Toward generating neural network structures for function approximation. *Neural Networks*, 7(1):89–99, 1994.

[263] Jean-Pierre Nadal. Study of a growth algorithm for a feedforward network. *Int. Journal of Neural Systems*, 1(1):55–59, 1989.

[264] B.K. Natarajan. On learning sets and functions. *Machine Learning*, 4(1), 1989.

[265] Kenney Ng and Richard P. Lippmann. A comparative study of the practical characteristics of neural network and conventional pattern classifiers. In *[221]*, pages 970–976, 1991.

[266] John R. Nickolls. The design of the MasPar MP-1: A cost effective massively parallel computer. In *Proc. of the COMPCON Spring 1990 — The 35th IEEE Computer Society Int. Conf.*, pages 25–28, San Francisco, CA, 1990.

[267] Soren S. Nielsen and Stavros A. Zenios. Data structures for network algorithms on massively parallel architectures. *Parallel Computing*, 18:1033–1052, 1992.

[268] Hans-Werner Niemann. *Vom Faustkeil zum Computer: Technikgeschichte, Kulturgeschichte, Wirtschaftsgeschichte.* Ernst Klett Verlage, Stuttgart, 1985.

[269] A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Browne, and G. Aybay. The S3.mp architecture: A local area multiprocessor. In *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 140–141, Velen, Germany, June 1993. ACM SIGACT/SIGARCH, EATCS, ACM Press.

[270] Steven J. Nowlan and Geoffry E. Hinton. Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4):473–493, 1992.

[271] Wilfried Oed and Martin Walker. An overview of Cray Research computers including the Y-MP/C90 and the new MPP T3D. In *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 271–272, Velen, Germany, June 1993. ACM SIGACT/SIGARCH, EATCS, ACM Press.

[272] O. M. Omidvar and C. L. Wilson. Optimization of neural network topology and information content using boltzmann methods. In *Proc. Int. Joint Conf. on Neural Networks, vol. 4*, pages 594–599, Baltimore, 1992.

[273] Seymour Papert. Some mathematical models of learning. In C. Cherry, editor, *Proc. of the 4th London Symposium of Information Theory*, New York, 1961. Academic Press.

[274] D.B. Parker. Learning logic. Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA, 1985.

[275] David A. Patterson and John L. Hennessy. *Computer Organization and Design, The Hardware/Software Interface*. Morgan Kaufmann, San Mateo, CA, 1994.

[276] Helene Paugam-Moisy. Parallel neural computing based on network duplicating. In Ioannis Pitas, editor, *Parallel Algorithms for Digital Image Processing, Computer Vision, and Neural Networks*, chapter 10, pages 305–340. John Wiley and Sons, Chichester, 1993.

[277] Michael Philippsen. Automatic data distribution for nearest neighbor networks. In *Frontiers '92: The Fourth Symposium on the*

Frontiers of Massively Parallel Computation, pages 178–185, McLean VA, October 1992. Also as Technical Report 3/92, Fakultät für Informatik, Universität Karlsruhe, Germany.

[278] Michael Philippsen. *Optimierungstechniken zur Übersetzung paralleler Programmiersprachen.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, December 1993. Fortschrittsberichte Informatik 292, VDI Verlag.

[279] Michael Philippsen and Markus U. Mock. Data and process alignment in Modula-2*. In *AP '93, Int. Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution, and Automatic Parallel Performance Prediction*, pages 141–149, Saarbrücken, Germany, March 1993.

[280] John Platt. A resource-allocating network for function interpolation. *Neural Computation*, 3:213–225, 1991.

[281] Mike Plonski and Chuck Joyce. RCS, GENESIS and SFINX: Three "public-domain" simulators for neural networks. *Neural Network Review*, 4(3/4), 1990.

[282] Mark Plutowksi, Garrison Cottrell, and Halbert White. Learning Mackey-Glass from 25 examples, plus or minus 2. In *[81]*, 1994.

[283] Mark Plutowski. *Selecting Training Exemplars For Neural Network Learning.* PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, CA, 1994.

[284] D. Pollard. *Convergence of Stochastic Processes.* Springer Verlag, 1984.

[285] M.J.D. Powell. Restart procedures for the conjugate gradient method. *Mathematical Programming*, 12:241–254, 1977.

[286] Lutz Prechelt. CuPit — a parallel language for neural algorithms: Language reference and tutorial. Technical Report 4/94, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, January 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-4.ps.Z on ftp.ira.uka.de.

[287] Lutz Prechelt. PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, September 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-21.ps.Z on ftp.ira.uka.de.

[288] Lutz Prechelt. A study of experimental evaluations of neural network learning algorithms: Current research practice. Technical Report 19/94, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, August 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-19.ps.Z on ftp.ira.uka.de.

[289] Lutz Prechelt. The CuPit compiler for the MasPar — a literate programming document. Technical Report 1/95, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, January 1995. Anonymous FTP: /pub/papers/techreports/1995/1995-1.ps.Z on ftp.ira.uka.de.

[290] D. Psaltis, D. Brady, X. Gu, and K. Hsu. Optical implementation of neural computers. In H. Arsenault, T. Szoplik, and B. Macukow, editors, *Optical Processing and Computing.* Academic Press, 1989.

[291] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, and M. Weßeling. Synapse-1 — A general purpose neurocomputer. Siemens AG, Corporate Research and Development Division, Proprietary information, February 1993.

[292] Nicholas J. Redding, Adam Kowalczyk, and Tom Downs. Constructive higher-order network algorithm that is polynomial time. *Neural Networks*, 6:997–1010, 1993.

[293] Russel Reed. Pruning algorithms — a survey. *IEEE Transactions on Neural Networks*, 4(5):740–746, 1993.

[294] Michael D. Richard and Richard P. Lippmann. Neural network classifiers estimate Bayesian a-posteriori probabilities. *Neural Computation*, 3:461–483, 1991.

[295] G.D. Richards and Tom Tollenaere. Documentation for RHWYDWAITH version

2.1. Technical Report ECSP-UG-7 Version 3.0, Edinburgh Concurrent Supercomputer Project, University of Edinburgh, Scotland, 1989.

[296] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Int. Conf. on Neural Networks*, San Francisco, CA, April 1993.

[297] A.K. Rigler, J.M. Irvine, and T.P. Vogl. Rescaling of variables in backpropagation learning. *Neural Networks*, 4(2):225–229, 1991.

[298] B.D. Ripley. Statistical aspects of neural networks. In O.E. Barndorff-Nielsen, J.L. Jensen, and W.S. Kendall, editors, *Networks and Chaos: Statistical and Probabilistic Aspects*. Chapman and Hall, London, 1993.

[299] Anne Rogers and Keshav Pingali. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):281–298, March 1994.

[300] Galina Rogova. Combining the results of neural network classifiers. *Neural Networks*, 7(5):777–781, 1994.

[301] Steve G. Romaniuk and Lawrence O. Hall. Divide and conquer networks. *Neural Networks*, 6:1105–1116, 1993.

[302] Dirk Roose and Raf Van Drische. Distributed memory parallel computers and computational fluid dynamics. TW 186, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, March 1993.

[303] Frank Rosenblatt. The perceptron: A perceiving and recognizing automaton (project PARA). Technical Report 85-460-1, Cornell Aeronautical Laboratory, January 1957.

[304] Frank Rosenblatt. *Principles of Neurodynamics*. Spartan Books, New York, 1962.

[305] Edward Rothberg and Anoop Gupta. Parallel ICCG on a hierarchical memory multiprocessor — addressing the triangular solve bottleneck. *Parallel Computing*, 18:719–741, 1992.

[306] Duncan Roweth. The Meiko CS-2 system architecture. In *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, page 213, Velen, Germany, June 1993. ACM SIGACT/SIGARCH, EATCS, ACM Press.

[307] Asim Roy, Lark Sang Kim, and Somnath Mukhopadhyay. A polynomial time algorithm for the construction and training of a class of multilayer perceptrons. *Neural Networks*, 6:535–545, 1993.

[308] David Rumelhart and John McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, Cambridge, MA, 1986.

[309] Lothar Sachs. *Angewandte Statistik*. Springer Verlag, Berlin Heidelberg New York, 1974. (7. Auflage, 1992).

[310] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–223, 1959.

[311] Ananth Sankar and Richard J. Mammone. Optimal pruning of neural tree networks for improved generalization. In *Proc. Int. Joint Conf. on Neural Networks, Vol. 2*, pages 219–224, Seattle, 1991.

[312] Warren S. Sarle. Neural networks and statistical models. In *Proc. of the 19th Annual SAS Users Group Int. Conf.*, April 1994.

[313] SAS Institute, Inc., Cary, NC 27511-8000. *SAS/STAT User's Guide, Version 6, Volume 1 and 2*, 4th edition.

[314] Yuji Sato, Katsunari Shibata, Mitsuo Asai, Masaru Ohki, Mamoru Sugie, Takahiro Sakaguchi, Masashi Hashimoto, and Yoshihiro Kuwabara. Development of a high-performance general purpose neurocomputer composed of 512 digital neurons. In *Proc. Int. Joint Conf. on Neural Networks*, pages 1967–1970, 1993.

[315] Cullen Schaffer. A conservation law for generalization performance. In William W. Cohen and Haym Hirsh, editors, *Machine Learning: Proc. of the 11th Int. Conf.*, San Francisco, CA, 1994. Morgan Kaufmann.

[316] W. Schiffmann, M. Joost, and R. Werner. Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical report, Institute of Physics, University of Koblenz, Koblenz, Germany, 1992.

[317] W. Schiffmann, M. Joost, and R. Werner. Synthesis and performance analysis of multilayer neural network architectures. Technical Report 16/1992, Institut für Physik, Universität Koblenz, 1992.

[318] W. Schiffmann, M. Joost, and R. Werner. Application of genetic algorithms to the construction of topologies for multilayer perceptrons. In *[3]*, pages 675–682, 1993.

[319] Wolfgang Schreiner. Parallel functional programming: An annotated bibliography. Technical report, Research Institute for Symbolic Computation, Johannes Kepler Universität, Linz, Austria, May 1993.

[320] Gary M. Scott and W. Harmon Ray. Neural network process models based on linear model structures. *Neural Computation*, 6(4):718–738, 1994.

[321] Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, pages 145–168, 1987.

[322] Jiri Sgall. *On-Line Scheduling on Parallel Machines*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1994. CMU-CS-94-144.

[323] Jude W. Shavlik, Raymond J. Mooney, and Geoffrey G. Towell. Symbolic and neural network learning algorithms: An experimental comparison. *Machine Learning*, 6:111–143, 1991.

[324] Hava T. Siegelmann. A neural programming paradigm. In *Proc. of the AAAI '94*, 1994.

[325] Jocelyn Sietsma and Robert J. F. Dow. Neural network pruning — why and how. In *IEEE Int. Conf. on Neural Networks, Vol. 1*, pages 325–333, 1988.

[326] Jocelyn Sietsma and Robert J. F. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4(1):67–79, 1991.

[327] Fernando M. Silva and Luis B. Almeida. Speeding up backpropagation. In R. Eckmiller, editor, *Advanced Neural Computers*, pages 151–158. Elsevier North Holland, Amsterdam, 1990.

[328] Natalio Simon. Constructive supervised learning algorithms for artificial neural networks. Master's thesis, Delft University of Technology, Department of Electrical Engineering, Delft, Netherlands, June 1993.

[329] Natalio Simon, Henk Corporaal, and Eugene Kerckhoffs. Variations on the Cascade-Correlation learning architecture for fast convergence in robot control. In *Proc. Neuro-Nimes*, pages 455–464, Nimes, France, November 1992. EC2.

[330] Alexander Singer. Implementations of artificial neural networks on the Connection Machine. Technical Report RL 90-2, Thinking Machines Corporation, Cambridge, MA, January 1990.

[331] J. A. Sirat and J.-P. Nadal. Neural trees: a new tool for classification. *Network*, 1:423–438, 1990.

[332] Steen Sjøgaard. *A Conceptual Approach to Generalisation in Dynamic Neural Networks*. PhD thesis, Aarhus University, Aarhus, Danmark, 1991.

[333] Steen Sjøgaard. Generalization in Cascade-Correlation networks. In *Workshop on Neural Networks for Signal Processing 1992, Vol. 2*, pages 59–68, 1992.

[334] Marc Snir. Scalable parallel computing — the IBM 9076 scalable POWERparallel 1. In *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Velen, Germany, June 1993. ACM SIGACT/SIGARCH, EATCS, ACM Press.

[335] Jianjian Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Computing*, 20:853–868, 1994.

[336] Eduardo Sontag. Feedforward nets for interpolation and classification. *J. Comp. Syst. Sci.*, 45:20–48, 1992.

[337] Eduardo D. Sontag. Some topics in neural networks and control. Technical Report LS93-02, Rutgers University, New Brunswick, NJ, 1993.

[338] M. L. Southcott and R. E. Bogner. Classification of incomplete data using neural networks. In *Proc. of 4th Australian Conf. on NN*, 1993.

[339] Alessandro Sperduti and Antonina Starita. Speed up learning and network optimization with extended back propagation. *Neural Networks*, 6:365–383, 1993.

[340] Guy L. Steele. Making asynchronous parallelism safe for the world. In *Proc. 17th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, San Francisco, CA, January 1990.

[341] Karl Steinbuch. Die Lernmatrix. *Kybernetik (Biological Cybernetics)*, 1(1):36–45, 1961.

[342] M. Stone. Cross-validatory choice and assessment of statistical predictors (with discussion). *Journal of the Royal Statistical Society*, B36:111–147, 1974.

[343] Richard V. Stone. Optoelectronic processor is programmable and flexible. *Laser Focus World*, August 1994.

[344] V.P. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–335, 1990.

[345] Brian A. Telfer and Harold H. Szu. Energy functions for minimizing misclassification error with minimum-complexity networks. *Neural Networks*, 7(5):809–818, 1994.

[346] Manoel Fernando Tenorio and Wei-Tsih Lee. Self organizing neural networks for the identification problem. In *[360]*, pages 57–64, 1989.

[347] Thinking Machines. Connection Machine model CM-2 technical summary. Technical Report TR-170, TR89-1, Thinking Machines Corp., Cambridge, MA, 1989.

[348] Thinking Machines Corp., Cambridge, MA. *C* Reference Manual, Version 6.0*, 1990.

[349] Hans Henrik Thodberg. Ace of Bayes: Application of neural networks with pruning. Technical Report 1132E, Danish Meat Research Institute, Maglegaardvej 2, DK-4000 Roskilde, Danmark, May 1993.

[350] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Džeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang. The MONK's problems: A performance comparison of different learning algorithms. Technical Report CS-91-197, Carnegie Mellon Univ., Pittsburgh, PA, 1991.

[351] Walter F. Tichy and Christian G. Herter. Modula-2*: An extension of modula-2 for highly parallel, portable programs. Technical Report 4/90, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, January 1990.

[352] Walter F. Tichy and Michael Philippsen. Hochgradiger Parallelismus. Technical Report 14/91, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, August 1991.

[353] Walter F. Tichy, Michael Phlippsen, and Phil Hatcher. A critique of the programming language C*. *Communications of the ACM*, 35(6):21–24, June 1992.

[354] Tom Tollenaere. SuperSAB: Fast adaptive backpropagation with good scaling properties. *Neural Networks*, 3:561–573, 1990.

[355] Tom Tollenaere. Neural network simulations on transputers. Technical Report TR 91 0021, Version 1, Katholieke Universiteit te Leuven, Leuven, Belgium, May 1991.

[356] Tom Tollenaere and Guy A. Orban. Simulating modular neural networks on message-passing multiprocessors. *Parallel Computing*, 17, 1991.

[357] Tom Tollenaere and Guy A. Orban. Transparent problem decomposition and mapping: A CSTools based implementation. In *Proceeding First World Conf. of Transputer Users, Transputing*, Amsterdam, April 1991. IOS.

[358] Tom Tollenaere and Guy A. Orban. Decomposition and mapping of locally connected layered neural networks on message-passing multiprocessors. *Parallel Algorithms and Applications*, 1:43–56, 1993.

[359] Aimo Törn and Antanas Žilinskas. *Global Optimization.* Lecture Notes in Computer Science 350. Springer Verlag, Berlin, Heidelberg, 1989.

[360] David S. Touretzky, editor. *Advances in Neural Information Processing Systems 1*, San Mateo, CA, 1989. Morgan Kaufman Publishers Inc.

[361] David S. Touretzky, editor. *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. Morgan Kaufman Publishers Inc.

[362] Geoffrey Towell and Jude W. Shavlik. Interpretation of artificial neural networks: Mapping knowledge-based neural networks into rules. In *[254]*, 1992.

[363] Philip Treleaven and Michael Recce. Programming languages for neurocomputers. In *[99]*, February 1988.

[364] Philip C. Treleaven. PYGMALION neural network programming environment. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 569–578. Elsevier Science Publishers, 1991.

[365] V. Tresp, S. Ahmad, and R. Neuneier. Training neural networks with deficient data. In *[81]*, 1994.

[366] Volker Tresp, Jürgen Hollatz, and Subutai Ahmad. Network structuring and training using rule-based knowledge. In *[143]*, pages 871–878, 1993.

[367] Ulrich Troitzsch and Wolfhard Weber, editors. *Die Technik von den Anfängen bis zur Gegenwart.* Georg Westermann Verlag, Unipart Verlag, Stuttgart, 1987.

[368] A.C. Tsoi and R.A. Pearson. Comparison of three classification techniques, CART, C4.5, and multi-layer perceptrons. In *[221]*, pages 963–969, 1991.

[369] D. L. Tuomenoksa and H. J. Siegel. Application of two-dimensional bin packing algorithms for task scheduling in the PASM multimicrocomputer system. In *19th Allerton Conference on Communication, Control and Computing*, page 542, October 1981.

[370] John Turek, Joel L. Wolf, and Phlip S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Supercomputing*, pages 323–332, Albuquerque, NM, November 1991. IEEE Computer Society Press.

[371] Jari Turunen, Mohammad R. Taghizadeh, and Antti Vasara. Computer-generated holograms for optical interconnections. In A.M. Goncharenko, F.V. Karpushko, G.V. Sinitsyn, and S.P. Apanasevich, editors, *ICO Topical Meeting on Optical Computing*, pages 89–97, Minsk, Belarus, June 1992. SPIE Volume 1806.

[372] Ten H. Tzen and Lionel M. Ni. Dependence uniformization: A loop parallelization technique. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):547–558, May 1993.

[373] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[374] United States Department of Defense, Springer Verlag, Lecture Notes in Computer Science 106. *The Programming Language Ada*, 1981.

[375] K.P. Unnikrishnan and K.P. Venugopal. Alopex: A correlation-based learning algorithm for feed-forward and recurrent neural networks. *Neural Computation*, 6(3):469–490, 1994.

[376] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[377] P. Vamplew and A. Adams. Missing values in a backpropagation net. In *Australian Conf. on Neural Networks*, pages 64–67, 1992.

[378] Drew van Camp. A users guide for the Xerion neural network simulator version 3.1.

Technical report, Department of Computer Science, University of Toronto, Toronto, Canada, May 1993.

[379] V.N. Vapnik and A.Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[380] Alistair Craig Veitch and Geoffrey Holmes. A modified Quickprop algorithm. *Neural Computation*, 3:310–311, 1991.

[381] Marley M.B.R. Vellasco. *Pygmalion Neural Network Programming Environment – nC Manual.* Esprit Project 2059, version 1.02 edition, 1991.

[382] Reinhard von Hanxleden. Parallelizing dynamic processes on message passing architectures. In *Proc. 5th SIAM Conf. on Parallel Processing for Scientific Computing*, pages 451–455, Houston, TX, March 1991.

[383] Reinhard von Hanxleden and Ken Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 188–199, San Francisco, CA, June 1992. ACM Press.

[384] John von Neumann. First draft report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, 1945. In [20].

[385] Alex Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1(1):39–46, 1989.

[386] David W. Wall. Predicting program behavior using real or estimated profiles. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 59–70, Toronto, Canada, June 1991.

[387] Changfeng Wang and Santosh S. Venkatesh. Optimal stopping and effective machine complexity in learning. In *[81]*, 1994.

[388] Zhenni Wang, Christine Di Massimo, Ming T. Tham, and A. Julian Morris. A procedure for determining the topology of multilayer feedforward neural networks. *Neural Networks*, 7(2):291–300, 1994.

[389] Thomas M. Warschko, Christian G. Herter, and Walter F. Tichy. Latency hiding in parallel systems: A quantitative approach. Technical Report 10/94, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, March 1994.

[390] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. Backpropagation, weight-elimination and time series prediction. In D. Touretzky, editor, *Proc. Connectionist Models Summer School*, pages 105–116, 1990.

[391] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. Generalization by weight-elimination with application to forecasting. In *[221]*, pages 875–882, 1991.

[392] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. Generalization by weight-elimination applied to currency exchange rate prediction. In *Proc. Int. Joint Conf. on Neural Networks, Vol.1*, pages 837–841, Seattle, 1991.

[393] John (Juyang) Weng, Narendra Ahuja, and Thomas S. Huang. Cresceptron: A self-organizing neural network which grows adaptivly. In *Proc. Int. Joint Conf. on Neural Networks, vol. 1*, pages 576–581, Baltimore, 1992.

[394] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[395] Nico Weymaere and Jean-Pierre Martens. A fast and robust learning algorithm for feedforward neural networks. *Neural Networks*, 4:361–369, 1991.

[396] H. White. Learning in artificial neural networks: A statistical perspective. *Neural Computation*, 1:425–464, 1989.

[397] Norbert Wiener. *Cybernetics, or: Control and Communication in the Animal and the Machine.* Wiley, New York, 1948.

[398] Peter Wilke and Ralf Scholz. NeuroGraph: Ein Simulator für künstliche neuronale Netze auf paralleler Hardware. Technical report, Lehrstuhl für Programmiersprachen

der Universität Erlangen-Nürnberg, Erlangen, Germany, 1993.

[399] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):979–993, September 1993.

[400] D.J. Willshaw, O.P. Buneman, and H.C. Longuet-Higgins. Non-holographic associative memory. *Nature*, 222:900–962, 1969. Reprinted in [13].

[401] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, 1985.

[402] Michael Witbrock and Marco Zagha. An implementation of backpropagation learning on GF11, a large SIMD parallel computer. *Parallel Computing*, 14:329–346, 1990.

[403] B.S. Wittner and J.S. Denker. Strategies for teaching layered networks classification tasks. In *[360]*, pages 850–859, 1988.

[404] William H. Wolberg and O.L. Mangasarian. Multisurface method of pattern separation for medical diagnosis applied to breast cytology. *Proc. of the National Acaedemy of Sciences, USA*, 87:9193–9196, December 1990.

[405] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 30–44, Toronto, Canada, June 1991. ACM Press.

[406] Michael Wolfe and Chau-Wen Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[407] David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

[408] Barbara B. Wyatt, Krishna Kavi, and Steve Hufnagel. Parallelism in object-oriented languages: A survey. *IEEE Software*, pages 56–66, November 1992.

[409] Mike Wynne-Jones. Node splitting: A constructive algorithm for feed-forward neural networks. In *[254]*, pages 1072–1079, 1991.

[410] Jingling Xue. Automatic non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20:711–728, 1994.

[411] Xiahua Yang. A convenient method to prune multilayer neural networks via transform domain backpropagation algorithm. In *Proc. Int. Joint Conf. on Neural Networks, vol. 3*, pages 817–822, Baltimore, 1992.

[412] Xin Yao. Evolutionary artificial neural networks. *Int. Journal of Neural Systems*, 4(3):203–222, September 1993.

[413] Dit-Yan Yeung. A neural network approach to constructive induction. In Greg C. Collins Lawrence A. Birnbaum, editor, *Machine Learning – Proc. of the 8th Int. Workshop*, pages 228–232, San Mateo, CA, June 1991. Morgan Kaufman.

[414] Andreas Zell, Thomas Korb, Niels Mache, and Tilman Sommer. *Nessus Handbuch.* Institut für parallele und verteilte Höchstleistungsrechner, Universität Stuttgart, Germany, 1991.

[415] Andreas Zell, Niels Mache, Ralf Hübner, Günter Mamier, Michael Vogt, Kai-Uwe Herrmann, Michael Schmalzl, Tilman Sommer, Artemis Hatzigeorgiou, Sven Döring, and Dietmar Posselt. SNNS user manual, version 3.0. Technical Report 3/93, Universität Stuttgart, Institut für verteilte und parallele Höchstleistungsrechner, Stuttgart, Germany, 1993.

[416] Byoung-Tak Zhang and Gerd Veenker. Focused incremental learning for improved generalization with reduced training sets. In T. Kohonen, Kai Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks, Proc. Int. Conf. on Artificial Neural Networks (ICANN)*, pages 227–232, Epsoo, Finland, June 1991. North Holland.

[417] Xiru Zhang, Michael McKenna, Jill P. Mesirov, and David Waltz. An efficient implementation of the backpropagation algorithm on the Connection Machine CM-2. Technical Report RL 89-1, Thinking Machines Corp., Cambridge, MA, August 1989.

[418] R. Zollner, H. J. Schmitz, F. Wünsch, and U. Krey. Fast generating algorithm for a

general three-layer perceptron. *Neural Networks*, 5, 1992.

[419] Steven F. Zornetzer, Joel L. Davis, and Clifford Lau, editors. *An Introduction to Neural and Electronic Networks.* Academic Press, San Diego, CA, 1990.

Index

W

X

Y

Z

Lebenslauf

Name:	Lutz Prechelt
Geburtstag:	16. Februar 1965
Geburtsort:	Bielefeld
Eltern:	Monika Prechelt, geb. Schadwell
	Klaus Prechelt

Ausbildung

3/90	Diplom in Informatik (Note gut), Universität Karlsruhe
10/85 – 3/90	Studium der Informatik an der Universität Karlsruhe
9/87	Vordiplom in Informatik (Note befriedigend), Universität Karlsruhe
7/84 – 9/85	Grundwehrdienst beim Sanitätsbatallion 3, Hamburg-Harburg
6/84	Allgemeine Hochschulreife (Note 1,7), Max-Planck-Gymnasium Bielefeld
8/75 – 6/84	Max-Planck-Gymnasium Bielefeld
8/71 – 7/75	Bültmannshofschule Bielefeld

Berufstätigkeit

3/86 – 3/90	Selbständige Tätigkeit als Softwareentwickler
4/88 – heute	Leiter und Referent auf Seminaren der überparteilichen politischen Bildung für Schüler
3/90 – heute	Wissenschaftlicher Mitarbeiter bei Prof. Tichy am Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe
10/92 – heute	Leiter und Referent auf Rhetorik- und Seminarleiterseminaren

Veröffentlichungen

[1] Lutz Prechelt. Ein Fallschablonenzerteiler für Deutsch. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 1989.

[2] Lutz Prechelt. The SIS project: Software reuse with a natural language approach. Technischer Report 2/92, Fakultät für Informatik, Universität Karlsruhe, Juni 1992.

[3] Nico Habermann, Walter Tichy, Lutz Prechelt (Ed.). Future Directions in Software Engineering. Dagstuhl Seminar Report 32, IBFI, Schloß Dagstuhl, Februar 1992. Auch in: Software Engineering Notes 18(1), Januar 1993, Seiten 35–48, ACM Press.

[4] Lutz Prechelt. Springertouren. *Informatik Spektrum*, 15(3):169–172, Juni 1992.

[5] Lutz Prechelt. Grundüberlegungen zur Ausbildung in Software–Engineering im Studiengang Informatik. In Jochen Ludewig and Kurt Schneider, editors, *Software Engineering im Unterricht der Hochschulen*, Stuttgart, 1992. Berichte des German Chapter of the ACM, Band 37, Teubner Verlag.

[6] Lutz Prechelt. Praxis versus Praktika: Die Naivität der SE-Ausbildung in der Hochschule. In Jochen Ludewig and Kurt Schneider, editors, *Software Engineering im Unterricht der Hochschulen*, Seiten 96–98, Stuttgart, 1992. Berichte des German Chapter of the ACM, Band 37, Teubner Verlag.

[7] Lutz Prechelt, Finn Dag Buø, and Rolf Adams. Engineering cost-effective natural language interfaces for knowledge based systems. In *Software Engineering and its Applications*, Seiten 649–658, Toulouse, France, December 1992.

[8] Lutz Prechelt, Finn Dag Buø, and Rolf Adams. Transportable natural language interfaces for taxonomic knowledge representation systems. In *Conference on Artificial Intelligence Applications*, Seiten 149–155, Orlando, Florida, March 1-5, 1993. IEEE.

[9] Lutz Prechelt. Measurements of MasPar MP-1216A communication operations. Technischer Report 1/93, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Januar 1993.

[10] Lutz Prechelt. Ziele und Wege für Softwaretechnik-Praktika. In Jörg Raasch and Thomas Bassler, editors, *Software Engineering im Unterricht der Hochschulen*, Seiten 78–82, Stuttgart, 1993. Berichte des German Chapter of the ACM, Band 38, Teubner Verlag.

[11] Lutz Prechelt. Comparison of MasPar MP-1 and MP-2 communication operations. Technischer Report 16/93, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 1993.

[12] Lutz Prechelt. CuPit — A parallel language for neural algorithms: Language reference and tutorial. Technischer Report 4/94, Fakultät für Informatik, Universität Karlsruhe, Januar 1994.

[13] Lutz Prechelt. A Study of Experimental Evaluations of Neural Network Learning Algorithms: Current Research Practice, Technischer Report 19/94, Fakultät für Informatik, Universität Karlsruhe, August 1994.

[14] Lutz Prechelt. PROBEN1 — A Set of Benchmarks and Benchmarking Rules for Neural Network Training Algorithms, Technischer Report 21/94, Fakultät für Informatik, Universität Karlsruhe, September 1994.

[15] Lutz Prechelt. A Motivating Example Problem for Teaching Adaptive Systems Design. ACM SIGCSE Bulletin 26(4):25–28, Dezember 1994.

[16] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, Seiten 9–18, Januar 1995. Wird erscheinen.

[17] Lutz Prechelt. The CuPit Compiler for the MasPar — A Literate Programming Document. Technischer Report 1/95, Fakultät für Informatik, Universität Karlsruhe, Januar 1995. Wird erscheinen.

[18] Lutz Prechelt. INCA: A Multi-Choice Model of Cooperation Under Restricted Communication. *BioSystems*, Herbst 1995. Wird erscheinen.