Vergleichende Untersuchung von verschiedenen Verfahren zur Verbesserung der Konvergenz beim "Supervised Learning" von Neuronalen Netzen mit Hilfe von "Error-Back-Propagation"

Diplomarbeit im Studiengang Informatik

von

Steffen Beyer

Matr.-Nr. 142 832

Referent: Prof. Dr. phil. W. Huber

Koreferent: Prof. Dr. rer. nat. W. Oberschelp

Betreuer: Dr. rer. nat. Dipl.-Psych. Dipl.-Math. K. Willmes

Rheinisch-Westfälische Technische Hochschule Aachen

14. Mai 1992

Inhaltsverzeichnis

1 Biologische Grundlagen **3**

1.1 Zytologie und Nervenzellen 3

1.2 Die Zellmembran 5

1.3 Das Membranpotential 5

1.4 Das Aktionspotential 10

1.5 Die Vorgänge an der Synapse 12

1.6 Informationsverarbeitung in natürlichen Nervensystemen 13

1.7 Weitere wichtige Eigenschaften natürlicher Nervenzellen 16

1.8 Glossar der biologischen Termini 17

2 Einführung **21**

2.1 Funktionsweise von Neuronen im Modell 21

2.2 Verbindungsstrukturen und Gewichte 24

2.3 Verarbeitung von Daten in Neuronalen Netzen 29

2.4 Lernen in Neuronalen Netzen 31

2.5 Umgebung und Kodierung 32

2.6 Aktivierungsfunktionen 35

2.6.1 Lineare Aktivierungsfunktion 35

2.6.2 "Linear Threshold"-Aktivierungsfunktion 36

2.6.3 "Brain State in a Box"-Aktivierungsfunktion 36

2.6.4 Stochastische Aktivierungsfunktion ("Boltzmann Machine") 37

2.6.5 Logistische (Sigmoide) Aktivierungsfunktion 38

2.6.6 Realisierung des Schwellenwerts in der Praxis 38

2.7 Lernregeln 39

2.7.1 Die Hebb'sche Lernregel 40

2.7.2 Die Delta-Lernregel 41

2.8 Standard Back-Propagation 42

2.9 Initialisierung der Gewichte 50
2.10 Abbruchkriterien und Fehlermaße 52
2.11 Lernmodi und Komplexitätsmaße 55
2.11.1 "Sequential Learning" 56
2.11.2 "Periodical Learning" 56
2.11.3 "Sum-of-Derivatives" Batch-Modus 58
2.11.4 "Sum-of-Updates" Batch-Modus 59
2.11.5 "Dynamic Learning" 60
2.11.6 "Dynamic Batch Learning" 62
2.11.7 Zyklen, Epochen und andere Komplexitätsmaße 64
2.12 Generalisierung und "Weight Decay" 70
2.13 "Skeletonizing" (Mozer/Smolensky) 72

3 Lernregeln **75**
3.1 Nullstellensuche (Schmidhuber) 75
3.1.1 Motivierung des Verfahrens 75
3.1.2 Herleitung des Verfahrens 76
3.1.3 Einige Eigenschaften des Verfahrens 77
3.2 Automatische Wahl von η und α (Chan/Fallside) 78
3.2.1 Anmerkungen zur Wirkung von η und α 78
3.2.2 Strategien zur Erkennung von Tälern und Plateaus 80
3.2.3 Adaption der Lernrate η 81
3.2.4 Adaption des Trägheitsfaktors α 81
3.2.5 Der Algorithmus 84
3.3 QuickProp (Fahlman) 84
3.3.1 Der QuickProp-Algorithmus 84
3.3.2 Herleitung des QuickProp-Verfahrens 85
3.3.3 Implementierung des Verfahrens 86
3.3.4 Anmerkungen zur Implementierung 88
3.4 Delta-bar-delta-Regel (Jacobs) 89
3.4.1 Schwächen von Standard-Back-Propagation 89
3.4.2 Heuristiken zur Verbesserung der Konvergenz 91
3.4.3 Entwicklung eines geeigneten Verfahrens 92
3.4.4 Die Delta-bar-delta-Regel 93
3.4.5 Hinweise zur Anwendung der Delta-bar-delta-Regel 94
3.5 Adaptives Verfahren (Silva/Almeida) 95

3.6 Perceptron (Rosenblatt/Rumelhart) 96
3.6.1 Das „klassische“ Perceptron 96
3.6.2 Lineare Modelle 96
3.6.3 Lernregeln und Eigenschaften von Perceptrons 97
3.6.4 Klassifikationen und lineare Separierbarkeit 98
3.7 Eigenes Verfahren 99
3.8 Modifikationen 100
3.8.1 Symmetrische Aktivierungsfunktion 100
3.8.2 Vermeidung der „falschen“ Nullstellen der Ableitungen 102
3.8.3 ATANH-Fehlerfunktion 102
3.8.4 Vernachlässigen kleiner Fehler 103
3.8.5 “Split Eta”-Technik 104

4 Testdaten (Benchmarks) 105
4.1 Das XOR-Problem 106
4.2 Das Parity-Problem 107
4.3 Das „zwei-oder-mehr-Blöcke“-Prädikat 107
4.4 Das Encoder-Decoder-Problem 109
4.5 Das Multiplexer-Problem 110
4.6 Das Mesh-Problem 111
4.7 Die Rotkäppchen-Simulation 113
4.8 Approximation reellwertiger Funktionen 113
4.9 Das Primzahlen-Prädikat 116
4.10 Der Aachener Aphasie Test (AAT) 116

5 Simulationsergebnisse 120
5.1 Das XOR-Problem 123
5.2 Die Rotkäppchen-Simulation 145
5.3 Das Parity-Problem 163
5.4 Das Primzahlen-Problem 165
5.5 Das „Zwei-oder-mehr-Blöcke“-Prädikat 167
5.6 Das Mesh-Problem 170
5.7 Das Multiplexer-Problem 172
5.8 Approximation reeller Funktionen 173
5.9 Der Aachener Aphasie Test 175

6 Schluß **178**

6.1 Bewertung der Ergebnisse 178

6.2 Ausblick 180

A Abbildungen **182**

B Programmbeschreibung **203**

Vorwort

Die vorliegende Arbeit wurde an der Neurologischen Klinik, Lehrgebiet Neurolinguistik, des Klinikums der RWTH Aachen geschrieben, wo ich bereits vorher für geraume Zeit als studentische Hilfskraft beschäftigt war.

Meine Aufgaben bestanden dabei in der Applikation von Neuronalen Netzen (besonders der beiden Modelle "Competitive Learning" und "Back-Propagation") auf Anwendungen der Neurolinguistischen Forschung (besonders der Klassifikation von Patientenprofilen anhand des Aachener Aphasie Tests (AAT) und der Untersuchung der Zusammenhänge zwischen Läsionen des Gehirns, ermittelt und lokalisiert anhand von Computer-Tomographien, und den dabei auftretenden Sprachstörungen).

Die zur Simulation der Neuronalen Netzwerke erforderlichen Programme wurden auf einem VAX 11/780-System mit Betriebssystem VMS in PASCAL geschrieben.

Das Manuskript dieser Arbeit selbst wurde auf einem IBM PC (kompatiblen) geschrieben und mit Hilfe von LaTeX gesetzt.

In dieser Arbeit werden mehrere Verfahren aus der Literatur (sowie ein eigenes) vorgestellt, die die Konvergenzeigenschaften des "Back-Propagation"-Verfahrens verbessern sollen.

Bei "Back-Propagation" handelt es sich um ein Lernverfahren im sogenannten "Supervised Mode", d.h., dem Netzwerk werden explizit die Ausgabemuster vorgegeben, die es als Antwort auf bestimmte Eingabemuster produzieren soll. Im Unterschied dazu ist beispielsweise das "Competitive Learning" ein Verfahren im "Free-running Mode", bei dem das Netzwerk seine Ausgaben selbst bestimmt.

Das Ziel dieser Arbeit ist ein zweifaches: Einmal sollen die grundlegenden Konzepte von Konnektionistischen Modellen sowie die verwendeten speziellen Verfahren möglichst umfassend und verständlich erläutert werden, ohne jedoch andere ausgezeichnete Arbeiten zu diesem Thema wie beispielsweise von Rumelhart, McClelland und weiteren Autoren (besonders in [24], [25] und [17]) zu wiederholen.

Ich widme diese Arbeit daher denjenigen, die sich in das Gebiet der Konnektionistischen Modelle einarbeiten möchten und hoffe, daß sie sie nützlich finden werden. Aus demselben Grunde ist besonders das Kapitel über die biologischen Grundlagen ausführlicher gehalten als das zur Erfüllung des zweiten Zieles alleine (siehe hierunter) nötig gewesen wäre.

Das zweite Ziel dieser Arbeit besteht darin, eine möglichst günstige Alternative für das Standard "Back-Propagation"-Verfahren zu finden, das zum Einen sehr aufwendig in der Handhabung ist, bedingt durch eine Vielzahl von Parametern, die, vom Problem abhängig, erst durch Versuch und Irrtum optimal eingestellt werden müssen, und das zum Anderen nur sehr langsam konvergiert; die zu lernenden Muster müssen dem Netzwerk typischerweise einige tausend oder zigtausend Male präsentiert werden.

Zu diesem Zwecke wurden verschiedene Verbesserungsvorschläge aus der Literatur implementiert, untersucht und verglichen.

An dieser Stelle möchte ich mich außerdem bei meinen Betreuern, Dr. K. Willmes und Prof. Dr. W. Huber, sehr für ihre stets wohlwollende Unterstützung bedanken sowie ihnen und den übrigen Mitarbeitern der Abteilung für das menschlich angenehme Arbeitsklima meinen Dank aussprechen.

Kapitel 1

Biologische Grundlagen

Ein Grund für die Beschäftigung mit Neuronalen Netzen und die Faszination, die diese ausüben, liegt sicher in der Analogie mit Nervenzellen und Nervensystemen von Lebewesen. Lebewesen, die oft sehr viel kleiner sind als ein integrierter Schaltkreis in seinem DIP-Gehäuse (z.B. Insekten), aber dennoch Leistungen der Orientierung in der Umwelt, der Steuerung situationsangepassten Verhaltens und der Regelung des eigenen Körpermilieus vollbringen, die kein herkömmliches Computerprogramm zu imitieren vermag.

Noch viel schwerer tat man sich bisher (trotz „Künstlicher Intelligenz“) bei der Nachahmung von Funktionen des menschlichen Gehirns wie z.B. das Erkennen von Gesichtern, Sprachproduktion, Sprachverständnis, um nur einige wenige zu nennen.

Durch die Verwendung ähnlicher „Bausteine“ und Konzepte wie die Natur hofft man (neben der erfolgreichen Anwendung in vielen sonst nicht oder nur sehr schwer lösbaren Problemen) etwas darüber zu lernen, wie die Natur die ihr gestellten Aufgaben auf so bewundernswert effiziente, scheinbar einfache Weise gelöst hat.

Die Übertragung von Ergebnissen, die anhand von konnektionistischen Modellen gewonnen wurden, auf die Verhältnisse in der Natur oder beim Menschen ist jedoch problematisch, da alle konnektionistischen Modelle extreme Simplifizierungen der Verhältnisse in biologischen Nervensystemen darstellen.

Wie komplex diese Verhältnisse in Wirklichkeit sind, davon soll das folgende Kapitel eine Vorstellung geben.

Gleichzeitig sollen dabei einige Konzepte, die in konnektionistischen Systemen Anwendung finden, motiviert werden.

1.1 Zytologie und Nervenzellen

Jede tierische Zelle besitzt eine *Zellmembran*, welche das Zellinnere, *Zell-* oder *Zytoplasma* genannt, gegen die Umwelt abgrenzt. Das Zellplasma besteht einmal aus der halbflüssigen, gelartigen Grundsubstanz, *Zytosol* genannt, zum anderen aus den *Zellorganellen*, die im Zytosol liegen. Zu diesen Zellorganellen zählen z.B.

- der *Zellkern*, Träger der genetischen Information und für die Steuerung der Stoffwechselvorgänge in der Zelle verantwortlich,

- die *Mitochondrien* zur Energiegewinnung durch Verbrennung organischer Substanzen mit Sauerstoff,

- das *Endoplasmatische Retikulum*, ein System von durch Membranen begrenzten Hohlräumen, die als Reaktionsgefäße für die verschiedenen Stoffwechselvorgänge in der Zelle dienen,

- die am Endoplasmatischen Retikulum („*rauhes ER*“) befestigten *Ribosomen*, auch *Polysomen* genannt, die die Synthese der in der Zelle benötigten Eiweiße durchführen;

um nur die wichtigsten kurz zu nennen.

(Siehe dazu auch Abbildung 1.1 Seite 4: Die Strukturen der Zelle sind nicht starr, sondern verändern sich ständig; die Zelle ist ein dynamisches System. Im Zytoplasma befinden sich auch Stoffwechselprodukte.)

Abbildung 1.1: Schema der Zelle. 1 Endoplasmatisches Retikulum (ER), 2 Verbindung des ER mit dem ER der Nachbarzelle, 3 Zentriol quer, 4 Zentriol längs, 5 Lysosom, 6 Zellmembran, 7 Kernhülle, 8 Nukleolus, 9 Kernpore, 10 Mikrotubuli, 11 Kernplasma mit DNA, 12 Golgivesikel, 13 Dictyosom, 14 Entstehung von Lysosomen, 15 freie Ribosomen und Polysomen, 16 Zytosol, 17 Ribosomen an ER gebunden, 18 Cristae des Mitochondriums. (Aus: Linder [16] Seite 23)

Nervenzellen oder „*Neuronen*“ verfügen darüber hinaus über einige spezielle Anpassungen: Einmal ein am Zellkörper (*Soma*) ansetzendes, baumartig verzweigtes Geflecht, die *Dendriten* (in seiner Gesamtheit auch „*Dendritenbaum*“ genannt), zum anderen einen langen Fortsatz (beim Menschen bis zu über einen Meter lang), die sogenannte *Nervenfaser*, auch der „*Neurit*“ oder das „*Axon*“ genannt. Die Ursprungsstelle des Axons am Zellkörper der Nervenzelle wird *Axonhügel* genannt. (Siehe dazu auch Abbildung 1.2 Seite 6)

Das Axon dient der Erregungsfortleitung, es endet in einer (oder mehreren) kleinen Verdickung, *Synapse* genannt, welche die Verbindung mit einer nachfolgenden Zelle herstellt — einer Muskel-, Drüsen- oder einer weiteren Nervenzelle. (Im Falle einer Nervenzelle endet die Synapse (in der Regel) am Dendritenbaum. An Muskelzellen endende Synapsen nennt man auch „*motorische Endplatten*“)

1.2 Die Zellmembran

Die Zellmembran, auch *Plasmalemma* genannt, besteht aus vier Schichten: Die inneren zwei Schichten bestehen aus *Lipid*molekülen, Verwandte der *Fette*. (Natürlich vorkommende Fette sind sogenannte *Triglyzeride*; Verbindungen (*Ester*) aus *Glyzerin* (einem dreiwertigen Alkohol) und drei *Fettsäuren*)

Im Unterschied zu Fetten enthalten Lipide anstelle einer der Fettsäuren eine *polare* (d.h. lokal elektrisch geladene) und daher *hydrophile* („wasseranziehende“) Verbindung, meist ein Phosphatsäurerest. (Siehe Abbildung 1.3 Seite 7)

Diese hydrophilen Verbindungen der Lipide zeigen in der Membran jeweils zu den Außenseiten, die beiden anderen, *hydrophoben* („wasserabstoßenden“) oder auch *lipophilen* („fettanziehenden“), sind nach innen gerichtet.

Diese zwei aus Lipiden bestehenden Membranschichten werden ausschließlich durch elektrostatische (im polaren Teil) und durch *van-der-Waals*-Kräfte (im unpolaren, d.h. hydrophoben Teil) zusammengehalten; dadurch hat die Membran eine halbflüssige Struktur („*Fluid-Mosaic-Modell*“).

Die zwei äußeren Schichten der Membran bestehen aus *Proteinen*, wobei man die „*peripheren*“ Membranproteine, die den Lipidschichten nur aufgelagert sind, und die „*integralen*“ Membranproteine, die mehr oder weniger tief in die Lipidphase eindringen, unterscheidet.

Die Membranproteine, die durch die gesamte Membran hindurchgehen (die eine wichtige Rolle für alle aktiven und einige passive Transportvorgänge an der Membran spielen), nennt man auch *Transmembranproteine* (nach P. Schmidt [27]).

Auf der Außenseite der Zellmembran findet man außerdem noch Zuckermoleküle, die teils mit den Membranproteinen, teils mit den Lipidmolekülen verbunden sind, die zusammen als die *Glykokalix* der Zelle bezeichnet werden und für die Erkennung körpereigener Gewebe und die *Immunreaktion* (Abwehr körperfremder Stoffe) als *Antigene* eine wichtige Rolle spielen. Die Zellmembran ist insgesamt etwa 60 – 120 Å (6 – 12 nm) dick. (Siehe Abbildungen 1.4, 1.5 und 1.6 auf den Seiten 8, 8 und 9)

1.3 Das Membranpotential

Sticht man eine Elektrode in eine Zelle und taucht eine zweite in das die Zelle umgebende Medium, so kann man eine elektrische Spannung zwischen den beiden Elektroden messen. Das Zytoplasma ist gegenüber dem umgebenden Medium negativ geladen. Diese Spannung,

Abbildung 1.2: Nervenzelle (Motoneuron im Rückenmark), schematisch. Durchmesser des Axons etwa 5 bis 20 μm. Das linke Einschaltbild zeigt Fasern von anderen Nervenzellen, die Synapsen mit Zellkörper und Dendriten des Neurons bilden. Im rechten Einschaltbild ist das Axon mit der Schwannschen Scheide vergrößert abgebildet. (Aus: Linder [16] Seite 155)

Abbildung 1.3: Phosphatidylcholin, ein Phospholipid. Man beachte die elektrischen Ladungen, die der Kopfgruppe ihren polaren Charakter verleihen. (Aus: Eckert/Randall [6] Seite 91)

Abbildung 1.4: Modell der Elementarmembran. Aus der Außenmembran tierischer Zellen ragen Ketten von Zuckermolekülen (oval und quadratisch) als Glykokalix heraus. Die Zuckermoleküle sind mit den Protein- und den Lipidmolekülen verbunden. (Aus: Linder [16] Seite 24)

Abbildung 1.5: Querschnitt des "mosaic bilayer" Modells. Es zeigt die geladenen hydrophilen Aminosäureseitengruppen der Proteine, die in die wässrige Phase hineinragen, und die ungeladenen hydrophoben Gruppen, die sich im Kontakt mit der Lipidphase befinden. (Aus: Eckert/Randall [6] Seite 96)

Abbildung 1.6: Dreidimensionales Diagramm des "fluid mosaic"-Modells der Membran. Es zeigt die globulären integralen Proteine, die in die Lipid-Doppelschicht eingebettet sind. (Aus: Eckert/Randall [6] Seite 96)

Membranpotential genannt, beträgt bei ruhenden (nicht gereizten) Zellen $50 - 200$ mV. Man nennt dies auch das *Ruhepotential* der Zelle.

Das Ruhepotential ist eine Eigenschaft aller lebenden Zellen. Es wird unter Energieverbrauch aktiv erzeugt, wobei Membranproteine in Form der sogenannten *Natrium-Kalium-Pumpe* Natriumionen aus dem Zellinneren nach außen und Kaliumionen von außen nach innen transportieren.

Dieser Transport erfolgt offenbar aneinander gekoppelt, wie Experimente belegen, daher spricht man auch von *Kotransport* bzw., da die Ionen in entgegengesetzte Richtung transportiert werden, von *Antiport* (englisch *countertransport*).

Die Natrium- und Kaliumionen werden dabei im Verhältnis 3 : 2 durch die Membran hindurchgeschleust. (Siehe Abbildung 1.7 Seite 10)

Da die Membran in gewissem Maße für Kaliumionen durchlässig ist, jedoch kaum für Natriumionen (die *Permeabilität* der Membran im Ruhezustand ist für Kaliumionen etwa 30mal höher als für Natriumionen), diffundieren Kaliumionen aufgrund des durch die Natrium-Kalium-Pumpe erzeugten Konzentrationsgefälles wieder nach außen. Es stellt sich ein Gleichgewicht zwischen Konzentrationsgefälle und der aus der Ladungsverschiebung resultierenden elektrischen Spannung ein.

Schließlich tragen zur Ungleichverteilung der Ionen an der Membran auch die im Zellinneren vorhandenen organischen Ionen (meist Proteine mit negativ geladenen Seitengruppen) bei, da sie die Membran nicht passieren können und dadurch (passiv) Chloridionen nach außen verdrängen und Kaliumionen in der Zelle zu halten helfen. (*Donnan-Verteilung*, *Donnan-Gleichgewicht*)

Abbildung 1.7: Direkte und indirekte Beiträge der Natrium-Kalium-Pumpe zum Ruhepotential. Wegen der 3 Na : 2 K-Austauschrate trägt die Pumpe *direkt* zum Ruhepotential bei, indem sie positive Ladung aus dem Zellinneren entfernt. Indem sie eine hohe interne Kaliumkonzentration aufrechterhält, trägt die Pumpe *indirekt* zum Ruhepotential bei. (Aus: Eckert/Randall [6] Seite 150)

Wird eine Zelle gereizt (mechanisch, chemisch, elektrisch oder durch Licht- oder Temperaturänderungen), so verändert sich bei den meisten Zellen das Membranpotential. Es wird in der Regel dabei niedriger, d.h. die Spannung zwischen dem Zytoplasma und dem Außenmedium wird kleiner. Je höher die Reizintensität, desto stärker wird das Membranpotential verändert.

1.4 Das Aktionspotential

In Nervenzellen und manchen Muskelfasern gibt es eine besondere Reaktion auf eine Erniedrigung des Membranpotentials (man spricht auch von *Depolarisation* des Membranpotentials), das *Aktionspotential*: Fällt das Membranpotential unter einen gewissen Schwellwert ab, wird die Membran schlagartig für Natriumionen durchlässig (der Wert der Permeabilität der Membran für Natriumionen schnellt auf das 300fache des Ruhewerts und damit auf das 10fache der Permeabilität der Membran für Kaliumionen).

Die Natriumionen schießen daraufhin in die Zelle ein, die kurzzeitig sogar positiv gegenüber dem Außenmedium wird. Dies dauert ca. $1 - 2$ ms.

Danach erhöht sich die Permeabilität der Membran für Kaliumionen, die nun nach außen strömen, wodurch das Membranpotential rasch wieder zum Ruhewert zurückkehrt (kurzzeitig wird dabei das Ruhepotential sogar unterschritten). Gleichzeitig fällt die Permeabilität der Membran für Natriumionen wieder auf ihren ursprünglichen Wert.

Diese Phase dauert etwas länger als die vorherige; während dieser Zeit ist eine erneute Auslösung eines Aktionspotentials nicht möglich (sogenannte *Refraktärzeit*). (Siehe Abbildungen 1.8 und 1.10 auf den Seiten 11 und 12)

Schließlich fällt auch die Durchlässigkeit der Membran für Kaliumionen wieder ab. Der

Abbildung 1.8: Richtung und Stärke der Ionenströme beim Ruhe- und beim Aktionspotential, schematisch. Die Dicke der Pfeile veranschaulicht die unterschiedlichen Mengen der durch die Membran hindurchtretenden Ionen. (Aus: Linder [16] Seite 157)

Einstrom an Natrium- und der Ausstrom an Kaliumionen während eines Aktionspotentials wird durch den aktiven Transport der Natrium-Kalium-Pumpe allmählich wieder ausgeglichen. Die Menge der wandernden Ionen ist jedoch außerordentlich gering, so daß nach einem einzigen Aktionspotential keine meßbare Veränderung der Ionenkonzentrationen eintritt.

Amplitude und Dauer des Aktionspotentials sind von der Reizstärke unabhängig; wenn der Reiz den Schwellenwert übersteigt, entsteht ein Aktionspotential, wird der Schwellenwert nicht erreicht, unterbleibt es (*Alles-oder-Nichts-Gesetz*). (Siehe Abbildung 1.9 Seite 11)

Abbildung 1.9: Aktionspotential der Membran einer Nervenzelle als Antwort auf einen depolarisierenden Stimulus, der das Membranpotential bis zum Schwellwert erniedrigte. Niedrigere Reizstärken konnten die Alles-oder-Nichts-Antwort nicht auslösen. (Aus: Eckert/Randall [6] Seite 152)

Nach dem gleichen Mechanismus funktioniert die Erregungsfortleitung in einer Nervenfaser: Wird die Membran der Nervenfaser an einer beliebigen Stelle — z.B. durch eine von außen angelegte Spannung — über einen gewissen Schwellwert hinaus depolarisiert (erniedrigt), so entsteht an dieser Stelle ein Aktionspotential. Durch elektrische Ausgleichsströmchen

Abbildung 1.10: (A) Refraktäres Verhalten nach einem Aktionspotential. Fünf Stimuluspaare (A_1, B_1 bis A_5, B_5) wurden angelegt; mit abnehmendem zeitlichem Abstand zwischen erstem (A) und zweitem Stimulus (B) jeden Paares. Während der relativen Refraktärzeit wurde das Aktionspotential B kleiner. Während der absoluten Refraktärzeit konnte kein zweites Aktionspotential ausgelöst werden. (B) Zeitlicher Verlauf der Änderung der Erregbarkeit. (Aus: Eckert/Randall [6] Seite 153)

zwischen dieser und den benachbarten Stellen werden nun diese ebenfalls depolarisiert, die dann ihrerseits ein Aktionspotential erzeugen.

Durch erneute Ausgleichsströme werden wiederum die benachbarten Stellen depolarisiert — auch die Stellen, von denen die Erregung gerade kam und die bereits wieder das Ruhepotential erreicht haben. Diese befinden sich aber noch in der Refraktärzeit und erzeugen daher kein weiteres Aktionspotential.

Dadurch wandert das Aktionspotential die Nervenfaser entlang; prinzipiell ist dies in beide Richtungen möglich. Bei Nervenzellen entsteht jedoch (bei entsprechender Reizung der Zelle) das Aktionspotential am Axonhügel und wandert dann über das Axon bis zu der/den Synapse(n). (Siehe Abbildung 1.11 Seite 13)

1.5 Die Vorgänge an der Synapse

Erreicht ein über das Axon laufendes Aktionspotential schließlich eine Synapse, die das Ende des Axons bildet, so führt das Aktionspotential in der Synapse dazu, daß ein Stoff, ein sogenannter *Neurotransmitter*, von der Synapse in den *synaptischen Spalt* zwischen Synapse und der nächsten Zelle, der Folgezelle (die eine weitere Nervenzelle, eine Muskel- oder Drüsenzelle sein kann), ausgeschüttet wird. Die Neurotransmitter werden in den Synapsen in membranumschlossenen Bläschen (*Vesikeln*) aufbewahrt, die bei einem Aktionspotential mit der Außenmembran verschmelzen und sich dadurch nach außen entleeren (*Exozytose*).

An der Membran der Folgezelle befinden sich Rezeptoren, die die Neurotransmittermoleküle an sich binden und daraufhin die Permeabilität der Membran für bestimmte Ionen verändern

Abbildung 1.11: (A) Weiterleitung eines Aktionspotentials, schematisch. (B) Sprunghafte (saltatorische) Erregungsleitung in einer Nervenfaser mit Schwannscher Scheide. (Aus: Linder [16] Seite 158)

(bei Nervenzellen in der Regel Kalium, bei Muskelzellen Calcium).

Dadurch verändert sich in gewissem Maße das Membranpotential der Folgezelle.

Durch *Enzyme* (Biokatalysatoren) der Folgezelle werden die Neurotransmitter in eine inaktive Form umgewandelt, die dann von der Synapse wieder aufgenommen und in die aktive Form zurückversetzt werden. (Siehe Abbildung 1.12 Seite 14)

Die Differenz zwischen dem Membranpotential im Ruhezustand (Ruhepotential) und im gereizten Zustand wird *Rezeptorpotential* genannt.

Manche der Neurotransmitter erhöhen, andere senken die Permeabilität der Membran für „ihre“ Ionen. Dementsprechend wirken manche Synapsen erregend, manche hemmend auf die Folgezelle; diese verrechnet alle eintreffenden Impulse (sowohl die hemmenden als auch die erregenden) durch Überlagerung.

Die Anzahl der Rezeptoren für die Neurotransmittermoleküle kann (an ein- und derselben Folgezelle) von Synapse zu Synapse stark variieren; dadurch und durch unterschiedlichen Abstand der Synapsen vom Axonhügel, an dem das neue Aktionspotential erzeugt wird, erfahren eintreffende Signale eine unterschiedliche Wichtung.

Zusätzlich kann auch die Menge an Neurotransmitter, die pro Aktionspotential von der Synapse ausgeschüttet wird, von Synapse zu Synapse variieren. Man nimmt an, daß sich auch im Laufe der Zeit bei ein- und derselben Synapse diese Menge verändert — man vermutet, daß hier die Grundlage des Lernens (d.h. der Informationsspeicherung) liegt.

1.6 Informationsverarbeitung in natürlichen Nervensystemen

Reizintensitäten werden im Nervensystem durch die Frequenz einer Folge von Aktionspotentialen kodiert, deren Dauer in der Regel mit der Dauer des Reizes übereinstimmt. (Siehe

Abbildung 1.12: Schema des Acetylcholinkreislaufs an der Synapse. Weil synaptische Bläschen mit Überträgerstoff A nur im Axonende liegen, nicht aber im Dendriten, kann eine Erregung nur in eine Richtung — vom Axon zum Dendriten des folgenden Neurons (bzw. zur Drüsenzelle oder Muskelfaser) — und nicht umgekehrt geleitet werden (Gleichrichterfunktion der Synapsen). Der synaptische Spalt ist unverhältnismäßig breit gezeichnet, in Wirklichkeit hat er etwa doppelte Membrandicke. (Aus: Linder [16] Seite 159)

dazu auch weiter unten)

Im Allgemeinen führt ein einziges Aktionspotential einer Nervenzelle nicht zur Auslösung eines Aktionspotentials in der Folgezelle; dazu sind in der Regel viele erregende Impulse notwendig. (Die sich auch gegen die hemmenden Impulse, die ggfs. gleichzeitig an derselben Zelle ankommen, durchsetzen müssen)

Die Überlagerung aller an den Dendriten eintreffenden hemmenden und erregenden Signale und damit die Änderung des Membranpotentials (also das Rezeptorpotential) breitet sich passiv über den Zellkörper bis zum Axonhügel hin aus, wo es zu einem Aktionspotential kommt, falls das Rezeptorpotential an dieser Stelle einen gewissen Schwellwert überschreitet (man spricht dann von einem *überschwelligen* Reiz). (Siehe dazu auch die Abbildung 1.13 Seite 15)

Erläuterung zur Abbildung 1.13 Seite 15: Das von den Dendriten kommende Rezeptorpotential (e.p.s.p. = excitatory post-synaptic potential) breitet sich elektrotonisch aus und wird mit zunehmendem Abstand kleiner. Die Dichte der Natriumkanäle (gepunktet) bestimmt den Schwellwert (threshold) für die Erzeugung des Aktionspotentials. Obwohl das synaptische Potential auf dem Wege zum Axon hin kleiner wird (siehe oberer Teil der Abbildung), wird das Aktionspotential in der Zone am Beginn des Axons (am Axonhügel) erzeugt, wo die Dichte der Natriumkanäle hoch und der Schwellwert niedrig ist. Die gestrichelte Linie gibt den Verlauf des Synapsenpotentials an, den es bei nicht zustandekommendem Aktionspotential nähme.

Ist die Reizintensität sehr hoch und/oder hält der Reiz länger an, dann kommt es zur Bildung mehrerer Aktionspotentiale hintereinander; wobei der zeitliche Abstand (also die Frequenz) der Aktionspotentiale untereinander von der Reizintensität abhängt. (Vergleiche Abbildung 1.14 Seite 15)

Im Allgemeinen stellt man einen logarithmischen Zusammenhang zwischen der Reizinten-

Abbildung 1.13: Räumliche Abnahme des Rezeptorpotentials und Impulsauslösung. Erläuterung im Text. (Aus: Eckert/Randall [6] Seite 217)

Abbildung 1.14: Obere Reihe: Reizung einer Sinneszelle mit Reizen verschiedener Höhe und Dauer. Mittlere Reihe: Das von den Reizen erzeugte Rezeptorpotential (intrazelluläre Ableitung aus der Sinneszelle). Untere Reihe: Aktionspotentiale in der Nervenfaser (extrazelluläre Ableitung von der zugehörigen Nervenfaser). Schematisch. (Aus: Linder [16] Seite 161) sität und der Frequenz der Aktionspotentiale fest. Die Logarithmusbildung kann jedoch (zumindest theoretisch) an verschiedenen Stellen der Wirkungskette erfolgen: zwischen der Reizintensität und dem entsprechenden Rezeptorpotential in der Sinneszelle, zwischen der Frequenz der Aktionspotentiale und der Menge des ausgeschütteten Neurotransmitters an der Synapse, zwischen der Menge des an Rezeptoren der Folgezelle gebundenen Neurotransmitters und dem dadurch erzeugten Rezeptorpotential, oder auch am Axonhügel zwischen dem Rezeptorpotential und der Frequenz der abgegebenen Aktionspotentiale.

Ausgehend von diesen (und weiteren) Beobachtungen (man hat z.B. Synapsen beobachtet, die an einem anderen Axon kurz vor dessen Synapse enden) wurden beispielsweise die sogenannten *Sigma-Pi-Units* inspiriert, die in manchen konnektionistischen Modellen verwendet werden (diese multiplizieren jeweils eine bestimmte Anzahl ihrer Eingänge miteinander und summieren anschließend die Produkte auf. Siehe Rumelhart/McClelland [24]). Ein weiterer ähnlicher Ansatz wird in der Arbeit von Durbin/Rumelhart [5] verfolgt.

Bei vielen Zellen beobachtet man allerdings auch eine Abhängigkeit der Frequenz von der Zeit: Man unterscheidet *tonisches* (von der Zeit unabhängiges) Verhalten, *phasisches* Verhalten (dabei sinkt die Frequenz asymptotisch auf Null) sowie *phasisch-tonisches* Verhalten (dabei sinkt die Frequenz asymptotisch auf einen niedrigeren Wert ab). (Vergleiche dazu Abbildung 1.15 Seite 16)

Abbildung 1.15: Das Verhalten verschiedener Sinneszelltypen auf einen konstanten Reiz. (Aus: Linder [16] Seite 161)

1.7 Weitere wichtige Eigenschaften natürlicher Nervenzellen

Es gibt noch einige andere wichtige Eigenschaften, die alle lebenden Nervenzellen von ihren „Pendants“ in konnektionistischen Modellen unterscheiden: Das ist einmal ihre *Ermüdbarkeit*, wie sie bereits in Abbildung 1.10 Seite 12 kurz angedeutet wird. Aber auch noch andere innere Faktoren des Organismus' oder der Zelle selbst können Schwankungen in der Reizantwort einer Zelle bewirken, sowohl was die Auslösbarkeit, als auch was die Stärke (Amplitude und/oder Frequenz) der Reaktionen angeht.

Eine andere, wesentliche Eigenschaft aller Nervenzellen ist ihre *spontane Aktivität*. Nervenzellen sind nicht im mathematischen oder „informatischen“ Sinne deterministisch; vielmehr feuern sie auch spontan (d.h. ohne Einwirkung von außen) und zufällig Aktionspotentiale ab. Dieses „Hintergrundrauschen“ bildet den normalen Grundzustand des Gehirns. Man vermutet, daß negative Größen im Gehirn durch mehr oder weniger starke Inhibition (Unterdrückung) dieses Hintergrundrauschens (das so etwas wie den Nullwert darstellt) kodiert werden. Ergebnisse aus der Chaos-Forschung bzw. der Theorie der Fraktale legen den Verdacht nahe, daß dieses Grundrauschen im Gehirn sogar ein wesentliches Funktionsmerkmal sein könnte; wobei Gedanken, Ideen und Bilder (also alle internen Repräsentationen von Gedankeninhalten) in chaotischen Attraktoren eine Entsprechung besitzen (siehe in GEO [8], Peitgen/Richter [20] u.a.)

Eine weitere Form spontaner, allerdings nicht stochastischer, sondern periodischer elektrischer Aktivität von Körper- und Nervenzellen ist für (fast) alle tierischen Lebewesen von lebenswichtiger Bedeutung: Von ihr hängt beispielsweise die regelmäßige Kontraktion des Herzmuskels ab, dessen regelmäßige Erregung autonom in den Herzmuskelzellen erzeugt wird (und die von außen lediglich in ihrer Frequenz mehr oder weniger modifiziert werden kann).

Ein anderes Beispiel sind die rhythmischen Flossenbewegungen von Fischen, die autonom in ihrem Rückenmark erzeugt werden — sie werden vom Gehirn nur modifiziert (zur Richtungs- und Geschwindigkeitskontrolle sowie zur Synchronisation der Vorder- und Hinterflossen) oder ganz gehemmt. Durchtrennt man das Rückenmark im Nacken, schwimmen die Fische langsam (aber unkoordiniert) geradeaus weiter.

Bei der Erzeugung dieser autonomen Rhythmen spielen mehrere ionische Mechanismen an der Zellmembran eine Rolle, die auf bisher noch nicht vollständig geklärte Weise interagieren und zu periodischen Schwankungen des Membranpotentials führen. Sind diese Oszillationen regelmäßig, relativ langsam und führen sie zu Aktionspotentialen, so spricht man von *„Schrittmacher“-Potentialen* (englisch “*pacemaker-potentials*”).

1.8 Glossar der biologischen Termini

Aktionspotential: Impulsförmige Änderung des Membranpotentials, das zur Erregungsfortleitung dient
Aktivität, spontane: Eigenschaft aller Nervenzellen, auch ohne äußere Einwirkung (also spontan), zufällig (bzw. periodisch) zu „feuern“
Alles-oder-Nichts-Gesetz: Die Auslösung eines Aktionspotentials hängt nur davon ab, ob der auslösende Reiz über- oder unterschwellig ist; die Amplitude des Aktionspotentials ist vom Reiz unabhängig
Aminosäuren: Die Bausteine der Proteine (Eiweiße) mit der allgemeinen Formel $C(H)(NH_2)(COOH)(Rest)$
Antigene: Bestimmte Stoffgruppen an der Oberfläche von Zellen und Viren, die wie „Etiketten“ die Unterscheidung von körpereigenen und körperfremden Substanzen erlauben
Antiport (countertransport): In Gegenrichtung verlaufender, gekoppelter Transport von Stoffen durch die Zellmembran hindurch
Axon (Neurit): Nervenfaser

Axonhügel: Ursprungsstelle des Axons am Neuron, Entstehungsort des Aktionspotentials
Dendriten, Dendritenbaum: Fortsätze von Nervenzellen, an denen die Synapsen anderer Nervenzellen enden
Depolarisation, depolarisieren: Von außen erzwungene Erniedrigung eines (Membran-) Potentials
Donnan-Verteilung, Donnan-Gleichgewicht: Effekt der Verschiebung von Ionenkonzentrationen in durch Membranen getrennten Flüssigkeiten durch die Anwesenheit eines impermeablen Ions
Eiweiß (Protein): Wesentlicher Bestandteil aller Körpersubstanzen
Elementarmebran: Membranen kommen an vielen Stellen in Zellen vor; die Zellmembran ist nur eine davon (siehe auch „Fluid-Mosaic-Modell“)
Endoplasmatisches Retikulum: Reaktionsräume für Stoffwechselvorgänge in der Zelle, von einer Elementarmembran umgeben
Endplatte, motorische: Synapse an einer Muskelzelle
Enzyme: Biokatalysatoren, beschleunigen oder ermöglichen bestimmte chemische Reaktionen durch ihre Anwesenheit, ohne sich dabei zu verbrauchen; bestehen hauptsächlich aus Eiweiß
Ester: Unter Wasserabspaltung entstehende chemische Verbindung eines Alkohols und einer Säure
Exozytose: Zellausscheidung
Fette: Ester aus Glyzerin und drei Fettsäuren
Fettsäuren: In Fetten vorkommende organische Säuren, meist langkettige Kohlenwasserstoffe mit einer Säuregruppe (-COOH) an einem Ende
Fluid-Mosaic-Modell: Modell der Elementarmembran, bestehend aus zwei (inneren) halb flüssigen Lipidschichten ("lipid bilayer") und zwei (äußeren) Schichten von Proteinen, die auf der Lipidphase „schwimmen“
Glykokalix: Zuckermoleküle auf der Zelloberfläche, dienen als Antigene
Glyzerin: Dreiwertiger Alkohol ($CH_2(OH)$-$CH(OH)$-$CH_2(OH)$)
hydrophil („wasserfreundlich“): Wasseranziehend, da elektrisch geladen (polar)
hydrophob („wasserfürchtend“): Wasserabstoßend, da elektrisch neutral (unpolar)
Immunreaktion: Abwehr des Körpers gegen körperfremde Stoffe und Organismen
Lipide: Verwandte der Fette, Bausteine der Elementarmembran
Kotransport: Gekoppelter Transport von Stoffen durch die Zellmembran hindurch
lipophil („fettfreundlich“): Fettanziehend, da elektrisch neutral (unpolar)
lipophob („fettfürchtend“): Fettabstoßend, da elektrisch geladen (polar)
Membranpotential: Elektrisches Potential der Zelle gegenüber dem Außenmedium, durch die Membran aufrechterhalten
Membranproteine: (Siehe *Fluid-Mosaic-Modell* im Text)

- periphere
- integrale
- Trans-

Mitochondrium: Zellorgan zur Energiegewinnung durch Oxidation organischer Substanzen (in der Regel Glukose (Traubenzucker), bei Mangel daran auch Eiweiße)
Natrium-Kalium-Pumpe: Integrale Membranproteine zur Aufrechterhaltung des Membranpotentials durch Austausch von Natrium- gegen Kaliumionen in der Zelle
Neurit (Axon): Nervenfaser
Neuron: Nervenzelle
Neurotransmitter: Chemischer Botenstoff zur Erregungsübertragung
Permeabilität: Durchlässigkeit
Plasmalemma (Zellmembran): Zellbegrenzung
polar: Ungleichverteilung elektrischer Ladungen in einem Molekül oder elektrische Nettoladung desselben
Polysomen (Ribosomen): Dienen zur Synthese der zelleigenen Proteine
Potential: (Siehe jeweils dort)

- Membran-
- Ruhe-
- Rezeptor-
- Aktions-
- Schrittmacher-

Protein: Eiweiß, Ketten aus Aminosäuren
Refraktärzeit: Phase nach einem Aktionspotential, in der kein weiteres Aktionspotential ausgelöst werden kann
Rezeptorpotential: Differenz zwischen dem Ruhe- und dem Membranpotential bei Reizung
Ribosomen (Polysomem): Dienen zur Synthese der zelleigenen Proteine
Ruhepotential: Membranpotential der Zelle in Ruhe, -50 bis -200 mV
Schrittmacher- (pacemaker-) Potential: Periodisch schwankendes Membranpotential, das zur Erzeugung autonomer Rhythmen dient
Soma: Zellkörper
Synapse: Das verdickte Ende eines Axons, mit dem eine Nervenzelle die Verbindung zu einer nachfolgenden Zelle herstellt, über die Erregungen weitergereicht werden
synaptische Bläschen (Vesikel): Enthalten den Neurotransmitter
synaptischer Spalt: Der enge Zwischenraum zwischen Synapse und Folgezelle
Triglyzeride: Dreifache Ester des Glyzerins
unpolar: Elektrische Neutralität eines Moleküls; keine elektrische Nettoladung bzw. gleichmäßige elektrische Ladungsverteilung
van-der-Waals-Kräfte: Bindungskräfte zwischen unpolaren Molekülen, hervorgerufen durch fortwährende Schwankungen in der Ladungsverteilung in den Molekülen, die Ladungsverschiebungen in den benachbarten Molekülen induzieren und dadurch zu (schwacher) elektrischer Anziehung führen
Verhalten: (von Sinnes- und Nervenzellen; siehe Text)

- phasisches
- tonisches

- phasisch-tonisches

Vesikel (Bläschen): Membranumschlossene Behältnisse für Stoffwechselprodukte in Zellen
Zellkern: Dient der Steuerung der Zelle, enthält die Erbanlagen
Zellmembran (Plasmalemma): Zellbegrenzung
Zellorganellen: Die „Organe“ der Zelle
Zellplasma: Zellinneres
Zytoplasma: Zellplasma
Zytosol: Zellgrundsubstanz

Kapitel 2

Einführung

Theorie konnektionistischer Modelle

2.1 Funktionsweise von Neuronen im Modell

Die komplexen Vorgänge in natürlichen Nervensystemen, wie sie im Kapitel 1 jeweils kurz angesprochen wurden, finden in Konnektionistischen Modellen[1] nur bedingt ihre Entsprechung. Viele Eigenschaften natürlicher Nervensysteme bleiben unberücksichtigt, manche werden in den Modellen nur in stark vereinfachter (abstrahierter) und formalisierter Form verwirklicht.

In Abbildung 2.1 (Seite 22) ist ein Schema zweier miteinander verbundener Neurone dargestellt.

Die beiden Zellkörper werden jeweils durch die beiden großen Kreise dargestellt. An ihrer linken Seite befinden sich mehrere kleine Kreise, welche die Rezeptoren an den Stellen symbolisieren, an denen Axone (im Schema Linien) von anderen Neuronen mit dem Dendritenbaum dieser Nervenzelle Synapsen bilden (dargestellt durch die annähernd dreieckigen Endigungen der Axone). Das größere Dreieck an der jeweils rechten Seite der beiden Nervenzellen stellt den Axonhügel dar, an dem das Axon der betreffenden Nervenzelle entspringt, das sich anschließend in mehrere Fasern aufteilt, die jeweils mit anderen Nervenzellen Synapsen bilden.

In konnektionistischen Modellen spricht man statt von Neuronen oder Nervenzellen von *Units* („Einheiten"). Statt von Axonen und Synapsen spricht man einfach von *Verbindungen* zwischen zwei Units, die als gerichtet angesehen werden („*Gleichrichterfunktion der Synapsen*", siehe Kapitel 1). Formal lassen sich Neuronale Netze also als Graphen mit gerichteten Kanten auffassen.

Ein an einer Synapse eintreffendes Aktionspotential löst durch Ausschüttung des Neurotransmitters in den synaptischen Spalt in der Folgezelle eine Veränderung des Membranpotentials aus. Je nach der Menge des ausgeschütteten Neurotransmitters und der Menge der vorhandenen Rezeptoren auf der Seite der Folgezelle ist die Größe dieser Potentialänderung verschieden. Alle Faktoren, die die Größe der Potentialänderung je Aktionspotential im

[1] Um möglichen Verwirrungen vorzubeugen, weise ich hier darauf hin, daß ich im Folgenden die Begriffe „Konnektionistisches Modell" und „Neuronales Netz" synonym verwende.

Abbildung 2.1: Schema zweier Neurone

natürlichen System beeinflußen, werden im Modell in einer einzigen Zahl, *Gewicht* genannt, zusammengefaßt. Mögliche zeitliche Abhängigkeiten bzw. gegenseitige Beeinflussungen mehrerer Aktionspotentiale hintereinander bleiben dabei unberücksichtigt.

Zu jeder Synapse, d.h. zu jeder Verbindung zwischen zwei Units im Modell gehört ein solches Gewicht. Das Gewicht der Verbindung von einer Unit i zu einer Unit j wird mit w_{ji} bezeichnet[2]. Während im natürlichen System Informationen (Reizintensitäten) kodiert durch die Frequenz von Aktionspotentialen über das Axon von Zelle zu Zelle übermittelt werden, wird im Modell über die Verbindung zwischen zwei Units jeweils nur eine einzige Zahl (die der Reizintensität entspricht) weitergegeben. Der Größe der durch einen Reiz hervorgerufenen Änderung des Membranpotentials der Folgezelle entspricht im Modell das Produkt aus Reizintensität und dem Gewicht der betreffenden Verbindung.

Alle an den Dendriten einer Nervenzelle eintreffenden Signale werden miteinander verrechnet. Wie im Kapitel 1 erläutert wurde, handelt es sich dabei im Wesentlichen um die elektrotonische Überlagerung (Summation) aller durch die Synapsen hervorgerufenen Änderungen des Membranpotentials. Das Resultat dieser Überlagerung, die effektive Änderung des Membranpotentials, wird als das Rezeptorpotential bezeichnet.

Man vermutet, daß eintreffende Signale teilweise auch auf andere, komplexere Weise miteinander verrechnet werden als durch Summation. Ein Beispiel ist die multiplikative Verknüpfung zweier oder mehrerer Signale, die allerdings auch durch Logarithmierung der Reizintensitäten in den Synapsen und anschließender Summation (Überlagerung) an der Membran der Folgezelle realisiert sein kann.

[2]Die Vertauschung der Indizes hängt mit der später einzuführenden Matrixnotation zusammen.

Im konnektionistischen Modell wird diese Verrechnung durch eine Funktion g realisiert, deren Argumente die Produkte aus der Reizintensität des eintreffenden Signals und dem Gewicht der betreffenden Verbindung sind. Diese Funktion kann für verschiedene Units unterschiedlich sein, weswegen man die Nummer der betreffenden Unit in den Index schreibt. Das Ergebnis der Verrechnung der eintreffenden Signale wird mit "net" („*Netzeingabe*") bezeichnet:

$$\mathrm{net}_j \quad = \quad g_j\,(\; w_{ji_1} \cdot o_{i_1} \;,\; w_{ji_2} \cdot o_{i_2} \;,\; \ldots \;,\; w_{ji_n} \cdot o_{i_n} \;) \tag{2.1}$$

Dabei bezeichnet o_i die Ausgabe (den "output") der Unit i, w_{ji} das Gewicht der Verbindung von Unit i zur Unit j und n die Anzahl der Verbindungen, die zur Unit j hinführen. (Vergleiche auch Abbildung 2.1 Seite 22)

In aller Regel wird als Verrechnungsfunktion g die Summe der Argumente verwendet:

$$\mathrm{net}_j \quad = \quad \sum_{k=1}^{n} w_{ji_k} \cdot o_{i_k} \tag{2.2}$$

Mehr zu anderen Verrechnungsfunktionen siehe beispielsweise unter dem Stichwort "*Sigma-Pi-Units*" in Rumelhart/McClelland [24] (Kapitel 2) oder unter "*Product Units*" in Durbin/Rumelhart [5].

Das Rezeptorpotential breitet sich passiv über den gesamten Zellkörper bis zum Axonhügel hin aus (vergleiche Abbildung 1.13 (Seite 15) im Kapitel 1). Im Modell entspricht das sich ausbreitende Rezeptorpotential einer sogenannten *Aktivierung* der Unit, die aus der Netzeingabe mit Hilfe einer *Aktivierungsfunktion* F berechnet wird. Diese Funktion kann für verschiedene Units unterschiedlich sein, weswegen man die Nummer der betreffenden Unit in den Index der Funktion schreibt. Die Aktivierung einer Unit j wird mit a_j bezeichnet:

$$a_j \quad = \quad F_j\,(\; \mathrm{net}_j \;) \tag{2.3}$$

Die Aktivierungsfunktion kann außerdem von der vorhergehenden Aktivierung abhängen:

$$a_j\,(\mathrm{t{+}1}) \quad = \quad F_j\,(\; a_j\,(\mathrm{t}) \;,\; \mathrm{net}_j\,(\mathrm{t}) \;) \tag{2.4}$$

Mehr über Aktivierungsfunktionen siehe im Abschnitt 2.6.

Übersteigt die am Axonhügel durch das Rezeptorpotential hervorgerufene Depolarisation der Membran einen bestimmten Schwellenwert, werden ein oder mehrere Aktionspotentiale erzeugt, deren Frequenz (d.h. deren reziproker Wert ihres zeitlichen Abstandes voneinander) der Reizintensität entspricht. In konnektionistischen Modellen werden von den Units dagegen keine Aktionspotentiale ausgegeben, sondern eine Zahl, deren Wert der Reizintensität entspricht. Dieser Ausgabewert ("output") wird aus der Aktivierung der Unit mit Hilfe einer *Ausgabefunktion* h berechnet, die von Unit zu Unit auch verschieden sein kann. Die Ausgabe (der Ausgabewert) einer Unit j wird dabei mit o_j bezeichnet:

$$o_j \quad = \quad h_j\,(\; a_j \;) \tag{2.5}$$

Häufig verwendete Ausgabefunktionen sind Schwellwertfunktionen, wie z.B. die folgende, aus einem diskreten Modell:

$$h\,(x) \quad = \quad \begin{cases} 1 & \text{falls} \quad x \;>\; 0 \\ 0 & \text{falls} \quad x \;\leq\; 0 \end{cases} \tag{2.6}$$

In analogen Modellen gibt es viele Möglichkeiten, eine Schwellwertfunktion zu definieren. Eine davon lautet:

$$h(x) = \begin{cases} x & \text{falls} \quad x > 0 \\ 0 & \text{falls} \quad x \leq 0 \end{cases} \tag{2.7}$$

Eine weitere sehr häufig verwendete Ausgabefunktion ist die Identität, da man oft die Aktivierungs- und die Ausgabefunktion in einer einzigen Funktion, meist der Aktivierungsfunktion, zusammenfaßt, so daß gilt:

$$o_j = a_j \tag{2.8}$$

Man beachte, daß bei *Eingabeunits* (mehr dazu siehe in den folgenden Abschnitten) die Verrechnungsfunktion g und die Aktivierungsfunktion F entfallen, da die Aktivierung a dieser Units von außen gesetzt wird. Aus Gründen der Vereinfachung bzw. der Praktikabilität verzichtet man in konnektionistischen Modellen auf die Simulation der Analoga von Sinneszellen, die im natürlichen System die Eingaben des Netzwerks liefern.

2.2 Verbindungsstrukturen und Gewichte

Durch Axone und Synapsen sind Nervenzellen im Nervensystem zu einem großen Geflecht, einem „Netzwerk“, verbunden. Im menschlichen Gehirn sind nach Schätzungen mehrere hundert Millionen Nervenzellen vorhanden. Jede dieser Nervenzellen besitzt auf ihren Dendriten im Schnitt etwa zehntausend Synapsen, durch die sie Signale von anderen Nervenzellen empfängt.

Die Nervenzellen des Nervensystems entsprechen in konnektionistischen Modellen den Knoten eines Graphen, “Units” genannt, die durch gewichtete und gerichtete Kanten miteinander verbunden sind (vergleiche voriger Abschnitt).

Zur Beschreibung der Verbindungsstruktur eines solchen Graphen bedient man sich einer (quadratischen) Matrix, deren Zeilen und Spalten den Knoten des Graphen entsprechen. Diese Matrix wird *Gewichtsmatrix* genannt und mit W (von “weight”) bezeichnet. Der Eintrag w_{ji} der Matrix W enthält das Gewicht der Kante vom Knoten i zum Knoten j[3]. Da die Kanten gerichtet sind, ist die entstehende Matrix nicht notwendig symmetrisch, d.h. obere und untere Dreiecksmatrix weisen im Allgemeinen verschiedene Einträge auf. Ein Eintrag mit dem Wert Null zeigt das Fehlen der entsprechenden Kante an. Dadurch können beliebige Verbindungsstrukturen realisiert werden, die sich außerdem sehr leicht verändern lassen.

Da im menschlichen Gehirn jede Nervenzelle nur etwa zehntausend Synapsen besitzt, ist offensichtlich, daß nicht jede Nervenzelle mit jeder anderen verbunden sein kann. Auch in konnektionistischen Modellen ist ein vollständig verbundenes Netz (ein vollständiger Graph), in dem jede Unit mit jeder anderen (in beiden Richtungen) verbunden ist, bei einer geringen Anzahl von Units zwar noch realisierbar, aber dennoch eine relativ seltene Ausnahme.

[3] Diese Definition hat die Eigenschaft, daß bei Verwendung von Units mit der Identität als Aktivierungs- und Ausgabefunktion die Abbildung des Netzes von den Eingabe- auf die Ausgabevektoren gerade der Multiplikation der Gewichtsmatrix mit dem jeweiligen Eingabevektor entspricht.

Je nach ihrer Verbindungsstruktur teilt man konnektionistische Modelle in mehrere systematische Klassen ein.

Eines der wichtigsten Unterscheidungsmerkmale ist die *Zykelfreiheit* des entsprechenden Graphen (vergleiche Graphentheorie). Netzwerke, die Zykeln enthalten, bezeichnet man als *rekurrent*, zykelfreie Netze als *nicht rekurrent*.

Nicht rekurrente Netzwerkmodelle sind statisch, d.h. ihre Ausgabe hängt allein von der momentanen Eingabe ab, während die Ausgabe von rekurrenten Netzen im Allgemeinen auch von allen vorhergehenden Eingaben abhängt; rekurrente Netze verhalten sich dynamisch.

Abbildung 2.2 (Seite 25) zeigt ein solches rekurrentes Netz mit mehreren Zykeln: 6-6, 4-7-4, 1-4-5-1, 1-2-5-1, 4-5-6-4, 4-5-7-4, 4-5-6-7-4, 1-2-3-5-1, ...

Abbildung 2.2: Schema eines (beliebigen) Neuronalen Netzes

Ein Verbot aller Zykeln betrifft auch alle Verbindungen einer Unit mit sich selbst, wie im Falle der Unit #6. Zykelfreie Graphen enthalten daher in der Diagonalen ihrer Verbindungsmatrix stets Nullen.

Drei Typen von Knoten (Units) lassen sich in dieser Abbildung unterscheiden: Einmal solche, in die von außerhalb des Netzes Kanten hineinführen (1, 2, 3); diese werden als die *Input*- oder *Eingabe*-Units des Netzes bezeichnet. Zum Zweiten gibt es Knoten, von denen Kanten nach außen führen (1, 4, 7); diese werden als die *Output*- oder *Ausgabe*-Units des Netzes bezeichnet. Und schließlich gibt es noch Knoten, die in keine dieser beiden Kategorien gehören, die keinerlei Verbindungen nach oder von außerhalb des Netzes besitzen (5, 6). Deren Ein- und Ausgaben sind außerhalb des Netzes nicht sichtbar, sie werden daher als "*hidden*" (versteckte) Units bezeichnet.

In Analogie zum Gehirn, in dem sich verschiedene anatomische Schichten deutlich unterscheiden lassen, werden in den meisten Netzwerkmodellen die Units ebenfalls in Schichten, auch "*layer*" genannt, angeordnet. Verschiedene Einschränkungen in der Verbindungsstruktur schaffen hierbei eine Vielzahl unterschiedlicher Klassen von Modellen (vergleiche besonders Rumelhart/McClelland [24] Kapitel 2):

Eine sehr häufig anzutreffende Einschränkung, die besonders bei Verwendung von Back-Propagation als Lernverfahren die Berechnung vereinfacht, läßt keine Verbindungen zwischen den Units aus ein- und derselben Schicht zu (auch dafür gibt es Entsprechungen im Gehirn). In anderen Modellen wie z.B. "Competitive Learning" (siehe Rumelhart/McClelland [24] Kapitel 5) sind jedoch gerade diese Verbindungen innerhalb einer Schicht wesentlich für die Eigenschaften des Modells (siehe auch unter dem Stichwort „*laterale Inhibition*“ in Linder [16], Eckert/Randall [6], Schmidt/Thews [28], Schmidt [27] u.a.).

Eine weitere sehr wichtige systematische Einschränkung besteht darin, diese Schichten hierarchisch anzuordnen und Verbindungen nur von Units aus niedrigeren Schichten hin zu Units in höheren Schichten zu erlauben.

Eine Verschärfung dieser hierarchischen Ordnung läßt Verbindungen nur von den Units einer Schicht zu den Units der nächst höheren (also benachbarten) Schicht zu.

In der Mehrzahl der Fälle sind dabei die Units zweier benachbarter Schichten vollständig vernetzt, d.h. jede Unit der einen ist mit jeder Unit der anderen Schicht verbunden. Es wird jedoch auch mit Modellen experimentiert, die eine niedrigere Konnektivität besitzen, so daß z.B. jede Unit der niedrigeren Schicht Verbindungen zu genau zwei oder einer anderen festen Anzahl von Units der nächstfolgenden Schicht entsendet.

Die niedrigste Schicht in der Hierarchie besteht aus den Eingabeunits, die höchste Schicht aus den Ausgabeunits des Netzes. Alle dazwischenliegenden Schichten, falls vorhanden, werden als "*hidden layers*" (versteckte Schichten) bezeichnet, da ihre Ein- und Ausgaben außerhalb des Netzes nicht sichtbar sind.

Theoretisch können Ein- und Ausgabeunits zusätzlich auch in anderen Schichten vorkommen; eine Unit kann sogar gleichzeitig sowohl Ein- als auch Ausgabeunit sein. Dies kommt jedoch in der Praxis in hierarchischen Modellen nicht vor.

Abbildung 2.3 (Seite 27) zeigt ein solches hierarchisches Netz[4], dessen Struktur kennzeichnend für die Modelle ist, die Back-Propagation als Lernverfahren (mehr darüber im Abschnitt 2.8) verwenden, d.h., das in dieser Abbildung gezeigte Netzwerk ist die Grundlage, auf der alle im weiteren Verlauf besprochenen Verfahren beruhen. (Auf die speziellen Units I_0 und H_0 komme ich im Abschnitt 2.6 zurück)

[4]auch "Feed-Forward Network" genannt

Abbildung 2.3: Schema des im Folgenden verwendeten Netzwerktyps

Während man von den Units auf einer Ebene des Netzes als einer „Schicht“ spricht, werden zwei benachbarte Schichten mit den zugehörigen Verbindungen zwischen ihnen auch als eine „*Stufe*“ bezeichnet. Das Netzwerk aus Abbildung 2.3 besitzt also drei Schichten bzw. zwei Stufen. (Aufgrund dieser Definition gilt stets, daß die Anzahl der Stufen eines hierarchischen Netzes um Eins kleiner ist als die Anzahl der Schichten)

Man beachte, daß das Back-Propagation-Verfahren auch für mehr als nur zwei Stufen geeignet ist (vergleiche Abschnitt 2.8) und daß einige Autoren in ihren Simulationen auch tatsächlich mehr als nur einen "hidden layer" verwenden. Hierarchische Netze mit nur zwei Stufen sind jedoch leichter zu simulieren, da weniger Speicherplatz und weniger Rechenzeit benötigt wird. Auch sind zweistufige Netze insofern „vollständig“, als daß mit ihnen bereits alle *booleschen* Funktionen darstellbar[5] sind, analog zur Berechnung in einem PLA (Programmable Logic Array) nach der Disjunktiven Normalform (DNF). (Die allgemeine Konstruktionsvorschrift zur Darstellung von booleschen Funktionen mit zweistufigen Neuronalen Netzen siehe in Denker et al. [4])

Es gibt allerdings durchaus Funktionen, die nicht durch zweistufige Netzwerke berechnet werden können. (siehe Verweis in Denker et al. [4])

Es hat sich eingebürgert, die Struktur eines hierarchischen Netzwerks abkürzend durch die Anzahl der Units in den einzelnen Schichten zu beschreiben: Ein 11–8–4 - Netz ist ein Netz mit 11 Input Units, 8 "hidden" Units und 4 Output Units; ein 1–20–20–1 - Netz ist ein Netz mit je einer Input- und Output-Unit sowie je 20 "hidden" Units in zwei "hidden layers". Man beachte, daß die speziellen Units I_0 und H_0 (bzw. H_0^1 und H_0^2 im letzteren Fall) nicht mitgezählt werden.

Da Verbindungen nur jeweils zwischen benachbarten Schichten existieren, enthält die Verbindungsmatrix eines hierarchischen Netzes wie in Abbildung 2.3 sehr viele Nulleinträge. Speichert man die Gewichte des Netzes in der Simulation in Form einer Matrix, wird unnötig Speicherplatz verschwendet, der bei großen Netzen beträchtlich sein kann. Es ist daher zweckmäßig, separate Matrizen für die einzelnen Stufen einzuführen. Im Folgenden sowie in meinem Programm werden diese zwei Matrizen (da nur zweistufige Netze simuliert werden) mit W_{ji} (für die erste Stufe von den Input- zu den "hidden" Units) und mit W_{kj} (für die zweite Stufe von den "hidden" zu den Output-Units) bezeichnet.

Man beachte, daß es für die Benennung (Durchnumerierung) von Gewichten mehrere Alternativen gibt, die sowohl in der Literatur als auch in dieser Arbeit gleichwertig nebeneinander verwendet und je nach Bedarf gewechselt werden:

- Die erste und einfachste Möglichkeit besteht in der fortlaufenden Durchnumerierung aller Gewichte. Die Gewichte werden dabei auch als Komponenten eines Gewichtsvektors interpretiert. In dieser Schreibweise haben alle Gewichte nur einen einzigen Index; in allgemeinen Ausdrücken in der Regel ein „i“. Wenn dieser Index für die Darstellung eines bestimmten Sachverhalts unwesentlich ist, wird er oft auch weggelassen.

- Bei der zweiten Möglichkeit numeriert man die Gewichte entsprechend den Zeilen und Spalten der Verbindungsmatrix des *gesamten* Netzes. In dieser Konvention hat jedes Gewicht zwei Indizes, die seiner Zeilen- und Spaltennummer in der Verbindungsmatrix entsprechen; in allgemeinen Ausdrücken werden in der Regel die Indizes „i“ und „j“ verwendet.

[5] Dies ist keine Aussage darüber, ob das Netz diese Funktionen im Einzelfall auch lernen kann!

- Die dritte Möglichkeit besteht in der Durchnumerierung der Gewichte entsprechend den Zeilen und Spalten der Verbindungsmatrizen *der einzelnen Stufen*. Bei dieser Variante haben alle Gewichte ebenfalls zwei Indizes, die der Zeilen- und Spaltennummer *in der betreffenden Verbindungsmatrix* entsprechen. Hier werden als Indizes „i" und „j" für die Gewichte der ersten Stufe, „j" und „k" für die Gewichte der zweiten Stufe verwendet.

2.3 Verarbeitung von Daten in Neuronalen Netzen

Im vorangegangenen Abschnitt wurde beschrieben, wie sich die elementaren Verarbeitungseinheiten (vergleiche Abschnitt 2.1) eines konnektionistischen Modells zu größeren Einheiten, zu sogenannten „Netzwerken", zusammenschalten lassen.

Mit Hilfe dieser Netzwerke können Daten verarbeitet werden: Jede Unit stellt einen kleinen Prozessor dar, der nach relativ einfachen Regeln (mathematischen Funktionen) ankommende Daten verrechnet und das Ergebnis dieser Berechnungen an nachfolgende Units weiterreicht (siehe Abschnitt 2.1). Auf dem Wege von den Eingabe- zu den Ausgabeunits können daher im Prinzip alle nach der Church'schen These[6] berechenbaren Funktionen realisiert werden.

In der Praxis beschränkt man sich meist auf wenige verschiedene Typen von Units mit relativ einfachen Funktionen; die Summation zur Verrechnung der ankommenden Daten, die Identität als Ausgabefunktion und einige wenige verschiedene Aktivierungsfunktionen (siehe Abschnitt 2.6).

Die Gründe hierfür liegen darin, daß man vom herkömmlichen Berechnungsmodell eines John-von-Neumann-Rechners (mit serieller Verarbeitung nach dem Prinzip "single data, single instruction") loskommen möchte, hin zu einer „*verteilten Verarbeitung*" wie in natürlichen Nervensystemen, in der Daten parallel verarbeitet werden und in dem Informationen wie in einer Holografie[7] *verteilt gespeichert* sind: Durch Ausfall einer Zelle oder einer Verbindung wird nicht das gesamte Netz funktionsuntüchtig, sondern die Funktionsfähigkeit verschlechtert sich nur (die Ausgaben streuen weiter als vorher um die jeweils „richtige" Ausgabe herum), und gespeicherte Informationen gehen nicht verloren, sondern werden nur „unschärfer". (Man nennt diese Eigenschaft natürlicher Systeme auch "*graceful degradation*", was man ungefähr mit „gleitender, progressiver Verfall" übersetzen könnte)

In natürlichen Systemen scheint darüber hinaus vielfach, im Gegensatz zu den herkömmlichen Rechnern, die Trennung von Wissen in prozedurales und lexikalisches Wissen verwischt zu sein, was eine wesentliche Grundlage für die Flexibilität und Leistungsfähigkeit der natürlichen kognitiven Prozesse zu sein scheint. Erste Forschungsergebnisse in dieser Richtung, mit Hilfe von rekursiven verteilten Repräsentationen, die letztlich sogar selbstähnlich (eine Eigenschaft aller Fraktale) werden und damit die Brücke zur Chaos-Theorie schlagen, ist in Pollack [22] zu finden.

Diese verteilte, parallele Verarbeitung besitzt große Vorteile, wie man an der Leistungsfähigkeit des Gehirns ermessen kann: Wenn man in einem Versuch einer Versuchsperson

[6] Die Church'sche These besagt, daß alle im intuitiven Sinne berechenbaren Funktionen (u.a.) durch eine Turing-Maschine berechnet werden können.

[7] Jede Teilfläche einer Holografie enthält das gesamte Bild. Zerschneidet man die Holografie, läßt sich aus jedem Teil wieder das vollständige Bild rekonstruieren; je kleiner das Teil, desto unschärfer ist jedoch das wiedergewonnene Bild.

verschiedene Gesichter zeigt und die Zeitspanne mißt, bis das Gesicht entweder erkannt ist oder als unbekannt eingeordnet wird (durch Drücken eines entsprechenden Knopfes), dann stellt man fest (durch Division der mittleren Reaktionszeit durch die Zeit, die ein Signal benötigt, um von einer Nervenzelle zur nächsten zu gelangen), daß das Signal auf dem Wege von der Netzhaut über den Sehnerven zum Gehirn und von dort über das Rückenmark bis zum ausführenden Muskel (zum Drücken des Knopfes) nur etwa 100 Neurone durchläuft, d.h., daß die gesamte, komplexe Berechnung zur Musterwiedererkennung weniger als 100 Schritte benötigt!

Im natürlichen System ist jede Nervenzelle eine autonome Einheit. Im Gegensatz zu einem Rechner gibt es keinen zentralen Takt, der die Aktivität aller Nervenzellen des Nervensystems miteinander synchronisiert; insbesondere entstehen auch Aktionspotentiale in verschiedenen Nervenzellen zeitlich unabhängig voneinander. Daß dadurch keine Aktionspotentiale „verlorengehen" oder „verpaßt" werden, liegt an der Speicherung eintreffender Aktionspotentiale im Membranpotential der Folgezelle, das erst nach einiger Zeit seinen Ruhewert wiedererreicht (falls keine weiteren Aktionspotentiale folgen).

Da konnektionistische Modelle bisher in der Regel auf herkömmlichen Rechnern simuliert werden, läßt sich die Autonomie von Nervenzellen nur unzureichend nachbilden, da alle Berechnungen an den zentralen Takt gebunden erfolgen (dies gilt selbst noch für die Hardware-Realisierungen in VLSI-Technik, nicht jedoch für Realisierungen in Analogtechnik).

Man unterscheidet konnektionistische Modelle mit *synchronem* und solche mit *asynchronem Update*. Beim synchronen Update werden periodisch die Aktivierungs- und Ausgabewerte *sämtlicher* Units des Netzes *gleichzeitig* neu berechnet. Dazu werden die Ein- und Ausgaben aller Units festgehalten, woraufhin die neuen Aktivierungen und Ausgaben aller Units nacheinander (bei Simulation auf einem sequentiellen Rechner) oder gleichzeitig (bei Simulation auf einem parallelen Multiprozessor-System) berechnet werden. Anschliessend werden die neuen Ausgaben über die Verbindungen des Netzes weitergeleitet (man sagt *„propagiert"*).

Beim asynchronen Update werden die Aktivierungen und Ausgaben der Units des Netzes *nacheinander* neu berechnet: Man hält die Ein- und Ausgaben *einer* Unit fest, berechnet ihre Aktivierung und ihre Ausgabe neu und propagiert anschließend den neuen Ausgabewert zu den nachfolgenden Units. Die Unit, deren Aktivierung und Ausgabe neu berechnet werden soll, wird entweder zufällig ausgewählt (stochastischer asynchroner Update), oder alle Units werden in einer bestimmten Reihenfolge angeordnet, die dann durchlaufen wird (sequentieller asynchroner Update).

Den Update in hierarchischen Netzen wie in Abbildung 2.3 (Seite 27) kann man als eine Mischform aus synchronem und asynchronem Update ansehen: Hier werden zuerst die Aktivierungen der Eingabeunits (und damit gleichzeitig ihre Ausgaben) von außen gesetzt, anschließend werden die Aktivierungen und Ausgaben der "hidden" Units (der mittleren Schicht) berechnet und zuletzt die Aktivierungen und Ausgaben der Ausgabeschicht. Die Aktivierungen und Ausgaben der Units *einer* Schicht werden also (im Prinzip) *gleichzeitig* (synchron) berechnet (wenn auch auf einem seriellen Rechner in Wirklichkeit nacheinander), die der Units in den *verschiedenen* Schichten jedoch *nacheinander* (asynchron).

Dies ist allerdings gleichwertig damit, die Units in der Reihenfolge Inputunits, "hidden" Units und Outputunits durchzunumerieren und die Aktivierungen und Ausgaben der Units dann in dieser Reihenfolge zu berechnen, was einem rein sequentiellen asynchronen Update entspricht.

Einen Durchgang dieser Art, bei dem die Aktivierungen und Ausgaben aller Units des Netzes für eine Eingabe neu berechnet werden, bezeichnet man als einen "*Propagation*"-Schritt oder einen "*Forward-Pass*".

2.4 Lernen in Neuronalen Netzen

Lernen in Neuronalen Netzen bedeutet die Anpassung der Abbildung von den Netzeingaben auf die Netzausgaben, die das Netzwerk realisiert, an eine bestimmte Aufgabe oder eine gewünschte Abbildung.

Im Prinzip kann diese Anpassung auf zwei Wegen erreicht werden. Der direkte Ansatz besteht darin, die Funktionen (die Verrechnungs-, die Aktivierungs- und die Ausgabefunktion) in den einzelnen Units zu verändern. Der zweite, mehr indirekte Ansatz besteht darin, die Gewichte der Verbindungen innerhalb des Netzes zu modifizieren (und die Funktionen in den Units unverändert zu lassen).

Obwohl der erste Ansatz intuitiv effizienter erscheint, hat er den Nachteil, daß sich die Suche im Raum aller möglichen Funktionen nicht automatisieren läßt.

Die Automatisierbarkeit ist deswegen wichtig, weil man mit Hilfe von Neuronalen Netzen das selbständige (d.h. automatische) Lernen von bestimmten Zuordnungen zwischen Mengen von Ein- und Ausgabetupeln anhand von Beispielen realisieren möchte: Das Netzwerk soll selbständig aus der Menge der Beispiele die zugrundeliegenden Regeln der jeweiligen Zuordnung extrahieren, ohne daß diese explizit vorgegeben werden.

Dies ist besonders in den Fällen von entscheidender Bedeutung, in denen die einer bestimmten Abbildung zugrundeliegenden Regeln gar nicht bekannt sind.

Ein sehr einfaches Beispiel ist die Erkennung der (handschriftlichen) Ziffern von 0 bis 9: Es gibt keine Regel, welche die Zuordnung eines handschriftlichen Zeichens zu einer der zehn Ziffern festlegt. Diese Zuordnung ist allein anhand von Beispielen definierbar und nur durch Einordnung nach dem Kriterium des geringsten Abstandes möglich.

Obwohl die Anpassung der Funktionalität eines Netzwerks durch Modifikation der Gewichte seiner Verbindungen auf den ersten Blick unhandlich erscheinen kann, ist diese Methode sehr effektiv (wenn auch nicht unproblematisch). Sie bildet außerdem das Analogon zu den Grundlagen, auf denen in natürlichen Systemen das Lernen wahrscheinlich beruht (Vermutungen der Neurophysiologen zufolge). Ein zweistufiges Netzwerk wie in Abbildung 2.3 (Seite 27) mit einer sigmoiden Aktivierungsfunktion mit Schwellenwert (siehe Abschnitt 2.6), der Summation als Verrechnungsfunktion und der Identität als Ausgabefunktion stellt einen *universellen Approximator* dar, der bei geeigneter Wahl der Gewichte des Netzes jede beliebige stetige Funktion zu approximieren vermag. (Vergleiche auch Linden [15])

Beim Lernen oder „Trainieren" eines Neuronalen Netzes unterscheidet man zwei verschiedene Vorgehensweisen: Zum Einen das "*Supervised Learning*" („Überwachtes Lernen"), das so genannt wird, weil ein (fiktiver) „Lehrer" für jede Netzwerkeingabe die gewünschte Netzwerkausgabe vorgibt, das Lernen also „überwacht", sowie zum Anderen das "*Unsupervised Learning*" („nicht überwachtes Lernen"), bei dem das Netzwerk seine Netzausgabe selbst bestimmt.

Letzteres Verfahren hat den Vorteil, daß die Lernregeln im Allgemeinen einfacher sind als beim "Supervised Learning"; und den Nachteil, daß die Netzwerkausgaben verschieden sein können (sowohl funktionell verschieden als auch Permutationen voneinander), wenn man die Gewichte des Netzes reinitialisiert und das Netz von neuem trainiert, d.h., die Netzausgaben bedürfen der Interpretation.
(Siehe z.B. "Competitive Learning" in Rumelhart/McClelland [24] Kapitel 5)

Lernverfahren werden in den Abschnitten 2.7 und 2.8 sowie im Kapitel 3 ausführlicher behandelt.

2.5 Umgebung und Kodierung

Der vorangegangene Abschnitt beschreibt, wie mit Hilfe von Netzwerken Berechnungen durchgeführt und Abbildungen von den Netzwerkeingaben auf die Netzwerkausgaben realisiert werden können, die ihrerseits anhand von Beispielen gelernt werden.

Die „Schnittstelle" zwischen Netzwerk und Umwelt, d.h. die Kodierung der Ein- und Ausgabesignale eines Netzwerks ist der Gegenstand dieses Abschnitts.

Wie bereits zuvor erläutert, werden Signale in konnektionistischen Modellen durch Zahlen dargestellt.

Um mit Hilfe eines konnektionistischen Modells eine bestimmte Aufgabe der natürlichen Umwelt (Beispiele siehe hiernach) zu lösen, muß sie zuvor geeignet kodiert werden.

Einige Beispiele für Aufgaben, die man mit konnektionistischen Modellen lösen möchte:

Steuerung eines Rollstuhls für Querschnittsgelähmte: Gegeben ist eine Menge gesprochener Wörter und eine Menge von Aktionen, z.B. { Vorwärts, Rückwärts, Links, Rechts, Schneller, Langsamer, Stop }. Die Aufgabe besteht in der richtigen Zuordnung des gesprochenen Worts zur gewünschten Aktion.

Maschinelles Lesen: Gegeben ist das Bild einer Videokamera und die Menge der Buchstaben, Zahlen und Zeichen. Die Aufgabe besteht in der korrekten Zuordnung der (maschinen- oder handschriftlichen) Zeichen im Bild der Kamera auf die Menge der Buchstaben, Zahlen und Zeichen. Mögliche Anwendungen sind das automatische Erfassen von Texten, die automatische Erkennung von Unterschriften, das Lesen des Betrages auf einem Scheck, etc.

Diagnoseunterstützung in der Medizin: a) Gegeben sind die Signale mehrerer Elektroden, die an Brust und Rücken befestigt die Herzstromkurven messen. Die Aufgabe besteht darin festzustellen, ob ein Herzinfarkt vorliegt und wenn ja, diesen (grob) zu lokalisieren.

Diagnoseunterstützung in der Medizin: b) Gegeben sind die Ergebnisse (Punktwerte) der Testaufgaben aus dem Aachener Aphasie Test (einem Sprachtest). Die Aufgabe besteht darin festzustellen, ob eine Aphasie (eine Sprachstörung aufgrund eines Hirninfarkts oder einer Hirnverletzung) vorliegt und falls ja, diese einem der Standardsyndrome (globale, Wernicke, Broca, amnestische Aphasie) zuzuordnen.

Medizinische Grundlagenforschung: Gegeben sind Aufnahmen des Gehirns mittels der Computer-Tomographie, in denen Hirnläsionen zu erkennen sind. Die Aufgabe besteht darin, diese Läsionen den zu erwartenden Aphasie-Symptomen (oder den Standardsyndromen) zuzuordnen. Man hofft, dadurch spezifische sprachliche Leistungen bestimmten Hirnregionen zuordnen zu können.

Die geeignete Kodierung von Aufgaben dieser Art ist ein echtes Problem, besonders da es keine exakten Kriterien dafür gibt, was eine geeignete Kodierung eines bestimmten Problems ist und was nicht.

Um die fundamentale Wichtigkeit der Kodierung zu unterstreichen, hier einige Beispiele (aus Denker et al. [4]):

Die Eingaben des Netzes seien binär kodierte Zahlen, d.h. jede Eingabeunit entspricht einem Bit der Binärdarstellung. Die Aktivierungen der Eingabeunits (vergleiche Abschnitt 2.1) werden auf die Werte 0.0 (für "false") bzw. 1.0 (für "true") gesetzt.

Man betrachte das Prädikat „x ist eine ungerade Zahl". Es ist offensichtlich, daß selbst ein sehr einfaches, einstufiges Netzwerk dieses Prädikat realisieren (und anhand von nur wenigen Beispielen lernen) kann, indem es die Unit des niedrigsten Eingabebits mit der Ausgabeunit verbindet und alle übrigen Bits ignoriert.

Man betrachte nun das Prädikat „x hat eine ungerade Anzahl von Primfaktoren". Möglicherweise gibt es tatsächlich ein Netzwerk, das dieses Prädikat (auf der Grundlage binär kodierter Zahlen) realisiert (und lernt), aber das erscheint kaum vorstellbar. Die Primfaktorzerlegung gilt als ein schwieriges, für große Zahlen nahezu unlösbares Problem, und ein Neuronales Netz ist in dieser Beziehung sicherlich keinem herkömmlichen Computer überlegen.

Offensichtlich sind triviale, oberflächliche strukturelle Eigenschaften wie gerade/ungerade leichter zu erkennen und zu lernen als tiefere, abstrakte Eigenschaften, wie z.B. für eine Zahl, prim zu sein.

Man beachte jedoch, wie sehr die Frage, was oberflächlich und was in hohem Grade nichttrivial ist, von der Wahl der Repräsentation des Problems abhängt:

Im Zahlensystem der Basis 3 beispielsweise ist das gerade bzw. ungerade sein einer Zahl keine offen zutage liegende Eigenschaft.

Umgekehrt läßt sich aber sehr leicht eine Repräsentation definieren, in der Zahlen durch ihre Primfaktoren dargestellt werden, so daß das Prädikat, eine ungerade Anzahl von Primfaktoren zu besitzen, sehr leicht zu berechnen ist.

Um von einem Neuronalen Netz verarbeitet werden zu können, müssen die Rohdaten eines gegebenen Problems nicht nur kodiert, sondern auch auf geeignete Weise vorverarbeitet werden.

Im Falle der automatischen Spracherkennung (siehe oben) bedeutet dies z.B., daß ein Mikrofon mit einem Verstärker vorhanden sein muß, das die gesprochenen Befehle aufnimmt. Ein elektronischer Schaltkreis wird in der Regel eingesetzt, um die Fouriertransformation der Sprachsignale in das jeweilige Frequenzspektrum zu berechnen. Ein Analog/Digital-Wandler macht anschließend diese Daten einem Rechner zugänglich, auf dem das Neuronale Netz simuliert wird. Schließlich setzt der Rechner ggfs. die Ausgaben des Netzes mit Hilfe von elektronischen Schaltern und elektrischen Motoren in Aktionen um.

Die Bedeutung dieser Vorverarbeitung kann nicht überschätzt werden, wie der folgende (etwas überspitzte) Beweis zeigt, daß maschinelles (automatisches) Lernen *immer* eine Lösung (d.h. eine Wahl der Gewichte der Verbindungen des Netzes) für eine *beliebige* Aufgabe finden kann, die richtige Vorverarbeitung vorausgesetzt:

Seien die Rohdaten in einer beliebigen Weise kodiert gegeben. An diese Eingabetupel werden zusätzlich die gewünschten Ausgaben des Netzes angehängt (durch Konkatenation der entsprechenden Tupel). Ein einstufiges Netzwerk, das mit diesen Eingaben gefüttert wird, lernt ohne weiteres, den Rohdatenteil seiner Eingaben zu ignorieren und die gewünschten Ausgaben zu den Ausgabeunits weiterzuleiten, ganz wie das Netz im Falle der Erkennung der ungeraden Zahlen.

Es wäre jedoch falsch, hieraus schließen zu wollen, daß maschinelles Lernen stets in der Lage sei, für jede beliebige Aufgabe eine Lösung zu finden; leider gibt es keine allgemeine, automatische Prozedur zur Durchführung der jeweils erforderlichen Vorverarbeitung.

Eine umfassende Darstellung der Kodierungsproblematik ist außer in Denker et al. [4] vor allem in Rumelhart/McClelland [24] (Kapitel 3) zu finden. Darin werden besonders die Vorteile von „*verteilten Kodierungen*" ("Distributed Representations") diskutiert. Grob zusammengefaßt kann man sagen, daß digitale Kodierungen für konnektionistische Modelle eher ungeeignet sind, da sie die spezifischen Fähigkeiten von Neuronalen Netzen nicht ausnutzen, wie z.B., auch mit unvollständigen oder gestörten (mit Rauschen überlagerten) Mustern die „richtigen" Ergebnisse zu liefern. Analoge, möglichst verteilte Kodierungen sind der Verarbeitungsweise Neuronaler Netze (siehe Abschnitt 2.3) besser angepaßt. Darüber hinaus erhöhen Dimensionserweiterungen des Inputs in der Regel die Konvergenzgeschwindigkeit beim Lernen (siehe dazu auch Linden [15]).

Insbesondere schlagen Rumelhart et al. in jenem Kapitel die Kodierung von zweidimensionalen (Bild-) Daten mit Hilfe von „*rezeptiven Feldern*" vor (dort "*coarse coding*" genannt), in Analogie zur Verarbeitung von Bildinformationen in der Retina (Netzhaut) von Wirbeltieren oder dem Facettenauge des Pfeilschwanzkrebses *Limulus* (siehe unter dem Stichwort „*rezeptive Felder*" in Linder [16], Eckert/Randall [6] u.a.). Diese Form der Kodierung besitzt eine große Bedeutung für viele Anwendungen.

Einige weitere wichtige Kodierverfahren zählt Linden [15] in seiner Arbeit auf:

- Eine direkte Art der Kodierung besteht darin, jedes Eingabemerkmal als reelle Zahl darzustellen und diese dem Netz als Aktivierungswert einer der Eingabeunits einzugeben. Hat ein Merkmal beispielsweise den Wert 2.5, so bekommt die entsprechende Eingabeunit den Aktivierungswert 2.5 zugewiesen. Eine Verfeinerung dieser einfachen Kodierung besteht darin, alle Eingaben jeweils auf den Bereich $[\,0\,,\,1\,]$ zu normieren, falls die jeweiligen Minimal- und Maximalwerte bekannt sind. Falls die Werte prinzipiell nicht beschränkt sind (also aus dem Intervall $[\,-\infty\,,\,+\infty\,]$ stammen), kann man zur Normierung die logistische Funktion (siehe Abschnitt 2.6 über Aktivierungsfunktionen) verwenden.

- Obwohl sie für Neuronale Netze eigentlich inadäquat ist, hat die *Binärdarstellung* von Zahlen in konnektionistischen Modellen nach wie vor eine große Bedeutung. Jede Eingabeunit entspricht dabei einem Bit der darzustellenden Zahl, die entweder den Aktivierungswert Null oder Eins zugewiesen bekommt. Schwierigkeiten ergeben sich, falls zusätzlich Zwischenwerte dargestellt werden sollen; außerdem sind die Kodes aufeinanderfolgender Zahlen nicht äquidistant (Abhilfe schafft im letzteren Fall die Verwendung eines Gray-Kodes).

- Bei der *lokalen Kodierung* wird für jeden Wert, den ein Merkmal annehmen kann, eine eigene Eingabeunit vorgesehen; die Eingabeunits haben dementsprechend nur die Aktivierungswerte Null oder Eins. Man nennt diese Kodierung auch *1-aus-n-Kodierung*. Sie eignet sich z.B. gut zur Darstellung der Buchstaben des Alphabets.
- Bei der *Thermometerkodierung* werden für jedes darzustellende Merkmal eine ganze Reihe von Eingabeunits vorgesehen, die je nach Wert des Merkmals analog zu einem Thermometer oder einer Leuchtbalkenanzeige sukzessive den Aktivierungswert Eins erhalten und ansonsten die Aktivierung Null. Der Wert 6 wird dann dargestellt durch $(1, 1, 1, 1, 1, 1, 0, \ldots, 0)$. Zwischenwerte wie z.B. 3.46 können dargestellt werden durch Aktivierung der ersten drei Eingabeunits mit dem Wert Eins und der vierten Eingabeunit mit dem Wert 0.46: (1.0 , 1.0 , 1.0 , 0.46 , 0.0 , ... , 0.0).

Die Wahl einer sowohl dem Problem als auch dem Netzwerk angepaßten Kodierung der Ausgabe und die Festlegung ihrer Bedeutung erfolgt analog; die oben genannten Kodierungen lassen sich auch zur Strukturierung der Ausgaben verwenden. Man beachte aber, daß bei Verwendung kontinuierlicher Ausgabefunktionen in den Ausgabeunits und der Verwendung einer diskreten oder semi-diskreten Kodierung der Ausgabe (wie z.B. der lokalen, der binären oder der Thermometerkodierung) Werte in der Ausgabe vorkommen können, deren Bedeutung in der jeweiligen Kodierung nicht definiert ist bzw. deren sinnvolle Interpretation Sache des Anwenders ist.

Kombinationen der obigen Kodierungen untereinander (z.B. zur Darstellung von Haupt- und Nebenmerkmalen oder zur Klassifizierung in Haupt- und Untergruppen) sind ebenfalls möglich.

2.6 Aktivierungsfunktionen

Einige der in der Praxis gängigen Aktivierungsfunktionen sollen hier kurz vorgestellt werden. Es wird dabei davon ausgegangen, daß als Ausgabefunktion der Units die Identität verwendet wird.

2.6.1 Lineare Aktivierungsfunktion

Die einfachste Aktivierungsfunktion ist die Identität:

$$F(x) \quad = \quad x \tag{2.9}$$

Units mit der linearen Aktivierungsfunktion und der Summation als Verrechnungsfunktion (siehe Abschnitt 2.1) berechnen Linearkombinationen der Ausgaben ihrer Vorgängerunits. Besteht ein Netz wie in Abbildung 2.3 (Seite 27) aus solchen Units, ist die Berechnung jeder Stufe des Netzes äquivalent zur Matrixmultiplikation der Ausgaben der niedrigeren Schicht mit der Gewichtsmatrix der Verbindungen zwischen dieser und der nächsthöheren Schicht. Das bedeutet, daß sich mit einem solchen Netz alle linearen Abbildungen realisieren lassen. (Eine ausführlichere Besprechung ist in Kapitel 3.6 zu finden)

Es ist zu beachten, daß die Ausgabewerte einer Unit bei dieser Aktivierungsfunktion nicht beschränkt sind!

2.6.2 “Linear Threshold”-Aktivierungsfunktion

Um (in hierarchischen Netzen) außer den linearen Abbildungen auch andere Klassen von Funktionen berechnen zu können, ist die Einführung von Nichtlinearitäten erforderlich.

Eine einfache und oft verwendete Möglichkeit ist die Einführung eines *Schwellenwertes*: Erst wenn die Netzeingabe net_j einer Unit j einen vorgegebenen Schwellenwert, genannt θ_j, übersteigt, erfolgt eine Reaktion dieser Unit (in Form einer Ausgabe), analog zum „Alles-oder-Nichts-Gesetz“ aus Kapitel 1.

Es gibt mehrere Möglichkeiten, eine solche Aktivierungsfunktion mit Schwellenwert zu definieren. Nur zwei Funktionen sollen hier exemplarisch, je eine für den diskreten und eine für den kontinuierlichen Fall, aufgeführt werden:

$$F(x) = \begin{cases} 1 & \text{falls } x > \theta \\ 0 & \text{falls } x \leq \theta \end{cases} \tag{2.10}$$

sowie

$$F(x) = \begin{cases} x - \theta & \text{falls } x > \theta \\ 0 & \text{falls } x \leq \theta \end{cases} \tag{2.11}$$

Diese Aktivierungsfunktionen haben allerdings den Nachteil, nicht differenzierbar zu sein, was für manche Lernregeln (siehe dazu die Abschnitte 2.7 und 2.8) Voraussetzung ist.

Eine Diskussion einer Anwendung der diskreten Variante dieser beiden Aktivierungsfunktionen ist in Kapitel 3.6 zu finden (Stichwort „Lineare Separierbarkeit“).

2.6.3 “Brain State in a Box”-Aktivierungsfunktion

Das “Brain State in a Box”-Modell (siehe Rumelhart/McClelland [24] sowie McClelland/Rumelhart [17]) ist eines der wenigen Modelle mit vollständiger Verbindungsstruktur, d.h. jede Unit ist mit jeder anderen (in beiden Richtungen) sowie mit sich selbst verbunden. Alle Units des Netzes dienen gleichzeitig sowohl als Ein- wie auch als Ausgabeunits.

Ein solches Netz dient als Auto-Assoziator, d.h. es kann dazu benutzt werden, um einmal gelernte Muster anhand von unvollständigen oder gestörten (also leicht fehlerhaften) Eingaben derselben Muster zu rekonstruieren. Bei der Eingabe eines dem Netzwerk unbekannten Musters wird das ähnlichste der gelernten Muster ausgegeben.

Das “Brain State in a Box”-Modell (abgekürzt „BSB-Modell“) verwendet eine lineare Aktivierungsfunktion, die außerdem vom vorherigen Aktivierungszustand der Unit abhängt:

$$a_j(\text{t+1}) = F(\, a_j(\text{t}) \,,\, \text{net}_j(\text{t}) \,) \tag{2.12}$$

mit

$$F(x, y) = F_{\text{clip}}(\, F_{\text{actv}}(x, y) \,) \tag{2.13}$$

und

$$\begin{aligned} F_{\text{actv}}(x, y) &= x + y \\ F_{\text{clip}}(z) &= \begin{cases} +1 & \text{falls } z > +1 \\ -1 & \text{falls } z < -1 \\ z & \text{sonst} \end{cases} \end{aligned} \tag{2.14}$$

Anders ausgedrückt, lautet die Aktivierungsvorschrift:

$$a_j\,(\mathrm{t+1}) \quad = \quad a_j\,(\mathrm{t}) \; + \; \mathrm{net}_j\,(\mathrm{t}) \tag{2.15}$$

Wobei zu beachten ist, daß die neue Aktivierung auf den Wert -1 bzw. $+1$ gesetzt wird, falls sie diese Werte unter- bzw. überschreiten sollte.

Eine (experimentelle) Anwendung dieses Modells ist die Klassifikation von Räumen anhand von darin enthaltenen Möbeln und Gebrauchsgegenständen, nach Vorgabe einiger Modellräume mit entsprechender Einrichtung (wie Küche, Wohnzimmer, Bad, Schlafzimmer usw.). (Siehe McClelland/Rumelhart [17])

2.6.4 Stochastische Aktivierungsfunktion ("Boltzmann Machine")

Nicht nur deterministische, sondern auch stochastische Funktionen finden als Aktivierungsfunktion Verwendung, die den Units in den betreffenden Modellen Eigenschaften verleihen, die entfernt an die *spontane Aktivität* von Nervenzellen erinnern (siehe Kapitel 1).

Im Falle der „Boltzmann-Maschine" ist die Aktivierungsfunktion eine (diskrete) Zufallsvariable mit den Werten Null und Eins, deren Wahrscheinlichkeitsverteilung wie folgt lautet:

$$\begin{aligned} p\,(\; a_j \; = \; 1 \;) \quad &= \quad f\,(\; \tfrac{net_j \, + \, \iota_j \, - \, \theta_j}{T} \;) \\ &= \quad \frac{1}{1 \; + \; e^{-\frac{net_j \, + \, \iota_j \, - \, \theta_j}{T}}} \end{aligned} \tag{2.16}$$

Dabei steht ι_j für eine Eingabe in Unit j von außen und θ_j für einen Schwellenwert, T ist ein Parameter.

Die dieser Wahrscheinlichkeitsverteilung zugrundeliegende Funktion f wird *logistische Funktion* genannt:

$$f(x) \quad = \quad \frac{1}{1 \; + \; e^{-x}} \tag{2.17}$$

Wenn die Netzeingabe net_j (plus einer etwaigen Eingabe von außen) größer als der Schwellenwert θ_j ist, so ist die Wahrscheinlichkeit $p\,(\; a_j \; = \; 1 \;)$ größer als 1/2. Ist die Netzeingabe net_j (plus der etwaigen Eingabe von außen) dagegen kleiner als der Schwellenwert θ_j, dann ist diese Wahrscheinlichkeit kleiner als 1/2.

Liegt die Netzeingabe weit genug über dem Schwellenwert, so wird die betreffende Unit (so gut wie) immer aktiviert. Liegt die Netzeingabe weit genug darunter, bleibt sie (so gut wie) stets inaktiv.

Der Parameter T bestimmt dabei die Steilheit der Verteilungsfunktion. Je größer er ist, desto größeren Einfluß hat der Zufall auf die Aktivierung der Units. Für $T \rightarrow 0$ geht die Aktivierungsfunktion in die "Linear Threshold"-Aktivierungsfunktion über.

Für die Boltzmann-Maschine gelten (per formalem Analogieschluß) Sätze aus der Thermodynamik, die etwas über das Verhalten dieses Modells in Abhängigkeit von dem Parameter T, der der „*Temperatur*" des Systems entspricht, aussagen, weshalb man auch von einem *Thermodynamischen Modell* (und einer *thermodynamischen Aktivierungsfunktion*) spricht.

Durch schrittweises Verringern des Parameters T läßt sich das Modell in einen niedrigen Energiezustand überführen, einen Zustand maximaler Übereinstimmung zwischen den

Eingaben in das Netz und den durch die Gewichte der Verbindungen innerhalb des Netzes definierten einschränkenden Bedingungen ("constraints"). Man nennt dieses Verfahren "*Simulated Annealing*" („Simuliertes Abkühlen").

Nähere Einzelheiten zu diesem Modell kann man Rumelhart/McClelland [24] sowie McClelland/Rumelhart [17] entnehmen.

2.6.5 Logistische (Sigmoide) Aktivierungsfunktion

Viele Lernverfahren setzen eine differenzierbare Aktivierungsfunktion voraus. Gleichzeitig ist jedoch eine nichtlineare Funktion erforderlich, um in hierarchischen Netzen (vergleiche Abbildung 2.3) andere als nur lineare Abbildungen realisieren zu können. Die "Linear Threshold"-Aktivierungsfunktion erfüllt zwar die letztere, nicht jedoch die erste Bedingung. Eine Klasse von Funktionen, die beide Bedingungen erfüllen, sind die sigmoiden (S-förmigen) Funktionen.

Einer ihrer Vertreter beruht auf der bereits erwähnten *logistischen Funktion*:

$$f(x) \quad = \quad \frac{1}{1 \; + \; e^{-x}} \tag{2.18}$$

Die *logistische Aktivierungsfunktion* lautet:

$$\begin{aligned} F(x) \quad &= \quad f(\ \text{net} \; - \; \theta \ ) \\ &= \quad \frac{1}{1 \; + \; e^{-(net \, - \, \theta)}} \end{aligned} \tag{2.19}$$

Diese Funktion ist beschränkt (durch Null und Eins), stetig, differenzierbar, streng monoton steigend, auf dem ganzen Intervall $(-\infty, +\infty)$ definiert und semilinear in einem kleinen Bereich um den Arbeitspunkt (im Falle der logistischen Funktion der Nullpunkt) herum, der durch den Schwellwert θ bestimmt wird.

Die Aktivierung einer Unit j berechnet sich dann wie folgt:

$$a_j \quad = \quad f(\ \text{net}_j \; - \; \theta_j \ ) \tag{2.20}$$

Auf der logistischen Aktivierungsfunktion beruhen alle Modelle, die Back-Propagation als Lernverfahren benutzen (siehe auch Abschnitt 2.8).

2.6.6 Realisierung des Schwellenwerts in der Praxis

Das Argument der logistischen Aktivierungsfunktion, $\text{net}_j \; - \; \theta_j$, lautet ausgeschrieben (vergleiche mit der Definition der Verrechnungsfunktion g in Abschnitt 2.1):

$$\sum_{k=1}^{n} w_{ji_k} \cdot o_{i_k} \; - \; \theta_j \tag{2.21}$$

Indem man ein neues Gewicht $w_{j0} \; = \; w_{ji_0}$ definiert durch:

$$w_{j0} \quad = \quad - \; \theta_j \tag{2.22}$$

sowie durch Hinzufügen einer fiktiven Unit #0, deren Output $o_0 = o_{i_0}$ stets gleich Eins ist, läßt sich der Schwellenwert in die Berechnung der Summe (2.21) mit hineinziehen:

$$\sum_{k=0}^{n} w_{ji_k} \cdot o_{i_k} \tag{2.23}$$

Die Aktivierung einer Unit j berechnet sich dann durch:

$$a_j = \frac{1}{1 + e^{-\left(\sum_{k=0}^{n} w_{ji_k} \cdot o_{i_k}\right)}} \tag{2.24}$$

Da die Schwellenwerte, die den Arbeitspunkt der sigmoiden logistischen Aktivierungsfunktion jeweils festlegen, genauso gelernt werden müssen wie die Gewichte der Verbindungen des Netzes, ist es in der Simulation günstig (da es die Berechnung sehr vereinfacht), die Schwellenwerte als Gewichte von Verbindungen einer (oder mehrerer) speziellen Unit mit dem konstanten Aktivierungswert Eins zu realisieren, so wie das in den obigen Formeln bereits geschehen ist.

In der Abbildung 2.3 (Seite 27) gibt es zwei dieser speziellen Units, I_0 und H_0 genannt, wobei die Gewichte der Verbindungen von diesen beiden zu den Units der jeweils nächsthöheren Schicht die (negativen) Schwellenwerte der Units dieser (der höheren) Schicht darstellen.

Im Simulationsprogramm werden die Aktivierungen von allen Units einer Schicht jeweils in einem Vektor zusammengefaßt, dessen „nullte" Komponente gerade die spezielle Unit, I_0 oder H_0, realisiert, deren Wert jeweils vor einem "Propagation"-Schritt oder "Forward-Pass" (vergleiche Abschnitt 2.3) auf Eins gesetzt wird.

Die Komponenten w_{j0} bzw. w_{k0} der Gewichtsmatrizen W_{ji} bzw. W_{kj} (für die beiden Stufen des Netzes) stellen konsequenterweise die (negativen) Schwellenwerte der "hidden"- bzw. der "output"-Units dar.

2.7 Lernregeln

Konnektionistische Modelle besitzen die Fähigkeit, bestimmte Aufgaben „selbständig" zu lösen, indem sie diese Lösung anhand von Beispielen selbständig erlernen, d.h. ohne daß die interne Struktur des Netzes und die interne Repräsentation des Problems explizit vorgegeben werden muß.

Man unterscheidet im Wesentlichen vier verschiedene Arten des Lernens in konnektionistischen Modellen:

Auto-Assoziation: Eine Menge von *Patterns* (= „Mustern") wird dem Netz wiederholt eingegeben, die das Netz „lernen" (speichern) soll. Wenn das Netz ausgelernt hat, werden unvollständige oder veränderte Versionen der gelernten Patterns angelegt, die das Netzwerk vervollständigen bzw. korrigieren soll. Das unvollständige oder veränderte Muster dient dabei als Schlüssel zum Wiederfinden der ursprünglichen Information.

Pattern-Assoziation: Eine Menge von *Paaren* von Patterns wird dem Netz wiederholt präsentiert. Das Netzwerk soll lernen, das zweite Pattern eines Paares auszugeben, wenn das erste Pattern des Paares eingegeben wird. Hier muß das Netzwerk die Zuordnung von einer (im Prinzip) willkürlichen Menge von Eingabepatterns auf eine andere, ebenso willkürliche Menge von Ausgabepatterns leisten.

Klassifikation: Hierbei soll eine Menge von Patterns oder „Stimuli“ in eine Reihe von vorgegebenen Kategorien (Klassen) eingeteilt werden. Dem Netz werden während der Lernphase eine Reihe von Patterns sowie die Klasse eingegeben, in die das Pattern jeweils gehört. Nach Abschluß der Lernphase soll das Netz in der Lage sein, die gelernten Stimuli selbst korrekt zu klassifizieren. Man hofft oder erwartet, daß außerdem andere, nicht gelernte Stimuli aus der gleichen Quelle wie die bereits gelernten ebenfalls korrekt bzw. sinnvoll klassifiziert werden. Diese Transferleistung bezeichnet man als „*Generalisierungsvermögen*“ des Netzes. Bei dieser Art des Lernens handelt es sich um eine Form der Pattern-Assoziation, allerdings mit einer speziellen Zielsetzung.

Regularitäts-Extraktion: Eine Menge von Patterns wird dem Netzwerk in der Lernphase präsentiert. Das Netzwerk soll selbständig relevante Merkmale der Trainingsmenge herausfinden, um diese Patterns in verschiedene, bezüglich dieser Merkmale möglichst homogene Klassen einzuteilen. Im Gegensatz zur *Klassifikation* gibt es hierbei keine a-priori festgelegten und von außen vorgegebenen Kategorien. Das Netz muß vielmehr selbständig eine interne Repräsentation der relevanten Merkmale der Menge der Eingabestimuli entwickeln, die dann eine Einteilung dieser Patterns in Klassen erlaubt.

Die für die hier untersuchten Modelle relevanten Arten des Lernens sind die *Pattern-Assoziation* und die *Klassifikation*.

Zwei Lernregeln sollen im Folgenden vorgestellt werden, die diese Arten des Lernens in Neuronalen Netzen ermöglichen.

2.7.1 Die Hebb'sche Lernregel

Die erste Lernregel geht auf Hebb [10] zurück. Hebb untersuchte in seiner Arbeit die neurophysiologischen Grundlagen des Lernens. Er stellte dabei die folgende Hypothese auf:

> Wenn das Axon eines Neurons A mit einem Neuron B eine Synapse bildet und wiederholt oder ständig daran beteiligt ist, in B ein Aktionspotential auszulösen, dann findet ein Wachstumsprozess oder eine metabolische Veränderung in einer der oder beiden Zellen statt, so daß sich die Wirksamkeit von A bei der Auslösung eines Aktionspotentials in B erhöht.

In der quantifizierten Konkretisierung von Rumelhart et al. [24], die auch negativen Aktivierungen und inhibitorischen Synapsen Rechnung trägt, lautet die Hebb'sche Lernregel wie folgt:

Definition 2.1 *Modifiziere die Stärke der Verbindung von einer Unit i zu einer Unit j proportional zu dem Produkt aus den Aktivierungswerten der beiden Units.*

In mathematischer Schreibweise lautet die Hebb'sche Lernregel:

$$\begin{aligned} \Delta w_{ji} &= \eta \cdot a_j \cdot a_i \\ \text{und} \qquad & \\ w'_{ji} &= w_{ji} + \Delta w_{ji} \end{aligned} \tag{2.25}$$

Der Parameter η ist der Proportionalitätsfaktor aus Definition 2.1.

Wenn das Produkt der beiden Aktivierungswerte positiv ist, wird die Stärke (= das Gewicht) der Verbindung vergrößert, hin zu einer größeren *exzitatorischen* (erregenden) Wirkung auf die nachfolgende Unit. Ist das Produkt negativ, wird das Gewicht der Verbindung vermindert, hin zu einer größeren *inhibitorischen* (hemmenden) Wirkung auf die nachfolgende Unit.

Ein einstufiges Netzwerk kann mit Hilfe dieser Regel lernen, ein Pattern A mit einem Pattern B zu assoziieren, indem man den Eingabeunits das Pattern A eingibt und den Ausgabeunits das Pattern B. Anschließend werden die Gewichte nach der obigen Lernregel modifiziert.

Erstaunlicherweise kann das Netzwerk mehr als nur ein solches Paar von Patterns lernen, obwohl die Gewichte, in denen diese Muster intern gespeichert werden, für alle Paare dieselben sind. (Systematische Einschränkungen hiervon bzw. Bedingungen, die die Patterns dafür erfüllen müssen, siehe in Kapitel 3.6 bzw. in Rumelhart/McClelland [24] und McClelland/Rumelhart [17])

Die verschiedenen Paare von Patterns werden wiederholt der Reihe nach angelegt, wobei jedesmal die Gewichte nach der Lernregel modifiziert werden. Der Proportionalitätsfaktor η wird so klein gewählt, daß alle Patterns gleichmäßig gelernt werden und die Gewichte der Verbindungen langsam wachsen, bis das Netzwerk die gewünschten Assoziationen leistet, d.h., bis das Netzwerk beim Anlegen eines Eingabepatterns und Propagieren der Signale zu den Ausgabeunits das gewünschte Ausgabepattern produziert.

2.7.2 Die Delta-Lernregel

Die zweite Lernregel wurde von mehreren Autoren unabhängig voneinander formuliert. Der Name „Delta-Regel“ geht auf Rumelhart et al. [24] zurück; er bezieht sich auf die Differenz zwischen der gewünschten und der vom Netzwerk produzierten Ausgabe, die in diese Lernregel als wesentlichste Größe eingeht. Nach zwei weiteren Autoren wird diese Lernregel auch *Widrow-Hoff-Regel* genannt.

Die Menge P der Paare von Patterns, die das Netzwerk lernen soll, ist gegeben durch:

$$P = \{ \ (\ \vec{i}_p \ , \ \vec{t}_p \ ) \ \mid \ 1 \leq p \leq n_{pat} \ \} \tag{2.26}$$

Nach dem Anlegen eines Inputvektors $\vec{i}_p$ berechnet das Netz zuerst seine eigene Ausgabe, $\vec{o}_p$. Anschließend wird diese Ausgabe mit der *gewünschten* Ausgabe, dem *„Target-Vektor“* $\vec{t}_p$, verglichen. Besteht kein Unterschied, findet auch kein Lernen statt. Andernfalls werden die Gewichte des Netzes modifiziert, um diese Differenz zu verringern.

Die Regel zur Veränderung der Gewichte des Netzes nach dem Anlegen eines Input/Output-Paares p lautet:

$$\begin{aligned} \Delta_p w_{ji} &= \eta \cdot (t_{pj} - o_{pj}) \cdot o_{pi} \\ \text{mit} \qquad w'_{ji} &= w_{ji} + \Delta_p w_{ji} \end{aligned} \tag{2.27}$$

Man definiert:

$$\Delta_{pj} = (\ t_{pj} - o_{pj} \ ) \tag{2.28}$$

(Nicht zu verwechseln mit den Differenzen $\Delta_p w_{ji}$, um die die Gewichte w_{ji} modifiziert werden!)

Die nach dieser Differenz Δ_{pj} benannte *Delta-Regel* lautet dann:

$$\begin{array}{lrcl} & \Delta_p w_{ji} & = & \eta \cdot \Delta_{pj} \cdot o_{pi} \\ \text{mit} & & & \\ & w'_{ji} & = & w_{ji} + \Delta_p w_{ji} \end{array} \tag{2.29}$$

Dabei ist t_{pj} die j-te Komponente des Targetvektors des Patternpaares p, o_{pj} ist das j-te Element des vom Netzwerk produzierten Outputvektors, d.h. die Ausgabe der j-ten Ausgabeunit des Netzes, bei Anlegen des Inputvektors des Patternpaares p. o_{pi} ist der Ausgabe- bzw. Aktivierungswert der i-ten Eingabeunit, der gleich dem i-ten Element des Inputvektors, i_{pi}, ist. $\Delta_p w_{ji}$ ist der Wert, um den das Gewicht der Verbindung von Eingabeunit i zu Ausgabeunit j nach Anlegen des Patternpaares p verändert wird. Der Parameter η ist ein Proportionalitätsfaktor.

Analog wie bei der Hebb'schen Regel wird der Parameter η klein genug gewählt, damit sich die Gewichte durch wiederholtes Anlegen der Patterns aus P nach und nach, möglichst stetig aufbauen.

Der Vorteil dieser Lernregel gegenüber der Hebb'schen Regel liegt darin, daß die Voraussetzungen, die die Menge der Inputvektoren erfüllen muß, damit ein Netzwerk die Menge der Patternzuordnungen P lernen kann, schwächer sind als im Falle der Hebb'schen Regel. (Siehe dazu auch Kapitel 3.6 sowie Rumelhart/McClelland [24] und McClelland/Rumelhart [17])

Man beachte, daß sowohl die Hebb'sche als auch die Delta-Lernregel nur für solche (ansonsten jedoch beliebige) Netzwerke geeignet sind, die keine "hidden" Units enthalten. Eine Lernregel für hierarchische Netze mit beliebig vielen Stufen wird im folgenden Abschnitt vorgestellt.

2.8 Standard Back-Propagation

Ein großer Nachteil der Hebb'schen und der Delta-Lernregel liegt darin, daß man mit ihnen keine mehrstufigen Netze trainieren kann, da es für die "hidden" Units keine "target"- oder Sollausgaben gibt. Das wiederum liegt daran, daß das Netz beim Lernen selbständig, ohne Vorgaben, interne Repräsentationen in den "hidden" Units bilden soll, die die Lösung von komplexeren Aufgaben erlauben, als das durch direkte Assoziation durch eine einzige Stufe möglich ist. Das dafür inzwischen klassische Beispiel ist das XOR- (Exklusiv-Oder) Problem (siehe auch Kapitel 4):

Der Output des Netzes zur Simulierung eines XOR-Gatters soll genau dann Eins betragen (und sonst Null), wenn eine der beiden Input-Units den Wert Eins hat, d.h. wenn beide Input-Units den Wert Eins haben, soll das Netz den Wert Null ausgeben.

Es läßt sich zeigen (Linden [15], Rumelhart/McClelland [24]), daß ein einstufiges Netzwerk nicht in der Lage ist, diese Funktion zu realisieren. Erst durch die interne Verknüpfung der beiden Inputs in einer "hidden" Unit, die die Output-Unit hemmt, falls beide Input-Units den Wert Eins haben, läßt sich diese Aufgabe lösen.

Im Gegensatz zur Hebb'schen oder der Delta-Regel erfordert das "*Back-Propagation*"-Verfahren eine **differenzierbare** Aktivierungsfunktion in den Units des Netzes. Außerdem beschränkt sich seine Anwendung in der Regel auf *hierarchische* Netze wie in Abbildung 2.3 (Seite 27). (Andere Anwendungen wie z.B. in rekurrenten Netzen siehe u.a. in Linden [15])

Man beachte, daß dieses Verfahren (ebenfalls im Unterschied zur Hebb'schen oder der Delta-Regel) für hierarchische Netze *mit beliebig vielen Stufen* geeignet ist.

Eine sehr ausführliche und umfassende Darstellung dieses Verfahrens ist in Rumelhart/McClelland [24] (Kapitel 8) zu finden. Da die Delta-Regel unter bestimmten Voraussetzungen ein Sonderfall von Back-Propagation ist, wird dieses Verfahren dort auch *Generalized Delta-Rule* (Verallgemeinerte Delta-Regel) genannt.

Das *Ziel des Verfahrens* besteht darin, die Gewichte des Netzes so zu modifizieren, daß der Unterschied zwischen den Ausgaben des Netzes, $\vec{o}_p$, und den gewünschten Ausgaben $\vec{t}_p$ für alle Paare p von Patterns aus P Null wird (oder zumindest Null so nahe wie möglich kommt).

Die Menge P der zu lernenden Patterns ist gegeben durch:

$$P \quad = \quad \{ \ (\ \vec{i}_p \ , \ \vec{t}_p \ ) \ \ | \ \ 1 \leq p \leq n_{pat} \ \ \} \tag{2.30}$$

(Vergleiche auch Abschnitt 2.7)

Man beachte, daß P nicht unbedingt eine endliche Menge sein muß, sondern beispielsweise auch aus dem Definitions- und Wertebereich einer Funktion (auch einer stochastischen Funktion) bestehen kann, aus der zum Zeitpunkt des Trainings des Netzes „Proben“ (Samples) entnommen werden. (Dies gilt besonders für Real-Time-Anwendungen)

Die *Methode* des Back-Propagation-Verfahrens ist ein „*Gradientenabstieg*“ (Gradient Descent) auf einer zuvor geeignet definierten *Fehlerfunktion*, die von den Gewichten des Netzes als ihren Parametern abhängt und deren Funktionswert durch das Verfahren minimiert werden soll. Man spricht auch von der „*Methode des steilsten Abstiegs*“ (Steepest Descent). Der Gradient, der hierzu benötigt wird, besteht aus den partiellen Ableitungen der Fehlerfunktion nach den Gewichten des Netzes.

Die Fehlerfunktion mißt den Unterschied zwischen Ist- und Sollausgabe des Netzes. Da in die Berechnung der Istausgabe die Aktivierungsfunktion in den Units mit eingeht, muß diese zur Berechnung der partiellen Ableitungen differenzierbar sein.

Im Allgemeinen besitzen die Netzwerke mehr als nur eine Output-Unit, daher besteht der „Unterschied“ zwischen der Ist- und der Sollausgabe des Netzes für ein gegebenes Paar von Input/Output-Patterns p aus mehreren Komponenten:

$$\Delta_{pk} \quad = \quad (\ t_{pk} \ - \ o_{pk} \ ) \tag{2.31}$$

(Analog zur Definition in Gleichung (2.28) der Delta-Regel in Abschnitt 2.7)

Basierend auf diesen Differenzen Δ_{pk} muß ein geeignetes Fehlermaß gefunden werden, das die Fehler in den einzelnen Komponenten in einer einzigen Zahl zusammenfaßt. Eine dafür geeignete Funktion ist beispielsweise die *Euklidische Norm*[8]. Diese hat außerdem die erwünschte Eigenschaft, größere Fehler durch die Quadrierung der Differenzen stärker zu berücksichtigen als kleine Fehler.

[8] Im Prinzip ist hierfür jede Norm geeignet.

Der „Fehler" für ein Paar von Input/Output-Patterns, E_p, wird allgemein als das Quadrat der Euklidischen Norm über den Differenzen Δ_{pk} definiert:

$$E_p = \frac{1}{2} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 \tag{2.32}$$

(Der Faktor $\frac{1}{2}$ wird eingeführt, weil sich dadurch die partiellen Ableitungen vereinfachen)

Da das Netz jedoch nicht nur einzelne Paare von Patterns, sondern alle Patterns aus P lernen soll, muß die Fehlerfunktion auf der gesamten Menge P definiert werden. Man bedient sich hier einfach der Summe der Fehler in den einzelnen Patterns:

$$\begin{aligned} E &= \sum_{p=1}^{n_{pat}} E_p \\ &= \sum_{p=1}^{n_{pat}} \frac{1}{2} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 \\ &= \frac{1}{2} \sum_{p=1}^{n_{pat}} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 \end{aligned} \tag{2.33}$$

Die so definierte Fehlerfunktion wird als „*LMS-Fehlermaß*" (Least Mean Squares error measure) bezeichnet.

Durch Ableitung dieses Fehlermaßes nach den Gewichten des Netzes (die implizit in den Werten o_{pk} enthalten sind) erhält man einen Vektor von partiellen Ableitungen, den Gradientenvektor. Dieser Vektor zeigt stets in die Richtung des steilsten *Anstiegs* der abgeleiteten Funktion an der Stelle der Auswertung, d.h. für die aktuellen Werte der Gewichte des Netzes.

Die Werte von o_{pk} hängen jeweils von der verwendeten Aktivierungsfunktion ab. Als Aktivierungsfunktion wird in Back-Propagation-Modellen in der Regel die *logistische Aktivierungsfunktion* (siehe Abschnitt 2.6) verwendet.

Bei der Berechnung des Gradientenvektors wird die Ableitung der Aktivierungsfunktion benötigt. Die Ableitung der logistischen Funktion berechnet sich wie folgt:

$$\begin{aligned} f(x) &= \frac{1}{1 + e^{-x}} \\ f'(x) &= \frac{0 \cdot (1+e^{-x}) - 1 \cdot (-e^{-x})}{(1+e^{-x})^2} \\ &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1}{(1+e^{-x})} \cdot \frac{e^{-x}}{(1+e^{-x})} \\ &= \frac{1}{(1+e^{-x})} \cdot \frac{1+e^{-x}-1}{(1+e^{-x})} \\ &= f(x) \cdot [\, 1 - f(x) \,] \end{aligned} \tag{2.34}$$

Die Berechnung des Gradientenvektors erfolgt stufenweise, beginnend mit der letzten Stufe (der Gewichte zwischen der (letzten) "hidden" und der Ausgabeschicht), dann weiter mit eventuell dazwischenliegenden Stufen (falls mehrere "hidden layer" vorhanden sind) und endend bei der ersten Stufe (der Gewichte zwischen der Eingabe- und der (ersten) "hidden" Schicht).

Sei $w_{k_0 j_0}$ ein Gewicht der letzten Stufe (vergleiche dazu auch Abbildung 2.3 Seite 27). Aufgrund der Linearität der Differentiation gilt die folgende Beziehung:

$$\frac{\partial E}{\partial w_{k_0 j_0}} = \sum_{p=1}^{n_{pat}} \frac{\partial E_p}{\partial w_{k_0 j_0}} \tag{2.35}$$

Die partiellen Ableitungen der Gewichte der letzten Stufe werden wie folgt nach der Kettenregel berechnet:

$$\underbrace{\frac{\partial E_p}{\partial w_{k_0 j_0}}}_{(2.35)} = \underbrace{\frac{\partial E_p}{\partial o_{pk_0}}}_{(2.37)} \cdot \underbrace{\frac{\partial o_{pk_0}}{\partial net_{pk_0}}}_{(2.38)} \cdot \underbrace{\frac{\partial net_{pk_0}}{\partial w_{k_0 j_0}}}_{(2.39)} \tag{2.36}$$

Die Abhängigkeit des Fehlers von der Ausgabe der Ausgabeunits wird durch die folgende Beziehung quantifiziert:

$$\begin{aligned} \frac{\partial E_p}{\partial o_{pk_0}} &= \frac{\partial}{\partial o_{pk_0}} \left(\frac{1}{2} \cdot \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 \right) \\ &= \frac{\partial}{\partial o_{pk_0}} \left(\frac{1}{2} \cdot (t_{pk_0} - o_{pk_0})^2 \right) \\ &= \frac{1}{2} \cdot 2 \cdot (t_{pk_0} - o_{pk_0}) \cdot (-1) \\ &= - (t_{pk_0} - o_{pk_0}) \end{aligned} \tag{2.37}$$

Die Ableitung der Ausgabe einer Unit nach ihrer Netzeingabe ist gleich der Ableitung ihrer Aktivierungsfunktion (f ist die logistische Funktion):

$$\begin{aligned} o_{pk} &= f(\mathrm{net}_{pk}) \\ \frac{\partial o_{pk_0}}{\partial net_{pk_0}} &= \frac{\partial}{\partial net_{pk_0}} (f(\mathrm{net}_{pk_0})) \\ &= f'(\mathrm{net}_{pk_0}) \\ &= f(\mathrm{net}_{pk_0}) \cdot [1 - f(\mathrm{net}_{pk_0})] \\ &= o_{pk_0} \cdot (1 - o_{pk_0}) \end{aligned} \tag{2.38}$$

Die Ableitung der Netzeingabe einer Ausgabeunit nach einem Gewicht einer ihrer Verbindungen mit einer vorhergehenden Unit schließlich lautet (unter Beachtung des Hinweises aus Abschnitt 2.6 über die Realisierung der Schwellenwerte als Gewichte):

$$\begin{aligned} \frac{\partial net_{pk_0}}{\partial w_{k_0 j_0}} &= \frac{\partial}{\partial w_{k_0 j_0}} \left(\sum_{j=0}^{n_{hid}} w_{k_0 j} \cdot o_{pj} \right) \\ &= o_{pj_0} \end{aligned} \tag{2.39}$$

Zusammensetzen der einzelnen Terme ergibt:

$$\frac{\partial E_p}{\partial w_{k_0 j_0}} = - (t_{pk_0} - o_{pk_0}) \cdot o_{pk_0} \cdot (1 - o_{pk_0}) \cdot o_{pj_0} \tag{2.40}$$

Durch Einsetzen in Gleichung (2.35) erhält man schließlich das Ergebnis:

$$\frac{\partial E}{\partial w_{k_0 j_0}} = - \sum_{p=1}^{n_{pat}} (t_{pk_0} - o_{pk_0}) \cdot o_{pk_0} \cdot (1 - o_{pk_0}) \cdot o_{pj_0} \tag{2.41}$$

Die Ergebnisse aus den Gleichungen (2.37) und (2.38) faßt man auch in einem einzigen Term zusammen, da dieser Term in der Berechnung der Ableitungen der nächstniedrigeren Stufe benötigt wird. Dieser Term wird mit δ_{pk} bezeichnet:

$$\begin{aligned} \delta_{pk} &= (t_{pk} - o_{pk}) \cdot o_{pk} \cdot (1 - o_{pk}) \\ &= \Delta_{pk} \cdot o_{pk} \cdot (1 - o_{pk}) \end{aligned} \tag{2.42}$$

Dieser Term wird außerdem „*Fehlersignal*“ genannt.

Sei $w_{j_0 i_0}$ ein Gewicht der nächstniedrigeren Stufe (in einem Netz wie in Abbildung 2.3 ist das bereits die Eingangsstufe).

Wieder gilt aufgrund der Linearität der Differentiation die folgende Beziehung:

$$\frac{\partial E}{\partial w_{j_0 i_0}} = \sum_{p=1}^{n_{pat}} \frac{\partial E_p}{\partial w_{j_0 i_0}} \tag{2.43}$$

Ebenfalls analog erfolgt die Berechnung der partiellen Ableitungen dieser Stufe nach der Kettenregel:

$$\underbrace{\frac{\partial E_p}{\partial w_{j_0 i_0}}}_{(2.43)} = \underbrace{\frac{\partial E_p}{\partial o_{pj_0}}}_{(2.46)} \cdot \underbrace{\frac{\partial o_{pj_0}}{\partial net_{pj_0}}}_{(2.47)} \cdot \underbrace{\frac{\partial net_{pj_0}}{\partial w_{j_0 i_0}}}_{(2.48)} \tag{2.44}$$

Zur Erläuterung der Berechnung des ersten dieser Glieder hier zunächst eine ausführlichere Darstellung des Fehlers E_p (f ist die logistische Aktivierungsfunktion):

$$\begin{aligned} E_p &= \frac{1}{2} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 \\ &= \frac{1}{2} \sum_{k=1}^{n_{out}} [t_{pk} - f(\mathrm{net}_{pk})]^2 \\ &= \frac{1}{2} \sum_{k=1}^{n_{out}} [t_{pk} - f(\sum_{j=0}^{n_{hid}} w_{kj} \cdot o_{pj})]^2 \end{aligned} \tag{2.45}$$

Die Ableitung des Fehlers E_p nach der Ausgabe einer “hidden” Unit o_{pj_0} ergibt dann (unter mehrfacher Anwendung der Kettenregel):

$$\begin{aligned} \frac{\partial E_p}{\partial o_{pj_0}} &= \frac{\partial}{\partial o_{pj_0}} \left(\frac{1}{2} \cdot \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 \right) \\ &= \frac{1}{2} \cdot \sum_{k=1}^{n_{out}} 2 \cdot (t_{pk} - o_{pk}) \cdot [- f'(\mathrm{net}_{pk})] \cdot \frac{\partial net_{pk}}{\partial o_{pj_0}} \\ &= - \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk}) \cdot o_{pk} \cdot (1 - o_{pk}) \cdot w_{kj_0} \\ &= - \sum_{k=1}^{n_{out}} \delta_{pk} \cdot w_{kj_0} \end{aligned} \tag{2.46}$$

Die Ableitung der Ausgabe einer Unit nach ihrer Netzeingabe ist gleich der Ableitung ihrer Aktivierungsfunktion (f ist die logistische Funktion):

$$\begin{aligned} o_{pj} &= f(\ \mathrm{net}_{pj}\ ) \\ \frac{\partial o_{pj_0}}{\partial net_{pj_0}} &= f'(\ \mathrm{net}_{pj_0}\ ) \\ &= o_{pj_0} \cdot (\ 1 - o_{pj_0}\ ) \end{aligned} \tag{2.47}$$

Die Ableitung der Netzeingabe einer Unit nach einem Gewicht einer ihrer Verbindungen mit einer vorhergehenden Unit (hier einer Eingabeunit) schließlich lautet:

$$\begin{aligned} \frac{\partial net_{pj_0}}{\partial w_{j_0 i_0}} &= \frac{\partial}{\partial w_{j_0 i_0}} \left(\sum_{i=0}^{n_{inp}} w_{j_0 i} \cdot o_{pi} \right) \\ &= o_{pi_0} \end{aligned} \tag{2.48}$$

Das Zusammensetzen der einzelnen Terme ergibt:

$$\begin{aligned} \frac{\partial E_p}{\partial w_{j_0 i_0}} &= - \sum_{k=1}^{n_{out}} [\,(t_{pk} - o_{pk}) \cdot o_{pk}\ (1 - o_{pk}) \cdot w_{kj_0}\,] \cdot o_{pj_0}\ (1 - o_{pj_0}) \cdot o_{pi_0} \\ &= - \sum_{k=1}^{n_{out}} [\ \delta_{pk} \cdot w_{kj_0}\ ] \cdot o_{pj_0}\ (1 - o_{pj_0}) \cdot o_{pi_0} \end{aligned} \tag{2.49}$$

Durch Einsetzen in Gleichung (2.43) erhält man schließlich das Ergebnis:

$$\frac{\partial E}{\partial w_{j_0 i_0}} = - \sum_{p=1}^{n_{pat}} \sum_{k=1}^{n_{out}} [\ \delta_{pk} \cdot w_{kj_0}\ ] \cdot o_{pj_0}\ (1 - o_{pj_0}) \cdot o_{pi_0} \tag{2.50}$$

(Diese Formel ist analog auch für alle weiteren, niedrigeren Stufen gültig. Der Laufindex i bezieht sich dann jeweils auf die niedrigere der beiden Schichten der betrachteten Stufe, der Laufindex j auf die höhere und der Laufindex k auf die darüberliegende Schicht. Außerdem sind die entsprechende Summationsgrenze n_k sowie die entsprechenden, in der Stufe zuvor berechneten Terme δ_{pk} einzusetzen.)

Wiederum werden die Ergebnisse aus den Gleichungen (2.46) und (2.47) (analog zur Definition von δ_{pk} in Gleichung (2.42)) in einem einzigen Term zusammengefaßt, der in die Berechnung der Ableitungen der nächstniedrigeren Stufe (falls vorhanden) eingeht. Dieser Term wird mit δ_{pj} bezeichnet:

$$\delta_{pj} = \sum_{k=1}^{n_{out}} [\ \delta_{pk} \cdot w_{kj}\ ] \cdot o_{pj}\ (1 - o_{pj}) \tag{2.51}$$

Der Name des Verfahrens, *Back-Propagation*, geht darauf zurück, daß die mit δ_{pk} und δ_{pj} bezeichneten Terme jeweils in der nächstniedrigeren Stufe bei der Berechnung der partiellen Ableitungen benötigt werden. Bildlich gesprochen werden diese Werte als „*Fehlersignale*" ausgehend von den Ausgabeunits *rückwärts* über die Verbindungen des Netzes zu den Units der *vorhergehenden* Schichten propagiert. Wie bei der (Vorwärts-) Propagation der Signale von den Eingabe- zu den Ausgabeunits werden auch diese „Fehlersignale" jeweils mit den Gewichten der Verbindungen multipliziert und anschließend aufsummiert (siehe auch Gleichung (2.46)).

Dies deutet bereits an, wie die Berechnung der partiellen Ableitungen auf einem Multiprozessorsystem realisiert werden kann: Jeder Prozessor erhält die Fehlersignale seiner Nachfolger über die entsprechenden Verbindungen, multipliziert mit dem Gewicht dieser Verbindungen. Diese Werte werden aufsummiert; anschließend wird das neue Fehlersignal berechnet sowie die partiellen Ableitungen der Verbindungen zu den Units der nächstniedrigeren Schicht bestimmt.

Auf der konzeptuellen Ebene nimmt man dabei in Kauf, daß die Verbindungen des Netzes nicht mehr gerichtete, sondern bidirektionale Kanten sind, was biologisch unplausibel ist aufgrund der Gleichrichterfunktion der Synapsen.

Die Berechnung der partiellen Ableitungen aller Gewichte des Netzes (für **ein** Patternpaar) wird aufgrund der Analogie zu einem *Propagation-Schritt* oder *Forward-Pass* als ein *Back-Propagation-Schritt* oder ein *Backward-Pass* bezeichnet.

Der Gradientenvektor der Fehlerfunktion E besteht aus den partiellen Ableitungen aller Gewichte des Netzes (sämtlicher Stufen zusammengenommen), so wie sie oben bestimmt wurden.

Wie bereits erwähnt, zeigt der Gradientenvektor einer mehrdimensionalen Funktion stets in die Richtung des größten *Anstiegs* der Funktion am Punkt der Auswertung. Das *Verfahren* des „steilsten Abstiegs" bzw. „Gradientenabstiegs" ist eine Fixpunktiteration, deren Fixpunkte die lokalen Minima der betrachteten Funktion sind. Man startet das Verfahren an einem beliebigen Punkt des Vektorraums (der durch die Variablen, von denen die Funktion abhängt, aufgespannt wird) und iteriert die folgende Vorschrift:

Man berechnet den Gradientenvektor, ausgewertet am aktuellen Punkt, und addiert ein **negatives** *Vielfaches zum aktuellen Punkt hinzu. Dies ergibt den neuen aktuellen Punkt.*

Durch diese Vorschrift folgt man schrittweise der Richtung des steilsten *Abstiegs*, wobei die Größe des verwendeten Faktors die Schrittweite bestimmt.

Im Falle eines Neuronalen Netzes besteht der aktuelle Punkt aus den Werten der Gewichte des Netzes. Die Iterationsvorschrift des Gradientenabstiegs für ein Neuronales Netz lautet:

$$\begin{array}{lrcl} & \Delta w_{ji} & = & -\ \eta \ \cdot\ \frac{\partial E}{\partial w_{ji}} \\ \text{mit} & & & \\ & w'_{ji} & = & w_{ji}\ +\ \Delta w_{ji} \end{array} \tag{2.52}$$

Der Parameter η ist die Schrittweite der Iterationsvorschrift.

Da die hier vorliegenden partiellen Ableitungen alle ein negatives Vorzeichen besitzen, hebt sich dieses mit dem negativen Vorzeichen der Iterationsvorschrift auf. Daher wird das Vorzeichen in der Berechnung der partiellen Ableitungen im Simulationsprogramm weggelassen.

Die Schrittweite des Verfahrens, η, muß möglichst klein gewählt werden, damit das Verfahren konvergiert, andererseits möglichst groß, damit dies in möglichst kurzer Zeit geschieht. Mehr zur Problematik der geeigneten Wahl dieses Parameters siehe in Kapitel 3.

In der Praxis hat sich gezeigt, daß dieses Verfahren nicht immer konvergiert (z.B. bei zu großer Wahl der Schrittweite η) bzw. daß es nicht immer das beste Minimum findet (was von diesem Verfahren auch nicht erwartet werden kann).

Um diese Schwierigkeiten zu mildern, hat sich in der Praxis die Einführung eines zusätzlichen Terms in die Iterationsformel bewährt. Es handelt sich dabei um eine bei Iterationsverfahren häufig verwendete Technik, *Relaxations*-Verfahren genannt.

Die Iterationsvorschrift mit diesem *Relaxations*-Term lautet:

$$\begin{array}{lrcl}
 & \Delta^t w_{ji} & = & -\,\eta \cdot \frac{\partial E}{\partial w_{ji}} \quad + \quad \alpha \cdot \Delta^{t-1} w_{ji} \\
\text{mit} & & & \\
 & w'_{ji} & = & w_{ji} \quad + \quad \Delta^t w_{ji}
\end{array} \tag{2.53}$$

Der Parameter α ist der *Relaxations*-Parameter.

Dieser zusätzliche Term wird im Zusammenhang mit Back-Propagation auch "*Momentum*"- oder „*Trägheits*"-Term genannt, der Parameter α dementsprechend "Momentum"- oder „Trägheits"-Faktor.

Diese Namensgebung beruht auf der Analogie dieses Terms mit der Trägheit einer Masse in der Physik. Man stellt sich den aktuellen Punkt des Iterationsverfahrens als eine Kugel vor, die eine Fläche hinunterrollt und in den Tälern (den lokalen Minima) dieser Fläche zur Ruhe kommt. Durch das Hinzufügen des Trägheitstermes bekommt die rollende Kugel Masse und damit Trägheit, wodurch sie eventuell kleinere Minima mit Hilfe ihres Schwunges überwinden kann, um schließlich in einem tieferen Minimum liegenzubleiben.

Die Wahl des Trägheitsfaktors α ist wie die Wahl der Schrittweite η nicht unkritisch (vergleiche Kapitel 3). In vielen Iterationsverfahren ist die Konvergenz nur für Werte des Relaxationsparameters zwischen Null und Zwei gegeben. Bewährt haben sich für Back-Propagation Werte von α zwischen Null und geringfügig kleiner als Eins. (Ein Wert von α gleich Null entspricht dem Verfahren *ohne* Momentumterm)

Man beachte, daß das Verfahren mit dem Trägheitsterm keinen „steilsten Abstieg" im strengen Sinne mehr realisiert.

Weitere Variationen des Verfahrens sind möglich (vergleiche Kapitel 3). Das um den Trägheitsterm erweiterte Gradientenverfahren aus Gleichung (2.53) stellt jedoch das Standard-Verfahren von Back-Propagation dar.

Der in diesem Verfahren verwendete Gradient besteht aus der Summe der Ableitungen beim Anlegen aller Patternpaare p aus der Menge P an das Netz. Mit anderen Worten: Um einen Iterationsschritt des Verfahrens auszuführen, müssen zuvor alle Patternpaare durchlaufen werden, wobei die einzelnen partiellen Ableitungen aufsummiert werden.

Man kann aber auch nach *jedem* Pattern einen Iterationsschritt, alleine basierend auf den partiellen Ableitungen dieses einen Patterns, ausführen. Dies bedeutet zwar eine Abweichung vom reinen Gradientenverfahren, in der Praxis hat sich diese Methode jedoch teilweise sogar besser bewährt als das „korrekte" Verfahren.

In den Lernregeln in Kapitel 3 wird daher manchmal nicht zwischen der Ableitung des Fehlers des Patterns, $\frac{\partial E_p}{\partial w_{ji}}$, und der Ableitung des Gesamtfehlers, $\frac{\partial E}{\partial w_{ji}}$, unterschieden, da es dem Anwender überlassen bleibt, ob er erst alle Ableitungen aufsummiert oder ob er nach jedem Anlegen eines Patterns einen Iterationsschritt ausführt.

Werden vor jedem Iterationsschritt alle Patterns angelegt und die Ableitungen aufsummiert, spricht man von "*Batch-Learning*" oder von Lernen im „*Batch*-Modus". Wird nach dem Anlegen jedes einzelnen Patterns ein Iterationsschritt durchgeführt, spricht man von "*On-Line-Learning*" oder von Lernen im „*On-Line*-Modus". (Der letztere Begriff wurde besonders durch Real-Time-Anwendungen geprägt, in denen nicht die gesamte Menge P verfügbar ist, sondern jeweils nur Proben daraus entnommen werden können, so daß es sinnvoll ist, nach jeder Probe einen Iterationsschritt auszuführen)

Wie bereits erläutert wurde, läßt sich die Berechnung der partiellen Ableitungen durch das Rückwärts-Propagieren der „Fehlersignale“ leicht auf einem parallelen Mehrprozessorsystem implementieren. Mit Hilfe dieser Fehlersignale kann auch ein Iterationsschritt zur Modifikation der Gewichte des Netzes sehr leicht parallel implementiert werden:

$$\begin{array}{rcl} \Delta_p w_{ji} & = & -\,\eta \cdot \delta_{pj} \cdot o_{pi} \\ \text{mit} \quad w'_{ji} & = & w_{ji} + \Delta_p w_{ji} \end{array} \tag{2.54}$$

(Dies gilt für beliebige Stufen; der Index j bezieht sich auf die obere, der Index i auf die untere der beiden Schichten einer Stufe)

Diese Schreibweise belegt übrigens die Verwandtschaft des Gradientenverfahrens mit der Delta-Regel (siehe Gleichung (2.29) in Abschnitt 2.7): Statt δ_{pj} heißt es dort lediglich Δ_{pj}!

Allgemeiner lautet diese Regel:

$$\begin{array}{rcl} \Delta^t w_{ji} & = & -\,\eta \cdot \sum_{p=1}^{n_{pat}} \delta_{pj} \cdot o_{pi} + \alpha \cdot \Delta^{t-1} w_{ji} \\ \text{mit} \quad w'_{ji} & = & w_{ji} + \Delta^t w_{ji} \end{array} \tag{2.55}$$

Mit Hilfe dieser Regel kann jede Unit (= jeder Prozessor) sehr einfach alle Gewichte der Verbindungen, die von den Units der nächstniedrigeren Schicht kommen, der Iterationsvorschrift entsprechend modifizieren, ohne daß dafür noch andere Prozessoren benötigt würden. Dies ist eine sehr willkommene Eigenschaft des Back-Propagation-Verfahrens, wodurch dieses Verfahren eine gewisse biologische Plausibilität erhält, denn es ist kaum vorstellbar, daß die Stärken von Synapsen im Nervensystem von anderen als den beiden betroffenen (über diese Synapse miteinander verbundenen) Zellen verändert werden können.

Man spricht im Zusammenhang mit dieser Eigenschaft, daß alle für das Netzwerk relevanten Berechnungen von den Units des Netzes durchgeführt werden können, ohne daß dafür noch andere, externe Units oder Prozessoren benötigt werden, von der “locality constraint” oder „Lokalitätsbedingung“. (Siehe auch Kapitel 3)

2.9 Initialisierung der Gewichte

Bevor ein Lernverfahren wie „Back-Propagation“ (siehe vorhergehender Abschnitt) angewendet werden kann, muß ein geeigneter Startwert für das Verfahren gewählt werden; d.h. die Gewichte des Netzes müssen auf irgendeine Weise initialisiert werden.

Zwar kann das Lernverfahren im Prinzip mit fast jedem beliebigen Startwert beginnen, manche Initialisierungen sind jedoch für die Praxis geeigneter als andere.

Unbrauchbar sind beispielsweise alle Initialisierungen, die sämtlichen Gewichten des Netzes denselben Wert zuweisen. Wenn die Lösung eines gegebenen Problems verschiedene Gewichte in den Verbindungen einer Unit erfordert (weil die Lösung der Aufgabe an dieser Stelle nicht symmetrisch ist), dann kann das Netz diese Lösung niemals finden (d.h. lernen).

Das liegt daran, daß die von dieser Unit zurückpropagierten Fehlersignale (naturgemäß) identisch sind und durch die Multiplikation mit den jeweils identischen Gewichten auf dem

Weg rückwärts durch das Netz auch dieselben Änderungen erfahren. Dadurch sind die partiellen Ableitungen der Gewichte der einander entsprechenden Verbindungen auf allen möglichen Wegen zu dieser Unit identisch, wodurch sich auch nach dem Modifizieren der Gewichte durch das Lernverfahren nichts an der Koppelung der Werte der Gewichte aneinander ändert.

Eine derartige Initialisierung stellt gewissermaßen ein lokales Maximum der Fehlerfunktion dar bzw. einen (abstoßenden) Fixpunkt des Lernverfahrens: Der Iterationsvorschrift gelingt es nicht, die einmal vorgegebene Symmetrie des Systems zu durchbrechen. Wird das Verfahren allerdings in einem anderen Punkt gestartet, wird dieser Fixpunkt nie erreicht; der Einzugsbereich dieses Fixpunktes besteht nur aus dem Punkt[9] selbst, er ist abstoßend.

Um das Lernvermögen des Netzes also nicht einzuschränken, ist es wichtig, alle Gewichte mit *verschiedenen* Werten zu initialisieren. Auf diese Weise werden durch den Startwert bedingte, das Netz in eine falsche Richtung führende Symmetrien gebrochen ("*symmetry breaking*", siehe Rumelhart/McClelland [24] oder McClelland/Rumelhart [17]).

Das schließt jedoch nicht aus, daß die der Lösung einer Aufgabe inhärenten Symmetrien gefunden (also gelernt) werden können.

Die am einfachsten zu realisierende Art der Initialisierung besteht in der Verwendung von Zufallszahlen. Zwar werden dadurch vereinzelt identische Gewichte nicht völlig ausgeschlossen, diese sind aber sehr unwahrscheinlich und in den meisten Fällen (wenn vereinzelt auftretend) unkritisch oder zumindest tolerierbar. (Im ungünstigsten Fall muß der Lernversuch abgebrochen und das Netz neu initialisiert werden)

Es hat sich eingebürgert, *gleichverteilte* Zufallszahlen zu verwenden, wohl weil diese in der Regel auf allen Rechnern zur Verfügung stehen.

Gleichverteilte Zufallszahlen liegen in einem bestimmten Intervall; man wählt normalerweise einen symmetrischen Bereich um den Nullpunkt herum und bezeichnet die (positive) Schranke mit r: $[-r, +r]$.

Im Prinzip können jedoch auch anders verteilte Zufallszahlen verwendet werden, z.B. normal verteilte Zahlen mit dem Mittelwert Null und der Standardabweichung r.

Der Parameter r, wie auch die Initialisierung der Gewichte mit Zufallszahlen überhaupt, hat einen nicht zu vernachlässigenden Einfluß auf das Lernverhalten des Netzes, der aber leider nicht genau zu kontrollieren ist. Je nach Zufall und nach Problem können die benötigten Lernzeiten mehr oder weniger stark schwanken. Es kommt sogar vor, daß das Lernverfahren aufgrund einer ungünstigen Initialisierung gelegentlich steckenbleibt.

Man wählt daher den Parameter r relativ klein; einmal, um dem Netz keine falschen Vorgaben zu machen (um Umwege beim Lernen zu ersparen und das Steckenbleiben zu verhindern), andererseits, weil Gewichte nahe bei Null die Netzeingaben der Units nahe dem Arbeitspunkt der logistischen Aktivierungsfunktion halten, d.h. im linearen Bereich dieser Funktion (im Gegensatz zu den Sättigungsbereichen, in denen kleinere Unterschiede in der Eingabe keine Unterschiede in der Ausgabe mehr hervorrufen). Letzteres hat auch den Vorteil, daß die Ableitungen der Aktivierungsfunktion am größten sind, wodurch das Netz zu Beginn am schnellsten lernt.

[9] bzw. aus der Menge der äquivalenten Punkte, d.h. der Menge der Belegungen aller Gewichte des Netzes mit jeweils demselben (beliebigen) Wert

In der Regel verwendet man für den Parameter r Werte in der Größenordnung von 0.1 bis 0.2 (manche Autoren verwenden vereinzelt auch wesentlich größere Werte wie z.B. 1.0 oder 2.0).

2.10 Abbruchkriterien und Fehlermaße

Jede Fixpunktiteration, wie z.B. „Back-Propagation", benötigt neben der Iterationsvorschrift und einem Startwert auch ein Abbruchkriterium, um die Iteration abzubrechen, sobald ein Fixpunkt gefunden ist.

Theoretisch ist die Iteration dann beendet, wenn sich durch Anwendung der Iterationsvorschrift φ der aktuelle Punkt x^* nicht mehr verändert, d.h. wenn gilt:

$$x^* = \varphi(x^*) \tag{2.56}$$

Aufgrund der endlichen Stellenanzahl des Rechners und den damit verbundenen Rundungsfehlern in der Berechnung der Iterationsvorschrift φ wird diese Bedingung in den seltensten Fällen numerisch exakt erfüllt.

Man umgeht diese Schwierigkeit in der Regel dadurch, daß man eine Fehlerschranke ϵ festlegt, unter die die Differenz aus dem aktuellen und dem vorherigen Punkt fallen muß, damit der zuletzt berechnete Punkt als Fixpunkt akzeptiert wird:

$$\| x^* - \varphi(x^*) \| < \epsilon \tag{2.57}$$

Bei Back-Propagation ist dieses Abbruchkriterium äquivalent zu:

$$\| \Delta \vec{w} \| < \epsilon \tag{2.58}$$

(Vergleiche Gleichung (2.53) des Standard Back-Propagation-Verfahrens im Abschnitt 2.8)

Vermutlich aus historischen Gründen wird dieses Abbruchkriterium jedoch für das Lernen in konnektionistischen Modellen nicht verwendet. Es wäre allerdings sicherlich eine Untersuchung wert, ob dieses Kriterium nicht in manchen Fällen sogar geeigneter wäre als die üblicherweise benutzten, im Folgenden vorgestellten Kriterien.

Fahlman [7] zählt in seiner Arbeit einige häufig verwendete Erfolgskriterien für das Lernen mit Back-Propagation auf (einige davon sind nur für *binäre* Targets (Null oder Eins) geeignet):

- Das "*sharp threshold*"- (scharfer Schwellenwert) Kriterium: Jeder Ausgabewert größer oder gleich 0.5000000 wird als Eins angesehen, während jeder Ausgabewert kleiner als diese Schwelle als Null gilt. Sobald alle Ausgabeunits für alle zu lernenden Patterns die jeweils richtigen Werte liefern, ist das Lernen beendet. Diese auf der Hand liegende Einteilung hat jedoch ihre Nachteile, beispielsweise dann, wenn man sie in analogen Hardware-Realisierungen einsetzt, da dort bereits das kleinste bißchen Rauschen den Wert von einigen Ausgabe-Bits umkippen lassen kann.

- Das "*small individual error*"- (kleiner individueller Fehler) Kriterium: Der Ausgabewert einer jeden Ausgabeunit muß für alle zu lernenden Patterns innerhalb einer Schranke ϵ um den entsprechenden Targetwert herum liegen. Falls das Netzwerk eine Aufgabe mit binären Zielwerten lösen soll, ist dieses Kriterium jedoch unnötig streng, da nur von Interesse ist, ob eine bestimmte Ausgabe den Wert Null oder Eins repräsentiert.

- Das "*small composite error*"- (kleiner globaler Fehler) Kriterium: Die Summe über alle Ausgabeunits und alle zu lernenden Patterns der quadrierten Differenzen zwischen Ausgabe- und Targetwert muß unter eine gewisse Fehlerschranke ϵ fallen. Für binäre Applikationen (zumindest solange ϵ nicht sehr klein ist) ist dieses Kriterium nicht sehr geeignet, da *alle* Ausgabe-Bits korrekt sein sollen, und nicht ein großer Fehler an einer Ausgabeunit durch einige sehr kleine Fehler an anderen Ausgabeunits ausgeglichen werden soll.

- Das "*winner takes all*"- (Gewinner bekommt alles) Kriterium: Der Wert der „richtigen" Ausgabeunit muß größer sein als der Wert jeder anderen Ausgabeunit. Dieses Kriterium ist allerdings nur anwendbar, wenn die Ausgabeunits jeweils disjunkte Ereignisse darstellen sollen, d.h. wenn die Aktivierung einer Ausgabeunit die Aktivierung aller übrigen Ausgabeunits ausschließen soll (z.B. bei Klassifikationsproblemen). Diese Methode hat den Nachteil, wie das *sharp threshold*-Kriterium für Rauschen anfällig zu sein, falls die größten zwei Aktivierungen fast gleich groß sind. Außerdem ist in einer Hardware-Realisierung ein zusätzlicher Schaltkreis zur Bestimmung der Ausgabeunit mit der größten Aktivierung erforderlich.

- Das "*threshold with margin*"- (Schwellenwert mit Übergangsbereich) Kriterium: Man wählt einen Schwellenwert wie für das *sharp threshold*-Kriterium, beispielsweise 0.5, und betrachtet alle Ausgaben, die nahe bei diesem Schwellenwert liegen, als inkorrekt. Dazu legt man außer dem Schwellenwert noch einen Bereich um diesen Schwellenwert herum fest. Z.B. vereinbart man, alle Werte unterhalb von 0.4 als Null anzusehen, alle Werte oberhalb von 0.6 als Eins und alle Werte dazwischen als unbestimmt. Im Gegensatz zum *sharp threshold*-Kriterium ist dieses Kriterium viel unempfindlicher gegenüber Rauschen.

Eine Mischung aus dem "*small individual error*"- (kleiner individueller Fehler) und dem "*small composite error*"- (kleiner globaler Fehler) Kriterium ist das folgende, das ich "*small local error*"- (kleiner lokaler Fehler) Kriterium nenne: Für *jedes* Pattern muß der Wert einer geeignet gewählten Fehlerfunktion, die jeweils von den Ausgabe- und Targetwerten *eines* Patterns abhängt, unter eine gewisse Fehlerschranke, ϵ, fallen.

Diese Fehlerfunktion kann mit der Fehlerfunktion, die bei der Berechnung der partiellen Ableitungen eines Patterns zugrundegelegt wird (siehe Gleichung (2.32) in Abschnitt 2.8), identisch sein, muß es aber nicht. Die Fehlerfunktion für das "*small local error*"-Kriterium nenne ich "*local error*" (lokaler Fehler).

Die Fehlerschranke ϵ für dieses Kriterium bezeichne ich als "*pattern error limit*" (Pattern-Fehlerschranke), im Unterschied zur Fehlerschranke des "*small composite*"- bzw. "*small global error*"-Kriteriums, die ich als "*absolute*" oder "*global*" *error limit* bezeichne.

Allgemeiner formuliert, muß beim "*small global error*"-Kriterium ebenfalls eine (geeignet gewählte) Fehlerfunktion unter eine Fehlerschranke fallen, diese Fehlerfunktion wird allerdings auf der gesamten Menge der zu lernenden Patterns definiert und daher als "*global error*" (globaler Fehler) bezeichnet.

Im einfachsten Fall besteht der *globale Fehler* aus der *Summe* der *lokalen Fehler* der einzelnen Patterns. Falls außerdem der lokale Fehler mit der Fehlerfunktion aus Gleichung (2.32) (Abschnitt 2.8) identisch ist, entspricht der *globale Fehler* dem Fehlermaß, das zur Berechnung der partiellen Ableitungen verwendet wird (siehe Gleichung (2.33) in Abschnitt 2.8).

Dieses Fehlermaß wird als "LMS (Least Mean Squares) error measure" („kleinstes mittleres Quadrate-Fehlermaß") bezeichnet.

Statt der Fehlerfunktion (2.32) aus Abschnitt 2.8 wird als lokaler Fehler in der Regel jedoch eine normierte Version der euklidischen Norm verwendet, deren Größe durch die Normierung von der Anzahl der Ausgabeunits unabhängig wird. Eine derartige Fehlerfunktion wird "*RMS (Root Mean Square) error measure*" genannt:

$$E_{local} = \sqrt{\frac{1}{n_{out}} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2} \tag{2.59}$$

(Bei Verwendung der symmetrischen Aktivierungsfunktion (siehe Kapitel 3.8.1) wird dieser Fehler zusätzlich durch Zwei geteilt, um den größeren Wertebereich (von minus Eins bis Eins statt von Null bis Eins) zu kompensieren)

Dazu analog wird als globaler Fehler in der Regel statt des LMS-Fehlermaßes (Gleichung (2.33) in Abschnitt 2.8) ebenfalls das RMS-Fehlermaß verwendet:

$$\begin{aligned} E_{global} &= \sqrt{\frac{1}{n_{pat}} \sum_{p=1}^{n_{pat}} (E_{local})^2} \\ &= \sqrt{\frac{1}{n_{pat}} \sum_{p=1}^{n_{pat}} \frac{1}{n_{out}} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2} \\ &= \sqrt{\frac{1}{n_{pat} \cdot n_{out}} \sum_{p=1}^{n_{pat}} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2} \end{aligned} \tag{2.60}$$

Durch die Division des RMS-Fehlermaßes durch die Anzahl seiner Summanden wird die Größe des Fehlers von der Anzahl der zu lernenden Patterns und der Anzahl der Ausgabeunits unabhängig und damit *problemunabhängig*.

Dies hat für Experimente mit vielen verschiedenen Problemen mehrere Vorteile; z.B. läßt sich die Grafik zur Anzeige des Gesamt- (globalen) Fehlers während des Lernens problemunabhängig dimensionieren. Auch lassen sich die Werte der Fehlerschranken (ϵ) viel leichter festlegen, da ein Fehler von (beispielsweise) 0.1 immer (in etwa) gleich „gut" (oder „schlecht") ist.

Man muß nur beachten, daß die Schritte des RMS-Fehlermaßes während des Lernens je nach der Anzahl der Patterns und der Output-Units eines Problems verschieden „fein" sind: Bei einem Problem mit wenigen Patterns und/oder wenigen Ausgabeunits (also mit wenigen Summanden in der RMS-Fehlerfunktion) kann sich eine Verbesserung des Fehlers einer einzigen Ausgabeunit beispielsweise bereits in der zweiten oder dritten Dezimale des RMS-Fehlers bemerkbar machen, während sich bei einem größeren Problem die gleiche Verbesserung an einer Unit vielleicht erst in der sechsten oder siebten Dezimale nach dem Komma zeigt.

Eine Variation des "*small global error*"-Kriteriums besteht darin, statt der Summe (bzw. der Wurzel der gemittelten Summe) das *Maximum* der lokalen Fehler unter eine Fehlerschranke ϵ zu drücken (also im Gegensatz zum "*small local error*"-Kriterium nicht alle lokalen Fehler unter dieselbe Fehlerschranke, sondern den *größten* lokalen Fehler, so daß alle anderen lokalen Fehler im Allgemeinen sogar noch wesentlich kleiner sind). Dieses Kriterium bezeichne ich als „*Maximum-Norm*-Kriterium".

In meinem Programm sind als Abbruchkriterien das "*small global error*"-Kriterium (mit RMS-Fehlerfunktion) und das „*Maximum-Norm*"-Kriterium implementiert, zwischen denen man durch Setzen eines Flags im Lernparameter-Menü hin- und herschalten kann.

Zusätzlich findet in vielen Lernmodi (vergleiche nächsten Abschnitt) das "*small local error*"-Kriterium (mit RMS-Fehlerfunktion) Verwendung: Falls der lokale Fehler eines Patterns unterhalb des "Pattern error limits" liegt, werden die Gewichte des Netzes nicht modifiziert, und es wird mit dem nächsten Pattern fortgefahren.

Für Klassifikationsprobleme läßt sich außerdem eine weitere Option im Lernparameter-Menü hinzuschalten, die einen "Penalty"-Term zum lokalen Fehler hinzuaddiert, der nicht eindeutige oder falsche Klassifikationen „bestraft": Dieser Term ist umso größer, je geringer der Abstand zwischen dem Maximum der Ausgabe und dem Maximum der übrigen Ausgabeunits (also dem nächstniedrigeren Wert) ist. Ist das Maximum der Ausgabe an der falschen Stelle (d.h. nicht an derselben Stelle wie das Maximum der Target-Werte), wird zusätzlich die Konstante 1.0 zum lokalen Fehler hinzuaddiert. Der Penalty-Term selbst ist im Wesentlichen eine Differenz von Differenzen: Zuerst wird die Differenz zwischen Maximum und Minimum der Targetwerte bestimmt. Anschließend wird von dieser die Differenz zwischen dem Maximum der Ausgabe und dem nächstniedrigeren Wert abgezogen. Zuletzt wird dieser Wert noch durch die Differenz aus Maximum und Minimum der Targetwerte geteilt und damit auf den Bereich von Null bis Eins normiert. (Dies für den Fall, daß statt der Targetwerte Null und Eins andere Werte verwendet werden, wie z.B. die Werte 0.1 und 0.9 oder wie im Falle der symmetrischen Aktivierungsfunktion (siehe Kapitel 3.8.1))

Zu beachten ist, daß dieser Penalty-Term *nicht* in die Berechnung des (globalen) RMS-Fehlers eingeht, sondern nur in das „*Maximum-Norm*"- und das "*small local error*"-Kriterium sowie in die Berechnung der Gewichte *der Patterns* beim "Dynamic"-Lernmodus (siehe folgender Abschnitt).

Abschließend bleibt noch hinzuzufügen, daß manche Lernverfahren bei bestimmten Problemen gar nicht oder aufgrund einer zufällig ungünstigen Initialisierung der Gewichte des Netzes nicht konvergieren. Aus diesem Grund ist es zweckmäßig, zusätzlich zu dem Abbruchkriterium noch eine Abfrage vorzusehen, die das Lernen nach einer bestimmten Maximalzahl von Durchläufen abbricht, so daß ein menschlicher Beobachter entscheiden kann, ob sich eine Fortsetzung des Lernens lohnt.

Noch zweckmäßiger, besonders für Experimente, ist die laufende Anzeige des globalen Fehlers, wie er nach jedem Durchlauf errechnet wird; am günstigsten in Form einer Grafik. Auf diese Weise kann der Experimentator den Lernfortschritt besser beurteilen und bei stationärem oder steigendem Fehler das Lernen abbrechen.

2.11 Lernmodi und Komplexitätsmaße

Wie im Abschnitt 2.8 bereits angedeutet, gibt es verschiedene Möglichkeiten, einem Neuronalen Netz die zu lernenden Patterns zu präsentieren und dabei die Gewichte des Netzes zu modifizieren.

Man unterscheidet grundsätzlich zwei verschiedene Vorgehensweisen:

Die eine besteht darin, die partiellen Ableitungen aller zu lernenden Patterns nacheinander zu bestimmen und *aufzusummieren*, bevor anschließend die Gewichte des Netzes modifiziert werden. Dies wird solange wiederholt, bis das Netz die gewünschte Abbildung beherrscht. Man bezeichnet dieses Verfahren als "Batch-Learning" oder als Lernen im "Batch"-Modus (vergleiche Abschnitt 2.8).

Die andere Methode besteht darin, die Gewichte des Netzes separat für jedes einzelne der zu lernenden Patterns anzupassen. Dazu werden die Patterns der Reihe nach angelegt und die Gewichte jeweils *nach jedem* Pattern modifiziert. Man nennt diese Methode historisch bedingt "On-Line-Learning" oder Lernen im "On-Line"-Modus. Obwohl diese Methode eine Abweichung vom „korrekten" Gradientenabstieg bedeutet, hat sie sich in der Praxis sehr bewährt.

Diese Unterscheidung ist jedoch nur von theoretischer Bedeutung, da es in der Praxis eine ganze Reihe von verschiedenen Lernkonzepten, hier „Lernmodi" genannt, gibt, die sich nicht mehr in dieses Schema pressen lassen, wie die folgenden Abschnitte zeigen werden.

2.11.1 "Sequential Learning"

Die Idee des "Sequential Learning" oder des „sequentiellen Lernmodus'" besteht darin, das Lernen jedes einzelnen Patterns so oft zu wiederholen, bis das Netzwerk die gewünschte Ausgabe für dieses Pattern mit genügender Genauigkeit (d.h. mit genügend kleinem *lokalem Fehler*) reproduziert, bevor mit dem nächsten Pattern in der gleichen Weise fortgefahren wird.

Sind alle Patterns durchlaufen, sollte das Netzwerk theoretisch in der Lage sein, für alle Patterns die gewünschten Ausgaben zu liefern. In der Praxis ist dies jedoch selten auf Anhieb der Fall, da sich die von den verschiedenen Patterns hervorgerufenen Gewichtsänderungen in der Regel mehr oder weniger überlagern. (Je weniger sie sich überlagern, desto schneller konvergiert das "sequential learning". Bei extrem ungünstigen Überlagerungen kann es jedoch auch vorkommen, daß das sequentielle Lernen steckenbleibt oder divergiert)

Man bestimmt daher nach Durchlauf aller Patterns den *globalen Fehler* und beginnt einen erneuten Durchlauf aller Patterns, falls das *Abbruchkriterium* nicht erfüllt ist. (Für eine Erläuterung der Begriffe *Abbruchkriterium*, *lokaler* und *globaler* Fehler siehe Abschnitt 2.10)

Manchmal kann das Lernen eines Patterns sehr lange dauern, wenn die augenblickliche Konfiguration der Gewichte gerade ungünstig ist. Statt darauf zu warten, daß das Netzwerk dieses Pattern lernt, kann es sinnvoller sein, nach einer bestimmten, vorher festgelegten Anzahl von Schleifendurchläufen das Lernen dieses Patterns abzubrechen und mit dem nächsten Pattern fortzufahren, in der (meist begründeten) Hoffnung, daß das Lernen der übrigen Patterns im nächsten Durchlauf der Patterns eine für das Lernen dieses Patterns günstigere Ausgangssituation schafft.

Abbildung 2.4 (Seite 57) zeigt die Kontrollstruktur des Lernens im sequentiellen Modus.

2.11.2 "Periodical Learning"

Die Idee des "Periodical Learning" oder des „periodischen Lernmodus'" besteht darin, *periodisch* immer wieder alle Patterns (stets in derselben Reihenfolge) zu durchlaufen, dabei nach *jedem* Pattern die Gewichte zu modifizieren und dies solange zu wiederholen, bis das Netz für jedes Pattern die gewünschten Ausgaben mit genügender Genauigkeit reproduziert, d.h. bis das Abbruchkriterium erfüllt bzw. der globale Fehler klein genug geworden

Abbildung 2.4: Struktogramm des Lernens mit "Sequential Learning" ist. Der periodische Lernmodus ist aufgrund des Umstandes, daß nach jedem Pattern die Gewichte des Netzes modifiziert werden, ein typischer Vertreter des "On-Line-Learning".

Häufig wird jedoch eine Variante benutzt, die dem Netz jedes Pattern gleich mehrere Male hintereinander präsentiert und die Gewichte modifiziert. Wie oft das Lernen eines Patterns wiederholt wird, bestimmt ein Parameter, der "*repeat factor*" genannt wird.

Da es in der Regel unnötig ist, bereits sehr gut gelernte Patterns weiter zu trainieren, wird zusätzlich eine Abfrage in das Programm eingefügt, die das Training eines Patterns abbricht oder ggfs. gar nicht erst beginnt, sondern gleich zum nächsten Pattern übergeht, falls der lokale Fehler des betreffenden Patterns unterhalb der angestrebten Fehlerschranke ϵ, "pattern error limit" genannt, liegt.

Zu beachten ist, daß das periodische Lernen mit dieser Abfrage mit dem sequentiellen Lernen äquivalent ist, vorausgesetzt es wird ein genügend großer "repeat factor" verwendet.

Den Fluß der Verarbeitung im periodischen Lernmodus zeigt die folgende Abbildung 2.5 (Seite 58).

Abbildung 2.5: Struktogramm des Lernens mit "Periodical Learning"

2.11.3 "Sum-of-Derivatives" Batch-Modus

Dies ist der klassische Batch-Lernmodus bzw. die dem „korrekten" Gradientenabstiegsverfahren zugrundeliegende Vorgehensweise: Alle zu lernenden Patterns werden periodisch durchlaufen, die jeweiligen partiellen Ableitungen werden berechnet und *aufsummiert*. Nach je einem Durchgang durch alle Patterns werden die Gewichte des Netzes entsprechend der Iterationsvorschrift des Gradientenverfahrens (mit Trägheitsterm) oder einer anderen Lernregel (siehe Kapitel 3) modifiziert. Dies wird solange wiederholt, bis das gewählte Abbruchkriterium (vergleiche Abschnitt 2.10) erfüllt ist.

Abbildung 2.6 (Seite 59) zeigt den Verlauf des Lernens im (Standard) Batch-Modus.

Abbildung 2.6: Struktogramm des Lernens mit "Sum-of-Derivatives" Batch Learning

2.11.4 "Sum-of-Updates" Batch-Modus

Das "Sum-of-Updates" Batch-Learning ist ein Verfahren, bei dem periodisch alle zu lernenden Patterns durchlaufen werden, wobei nach *jedem* Pattern eine Modifikation der Gewichte des Netzes berechnet wird. Demnach könnte man dieses Verfahren als einen Vertreter des "On-Line-Learning" ansehen. Die Gewichte des Netzes werden jedoch nicht nach jedem Pattern modifiziert, vielmehr werden die errechneten Modifikationen *aufsummiert*, und erst wenn alle zu lernenden Patterns durchlaufen sind, wird die akkumulierte Summe der Modifikationen zu den Gewichten des Netzes hinzuaddiert. Insofern hat dieses Verfahren eher Ähnlichkeit mit dem (Standard) Batch-Modus aus dem vorangegangenen Abschnitt.

Diese Durchläufe werden solange wiederholt, bis das gewählte Abbruchkriterium (vergleiche Abschnitt 2.10) erfüllt ist.

Die Bezeichnung dieses Verfahrens als "Batch"-Verfahren erhält ihre Rechtfertigung aus dem Umstand, daß bei Verwendung der Standard-Lernregel, d.h. dem Gradientenabstieg des Standard-Back-Propagation-Verfahrens, das "Sum-of-Updates" und das "Sum-of-Derivatives" Batch-Learning äquivalent sind, und zwar aufgrund der Linearität der Gradientenabstiegsregel (siehe Abschnitt 2.8).

Nur bei Verwendung nicht-linearer Lernregeln, wie sie in Kapitel 3 vorgestellt werden, besteht ein Unterschied zwischen dem "Sum-of-Updates" und dem "Sum-of-Derivatives" Batch-Learning.

Abbildung 2.7 (Seite 60) zeigt das Verfahren des "Sum-of-Updates" Batch-Learning.

Abbildung 2.7: Struktogramm des Lernens mit "Sum-of-Updates" Batch Learning

Zu beachten ist, daß beim "Sum-of-Updates" Batch-Learning der Trägheitsterm nicht jeweils zu den einzelnen Modifikationen hinzuaddiert wird, die anhand der einzelnen Patterns errechnet und aufsummiert werden, sondern direkt zu den Gewichten des Netzes, sobald die Summe der Modifikationen nach einem Durchlauf durch alle Patterns zu den Gewichten des Netzes hinzuaddiert wird. Die Summe der errechneten Modifikationen des einen Durchlaufs stellt gleichzeitig den Delta-Wert dar, der, mit dem Trägheitsfaktor α multipliziert, den Trägheitsterm des darauffolgenden Durchlaufs bildet.

Da dieser Trägheitsterm die Effektivität der meisten nicht-linearen Lernregeln (siehe Kapitel 3) negativ beeinflussen würde, muß der Trägheitsterm bei Verwendung dieser Lernregeln abgeschaltet werden, indem der Trägheitsfaktor α im Parameter-Menü des Simulationsprogramms auf Null gesetzt wird.

2.11.5 "Dynamic Learning"

Das "Dynamic Learning" ist eine Eigenentwicklung, deren Anliegen es war, die „wichtigeren“ Patterns innerhalb der Menge der zu lernenden Patterns intensiver zu trainieren als die nicht so „wichtigen“. Bei Verwendung der hier bereits vorgestellten, „klassischen“ Lernverfahren wie Sequential, Periodical und Batch Learning stellt man außerdem fest, daß der

Lernerfolg aufgrund des Determinismus dieser Verfahren wesentlich von der Initialisierung der Gewichte des Netzes abhängt und sie bei der Fixpunktiteration, auf denen alle diese Lernverfahren beruhen, häufig vorzeitig in lokalen Minima der verwendeten Fehlerfunktion zum Stillstand kommen. Ein gewisses Maß an zufälliger Streuung kann dagegen (analog zum "Simulated Annealing" der Boltzmann-Machine; vergleiche Rumelhart/McClelland [24]) diese lokalen Minima überwinden helfen.

Eine wesentliche Idee des "Dynamic Learning" besteht darin, jedem der zu lernenden Patterns eine reelle Zahl zuzuordnen. Diese reelle Zahl nenne ich das „*Gewicht*" eines Patterns.

Das Gewicht eines **Patterns** *darf jedoch nicht mit den Gewichten der Verbindungen innerhalb eines Neuronalen Netzes verwechselt werden.*

Diese Gewichte können unter anderem dazu verwendet werden, um eine (anderweitig vorgegebene) relative Wichtigkeit der einzelnen Patterns festzulegen, oder sie können die Wahrscheinlichkeiten einer Wahrscheinlichkeitsverteilung darstellen, falls es sich bei den zu lernenden Patterns um eine Stichprobe aus einer Grundgesamtheit handelt.

Im Falle des "Dynamic Learning" werden diese Gewichte dazu benutzt, um die Patterns mit dem größten *lokalen Fehler* häufiger zu trainieren als solche, die das Netz bereits besser beherrscht.

Dazu wird vor Beginn des eigentlichen Lernens der lokale Fehler jedes einzelnen Patterns berechnet und mit diesem zusammen abgespeichert.

Während des Lernens werden die zu lernenden Patterns nicht periodisch durchlaufen, sondern *zufällig* ausgewählt. Die Wahrscheinlichkeit dafür, daß ein bestimmtes Pattern ausgewählt wird, ist dabei gleich dem lokalen Fehler dieses Patterns, geteilt durch die Summe aller Gewichte (Normierung der lokalen Fehler auf das Intervall $[0, 1]$, so daß ihre Summe Eins ergibt).

Realisiert wird dies durch die Erzeugung einer Zufallszahl mit Hilfe eines Zufallszahlengenerators, der gleichverteilte Zufallszahlen in einem beliebigen Intervall liefert, hier im Intervall von Null bis zur aktuellen Summe der Gewichte aller Patterns. Anschließend wird die Liste der zu lernenden Patterns durchlaufen, wobei jeweils das Gewicht des betreffenden Patterns von der erzeugten Zufallszahl subtrahiert wird, bis diese Zahl Null oder negativ wird. Das Pattern, bei dem dies eintritt, ist das gesuchte.

Diese zufällige Auswahl der Patterns hat nicht nur die gewünschte Eigenschaft, daß Patterns mit einem größeren lokalen Fehler häufiger ausgewählt und damit intensiver gelernt werden, sondern bewirkt auch genügend Streuung, so daß das Verfahren etwas unabhängiger von der anfänglichen Initialisierung der Gewichte des Netzes wird und kleinere lokale Minima in der Regel von selbst überwinden kann.

Nachdem ein Pattern ausgewählt worden ist, wird dieses dem Netz eingegeben. Daraufhin werden die Ausgaben des Netzes berechnet, dann die partiellen Ableitungen und schließlich die erforderlichen Modifikationen der Gewichte des Netzes (die Gewichte des Netzes werden also nach jedem Pattern modifiziert). Bei diesem Lernmodus handelt es sich also um ein "On-Line"-Lernverfahren.

Wie bei den übrigen Lernmodi wird das Trainieren der Patterns solange fortgesetzt, bis das Abbruchkriterium erfüllt ist.

Abbildung 2.8 (Seite 63) zeigt den Ablauf des "Dynamic Learning".

Zu beachten ist, daß sich im Allgemeinen die lokalen Fehler *aller* Patterns ändern, sobald die Gewichte des Netzes modifiziert werden. Es muß daher stets dafür gesorgt werden, daß die Gewichte *der Patterns* auf den neuesten Stand gebracht werden: Im Prinzip nach jedem Anlegen eines Patterns.

In der Praxis wäre dies jedoch äußerst rechenzeitaufwendig. In der jetzigen Implementation des "Dynamic Learning" wird daher nur das Gewicht *des gerade gelernten* Patterns neu berechnet, und erst nach einer bestimmten Anzahl von gelernten Patterns (die einstellbar ist) werden periodisch die Gewichte *aller* Patterns neu berechnet.

Das Trainieren der vom Anwender festgelegten Anzahl von Patterns mit anschließender Neuberechnung der Gewichte aller Patterns wird in Analogie zu den übrigen Lernmodi als „ein Durchlauf" durch alle Patterns bezeichnet, obwohl nicht notwendigerweise jedes Pattern auch tatsächlich, bedingt durch die zufällige Auswahl, mindestens einmal trainiert wird.

"Dynamic Learning" heißt dieser Lernmodus deshalb, weil sich die relative „Wichtigkeit" eines jeden Patterns dynamisch, je nach den bereits erzielten Lernfortschritten anpaßt und das Verfahren sich dadurch stets besonders auf die jeweils „problematischen" Patterns konzentriert.

Analog zum "Periodical Learning" besteht beim "Dynamic Learning" die Möglichkeit, das einmal ausgewählte Pattern mehrmals hintereinander zu trainieren, wobei die Anzahl der Wiederholungen durch denselben "repeat factor" festgelegt wird wie für das "Periodical Learning". Ebenfalls analog wird das wiederholte Lernen eines Patterns abgebrochen oder gar nicht erst begonnen, falls der lokale Fehler das "pattern error limit" unterschreitet.

2.11.6 "Dynamic Batch Learning"

Das "Dynamic Batch Learning" ist eine Kombination aus Dynamic und Batch Learning.

Im Prinzip handelt es sich bei diesem Lernmodus um Dynamic Learning. Der einzige Unterschied zum Dynamic Learning besteht darin, daß (im Falle des Dynamic "Sum-of-Derivatives" Batch Learning) die partiellen Ableitungen bzw. (im Falle des Dynamic "Sum-of-Updates" Batch Learning) die errechneten Modifikationen (der Gewichte des Netzes) während eines Durchgangs (der, bedingt durch das zufällige Auswahlprinzip, eine beliebige Anzahl von Patternpräsentationen umfassen kann) *aufsummiert* werden, die Gewichte des Netzes also nicht mehr nach *jedem* Pattern modifiziert werden. Erst nach einem Durchgang, wenn die Gewichte der Patterns neu berechnet werden, werden zuvor die Gewichte des Netzes modifiziert. Im Falle des Dynamic "Sum-of-Derivatives" Batch Learning durch Anwenden der gewählten Lernregel auf die gebildete Summe der Ableitungen und anschliessender Applikation der berechneten Modifikationen auf die Gewichte des Netzes oder, im Falle des Dynamic "Sum-of-Updates" Batch Learning, durch Hinzuaddieren der aufsummierten Modifikationen zu den Gewichten des Netzes, einschließlich des Trägheitsterms (siehe im Abschnitt 2.11.4 über "Sum-of-Updates" Batch Learning, wie der Trägheitsterm in diesem Falle berechnet wird).

Wie im Falle des "Sum-of-Derivatives" und des "Sum-of-Updates" Batch Learning gilt auch hier, daß bei Verwendung der Standard-Lernregel, des steilsten Abstiegs (Abschnitt 2.8), das Dynamic "Sum-of-Derivatives" und das Dynamic "Sum-of-Updates" Batch Learning äquivalent sind aufgrund der Linearität dieser Lernregel.

Abbildung 2.8: Struktogramm des Lernens mit “Dynamic Learning”

Die Abbildung 2.9 (Seite 64) und die Abbildung 2.10 (Seite 65) verdeutlichen die Funktionsweise dieser zwei Lernmodi.

Abbildung 2.9: Struktogramm des Lernens mit Dynamic "Sum-of-Derivatives" Batch Learning

2.11.7 Zyklen, Epochen und andere Komplexitätsmaße

Die wesentliche Grundlage zum Vergleich der Effizienz verschiedener Lernverfahren sind geeignete Komplexitätsmaße.

Während die Ermittlung des bei der Netzwerksimulation benötigten Speicherplatzes sehr einfach ist, im Wesentlichen gegeben durch die Größe der benötigten Matrizen, plus einigen Lauf- und Zwischenspeicherungsvariablen, ist die Festlegung geeigneter Maße für die verbrauchte Rechenzeit, das wichtigste Vergleichskriterium bei der Simulation Neuronaler Netze auf seriellen Rechnern, ziemlich schwierig.

Die zum Trainieren eines bestimmten Problems erforderliche Zeit ist einmal von der Problemgröße, der Anzahl und Länge der zu lernenden Vektoren (Patterns), von der Komplexität des Problems selbst und von dem gewählten Abbruchkriterium abhängig. Für die Komplexität eines Problems gibt es bis heute so gut wie keine praktikablen Maße (siehe dazu auch unter dem Stichwort „lineare Separierbarkeit“ in Kapitel 3.6).

Abbildung 2.10: Struktogramm des Lernens mit Dynamic "Sum-of-Updates" Batch Learning

Zum anderen ist die verbrauchte Rechenzeit von der Implementation der Simulation abhängig. Ein möglichst allgemein geschriebenes Programm, ein Netzwerksimulator, der die Simulation möglichst vieler verschiedener Netzwerke erlaubt, ist in der Ausführung stets langsamer als ein an eine spezielle Aufgabe angepaßtes und optimiertes Programm. Außerdem werden meist mehr oder weniger aufwendige Graphiken zur Anzeige des Geschehens während des Lernens eingesetzt, die die Ausführung des Programmes zusätzlich verlangsamen.

Und schließlich hängt die benötigte Rechenzeit nicht zuletzt wesentlich vom verwendeten Rechner ab, von der Mächtigkeit seines Befehlssatzes, der Breite seiner internen Zahlendarstellung und seiner Taktfrequenz.

Bei der Verwendung von parallelen Multiprozessorsystemen hängt die Rechenzeit zusätzlich von Art und Anzahl der Prozessoren, ihrer Verbindungsstruktur und der Parallelisierbarkeit der verwendeten Algorithmen ab.

Aus diesen Gründen verbietet sich die Verwendung der Rechenzeit als Vergleichskriterium zur Beurteilung unterschiedlicher Verfahren von vornherein; ganz davon abgesehen, daß die benötigte Rechenzeit auf einem Mainframe (Großrechner mit Mehrbenutzerbetrieb) von der jeweiligen Auslastung des Systems abhängt, und auch die Verwendung der tatsächlich verbrauchten CPU-Zeit das Problem der Rechner- und Implementationsabhängigkeit nicht löst.

Der allgemeine Ansatz der Komplexitätstheorie in solchen Fällen besteht darin, von der jeweiligen Implementation und dem verwendeten Rechner zu abstrahieren, indem elementare Rechenschritte gesucht werden, die universell in allen zu vergleichenden Verfahren vorkommen, so daß diese dann als elementare Maßeinheit verwendet werden können.

Im Falle des Lernens mit Back-Propagation, unter Verwendung einer explizit vorliegenden Menge von Patterns, haben sich in der Literatur zwei grundlegende Maßeinheiten eingebürgert:

- Das Anlegen eines der zu lernenden Patterns an das Netz wird als ein „*Zyklus*“ (*cycle*) bezeichnet.
- Ein Durchgang durch alle zu lernenden Patterns wird als eine „*Epoche*“ (*epoch*) bezeichnet.

Eine Epoche umfaßt also soviele Zyklen, wie die Menge der zu lernenden Patterns Patterns enthält.

Diese Definition ist jedoch ungenau und nicht immer praktikabel:

Es wird z.B. bei einem „Zyklus“ nicht zwischen dem Anlegen eines Patterns *mit anschliessender Modifikation* der Gewichte des Netzes in einem “On-Line”-Lernmodus und dem Anlegen eines Patterns und bloßem Aufsummieren der partiellen Ableitungen im Batch-Modus unterschieden.

Ein anderes Beispiel ist das “Dynamic Learning”, bei dem das Konzept der „Epoche“ nicht anwendbar ist.

Wie man aus den Struktogrammen der einzelnen Lernmodi (siehe dort) erkennen kann, gibt es andere, elementarere Konzepte, die in allen Lernmodi in gleicher Weise Verwendung finden und die sich daher als Basismaßeinheiten wesentlich besser eignen:

- Da ist zunächst die *Pattern-Fetch-Phase*, mit der das nächste zu lernende Pattern ausgesucht und angelegt wird. Dieses Konzept ist so allgemein, daß es sowohl auf "On-Line"-Lernverfahren, auf den Batch-Modus als auch auf das "Dynamic Learning" anwendbar ist. (Es ist sogar dann noch anwendbar, wenn die zu lernenden Patterns nicht als abgeschlossene Menge explizit vorliegen, sondern erst zur Lernzeit in Form einer Stichprobe eines in Echtzeit ablaufenden Prozesses gewonnen werden)

- Eine zweite, für Neuronale Netze elementare Phase ist die Berechnung der Ausgaben des Netzes ausgehend von den Eingaben; in hierarchischen Netzen wird diese Phase *Propagation-Phase* genannt. Diese Berechnung läuft stets und bei allen Verfahren in exakt derselben Weise ab, weshalb sich die Anzahl der benötigten Propagation-Phasen besonders gut als Vergleichsbasis verschiedener Lernverfahren eignet.

- Kennzeichnend für das Lernen mit Back-Propagation ist die *Back-Propagation-Phase*, in der die „Fehlersignale" von den Ausgabeunits aus rückwärts durch das Netz propagiert werden, mit deren Hilfe dabei die partiellen Ableitungen der Fehlerfunktion nach den Gewichten des Netzes berechnet werden. Je nach Lernmodus werden diese partiellen Ableitungen in (den Gewichtsmatrizen entsprechenden) Matrizen abgelegt oder aufsummiert.

- Anschließend wird in der "*Calculate Update*"-Phase eine der *Lernregeln* angewendet; sei es die Standard-Lernregel des steilsten Abstiegs mit Trägheitsterm, sei es eine der anderen Lernregeln, die in Kapitel 3 vorgestellt werden. Mit Hilfe der *Lernregel* wird aus den partiellen Ableitungen eine Anpassung oder Modifikation (*Update*) der Gewichte des Netzes berechnet, die dann (je nach Lernmodus) entweder zu den Gewichten des Netzes hinzuaddiert oder (im Falle des "Sum-of-Updates" Batch Learning) in separaten Matrizen aufsummiert werden.

- Die *Update-Phase*, in der die errechneten Modifikationen zu den Gewichten des Netzes hinzuaddiert werden, stellt nur im Batch-Modus eine eigenständige Phase dar, in den übrigen Lernmodi wird diese Phase in die "Calculate Update"-Phase mit einbezogen, da dies zusätzliche Schleifen und damit etwas Rechenzeit einspart. Aus diesem Grund eignet sich diese Phase nicht als Basiseinheit zur Komplexitätsmessung. Vereinfachend spreche ich im Folgenden von der kombinierten "Calculate-Update"- und "Update"-Phase als der "Update-Phase".

Wie aus den Struktogrammen der einzelnen Lernmodi ersichtlich ist, werden aufgrund der systematischen Unterschiede zwischen den Lernmodi (besonders zwischen den beiden Batch- und den übrigen Lernmodi) die oben genannten Phasen verschieden oft durchlaufen (es kommen aber alle Phasen in allen Lernmodi vor). Zusätzlich kann je nach Versuchslauf (abhängig von der zufälligen Initialisierung der Gewichte des Netzes sowie beim "Dynamic Learning" zusätzlich abhängig von der zufälligen Reihenfolge der präsentierten Patterns) die Anzahl der benötigten Durchläufe der einzelnen Phasen zum Teil erheblich variieren, bedingt durch die Schleife in einigen der Lernmodi, die das Training eines Patterns mehrmals hintereinander wiederholt, solange der lokale Fehler das "pattern error limit" nicht unterschreitet.

Aufgrund ihres universellen Charakters eignen sich diese Phasen zum Vergleich des Zeitbedarfs verschiedener Verfahren (unter denselben Startbedingungen), verschiedener Probleme (unter denselben Verfahren und Startbedingungen) oder verschiedener Lernversuche (desselben Problems).

Nachteilig ist dabei allerdings, daß diese Phasen keinen *eindeutigen* Vergleich erlauben, da die benötigten Anzahlen von Durchläufen der verschiedenen Phasen nicht miteinander gekoppelt sind. (Genaueres dazu siehe unten)

Zwar läßt sich in gewissem Umfang ein "Trade-Off" zwischen den verschiedenen Phasen beobachten (der Standard Batch-Modus benötigt beispielsweise nur sehr wenige Update-Schritte, dafür aber in der Regel mehr Propagation- und Back-Propagation-Schritte als andere Lernmodi), dennoch lassen sich diese Phasen nicht 1:1 gegeneinander aufrechnen, da sie sich weder in ihrer jeweiligen Ausführungsdauer noch in ihrer Wichtigkeit für den Lernfortschritt des Netzes entsprechen.

Zu beachten ist ferner, daß in meiner Implementation die Anzahl der Zyklen mit der Anzahl der Pattern-Fetches identifiziert wird. Dadurch wird auch das mehrfache Training ein und desselben Patterns mehrmals hintereinander, wie im Sequential, Periodical und Dynamic Learning, als ein einziger Zyklus gewertet.

Im Falle der „dynamischen“ Lernmodi wird eine Epoche als die Anzahl von Zyklen bzw. Pattern-Fetches definiert, nach der die Gewichte der Patterns neu berechnet werden (vergleiche Abschnitt 2.11.5).

Die hier angestellten Betrachtungen sind lediglich als ein Versuch anzusehen, geeignete Komplexitätsmaße zu finden sowie gängige Begriffe wie „Zyklus“ und „Epoche“ genauer zu definieren; weitere Forschungen und Vereinheitlichungen auf diesem Gebiet sind jedoch erforderlich.

Ob die Anzahl der Pattern-Fetches und Propagation-Phasen in der Berechnung des globalen Fehlers nach jeder Epoche mitgezählt wird oder nicht, ist beispielsweise eine Frage der Definition und hängt vom jeweiligen Untersucher ab.

Da die Berechnung des Gesamtfehlers für den Test des Abbruchkriteriums notwendig ist und dieser Test ein wichtiger Bestandteil des kompletten Verfahrens ist, sollten die dafür benötigten Pattern-Fetches und Propagation-Phasen meines Erachtens mitgezählt werden, auch wenn dieser Test für das Lernen des Netzes selbst nichts beiträgt.

Die Anzahlen der einzelnen Lernphasen in den verschiedenen Lernmodi hängen wie folgt zusammen (vergleiche dazu die Struktogramme):

Der übersichtlicheren Darstellung wegen seien zuvor die folgenden Abkürzungen vereinbart:

p	=	Anzahl der Patterns
E	=	Anzahl der Epochen
F	=	Anzahl der Pattern-Fetches
P	=	Anzahl der Propagation-Phasen
B	=	Anzahl der Back-Propagation-Phasen
U	=	Anzahl der (Calculate-) Update-Phasen

Sequential Learning:

$$\begin{array}{rcccl} & & F & = & E \cdot p \cdot 2 \\ E \cdot p \cdot 2 & \leq & P & \leq & E \cdot p \cdot (k+2) \\ 0 & \leq & B & \leq & E \cdot p \cdot k \\ & & U & = & B \end{array}$$

Dabei ist k die maximale Anzahl von Patternwiederholungen; diese kann beliebig groß (auch unendlich) gewählt werden.

Periodical Learning:

$$\begin{array}{rcccl} & & F & = & E \cdot p \cdot 2 \\ E \cdot p \cdot 2 & \leq & P & \leq & E \cdot p \cdot (r+2) \\ 0 & \leq & B & \leq & E \cdot p \cdot r \\ & & U & = & B \end{array}$$

Dabei ist r (= "repeat factor") die maximale Anzahl von Patternwiederholungen.

"Sum-of-Derivatives" Batch Learning:

$$\begin{array}{rcl} F & = & E \cdot p \cdot 2 \\ P & = & E \cdot p \cdot 2 \\ B & = & E \cdot p \\ U & = & E \end{array}$$

"Sum-of-Updates" Batch Learning:

$$\begin{array}{rcl} F & = & E \cdot p \cdot 2 \\ P & = & E \cdot p \cdot 2 \\ B & = & E \cdot p \\ U & = & E \cdot p \end{array}$$

Dynamic Learning:

$$\begin{array}{rcccl} & & F & = & E \cdot p \cdot 2 \\ E \cdot p \cdot 2 & \leq & P & \leq & E \cdot p \cdot (r+2) \\ 0 & \leq & B & \leq & E \cdot p \cdot r \\ & & U & = & B \end{array}$$

Dabei ist r (= "repeat factor") die maximale Anzahl von Patternwiederholungen. Falls die Anzahl der Zyklen, nach der alle Gewichte der Patterns neu berechnet werden, nicht mit p übereinstimmt, sind diese Formeln nicht korrekt.

Dynamic "Sum-of-Derivatives" Batch Learning:

$$\begin{array}{rcl} F & = & E \cdot p \cdot 2 \\ P & = & E \cdot p \cdot 2 \\ B & = & E \cdot p \\ U & = & E \end{array}$$

Dynamic "Sum-of-Updates" Batch Learning:

$$\begin{aligned} F &= E \cdot p \cdot 2 \\ P &= E \cdot p \cdot 2 \\ B &= E \cdot p \\ U &= E \cdot p \end{aligned}$$

Für die zwei Batch-Varianten von Dynamic Learning gilt ebenfalls, daß wenn die Anzahl der Zyklen, nach der alle Gewichte der Patterns neu berechnet werden, nicht mit p übereinstimmt, diese Formeln nicht korrekt sind.

2.12 Generalisierung und "Weight Decay"

Beim Lernen mit Neuronalen Netzen geht es im Allgemeinen nicht nur darum, eine vorgegebene Menge von Patterns zu speichern und mit den gewünschten Ausgaben zu verknüpfen (zu assoziieren), vielmehr soll das Netz dabei selbständig die der Abbildung zugrundeliegenden Regeln extrahieren und diese folgerichtig auf andere, nicht gelernte Beispiele anwenden können bzw. (wie im Beispiel der Erkennung handschriftlicher Ziffern) die diskriminierenden Merkmale entdecken, die auch bei leicht veränderten (oder verrauschten) Eingaben noch zu den gewünschten Ausgaben führen. Bei Klassifikationsproblemen soll das Netzwerk in der Lage sein, nicht gelernte Patterns richtig (oder sinnvoll) einzuordnen. (Vergleiche dazu auch den Abschnitt 2.4)

Diese Transferleistung wird allgemein mit dem Begriff „*Generalisierung*" bzw. „*Generalisierungsvermögen*" des Neuronalen Netzes bezeichnet.

Es gibt zahlreiche Versuche, diesen etwas vagen Begriff zu konkretisieren. Die allgemeinste und gleichzeitig präziseste Definition ist die von Denker et al. [4]:

Definition 2.2 *Sei* $\mathbf{I}$ *die Menge aller möglichen Eingabevektoren für ein gegebenes Neuronales Netz und* $\mathbf{O}$ *die Menge aller möglichen Ausgabevektoren des Netzes. Definiere daraus das „Universum"* $\mathbf{U} = \mathbf{I} \times \mathbf{O}$, *die Menge aller möglichen geordneten Paare* (*Eingabevektor*, *Ausgabevektor*) .

Alle möglichen Funktionen oder Relationen von $\mathbf{I}$ *nach* $\mathbf{O}$ *sind dann Teilmengen von* $\mathbf{U}$.

Sei weiter die Menge $\mathbf{R} \subset \mathbf{U}$ *die vom Netzwerk zu lernende Regel und* $\mathbf{P} \subset \mathbf{R}$ *die Menge der Patterns, mit denen das Netzwerk trainiert wird.*

Sei schließlich die Menge $\mathbf{X} \subset \mathbf{R}$ *eine von der Trainingsmenge* $\mathbf{P} \subset \mathbf{R}$ *disjunkte Testmenge, d.h. es sei* $\mathbf{X} \subseteq \mathbf{R} \setminus \mathbf{P}$.

Nachdem das Neuronale Netz die Regel $\mathbf{R}$ *anhand der Trainingsmenge* $\mathbf{P}$ *gelernt hat, wird die* Regelextraktion *der zugrundeliegenden Regel* $\mathbf{R}$ *durch das Netz mit Hilfe der Testmenge* $\mathbf{X}$ *überprüft.*

Der globale Fehler *auf der Testmenge* $\mathbf{X}$ *dient als Maß für die Güte der vom Netzwerk geleisteten Regelextraktion.*

Man beachte, daß sowohl die Trainingsmenge $\mathbf{P}$ als auch die Testmenge $\mathbf{X}$ für die zu lernende Regel $\mathbf{R}$ *repräsentativ* sein müssen, also zwar disjunkt sein müssen, aber keine *systematischen* Unterschiede aufweisen dürfen. In der Regel genügt es, die Elemente von $\mathbf{P}$ und von $\mathbf{X}$ durch denselben zufälligen Prozess aus $\mathbf{R}$ auszuwählen, um diese Bedingung zu erfüllen.

Generalisierung wird in Denker et al. [4] von der oben definierten *Regelextraktion* strikt unterschieden und wie folgt definiert:

Definition 2.3 *Seien die Mengen* $\mathbf{P}$ *und* $\mathbf{U}$ *wie oben definiert mit* $\mathbf{P} \subset \mathbf{U}$ *.*

Betrachte nun Mengen $\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3, \ldots \subset \mathbf{U}$ *.*

Eine Menge $\mathbf{G}_i$ *heißt* Generalisierung *der Menge* $\mathbf{P}$*, falls* $\mathbf{G}_i$ *eine echte Obermenge der Menge* $\mathbf{P}$ *ist, d.h. wenn gilt* $\mathbf{G}_i \supset \mathbf{P}$ *.*

Das bedeutet, daß die Relation $\mathbf{G}_i$ einen größeren Definitionsbereich besitzt als die Relation $\mathbf{P}$ und daß beide Relationen dort übereinstimmen, wo sich ihre jeweiligen Definitionsbereiche überdecken.

Man beachte, daß hier nicht von *der* Generalisierung die Rede ist, sondern daß es für jede Trainingsmenge $\mathbf{P}$ unzählige Generalisierungen gibt, die alle mit der Trainingsmenge vereinbar sind.

(Falls die Elemente von $\mathbf{I}$ und $\mathbf{O}$ binäre Vektoren sind und das Netz N Eingabebits und a Ausgabebits besitzt sowie die Trainingsmenge $\mathbf{P}$ m Paare von Ein- und Ausgabevektoren umfaßt, gibt es $(2^a)^{2^N - m}$ mögliche Generalisierungen)

Die *Regelextraktion* ist nach dieser Definition der Vergleich der vom Netzwerk erzeugten Generalisierung $\mathbf{G}$ mit einer ganz bestimmten, gewünschten Generalisierung $\mathbf{R}$ der Trainingsmenge $\mathbf{P}$.

In der Literatur wird zwischen Regelextraktion und Generalisierung meist nicht klar unterschieden; in der Regel wird aber aus dem Kontext klar, welches von beiden Konzepten gemeint ist.

Einfache Beispiele für die Regelextraktion und die Generalisierung sind die Interpolation und Extrapolation reellwertiger mathematischer Funktionen, die anhand einer Menge von Stützstellen gelernt wurden. (Beispiele zu dieser Art von Aufgabe siehe im Kapitel 4)

Man hat in der Praxis festgestellt, daß ein (hierarchisches) Neuronales Netz in der Regel umso schneller lernt, je mehr "hidden" Units ihm zur Verfügung stehen. Je mehr "hidden" Units jedoch vorhanden sind, desto leichter kann das Netz die Trainingsmenge $\mathbf{P}$ „auswendig lernen", anstatt die der Abbildung zugrundeliegenden Regeln zu extrahieren.

Das heißt mit anderen Worten, daß mit steigender Anzahl von "hidden" Units zwar die Konvergenzgeschwindigkeit zunimmt, dafür aber das Generalisierungsvermögen des Netzes abnimmt.

Es gibt im Wesentlichen zwei Ansätze, die aus diesem Dilemma herausführen.

Der erste Ansatz besteht darin, dem Netz zu Beginn der Trainingsphase genügend oder mehr als genügend "hidden" Units zur Verfügung zu stellen (dafür gibt es bisher keine exakten Regeln; man verfährt nach Daumenregeln und Erfahrung) und anschließend während des Trainings dem Netzwerk die „überflüssigen" "hidden" Units nach und nach zu entziehen.

Der zweite Ansatz besteht darin, die Größe der Gewichte des Netzes zu beschränken oder manche Verbindungen ganz stillzulegen, was ebenfalls indirekt das Generalisierungsvermögen des Netzes verbessert.

Für beide Ansätze gibt es zwei *Methoden*.

Die erste Methode beruht darauf, die bei der Berechnung der partiellen Ableitungen der Gewichte des Netzes verwendete *Fehlerfunktion* um einen sogenannten *Penalty*-Term (teilweise auch *Energy*- oder *Bias*-Term genannt) zu erweitern, der umso größer ist, je mehr "hidden" Units im Netzwerk aktiv sind oder je größer die Gewichte des Netzes sind (je nach Ansatz). Dieser Term bewirkt eine „Bestrafung" der Aktivität der "hidden" Units bzw. der Verbindungen des Netzes, was dazu führt, daß beim Training des Netzes nicht nur der Fehler des Netzes in den Ausgaben, sondern gleichzeitig auch die Anzahl der im Netz aktiven "hidden" Units bzw. Verbindungen minimiert wird. (Eine ausführliche Erörterung dieser Methode ist z.B. in Chauvin [3] und Hanson/Pratt [9] zu finden)

Für den ersten Ansatz (die Minimierung der Anzahl der aktiven "hidden" Units) lautet die Fehlerfunktion E:

$$E = \mu_{err} \cdot \sum_{p=1}^{n_{pat}} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 + \mu_{en} \cdot \sum_{p=1}^{n_{pat}} \sum_{j=1}^{n_{hid}} e (o_{pj}^2) \tag{2.61}$$

Die zweite Methode geht auf Mozer und Smolensky [19] zurück und beruht darauf, die „*Relevanz*" einer "hidden" Unit oder einer Verbindung für die Funktionalität des Netzwerkes zu bestimmen und ggfs. weniger relevante "hidden" Units oder Verbindungen aus dem Netzwerk zu entfernen. Unter geeigneten Simulationsbedingungen kann diese Methode auch eine geringfügige Konvergenzbeschleunigung bewirken aufgrund der Einsparung der Berechnungen, in denen die entfernten Units oder Verbindungen beteiligt waren.

Diese zweite Methode wird im folgenden Abschnitt kurz erläutert.

Eine Variante der ersten Methode ist das sogenannte "Weight Decay". Durch einen Penalty-Term, der aus der Summe der Quadrate aller Gewichte des Netzes besteht, werden alle Gewichte in jedem Update-Schritt zusätzlich um einen kleinen Bruchteil ihres aktuellen Wertes vermindert (dies ergibt sich durch die Ableitung des obigen Penalty-Terms). Nicht relevante Gewichte werden dadurch ständig kleiner, bis sie nahezu Null werden und damit die entsprechende Verbindung inaktiv wird. Außerdem wird verhindert, daß die Gewichte beliebig groß werden können.

Die Fehlerfunktion E für das "Weight Decay" hat die folgende Gestalt:

$$E = \mu_{err} \cdot \sum_{p=1}^{n_{pat}} \sum_{k=1}^{n_{out}} (t_{pk} - o_{pk})^2 + \mu_w \cdot \sum_{j=1}^{n_{unit}} \sum_{i=1}^{n_{unit}} w_{ji}^2 \tag{2.62}$$

Die Untersuchung dieser Methoden ist ein eigenes und umfangreiches Forschungsgebiet und daher nicht weiter Gegenstand dieser Arbeit.

2.13 "Skeletonizing" (Mozer/Smolensky)

Das hier vorgestellte Verfahren von Mozer und Smolensky [19] dient dazu, mit Hilfe des im Netzwerk gespeicherten Wissens die Funktionalität oder „*Relevanz*" seiner Komponenten zu bestimmen, sowohl um das Funktionsverhalten des gegebenen Netzes (und seiner Komponenten) genauer verstehen als auch um die Leistungsfähigkeit des Netzes verbessern zu können.

Das Verfahren besteht darin, das Netzwerk so lange zu trainieren, bis das Erfolgs- oder Abbruchkriterium erfüllt ist. Sodann wird die Relevanz der Komponenten des Netzes berechnet, die angibt, welche der Units und/oder Verbindungen für die Funktion des Netzes wesentlich sind. Die am wenigsten relevanten Units oder Verbindungen werden ggfs. „von Hand“ oder automatisch durch ein entsprechendes Programm entfernt. Anschließend wird das Netz erneut trainiert, das durch diese „Operation“ eine in der Regel unbedeutende Funktionseinbuße erleidet, bis das Erfolgskriterium wieder erfüllt ist.

Falls man ein minimales Netz erhalten möchte, besteht die Möglichkeit, dieses Verfahren zu iterieren.

Trainiert man das Netzwerk mit überschüssigen "hidden" Units, kann man die Lerngeschwindigkeit erhöhen. Durch anschließendes Entfernen der überflüssigen oder redundanten "hidden" Units wird das Netzwerk vereinfacht und sein Generalisierungsvermögen verbessert. Gleichzeitig erlaubt die Berechnung der Relevanz ein besseres Verständnis der Funktion der "hidden" Units des Netzes in Hinblick auf die der zu lernenden Abbildung zugrundeliegenden Regeln. Man hofft, das Netzwerk durch das Entfernen der überflüssigen "hidden" Units dazu zu bringen, die minimalen, wesentlichen Regeln zu finden und diese in den verbleibenden "hidden" Units festzuhalten.

Der Name des Verfahrens, "Skeletonization" oder kürzer "Skeletonizing", ergibt sich aus dem sukzessiven Entfernen von überschüssigen "hidden" Units aus dem Netz, dem „Fleisch“, bis nur noch das minimale Netz übrigbleibt, das „Gerippe“.

Im Prinzip kann jedoch mit Hilfe dieses Verfahrens die Relevanz sowohl von "hidden" als auch von Input-Units, als auch von Verbindungen innerhalb des Netzes berechnet werden.

Sind die Gewichte der Verbindungen, die von einer Unit ausgehen, relativ groß im Vergleich zu den Gewichten der Verbindungen, die von den übrigen Units derselben Schicht ausgehen, könnte man auf den Gedanken kommen, daß sowohl diese Verbindungen als auch die betreffende Unit besonders relevant für die Funktion des Netzes sein müssen.

Dies muß jedoch nicht der Fall sein, da sich die Wirkungen dieser Verbindungen gegenseitig aufheben können, oder es kann sein, daß die betreffende Unit trotz großer Aktivität keinen nennenswerten Einfluß auf die nachfolgenden Units hat, falls diese sich bereits im Sättigungsbereich befinden (aufgrund ihrer sigmoiden Aktivierungsfunktion), oder die Gewichte der Verbindungen, die von den Nachfolgeunits ausgehen, sind sehr klein, so daß diese kaum Einfluß auf höhere Schichten haben.

Demzufolge ist die Größe der Gewichte der Verbindungen des Netzes kein geeignetes Relevanzmaß.

Was in Wirklichkeit von Interesse ist, ist die Antwort auf die Frage, wie sich das Entfernen einer Unit oder einer Verbindung auf die Funktionalität des Netzes auswirkt, d.h., wie verändert sich der globale Fehler, berechnet über alle zu lernenden Patterns, falls eine bestimmte Unit oder Verbindung aus dem Netz herausgenommen wird?

Oder mit anderen Worten, wie groß ist der globale Fehler *mit* der Komponente (Unit oder Verbindung) i und wie groß ist er *ohne* diese Komponente?

Dadurch erhält man ein sehr einfaches Maß für die Relevanz ϱ_i einer Komponente i:

$$\varrho_i \quad = \quad E_{ohne\ Komponente\ i} \ - \ E_{mit\ Komponente\ i} \tag{2.63}$$

Man beachte, daß aufgrund dieser Definition auch *negative* Relevanzen existieren. Diese treten dann auf, wenn das Herausnehmen einer Komponente den globalen Fehler *verringert* anstatt erhöht. In der Praxis kommen negative Relevanzen jedoch nur bei noch untrainierten Netzen und direkt nach dem Herausnehmen einer Komponente vor.

Mozer und Smolensky [19] schlagen in ihrer Arbeit die Verwendung von Abschätzungen für die Berechnung der Relevanz der Units vor, mit der Begründung, daß die exakte Berechnung der Relevanz aller (interessierenden) Units den Zeitaufwand $\mathrm{O}\,(\,n\cdot p\,)$ erfordert, wenn n die Anzahl der Units des Netzes (ohne die Ausgabeunits) und p die Anzahl der zu lernenden Patterns ist, was bei wiederholten Berechnungen, zumal bei der automatischen Reduzierung der Netzwerkgröße durch die sukzessive Herausnahme von "hidden" Units, zu teuer sei.

Die Herleitung dieser Abschätzungen bzw. die Begründung für die letztendlich verwendete Formel (die aus einer exponentiellen Summe von Ableitungen besteht) ist jedoch lückenhaft und den Autoren zufolge eher durch den empirischen Erfolg gerechtfertigt. Außerdem ist auch die Berechnung dieser Abschätzungen nicht ohne Aufwand, der zudem von derselben Ordnung ist wie die exakte Formel (Gleichung (2.63) oben).

Da ein Vorschlag zur Automatisierung des Verfahrens in der Arbeit von Mozer/Smolensky [19] fehlt, habe ich mich darauf beschränkt, in meinem Programm die exakte Formel (Gleichung (2.63) oben) zu verwenden und nur das Entfernen von "hidden" Units „per Hand“ vorgesehen.

Das Herausnehmen von einzelnen Verbindungen ist zum Einen in meiner Implementation sehr schwierig zu realisieren, zum Anderen weniger effektiv als die Herausnahme von "hidden" Units, bei der sogar eine ganze Reihe von Verbindungen aufhören zu existieren. Die Berechnung der Relevanz von Verbindungen ist aus diesem Grunde nicht vorgesehen, wenngleich prinzipiell möglich.

Die Berechnung der Relevanz von Input-Units ist hingegen realisiert, obwohl wegen des Formats der Eingabedaten in den Eingabefiles die Entfernung von Input-Units nicht vorgesehen ist; dies ist die Aufgabe des Anwenders, der ausgehend von den ermittelten Relevanzen ggfs. Input-Units entfernt, indem er den Inhalt des Eingabefiles (das die Menge der zu lernenden Patterns enthält) entsprechend verändert und das Programm mit einer entsprechenden Anzahl von Input-Units neu startet.

Da das Verfahren nicht automatisiert ist, kann der Benutzer zu jedem beliebigen Zeitpunkt die Berechnung der Relevanzen der Units des Netzes durchführen, diese anzeigen lassen (auf einem Grafikterminal auch in Form eines Histogramms) und eine beliebige Anzahl von "hidden" Units entfernen.

Kapitel 3

Lernregeln

In diesem Kapitel werden die Lernregeln vorgestellt, die in meinem Programm (siehe Programmbeschreibung) implementiert sind.

Allen betrachteten Lernregeln ist gemeinsam, daß sie zur Änderung der Gewichte des Netzes einen Gradienten verwenden: Die partiellen Ableitungen eines geeignet gewählten Fehlermaßes, abgeleitet nach den Gewichten des Netzes. Sie sind daher eng mit dem ursprünglich von Rumelhart, Hinton und Williams ([24] Kapitel 8) entwickelten Verfahren des „Steilsten Abstiegs“ (“Steepest Descent”) verwandt, das dort „verallgemeinerte Delta-Regel“ genannt wird.

Die Verwendung des Gradienten ist jedoch keineswegs grundsätzlich notwendig, wie andere Verfahren zeigen, die ohne die Berechnung der partiellen Ableitungen auskommen, so z.B. der “Gradient-Free Learning Algorithm” von Birmiwal, Sarwal und Sinha [1], der auf dem (iterierten) Lösen von Gleichungssystemen beruht und außerdem nicht-differenzierbare Aktivierungsfunktionen erlaubt.

Da es jedoch unmöglich ist, in dieser Arbeit alle denkbaren oder interessanten Verfahren zu behandeln, wurden nur Gradientenverfahren untersucht.

Das bei der Berechnung der partiellen Ableitungen im Folgenden zugrundegelegte Fehlermaß ist das allgemein übliche (vergleiche Kapitel 2.8):

$$\begin{aligned} E_p &= \frac{1}{2} \sum_k (t_{pk} - o_{pk})^2 \\ \text{bzw.} \qquad & \\ E &= \frac{1}{2} \sum_p \sum_k (t_{pk} - o_{pk})^2 \end{aligned} \tag{3.1}$$

wobei p über alle zu lernenden Patterns und k über die Outputunits des Netzes variiert.

3.1 Nullstellensuche (Schmidhuber)

3.1.1 Motivierung des Verfahrens

Die Gradientenverfahren versuchen durch schrittweisen Abstieg (mehr oder weniger entlang der Richtung des Gradientenvektors) ein Minimum der Fehlerfläche zu finden — in der

Hoffnung, nicht in einem lokalen Minimum zu enden, sondern ein globales zu erreichen — oder doch zumindest ein lokales Minimum, das „gut genug“ ist.

Was aber eigentlich gesucht wird, sind Nullstellen der Fehlerfläche; d.h. solche Gewichte, die das Netz in die Lage versetzen, zu den gegebenen Eingabevektoren die gewünschten Ausgabevektoren zu produzieren.

Dabei ist Voraussetzung, daß überhaupt eine Lösung der Lernaufgabe existiert. Allgemeine hinreichende Bedingungen dafür zu finden, ob für eine gegebene Lernaufgabe eine Lösung existiert, ist ein bisher ungelöstes Problem (siehe dazu auch Rumelhart/McClelland [24], McClelland/Rumelhart [17] oder Abschnitt 3.6).

Unter der Annahme der Existenz einer Lösung macht es jedoch Sinn, statt möglicherweise nur lokaler Minima direkt die Nullstellen der Fehlerfläche zu suchen.

Die folgende Lernregel von Schmidhuber [26] versucht, diese Nullstellen durch ein Tangentenverfahren zu approximieren.

3.1.2 Herleitung des Verfahrens

Zur Herleitung der Regel soll zuerst der zweidimensionale Fall anhand eines einfachen Netzes mit nur zwei Input-Units, die über je eine (gewichtete) Kante mit einer Output-Unit verbunden sind, betrachtet werden. Demnach gibt es zwei Gewichte, w_1 und w_2.

Diese zwei Variablen spannen eine reelle Zahlenebene auf, über der sich die Fehlerfläche erhebt, die durch das verwendete Fehlermaß definiert ist.

Seien die zwei Gewichte zu Beginn willkürlich gewählt. Der dazugehörige Punkt auf der Fehlerfläche besitzt die Koordinaten $(\, w_1\,,\; w_2\,,\; E_p\,(w_1, w_2)\,)$.

Diejenige Fläche, die sich tangential an diesen Punkt der Fehlerfläche anschmiegt, besitzt die Darstellung:

$$\begin{pmatrix} w_1 \\ w_2 \\ E_p\,(w_1, w_2) \end{pmatrix} \quad + \quad \lambda_1 \cdot \begin{pmatrix} 1 \\ 0 \\ \frac{\partial E_p}{\partial w_1} \end{pmatrix} \quad + \quad \lambda_2 \cdot \begin{pmatrix} 0 \\ 1 \\ \frac{\partial E_p}{\partial w_2} \end{pmatrix} \tag{3.2}$$

Solange der Gradientenvektor $\left(\frac{\partial E_p}{\partial w_1}\,,\; \frac{\partial E_p}{\partial w_2} \right)$ nicht Null ist, ist diese Tangentialebene geneigt und schneidet die (w_1, w_2)-Ebene.

Gesucht ist nun ein Punkt in der (w_1, w_2)-Ebene, für den der zugehörige Funktionswert $E_p\,(w_1, w_2)$ Null wird. Diese Nullstelle kann approximiert werden durch einen Punkt auf der Geraden, die durch den aktuellen Punkt (w_1, w_2) sowie den Gradientenvektor als Richtungsvektor gegeben ist:

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \quad + \quad \eta \cdot \begin{pmatrix} \frac{\partial E_p}{\partial w_1} \\ \frac{\partial E_p}{\partial w_2} \end{pmatrix} \tag{3.3}$$

Um auf dieser Geraden einen Punkt festzulegen, wird sie mit der Tangentialebene (3.2) geschnitten:

$$\begin{pmatrix} w_1 \\ w_2 \\ E_p\,(w_1, w_2) \end{pmatrix} + \lambda_1 \cdot \begin{pmatrix} 1 \\ 0 \\ \frac{\partial E_p}{\partial w_1} \end{pmatrix} + \lambda_2 \cdot \begin{pmatrix} 0 \\ 1 \\ \frac{\partial E_p}{\partial w_2} \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ 0 \end{pmatrix} + \eta \cdot \begin{pmatrix} \frac{\partial E_p}{\partial w_1} \\ \frac{\partial E_p}{\partial w_2} \\ 0 \end{pmatrix} \tag{3.4}$$

Das ergibt das folgende Gleichungssystem:

$$\left[\begin{array}{ccccccccc} w_1 & + & \lambda_1 & & & = & w_1 & + & \eta \left(\frac{\partial E_p}{\partial w_1}\right) \\ w_2 & & & + & \lambda_2 & = & w_2 & + & \eta \left(\frac{\partial E_p}{\partial w_2}\right) \\ E_p(w_1, w_2) & + & \lambda_1 \left(\frac{\partial E_p}{\partial w_1}\right) & + & \lambda_2 \left(\frac{\partial E_p}{\partial w_2}\right) & = & 0 & & \end{array}\right] \tag{3.5}$$

Das ergibt sofort:

$$\lambda_1 = \eta \cdot \left(\frac{\partial E_p}{\partial w_1}\right)$$
$$\lambda_2 = \eta \cdot \left(\frac{\partial E_p}{\partial w_2}\right)$$

Durch Einsetzen in (3.5) erhält man:

$$E_p(w_1, w_2) + \eta \cdot \left(\frac{\partial E_p}{\partial w_1}\right)^2 + \eta \cdot \left(\frac{\partial E_p}{\partial w_2}\right)^2 = 0 \tag{3.6}$$

Durch Umformen und Auflösen nach η ergibt sich:

$$\eta = -\frac{E_p(w_1, w_2)}{\sum\limits_{i=1}^{2} \left(\frac{\partial E_p}{\partial w_i}\right)^2} \tag{3.7}$$

Der gesuchte Punkt errechnet sich dann durch Einsetzen dieses Wertes in die Gleichung der Geraden (3.3):

$$\begin{pmatrix} w_1{}' \\ w_2{}' \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \frac{E_p(w_1, w_2)}{\sum\limits_{i=1}^{2} \left(\frac{\partial E_p}{\partial w_i}\right)^2} \cdot \begin{pmatrix} \frac{\partial E_p}{\partial w_1} \\ \frac{\partial E_p}{\partial w_2} \end{pmatrix} \tag{3.8}$$

Die Herleitung für den n-dimensionalen Fall erfolgt analog; die Lernregel lautet:

$$\begin{array}{lrcl} & \vec{w}\,' & = & \vec{w} + \Delta\vec{w} \\ \text{mit} & & & \\ & \Delta\vec{w} & = & \eta \cdot \operatorname{grad} E_p \\ & \eta & = & -\frac{E_p(w_1, \ \ldots \ , w_n)}{\sum\limits_{i=1}^{n} \left(\frac{\partial E_p}{\partial w_i}\right)^2} \\ & \operatorname{grad} E_p & = & \left(\frac{\partial E_p}{\partial w_1}, \ \ldots \ , \frac{\partial E_p}{\partial w_n}\right) \end{array} \tag{3.9}$$

Der Faktor η ist kein von außen vorgegebener Parameter wie im Falle des "Steepest Descent"-Verfahrens, sondern er wird für jeden Lernschritt neu berechnet.

3.1.3 Einige Eigenschaften des Verfahrens

Der Vorteil der Nullstellensuche nach Schmidhuber besteht darin, daß im Gegensatz zu "Steepest Descent" und verwandten Verfahren keine (in der Regel problemabhängigen) Parameter von Hand optimiert werden müssen, bevor akzeptable Lernzeiten oder überhaupt Konvergenz erzielt werden können.

Wenn das Verfahren beim Iterieren der Regel (3.9) an eine Stelle gelangt, an der der Gradientenvektor nahezu Null wird, wächst η gegen unendlich. Dieses Verhalten ist erwünscht, da es erlaubt, durch einen großen, nahezu zufälligen Sprung aus einem lokalen Minimum zu entkommen oder aus dem flachen Bereich der Fehlerfläche. Damit diese Sprünge aber nicht zu groß werden, ist es sinnvoll, η zu begrenzen. Im Programm ist daher eine Obergrenze für η eingebaut. Der Autor des Verfahrens empfiehlt einen Maximalwert von 20.

Große Sprünge wirken sich nicht unbedingt negativ auf die übrigen zu lernenden Patterns aus, vorausgesetzt die Patterns interferieren nur wenig. Um dies zu erreichen, empfiehlt Schmidhuber eine „dünne“ Kodierung (sparse coding) der Eingaben des Netzes, so daß immer nur ein geringer Anteil aller Units des Netzes zur gleichen Zeit aktiv ist. Eine „dünne“ Kodierung kann zu nahezu oder ganz linear unabhängigen (oder orthogonalen) Eingabevektoren führen, was ganz allgemein zur besseren Lernbarkeit beiträgt. Mehr hierzu siehe Abschnitt 3.6.

Die Lernregel von Schmidhuber erfüllt die sogenannte „Lokalitätsbedingung“ nicht; die Bedingung, daß sämtliche Berechnungen zur Änderung der Gewichte nur lokal verfügbare Informationen (wie die partielle Ableitung) verwenden dürfen, jedoch keine Informationen, die sich erst aus der Gesamtheit aller Gewichte des Netzes ergeben (wie z.B. hier der Wert von η). (Mehr dazu siehe Kapitel 2 und Abschnitt 3.4.1)

Das Verfahren läßt sich trotzdem auch auf einem parallelen System implementieren, wenn man nach dem Vorwärtspropagieren der Aktivierungen und dem Rückwärtspropagieren der Fehler (d.h. der partiellen Ableitungen nach der Kettenregel) jede Unit die Quadrate der Ableitungen aller hereinkommenden Verbindungen aufsummieren und an die nächstfolgende Schicht weitergeben läßt. Eine einzelne Unit wird zusätzlich benötigt, um die Werte der Output-Schicht zu sammeln. Anschließend wird der ermittelte Wert von η an alle Units zurückpropagiert.

3.2 Automatische Wahl von η und α (Chan/Fallside)

3.2.1 Anmerkungen zur Wirkung von η und α

Das folgende Verfahren von Chan und Fallside [2] basiert auf der von Rumelhart, Hinton und Williams ([24] Kapitel 8) entwickelten Lernregel (dort „verallgemeinerte Delta-Regel“ genannt) für den steilsten Abstieg (steepest descent) auf der Fehlerfläche, die um den “Momentum”- bzw. „Trägheits“-Term erweitert wurde:

$$\vec{w}_{t+1} = \vec{w}_t + \Delta\vec{w}_t$$

mit

$$\Delta\vec{w}_t = -\,\eta \cdot \operatorname{grad} E_p + \alpha \cdot \Delta\vec{w}_{t-1}$$

$$(\ \Delta\vec{w}_0 = \vec{0}\ ) \tag{3.10}$$

$$\operatorname{grad} E_p = \left(\frac{\partial E_p}{\partial w_1}, \ \ldots\ , \frac{\partial E_p}{\partial w_n}\right)^t$$

und

$$\eta \in \mathrm{R}^+ \quad , \quad \alpha \in \mathrm{R}_0^+$$

Dieses Verfahren ist das Standardverfahren, das man üblicherweise unter dem Begriff “Back-Propagation” versteht (vergleiche Kapitel 2.8).

Ein großer Nachteil dieser Lernregel ist ihre Abhängigkeit von der „richtigen" Wahl der zwei Parameter, η und α, die zudem problemspezifisch ist: Bei ungünstiger Wahl kann sich das Lernen extrem verlangsamen oder das Verfahren lernt überhaupt nicht mehr. Selbst in günstigen Bereichen für beide Parameter können geringfügige Änderungen noch zu erheblichen Schwankungen in den benötigten Lernzeiten führen. Daher müssen diese Parameter von Hand, durch Versuch und Irrtum, optimiert werden.

Darüber hinaus kann es passieren, daß eine bestimmte Wahl von η und α nur am Anfang günstig ist, mit fortgeschrittenem Lernen aber wiederum andere Werte.

Daher gibt es die Strategie, das Lernen nach einer bestimmten Anzahl von Zyklen anzuhalten, die Parameter η und α neu zu setzen und anschließend das Lernen weiterlaufen zu lassen.

Das erhöht jedoch nur die Anzahl der zu optimierenden Parameter, da als weiterer Parameter die Anzahl der Zyklen, nach der η und α verändert werden, hinzukommt.

Das Optimieren der Parameter von Hand ist nicht nur lästig, sondern auch äußerst zeitaufwendig. Daher versucht das Verfahren von Chan und Fallside [2], die Werte von η und α automatisch und fortlaufend neu zu bestimmen.

Wenn die Lernrate η infinitesimal klein ist, folgt der Gewichtsvektor des Netzes, als Punkt im (Vektor-) Raum der Gewichte aufgefaßt, der Fehlerfläche entlang der Linie des steilsten Abstiegs (die durch den Gradientenvektor gegeben ist) hinunter. *(Modell einer masselosen (also trägheitslosen) Kugel, die die Fehlerfläche hinunterrollt)*

Dadurch werden z.B. auch Täler in der Fehlerfläche auf dem kürzesten Wege durchlaufen: Vom Rand des Tals hinunter zur Sohle und dann die (geneigte) Talsohle hinunter. Falls die Talsohle nicht abwärts geneigt ist, liegt ein lokales Minimum („Talkessel") vor.

Die Bewegung des Gewichtsvektors wird jedoch, der Größe der Lernrate entsprechend, ziemlich langsam sein, daher wird man einen größeren Wert für η wählen, um die Lerngeschwindigkeit zu erhöhen.

Allerdings ergeben sich einige schwerwiegende Probleme, sobald die Schrittweiten nicht mehr infinitesimal klein sind: Es können dann beispielsweise Oszillationen auftreten, die dadurch entstehen, daß der Gewichtsvektor zwischen zwei Talwänden in der Fehlerfläche hin- und herspringt. Das verlangsamt das Lernen und macht den Geschwindigkeitsvorteil durch die größere Lernrate wieder zunichte.

Um solche Oszillationen zu dämpfen, führt man zusätzlich den "Momentum"- oder „Trägheits"-Term mit dem Koeffizienten α ein. Er bewirkt, daß jeder Lernschritt die Resultierende aus der (Vektor-) Summe eines bestimmten Anteils des vorhergehenden Lernschritts und des augenblicklichen Gradientenvektors ist. Dadurch werden Richtungsänderungen, die der Gradientenvektor hervorruft, „geglättet". *(Modell einer massebehafteten und daher trägen Kugel, die die Fehlerfläche hinunterrollt)*

Außerdem bekommt die „rollende Kugel" durch den Trägheitsterm beim „Fallen" Bewegungsenergie; mit diesem „Schwung" ist sie in der Lage, kleine lokale Minima („Talkessel") zu überwinden.

Je kleiner die Lernrate η ist, desto weniger weicht das Verfahren von der Fallinie (die durch den Gradientenvektor gegeben ist) ab, und desto geringer werden die oben beschriebenen Probleme. Bei kleiner Lernrate ist auch die Notwendigkeit entsprechend gering, mit Hilfe des Trägheitsterms zu glätten.

Je größer η ist, desto notwendiger und desto wirksamer wird die Glättung durch den Trägheitsterm. Daher sollte man α um so größer wählen, je größer die Lernrate η ist. Allerdings darf man α nicht zu groß setzen, da sonst die Trägheit zu groß wird und das Verfahren nicht mehr auf die Änderungen des Gradientenvektors reagiert.

Ein weiteres Problem während des Lernens stellen flache „Plateaus" in der Fehlerfläche dar, da dort der Gradientenvektor gegen Null strebt, wodurch die Änderungen der Gewichte und damit auch die Lernfortschritte nahezu auf Null reduziert werden. Um diese flachen Stellen möglichst schnell zu überwinden und um den kleinen Gradientenvektor zu kompensieren, ist eine möglichst große Lernrate η zu wählen. Wenn man gleichzeitig auch den Trägheitskoeffizienten α möglichst groß wählt, erhöht dies den gewünschten positiven Effekt, vorausgesetzt die Fehlerfläche ist in der eingeschlagenen Richtung auch weiterhin flach.

3.2.2 Strategien zur Erkennung von Tälern und Plateaus

Eine geeignete Wahl von η und α hängt von der „Landschaft" der Fehlerfläche ab, die das Verfahren während des Lernens vorfindet. Wie zuvor erläutert, stellen dabei besonders Täler und Plateaus schwierige „Hindernisse" dar.

Um Täler und Plateaus zu erkennen, eignen sich die Winkel zwischen dem Gradientenvektor ($\operatorname{grad} E_p$), dem vorherigen ($\Delta\vec{w}_{t-1}$) und dem aktuellen (letzten) Änderungsschritt des Gewichtsvektors ($\Delta\vec{w}_t$), die bei einem Lernschritt des Standardverfahrens (siehe Gleichung (3.10) Abschnitt 3.2.1) auftreten:

Einmal ist der Winkel zwischen dem Gradientenvektor und dem letzten (vorhergehenden) Änderungsschritt von Interesse, gegeben durch:

$$\cos\theta \quad = \quad \frac{\operatorname{grad} E_p \cdot \Delta\vec{w}_{t-1}}{\|\operatorname{grad} E_p\| \cdot \|\Delta\vec{w}_{t-1}\|} \tag{3.11}$$

und zum anderen der Winkel zwischen aufeinanderfolgenden Änderungsschritten, gegeben durch:

$$\cos\phi \quad = \quad \frac{\Delta\vec{w}_t \cdot \Delta\vec{w}_{t-1}}{\|\Delta\vec{w}_t\| \cdot \|\Delta\vec{w}_{t-1}\|} \tag{3.12}$$

Diese Winkel sind in Abbildung 3.1 (Seite 81) veranschaulicht.

Dabei gibt der Winkel θ einen guten Aufschluß über den Verlauf der Fehlerfläche, während $\theta - \phi$ einen Anhaltspunkt über die vom Trägheitsterm hervorgerufene Abweichung von der Richtung des Gradientenvektors liefert.

Der Winkel ϕ selbst schließlich zeigt die „Laufruhe" beim Lernen an: Je weniger er von 0° abweicht, desto weniger „Kurven" in der Fehlerfläche muß das Verfahren folgen, desto „zielstrebiger" und desto schneller ist es. (Für eine ausführlichere Besprechung siehe [2])

Aus der Abbildung 3.2 (a) (Seite 82) kann man erkennen, daß es bei einem Winkel θ im Bereich um 180° wahrscheinlich ist, daß sich das Verfahren gerade in einem Tal der Fehlerfläche befindet und demzufolge der Wert von η reduziert werden sollte, um ein Hin- und Herpendeln zwischen den Talwänden zu verhindern.

Aus der Abbildung 3.2 (b) (Seite 82) kann man weiter entnehmen, daß das Verfahren sich wahrscheinlich auf einem Plateau befindet, wenn sich der Winkel θ im Bereich um 0° bzw. 360° bewegt. In diesem Fall sollte der Wert von η erhöht werden, um den kleinen Gradienten zu kompensieren und um die Ebene möglichst schnell wieder zu verlassen.

Abbildung 3.1: Winkel θ und ϕ in der Fehlerfläche (aus Chan/Fallside [2])

3.2.3 Adaption der Lernrate η

Eine Funktion, welche die beiden Bedingungen aus Abschnitt 3.2.2 erfüllt, ist die folgende:

$$\eta_t = \eta_{t-1} \cdot \left(1 + \tfrac{1}{2} \cos \theta\right) \tag{3.13}$$

(Siehe auch Abbildung 3.3 Seite 83)

Der Term $(1 + \frac{1}{2} \cos \theta)$ besitzt für einen Winkel θ von $180°$ ein Minimum von 0.5 und ein Maximum von 1.5 für einen Winkel θ von $0°$. Bei rechten Winkeln ($\theta = 90°, 270°$) nimmt dieser Term den Wert 1.0 an.

Durch Multiplikation dieses Terms mit der vorhergehenden Lernrate η_{t-1} wird die aktuelle Lernrate η_t verkleinert, wenn der Winkel θ im Bereich zwischen $90°$ und $270°$ liegt, d.h. wenn sich das Verfahren in einem Tal befindet. Liegt der Winkel θ im Bereich zwischen $270°$ und $360°$ bzw. zwischen $0°$ und $90°$, d.h. befindet sich das Verfahren auf einem Plateau, wird die Lernrate η_t erhöht.

Die Lernrate kann sich in einem Schritt maximal halbieren oder auf das anderthalbfache (nicht auf das doppelte!) anwachsen. Die damit verbundene Präferenz zur Verkleinerung der Lernrate soll die Stabilität des Verfahrens erhöhen.

Fortgesetzte Oszillationen mit Richtungsumkehr bewirken eine Verkleinerung der Lernrate, bis diese der Breite des Tals entsprechend angepaßt ist. Befindet sich das Verfahren während mehrerer aufeinanderfolgender Lernschritte auf einem Plateau, erhöht sich die Lernrate fortlaufend, um den kleinen Gradienten zu kompensieren und um das Plateau möglichst schnell zu verlassen.

3.2.4 Adaption des Trägheitsfaktors α

Die von den beiden Autoren des Verfahrens vorgeschlagene Funktion zur Anpassung des Trägheits- (Momentum-) Faktors α soll verhindern, daß der Trägheitsterm zu groß wird

Abbildung 3.2: Winkel in der Fehlerfläche (a): in einem Tal (b): auf einem Plateau (aus Chan/Fallside [2])

Abbildung 3.3: Adaptionsfunktion für die Lernrate η (aus Chan/Fallside [2])

und damit das Lernen behindert. Gleichzeitig soll der Wert von α im Wesentlichen dem von der Lernrate η gesetzten Trend folgen, wie bereits in den Vorbemerkungen erwähnt.

Eine Funktion, die diese Forderungen erfüllt, lautet:

$$
\begin{aligned}
\alpha_t &= \lambda_t \, \eta_t \\
\text{mit} \quad & \\
\lambda_t &= \lambda_0 \frac{\|\mathrm{grad}\, E_p\|}{\|\Delta \vec{w}_{t-1}\|} \\
\text{und} \quad & \\
0 \leq \lambda_0 &< 1
\end{aligned}
\tag{3.14}
$$

Wenn die Lernschrittweiten relativ (zur Fehlerfläche) klein sind, ändert sich nach einem Lernschritt am Gradienten nicht sehr viel. In diesem Fall liegt das Verhältnis $\|\mathrm{grad}\, E_p\| \; / \; \|\Delta \vec{w}_{t-1}\|$ nahe bei Eins, da es im Wesentlichen das Verhältnis zwischen den Längen aufeinanderfolgender Lernschritte ist. Der Wert von λ_t ist dann etwa gleich dem Wert von λ_0, der Grundeinstellung der Dämpfung.

Werden die Schrittweiten aufgrund zunehmender Länge des Gradientenvektors (zunehmender Steilheit des Geländes) größer, wächst dieses Verhältnis. Dies wird dazu benutzt, um die Dämpfung aus den Gründen, die in Abschnitt 3.2.1 erläutert sind, zu erhöhen. Werden die Schrittweiten hingegen kleiner, reicht ein niedrigeres Maß an Dämpfung aus: Das Verhältnis wird kleiner, und somit auch, wie erwünscht, die Dämpfung mittels λ_t.

3.2.5 Der Algorithmus

Das vollständige Verfahren wird durch die folgenden Beziehungen beschrieben:

$$\begin{aligned} \vec{w}_{t+1} &= \vec{w}_t + \Delta\vec{w}_t \\ \Delta\vec{w}_t &= \eta_t \cdot (-\operatorname{grad} E_p + \lambda_t \cdot \Delta\vec{w}_{t-1}) \\ (\quad \Delta\vec{w}_0 &= \vec{0} \quad) \\ \operatorname{grad} E_p &= \left(\tfrac{\partial E_p}{\partial w_1}, \ \ldots \ , \tfrac{\partial E_p}{\partial w_n}\right)^t \end{aligned} \tag{3.15}$$

mit den Definitionen für η_t und λ_t wie in den Abschnitten 3.2.3 und 3.2.4.

Obzwar wir bei diesem Vorgehen gegenüber dem Verfahren mit festen Koeffizienten die Wahl von η und α gegen die Wahl von η_0 und λ_0 eintauschen, ist dieses Verfahren dennoch von Vorteil, da nach den Simulationsergebnissen der Autoren der Lernerfolg von den letzteren beiden Koeffizienten weitestgehend unabhängig ist.

Der zusätzliche Aufwand für die Berechnung der Längen $\|\operatorname{grad} E_p\|$, $\|\Delta\vec{w}_{t-1}\|$ und des Produkts $\operatorname{grad} E_p \cdot \Delta\vec{w}_{t-1}$ ist gering. Der Speicherplatzbedarf dieses Verfahrens ist darüber hinaus von derselben Ordnung wie der des Standardverfahrens.

Da die Lernrate η_t bei diesem Verfahren exponentiell wachsen und schrumpfen kann, sind in meinem Programm zusätzlich je eine (einstellbare) Ober- und Unterschranke vorgesehen.

Dieses Verfahren genügt der Lokalitätsbedingung ("locality constraint") nicht (mehr darüber siehe Kapitel 2 und Abschnitt 3.4.1), da die Berechnung von η und α nicht lokal (von den Units des Netzes), sondern global (extern) erfolgt. Es läßt sich aber analog zum Verfahren von Schmidhuber (siehe Abschnitt 3.1.3) parallelisieren.

3.3 QuickProp (Fahlman)

3.3.1 Der QuickProp-Algorithmus

Zur Motivation seines Verfahrens schreibt Fahlman[1] [7] in seinem Artikel:

„Back-Propagation und seine Verwandten basieren auf der Berechnung der ersten partiellen Ableitungen des Gesamtfehlers bezüglich der einzelnen Gewichte. Mit Hilfe dieser Information können wir einen Gradientenabstieg im Gewichtsraum vornehmen. Falls wir infinitesimale Schritte entlang des Gradienten abwärts gehen, ist gesichert, daß wir schließlich ein lokales Minimum erreichen. Es hat sich durch die Erfahrung gezeigt, daß diese lokalen Minima bei vielen Problemen sogar global sind, oder zumindest für die meisten Anwendungen eine „genügend gute“ Lösung.

Wenn wir eine Lösung in der kürzest möglichen Zeit suchen, wollen wir natürlich keine infinitesimalen Schritte machen: Wir wollen größtmögliche Schritte ohne über das Ziel hinauszuschießen. Leider sagen uns eine Reihe von ersten partiellen Ableitungen an einem einzigen Punkt sehr wenig darüber, wie groß der Schritt im Gewichtsraum maximal sein

[1] Ich möchte dem Autor dieses Verfahrens, Scott E. Fahlman, an dieser Stelle für seine freundliche Unterstützung danken: Er sandte mir per "electronic mail" das Listing seines C-Programmes zu, in dem sein Verfahren implementiert ist, was dabei half, einige noch offene Fragen zu klären.

darf, den wir unbeschadet machen können. Wenn wir etwas über die höheren Ableitungen wüßten — die „Kurvigkeit“ der Fehlerfläche — dann könnten wir eine viel bessere Wahl treffen.

Mit Hilfe von zwei verschiedenen Ansätzen wurde bisher versucht, diesem Problem beizukommen: Beim ersten versucht man, die Lernrate dynamisch anzupassen, entweder global oder für jedes Gewicht separat, basierend auf Heuristiken, die sich an der Vorgeschichte während eines Lerndurchlaufs orientieren. (Vergleiche die folgenden beiden Kapitel)

Der beim Standard-Backpropagation-Verfahren verwendete Trägheitsterm stellt ein Beispiel für eine solche Strategie dar. (...)

Der andere Ansatz macht expliziten Gebrauch von den zweiten Ableitungen des Fehlers bezüglich der jeweiligen Gewichte. Mit Hilfe dieser Informationen können wir einen neuen Satz von Gewichten bestimmen, sei es unter Verwendung des Newton-Verfahrens oder einer noch verfeinerteren Optimierungstechnik. Leider erfordert die Bereitstellung der echten zweiten Ableitung eine sehr kostenintensive globale Berechnung, so daß man meist irgendeine Form von Approximation benutzt. (...)

Ich selbst habe einen Algorithmus entwickelt, den ich „QuickProp“ nenne und der etwas mit allen beiden Ansätzen zu tun hat. Es handelt sich um eine Methode zweiter Ordnung, die lose auf dem Newtonverfahren basiert, aber eher von heuristischer als formaler Natur ist. Alles läuft wie beim Standardverfahren für Back-Propagation, allerdings wird für jedes Gewicht eine Kopie von $\frac{\partial E}{\partial w}\,(\text{t}-1)$ (der ersten partiellen Ableitung des Fehlers im vorhergehenden Lernschritt) gespeichert, zusammen mit der Differenz zwischen dem aktuellen und dem vorherigen Wert dieses Gewichts, Δw. Der Wert von $\frac{\partial E}{\partial w}\,(\text{t})$ ist zu der Zeit, zu der die Gewichtsänderung berechnet werden soll, ebenfalls vorhanden.

QuickProp basiert auf zwei relativ stark vereinfachenden Annahmen: Erstens, daß die Fehlerkurve in Abhängigkeit von dem Wert des betrachteten Gewichts durch eine Parabel approximiert werden kann, die nach oben offen ist; zweitens, daß die Änderung der Steigung der Fehlerkurve für ein bestimmtes Gewicht nicht durch alle übrigen Gewichte beeinflußt wird, die zur selben Zeit verändert werden.

Für jedes Gewicht benutzen wir, von anderen Gewichten unabhängig, die vorhergehende und die aktuelle Steigung der Fehlerkurve sowie die Differenz zwischen aktuellem und vorigem Wert, an denen diese beiden Steigungen bestimmt wurden. Daraus wird eine Parabel berechnet, zu deren Minimum wir dann direkt springen. Diese Berechnung ist sehr einfach, und sie benutzt ausschließlich Informationen, die lokal an jedem Gewicht vorhanden sind:

$$\Delta w\,(\text{t}) \quad = \quad \frac{\frac{\partial E}{\partial w}(t)}{\frac{\partial E}{\partial w}(t-1) \;-\; \frac{\partial E}{\partial w}(t)} \cdot \Delta w\,(\text{t}-1) \tag{3.16}$$

Natürlich ist das nur eine sehr grobe Approximierung für den optimalen Wert des Gewichts, aber wenn man diese Methode iteriert, ist sie überraschend effektiv. Man beachte, daß der Parameter α hier verschwunden ist; wir werden jedoch weiterhin ein η benötigen.“

3.3.2 Herleitung des QuickProp-Verfahrens

Man geht von den Ableitungen an zwei Stützstellen $w\,(\text{t})$ und $w\,(\text{t}-1)$ aus und erhält somit zwei lineare Gleichungen. Die dritte Gleichung beschreibt die gesuchte dritte Stelle $w\,(\text{t}+1)$, an der die Ableitung des Fehlers Null sein soll. Zusammen mit der Differenz zwischen den zwei Stützstellen, Δw, erhält man das folgende Gleichungssystem:

$$\begin{array}{lrcl}
(1): & a \cdot w\,(\mathrm{t{-}1}) + b & = & \frac{\partial E}{\partial w}\,(\mathrm{t{-}1}) \\
(2): & a \cdot w\,(\mathrm{t}) + b & = & \frac{\partial E}{\partial w}\,(\mathrm{t}) \\
(3): & a \cdot w\,(\mathrm{t{+}1}) + b & = & 0 \\
(4): & w\,(\mathrm{t}) & = & w\,(\mathrm{t{-}1}) + \Delta w
\end{array}$$

Durch Umformen erhält man:

$$\begin{array}{lrcl}
(1)-(2): & a \cdot [\, w\,(\mathrm{t{-}1}) - w\,(\mathrm{t})\,] & = & \frac{\partial E}{\partial w}\,(\mathrm{t{-}1}) - \frac{\partial E}{\partial w}\,(\mathrm{t}) \\
(2)\text{'}: & b & = & \frac{\partial E}{\partial w}\,(\mathrm{t}) - a \cdot w\,(\mathrm{t}) \\
(3)\text{'}: & w\,(\mathrm{t{+}1}) & = & -\,\frac{b}{a} \\
(4)\text{'}: & w\,(\mathrm{t{-}1}) - w\,(\mathrm{t}) & = & -\,\Delta w
\end{array}$$

Durch Einsetzen von (4)' in (1) − (2) ergibt sich:

$$(5): \quad a \cdot (\,-\,\Delta w\,) = \frac{\partial E}{\partial w}\,(\mathrm{t{-}1}) - \frac{\partial E}{\partial w}\,(\mathrm{t})$$

Und durch Umformen:

$$(5)\text{'}: \quad a = \frac{\frac{\partial E}{\partial w}(t) - \frac{\partial E}{\partial w}(t-1)}{\Delta w}$$

Einsetzen von (2)' in (3)' ergibt:

$$(6): \quad w\,(\mathrm{t{+}1}) = -\,\frac{\frac{\partial E}{\partial w}(t)}{a} + w\,(\mathrm{t})$$

Durch Einsetzen von (5)' in (6) erhält man:

$$(6)\text{'}: \quad w\,(\mathrm{t{+}1}) = -\,\frac{\frac{\partial E}{\partial w}(t) \cdot \Delta w}{\frac{\partial E}{\partial w}(t) - \frac{\partial E}{\partial w}(t-1)} + w\,(\mathrm{t})$$

Und durch Umformung folgt schließlich die Lernregel:

$$w\,(\mathrm{t{+}1}) = w\,(\mathrm{t}) + \frac{\frac{\partial E}{\partial w}(t)}{\frac{\partial E}{\partial w}(t-1) - \frac{\partial E}{\partial w}(t)} \cdot \Delta w$$

q.e.d.

3.3.3 Implementierung des Verfahrens

Zur Implementierung seines Verfahrens stellt Fahlman [7] die folgenden Überlegungen an: „Drei verschiedene Fälle sind bei diesem Verfahren zu unterscheiden:

1. Wenn die aktuelle Ableitung vom Betrag her etwas kleiner ist als die vorherige (aber das gleiche Vorzeichen besitzt), so erfolgt die Gewichtsänderung in die gleiche Richtung wie im vorangegangenen Schritt. Die Größe des neuen Schritts hängt dann davon ab, wie stark sich die Steigung des Fehlers durch den letzten Schritt bereits verringert hat.

2. Falls die aktuelle und die vorherige Ableitung verschiedenes Vorzeichen haben, bedeutet dies, daß wir das Minimum bereits übersprungen haben und uns nun auf dem entgegengesetzten Ast der Parabel befinden. In diesem Fall wird uns der nächste Schritt irgendwohin zwischen den vorherigen und den jetzigen Wert des Gewichts versetzen.

3. Der dritte Fall tritt ein, wenn aktuelle und vorherige Ableitung das gleiche Vorzeichen besitzen, die aktuelle Ableitung aber betragsmäßig gleich oder größer als die vorhergehende ist. Würden wir der Formel einfach blind folgen, käme ein unendlich großer oder ein Schritt rückwärts zustande, die Fehlerkurve hinauf und zu einem lokalen Maximum.

Es gibt mehrere Möglichkeiten, diesen dritten Fall zu behandeln. Die Methode, die anscheinend am besten funktioniert, besteht darin, einen neuen Parameter ins Leben zu rufen, μ, den „maximalen Wachstumsfaktor": Keine Änderung eines Gewichtes darf betragsmäßig größer sein als μ multipliziert mit der vorhergehenden Änderung. Wenn der durch die QuickProp-Formel berechnete Schritt zu groß wird, unendlich ist oder die Fehlerkurve wieder hinaufgeht, benutzen wir statt dessen μ mal dem letzten Schritt als den neuen Schritt. Die Idee dabei ist, daß man es sich leisten kann, innerhalb gewisser Grenzen zu beschleunigen, wenn die Steigung, anstatt geringer zu werden, während des Abstiegs wächst. Da die Gewichte sich in ihrer Wirkung gegenseitig beeinflussen können, kommt es bei der gleichzeitigen Veränderung aller Gewichte zu Unwägbarkeiten, die es ratsam erscheinen lassen, nicht allzu weit zu extrapolieren und nur mäßig zu beschleunigen.

Experimente zeigen, daß sich das Netzwerk chaotisch verhält und nicht lernt, wenn μ zu groß gewählt wird. Der optimale Wert für μ hängt wahrscheinlich in gewissem Umfang vom Problem ab, dennoch funktionierte ein Wert von 1.75 sehr gut für einen weiten Bereich von "Encoder"-Problemen."

(Anmerkung: "Encoder"-Probleme sind die von Fahlman getesteten Benchmark-Probleme: Dabei muß das Netzwerk einen binären Eingabevektor der Länge n, der an einer Stelle eine Eins enthält und sonst nur Nullen, identisch auf die Ausgabe abbilden, darf aber nur eine sehr geringe Anzahl von "hidden" Units verwenden, meist $\log_2(n)$. Siehe dazu auch Abschnitt 3.8.1)

„Da QuickProp die Gewichte in Abhängigkeit vom vorhergehenden Lernschritt verändert, wird eine Methode benötigt, um QuickProp damit zu starten. Zusätzlich benötigen wir etwas, um den Lernprozeß erneut zu starten, falls ein Gewicht einmal einen Schritt der Länge Null gemacht hat und nun wieder eine von Null verschiedene Steigung hat, dadurch daß sich woanders im Netzwerk etwas verändert hat. Die naheliegende Lösung ist, einfach einen Schritt des normalen Gradientenabstiegs (mit Lernrate η) auszuführen.

Meine augenblickliche Version von QuickProp addiert immer einen Gradientenabstiegsterm η mal der aktuellen Steigung zu dem von der QuickProp-Formel berechneten Wert von Δw hinzu. Allerdings mit einer Ausnahme: Der Gradientenabstiegsterm wird nicht zu Δw hinzuaddiert, falls die aktuelle Steigung ungleich Null ist und das entgegengesetzte Vorzeichen der vorherigen Steigung besitzt. In dieser Situation (der zweite Fall von oben) genügt die quadratische Formel von QuickProp alleine, um das Minimum zu finden. Das Hinzuaddieren des Gradiententerms würde sonst zum erneuten Überspringen des Minimums führen und Oszillationen hervorrufen.

Eine letzte Verfeinerung wird noch benötigt: Für manche Probleme läßt QuickProp die Gewichte unverhältnismäßig stark wachsen. Das kann zu einem Laufzeitfehler und damit zum Programmabbruch durch Fließkommaüberlauf mitten während des Lernens führen. Um das zu starke Anwachsen der Gewichte zu verhindern, addieren wir einen kleinen "weight-decay"-Term zur Steigung für jedes einzelne Gewicht hinzu. Man muß dabei beachten, daß es nicht reicht, lediglich alle Gewichte periodisch um einen gewissen Bruchteil ihres aktuellen Wertes zu reduzieren: Der QuickProp-Algorithmus ist in der Lage, eine solche einfache Form von "weight-decay" mit Leichtigkeit zu überspielen. Wenn stattdessen der "weight-decay"-Term dazu benutzt wird, die berechnete Steigung zu reduzieren, bewirkt die QuickProp innewohnende „Beschleunigung" automatisch eine Reduzierung und Begrenzung der Größe der Gewichte. (...)"

3.3.4 Anmerkungen zur Implementierung

1. Wie wird die von Fahlman empfohlene Form von "weight-decay" implementiert?

   QuickProp ist als ein Verfahren konzipiert, das im Batchmodus arbeitet, d.h., für einen Lernschritt dieses Verfahrens werden zuvor alle zu lernenden Patterns durchlaufen. Dabei werden jeweils die partiellen Ableitungen für sämtliche Gewichte bestimmt. Die partiellen Ableitungen desselben Gewichts für die verschiedenen Patterns werden jeweils aufsummiert. Diese Summe ist dann die Steigung, die in die Berechnung der QuickProp-Formel eingeht. Zu jedem Gewicht gehört also eine solche Summe und damit die aktuelle Steigung für dieses Gewicht. Statt nun diese Summe vor dem Durchlauf durch alle Patterns mit dem Wert Null zu initialisieren, wird sie mit dem Wert des entsprechenden Gewichts, multipliziert mit einem kleinen negativen Faktor (beispielsweise -0.0001), vorbesetzt.

2. Der „maximale Wachstumsfaktor" μ wurde eingeführt, um unendlich große Schritte oder Schritte den Gradienten hinauf zu verhindern, falls die aktuelle und die vorherige Steigung das gleiche Vorzeichen besitzen, die neue Steigung aber vom Betrage her größer ist (das ist der dritte Fall aus Abschnitt 3.3.3).

   Statt aber die Beträge der aktuellen und der vorherigen Steigung direkt miteinander zu vergleichen, wird in der Implementierung von Fahlman der Wert der vorherigen Steigung zuvor mit einem Schrumpfungsfaktor σ multipliziert, der aus dem maximalen Wachstumsfaktor berechnet wird nach der Formel:

   $$\sigma = \frac{\mu}{1 + \mu} \tag{3.17}$$

   Diese Funktion geht durch den Nullpunkt und nähert sich asymptotisch der Eins.

   Wenn also der Betrag der aktuellen Steigung größer ist als der Betrag der vorherigen Steigung mal σ, dann lautet der nächste Lernschritt:

   $$\Delta w\,(t+1) = \mu \cdot \Delta w\,(t) \tag{3.18}$$

   Falls nicht, wird die QuickProp-Formel benutzt.

   Das hat zur Folge, daß einfach ein Schritt in die bisherige Richtung vorgenommen wird, auch wenn die aktuelle und die vorherige Steigung nur *annähernd* gleich sind. Das ist sinnvoll, weil die QuickProp-Formel keine gute Schätzung liefern kann, wenn sich die beiden Steigungen nur unwesentlich unterscheiden.

Der Vergleichbarkeit der Ergebnisse wegen habe ich das QuickProp-Verfahren genauso implementiert (wenn auch in PASCAL statt in C) wie das Original von Fahlman.

3.4 Delta-bar-delta-Regel (Jacobs)

3.4.1 Schwächen von Standard-Back-Propagation

Es gibt eine ganze Reihe von (iterativen) Optimierungsverfahren, die eine gegebene Funktion minimieren, welche von mehreren Parametern abhängt. Im Falle von konnektionistischen Modellen ist meist eine Fehlerfunktion, d.h. die Differenz zwischen Soll- und Ist-Ausgabe des Netzes zu minimieren. Diese Fehlerfunktion ist von den Gewichten der Verbindungen als den Parametern abhängig.

Alle diese Optimierungsverfahren lassen sich in zwei Klassen einteilen: Einmal in *lokale* Verfahren, die jeden Parameter allein basierend auf Informationen über diesen Parameter modifizieren, d.h. in solche Verfahren, die ausschließlich *lokale* Berechnungen ausführen; sowie zweitens in *globale* (nicht lokale) Verfahren, die auch andere Informationen zur Bestimmung der Parameter heranziehen. Dieses Unterscheidungskriterium ist unter dem Begriff „Lokalitätsbedingung“ (“locality constraint” bekannt (vergleiche Kapitel 2).

Obwohl globale Verfahren oft sehr gute Konvergenzeigenschaften besitzen, wird in konnektionistischen Modellen meist einem lokalen Verfahren (in der Regel Back-Propagation) der Vorzug gegeben.

Ein Anliegen der konnektionistischen Forschung ist gerade die Untersuchung der Möglichkeiten, die rein lokale Berechnungen in Neuronalen Netzen bieten. Das hat zwei Gründe: Einmal bleibt dadurch eine grundlegende Eigenschaft biologischer Netze in den Modellen erhalten, zum anderen geht man davon aus, daß lokale Verfahren auf parallelen Rechnerarchitekturen leichter zu implementieren sind.

Geometrisch gesprochen, legt die gewählte Fehlerfunktion eine Fehlerfläche über dem Parameterraum (dem Raum der Gewichte des Netzes) fest.

Back-Propagation ist im Wesentlichen eine Methode des steilsten Abstiegs: Bei jedem Iterationsschritt werden die Gewichte vergrößert oder verkleinert, je nachdem in welcher Richtung die Fehlerfläche am steilsten abnimmt. Diese Richtung wird durch den Gradientenvektor der Fehlerfunktion angezeigt. Die Größe der Änderung des Gewichts ist ein konstanter Bruchteil der Größe der zugehörigen Komponente des Gradientenvektors, d.h. der partiellen Ableitung der Fehlerfunktion nach diesem Gewicht. Dieser konstante Bruchteil wird durch den Faktor η bestimmt, „Lernrate“ oder „Schrittweite“ genannt.

Die Größe der partiellen Ableitung der Fehlerfunktion nach einem bestimmten Gewicht kann jedoch so beschaffen sein, daß eine Änderung dieses Gewichts um einen konstanten Bruchteil seiner Ableitung nur eine minimale Verringerung des Gesamtfehlers zur Folge hat. Das tritt in den folgenden beiden Fällen ein:

1. Wenn die Fehlerfläche entlang der Dimension eines Gewichts ziemlich flach ist, ist die entsprechende partielle Ableitung vom Betrage her sehr klein. Daher wird auch das Gewicht nur um einen sehr kleinen Betrag geändert, und viele Schritte sind nötig, um eine nennenswerte Verringerung des Gesamtfehlers herbeizuführen.

Wenn die Fehlerfläche entlang der Dimension eines Gewichts dagegen sehr steil ist, ist auch die zugehörige Ableitung (vom Betrage) sehr groß. Dadurch wird der Wert des Gewichts um einen großen Betrag verändert und überspringt möglicherweise das Minimum der Fehlerfläche entlang dieser Dimension.

2. Ein weiterer Grund für die langsame Konvergenz von "Steepest Descent"-Algorithmen liegt darin, daß der (negative) Gradientenvektor nicht notwendigerweise in die Richtung des Minimums der Fehlerfläche zeigt. Warum das so sein kann, zeigt die Abbildung 3.4 (Seite 90). Die Fehlerfläche ist mittels Konturlinien dargestellt, die Punkte gleichen Fehlers miteinander verbinden. Das Minimum der Fehlerfläche ist durch den dicken schwarzen Punkt in der Mitte der konzentrischen Ovale gekennzeichnet. Der aktuelle Wert der beiden Gewichte w_1 und w_2 ist durch den Punkt $w\,(\mathrm{t})$ gegeben. Da an diesem Punkt die Fehlerfläche entlang der w_2-Dimension steiler ist als entlang der w_1-Dimension, ist auch die partielle Ableitung von w_2 größer als die von w_1.

   Damit für alle Punkte des Gewichtsraums der Gradientenvektor stets genau in die Richtung des Minimums der Fehlerfläche zeigt, müssen alle Konturlinien (von Punkten gleichen Fehlers) exakt kreisförmig sein. Im Allgemeinen sind diese Konturlinien jedoch elliptisch.

Abbildung 3.4: Fehlerfläche über einem zweidimensionalen Gewichtsraum (aus Jacobs [13])

Selbst wenn wir annehmen, daß die Achsen dieser Ellipsen parallel zu den Achsen der Gewichte liegen[2], kann eine konstante Lernrate in einem Fall entlang der (längeren) Hauptachse der Ellipse zu vernünftigen Schrittgrößen führen, während die Schritte in Richtung der

[2] Täler, deren Achsen nicht parallel zu den Koordinatenachsen liegen, kommen nach Sutton [30] nur dann vor, wenn eine enge Korrelation zwischen der Wirkung von zwei (oder mehr) Gewichten auf die Ausgabe des Netzes besteht.

(kürzeren) Nebenachse viel zu groß sind. Dadurch springt das Verfahren in dieser Richtung andauernd hin- und her und der Gesamtfehler verringert sich nur äußerst langsam.

Ist dagegen die Lernrate für die Nebenachse der Ellipse passend gewählt, ergibt dies kleine Schritte in Richtung der Hauptachse. Das vermeidet zwar Oszillationen des Verfahrens in der Richtung der Nebenachse, verringert aber gleichfalls den Gesamtfehler in jedem Lernschritt nur geringfügig, da hier die Bewegung entlang der Hauptachse zum begrenzenden Faktor wird.

3.4.2 Heuristiken zur Verbesserung der Konvergenz

Jacobs [13] schlägt die folgenden vier Heuristiken zur Verbesserung der Konvergenz der "Steepest Descent"-Methode vor:

1. Jeder der Parameter der zu minimierenden Funktion sollte eine eigene Lernrate besitzen. Wie im vorigen Abschnitt erläutert wurde, ist eine Schrittweite, die für eine Dimension des Parameterraums geeignet ist, nicht unbedingt auch für andere Dimensionen geeignet.

2. Jede Lernrate sollte während der Dauer des Lernens variieren können. Wie oben erläutert wurde, besitzt die Fehlerfläche gewöhnlich an verschiedenen Stellen entlang einer einzelnen Dimension auch verschiedene Eigenschaften: Mal ist sie steiler, mal flacher, so daß einmal kleinere, ein anderes Mal größere Schrittweiten vorteilhafter sind.

3. Besitzt die Ableitung eines Gewichts in mehreren aufeinanderfolgenden Lernschritten das gleiche Vorzeichen, so ist meist die Fehlerfläche an dieser Stelle und in dieser Richtung sehr flach, so daß die Lernrate dieses Gewichts vergrößert werden sollte.

4. Wenn das Vorzeichen der Ableitung eines Gewichts während mehrerer aufeinanderfolgender Lernschritte alterniert, sollte die Schrittweite dieses Gewichts verkleinert werden, um Oszillationen zu verhindern.

Diese Heuristiken sind allerdings nicht immer erfolgreich: Man betrachte beispielsweise eine Fehlerfläche über einem zweidimensionalen Gewichtsraum, die ein Tal aufweist, das jeweils in einem Winkel von 45° zu beiden Koordinatenachsen (denen der beiden Gewichte) liegt. Am Rande dieses Tals ist die Fehlerfläche in beiden Achsenrichtungen sehr steil. Befindet sich nun der aktuelle Gewichtsvektor (d.h. der Punkt mit den Koordinaten, die durch die beiden Gewichte gegeben sind) innerhalb dieses Tals, wird durch die obigen Heuristiken die Lernrate beider Gewichte verringert. Um den Gesamtfehler so schnell wie möglich zu reduzieren, müßten jedoch beide Lernraten gemeinsam erhöht werden.

Das Versagen der Heuristiken unter diesen Umständen ist darauf zurückzuführen, daß sie der Beschränkung durch die eingangs erwähnte „Lokalitätsbedingung" (Abschnitt 3.4.1) unterliegen.

3.4.3 Entwicklung eines geeigneten Verfahrens

Der Trägheitsterm

Im Trägheitsterm des Standard-Back-Propagation-Verfahrens sind die Heuristiken aus dem vorigen Abschnitt (3.4.2) bereits teilweise verwirklicht.

Zum Zeitpunkt t wird jedes Gewicht $w\,(\mathrm{t})$ nach der folgenden Regel modifiziert:

$$\Delta w\,(\mathrm{t}) \quad = \quad -\,\eta \cdot \frac{\partial E}{\partial w}\,(\mathrm{t}) \quad + \quad \alpha \cdot \Delta w\,(\mathrm{t{-}1}) \tag{3.19}$$

Das ist äquivalent zu:

$$\Delta w\,(\mathrm{t}) \quad = \quad -\,\eta \cdot \sum_{i=0}^{t} \alpha^i \cdot \frac{\partial E}{\partial w}\,(\mathrm{t{-}i}) \tag{3.20}$$

Wenn aufeinanderfolgende Ableitungen eines Gewichts das gleiche Vorzeichen besitzen, wird die obige exponentiell gewichtete Summe vom Betrage sehr groß, und das entsprechende Gewicht wird um einen großen Betrag verändert.

Wechselt das Vorzeichen dagegen mehrmals hintereinander, wird diese Summe (vom Betrage) klein, und auch das Gewicht wird nur um einen kleinen Betrag verändert.

Der Effektivität des Trägheitsterms sind jedoch Grenzen gesetzt: Nimmt man vereinfachend an, daß sämtliche Ableitungen stets gleich Eins sind, konvergiert die obige Summe für t gegen unendlich gegen den Wert $\frac{1}{1\,-\,\alpha}$. Die maximale Veränderung eines Gewichts wird in diesem Fall durch den Wert $\frac{\eta}{1\,-\,\alpha}$ begrenzt.

Zweitens kann diese Summe das entgegengesetzte Vorzeichen der augenblicklichen Ableitung aufweisen. Dadurch kann der Trägheitsterm eine Veränderung eines Gewichts den Gradienten hinauf statt hinunter bewirken. Das kann vorteilhaft sein (z.B. falls ein lokales Minimum zu überwinden ist), muß es aber nicht.

Die Delta-delta-Regel

Die Delta-delta-Regel [13] besteht sowohl aus einer Regel für die Veränderung der Gewichte als auch aus einer Regel zur Veränderung der Lernraten, von denen jedes Gewicht eine eigene besitzt.

Die Regel für die Änderung der Gewichte wird beschrieben durch:

$$w\,(\mathrm{t{+}1}) \quad = \quad w\,(\mathrm{t}) \quad - \quad \eta\,(\mathrm{t{+}1}) \;\cdot\; \frac{\partial E}{\partial w}\,(\mathrm{t}) \tag{3.21}$$

Als Regel für die Veränderung der Lernraten nimmt Jacobs den „steilsten Abstieg" auf einer Fehlerfläche, die aus Gründen der vereinfachten Herleitung über einem Netzwerk von Units mit *linearer* Aktivierungsfunktion definiert wird (siehe [13]).

Für die so definierte Fehlerfunktion E' lauten die partiellen Ableitungen der Lernraten:

$$\frac{\partial E'}{\partial \eta}\,(\mathrm{t}) \quad = \quad -\,\frac{\partial E}{\partial w}\,(\mathrm{t}) \cdot \frac{\partial E}{\partial w}\,(\mathrm{t{-}1}) \tag{3.22}$$

Die Regel für die Veränderung der Lernraten in Form eines „steilsten Abstiegs" entlang dieses Gradienten lautet dann

$$\begin{aligned} \eta\,(\mathrm{t{+}1}) \quad &= \quad \eta\,(\mathrm{t}) \;+\; \Delta\eta\,(\mathrm{t}) \\ \Delta\eta\,(\mathrm{t}) \quad &= \quad \gamma \cdot \frac{\partial E}{\partial w}\,(\mathrm{t}) \cdot \frac{\partial E}{\partial w}\,(\mathrm{t{-}1}) \end{aligned} \tag{3.23}$$

Leider hat die Delta-delta-Regel nur einen beschränkten praktischen Wert:

Wenn die Fehlerfläche an einem gegebenen Punkt des Gewichtsraums entlang der Dimension eines Gewichts sehr flach ist, haben zwei aufeinanderfolgende Ableitungen dieses Gewichts das gleiche Vorzeichen und sind vom Betrag her relativ klein. Ihr Produkt ist dann eine sehr kleine positive Zahl. Um die in diesem Fall wünschenswerte, möglichst große Lernrate zu erhalten, muß der Wert der Konstanten γ möglichst groß gewählt werden.

Da das Vorzeichen der Ableitung jedoch mit großer Wahrscheinlichkeit (der flachen Fehlerfläche wegen) auch für einige weitere Lernschritte das gleiche bleibt, kann die Lernrate in diesem Fall unangemessen groß werden, was das Gewicht unter Umständen weit über das Minimum in dieser Dimension hinausschießen läßt.

Andererseits kann es auch zu Problemen führen, wenn die Fehlerfläche entlang einer Gewichts-Dimension sehr steil ist. Hier haben sehr häufig zwei aufeinanderfolgende Ableitungen entgegengesetzte Vorzeichen, ihr Betrag ist meist sehr groß. Ihr Produkt ist dann eine betragsmäßig sehr große, negative Zahl. Wenn die Konstante γ sehr groß ist, kann es passieren, daß die Lernrate vermindert wird, bis sie negativ wird. In diesem Fall wird das Gewicht den Gradienten aufwärts statt abwärts bewegt, der Gesamtfehler wird größer statt kleiner. Um dieses Problem zu vermeiden, müßte die Konstante γ möglichst klein gewählt werden.

Aus diesen Gründen ist die Delta-delta-Regel kein geeignetes Verfahren zur Verbesserung der Konvergenz von Back-Propagation.

3.4.4 Die Delta-bar-delta-Regel

Die Delta-bar-delta-Regel [13] besteht ebenfalls (wie die Delta-delta-Regel) sowohl aus einer Regel für die Veränderung der Gewichte, als auch aus einer Regel zur Veränderung der (individuellen) Lernraten (für jedes Gewicht eine).

Die Regel zur Modifizierung der Gewichte ist dieselbe wie die der Delta-delta-Regel:

$$w\,(\mathrm{t+1}) \quad = \quad w\,(\mathrm{t}) \quad - \quad \eta\,(\mathrm{t+1}) \cdot \frac{\partial E}{\partial w}\,(\mathrm{t}) \tag{3.24}$$

Die Delta-bar-delta-Regel zur Modifizierung der Lernraten lautet jedoch:

$$\begin{aligned} \eta\,(\mathrm{t+1}) &= \eta\,(\mathrm{t}) + \Delta\eta\,(\mathrm{t}) \\ \Delta\eta\,(\mathrm{t}) &= \begin{cases} \kappa & \text{falls } \bar{\delta}\,(\mathrm{t-1}) \cdot \delta\,(\mathrm{t}) > 0 \\ -\,\phi \cdot \eta\,(\mathrm{t}) & \text{falls } \bar{\delta}\,(\mathrm{t-1}) \cdot \delta\,(\mathrm{t}) < 0 \\ 0 & \text{sonst} \end{cases} \end{aligned} \tag{3.25}$$

wobei

$$\delta\,(\mathrm{t}) \quad = \quad \frac{\partial E}{\partial w}\,(\mathrm{t}) \tag{3.26}$$

und

$$\begin{aligned} \bar{\delta}\,(\mathrm{t}) &= (1-\theta) \cdot \delta\,(\mathrm{t}) + \theta \cdot \bar{\delta}\,(\mathrm{t-1}) \\ \bar{\delta}\,(-1) &= 0 \\ \text{mit} \quad \theta &\in [\,0\,,1\,] \end{aligned} \tag{3.27}$$

Die rekursive Definition von $\bar{\delta}\,(\mathrm{t})$ läßt sich auch als Summe schreiben:

$$\bar{\delta}\,(\mathrm{t}) \quad = \quad (1-\theta)\cdot\sum_{i=0}^{t}\theta^i\cdot\delta\,(\mathrm{t{-}i}) \tag{3.28}$$

Die vier Heuristiken aus Abschnitt 3.4.2 werden durch die Delta-bar-delta-Regel (3.25) wie folgt realisiert:

Wenn die aktuelle Ableitung eines Gewichts und die exponentiell gewichtete Summe der vorhergehenden Ableitungen desselben Gewichts das gleiche Vorzeichen besitzen, wird die Lernrate um den konstanten Wert κ erhöht.

Haben dagegen die aktuelle Ableitung dieses Gewichts und die exponentielle Summe seiner bisherigen Ableitungen verschiedenes Vorzeichen, wird die Lernrate um den Faktor ϕ verkleinert.

Die Delta-bar-delta-Regel erhöht also die Lernraten *linear*, verkleinert sie aber *exponentiell*. Das lineare Wachstum soll verhindern, daß die Lernraten zu schnell zu groß werden. Die exponentielle Schrumpfung soll einerseits verhindern, daß die Lernraten negativ werden. Andererseits soll sie dafür sorgen, daß die Lernraten schnell genug kleiner werden können, um Oszillationen durch Hin- und Herspringen zu verhindern.

3.4.5 Hinweise zur Anwendung der Delta-bar-delta-Regel

Lernmodus

Die Delta-bar-delta-Regel ist für die Verwendung im Batch-Modus ausgelegt, d.h., vor jeder Anwendung der Lernregel wird ein Durchgang durch alle zu lernenden Patterns ausgeführt und die jeweiligen partiellen Ableitungen werden aufsummiert. Diese Summe ist das $\delta\,(\mathrm{t})$ aus der Formel (3.26) in Abschnitt 3.4.4.

Verschiedene Patterns haben im Allgemeinen, einzeln betrachtet, unterschiedliche Optima für die Gewichte eines Netzes. Die jeweiligen partiellen Ableitungen der Gewichte unter den verschiedenen Patterns weichen daher in der Regel stark voneinander ab. Es kann häufig vorkommen, daß sie sogar unterschiedliche Vorzeichen besitzen. Führt man nach dem Anlegen eines jeden einzelnen Patterns einen Lernschritt mit Hilfe der Delta-bar-delta-Regel aus, werden deshalb die Lernraten unnötig oft, im ungünstigsten Fall bei jedem Lernschritt verringert. Im Extremfall lernt das Netz überhaupt nicht mehr, wenn dadurch die Lernraten zu (numerisch) Null reduziert werden.

Das Problem der entgegengesetzten Vorzeichen der partiellen Ableitungen von verschiedenen Patterns wird durch das Lernen im Batch-Modus jedoch nicht behoben. Vielmehr löschen sich im Batch-Modus die partiellen Ableitungen durch die Summation teilweise gegenseitig aus.

Parameter

Die individuellen Lernraten müssen vor dem Start des Verfahrens initialisiert werden. Sie werden dazu jeweils mit dem globalen Parameter η vorbesetzt.

Die Verwendung der Delta-bar-delta-Regel ist auch in Kombination mit dem Trägheitsterm (mit dem Faktor α) möglich.

Für diese beiden sowie die neu hinzugekommenen Parameter κ, ϕ und θ verwendet Jacobs [13] in seinen Benchmark-Tests die folgenden Werte:

$$\begin{array}{lllll} \eta & : & 0.8 & & \\ \alpha & : & 0.9 & & \\ \kappa & : & 0.035 & \text{bis} & 0.095 \\ \phi & : & 0.01 & \text{bis} & 0.333 \\ \theta & : & 0.7 & & \end{array}$$

3.5 Adaptives Verfahren (Silva/Almeida)

Das adaptive Verfahren von Silva und Almeida [29] ist eine Variation des Verfahrens von Jacobs [13] und beruht auf den gleichen Überlegungen wie jenes (Abschnitt 3.4.1).

Eine ausführliche Darstellung der mit Standard-Backpropagation verbundenen Probleme ist darüber hinaus in Sutton [30] zu finden.

Das Verfahren von Silva und Almeida besteht wie das Verfahren von Jacobs aus zwei Regeln: Einer zur Veränderung der Gewichte des Netzes und einer zur Veränderung der individuellen Lernraten (der jeweiligen Gewichte).

Die Regel zur Veränderung eines Gewichts w_i (i variiert über alle Gewichte des Netzes) zum Zeitpunkt t ist die folgende:

$$w_i\,(\mathrm{t+1}) \quad = \quad w_i\,(\mathrm{t}) \quad - \quad \eta_i\,(\mathrm{t}) \quad \cdot \quad \frac{\partial E}{\partial w_i}\,(\mathrm{t}) \tag{3.29}$$

Die Regel zur Veränderung der Lernraten lautet:

$$\eta_i\,(\mathrm{t}) \quad = \quad \begin{cases} u \cdot \eta_i\,(\mathrm{t}-1) & \text{falls} \quad \frac{\partial E}{\partial w_i}\,(\mathrm{t}) \cdot \frac{\partial E}{\partial w_i}\,(\mathrm{t}-1) \; > \; 0 \\ d \cdot \eta_i\,(\mathrm{t}-1) & \text{falls} \quad \frac{\partial E}{\partial w_i}\,(\mathrm{t}) \cdot \frac{\partial E}{\partial w_i}\,(\mathrm{t}-1) \; < \; 0 \\ \eta_i\,(\mathrm{t}-1) & \text{sonst} \end{cases} \tag{3.30}$$

Eine wesentliche Eigenschaft dieser Lernregel ist das exponentielle Wachstum der Lernraten. Dadurch können die Lernraten in wenigen Iterationen um mehrere Größenordnungen wachsen oder schrumpfen, was die Einstellung der günstigsten Werte unabhängig von den Ausgangswerten erlaubt.

Das hat den Vorteil, daß die Parameter des Verfahrens nicht mehr durch Versuch und Irrtum von Hand optimal eingestellt werden müssen, bevor das Netz erfolgreich lernt.

Ein Wert von u zwischen 1.1 und 1.3 ergab in vielen unterschiedlichen Simulationen der Autoren gute Ergebnisse. Der Parameter d sollte etwas kleiner als $1/u$ gewählt werden, was der Verkleinerung der Lernraten eine geringfügige Präferenz gibt und damit die Stabilität des Verfahrens erhöht. (Silva und Almeida empfehlen die Werte $u = 1.2$ und $d = 0.7$)

Da die Einstellung der Lernraten auf den optimalen Wert mehrere Iterationen erfordern kann, können Lernschritte in dieser Zeit die Gewichte den Gradienten hinauf verändern, falls die Lernraten zu groß sind. Um dies zu vermeiden, schlagen Silva und Almeida vor, nach jedem Lernschritt zu testen, ob sich der Gesamtfehler erhöht hat. Falls ja, wird der Lernschritt rückgängig gemacht, nicht jedoch die Änderung der Lernraten. (In meinem Programm ist dies als zusätzliche Option möglich)

Das Verfahren von Silva und Almeida ist wie die Delta-bar-delta-Regel für die Verwendung im Batch-Modus ausgelegt. Bei der Verwendung im sogenannten "On-line"-Modus (bei dem nach *jedem* angelegten Pattern ein Lernschritt durchgeführt wird) sind schlechtere Resultate zu erwarten. (Vergleiche Abschnitt 3.4.5)

3.6 Perceptron (Rosenblatt/Rumelhart)

3.6.1 Das „klassische" Perceptron

Das „Perceptron" geht ursprünglich auf Rosenblatt [23] zurück, der in seiner Arbeit versucht hat, seine an natürlichen Nervensystemen gewonnenen Erkenntnisse in ein formales Modell umzusetzen.

Sein Werk gehört inzwischen zu den „Klassikern" der konnektionistischen Literatur. Offenbar liegt es daran, daß heute mit dem Begriff „Perceptron" eine ganze Reihe zum Teil ganz unterschiedlicher Modelle bezeichnet wird.

Rosenblatt geht in seiner Arbeit von Nervenzellen aus, die nach dem *Alles-oder-Nichts-Gesetz* (vergleiche Kapitel 1) funktionieren: Überschreitet die Summe der ankommenden Erregungen einen Schwellwert θ, feuert die Nervenzelle, ansonsten bleibt sie inaktiv. Er geht weiter vereinfachend davon aus, daß die Stärke der Aktivierung einer Nervenzelle konstant und von der Reizintensität unabhängig ist.

In konnektionistischen Modellen (McClelland/Rumelhart [17]) nennt man diesen Typ von Units "linear threshold units", manchmal auch einfach "threshold units", was aber zu Verwechslungen mit den meist verwendeten Units mit *sigmoider* (nichtlinearer) Aktivierungsfunktion führt, die ebenfalls über einen "threshold" (auch "bias" genannt) verfügen.

Die "linear threshold units" („lineare Schwellwerteinheiten") besitzen die folgende einfache Aktivierungsfunktion: Überschreitet die Summe der eingehenden Signale einen Schwellwert θ, dann ist die Ausgabe dieser Unit gleich Eins, ansonsten ist sie Null.

In der Regel wird unter einem „Perceptron" ein Netzwerk verstanden, das aus einer Schicht von Inputunits besteht (deren Aktivierungen von außen gesetzt werden; oft nur auf Null oder Eins) sowie einer Ausgabeschicht, bestehend aus "linear threshold units", wobei jede Input- mit jeder Outputunit über eine gewichtete Kante verbunden ist. (Es gibt also keine "hidden" Units)

3.6.2 Lineare Modelle

In manchen Modellen (Rumelhart/McClelland [24], McClelland/Rumelhart [17]) verwendet man statt der "linear threshold units" in der Ausgabeschicht Einheiten mit *linearer* Aktivierungsfunktion. Die lineare Aktivierungsfunktion weist einer Unit einen Aktivierungswert zu, der nicht wie bei den "linear threshold units" entweder Null oder Eins ist, sondern einfach die Summe der eingehenden Erregungen.

Die mathematische Analyse dieses letzteren Modells ist besonders einfach, da die Abbildung der Eingabe- auf die Ausgabevektoren, die das Netzwerk leistet, einer Matrixmultiplikation (der Matrix der Gewichte des Netzes mit dem Eingabevektor) entspricht, also eine lineare Abbildung darstellt.

Netze dieser Art mit mehr als zwei Schichten sind außerdem stets äquivalent zu einem solchen mit nur zwei Schichten, da jede zusätzliche Schicht lediglich der Multiplikation mit einer weiteren Matrix entspricht, so daß ein Netz existiert, dessen Gewichte die Komponenten der resultierenden Produktmatrix sind.

Man beachte, daß auch das Hinzufügen eines „Schwellwertes“ (der lediglich von den eingehenden Erregungen subtrahiert wird) zu Units mit linearer Aktivierungsfunktion nichts an dieser Äquivalenz ändert. Das Hinzuaddieren (oder Subtrahieren) eines Vektors von „Schwellwerten“ entspricht einer affinen Verschiebung. Geht man in den Vektorraum mit einer um eins höheren Dimension über[3], erhält man wieder eine lineare Abbildung, die einer Matrixmultiplikation homomorph ist (entspricht).

3.6.3 Lernregeln und Eigenschaften von Perceptrons

Je nach der verwendeten Lernregel können sich die Eigenschaften des Perceptrons (bei ansonsten identischer Architektur und Aktivierungsfunktion) wesentlich unterscheiden.

Zwei Lernregeln sind für das Perceptron von Bedeutung:

- Die *Hebb'sche Regel* (Hebb [10], Rumelhart/McClelland [24], McClelland/Rumelhart [17], sowie Kapitel 2.7), die besagt, daß die Stärke derjenigen Verbindungen erhöht werden sollte, deren zugehörige beiden Einheiten gleichzeitig aktiv sind:

$$\Delta w_{ji} = \eta \cdot a_j \cdot a_i \tag{3.31}$$

- Die *Delta-Regel* (Rumelhart/McClelland [24], McClelland/Rumelhart [17], sowie Kapitel 2.7), die die *Differenz* zwischen Ist- und Soll-Wert der Ausgabe (also den Fehler) zu minimieren sucht:

$$\begin{aligned} \Delta w_{ji} &= \eta \cdot e_j \cdot a_i \\ e_j &= t_j - a_j \end{aligned} \tag{3.32}$$

(w_{ji} ist das Gewicht der Verbindung zwischen Eingabeunit i und Ausgabeunit j, a_i und a_j sind die Aktivierungswerte der entsprechenden Units, t_j ist der Soll- (“target”-) Wert der Ausgabeunit j und e_j ihr Fehler)

Interessanterweise gilt (siehe Rumelhart/McClelland [24]), daß für ein Perceptron (ohne “hidden” Units!) mit linearer Aktivierungsfunktion die Lösung eines gegebenen Lernproblems, d.h. die injektive Abbildung der Menge der Eingabevektoren auf die Menge der (gewünschten) Ausgabevektoren, genau dann existiert (und mit Hilfe der Lernregeln auch bestimmt werden kann), wenn die Menge der Eingabevektoren die folgenden Voraussetzungen erfüllt:

- Bei Verwendung der Hebb'schen Regel müssen die Eingabevektoren paarweise *orthogonal* sein

- Bei Verwendung der Delta-Regel müssen die Eingabevektoren paarweise *linear unabhängig* sein

[3] Durch Erweiterung aller Vektoren um eine erste Komponente konstant gleich Eins und Erweiterung der Abbildungsmatrix um den Schwellwert-Spaltenvektor als neuer erster Spalte und um eine Kopfzeile, die in der ersten Komponente eine Eins und sonst nur Nullen besitzt.

(Diese Modelle lassen sich daher auch umgekehrt dazu einsetzen, um die Orthogonalität bzw. lineare Unabhängigkeit einer Menge von Eingabevektoren zu untersuchen)

Für Netze, deren Units eine nichtlineare Aktivierungsfunktion verwenden (z.B. solche mit "linear threshold" oder sigmoider (logistischer) Aktivierungsfunktion) oder die "hidden" Units enthalten (also aus mehr als zwei Schichten bestehen), sind bis auf das im folgenden Abschnitt besprochene Kriterium der „linearen Separierbarkeit“ keine solchen einfachen Konvergenzbedingungen bekannt (Rumelhart/McClelland [24], McClelland/Rumelhart [17]).

3.6.4 Klassifikationen und lineare Separierbarkeit

Für zweischichtige Perceptrons (d.h. solche ohne "hidden" Units) mit "linear threshold"-Aktivierungsfunktion gibt es ein Konvergenzkriterium, das für Klassifikationsprobleme (eine häufige Anwendung Neuronaler Netze) von Interesse ist:

Definition 3.1 *Ist eine Menge von Vektoren* $\mathbf{M}$ *aus einem Vektorraum* $\mathbf{V}$ *in zwei disjunkte (nichtleere) Teilmengen („Klassen“ genannt) eingeteilt, so heißen diese zwei Klassen* linear separierbar *genau dann, wenn in diesem Vektorraum eine Hyperebene existiert, die die beiden Klassen vollständig voneinander trennt.*

Sowie

Definition 3.2 *Ist eine Menge von Vektoren* $\mathbf{M}$ *aus einem Vektorraum* $\mathbf{V}$ *in eine (endliche) Anzahl von disjunkten (nichtleeren) Teilmengen („Klassen“ genannt) eingeteilt, so heißt diese Einteilung* linear separierbar *genau dann, wenn sie sich durch (endliche) Vereinigung und Schnitt aus dualen Einteilungen nach* Definition 3.1 *zusammensetzen läßt.*

Eine beliebige Hyperebene in einem Vektorraum ist durch die Angabe eines Normalenvektors $\vec{n}$ und eines Verschiebevektors $\vec{v}$ eindeutig bestimmt.

Definition 3.1 ist daher (für endliche Vektorräume) äquivalent zu (ohne Beweis):

Definition 3.3 *Sei ein Vektorraum* $\mathbf{V}$ *mit Basis* $\mathbf{B}$ *gegeben. Ist eine Menge von Vektoren* $\mathbf{M}$ *aus dem Vektorraum* $\mathbf{V}$ *in zwei disjunkte (nichtleere) Teilmengen („Klassen“)* $\mathbf{X}$ *und* $\mathbf{Y}$ *eingeteilt, so heißen diese zwei Klassen* linear separierbar *genau dann, wenn ein Normalenvektor* $\vec{n} \in R^n$ *und ein Verschiebevektor* $\vec{v} \in R^n$ *existiert, so daß für alle Vertreter* $x \in \mathbf{X}$ *und* $y \in \mathbf{Y}$ *gilt:*

$$\begin{aligned} \vec{n} \cdot x_B &> \vec{n} \cdot \vec{v} \\ \vec{n} \cdot y_B &\leq \vec{n} \cdot \vec{v} \end{aligned} \tag{3.33}$$

(x_B und y_B sind die Koordinatenvektoren von x und y bezüglich der Basis $\mathbf{B}$)

Ein Perceptron mit einer "linear threshold"-Outputunit leistet genau diese Einteilung: Der Normalenvektor $\vec{n}$ entspricht den Gewichten der Verbindungen von den Inputunits zur Outputunit, die Zahl $\vec{n} \cdot \vec{v}$ entspricht dem Schwellenwert θ. Die Ausgabe der Unit ist Eins oder Null, je nachdem ob das Skalarprodukt des Eingabe- und des Normalenvektors größer oder kleiner als der Schwellenwert θ ist.

Ein Perceptron mit mehr als einer Ausgabeeinheit realisiert gleichzeitig mehrere (duale) Klassifikationen und damit eine linear separierbare Klassifikation nach Definition 3.2.

Es läßt sich daher mit Hilfe eines zweischichtigen Perceptrons mit "linear threshold units" feststellen, ob eine vorgegebene Klassifikation (vorgegeben durch die entsprechenden "target"-Werte) linear separierbar ist.

Um die Komplexität von Lernaufgaben ("Benchmarks") besser beurteilen zu können, habe ich ein derartiges Perceptron in meinem Programm implementiert, da für die in dieser Arbeit untersuchten mehrschichtigen (dreischichtigen) Netze mit sigmoider Aktivierungsfunktion bei nicht linear separierbaren Problemen längere Lernzeiten zu erwarten sind, während das Lernen von linear separierbaren Problemen durch diese Netze als zu einfach anzusehen ist.

3.7 Eigenes Verfahren

Ein von mir entwickeltes Verfahren ist die folgende einfache Variation des Verfahrens von Chan und Fallside [2] (Motivation und Darstellung dieses Verfahrens siehe Abschnitt 3.2):

Durch die von Chan und Fallside verwendete Funktion (3.13) zur Anpassung der Lernrate

$$\eta_t \quad = \quad \eta_{t-1} \cdot \left(1 + \tfrac{1}{2} \cos \theta\right) \tag{3.34}$$

kann diese theoretisch unbegrenzt wachsen oder aber zu (numerisch) Null werden, wodurch das Netz nicht mehr lernt.

In meiner Implementation des Verfahrens von Chan und Fallside habe ich bereits eine (einstellbare) Ober- und Unterschranke hinzugefügt; dennoch ist diese Lösung unbefriedigend:

Wenn sich das Verfahren einer radikalen Richtungsumkehr gegenüber sieht (z.B. in einem Tal), muß die Lernrate sofort verringert werden, und nicht erst nach einigen Iterationen, sonst springt das Verfahren noch mehrmals zwischen den Talwänden hin und her oder geht in eine völlig falsche Richtung.

Bei häufigen Richtungswechseln darf die Lernrate nicht immer kleiner werden, sonst lernt das Netz nicht weiter. Hier ist ein minimaler Wert für die Lernrate sinnvoll.

Zeigt der aktuelle Gradient in etwa die gleiche Richtung wie der vorhergehende Lernschritt (z.B. auf einem Plateau), ist eine möglichst sofortige Anpassung (Vergrößerung) der Lernrate ebenfalls von Vorteil.

Bleibt die Richtung des Gradienten in mehreren aufeinanderfolgenden Schritten in etwa gleich, sollte die Lernrate nicht immer weiter wachsen, da sonst beim ersten Schritt in eine andere Richtung die Lernrate nicht schnell genug verringert werden kann und das Verfahren über das Ziel hinausspringt.

Aus diesen Gründen sollte die Lernrate zum Zeitpunkt t nicht von der Lernrate zum Zeitpunkt $t-1$ abhängen, sondern für jeden Lernschritt neu berechnet werden. Zusätzlich muß je eine Unter- (a) und eine Oberschranke (b) eingeführt werden. Das Prinzip der Anpassung der Lernrate in Abhängigkeit vom Kosinus des Winkels (θ) zwischen aktuellem Gradienten und vorhergehendem Lernschritt soll jedoch erhalten bleiben.

Eine Formel, die dies erfüllt, lautet:

$$\eta_t \quad = \quad a \; + \; (b - a) \cdot \tfrac{1}{2} \cdot (1 + \cos \theta) \tag{3.35}$$

Diese Funktion normiert den Wert der Kosinus-Funktion (des besagten Winkels θ) auf den Bereich von Null bis Eins und transformiert diesen Wert anschließend auf das Intervall $[\, a\, ,\, b\,]$.

Die Funktion zur Anpassung des Trägheitsfaktors ist (bis auf eine Umbenennung von λ_0 in α_0) die gleiche wie die von Chan und Fallside verwendete (3.14):

$$\begin{aligned} \alpha_t &= \eta_t \cdot \alpha_0 \cdot \frac{\|\mathrm{grad}\, E_p\|}{\|\Delta \vec{w}_{t-1}\|} \\ \text{mit} \qquad 0 \leq &\ \alpha_0 < 1 \end{aligned} \tag{3.36}$$

Mit Hilfe der so modifizierten Lernregel gelang es, in einigen Testproblemen bis dahin unerreichte Verkürzungen der benötigten Lernzeiten zu erzielen. (Siehe Kapitel 5)

3.8 Modifikationen

Kleinere Modifikationen bestehender Verfahren zur Verbesserung der Konvergenzeigenschaften werden häufig in der Literatur vorgeschlagen.

Die wichtigsten Modifikationen, die auch in meinem Programm implementiert sind, werden hier vorgestellt.

3.8.1 Symmetrische Aktivierungsfunktion

Mehrere Autoren (Fahlman [7], Linden [15] u.a.) schlagen die Verwendung einer symmetrischen Aktivierungsfunktion statt der üblichen logistischen Funktion

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.37}$$

vor und begründen dies mit einem Geschwindigkeitszuwachs beim Lernen.

Die symmetrische Aktivierungsfunktion lautet im einfachsten Fall:

$$S(x) = \frac{1}{1 + e^{-x}} - \frac{1}{2} \tag{3.38}$$

Der Funktionswert dieser Funktion liegt im Bereich $[\, -\frac{1}{2}\, ,\, \frac{1}{2}\,]$.

Bei Normierung des Funktionswertes auf das Intervall $[\, -1\, ,\, 1\,]$ lautet sie:

$$S(x) = \frac{2}{1 + e^{-x}} - 1 \tag{3.39}$$

Die Normierung auf das Intervall $[\, -1\, ,\, 1\,]$ bietet den Vorteil, daß Testdaten (die in der Regel im Bereich $[\, 0\, ,\, 1\,]$ liegen) auch ohne Umkodierung weiterhin verwendet werden können. In einigen Fällen führte dies sogar zu kürzeren Lernzeiten als mit den auf den Bereich $[\, -1\, ,\, 1\,]$ transformierten Daten.

Außer im Falle der Encoder- und der Complement-Encoder-Probleme (Fahlman [7]) ist nicht klar, worauf der Geschwindigkeitsvorteil bei Verwendung der symmetrischen Aktivierungsfunktion beruht.

Bei Encoder-Problemen erhält das Netzwerk binäre Eingabevektoren, die nur an einer Stelle eine Eins aufweisen und sonst aus Nullen bestehen. Das Netzwerk soll diese Vektoren identisch auf die Ausgabe abbilden, wobei es in der Regel einen „Flaschenhals“ von sehr wenigen “hidden” Units überwinden muß, indem es sehr komprimierte Kodierungen der Eingabevektoren in den “hidden” Units vornimmt. Da immer nur eine einzige Eingabeunit aktiv ist, werden bei jedem Lernschritt auch nur diejenigen Gewichte des Netzes verändert, die von der betreffenden (aktiven) Eingabeunit bzw. von dadurch aktivierten “hidden” Units ausgehen.

Bei Complement-Encoder-Problemen sind lediglich alle Bits der Ein- und Ausgabevektoren des entsprechenden Encoder-Problems invertiert. Da genau diejenigen Gewichte, die von der Eingabeunit ausgehen, die das Pattern charakterisiert (nämlich derjenigen, die inaktiv ist), beim Lernen des betreffenden Patterns nicht verändert werden (sondern nur indirekt beim Lernen anderer Patterns), stellte Fahlman [7] in seinen Simulationen erwartungsgemäß längere Lernzeiten (mehr als doppelt so lang) fest.

Bei Verwendung der symmetrischen Aktivierungsfunktion (und Verwendung der Werte $+1$ und -1 zur Kodierung der booleschen Werte wahr und falsch) sind die Encoder- und die entsprechenden Complement-Encoder-Probleme äquivalent. Dementsprechend stellte Fahlman gleichlange Lernzeiten für beide Arten von Problemen fest. Interessanterweise waren diese Lernzeiten jedoch geringfügig länger als die der Encoder-Probleme unter Verwendung der asymmetrischen (logistischen) Aktivierungsfunktion, aber wesentlich kürzer als die der Complement-Encoder-Probleme, was den Vorteil der symmetrischen Aktivierungsfunktion in manchen Fällen nachdrücklich demonstriert.

Dennoch gilt: Die von einem Netz mit logistischer und einem Netz mit symmetrischer Aktivierungsfunktion berechenbaren (lernbaren) Funktionen unterscheiden sich prinzipiell nicht: Wie in Linden [15] gezeigt wird, läßt sich jedes Netz (bis auf die Ausgabeschicht) mit Aktivierungen in einem beliebigen Intervall durch eine einfache (lineare) Transformation der Gewichte in ein Netz mit Aktivierungen im Intervall $[\,0\,,\,1\,]$ (oder in einem anderen beliebigen Intervall) transformieren.

Bei der Verwendung der symmetrischen Aktivierungsfunktion ist zu beachten, daß sich ihre Ableitung von der der logistischen Funktion unterscheidet. Während die Ableitung der logistischen Funktion

$$f'(x) \quad = \quad f(x)\cdot(1-f(x)) \tag{3.40}$$

lautet (vergleiche Kapitel 2.8), lautet die Ableitung der symmetrischen Aktivierungsfunktion (im Intervall $[\,-1\,,\,1\,]$):

$$S'(x) \quad = \quad \tfrac{1}{2}\cdot\Big(1 \;-\; [\,S(x)\,]^2\Big) \tag{3.41}$$

Außerdem ist zu beachten, daß der Wertebereich von $f'(x)$ (für x im Intervall $[\,0\,,\,1\,]$) das Intervall $[\,0\,,\,\frac{1}{4}\,]$ ist, während die Werte von $S'(x)$ (für x im Intervall $[\,-1\,,\,1\,]$) im Bereich $[\,0\,,\,\frac{1}{2}\,]$ liegen. Dies kann unter Umständen zu zu großen Schrittweiten beim Lernen führen, was ggfs. durch Halbierung der Lernrate η ausgeglichen werden kann.

Unter Umständen kann es jedoch sinnvoller sein, statt $S'(x)$

$$\tfrac{1}{2}\cdot S'(x) \quad = \quad \tfrac{1}{4}\cdot\Big(1 \;-\; [\,S(x)\,]^2\Big) \tag{3.42}$$

zu verwenden, damit die Werte ebenfalls im Bereich$[\,0\,,\,\frac{1}{4}\,]$ liegen.

3.8.2 Vermeidung der „falschen“ Nullstellen der Ableitungen

Die Ableitung der logistischen (bzw. der symmetrischen) Aktivierungsfunktion ist eine quadratische Funktion (siehe auch voriger Abschnitt), die an den Intervallgrenzen ($[0, 1]$ für die logistische, $[-1, 1]$ für die symmetrische Aktivierungsfunktion) den Wert Null hat und in der Intervallmitte ihren Maximalwert erreicht ($\frac{1}{4}$ bzw. $\frac{1}{2}$).

Das führt dazu, daß die partiellen Ableitungen aller Gewichte derjenigen Verbindungen zu Null werden, die zu einer Unit führen, deren Aktivierung nahe oder gleich einer der beiden Intervallgrenzen ist. Und zwar selbst dann, wenn es sich um eine Ausgabeunit handelt, deren Aktivierungswert maximal von ihrem Sollwert entfernt ist.

Das führt nach Fahlman [7] dazu, daß manche der Units eines Netzes während des Lernens in einen ständig inaktiven Zustand übergehen, in dem sie unter Umständen für den Rest der Simulation gefangen bleiben; was desto häufiger auftritt, je mehr "hidden" Units verwendet werden.

Um dieses Problem zu beseitigen, schlägt Fahlman vor, einen kleinen konstanten Wert (z.B. 0.1), "offset" genannt, zum Wert der Ableitung der Aktivierungsfunktion hinzuzuaddieren. Die Ableitung hat an den Intervallgrenzen dann nicht mehr den Wert Null, sondern den des Offsets, und das Maximum erhöht sich entsprechend:

$$f'^{*}(x) = f(x) \cdot (1 - f(x)) + \text{offset} \tag{3.43}$$

bzw.

$$S'^{*}(x) = \tfrac{1}{2} \cdot \left(1 - [S(x)]^2\right) + \text{offset} \tag{3.44}$$

Nach Fahlman war diese Methode unter mehreren, die er ausprobiert hat, die effektivste. Bei einem seiner Encoder-Probleme verkürzte sich dadurch die Laufzeit um fast die Hälfte.

In meinem Programm ist dieser Offset als wählbarer Parameter implementiert; er heißt dort „bias error“.

3.8.3 ATANH-Fehlerfunktion

Um diejenigen Patterns verstärkt zu lernen, deren Fehler besonders groß sind, schlägt Fahlman [7] die Verwendung einer modifizierten Fehlerfunktion vor.

Bei der Berechnung des Fehlergradienten geht die Differenz zwischen Ist- und Sollwert einer jeden Ausgabeunit k als (linearer) Faktor Δ_k ein (vergleiche Kapitel 2.8):

$$\Delta_k = (t_k - o_k) \tag{3.45}$$

Dabei ist t_k der "target"- oder Sollwert, o_k der "output"- oder Istwert.

Um größere Fehler stärker zu berücksichtigen, schlägt Fahlman vor, diesen Faktor nichtlinear ansteigen zu lassen: Für sehr kleine Fehler soll dieser Faktor weiterhin linear sein. Nur wenn die Differenz zwischen "target"- und "output"-Wert gegen den maximal möglichen Wert geht (± 1 im Falle der logistischen, ± 2 im Falle der symmetrischen Aktivierungsfunktion), so soll der Faktor Δ gegen $\pm\infty$ gehen.

Eine Funktion, die diese gewünschten Eigenschaften hat, ist die Area-Tangens-Hyperbolicus-Funktion:

$$\operatorname{atanh}(x) = \tfrac{1}{2} \ln \frac{1 + x}{1 - x} \qquad \text{für} \qquad |x| < 1 \tag{3.46}$$

Um im Programm nicht mit unendlichen Werten umgehen zu müssen, wird der Wert der Funktion auf ± 17.0 gesetzt, falls der Betrag der Differenz zwischen "target"- und "output"-Wert den Wert 0.9999999 übersteigt.

Der neue Faktor Δ_k^* hat dann die Form:

$$\Delta_k^* = \begin{cases} +17.0 & \text{falls } (t_k - o_k) > +0.9999999 \\ -17.0 & \text{falls } (t_k - o_k) < -0.9999999 \\ 2 \cdot \operatorname{atanh}(t_k - o_k) & \text{sonst} \end{cases} \tag{3.47}$$

Fahlman berichtet, daß die Verwendung dieser modifizierten Fehlerfunktion in seinen Simulationen die benötigten Lernzeiten weiter verkürzt hat; bei dem 10–5–10-Encoder-Problem (10 input, 5 hidden, 10 output units) um etwa 25%.

Es bleibt noch anzumerken, daß obwohl Fahlman diese Modifikation „ATANH-Fehlerfunktion" nennt, die ATANH-Funktion selbst eigentlich nicht in der Fehlerfunktion, sondern in deren Ableitung vorkommt.

3.8.4 Vernachlässigen kleiner Fehler

Oft werden als Zielwerte ("target"-Werte) für die Ausgabeunits die binären Werte 0 und 1 (bzw. -1 und $+1$) vorgegeben.

Da die logistische (bzw. symmetrische) Aktivierungsfunktion diesen Werten nur asymptotisch zustrebt, werden unter Umständen die Gewichte des Netzes sehr groß bei dem Versuch, diese Zielwerte tatsächlich zu erreichen.

Um diesen in der Regel unnötigen Rechenaufwand zu vermeiden (sowie die möglichen Nachteile überproportionierter Gewichte), gibt es zwei Strategien:

- Die erste besteht darin, als Zielwerte nicht die Zahlen 0.0 und 1.0 vorzugeben, sondern 0.1 und 0.9 (bzw. -0.9 und $+0.9$ o.ä. für die symmetrische Aktivierungsfunktion).
- Die zweite besteht darin, den Faktor Δ_k (siehe Gleichung (3.45) im vorigen Abschnitt) auf Null zu setzen, falls sein Betrag kleiner als eine gewisse Schranke (z.B. 0.1) ist.

Die erste Strategie ist eine Frage der Kodierung der Ein- und Ausgabedaten eines zu lernenden Problems und wird hier nicht weiter verfolgt (ihre Verwendung in Kombination mit meinem Programm ist jedoch ohne weiteres möglich).

Die zweite Strategie betrachtet solche Ausgabewerte als korrekt, die im Bereich der gewählten Schranke um den Zielwert herum liegen, und sie verändert die Gewichte der zu dieser Ausgabeunit führenden Verbindungen nicht weiter. (!)

Man muß bei dieser Strategie jedoch darauf achten, daß bei Verwendung eines automatischen Lernabbruchkriteriums dieses nicht „strenger" eingestellt werden darf als die hier gewählte Schranke, da sonst die Simulation in eine unendliche Schleife gerät, sobald aufgrund dieser Strategie keine Gewichte mehr verändert werden, das Netz aber immer weiter trainiert wird in der Erwartung einer Verbesserung des Gesamtfehlers.

3.8.5 "Split Eta"-Technik

Eine weitere von Fahlman [7] verwendete Modifikation ist eine Aufteilung ("Splitting") der globalen Lernrate η jeweils auf die zu verändernden Gewichte aller Verbindungen, die in derselben Unit der nachfolgenden Schicht zusammenlaufen.

Die effektive Lernrate eines Gewichtes wird ermittelt durch die Division der (globalen) Lernrate η durch den "fan-in" der Unit, zu der die betrachtete Verbindung hinläuft. Der "fan-in" einer Unit ist die Anzahl der Verbindungen, die zu ihr hinführen. (Im Gegensatz zum "fan-out", der die Anzahl der Verbindungen bezeichnet, die von einer Unit ausgehen)

Fahlman begründet diese Modifikation damit, daß ohne sie einige seiner Testprobleme überhaupt nicht konvergierten.

Diese Modifikation ist in meinem Programm nur im QuickProp-Lernalgorithmus von Fahlman wirksam (falls sie vom Benutzer aktiviert wird).

Kapitel 4

Testdaten (Benchmarks)

Die Wahl geeigneter Testprobleme, an denen die verschiedenen Lernverfahren erprobt und verglichen werden können, ist schwierig.

Um die Ergebnisse des Trainings eines Neuronalen Netzes leichter beurteilen zu können, bedient man sich meist sehr einfacher, kleiner Probleme, deren Lösungen in der Regel bekannt sind; sei es, daß man die Aufgabe bereits mit Hilfe eines Netzwerks gelöst hat, sei es, daß man „von Hand“ ein Netzwerk konstruieren kann, das die Aufgabe löst. Kleine Probleme haben zudem den Vorteil, daß die durch das Netzwerk gefundene Lösung leichter nachvollziehbar ist (denn in der Mehrzahl der Fälle gibt es viele verschiedene mögliche Lösungen).

Die Wahl solcher kleinen, überschaubaren Testprobleme birgt jedoch eine Gefahr: Es kann nicht davon ausgegangen werden, daß die anhand dieser kleinen Probleme gewonnenen Ergebnisse und Einsichten auch auf wesentlich größere “Real-World”-Anwendungen übertragbar sind.

Dies liegt einmal an der Größe des verwendeten Netzwerks, denn es ist nicht auszuschließen, daß sehr große Netzwerke ein von den meist nur winzigen getesteten Netzen stark abweichendes Verhalten zeigen. (Eine Düne zeigt andere Eigenschaften als ein paar Sandkörner; das Ganze ist mehr als die Summe seiner Teile)

Problematisch ist neben der unterschiedlichen Größe der Netzwerke (Quantität) aber auch die Andersartigkeit der Aufgaben (Qualität). Die kleinen Testprobleme werden meist künstlich konstruiert und sind mit Hilfe eines konventionellen Rechners in der Regel viel effizienter lösbar. Die besondere Schwierigkeit der Testprobleme für das Netzwerk unterscheidet sich meist erheblich von den besonderen Schwierigkeiten von “Real World”-Anwendungen (wie z.B. das Lernen einer bestimmten booleschen Funktion und die Klassifikation (Diagnoseerstellung) von Patienten anhand von medizinischen Testprofilen).

Aus diesen Gründen sind „echte“ Anwendungen als Testprobleme sicherlich aussagekräftiger und den „künstlichen“ kleinen Benchmarks vorzuziehen.

Dem stellen sich jedoch zwei Schwierigkeiten entgegen: Oft sind für wirklich große Probleme und/oder umfangreiche Untersuchungen die Rechner, auf denen die Simulationen durchgeführt werden sollen, zu klein oder zu langsam.

Zum Zweiten sind sinnvolle neue “Real World”-Anwendungen gar nicht so häufig anzutreffen. Viele Anwendungen erfordern zudem eine spezielle Netzwerkarchitektur oder zusätzliche spezielle Hardware und sind daher nicht für Standard-Back-Propagation-Netze geeignet.

Für einen sinnvollen Vergleich unterschiedlicher Lernverfahren muß daher ein Kompromiß eingegangen werden zwischen möglichst großer Praxisnähe, Verfügbarkeit der erforderlichen Daten, der Problemgröße und der benötigten Rechenzeit.

4.1 Das XOR-Problem

Das „Exklusiv-Oder"- bzw. XOR-Problem ist der „Klassiker" unter den Testproblemen. Dies ist historisch bedingt.

Nachdem man mit Hilfe von einstufigen Perceptrons (vergleiche Kapitel 3.6) eine ganze Reihe von (aufgrund der damaligen, sehr begrenzten Rechnerkapazitäten relativ einfachen) Aufgaben gelöst hatte, glaubten die Untersucher euphorisch, das Ei des Kolumbus zur Computertechnischen Lösung von kognitiven Aufgaben wie z.B. das Erkennen von Gesichtern oder das Lesen handgeschriebener Schrift gefunden zu haben.

Die Arbeit von Minsky und Papert [18] machte 1969 diesem Traum ein Ende, indem sie nachwies, daß Perceptrons nicht in der Lage sind, selbst einfache boolesche Funktionen (wie das Exklusiv-Oder) zu lernen — wieviel weniger also komplexe kognitive Aufgaben, so schlossen die Autoren.

Die XOR-Funktion besitzt die folgende Funktionstabelle:

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

Diese Funktion hat den Wert Eins, falls entweder das Argument x_1 oder das Argument x_2 den Wert Eins hat, nicht jedoch, wenn beide Argumente den Wert Eins haben.

Von dieser Funktion läßt sich sehr leicht zeigen, daß sie nicht *linear separierbar* ist (vergleiche Kapitel 3.6, Beweis siehe u.a. in Linden [15] und Rumelhart/McClelland [24]): Die vier Argumentetupel $(0,0)$, $(0,1)$, $(1,0)$ und $(1,1)$ entsprechen den Ecken eines Quadrats in der Euklidischen Ebene mit diesen Koordinaten. Es läßt sich keine Gerade finden, die das Quadrat so zerteilt, daß die beiden (gegenüberliegenden) Ecken mit dem Funktionswert Null auf der einen, die beiden anderen Ecken (ebenfalls einander gegenüberliegend) mit dem Funktionswert Eins auf der anderen Seite dieser Geraden liegen.

Als Paradebeispiel einer Funktion, die von einem (einstufigen) Perceptron prinzipiell nicht gelernt werden kann, wird das XOR-Problem gerne als Test für mehrstufige Netzwerke verwendet.

Durch die Verwendung eines nicht linear separierbaren Testproblems möchte man sicherstellen, daß das Netzwerk nicht mit einer „trivialen" Aufgabe gefüttert wird, die auch ein wesentlich einfacheres Netzwerk, nämlich das Perceptron, lösen kann.

Es gibt fast keine Arbeit über verbesserte Lernverfahren, die nicht die Überlegenheit ihrer Methode über das Standard-Back-Propagation-Verfahren mit Hilfe von Simulationsergebnissen des XOR-Problems nachzuweisen versucht. Das liegt einmal an der historischen

Bedeutung dieses Problems, zum anderen daran, daß dazu nur ein minimaler Aufwand an Platz (Netzwerkgröße) und Rechenzeit erforderlich ist.

Fahlman [7] schreibt dazu jedoch: „Wenn das Ziel darin besteht, möglichst gute Lernalgorithmen für “Real World”-Anwendungen zur Pattern-Klassifikation zu entwickeln, sind das XOR- und das “Parity”-Problem (siehe Abschnitt 4.2) der falsche Test. Die Lösung von Klassifikationsaufgaben beruht wesentlich auf der Fähigkeit des Netzwerks, von den gelernten Patterns auf ähnliche Patterns aus dem gleichen Eingaberaum zu verallgemeinern. Gelegentlich muß das Netzwerk dabei zwischen zwei sehr ähnlichen Patterns eine scharfe Unterscheidung machen, aber dies ist eher die Ausnahme als die Regel. XOR und Parity haben dagegen den genau entgegengesetzten Charakter: Generalisierung im Eingaberaum wird sogar „bestraft“, da die nächsten Nachbarn eines Patterns die diesem Pattern genau entgegengesetzte Antwort produzieren müssen.

Andere beliebte Benchmark-Probleme wie das Zählen der Anzahl zusammenhängender Blöcke oder “clumps” (siehe auch Abschnitt 4.3) haben etwas von der gleichen Tendenz, jeden Algorithmus zu „bestrafen“, der im Eingaberaum generalisiert. Hier hat ebenfalls die Veränderung eines beliebigen Eingabebits fast immer eine große Veränderung in der Ausgabe zur Folge; sehr ähnliche Patterns mit geringem Hamming-Abstand führen zu vollkommen verschiedenen Ausgaben. Als Teil einer Serie von Testproblemen sind diese Aufgaben sicherlich sinnvoll, aber wenn sie ausschließlich verwendet werden, können sie dazu führen, daß Lernalgorithmen entwickelt werden, die schlecht generalisieren.“

4.2 Das Parity-Problem

Das “Parity”- oder Paritäts-Problem ist die Erweiterung der XOR-Funktion auf mehr als zwei Argumente.

Die Aufgabe wird meist so gestellt, daß die Ausgabe des Netzes Eins betragen soll, wenn die Anzahl der Eingabebits mit dem Wert Eins *ungerade* ist (“odd parity”). (Das umgekehrte Problem mit der Ausgabe gleich Eins bei einer *geraden* Anzahl von Eingabebits mit dem Wert Eins wird als “even parity” bezeichnet)

Aus der booleschen Algebra ist bekannt, daß sich das Paritäts-Problem mit n Argumenten mit Hilfe von $n-1$ XOR-Gattern lösen läßt, indem immer je zwei Argumente oder Ausgänge von XOR-Gattern kaskadenartig durch ein XOR-Gatter zusammengefaßt werden.

Zur Lösung dieser Aufgabe muß das Netzwerk eine relativ komplexe und sehr genau ausbalancierte Belegung seiner Gewichte entwickeln, da von der Hamming-Distanz her benachbarte Eingabetupel die entgegengesetzte Ausgabe produzieren müssen.

Es handelt sich hier um ein hochgradig nicht linear separierbares Problem, das aus diesem Grunde oft als Benchmark verwendet wird.

4.3 Das „zwei-oder-mehr-Blöcke“-Prädikat

Das “Two-or-more-clumps-predicate” oder „Zwei-oder-mehr-Blöcke-Prädikat“ ist ein Verwandter des Paritäts-Problems.

Die Aufgabe besteht darin, in einem String von Bits (einem Binärvektor) die Anzahl der zusammenhängenden Blocks von Einsen zu bestimmen. Das Prädikat ist erfüllt, falls die Anzahl dieser Blocks zwei oder mehr beträgt.

Einige Beispiele:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	f
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1
0	0	0	0	1	1	0	0	0
1	0	1	0	1	0	1	0	1
0	1	1	1	1	1	1	0	0
1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0

Es läßt sich sogar explizit ein Netzwerk angeben, das dieses Prädikat berechnet (vergleiche Denker et al [4]). Die Konstruktion geht wie folgt:

Sei n die Anzahl der Eingabebits. Die Eingabeunits des Netzwerkes seien von 1 bis n durchnumeriert. Zusätzlich denke man sich zwei weitere, imaginäre Eingabeunits mit den Nummern 0 und $n+1$, die konstant den Aktivierungswert Eins haben, die die Ränder des Eingabevektors markieren.

Seien außerdem $n+1$ "hidden" Units vorhanden mit den Nummern 0 bis n.

Seien schließlich b und p zwei Zahlen größer Null. (Je größer diese sind, desto näher kommen die Ausgaben der "hidden" Units bzw. der Ausgabeunit den Sättigungswerten Null und Eins)

Die Anzahl zusammenhängender Blocks von Einsen in einem Binärvektor ist gleich der Anzahl der linken oder rechten Seiten der einzelnen Blöcke. Die linke Seite eines Blocks ist durch zwei benachbarte Bits mit den Werten $(0,1)$ gekennzeichnet, die rechte durch zwei Bits mit den Werten $(1,0)$.

Belege die Gewichte von den Eingabeunits i zu den "hidden" Units j wie folgt:

$$w_{ji} \quad = \quad \begin{cases} -2b & \text{falls} \quad i \;=\; j \\ +\,2b & \text{falls} \quad i \;=\; j+1 \\ \;0 & \text{sonst} \end{cases} \tag{4.1}$$

Setze die Schwellenwerte der "hidden" Units außerdem einheitlich auf den Wert $-b$ fest.

Die Netzeingabe einer "hidden" Unit j kann dann die folgenden Werte annehmen:

$$\mathrm{net}_j \quad = \quad \begin{cases} -b & \text{falls} \quad (\mathrm{I}_j, \mathrm{I}_{j+1}) \;=\; (0,0) \\ +\,b & \text{falls} \quad (\mathrm{I}_j, \mathrm{I}_{j+1}) \;=\; (0,1) \\ -\,3b & \text{falls} \quad (\mathrm{I}_j, \mathrm{I}_{j+1}) \;=\; (1,0) \\ -\,b & \text{falls} \quad (\mathrm{I}_j, \mathrm{I}_{j+1}) \;=\; (1,1) \end{cases} \tag{4.2}$$

Dabei steht I_i für den Wert des Eingabevektors an der Stelle i.

Da die "hidden" Unit #0 nur die Netzeingaben $-3b$ und $-b$ erhalten kann (da die Inputunit #0 stets den Aktivierungswert Eins hat) und somit (für $b \to \infty$) konstant den Ausgabewert Null liefert, kann man sowohl die "hidden" Unit #0 als auch die Inputunit #0 weglassen. Die Inputunit $\#n+1$ kann ebenfalls weggelassen werden, da die Verbindung von dieser Unit (mit dem konstanten Ausgabewert Eins) zur "hidden" Unit $\#n$ (mit dem Gewicht $+2b$) nur einen konstanten Beitrag von $+2b$ zur Netzeingabe dieser "hidden" Unit leistet. Man erhöht dazu lediglich den Schwellenwert der "hidden" Unit $\#n$ um den Betrag $+2b$, was insgesamt einen Schwellenwert von $+b$ für diese Unit ergibt, und läßt die Eingabeunit $\#n+1$ samt Verbindung weg.

Man erhält also insgesamt ein Netzwerk mit n Eingabe- und n "hidden" Units.

Die Gewichte von den "hidden" Units zur Ausgabeunit erhalten sämtlich den Wert $+2p$.

Wenn c die Anzahl der zusammenhängenden Blöcke oder "clumps" von Einsen im Eingabevektor ist, erhält die Ausgabeunit angenähert (da die Ausgaben der "hidden" Units nur für $b \to \infty$ die Werte Null und Eins annehmen) den Wert $2p \cdot c$ als Eingabe von den "hidden" Units.

Die exakte Formel für die Netzeingabe der Ausgabeunit lautet:

$$\text{net} \quad = \quad 2p \cdot [\, c \cdot f(+b) + c \cdot f(-3b) + (n-2c) \cdot f(-b) \,] \quad + \quad \theta \tag{4.3}$$

(f ist die logistische Funktion)

Wenn der Eingabevektor C oder mehr zusammenhängende Blöcke von Einsen enthalten muß, um das Prädikat zu erfüllen, wird der Schwellenwert θ der Ausgabeunit auf den Wert

$$\begin{aligned} \theta &= -\,2p \cdot C \;+\; p \\ &= -\,(\,2 \cdot C \;-\; 1\,) \cdot p \end{aligned} \tag{4.4}$$

gesetzt.

Die Netzeingabe der Ausgabeunit beträgt somit angenähert

$$\text{net} \quad = \quad 2p \cdot (\, c \;-\; C\,) \;+\; p \tag{4.5}$$

Für endliche p liefert die Ausgabeunit somit angenähert den Wert Null, falls der Eingabevektor weniger als C, und angenähert den Wert Eins, falls er C oder mehr zusammenhängende Blöcke von Einsen enthält. (Nur für $p \to \infty$ werden die exakten Werte Null und Eins angenommen)

Man beachte, daß diese Konstruktion weder die einzige noch notwendigerweise die beste Lösung für dieses Problem darstellt.

Diese Lösung wird in Denker et al. [4] als „geometrische" Lösung bezeichnet bzw. auch als "human solution" (menschliche Lösung), da Menschen in der Regel diese Konstruktion wählen, wenn sie ein Netzwerk entwerfen sollen, das dieses Prädikat berechnet.

4.4 Das Encoder-Decoder-Problem

Bei diesem bereits in Kapitel 3 erwähnten Problem werden dem Netzwerk n verschiedene Patterns präsentiert, bei denen jeweils nur ein einziges Input-Bit „an" ist (den Wert 1.0

hat), während alle übrigen Input-Bits „aus" sind (den Wert 0.0 haben). Die Aufgabe besteht darin, diese Eingabepatterns identisch auf die Ausgabe abzubilden.

Da alle Informationen die "hidden" Units passieren müssen, muß das Netzwerk für jedes der n Eingabepatterns eine eindeutige Kodierung in den m "hidden" Units entwickeln und seine Gewichte so anpassen, daß sie die notwendigen Kodier- und Dekodier-Operationen durchführen.

Wenn ein Netzwerk $m = \log_2(n)$ "hidden" Units besitzt, spricht man von einem „engen" ("tight") Encoder. Z.B. sind der 8–3–8- und der 16–4–16-Encoder „eng" in diesem Sinne.

Falls die "hidden" Units nur zwei Ausgabewerte, Null und Eins, annehmen, muß das Netzwerk jedem Inputpattern eines der n möglichen Binärwörter über den "hidden" Units zuweisen, ohne auch nur eins davon zu verschenken.

In der Praxis, so Fahlman [7] in seinem Artikel, wird ein Back-Propagation-Netz oft drei oder mehr verschiedene analoge Werte in einigen der "hidden" Units verwenden, so daß „enge" Encoder-Netzwerke in Wirklichkeit nicht gezwungen sind, nach optimalen Binärkodes zu suchen.

Es ist sogar möglich, berichtet Fahlman, die Encoder-Decoder-Aufgabe mit einem „superengen" ("ultra-tight") 8–2–8-Netzwerk zu lösen; das Lernen dauert jedoch wesentlich länger als mit dem 8–3–8-Netzwerk.

In einer "Real World"-Klassifizierungsaufgabe wird man einem Netzwerk in der Regel genügend "hidden" Units zur Verfügung stellen, damit es die Aufgabe ohne Schwierigkeiten lösen kann. Zwar wird man nicht zu viele "hidden" Units vorsehen — das erlaubt es dem Netzwerk, die zu lernenden Patterns einfach nur abzuspeichern, anstatt die zugrundeliegenden Regeln zu extrahieren, was die Bedingung dafür ist, daß das Netzwerk später auch mit nicht gelernten Fällen sinnvoll umgehen kann — aber man möchte das Netzwerk auch nicht zwingen, eine Menge zusätzliche Zeit damit zu verbringen, eine optimale Repräsentation zu finden.

Das legt die Vermutung nahe, daß relative „lose" ("loose") Encoder-Probleme wie 10–5–10 oder 64–8–64 realistischere und informativere Benchmarks abgeben als „enge" oder „superenge" Encoder.

4.5 Das Multiplexer-Problem

Der *Multiplexer* ist ein elektronisches Bauteil, das über mehrere Dateneingänge, einige Selektionseingänge und einen Datenausgang verfügt. Je nach dem an den Selektionseingängen angelegten Binärwort wird einer der Dateneingänge ausgewählt und zum Datenausgang durchgeschaltet.

Bei einem idealen Multiplexer wird das Signal des ausgewählten Dateneingangs ohne Verzerrungen, Abschwächungen oder Übersprechen mit den anderen Eingängen an den Ausgang weitergeleitet. In der Praxis gelingt dies jedoch nie vollkommen.

Zur Simulation eines Multiplexers auf einem Neuronalen Netz werden dem Netz Patterns der Länge $n + 2^n$ präsentiert, wobei die ersten 2^n Komponenten den Dateneingängen entsprechen und die letzten n Komponenten den Selektionseingängen.

Der kleinste Multiplexer besitzt also drei Eingänge: Einen Selektions- und zwei Dateneingänge. Der nächstgrößere Multiplexer besitzt sechs Eingänge, der nächste elf, dann zwanzig, usw.

Während an die Selektionseingänge nur die diskreten Werte Null und Eins angelegt werden, können an die Dateneingänge im Prinzip nicht nur binäre (diskrete), sondern auch analoge Werte angelegt werden.

Es ist jedoch einfacher, das Netz lediglich mit den insgesamt 2^{n+2^n} möglichen verschiedenen binären Patterns zu trainieren.

Von Interesse ist dabei besonders, das Übertragungsverhalten des trainierten Netzwerks zu überprüfen; d.h. festzustellen, inwieweit Übersprechen zwischen den Dateneingängen stattfindet und wie sehr das ausgewählte Signal durch das Netz verzerrt wird. (Es handelt sich hierbei um eine Überprüfung der Generalisierungsleistung des Netzwerks, da es nur an den Randpunkten des Eingabebereichs (des Intervalls $[\,0\,,\,1\,]$) trainiert wurde und nun die Zwischenwerte interpolieren muß)

Eine Variante dieses diskreten Multiplexer-Problems besteht in der Hinzunahme eines oder mehrerer Zwischenwerte in die Menge der zu lernenden Patterns, was beispielsweise den Vergleich der Generalisierungsleistung des Netzwerks in Abhängigkeit von der Anzahl der Stützstellen ermöglicht.

Im einfachsten Fall nimmt man den Wert 0.5 hinzu. Die Menge der zu lernenden Patterns besteht dann aus allen Kombinationen der drei Werte 0.0 , 0.5 und 1.0 für die Dateneingänge kombiniert mit allen Binärworten der Länge n für die Selektionseingänge. Bei n Selektionseingängen gibt es 2^n Dateneingänge und somit $3^{(2^n)} \cdot 2^n$ mögliche Patterns; für $n = 2$ also $81 \cdot 4 = 324$ Patterns.

4.6 Das Mesh-Problem

Das "Mesh"-Problem ist eine graphische Klassifikationsaufgabe, deren Schwierigkeit darin besteht, daß Exemplare zweier Kategorien (Punkte zweier Punktmengen) willkürlich auf einer zweidimensionalen Ebene verteilt sind, so daß sie sich gegenseitig durchdringen (daher der Name dieses Problems).

Das Netzwerk soll dabei lernen, den beiden Kategorien die ausgewählten Punkte korrekt zuzuordnen. Je mehr sich die zwei Punktmengen gegenseitig durchdringen, desto komplexer wird das Problem, und desto mehr (lineare) Schnitte muß das Netzwerk kombinieren, um diese Aufgabe zu lösen, d.h. um die beiden Klassen zu separieren. Es handelt sich also um ein (außer in Trivialfällen) mehr oder weniger hochgradig nicht linear separierbares Problem.

Abbildung 4.1 (Seite 112) zeigt die hier verwendete Variante des Mesh-Problems, bestehend aus zwei dünnen, annähernd C-förmigen Punktmengen, bei denen jeweils einer der Arme der einen Menge in die Bucht der anderen Menge hineinragt. Diese Variante geht auf Minsky und Papert [18] zurück; hier wurde sie jedoch aus Hanson und Pratt [9] entnommen.

Die vier Eckpunkte der Bildfläche haben die Koordinaten $(0,0)$, $(0,1)$, $(1,0)$ und $(1,1)$. Die Koordinaten der Punkte der ersten Menge sind die Folgenden:

Abbildung 4.1: Das Mesh-Problem. Die Datenpunkte (links) und ihre Zuordnung (rechts).

A	
0.5	0.5
0.4	0.5
0.3	0.6
0.3	0.7
0.4	0.8
0.5	0.9
0.6	0.9

Und die Koordinaten der Punkte aus der zweiten Punktmenge lauten:

B	
0.3	0.2
0.4	0.3
0.5	0.3
0.6	0.4
0.7	0.5
0.6	0.6
0.5	0.6
0.4	0.6

In einer anderen Variante bilden die zwei Punktmengen zwei parallele Linien, die spiralförmig „aufgewickelt" sind. Dadurch wird jedes Stück Linie der einen Menge auf beiden Seiten von je einem Stück Linie der anderen Menge eingeschlossen. In dieser Variante ist die Separierung der beiden Punktmengen extrem nichtlinear. Da jedoch bereits das Mesh-Problem aus Abbildung 4.1 (Seite 112) erhebliche Rechenzeit benötigte (vergleiche Kapitel 5), wurde diese Variante nicht weiter untersucht.

4.7 Die Rotkäppchen-Simulation

Diese Anwendung stammt aus einem Public Domain-Programm zur Simulation von Back-Propagation-Netzen. Das Programm nennt sich BPSIM.C und wurde Juni 1987 von Josiah C. Hoskins in C geschrieben.

Das kleine mitgelieferte Beispiel ist ein einfaches *Pattern-Assoziations*-Problem (vergleiche Kapitel 2.7).

Das Netzwerk soll dabei Rotkäppchen simulieren, das anhand von gewissen Merkmalen der Personen, denen es begegnet, entscheiden muß, wie es reagieren soll.

Die Menge der dem Rotkäppchen zur Verfügung stehenden Erkennungsmerkmale und die in diesem Märchen vorkommenden Personen sind die Folgenden:

	Big Ears	Big Eyes	Big Teeth	Kindly	Wrinkled	Handsome
Wolf	*	*	*			
Grandma		*		*	*	
Woodcutter	*			*		*

Die möglichen Reaktionen von Rotkäppchen auf diese Personen sind:

	Scream	Run Away	Look for Woodcutter	Approach	Kiss on Cheek	Offer Food	Flirt with
Wolf	*	*	*				
Grandma				*	*	*	
Woodcutter				*		*	*

Das Netzwerk benötigt demnach (bei binärer Kodierung) 6 Inputunits zur Darstellung der Merkmale der vorkommenden Personen und 7 Outputunits zur Darstellung der möglichen Reaktionen des Rotkäppchens.

Da die Trainingsmenge nur die drei Personen, d.h. drei Paare von Patterns umfaßt, ist es nicht sinnvoll, dem Netzwerk mehr als 3 "hidden" Units zur Verfügung zu stellen, denn mit diesen ist es bereits in der Lage, jeder Person eine "hidden" Unit zuzuordnen und somit, die gewünschte Zuordnung zu leisten.

4.8 Approximation reellwertiger Funktionen

Wie bereits in Kapitel 2.4 erwähnt, stellt ein zweistufiges hierarchisches Netzwerk, dessen Units eine nichtlineare Aktivierungsfunktion besitzen, einen universellen Approximator dar, der jede stetige Funktion beliebig genau approximieren kann.

Die einfachste Möglichkeit, diese Eigenschaft zu untersuchen und etwas über das Generalisierungsvermögen (hier in Form von Interpolation) von Netzwerken zu erfahren, besteht darin, einem Netzwerk eine Reihe von reellen, der größeren Anschaulichkeit wegen hier nur eindimensionalen Funktionen beizubringen.

Die Funktionen wurden auf dem Intervall $[0,1]$ an 201 äquidistanten Stützstellen (mit dem Abstand $1/200$; einschließlich den Intervallgrenzen) ausgewertet, die die Menge der zu lernenden Patterns bildeten.

Lineare Funktionen (Gerade):

Die hier untersuchten zwei linearen Funktionen lauten:

a) : $y = 0.9 \cdot x + 0.1 \qquad x \in [0, 1]$

b) : $y = -1.0 \cdot x + 1.0 \qquad x \in [0, 1]$

Zur Anschauung hier noch einige der markantesten Punkte dieser beiden Funktionen:

a) :

$$\begin{array}{rcl} 0.0 & \rightarrow & 0.1 \\ 0.5 & \rightarrow & 0.55 \\ 1.0 & \rightarrow & 1.0 \end{array}$$

b) :

$$\begin{array}{rcl} 0.0 & \rightarrow & 1.0 \\ 0.5 & \rightarrow & 0.5 \\ 1.0 & \rightarrow & 0.0 \end{array}$$

Quadratische Funktion (Parabel):

Eine einfache nach unten geöffnete Parabel wurde untersucht:

$$y = -4.0 \cdot x^2 + 4.0 \cdot x \qquad x \in [0, 1]$$

Die charakteristischen Punkte dieser Funktion sind:

$$\begin{array}{rcl} 0.0 & \rightarrow & 0.0 \\ 0.5 & \rightarrow & 1.0 \\ 1.0 & \rightarrow & 0.0 \end{array}$$

Polynom 5. Grades:

Das Polynom wurde so gewählt, daß die meisten interessanten Punkte (Nullstellen, Minima, Maxima und Wendepunkte) im Intervall $[0, 1]$ liegen:

$$y = -40.5 \cdot x^5 + 101.25 \cdot x^4 - 81.0 \cdot x^3 + 20.25 \cdot x^2 + x$$

mit $x \in [0, 1]$.

Die wichtigsten charakteristischen Punkte dieser Funktion sind:

$$\begin{array}{rcll} 0.0 & \rightarrow & 0.0 & \text{(NS, Min)} \\ 1/3 & \rightarrow & 2/3 & \text{(Max)} \\ 0.5 & \rightarrow & 0.5 & \text{(WP)} \\ 2/3 & \rightarrow & 1/3 & \text{(Min)} \\ 1.0 & \rightarrow & 1.0 & \text{(Max)} \end{array}$$

Sinus-Funktion:

Die Sinusfunktion wurde leicht gestaucht und in das Intervall $[0,1]$ transformiert:

$$y = 0.4 \cdot \sin(2\pi \cdot x) + 0.5$$

mit $x \in [0,1]$.

Die charakteristischen Punkte dieser Funktion sind:

$$\begin{array}{lcl} 0.0 & \to & 0.5 \\ 0.25 & \to & 0.9 \\ 0.5 & \to & 0.5 \\ 0.75 & \to & 0.1 \\ 1.0 & \to & 0.5 \end{array}$$

Die Stauchung der Sinusfunktion hat folgenden Grund: Würde man als Minimum und als Maximum der obigen Funktion die Intervallgrenzen Null und Eins wählen, bräuchte das Netzwerk erheblich längere Trainigszeiten, um die Gewichte so groß werden zu lassen, so daß die logistische Aktivierungsfunktion der Ausgabeunits ihren beiden Grenzwerten, denen sie asymptotisch zustrebt, nahe kommt.

Beim Lernen bietet es sich aufgrund der Symmetrie der Sinusfunktion an, die symmetrische Aktivierungsfunktion zu verwenden. Durch die entsprechende Option des Simulationsprogramms können dafür alle Werte mit Hilfe der Funktion $y = 2.0 \cdot x - 1.0$ in das Intervall $[-1,1]$ transformiert werden.

Die so transformierte Sinusfunktion lautet dann:

$$y = 0.8 \cdot \sin(\pi \cdot (x + 1.0))$$

mit $x \in [-1,1]$.

Die charakteristischen Punkte der transformierten Funktion sind:

$$\begin{array}{rcr} -1.0 & \to & 0.0 \\ -0.5 & \to & 0.8 \\ 0.0 & \to & 0.0 \\ 0.5 & \to & -0.8 \\ 1.0 & \to & 0.0 \end{array}$$

Betragsfunktion:

Die Betragsfunktion wurde in das Intervall $[0,1]$ verschoben:

$$y = |\, 2.0 \cdot x - 1.0 \,|$$

mit $x \in [0,1]$.

Die drei bestimmenden Punkte dieser Funktion sind:

$$\begin{array}{lcl} 0.0 & \to & 1.0 \\ 0.5 & \to & 0.0 \\ 1.0 & \to & 1.0 \end{array}$$

Von Interesse ist hier besonders das Verhalten des Netzwerks im Bereich des Punktes $(0.5, 0.0)$, an dem diese Funktion nicht differenzierbar ist.

Unstetige quadratische Funktion:

Wie eingangs erwähnt, können nur *stetige* Funktionen von einem universellen Approximator wie die hier betrachteten Netzwerke beliebig genau angenähert werden.

Interessant war es daher zu untersuchen, was geschieht, falls das Netzwerk eine Funktion mit einer Sprungstelle approximieren soll. Dazu wurde die folgende Funktion gewählt:

$$
\begin{aligned}
z &= \begin{cases} x^2 & \text{falls } x < 0.5 \\ +\sqrt{0.25 - (x - 0.5)^2} & \text{falls } x \geq 0.5 \end{cases} \\
y &= 1.8 \cdot z + 0.05
\end{aligned}
$$

mit $x \in [0, 1]$.

Diese Funktion beginnt im Nullpunkt mit dem Wert 0.05, folgt bis $x = 0.5$ (dieser Punkt nicht mit eingeschlossen) einem Parabelast aufwärts bis zum (uneigentlichen) Wert 0.5. An dieser Stelle ($x = 0.5$) springt die Funktion auf den Wert 0.95 und fällt anschließend bis $x = 1.0$ in einem Viertelkreis wieder auf den Wert 0.05 ab.

4.9 Das Primzahlen-Prädikat

Zu Vergleichszwecken wurden die Zahlen von 2 bis 255 binär kodiert (8 Bit; jeweils durch die Werte 0.0 und 1.0 repräsentiert) und in einem Testdatensatz zusammengefaßt. Als Targetwert wurde jeder Zahl der Wert 1.0 zugeordnet, falls es sich um eine Primzahl handelt, ansonsten der Wert 0.0.

Das Primzahlenprädikat besitzt für ein nichtrekurrentes, zweistufiges Netz keine erkennbare Regel (vergleiche Denker et al. [4]). Dies läßt sich sehr leicht daran erkennen, daß sehr häufig, wenn beim Training ein oder einige Elemente dieses Testdatensatzes weggelassen werden, das Netzwerk nach Beendigung des Trainings diese nicht gelernten Patterns falsch klassifiziert.

Je nach der Anzahl der "hidden" Units kann das Netzwerk diese Patterns einfach „auswendig“ lernen, oder es wird eigene Merkregeln, „Eselsbrücken“, erfinden.

Von Interesse ist hier der Vergleich, wieviel länger das Netzwerk zum Lernen dieses „regellosen“ Problems benötigt, im Gegensatz zu regelhaften Problemen gleicher Größe.

4.10 Der Aachener Aphasie Test (AAT)

Der Aachener Aphasie Test wurde von Huber et al. [12] an der Neurologischen Klinik (Abteilung Neurolinguistik) des Klinikums der RWTH Aachen zur Diagnoseerstellung bei Patienten mit Aphasien (Sprachstörungen) nach Hirnschlägen, Tumoren oder Unfällen für die Anwendung im klinischen und/oder sprachtherapeutischen Bereich (Logopädie) entwickelt.

Bei diesem Test handelt es sich um den ersten standardisierten und hinsichtlich seiner Gütekriterien (Validität, Reliabilität und Auswertungsobjektivität) überprüften Aphasietest für den deutschsprachigen Raum. Inzwischen wurde der Test außerdem ins Italienische, Niederländische und Englische übertragen. (Siehe Poeck et al. [21])

Der AAT setzt sich aus mehreren Untertests zusammen, die jeweils spezifische sprachliche Leistungen erfassen, und zwar sowohl der Sprachproduktion als auch des Sprachverständnisses, sowohl mündlich als auch schriftlich.

Aus den Ergebnissen der verschiedenen Tests (in Form von Punktwerten) kann der Facharzt oder Logopäde feststellen, ob eine Aphasie vorliegt, den Schweregrad bestimmen und die Aphasie nach den vier Standardsyndromen (globale, Wernicke-, Broca- und amnestische Aphasie) klassifizieren.

Mit Hilfe eines Statistik-Programmpakets zur nichtparametrischen Klassifikationsanalyse sowie einer Normstichprobe von 314 Patienten wurde dazu übergegangen, die Einteilung von Patientenprofilen in die vier Standardklassen zu automatisieren.

Die Normstichprobe setzt sich wie folgt zusammen:

Globale Aphasie	90
Wernicke-Aphasie	74
Broca-Aphasie	79
Amnestische Aphasie	71
Gesamt	314

Nachteilig ist bei dieser statistischen Methode, daß für jedes zu klassifizierende Patientenprofil die Klassifikationswahrscheinlichkeiten für sämtliche Profile der Normstichprobe neu berechnet werden müssen.

Es bot sich daher an, diese Klassifikation mit Hilfe eines Neuronalen Netzes, das anhand der Normstichprobe trainiert werden kann, durchzuführen.

Das hat den Vorteil, daß wenn das Netz einmal ausreichend trainiert ist, die Klassifikation eines neuen Patienten mit Hilfe einer einzigen Propagations-Phase des Netzwerks erfolgen kann, also sehr schnell geht.

Ein weiterer Vorteil besteht darin, daß im Unterschied zu den Klassifikationswahrscheinlichkeiten, die für ein Profil, über alle vier Klassen summiert, stets Eins ergeben, die Ausgabeunits des Netzwerks (die jeweils die Zugehörigkeit zu einem der Standardsyndrome anzeigen) unabhängig voneinander aktiviert werden, und somit die Stärke der Aktivierung ein Maß dafür ist, wie gut sich das Profil in die jeweilige Klasse einordnen läßt. Das erlaubt insbesondere die Unterscheidung zweier verschiedener Arten von nichtklassifizierbaren Aphasien: Solche, bei denen keine Ausgabeunit eine nennenswerte Aktivierung zeigt, und solche, bei denen zwei oder mehr Ausgabeunits hohe Aktivierungen zeigen und miteinander um die höchste Aktivierung „konkurrieren“.

Der AAT setzt sich aus den Untertests wie folgt zusammen:

	Spontansprache:	
1.	Kommunikationsverhalten	0 – 5
2.	Artikulation und Prosodie	0 – 5
3.	Automatisierte Sprache	0 – 5
4.	Semantische Struktur	0 – 5
5.	Phonematische Struktur	0 – 5
6.	Syntaktische Struktur	0 – 5

7. Tokentest:	
Tokentest 1	10 – 0
Tokentest 2	10 – 0
Tokentest 3	10 – 0
Tokentest 4	10 – 0
Tokentest 5	10 – 0
	50 – 0

8. Nachsprechen:	
Nachsprechen 1	0 – 30
Nachsprechen 2	0 – 30
Nachsprechen 3	0 – 30
Nachsprechen 4	0 – 30
Nachsprechen 5	0 – 30
	0 – 150

9. Schriftsprache:	
Schriftsprache 1	0 – 30
Schriftsprache 2	0 – 30
Schriftsprache 3	0 – 30
	0 – 90

10. Benennen:	
Benennen 1	0 – 30
Benennen 2	0 – 30
Benennen 3	0 – 30
Benennen 4	0 – 30
	0 – 120

11. Sprachverständnis:	
Sprachverständnis 1	0 – 30
Sprachverständnis 2	0 – 30
Sprachverständnis 3	0 – 30
Sprachverständnis 4	0 – 30
	0 – 120

Bis auf den Tokentest bedeutet ein Punktwert von Null die schlechteste, der jeweils angegebene maximale Punktwert die bestmögliche Leistung im betreffenden Test. Beim Tokentest ist es genau umgekehrt, hier gibt der Punktwert die Anzahl der Fehler an.

Um ein Netzwerk mit den Normstichproben trainieren zu können, wurden die (Gesamt-) Punktwerte in den mit 1. bis 11. bezeichneten Tests (die Punktwerte aus den Untertests einer Gruppe werden dabei jeweils aufsummiert) durch Division durch den jeweiligen Maximalwert auf das Intervall $[0, 1]$ normiert. Bis auf den Wert des Tokentests wurden diese Werte anschließend von Eins abgezogen, um einheitlich einen umso größeren Wert zu erhalten, je größer der Fehler ist.

Das Netzwerk benötigt demnach 11 Inputunits zur Eingabe der Ergebnisse aus den elf Tests, sowie 4 Outputunits für die Klassifizierung der Testprofile in die vier Standardsyndrome.

Eine Variante dieser Aufgabe, die Klassifikation der Testprofile aus der Normstichprobe zu erlernen, besteht darin, die zu einer Gruppe gehörenden Untertests nicht aufzusummieren (wie das für den obigen Testdatensatz geschah), sondern für jeden der Untertests eine eigene Inputunit vorzusehen; bis auf die Tokentests 1–4, die weiterhin aufsummiert werden, da es sich um dieselbe Aufgabe (Zeigen), nur mit anderen Objekten, handelt.

Diese Variante benötigt dementsprechend 24 Inputunits, bei der gleichen Art der Kodierung wie oben (die Anzahl der Outputunits bleibt unverändert).

Kapitel 5

Simulationsergebnisse

Der Back-Propagation-Algorithmus ist ein weitverbreitetes und überaus erfolgreich eingesetztes Verfahren zum Training von Neuronalen Netzen. Die Popularität von Back-Propagation beruht besonders darauf, daß es sich dabei um das erste Verfahren handelt, das das "Supervised Learning" („überwachte Lernen") von *mehrstufigen* Netzen erlaubte. (Das Trainieren von mehrstufigen Netzen im "free running mode", d.h. ohne Vorgabe von Sollausgaben, ist bereits mit wesentlich einfacheren Mitteln möglich, wie z.B. dem "Competitive Learning")

Trotz seines großen Erfolgs hat Back-Propagation zwei schwere Nachteile: Zum einen lernt es nur äußerst langsam (mehrere tausend Wiederholungen der zu lernenden Patterns sind normal), zum anderen ist die Konvergenzgeschwindigkeit wesentlich (kritisch) von der Wahl der Parameter abhängig. Die optimalen Werte dieser Parameter für ein gegebenes Problem müssen durch Versuch und Irrtum bestimmt werden. Um jedoch die optimalen Werte herauszufinden, muß das eigentliche Anwendungsproblem mehrmals mit Hilfe des Verfahrens gelernt werden — was unsinnig ist, falls man nur an der einmaligen Lösung einer ganz bestimmten Lernaufgabe interessiert ist.

Im Falle des Standard-Verfahrens gibt es im Wesentlichen zwei Parameter: Die Lernrate η und den Momentum- (Trägheits-) Faktor α. (Sowie den Parameter r für die Initialisierung der Gewichtsmatrizen mit (gleichverteilten) Zufallszahlen zwischen $-r$ und $+r$)

Weitere Parameter kommen hinzu, falls man kleinere Variationen des Back-Propagation-Verfahrens benutzt:

Die Verwendung der symmetrischen Aktivierungsfunktion, die Verwendung einer hyperbolischen (*atanh*) Fehlerfunktion, das Nullsetzen von partiellen Ableitungen, falls der Fehler einer Ausgabeunit unter eine bestimmte Schranke fällt, oder das Addieren eines Offsets zur Ableitung der Aktivierungsfunktion zur Vermeidung ihrer Nullstellen (um ein Steckenbleiben des Verfahrens zu verhindern).

Je nach dem verwendeten Lernmodus kommen zusätzliche Parameter hinzu:

Beim Sequential Learning die maximale Anzahl von Wiederholungen eines Patterns, beim Periodical Learning der Pattern "repeat factor" (was im Grunde genommen dasselbe ist), und beim Dynamic Learning die Anzahl der Zyklen, nach der alle Gewichte der Patterns neu berechnet werden, sowie die Verwendung eines Penalty-Terms, der die Aktivierung von mehr als einer Ausgabeunit zusätzlich „bestraft" (für Klassifikationsprobleme).

Die Wahl des Abbruchkriteriums schließlich hat zwar keinen Einfluß auf das Training des Netzes selbst, je nach Typ des Kriteriums (hier Maximumnorm oder *Root Mean Squares* Error Measure) und der Größe der gewählten Fehlerschranke(n) (hier das "absolute error limit" ϵ_{abs} und das "pattern error limit" ϵ_{pat}) beeinflußt es die zu seiner Erfüllung erforderliche Trainingszeit des Netzes.

Das Ziel dieser Arbeit besteht nun darin, einmal die Abhängigkeit der Konvergenzgeschwindigkeit von der Wahl dieser Parameter zu untersuchen, und zum anderen die in Kapitel 3 vorgestellten Verfahren miteinander zu vergleichen.

Gesucht wird dabei ein Verfahren, das möglichst schnell konvergiert, möglichst zuverlässig ist (d.h. möglichst unabhängig vom Problem und von der Initialisierung der Gewichtsmatrix gleichbleibend gute Eigenschaften besitzt) und/oder so wenig Parameter wie möglich benötigt (oder zumindest möglichst unkritisch von diesen abhängt).

Außerdem sollte ein Verfahren der Wahl über eine hohe Generalisierungsfähigkeit verfügen.

Der Begriff der „Generalisierungsfähigkeit" ist jedoch unter Forschern von Konnektionistischen Modellen ein bisher nur vage definierter und außerdem kontrovers diskutierter[1] Begriff, der meist sehr von der Problemstellung abhängt. Zudem ist die Überprüfung der Generalisierungsleistung eines Netzes äußerst aufwendig. Aus diesen Gründen wurden in dieser Arbeit von einer Untersuchung der Generalisierung abgesehen.

Die Implementation der vorgestellten Verfahren, Variationen und Lernmodi in meinem Programm erlaubt die Kombination fast jedes Verfahrens mit fast jeder Variation und mit fast jedem Lernmodus. Dadurch entsteht ein vieldimensionaler Parameterraum; man kann berechtigterweise von einer kombinatorischen Explosion von möglichen Verfahren sprechen.

Da die Untersuchung aller dieser Möglichkeiten im Rahmen dieser Arbeit unmöglich ist, mußte ich mich auf relativ wenige Versuche beschränken. Nicht alle Kombinationen und nicht alle Parameter werden hier behandelt.

Um die Darstellung der Simulationsergebnisse möglichst übersichtlich zu halten, folgt hier eine Liste sämtlicher Parameter, die in meinem Programm vorkommen, zusammen mit ihrer standardmäßigen (default) Einstellung. *Falls nicht anders vermerkt, wird in allen beschriebenen Versuchen von dieser Standardeinstellung ausgegangen.*

Historisch bedingt, und zur Vermeidung von Irrtümern während der Benutzung, ist die Verwendung der symmetrischen Aktivierungsfunktion in meinem Programm kein während des Programmlaufs veränderbarer Parameter, sondern eine grundsätzliche Einstellung, die zu Beginn der Programmausführung getroffen wird. Falls nicht anders vermerkt, wurde in den folgenden Simulationen die normale (logistische) Aktivierungsfunktion verwendet.

```
          --- Simulation Parameters ---

(I) [I]nitialization range          =    0.2000000
(E) [E]ta (learning rate)           =    0.5000000
(M) [M]omentum term                 =    0.9000000
(R) [R]oot Mean Squares/MAXimum norm error:    MAX
(H) [H]yperbolic (atanh) error function:       OFF
(S) [S]everal output activations penalty term:  ON
```

[1] "Connectionists" Network (electronic mail)

```
(Z) [Z]ero error if | (tar - out) | <    0.0000000
(B) [B]ias error ( O * (1-O) + ... ):    0.0000000
(A) [A]bsolute error limit         =     0.0020000
(P) [P]attern error limit          =     0.0020000
(C) [C]ancel learn steps that increased error: OFF
(F) [F]actor for pattern repetition =    1
(T) [T]oggle epochs/cycles below:           EPOCHS
(U) [U]pdate pattern weights every  =    1  epochs
(X) e[X]tra update (learn) rules: Standard

(O) [O]ption: save best matrix automatically   OFF
(K) [K]eep a copy of best matrix found          ON
```

Mit Hilfe der X-Funktion des Parameter-Menüs können die speziellen Lernregeln 1–7 angewählt werden. Regel 0 ist das Standardverfahren:

```
0  :  Standard
1  :  Schmidhubers Zero-Point-Search
2  :  ChanFallside (autom. eta/mom)
3  :  Fahlmans QuickProp
4  :  Jacobs delta-bar-delta-rule
5  :  Almeidas adaptive BP
6  :  Perceptron
7  :  Steffen's autom. eta/mom
```

```
(1)          --- Schmidhubers algorithm parameters ---

             (M) [M]aximum step size         =   20.0000

(2)          --- Chan & Fallside's automatic choice of eta & mom ---

             (N) mi[N]imum for eta (learning rate)   =    0.0001000
             (X) ma[X]imum for eta (learning rate)   =   20.0000000
             (L) [L]ambda (for momentum calculation) =    0.9000000

(3)              --- QuickProp parameters ---

             (H) [H]yperbolic (atanh) error function:            OFF
             (S) [S]plit learn rate ("split eta"):               OFF
             (T) [T]hreshold (below:descent/above:jump) =    0.0000
             (M) [M]aximum step factor                  =    1.7500
             (D) weight [D]ecay factor                  =    0.0001000

(4)      --- delta-bar-delta rule parameters ---

             (T) [T]heta         =    0.7000
             (K) [K]appa         =    0.0500
             (P) [P]hi           =    0.1000
```

```
(5)        --- Almeidas adaptive algorithm parameters ---

               (U) [U]p factor           =    1.2000
               (D) [D]own factor         =    0.7000
```

Das Perceptron besitzt keine eigenen Parameter außer der Lernrate η.

```
(7)            --- Steffen's automatic choice of eta & mom ---

               (N) mi[N]imum for eta (learning rate)   =    0.0025000
               (X) ma[X]imum for eta (learning rate)   =    0.5000000
```

5.1 Das XOR-Problem

Obwohl das XOR-Problem für "Real World"-Anwendungen nicht repräsentativ ist, sowohl was seine Größe als auch seine Qualität als boolesches Problem anbelangt, das für ähnliche Eingaben (Hamming-Abstand) entgegengesetzte Ausgaben verlangt (vergleiche Kapitel 4), eignet es sich gerade seiner geringen Größe wegen zu einer ausgiebigeren Untersuchung des Parameterraums. Darüber hinaus erlaubt es einen Vergleich mit den Ergebnissen von anderen Forschern, da das XOR-Problem in nahezu jeder Arbeit untersucht wird.

Die folgenden Simulationen wurden alle mit zwei hidden Units durchgeführt; da nur eine einzige Outputunit vorhanden ist, wurde die "[S]everal output activations penalty term"-Option abgeschaltet.

Vor Beginn der Simulationen wurden zehn Gewichtsmatrizen erzeugt, die jeweils mit gleichverteilten Zufallszahlen im Bereich $[-0.2, +0.2]$ belegt wurden, und die im folgenden mit X0 bis X9 bezeichnet werden.

Vor Beginn eines Versuchslaufs wurde stets eine dieser Matrizen eingelesen. Ziel dieser Anordnung des Experiments war es unter anderem, den Einfluß der zufälligen Initialisierung der Gewichtsmatrix zu bestimmen.

Das "absolute error limit" ϵ_{abs} und das "pattern error limit" ϵ_{pat} wurde einheitlich auf den Wert 0.1 festgelegt.

Die verschiedenen Lernregeln (0–7) wurden in der zu Eingang des Kapitels beschriebenen Standardreihenfolge untersucht. Innerhalb dieser Abschnitte wurden wiederum die Lernmodi (Sequential, Periodical, Batch, Dynamic) in teils wechselnder Reihenfolge variiert.

Um den Einfluß der Lernrate auf das Standardverfahren zu untersuchen, wurde zunächst eine Versuchsserie ohne Trägheitsterm (mit $\alpha = 0.0$) durchgeführt.

Ausgehend von extrem kleinen Werten wurde die Lernrate η dann stufenweise erhöht.

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen
X0	0.01	0.0	$\perp$
X9			$\perp$
X0	0.05	0.0	$\perp$
X9			$\perp$
X0	0.1	0.0	$\perp$
X9			$\perp$
X0	0.2	0.0	$\perp$
X9			6053

Das Zeichen "$\perp$" bedeutet, daß das Verfahren in einem lokalen Minimum steckengeblieben ist.

Dieser Vorversuch (im Verein mit den folgenden Versuchen) zeigt deutlich, daß zu kleine Werte von η das Lernen extrem verlangsamen oder sogar ganz verhindern.

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.5	0.0	2568	0.7	0.0	1767	0.9	0.0	1380
X1			2227			1564			1268
X2			2324			1589			1216
X3			2855			2080			1732
X4			2290			1566			1211
X5			2207			1503			1145
X6			2084			1429			1086
X7			2728			$\perp$			$\perp$
X8			$\perp$			1701			1239
X9			2065			1422			1080

Mit zunehmender Lernrate werden die benötigten Trainingszeiten bis zur Erfüllung des Abbruchkriteriums (das Maximum der lokalen Fehler der vier Patterns muß kleiner werden als $\epsilon_{abs} = 0.1$) kürzer.

Vergleicht man die Trainingszeiten der verschiedenen Initialisierungen untereinander, fällt beispielsweise auf, daß in allen Fällen X9 die kürzesten Trainingszeiten benötigt, dicht gefolgt von X6, während X7 das schlechteste Ergebnis liefert. Diese Beobachtung wird durch die folgenden Versuche weiter erhärtet.

Der nächste Versuch zeigt in groben Schritten die Tendenz der benötigten Trainingszeiten in Abhängigkeit von großen und sehr großen Werten für die Lernrate:

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	1.0	0.0	1255	3.0	0.0	824	20.0	0.0	$\perp$
X9			965			365			590
X0	1.2	0.0	1084	5.0	0.0	1245	25.0	0.0	–
X9			801			306			319
X0	1.5	0.0	937	10.0	0.0	$\perp$	30.0	0.0	–
X9			642			348			3083
X0	2.0	0.0	832	15.0	0.0	$\perp$			
X9			495			1081			

"–" : nicht untersucht

Zuerst nehmen die Trainingszeiten ab, bis sie nach Erreichen eines Optimums wieder ansteigen, bis schließlich das Verfahren bei zu großen Lernraten steckenbleibt. Interessant ist, daß diese Veränderungen zwar im Wesentlichen „glatt" zu verlaufen scheinen, einzelne „Ausrutscher" aber durchaus vorkommen, wie z.B. für $\eta = 15.0$ bei X9. Außerdem sieht man, daß das Optimum für η je nach Initialisierung unterschiedlich ist.

Das Steckenbleiben hatte im Übrigen mehrere Ausprägungen, d.h es gab mehrere verschiedene lokale Minima der Fehlerfläche. Eines der häufigsten bestand darin, daß für jedes Pattern die Aktivierung der Ausgabeunit 0.5 betrug.

Die Hinzunahme des Trägheitsterms bei noch relativ kleiner Lernrate zeigt den beschleunigenden Effekt dieses Terms, der mit steigender Größe des Trägheitsfaktors α zunimmt:

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.5	0.1	2302	0.5	0.3	1796	0.5	0.5	1299
X1			2003			1559			1123
X2			2080			1622			1172
X3			2577			2017			1454
X4			2054			1601			1160
X5			1975			1536			1112
X6			1866			1451			1047
X7			2552			2083			1465
X8			2881			1783			1261
X9			1850			1438			1035

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.5	0.7	801	0.5	0.9	289	0.5	0.95	168
X1			686			245			138
X2			721			257			146
X3			890			328			217
X4			713			254			144
X5			682			244			138
X6			643			230			129
X7			859			306			217
X8			778			278			162
X9			633			225			127

Wieder kann man beobachten, daß X9 die kürzesten Trainingszeiten erfordert, gefolgt von X6. X7 schneidet nicht immer als schlechteste Matrix ab, ist jedoch in der Regel eine der schlechtesten Initialisierungen.

Dies zeigt deutlich den nicht zu unterschätzenden Effekt einer günstigen Initialisierung auf die Trainingsdauer.

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	Matrix	η	α	Epochen
X0	0.5	0.99	$\perp$	X5	0.5	0.99	185
X1			$\perp$	X6			132
X2			435	X7			$\perp$
X3			$\perp$	X8			143
X4			$\perp$	X9			97

Bei extremen Werten von α erzielen einige Matrizen (z.B. X9) Spitzenresultate, während bei anderen das Verfahren bereits steckenbleibt.

Steigert man zusätzlich zum Trägheitsterm die Lernrate, verkürzen sich die Trainingszeiten weiter:

Standard / Periodical
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.7	0.3	1241	0.7	0.5	892	0.7	0.7	547
X1			1099			786			478
X2			1115			804			493
X3			1477			1051			632
X4			1099			794			486
X5			1050			759			466
X6			999			719			439
X7			$\perp$			$\perp$			1219
X8			1151			833			512
X9			990			713			436

Standard / Periodical
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.7	0.9	201	0.7	0.95	138	0.7	0.99	$\perp$
X1			174			109			$\perp$
X2			178			108			$\perp$
X3			249			288			$\perp$
X4			177			110			$\perp$
X5			171			103			$\perp$
X6			161			98			$\perp$
X7			404			$\perp$			$\perp$
X8			189			115			$\perp$
X9			157			95			238

Wiederum verschlechtern sich die Ergebnisse dramatisch, sobald α einen extremen großen Wert annimmt.

Eine weitere Steigerung der Lernrate führt zu noch besseren Ergebnissen:

Standard / Periodical
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.9	0.3	972	0.9	0.5	693	0.9	0.7	420
X1			891			624			374
X2			853			616			374
X3			1237			869			517
X4			853			611			373
X5			799			578			352
X6			762			549			339
X7			$\perp$			$\perp$			$\perp$
X8			868			630			387
X9			755			545			333

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.9	0.9	162	0.9	0.95	155	0.9	0.99	–
X1			138			120			–
X2			139			97			–
X3			231			$\perp$			–
X4			139			105			–
X5			133			88			–
X6			123			84			–
X7			$\perp$			$\perp$			–
X8			143			95			–
X9			121			81			$\perp$

Der erste Teil des folgenden Versuchs zeigt bei konstanter, ziemlich großer Lernrate erst eine Abnahme, dann wieder eine Zunahme der Trainingszeiten mit wachsendem Trägheitsfaktor.

Der zweite Teil zeigt, daß der für X9 optimale Wert für die anderen Matrizen teilweise bereits zu groß ist, für manche jedoch ebenfalls vorteilhaft ist. Die Lernrate von 1.0 ist im Vergleich zum obigen Versuch mit kleinerer Lernrate (0.9) und dafür größerem Trägheitsfaktor (0.95) für manche Matrizen günstiger (z.B. X0, X3), für manche schlechter (z.B. X1, X2). Für X6 dagegen ist bei dieser Wahl der beiden Parameter die Anzahl der benötigten Epochen (84) die gleiche wie bei $\eta = 2.0$ und $\alpha = 0.9$.

Standard / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen
X9	2.0	0.5	250	2.0	0.95	155
X9	2.0	0.9	69	5.0	0.9	$\perp$

Matrix	η	α	Epochen	η	α	Epochen
X0	1.0	0.9	151	2.0	0.9	218
X1			130			$\perp$
X2			127			105
X3			238			$\perp$
X4			127			180
X5			119			82
X6			112			84
X7			$\perp$			$\perp$
X8			130			98
X9			110			69

Zusammenfassend kann man sagen, daß weder sehr kleine noch extrem große Lernraten vorteilhaft sind. Günstig, wenn auch nicht optimal, war für alle Initialisierungen ein Bereich etwa ab 0.5 bis etwa 1.0, vorausgesetzt der Trägheitsfaktor betrug etwa 0.9. Vereinzelt ließen sich durch extremere Einstellungen bessere Ergebnisse erzielen.

Der Vergleich der Ergebnisse bei gleicher Lernrate mit wechselndem Trägheitsfaktor zeigt eindeutig den Geschwindigkeitszuwachs, den der Trägheitsterm mit sich bringt.

Stellenweise kann man einen Trade-Off zwischen Lernrate und Trägheitsfaktor beobachten. Die von Chan & Fallside (siehe Kapitel 3) gegebene Empfehlung, die beiden Parameter proportional zueinander anwachsen zu lassen, wird dadurch bestätigt: Die Steigerung des einen Parameters ohne den anderen ist nicht so effektiv wie die gleichzeitige Steigerung beider Werte.

Die Verwendung derselben Initialisierungsmatrizen in allen Versuchen zeigt die Bedeutung der Initialisierung für die Konvergenzgeschwindigkeit: Die Matrizen zeigten nahezu in allen Versuchen übereinstimmend dieselbe Rangfolge in den benötigten Trainingszeiten.

Die folgenden Versuche sollen die Frage erhellen, inwieweit die für das Periodical Learning gefundenen Ergebnisse auch auf andere Lernmodi, hier den Batch-Modus, übertragbar sind.

Mit "Batch1" wird dabei abkürzend das Standard-Batch-Verfahren, das "Sum-of-Derivatives" Batch Learning, bezeichnet. Das "Sum-of-Updates" Batch Learning wird mit "Batch2" bezeichnet. Bei Verwendung der Standard-Lernregel sind diese beiden Verfahren jedoch äquivalent (vergleiche Kapitel 2), weswegen das letztere hier nicht nochmals separat untersucht wurde.

Ein Vorversuch diente dazu, grob den relevanten Bereich der beiden Parameter einzugrenzen:

Vorversuch
Standard / Batch1
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X9	0.5	0.9	$\perp$	0.9	0.95	848	15.0	0.95	$\perp$
X9	0.7	0.9	$\perp$	1.0	0.95	379	20.0	0.9	$\perp$
X9	0.7	0.95	$\perp$	2.0	0.95	155	20.0	0.95	$\perp$
X9	0.7	0.99	221	5.0	0.95	74			
X9	0.9	0.9	$\perp$	10.0	0.95	39			

Dieser Versuch legte den Gedanken nahe, daß im Gegensatz zum Periodical Learning hier ein größerer Trägheitsfaktor günstig sei:

Standard / Batch1
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.7	0.9	658	0.7	0.95	443	0.7	0.99	382
X1			528			316			189
X2			1314			722			324
X3			1069			1023			$\perp$
X4			697			449			353
X5			1856			626			182
X6			586			361			$\perp$
X7			1616			912			467
X8			453			301			203
X9			$\perp$			$\perp$			221

Diese Ergebnisse sind schlechter als beim Periodical Learning mit denselben Parametern, zumindest was die Anzahl der Epochen betrifft. Während beim Batch1-Learning die Anzahl der Updates mit der Anzahl der Epochen übereinstimmt, kann die Anzahl der Updates beim Periodical Learning (bei einem "repeat factor" von Eins) maximal die Anzahl der Patterns (hier vier) mal der Anzahl der Epochen betragen.

Wären die Anzahlen der Epochen annähernd gleich, wäre das Batch1-Verfahren mit großer Wahrscheinlichkeit in der Anzahl der Updates überlegen. So jedoch lohnt sich eine genauere Untersuchung des Batch1-Learnings sowie ein Vergleich der Anzahl der Updates zwischen Periodical und Batch1 Learning nicht, da keine nennenswerten Verbesserungen zu erwarten sind.

Bei zu großem Trägheitsfaktor (0.99) trat bereits wieder Steckenbleiben auf (nur bei X9 war genau das umgekehrte Verhalten zu beobachten). Aus diesem Grund wurde der Wert $\alpha = 0.99$ im folgenden Versuch nicht mehr untersucht.

Standard / Batch1

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
X0	0.9	0.9	530	0.9	0.95	377	2.0	0.95	260
X1			418			258			140
X2			1032			583			286
X3			988			811			158
X4			559			375			223
X5			1547			401			149
X6			468			296			163
X7			1274			735			386
X8			373			250			145
X9			$\perp$			848			155

Standard / Batch1

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	η	α	Epochen
X0	5.0	0.95	152	10.0	0.95	$\perp$
X1			72			$\perp$
X2			128			46
X3			$\perp$			$\perp$
X4			165			$\perp$
X5			66			$\perp$
X6			$\perp$			$\perp$
X7			198			$\perp$
X8			82			47
X9			74			39

Sehr große Werte der Lernrate führten auch hier zur Beschleunigung; bei extremen Werten wie $\eta = 10.0$ trat wieder massives Steckenbleiben auf.

Der Vergleich zwischen Periodical und Batch1 Learning zeigt annähernd gleiche Reaktionen auf die Parameterwerte, beim Batch1 Learning waren jedoch geringfügig höhere Werte möglich, bevor die Lernversuche steckenblieben.

Bei den nahezu optimalen Werten von $\eta = 2.0$ und $\alpha = 0.9$ für das Periodical und von $\eta = 5.0$ und $\alpha = 0.95$ für das Batch1 Learning war (unter Berücksichtigung der Anzahl der Updates) das Batch1 Learning leicht überlegen.

Bei den folgenden Versuchen mit Sequential Learning wurde eine Maximalzahl von hundert Wiederholungen festgelegt (daher die Bezeichnung "Sequential 100").

Da die Anzahl der Epochen beim Sequential Learning nicht sehr aussagekräftig ist, wurde jeweils auch die Anzahl der Updates angegeben. (Diese kann bei einem maximalen Wiederholungsfaktor von hundert und bei vier Patterns bis zu vierhundertmal großer sein als die Anzahl der Epochen)

Standard / Sequential 100
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	Updates	η	α	Epochen	Updates
X0	0.5	0.9	203	37,489.	0.7	0.9	$\perp$	–
X1			65	10,647.			36	4,467.
X2			85	11,672.			$\perp$	–
X3			$\perp$	–			$\perp$	–
X4			$\perp$	–			51	8,120.
X5			168	19,206.			$\perp$	–
X6			122	21,160.			$\perp$	–
X7			$\perp$	–			$\perp$	–
X8			233	21,758.			32	4,941.
X9			$\perp$	–			$\perp$	–

Zuerst wurde mit den Standardwerten von $\eta = 0.5$ und $\alpha = 0.9$ begonnen. Diese Ergebnisse waren im Vergleich zum Periodical Learning extrem schlecht. Entgegen dem Verhalten beim Periodical Learning führte eine geringfügige Erhöhung der Lernrate nicht zu einer moderaten Verbesserung, sondern bei einigen Matrizen zu sprunghaft besseren Ergebnissen, bei den meisten Matrizen jedoch zum Steckenbleiben.

Standard / Sequential 100
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	Updates	η	α	Epochen	Updates
X0	0.5	0.5	34	2,874.	0.5	0.0	45	6,486.
X1			29	3,325.			33	6,378.
X2			38	3,179.			41	5,748.
X3			33	2,906.			40	5,863.
X4			33	3,778.			40	5,942.
X5			29	3,432.			$\perp$	–
X6			$\perp$	–			171	23,499.
X7			30	3,468			$\perp$	–
X8			28	3,318.			37	6,755.
X9			28	3,186.			36	6,474.

Die *Verringerung* des Trägheitsfaktors erbrachte zunächst eine wesentliche Verbesserung der Resultate. Das völlige Abschalten des Trägheitsterms ließ die Lernzeiten wieder ansteigen, sie waren aber immer noch weit kürzer als die, die im vorigen Versuch mit den Standardparametern erzielt wurden.

Da dieses Verhalten sich sehr von dem des Periodical Learning unterscheidet, diente ein Zwischenversuch dazu, günstige Parameterkombinationen zu finden.

Zwischenversuch
Standard / Sequential 100
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	Updates	η	α	Epochen	Updates
X1	0.7	0.5	27	2,354.	2.0	0.2	29	1,336.
X1	0.9	0.5	27	1,803.	5.0	0.5	26	408.
X1	2.0	0.5	25	786.	10.0	0.5	19	319.

Die aussichtsreichsten Parameterwerte wurden anschließend noch einmal an allen Matrizen getestet.

Standard / Sequential 100
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	Updates	η	α	Epochen	Updates
X0	2.0	0.5	28	840.	5.0	0.5	24	348.
X1			25	786.			26	408.
X2			25	724.			20	292.
X3			27	817.			58	926.
X4			32	1,095.			$\perp$	–
X5			24	714.			22	323.
X6			23	685.			20	286.
X7			26	816			$\perp$	–
X8			24	720.			20	297.
X9			29	879.			$\perp$	–

Standard / Sequential 100
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen	Updates
X0	10.0	0.5	18	243.
X1			19	319.
X2			28	422.
X3			$\perp$	–
X4			$\perp$	–
X5			22	292.
X6			16	299.
X7			$\perp$	–
X8			17	226.
X9			51	894.

Die mit diesen Parametern erzielten Ergebnisse waren um ein Vielfaches schlechter als beim Periodical Learning. Außerdem blieb das Verfahren bei einigen der Matrizen bereits stecken, weswegen keine größere Lernrate mehr untersucht wurde.

Da das Sequential Learning jedes Pattern solange trainiert, bis es gelernt ist (oder die Maximalzahl von Wiederholungen erreicht ist), besteht die Gefahr, daß bereits vorher gelernte Patters wieder „vergessen" werden, vor allem, wenn die Ausgaben ähnlicher Patterns nicht ebenfalls ähnlich sind und sich gegenseitig verstärken, wie das beim XOR-Problem der Fall ist.

Dadurch erscheint das wesentlich schlechtere Abschneiden des Sequential Learnings plausibel. Ein Vergleich mit homogeneren Testdaten wird diese Vermutung bestätigen oder widerlegen helfen.

Schließlich wurde das Dynamic Learning getestet.

Da beim Dynamic Learning die Auswahl der Patterns zufällig ist, ist eine gewisse zufällige Schwankung der Lernzeiten zu erwarten. Zwar reicht die Anzahl der durchgeführten Versuche nicht zu einer statistischen Untersuchung, dennoch läßt sich auch so eine gewisse Vorstellung von der Größe dieser Schwankungen gewinnen.

Wieder wurde zunächst der Parameterraum relativ weitmaschig nach aussichtsreichen Parameterkombinationen durchsucht.

Vorversuch
Standard / Dynamic
1 Epoche = 8 Zyklen
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	η	α	Epochen 1. Vers.	Epochen 2. Vers.	Epochen 3. Vers.	Epochen 4. Vers.	Epochen 5. Vers.
X9	1.0	0.95	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
X9	1.0	0.5	252	248	223	253	212
X9	2.0	0.5	127	209	134	124	130
X9	5.0	0.5	98	101	151	87	451
X9	10.0	0.5	$\perp$	$\perp$	$\perp$	424	138
X9	2.0	0.0	237	$\perp$	259	273	247
X9	1.0	0.0	592	506	494	511	515
X9	5.0	0.0	191	126	136	116	133
X9	10.0	0.0	90	105	77	166	103
X9	20.0	0.0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
X9	0.9	0.9	$\perp$	$\perp$	185	$\perp$	$\perp$
X9	2.0	0.2	180	211	216	275	185
X9	5.0	0.2	110	90	140	94	101

Eine der Parameterkombinationen mit den besten Ergebnissen, die nach den Erfahrungen mit dem Periodical Learning schon als relativ groß, aber noch „vorsichtig" gelten kann, wurde anschließend ausführlicher untersucht.

Standard / Dynamic
1 Epoche = 4 Zyklen
$\epsilon_{abs} = 0.1$ $\epsilon_{pat} = 0.1$

Matrix	η	α	Epochen 1. Vers.	Epochen 2. Vers.	Epochen 3. Vers.	Epochen 4. Vers.	Epochen 5. Vers.
X0	2.0	0.5	417	469	302	287	429
X1			$\perp$	$\perp$	$\perp$	504	498
X2			290	626	333	307	335
X3			453	569	659	474	977
X4			419	391	369	391	293
X5			229	326	–	–	–
X6			270	340	–	–	–
X7			690	514	–	–	–
X8			360	294	–	–	–
X9			338	299	–	–	–

Die Ergebnisse fielen jedoch schlechter aus als nach dem Vorversuch erwartet.

Da die Lernzeiten um einiges größer waren als beim Periodical Learning (mit ähnlichen Parametern) und die Differenzen zwischen den Versuchen an einer Matrix mit bis zu rund 300 Epochen sehr groß waren (wobei die Durchführung mehrerer Versuche pro Matrix sehr zeitaufwendig ist) wurde der Parameterraum der Standard-Lernregel unter dem Dynamic Learning nicht weiter untersucht in der Hoffnung, unter den speziellen Lernregeln ohne Parametersuche noch wesentlich bessere Ergebnisse zu finden.

Als erste der speziellen Lernregeln wurde das Verfahren von Schmidhuber zur Nullstellensuche untersucht.

Schmidhuber
$\epsilon_{abs} = 0.1$ $\epsilon_{pat} = 0.1$

Periodical

Matrix	Epochen	Updates
X0	264	918.
X1	1119	3,521.
X2	>2000	–
X3	568	1,851.
X4	>2000	–
X5	1339	4,178.
X6	220	780.
X7	210	756.
X8	496	1,637.
X9	224	797.

Sequential 100

Matrix	Epochen	Updates
X0	29	297.
X1	26	294.
X2	26	270.
X3	29	302.
X4	34	382.
X5	26	275.
X6	24	258.
X7	28	301.
X8	25	270.
X9	27	311.

Schmidhuber

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Dynamic

Matrix	Epochen	Updates
X0	324	1,224.
X1	>2000	–
X2	456	1,705.
X3	820	3,105.
X4	198	739.
X5	249	918.
X6	1166	4,418.
X7	>2000	–
X8	878	3,281.
X9	214	770.

Batch1

Matrix	Epochen
X0	184
X1	171
X2	182
X3	229
X4	154
X5	149
X6	139
X7	>2000
X8	150
X9	143

Batch2

Matrix	Epochen
X0	61
X1	116
X2	111
X3	122
X4	$\perp$
X5	93
X6	80
X7	108
X8	110
X9	110

Der Vorteil des Verfahrens von Schmidhuber besteht darin, keine Parameter zu benötigen (außer einer maximalen Schrittweite, die man jedoch auch als festen Bestandteil des Verfahrens ansehen kann, da ihre Wahl unkritisch ist).

Die Ergebnisse dieses Verfahrens in Kombination mit Sequential Learning sind im Vergleich zum Standardverfahren mit Sequential Learning exzellent. Die Ergebnisse in den übrigen Lernmodi sind im Vergleich zwar nicht optimal, aber durchaus als passabel bis gut zu bezeichnen.

Eine gewisse Überraschung ist das wesentlich bessere Abschneiden des Batch2 Learnings im Vergleich zum Batch1 Learning sowie auch im Vergleich zu den übrigen Lernmodi.

Chan&Fallside / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

$\eta_{min} = 0.0001 \qquad \eta_{max} = 20.0$

Matrix	η_0	λ_0	Epochen
X0	0.5	0.9	$\perp$
X1			$\perp$
X2			$\perp$
X3			$\perp$
X4			$\perp$
X5			$\perp$
X6	0.9	0.95	$\perp$
X7	2.0	0.95	$\perp$
X8	5.0	0.95	$\perp$
X9	20.0	0.95	$\perp$

Das Verfahren von Chan & Fallside blieb beim Periodical Learning auch unter etwas variierenden Anfangswerten stets stecken, wobei die Lernrate auf den niedrigsten Wert sank und dort blieb.

Chan&Fallside

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

$\eta_{min} = 0.0001 \qquad \eta_{max} = 20.0$

Batch1

Matrix	Epochen
X0	93
X1	92
X2	186
X3	116
X4	125
X5	217
X6	85
X7	229
X8	419
X9	94

Sequential 100

Matrix	Epochen	Updates
X0	93	2,363.
X1	99	1,538.
X2	$\perp$	–
X3	$\perp$	–
X4	33	612.
X5	$\perp$	–
X6	$\perp$	–
X7	$\perp$	–
X8	$\perp$	–
X9	$\perp$	–

Allein das Batch1-Learning führte beim Verfahren von Chan & Fallside zum Erfolg, mit im Vergleich zum Standardverfahren sehr guten Ergebnissen.

(Beim Batch2 Learning blieb das Verfahren stets stecken, was plausibel ist, da die automatische Anpassung des Trägheitsfaktors ins Leere läuft, da der Trägheitsterm bei diesem Lernmodus erst am Schluß einer Epoche berechnet wird und dafür der Trägheitsfaktor des letzten Patterns verwendet wird)

Fahlmans QuickProp / Batch1

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

ohne atanh-Fehler
ohne split-eta

Matrix	Epochen
X0	$\perp$
X1	$\perp$
X2	$\perp$
X3	$\perp$
X4	$\perp$
X5	$\perp$
X6	45
X7	$\perp$
X8	59
X9	63

ohne atanh-Fehler
mit split-eta

Matrix	Epochen
X0	$\perp$
X1	$\perp$
X2	74
X3	$\perp$
X4	$\perp$
X5	$\perp$
X6	$\perp$
X7	$\perp$
X8	$\perp$
X9	54

Fahlmans QuickProp / Batch1

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

mit atanh-Fehler *ohne* split-eta

Matrix	Epochen
X0	$\perp$
X1	69
X2	60
X3	$\perp$
X4	$\perp$
X5	$\perp$
X6	46
X7	$\perp$
X8	58
X9	$\perp$

mit atanh-Fehler *mit* split-eta

Matrix	Epochen
X0	$\perp$
X1	$\perp$
X2	$\perp$
X3	$\perp$
X4	$\perp$
X5	$\perp$
X6	53
X7	80
X8	67
X9	54

Das Verfahren von Fahlman, QuickProp, bewährte sich bei diesem Testproblem nicht. Möglicherweise wären die Ergebnisse durch Variation der übrigen Parameter dieses Verfahrens zu verbessern gewesen; da jedoch ein Verfahren gesucht wird, das möglichst ohne "parameter tuning" auskommt, wurde diese Möglichkeit nicht weiter untersucht.

Jacobs Delta-bar-delta-rule

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Batch1

Matrix	Epochen
X0	232
X1	231
X2	286
X3	470
X4	212
X5	340
X6	214
X7	276
X8	172
X9	$\perp$

Batch2

Matrix	Epochen
X0	$\perp$
X1	$\perp$
X2	–
X3	–
X4	–
X5	–
X6	–
X7	–
X8	–
X9	–

Jacobs Delta-bar-delta-rule

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Periodical

Matrix	Epochen
X0	785
X1	$\perp$
X2	$\perp$
X3	$\perp$
X4	$\perp$
X5	$\perp$
X6	$\perp$
X7	$\perp$
X8	$\perp$
X9	$\perp$

Sequential 100

Matrix	Epochen
X0	$\perp$
X1	$\perp$
X2	$\perp$
X3	$\perp$
X4	$\perp$
X5	–
X6	–
X7	–
X8	–
X9	–

Das Verfahren von Jacobs, die Delta-bar-delta-Regel, konvergierte bis auf eine Ausnahme nur mit Batch1 Learning. Diese Ergebnisse waren jedoch nicht besser als die des Standardverfahrens; die Ergebnisse des Verfahrens von Chan & Fallside waren außerdem weit besser.

Möglicherweise zeigten die individuellen Lernraten dieses Verfahrens aufgrund der geringen Anzahl von Patterns und von Gewichten bei diesem Problem kaum eine beschleunigende Wirkung.

Silva&Almeida

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Batch1

Matrix	Epochen
X0	253
X1	113
X2	312
X3	171
X4	$\perp$
X5	338
X6	170
X7	$\perp$
X8	$\perp$
X9	$\perp$

Periodical

Matrix	Epochen
X0	$\perp$
X1	$\perp$
X2	$\perp$
X3	$\perp$
X4	$\perp$
X5	–
X6	–
X7	–
X8	–
X9	–

Sequential 100

Matrix	Epochen
X0	$\perp$
X1	$\perp$
X2	$\perp$
X3	$\perp$
X4	$\perp$
X5	–
X6	–
X7	–
X8	–
X9	–

Das Verfahren von Silva & Almeida ist dem Verfahren von Jacobs sehr ähnlich. Das spiegelt sich hier auch in den Ergebnissen wieder, die sich nicht wesentlich von denen des Verfahrens von Jacobs unterscheiden.

Das von mir entwickelte Verfahren ist eine Modifikation des Verfahrens von Chan & Fallside. Beträgt der Winkel zwischen dem vorangegangenen Lernschritt und dem aktuellen Gradienten 0°, nimmt die Lernrate den vorher eingestellten Maximalwert an, bei einem Winkel von 180° den Minimalwert. Bei einem Winkel von 90° oder 270° nimmt die Lernrate den Wert in der Mitte zwischen Minimal- und Maximalwert an; bei anderen Winkeln entsprechende Zwischenwerte gemäß einer Kosinusfunktion.

Eigenes Verfahren / Batch1

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	α_0	η_{min}	η_{max}	Epochen	α_0	η_{min}	η_{max}	Epochen
X0	0.9	0.0025	0.5	$\perp$	0.95	0.1	2.0	1058
X1				$\perp$				832
X2				$\perp$				$\perp$
X3				4717				1506
X4				4587				1114
X5				$\perp$				$\perp$
X6				3959				937
X7				$\perp$				$\perp$
X8				2774				675
X9				$\perp$				$\perp$

Die Ergebnisse dieses Verfahrens mit seinen Standardparametern und mit leicht erhöhten Werten waren wesentlich schlechter als beim Standardverfahren mit Batch1 Learning.

Eigenes Verfahren / Batch1

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	α_0	η_{min}	η_{max}	Epochen	α_0	η_{min}	η_{max}	Epochen
X0	0.95	0.5	5.0	133	0.95	1.0	5.0	125
X1				115				112
X2				176				160
X3				146				172
X4				140				131
X5				113				110
X6				115				111
X7				$\perp$				$\perp$
X8				148				140
X9				111				109

Eine weitere Erhöhung der Parameter brachte jedoch (ohne aufwendige Suche wie beim Standardverfahren) sehr gute Ergebnisse mit niedrigen Schwankungen, die zwar vereinzelt vom Standardverfahren mit Batch1 Learning unterboten wurden, dessen Ergebnisse jedoch nicht so gleichmäßig gut waren.

Mit den Parametern $\alpha_0 = 0.95$, $\eta_{min} = 0.5$ und $\eta_{max} = 10.0$ konvergierte das Verfahren mit dem "Sum-of-Derivatives Batch Learning" für keine der zehn Initialisierungsmatrizen X0–X9. ($\perp$)

Eigenes Verfahren
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

				Sequential 100		Dynamic	
Matrix	α_0	η_{min}	η_{max}	Epochen	Updates	Epochen	Updates
X0	0.9	0.0025	0.5	28	2,774.	469	1,776.
X1				27	3,518.	474	1,775.
X2				39	3,641.	512	1,916.
X3				31	3,044.	*518	1,891.
X4				31	4,074.	446	1,642.
X5				30	3,830.	*503	1,810.
X6				$\perp$	–	*485	1,815.
X7				29	3,755.	662	2,436.
X8				27	3,560.	728	2,667.
X9				27	3,357.	504	1,855.

"*" : Beim ersten Versuch konvergierte das Verfahren nicht.
Angegeben sind die Ergebnisse des zweiten Versuchs.

Die Ergebnisse mit Sequential Learning entsprechen in etwa denen des Standardverfahrens mit $\eta = \alpha = 0.5$, was dort bereits gut, aber noch nicht optimal war.

Die Ergebnisse des Dynamic Learning lagen in etwa in demselben Bereich wie beim Standardverfahren.

Eigenes Verfahren
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

				Sequential 100		Dynamic	
Matrix	α_0	η_{min}	η_{max}	Epochen	Updates	Epochen	Updates
X0	0.95	0.1	2.0	26	785.	$\perp$	–
X1				21	912.	419	1,624.
X2				21	889.	*228	866.
X3				22	923.	*127	437.
X4				22	930.	137	487.
X5				26	944.	99	336.
X6				19	741.	*143	496.
X7				24	1,056.	117	423.
X8				21	868.	$\perp$	–
X9				19	778.	101	361.

"*" : Beim ersten Versuch konvergierte das Verfahren nicht.
Angegeben sind die Ergebnisse des zweiten Versuchs.

Die Ergebnisse des Sequential Learning mit etwas erhöhten Parametern entsprechen den guten Ergebnissen des Standardverfahrens mit $\eta = 2.0$ und $\alpha = 0.5$. Mit dem Standardverfahren wurden zwar noch bessere Resultate erzielt, jedoch nur mit extrem großer Lernrate, die das Ergebnis vieler vorheriger Versuche war.

Beim Dynamic Learning sind die Ergebnisse zum Teil erheblich besser als beim Standardverfahren. Dies kann jedoch nicht als gültiger Vergleich angesehen werden, da beim Standardverfahren zu wenige Parameterkombinationen getestet wurden und die dort erzielten

Ergebnisse nicht unbedingt repräsentativ sind.

Insgesamt läßt sich feststellen, daß die erzielten Trainingszeiten im Vergleich mit dem Standardverfahren zwar nicht optimal waren, dafür aber ohne aufwendige Suche geeigneter Parameter zustande kamen, was meines Erachtens bereits ein großer Vorteil ist.

Eigenes Verfahren / Batch2

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	α_0	η_{min}	η_{max}	α_0	η_{min}	η_{max}	α_0	η_{min}	η_{max}	Epochen
X0	0.9	0.0025	0.5	0.95	0.1	2.0	0.95	0.5	5.0	$\perp$
X1										$\perp$
X2										$\perp$
X3										$\perp$
X4										$\perp$
X5										$\perp$
X6										–
X7										–
X8										–
X9										–

Das Versagen dieses Verfahrens beim Batch2 Learning ist nach der Bemerkung zum Verfahren von Chan & Fallside relativ plausibel, da die Anpassung des Trägheitsfaktors bei beiden Verfahren identisch ist, diese aber durch das Batch2 Learning unterlaufen wird.

Eigenes Verfahren / Periodical

$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	α_0	η_{min}	η_{max}	Epochen
X0	0.9	0.0025	0.5	545
X1				695
X2				$\perp$
X3				$\perp$
X4				1048
X5				1678
X6				673
X7				$\perp$
X8				804
X9				360

Eigenes Verfahren / Periodical
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$

Matrix	α_0	η_{min}	η_{max}	Epochen	α_0	η_{min}	η_{max}	Epochen
X0	0.95	0.1	2.0	$\perp$	0.95	0.5	5.0	$\perp$
X1				$\perp$				$\perp$
X2				$\perp$				$\perp$
X3				$\perp$				$\perp$
X4				366				$\perp$
X5				$\perp$				137
X6				91				280
X7				$\perp$				$\perp$
X8				210				90
X9				93				$\perp$

Im Unterschied zum Standardverfahren, wo sich mittels Periodical Learning nach einigen Versuchen sehr gute Ergebnisse erzielen ließen, scheint sich dieses Verfahren nicht für Periodical Learning zu eignen, wie das extrem häufige Steckenbleiben zeigt.

Als Ergebnis der bisherigen Simulationen läßt sich festhalten, daß mit optimalen Parametern mit Hilfe des Standardverfahrens und des Periodical oder Batch1 Learning, in geringerem Maße auch mit dem Sequential Learning, die besten Ergebnisse erzielt wurden.

Die speziellen Lernregeln erzielten teilweise gleichwertige oder geringfügig schlechtere Ergebnisse, jedoch verbunden mit dem Vorteil, keine aufwendige Parametersuche zu benötigen, teilweise versagten sie auch ganz.

Es ließ sich ebenfalls feststellen, daß sich nicht jede Lernregel (das Standardverfahren eingeschlossen) mit jedem Lernmodus gleich gut kombinieren läßt. Vielmehr gab es für jede Lernregel ein oder zwei Lernmodi, mit denen sich die besten Ergebnisse erzielen ließen.

Das bisher gesagte gilt (bis jetzt) jedoch nur für das XOR-Problem und die Verwendung der logistischen Aktivierungsfunktion.

Die Verwendung der symmetrischen Aktivierungsfunktion soll jedoch noch in einigen wenigen stichprobenartigen Versuchen (mit den bisher erfolgreichsten Verfahren) untersucht werden.

Man beachte, daß bei meiner Implementierung dieser Funktion die Möglichkeit besteht, die zu lernenden Testdaten, deren Werte sämtlich im Bereich $[0, 1]$ liegen, entweder in diesem Bereich zu belassen oder aber sie in den Bereich $[-1, +1]$ zu transformieren. (Das Belassen der Daten im ursprünglichen Bereich führt manchmal zu größeren Steigerungen der Lerngeschwindigkeit)

Vorversuch
Standard / Periodical
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$
Symmetrische Aktivierungsfunktion
Daten konvertiert

Matrix	η	α	Epochen
X9	0.5	0.9	$\perp$
X9	0.9	0.9	$\perp$
$r = 0.5$			$\perp$
X9	5.0	0.95	$\perp$
$r = 0.5$			$\perp$

"$r = 0.5$" : Zufallsmatrix mit Werten zwischen -0.5 und $+0.5$.

(Die Verwendung eines größeren Initialisierungsbereichs von $r = 0.5$ statt $r = 0.2$ führte teilweise bei anderen Versuchen mit dem XOR-Problem zu besserer Konvergenz)

Vorversuch
Standard / Periodical
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$
Symmetrische Aktivierungsfunktion
Daten *nicht* konvertiert

Matrix	η	α	Epochen
X9	0.9	0.9	$\perp$
X9	2.0	0.9	$\perp$

Das bisher relativ erfolgreiche Standardverfahren scheint hier völlig zu versagen.

Schmidhuber
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$
Symmetrische Aktivierungsfunktion
Daten konvertiert

Sequential 100

Matrix	Epochen	Updates
X0	25	185.
X1	$\perp$	–
X2	20	147.
X3	10	81.
X4	14	112.
X5	12	95.
X6	13	100.
X7	8	58.
X8	>500	–
X9	14	115.

Dynamic

Matrix	Epochen	Updates
X0	111	405.
X1	48	183.
X2	52	185.
X3	68	254.
X4	30	102.
X5	*38	129.
X6	38	133.
X7	32	112.
X8	41	147.
X9	30	103.

"*" : Beim ersten Versuch konvergierte das Verfahren nicht (>1000).
Angegeben sind die Ergebnisse des zweiten Versuchs.

Der Vergleich mit den Ergebnissen des gleichen Versuchs mit logistischer Aktivierungsfunktion (siehe oben) zeigt einen erheblichen Geschwindigkeitszuwachs bei Verwendung der symmetrischen Aktivierungsfunktion.

Eigenes Verfahren
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$
$\alpha_0 = 0.9 \qquad \eta_{min} = 0.0025 \qquad \eta_{max} = 0.5$
Symmetrische Aktivierungsfunktion
Daten konvertiert

Sequential 100

Matrix	Epochen	Updates
X0	8	243.
X1	20	527.
X2	16	421.
X3	12	356.
X4	8	263.
X5	13	375.
X6	9	297.
X7	12	368.
X8	12	404.
X9	$\perp$	–

Dynamic

Matrix	Epochen	Updates
X0	57	202.
X1	61	225.
X2	74	257.
X3	82	297.
X4	63	237.
X5	64	214.
X6	73	268.
X7	85	323.
X8	48	171.
X9	64	234

Auch bei meinem Verfahren zeigt sich eine vielfache Geschwindigkeitssteigerung gegenüber dem gleichen Versuch mit logistischer Aktivierungsfunktion.

Eigenes Verfahren
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$
$\alpha_0 = 0.95 \qquad \eta_{min} = 0.1 \qquad \eta_{max} = 2.0$
Symmetrische Aktivierungsfunktion
Daten konvertiert

Sequential 100

Matrix	Epochen	Updates
X0	5	87.
X1	$\perp$	–
X2	10	214.
X3	6	114.
X4	$\perp$	–
X5	28	429.
X6	228	4429.
X7	5	72.
X8	8	119.
X9	$\perp$	–

Dynamic

Matrix	Epochen	Updates
X0	*$\perp$	–
X1	12	42.
X2	13	48.
X3	*18	66.
X4	13	48.
X5	*175	657.
X6	26	97.
X7	16	55.
X8	*24	88.
X9	12	40.

"*" : Beim ersten Versuch konvergierte das Verfahren nicht.
Angegeben sind die Ergebnisse des zweiten Versuchs.

Während die Ergebnisse beim Dynamic Learning auch hier wesentlich besser sind als bei Verwendung der logistischen Aktivierungsfunktion, zeigen sich beim Sequential Learning schon erste „Ausreißer“, wo das Verfahren steckenblieb oder unverhältnismäßig lange brauchte, um zu konvergieren.

Eine bereits erwähnte Variante besteht darin, die Daten *nicht* auf den Bereich $[-1, +1]$ zu transformieren:

Schmidhuber / Sequential 100
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$
Symmetrische Aktivierungsfunktion
Daten *nicht* konvertiert

Matrix	Epochen	Updates
X0	$\perp$	–
X1	64	475.
X2	79	582.
X3	$\perp$	–
X4	59	459.
X5	63	498.
X6	74	563.
X7	72	539.
X8	$\perp$	–
X9	65	494.

Eigenes Verfahren / Sequential 100
$\epsilon_{abs} = 0.1 \qquad \epsilon_{pat} = 0.1$
$\alpha_0 = 0.9 \qquad \eta_{min} = 0.0025 \qquad \eta_{max} = 0.5$
Symmetrische Aktivierungsfunktion
Daten *nicht* konvertiert

Matrix	Epochen	Updates
X0	34	921.
X1	30	770.
X2	30	756.
X3	31	809.
X4	34	894.
X5	30	784.
X6	36	970.
X7	31	821.
X8	32	888.
X9	33	869.

Die so erzielten Ergebnisse sind jedoch wesentlich schlechter als mit den transformierten Daten.

5.2 Die Rotkäppchen-Simulation

Ein weiteres sehr kleines Problem ist das Rotkäppchen-Problem, welches nur aus drei Patterns besteht und sich deshalb für eine umfangreichere Untersuchung anbietet, obwohl die Gefahr besteht, daß die daran gewonnenen Ergebnisse nicht repräsentativ sind für anders geartete und/oder größere Probleme.

Der Vorteil dieses Problems besteht jedoch darin, daß es sich als „Gegengewicht“ zum XOR-Problem eignet: Während das XOR-Problem eine Klassifikationsaufgabe ist, geht es beim Rotkäppchen-Problem um Pattern-Assoziation.

Während das XOR-Problem extrem nicht-linear separierbar ist, ähnliche Patterns entgegengesetzte Ausgaben hervorrufen müssen, ist das Rotkäppchen-Problem sogar linear separierbar.

Mit einer Fehlerschranke von $\epsilon_{abs} = \epsilon_{pat} = 0.1000001$ konvergierte das Perceptron mit $\eta = 0.5$ in nur 2 Epochen und benötigte dabei nur 4 Updates.

Die seltsame Fehlerschranke hängt damit zusammen, daß das Perceptron nur die Ausgabewerte 0.0 und 1.0 erzeugen kann, die Targetwerte des Datensatzes des Rotkäppchen-Problems aber aus den Werten 0.1 und 0.9 bestehen.

Damit die Fehler kleiner als die Fehlerschranke werden können und somit das Verfahren konvergieren kann, muß die Fehlerschranke geringfügig größer sein als der Mindestabstand zwischen Target- und Ausgabewerten, der 0.1 beträgt.

Obwohl das Problem also eigentlich trivial ist, ist trotzdem interessant, wie die verschiedenen Verfahren im Vergleich mit dem XOR-Problem abschneiden.

In den folgenden Simulationen wurden im Unterschied zum XOR-Problem die Fehlerschranken $\epsilon_{abs} = 0.01$ und $\epsilon_{pat} = 0.01$ verwendet. Das hat damit zu tun, daß beim XOR-Problem (mit den Targetwerten 0.0 und 1.0) aufgrund der Fehlerschranken $\epsilon_{abs} = \epsilon_{pat} = 0.1$ die beim ausgelernten Netzwerk tatsächlich erzeugten Ausgabewerte (etwa) 0.1 und 0.9 betrugen. Da hier bereits die Targetwerte 0.1 und 0.9 betragen, ist eine etwas geringere Fehlertoleranz statthaft, ohne die Vergleichbarkeit der Ergebnisse zu beeinträchtigen.

Die Versuche sind analog zu denen des XOR-Problems konzipiert.

Vor Versuchsbeginn wurden zehn zufällig initialisierte Matrizen mit Werten im Bereich $[-0.2\,, +0.2]$ erzeugt, die im Folgenden mit R0 bis R9 bezeichnet werden. Vor jedem Versuch wurde eine dieser Matrizen eingelesen, wiederum mit dem Zweck, den unkontrollierbaren Einfluß der zufälligen Initialisierung so weit wie möglich auszuschalten.

Die Lernregeln wurden ebenfalls in der zu Anfang des Kapitels beschriebenen Standardreihenfolge untersucht; wobei zusätzlich die Lernmodi in wechselnder Reihenfolge variiert wurden.

In allen Simulationen wurden drei hidden Units verwendet. Da mehrere gleichzeitige Aktivierungen der Outputunits möglich waren, wurde der "[S]everal output activations penalty term" abgeschaltet.

Begonnen wurde wieder mit dem Standardverfahren und Periodical Learning.

Vorversuch
Standard / Periodical
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen
R0	0.01	0.0	>5000
R0	0.1	0.0	3778
R0	0.2	0.0	1888

Der Vorversuch zeigt auch hier, daß Lernraten kleiner als 0.2 unpraktikabel sind, wenn auch das Verfahren hier nicht steckenbleibt wie im Falle des XOR-Problems, was sicherlich mit der linearen Separierbarkeit dieses Problems zusammenhängt.

Standard / Periodical
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen
R0	0.5	0.0	755	0.9	0.0	419
R1			584			339
R2			770			441
R3			772			431
R4			538			296
R5			674			357
R6			521			287
R7			642			344
R8			742			410
R9			758			423

Zwischenversuch
Standard / Periodical
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	1.0	0.0	377	3.0	0.0	126	20.0	0.0	227
R6			259			97			91
R0	1.2	0.0	314	5.0	0.0	76	25.0	0.0	239
R6			216			106			$\perp$
R0	1.5	0.0	251	10.0	0.0	66	30.0	0.0	$\perp$
R6			173			58			$\perp$
R0	2.0	0.0	188	15.0	0.0	41			
R6			132			57			

Standard / Periodical
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen
R0	15.0	0.0	41
R1			41
R2			47
R3			51
R4			84
R5			41
R6			57
R7			33
R8			47
R9			50

Der Vergleich mit den entsprechenden Versuchen des XOR-Problems zeigt die gleichen Effekte: Ausgehend von kleinen Lernraten nehmen mit zunehmender Lernrate die benötigten Trainingszeiten immer weiter ab, bis sie (bei annähernd der gleichen Größenordnung der Lernrate) ihr Minimum erreichen. Danach steigen die Trainingszeiten mit der Lernrate wieder steil an.

Die benötigten Trainingszeiten sind lediglich insgesamt kürzer, was durch die unterschiedliche Komplexität der beiden Probleme erklärt wird.

Standard / Periodical
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	0.5	0.5	377	0.5	0.9	66	0.5	0.95	92
R1			285			60			71
R2			387			92			138
R3			394			95			108
R4			269			55			111
R5			338			97			136
R6			261			67			221
R7			321			74			104
R8			371			65			93
R9			377			65			95

Wie beim XOR-Problem sinken zunächst die Trainingszeiten mit zunehmendem Trägheitsfaktor. Ab einem bestimmten Optimum steigen die Trainingszeiten wieder an; hier jedoch schon viel früher als dort.

Standard / Periodical
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	0.9	0.5	210	0.9	0.9	28	0.9	0.95	143
R1			164			41			58
R2			224			63			160
R3			226			60			124
R4			148			44			270
R5			179			69			77
R6			144			63			195
R7			163			47			168
R8			206			29			103
R9			210			34			98

Eine Erhöhung der Lernrate bringt noch bessere Resultate. Auch hier wieder ist ein Abfall nach zu großer Erhöhung des Trägheitsfaktors zu beobachten.

Standard / Periodical
$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen
R0	1.2	0.5	157	1.2	0.9	37
R1			126			38
R2			174			60
R3			176			51
R4			109			49
R5			130			49
R6			110			61
R7			116			41
R8			153			40
R9			158			45

Erhöht man die Lernrate weiter, setzt bereits ein Rückgang der Konvergenzgeschwindigkeit ein. Steigert man jedoch zusätzlich den Trägheitsfaktor, werden wiederum sehr gute Ergebnisse erzielt, ein Phänomen, das beim XOR-Problem nicht beobachtet wurde mangels eines entsprechenden Versuchs.

Standard / Periodical
$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen
R0	0.7	0.9	38	0.8	0.9	33
R1			48			44
R2			72			68
R3			70			64
R4			45			43
R5			79			73
R6			91			69
R7			55			49
R8			41			35
R9			40			93

„Herantasten“ von unten an die bisher beste Kombination mit $\eta = \alpha = 0.9$ zeigt mit zwei Ausnahmen (R4, R9) einen äußerst glatten Verlauf.

Standard / Periodical

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	1.0	0.9	23	1.1	0.9	40	2.0	0.9	38
R1			41			42			36
R2			60			59			50
R3			57			56			45
R4			45			46			109
R5			62			55			38
R6			62			61			69
R7			48			45			44
R8			25			39			43
R9			34			44			55

Die weitere (geringfügige) Steigerung der Lernrate ergibt bei einigen Matrizen bessere, bei einigen gleichbleibende und bei anderen auch schlechtere Werte, ebenso nach einem größeren Sprung der Lernrate auf fast das Doppelte. Der Verlauf ist jedoch nicht mehr glatt; mal hat eine Matrix einen besseren, dann wieder einen schlechteren und schließlich einen noch besseren Wert, oder umgekehrt.

Standard / Batch1

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	0.2	0.9	155	0.3	0.9	72	0.5	0.9	121
R1			154			126			102
R2			224			184			184
R3			206			161			165
R4			305			144			112
R5			252			205			186
R6			127			139			147
R7			269			222			188
R8			156			86			121
R9			155			89			124

Auch beim Batch1 Learning führt die Vergrößerung der Lernrate im Allgemeinen zu einer Geschwindigkeitssteigerung, im Einzelfall ist diese Steigerung jedoch nicht immer monoton.

Standard / Batch1
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	0.7	0.9	129	0.9	0.9	133	1.0	0.9	136
R1			103			111			127
R2			240			184			211
R3			169			173			175
R4			136			147			165
R5			182			188			191
R6			179			243			285
R7			177			183			193
R8			128			133			136
R9			130			132			134

Im vorangehenden Versuch tritt sogar die Umgekehrung dieser Tendenz auf: Mit steigender Lernrate werden die Ergebnisse (von R2 abgesehen) schlechter.

Standard / Batch1
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	0.5	0.7	235	0.7	0.7	166	0.9	0.7	127
R1			177			129			101
R2			246			177			139
R3			243			174			135
R4			193			162			150
R5			247			182			144
R6			161			114			87
R7			235			175			142
R8			232			164			125
R9			236			166			127

Bei kleinerem Trägheitsfaktor erfolgt wiederum eine Verkürzung der Trainingszeiten bei einer Zunahme der Lernrate.

Standard / Batch1
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	1.0	0.5	198	2.0	0.0	200	2.0	0.1	180
R1			146			144			130
R2			204			203			183
R3			202			201			181
R4			152			144			131
R5			200			190			173
R6			135			135			121
R7			188			179			163
R8			195			197			177
R9			198			200			180

Wie beim XOR-Problem ist aufgrund der Gleichheit der Anzahl der Epochen und der Anzahl der Updates beim Batch1 Learning die Anzahl der Epochen generell größer als beim Periodical Learning. Aber selbst wenn man annimmt, daß beim Periodical Learning pro Epoche drei Updates (für jedes Pattern eines) erfolgten (was das Maximum darstellt und sicher zu hoch geschätzt ist), sind die hier mit Batch1 Learning erzielten Ergebnisse in der Regel schlechter.

Standard / Batch1
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	η	α	Epochen	η	α	Epochen
R0	2.0	0.3	139	2.0	0.5	97	2.0	0.7	47
R1			102			76			49
R2			143			104			69
R3			141			101			62
R4			107			100			70
R5			139			104			65
R6			94			67			38
R7			131			101			79
R8			137			96			47
R9			139			98			47
R0	2.0	0.9	643						
R9			644						

Erst bei diesem Versuch (mit $\eta = 2.0$ und $\alpha = 0.7$) konnte ein echter Durchbruch erzielt werden. Zwar ist die Anzahl der Epochen noch etwa anderthalbmal so groß wie beim besten Ergebnis mit Periodical Learning (mit $\eta = \alpha = 0.9$), unter der Berücksichtigung des Umstandes, daß das Periodical Learning jedoch bis zu drei Updates pro Epoche benötigen kann, kann man davon ausgehen, daß diese Ergebnisse jenen überlegen sind.

Eine Erhöhung des Trägheitsfaktors über den optimalen Wert hinaus hat massive Geschwindigkeitseinbußen zur Folge.

Standard / Batch1
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen	Matrix	η	α	Epochen
R0	5.0	0.0	80				
R0	10.0	0.0	39				
R0	15.0	0.0	35	R5	15.0	0.0	44
R1			34	R6			55
R2			55	R7			116
R3			124	R8			89
R4			34	R9			46
R0	20.0	0.0	204				

Wie beim Periodical Learning ließen sich auch hier durch das Weglassen des Trägheitsterms (durch das auf Null Setzen des Trägheitsfaktors) und (zum Ausgleich) durch eine extreme Steigerung der Lernrate ausgezeichnete Ergebnisse erzielen. Diese waren aber nicht wesentlich besser als die besten Ergebnisse mit Trägheitsterm (siehe Versuch davor) und vor allem nicht so gleichmäßig gut.

Standard / Dynamic
$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

Matrix	η	α	Epochen 1. Vers.	Epochen 2. Vers.	η	α	Epochen 1. Vers.	Epochen 2. Vers.
R0	0.5	0.9	62	60	1.0	0.9	73	77
R1			64	69			161	65
R2			85	111			231	241
R3			186	85			350	178
R4			100	114			52	233
R5			72	65			193	99
R6			152	66			762	144
R7			80	57			72	157
R8			98	138			84	200
R9			59	73			84	263

Der erste Versuch mit Dynamic Learning zeigt das gleiche Phänomen wie beim Batch1 Learning: Bei einem Trägheitsfaktor von 0.9 trat im Allgemeinen eine Verschlechterung der Ergebnisse mit zunehmender Lernrate auf, was durch den ersten Teil des folgenden Versuchs noch einmal unterstrichen wird.

Standard / Dynamic
$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

Matrix	η	α	Epochen 1. Vers.	Epochen 2. Vers.	η	α	Epochen 1. Vers.	Epochen 2. Vers.
R0	2.0	0.9	368	1539	2.0	0.7	46	48
R1			>2000	>2000			61	52
R2			–	–			87	52
R3			–	–			86	60
R4			–	–			48	51
R5			–	–			47	51
R6			–	–			73	44
R7			–	–			52	40
R8			–	–			59	49
R9			–	–			54	46

Die für das Batch1 Learning optimalen Parameterwerte erbrachten auch beim Dynamic Learning ausgezeichnete Ergebnisse, die sich ziemlich genau in derselben Größenordnung bewegten.

Standard / Dynamic
$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

Matrix	η	α	Epochen 1. Vers.	Epochen 2. Vers.
R0	15.0	0.0	104	43
R1			35	31
R2			81	91
R3			27	53
R4			21	294
R5			95	23
R6			74	113
R7			141	35
R8			39	52
R9			60	70

Der beim Batch1 Learning beste Wert für die Lernrate unter Weglassung des Trägheitsterms erbrachte auch beim Dynamic Learning sehr gute Ergebnisse, einige Werte waren besser als bei den besten Ergebnissen mit Trägheitsterm (siehe vorherigen Versuch), dafür waren viele Ergebnisse erheblich schlechter und insgesamt die Streuung erheblich größer als mit Trägheitsterm.

Schmidhuber
$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

	Periodical	Batch1	Batch2
Matrix	Epochen	Epochen	Epochen
R0	419	445	133
R1	339	318	111
R2	441	449	184
R3	431	447	173
R4	296	315	147
R5	357	416	188
R6	287	299	243
R7	344	391	183
R8	410	439	133
R9	423	443	132

Schmidhuber
$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

	Sequential 100		Dynamic	
Matrix	Epochen	Updates	Epochen 1. Vers.	Epochen 2. Vers.
R0	25	3,593.	367	375
R1	14	2,078.	368	298
R2	35	5,306.	400	407
R3	14	1,949.	413	801
R4	17	2,814.	297	306
R5	22	2,398.	373	339
R6	17	2,580.	423	366
R7	19	2,304.	281	329
R8	16	2,423.	382	385
R9	14	1,818.	378	392

Das Verfahren von Schmidhuber erzielte hier im Vergleich zum Standardverfahren nur einigermaßen akzeptable Resultate. Es ist jedoch zu bedenken, daß die Ergebnisse mittels des Standardverfahrens erst nach einer aufwendigen Parametersuche erzielt wurden, letztendlich also das Verfahren von Schmidhuber günstiger ist.

Chan&Fallside

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

$\eta_{min} = 0.0001 \qquad \eta_{max} = 20.0$

	Periodical	Batch1	Sequential 100	
Matrix	Epochen	Epochen	Epochen	Updates
R0	$\perp$	21	66	7,255.
R1	$\perp$	21	>100	>10,015.
R2	$\perp$	31	>100	>10,058.
R3	$\perp$	29	>100	>10,200.
R4	$\perp$	45	>100	>9,741.
R5	$\perp$	32	>100	>8,468.
R6	–	24	62	14,606.
R7	–	34	>100	>9,964.
R8	–	35	>100	>9,196.
R9	–	32	>100	>9,988.
$\eta_{min} = 0.1 \qquad \eta_{max} = 2.0$				
R0	648	108	8	596.
$\eta_{min} = 0.5 \qquad \eta_{max} = 5.0$				
R0	153	46	11	482.

Wie beim XOR-Problem erzielte das Verfahren von Chan & Fallside nur beim Batch1 Learning (im Übrigen ausgezeichnete) Ergebnisse. Zwar hätte auch das Sequential Learning bei einer höheren Maximalzahl von Epochen vermutlich noch bei einigen Matrizen zur Konvergenz geführt, diese Ergebnisse wären jedoch nutzlos gewesen angesichts der unverhältnismäßig hohen Anzahl der benötigten Updates.

Die Ergebnisse dieses Verfahrens mit Batch1 Learning sind jedoch sogar noch viel besser als die mittels Standardverfahren erzielten.

Das Beschränken der Lernrate in einem kleineren als dem standardmäßigen Bereich (siehe Standard-Parameter zu Anfang dieses Kapitels) brachte bei den anderen Lernmodi eine gewaltige Verbesserung der Ergebnisse, beim Batch1 Learning jedoch eine Verschlechterung.

Fahlmans QuickProp / Batch1

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

ohne atanh-Fehler
ohne split-eta

Matrix	Epochen
X0	29
X1	19
X2	35
X3	30
X4	30
X5	38
X6	22
X7	23
X8	27
X9	33

ohne atanh-Fehler
mit split-eta

Matrix	Epochen
X0	31
X1	30
X2	47
X3	41
X4	38
X5	28
X6	33
X7	38
X8	33
X9	49

Fahlmans QuickProp / Batch1

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

mit atanh-Fehler
ohne split-eta

Matrix	Epochen
X0	26
X1	18
X2	30
X3	27
X4	19
X5	27
X6	23
X7	27
X8	29
X9	28

mit atanh-Fehler
mit split-eta

Matrix	Epochen
X0	28
X1	21
X2	33
X3	34
X4	39
X5	26
X6	30
X7	32
X8	28
X9	34

Die mit Fahlmans QuickProp-Verfahren erzielten Ergebnisse sind vergleichsweise sehr gut, besser als die mit dem Standardverfahren (und Batch1 Learning) erzielten. Dazu kommt, daß dazu kein aufwendiges “parameter tuning” erforderlich war.

Die besten Ergebnisse wurden mit hyperbolischer Fehlerfunktion, jedoch ohne aufsplitten der Lernrate erzielt.

Jacobs Delta-bar-delta-rule

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

	Periodical		Batch1
Matrix	Epochen	Updates	Epochen
R0	85	215.	116
R1	59	152.	85
R2	103	284.	115
R3	118	306.	96
R4	62	167.	122
R5	134	317.	184
R6	60	169.	115
R7	115	296.	99
R8	87	224.	98
R9	99	275.	112

Jacobs Delta-bar-delta-rule

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

	Sequential 100		Dynamic	
Matrix	Epochen	Updates	Epochen 1. Vers.	Epochen 2. Vers.
R0	>100	>29,908.	79	115
R1	>100	>29,908.	$\perp$	114
R2	–	–	103	*120
R3	–	–	118	51
R4	–	–	115	132
R5	–	–	153	$\perp$
R6	–	–	*93	167
R7	–	–	123	97
R8	–	–	78	61
R9	–	–	93	220

"*" : Das Verfahren hatte nach 500 Epochen das Abbruchkriterium noch nicht erfüllt und wurde neu gestartet.

Die Delta-bar-delta-Regel von Jacobs erzielte im Vergleich zum Standardverfahren mäßig gute Resultate. Wie beim Verfahren von Schmidhuber war dazu jedoch keine aufwendige Parametersuche erforderlich (wobei das Verfahren jedoch eigene Parameter besitzt, so daß möglicherweise eine geeignetere Wahl bessere Ergebnisse zur Folge hätte). Im Vergleich zum Verfahren von Schmidhuber waren die benötigten Trainingszeiten sogar besser.

Silva&Almeida

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

	Periodical	Batch1
Matrix	Epochen	Epochen
R0	$\perp$	$\perp$
R1	$\perp$	$\perp$
R2	$\perp$	$\perp$
R3	$\perp$	$\perp$
R4	$\perp$	$\perp$
R5	$\perp$	$\perp$
R6	–	–
R7	–	–
R8	–	–
R9	–	–

Nicht ganz verständlich ist, warum hier das Verfahren von Silva & Almeida vollständig versagte, obwohl es dem Verfahren von Jacobs sehr ähnelt.

Auch mit Hilfe der "Cancel Step"-Option (die eine Gewichtsänderung rückgängig macht, falls sich durch diese der Gesamtfehler verschlechtert hat) konvergierte das Verfahren für keine der beiden Initialisierungsmatrizen R0 und R1 (und blieb stecken).

Diese Option ist nach den bisherigen Erfahrungen jedoch ohnehin sinnlos, da z.B. beim XOR-Problem bei *allen* Versuchen der Fehler zuerst stark anstieg, um dann auf nahezu Null abzufallen (vergleiche dazu die Abbildungen im Anhang A). Da die Überwindung dieses „Aktivierungsenergie-Berges" offensichtlich notwendig war, hätte die "Cancel Step"-Option die Konvergenz in allen Versuchen verhindert.

Eigenes Verfahren

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

$\alpha_0 = 0.9 \qquad \eta_{min} = 0.0025 \qquad \eta_{max} = 0.5$

	Periodical	Batch1	Batch2
Matrix	Epochen	Epochen	Epochen
R0	516	425	301
R1	343	305	297
R2	469	430	638
R3	509	428	298
R4	442	307	310
R5	417	408	290
R6	330	287	345
R7	450	384	881
R8	403	418	295
R9	447	423	1384

Eigenes Verfahren

$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

$\alpha_0 = 0.9$ $\eta_{min} = 0.0025$ $\eta_{max} = 0.5$

	Sequential 100		Dynamic	
Matrix	Epochen	Updates	Epochen 1. Vers.	Epochen 2. Vers.
R0	14	2,593.	359	359
R1	11	1,710.	357	276
R2	15	2,659.	398	379
R3	11	1,758.	>500	376
R4	13	1,907.	312	361
R5	12	1,729.	415	345
R6	12	2,068.	310	275
R7	16	2,179.	276	285
R8	12	1,855.	358	358
R9	12	1,811.	373	356

Das von mir modifizierte Verfahren von Chan & Fallside erzielte mit den Standardparametern im Vergleich zum Standardverfahren keine besonders guten Resultate. Die Ergebnisse beim Sequential Learning sind jedoch besser als beim Verfahren von Schmidhuber, bei den übrigen Lernmodi in etwa gleich gut. Dort wie hier waren erstaunlicherweise die Ergebnisse beim Batch2 Learning besser als beim Batch1 Learning. Erstaunlich deshalb, weil das Batch2 Learning die automatische Anpassung des Trägheitsfaktors durch mein Verfahren nahezu wirkungslos macht, und weil das Batch2 Learning mit meinem Verfahren beim XOR-Problem in keinem Fall konvergierte.

Eigenes Verfahren

$\epsilon_{abs} = 0.01$ $\epsilon_{pat} = 0.01$

$\alpha_0 = 0.95$ $\eta_{min} = 0.1$ $\eta_{max} = 2.0$

	Periodical	Batch1	Batch2
Matrix	Epochen	Epochen	Epochen
R0	122	105	>2000
R1	93	81	>2000
R2	143	111	–
R3	142	109	–
R4	91	110	–
R5	90	108	–
R6	90	73	–
R7	86	107	–
R8	89	104	–
R9	108	105	>2000

Eigenes Verfahren

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

$\alpha_0 = 0.95 \qquad \eta_{min} = 0.1 \qquad \eta_{max} = 2.0$

	Sequential 100		Dynamic	
Matrix	Epochen	Updates	Epochen 1. Vers.	Epochen 2. Vers.
R0	8	605.	84	79
R1	8	551.	89	109
R2	12	875.	83	85
R3	9	578.	88	88
R4	10	669.	97	79
R5	10	709.	89	98
R6	10	765.	125	86
R7	12	779.	72	89
R8	9	595.	83	79
R9	10	725.	73	74

Die Erhöhung der Schranken für die Lernrate hatte (wie beim XOR-Problem) eine zum Teil beträchtliche Verkürzung der Lernzeiten zur Folge. Beim Batch2 Learning erhöhten sich dagegen die Lernzeiten bis über die reichlich bemessene Maximalzahl von Epochen (2000) hinaus.

Eigenes Verfahren

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

$\alpha_0 = 0.95 \qquad \eta_{min} = 0.5 \qquad \eta_{max} = 5.0$

	Periodical	Batch1	Batch2
Matrix	Epochen	Epochen	Epochen
R0	48	45	>2000
R1	46	53	>2000
R2	51	51	–
R3	62	59	–
R4	55	52	–
R5	36	52	–
R6	62	41	–
R7	34	54	–
R8	38	44	–
R9	40	45	>2000

Eigenes Verfahren

$\epsilon_{abs} = 0.01 \qquad \epsilon_{pat} = 0.01$

$\alpha_0 = 0.95 \qquad \eta_{min} = 0.5 \qquad \eta_{max} = 5.0$

	Sequential 100		Dynamic	
Matrix	Epochen	Updates	Epochen 1. Vers.	Epochen 2. Vers.
R0	13	482.	38	31
R1	9	322.	60	62
R2	15	613.	54	102
R3	15	578.	>500	68
R4	12	459.	37	40
R5	19	841.	56	76
R6	15	585.	36	211
R7	13	503.	36	36
R8	10	383.	88	47
R9	10	430.	32	45

Die weitere Erhöhung der Schranken für die Lernrate hatte eine weitere, erhebliche Steigerung der Lerngeschwindigkeit zur Folge, wenn auch die Schwankungen beim Dynamic Learning etwas an Amplitude zunahmen.

Diese Ergebnisse waren (beim Batch1 Learning) fast so gut wie die des QuickProp-Verfahrens und des Verfahrens von Chan & Fallside und (in allen untersuchten Lernmodi außer dem Batch2 Learning) wesentlich besser als die des Verfahrens von Schmidhuber und der Delta-bar-delta-Regel von Jacobs. Im Unterschied zu diesen waren dazu jedoch drei Versuche mit unterschiedlichen Parameterwerten erforderlich.

Insgesamt bleibt festzuhalten, daß im Vergleich zum XOR-Problem die benötigten Lernzeiten trotz verringerter Fehlerschranke im Allgemeinen kürzer waren und ein Steckenbleiben nur sehr selten auftrat. Dies ist sicherlich auf den Umstand zurückzuführen, daß es sich hier um ein linear separierbares Problem handelt.

Weiter ergaben sich einige Unterschiede, z.B. was das Batch2 Learning in Kombination mit meinem eigenen Verfahren betrifft, das beim XOR-Problem in keinem Fall konvergierte (steckenblieb), beim Rotkäppchen-Problem jedoch teils akzeptable Lernzeiten, teils über 2000 Epochen benötigte.

Ein weiterer Unterschied trat beim Standardverfahren zutage, wo in einigen Fällen (bei zu großem Trägheitsfaktor) die Lernzeiten mit größer werdender Lernrate zu- statt abnahmen. Im Wesentlichen waren die beobachteten Trends jedoch gleich: Mit zunehmender Lernrate nahm auch die Lerngeschwindigkeit zu, bei Überschreitung eines optimalen Wertes nahm sie jedoch sehr schnell ab. Auch die Erhöhung des Trägheitsfaktors hatte bis zu einem Optimum eine beschleunigende Wirkung, danach setzte wiederum ein schneller Abfall ein.

Bei optimaler Wahl der Parameter ließen sich mit Hilfe des Standardverfahrens oft ausgezeichnete Resultate erzielen.

Das Verfahren von Schmidhuber erzielte mäßige Resultate, jedoch verbunden mit dem Vorteil, dafür keine aufwendige Parametersuche zu benötigen.

Beim Rotkäppchen-Problem war das Verfahren von Chan & Fallside erfolgreicher als beim

XOR-Problem. Während dort die Ergebnisse lediglich nicht schlecht waren, erzielte dieses Verfahren beim Rotkäppchen-Problem die besten Ergebnisse überhaupt.

Beim XOR-Problem erzielte das QuickProp-Verfahren von Fahlman zwar bei einigen Matrizen die kürzesten Lernzeiten, bei den meisten Matrizen blieb es jedoch stecken. Beim Rotkäppchen-Problem waren die Ergebnisse sehr gut.

Bei beiden Problemen ergab die Delta-bar-delta-Regel von Jacobs nur mäßige Resultate.

Gleiches galt für das Verfahren von Silva & Almeida beim XOR-Problem, während es beim Rotkäppchen-Problem sogar ganz versagte.

Das von mir entwickelte Verfahren ergab bei beiden Problemen bei günstiger Wahl der Schranken für die Lernrate gute Ergebnisse. Diese reichten zwar an die kürzesten gemessenen Lernzeiten nicht heran, erforderten dafür aber keine so ausgiebige Parametersuche. Beim XOR-Problem war das Verfahren von Schmidhuber besser, beim Rotkäppchen-Problem wesentlich schlechter.

Wie auch schon beim XOR-Problem festgestellt wurde, eigneten sich nicht alle Lernmodi für alle Verfahren (Lernregeln) gleich gut.

Während das Verfahren von Fahlman (QuickProp) von vornherein nur für das Batch1 Learning gedacht ist, konvergierten auch das Verfahren von Chan & Fallside, die Delta-bar-delta-Regel von Jacobs und das Verfahren von Silva & Almeida nur im Batch-Modus. Dies ist jedoch nicht weiter verwunderlich, da alle diese Verfahren den vorherigen Lernschritt und den augenblicklichen Gradienten zur Berechnung des nächsten Lernschrittes verwenden. Bei einem On-Line-Lernmodus sind die Schwankungen in der Richtung des Gradienten naturgemäß größer als bei Verwendung eines Mittelwerts (den die Summe der partiellen Ableitungen aller Patterns im Batch-Modus im Prinzip darstellt), was offensichtlich die Stabilität dieser Verfahren überfordert. (Die Lernraten nahmen häufig einen Extremwert an und blieben auf diesem stehen. Dadurch blieb dann auch das Netzwerk in einem lokalen Minimum stecken)

Das Standardverfahren, das Verfahren von Schmidhuber und mein eigenes konvergierten dagegen auch bei On-Line-Lernmodi wie Sequential, Periodical und Dynamic Learning.

5.3 Das Parity-Problem

Durch die geringe Größe des XOR-Problems, sowohl was die Anzahl der Gewichte des Netzes als auch was die Anzahl der zu lernenden Patterns betrifft, wurden möglicherweise die anhand dieses Problems gewonnenen Ergebnisse verzerrt.

Das Parity-Problem wurde deshalb ausgewählt, weil es ein dem XOR-Problem von der Art der Aufgabe her engstens verwandtes Problem darstellt, so daß man hoffen kann, daß eventuelle Unterschiede in den Versuchsergebnissen zwischen beiden Problemen vor allem auf deren unterschiedliche Größe zurückzuführen sind.

Dies muß nicht unbedingt der Fall sein, da aus der Sicht eines Netzwerkes die Verwandtschaft beider Probleme nicht unbedingt ersichtlich ist, d.h., für das Netz ist die Zuordnung der Patterns des Parity-Problems möglicherweise völlig willkürlich, so daß die Netzwerklösung dieses Problems keine Ähnlichkeit mit der des XOR-Problems mehr hat, oder mit anderen Worten, das Netzwerk speichert einfach die Patterns, ohne die zugrundeliegende Regel zu entdecken.

Da die Analyse eines Netzwerkes äußerst schwierig ist, wurde diese Frage jedoch nicht weiter untersucht.

Der Datensatz des Parity-Problems umfaßt 256 Patterns, d.h. alle Binärzahlen der Länge 8 Bit. Das Netzwerk bestand aus 8 Input-Units zur Aufnahme dieser Binärzahlen und aus einer Output-Unit, die die Parität der eingegebenen Binärzahl anzeigen sollte.

In allen folgenden Simulationen wurden 16 hidden Units verwendet.

Da nur eine einzige Output-Unit vorhanden war, wurde die "[S]everal output activations penalty term"-Option abgeschaltet.

Wie beim XOR-Problem wurden als Fehlerschranken die Werte $\epsilon_{abs} = 0.1$ und $\epsilon_{pat} = 0.1$ verwendet.

Es wurden zwei nahezu identische Versuchsserien durchgeführt, die sich durch die Verwendung der logistischen bzw. der symmetrischen Aktivierungsfunktion unterschieden.

Falls nicht anders angegeben, wurden die jeweiligen Standardparameter benutzt.

Bis auf das Sequential Learning wurden alle Versuche mit einer Maximalzahl von 10.000 Epochen gestartet. Beim Sequential Learning betrug die Maximalzahl 5.000 Epochen.

Verfahren	Epochen	Updates
Standard / Periodical $\eta = 0.9$ $\alpha = 0.9$	$\perp$	–
Schmidhuber / Sequential 100	$\perp$	–
Schmidhuber / Dynamic	$\perp$	–
Chan&Fallside / Batch1	$\perp$	–
QuickProp (+hyperr) / Batch1	$\perp$	–
Eigenes Verfahren / Batch1	$\perp$	–
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.1$ $\eta_{max} = 2.0$ / Batch1	$\perp$	–
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.5$ $\eta_{max} = 5.0$ / Batch1	$\perp$	–
Eigenes Verfahren / Dynamic	$\perp$	–
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.1$ $\eta_{max} = 2.0$ / Dynamic	$\perp$	–
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.5$ $\eta_{max} = 5.0$ / Dynamic	$\perp$	–
Gemischtes Verfahren $^{(*)}$	$\perp$	–

"$\perp$" : Das Verfahren blieb in einem lokalen Minimum stecken.

Das gemischte Verfahren bestand darin, zuerst 1.000 Epochen lang das QuickProp-Verfahren von Fahlman (mit hyperbolischem Fehler) laufen zu lassen, zur besten Matrix bis dahin zurückzukehren und ab da das Verfahren von Schmidhuber mit Dynamic Learning und einer Maximalzahl von 10.000 Epochen laufen zu lassen. (Dieses gemischte Verfahren hat sich bei früheren Untersuchungen bereits sehr bewährt)

Symmetrische Aktivierungsfunktion / Daten konvertiert

Verfahren	Epochen	Updates
Standard / Periodical $\eta = 0.9$ $\alpha = 0.9$	$\perp$	–
Schmidhuber / Sequential 100	$\perp$	–
Schmidhuber / Dynamic	$\perp$	–
Chan&Fallside / Batch1	$\perp$	–
QuickProp (+hyperr) / Batch1	$\perp$	–
Eigenes Verfahren / Batch1	$\perp$	–
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.1$ $\eta_{max} = 2.0$ / Batch1	$\perp$	–
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.5$ $\eta_{max} = 5.0$ / Batch1	$\perp$	–
Eigenes Verfahren / Dynamic	161	16,240.
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.1$ $\eta_{max} = 2.0$ / Dynamic	$\perp$	–
Eigenes Verfahren $\alpha_0 = 0.95$ $\eta_{min} = 0.5$ $\eta_{max} = 5.0$ / Dynamic	$\perp$	–
Gemischtes Verfahren	334	42,820.

Zur Sicherheit wurden die Versuche der beiden erfolgreichen Verfahren noch einmal wiederholt:

Symmetrische Aktivierungsfunktion / Daten konvertiert

Verfahren	Epochen	Updates	Epochen	Updates
	2. Versuch		3. Versuch	
Eigenes Verfahren / Dynamic	44	5,372.	29	3,893
Gemischtes Verfahren	$\perp$	–	–	–

Beim gemischten Verfahren zählen nicht die ganzen 1.000 Epochen des QuickProp-Verfahrens mit, sondern nur die bis zur besten Matrix bereits durchlaufenen Epochen und Updates sowie die ab da zusätzlich benötigten Epochen und Updates des Verfahrens von Schmidhuber.

Daß fast keines der untersuchten Verfahren konvergierte, mag zwar vereinzelt auf ungünstige Initialisierungen zurückzuführen sein, bei einem derart massiven Versagen muß jedoch eher von der Schwierigkeit des Problems ausgegangen werden, das bekanntermaßen hochgradig nicht linear separierbar ist.

Erstaunlich ist es jedoch, daß mein eigenes Verfahren in so extrem kurzer Zeit konvergierte. Wären in jeder Epoche 256 Updates aufgetreten (für jedes Pattern ein Update), hätte das Verfahren im ersten Versuch 41.216 Updates benötigen müssen. Tatsächlich hat es weniger als die Hälfte davon gebraucht.

Das gemischte Verfahren hat ebenfalls nur rund halb so viele Updates benötigt, wie maximal bei der gegebenen Zahl von Epochen möglich gewesen wären.

Eine befriedigende Erklärung dieses Phänomens ist zur Zeit leider noch nicht möglich.

5.4 Das Primzahlen-Problem

Das Primzahlen-Problem war als Test gedacht, wie ein Netzwerk auf eine (für ein Netzwerk) vollkommen willkürliche Klassifizierung der binär kodierten Zahlen von 2 bis 255 reagieren würde, deren Regelhaftigkeit wohl kein Netzwerk in der Lage sein dürfte zu erkennen, mit dem Ziel, die dabei erzielten Ergebnisse anderen Versuchen der gleichen Größe als maximale Trainingszeiten gegenüberzustellen.

Es scheint nun so, als hätte bereits das Parity-Problem diese Aufgabe übernommen.

Die Untersuchung des Primzahlen-Prädikats, dessen Datensatz 254 Patterns von binär kodierten Zahlen der Länge 8 Bit umfaßt und damit zwar zwei Patterns weniger besitzt als das Parity-Problem, ansonsten aber dieselbe Größe hat, kann dazu dienen festzustellen, ob die beim Parity-Problem erzielten Ergebnisse pathologische Einzelfälle waren (oder auf einem Programm- oder Systemfehler beruhen) oder ob ein anderes Problem vergleichbarer Größe und Komplexität ähnliche Ergebnisse erzielt.

Zu diesem Zweck wurden die beiden einzigen Verfahren, die beim Parity-Problem konvergierten, eingesetzt.

Die Parameter waren dieselben wie beim Parity-Problem:

- 8 Input-Units, 16 hidden Units, 1 Output-Unit
- abgeschalteter "[S]everal output activations penalty term"
- $\epsilon_{abs} = 0.1$ und $\epsilon_{pat} = 0.1$

Symmetrische Aktivierungsfunktion / Daten konvertiert

Verfahren	Epochen	Updates	Epochen	Updates	Epochen	Updates
	1. Versuch		2. Versuch		3. Versuch	
Eigenes / Dynamic	39	4,627.	45	5,106	$\perp$	–
Gemischtes Verfahren	$\perp$	–	$\perp$	–	$\perp$	–

Diese Ergebnisse, die im Falle von meinem Verfahren in ziemlich genau der gleichen Größenordnung liegen wie beim Parity-Problem, bestätigen die dort erzielten Resultate.

Das Versagen des gemischten Verfahrens bei diesem Problem ist nicht unbedingt aussagekräftig. Normalerweise wurde dieses Verfahren von mir nur im interaktiven Betrieb eingesetzt, wo ich bei schlechter Konvergenz während der QuickProp-Phase den Versuch einfach neu startete. Dies führte in aller Regel zum Erfolg. Hier jedoch wurden die Versuche als Batch-Job an den Großrechner abgeschickt, ohne jede weitere Möglichkeit der Einflußnahme von außen.

In einer weiteren Versuchsserie wurde das Primzahlen-Problem mit Hilfe des gemischten Verfahrens im interaktiven Betrieb trainiert, jedoch ohne die Verwendung der symmetrischen Aktivierungsfunktion.

Dabei wurde die Anzahl der zur Verfügung gestellten hidden Units variiert.

Als Fehlermaß wurde der RMS-Fehler benutzt, die Fehlerschranken wurden auf die Werte $\epsilon_{abs} = 0.002$ und $\epsilon_{pat} = 0.002$ festgesetzt, um zu demonstrieren, daß das Netzwerk diese willkürliche Klassifikationsaufgabe mit beträchtlicher Genauigkeit zu lösen in der Lage ist. Außerdem hat diese niedrige Fehlerschranke den Effekt, die Schwankungen der Lernzeiten, die durch das Dynamic Learning bedingt sind, herabzusetzen. Dies beruht darauf, daß im Bereich sehr kleiner Fehler die zufällige Auswahl der Patterns kaum noch einen Einfluß auf die Trainingsdauer hat. In diesem Bereich verläuft die Kurve des globalen Fehlers asymptotisch und praktisch ohne Abweichungen nach unten oder oben gegen Null (siehe dazu auch die Abbildungen in Anhang A).

Symmetrische Aktivierungsfunktion / Daten konvertiert

hidden Units	Epochen	BP-Phasen	Updates	RMS-Fehler
16	3154	793,279.	410,490.	0.0019941
15	3226	809,614.	426,825.	0.0019979
14	3360	840,003.	457,214.	0.0019969
13	3411	851,585.	468,796.	0.0019944
12	3566	886,806.	504,017.	0.0019984
11	3717	922,093.	539,304.	0.0019940
10	3932	971,585.	588,796.	0.0019961
9	4038	996,285.	613,496.	0.0019939
8	8435	2,036,308.	1,359,533.	0.0019885

Diese Ergebnisse bestätigen das von allen Untersuchern berichtete Verhalten, wonach bei zunehmender Anzahl von hidden Units ein Netzwerk in der Regel besser lernt. Bei einer geringen Zahl von hidden Units steigt die benötigte Trainingszeit stark an, da hier das Netzwerk die zugrundeliegenden Regeln stärker berücksichtigen und extrahieren muß, anstatt die Patterns einfach nur abzuspeichern.

So lautet jedenfalls die gängige Theorie. Möglicherweise treten jedoch lediglich Effekte auf, wie sie von Fahlman [7] von seinen Encoder-Netzwerken berichtet werden, wonach „super-enge" Netzwerke wie 8–2–8 die gewünschte Abbildung ebenfalls lernen können, indem sie mehrere analoge Aktivierungswerte in den hidden Units zur Kodierung der Patterns einsetzen. Möglicherweise handelt es sich auch bei anderen Problemen mit nur sehr wenigen hidden Units um das gleiche Phänomen und nicht um Regel-Extraktion im mehr menschlichen Sinne.

5.5 Das „Zwei-oder-mehr-Blöcke"-Prädikat

Das "Two-or-more-clumps"-Prädikat ist ein weiteres binäres Problem, das ebenfalls in einer Version mit 8 Bit langen Binärwörtern untersucht wurde. Dadurch lassen sich die hier gewonnenen Ergebnisse im Prinzip mit denen des Parity- und des Primzahlen-Problems vergleichen.

Interessant ist die Untersuchung dieses Problems jedoch vor allem deshalb, weil es ein explizit konstruierbares Netzwerk gibt, das dieses Prädikat berechnet (siehe Kapitel 4 oder Denker et al. [4]).

In einem ersten Versuch wurde mit Hilfe des gemischten Verfahrens (siehe auch die Abschnitte 5.3 und 5.4) die Hälfte des Datensatzes (zur späteren Überprüfung des Generalisierungsvermögens des ausgelernten Netzes) trainiert. Dazu wurden zufällig 128 Patterns aus dem Datensatz mit 256 Patterns, entsprechend den Binärzahlen von 0 bis 255, ausgewählt.

Das Netzwerk besaß 8 Input-Units zur Aufnahme der Eingabepatterns, eine Ausgabe-Unit zur Anzeige, ob das Prädikat zutrifft oder nicht sowie zu Beginn 8 hidden Units. 8 hidden Units deshalb, weil die explizit konstruierbare Lösung für dieses Problem ebenfalls 8 hidden Units benötigt.

Abbildung 5.1: Reduktion der hidden Units beim „Zwei-oder-mehr-Blöcke“-Prädikat

Der “[S]everal output activations penalty term” wurde abgeschaltet.

Nach Erreichen der Fehlerschranke von $\epsilon_{abs} = \epsilon_{pat} = 0.01$ (mit RMS-Fehlermaß) wurde mit Hilfe des Skeletonizing-Verfahrens von Mozer & Smolensky die Relevanz der hidden Units bestimmt und die am wenigsten relevante Unit entfernt.

Danach wurde das Netzwerk erneut trainiert, um den durch das Entfernen der einen hidden Unit gestiegenen Gesamtfehler wieder unter die Fehlerschranke zu drücken. Dies war nach wenigen, ca. 50 Epochen der Fall.

Erneut wurde die Relevanz der hidden Units bestimmt, die hidden Unit mit der niedrigsten Relevanz entfernt und das Netzwerk bis zum Erreichen der Fehlerschranke trainiert.

Nach der fünften Wiederholung dieser Prozedur gelang es nicht mehr, den Gesamtfehler des Netzwerks mit inzwischen nur noch drei hidden Units unter die Fehlerschranke zu drücken, vielmehr schwankte er chaotisch in einem begrenzten Bereich weit oberhalb des gewünschten Niveaus hin und her.

Dieser Versuch ist in Abbildung 5.1 (Seite 168) dokumentiert. (Gepunktete senkrechte Linien zeigen die Unterbrechung des Lernens durch Erreichen der eingegebenen Maximalzahl von Epochen an, gestrichelte Linien das Herausnehmen einer hidden Unit)

Der Versuch belegt die Tatsache, daß die konstruierte Lösung für dieses Problem mit 8 hidden Units nicht optimal ist, und daß offensichtlich ein Netzwerk mit nur 4 hidden Units existiert, das das Problem zumindest für den halben Datensatz löst.

In einem zweiten Versuch wurde daher ein Netzwerk mit 5 hidden Units (um es dem Netzwerk nicht allzu schwer zu machen) mit dem gesamten Datensatz (mit 256 Patterns) trainiert.

Nach Erreichen der Fehlerschranke von $\epsilon_{abs} = \epsilon_{pat} = 0.002$ (mit RMS-Fehlermaß) fiel beim Betrachten der Gewichtsmatrix eine gewisse Symmetrie der Lösung auf, wenn man sich alle Einträge zu ganzen Zahlen gerundet vorstellte.

```
<  0>  -12.9969397    2.9866595  -11.7235308    3.0489502    4.5501976
<  1>    0.0462352    0.1353238   -0.8584738   -6.7854424   -2.3041487
<  2>    0.0718420   -0.1006416    4.0823975   -8.2025385    3.3873096
<  3>   -0.1851888    0.2089740    5.4493661   -4.3759661   -3.5781279
<  4>    0.2185943   -0.3114010    5.2824388    2.1631222  -10.2712317
<  5>    6.5954409    1.0737677    0.2693349   -0.4285462  -10.3463249
<  6>    6.6444316   -4.5124192   -0.2735523    0.3190915   -3.3872857
<  7>    4.1969543   -7.6088910    0.1428923   -0.2621806    3.6906025
<  8>   -2.0537071   -5.7737131   -0.0138500    0.2752909   -2.5508432
<  0>   22.3672829
<  1>  -14.7934427
<  2>  -14.6637211
<  3>  -15.1494083
<  4>  -14.6237040
<  5>  -14.5017319

Standard logistic function
256 patterns     NORMAL
325120 pattern fetch cycles
```

```
1269 learning epochs
384532 propagation cycles
315412 backpropagation cycles
60412 weight matrix updates
Error: 0.0019996 (RMS)
```

Daher erstellte ich manuell eine Gewichtsmatrix mit den entsprechenden ganzzahligen Einträgen:

```
<  0>  -12.0000000    3.0000000  -12.0000000    3.0000000    4.0000000
<  1>    0.0000000    0.0000000   -1.0000000   -6.0000000   -3.0000000
<  2>    0.0000000    0.0000000    4.0000000   -8.0000000    4.0000000
<  3>    0.0000000    0.0000000    6.0000000   -5.0000000   -3.0000000
<  4>    0.0000000    0.0000000    6.0000000    2.0000000  -10.0000000
<  5>    6.0000000    2.0000000    0.0000000    0.0000000  -10.0000000
<  6>    6.0000000   -5.0000000    0.0000000    0.0000000   -3.0000000
<  7>    4.0000000   -8.0000000    0.0000000    0.0000000    4.0000000
<  8>   -1.0000000   -6.0000000    0.0000000    0.0000000   -3.0000000
<  0>   23.0000000
<  1>  -15.0000000
<  2>  -15.0000000
<  3>  -15.0000000
<  4>  -15.0000000
<  5>  -15.0000000
```

Nach dem Einlesen dieser Matrix in das Simulationsprogramm ergab der Test anhand des vollständigen Datensatzes mit 256 Patterns einen globalen Fehler von 0.0489855 (Maximumnorm), einem durchaus akzeptablen Fehler.

Die Eleganz dieser mit Hilfe eines Netzwerks gefundenen Lösung besteht zum Einen vor allem in der Tatsache, daß sie weniger hidden Units benötigt als die konstruierte Lösung, zweitens ist sie ebenfalls ganzzahlig und drittens von auffälliger Symmetrie.

5.6 Das Mesh-Problem

Das Mesh-Problem ist eine Klassifikationsaufgabe mit relativ gut dosierbarem Schwierigkeitsgrad. Je nachdem wie stark die zwei Punktmengen sich gegenseitig umschließen, muß das Netzwerk mehr oder weniger Schnitte durch die Figur legen, um die zwei Punktmengen zu separieren. Außer im trivialen Fall zweier paralleler Geraden ist das Problem nicht linear separierbar. Die minimale Anzahl der zur Lösung eines Mesh-Problems erforderlichen Schnittgeraden ist gleich der Anzahl der hidden Units, die das Netzwerk zur Lösung der Aufgabe benötigt, und gleichzeitig ein direktes Maß für die Komplexität der Aufgabe.

Das Mesh-Problem ist damit das einzige mir bekannte Problem, dessen Komplexität meßbar ist und bei dem die Zahl der benötigten hidden Units exakt bestimmbar ist.

Mit Hilfe des QuickProp-Verfahrens wurde die auf Minsky und Papert [18] zurückgehende Version des Mesh-Problems (siehe Kapitel 4 bzw. in Hanson und Pratt [9]) zunächst

3000 Epochen lang auf einem Netzwerk mit 2 Input-, 8 hidden und 2 Output-Units trainiert. Anschließend wurde die bis dahin beste Matrix zurückgeholt und das Training mit dem Verfahren von Schmidhuber und Dynamic Learning fortgesetzt.

Nach insgesamt 9616 Epochen (und 108.668 Back-Propagation-Phasen sowie 66.738 Updates) erreichte das Netzwerk die Fehlerschranke von $\epsilon_{abs} = \epsilon_{pat} = 0.01$ (Maximumnorm).

Abbildung 5.2: Das Mesh-Problem. Einzugsbereiche (links) und Separierung (rechts).

In Anhang A ist eine weitere Abbildung der Separierung des Mesh-Problems durch das Netzwerk nach einem anderen Lernversuch zu finden.

Zum Vergleich der verschiedenen Verfahren wurden die folgenden Versuche durchgeführt:

(Die Fehlerschranken betrugen $\epsilon_{abs} = \epsilon_{pat} = 0.01$, das Netzwerk besaß 8 hidden Units)

Verfahren	Epochen	Updates
Standard / Periodical	$\perp$	–
Standard / Dynamic	$\perp$	–
Schmidhuber / Dynamic	$\perp$	–
Schmidhuber / Sequential 100	1488	694,396.
Chan&Fallside / Batch1	$\perp$	–
QuickProp / Batch1	> 20.000	–
Delta-bar-delta-Rule / Batch1	1458	1,458.
Silva&Almeida / Batch1	$\perp$	–
Eigenes Verfahren / Dynamic	> 20.000	–
Eigenes Verfahren / Sequential 100	5082	2,060,718.

Beim Vergleich dieser Ergebnisse mit dem gemischten Verfahren (weiter oben) fällt auf, daß dieses gemessen an der Zahl der Updates allen anderen Verfahren überlegen ist, von der Delta-bar-delta-Regel von Jacobs abgesehen, die hier ein verblüffendes Resultat brachte, das alle anderen Ergebnisse weit überflügelt.

Im Vergleich mit dem Parity-Problem fällt auf, daß besonders Kombinationen des Verfahrens von Schmidhuber und meines eigenen Verfahrens konvergieren, während die anderen Verfahren in der Regel steckenbleiben.

Symmetrische Aktivierungsfunktion / Daten konvertiert

Verfahren	Epochen	Updates
Standard / Periodical	$\perp$	–
Standard / Dynamic	$\perp$	–
Schmidhuber / Dynamic	649	7,177.
Schmidhuber / Sequential 100	308	148,706.
Chan&Fallside / Batch1	$\perp$	–
QuickProp / Batch1	> 20.000	–
Delta-bar-delta-Rule / Batch1	$\perp$	–
Silva&Almeida / Batch1	$\perp$	–
Eigenes Verfahren / Dynamic	6084	44.680
Eigenes Verfahren / Sequential 100	6138	1,711,984.

Dieser Versuch bestätigt die Beobachtung anderer Untersucher, wonach im Allgemeinen bei Verwendung der symmetrischen Aktivierungsfunktion kürzere Trainingszeiten zu erzielen sind als bei Verwendung der logistischen Aktivierungsfunktion. Einige Verfahren, die im vorigen Versuch steckenblieben, konvergierten hier, zum Teil sehr langsam, im Falle von Schmidhuber / Dynamic sogar sehr schnell (und von allen Verfahren mit der geringsten Anzahl Updates).

Das QuickProp-Verfahren war in beiden Versuchen auf dem Wege zu konvergieren, hatte aber beim Limit von 20.000 Epochen die Fehlerschranke noch nicht erreicht.

Der Erfolg der Delta-bar-delta-Regel von Jacobs im vorigen Versuch scheint ein Einzelfall gewesen zu sein.

5.7 Das Multiplexer-Problem

Beim Multiplexer-Problem ist eine interessante Fragestellung, ob es einem Netzwerk gelingt, diesen speziellen Typ von Aufgabe zu lösen, dessen besondere Schwierigkeit darin liegt, daß das Signal eines Eingabekanals auf die Ausgabe geschaltet werden muß, unabhängig von der Art des angelegten Signals und unabhängig von der Belegung der übrigen Eingänge.

Die ideale (Spannungs-) Kennlinie eines Multiplexers ist eine Gerade. Während die übrigen Eingänge mit Rauschen belegt werden, muß die Ausgangsspannung streng proportional zur Eingangsspannung des durch die Selektionseingänge ausgewählten Eingabekanals ansteigen.

Zuerst wurde ein Datensatz für einen Multiplexer mit 4 Dateneingängen und 2 Selektionseingängen zusammengestellt, der alle 64 möglichen binären Eingabekombinationen enthielt.

Mit Hilfe des Verfahrens von Fahlman (QuickProp) wurde dieser Datensatz auf einem Netzwerk mit 12 hidden Units trainiert bis zum Erreichen der Fehlerschranke $\epsilon_{abs} = 0.03$ (RMS Fehlermaß). Anschließend wurde mit dem Verfahren von Schmidhuber und Dynamic Learning weitertrainiert bis zum Erreichen der Fehlerschranke $\epsilon_{abs} = 0.002$. (Der "[S]everal output activations penalty term" war dabei abgeschaltet)

Mit Hilfe des Skeletonizing von Mozer & Smolensky wurden die Relevanzen der hidden Units berechnet und die 8 hidden Units mit der niedrigsten Relevanz entfernt.

Das verbleibende Netzwerk mit 4 hidden Units wurde dann nochmals (mit Schmidhuber / Dynamic) trainiert, um den durch das Herausnehmen der hidden Units gestiegenen Gesamtfehler wieder wettzumachen, bis zum Erreichen der Fehlerschranke $\epsilon_{abs} = 0.003$. (350 Epochen)

Insgesamt wurden 1488 Epochen zum Training dieses Netzwerks benötigt.

Die Ein/Ausgabemeßkurven dieses Versuchs sind im Anhang A zu finden; dabei wurden einmal die übrigen Dateneingänge mit Rauschen (in Form von Zufallszahlen) belegt, im anderen Falle mit einem konstanten Wert (hier 0.0).

Die Kennlinien des Multiplexer-Netzwerks sind keine Geraden, was sie idealerweise sein sollten, sondern sigmoide (S-förmige) Kurven. Dies hängt offensichtlich mit der ebenfalls sigmoiden logistischen Aktivierungsfunktion der Units des Netzes zusammen.

Beim Anlegen von Rauschen zeigten die Eingänge untereinander erhebliches Übersprechen.

Ein zweiter Datensatz für einen Multiplexer mit 8 Dateneingängen und 4 Selektionseingänge wurde erstellt, der alle 2048 möglichen binären Eingabekombinationen enthielt.

Das Training eines 12–8–1-Netzwerkes mit diesem Datensatz mittels der Kombination Schmidhuber / Dynamic erforderte 357 Epochen bis zum Erreichen der Fehlerschranke $\epsilon_{abs} = 0.001$.

Da die Ein/Ausgabekennlinien denen des vorherigen Versuchs sehr ähnelten, wurden diese hier nicht abgebildet.

Ein letzter Versuch wurde mit einem dritten Datensatz für einen Multiplexer mit 4 Dateneingängen und 2 Selektionseingängen durchgeführt, der jedoch nicht nur alle binären Eingabekombinationen enthielt, sondern zusätzlich auch alle Kombinationen mit dem Eingabewert 0.5. Dies ergab insgesamt 324 Patterns (siehe auch Kapitel 4.5).

Ein Netzwerk mit 4 hidden Units benötigte 7065 Epochen Trainingszeit mit dem Verfahren von Schmidhuber und Dynamic Learning bis zum Erreichen der Fehlerschranke $\epsilon_{abs} = 0.01$.

Wie aufgrund der größeren Zahl von Stützstellen des Datensatzes zu erwarten war, waren die Kennlinien dieses Netzwerks (siehe Anhang A) erheblich glatter. Selbst mit Rauschen auf den übrigen Eingängen waren die Kurven fast völlig glatt, die Übersprechdämpfung also sehr hoch.

Die Kennlinien waren jedoch nach wie vor nicht linear, sondern sigmoid. Möglicherweise ließe sich die Linearität der Kennlinien mit einer noch größeren Anzahl von Stützstellen erreichen.

Anlaß zu dieser Vermutung gibt insbesondere der folgende Abschnitt.

5.8 Approximation reeller Funktionen

Mehrere reelle Funktionen wurden als Datensätze mit je 201 äquidistanten Stützstellen im Intervall $[0, 1]$ angelegt: Zwei Geraden, eine Parabel, ein Polynom 5. Grades, die Sinus-Funktion, die Betragsfunktion (seitlich verschoben) und eine unstetige (im Wesentlichen quadratische) Funktion.

Zur Approximation dieser Funktionen wurde ein 1–20–1-Netzwerk verwendet.

Die Abbildungen aller dieser Funktionen sind in Anhang A zu finden. Dünn gepunktet sind darin die Funktionswerte an den Stützstellen der Datensätze zu erkennen, die durchgezogene Kurve zeigt die Ausgaben des Netzwerkes an. Man beachte, daß dabei wesentlich mehr Punkte ausgewertet wurden als nur die an den Stützstellen, das Netzwerk also interpolieren mußte. Diese Interpolation ist im Zusammenhang mit der Untersuchung der Generalisierungsfähigkeit von Netzwerken von Bedeutung.

Das Lernverhalten der verschiedenen Funktionen unterschied sich in den durchgeführten Vorversuchen (unter Verwendung des gemischten Verfahrens aus Fahlmans QuickProp und Schmidhubers Verfahren mit Dynamic Learning im interaktiven Betrieb) nicht wesentlich.

Daher wurde eine Funktion, die Sinus-Funktion, herausgegriffen, mit der eine umfangreichere Untersuchung durchgeführt wurde.

Die Versuche wurden laufend durchnumeriert; immer zwei aufeinanderfolgende Versuche verwendeten dieselbe Verfahrenskombination, der erste jedoch mit der logistischen, der zweite mit der symmetrischen Aktivierungsfunktion und mit transformierten Daten.

Als Fehlerschranken wurden $\epsilon_{abs} = 0.02$ und $\epsilon_{pat} = 0.02$, als Fehlermaß die Maximumnorm verwendet, der "[S]everal output activations penalty term" war abgeschaltet.

Da viele der Verfahren diese Fehlerschranke nach 10.000 Epochen noch nicht erreicht hatten, wird zusätzlich bei jedem Verfahren für die beste bis dahin gefundene Matrix die Anzahl der benötigten Epochen und der bei dieser Matrix noch bestehende globale Fehler angegeben. (Man beachte, daß die Verfahren bei einem Fehler von etwa 0.5 – 0.7 starteten, ein Fehler bis hinunter zu 0.1 also noch als sehr groß anzusehen ist)

		Bei Konvergenz oder Abbruch		Beste bis dahin gefundene Matrix	
Ref.	Verfahren	Epochen	Restfehler	Epochen	Fehler
Sin1	Schmidhuber / Periodical	> 10.000	0.0470928	5933	0.0321862
Sin2*	Schmidhuber / Periodical	> 10.000	0.0468341	9859	0.0396899
Sin3	Schmidhuber / Dynamic	4542	0.0199505	–	–
Sin4*	Schmidhuber / Dynamic	> 10.000	0.0604289	5460	0.0330798
Sin5	Standard / Dynamic	1402	0.0198877	–	–
Sin6*	Standard / Dynamic	> 10.000	0.3408118	9884	0.1682246
Sin7	QuickProp / Batch1	> 10.000	0.0303399	5585	0.0279356
Sin8*	QuickProp / Batch1	> 10.000	0.5500270	7548	0.3065277
Sin9	Eigenes / Dynamic	3148	0.0199889	–	–
Sin10*	Eigenes / Dynamic	> 10.000	0.0987093	3871	0.0486198
Sin11	Standard / Periodical	$\perp$	0.5044286	3	0.4011194
Sin12*	Standard / Periodical	> 10.000	0.4406312	3773	0.3993409
Sin13	Schmidhuber / Sequ. 100	> 5.000	0.0438715	4998	0.0348477
Sin14*	Schmidhuber / Sequ. 100	> 5.000	0.0560530	2875	0.0411535
Sin15	Standard / Sequ. 100	> 5.000	0.3807237	5000	0.3807237
Sin16*	Standard / Sequ. 100	$\perp$	0.4028344	41	0.4028344

"*" : Symmetrische Aktivierungsfunktion / Daten konvertiert

Die einzigen Verfahrenskombinationen, die konvergierten (sogar in weit weniger als

10.000 Epochen) waren Schmidhuber / Dynamic, Standard / Dynamic und Eigenes Verfahren / Dynamic, interessanterweise hier jedoch ausschließlich bei Verwendung der logistischen Aktivierungsfunktion.

Überraschenderweise schnitt hier das Standardverfahren (mit Dynamic Learning) am besten ab.

Nach den 10.000 Epochen hatte das QuickProp-Verfahren den geringsten Restfehler unter den übrigen Verfahren und auch die bis dahin beste gefundene Matrix hatte den kleinsten Fehler von allen übrigen Verfahren.

Zwei Verfahren blieben sogar in lokalen Minima stecken (Standard / Periodical mit logistischer und Standard / Sequential 100 mit symmetrischer Aktivierungsfunktion).

Die Versuche mit symmetrischer Aktivierungsfunktion wurden noch einmal wiederholt, jedoch *ohne* Transformierung der Daten.

Die Fehlerschranken wurden für diesen Versuch gegenüber dem vorigen halbiert ($\epsilon_{abs} = 0.01$ und $\epsilon_{pat} = 0.01$), da sich sonst der relative Fehler bei den Patterns verdoppelt hätte. (Durch die Verwendung der nicht transformierten Daten zusammen mit der symmetrischen Aktivierungsfunktion ist die Amplitude der Funktion nur noch halb so groß wie bei den transformierten Daten, derselbe absolute Fehler wirkt sich damit relativ zur Amplitude doppelt so stark aus)

		Bei Konvergenz oder Abbruch		Beste bis dahin gefundene Matrix
Ref.	Verfahren	Epochen	Restfehler	Fehler
Sin2+	Schmidhuber / Periodical	$\perp$	0.1983133	0.1972953
Sin4+	Schmidhuber / Dynamic	6934	0.0099398	–
Sin6+	Standard / Dynamic	> 10.000	0.1152251	0.0513405
Sin8+	QuickProp / Batch1	> 10.000	0.4498067	0.1684152
Sin10+	Eigenes / Dynamic	962	0.0099313	–
Sin12+	Standard / Periodical	> 10.000	0.2046274	0.2046274
Sin14+	Schmidhuber / Sequ. 100	> 5.000	0.1984077	0.1973438
Sin16+	Standard / Sequ. 100	> 5.000	0.2043679	0.2004272

Symmetrische Aktivierungsfunktion / Daten *nicht* konvertiert

Die einzigen zwei Verfahrenskombinationen, die bei diesem Versuch konvergierten, waren Schmidhuber / Dynamic und Eigenes Verfahren / Dynamic, wobei letzteres das beste Ergebnis unter allen Versuchen überhaupt erzielte.

Das Verfahren von Schmidhuber mit Periodical Learning blieb in einem lokalen Minimum stecken.

5.9 Der Aachener Aphasie Test

Der Datensatz zu diesem Problem umfaßt die 314 Patientenprofile der Normstichprobe, die als Vergleichsbasis sowohl von den Fachärzten als auch von einem Statistik-Paket zur Diagnoseerstellung herangezogen wird.

Im ersten Fall wurden die Ergebnisse aus den Untertests zusammengefaßt, so daß sich 11 verschiedene Punktwerte ergaben, mit denen das Netzwerk gefüttert wurde. Die Punktwerte wurden dabei (durch Subtraktion vom Maximalwert und anschließender Division durch denselben) auf das Intervall $[0, 1]$ transformiert.

Netzwerke mit verschiedenen Anzahlen von hidden Units wurden mit diesen Daten trainiert. Die Versuche reichten von 8 hidden Units bis hinunter zu 2 hidden Units. Die Trainingszeiten nahmen mit abnehmender Zahl der hidden Units zu. Bei der Überprüfung der Generalisierungsleistung dieser Matrizen anhand von weiteren Patientendaten ergaben sich mehr Unstimmigkeiten bei den Netzen mit extrem wenigen hidden Units.

Aus diesem Grund wurden die folgenden Versuche mit 8 hidden Units durchgeführt, was nach den vorherigen Versuchen eine ausreichend bemessene Anzahl ist.

Das 11–8–4-Netzwerk wurde mit den Fehlerschranken $\epsilon_{abs} = 0.01$ und $\epsilon_{pat} = 0.1$ mit verschiedenen Verfahren trainiert.

Die Versuche wurden einmal mit der logistischen und ein weiteres Mal mit der symmetrischen Aktivierungsfunktion (und transformierten Daten) durchgeführt.

Ref.	Verfahren	Epochen	Updates
AAT00	Standard / Periodical	> 10.000	–
AAT01	Standard / Batch1	$\perp$	–
AAT02	Standard / Dynamic	4743	207,106.
AAT03	Standard / Sequ. 100	$\perp$	–
AAT10	Schmidhuber / Dynamic	639	80,759.
AAT11	Schmidhuber / Sequ. 100	523	268,155.
AAT12	Schmidhuber / Batch2	$\perp$	–
AAT13	Schmidhuber / Batch1	> 10.000	–
AAT20	Chan&Fallside / Batch1	$\perp$	–
AAT30	QuickProp / Batch1	$\perp$	–
AAT40	Delta-bar-delta-Rule / Batch1	$\perp$	–
AAT50	Silva&Almeida / Batch1	$\perp$	–
AAT70	Eigenes / Dynamic	5809	805,103.
AAT71	Eigenes / Sequ. 100	697	1,385,512
AAT72	Eigenes / Batch2	$\perp$	–
AAT73	Eigenes / Batch1	$\perp$	–

Als einzige Verfahren konvergierten: Standard / Dynamic, Schmidhuber / Dynamic, Schmidhuber / Sequential 100, Eigenes Verfahren / Dynamic und Eigenes Verfahren / Sequential 100.

Standard / Periodical und Schmidhuber / Batch1 waren auf dem Wege zu konvergieren, hatten aber nach 10.000 Epochen die Fehlerschranke noch nicht erreicht. Die übrigen Verfahren blieben in lokalen Minima stecken.

Symmetrische Aktivierungsfunktion / Daten konvertiert

Ref.	Verfahren	Epochen	Updates
AAT00s	Standard / Periodical	$\perp$	–
AAT01s	Standard / Batch1	$\perp$	–
AAT02s	Standard / Dynamic	$\perp$	–
AAT03s	Standard / Sequ. 100	$\perp$	–
AAT10s	Schmidhuber / Dynamic	290	79,823.
AAT11s	Schmidhuber / Sequ. 100	94	33,411.
AAT12s	Schmidhuber / Batch2	$\perp$	–
AAT13s	Schmidhuber / Batch1	> 10.000	–
AAT20s	Chan&Fallside / Batch1	$\perp$	–
AAT30s	QuickProp / Batch1	$\perp$	–
AAT40s	Delta-bar-delta-Rule / Batch1	$\perp$	–
AAT50s	Silva&Almeida / Batch1	$\perp$	–
AAT70s	Eigenes / Dynamic	146	8,984.
AAT71s	Eigenes / Sequ. 100	$\perp$	–
AAT72s	Eigenes / Batch2	$\perp$	–
AAT73s	Eigenes / Batch1	$\perp$	–

Im zweiten Versuchsdurchgang mit der symmetrischen Aktivierungsfunktion konvergierten als einzige Verfahren nur Schmidhuber / Dynamic, Schmidhuber / Sequential 100 und Eigenes Verfahren / Dynamic, also weniger als bei Verwendung der logistischen Aktivierungsfunktion. Dafür waren die Ergebnisse in allen drei Fällen besser als mit der logistischen Aktivierungsfunktion.

Das eigene Verfahren mit Dynamic Learning erzielte das weitaus beste Resultat. (Wie aus den Abbildungen im Anhang A zu entnehmen ist, fällt es außerdem in allen Fällen früher steil ab als die übrigen Verfahren, d.h. bei einer größeren Fehlerschranke wäre der Geschwindigkeitsvorsprung sogar noch größer)

Unter den gleichen Versuchsbedingungen wie in den vorangegangenen beiden Versuchen wurde ein 24–8–4-Netzwerk mit einem modifizierten Datensatz trainiert, bei dem die Ergebnisse aus den Untertests *nicht* zusammengefaßt waren, weshalb es hier nicht 11, sondern 24 Eingabewerte gab.

Es wurde jedoch nur das bisher erfolgreichste Verfahren, zweimal wiederholt, getestet.

Symmetrische Aktivierungsfunktion / Daten konvertiert

Verfahren	Epochen	Updates
Eigenes Verfahren / Dynamic	50	10,108.
Eigenes Verfahren / Dynamic	29	4,012.

Erstaunlicherweise konvergierte das Verfahren bei diesem größeren Netzwerk sogar noch schneller. Möglicherweise ist dies jedoch auf günstige Initialisierungen der Gewichtsmatrizen und/oder eine günstige zufällige Auswahl der Patterns durch das Dynamic Learning zurückzuführen.

Der Versuch bestätigt jedoch die Güte des von mir entwickelten Verfahrens in Verbindung mit Dynamic Learning für größere, "Real-World"-Anwendungen.

Kapitel 6

Schluß

6.1 Bewertung der Ergebnisse

Die Versuchsergebnisse aus den Simulationen ergeben ein geteiltes Bild: Während in den Miniproblemen wie XOR und Rotkäppchen nahezu alle Verfahren stets konvergierten, wenn auch mit wechselnder Schnelligkeit, blieben die meisten Verfahren in größeren Problemen wie Parity, dem Mesh-Problem, der Sinus-Funktion und dem Aachener Aphasie Test stecken.

Zwar kann man einwenden, daß möglicherweise ungünstige Initialisierungen die Ursache hierfür seien und daß die einmalige Durchführung der einzelnen Versuche daher nicht ausreiche. Dieses Argument ist jedoch einmal angesichts des immer gleichartig wiederkehrenden Profils der Leistungen der verschiedenen Verfahren nicht überzeugend, zum Anderen sind nach meiner Erfahrung ungünstige Initialisierungen nicht so häufig, wie man zur Aufrechterhaltung dieser Hypothese annehmen müßte. Oder man müßte annehmen, daß die betreffenden Verfahren viel empfindlicher auf für andere Verfahren nur geringfügig schlechtere Initialisierungen reagieren, dies wäre den Verfahren jedoch ebenso anzulasten.

Vor allem bei den kleineren Problemen hat das Standardverfahren gezeigt, daß es im Vergleich mit den anderen Verfahren (optimale Parameterwerte jedoch vorausgesetzt) nicht so schlecht ist wie manchmal sein Ruf.

Eine mögliche Ursache des relativ schlechten Abschneidens des Verfahrens von Schmidhuber bei den Miniproblemen wie XOR oder Rotkäppchen kann in dem Umstand begründet liegen, daß das Verfahren eine möglichst „glatte“ Fehlerfläche voraussetzt, so daß die lineare Approximation des Verfahrens genügend gute Approximationen der Nullstellen liefern kann. Bei sehr niedrig dimensionalen Problemen, sowohl was die Anzahl der Gewichte als auch die Anzahl der Patterns anbelangt, ist die Fehlerfläche naturgemäß sehr viel scharfkantiger und spitzer als bei großen Problemen, bei denen das Verfahren von Schmidhuber ausgezeichnete Ergebnisse erzielte.

Das Verfahren von Chan & Fallside leidet (wie auch die Verfahren von Jacobs und von Silva & Almeida) daran, daß die Lernrate oft in die „Sättigung“ geht, entweder immer kleiner wird und praktisch zu Null wird (wenn auch den Formeln gemäß nicht mathematisch, so doch numerisch) oder immer weiter wächst. Ohne die von mir zusätzlich eingebauten Begrenzungen kamen teilweise etwa 60-stellige Lernraten vor!

Der Versuch beim Rotkäppchen-Problem mit stärker eingeschränktem Bereich für die Lernrate ließ bei einigen Lernmodi eine deutliche Verbesserung erkennen; im gleichen Versuch waren die Ergebnisse unter Batch1 Learning mit dem sehr weiten Bereich sogar die besten der gesamten Versuchsserie.

Das QuickProp-Verfahren von Fahlman blieb beim XOR-Problem öfter stecken, erzielte sonst aber bei den kleinen Problemen sehr gute Resultate. Bei den großen Problemen blieb es ebenfalls des öfteren stecken, in einigen Fällen erreichte es jedoch lediglich die Fehlerschranke bis zur vorgegebenen Maximalzahl von Epochen nicht, war aber im Prinzip konvergent.

Bis auf einen einzigen Versuch bei den größeren Problemen, in dem die Delta-bar-delta-Regel von Jacobs sogar das mit Abstand beste Resultat der Versuchsserie erbrachte, blieb dieses Verfahren bei allen anderen der großen Probleme stecken, ebenso das diesem sehr ähnliche Verfahren von Silva & Almeida, das bei keinem dieser Probleme konvergierte.

Zwar läßt sich einwenden, daß in dieser Arbeit die Möglichkeiten der Verfahren von Jacobs und von Silva & Almeida nicht ausgeschöpft wurden, indem keine anderen Parameter als die von den Autoren empfohlenen getestet wurden, die Versuchsanordnung diesen Verfahren also nicht gerecht wird.

Da die Autoren behaupten, diese Parameter hätten sich bereits bei einer Reihe verschiedener Probleme bewährt, muß jedoch von der prinzipiellen Eignung dieser Werte ausgegangen werden.

Ziel dieser Untersuchung war es nicht, die optimalen Parameter dieser Verfahren herauszufinden, sondern „gebrauchsfertige“ Verfahren zu implementieren und zu vergleichen, wobei ein bei möglichst vielen Problemen gleichermaßen geeignetes Verfahren gesucht war, das möglichst wenige Parameter erfordert oder von diesen nicht kritisch abhängt.

Zudem stellt sich die Frage, ob sich der zusätzliche Aufwand für die individuellen Lernraten lohnt. Im Hinblick auf die anderen hier getesteten Verfahren lautet die Antwort eindeutig Nein.

Das von mir lediglich versuchsweise aus dem Verfahren von Chan & Fallside weiterentwickelte Verfahren erzielte gleich beim allerersten Test (anhand des AAT-Problems) einen solchen Durchbruch gegenüber den früheren Lernzeiten mit anderen Verfahren, daß ich es in den Vergleich mit den anderen Verfahren mit aufnahm. Erfreulicherweise zeigte es sich dabei auch für andere Probleme in hohem Maße geeignet.

Das ideale Verfahren, daß immer hervorragende Ergebnisse erzielt und dabei von keinerlei Parametern abhängt, wurde auch hier nicht entdeckt.

Das Verfahren von Schmidhuber kommt diesem Ideal jedoch sehr nahe. Zwar erzielte es bei den kleineren Problemen keine sehr guten Resultate, dafür war es stets verläßlich. Die wenigen Fälle, wo dieses Verfahren einmal nicht konvergierte, waren oft (tatsächlich) auf ungünstige Initialisierungen zurückzuführen, was durch einen Neustart behoben werden konnte.

Zudem besitzt dieses Verfahren den großen Vorteil, als einziges von den getesteten Verfahren tatsächlich keine Parameter zu benötigen. (Die maximale Schrittweite wurde von mir noch nie geändert)

Als zweites Verfahren der Wahl ist das von mir entwickelte Verfahren zu nennen, das bei den großen Problemen in der Regel außerordentlich gute Ergebnisse erzielte. Mit nur drei verschiedenen Sätzen von Parametern waren alle untersuchten Probleme in guten bis exzellenten Lernzeiten trainierbar.

Als drittes Verfahren ist das QuickProp-Verfahren von Fahlman zu empfehlen, obwohl es in den hier geschilderten Versuchen relativ schlecht abschnitt.

Diese Empfehlung beruht auf der Beobachtung im interaktiven Betrieb mit dem Simulationsprogramm, daß dieses Verfahren in sehr vielen Fällen innerhalb weniger Epochen (teilweise unter 100, manchmal bis zu 1000 oder 2000) einen plötzlichen und sehr steilen Abfall des globalen Fehlers auf sehr niedrige Werte zu Folge hat, danach aber kaum noch eine Verbesserung des Fehlers eintritt.

Nach Erreichen dieses steilen Abfalls hat sich das Weiterlernen mit einem anderen Verfahren, vorzugsweise dem Verfahren von Schmidhuber mit Dynamic Learning, sehr bewährt. Auf diese Weise konvergierten viele Probleme, die beispielsweise mit dem Schmidhuber-Verfahren und Dynamic Learning allein nicht konvergierten.

Das von mir entwickelte Dynamic Learning hat sich in den meisten Versuchen hervorragend bewährt. Auch in Kombination mit dem Standardverfahren erwies es sich als besonders vorteilhaft (siehe z.B. im Versuch Sin5).

Besonders jedoch in Kombination mit dem Verfahren von Schmidhuber sowie meinem eigenen Verfahren erwies es sich als sehr effektiv.

6.2 Ausblick

Der Raum der Parameter und möglichen Verfahrenskombinationen konnte nur stichprobenartig erforscht werden. Weitere systematische und ausführliche Untersuchungen größerer Probleme ähnlich dem AAT-Problem sind wünschenswert, unter anderem auch um die Aussagekraft der hier gewonnenen Ergebnisse zu erhöhen.

Es wäre außerdem interessant, noch einige Variationen der hier vorgestellten Verfahren zu untersuchen. Beispielsweise könnte man die Beschränkung des QuickProp-Verfahrens von Fahlman auf das Batch1 Learning aufheben, da nicht einzusehen ist, warum dieses Verfahren nicht auch mit den On-Line-Lernmodi funktionieren sollte; denn schließlich handelt es sich um ein Approximationsverfahren ähnlich dem Verfahren von Schmidhuber, das gerade im On-Line-Lernmodus die besten Ergebnisse erzielte.

Ein Vergleich des Rechenzeitbedarfs der hier untersuchten Back-Propagation-Verfahren, die alle auf der Berechnung eines Fehlergradienten beruhen, mit anderen Verfahren, die ohne diesen Gradienten auskommen wie z.B. das Verfahren von Birmiwal, Sarwal und Sinha [1], wäre sicherlich sehr aufschlußreich und für "Real World"-Anwendungen von großem Interesse.

Weitere interessante Arbeiten konnten aus Zeitgründen und um den Umfang dieser Arbeit nicht noch mehr zu vergrößern nicht berücksichtigt werden, wie z.B. der "Minimal Disturbance" Back-Propagation Algorithmus von Heileman, Georgiopoulos und Brown [11], der die Störung bereits gelernter Patterns im Netzwerk durch neu dazukommende Patterns zu minimieren sucht (bisher mußte bei einer nachträglichen Erweiterung der Patternmenge, die das Netzwerk beherrschen soll, immer die gesamte Menge aller Patterns von neuem trainiert werden) oder die Arbeit von Yu und Simmons [31], die eine Verbesserung der Generalisierungsleistung verspricht (ein Untersuchungsgebiet, das hier ganz ausgespart werden mußte).

Die bisherigen Versuche deuten darauf hin, daß hier jedoch eher Verbesserungen im Detail statt große Durchbrüche zu erwarten sind.

Die Versuche beispielsweise zum Aachener Aphasie Test haben gezeigt, daß ein Netzwerk immer nur so gut ist wie die Aufgabenstellung (und ihre Kodierung).

Gerade beim AAT war das Netzwerk nicht in der Lage, Dinge zu vollbringen, die nicht ein Experte zumindest im Prinzip genauso hätte tun können.

Wenn auch im Zusammenhang mit Neuronalen Netzwerken oft von „Regel-Extraktion" die Rede ist, darf man diese nicht mit *Einsicht in Zusammenhänge* verwechseln. Simple zweistufige Netzwerke mit der (wenn auch gewichteten) Addition von Signalen und anschließender Stauchung durch die logistische Aktivierungsfunktion sind dazu sicherlich nicht in der Lage.

Ob komplexere Schaltungen aus mehreren Netzwerken wie z.B. in Rumelhart/McClelland [24] (Kapitel 12 Seite 473ff) jemals erklärende Kraft oder Modellcharakter für höhere kognitive Funktionen des Menschen erlangen werden, ist eine noch offene Frage.

Fest steht, daß nach einer Weile der Beschäftigung mit Neuronalen Netzen durch die Selbstbeobachtung Gemeinsamkeiten mit dem menschlichen Lernen auffielen, andererseits auch Unterschiede.

Davon gänzlich unberührt haben Neuronale Netzwerke jedoch längst ihre Nützlichkeit in vielen Anwendungsproblemen als Softwarewerkzeuge oder auch als Hardwareschaltung bewiesen.

Als solche sollten sie meines Erachtens verstanden werden; wobei ich hoffe, mit dieser Arbeit einen kleinen Beitrag zur Schärfung dieser Werkzeuge für künftige Anwendungen geleistet zu haben.

im Mai 1992

Anhang A

Abbildungen

Bei den folgenden Abbildungen handelt es sich zum überwiegenden Teil um Protokolle von Lernversuchen. Dabei ist der Verlauf des Gesamtfehlers während des Lernens dargestellt. Bei allen diesen Lernversuchen wurde als Fehlermaß die Maximumnorm der lokalen Fehler verwendet.

Die Graphiken wurden mit einem speziell hierfür geschriebenen Programm ("PLOTERR"; siehe Anhang B) erstellt, welches die vom Simulationsprogramm erstellten LOG-Files als Eingabe benutzt.

Senkrecht ist die Größe des Fehlers aufgetragen; ein Faktor am oberen Ende der Skala in Form einer Zehnerpotenz gibt ggfs. die Größenordnung der Skala an, falls die Werte nicht zwischen 0.1 und 1.0 liegen. Waagerecht sind die Epochen aufgetragen, jeder Pixel der Graphik entspricht einer Epoche. In einigen Graphiken sind die Pixel aufeinanderfolgender Epochen jeweils durch Linien miteinander verbunden, in anderen nicht.

Falls nicht anders vermerkt, wurden die gleichen Standard-Parameter benutzt, wie sie in den entsprechenden Versuchen im Kapitel 5 dokumentiert sind.

Die übrigen Graphiken wurden mit Hilfe der Utility-Routinen erzeugt, die in das Simulationsprogramm selbst eingebaut sind (siehe Anhang B).

X9 Standard / Period. $\eta = 1.5$ $\alpha = 0.0$ X9 Standard / Period. $\eta = 2.0$ $\alpha = 0.0$

X9 Standard / Period. $\eta = 3.0$ $\alpha = 0.0$

X9 Standard / Period. $\eta = 5.0$ $\alpha = 0.0$

X9 Standard / Period. $\eta = 10.0$ $\alpha = 0.0$

X9 Standard / Period. $\eta = 20.0$ $\alpha = 0.0$

X9 Standard / Period. $\eta = 25.0$ $\alpha = 0.0$

X9 Standard / Period. $\eta = 30.0$ $\alpha = 0.0$

X9 Standard / Period. $\eta = 0.5$ $\alpha = 0.9$

X9 Standard / Period. $\eta = 0.5$ $\alpha = 0.95$

X9 Standard / Period. $\eta = 0.5$ $\alpha = 0.99$

X9 Standard / Period. $\eta = 0.5$ $\alpha = 0.7$

X9 Standard / Period. $\eta = 0.7$ $\alpha = 0.9$

X9 Standard / Period. $\eta = 0.7$ $\alpha = 0.95$

X9 Standard / Period. $\eta = 0.7$ $\alpha = 0.99$

X9 Standard / Period. $\eta = 0.9$ $\alpha = 0.9$

X9 Standard / Period. $\eta = 0.9$ $\alpha = 0.95$

X9 Standard / Period. $\eta = 2.0$ $\alpha = 0.5$

X9 Standard / Period. $\eta = 2.0$ $\alpha = 0.9$

X9 Standard / Period. $\eta = 2.0$ $\alpha = 0.95$

X9 Standard / Period. $\eta = 1.0$ $\alpha = 0.9$

X9 Standard / Batch1 $\eta = 2.0$ $\alpha = 0.95$

X9 Standard / Batch1 $\eta = 2.0$ $\alpha = 0.9$

X1 Standard / Batch1 $\eta = 2.0$ $\alpha = 0.9$

X1 Standard / Batch1 $\eta = 2.0$ $\alpha = 0.95$

X1 Standard / Batch1 $\eta = 0.9$ $\alpha = 0.95$

X1 Standard / Sequ. 100 $\eta = 2.0$ $\alpha = 0.5$

X9 Standard / Dynamic $\eta = 2.0$ $\alpha = 0.5$

X1 Standard / Dynamic $\eta = 2.0$ $\alpha = 0.5$

X1 Standard / Dynamic $\eta = 5.0$ $\alpha = 0.2$

X9 Standard / Dynamic $\eta = 5.0$ $\alpha = 0.2$

X0 Schmidhuber / Period.

X0 Schmidhuber / Sequ. 100

X0 Schmidhuber / Batch1

X3 Schmidhuber / Batch1

X0 Schmidhuber / Batch2

X0 Schmidhuber / Dynamic

X0 Chan&Fallside / Batch1

X9 QuickProp / Batch1

X9 QuickProp (hyperr, split-eta) / Batch1

X0 Delta-bar-delta / Batch1

X0 Silva&Almeida / Batch1

X0 Eigenes / Sequ. 100

—

R0 Standard / Period. $\eta = 15.0$ $\alpha = 0.0$

R0 Standard / Period. $\eta = 0.5$ $\alpha = 0.9$

R0 Standard / Batch1 $\eta = 1.0$ $\alpha = 0.9$

R0 Standard / Batch1 $\eta = 15.0$ $\alpha = 0.0$

R0 Standard / Dynamic $\eta = 0.5$ $\alpha = 0.9$

R0 Schmidhuber / Sequ. 100

R0 Chan&Fallside / Batch1

R0 QuickProp / Batch1

R0 Eigenes / Sequ. 100

R0 Eigenes / Period.

R0 Eigenes / Batch1

R0 Eigenes / Batch2

R0 Eigenes / Dynamic

R0 Eigenes (0.95/0.5/5.0) / Sequ. 100

R0 Eigenes (0.95/0.5/5.0) / Period.

R0 Eigenes (0.95/0.5/5.0) / Batch1

R0 Eigenes (0.95/0.5/5.0) / Batch2

R0 Eigenes (0.95/0.5/5.0) / Dynamic

Parity Eigenes / Dynamic (symm.)

Prim Eigenes / Dynamic (symm.)

Mesh-Problem: QuickProp/Batch1 | Schmidh./Dyn. | Schmidh./Sequ. 100 I + II

Mesh-Problem: QuickProp/Batch1 | Schmidh./Dyn. | Schmidh./Sequ. 100 III + IV

Mux1: 64 Pat. QuickProp/Batch1 | Schmidh./Dyn. hid=12/4

Multiplexer input #1 ohne Rauschen

Multiplexer input #2 ohne Rauschen

Multiplexer input #3 ohne Rauschen

Multiplexer input #4 ohne Rauschen

Multiplexer input #1 mit Rauschen

Multiplexer input #2 mit Rauschen

Multiplexer input #3 mit Rauschen

Multiplexer input #4 mit Rauschen

Muxx: 2048 Pat. Schmidh./Dyn. 357 Ep.

Mux2: 324 Pat. Schmidh./Dyn. 7065 Ep.

Multiplexer 2 input #1 mit Rauschen

Multiplexer 2 input #2 mit Rauschen

Multiplexer 2 input #3 mit Rauschen

Multiplexer 2 input #4 mit Rauschen

Lineare Funktion (a)

Lineare Funktion (b)

Parabel

Polynom 5. Grades

Betragsfunktion

Parabelabschnitt und Viertelkreis

Sin1: Schmidhuber / Periodical

Sin2: Schmidhuber / Periodical (symm. Akt.-fktn.)

Sin3: Schmidhuber / Dynamic

Sin4: Schmidhuber / Dynamic (symm. Akt.-fktn.)

Sin5: Standard / Dynamic

Sin6: Standard / Dynamic (symm. Akt.-fktn.)

Sin7: QuickProp / Batch1

Sin8: QuickProp / Batch1 (symm. Akt.-fktn.)

AAT10: Schmidhuber / Dynamic

AAT11: Schmidhuber / Sequ. 100

AAT10s: Schmidh. / Dynamic (symm.)

AAT11s: Schmidh. / Sequ. 100 (symm.)

AAT71 : Eigenes / Sequ. 100

AAT70s: Eigenes / Dynamic (symm.)

AAT_24_8_4: Eigenes / Dynamic (symm.)

—

AAT_11_8_4: Relevanzen (**inp** oben, **hid** unten)

AAT_11_8_4: Verteilung der Fehler

Separierung des Mesh-Problems durch ein 2–8–2-Netzwerk

Anhang B

Programmbeschreibung

Das zur Simulation der Neuronalen Netzwerke benötigte Programm wurde auf dem VAX 11/780-System des Instituts für Medizinische Statistik und Dokumentation des Klinikums der RWTH Aachen mit Betriebssystem VMS in PASCAL geschrieben. Augenblicklich umfaßt es 4337 Zeilen Kode.

Da das Simulationsprogramm nur zum Experimentieren konzipiert war, wurde es oft je nach den augenblicklichen Erfordernissen umgeschrieben und erweitert.

Manche Eigenarten der Programmierung sind so historisch gewachsen und nicht das Ergebnis vorausschauender Planung. Dies wäre auch unmöglich gewesen, da viele anfängliche Versuche (und Irrtümer) erst den richtigen Weg gewiesen haben.

Weil dies einfacher zu programmieren und wieder umzuändern war, fiel die Wahl zwischen Menü-Steuerung und Kommandosprache (wie z.B. von McClelland & Rumelhart in [17] realisiert) zugunsten einer Ein-Buchstaben-Befehl Menü-Steuerung aus, die einerseits auch vom ungeübten Anwender dank der Menüs und der mnemotechnisch leicht zu merkenden Befehlsbuchstaben leicht zu bedienen ist, und die dem geübten Anwender eine schnellere Bedienung ermöglicht, als das durch das bei einer Kommandosteuerung erforderliche Eintippen von langen Befehlsnamen möglich wäre.

```
---------------------------------------------------------------------------
                   Back-Propagation Networks Simulator
---------------------------------------------------------------------------
                 Implementation of Advanced Learning Rules
---------------------------------------------------------------------------
                  Copyright (C) 1992  by  Steffen Beyer
---------------------------------------------------------------------------
                              Written at
                         Neurologische Klinik
                       Abteilung Neurolinguistik
                       Klinikum der RWTH Aachen
                            Pauwelsstr. 30
                             5100  Aachen
                               Germany
---------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------
              This program may be used and distributed freely
                  for personal use or scientific research
            --- Commercial use and distribution prohibited ---
---------------------------------------------------------------------------
```

Nach Aufruf des Programms von der Betriebssystemebene aus mit "RUN BP" werden zunächst einige grundsätzliche Programmparameter abgefragt, wie die Anzahl der Input-, der hidden und der Output-Units und ob die symmetrische Aktivierungsfunktion verwendet werden soll.

Anschließend kann man den Hauptteil eines File-Namens (d.h. ohne ein Suffix wie z.B. ".DAT") eingeben, der anschließend bei allen Ein- und Ausgaben, um ein passendes Suffix ergänzt, als Default-Wert für den File-Namen angeboten wird.

Anschließend fragt das Programm nach einem Namen für das LOG-File, das den Programmlauf protokolliert und alle relevanten Daten enthält, aus dem sich jeder Lernversuch mit allen Parametern rekonstruieren läßt. Das Format ist so gewählt, daß es einerseits für den menschlichen Benutzer leicht lesbar ist (es werden z.B. das Parameter-Menü und die Status-Display-Seite in dieses File geschrieben), andererseits kann man auch die während des Programmlaufs erzeugte Graphik des globalen (Gesamt-) Fehlers (nur auf TEK 4205 und kompatiblen Terminals) mit Hilfe des zusätzlichen "PLOTERR" Utility-Programms automatisch skaliert und beschriftet anzeigen und/oder plotten lassen (siehe auch am Schluß dieses Anhangs).

Falls der Benutzer während der Simulation einen Lernversuch abbricht und die Gewichtsmatrix neu initialisiert, wird das aktuelle LOG-File geschlossen und ein neues geöffnet. Dem neuen LOG-File wird dabei eine laufende Nummer an den Namen angehängt. Hieß das ursprüngliche LOG-File z.B. "AAT.LOG", wird das nächste LOG-File "AAT1.LOG" genannt, das folgende "AAT2.LOG" usw.

Schließlich wird der Benutzer gefragt, ob sein Terminal graphikfähig ist. (Nur für TEK 4205 Terminals und kompatible geeignet)

```
Please enter number of INPUT units  (1..100): 11

Please enter number of HIDDEN units (1..100): 8

Please enter number of OUTPUT units (1..100): 4

Input  units:  11
Hidden units:   8
Output units:   4

Is this correct? (yes/no) y


In the following simulations,
the SYMMETRIC logistic function and values in the range [-1, 1] are used.

Is this correct? (yes/no) y
```

```
Please enter name of default identifier for a file (default=BPS_11_8_4):
aat_11_4

Is 'aat_11_4' correct? (yes/no) y


Please enter name of log file (default=aat_11_4.LOG):

Is 'aat_11_4.LOG' correct? (yes/no) y

Using 'aat_11_4.LOG' as name of logfile.


Is your terminal capable of graphics (TEK 4205)? (yes/no) y


Initializing...

Random number seed:    9896845
```

Der Zufallszahlengenerator von VMS-PASCAL benötigt einen UNSIGNED Integer-Wert als Startzahl. Um diese zu erzeugen, wird nach jeder Tastaturabfrage des Programms die Uhr des Systems ausgelesen. Die Zehntel- und Hundertstelsekunden (deren Wert nach einer Tastatureingabe garantiert zufällig ist) werden anschließend in einen String geschoben, der dabei jedesmal um zwei Stellen weiterrückt, wobei die ältesten zwei Stellen verlorengehen. Dieser String wird dann in eine UNSIGNED-Zahl umgewandelt (von maximal 9 Stellen) und als neuer Startwert des Zufallszahlengenerators verwendet. Um den 10-stelligen String vollständig zu füllen oder auszutauschen, sind mindestens 5 Tastatureingaben erforderlich.

Nach diesen Abfragen erscheint das Hauptmenü:

```
                    --- Main Menu ---

               (C)  [C]lear data buffer
               (D)  read [D]ata file (patterns) into buffer
               (Z)  initiali[Z]e weight matrix
               (R)  [R]ead weight matrix from file
               (W)  [W]rite weight matrix to disk
               (K)  return to best matrix [K]ept

               (L)  [L]earn parameters
               (S)  [S]equential learning
               (P)  [P]eriodical learning
               (V)  sum of deri[V]atives batch learning
               (B)  sum of updates [B]atch learning
               (Y)  d[Y]namic learning ( & 'YV', 'YB' )
               (X)  Skeletonizing (Mozer&Smolensky)
```

```
              (I)   status [I]nformation display
              (U)   [U]ser functions menu
              (O)   write [O]utput file (pattern list)
              (T)   [T]ype file
              (Q)   [Q]uit

Please select: 1
```

Die zu lernenden Patterns werden während des Programmlaufs als lineare Liste (die mit Hilfe von Pointern verkettet ist) dynamisch im Hauptspeicher gespeichert. Diese Liste wird als "Buffer" bezeichnet, obwohl es sich dabei nicht um einen Speicherbereich fester Länge handelt.

(C) : Der Menü-Befehl "C" löscht diesen Buffer, d.h. der belegte Speicherplatz wird dem Betriebssystem zurückgegeben und einige Systemvariablen werden entsprechend zurückgesetzt.

(D) : Mit Hilfe des Befehls "D" lassen sich die zu lernenden Patterns von einem File einlesen. Den Default-Wert für den File-Namen kann man dadurch akzeptieren, indem man die RETURN-Taste ohne eine Eingabe drückt. Falls die symmetrische Aktivierungsfunktion verwendet wird, wird zusätzlich gefragt, ob die Daten (die stets im Bereich $[\,0\,,\,1\,]$ liegen sollten) auf das Intervall $[\,-1\,,\,+1\,]$ transformiert werden sollen.

Die eingelesenen Daten werden den ggfs. bereits im Buffer vorhandenen Daten hinzugefügt. Man kann also z.B. die zu lernenden Patterns auf mehrere Daten-Files verteilen und dann nacheinander einlesen.

Das Format dieser Daten-Files ist das folgende:

Leerzeichen und Leerzeilen können zwischen den verschiedenen Items beliebig verwendet werden, so daß ein Input-File auch für den menschlichen Benutzer lesbar gestaltet werden kann.

Das erste Item ist ein String, der zur näheren Kennzeichnung des Patterns vom Benutzer frei gewählt werden kann, aber nicht fehlen darf, sondern mindestens ein Zeichen umfassen muß. Sind mehr als 30 Zeichen vorhanden, werden alle weiteren Zeichen nach dem 30. Charakter bis zum nächsten Leerzeichen oder Zeilenende ignoriert. Dieser "Tag" (Etikett) genannte Bezeichner darf keine Leerzeichen enthalten (bei Bedarf kann man diese durch einen Underscore ("_") ersetzen).

Das zweite Item ist eine reelle Zahl, die als Gewicht des Patterns dient, falls ein solches von vornherein vorgegeben werden soll. Bisher ist aber noch kein Lernverfahren implementiert, das dieses von außen vorgegebene Gewicht verwendet. Nur das Dynamic Learning verwendet Gewichte für die Patterns, diese werden jedoch automatisch errechnet und überschreiben die aus dem Daten-File gelesenen. Dennoch darf diese Zahl im Input-File nicht fehlen.

Das Format einer reellen Zahl ist wie folgt festgelegt: Die Zahl darf nur aus den Zeichen "+", "–", "." und den Ziffern 0 bis 9 bestehen. Alle anderen Zeichen werden als Trennzeichen aufgefaßt und ignoriert. Ein String zwischen zwei Trennzeichen, nur aus den

Zeichen "+", "–", "." und den Ziffern 0 bis 9 bestehend, wird einer Betriebssystem-Routine zur Umwandlung in eine Real-Zahl übergeben. Falls der String keine gültige Real-Zahl ist, erfolgt ein systembedingter Programmabbruch. Ein Komma und ein Nachkommateil sind nicht zwingend vorgeschrieben.

Als nächstes kommen die Eingabewerte in diesem Real-Format, anschließend die Target-Werte.

Eine Unterscheidung zwischen den Eingabe- und den Target-Werten innerhalb des Files erfolgt nicht. Es ist Sache des Benutzers, dafür Sorge zu tragen, daß ihre Anzahl mit der später verwendeten Anzahl von Input- und Output-Units übereinstimmt. Sind zuwenig Eingabewerte vorhanden, liest das Programm statt dessen einfach die folgenden Target-Werte ein. Sind zuviele Input-Werte vorhanden, liest das Programm die überzähligen Input-Werte als Target-Werte ein. Sind insgesamt zuviele Werte vorhanden, werden die überzähligen Werte ignoriert. Sind zuwenig Werte vorhanden, werden statt der fehlenden Werte Nullen eingelesen.

```
Please enter name of data file (default=aat_11_4.DATA):

Is 'aat_11_4.DATA' correct? (yes/no) y

Data conversion (from range [0, 1] to [-1, 1]) is now ON

Is this correct? (yes/no) y

Reading and converting aat_11_4.DATA...

Read 314 patterns.
Now 314 patterns contained in buffer.
Data status:  CONVERTED

(Press <CR> to return to main menu)
```

(Z) : Mit Hilfe des "Z"-Befehls können die Gewichtsmatrizen neu initialisiert werden. Die Gewichte werden dabei mit gleichverteilten Zufallszahlen im gewählten Bereich (siehe Parameter-Menü) von $-r$ bis $+r$ belegt.

(W) : Der "W"-Befehl speichert die Gewichtsmatrix als File mit dem Suffix ".MAT" ab (falls kein anderes Suffix angegeben wird). Zur Sicherheit wird eine zweite Kopie unter dem gleichen Namen mit dem Suffix ".MATQ" abgelegt (oder mit dem angegebenen Suffix, an das der Buchstabe "Q" angehängt wird).

Die mit `<   0>` bezeichnete erste Zeile dieses Files enthält dabei die Schwellenwerte der Inputunits in ihrer natürlichen Reihenfolge. Die mit `<   1>` bis `<   n>` bezeichneten Zeilen enthalten jeweils die m Gewichte von den n Input-Units zu den m hidden Units (in natürlicher Reihenfolge).

Eine weitere Zeile mit der Nummer `<   0>` enthält die Schwellenwerte der hidden Units. Die Zeilen mit den Nummern `<   1>` bis `<   m>` enthalten schließlich die Gewichte von den m hidden Units zu den Output-Units (stets in ihrer natürlichen Reihenfolge).

Es folgen fünf Zahlen, die die Zählerstände der Komplexitätsmaße (Anzahl der Pattern-Fetches, der Epochen, der Propagation-, Back-Propagation- und Update-Phasen) enthalten.

Am Schluß des Files folgt eine Kopie der aktuellen Status-Seite (siehe unter dem "I"-Befehl des Haupt-Menüs).

(R) : Mit Hilfe des "R"-Befehls kann man eine zuvor abgespeicherte (oder von Hand eingegebene) Matrix einlesen.

Man beachte, daß obwohl beim Abspeichern soviele Nachkomma-Stellen geschrieben werden wie eine Real-Zahl in VMS-PASCAL besitzt (7), gelegentlich leichte Veränderungen im Gesamtfehler bemerkbar sind, wenn man eine Matrix in ein File schreibt und sofort wieder einliest.

Die Zählerstände der Komplexitätsmaße werden automatisch wieder eingelesen (falls diese in einem manuell erstellten File fehlen, werden statt dessen Nullen eingelesen).

Für das Format der Zahlen in einem Matrix-File gelten dieselben Regeln wie für Daten-Files.

(K) : Mit Hilfe der "K"-Option des Parameter-Menüs kann man automatisch eine Kopie der besten Gewichtsmatrix erhalten, die während des Lernens aufgetreten ist. Dies ist deshalb sinnvoll, weil der Gesamtfehler oft erheblich schwankt und nach Vollendung einer bestimmten Anzahl von Lernzyklen nicht immer das gerade beste Ergebnis vorliegt. Durch den "K"-Befehl des Hauptmenüs kann man diese beste Matrix zurückholen, falls eine solche vorhanden ist und die "K"-Option eingeschaltet ist.

(L) : Der "L"-Befehl verzweigt zum Parameter-Menü. Die einzelnen Parameter wurden bereits in Kapitel 3 theoretisch erläutert.

```
    --- Simulation Parameters ---

(I) [I]nitialization range          =    0.2000000
(E) [E]ta (learning rate)           =    0.5000000
(M) [M]omentum term                 =    0.9000000
(R) [R]oot Mean Squares/MAXimum norm error:    MAX
(H) [H]yperbolic (atanh) error function:       OFF
(S) [S]everal output activations penalty term:  ON
(Z) [Z]ero error if | (tar - out) | <    0.0000000
(B) [B]ias error ( O * (1-O) + ... ):    0.0000000
(A) [A]bsolute error limit          =    0.0020000
(P) [P]attern error limit           =    0.0020000
(C) [C]ancel learn steps that increased error: OFF
(F) [F]actor for pattern repetition =    1
(T) [T]oggle epochs/cycles below:           EPOCHS
(U) [U]pdate pattern weights every  =    1  epochs
(X) e[X]tra update (learn) rules: Standard

(O) [O]ption: save best matrix automatically   OFF
(K) [K]eep a copy of best matrix found          ON
```

```
            (Q) [Q]uit

Please select: x
```

(I) : Dieser Parameter bestimmt den Bereich (von $-r$ bis $+r$), in dem die Zufallszahlen liegen, mit denen die Gewichtsmatrizen initialisiert werden.

(E) : Die Lernrate η

(M) : Der Trägheitsfaktor α

(R) : Dieser Parameter bestimmt die Art des Fehlermaßes, das für das Abbruchkriterium verwendet wird, die Maximum-Norm (MAX) oder das Root Mean Squares Error Measure (RMS).

(H) : Diese Option schaltet den hyperbolischen (atanh) Fehler ein und aus, der von Fahlman [7] vorgeschlagen wurde.

(S) : Der von mir vorgeschlagene Penalty-Term, der in Verbindung mit Dynamic Learning das Gewicht derjenigen Patterns besonders groß werden läßt, bei denen mehr als eine Output-Unit aktiv ist. (Nur für solche Probleme verwendbar, die mehrere Output-Units umfassen und bei denen nie mehr als ein Target-Wert gesetzt ist)

(Z) : Falls ein Wert größer als Null eingegeben wird, setzt diese Option das Fehlersignal derjenigen Output-Unit auf Null, deren Differenz zwischen Aktivierung und Target-Wert die eingegebene Schranke unterschreitet.

(B) : Falls ein Wert größer als Null eingegeben wird, addiert diese Option den eingegebenen Wert zur Ableitung der Aktivierungsfunktion $O \cdot (1 - O)$ (bzw. dem Äquivalent bei Verwendung der symmetrischen Aktivierungsfunktion).

(A) : Die Fehlerschranke des Abbruchkriteriums ϵ_{abs} , unter die der globale bzw. Gesamtfehler fallen soll.

(P) : Die Pattern-Fehlerschranke ϵ_{pat} des Periodical, Sequential und Dynamic Learning. Falls der lokale Fehler eines Patterns kleiner ist als dieser Wert, wird dieses Pattern nicht weiter gelernt.

(C) : Dies ist die von Silva & Almeida [29] vorgeschlagene Option, einen Lernschritt rückgängig zu machen, falls sich durch diesen der Gesamtfehler verschlechtert hat. Diese Option ist nur beim Batch Learning wirksam.

(F) : Dies ist der "repeat factor" für das Periodical und das Dynamic Learning.

(T) : Dieser Befehl schaltet lediglich zwischen der Einheit "Epochen" und der Einheit "Zyklen" für den danach folgenden "U"-Parameter um.

(U) : Dieser Wert gibt an, alle wieviel Zyklen oder Epochen (= hier nur eine Abkürzung für soviele Zyklen, wie Patterns vorhanden sind) beim Dynamic Learning die Gewichte der Patterns neu berechnet („aufgefrischt") werden.

(K) : Falls aktiviert, wird immer automatisch eine Kopie der besten bisher während des Lernens gefundenen Gewichtsmatrix aufgehoben, die mit Hilfe des "K"-Befehls des Hauptmenüs zurückgeholt werden kann.

(O) : Falls aktiviert, wird während des Lernens jedesmal, wenn der Gesamtfehler unter den bis dahin besten Wert fällt, die aktuelle Gewichtsmatrix abgespeichert. Zur Sicherheit vor Systemabstürzen wird die Matrix doppelt abgespeichert (siehe unter dem "W"-Befehl des Haupt-Menüs). Zu jedem Zeitpunkt ist entweder die bisher beste (letzte) oder die vorletzte Matrix auf Platte vorhanden. Dies wird dadurch erreicht, daß die Abspeicherroutine zuerst die alte Version unter dem Namen mit angehängtem "Q" löscht, dann die neue Matrix unter demselben Namen speichert, anschließend die alte Version unter dem Namen ohne angehängtes "Q" löscht und schließlich die neue Matrix noch einmal unter diesem Namen abspeichert.

(X) : Mit Hilfe dieses Befehls lassen sich die speziellen Lernregeln, die in Kapitel 3 beschrieben sind, durch Angabe der entsprechenden Nummer (siehe unten) anwählen. Anschließend verzweigt das Programm jeweils in ein Untermenü, in dem die Parameter des jeweiligen Verfahrens eingestellt werden können.

```
Selected: Standard  (mode #0)

  0  :  Standard
  1  :  Schmidhubers Zero-Point-Search
  2  :  ChanFallside (autom. eta/mom)
  3  :  Fahlmans QuickProp
  4  :  Jacobs delta-bar-delta-rule
  5  :  Almeidas adaptive BP
  6  :  Perceptron
  7  :  Steffen's autom. eta/mom

Please enter the new value for [mode]: 0
```

Die Parameter der nun folgenden Verfahren sind in Kapitel 3 ausführlich erläutert.

```
              --- Schmidhubers algorithm parameters ---

              (M) [M]aximum step size        =   20.0000
              (Q) [Q]uit

Please select: q

              --- Chan & Fallside's automatic choice of eta & mom ---

              (N) mi[N]imum for eta (learning rate)   =    0.0001000
              (X) ma[X]imum for eta (learning rate)   =   20.0000000
              (L) [L]ambda (for momentum calculation) =    0.9000000
              (D) [D]efault values for learning rate and momentum:
                          eta ->    0.5       mom ->    0.9
```

```
                    (eta  =    0.5000    mom  =    0.9000)
          (Q) [Q]uit

Please select: q
```

Der Befehl "D" im Parameter-Menü des Verfahrens von Chan & Fallside setzt die Lernrate η und den Trägheitsfaktor α auf die Standard-Werte 0.5 bzw. 0.9 zurück. Diese zwei Parameter werden durch das Verfahren verändert.

```
***** Only in conjunction with batch learning (main menu option [V])!! *****

                  --- QuickProp parameters ---

              (H) [H]yperbolic (atanh) error function:           OFF
              (S) [S]plit learn rate ("split eta"):              OFF
              (T) [T]hreshold (below:descent/above:jump) =    0.0000
              (M) [M]aximum step factor                  =    1.7500
              (D) weight [D]ecay factor                  =    0.0001000
              (Q) [Q]uit

Please select: q
```

Die Option des QuickProp Parameter-Menüs "H" ist mit der Option gleichen Namens des Haupt-Parameter-Menüs identisch.

Der "maximum step factor" ist das μ aus der Arbeit von Fahlman [7].

Der "Weight-Decay"-Faktor ist hier positiv, da das eigentlich erforderliche negative Vorzeichen im Programm selbst hinzugefügt wird.

```
          --- delta-bar-delta rule parameters ---

              (T) [T]heta        =    0.7000
              (K) [K]appa        =    0.0500
              (P) [P]hi          =    0.1000
              (Q) [Q]uit

Please select: q

          --- Almeidas adaptive algorithm parameters ---

              (U) [U]p factor           =    1.2000
              (D) [D]own factor         =    0.7000
              (Q) [Q]uit

Please select: q
```

Das Perceptron besitzt keine eigenen Parameter außer der Lernrate η.

```
          --- Steffen's automatic choice of eta & mom ---

          (N) mi[N]imum for eta (learning rate)   =    0.0025000
          (X) ma[X]imum for eta (learning rate)   =    0.5000000
          (Q) [Q]uit

Please select: q
```

Als Wert für das in meinem eigenen Verfahren erforderliche α_0 wird der Wert von α (unter "M" im Haupt-Parameter-Menü) verwendet.

Im Unterschied zum Verfahren von Chan & Fallside wird bei meinem Verfahren der Wert von η und α nicht verändert.

Gehen wir nun aus diesen Untermenüs über das Haupt-Parameter-Menü wieder in das Haupt-Menü zurück. (Dies war bereits die tiefste Ebene von Unter-Menüs, die in diesem Programm möglich ist)

(S) : Mit diesem Befehl wird das Sequential Learning gestartet. Falls die "O"-Option zum automatischen Abspeichern der besten Matrix aktiv ist, wird nach einem File-Namen für die abzuspeichernde Matrix gefragt.

Anschließend muß die maximale Anzahl von Zyklen *pro Pattern* eingegeben werden (eine negative Zahl bedeutet kein Limit, eine Null führt zum Haupt-Menü zurück) sowie die maximale Anzahl von Epochen, die ausgeführt werden sollen (wieder bedeutet eine negative Eingabe kein Limit und eine Null führt zum Haupt-Menü zurück).

Falls das Terminal graphikfähig ist (TEK 4205), wird der Verlauf des globalen Fehlers graphisch dargestellt. Jeder Punkt entspricht dabei einer Epoche. Die drei Markierungen am linken Rand entsprechen einem Fehler von 1.0, 2.0 und 3.0. Nach 500 Epochen (der Bildschirm faßt horizontal nur 512 Pixel) wartet das Programm auf eine Quittierung durch den Benutzer durch Drücken der RETURN-Taste.

(P) : Mit diesem Befehl wird das Periodical Learning gestartet. Analog wie beim Sequential Learning (wie auch bei den folgenden Lernmodi) wird zuerst, falls die "O"-Option zum automatischen Abspeichern der besten Matrix aktiv ist, nach einem File-Namen für die abzuspeichernde Matrix gefragt.

Danach wird die Anzahl der Epochen eingegeben, die maximal ausgeführt werden sollen (falls das Abbruchkriterium bereits vorher erfüllt ist, stoppt das Verfahren und kehrt nach einer entsprechenden Meldung, die mit RETURN quittiert werden muß, zum Haupt-Menü zurück).

Wieder bedeutet ein negativer Wert kein Limit, die Null führt zum Haupt-Menü zurück.

Die Graphik ist für alle Lernmodi implementiert.

(V) : Das "Sum-of-Derivatives" Batch Learning, das Standard-Batch-Verfahren, abkürzend auch mit "Batch1" bezeichnet, wird durch diesen Befehl gestartet.

Die Bedienung ist wie beim Periodical Learning.

(B) : Das "Sum-of-Updates" Batch Learning, abkürzend auch mit "Batch2" bezeichnet, wird durch diesen Befehl gestartet.

Die Bedienung ist wie beim Periodical Learning.

(Y) : Durch diesen Befehl wird das Dynamic Learning gestartet.

Durch die Eingabe von "YV" bzw. "YB" statt "Y" wird das Dynamic "Sum-of-Derivatives" Batch Learning bzw. das Dynamic "Sum-of-Updates" Batch Learning gestartet.

Die Bedienung ist wie beim Periodical Learning, nur daß hier (historisch bedingt) keine negative Eingabe bei der Anzahl der Epochen möglich ist.

(X) : Mit Hilfe dieses Befehls können die Relevanzen der Input- und hidden Units (nach Mozer & Smolenksy [19]) berechnet und ggfs. hidden Units entfernt werden.

Zuerst wird gefragt, ob die Relevanzen der Input- oder der hidden Units oder von beiden berechnet werden sollen.

Sobald die Berechnung abgeschlossen ist, kann man die Ergebnisse auflisten, wahlweise auch nach Größe sortiert. (Die Sortierung läßt sich rückgängig machen)

Falls man auf einem graphikfähigen Terminal (TEK 4205) arbeitet, können die Relevanzen auch in Form eines Histogramms dargestellt werden.

Falls gewünscht, können eine oder mehrere hidden Units entfernt werden. Dies geschieht durch Überschreiben der betreffenden Zeile bzw. Spalte in den beiden Gewichtsmatrizen durch die Zeile bzw. Spalte der letzten hidden bzw. Output-Unit, wodurch aufwendige Shift-Operationen entfallen.

```
Calculate relevance of [I]nput or [H]idden units or [B]oth? b

Calculating relevance of both input and hidden units

Is this correct? (yes/no) y

Calculating relevance...

                   --- Skeletonizing Menu ---

              (L)  [L]ist the relevance values
              (D)  [D]isplay bar-histogram
              (S)  [S]ort
              (U)  [U]nsort
              (R)  [R]emove a (hidden) unit

              (Q)  [Q]uit

Please select: q
```

(I) : Dieser Befehl zeigt die Status-Seite des Simulators an.

Das gerade aktive Fehlermaß (RMS oder Maximumnorm) wird durch drei Kleiner-Zeichen ("<<<") kenntlich gemacht.

Der LMS-Fehler ist der Fehler nach der Back-Propagation-Formel, der in die Berechnung der partiellen Ableitungen eingeht.

"weightsum" ist die Summe der Gewichte aller Patterns.

Es gibt vier mögliche Datenzustände: NONE, NORMAL, CONVERTED (für transformierte Daten) und MIXED (für gemischt normale und transformierte Daten; was beim Einlesen mehrerer Daten-Files vorkommen kann)

```
Collecting data...


--- Backpropagation Networks Simulator Status Information ---

      11  input units
       8 hidden units
       4 output units

SYMMETRIC logistic function    (range [-1, 1])

     314 patterns of  CONVERTED data in buffer

    LMS error :    0.0015556         RMS error :    0.0007869
    weightsum :    0.3670742     maximum error :    0.0099844   <<<
   best error :    0.0099844

   57462 pattern fetch cycles
      59 epochs of learning
   64777 propagation routine calls
    7315 backpropagation routine calls
    7315 weight matrix updates

last filename used: 'aat_11_8_4.MAT'

(Press <CR> to return to main menu)
```

(U) : Mit diesem Befehl gelangt man in das "User Functions Menu", eine Sammlung verschiedener Utilities, zu denen sich die Ideen oder die Notwendigkeit während der Benutzung dieses Programms ergeben haben und die sich daraufhin als nützlich oder interessant erwiesen haben.

```
                   --- User Functions Menu ---

              (F)  real [F]unctions plot
              (A)  mesh problem [A]rea plot
              (L)  mesh problem [L]ine plot
```

```
(M)  [M]ultiplexer I/O characteristics plot
(D)  [D]istribution of errors histogramm
(E)  list individual learn rates [E]ta(j,i)
(R)  [R]ound all matrix elements
(C)  [C]ount # of correct/incorrect patterns
(O)  [O]utput list of correct/incorrect patterns

(Q)  [Q]uit

Please select: q
```

(F) : Mit Hilfe dieses kleinen Programms lassen sich die mathematischen Funktionen, die in Kapitel 4 beschrieben sind, auf einem graphikfähigen Terminal plotten.

Es werden dabei sowohl die Target-Werte an den Stützstellen als auch die vom Netzwerk erzeugten Ausgabewerte (wahlweise auch an beliebig vielen Zwischenstellen) gezeichnet.

(A) : Mit Hilfe dieses Programms lassen sich für das Mesh-Problem die Einzugsbereiche der beiden Mengen graphisch darstellen.

(L) : Dieses Programm stellt die zwei Punktmengen des Mesh-Problems dar und zeichnet in dieses Schema die Geraden, mit Hilfe derer die Gewichtsmatrix die Separation vornimmt. Anders als "Linear Threshold Units" haben Units mit der logistischen Aktivierungsfunktion keinen scharfen Schwellenwert. Die Geraden wurden daher durch den Punkt der größten Steilheit der sigmoiden logistischen Funktion gelegt, d.h. durch den Wert des Schwellenwerts, da dieser den Arbeitspunkt der logistischen Aktivierungsfunktion bestimmt.

Die Formel lautet:

$$y \quad = \quad - \; (\; w_{j1} \cdot x \; + \; w_{j0} \;) \; / \; w_{j2}$$

Dabei ist j die Nummer der betreffenden hidden Unit und w_{j0} der Schwellenwert dieser Unit. Die erste Input-Unit entspricht der x-Koordinate, die zweite Input-Unit der y-Koordinate.

(M) : Mit Hilfe dieser Funktion lassen sich die Ein-/Ausgabecharakteristiken für Multiplexer verschiedener Größe ausgeben. Nacheinander wird die Übertragungskennlinie jedes Eingangs geplottet. Die übrigen Eingänge können währenddessen auf einen bestimmten Wert gesetzt (clamping) oder aber mit Rauschen (Zufallszahlen zwischen Null und Eins) gefüttert werden.

Letzteres eignet sich besonders zum Test der Übersprechdämpfung.

(D) : Diese Funktion zeigt die Verteilungskurve der lokalen Fehler der gelernten Patterns an. Die Skalierung erfolgt automatisch.

Idealerweise nimmt die Verteilung der Fehler eine L-förmige Kurve an: Die meisten Patterns haben niedrige Fehler, und immer weniger Patterns haben immer größere Fehler, wobei der größte Fehler noch unterhalb der Fehlerschranke liegt.

Interessant ist der Vergleich dieser Verteilungen zwischen noch kaum trainierten und ausgelernten Gewichtsmatrizen.

(E) : Die Verfahren von Jacobs [13] und Silva & Almeida [29] verwenden individuelle Lernraten separat für jedes Gewicht. Mit Hilfe dieser Funktion kann die Matrix der Lernraten ausgegeben werden.

Das File hat im Wesentlichen das gleiche Format wie das File einer Gewichtsmatrix.

(R) : Mit Hilfe dieser Funktion kann die Gewichtsmatrix auf eine bestimmte Anzahl von Stellen nach dem Komma gerundet werden.

In der Regel verschlechtert sich dadurch der Gesamtfehler. In einigen überraschenden Fällen wurden jedoch auch ganzzahlige Lösungen gefunden.

(C) : Diese Funktion bestimmt die Anzahl der korrekt klassifizierten und der mißklassifizierten Patterns.

Das Kriterium klassifiziert eher Patterns als richtig, die noch relativ große lokale Fehler haben, als umgekehrt.

Die Routine testet lediglich, ob die Aktivierungen aller Output-Units in derselben Hälfte des möglichen Wertebereichs ($[0, 1]$ im Falle der logistischen, $[-1, +1]$ im Falle der symmetrischen Aktivierungsfunktion) liegen wie die entsprechenden Target-Werte.

(O) : Dieser Befehl erstellt ein Ausgabe-File mit einer Liste aller Patterns. Die nach dem obigen Kriterium als „korrekt“ und als „falsch klassifiziert“ angesehenen Patterns werden dabei getrennt aufgelistet, was das Auffinden der Patterns mit besonders großem Fehler in der oft langen Liste aller Patterns (wie sie von der gleichnamigen Funktion des Haupt-Menüs ausgegeben wird) sehr erleichtert.

```
Correct patterns:

#: 1   tag: Pattern#1
      0.00   0.10
#: 2   tag: Pattern#2
      1.00   0.90
#: 3   tag: Pattern#3
      1.00   0.90

Incorrect patterns:

#: 4   tag: Pattern#4
      0.00   0.90
```

Zurück zum Haupt-Menü:

(O) : Dieser Befehl erstellt ein Ausgabe-File mit einer Liste aller Patterns, wobei für jedes Pattern die Nummer im Buffer angezeigt wird, die Anzahl der Male, die es gelernt worden ist, sein Gewicht (wie vom Daten-File eingelesen oder vom Dynamic Learning berechnet), seinen lokalen Fehler und seinen Bezeichner.

Außerdem werden die Aktivierungen aller Input-, hidden und Output-Units sowie die entsprechenden Target-Werte aufgelistet.

Am Schluß des Files folgt eine verkürzte Version der Status-Seite.

```
#: 1   fetches: 538   weight: 0.0994   error: 0.0994   tag: Pattern#1
inp:  0.00   0.00
hid:  0.86   0.99
tar:  0.00
out:  0.10
#: 2   fetches: 538   weight: 0.0985   error: 0.0985   tag: Pattern#2
inp:  0.00   1.00
hid:  0.02   0.79
tar:  1.00
out:  0.90
#: 3   fetches: 538   weight: 0.0990   error: 0.0990   tag: Pattern#3
inp:  1.00   0.00
hid:  0.02   0.79
tar:  1.00
out:  0.90
#: 4   fetches: 538   weight: 0.0999   error: 0.0999   tag: Pattern#4
inp:  1.00   1.00
hid:  0.00   0.11
tar:  0.00
out:  0.10

       2  input units
       2 hidden units
       1 output units

standard logistic function     (range [ 0, 1])
       4 patterns of     NORMAL data in buffer

    weightsum :    0.3968023
 global error :    0.0998649
(maximum norm)
```

(T) : Mit Hilfe dieses Befehls können während des Programmlaufs beliebige Text-Files auf dem Bildschirm gelistet werden. Dadurch ist es insbesondere möglich, die verschiedenen Ausgaben des Programms, die vielfach in Files geschrieben und nicht auf dem Bildschirm dargestellt werden, sofort anschließend (auch mehrmals) anzuschauen.

Praktischerweise verwendet dieser Befehl immer den zuletzt vom Programm benutzten File-Namen als Default, so daß der Name eines soeben erzeugten Ausgabe-Files nicht noch einmal eingetippt zu werden braucht.

Nach dem Verlassen des Haupt-Menüs mit "Q" wird, um ein irrtümliches Verlassen des Programmes zu verhindern (was z.B. bei der Rückkehr aus einem Unter-Menü ins Haupt-Menü sehr leicht passieren kann), noch einmal nachgefragt. Wird die Nachfrage mit "Y" beantwortet, beendet das Programm die Ausführung des Simulators.

Anschließend fragt das Programm, ob es neu gestartet werden soll. Dies erspart bei Versuchsserien das erneute Laden des Programms. Beim Neustart werden alle Variablen des Simulators neu initialisiert; einzige Ausnahme hiervon ist der Daten-Buffer, da dies bei

Versuchsserien das erneute Laden der Daten erspart. Eine entsprechende Meldung macht hierauf aufmerksam. Falls die Meldung “Still 0 patterns contained in buffer.” lautet, kann man sie ignorieren, das Programm ist damit bereits vollständig reinitialisiert.

```
Initializing...

Warning: Pattern buffer for learning not cleared!
Still 0 patterns contained in buffer.
Data status:       NONE

(Press <CR> to continue)

Random number seed:  353231313
```

Es ist Sache des Benutzers, die Übereinstimmung zwischen dem Format der eingelesenen Daten und der gewählten Anzahl von Input- und Output-Units zu garantieren. Sind die im Buffer noch vorhandenen Daten nach einem Neustart nicht mehr verwendbar, kann man sie durch den “C”-Befehl des Haupt-Menüs löschen.

Beantwortet man die letzte Frage des Programms mit “N” oder “Q”, wird die Programmausführung beendet.

```
                    (Q)  [Q]uit

     Please select: q

Are you sure you want to quit? (yes/no) y


Re-run program? (yes/no) q
```

Man beachte, daß ein einmal eingegebener File-Name eines bestimmten Typs (Matrix-File, Daten-File, Ausgabe-File etc.) zum Default-Namen für das darauffolgende Mal wird, wenn wieder nach einem File-Namen desselben Typs gefragt wird.

Man beachte außerdem, daß man mit Hilfe des Zeichens “*” den Default-Vorschlag für einen File-Namen in die eigene Eingabe einfügen kann. Folgen dem “*” noch Minuszeichen (“–”), werden dabei ebensoviele Zeichen vom rechten Ende des Default-Namens entfernt. So kann man beispielsweise aus dem Vorschlag “`AAT.MAT`” durch die Eingabe “`BEST_*---RESULT`” den Namen “`BEST_AAT.RESULT`” erzeugen.

Mit Hilfe des bereits weiter oben erwähnten “PLOTERR”-Programms lassen sich die während des Programmlaufs erzeugten Graphiken des Fehlerverlaufs anhand der vom Simulationsprogramm erstellten LOG-Files reproduzieren; zusätzlich automatisch skaliert und beschriftet, im Unterschied zur Graphik des Simulationsprogramms, die lediglich einen Anhaltspunkt über den Verlauf des (globalen) Fehlers während des Lernens geben soll (was beispielsweise das Steckenbleiben des Verfahrens sichtbar macht) und ein Maximum von Epochen (500) auf dem Bildschirm anzeigen soll.

An den Stellen, wo in der Simulation das Lernen dadurch unterbrochen wurde, daß die eingegebene maximal auszuführende Anzahl von Epochen erreicht war, wird in der Graphik eine gepunktete senkrechte Linie gezogen. Dies kann unter Umständen außerdem die Veränderung von Simulationsparametern bedeuten oder den Wechsel von einem Lernmodus in den anderen.

Eine senkrechte gestrichelte Linie zeigt an, wo mit Hilfe des Skeletonizing ("X"-Befehl des Haupt-Menüs) eine hidden Unit entfernt wurde.

Eine durchgezogene senkrechte Linie wird an den Stellen eingezeichnet, wo während der Simulation eine neue Gewichtsmatrix eingelesen wurde oder wo mit Hilfe des "K"-Befehls eine zuvor gespeicherte beste Matrix zurückgeholt wurde.

Das Programm beginnt eine neue Seite, sobald 450 bzw. 900 (bei halbierter Auflösung) Epochen geplottet sind. Eine Variante des "PLOTERR"-Programms namens "PLOTERRX" beginnt jedesmal an den Stellen eine neue Seite, wo beim "PLOTERR"-Programm eine durchgezogene senkrechte Linie eingezeichnet würde; dies ist zur Dokumentation von Versuchsserien oft zweckmäßiger.

Eine dritte Version des Programms namens "PLOTERROLD" ist ein einfacherer Vorläufer, der die LOG-Files früherer Versionen des Simulationsprogramms plottet (solche, bei denen die Werte des globalen Fehlers *nicht* mit "`@`" markiert sind).

Literaturverzeichnis

[1] Birmiwal, K., Sarwal, P., Sinha, S. (1989)
"A new gradient-free learning algorithm"
Proceedings of the 1989 IJCNN Conference
Southern Illinois University, Carbondale

[2] Chan, L.-W., Fallside, F. (1987)
"An adaptive training algorithm for back propagation networks"
Computer Speech and Language 2, 205–218

[3] Chauvin, Y. (1989)
"A Back-Propagation Algorithm with Optimal Use of Hidden Units"
In: D. Touretzky (ed.)
Advances in Neural Information Processing Systems 1, 519–526
Morgan Kaufman Palo Alto

[4] Denker, J., Schwartz, D., Wittmer, B., Solla, S., Howard, R., Jackel, L., Hopfield, J. (1987)
"Large Automatic Learning, Rule Extraction, and Generalization"
Complex Systems 1, 877–922

[5] Durbin, R., Rumelhart, D. E. (1989)
"Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks"
Neural Computation 1, 133–142

[6] Eckert, R., Randall, D.
Animal Physiology
W.H. Freeman & Co. 1983, 2nd edition

[7] Fahlman, S. E. (1988)
"Faster-Learning Variations on Back-Propagation: An Empirical Study"
Proceedings of the 1988 Connectionist Models Summer School
Carnegie Mellon University, Pittsburgh, PA

[8] „Chaos + Kreativität"
GEO Wissen 2 1990
Gruner + Jahr Hamburg

[9] Hanson, S. J., Pratt, L. Y. (1989)
"Comparing Biases for Minimal Network Construction with Back-Propagation"

In: D. Touretzky (ed.)
Advances in Neural Information Processing Systems 1, 177–185
Morgan Kaufman Palo Alto

[10] Hebb, D. O. (1949)
"The Organization of Behaviour"
Wiley New York
In: J. A. Anderson, E. Rosenfeld (eds.)
Neurocomputing. Foundations of Research.
MIT Press / Bradford Books

[11] Heileman, G. L., Georgiopoulos, M., Brown, H. K.
"The Minimal Disturbance Back-Propagation Algorithm"
University of Central Florida, Orlando

[12] Huber, W., Poeck, K., Weniger, D., Willmes, K. (1983)
Aachener Aphasie Test (AAT)
Hogrefe Göttingen

[13] Jacobs, R. A. (1988)
"Increased Rates of Convergence Through Learning Rate Adaptation"
Neural Networks 1, 295–307

[14] Kindermann, J., Linden, A. (1988)
"Draft: Learning from Partial Patterns and Pattern Completion with Back-Propagation"
GMD St. Augustin

[15] Linden, A. (1990)
„Untersuchung von Backpropagation in konnektionistischen Systemen"
Diplomarbeit an der Rheinischen Friedrich-Wilhelms-Universität Bonn
GMD St. Augustin

[16] Linder
Biologie. Lehrbuch für die Oberstufe.
J.B. Metzlersche Stuttgart 1980, 18. Auflage

[17] McClelland, J. L., Rumelhart, D. E. (1988)
Explorations in Parallel Distributed Processing:
A Handbook of Models, Programs, and Exercises
MIT Press / Bradford Books

[18] Minsky, M., Papert, S. (1969)
"Perceptrons"
MIT Press Cambridge, Mass.
In: J. A. Anderson, E. Rosenfeld (eds.)
Neurocomputing. Foundations of Research.
MIT Press / Bradford Books

[19] Mozer, M. C., Smolensky, P. (1989)
"Using Relevance to Reduce Network Size Automatically"
Connection Science 1, 3–16

[20] Peitgen, H.-O., Richter, P. H. (1985)
Schönheit im Chaos / Frontiers of Chaos
Forschungsgruppe Komplexe Dynamik
Universität Bremen

[21] Poeck, K. (Hrsg.)
Klinische Neuropsychologie
Thieme 1989, 2. Auflage

[22] Pollack, J. B.
"Implications of Recursive Distributed Representations"
Ohio State University, Columbus
(FTP Neuroprose archive)

[23] Rosenblatt, F. (1958)
"The perceptron: A probabilistic model for information storage and organization in the brain"
Psychological Review 65, 386–408
In: J. A. Anderson, E. Rosenfeld (eds.)
Neurocomputing. Foundations of Research.
MIT Press / Bradford Books

[24] Rumelhart, D. E., McClelland, J. L., and the PDP research group (1986)
Parallel Distributed Processing:
Explorations in the Microstructure of Cognition
Volume 1: Foundations
MIT Press / Bradford Books

[25] Rumelhart, D. E., McClelland, J. L., and the PDP research group (1986)
Parallel Distributed Processing:
Explorations in the Microstructure of Cognition
Volume 2: Psychological and Biological Models
MIT Press / Bradford Books

[26] Schmidhuber, J. (1989)
"Accelerated Learning in Back-Propagation Nets"
In: R. Pfeifer, Z. Schreter, F. Fogelman-Soulié, L. Steels (eds.)
Connectionism in Perspective
Elsevier Amsterdam

[27] Schmidt, P. (1987)
Skript „*Biologie für Mediziner*"
RWTH Aachen WS 87/88

[28] Schmidt, R. F., Thews, G.
Physiologie des Menschen
Springer 1980, 20. Auflage

[29] Silva, F. M., Almeida, L. B. (1990)
"Acceleration Techniques for the Backpropagation Algorithm"
In: L. B. Almeida, C. J. Wellekens (eds.)

Neural Networks
Proceedings of the 1990 EURASIP Workshop in Sesimbra, Portugal
Springer Berlin Heidelberg

[30] Sutton, R. S. (1986)
"Two Problems with Back-Propagation and other Steepest-Descent Learning Procedures for Networks"
Proceedings of the 8th Annual Conference of the Cognitive Science Society, 1986, 823–831

[31] Yu, Y.-H., Simmons, R. F. (1990)
"Descending Epsilon in Back-Propagation: A Technique for Better Generalization"
Proceedings of the 1990 IJCNN Conference
University of Texas, Austin