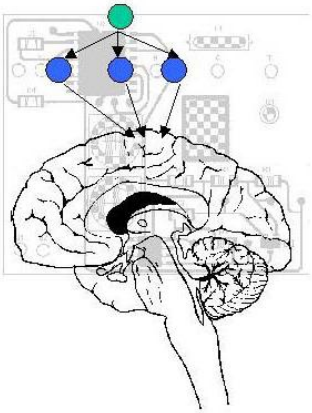
UNIVERSITÄT BAYREUTH
MATHEMATISCHES INSTITUT

Merkmalbasierte Zeichenerkennung mittels neuronaler Netze



Diplomarbeit

von

Lisa Sammer

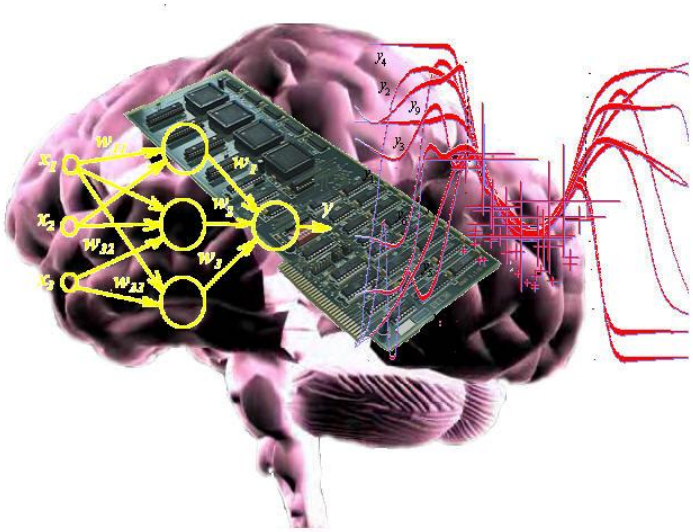
Datum: 10. Mai 2005

Aufgabenstellung / Betreuung:

Prof. Dr. Lars Grüne

Zweiter Gutachter:

Prof. Dr. Matthias Gerdts



Inhaltsverzeichnis

Abbildungsverzeichnis **7**

1. Einführung **9**
1.1. Idee der künstlichen neuronalen Netze . . . 9
1.2. Grundlagen aus der Biologie . . . 10
1.3. Modellierung eines neuronalen Netzes . . . 12
1.4. Motivation für neuronale Netze . . . 13
1.5. Geschichtliche Entwicklung der Forschung zu neuronalen Netzen . . . 14
1.5.1. Die Anfänge/erste Modelle: . . . 14
1.5.2. Die frühe Hochphase: . . . 15
1.5.3. Die Stagnation: . . . 16
1.5.4. Die Renaissance: . . . 18

2. Die Theorie der künstlichen neuronalen Netze **21**
2.1. Aufbau und Komponenten eines neuronalen Netzes . . . 21
2.1.1. Neuronen, Zellen . . . 22
2.1.2. Verbindungsstruktur zwischen Verarbeitungseinheiten . . . 23
2.1.3. Einführung in Netzstrukturen . . . 23
2.1.4. Propagierungsregel . . . 28
2.1.5. Kontrollstrategie . . . 29
2.1.6. Systemstatus und Aktivierung . . . 29
2.1.7. Lernregeln . . . 34
2.2. Standard Netzmodelle . . . 40
2.2.1. Perzeptron . . . 40
2.2.2. Adaline und Madaline . . . 45
2.3. Backpropagation . . . 47
2.3.1. Einführung und Prinzip . . . 47
2.3.2. Das Gradientenverfahren . . . 49
2.3.3. Backpropagation Lernprozess . . . 50
2.3.4. Herleitung . . . 53
2.3.5. Konvergenz von Backpropagation . . . 56

2.3.6. Probleme . . . 64
2.3.7. Modifikationen der Backpropagation Regel . . . 66
2.4. Neocognitron . . . 73
2.4.1. Architektur . . . 74
2.4.2. Lernstrategie (Training) . . . 78
2.5. Verkleinerung von Netzen / Pruning . . . 80
2.5.1. Weight Pruning . . . 81
2.5.2. Input pruning . . . 81
2.5.3. Weight Decay . . . 81
2.5.4. Magnitude Based Pruning . . . 82
2.5.5. Optimal Brain Damage (OBD) . . . 82
2.5.6. Optimal Brain Surgeon (OBS) . . . 84

3. Anwendung / Praxis 87
3.1. Anwendungen für mehrschichtige feedforward Netze mit Backpropagation . . . 87
3.2. Anwendungsgebiete . . . 87
3.3. Einführung in die Problemstellung der von mir entwickelten Software . . . 90
3.4. der SNNS . . . 91
3.5. Dokumentation meiner Software . . . 91
3.5.1. Vorverarbeitung und Laden der Bilder . . . 91
3.5.2. Geometrische Merkmale: . . . 92
3.5.3. Erstellen und Trainieren der neuronalen Netze . . . 96
3.5.4. Entscheidungskriterien bei Erkennung . . . 108
3.5.5. Ergebnisse . . . 109
3.6. Zusammenfassung und Ausblick . . . 112
3.6.1. Euphorie und Realität . . . 112
3.6.2. Fazit . . . 113

A. Anhang 115
.1. Kleines Deutsch-Englisches Fachwörterbuch . . . 115

Literaturverzeichnis 115

Abbildungsverzeichnis

1.1. Gehirn [13] . . . 10
1.2. Nervenzellen im Cortex [13] . . . 10
1.3. Synapse und Dendrite . . . 10
1.4. Neuron . . . 11
1.5. Synapse stark vergößert und idealisiert [13] . . . 11
1.6. D.O. Hebb [3] . . . 15
1.7. B. Widrow [3] . . . 16
1.8. M. Minsky [3] . . . 16
1.9. S. Papert [3] . . . 16
1.10. T. Kohonen [3] . . . 17
1.11. J. McClelland [3] . . . 17
1.12. J. Hopfield [3] . . . 18

2.1. biologisches Vorbild übertragen auf ein mathematisches Netzmodell [13] . . . 21
2.2. Aufbau eines einzelnen Verarbeitungselements [Rigoll] . . . 22
2.3. Netz mit zugehöriger Gewichtsmatrix [3] . . . 23
2.4. feedforward Netz 1. Ordung [3] . . . 24
2.5. feedforward Netz 2. Ordung [3] . . . 25
2.6. feedback Netz mit direkter Rückkopplung [3] . . . 26
2.7. feedback Netz mit indirekter Rückkopplung [3] . . . 26
2.8. feedback Netz mit Lateralverbindungen [3] . . . 27
2.9. feedback Netz mit vollständiger Vermaschung [3] . . . 27
2.10. verschiedene Propagierungsregeln [Scherer] . . . 28
2.11. bipolare Aktivierung . . . 31
2.12. Identität ($o_j = net_j$) . . . 31
2.13. semilineare Aktivierung . . . 31
2.14. sigmoide Aktivierung . . . 32
2.15. Fermi-Funktion . . . 32
2.16. binäre sigmoide Aktivierung . . . 33
2.17. binäre sigmoide Aktivierung . . . 33
2.18. Tangens hyperbolicus . . . 33

2.19. Überwachtes Lernen graphisch dargestellt [Rigoll] . . . 35
2.20. Fehlerfläche (in Abhängigkeit von 2 Gewichten) [Patterson] . . . 36
2.21. Unüberwachtes Lernen graphisch dargestellt [Rigoll] . . . 37
2.22. Quadratischer Fehler ϵ als Funktion des Gewichts w_i [Rigoll] . . . 39
2.23. Perzeptron [3] . . . 41
2.24. Funktionsschema des Perzeptrons . . . 42
2.25. Boolsche Funktion AND . . . 43
2.26. Boolsche Funktion OR . . . 43
2.27. Hyperebenen im $\mathbb{R}^2$ [20] . . . 43
2.28. Boolsche Funktion XOR . . . 44
2.29. [5] . . . 45
2.30. Adaline [3] . . . 46
2.31. [Rigoll] . . . 46
2.32. Gradientenmethode im $\mathbb{R}^2$ . . . 51
2.33. Gradientenmethode im $\mathbb{R}^3$ . . . 51
2.34. Stagnation in einem lokalen Minimum . . . 65
2.35. Stagnation auf einem Plateau . . . 65
2.36. Oszillation . . . 65
2.37. Überspringen eines Minimums . . . 65
2.38. Quickprop graphisch dargestellt [Zell] . . . 69
2.39. Gewichtsänderungen bei RProp [Zell] . . . 73
2.40. Neocognitron [10] . . . 75
2.41. S-Ebene mit S-Zellen [10] . . . 76
2.42. C-Ebene mit C-Zellen [10] . . . 77
2.43. hierarchische Merkmalsextraktion [10] . . . 78

3.1. Personalausweis (Muster) . . . 90
3.2. Zeichen im Orginalzustand, auf Standardgröße gebracht und in ein schwarz-weiß Bild gewandelt . . . 92
3.3. Zeichen mit 3 horizontalen Linien . . . 93
3.4. Zeichen ohne horizontale Linien . . . 93
3.5. Zeichen mit 2 vertikalen Linien . . . 93
3.6. Zeichen ohne vertikale Linien . . . 93
3.7. Zeichen mit Winkel von 40 Grad . . . 94
3.8. Zeichen ohne Winkel . . . 94
3.9. Zeichen mit vielen Elementen auf der Diagonale . . . 94
3.10. Zeichen mit wenigen Elementen auf der Diagonale . . . 94
3.11. Zeichen mit späten Schwarzwerten auf den mittleren Bildachsen . . . 94
3.12. Zeichen mit frühen Schwarzwerten auf den mittleren Bildachsen . . . 94
3.13. Zeichen mit Symmetrie bzgl. der horizontalen Achse . . . 95
3.14. nicht-symmetrisches Zeichen . . . 95

3.15. Zeichen mit Symmetrie bzgl. der vertikalen Achse 95
3.16. nicht-symmetrisches Zeichen 95
3.17. Zeichen mit eingezeichneten Schwerpunkten 96
3.18. schwarz-weiß Verhältnisse 96
3.19. untrainiertes neuronales Netz 97
3.20. SNNS Manager Panel 98
3.21. SNNS BigNet Panel 99
3.22. File Panel 100
3.23. Control Panel 100
3.24. Verlauf des Fehlers eines 1×31 feedforward Netzes mit zwei verdeckten Schichten bei Backpropagation 102
3.25. Null/Eins Netz 102
3.26. Verlauf des Fehlers eines 32×32 feedforward Netzes mit zwei verdeckten Schichten bei Backpropagation 103
3.27. Verlauf des Fehlers eines 1×31 feedforward Netzes mit zwei verdeckten Schichten bei Quickprop 104
3.28. Pruning Panel 105
3.29. General Parameters for Pruning 105
3.30. Netz vor und nach dem feedforward-Pruning 106
3.31. Plot der Ergebnisse 108

Abbildungsverzeichnis

1. Einführung

1.1. Idee der künstlichen neuronalen Netze

Neuronale Netze, oder auch künstliche neuronale Netze, sind massiv parallel agierende, informationsverarbeitende Systeme, die sich am Vorbild des menschlichen Gehirns orientieren. Sie sind sozusagen Modelle der Gehirnfunktion.
Man versucht, in Struktur und Funktionsweise Gehirnzellkomplexe nachzubilden und dadurch eine tragfähige Simulation komplexer menschlicher Denkvorgänge zu erzielen. (Ursprünglich wurden neuronale Netze entwickelt, um Abläufe im Gehirn besser zu verstehen.)
Sie bestehen aus einer großen Anzahl primitiver, uniformer Prozessoren, die stark untereinander vernetzt sind. Die Verarbeitungseinheiten kommunizieren miteinander, d.h. sie senden sich Informationen, in Form von Aktivierung der Zellen, über gerichtete Verbindungen zu.
Ebenso wie bei biologischen neuronalen Netzen besteht der Lernprozess in der Veränderung der synaptischen Verbindungen zwischen den Neuronen. Sie lernen - wie Menschen - an Beispielen und werden für bestimmte Aufgaben durch einen derartigen Lernprozess trainiert.

1.2. Grundlagen aus der Biologie

Das Gehirn empfängt über Sensoren (Sinnesorgane) Reize aus der Umwelt, die aufgenommen, verarbeitet und beantwortet werden.
Grundlegende Eigenschaften und Bausteine sind:

Abbildung 1.1.: Gehirn [13]

Abbildung 1.2.: Nervenzellen im Cortex [13]

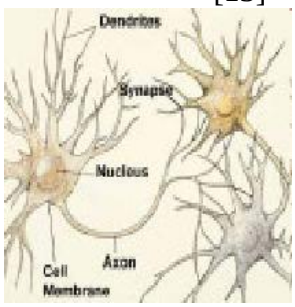


Abbildung 1.3.: Synapse und Dendrite

- Das Gehirn ist ein gigantisches Parallelverarbeitungssystem.
- Ort der Informationsverarbeitung ist die Hirnrinde (Neokortex).
- Verarbeitungseinheiten sind Neuronen (Zellkörper: Informationsträger), die elektrochemische Reize aus mehreren Quellen empfangen.
- Die Neuronen zeigen Reaktionen in Form von elektrochemischen Impulsen, die auf andere Neuronen übertragen werden.
- Das Axon ist der Ausgangskanal der Zelle und dient zur Weitervermittlung des Erregungszustands eines Zellkörpers.
- Der Dendrit ist der Eingangskanal der Zelle und dient zum Empfangen von Signalen.
- Die Synapse ist ein Übertragungspunkt zwischen den Neuronen und bestimmt die Auswirkung der Erregung, die über das Axon kommt, auf eine andere Zelle.
- Die Synapsen können erregend (größere Oberfläche, excitatiorisch) oder hemmend (kleinere Oberfläche, inhibitiorisch) sein.
- Über die Rezeptoren wird die Summe der synaptisch gefilterten Eingangserregungen von anderen Zellen auf einen Zellkörper übertragen.

Die Neuronen (Units, Verarbeitungseinheiten, Prozessoren) künstlicher neuronaler Netze sind im Vergleich zu ihren natürlichen Vorbildern stark idealisiert.
Nichtsdestotrotz gibt es viele **Übereinstimmungen**:

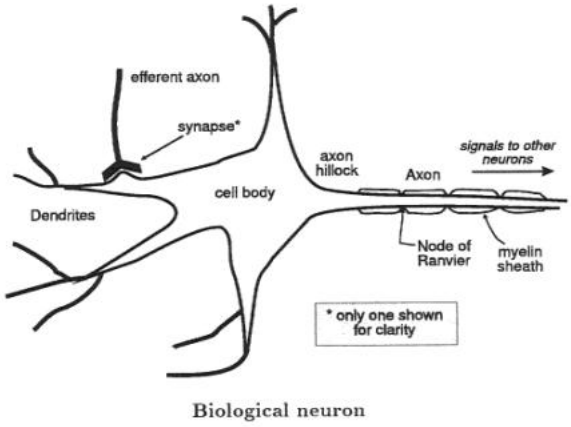


Abbildung 1.4.: Neuron

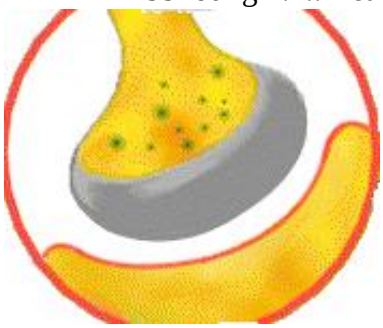


Abbildung 1.5.: Synapse stark vergößert und idealisiert [13]

- Die Neuronen arbeiten massiv parallel.
- Es handelt sich um relativ einfache Elemente.
- Neuronen verarbeiten die Aktivierungen der Vorgängerneuronen und die Stärke der Verbindungen zu einer Ausgabe.
- Die Neuronen sind durch gewichtete Verbindungen (Synapsen) miteinander verknüpft.
- Die Verbindungsgewichte bei künstlichen Neuronen sind modifizierbar, genauso wie die Plastizität der Synapsen bei dem biologischen Vorbild.
- Es herrscht eine hohe Konnektivität aufgrund der vielen Verbindungen unter den Neuronen.
- Das innere elektrische Potential einer Zelle oder auch Membranpotential bezeichnet man als Aktivität.
- Durch Empfang von Aktivitäten anderer Neuronen kann das Potential verstärkt oder abgeschwächt werden.
- Jedes Neuron hat einen Schwellenwert.
- Steigt das Aktionspotential über den Schwellenwert, so feuert die Zelle, d.h. eine Folge von Aktionspotentialen wird über das Axon ausgeschüttet.
- Diese Impulse erregen oder hemmen dann wiederum andere Neuronen.

Unterschiede zwischen biologischen und künstlichen Neuronen:

Künstliches neuronales Netz	**Biologisches Vorbild**
geringe Anzahl von Neuronen ($\mathbf{10^2}$ bis $\mathbf{10^4}$)	ca. $\mathbf{10^{11}}$ bis $\mathbf{10^{13}}$ Neuronen
geringe Anzahl von Verbindungen	höhere Anzahl von Verbindungen zw. den Neuronen (zwei- bis vierstellig)
Stärke einer Synapse wird ausschließlich durch das Gewicht bestimmt	Einfluss verschiedener Neurotransmitter auf die Stärke einer Synapse
numerischer Aktivierungswert (Amplitudenmodulation)	impulscodierte Informationsübertragung (Frequenzmodulation)
adressbezogene Speicherung	assoziative Speicherung
Schaltzeit eines Elements: ca. $\mathbf{10^{-9}}$ ms	Schaltzeit eines Elements: ca. $\mathbf{10^{-3}}$ ms
Schaltvorgänge insges.: ca. $\mathbf{10^{18}}$ pro sec	Schaltvorgänge insges: ca. $\mathbf{10^{13}}$ pro sec

Es gibt noch weitere Unterschiede zwischen künstlichen und biologischen Neuronen. Da jedoch Anpassungen an das biologische Vorbild wesentlich erhöhte Simulationsanforderungen mit sich bringen und sich sehr viele Probleme mit künstlichen neuronalen Netzen, die dem menschlichen Vorbild nur noch wenig ähneln, sehr gut lösen lassen, haben biologisch adäquatere Simulationen noch keine Vorteile gezeigt.

100 Schritt Regel: Dieses Beispiel zeigt die Notwendigkeit des massiven Parallelverarbeitung: Ein Mensch kann ein Bild einer bekannten Person, bzw. eines bekannten Gegenstandes in ca. 0,1 s erkennen. Bei einer Schaltzeit von 1 ms pro Neuron bedeutet das, dass die Erkennung in 100 Zeitschritten erfolgt. Vergleicht man diese Leistung mit einem (von Neumann) Rechner, dann wird deutlich, dass das Gehirn diesem weit überlegen ist, da dieser nach 100 sequentiellen Zeitschritten keinerlei Aussage treffen kann.

Bemerkung 1.1. *Man versucht sich von der Biologie anleiten zu lassen, aber nicht die biologischen Architekturen nachzumodellieren!*

1.3. Modellierung eines neuronalen Netzes

Definition 1.1. *Ein neuronales Netz ist ein Paar (N, V) mit einer Menge N von Neuronen und einer Menge V von Verbindungen. Es besitzt die Struktur eines gerichteten Graphen, für den die folgenden Einschränkungen und Zusätze gelten:*

- *Die Knoten des Graphen heißen Neuronen.*
- *Die Kanten heißen Verbindungen.*
- *Jedes Neuron kann eine beliebige Menge von Verbindungen empfangen, über die es seine Eingabe erhält.*
- *Jedes Neuron kann genau eine Ausgabe über eine beliebige Menge von Verbindungen aussenden.*

1.4. Motivation für neuronale Netze

Das Gehirn weist zahlreiche attraktive Eigenschaften auf, die die neuronalen Netze für sich gewinnen wollen.

Das Gehirn ist robust und fehlertolerant. Es ist flexibel und passt sich durch Lernen an neue Lebenssituationen an. Es arbeitet massiv parallel, verbraucht nur wenig Energie und kann mit unvollständigen und inkonsistenten Informationen umgehen.

Einige dieser Eigenschaften konnten auch bei neuronalen Netzen umgesetzt werden.
Durch die Lernfähigkeit werden sie anpassungsfähig in ihrem Verhalten. Sie werden mit einer großen Anzahl von Trainingsdaten gefüttert und können damit ihre Ausgaben bei veränderten Eingaben anpassen.
Sie haben eine verteilte Wissensrepräsentation, da sie ihr antrainiertes Wissen auf den im Netz verteilten Gewichten gespeichert haben. Ein Ausfall von einzelnen Subkomponenten kann aufgefangen werden. Dies gibt den Netzen eine höhere Fehlertoleranz im Vergleich zu herkömmlichen Algorithmen.
Die Speicherung von Information in einem neuronalen Netz erfolgt inhaltsbezogen, also assoziativ, nicht adressbezogen, wie es bisher meist der Fall war. Bei einer Eingabe eines Musters kann demnach leicht ein dazu ähnliches Muster bestimmt werden.
Neuronale Netze sind robuster gegen Störungen, z.B. verrauschten Daten. Bei ausreichendem Training können diese Fehler aufgefangen werden.
Durch adäquate Trainingsdaten und Lernstrategien können neuronale Netze Entscheidungsregeln ausbilden, deren Gültigkeit über den Trainingsprozess hinausgeht. Sie verfügen sozusagen über eine Art Generalisierungsfähigkeit.

Einer der Vorteile dieses Ansatzes liegt im Lernen anhand von Beispielen. Das neuronale Netz orientiert sich selbst, durch Training, an seiner Umwelt. Es arbeitet mit dem in Beispielen implizit

enthaltenem Wissen, es zeigt Strukturen auf, anstatt sie vorgeben zu lassen, es gibt wahrscheinliche Antworten, kann ungenaue oder widersprüchliche Daten handhaben und ergänzt teilweise fehlende Informationen. Es wird nicht für jedes Problem ein spezielles Programm erstellt, sondern das Netz selbst muss im Lernprozess die richtige Konfiguration finden.

Um ein Netz erfolgreich als Klassifikator einsetzen zu können, müssen viele verschiedene Faktoren beachtet werden, von denen ich allerdings nur die wichtigsten nennen werde.

- Trainingsdaten: Wenn das Netz sinnvolle Ergebnisse erzielen soll, insbesondere auch für Daten, die nicht in der Trainingsphase verwendet werden, so ist es notwendig, die Trainingsbeispiele repräsentativ auszuwählen.

- Merkmale: Bei der Auswahl der Merkmale für die Daten, mit denen das Klassifikationsproblem gelöst werden soll, muss mit besonders viel Sorgfalt gearbeitet werden.

- Richtiges Lernverfahren: In Abhängigkeit von den Anforderungen des vorliegenden Problems muss unter der Vielzahl von Lernverfahren genau abgewägt werden, welches Verfahren am besten geeignet ist.

- Validierung: Die Ausgabe des trainierten Netzes kann anhand von Testdaten geprüft werden. Dies trägt besonders zur Entscheidung unter den vielen Varianten des Trainings bei.

1.5. Geschichtliche Entwicklung der Forschung zu neuronalen Netzen

1.5.1. Die Anfänge/erste Modelle:

1943 beschrieben Warren McCulloch und Walter Pitts in ihrem Aufsatz ”A logical calculus of the ideas immanent in nervous activity” neurologische Netze, die auf dem McCulloch Pitts Neuron basieren. Dieses Werk handelt von den Eigenschaften eines einfachen binären Schwellenwerttyps eines Neurons mit zwei möglichen Statuszuständen (exzitatiorisch, oder inhibitorisch). Sie zeigten, dass auch einfache Klassen neuronaler Netze prinzipiell jede endliche Boolesche Funktion darstellen können. Diese Arbeit inspirierte weitere Forscher, wie z.B. Norbert Wiener und John von Neumann, auf diesem Gebiet Untersuchungen zu tätigen.

Abbildung 1.6.: D.O. Hebb [3]

1949 stellte Donald O. Hebb in seinem Buch ”The Organization of Behaviour” mit der Hebb’schen Lernregel als erster einen plausiblen Prozess für den neuronalen Lernvorgang vor. Diese Regel ist ein einfaches universelles Lernkonzept für individuelle Neuronen und ist noch heute Basis fast aller neuronalen Lernverfahren.

1.5.2. Die frühe Hochphase:

1957-1958 wurde der erste erfolgreiche Neurocomputer von Frank Rosenblatt, Charles Wightmann und Mitarbeitern am MIT entwickelt und für Mustererkennungsprobleme eingesetzt. Neben dieser technischen Leistung ist Frank Rosenblatt besonders durch sein Buch ”Principles of Neurodynamics” bekannt geworden. In ihm beschreibt er detailliert verschiedene Varianten des Perzeptrons und gibt auch einen Beweis dafür, dass das Perzeptron alles was es repräsentieren kann, durch das von ihm angegebene Lernverfahren lernen kann. Das grundlegende Perzepton Netzwerk war eine Schwellenwert-Logik, die aus drei Schichten bestand: einer (photo-) sensorischen Eingabeschicht, willkürlich mit einer Assoziationsschicht verbunden, welche wiederum mit einer Antwort- oder Klassifizierungs-Ausgabeschicht verknüpft ist.

Abbildung 1.7.: B. Widrow [3]

Mit der Publikation ”Adaptive switching circuits” von Bernard Widrow und Marcian E. Hoff wurde das Adaline vorgestellt. Dieses einfache neuronale Element, ähnlich dem Perzeptron, diente erstmals zur praktischen Anwendung neuronaler Netze. Die heute noch verwendeten Netzwerke, bestehend aus Adalines, nannte er Madalines.

Abbildung 1.8.: M. Minsky [3]

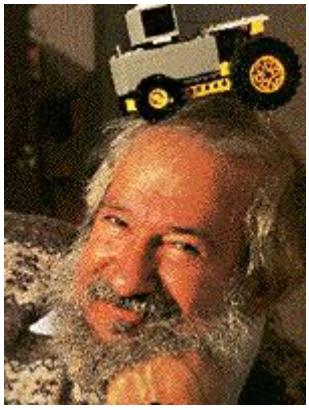


Abbildung 1.9.: S. Papert [3]

1.5.3. Die Stagnation:

1969 unternahmen Marvin Minsky und Seymour Papert in ihrer Arbeit ”Perceptrons” eine genaue mathematische Analyse dessen und zeigten, dass das Modell des Perzeptrons viele logische Funktionen gar nicht verarbeiten kann. Sie zogen die Schlussfolgerung, dass auch mächtigere Modelle die gleiche Problematik aufweisen würden und somit wurde das Buch der Todesstoß für die weitere Forschung zu neuronalen Netzen.

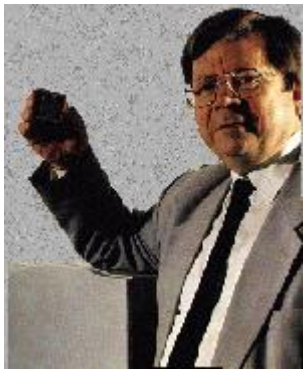


Abbildung 1.10.: T. Kohonen [3]

1972 stellte Teuvo Kohonen in seiner Veröffentlichung ”Correlation matrix memories” ein Modell eines speziellen Assoziativspeichers vor. Charakteristisch für dieses Modell ist die Verwendung linearer Aktivierungsfunktionen und kontinuierlicher Werte für Gewichte, Aktivierungen und Ausgaben.

Abbildung 1.11.: J. McClelland [3]

1974 entwickelte Paul Werbos in seiner Dissertation an der Harvard-Universität bereits das Backpropagation-Verfahren, das allerdings erst ca. 10 Jahre später durch die Arbeiten von Rumelhart und McClelland seine große Bedeutung erlangte.

Abbildung 1.12.: J. Hopfield [3]

1982 schrieb John Hopfield den einflussreichen Artikel ”Neural Networks and physical systems with emergent collective computational abilities” in dem er binäre, nach ihm benannte, Hopfield-Netze als neuronales Äquivalent der Ising-Modelle der Pysik untersuchte.

1983 stellten Fukushima, Miyake und Ito in ”Neocognitron” mit dem Neocognitron, einem hierarschischen feedforward Netzwerk, das durch überwachte oder nicht überwachte Methoden lernt, ein neuronales Modell zur positions-, größen- und abweichungsinvarianten Erkennung handgeschriebener Zeichen vor. Dieses war eine Erweiterung des schon 1975 entwickelten Cognitrons.

1.5.4. Die Renaissance:

1985 veröffentlichte John Hopfield den einflußreichen Artikel ”Neural Computation of Decisions in Optimization Problems” und zeigte darin, wie Hopfield Netze schwierige Optimierungsaufgaben (das traveling salesman problem) lösen können. Damit überzeugte er viele Forscher persönlich von der Wichtigkeit dieser Netze.

1986 fand das Gebiet der neuronalen Netze durch die Publikation des Lernverfahrens Backpropagation durch Rumelhart, Hinton und Williams einen besonderen Aufschwung. In zwei im gleichen Jahr veröffentlichten Artikeln ”Learning internal representations by error propagation” in dem von Rumelhart und McClelland herausgegebenen Buch ”Parallel Distributed Processing” und der Artikel in Nature: ”Learning representations by backpropagating errors” wurde mit Backpropagation ein, im Vergleich zu bisherigen Lernverfahren, sehr schnelles und robustes Lernverfahren vorgestellt. Diese Verfahren für mehrstufige, vorwärts gerichtete Netze ließen sich mathematisch elegant als Gradientenabstiegsverfahren des Netzwerkfehlers herleiten.

Seit 1986 hat sich das Gebiet geradezu explosiv entwickelt. Die Zahl der Forscher beträgt zur Zeit mehrere Tausend, es gibt eine Vielzahl von wissenschaftlichen Zeitschriften, die als Hauptthema neuronale Netze haben. Inzwischen existieren große anerkannte wissenschaftliche Gesellschaften, europäischer und internationale Fachgruppen, sowie Fachgruppen nationaler Informatik-Gesellschaften über neuronale Netze.

2. Die Theorie der künstlichen neuronalen Netze

2.1. Aufbau und Komponenten eines neuronalen Netzes

Definition 2.1. *Ein neuronales Netz ist ein Tupel mit folgenden Komponenten:*

1. *Menge von Knoten (Neuronen, Zellen, Units, Berechnungseinheiten, elementare Prozessoren) ($N = n_1, ..., n_n$), wobei Eingabeneuronen $I \subset N$ und Ausgabeneuronen $O \subset N$*
2. *Verbindungsnetzwerk der Knoten mit den Gewichten $\rightarrow \subset N \times N$ (gerichteter, gewichteter Graph)*
3. *Propagierungsregel für die Reihenfolge des Transports eines Inputs $net_j(t)$*
4. *Lernregel für das Training der Gewichte*

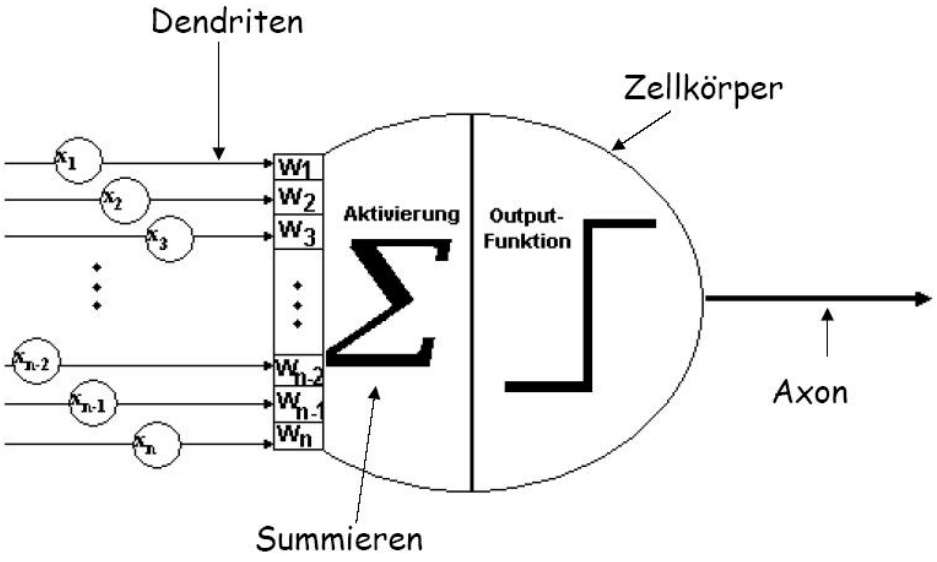


Abbildung 2.1.: biologisches Vorbild übertragen auf ein mathematisches Netzmodell [13]

2.1.1. Neuronen, Zellen

Definition 2.2. *Ein künstliches Neuron ist ein Abbildung* $o_i : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
(bzw. $o_i : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$*, falls kein Schwellenwert vorhanden) mit folgenden Komponenten:*

1. *Eingabevektor* $x = (x_1, x_2, ..., x_n)$
2. *Gewichtsvektor* $w = (w_1, w_2, ...w_m)$
3. *Aktivierungsfunktion* f_{act} *mit* $f_{act} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$
   (bzw. $f_{act} : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$*, falls kein Schwellenwert vorhanden)*
4. *Schwellenwert (Bias)* θ
5. *Ausgabefunktion* f_{out} *mit* $f_{out} : \mathbb{R}^n \rightarrow \mathbb{R}$

In dem Neuron wird für Netzeingabe net_i *und Aktivierung* a_i *mit Hilfe der Aktivierungsfunktion, der Ausgabefunktion und des Schwellenwerts eine Ausgabe erzeugt:*

$$(net_i, a_i) \rightarrow o_i = f_{out}(f_{act}(net_i, a_i, \theta_i))$$

dabei berechnet sich die Netzeingabe net_i *gemäß:*

$$net_j = \sum_i w_{ij} o_i \qquad bzw. \qquad net_j = \sum_i w_{ij} o_i - \theta_i$$

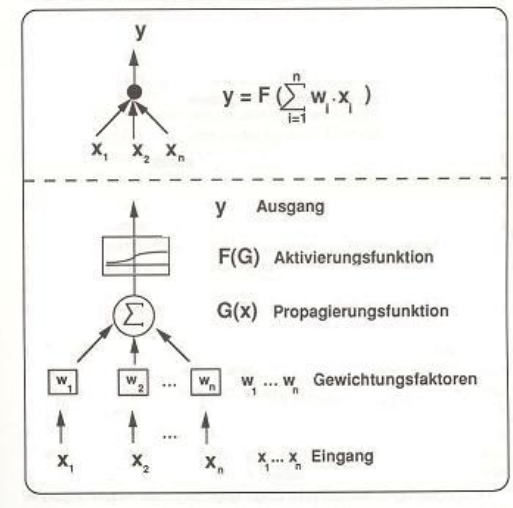


Abbildung 2.2.: Aufbau eines einzelnen Verarbeitungselements [Rigoll]

2.1.2. Verbindungsstruktur zwischen Verarbeitungseinheiten

Ein Verbund von mehreren Zellen wird als Schicht bezeichnet. Neuronen innerhalb von Schichten haben gleichartiges Verhalten und identische Rollen im Rahmen des Gesamt-Kommunikationsflusses. Auch Ein- und Ausgabezellen bilden jeweils eine Schicht.
Ein neuronales Netz kann als gerichteter, gewichteter Graph angesehen werden. Die Kanten stellen die Verbindungen zwischen den Neuronen dar und sind die Träger des Anwendungswissens eines neuronalen Netzes. (Gewichtungen des Informationsflusses = Codierung des Wissens)

Die Verbindungen in meinen Betrachtungen sind immer gerichtete Verbindungen, d.h. es handelt sich um eine definierte Ausrichtung des Informationsflusses.
w_{ij} ist dabei das Gewicht (weight) der Verbindung von Neuron n_i nach Neuron n_j.

- $w_{ij} > 0$ erregend (excitatory)
- $w_{ij} < 0$ hemmend (inhibitory)
- $w_{ij} = 0$ keine Verbindung vorhanden

Die Matrix der Verbindungen aller Zellen wird mit W bezeichnet und heißt Gewichtsmatrix.

Beispiel:

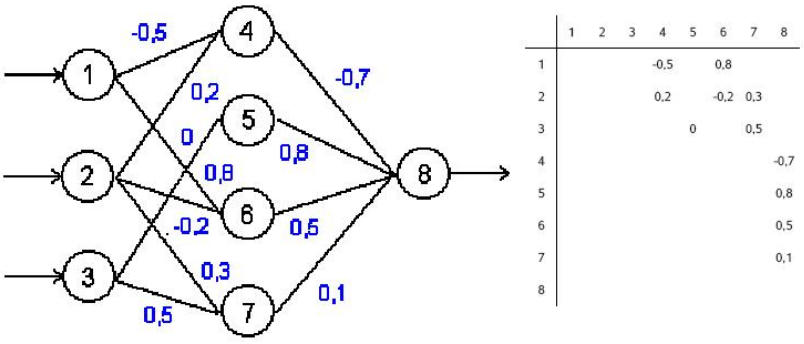


Abbildung 2.3.: Netz mit zugehöriger Gewichtsmatrix [3]

2.1.3. Einführung in Netzstrukturen

Neuronen werden zu einem netzartigen Objekt verbunden. Dieses kann als endlicher Graph gesehen werden.

Hierbei gibt es die Unterteilung in 2 Typen:

1. Netze ohne Rückkopplung (feedforward)
2. Netze mit Rückkopplung (feedbackward)

Netze ohne Rückkopplung (FF-Netze) Typische Eigenschaften der feedforward Netze:

- kontinuierliche Ein- und Ausgänge
- Verwendung der Sigmoid-Funktion als Aktivierungsfunktion
- meist unterschiedliche Dimension von Ein- und Ausgabevektoren
- Gewichtsmatrix: obere Dreiecksmatrix
- mathematische Beschreibung als statisches System (Berechnung des Ausgangsmusters erfolgt in einem einzigen Vorwärtsschritt)

feedforward Netz 1. Ordnung

Ein feedforward Netz 1. Ordnung hat nur gerichtete Verbindungen zwischen der Schicht $N(i)$ und der Schicht $N(i+1)$

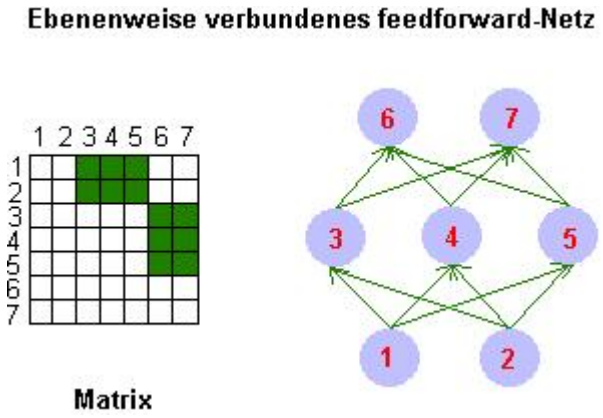


Abbildung 2.4.: feedforward Netz 1. Ordung [3]

feedforward Netz 2. Ordnung

Bei einem feedforward Netz 2. Ordnung haben Neuronen einer Schicht $N(i)$ nur Verbindungen zu Neuronen aus höheren Schichten $N(i+k)$ $(k > 0)$
(höhere Schicht = Schicht näher an der Ausgangsschicht)

Verbindungen, die nicht direkt zu der nächst höheren Schicht, sondern zu einer weiter entfernten Schicht laufen, heißen ”shortcut connections”. (Im Beispiel: von $\mathbf{1} \rightarrow \mathbf{6}$ und $\mathbf{2} \rightarrow \mathbf{7}$)

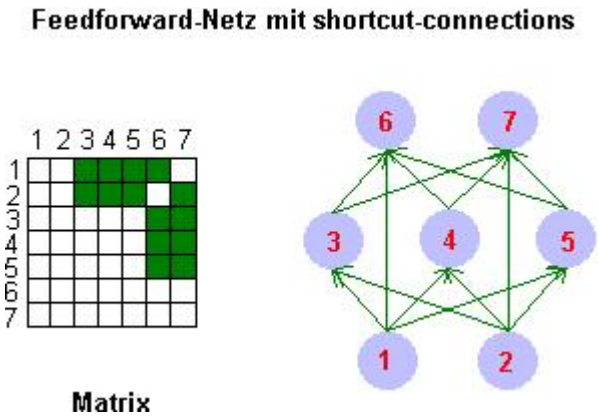


Abbildung 2.5.: feedforward Netz 2. Ordung [3]

Netze mit Rückkopplung In feedback Netzen gibts es eine Rückkopplung, die eine rekursive Berechnungsweise in mehreren Schritten zur Folge hat. Typische Eigenschaften der feedback Netze (nicht zwingend notwendig):

- oft nur eine Neuronenschicht
- meist gleiche Dimension von Ein- und Ausgabevektoren
- Treppenfunktion als Aktivierungsfunktion
- binäre Ein- und Ausgänge
- Hauptanwendung bei Assoziativspeichern

Direkte Rückkopplung Ein feedback Netz weist direkte Rückkopplung auf, wenn ein Neuron n_i seine Ausgabe sofort wieder als Eingabe erhält, die im nächsten Schritt erneut in die Berechnung des Aktivitätszustands einfließt. Diese Verbindungen bewirken oft, dass Neuronen den Grenzzustand ihrer Aktivität annehmen, weil sie sich selbst verstärken bzw. hemmen. (Im Beispiel: direkte Rückkopplung bei Neuronen 3 bis 7)

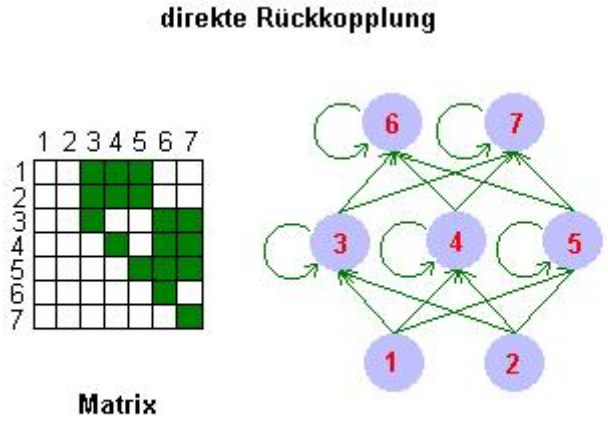


Abbildung 2.6.: feedback Netz mit direkter Rückkopplung [3]

Indirekte Rückkopplung Ein feedback Netz mit indirekter Rückkopplung hat Verbindungen von Schichten $N(j)$ zu $N(i)$, mit $i \leq j$ und gleichzeitig hat es auch Verbindungen von Neuronen von Schicht $N(i)$ zu Schicht $N(i+k)$ $(k > 0)$

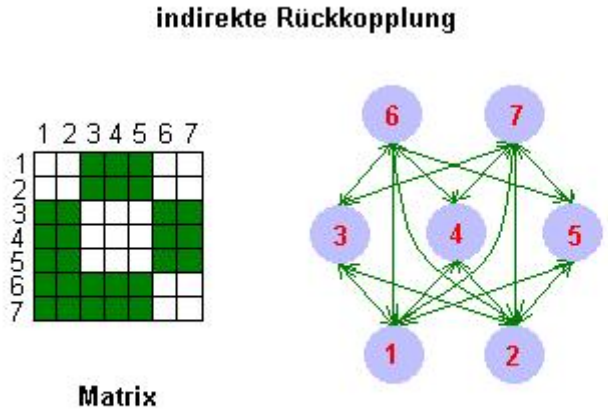


Abbildung 2.7.: feedback Netz mit indirekter Rückkopplung [3]

Rückkopplungen werden häufig verwendet, um die Aufmerksamkeit auf bestimmte Bereiche von Eingabeneuronen oder auf bestimmte Eingabemerkmale zu lenken.

Lateralverbindungen Netze mit Lateralverbindungen haben Rückkopplungen innerhalb einer Schicht. Das Ziel bei diesen Verknüpfungen ist, ein Neuron durch die aktivierende Rückkkopplung zu sich selbst, besonders hervorzuheben und gleichzeitig die Aktivierung benachbarter Neronen zu hemmen. (winner-takes-all)

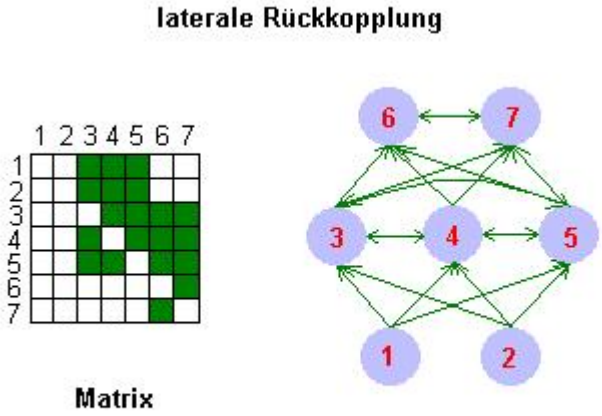


Abbildung 2.8.: feedback Netz mit Lateralverbindungen [3]

Vollständige Vermaschung Jedes Neuron n_i hat zu jedem Neuron n_j eine gerichtete Verbindung. (bekanntestes Beispiel: Hopfield-Netze)

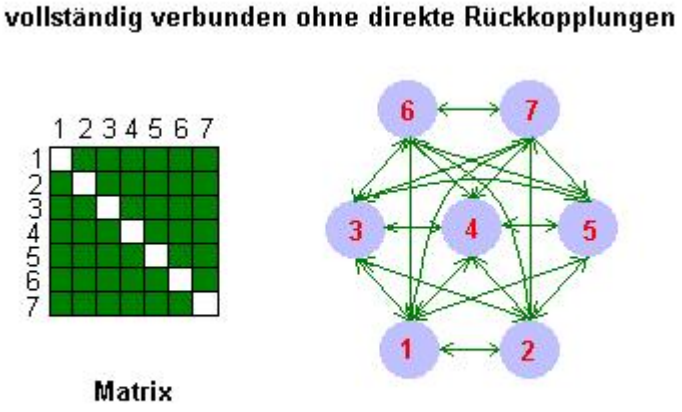


Abbildung 2.9.: feedback Netz mit vollständiger Vermaschung [3]

Die Gewichtsmatrix ist symmetrisch, die Hauptdiagonale ist mit 0 bestetzt.

2.1.4. Propagierungsregel

Die Propagierungsregel ist ein Modell für das Zusammenwirken der einzelnen Aktivierungen an den Eingängen des Neurons.
Diese Regel legt fest, wie sich die Netzeingabe einer Zelle aus den Ausgaben der anderen Zellen, zusammen mit den gewichteten Verbindungen errechnet. Normalerweise werden die Eingaben der Neuronen entsprechend den Gewichtungen der Eingangskanten aufsummiert.

Beispiel $net_j(t)$ bei Neuron n_j:

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

Oder auch:

$$net_j(t) = \sum_i (o_i(t) w_{ij}) - \theta_j$$

Die Netzeingabe berechnet sich also aus der Summe der Ausgaben $o_i(t)$ der Vorgängerzellen, multipliziert mit dem jeweiligen Gewicht w_{ij} der Verbindung von Neuron n_i nach Neuron n_j.

Es können jedoch auch andere Propagierungsregeln vorkommen:

Summation der gewichteten Eingabe

$$NET_j = \sum_{i=1}^{k} w_{ij} \cdot o_i$$

Maximalwert der gewichteten Eingabe

$$NET_j = \max(w_{ij} \cdot o_i)$$

Produkt der gewichteten Eingabe

$$NET_j = \prod_{i=1}^{k} w_{ij} \cdot o_i$$

Minimalwert der gewichteten Eingabe

$$NET_j = \min(w_{ij} \cdot o_i)$$

Abbildung 2.10.: verschiedene Propagierungsregeln [Scherer]

2.1.5. Kontrollstrategie

Die Kontrollstrategie gibt an in welcher Sequenz die Neuronen aktiviert werden.
Meist wird hierbei die Vorwärtsvermittlung benutzt (feedforward), eine Strategie, die die Aktivierung des Netzes schichtweise von Eingang bis Ausgang durchführt.
In seltenen Fällen wird eine andere Kontrollstrategie gewählt, z.B. eine selektive Kontrollstrategie (winner-take-all). Hier dürfen nur Neuronen mit bestimmten Aktivierungsverhalten/-eigenschaften ihre Aktivität weiter geben.

2.1.6. Systemstatus und Aktivierung

Aktivierungszustand und Aktivierungsregel

$$\boldsymbol{Aktivierungszustand}\begin{cases}\boldsymbol{kontinuierlich}\begin{cases}\boldsymbol{unbeschränkt}(\mathbb{R},\mathbb{N},...)\\ \boldsymbol{Intervalle}([0;1],...)\end{cases}\\ \boldsymbol{diskret}\begin{cases}\boldsymbol{binär}(\{0;1\},\{-1;1\},...)\\ \boldsymbol{mehrwertig}(\{-1;0;1\},...)\end{cases}\end{cases}$$

Der Aktivierungszustand Z hängt vom alten Zustand und der Veränderung der Aktivierungsfunktion ab.

Die Wahl des Wertebereichs für die Aktivierungen hat starken Einfluss auf die weiteren Design-Entscheidungen und ist bestimmend für die Charakteristik des Gesamtsystems.
So sind Netze mit diskreten Aktivierungszuständen zumeist schnelladaptierend, jedoch in ihren Klassifikationsfähigkeiten eingeschränkt.
Ein Neuron leitet seinen Aktivierungszustand durch die Verbindungsstruktur des Netzwerkes an die anderen Verarbeitungseinheiten weiter und umgekehrt. Die Aktivierung eines Neurons ist also von dem Zustand der anderen Zellen abhängig.

Die Aktivierungsregel gibt an, wie alle Inputs eines Neurons mit dessen momentanen Zustand zu einem neuen Zustand dieses Neurons kombiniert werden.

Die Aktivierungsregel stellt die Abhängigkeit zwischen dem eingehenden Nettoinput $\boldsymbol{net_i(t)}$ und dem Aktivationszustand $\boldsymbol{a_i(t)}$ zum Zeitpunkt $\boldsymbol{t}$ dar.
Folglich existiert für jede Unit $\boldsymbol{u_i}$ eine Aktivierungsfunktion:

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

oder auch:

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

und

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

Bias-Neuron

Jedes Neuron hat einen Schwellenwert (Bias, θ), ab dem es als aktiv angesehen wird.
Mathematisch gesehen, ist dies der Punkt, an dem monoton wachsende Aktivierungsfunktionen die größte Steigung haben.

Für die Positionierung des Bias sind zwei Varianten möglich:

- als Parameter innerhalb des Neurons, der bei der Berechnung der Aktivierung eingeht

$$net_j(t) = \sum_i o_i(t) w_{ij} \quad und \quad a_j(t+1) = f(a_j(t), net_j(t), \theta_j)$$

- als externes Neuron

$$net_j(t) = (\sum_i o_i(t) w_{ij}) - \theta_j \quad und \quad a_j(t+1) = f(a_j(t), net_j(t))$$

Aktivierungsfunktion

Die Aktivierungsfunktion legt fest, wie bei Neuron n_j ein Aktivierungszustand $a_j(t)$ zum Zeitpunkt t in einen Aktivierungszustand $a_j(t+1)$ zum Zeitpunkt $t+1$ übertragen wird.

Der Schwellenwert (Bias) ist ein Maß für die Tendenz eines Neurons n_j zur Aktivierung, bzw. Deaktivierung.

Folgende Aktivierungsfunktionen sind möglich:

- Identität oder lineare Abbildungen
- Schwellenwert- oder Treppenfunktionen
- Sigmoide Funktionen

Definition 2.3. *Eine Funktion* $s_c : \mathbb{R} \to [0,1]$ *heißt sigmoide Funktion wenn sie monoton wachsend und differenzierbar ist und wenn*

$$\lim_{\lambda \to \infty} s_c(\lambda) = k_2 \quad mit \quad k_2 < k_1$$

gelten.

einige konkrete Beispiele:

Perzeptronaktivierung

$$H(x) = \begin{cases} 1 & \text{für } x \geq 0 \\ 0 & \text{für } x < 0 \end{cases}$$

Bipolare Aktivierung

$$sgn(x) = \begin{cases} 1 & \text{für } x \geq 0 \\ -1 & \text{für } x < 0 \end{cases}$$

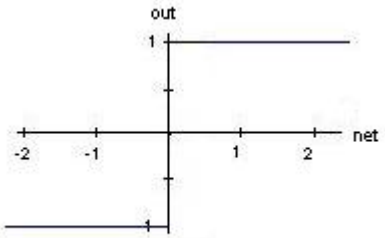


Abbildung 2.11.: bipolare Aktivierung

Identität

$$id(x) = x$$

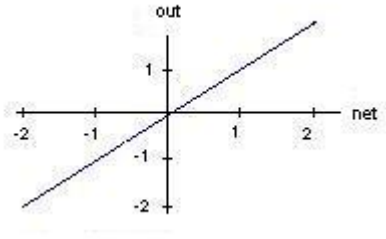


Abbildung 2.12.: Identität ($o_j = net_j$)

Semilineare Funktion

$$lin(x) = \begin{cases} 1 & \text{für } x \geq 1 \\ x & \text{für } -1 < x < 1 \\ -1 & \text{für } x \leq -1 \end{cases}$$

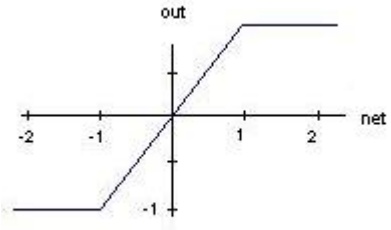


Abbildung 2.13.: semilineare Aktivierung

Sigmoide Funktion allgemein:

$$sgd(x) = \frac{1}{1 + e^{-(x+a)}}$$

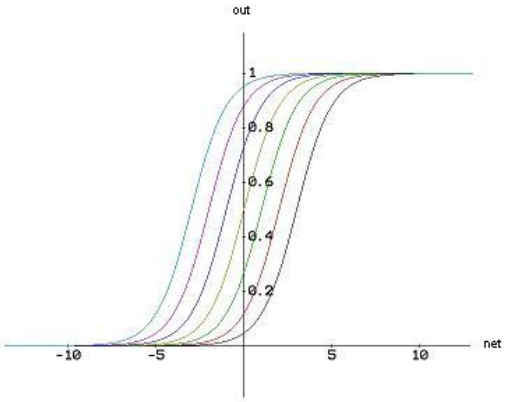


Abbildung 2.14.: sigmoide Aktivierung

Fermi-Funktion:

$$sgd(x) = \frac{1}{1 + e^{-x}}$$

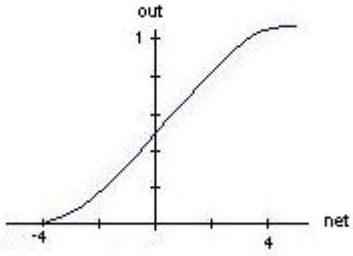


Abbildung 2.15.: Fermi-Funktion

binäre sigmoide Funktion

$$sgd(x) = \frac{1}{1 + e^{-\lambda x}}$$

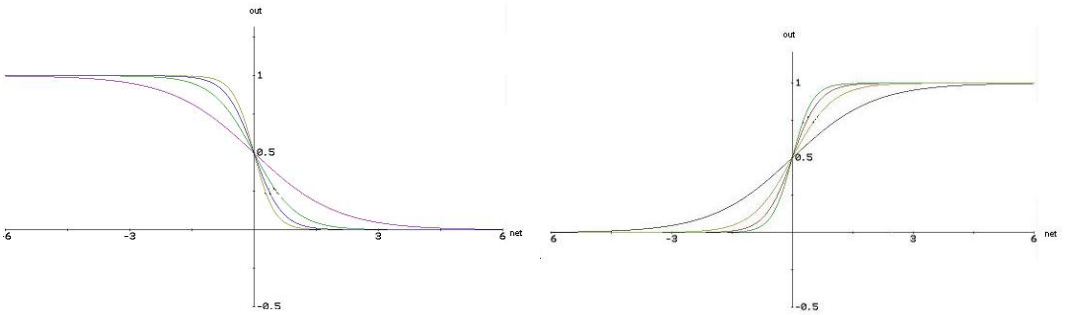


Abbildung 2.16.: binäre sigmoide Aktivierung

bipolare sigmoide Funktion

$$sgd(x) = \frac{2}{1 + e^{-\lambda x}} - 1$$

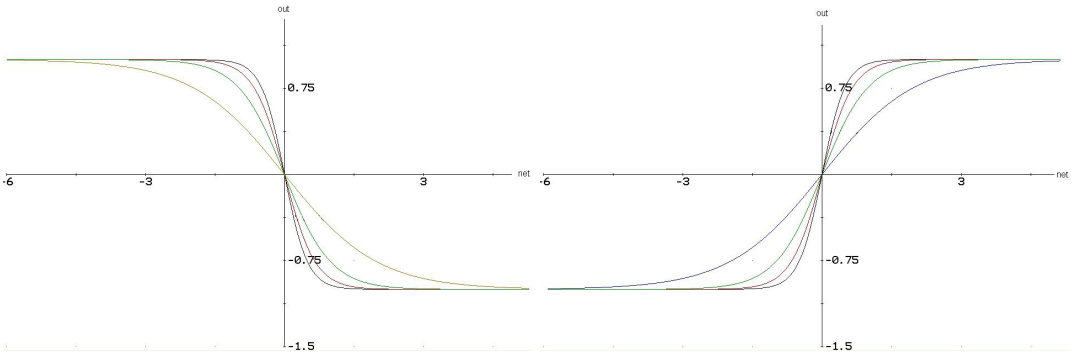


Abbildung 2.17.: binäre sigmoide Aktivierung

Tangens hyperbolicus

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

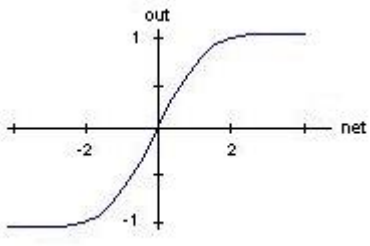


Abbildung 2.18.: Tangens hyperbolicus

Bemerkung 2.1. *Eine lineare Aktivierungsfunktion wird nur verwendet, wenn das Netz keine verdeckten Schichten aufweist.*
Wird als Aktivierungsfunktion die Identität gewählt, so ist die Ausgabe gleich der Aktivierung und der Wertebereich der Aktivierungsfunktion entspricht dem Wertebereich des Aktivierungszustands. Gewöhnlich wird für die Aktivierungsfunktion in Netzen mit verdeckten Schichten eine monoton wachsende, differenzierbare Funktion gewählt, wie z.B. die s-förmige Sigmoid Funktion. (Sie ist den in der Natur beobachteten Nichtlinearitäten bei biologischen neuronalen Netzen am ähnlichsten.)

Ausgabefunktion / Ausgaberegel

Die Ausgabefunktion f_{out} bestimmt aus der Aktivierung die Ausgabe des Neurons.
Sie legt den Ausgabewert eines Neurons, in Abhängigkeit vom aktuellen Aktivierungszustand, fest.

$$o_j = f_{out}(a_j)$$

2.1.7. Lernregeln

Ein neuronales Netz ”lernt”, indem es, gemäß einer festen Vorschrift, der Lernregel, modifiziert wird.
Dabei werden die Gewichte aufgrund der ”Erfahrungen” des Netzes in der Anwendungsumgebung adaptiert.
Mögliche Modifikationen:

1. Entwicklung neuer Verbindungen
2. Löschen existierender Verbindungen
3. Verändern der Gewichte
4. Verändern des Schwellenwertes
5. Verändern der Aktivierungs- bzw. Ausgabefunktion
6. Entwicklung neuer Neuronen
7. Löschen bestehender Neuronen

Von den gegebenen Möglichkeiten wird die dritte, also das Lernen durch Veränderung der Gewichte, am häufigsten verwendet. Die Punkte 4 bis 6 sind leider dem Netz-Designer frei überlassen und

keiner konkreten Regel unterlegen. Erst in letzter Zeit haben Verfahren, die auch eine Veränderung der Topologie beinhalten, an Bedeutung gewonnen.
Eine Studie hierzu findet man in [Herrmann].

Alle Lernregeln, die hier vorgestellt werden, beziehen sich auf Netze ohne verdeckte Schichten.

Das Lernen kann in folgender Weise unterteilt werden:

$$\boldsymbol{Lernen} \begin{cases} \text{überwachtes Lernen (supervised learning)} \begin{cases} \text{Ein- und Ausgabelernen} \\ \text{vollständiges Lernen} \end{cases} \\ \text{unüberwachtes Lernen (unsupervised learning)} \end{cases}$$

Die Parametrisierbarkeit ist allen Lernmethoden gemeinsam. Sie beeinflusst die Stärke der Auswirkung einer Fehlleistung des Netzes auf die Gewichtskonfiguration.
Der wichtigste Parameter hierbei ist die Lernrate η. Allerdings bleibt deren Wahl intuitiv, da es keine abgesicherte Methode zur Bestimmung von η gibt.

Überwachtes Lernen (supervised learning)

Beim überwachten Lernen ist ein externer Lehrer anwesend, der dem Netz zu jeder Eingabe die erwünschte Antwort oder die Differenz der tatsächlichen zur korrekten Ausgabe vorgibt (Lernfehler/Netzfehler). Anhand dieser Differenz wird dann das Netz über die Lernregel modifiziert.
Die Fehlergröße dient zur Veränderung der Gewichte, die variabel und dem aktuellen Lernfortschritt angepasst ist. Aus der Anpassung der Netzwerkparameter (Gewichte) soll eine Performanceverbesserung, also ein kleinerer Fehler für die einzelnen Trainingspaare, erzielt werden.

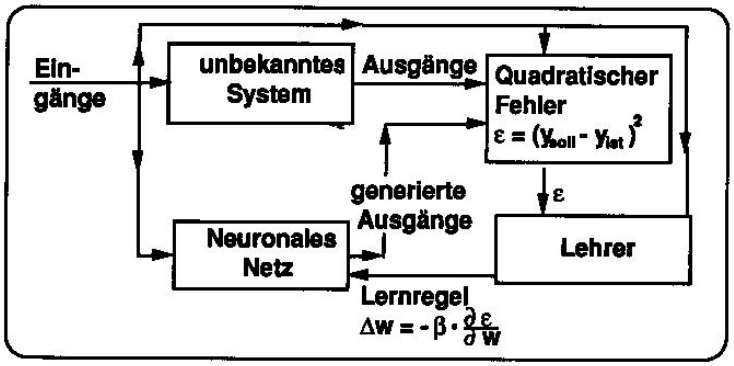


Abbildung 2.19.: Überwachtes Lernen graphisch dargestellt [Rigoll]

Ziel ist es also diesen Netzfehler auf einen möglichst kleinen Wert zu reduzieren. Mathematisch gesehen, stellt das Lernen ein Minierungsverfahren dar. Man will diejenigen variablen Verbindungsgewichte finden, die den ”Gesamtfehler” minimieren.

Graphisch gesehen, sucht man auf der Fehlerfläche des Netzes, aufgespannt durch die Gewichte, nach dem globalen Minimum.
Man erhält also ein Optimierungsproblem in den Variablen (Gewichten), über den Netzwerkfehler summiert über alle Trainingsmuster.
Minimiere:

$$E(w) = \frac{1}{2} \sum_{p} \sum_{i, i \in O} (t_{p,i} - o_{p,i}(w))^2$$

mit:
$t_{p,i}$: Sollausgabe des i-ten Ausgabeneurons bei Muster p
$o_{p,i}$: Ist-Ausgabe des i-ten Ausgabeneurons bei Muster p in Abhängigkeit der Gewichte

Abbildung 2.20.: Fehlerfläche (in Abhängigkeit von 2 Gewichten) [Patterson]

Um dieses Optimierungsproblem zu lösen, verwendet man beispielsweise Varianten des Gradientenverfahrens (2.3.2).
Diese Verfahren erzeugen Iterierte mit folgender Gestalt $w(t+1) = w(t) + \eta \cdot \delta w(t)$
$(t = 0, 1, ...)$

In der Praxis gibt man sich zufrieden, sobald die Gewichtungen gegen eine Wertemenge, mit der die erforderlichen Mustervorgarben erzielt werden können, konvergieren.
Das Lernen ist daher abgeschlossen, sobald der Fehler zwischen Ist- und Sollausgabe auf einen akzeptablen Wert minimiert wurde. Man findet meist nur ein lokales Minimum, was aber normalerweise ausreicht.

Nachteil:
Diese Technik setzt voraus, dass Trainingsdaten existieren, die aus Paaren von Ein- und Ausgabedaten bestehen.

Verstärkendes Lernen (reinforcement learning)

Das Grundprinzip hierbei ist eine ”unscharfe Beruteilung” des Klassifizierungserfolgs.
Im Gegensatz zum überwachten Lernen wird dem Netz hier, wiederum von einem externen Lehrer, lediglich mitgeteilt, ob die Ausgabe richtig oder falsch war. Die korrekte Antwort wird nicht präsentiert und somit wird auch der exakte Wert der Differenz nicht ermittelt.
Die Richtig/Falsch Information wird in das Netz rückgekoppelt, um damit die Performance zu verbessern.
Hierbei geht man mit folgender Strategie vor:

- Gewichtungen für Einheiten, die die richtige Antwort erzeugt haben, werden erhöht.
- Gewichtungen, die die falsche Antwort erzeugt haben, werden reduziert.

Unüberwachtes Lernen (unsupervised learning)

Das unüberwachte Lernen zählt nicht zu den populären Lernmethoden.
Hierbei gibt es keinen Lehrer und somit auch kein Feedback von außen. Daher heißt dieses Lernparadigma auch self-organized learning.
Das Netz strebt an, eigenständig die präsentierten Daten zu klassifizieren. Es extrahiert statistische Gleichmäßigkeiten der Trainingsbeispiele und passt sich an diese an. Ähnliche Eingaben werden also selbständig identifiziert und sodann auch einheitlich bewertet.

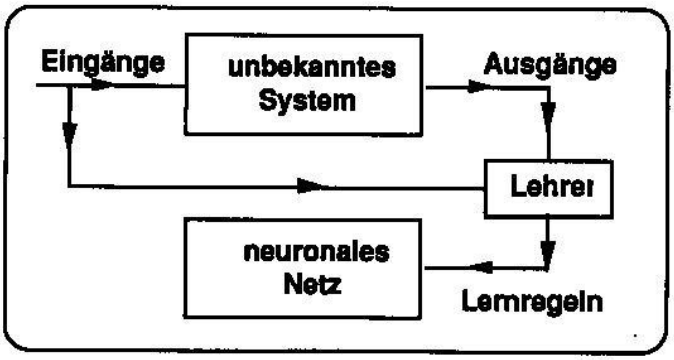


Abbildung 2.21.: Unüberwachtes Lernen graphisch dargestellt [Rigoll]

Hierbei werden durch Zufallszahlen ausgewählte Gewichtungen (die auf die meisten Eingabesignale reagieren) verstärkt und andere abgeschwächt.
Durch eine Bewertungsfunktion wird festgestellt, ob der neu berechnete Output (mit den veränderten Gewichten) besser ist, als der vorangegangene. In diesem Fall werden die modifizierten Gewichte gespeichert, andernfalls vergessen.
Bekannte Netze, die mit dieser Lernmethode arbeiten sind das Hammingnetz, das Neocognitron, oder auch das Kohonennetz.

Beispiele für konkrete Lernregeln

Hebb'sche Lernregel Idee: Bei starker, gleichzeitiger Aktivität von zwei Neuronen wird das Gewicht dieser Kantenverbindung erhöht.

Algorithmus nach Hebb:

Wenn ein Axon der Zelle A nahe genug ist, um eine Zelle B zu erregen und wiederholt oder dauerhaft sich am Feuern beteiligt, geschieht ein Wachstumsprozess oder metabolische Änderung in einer oder beiden Zellen dergestalt, dass A's Effizienz als eine der auf B (...) feuernden Zellen anwächst.

Übertragen auf das mathematische Modell bedeutet das:
Wenn eine Neuron n_j eine Eingabe von Neuron n_i erhält und beide gleichzeitig stark aktiviert sind, dann erhöhe das Gewicht w_{ij}. (verstärke die Verbindung von Neuron n_i zu Neuron n_j)

Gleichung für die Gewichtsänderung:

$$\triangle w_{ij} = \eta \cdot o_i \cdot a_j \quad und \quad w_{ij}(t+1) = w_{ij}(t) + \triangle w_{ij}$$

mit $\triangle w_{ij} =$ Änderung des Gewichts w_{ij}
$\eta =$ konstante Lernrate
$o_i =$ Ausgabe von Neuron n_i (vorher)
$a_j =$ Aktivität des nachfolgenden Neurons n_j

Die Gewichtsänderung ist abhängig von:

- der konstanten Lernrate
- der Ausgabe des Vorgängerneurons
- der Aktivitätsfunktion

Die Hebb'sche Lernregel stellt im Wesentlichen ein Kovarianz- oder Korrelationsbeziehung zwischen Ein- und Ausgabe dar.
$\Rightarrow$ Ein- und Ausgabemuster müssen zur Verfügung stehen. Die Brauchbarkeit hängt vom Korrelationsgrad zwischen den Eingabemustern ab.

Problem bei Hebb: Wenn η eine Konstante ist, können die Gewichtungen ins Unendliche wachsen (die Zellen kennen kein Vergessen). Es müssen also für die Eingabemuster oder die Gewichtungen Beschränkungen vorgegeben werden, bzw. ein Term für das Vergessen eingeführt werden.

Delta Regel Die Delta Regel (auch Widrow-Hoff-Regel) ist ein koordinatenweiser Abstieg. Die Gewichtungen werden so angepasst, dass die quadrierten Fehler über alle Muster minimiert werden (Last Mean Square - Methode). Es soll also möglichst schnell w_{ij} minimiert werden, so dass ein (wenn möglich globales) Minimum auf der Fehlerfläche gefunden wird. Man gibt sich allerdings meist mit einem lokalen Minimum zufrieden.

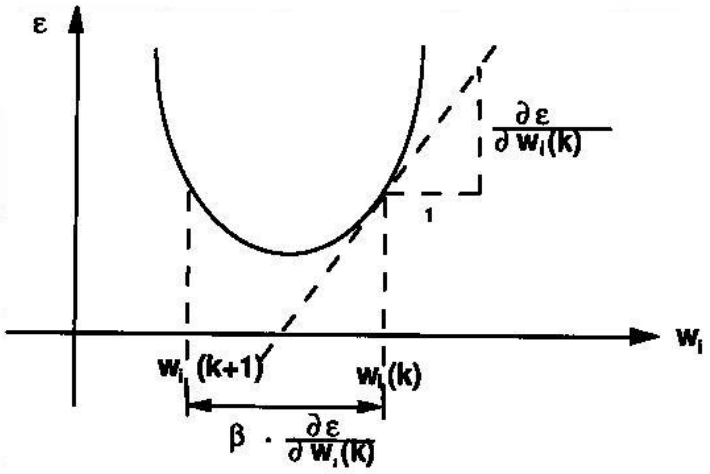


Abbildung 2.22.: Quadratischer Fehler ϵ als Funktion des Gewichts w_i [Rigoll]

Die Gleichung für die Gewichtsänderung lautet:

$$\triangle w_{ij} = \eta \cdot o_i \cdot \triangle a_j \qquad mit \qquad \triangle a_j = y_j - net_j$$

hierbei ist $\triangle w_{ij}$ die Gewichtsänderung, abhängig von:
- der Lernrate η
- der Ausgabe von Neuron n_i
- $\triangle a_j$, der Differenz zwischen Solloutput y_j und dem Netzinput net_j

Modifizierte Delta Regel:

$$\triangle w_{ij} = \eta \cdot o_i \cdot (y_j - o_j)$$

Ablauf: Kann zur Trainingszeit eine Abweichung zwischen der Ziel und Sollausgabe des Netzes festgestellt werden, so müssen die internen Gewichte der Verbindungen zwischen Ein- und Ausgabeschicht in der oben angegebenen Weise modifiziert werden.
Durch iterative Anwendung der Delta-Regel soll die Netzausgabe gegen die gewünschte Ausgabe konvergieren.

Eine weitere Modifikation der Delta Regel ist die Backpropagation Regel. Ich werde allerdings erst in Kapitel 2.3. genauer auf diese eingehen.

2.2. Standard Netzmodelle

In diesem Kapitel möchte ich Perzeptron und Adaline vorstellen. Diese beiden Netze haben in der Geschichte den Begriff der linearen Separierbarkeit stark geprägt.

Historisch gesehen handelte es sich bei dem Perzeptron um ein zweistufiges Netz, bei dem die Gewichte der untersten Stufe konstant und die der oberen Stufe lernfähig sind. Rosenblatt verwendete das sogenannte Photo-Perzeptron zur Klassifizierung visueller Muster, die die menschliche Retina liefert. Heutzutage bezeichnet man ein einstufiges, lernfähiges Netz schon als Perzeptron. Dieses ist wiederum sehr ähnlich zum Adaline. Die Unterschiede bestehen lediglich darin, dass das Perzeptron einen In- und Output von 0 und 1 hat und die Gewichtsänderungen beim Lernen mit den exakten Ausgabewerten $f(net)$ statt nur mit net berechnet werden.

2.2.1. Perzeptron

Das Perzeptron wurde erstmals von Rosenblatt entwickelt und wurde später auch durch Minsky und Papert bekannt.
Neuronale Netze mit folgender Struktur heißen Perzeptron: Die Eingabezellen sind durch feste, gewichtete Verbindungen alle mit einer Schicht von Ausgabeneuronen verbunden. Diese Neuronen können einfache Muster erkennen. Die Eingangsneuronen haben gewichtete Verbindungen zu einem weiteren Neuron, das als Klassifikator wirkt und angibt, ob das anliegende Muster erkannt wird. Die Verbindungen von der ersten Verarbeitungsschicht zum Ausgabeneuron sind trainierbar, also variabel. Weil es nur eine Ebene einstufiger Gewichte gibt, handelt es sich um ein einstufiges Netz mit nur einer trainierbaren Schicht.

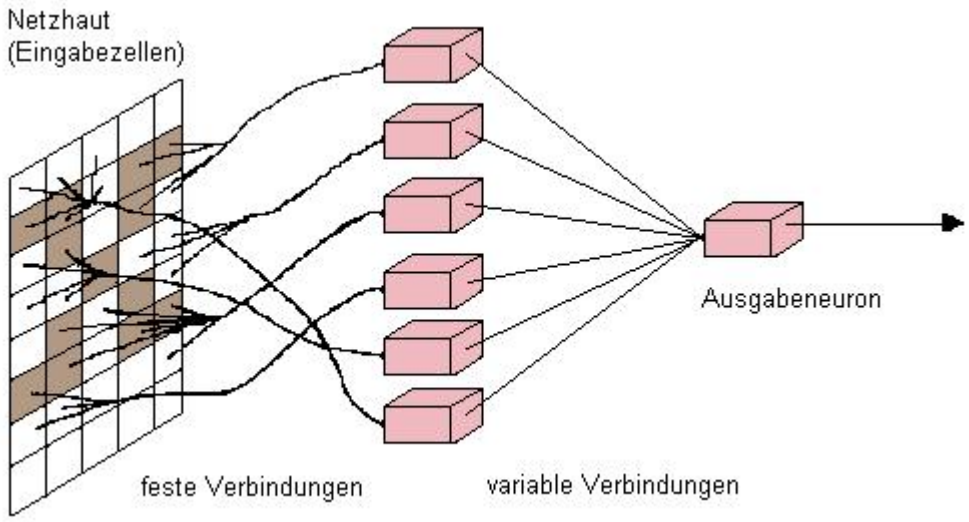


Abbildung 2.23.: Perzeptron [3]

Aufbau des Perzeptrons:

- Eingabezellen: binäre Netzeingabe
- $net_j = \sum w_{ij} o_j$
- binäre Schwellenwertfunktion
- Bias/Schwellenwert θ_j
- keine explizite Ausgabefunktion
- Ausgabe des Neurons n_j:

$$o_j = \begin{cases} 0 & \text{falls} \quad net_j \geq \theta_j \\ 0 & \text{sonst} \end{cases}$$

(beliebig viele Ausgabeneuronen)

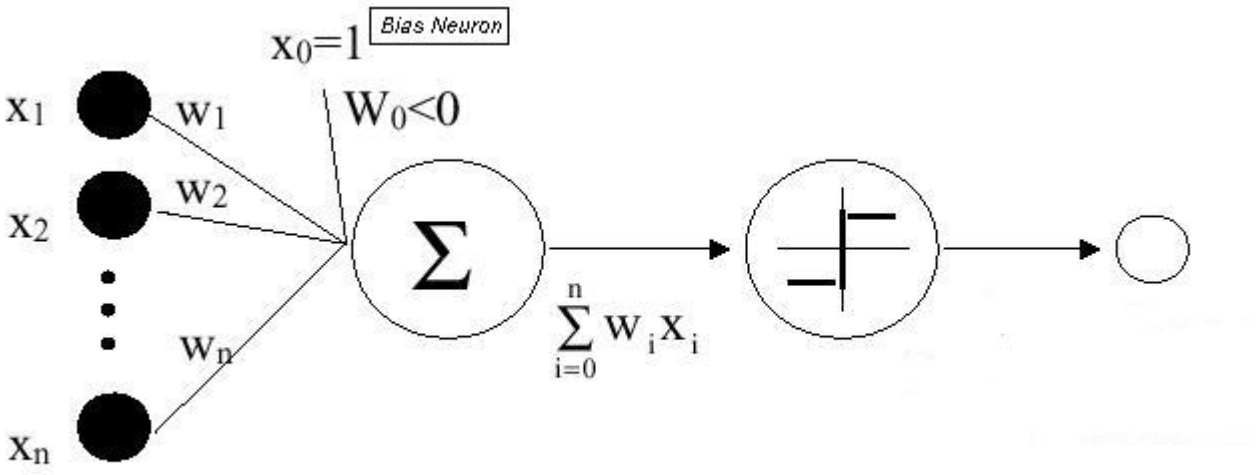


Abbildung 2.24.: Funktionsschema des Perzeptrons

Das Perzeptron ist ein typischer Anwendungsfall für das supervised learning. Die Lernregel lautet:

$$w_{ij}(t+1) = w_{ij}(t) + \eta \cdot (y_j - a_j) \cdot x_i$$

y_j = gewünschte Ausgabe
a_j = vom Netz erzeugte Ausgabe
x_i = Netzeingabe
η = Lernrate

Ist die tatsächliche Ausgabe gleich der erwarteten Ausgabe, so ist keine weitere Veränderung der Gewichte nötig. Andernfalls erfolgt eine Korrektur in Richtung des gewünschten Outputs durch $\triangle = \eta \cdot x_i$.

Die vom Perzeptron tatsächlich lösbaren Problemklassen sind stark eingeschränkt. Es können nur linear separierbare Mengen, also Mengen, die durch eine Hyperebene trennbar sind, dargestellt werden.
Beispielsweise können die Boolschen Funktionen AND (NAND), OR (NOR) problemlos realisiert werden.

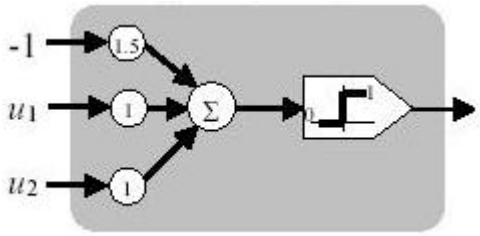


Abbildung 2.25.: Boolsche Funktion AND

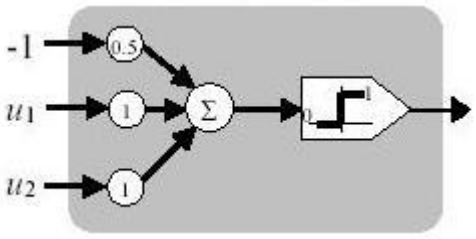


Abbildung 2.26.: Boolsche Funktion OR

Die XOR Funktion kann allerdings mit einem einstufigen Perzeptron nicht dargestellt werden. Um die Funktion realisieren zu können, müssten folgende Ungleichungen erfüllt sein:
$x_1 = 0, x_2 = 0, w_1 x_1 + w_2 x_2 = 0 \Rightarrow 0 < \Theta$
$x_1 = 0, x_2 = 1, w_1 x_1 + w_2 x_2 = w_2 \Rightarrow w_2 \geq \Theta$
$x_1 = 1, x_2 = 0, w_1 x_1 + w_2 x_2 = w_1 \Rightarrow w_1 \geq \Theta$
$x_1 = 1, x_2 = 1, w_1 x_1 + w_2 x_2 = w_1 + w_2 \Rightarrow w_1 + w_2 < \Theta$
Schon allein bei der Betrachtung des zu lösenden Gleichungssystems wird klar, dass die Gleichungen 2 und 3 nicht gleichzeitig mit der Gleichung 4 verifiziert werden können.

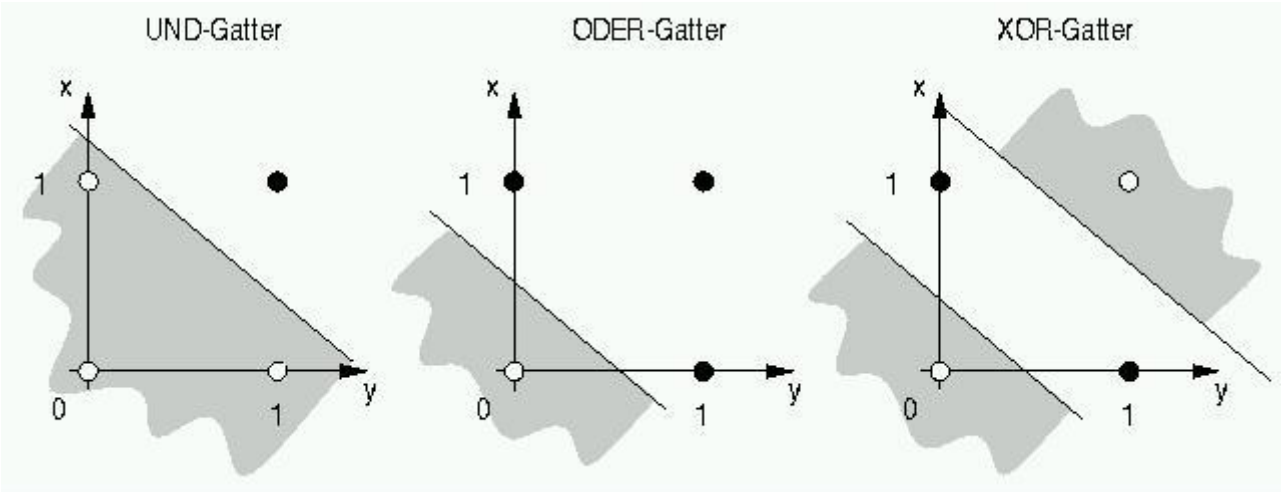


Abbildung 2.27.: Hyperebenen im $\mathbb{R}^2$ [20]

Das Perzeptron trifft die Entscheidung, ob ein Eingangsvektor zur Klasse ”0” oder zur Klasse ”1” gehört. Sieht man die Eingabevektoren als Punkte in einem n-dimensionalem Hyperrraum, so muss es möglich sein, eine Hyperebene durch diesen Raum zu legen, der die zwei Mengen ”0” und ”1” exakt trennt. Das Trainieren des Perzeptrons entspricht also dem Finden einer Trennebene zwischen den beiden Klassen - die natürlich nur gezogen werden kann, wenn das Problem linear separierbar ist.

Einstufige Perzeptrons sind also nur für sehr einfache Aufgaben mit einer geringen Zahl von Eingaben pro Zelle geeignet.

Zweistufige Perzeptrons, d.h. zwei hintereinander geschaltete Perzeptrons, hingegen können konvexe Probleme klassifizieren.

In der ersten und zweiten Ebene wird jeweils eine Hyperebene klassifiziert, so dass letztendlich durch den Schnitt der beiden, ein konvexes Polygon dargestellt werden kann.

Somit kann also auch die XOR Funktion gebildet werden:

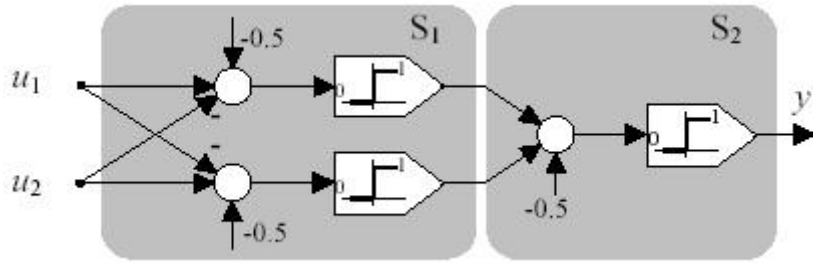


Abbildung 2.28.: Boolsche Funktion XOR

Mittels dreistufiger Perzeptrons können schließlich, durch Überlagerung und Schnitt konvexer Polygone, Mengen beliebiger Form repräsentiert werden.

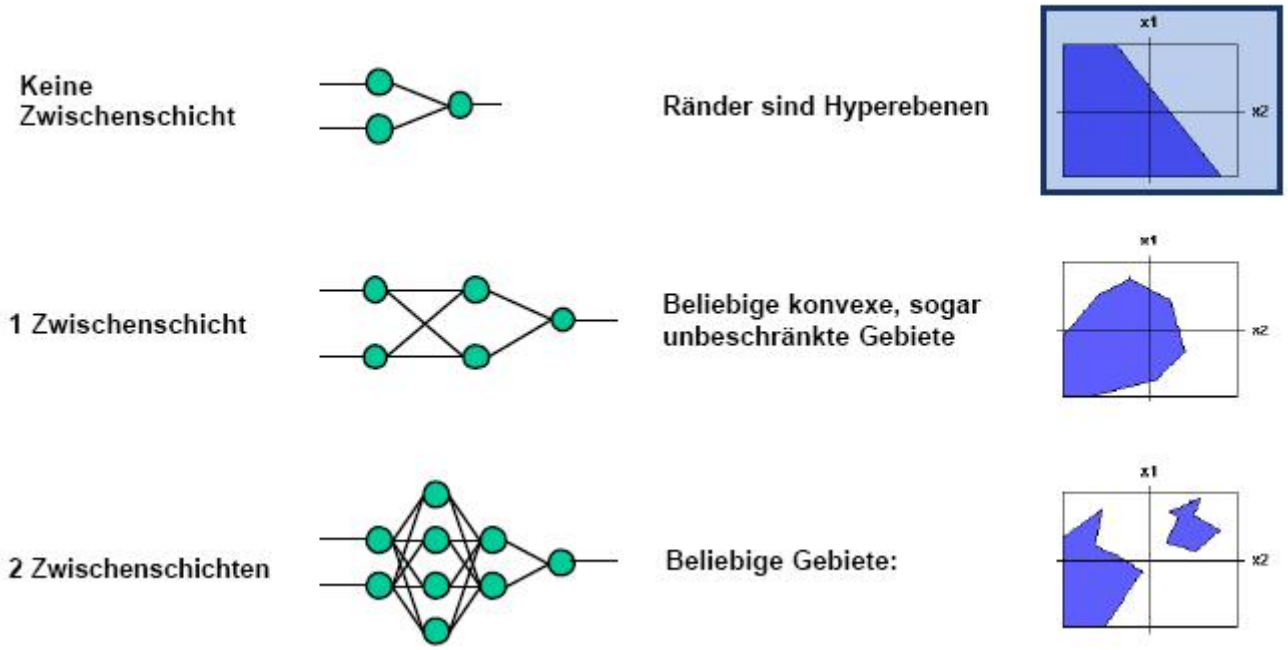


Abbildung 2.29.: [5]

Perzeptron-Konvergenz-Theorem

Theorem 2.1. *Der Lernalgorithmus des Perzeptrons konvergiert in endlicher Zeit, d.h. das Perzeptron kann in endlichen Schritten alles lernen, was es repräsentieren kann.*

Dieses Theorem garantiert also, dass ein Perzeptron einen Funktion in endlichen Zeitschritten lernen kann, sofern es sie überhaupt lernen kann.

Der Beweis ist bei [Kinnebrock] geführt und kann dort nachgelesen werden.

2.2.2. Adaline und Madaline

ADAptive LInear NEuron
Das Adaline hat eine starke Ähnlichkeit zum Perzeptron und kann ebenfalls nur lineare Klassifizierungen realisieren.

Das Adaline ist ein neuronales Netz mit nur einem einzigen Verarbeitungselement. Es besteht

aus Ein- und Ausgabeneuronen, wobei jedes Eingabeneuron mit jedem Ausgabeneuron verbunden ist. Die Gewichtungen sind variable Koeffizienten, die Aktivierungsfunktion und die Schwellenwertfunktion sind linear und die Ausgaben sind ± 1 . Es ist ein Bias vorhanden.

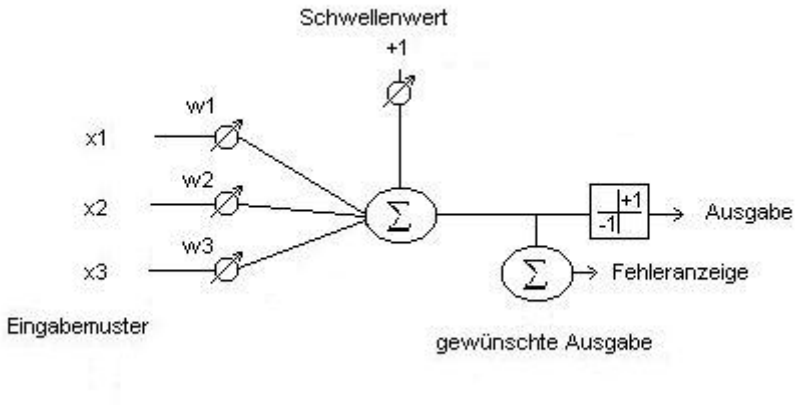


Abbildung 2.30.: Adaline [3]

Lernvorgang:
Aus der quadratischen Abweichung vom Sollwert wird eine Berechnungsvorschrift zur Verbesserung der Koeffizienten abgeleitet. (vgl. Delta-Regel)

Wie beim Perzeptron gibt es ein Theorem zur Konvergenz des Lernalgorithmus in endlich vielen Schritten.

Meist kann man sich mit einem Ausgabeneuron nicht begnügen. Um dies zu Realisieren geht man vom Adaline zum Madaline (multiple Adaline) über. Ziel hierbei ist, die Separation komplizierterer Klassen zu ermöglichen.

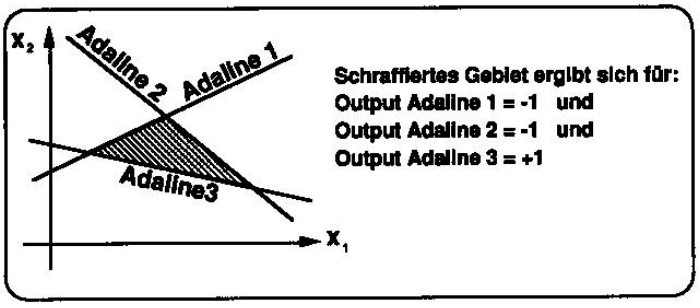


Abbildung 2.31.: [Rigoll]

Man lässt also den Eingangsvektor von mehreren Adalines verarbeiten und durch verschiedene Parameter wird der Parameterraum des Eingangsvektors in unterschiedliche Bereiche unterteilt. Das Training erfolgt ähnlich zu dem des Adaline, wobei die einzelnen Adaline separat trainiert werden.

Netzstruktur:
Die unterste Stufe ist die eines Adaline Netzes mit Gewichten w_{ij} und evtl. mit Bias.
In der zweiten Stufe gibt es von jedem vesteckten Neuron n_j genau einen Weg mit dem Gewicht 1 zum Ausgabeelement. Es gibt kein Bias.

2.3. Backpropagation

2.3.1. Einführung und Prinzip

Dieser Lernprozess, der dem überwachten Lernen zugeordnet wird, wurde von mehreren Forschern gleichzeitig erkannt. Erstmals beschrieben hat dieses numerische Verfahren P.J. Werbos 1974, aber erst die präzise mathematische Darstellung von D.E. Rumelhart, G.E. Hinton und R.J. Williams führte zum allgemeinen Durchbruch und zur Popularität dieser Lernmethode.

Backpropagation wurde speziell für mehrschichtige feedforward Netze (z.B. multilayer Perzeptron) entwickelt, die meist total verbundene Topologien, aber auch Topologien mit Shortcuts haben. Bei den bisher vorgestellten Lernmethoden wurde immer von Netzen ausgegangen, die keine verborgene Schicht besitzen.
Allerdings sind solch einfache Netze nur wenig interessant (sie können nur linear separieren) und eine Regel, mit der man auch verborgene Schichten trainieren kann, wurde benötigt.
Erst mit Hilfe der Backpropagation Regel konnte der Fehler für die Neuronen der verborgenen Schichten bestimmt und ein mehrschichtiges Netz sinnvoll trainiert werden. Generell gesehen ist das Backpropagation Verfahren eine Verallgemeinerung der bereits vorgestellen Delta-Regel, die nicht nur auf die sichtbaren Zellen am Ausgang, sondern auch auf die verdeckten Zellen in den inneren Schichten angewendet wird.
Zusätzlich wird also eine Regel gefordert, die angibt, wie die Gewichte in diesen Schichten modifiziert werden müssen.

Man geht davon aus, dass das Netz die folgenden Eigenschaften besitzt:

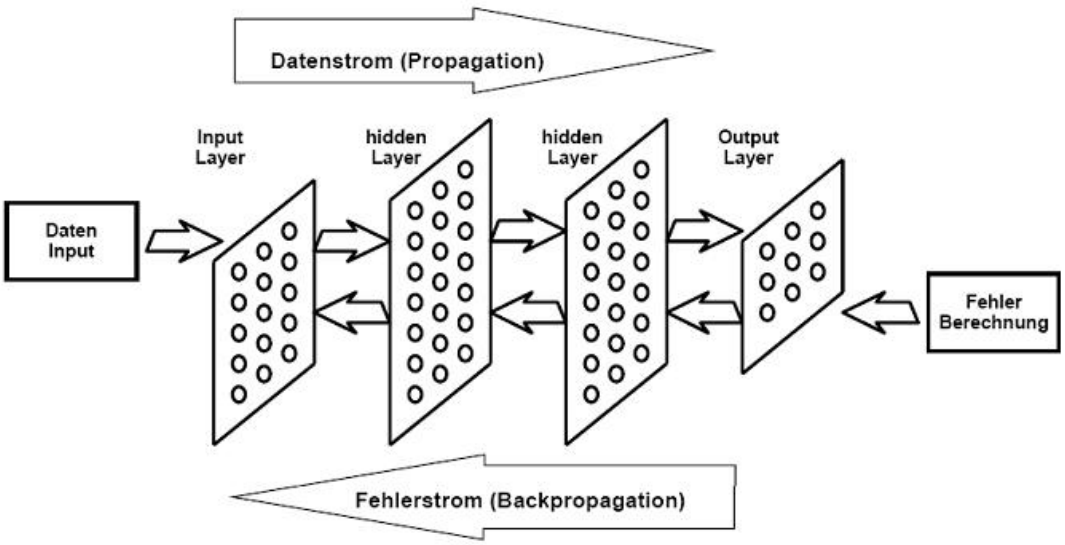
- mindestens eine verdeckte Schicht (hidden layer)
- eine nichtlineare Aktivierungsfunktion
  - an jeder Stelle differenzierbar
  - streng monoton steigend

- als Propagierungsfunktion die gewichtete Summe
- eine nach oben und unten beschränkte Aktivierungsfunktion, also eine kontinuierliche Aktivierung (z.B. $\in$ [0;1])
- Wendepunkt für $net_j = 0$ mit $f'(net_j) = maximal$ (nicht zwingend nötig)
- Identität als Ausgabefunktion der Ein- und Ausgabeschicht

Der Lernprozess besteht grundsätzlich aus 3 Phasen:

1. forward propagation
2. Bestimmung des Fehlers
3. backward propagation (”Backpropagation”)

Ablauf (gemäß dem überwachten Lernen):
Die Trainingsmuster, die erlernt werden sollen, werden dem Netz als Paare von Ein- und Ausgabevektoren präsentiert. Gemäß dem Feedforward-Prinzip werden die Eingabevektoren an der Eingabeschicht angelegt und über die funktionalen Komponenten zu den Neuronen in der Ausgabeschicht propagiert (forward propagation). Um den Fehler zu ermitteln, wird der Ausgabevektor der Ausgabeneuronen mit dem Soll-Ausgabevektor des Trainingsmusters verglichen.
Schließlich wird der Fehler in entgegengesetzter Richtung, also zurück zur Eingabeschicht, propagiert und die Verbindungsgewichte werden in Abhängigkeit der Fehlersignale adaptiert.



Der Backpropagation Algorithmus ist damit ein überwachtes Lernverfahren, bei dem die Verbindungsgewichte in einem feedforward Netz mit beliebig vielen Schichten durch das Rückwärtspropagieren eines Fehlersignals von der Ausgabeschicht durch alle verborgenen Schichten zur Eingabeschicht angepasst werden. [Köhle]

2.3.2. Das Gradientenverfahren

Das Gradientenverfahren, auch Verfahren des steilsten Abstiegs genannt, ist ein iteratives Minimierungsverfahren zur Minimierung einer stetig differenzierbaren Funktion $\boldsymbol{f} : \mathbb{R}^n \rightarrow \mathbb{R}$, mit der Iterationsvorschrift:

$$\boldsymbol{x^{(k+1)} = x^{(k)} + \eta^{(k)} d^{(k)}} \qquad \boldsymbol{k = 0, 1, 2, ...}$$

In einem Punkt $\boldsymbol{x^{(k)}}$ wird diejenige Richtung $\boldsymbol{d^{(k)}}$ bestimmt, in der f am stärksten fällt. Dieser Vorgang wird, unter Verwendung einer geeigneten Schrittweite $\boldsymbol{\eta^{(k)}}$, solange wiederholt, bis ein Minimum gefunden wird.

Es sei eine differenzierbare Funktion $\mathbb{R}^n \rightarrow \mathbb{R}$ gegeben. Ist im Punkt $\boldsymbol{x}$ der Gradient $\boldsymbol{\nabla f(x) \neq 0}$, so ist $\boldsymbol{-\nabla f(x)}$ eine Abstiegsrichtung von $\boldsymbol{f}$ in $\boldsymbol{x}$.

Unter allen Abstiegsrichtungen $\boldsymbol{d}$ der Länge 1 im Punkt $\boldsymbol{x}$ soll die Richtung $\boldsymbol{\bar{d}}$ des steilsten Abstiegs von f bestimmt werden.
Alternativ könnte man das Gradientenverfahren auch als Lösung $\boldsymbol{\bar{d}}$ des Problems

$$\boldsymbol{min_{d \in \mathbb{R}^n} \nabla f(x)^T d} \tag{2.1}$$

unter der Nebenbedingung $\boldsymbol{||d|| = 1}$ sehen.
Die Nebenbedingung ist notwendig, um eine Lösung zu finden, da die Zielfunktion nicht nach unten beschränkt ist.
Für die Richtung des steilsten Abstiegs gilt:

Lemma 2.1. *[Alt] Sei $\boldsymbol{f : \mathbb{R}^n \rightarrow \mathbb{R}}$ in $\boldsymbol{x}$ differenzierbar mit $\boldsymbol{\nabla f(x) \neq 0}$, dann ist:*

$$\boldsymbol{\bar{d} = -\frac{\nabla f(x)}{||\nabla f(x)||}}$$

die Lösung des Optimierungsproblems (2.1), d.h. der negative Gradient von $\boldsymbol{f}$ in $\boldsymbol{x}$ definiert die Richtung des steilsten Abstiegs im Punkt $\boldsymbol{x}$.

Beweis:
Für beliebiges $\boldsymbol{d \in \mathbb{R}^n}$ gilt:

$$\boldsymbol{\nabla f(x)^T d \geq -||\nabla f(x)||||d||}$$

Daher gilt für alle $\boldsymbol{d} \in \mathbb{R}^n$ mit $||\boldsymbol{d}|| = 1$:

$$\nabla \boldsymbol{f}(\boldsymbol{x})^T \boldsymbol{d} \geq -||\nabla \boldsymbol{f}(\boldsymbol{x})||$$

Für $\bar{\boldsymbol{d}}$ gilt:

$$\nabla \boldsymbol{f}(\boldsymbol{x})^T \bar{\boldsymbol{d}} = -||\nabla \boldsymbol{f}(\boldsymbol{x})||$$

q.e.d.

Algorithmus 2.1. *[Alt] Wählt man beim allgemeinen Abstiegsverfahren als Suchrichtung im Punkt $\boldsymbol{x}^{(k)}$ die Richtung des steilsten Abstiegs $\boldsymbol{d}^{(k)} = -\nabla \boldsymbol{f}(\boldsymbol{x}^{(k)})$, so erhält man das* ***Gradientenverfahren***

1. *Wähle einen Startpunkt $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$ und setze $\boldsymbol{k} := 0$.*
2. *Ist $\nabla \boldsymbol{f}(\boldsymbol{x}^{(k)}) = 0$, so stoppe.*
3. *Benutze als Suchrichtung $\boldsymbol{d}^{(k)} = -\nabla \boldsymbol{f}(\boldsymbol{x}^{(k)})$, berechne eine geeignete Schrittweite $\boldsymbol{\eta}$ und setze $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{\eta} \cdot \boldsymbol{d}^{(k)}$.*
4. *Setze k:=k+1 und gehe zu 2.*

Genauere Angaben zu Abbruchkriterien, Schrittweiten und den Beweis zur Konvergenz allgemeiner Abstiegsverfahren findet man in [19], bzw. [Alt].

2.3.3. Backpropagation Lernprozess

Ziel des Lernverfahrens für Backpropagation Netze ist es, eine Kombination von Gewichten $\boldsymbol{w_{ij}}$ zu finden, mit denen das Netzwerk ein vorgegebenes Eingabemuster möglichst fehlerfrei auf ein entsprechendes Zielmuster abbildet.

Bei der Backpropagation handelt es sich um eine Optimierungsprozedur, deren Basis das vereinfachte Gradientenabstiegsverfahren ist.
Idee:
Der Gradient ist die Richung des steilsten Anstieges auf der Fehleroberfläche. Allerdings wollen wir den Fehler minimieren und suchen somit den steilsten Abstieg (⟶ negativer Gradient).
Die Fehlerfläche E kann als Hyperfläche in einem n-dimensionalen Gewichtsraum betrachtet werden, wobei n die Anzahl der Gewichte im Netz ist.
Für einen ein- und zweidimensionalen Gewichtsraum kann die Fehlerfläche noch graphisch dargestellt werden.

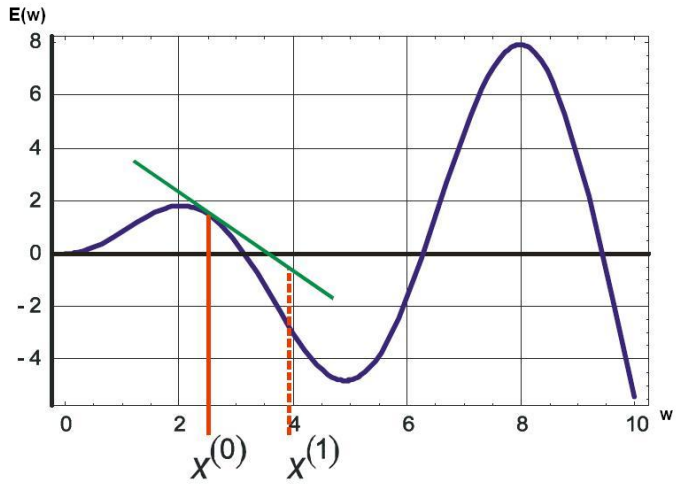


Abbildung 2.32.: Gradientenmethode im $\mathbb{R}^2$

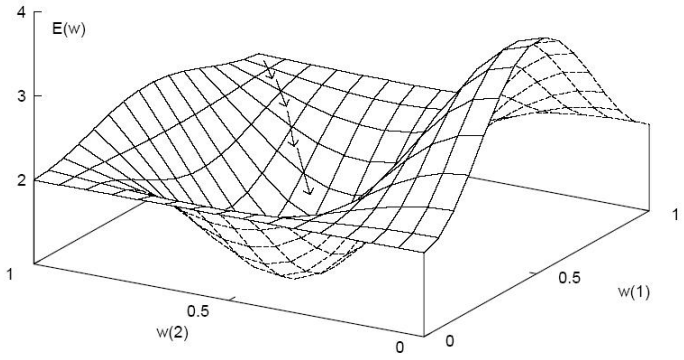


Abbildung 2.33.: Gradientenmethode im $\mathbb{R}^3$

Der Backpropagation Algorithmus gehört zu den Gradientenverfahren. Ein Gradientenverfahren berechnet den Gradienten einer Zielfunktion, hier der Fehlerfunktion, solange bis ein Minimum erreicht ist.

Die Gradientenmethode ermöglicht es, für jedes einzelne Verbindungsgewicht w_{ij} Werte zu finden, die den Lernfehler E minimieren.

Als Aktivierungsfunktion wird eine differenzierbare Funktion (mit Grenzwert) gewählt, um sicher zu gehen, dass der Gradient an jeder Stelle existiert. (Eine sigmoide Aktiverungsfunktion ist hierbei günstig, da sie die Fehleroberfläche glättet und somit den Gradientenabstieg erleichtert. Aber auch tanh, eine Logarithmus-Funktion, eine trigonometrische Funktion, o.ä. sind möglich.)
Der Algorithmus sucht nach dem Minimum der Fehlerfunktion eines bestimmten Lernproblems. Die Lösung ist eine Kombination von Gewichten, die den Netzfehler minimiert.

Ablauf:

- Ein Musterdatensatz wird an das Netz angelegt und Anhand des Vergleichs, zwischen tatsächlicher und erwünschter Ausgabe kann der Fehler des Netzes ermittelt werden.
- Fehlerfunktion:

$$E(w) = \frac{1}{2} \sum_{p} \sum_{i, i \in O} (t_{p,i} - o_{p,i}(w))^2$$

  mit:
  $t_{p,i}$: Sollausgabe des i-ten Ausgabeneurons bei Muster p
  $o_{p,i}$: Ist-Ausgabe des i-ten Ausgabeneurons bei Muster p in Abhängigkeit der Gewichte

Rückwärtsverknüpfungen: nur in der Lernrate aktiv
Vorwärtsverknüpfungen: in der Lern- und Arbeitsphase aktiv

Der Fehler wird über die feedback Verknüpfungen in das Netz rückgekoppelt, so dass die Verbindungsgewichte rückwärts Schicht für Schicht angepasst werden.
Der Fehler wird also iterativ weiter gereicht, wobei jeweils Korrekturen vorgenommen werden, immer abhängig vom Fehler der darauffolgenden Schicht.

Wiederhole den Prozess bis der gesamte Ausgabefehler gegen ein Minimum konvergiert, bzw. bis zu einer festen Anzahl an Trainingsbeispielen.
Die Gewichtsänderungen können auf zwei Arten erfolgen:

1. Offline, oder batch learning:
   Dem Netz werden alle Datensätze präsentiert und für jedes Muster werden die Gewichtsänderungen festgestellt.
   Nach dem Durchlauf aller Muster werden die Gewichtsmodifikationen für ein Gewicht summiert und zu diesem dazu addiert.

$$w_{ik} = w_{ik} + \sum_{p} \triangle_p w_{ik}$$

   Da dieses Verfahren zur Gewichtsadaption den Gradienten über dem Datensatz verwendet, ist es, mathematisch gesehen, das korrektere Verfahren.

2. Online, oder single-step learning:
   Bei diesem, oftmals schnelleren Verfahren, wird jedes Gewicht sofort nach der Präsentation eines Patterns angepasst. Die Gewichtsadaption folgt nur im Mittel dem Gradienten.

Weiterhin haben die Muster i.A. nicht die gleiche Fehleroberfläche und damit ist die Idee des Verfahrens, dass alle Muster den Fehler in einem gemeinsamen Tal minimieren, mathematisch nicht richtig.
Bei der Anwendung zeigt sich allerdings, dass dieses Verfahren schnell arbeitet und einigermaßen korrekte Resultate liefert.

2.3.4. Herleitung

Eigentlich müsste bei der kompletten Berechnung noch zusätzlich die Abhängigkeit von den Trainingsmustern berücksichtigt werden, bzw. der Fehler zusätzlich über diese summiert werden und die anderen Parameter müssten einen zusätlichen Index (meist p für ”pattern”) erhalten.
Um die Herleitung zu vereinfachen, habe ich darauf verzichtet.

Ausgangspunkt für die formale Herleitung des Backpropagation Algorithmus ist die Minimierungsvorschrift des Lernfehlers:

$$E(w_{ij}) = \frac{1}{2} \sum_{i, i \in O} (t_i - o_i(w_{ij}))^2 \rightarrow Min_{w_{ij}}$$

mit:
t_i: Sollausgabe des i-ten Ausgabeneurons
$o_i(w_{ij})$: Ist-Ausgabe des i-ten Ausgabeneurons in Abhängigkeit der Gewichte

Gewichtsänderung in den Ausgabezellen

Netto-Input der Neuronen der Schicht j:

$$net_j = \sum_i w_{ij} x_i \tag{2.2}$$

Output der Neuronen der Schicht j ($f_{out} = id$):

$$o_j = f_{act}(net_j) \tag{2.3}$$

Eine Schicht weiter: (k ist Ausgabeschicht)
Netto-Input der Neuronen der Schicht k:

$$net_k = \sum_j w_{jk} o_j \tag{2.4}$$

Output der Neuronen der Schicht k:

$$o_k = f_{act}(net_k) \tag{2.5}$$

Quadratischer Fehler:

$$E(w) = \frac{1}{2}\sum_i (t_i - o_i(w))^2 \tag{2.6}$$

(Faktor $\frac{1}{2}$ nur aus technischen Gründen)

Strategie: Wähle die Gewichtsänderung $\triangle w_{jk}$ proportional zu $\frac{\partial E}{\partial w_{jk}}$ (steilster Abstieg):

$$\triangle w_{jk} = -\eta \cdot \frac{\partial E}{\partial w_{jk}} \tag{2.7}$$

Es gilt:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{jk}}$$

Nach Gleichung (2.3.):

$$\frac{\partial net_k}{\partial w_{jk}} = o_j$$

Definiere das Fehlersignal δ:

$$\delta_k = -\frac{\partial E}{\partial net_k} \tag{2.8}$$

$$\Rightarrow \triangle w_{jk} = \eta \cdot \delta_k \cdot o_j \tag{2.9}$$

Vereinfache δ_k mit der Kettenregel:

$$\frac{\partial E}{\partial net_k} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k}$$

$$\frac{\partial E}{\partial o_k} = \frac{\partial}{\partial o_k} \cdot \frac{1}{2}\sum_m (t_m - o_m)^2 = -(t_k - o_k)$$

und

$$\frac{\partial o_k}{\partial net_k} = f'_{act}(net_k)$$

Also:

$$\delta_k = f'_{act}(net_k) \cdot (t_k - o_k) \tag{2.10}$$

Insgesamt gilt für alle Gewichtsänderungen von der verborgenen zur Ausgangsschicht:

$$\triangle w_{jk} = \eta \cdot f'_{act}(net_k) \cdot (t_k - o_k) \cdot o_j \tag{2.11}$$

Gewichtsänderung in den Zellen der verborgenen Schichten

Gewichtsänderung $\triangle w_{ij}$ proportional zu $\frac{\partial E}{\partial w_{ij}}$:

$$\triangle w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} \tag{2.12}$$

Kettenregel:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

Wiederum gilt:

$$\frac{\partial net_j}{\partial w_{ij}} = o_i$$

und

$$\frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_k} = \frac{\partial E}{\partial o_j} \cdot f'_{act}(net_j)$$

Definiere:

$$\delta_j = -\frac{\partial E}{\partial o_j} \cdot f'_{act}(net_j) \tag{2.13}$$

Insgesamt:

$$\triangle w_{jk} = \eta \cdot \delta_j \cdot o_j \tag{2.14}$$

Vereinfache (2.11.):

$$-\frac{\partial E}{\partial o_j} = -\sum_k \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial o_j} = \sum_k (-\frac{\partial E}{\partial net_k}) \frac{\partial}{\partial o_j} \sum_m w_{mk} o_m = \sum_k \delta_k w_{jk}$$

$$\Rightarrow \delta_j = (\sum_k \delta_k w_{jk}) \cdot f'_{act}(net_j) \tag{2.15}$$

Für alle Gewichtsänderungen innerhalb der verdeckten Schichten gilt:

$$\triangle w_{ij} = \eta \cdot (\sum_k \delta_k w_{jk}) \cdot f'_{act}(net_j) \cdot o_j \tag{2.16}$$

Konkrete Gewichtsänderungen unter Verwendung der Sigmoid-Funktion

In vielen Fällen wird in neuronalen Netzen als Aktivierungsfunktion die sigmoide Fermi-Funktion verwendet.

$$f(x) = \frac{1}{1 + e^x}$$

Ableitung:

$$f'(x) = \frac{e^x}{(1 + e^x)^2} = f(x) \cdot (1 - f(x)) \tag{2.17}$$

In diesem Fall lauten die Gewichtsänderungen unter Verwendung von (2.15.): folgendermaßen:

Gewichtsänderungen von der verborgenen zur Ausgangsschicht:

$$\triangle w_{jk} = \eta \cdot (t_k - o_k) \cdot o_k \cdot (1 - o_k) \cdot o_j \tag{2.18}$$

Gewichtsänderungen innerhalb der verdeckten Schichten:

$$\triangle w_{ij} = \eta \cdot (\sum_k \delta_k w_{jk}) \cdot o_j \cdot (1 - o_j) \cdot o_j \tag{2.19}$$

2.3.5. Konvergenz von Backpropagation

Der Beweis erfolgt in Anlehnung an [MangSol]
Es wird hier die Konvergenz des klassischen Backpropagtion Verfahrens eines feedforward Netzes mit einer verdeckten Schicht und einer festen Anzahl an verdeckten Neuronen gezeigt.
Es wird angenommen, man hat N Trainigsbeispiele und k Prozessoren ($N > 1, k > 1$). Die Menge der Trainingsdaten $\{1, ..., N\}$ wird in Teilmengen J_l unterteilt, mit $l = 1, ..., k$, so dass jedes Beispiel mindestens einem Prozessor präsentiert wird.

Um die Konvergenz des Backpropagation Verfahrens nachzuweisen, wird zunächst die Konvergenz von gestörten Algorithmen gezeigt.
Um eine Lösung für das nicht zwingende Minimierungsproblem zu finden, beginnt man mit dem Theorem für konvergente, nicht-monotone Algorithmen.

Definition 2.4. *[MangSol] Eine stetige Funktion $\sigma : \mathbb{R}_+ \to \mathbb{R}_+$ mit $\sigma(0) = 0$ und $\sigma(t) > 0$ für $t > 0$ heißt **zwingende Funktion**, wenn aus $t^i \geq 0$ und $\{\sigma(t^i)\} \to 0$ folgt, dass $\{t^i\} \to 0$, .*

Die beiden folgenden Lemmas kommen erst in späteren Beweisen zum Einsatz, werden aber hier schon angegeben:

Lemma 2.2. *[MangSol] Sei $f \in LC^1_L(\mathbb{R}^n)$, dann gilt:*

$$|f(y) - f(x) - \nabla f(x)^T (y - x)| \leq \frac{L}{2} \|y - x\|^2 \qquad \forall x, y \in \mathbb{R}^n$$

Lemma 2.3. *[MangSol] Seien $\{a^i\}$ und $\{\varepsilon^i\}$ zwei Folgen nichtnegativer reeller Zahlen mit $\sum_{i=0}^{\infty} \varepsilon^i < \infty$ und*
$0 \leq a^{i+1} \leq a^i + \varepsilon^i$ für i = 0, 1,...
Dann konvergiert die Folge $\{a^i\}$.

Jetzt kann das erste Resultat angegeben und bewiesen werden:

Theorem 2.2. *[MangSol] Sei* $f \in C^1(\mathbb{R}^n)$ *und sei* $\inf_{x\in\mathbb{R}^n} f(x) = \bar{f} > -\infty$.
$x^0 \in \mathbb{R}^n$ *sei ein beliebiger Startwert.*
Stoppe bei x^i*, sobald* $\nabla f(x^i) = 0$*, sonst:*
berechne $x^{i+1} = x^i + \eta_i d^i$*, bezüglich der Richtung* d^i *und der Schrittweite* η_i*, die wie folgt gewählt werden:*
Richtung d^i*:*

$$-\nabla f(x^i)^T d^i \geq \sigma(\|\nabla f(x^i)\|) - \lambda_i \tag{2.20}$$

wobei $\lambda_i \geq 0$ *und* $\sigma(\cdot)$ *zwingende Funktion*

Schrittweite η_i*:*

$$f(x^i) - f(x^{i+1}) \geq -\eta_i \nabla f(x^i)^T d^i - \nu_i \tag{2.21}$$

mit $\eta_i > 0$ *und* $\nu_i \geq 0$
Falls

$$\sum_{i=0}^{\infty} \eta_i = \infty, \qquad \sum_{i=0}^{\infty} \lambda_i \eta_i < \infty, \qquad \sum_{i=0}^{\infty} \nu_i < \infty \tag{2.22}$$

gilt, konvergiert die Folge $\{f(x^i)\}$ *und* $\inf_i \|\nabla f(x^i)\| = 0$*. Falls zusätzlich* $f \in LC^1_L(\mathbb{R}^n)$ *und* $\|d^i\| \leq c \quad \forall i$*,* $c > 0$*, folgt, dass* $\{\nabla f(x^i)\} \to 0$ *und für jeden Häufungspunkt* $\bar{x}$ *der Folge* $\{x^i\}$ *ist* $\nabla f(\bar{x}) = 0$.

Beweis:
Falls $\nabla f(x^{\bar{i}}) = 0$ für ein $\bar{i}$, dann terminiert der Algorithmus in einem stationären Punkt.
Annahme: Der Algorithmus terminiert nicht. Kombiniert man (2.20.) und (2.21.), dann folgt:

$$f(x^i) - f(x^{i+1}) \geq \eta_i \sigma(\|\nabla f(x^i)\|) - \lambda_i \eta_i - \nu_i \tag{2.23}$$

Folglich ist

$$0 \leq f(x^{i+1}) - \bar{f} \leq f(x^i) - \bar{f} + \lambda_i \eta_i + \nu_i.$$

Nach (2.22.) und Lemma 2.2. konvergiert die Folge $\{f(x^i) - \bar{f}\}$ und auch $\{f(x^i\}$.
Wendet man (2.23.) auf die erste Summation unten an, so erhält man:

$$\begin{aligned} f(x^0) - \bar{f} &\geq f(x^0) - f(x^i) = \sum_{j=0}^{i-1} (f(x^j) - f(x^{j+1})) \\ &\geq \sum_{j=0}^{i-1} \eta_j \sigma(\|\nabla f(x^j)\|) - \sum_{j=0}^{i-1} (\lambda_j \eta_j + \nu_j) \\ &\geq \inf_{0\leq j\leq i-1} \sigma(\|\nabla f(x^j)\|) \sum_{j=0}^{i-1} \eta_j - \sum_{j=0}^{i-1} \lambda_j \eta_j - \sum_{j=0}^{i-1} \nu_j \end{aligned} \tag{2.24}$$

Lässt man $i \longrightarrow \infty$, so gilt:

$$f(x^0) - \bar{f} \geq \inf_{j \geq 0} \sigma(\|\nabla f(x^j)\|) \sum_{j=0}^{\infty} \eta_j - \sum_{j=0}^{\infty} \lambda_j \eta_j - \sum_{j=0}^{\infty} \nu_j \tag{2.25}$$

Da die linke Seite und die beiden letzten Terme der rechten Seite in (2.25.) endlich sind, folgt aus der Divergenz von $\sum_{j=0}^{\infty} \eta_j$, dass $\inf_j \sigma(\|\nabla f(x^j)\|) = 0$ ist.
Nach Definition 2.4. der zwingenden Funktion ergibt sich sofort:

$$\inf_i \|\nabla f(x^i)\| = 0 \tag{2.26}$$

Sei $f \in LC_L^1(\mathbb{R}^n)$ und $\|d^i\| \leq c \; \forall i, c > 0$
Annahme: $\{\nabla f(x^i)\}$ konvergiert nicht gegen 0. Dann gibt es einige $\varepsilon > 0$ und wachsende Folgen $\{i_l\}$ von ganzen Zahlen, so dass $\|\nabla f(x^{i_l})\| \geq \varepsilon \; \forall l$ ist. Andererseits existierten nach (2.26.) zu jedem l einige $j > i_l$, so dass $\|\nabla f(x^j)\| \leq \frac{\varepsilon}{2}$ gilt. Für jedes l sei $j(l)$ die letzte Zahl, die diesen Bedingungen genügt. Mit der Dreiecksungleichung, $f \in LC_L^1(\mathbb{R}^n)$ und (2.22.) ergibt sich:

$$\begin{aligned} \frac{\varepsilon}{2} &\leq \|\nabla f(x^{i_l})\| - \|\nabla f(x^{j(l)})\| \leq \|\nabla f(x^{i_l}) - \nabla f(x^{j(l)})\| \\ &\leq L\|x^{i_l} - x^{j(l)}\| \leq L \sum_{t=i_l}^{j(l)-1} \eta_t \|d^t\| \leq Lc \sum_{t=i_l}^{j(l)-1} \eta_t \end{aligned}$$

Dann ist

$$\sum_{t=i_l}^{j(l)-1} \eta_t \geq \frac{\varepsilon}{2Lc} = \bar{c} > 0 \tag{2.27}$$

Unter Einsatz von (2.23.) und (2.27.) erhält man:

$$\begin{aligned} f(x^{i_l}) - f(x^{j(l)}) &\geq \sum_{t=i_l}^{j(l)-1} \eta_t \sigma(\|\nabla f(x^t)\|) - \sum_{t=i_l}^{j(l)-1} (\lambda_t \eta_t + \nu_t) \\ &\geq \bar{c} \inf_{i_l \leq t \leq j(l)-1} \sigma(\|\nabla f(x^t)\|) - \sum_{t=i_l}^{\infty} (\lambda_t \eta_t + \nu_t) \end{aligned}$$

Da die Folge $\{f(x^i)\}$ konvergiert und die letzte Summe in obiger Ungleichung für $l \longrightarrow \infty$ gegen 0 konvergiert, folgt dass:

$$\lim_{l \to \infty} \inf_{i_l \leq t \leq j(l)-1} \sigma(\|\nabla f(x^t)\|) = 0 \tag{2.28}$$

Mit der Wahl von i_l und $j(l)$, $\|\nabla f(x^t)\| \geq \frac{\varepsilon}{2}$, $\forall t$: $i_l \leq t \leq j(l)$ widerspricht dies (2.28.), da $\sigma(\cdot)$ eine zwingende Funktion ist. Dieses widerspricht der Annahme, dass $\nabla f(x^i)$ nicht gegen 0 konvergiert. Da der Gradient von f stetig ist folgt, dass $\bar{x}$ Häufungspunkt von $\{x^i\}$ ist. Es gilt: $\nabla f(\bar{x}) = 0$.
q.e.d.

Nun wird gezeigt, dass Theorem 2.2. zur Analyse von gestörten Gradientenverfahren angewendet werden kann. Dabei sind die Annahmen von (2.28.) in Korollar 2.12. sowohl in Abhängigkeit von f und der Lipschitz-Stetigkeit erfüllt, als auch in Abhänigkeit von ∇f in Bezug auf Backpropagation.

Korollar 2.1. *[MangSol] Sei*

$$f \in LC^1_L(\mathbb{R}^n), \qquad \|\nabla f(x)\| \leq M, \qquad f(x) \geq \bar{f} \qquad \forall x \in \mathbb{R}^n \tag{2.29}$$

für ein $M > 0$ und ein $\bar{f}$. Starte mit irgendeinem $x^0 \in \mathbb{R}^n$. Stoppe, falls $\nabla f(x^i) = 0$, sonst berechne

$$x^{i+1} = x^i + \eta_i d^i \tag{2.30}$$

wobei

$$d^i = -\nabla f(x^i) + e^i \tag{2.31}$$

für ein $e^i \in \mathbb{R}^n$, $\eta_i \in \mathbb{R}$, $\eta_i > 0$ und so dass

$$\sum_{i=0}^{\infty} \eta_i = \infty, \qquad \sum_{i=0}^{\infty} \eta_i^2 < \infty, \qquad \sum_{i=0}^{\infty} \eta_i \|e^i\| < \infty, \qquad \|e^i\| \leq \gamma \quad \forall i, \qquad \gamma > 0 \tag{2.32}$$

ist.
Es folgt, dass $\{f(x^i)\}$ konvergiert, $\{\nabla f(x^i)\} \to 0$ und für jeden Häufungspunkt $\bar{x}$ der Folge $\{x^i\}$ gilt $\nabla f(\bar{x}) = 0$.

Beweis:
Es reicht aus, zu zeigen, dass die Bedingungen (2.20.) - (2.22.) aus Theorem 2.2. erfüllt sind. Zuerst stellt man fest, dass durch (2.29.), (2.31.) und (2.32.), $\|d^i\| \leq M + \gamma$ für alle i gilt. Durch die Cauchy-Schwarz-Ungleichung, (2.29.) und (2.30.), gilt mit $\sigma(s) = s^2$:

$$\begin{aligned} -\nabla f(x^i)^T d^i &= \|\nabla f(x^i)\|^2 - \nabla f(x^i)^T e^i \\ &\geq \sigma(\|\nabla f(x^i)\|) - \|\nabla f(x^i)\| \|e^i\| \\ &\geq \sigma(\|\nabla f(x^i)\|) - M\|e^i\| \end{aligned} \tag{2.33}$$

Mit Lemma 2.1., (2.30.) und (2.31.) folgt:

$$\begin{aligned} f(x^i) - f(x^{i+1}) &\geq -\nabla f(x^i)^T (x^{i+1} - x^i) - \frac{L}{2}\|x^{i+1} - x^i\|^2 \\ &= -\eta_i \nabla f(x^i)^T d^i - \frac{L}{2}\eta_i^2 \|d^i\|^2 \\ &\geq -\eta_i \nabla f(x^i)^T d^i - \frac{L}{2}\eta_i^2 (M+\gamma)^2 \end{aligned} \tag{2.34}$$

Die Aussagen in (2.32.)-(2.34.) weisen nach, dass (2.20.) - (2.22.) aus Theorem 2.2. gelten, mit $\lambda_i = M\|e^i\|$, $\nu_i = \frac{L}{2}(M+\gamma)^2\eta_i^2$.
q.e.d.

Ab jetzt richtet sich die Aufmerksamkeit auf den Fall, dass die Zielfunktion des Problems von der Summe einer finiten Anzahl von Funktionen ($f(x) = \sum_{j=1}^{N} f_j(x)$) dargestellt wird.
Man nimmt an, es gibt k parallele Prozessoren, $k \leq 1$.
Sei $J_l \subseteq \{1,\dots,N\}$ mit $\bigcup_{l=1}^{k} J_l = \{1,\dots,N\}$, $J_{l_1} \cap J_{l_2} = \emptyset$ für $l_1 \neq l_2$. Sei N_l die Anzahl der Elemente von J_l. Definiere

$$f^l(x) = \sum_{j \in J_l} f_j(x) \tag{2.35}$$

Damit ist

$$f(x) = \sum_{l=1}^{k} f^l(x) \tag{2.36}$$

Nun kann die parallele Version von Korollar 2.2. dargestellt und bewiesen werden:

Theorem 2.3. *[MangSol] Jedes f_j ($j = 1,\dots,N$) genüge Bedingung (2.29.) aus Korollar 2.2. Beginne mit $x^0 \in \mathbb{R}^n$ beliebig. Stoppe, falls $x^i = x^{i-1}$, sonst berechne x^{i+1} wie folgt:*

1. *Parallelisierung: Für jeden Prozessor $l \in \{1,\dots,k\}$ berechne*

$$y_l^{i+1} = x^i + \eta_i d_l^i \tag{2.37}$$

   wobei

$$d_l^i = -\nabla f^l(x^i) + e_l^i, \eta_i > 0 \tag{2.38}$$

   gilt.

2. *Synchronisierung: Sei*

$$x^{i+1} = \frac{1}{k} \sum_{l=1}^{k} y_l^{i+1} \tag{2.39}$$

Falls für geeignete $\gamma > 0$

$$\sum_{i=0}^{\infty} \eta_i = \infty, \qquad \sum_{i=0}^{\infty} \eta_i^2 < \infty, \qquad \sum_{i=0}^{\infty} \eta_i \|e_l^i\| < \infty, \qquad \|e_l^i\| \leq \gamma \quad \forall i, \qquad l = 1,\dots,k \tag{2.40}$$

gilt, gelten alle Folgerungen aus Korollar 2.2. .

Beweis:
Mit (2.36.) und (2.37.)-(2.39.) ergibt sich:

$$x^{i+1} - x^i = \frac{1}{k} \sum_{l=1}^{k} y_l^{i+1} - x^i = \frac{1}{k} \sum_{l=1}^{k} (x^i + \eta_i d_l^i) - x^i$$

$$= \frac{1}{k} \sum_{l=1}^{k} \eta_i d_l^i = \frac{\eta_i}{k} \sum_{l=1}^{k} (-\nabla f^l(x^i) + e_l^i)$$

$$= \frac{\eta_i}{k}(-\nabla f(x^i) + \sum_{l=1}^{k} e_l^i).$$

Jetzt gilt Korollar 2.2., mit (2.40.) und $e^i = \sum_{l=1}^{k} e_l^i$.
q.e.d.

Eigentlicher Beweis zur Konvergenz von Backpropagation:
Ziel: Minimiere diese Fehlerfunktion:

$$min_{x \in \mathbb{R}^n} f(x) := \sum_{l=1}^{k} f^l(x) = \sum_{l=1}^{k} \sum_{j \in J_l} f_j(x)$$

mit $J_l \subseteq \{1, \ldots, N\}, l = 1, ..., k \bigcup_{l=1}^{k} J_l = \{1, \ldots, N\}$

Es wird ein paralleler Backpropagation Algorithmus mit Momentum Term verwendet, welcher aus der Differenz zwichen der aktuellen und der vorhergehenden Iterierten besteht.

Die Notation für den Beweis lautet folgendermaßen:

- $i = 1, 2, ...$ Index der "major" Iterationen des Backpropagation Algortihmus, von denen jede einzelne einen Durchlauf des kompletten Datensatzes bedeutet.
- $f_1(x), ... f_N(x)$ Fehlerfunktionen nach Iteration $1, ..., N$
- Eine Iteration wird seriell oder parallel von k Prozessoren durchgeführt, wobei Prozessor l zur Fehlerfunktion $f^l(x)$ ($l = 1, ..., k$) gehört.
  Jede "minor" Iteration besteht aus einem Schritt in Richtung des negativen Gradienten $-\nabla f^l_{m(j)}(z_l^{i,j})$
- $j = 1, ..., N_l$ Index der "minor" Iterationen des Backpropagation, die von einem parallelen Prozessor l durchgeführt werden ($l = 1, ..., k$).
- x_i Iterierte in $\mathbb{R}^n$ der "major" Iteration ($i = 1, 2, ...$)
- $z_j^{i,j}$ Iterierte in $\mathbb{R}^n$ der "minor" Iteration, innerhalb der "major" Iteration
- η_i Lernrate (konstant)
- α_i Momentum Term (konstant)

Algorithmus 2.2. *[MangSol]Paralleles Backpropagation mit Momentum Term*
Beginne mit einem beliebigen $x^0 \in \mathbb{R}^n$. Sobald x^i mit $\nabla f(x^i) = 0$ erreicht ist, stoppe, sonst berechne x^{i+1} wie folgt:

- *für jeden Prozessor:*

$$z_l^{i,j+1} = z_l^{i,j} + \eta_i \nabla f_{m(j)}^l(z_l^{i,j}) + \alpha \triangle z_l^{i,j} \qquad j = 1, ..., N_l \tag{2.41}$$

wobei

$$z_l^{i,1} = x_i, \qquad 0 < \eta^i < 1, \qquad 0 < \alpha^i < 1$$

$$\triangle z_l^{i,j} = \begin{cases} 0 & \text{für} j = 1 \\ z_l^{i,j} - z_l^{i,j-1} & \text{sonst} \end{cases} \tag{2.42}$$

- *Synchronisation:*

$$x^{i+1} = \frac{1}{k} \sum_{l=1}^{k} z_l^{i,N_l+1} \tag{2.43}$$

Jetzt wird die vorherige Analyse auf das Backpropagation Verfahren angewendet:

Theorem 2.4. *[MangSol] Sei $S \subset \mathbb{R}^n$ eine beliebige, beschränkte Menge. Falls die Lernrate und der Momentum Term so gewählt sind, dass*

$$\sum_{i=0}^{\infty} \eta_i = \infty, \qquad \sum_{i=0}^{\infty} \eta_i^2 < \infty, \qquad \sum_{i=0}^{\infty} \alpha_i \eta_i < \infty \tag{2.44}$$

gilt, folgt, dass für jede Folge $\{x^i)\} \subset S$, die durch Algorithmus 2.1. erzeugt wird, $\{f(x^i)\}$ konvergiert, $(\nabla f(x^i))_i \to 0$ und für jeden Häufungspunkt $\bar{x}$ der Folge $\{x^i\}$ $\nabla f(\bar{x}) = 0$ ist.

Beweis:
Zu zeigen ist, dass die Voraussetzungen aus Theorem 2.3. gelten. Die Fehlerfunktion ist glatt und daher genügt der Gradient der Bedingung (2.29.) auf einer beschränkten Menge S. Weiterhin ist f nichtnegativ, also nach unten beschränkt. Mit Teil 1 und 2 aus Algorithmus 2.1. gilt für jeden Durchlauf i, jeden Prozessor l und jegliches j mit $2 \leq j \leq N_l + 1$:

$$\begin{aligned} z_l^{i,j} - w^i = z_l^{i,j} - z_l^{i,1} &= \sum_{t=1}^{j-1} (z_l^{i,t+1} - z_l^{i,t}) \\ &= \sum_{t=1}^{j-1} (-\eta_i \nabla E_{m(t)}^l(z_l^{i,t}) + \alpha_i \Delta z_l^{i,t}) \\ &= -\eta_i \sum_{t=1}^{j-1} \nabla E_{m(t)}^l(z_l^{i,t}) + \alpha_i (z_l^{i,j-1} - w^i) \qquad (2.45) \\ = -\eta_i \sum_{t=1}^{j-1} \nabla E_{m(t)}^l(z_l^{i,t}) - \eta_i \sum_{s=1}^{j-2} (\alpha_i^{j-1-s} \sum_{t=1}^{s} \nabla E_{m(t)}^l(z_l^{i,t})) & \qquad (2.46) \end{aligned}$$

wobei Gleichung (2.46.) durch mehrmalige Anwendung von (2.45.) entsteht, wobei j durch j - 1, j - 2, ..., 2 ersetzt wird. Mit (2.45.) und (2.35.) gilt für $j = N_l + 1$:

$$\begin{aligned} z_l^{i,N_l+1} - w^i &= -\eta_i \sum_{t=1}^{N_l} \nabla E^l_{m(t)}(z_l^{i,t}) + \alpha_i(z_l^{i,N_l} - w^i) \\ &= -\eta_i(\nabla E^l(w^i) + a_l^i + \frac{\alpha_i}{\eta_i} b_l^i) \end{aligned} \tag{2.47}$$

wobei

$$a_l^i = \sum_{t=2}^{N_l} (\nabla E^l_{m(t)}(z_l^{i,t}) - \nabla E^l_{m(t)}(w^i)) \tag{2.48}$$

und

$$b_l^i = w^i - z_l^{i,N_l} \tag{2.49}$$

Sei

$$e_l^i = -a_l^i - \frac{\alpha_i}{\eta_i} b_l^i \tag{2.50}$$

Mit Theorem 2.3., (2.44.) und (2.47.) muss nur noch

$$\sum_{i=0}^{\infty} \eta_i \|e_l^i\| < \infty, \|e_l^i\| \leq \gamma, \gamma > 0 \qquad (l = 1, \ldots, k) \tag{2.51}$$

gezeigt werden.
Mit (2.48.), (2.46.), (2.29.), der Dreiecksungleichung und $\alpha_i \leq 1$ folgt:

$$\begin{aligned} \|a_l^i\| \leq \sum_{t=2}^{N_l} \|\nabla E^l_{m(t)}(z_l^{i,t}) - \nabla E^l_{m(t)}(w^i)\| \leq L \sum_{t=2}^{N_l} \|z_l^{i,t} - w^i\| \\ \leq L \sum_{t=2}^{N_l} (\eta_i \sum_{r=1}^{t-1} \|\nabla E^l_{m(r)}(z_l^{i,r})\| + \eta_i \sum_{s=1}^{t-2} (\alpha_i^{t-1-s} \sum_{r=1}^{s} \|\nabla E^l_{m(r)}(z_l^{i,r})\|)) \\ \leq L\eta_i(N_l^2 M + N_l^3 M) = c_1 \eta_i \end{aligned} \tag{2.52}$$

Ähnlich zeigt man mit (2.49.), (2.46.), (2.29.) der b_l^i, der Dreiecksungleichung und $\alpha_i \leq 1$:

$$\begin{aligned} \|b_l^i\| = \|z_l^{i,N_l} - w^i\| \leq \eta_i(\sum_{t=1}^{N_l-1} \|\nabla E^l_{m(t)}(z_l^{i,t})\| + \sum_{s=1}^{N_l-2} (\alpha_i^{N_l-1-s} \sum_{t=1}^{s} \|\nabla E^l_{m(t)}(z_l^{i,t})\|)) \\ \leq \eta_i(\sum_{t=1}^{N_l-1} M + \sum_{s=1}^{N_l-2} M N_l) \leq \eta_i(M N_l + M N_l^2) = c_2 \eta_i \end{aligned} \tag{2.53}$$

Mit (2.50.), (2.52.), (2.53.), der Dreiecksungleichung erhalten wir:

$$\|e_l^i\| \leq c_1 \eta_i + c_2 \alpha_i$$

Zusammen mit (2.44.) und (2.51.) ist die Behauptung bewiesen.
q.e.d.

Bemerkung 2.2. *[MangSol] Es wurde ein allgemeines Theorem zur nichtmonotonen Konvergenz von Backpropagation, zum Training künstlicher neuronalen Netze mit einer verdeckten Schicht, vorgestellt. Der serielle oder parallele Backpropagation Algorithmus mit oder ohne Momentum Term, kann als deterministische, gestörte Gradientenmethode gesehen werden. Jeder Häufungspunkt, der durch den Algorithmus erzeugten Folge von Gewichten, ist ein stationärer Punkt der Fehlerfunktion bezogen auf die gegebene Menge der Trainingsdaten. Das Ergebnis kann auch auf feedforward Netze mit mehreren verdeckten Schichten ausgeweitet werden.*

2.3.6. Probleme

- Die Anzahl der hidden neurons und der Lernrate ist vom Anwendungsgebiet abhängig.
  - Anzahl zu klein:
    - Die Funktion kann nicht mehr dargestellt werden.
  - Anzahl zu groß:
    - Die Zahl der unabhängigen Variablen der Fehlerfunktion steigt und somit steigt auch die Zahl der Nebenminima.
    - Gefahr des Überlernens (overtraining)
    - Das Problem wird nur teilweise durch Generalisierung erkannt.
- Eine schnelle Konvergenz zu einer (optimalen) Lösung ist nicht immer gegeben.
- Bei höherer Dimension des Netzes steigt die Anzahl der Verbindungen (Gewichte), so dass die Fehlerfläche, in Abhängigkeit von den Gewichten, immer stärker zerklüftet ist.
  - Die Gefahr nur ein lokales Minimum zu finden steigt.
    $\rightrightarrows$ *In der Praxis: trainiere mehrere Netze mit unteschiedlichen Anfangsgewichten und zufälliger Reihenfolge der Trainingsdaten.*
    Wird das gleiche Minimum mehrmals errechnet, so kann man meist darauf schließen, dass es sich um das globale Minimum handelt.

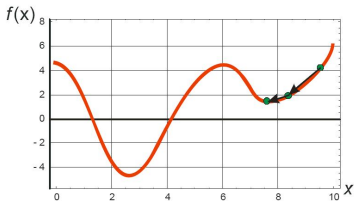


Abbildung 2.34.: Stagnation in einem lokalen Minimum

– Es kann zu einer Stagnation bei flachen Plateaus, oder zu Oszillation in steilen Schluchten kommen.
⇉ *Backpropagation mit Momentum Term*

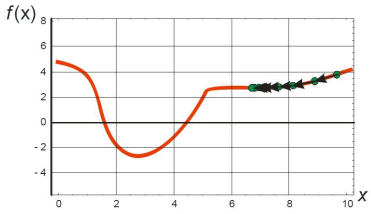


Abbildung 2.35.: Stagnation auf einem Plateau

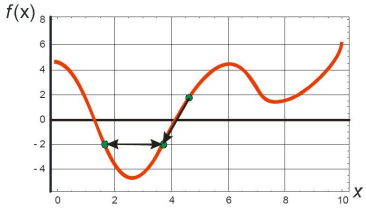


Abbildung 2.36.: Oszillation

– Gute Minima können aufgrund von starken Gradienten bzw. hohen Schrittweiten verlassen werden. ⇉ *Backpropagation mit Momentum Term*

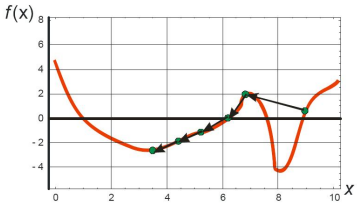


Abbildung 2.37.: Überspringen eines Minimums

- Symmetrie breaking:
  Die Startgewichte zwischen den Ebenen dürfen nicht alle gleich groß gewählt werden, da sonst der Output aller Neuronen einer Schicht gleich ist und das Netz keine unteschiedlichen Gewichte in diesen Schichten annehmen kann. Die Gewichte können sich zwar verändern, allerdings immer um den gleichen Betrag, so dass sie Symmetrie bestehen bleibt.
  Insbesondere ist die Vorbelegung der Gewichte durch 0 ungeeignet, da die Gewichte u.a. zur Bestimmung des Fehlermaßes in den inneren Schichten eingesetzt werden.
  $\rightrightarrows$ *günstige Initialisierung: gleichverteilte, zufällig ermittelte Werte. Wähle diese außerdem relativ klein, damit sich das Netz schneller anpassen kann.*

- Wahl der Schrittweite:
  Ist η zu klein, so ergibt sich ein hoher Zeitaufwand für das Training; Das Verfahren wird ineffizient.
  Ist η zu groß, so kann die Iteration nach Backpropagation entweder enge Täler nicht finden, oder sie springt wieder heraus oder sie oszilliert. Das Verfahren wird zu ungenau.

- Bei der Beschreibung des Backpropagation Algorithmus wird keine konkrete Abbruchbedingung angegeben.
  Normalerweise wird iteriert, bis der Fehler für die Trainingsbeispiele unter einem vorgegebenen Schwellenwert liegt. Da der Algorithmus zur Überspezialisierung (overfitting) neigt, ist diese Strategie allerdings sehr bedenklich.

  $\rightrightarrows$ *Eine Abhilfe könnte sein, eine separate Menge von Beispielen zur Validierung zu benutzen und dann solange zu iterieren, bis der Fehler für Validierungsmenge minimal ist, bzw. bis zu dem Punkt zu iterieren, an dem sich die Fehlerkurven der Trainings- und der Validierungsdaten schneiden.*
  Vorsicht: Der Fehler der Validierungsmenge fällt nicht immer monoton, im Gegenteil:
  Der Testfehler sinkt nur bis zu einem bestimmten Zeitpunkt und steigt dann wieder! Der Fehler in der Trainingsmenge dagegen, konvergiert mit steigender Epochenzahl.
  Durch längeres Training sinkt also die Generalisierungsfähigkeit, denn bei längerer Lernzeit spezialisiert sich das Netz immer mehr auf die Eigenheiten in den Trainingsfällen.
  Das Training sollte daher bei einem Minimum in der Testmenge beendet werden.

2.3.7. Modifikationen der Backpropagation Regel

Es gibt zahlreiche Möglichkeiten, die Backpropagation Regel zu modifizieren. Hier nur einige Beispiele:

- ändern der Fehlerfunktion
- Momentum-Term einführen
- Weight Decay
- Lernkonstanten und Output Funktion können für jede Schicht oder für jedes Neuron einzeln festgelegt werden

- Lernkonstanten können dynamisch angepasst werden
- Gewichtsanpassung modifizieren (Manhatten Training)
- Flat Spots eleminieren
- Lernrate wird an die Gewichte und den Gradienten gekoppelt
- Konvergenz durch Nutzung der zweiten Ableitung beschleunigen

Backpropagation mit Momentum Term

Das Momentum wurde erstmals 1986 von Rumelhart, Hinton und Williams eingeführt.
Das Verfahren ist ein online-Trainingsverfahren (Aktualisierung der Gewichte erfolgt direkt nach der Präsentation jedes einzelnen Musters). Dieses wird auch als konjugierter Gradientenabstieg bezeichnet.
Man führt einen festen Term $\alpha \cdot \triangle w_{ij}(t)$ ein, der bei einem Lernschritt die Richtung und das Ausmaß der bisherigen Gewichtsänderungen berücksichtigt.
Die Gewichte werden so modifiziert, dass das neue w_{ij} noch zusätzlich vom alten w_{ij} abhängt:

$$\triangle w_{ij}(t+1) = \eta o_j \delta_j + \alpha \triangle w_{ij}(t)$$

Das Fehlersignal δ_j ist durch

$$\delta_j = \begin{cases} (\frac{\partial}{\partial f_{ein}} f_{act}(f_{ein}) + c)(t_j - o_j) & \text{j ist Ausgabeneuron} \\ (\frac{\partial}{\partial f_{ein}} f_{act}(f_{ein}) + c)(\sum_k \delta_k w_{jk}) & \text{j ist verdecktes Neuron} \end{cases}$$

definiert.
Die Addition des Momentum Terms bewirkt eine Vergrößerung der Gewichtsveränderung bei kleinen Gradienten (Stagnation auf Plateaus der Hyperfläche des Gesamtfehlers) und eine Verringerung dieser bei großen Gradientenänderungen (in stark zerklüfteten Fehlerflächen wird abgebremst, so dass es nicht zu Oszillationen kommt).

Dynamische Lernraten

Eine weitere Möglichkeit, die Konvergenz der Gewichte anzutreiben, ist die Verwendung von dynamischen Lernraten.
Eine zu große Lernrate verursacht, dass steile Schluchten übersprungen werden, oder dass es zur Oszillation darin kommt. Eine zu kleine Lernrate bewirkt, dass auf Plateaus endlos lange gelernt wird und macht das Verfahren auch insgesamt gesehen viel zu langsam.

Strategie: Zu Beginn des Trainings wählt man die Lernrate eher groß (grobes Lernen) und mit fortschreitendem Training wird diese verringert.
Man vermindert sie gewöhnlich um einen festen Prozentsatz λ, so dass gilt:

$$\eta(t+1) = \eta(t)(1-\lambda)$$

Bemerkung 2.3. *Man sollte die Lernrate nicht vom Fehler abhängig wählen, da die Gefahr besteht, auf einem Plateau stehen zu bleiben. Weiterhin sollte man die Lernrate auch nicht von der Zeit abhängig machen, da man sonst eventuell stehen bleibt, bevor man zu Ende gelernt hat.*

Weight decay

Diese, von Paul Werbos entwickelte Modifikation, beschäftigt sich wiederum mit dem Problem einer steilen und zerklüfteten Fehlerfläche, auf der das normale Backpropagation Verfahren dazu neigt zu oszillieren oder auch zu große Sprünge zu machen.
Idee: Verwende einen kleinen Betrag der Gewichte bei gleichzeitiger Annäherung an die Zielvorgaben der Trainingsmenge.

$$\boldsymbol{E_{neu} = E + \frac{1}{2} d \sum_{i,j} (w_{ij})^2}$$

Der zweite Summand in diesem Term hat die Aufgabe, zu große Gewichte zu ”bestrafen”.

$$\boldsymbol{\Rightarrow \frac{\partial E_{neu}}{\partial w_{ij}} = \frac{\partial E}{\partial w_{ij}} + d w_{ij}}$$

$$\boldsymbol{\Longrightarrow \triangle w_{ij}(t+1) = \eta o_i \delta_j + d w_{ij}(t)}$$

Quickpropagation

Dieser Algorithmus wurde von Scott Fahlmann entwickelt, um das Backpropagation Verfahren zu beschleunigen.
Er hat allerdings nur noch in seinen Grundzügen mit Backpropagation zu tun. Quickpropagation ist ein Verfahren zweiter Ordnung zur Bestimmung des Minimums der Fehlerfunktion eines feedforward Netzes.
Idee:
Durch quadratische Approximation kann der Lernprozess erheblich schneller gemacht werden.

Annahmen:

- Die Fehlerfunktion in Abhängigkeit von nur **einem Gewicht** w ist lokal quadratisch und kann daher durch eine nach oben offene Parabel beschrieben werden. (Die anderen Gewichte bleiben konstant.)

- Die Änderung der Steigung der Fehlerkurve in Richtung des Gewichtes w_{ij} erfolgt unabhängig von den Änderungen der anderen Gewichte.

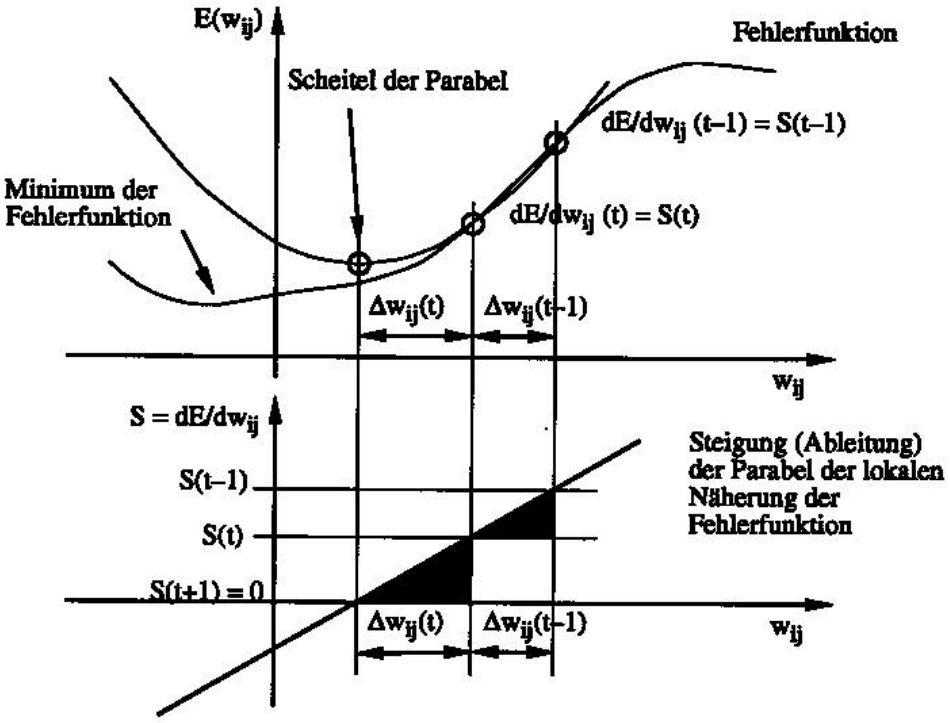


Abbildung 2.38.: Quickprop graphisch dargestellt [Zell]

Strategie: Für die Fehlerfunktion E wird ein quadratisches Interpolationspolynom der Form $E(w) = aw^2 + bw + c$ bestimmt. Suche nach dem Scheitel dieser Parabel, der dann gleichzeitig das Minimum der Fehlerfläche darstellt.
Hierzu sind folgende Parameter notwendig:

- zwei aufeinanderfolgende Werte der partiellen Ableitung der Fehlerfunktion (bezüglich des gleichen Trainigsmusters!).
  Der Algorithmus kann nicht fortlaufend, sondern erst nach einer kompletten Epoche aktualisiert werden. → Offline Training
  - Ableitung der Fehlerfunktion nach w_{ij} zur Zeit $t-1$: $\frac{\partial E}{\partial w_{ij}}(t-1)$
  - Ableitung der Fehlerfunktion in Richtung w_{ij} zur Zeit t: $\frac{\partial E}{\partial w_{ij}}(t)$
- letze Änderung des Gewichts $\triangle w_{ij}$

setze für die Steigung der Fehlerfunktion $S(t) = \frac{\partial E}{\partial w}$ (slope) in Richtung des Gewichtsvektors:

$$S(t) := \frac{\partial E}{\partial w_{ij}}(t) \tag{2.54}$$

und

$$S(t-1) := \frac{\partial E}{\partial w_{ij}}(t-1) \tag{2.55}$$

(2.54.) und (2.55.) sind die Interpolationsbedingungen aus denen man die Koeffizienten des Interpolationspolynoms bestimmen kann.

Die Fehlerkurve des Gewichts w_{ij} soll zur Zeit t als quadratisches Problem approximiert werden.
Annahme: $w(t+1)$ ist Minimum
(schreibe fortan nur noch w_t statt w_{ij}(t) bzw. $\triangle w_t$ statt $\triangle w_{ij}$(t))

$$E(w_t) = aw_t^2 + bw_t + c \qquad mit \quad (a > 0)$$

$$E'(w_t) = 2aw_t + b =^{(2.19.)} S(t) \tag{2.56}$$

$$\Rightarrow w_t = \frac{S(t) - b}{2a} \tag{2.57}$$

$$\Rightarrow \qquad E'(w_{t+1}) = 0 \Leftrightarrow w_{t+1} = -\frac{b}{2a} \tag{2.58}$$

Berechne a,b durch quadratischer Interpolation aus $S(t), S(t-1), w_{t-1}$ und $w_t = w_{t-1} + \triangle w_{t-1}$.
Daraus kann nun w_{t+1} bzw. $\triangle w_t = w_{t+1} - w_t$ berechnet werden.
Es gilt:

$$S(t-1) = 2aw_{t-1} + b$$

$$S(t) = 2aw_t + b = 2a(w_{t-1} + \triangle w_{t-1}) + b$$

$$\Rightarrow -\frac{1}{2a} = \frac{1}{S(t-1) - S(t)} \triangle w_{t-1} \tag{2.59}$$

Zusammen mit vorher ergibt sich für die Gewichtsänderung:

$$\triangle w_t = w_{t+1} - w_t = -\frac{S(t)}{2a} = \frac{S(t)}{S(t-1) - S(t)} \triangle w_{t-1} \tag{2.60}$$

Einige Auszüge aus der Arbeitsweise des Algorithmus:

- $|S(t)| < |S(t-1)|$ und $sgn(S(t)) = sgn(S(t-1))$
  Das Gewicht w wird in gleicher Richtung wie im Schritt vorher modifiziert; abhängig von der Verkleinerung der Ableitung im Verglich zum letzten Schritt.

- $sgn(S(t)) \neq sgn(S(t-1))$
  ein Minimum wurde übersprungen; bei $\triangle w_t$ ändert sich das Vorzeichen; der Algorithmus geht zurück und bleibt schließlich irgendwo zwischen w_{t-1} und w_t stehen.

- $S(t) = S(t-1)$ (oder $|S(t)| > |S(t-1)|$ und $sgn(S(t)) = sgn(S(t-1))$)
  $\Rightarrow |\triangle w_t| \to \infty$
  Das Verfahren benötigt einen Faktor μ, der das Wachsen stoppt (Faktor des maximalen Wachstums)
  $\to$ maximales Wachstum des Aktualisierungswertes: $\mu \triangle w_{t-1}$.
  Ein Gewicht wird also nie mit einem größeren Wert, als dem, der in der Aktualisierungsschranke angegeben ist, verändert.

Bemerkung 2.4. *Es handelt sich hier um eine starke Idealisierung des eigentlichen Problems.*
Man findet daher auch nur einen Punkt, der sich in der Nähe des tatsächlichen lokalen (evtl. auch globalen) Minimums befindet.

Ein Beweis zur Konvergenz des Quickpropagation Algorithmus findet sich in [VraMagPlag].

Resilient Propagation (RProp)

Dieses Verfahren kombiniert viele verschiedene Ansätze zur Verbesserung des Backpropagation Verfahrens. (Manhatten Training, Super SAB (vgl. [Zell]), Quickpropagation).
Grundidee:
Negative Einflüsse von $\frac{\partial E}{\partial w_{ij}}(t)$ auf die Adaption der Gewichte sollen vermieden werden.
($\frac{\partial E}{\partial w_{ij}}(t)$ sind die partiellen Ableitungen der Gesamtfehlerfunktion nach den Gewichten w_{ij} zum Zeitpunkt t.)
Die Gewichte werden demnach nicht mehr mit Werten der aktuellen partiellen Ableitung, sondern nur nach dem Vorzeichen des Gradienten der Fehlerfunktion von zwei aufeinanderfolgenden Ableitungswerten modifiziert. Jedes Gewicht besitzt einen Parameter für die Änderung der Schrittweite und man verwendet die Gradienten zum aktuellen und vorangegangen Zeitpunkt.

Sind die Vorzeichen gleich, dann war der letzte Aktualisierungsschritt in Ordnung und der nächste Schritt wird daher etwas größer ausfallen (in der gleichen Richtung), die Konvergenz in flachen Bereichen der Fehlerfläche wird beschleunigt. Jeder Vorzeichenwechsel wird als Sprung über ein lokales Minimum der Fehlerfläche interpretiert und daher wird der letzte Schritt zurückgenommen und danach verkleinert (mit dem Faktor η^-) wiederholt.

Definiere die Schrittweite $\triangle_{ij}(t)$ für die Aktualisierung des Gewichts w_{ij} zur Zeit t in Abhängigkeit von der vorherigen Schrittweite $\triangle_{ij}(t-1)$.
Berücksichtige dabei auch das Vorzeichen der partiellen Ableitungen:

$$\triangle_{ij}(t) = \begin{cases} \triangle_{ij}(t-1)\eta^+ & \text{für} \quad S(t-1)S(t) > 0 \\ \triangle_{ij}(t-1)\eta^- & \text{für} \quad S(t-1)S(t) < 0 \\ \triangle_{ij}(t-1) & \text{sonst} \end{cases}$$

mit:
$\triangle_{ij}(t)$ = Betrag der Gewichtsänderung zum Zeitpunkt t
$\triangle w_{ij}(t)$= Änderung des Gewichts w_{ij} zum Zeitpunkt t
$0 < \eta^- < 1 < \eta^+$
$S(t) = \frac{\partial E}{\partial w_{ij}}(t)$

$\Longrightarrow$ Der Betrag der Gewichtsänderungen hängt nicht mehr vom Betrag der Steigungen, sondern nur noch von den Vorzeichen zum aktuellen bzw. vorangegangen Zeitpunkt derer ab.

Ist $sign(\triangle w_{ij}(t)) \neq sign(\triangle w_{ij}(t-1))$, so war die vorgenommene Gewichtsänderung zu groß und ein lokales Minimum wurde übersprungen. Man macht diesen Fehler rückgängig, indem man das Gewichtes auf seinen vorherigen Wert ($-\triangle w_{ij}(t)$)zurücksetzt.

RProp verwendet nur das Vorzeichen der partiellen Ableitung zur Bestimmung der individuellen Richtung der Gewichständerung.
Insgesamt gilt also für die Gewichtsänderung:

$$w_{ij}(t+1) = w_{ij}(t) + \triangle w_{ij}(t)$$

mit:

$$\triangle w_{ij}(t) \begin{cases} -\triangle_{ij}(t) & \text{für} \quad S(t-1)S(t) > 0 \quad und \quad S(t) > 0 \\ +\triangle_{ij}(t) & \text{für} \quad S(t-1)S(t) > 0 \quad und \quad S(t) < 0 \\ -\triangle w_{ij}(t-1) & \text{für} \quad S(t-1)S(t) < 0 \\ -sgn(S(t))\triangle_{ij}(t) & \text{sonst} \end{cases} \tag{2.61}$$

oder auch so:

$$\triangle w_{ij}(t) \begin{cases} -\triangle_{ij}(t) & \text{für} \quad S(t) > 0 \\ +\triangle_{ij}(t) & \text{für} \quad S(t) < 0 \\ 0 \quad \text{sonst} \end{cases} \tag{2.62}$$

Beachte: Ist S(t)=0, so hat man bereits das gesuchte Minimum gefunden und w_{ij} muss nicht mehr modifiziert werden.

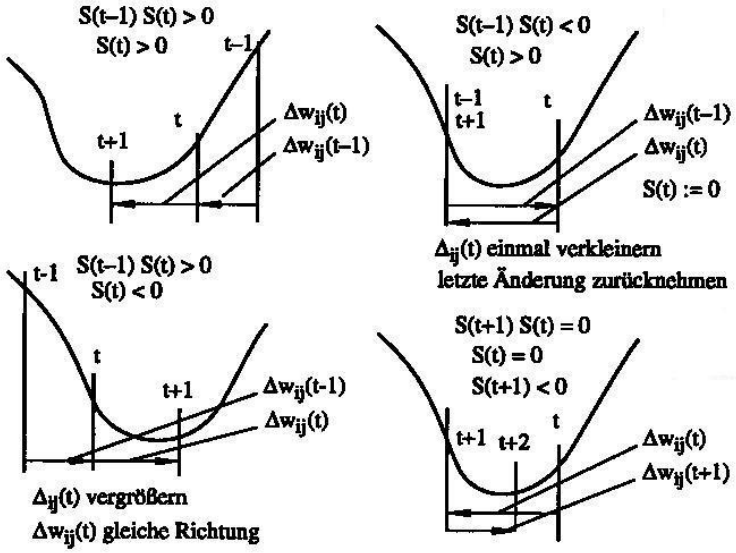


Abbildung 2.39.: Gewichtsänderungen bei RProp [Zell]

Nach Untersuchungen durch [Riedmiller] sind η^+ mit 1.2 und η^- mit 0.5 optimal belegt.
RProp ist ein offline-Trainingsverfahren, d.h. zu jedem Iterationsschritt erfolgt die Anpassung der Verbindungsgewichte erst nachdem alle Trainingsmuster präsentiert wurden.

Da die Resilient Propagation nicht die Werte der partiellen Ableitungen berücksichtigt, sondern lediglich deren Vorzeichen, scheint es sich nicht um ein GradientenVerfahren zu handeln. Allerdings sorgt Gleichung (2.62.) dafür, dass der Aktualisierungswert und die aktuelle partielle Ableitung stets verschiedene Vorzeichen haben (es sei denn, beide sind Null). Deshalb ist das Skalarprodukt zwischen Aktualisierungsvektor und Gradienten immer negativ, so dass (theoretisch jedenfalls) die Aktualisierung eine Verkleinerung des Fehlers zur Folge hat. Insofern handelt es sich also doch um eine Art Gradientenverfahren, so dass das Vorgehen auch mehrdimensional betrachtet, mathematisch gerechtfertigt werden kann.

2.4. Neocognitron

In diesem Abschnitt möchte ich kurz auf ein Netzwerk eingehen, das besonders im Zusammenhang mit der Mustererkennung bekannt geworden ist, das Neocognitron. Es wäre eine interesannte Alternative zu den von mir erstellen Netzen für die Zeichenerkennung (3.3).

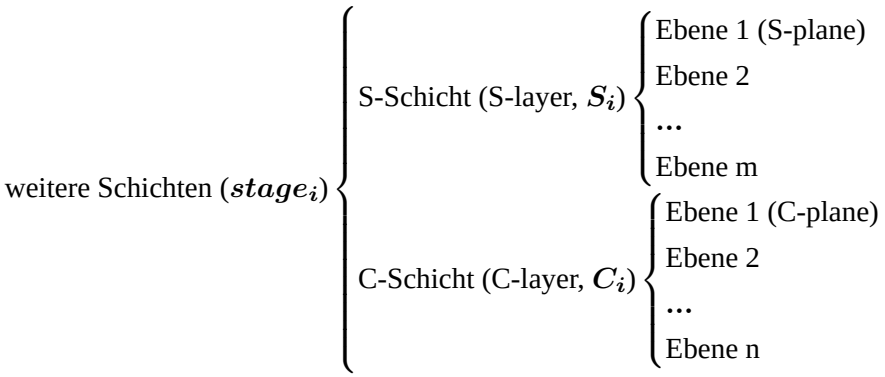
Kunihiko Fukushima stellte 1980 ein Modell zur deformationsinvarianten Mustererkennung vor, das Neocognitron, entstanden aus dem Cognitron, einem früheren Modell zur visuellen Mustererkennung im Gehirn.

Das Neocognitron erkennt Objekte unahbhängig von Position, Verformung, Überdeckung und Verzerrung. Fukushimas Ziel war es, die Sensibilität des Cognitrons gegenüber Verschiebungen (oder anderen Deformationen) zu beheben.
Allerdings ist das Neocognitron nicht invariant gegenüber Rotationen!
Mit seinen selektiven Verbindungen zwischen den hierarchisch aufgebauten Schichten, ist dieses Netzwerk einzigartig und sehr speziell. Der Verbindungsaufbau innerhalb dieses besonderen feedforward Netzes ist äußerst symmetrisch.

2.4.1. Architektur

Grundsätzlich kann das Netzwerk des Neocognitrons in folgende Bestandteile zerlegt werden:

- Eingabeschicht (input layer, $stage_0$)
-



- Die letzte C-Schicht ist die Ausgabeschicht (auch Kategorieschicht)

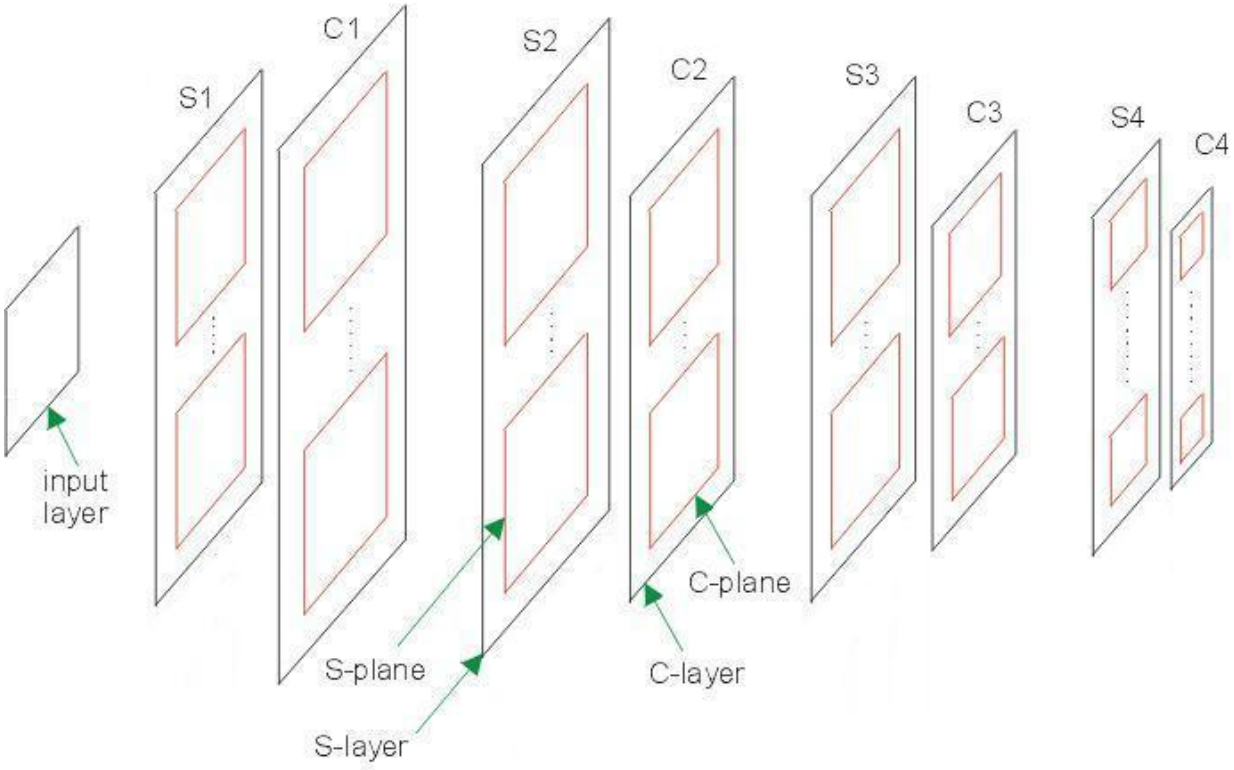


Abbildung 2.40.: Neocognitron [10]

Die Eingabeschicht ist ein rechtecktiges Feld aus Rezeptoren, die nur binäre oder reelle Eingaben verarbeiten können.
Es folgen (immer paarweise angeordnet) S-Schichten (simple Neurons) und C-Schichten (complex neurons)

Auf den S- und C-Schichten gibt es jeweils rechteckige Ebenen (S-plane/C-plane). Auf diesen Ebenen sind die Zellen (S-Zellen/C-Zellen) angeordnet.

S-Zellen (simple neurons)

Die S-Ebenen bestehen aus vielen S-Zellen. Die Anzahl der Zellen nimmt dabei in den höheren Schichten des Netzwerkes ab.
Innerhalb der Ebenen werden die Zellen gruppiert. Für alle S-Zellen in einer S-Schicht ist die Größe der Gruppe (connection area) gleichbleibend und wird beim Erstellen des Netzes festgelegt.

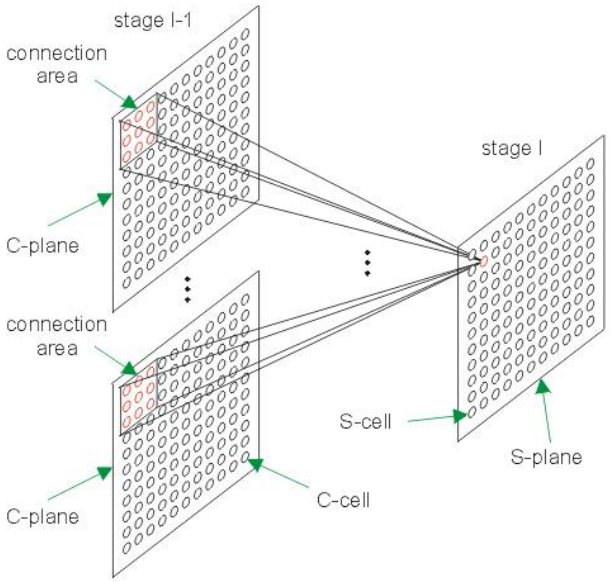


Abbildung 2.41.: S-Ebene mit S-Zellen [10]

Die S-Zellen extrahieren spezielle Eigenschaften an einer bestimmten Stelle des gegebenen Bildes und werten die Ausgabe der C-Zellen aus. Jede Ebene reagiert auf eine separate Eigenschaft des Musters.
Alle S-Zellen in einer Ebene reagieren auf die gleiche Eigenschaft.
Eine S-Zelle wird genau dann aktiv, wenn ein bestimmtes Muster an einer genau bestimmten Stelle der vorhergehenden C-Schicht gefunden wird.

Die erste S-Schicht erkennt primitive Eigenschaften (Liniensegmente), spätere Schichten können globalere Muster erkennen.

C-Zellen (complex neurons)

Die C-Ebenen bestehen, genau wie die S-Ebenen, aus vielen C-Zellen. Die Anzahl der Zellen nimmt dabei in den höheren Schichten des Netzwerkes so stark ab, dass in der letzten Ebene jede Gruppe nur noch aus einer Zelle besteht. Innerhalb der Ebenen werden die Zellen gruppiert. Für alle C-Zellen in einer C-Schicht ist die Größe der Gruppe (connection area) gleichbleibend und wird beim Erstellen des Netzes festgelegt.

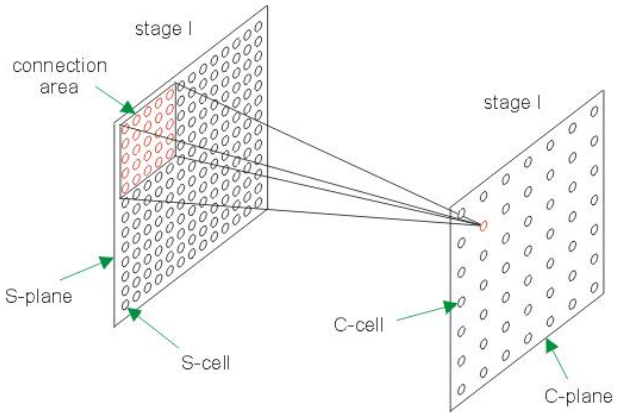


Abbildung 2.42.: C-Ebene mit C-Zellen [10]

Die C-Zellen erhalten Eingaben von kleinen konfigurierten Bereichen aus der S-Schicht und evaluieren deren Ausgabe.
Je mehr S-Zellen der Gruppe aktiv sind, bzw. je höher deren Aktivität ist, desto größer ist der Ausgabewert der C-Zelle.
Die C-Zelle fasst also die Aktivierung verschiedener S-Zellen zusammen und somit reagiert C_i auf dieselben Eigenschaften wie S_i.
Um eine Verschiebungs-Invarianz bei der Eingabe zu erreichen, sind die Verbindungen speziell in räumlichen Gruppen angeordnet.
Die C-Zellen sind also nicht nur für die Selektion der Merkmale aus der S-Schicht verantwortlich, sondern kompensieren zugleich Verschiebungen.
Die Zellen sind gegenüber Positionsänderungen in den Eingabemustern tolerant.
Damit wird die positions- und skalierungsinvariante Erkennung von Mustern möglich gemacht.

Ablauf:

Die Merkmalsextraktion im Neocognitron erfolgt hierarchisch.

- Niedrigere Schichten: einfachere Eigenschaften erkennen und kombinieren
  (S_1/C_1 erkennen die **lokal** vorhandenen Merkmale der Eingabeschicht.)
- Folgende Schichten:
  - Gruppen von Teilmustern in den vorhergehenden Schicht werden erkannt.
  - Erkenne größere Teilmuster, bestehend aus den einfacheren Teilmustern.
  - Merkmale werden zu einem größerem Bereich zusammengefasst, globale Eigenschaften/Konzepte höherer Ordnung werden erkannt.

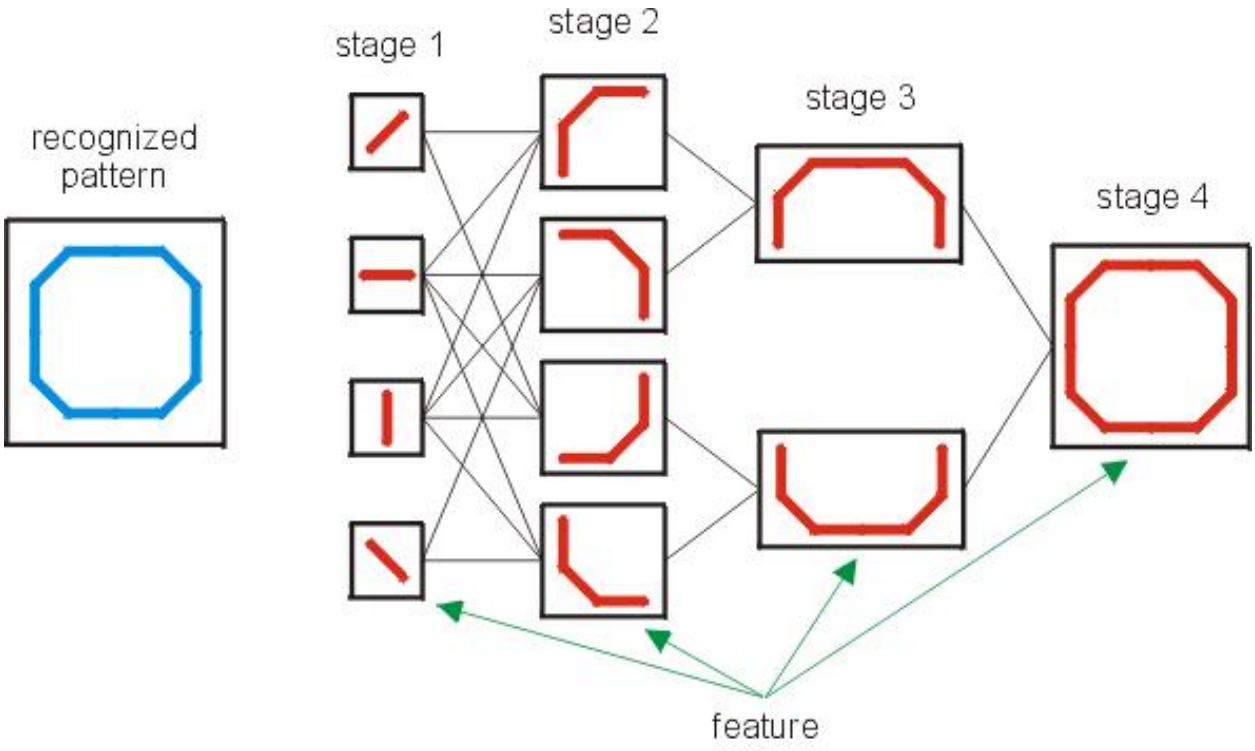


Abbildung 2.43.: hierarchische Merkmalsextraktion [10]

2.4.2. Lernstrategie (Training)

Das Neocognitron kann sowohl überwacht als auch unüberwacht trainiert werden.
Die besten Erfolge wurden durch überwachtes Einzeltraining der S-Schichten erzielt. Daher möchte ich auch nur auf diese Art von Lernen eingehen.

Man sollte beachten, dass nur die Verbindungen, die zu den S-Zellen hinführen, variabel also trainierbar sind. Alle Eingabeverbindungen die zu den C-Zellen führen haben konstante Gewichtungen und sind damit nicht trainierbar.

Der Trainingsprozeß erfolgt schrittweise von der niedrigsten Schicht S_1 bis hin zu S_n.

Die Gewichte der S-Zellen werden mit kleinen, unterschiedlichen, positiven Werte vorbelegt, die der C-Zellen mit festen, positiven Werten (werden auch nicht verändert).

Die Zelle einer Gruppe, die den stärksten Impuls erzeugt, am meisten reagiert, ist repräsentativ für die Gruppe.
Die Gewichte aller anderen Zellen in der überlegenen Zellengruppe bekommen den Gewichtungswert von dieser repräsentativen Zelle. Die Gruppe mit den stärksten Ausgaben ist die repräsentative Gruppe aller

Gruppen der Ebene für die Gewichtsmodifikation im Lernprozess.

Ziel des Trainings:

- Die Zelle reagiert noch mehr auf diesen Impuls.
- Bei verschiedenen Mustereigenschaften sollen verschiedene Ebenen reagieren.
- Alle repräsentativen Zellen werden sensibler gegenüber einem bestimmten Muster und reagieren weniger auf andere Muster.

Die Aktualisierungsregel und der Trainingsprozess für die Gewichtungen aller S-Schichten ist äuqivalent zu den Vorgaben der S_1 Schicht.

2.5. Verkleinerung von Netzen / Pruning

Das ”Pruning” ist ein Verfahren zur Reduzierung der Größe von neuronalen Netzen.
Es ist sinnvoll, wenn die Festlegung einer problemnahen Netztopologie a priori nicht möglich ist.

Meist gibt es eine sehr große Zahl an freien Parametern (Gewichte, Schwellenwerte, Parameter der Aktivierungsfunktion, etc.) innerhalb eines Netzes, aber nicht genug Trainingsmuster, um all diese Parameter optimal einzustellen.
Das Resultat ist ein Netzwerk, das zwar gut trainiert ist, aber bei neuen Daten relativ schlechte Ergebnisse liefert.

Möglichkeiten zur Größenreduktion

- Ausdünnen der Gewichtsmatrix (weight pruning)
  - weight-decay
  - optimal brain damage (OBD)
  - optimal brain surgeon (OBS)
- Löschen von Neuronen (unit pruning)
  - reduziere die Zahl der Neuronen in den verdeckten Schichten
  - reduziere die Anzal der Eingabezellen (input unit pruning)

Pruning Verfahren lassen sich mathematisch gesehen im Wesentlichen in drei Kategorien unterteilen:

- Sensitivitätsverfahren (OBD, OBS, etc.)
- Straftermverfahren
- stochastische Optimierungsverfahren

Die Taktik um ein Netz zu optimieren liegt meist in **destruktiven Pruning Algorithmen**:

- beginne mit einem trainiertem, voll vernetzem neuronalen Netzwerk
- lösche ausgewählte Netzwerkelemente (Verbindungen/Gewichte oder Neuronen)
- trainiere das Netz erneut
- wiederhole den letzten Schritt bis ein vorgegebenes Stoppkriterium erfüllt ist (hat ein Neuron keinerlei Verbindungen mehr, wird es automatisch gelöscht)

2.5.1. Weight Pruning

Dem ”Weight-Pruning” werden ganz allgemeine Verfahren zugeordnet, die aus einem überdimensionierten Netzwerk sukzessive alle nicht benötigten Gewichte entfernen.

Streng gesehen wäre das Weight Pruning der Überbegriff zum Neuron Pruning und Input Pruning, denn Weight Pruning kann ebenso komplette Units ”neutralisieren”, wie Inputs ”abklemmen”.

Das Training, mit weight pruning ist ein iteratives Verfahren. Der Backpropagation Trainingsprozess sorgt für eine Optimierung der gegebenen Zielfunktion.

Algorithmus:

- optimiere die Zielfunktion bis zum Stopp-Punkt
- berechne und nutze einen Testwert (Standard-Gewichtspruning) für jedes Gewicht
- Gewichte mit kleinsten Testwerten entfernen

2.5.2. Input pruning

Das Löschen von Eingabezellen ist ein Spezialfall des Weight Prunings.
Überflüssige Eingabedaten, die keine Relevanz auf das Ergebnis haben, werden hierbei entfernt.

Bei [ZimHerFinn] findet man den Ansatz, das Input Pruning über eine Sensitivitätsanalyse zu lösen. Dabei wird zu Beginn der Analyse der Fehler des Netzwerkes gemessen. Anschließend werden nach und nach alle Gewichte, die zu jeweils betrachteten Eingabeneuronen führen, auf Null gesetzt und der sich ergebende Fehler gemessen. Zuletzt braucht nur noch die Liste der Fehlerdifferenzen (Fehler bei Entfernung des ausgewählten Eingabeneurons minus usprünglicher Fehler) in aufsteigender Reihenfolge sortiert werden. Diejenigen Eingabeneuronen, nach deren Entfernen das Netzwerk die geringste Fehlerzunahme aufweist, stehen in der Liste ganz vorne. Sowohl die Rangfolge der Eingabeneuronen als auch die Höhe der Fehlerdifferenzen zeigen schnell, welche Eingabeneuronen entfernt werden müssen.

2.5.3. Weight Decay

Dieses Verfahren wurde schon als Erweiterung der Backpropagation erwähnt. Man führt zur Fehlerfunktion einen Term hinzu, der betragsmäßig große Gewichte ”bestraft”:

$$E_{neu} = E + \frac{1}{2} d \sum_{i,j} w_{ij}^2$$

Es entsteht eine Modifikationsregel, die gleichzeitig Netzwerkfehler und Gewichte minimiert:

$$\triangle w_{ij}(t+1) = \eta o_i \delta_i - d w_{ij}(t)$$

Strategie der Größenreduktion: Alle Verbindungsgewichte mit $|w_{ij}| \leq \epsilon$ (ϵ sehr klein) werden gelöscht. Problem: Eigentlich würde man, um gut zu generalisieren, für verschiedene Verbindungsgewichte verschiedene decay Konstanten d benötigen.
Mindestens bräuchte man 3 Stück:

1. Eingabe → Hidden
2. Hidden → Hidden
3. Hidden → Ausgabe

Mit dieser Modifikation steigt der Rechenaufwand allerdings gewaltig an.
Eine Alternative wäre das ”Bayesian learning”, siehe [Neal] oder [21]

2.5.4. Magnitude Based Pruning

Bei dem Magnitude Based Pruning (MBP) werden nach jedem Training alle Gewichte, die unterhalb einer Schwelle liegen, gelöscht. Falls nur ein Gewicht eliminiert werden soll, dann das kleinste. Dieses eher simple Verfahren hat sich in der Anwendung als sehr effizient erwiesen, insbesondere im Zusammenhang mit einem zusätzlichen Strafterm für die Größe der Gewichte (weight decay).

Der einfache und sehr schnelle Algorithmus des MBP liefert kaum schlechtere Ergebnisse wie andere komplexere Algorithmen, solange nicht zu viele Verbindungen gelöscht werden.

2.5.5. Optimal Brain Damage (OBD)

Optimal Brain Damage ist eine second-order Methode. Man versucht analytisch den Effekt einer Änderung des Gewichtsvektors auf die Fehlerfunktion E vorherzusagen.
Ziel: Setze das Gewicht auf ”0”, das die geringste Änderung der Fehlerfunktion bewirkt.

Betrachte die Approximation des Lernfehlers durch die Taylorreihe zweiten Grades der Zielfunktion mit dem Gewicht:

$$\triangle E = \underbrace{\sum_i \frac{\partial E}{\partial w} \triangle w}_{=0} + \underbrace{\frac{1}{2} \frac{\partial^2 E}{\partial w^2} \triangle w^2}_{dominiert!} + O(||\triangle W||^3)$$

Folgende Annahmen erleichtern die Näherung:

- Der aktuelle Trainigszustand des Netzes ist so weit fortgeschritten, dass man sich bereits in der Nähe eines lokalen Minimums von $\boldsymbol{E}$ befindet. Daher gilt: $\boldsymbol{E'(w) = 0}$
- Die Umgebung des Minimums sei lokal quadratisch.
- Die gemischten zweiten Ableitungen werden vernachlässigt. Die Hesse-Matrix wird durch eine Diagonalmatrix angenähert.

Es ist festzustellen, dass $\frac{\partial E}{\partial w} = \mathbf{0}$ in einem lokalen Minimum gilt und die Entwicklung der Zielfunktion bei kleinen Gewichtsveränderungen durch den Fehlerterm zweiten Grades dominiert wird. Daraus folgt, dass $\triangle \boldsymbol{E}$ sich auf folgenden Term reduziert:

$$\triangle \boldsymbol{E} = \frac{\boldsymbol{\partial^2 E}}{\boldsymbol{\partial w^2}} \triangle \boldsymbol{w^2}$$

Jetzt wird eine effiziente Methode zur Berechnung der zweiten Ableitung benötigt:

$$\boldsymbol{o_j = f_{act}(net_j)} \qquad \boldsymbol{net_j = \sum_i o_i w_{ij}} \qquad \boldsymbol{E = \sum_j (t_j - o_j)^2}$$

$$\boldsymbol{\Rightarrow \frac{\partial^2 E}{\partial w_{ij}^2} = \frac{\partial^2 E}{\partial net_j^2} o_i^2}$$

1) Für Neuronen der Hidden Schicht gilt:

$$\boldsymbol{\frac{\partial^2 E}{\partial net_j^2} = f'_{act}(net_j) \sum_k w_{jk}^2 \frac{\partial^2 E}{\partial net_k^2} - f''_{act}(net_j) \frac{\partial E}{\partial net_i}}$$

2) Für Neuronen der Ausgebeschicht gilt:

$$\boldsymbol{\frac{\partial^2 E}{\partial net_j^2} = 2 f'_{act}(net_j)^2 - 2(t_j - o_j) f''_{act}(net_j)}$$

Algorithmus 2.3. *[Zell]*

1. *Wähle eine vernünftige Netzwerkarchitektur.*
2. *Trainiere das Netz, bis eine Lösung gefunden ist (lokales Minimum).*
3. *Berechne $\boldsymbol{h_{kk}}$ für jedes Gewicht k.*
4. *Berechne die saliencies $\boldsymbol{s_k}$ für jedes Gewicht k: $\boldsymbol{s_k = \frac{1}{2} h_{kk} w_k^2}$.*
5. *Sortiere die Gewichte nach den saliencies $\boldsymbol{s_k}$ und setze einige der Gewichte mit geringstem Wert $\boldsymbol{s_k}$ auf Null.*
6. *Gehe zu 2.*

Bemerkung 2.5. *Die Methode kann erst im lokalen Minimum eingesetzt werden, so dass zu diesem Zeitpunkt in der Regel bereits eine Überanpassung an die Trainingsdaten eingetreten ist. Aus diesem Grund wird in der Praxis die Methode der statistischen Signifikanz bevorzugt eingesetzt.*

Im Unterschied von OBS begnügt man sich hier jedoch mit der Diagonale der Hessematrix und erspart sich dadurch Rechenzeit. Durch diese zusätzliche Vereinfachung geht jedoch der Vorsprung gegenüber der simpleren MBP-Methode verloren. Zudem sollte auch bei OBD in einem Iterationsschritt nur ein Gewicht entfernt werden.

2.5.6. Optimal Brain Surgeon (OBS)

Mathematisch eleganter ist die Verallgemeinerung von OBD, das von Hassibi und Storck entwickelte Verfahren, Optimal Brain Surgeon.
Bei OBS wird der Fehlerzuwachs abgeschätzt, der entsteht, wenn ein Verbindungsgewicht gelöscht wird. Die Gewichte, bei denen der kleinste Zuwachs entsteht, werden entfernt. Der Lernfehler wird durch die Taylorreihe zweiten Grades (quadratisches Polynom) um das lokale Minimum approximiert, das nach dem Lernvorgang mit Gradientenabstieg erreicht wurde.
OBS nutzt Informationen der zweiten Ableitung der Gesamtfehlerfunktion nach den Gewichten, zur Bestimmung derjenigen Netzgewichte, deren Löschen mit der geringsten Zunahme des Gesamtfehlers verbunden ist. Gleichzeitig werden die jeweils übrig gebliebenen Gewichte als Resultat der gelösten, restringierten Minimierungsaufgabe an die veränderte Situation angepasst. (Im Gegensatz zum OBD, wird die komplette Hessematrix verwendet und nicht nur durch eine Diagonalmatrix angenähert.)
Gehe wie folgt vor:
1) Zur Approximation wird für jedes einzelne Gewicht der Zuwachs des Lernfehlers beim Entfernen dieses Gewichts berechnet.
2) Das Gewicht mit dem geringsten Lernfehlerzuwachs wird dann entfernt und das Verfahren wird wiederholt.

$$\boldsymbol{\Delta E = (\frac{dE}{dW})^T \Delta W + \frac{1}{2}(\Delta W)^T H \Delta W + O(\|\Delta W\|^3)}$$

mit der Hesse-Matrix $\boldsymbol{H = \frac{\partial^2 E}{\partial W^2}}$. Die Annahmen zur Vereinfachung des Terms bei OBD werden ebenfalls verwendet. Das Löschen eines Gewichts $\boldsymbol{w_q}$ ist durch $\boldsymbol{\Delta w_q + w_q = 0}$ gegeben, wobei die Änderung $\boldsymbol{\Delta w_q}$ das Gewicht auf Null bringt, bzw. durch

$$\boldsymbol{e_q^T \Delta W + w_q = 0} \tag{2.63}$$

wobei $\boldsymbol{e_q}$ der q-te Einheitsvektor ist, gegeben. Es ist $\boldsymbol{e_q^T \Delta W = \Delta w_q}$.

Ziel: Finde folgendes Minimum:

$$\boldsymbol{\min_q \{\min_{\Delta W} \{\frac{1}{2} \Delta W^T H \Delta W\} : e_q^T \Delta W + w_q = 0\}} \tag{2.64}$$

Um die Gleichung zu lösen, bildet man die Lagrange-Funktion

$$L = \frac{1}{2}\Delta W^T H \Delta W + \lambda(e_q^T \Delta W + w_q), \tag{2.65}$$

wobei λ der Lagrange-Multiplikator ist.

Man erhält also ein restringiertes Optimierungsproblem
(Minimiere $\frac{1}{2}\Delta W^T H \Delta W$ unter der Nebenbedingung $e_q^T \Delta W + w_q = 0$)
für das die Karush-Kuhn-Tucker (KKT) Bedingungen (siehe [Alt]) angewendet werden können.
Nach Bildung funktionaler Ableitungen, Anwendung der Nebenbedingung und Matrixinversion nach [Zell] erhält man, zusammen mit den KKT Bedingungen die folgenden Gleichungen:

$$\Delta W = -\frac{w_q}{[H^{-1}]_{qq}} H^{-1} e_q \tag{2.66}$$

$$L_q = \frac{1}{2}\frac{w_q^2}{[H^{-1}]_{qq}}. \tag{2.67}$$

$$s_k = \frac{w_k^2}{[H^{-1}]_{kk}} \tag{2.68}$$

Algorithmus 2.4. *[HasSto]*

1. *Auswahl einer problemgerechten Netzwerkarchitektur*
2. *Training des Netzwerks in ein (lokales) Minimum der Gesamtfehlerfunktion*
3. *Berechnung von H^{-1} (der Inversen der Hesse-Matrix der Fehlerfunktion) für alle Gewichte des Netzwerks*
4. *Bestimmung der **saliency** L_q des Netzwerks gemäß der Gleichung (2.32.) für alle w_q*
5. *Falls der kleinste **saliency**-Wert eines Gewichts w_q viel kleiner als E ist, lösche Gewicht q und gehe zu 6, sonst zu 7*
6. *Anpassung aller restlichen Gewichte gemäß Gleichung (2.31.). Nachtrainieren um festgelegte Anzahl von Epochen. Zurück zu Schritt 3*
7. *Weiteres Löschen von Gewichten führt zu starkem Anwachsen von E. Den Zustand vor dem Löschen des letzten Gewichts wieder herstellen (fakultativ)*

Bemerkung 2.6. *Ein Problem ist die Berechnung von H^{-1}. Eine relativ effiziente Methode findet man in [HasSto]. Dort wird die Hesse-Matrix als Kovarianzmatrix bestimmter Gradienten geschrieben, wodurch sich H und H^{-1} durch Lösung iterativer Gleichungen berechnen lassen.*

Die Zeitkomplexität pro Matrixinversion liegt in der Ordnung $O(m \cdot p \cdot n_w^2)$ (m ist die Anzahl der Ausgabeneuronen, p die Anzahl der Trainingsmuster und n_w die Anzahl der Gewichte im Netz).

OBS ist von den hier vorgestellten Methoden das Verfahren, bei dem die meisten Gewichte des Netzes gelöscht werden.

Als Nebenprodukt dieser Berechnung gibt OBS auch den Gewichtänderungsvektor aus, mit dem die Taylor-Approximation diesen Lernfehlerzuwachs erzielt. Wäre die Approximation exakt, so hätten wir damit wieder ein lokales Minimum erreicht und das Verfahren könnte ohne Nachlernen iteriert werden.
Aufgrund der Ungenauigkeiten ist jedoch ein Nachtrainieren erforderlich.

Dieses Verfahren ist äußerst rechenintensiv, da in jedem Iterationsschritt die vollständige Hessematrix (zweite Ableitungen des Lernfehlers) berechnet werden muß. Darüber hinaus ist es im Gegensatz zu MBP bei OBS sehr kritisch, in einem Iterationsschritt mehrere Gewichte zu entfernen.

3. Anwendung / Praxis

3.1. Anwendungen für mehrschichtige feedforward Netze mit Backpropagation

Die mehrschichtigen feedforward Netze (Multilayer feedforward - MLFF) sind mit Abstand die beliebtesten Netze. Sie werden für eine Vielzahl von Aufgabenstellungen fast jedes Anwendungsbereiches eingesetzt. MLFF Netze haben besonders gute Abbildungsfähigkeiten. Steht für eine Anwendung eine geeignete Menge an Trainingsdaten zur Verfügung, so kann das Netz die gewünschte Aufgabe meist mit einer oder höchstens zwei verborgenen Schichten, erfüllen.

3.2. Anwendungsgebiete

Viele Aufgaben sind mit exakten Methoden bisher nur unzureichend gelöst. Die Möglichkeit der Automatisierung, obwohl erwünscht, ist nur partiell gegeben: Personenerkennung, Sprachverstehen, Autofahren, ... Sie haben gemein, dass eine mathematische Modellierung unmöglich oder aufwändig erscheint. Nichtsdestotrotz können die Aufgaben von Menschen nur aufgrund von partieller, expliziter Information, Erfahrung und Übung zufriedenstellend gelöst werden. Man könnte das auch als ”Vorhandensein von Beispielen” bezeichnen. Neuronale Netze sind eine Methode, eine Gesetzmäßigkeit (Funktion, Verhaltensweise, ...) nur mithilfe von Beispielen zu lernen.
Sie sind dabei weder das einzig mögliche Verfahren in diesem Bereich, noch ein Allheilmittel, obwohl sie scheinbar ein universeller Ansatz sind. Bei jedem neuen Problem wird man den Hauptteil der Arbeit für die Problemrepräsentation, die konkrete Anpassung und das Feintuning verwenden. Jedoch erzielen neuronale Netze in dem Fall, dass kein ausreichendes explizites Wissen vorhanden ist, oft erstaunliche Erfolge.
Einige Beispiele für Anwendungen:

Bildverarbeitung

- Erkennen von handgeschriebenen Ziffern
- Erkennen von Personen
- Erkennen von Fehlstellen in Materialien
- Krankheitsdiagnose anhand von Röntgenbildern

Klassifizierung und Diagnose, Prognose

- Krankheitsverlaufsprognose anhand von Daten (z.B. Klassifizierung von Zellen für die Krebsdiagnose)
- Kreditwürdigkeitsprognose
- Situationsbeurteilung

Probleme der Klassifizierung und Diagnose treten sehr häufig und in vielen verschiedenen Gebieten der Wissenschaft auf.
Gerade dieser Bereich ist für den Erfolg der neuronalen Netze vielversprechend, da die neuronalen Adaptionsschemata durch ihre schrittweise Anpassung den herkömmlichen statistischen Verfahren in Flexibilität weit überlegen sind.

Zeitreihenverarbeitung

- Börsenkursprognose
- Wettervorhersage
- Spracherkennung/-erzeugung
- Finanz-Prognose

Steuerung und Optimierung, Robotik Aufgaben der Steuerung und Optimierung sind für neuronale Netze schwer zu realisieren, da die Abbildungsfunktionen, die erlernt werden müssen, meist kompliziert und die Problemeinschränkungen oft widersprüchlich sind.
Nichtsdestotrotz werden MLFF Netze in den folgenden Bereichen recht erfolgreich eingesetzt:

- Steuerung eines führerlosen Fahrzeugs
- Robotersteuerung
- Steuerung von Maschinen
- Qualitätssicherung

Vorhersage Vorhersagen werden in sehr vielen Bereichen getroffen und es gibt zahllose Arten von Prognosen, wie z.B. ob, bzw. mit welcher Wahrscheinlichkeit ein Ereignis auftritt, die Zeit, zu der das Ereignis auftritt, etc.
Das MLFF Netz wird hier meist mit historischen Daten trainiert und muss lernen zu verallgemeinern, um schließlich relevante Prognosen abzugeben.

- Vorhersagen im finanztechnischen Bereich (Börsenkurse bzw. Aktiengewinne, Kreditwürdigkeit)
- Fehlervorhersage
- Vorhersage chaotischer Zeitfolgen

Mustererkennung Die Anwendungen zur Mustererkennung sind relativ anspruchsvoll und eng verwandt mit den kognitiven Aufgaben, die vom Menschen allerdings mühelos bewältigt werden.

- Sprach- / Zeichenerkennung (z.B Handschrifterkennung)
- visuelle Bilderkennung, bzw. Bildinterpretation
- Sprackerkennung- und Generierung
- Signal- / Tabellenanalyse

3.3. Einführung in die Problemstellung der von mir entwickelten Software

Die Abkürzung OCR steht für Optical Character Recognition oder auch optische Zeichenerkennung. Die Idee von OCR ist es, mit Hilfe von Computer und Scanner, geschriebene Texte in editierbare Dateien umzuwandeln.
OCR Software wird heute in Banken zur Automatisierung von Überweisungen oder Einzahlungen, bei der Post zum automatischen Lesen der Postleitzahlen, als Lesehilfe für Blinde, zur Archivierung großer Schriftmengen und in vielen anderen Bereichen eingesetzt.
Im Zuge der Automatisierung in diesen Bereichen bekommt die Zeichenerkennung mit dem ”digitalen Auge” immer mehr Bedeutung.

In meinem Fall handelt es sich um das Scannen und Erkennen der unteren zwei Zeilen eines Personalausweises.

Abbildung 3.1.: Personalausweis (Muster)

Die Software kann z.B. an einem Flughafenterminal eingesetzt werden, wo es aufgrund der steigenden Sicherheitsbedingungen nötig ist, genauere und schnellere Identitätsprüfungen vorzunehmen.

Zur Erkennung sind einige vorverarbeitende Schritte nötig.
Zum Einlesen des Dokuments muss es zuerst gescannt und die Lage der gesuchten Zeilen ermittelt werden. Ist diese gefunden, so wird die Lage der einzelnen Zeichen ermittelt. Sind die Ziffern bzw. Buchstaben extrahiert, so werden sie als kleine Bilddateien separiert und gespeichert. (eine genauere Darstellung dieses Vorgangs findet man u.a. in [18])

Für die Auswertung der Zeichen können verschiedene Technologien benutzt werden.
Eine Möglichkeit ist es, das schwarz-weiß Muster gescannter und separierter Zeichen mit abgespeicherten

Mustern zu vergleichen und bei genug Übereinstimmungen, das entsprechende Zeichen zuzuordnen.
Diese Technik ist allerdings weder größen-, noch verzerrungsinvariant, so dass bei stark verrauschten Daten nur noch geringe Erkennungsquoten zu erwarten sind.

Eine weitere Möglichkeit zur Erkennung ist die Merkmalsextraktion der Zeichen, mit der ich mich bei der Entwicklung meiner Software beschäftigt habe. Hierbei wird ein Zeichen nach seinen geometrischen Merkmalen, z.B. runde oder eckige Formen, Symmetrie, etc., untersucht.
Zur Qualifizierung des Zeichens habe ich sowohl eine eigene Methode als auch neuronale Netze implementiert. Erstere verwendet zur Klassifizierung die Differenzen der numerischen Daten der Merkmale zu denen der Musterzeichen, die Netzwerke arbeiten nach ihren Gesetzen von vorher. Ich habe die Netze mit den Daten der geometrischen Eigenschaften und verschiedenen Lernregeln trainiert.
Zusätzlich habe ich ein Netz mit dem Schwarz-Weiß Muster des Zeichens ”gefüttert” und trainiert.

Diese Methoden erzielen zusammen genommen sehr gute Resultate und liefern auch bei stark verrauschten Daten noch sehr annehmbare Ergebnisse.

3.4. der SNNS

Zur Erstellung der neuronalen Netze, die ich in der Software verwende, habe ich den Stuttgarter Neuronale Netze Simulator (SNNS) benutzt.
Bereits 1988 wurde in der Arbeitsgruppe von Andreas Zell am Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart ein erstest Projekt zur Simulation neuronaler Netze für UNIX Workstations entwickelt. Seither wurden zahlreiche Veränderungen und Neuerungen daran durchgeführt, so dass es sich inzwischen um eine sehr ausgereifte Software handelt.
Die aktuelle Version steht zum freien Download auf den Seiten der Universität Stuttgart und Tübingen zur Verfügung.

Der SNNS ist ein sehr effizienter Simulator zur Generierung, zum Training, zum Testen, oder zur Visualisierung neuronaler Netze.
Mit dem SNNS können verschiedene Netze erstellt werden und es sind alle gängen Lernverfahren implementiert.

3.5. Dokumentation meiner Software

3.5.1. Vorverarbeitung und Laden der Bilder

Die ursprünglichen Bilder sind sehr klein und in Graustufen abgespeichert. Um sie verarbeiten zu können, werden sie erst auf die gewünschte Größe gezoomt. Die Funktion zur Größenveränderung ist dynamisch programmiert und kann das Bild von beliebigen Größen vergrößern oder auch verkleinern - immer auf die Standardgröße $\mathbf{32 \times 32}$.
Schließlich wird das Bild in ein schwarz-weiß Bild gewandelt.

Dafür wird erst ein Schwellenwert für die Entscheidung ob schwarz oder weiß gesucht. Die Vorgabe hierfür ist, dass bei einem durchschnittlichen Zeichen zwischen 500 und 600 Felder von den 1024 (32*32) Feldern schwarz sind. Der Schwellenwert wird solange verändert, bis die Anzahl der schwarzen Kästchen in diesem Intervall liegt.

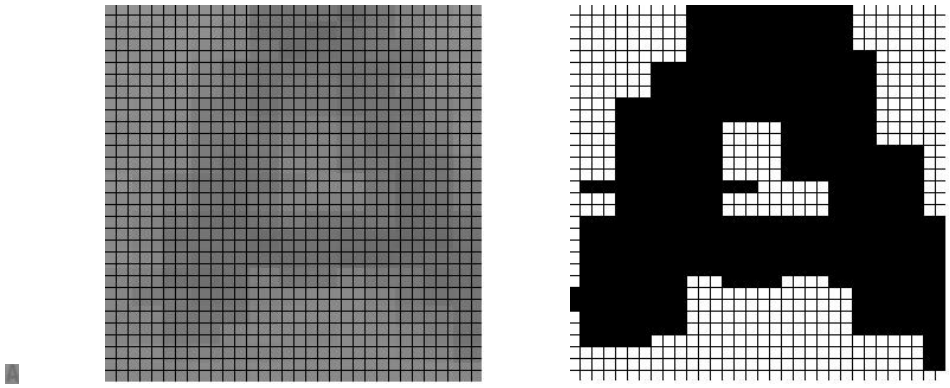


Abbildung 3.2.: Zeichen im Orginalzustand, auf Standardgröße gebracht und in ein schwarz-weiß Bild gewandelt

3.5.2. Geometrische Merkmale:

Für die Codierung der Zeichen werden folgende geometrische Merkmale verwendet:

- Wölbung in x Richtung
- Wölbung in y Richtung
- Schräge
- Diagonalen
- Profilerkennung
- Symmetrie in x Richtung
- Symmetrie in y Richtung
- Schwerpunkt schwarz
- Schwerpunkt weiß
- schwarz-weiß Verhältnisse

Wölbung in x Richtung

Wölbung ist eher ein verwirrender Name für die Funktion. Eigentlich werden die horizontalen, geraden Linen ermittelt und deren Position dann abgespeichert - maximal 3 Positionen pro Zeichen. Es hat sich als günstig erwiesen, das Bild durch 2 horizontale Linien in 3 Bereiche zu teilen. Sind mind. 70% der Linie gefüllt, so wird sie als durchgehend betrachtet und die Position (y-Wert) dieser horizontalen Linie wird gespeichert. Es kann pro Drittel eines Bildes jeweils maximal eine Position eingetragen werden.

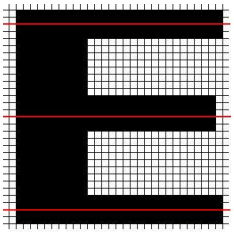


Abbildung 3.3.: Zeichen mit 3 horizontalen Linien

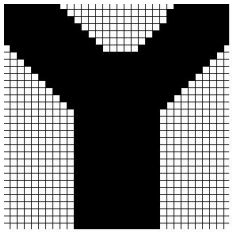


Abbildung 3.4.: Zeichen ohne horizontale Linien

Wölbung in y Richtung

Das gleiche gilt auch für die vertikalen Linien. Wenn mind. 75% der Linie gefüllt ist, wird sie als durchgehend betrachtet und die Position (x-Wert) dieser vertikalen Linie wird abgespeichert. Hierbei kann wiederum pro Drittel des Bildes jeweils maximal eine Position eingetragen werden.

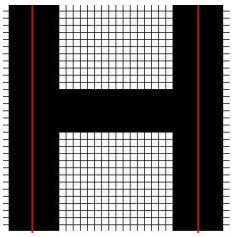


Abbildung 3.5.: Zeichen mit 2 vertikalen Linien

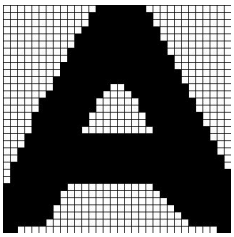


Abbildung 3.6.: Zeichen ohne vertikale Linien

Schräge

Diese Funktion misst (soweit vorhanden) den Neigungswinkel im Zeichen (0 bis 360 Grad).

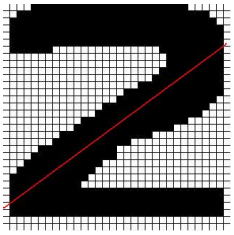


Abbildung 3.7.: Zeichen mit Winkel von 40 Grad

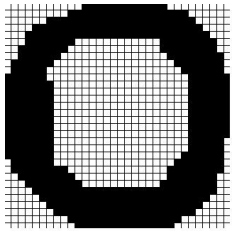


Abbildung 3.8.: Zeichen ohne Winkel

Diagonalen

Dieses Mermal repräsentiert die Anzahl der Diagonalelemente des Zeichens, bzw. bewertet es das schwarz-weiß Verhältnis auf den Diagonalen.

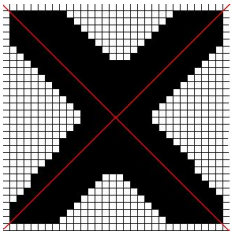


Abbildung 3.9.: Zeichen mit vielen Elementen auf der Diagonale

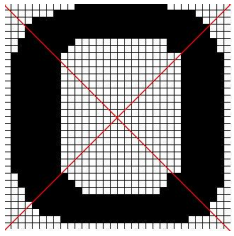


Abbildung 3.10.: Zeichen mit wenigen Elementen auf der Diagonale

Profilerkennung

Das Bild wird entlang der senkrechten und vertikalen Achse abgetastet. Für jede Achse wird von beiden Seiten der erste Schwarzwert ermittelt.

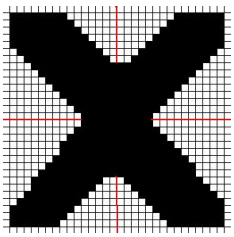


Abbildung 3.11.: Zeichen mit späten Schwarzwerten auf den mittleren Bildachsen

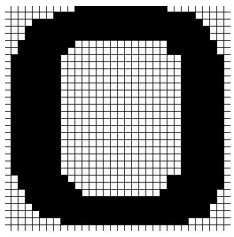


Abbildung 3.12.: Zeichen mit frühen Schwarzwerten auf den mittleren Bildachsen

Symmetrie in x Richtung

Die Symmetrie bezüglich einer horizontalen Achse in der Bildmitte (± 2 Positionen) wird ermittelt.

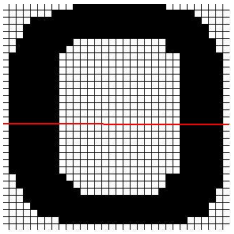


Abbildung 3.13.: Zeichen mit Symmetrie bzgl. der horizontalen Achse

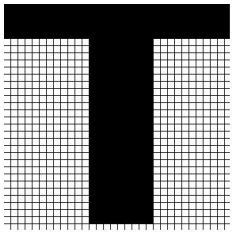


Abbildung 3.14.: nicht-symmetrisches Zeichen

Symmetrie in y Richtung

Die Symmetrie bezüglich einer vertikalen Achse in der Bildmitte (± 2 Positionen) wird ermittelt.

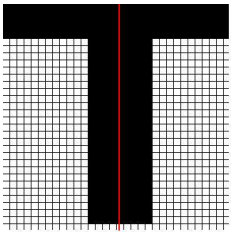


Abbildung 3.15.: Zeichen mit Symmetrie bzgl. der vertikalen Achse

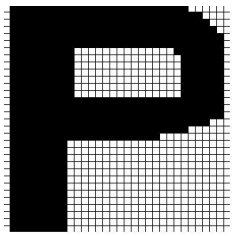


Abbildung 3.16.: nicht-symmetrisches Zeichen

Schwerpunkt schwarz / Schwerpunkt weiß

Von jeder Position aus wird gesucht, wieviele der Kästchen, die sich im umliegenden Quadrat befinden, auch diese Farbe haben. Die Koordinate, mit den meisten vollständigen, gleichfarbigen Quadraten um sich herum, ist der Schwerpunkt dieser Farbe (schwarz/weiß).

Leider ist diese Technik nicht sehr ausgefeilt und ist somit auch das fehleranfälligste Merkmal. Aus diesem Grund habe ich seine Bedeutung gegenüber den anderen Merkmalen durch eine Gewichtung mit dem Faktor $\mathbf{0.3}$ verringert.

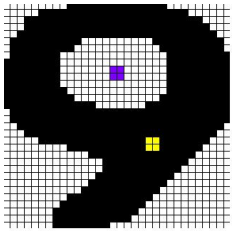


Abbildung 3.17.: Zeichen mit eingezeichneten Schwerpunkten

Schwarz-weiß Verhältnisse

Man teilt das Bild in 9 gleichmäß große Gebiete und untersucht das Schwarz-Weiß Verhältnis in den einzelnen Teilbildern.

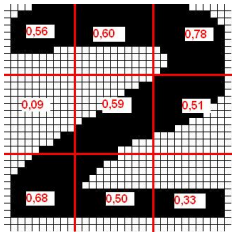


Abbildung 3.18.: schwarz-weiß Verhältnisse

Lernen des Musterzeichensatzes

Beim Lernen des Musterzeichensatzes werden alle ”Musterzeichen”, d.h. 0-9,<,A-Z, auf ihre geometrischen Eigenschaften geprüft und diese dann numerisch codiert in eine Matrix abgespeichert.
Hierbei können verschiedene Musterzeichensätze, also verschiedenen Schriftarten, bzw. evtl sogar Handschriften gelernt werden, allerdings genügt für meine Problemstellung ein einziger Zeichensatz, nämlich genau der, der für den Personalausweis verwendet wird.

3.5.3. Erstellen und Trainieren der neuronalen Netze

Insgesamt habe ich in die Software vier verschiedene, vollständig vernetze, feedforward Netze eingebaut, die ich allesamt mithilfe des SNNS erstellt und trainiert habe.
Es handelt sich dabei um jeweils ein äuqivalentes Netz, das einmal mit Standard Backpropagation und einmal mit Quickpropagation trainiert wurde. Weiterhin wurde dieses Netz mit einem Pruning Algorithmus trainiert und somit stark verkleinert. (Netze 1, 2 und 4)
Das ”Null-/Eins-Netz” ist ein Ansatz, bei dem nicht die geometrischen Merkmale als Eingangsdaten verwendet wurden, wie bei den anderen Netzen, sondern einfach das Bild in einer 0/1 (für schwarz/weiß) Matrix

kodiert. (Netz 3)
Ursprünglich beinhaltete die Software auch ein Netz, das mit Resilient Propagation trainiert war (2.3.7), allerdings lieferte dieses nur sehr schlechte Ergebnisse und wurde daher schließlich wieder entfernt.

Als Trainigsdaten wurden ca. 350 verschiedene Bilder eingelesen, darunter nicht nur die Bilder der Musterzeichen, sondern größten Teils verrauschte Bilder. Es wurde immer mit Trainings und Validierungsdaten trainiert. Als Abbruchkriterium für das Training wurde entweder der Schnitt der beiden Fehlergraphen gewählt, oder ein ausreichend kleiner Fehler der Trainingsdaten, falls der Fehler der Valiedierungsdaten konstant oder weiterhin fallend war.

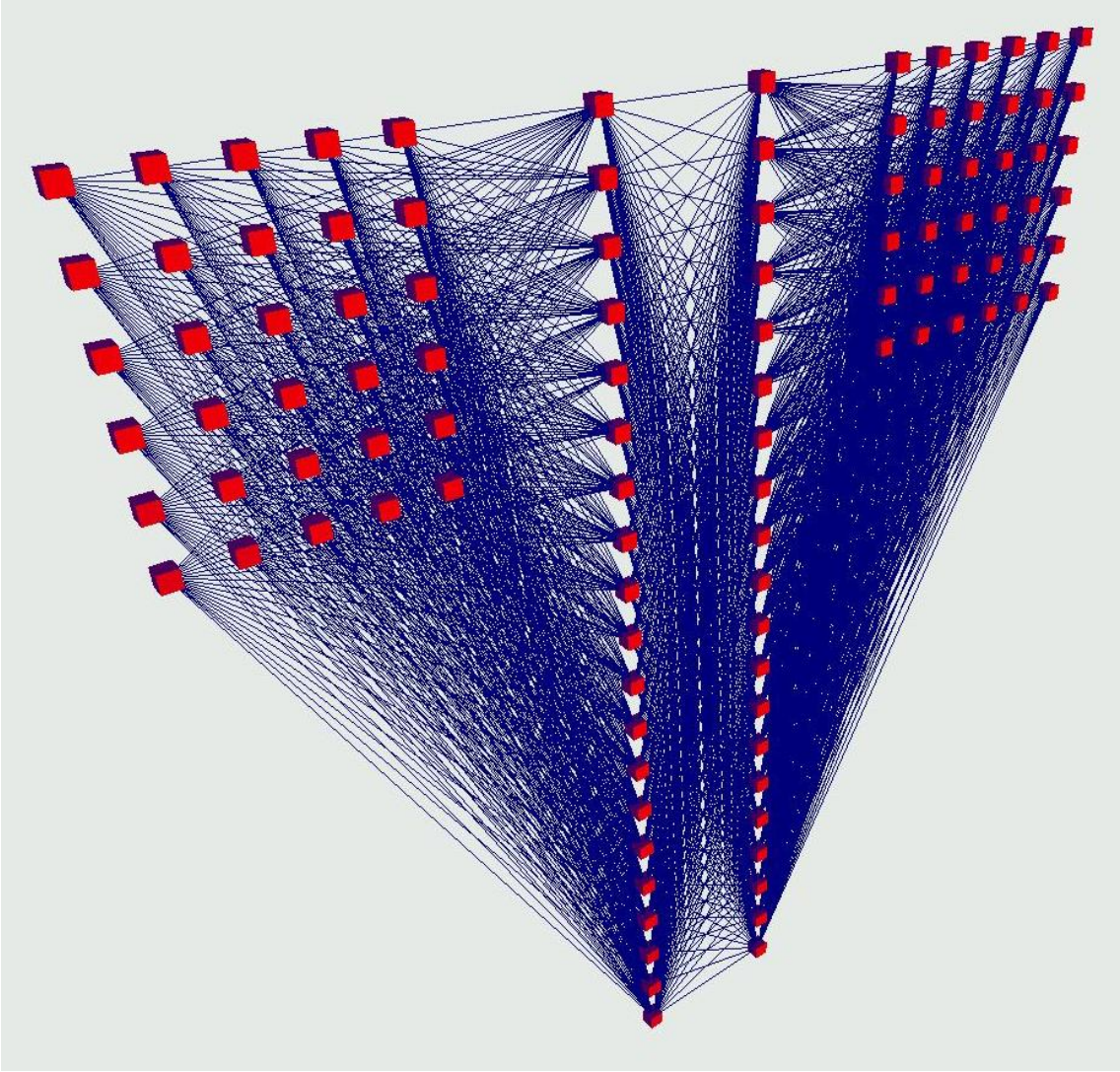


Abbildung 3.19.: untrainiertes neuronales Netz

Erstellen eines neuronalen Netzes

Abbildung 3.20.: SNNS Manager Panel

Vom Manager Panel aus können sämtliche Panels des SNNS aufgerufen werden. Es ist somit das Hauptfenster zur Bedienung des SNNS.

Für das Erstellen der Netze kann zuerst die Art des Netzes (im Manager-Panel unter BIGNET), dann die jeweilige Anzahl und Anordnung der Eingangsneuronen, Neuronen der verdeckten Schichten und der Ausgangsneuronen im oberen Abschnitt des BigNet-Panels festgelegt werden.

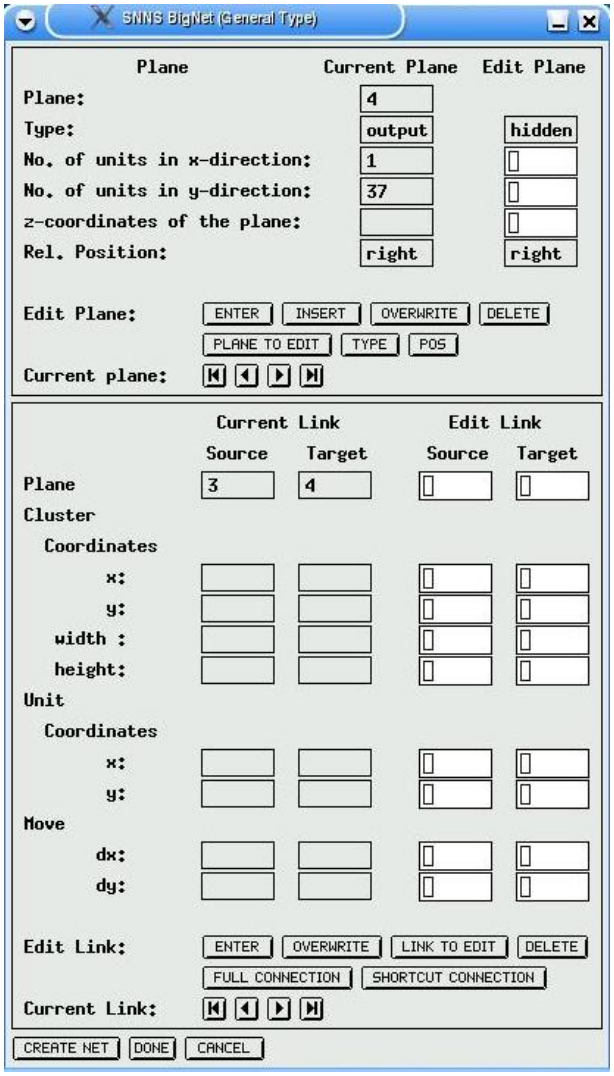


Abbildung 3.21.: SNNS BigNet Panel

Bei der Erstellung der Netze kann auch die Vernetzung der einzelnen Neuronen festgesetzt werden FULL CONNECT oder SHORTCUT CONNECTION, wobei ich erstere verwendet habe, so dass die Neuronen vollständig (ebenenweise) miteinander verbunden sind. Das neue Netz wird mit CREATE NET und DONE vollendet.
Nichtsdestotrotz kann das Netz während der Testphase noch modifiziert werden, beispielsweise können Zellen eingefügt oder entfernt werden, oder ihre Aktivierung verändert werden.

Trainieren eines neuronalen Netzes

Abbildung 3.22.: File Panel

Vom File Panel aus können gespeicherte Patterns, Netze und Konfigurationsdateien geladen werden oder neu erstellte Netze abgespeichert werden.

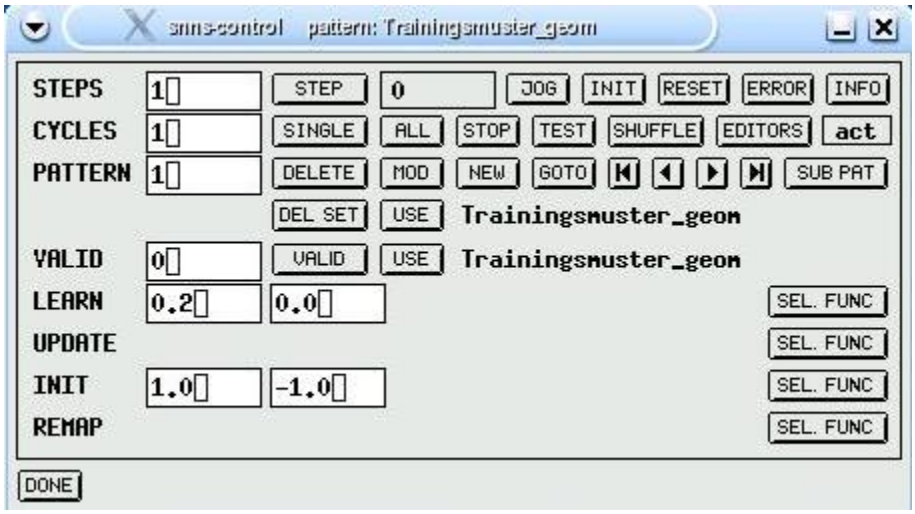


Abbildung 3.23.: Control Panel

Das Control Panel dient zur Steuerung, zum Training und Testen der Netze.

Auswahl von:

- Lernverfahren
- Lernfunktion

- Lernparameter
- Update-Funktion
- Art der Initialisierung der Gewichte
- Anzahl der Trainingszyklen

Wählt man zusätzlich noch SHUFFLE, so werden die Trainingsmuster in zufälliger Reihenfolge präsentiert, was zur Verbeserung der Generalisierungsfähigkeit des Netzes beitragen kann. Wird im SNNS Manager Panel GRAPH aktiviert, so wird der Verlauf des Netzwerkfehlers während des Trainigs graphisch dargestellt. Es kann zwischen den folgenden Fehlern gewählt werden: SSE, MSE und SSE/out.
Dabei gilt für den Sum Squared Error

$$\boldsymbol{SSE} = \sum_{\boldsymbol{p \in patterns}} \sum_{\boldsymbol{j \in output}} \boldsymbol{(t_{pj} - o_{pj})^2}.$$

Der Mean Squared Error (MSE) ist der SSE dividiert durch die Anzahl der Pattern und der SSE/out (SSE pro Ausgabeneuron) ist der SSE dividiert durch die Anzahl der Ausgabeneuronen.
Als Update-Funktion wurde Topological Order gewählt, da diese für feedforward Netze im Allgemeinen am geeignetsten ist [Zell].
Als Aktivierungsfunktion wurde die logistische Funktion und als Ausgabefunktion die Identität gewählt.

Um ds Netzes mit den Patterns zu trainieren, wird aus einer Datei ein Datensatz ($\boldsymbol{*.pat}$) eingelesen. Zu Begin wird die Gewichtung der Verbindungen initialisiert. Die Verbindungen der Neuronen werden mit kleinen, zufällig gewählten Werten, innerhalb eines vorgegebenen Intervalls [$\boldsymbol{\alpha, \beta}$], initialisiert. Hier gilt im Allgemeinen $\boldsymbol{\alpha = -1.0}$, $\boldsymbol{\beta = 1.0}$.
Die Patterns können dem Netz entweder in einer bestimmten Reihenfolge ($\boldsymbol{0-9}$ und $\boldsymbol{A-Z}$) vorgeführt werden, oder in zufälliger Reihenfolge. Die Lernrate und die Fehlerentwicklung kann mit dem Graphik-Panel (File Panel $\longrightarrow$ GRAPH) anhand einer Kurve verfolgt werden.

Netz 1

Eingabeschicht: 31 (1×31) Neuronen
2 verdeckte Schichten mit jeweils 20 Neuronen
Ausgabeschicht: 37 (1×37) Neuronen

Für das Netz wurde das Standard Backpropagation Verfahren (Online Backpropagation) (2.3) als Lerfunktion gewählt.
Bei einem Fehler von $\boldsymbol{0.04189}$ wurde das Training nach 4000 Durchläufen beendet. Ein längeres Training konnte den Fehler nur unwesentlich verringern. Der Fehler der Validierungsdaten blieb ab einem gewissen Zeitpunkt im Verlauf relativ konstant. Der Lernparameter wurde auf $\boldsymbol{\eta = 0.3}$ festgelegt. Der Verlauf des Netzwerkfehlers ist in Abbildung 3.26 dargestellt.

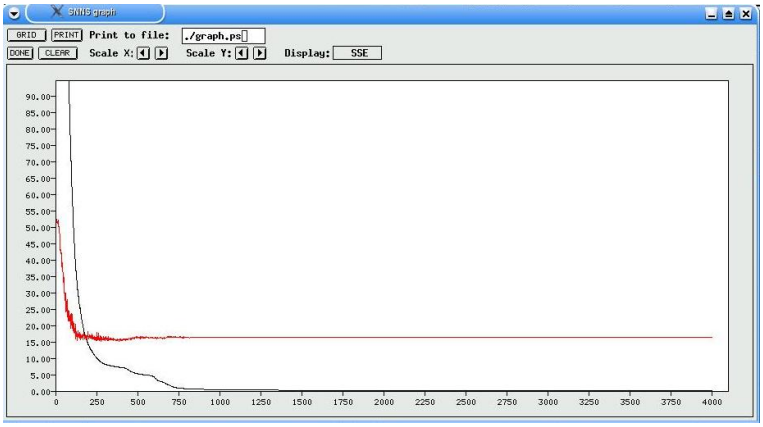


Abbildung 3.24.: Verlauf des Fehlers eines 1×31 feedforward Netzes mit zwei verdeckten Schichten bei Backpropagation

Netz 2

Null/Eins Netz
Eingabeschicht: 1024 (32×32) Neuronen
2 verdeckte Schichten mit jeweils 20 Neuronen
Ausgabeschicht: 37 (1×37) Neuronen

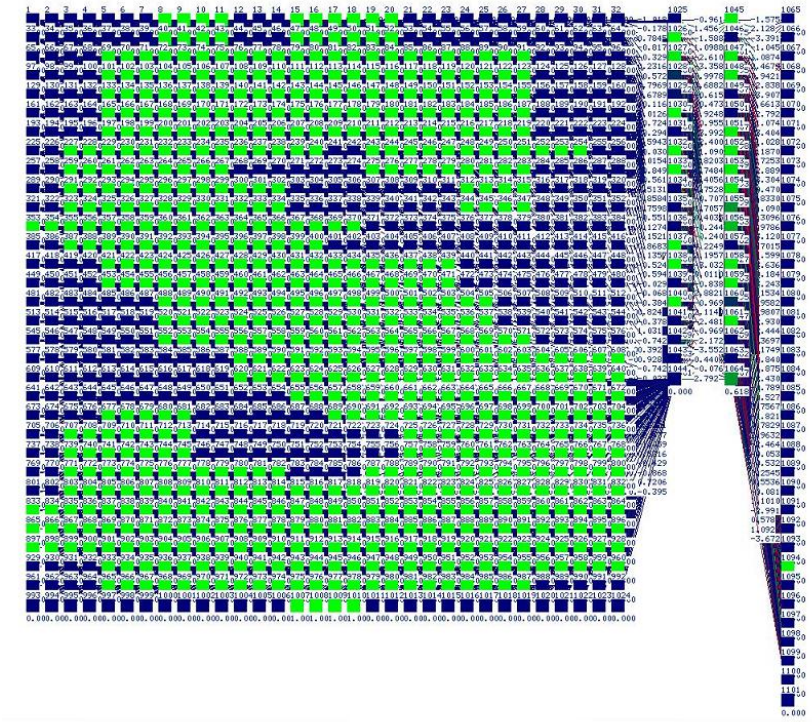


Abbildung 3.25.: Null/Eins Netz

Für das Netz wurde wiederum das Standard Backpropagation Verfahren (Online Backpropagation) (2.3) als Lerfunktion gewählt.
Bei einem Fehler von $\mathbf{1.16}$ wurde das Training wiederum nach 4000 Durchläufen beendet. Als Lernparameter wurde auf $\eta = \mathbf{0.2}$ verwendet. Der Verlauf des Netzwerkfehlers ist in Abbildung 3.26 dargestellt.

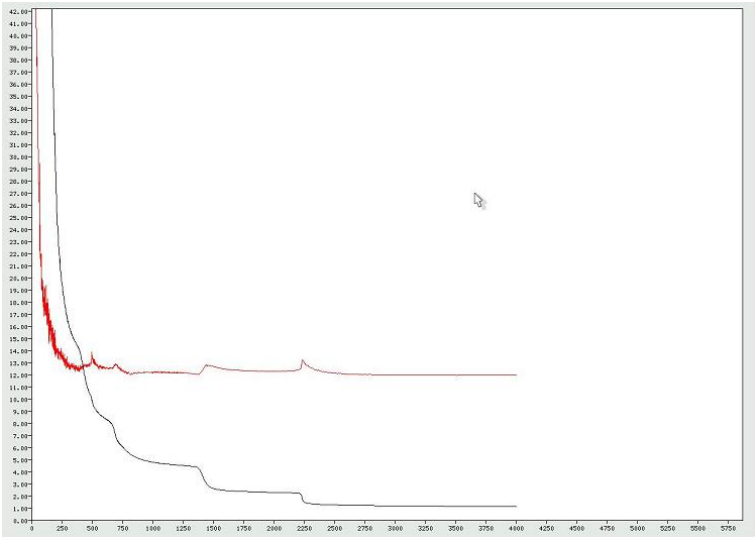


Abbildung 3.26.: Verlauf des Fehlers eines 32 $\times$ 32 feedforward Netzes mit zwei verdeckten Schichten bei Backpropagation

Netz 3

Eingabeschicht: 31 (1 $\times$ 31) Neuronen
2 verdeckte Schichten mit jeweils 20 Neuronen
Ausgabeschicht: 37 (1 $\times$ 37) Neuronen

Dieses Netz wurde mit Quickpropagation (2.3.7) trainiert.
Mit dieser Lernfunktion konnte das Lernen beschleunigt werden.
Die Lernparameter wurden wie folgt gewählt: $\eta = \mathbf{0.01}$, $\mu = \mathbf{1.3}$ und $d = \mathbf{0.0001}$.
Der Netzfehler konnte hierbei schon nach 1000 Durchläufen bis auf $\mathbf{0.0031}$ verringert werden.

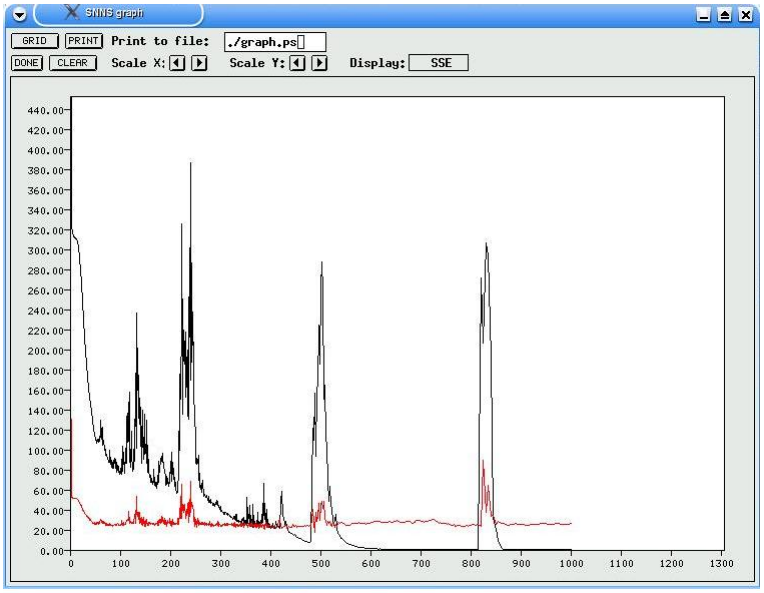


Abbildung 3.27.: Verlauf des Fehlers eines 1×31 feedforward Netzes mit zwei verdeckten Schichten bei Quickprop

Netz 4

Eingabeschicht: 31 (1 × 31) Neuronen
2 verdeckte Schichten mit jeweils 20 Neuronen
Ausgabeschicht: 37 (1 × 37) Neuronen

Hierfür habe ich das fertig trainierte Netz 1 (Standard Backpropagation) mit dem ”PruningFeedForward”-Algorithmus ”ausgedünnt” (2.5). Als Pruningmethode wurde das Magnitude Base Pruning verwendet, mit den Standard Parametern wie in Abbildung 3.29:

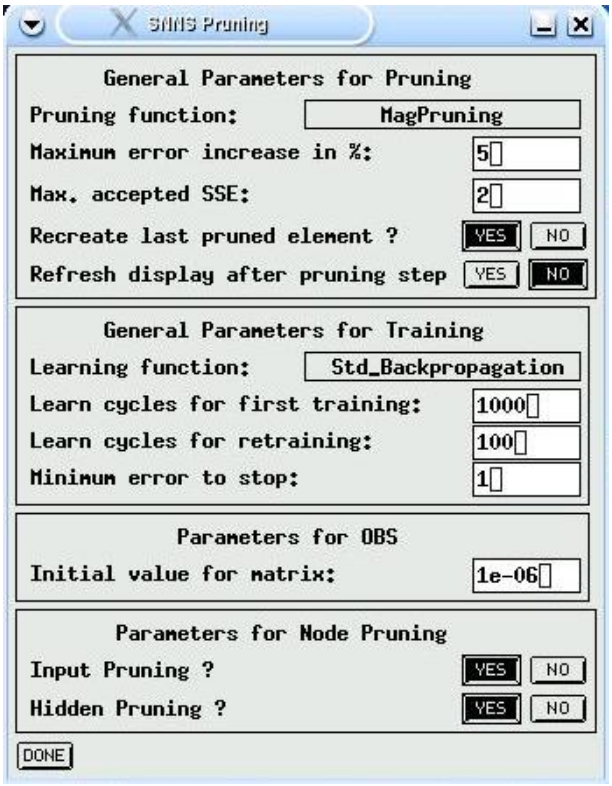


Abbildung 3.28.: Pruning Panel

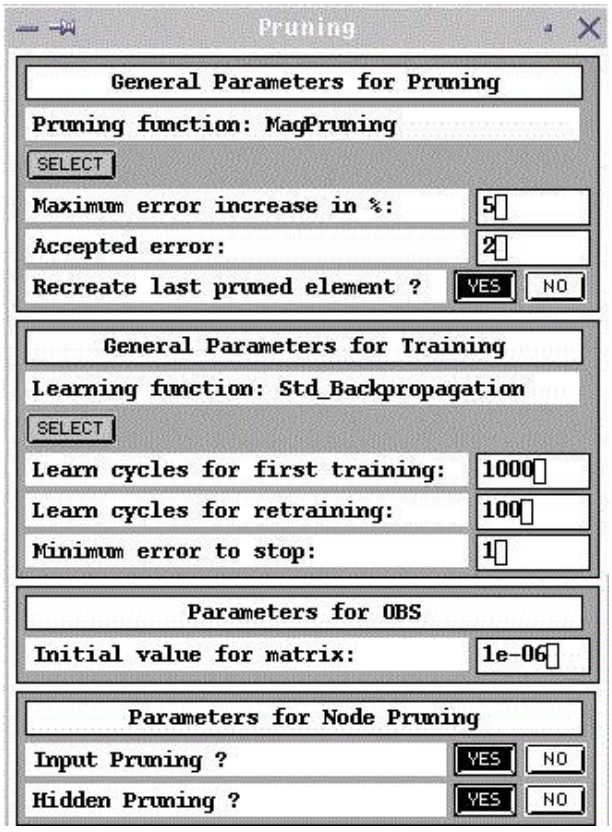


Abbildung 3.29.: General Parameters for Pruning

Ein Vergleich der beiden Netze vor und nach dem prunen:

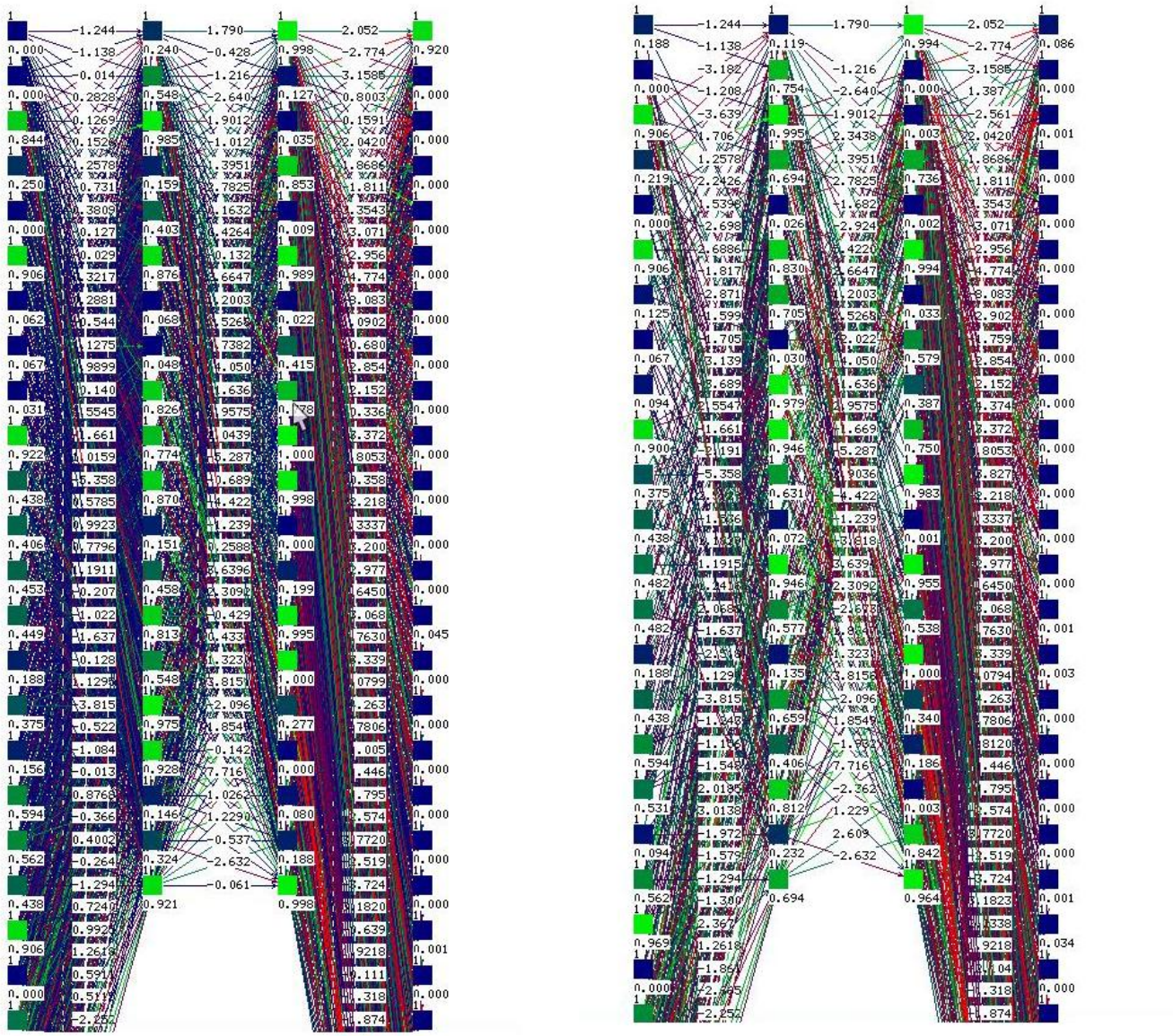


Abbildung 3.30.: Netz vor und nach dem feedforward-Pruning

Bemerkung 3.1. *Die Unterschiede in der Konvergenzgeschwindigkeit der einzelnen Lernverfahren, die vorher in der Theorie festgestellt wurden, bestätigen sich hier. Die auf dem Gradientenabstieg basierenden Verfahren konvergieren deutlich langsamer gegen ein Minimum als die auf der Quasi-Newton Methode basierenden Verfahren.*
Netze, die nur eine Schicht verdeckter Neuronen haben, können schneller trainiert werden. Wichtig ist aber auch die Generalisierungsfähigkeit, die erst bei verschiedenen Tests deutlich wird.

Beispiel für die als Trainingsdatensatz kodierte ”Muster-”Null:

1) unter Verwendung der geometrischen Merkmale:

Input pattern 0:
0.125 0 0.84375 0.21875 0 0.875 0.03125 0.05555 0.03125 0.9 0.28125 0.375 0.45117
0.47070 0.1875 0.5 0.125 0.5 0.625 0.46875 0.90625 0 0.59 0.5 0.65 0.7 0 0.5 0.68 0.31 0.7
Output pattern 0:
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

2) unter Verwendung der Null-/Eins-Matrix:

Input pattern 0:
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0
0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0
0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0
0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Output pattern 0:
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

3.5.4. Entscheidungskriterien bei Erkennung

Eigene Methode

Es wird ein Abgleich mit dem Mustertrainingssatz vorgenommen, d.h. die Daten der geometrischen Merkmale des aktuellen Zeichens werden Schritt für Schritt mit den Daten des kompletten Musterzeichensatzes verglichen.
Die jeweiligen Differenzen zwischen den einzelnen Merkmalen des Testzeichens und des Musterzeichens werden summiert und schließlich entscheidet die Software welches Zeichen erkannt wird, nämlich das Zeichen, das die geringste Summe aufweist.
In Anlehnung an die neuronalen Netze, die zur Erkennung die größte Wahrscheinlichkeit verwenden, habe ich die Zeichnung an dem größten, auftretendem Wert gespiegelt, so dass das angenommene Zeichen den höchsten Wert auf der Ordinate besitzt.

Die Summen können unter ”Resultate plotten”→”eigene Methode” graphisch betrachtet werden.

Neuronale Netze

Hier werden die geometrischen Merkmale als Eingangsvektor (mit 31 bzw. 1024 Einträgen) für die neuronalen Netze verwendet. Die Ausgabe der Netze ist wiederum ein (37-dimensionaler) Vektor mit den Wahrscheinlichkeiten für die einzelnen Zeichen in den Einträgen. Das Zeichen mit der höchsten Wahrscheinlichkeit wird erkannt.

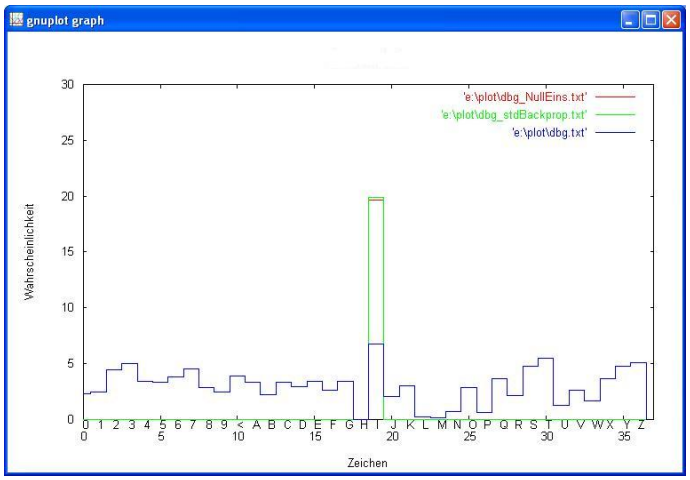


Abbildung 3.31.: Plot der Ergebnisse

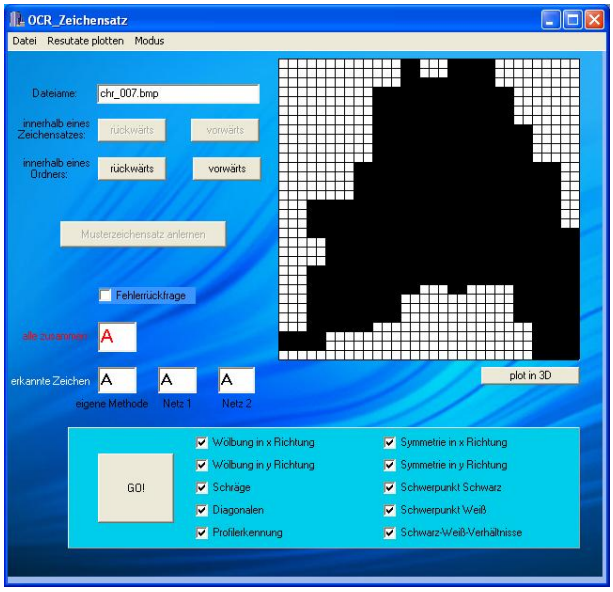
Erkennung gesamt

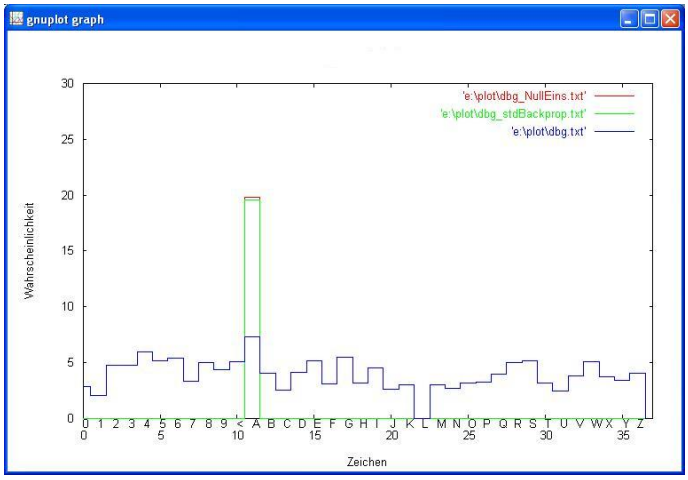
Da Netz 1 und Netz 2 mit Abstand die besten Ergebnisse liefern wird das Gesamtergebnis aus dem Resultat dieser beiden Netze mit 10-facher Gewichtung zusammen mit dem Resultat meiner eigenen Methode errechnet.

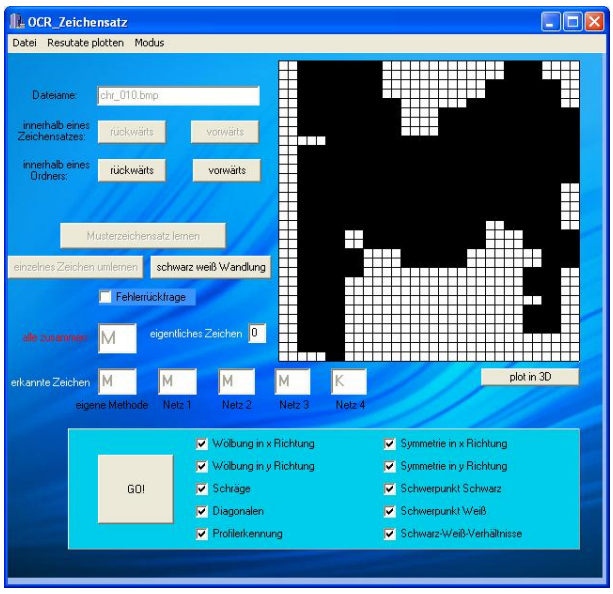
3.5.5. Ergebnisse

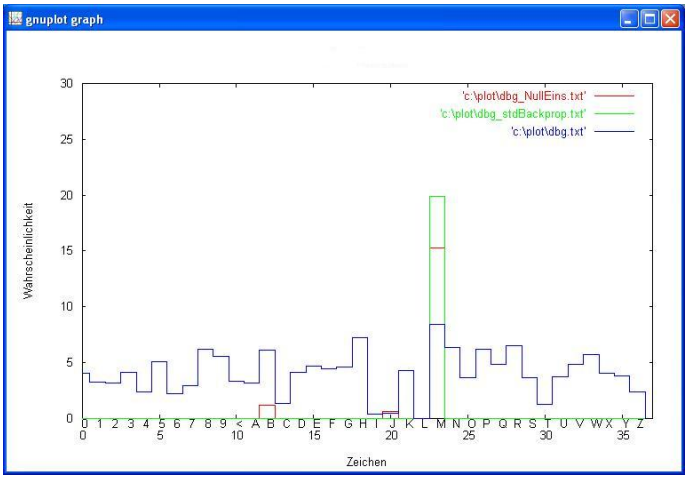
Das Ziel war die Zeichenerkennung mit neuronalen Netzen. Dafür habe ich mich mit verschiedenen neuronalen Netzen und den zugehörigen Trainigsmethoden auseinander gesetzt. Diejenigen Netze, die beim Training und in der Anwendung die besten Resultate erbracht haben, habe ich letztendlich in der von mir entwickelten Software implementiert. Man sollte die Netze mit möglichst vielen Trainingsdaten ”füttern”, denn erst nach ausreichendem Training erzielen die Netze auch annehmbare Ergebnisse. Meine Netze haben beispielsweise erst bei einem Training mit 350 Datensätzen gute Resultate geliefert.

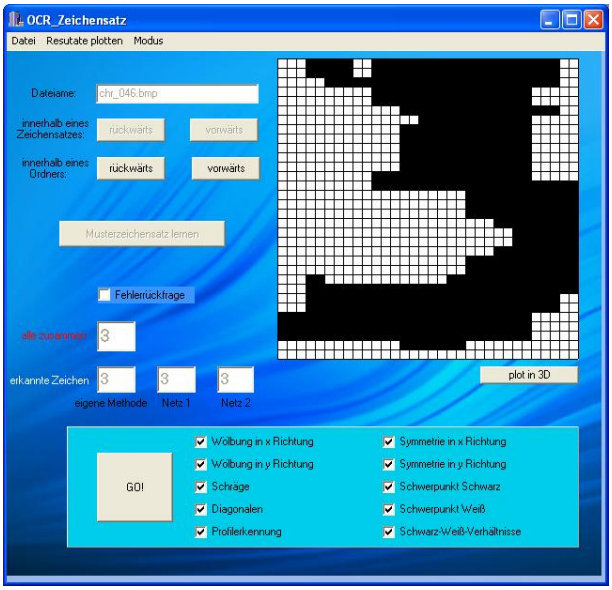
Jetzt werden auch stark verrauschte, bzw. verzerrte Bilder richtig erkannt:

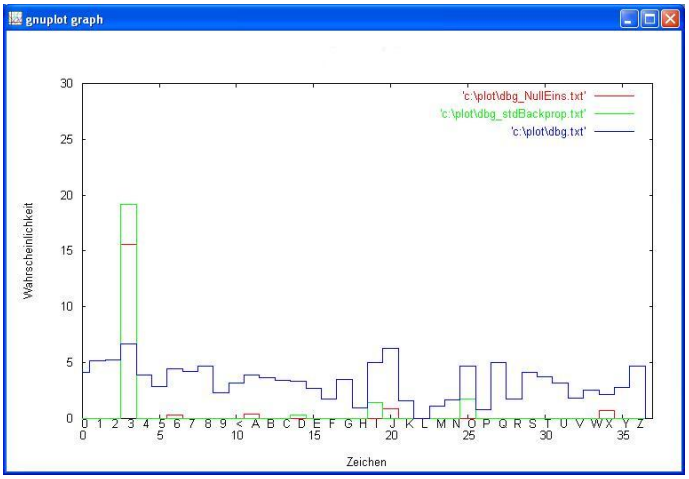




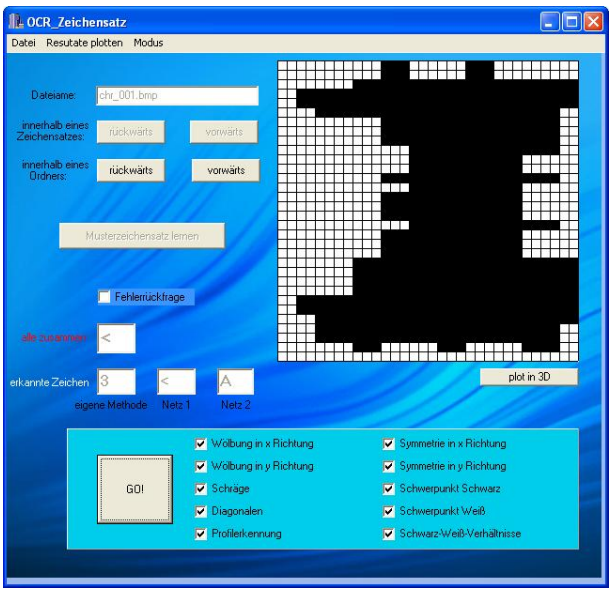


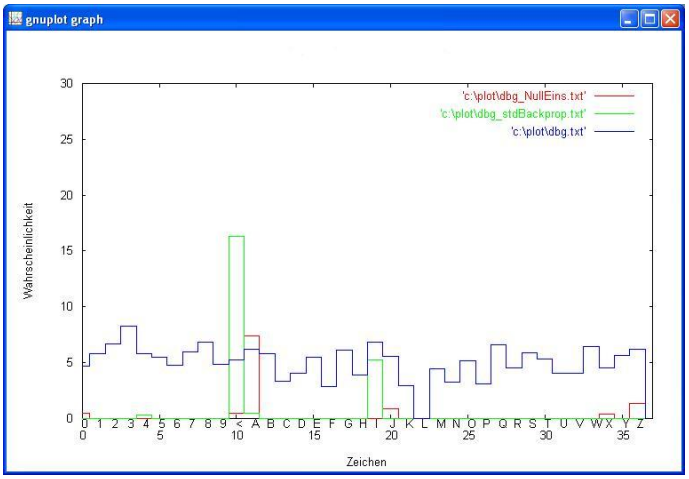


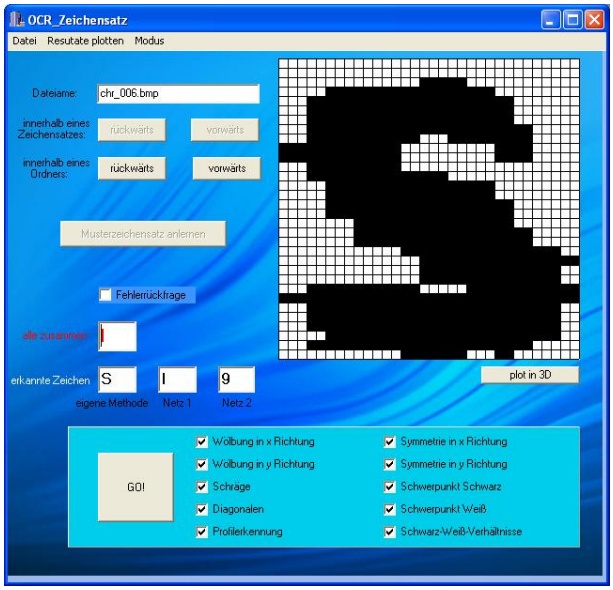


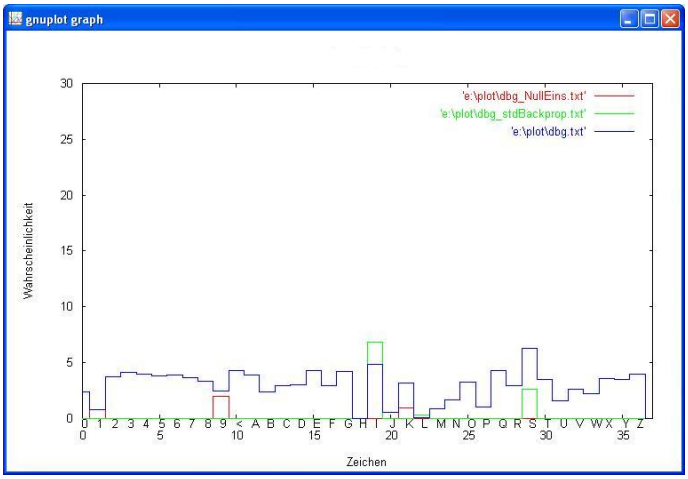


Allerdings versagen die Netze auch noch teilweise, obwohl das Zeichen für das menschliche Auge ”gut erkennbar” scheint.






OCR_Zeichensatz
Datei Resultate plotten Modus
Dateiname: chr_006.bmp
innerhalb eines Zeichensatzes:
rückwärts
vorwärts
innerhalb eines Ordners:
rückwärts
vorwärts
Musterzeichensatz anlernen
Fehlerrückfrage
alle zusammen
erkannte Zeichen
S
I
9
eigene Methode
Netz 1
Netz 2
plot in 3D
GO!
Wölbung in x Richtung
Wölbung in y Richtung
Schräge
Diagonalen
Profilerkennung
Symmetrie in x Richtung
Symmetrie in y Richtung
Schwerpunkt Schwarz
Schwerpunkt Weiß
Schwarz-Weiß-Verhältnisse


gnuplot graph
'e:\plot\dbg_NullEins.txt'
'e:\plot\dbg_stdBackprop.txt'
'e:\plot\dbg.txt'
Wahrscheinlichkeit
Zeichen
0 1 2 3 4 5 6 7 8 9 < A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

3.6. Zusammenfassung und Ausblick

Unter dem Sammelbegriff ”neuronal” sind mathematische Methoden (wie z.B. Gradienentmethode, Polynomklassifikator, etc.) zu einer breiteren Verwendung gelangt oder werden gelangen, dies alles begleitet von der inspirierenden Modellvorstellung eines sich enwickelnden ”Gehirns”, das in einem eng begrenzten Anwendungsbereich Kompetenz zeigt.

3.6.1. Euphorie und Realität

Neuronale Netze werden momentan gerne zu einem Allheilmittel hochstilisiert.
Die Wissenschaft verbreitet die Idee von einem autonom arbeitenden, denkenden System, das sich selbst konfiguriert, erweitert und entscheidet.
Dieser Stand ist allerdings kaum erreichbar, bzw. noch sehr weit entfernt.
Neuronale Netze weisen noch gravierende Mängel auf. Man kann diese grob in 3 Sektionen aufteilen:

1. Das methodologische Defizit
   - Das Design ist nach wie vor intuitiv, Entwurf und Konfigurierung des Netzes liegen in der Hand des Netzdesigners.
   - Für die Adaption des Netzes gibt es keine allgemein anwendbaren Vorschriften, auch diese ist komplett dem Anwender überlassen.
   - Es ist keine Angabe zum Umfang der Trainingsdaten vorhanden, meist ist allerings eine große Zahl erforderlich.
   - Eine Validierung ist meist nicht möglich
   - Es gibt keine Erkenntnisse zu sinnvollen und effizienten Codierungstechniken.
2. Das Anwendungsdefizit
   Es sind (außer im Bereich der Mustererkennung) noch realtiv wenige neuronale Netze im Einsatz. Weitreichende Erkenntnisse über die Arbeitsweise der Netzwerke können aber erst nach ausreichender Beobachtung von den Anwendungen erbracht werden und erst dann kann bei positiver Resonanz der Durchburch den neuronalen Netze erzielt werden.
3. Das Umgebungsdefizit
   Es gibt inzwischen eine sehr hohe Anzahl an Soft- und Hardwarlösungen für neuronale Netze, doch ist es schwierig sich einen Überblick über die vielen verschiedenen Möglichkeiten zu schaffen und eine Standardisierung ist nicht in Aussicht, obwohl sie sehr hilfreich für die Weiterentwicklung wäre.

Nichtsdestotroz haben neuronale Netze, den konventionellen Methoden gegenüber, auch gewaltige Vorteile:

- Hohe Geschwindigkeit durch massive Parallelität (Echtzeit)
- geringer Entwicklungsaufwand
- Flexibilität und schnelle Anpassung an neue Problemstellungen
- Hohe Fehlertoleranz gegen Ausfall einzelner Neuronen (besonders wichtig bei Speichern)

3.6.2. Fazit

Im Hinblick auf die rasante Entwicklung im Bereich der elektronischen Datenverarbeitung, strebt die Industrie nach immer besseren technischen Hilfsmitteln.
Ein vielversprechender Ansatz scheint die Nutzung der künstlichen Intelligenz zu sein, deren Teilgebiet die künstlichen neuronalen Netze sind.
Diese stellen besonders im Gebiet der Zeichenerkennung ein sehr mächtiges Instrument dar, so dass sie aufgrund der vielversprechenden Ergebnisse schon jetzt ein Schwerpunkt in der Forschung geworden sind.
Allerdings gibt es ein enormes Weiterentwicklungs- und Verbesserungspotzential der neuronalen Netze. Schon allein Entwicklung eines Netzes, sowie das Training derer, ist noch sehr intuitiv und keinen allgemeinen Regeln unterlegen.
Generell ist bei dem Einsatz der Netzwerke Vorsicht geboten, da die oftmals komplexen und aufwändigen Verfahren zur Erstellung künstlicher neuronaler Netze nicht immer eine Leistungssteigerung sein müssen, im Vergleich zu den traditionellen Methoden.

Nichtsdestotrotz lieferte der Ansatz, neuronale Netze für die Zeichenerkennung zu benutzen sehr vielversprechende Ergebnisse und zeichnete sich vorallem durch eine große Fehlertoleranz aus.
Daher ist der Einsatz dieser Netzwerke im Gebiet der OCR durchaus gerechtfertigt und sollte weiterhin verfolgt werden.

A. Anhang

.1. Kleines Deutsch-Englisches Fachwörterbuch

backpropagation	Rückvermittlung
backward pass	RückwärtsDurchlauf
continuous	fortlaufend
desired output	gewünschte Ausgabe
epoch	Epoche
error signal	Fehlersignal
feedback	Rückkopplung
forward pass	VorwärtsDurchlauf
gradient descent	GradientenAbstieg
hidden	verborgen
input	Eingabe
layer	Schicht
learning rate	Lernrate
neural networks	Neuronale Netze
neuron	Neuron
output	Ausgabe
overtraining	Übertrainieren
slope	Steigung
supervised learning	Beaufsichtigtes Lernen
target output vector	Zieldaten
unit	Neuron
weight	Gewicht
weight-decay term	Vergessensterm
weight update	Aktualisierung der Gewichte

[**?**, Spezifikation]

Literaturverzeichnis

[AhmTesHe] *Asymptotic Convergence of Backpropagation: Numerical Experiments*
Ahmad, Subutai
Tesauro, Gerald
He, Yu
1990

[Alt] *Nichtlineare Optimierung*
Alt, Walter
Vieweg
2002

[Anthony] *Neural network learning theoretical foundations*
Anthony, Martin
Bartlett, Peter L.
Cambridge Univ. Press
1999

[BerTsi] *Gradient convergence in gradient methods with errors*
Bertsekas, Dimitri P.
Tsitsiklis, John N.
SIAM J. OPTIM., Vol 10, No.3, S.627-642
2000

[Bose] *Neural network fundamentals with graphs, algorithms, and applications*Bose, Nirmal K.
Liang, Ping
McGraw-Hill
1996

[BrauFeu] *Praktikum neuronale Netze*
Braun, Heinrich
Feulner, Johannes
Malaka, Rainer
Springer
1996

[Braun] *Neuronale Netze Optimierung durch Lernen und Evolution*
Braun, Heinrich
Springer
1997

[Brause] *Neuronale Netze eine Einführung in die Neuroinformatik*
Brause, Rüdiger
Teubner
1991

[Brunak] *Neuronale Netze die nächste Computer-Revolution*
Brunak, Sren
Lautrup, Benny
Hanser
1993

[Fine] *Feedforward Neural Network Methodology*
Fine, Terrence L.
Springer
1999

[Grauel] *Neuronale Netze, Grundlagen und mathematische Modellierung*
Grauel, Adolf
BI-Wiss.-Verl.
1992

[HasSto] *Optimal Brain Surgeon*
Hassibi, B.
Stork, D.G.
Advances in Neural Information Processing Systems 5
1993

[Herrmann] *Evolutionäre Neuronale Netze - Theorie und Praxis*
Herrmann, S.
Diplomarbeit an der Fakultät für Mathematik und Physik
Universität Bayreuth
2004

[Hoffmann] *Kleines Handbuch neuronale Netze anwendungsorientiertes Wissen zum Lernen und Nachschlagen*
Hoffmann, Norbert
Vieweg
1993

[Hrycej] *Modular learning in neural networks a modularized approach to neural network classification*
Hrycej, Tomas

Wiley
1992

[Kinnebrock] *Neuronale Netze Grundlagen, Anwendungen, Beispiele*
Kinnebrock, Werner
Oldenbourg
1992

[Koehle] *Neuronale Netze*
Köhle, M.
Wien
Springer
1990

[Kratzer] *Neuronale Netze Grundlagen und Anwendungen*
Kratzer, Klaus Peter
Hanser
1991

[Lawrence] *Neuronale Netze Computersimulation biologischer Intelligenz*
Lawrence, Jeannette
Systhema-Verl.
1992

[MagVraAnr] *Improving the Convergence of the Backpropagation Algorithm Using Learning Rate Adaption Methods*
Magoulas, G.D.
Vrahatis, M.N.
Anroulakis, G.S.
Neural Computation 11 , S. 1769 - 1796
1999

[MangSol] *Serial and parallel Backpropagation convergence via nonmontone perturbed minimization*
Mangasarian O.L.
Solodov, M.V.
Optimization Methods and Software Vol 4 , S. 103 - 116
1994

[Mazzetti] *Praktische Einführung in neuronale Netze*
Mazzetti, Alessandro
Heise
1992

[Mueller] *Spärlich verbundene neuronale Netze und ihre Anwendung*
Müller, Klaus-Robert
Oldenbourg
1994

[Nauck] *Neuronale Netze und Fuzzy-Systeme Grundlagen des Konnektionismus, neuronaler Fuzzy-Systeme und der Kopplung mit wissensbasierten Methoden*
Nauck, Detlef
Klawonn, Frank
Kruse, Rudolf
Vieweg
1994

[Neal] *Bayesian Learning for Neuronal Networks*
Neal, R.M.
Springer Verlag
1996

[Oberhofer] *Wie künstliche neuronale Netze lernen*
Oberhofer, Walter
Universität Regensburg

[Patterson] *Künstliche neuronale Netze*
Patterson, Dan W.
Prentice Hall
1997

[PlaMagVra] *Optimization strategies and backpropagation neural networks*
Plagianakos, V.P.
Magoulas, G.D.
Vrahatis, M.N.

[Riedmiller] *Untersuchung zu Konvergenz und Generalisierungsfähigkeit überwachter Lernverfahren mit dem SNNS*
Riedmiller, M.
In: Zell, A.*Workshop SNNS 93* Seiten 107-113
Universität Stuttgart, Fakultät für Informatik
Bericht Nr 10/93

[Riedmiller] *RProp - Description and Implementation Details*
Riedmiller, M.
Technischer Bericht 01/94
Universität Karlsruhe
1994

[Rigoll] *Neuronale Netze eine Einführung für Ingenieure, Informatiker und Naturwissenschaftler*
Rigoll, Gerhard
Expert-Verlag
1994

[RiMaSchu] *Neuronale Netze, eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke*
Ritter, Helge

Martinetz, Thomas
Schulten, Klaus
Addison-Wesley
1991

[Scherer] *Neuronale Netze Grundlagen und Anwendungen*
Scherer, Andreas
Vieweg
1997

[Seraphin] *Neuronale Netze und Fuzzy-Logik Verknüpfung der Verfahren, Anwendungen, Vor- und Nachteile, Simulationsprogramm*
Seraphin, Marco
Franzis-Verlag
1994

[Tan] *Statistical mechanics of pattern recognition in a neural network*
Tan, Zuguo
Shaker
1997

[VraMagPlag] *Globally Convergent Modification of the Quickprop Method*
Vrahatis, Michael N.
Magoulas, George D.
Plagianakos, Vassilis P.
1999

[VraMagPlag] *Convergence Analysis fo the Quickprop Method*
Vrahatis, M.N.
Magoulas, G.D.
Plagianakos, V.P.
Proceedings of the INNS-IEEE international joint conference on neural networks
1999

[Zeidenberg] *Neural network models in artificial intelligence*
Zeidenberg, Matthew
Ellis Horwood
1990

[Zell] *Simulation neuronaler Netze*
Zell, Andreas
Addison-Wesley
1994

[ZimHerFinn] *Neuron pruning and merging methods for use in conjunction with weight elimination*
Zimmermann, H.G.

Hergert, F.
Finnhoff, W.
Siemens AG, Corporate Research and Development
1992

[1] *Universität Tübingen, Java Neural Network Simulator*
http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/manual/JavaNNS
manual-8.html

[2] *Vorlesung Fuzzy-Regelung und Neuronale Netze, Prof. Dr.-Ing. Chr. Schmid*
http://www.esr.ruhr-uni-bochum.de/mitarbeiter/cs/FuzzNN/FuzzNNVorlesung_Te

[3] *Institut für Informatik, Westfälische Wilhelms Universität Münster, Prof. Dr. Wolfram-M. Lippe*
http://www.math.uni-muenster.de/SoftComputing/lehre/material/wwwnnscript/

[4] *Stuttgart Neural Network Simulator*
http://www-ra.informatik.uni-tuebingen.de/SNNS/

[5] *Universität Würzburg, "Methoden der Forschung: Neuronale Netze", Prof. Dr. Hans-Peter Krüger*
http://www.psychologie.uni-wuerzburg.de/methoden/lehre/skripten/HS_Meth_Forsch

[6] *Hochschule für Technik, Rapperswil, Prof.Dr. J.M. Joller*
http://i.hsr.ch/Content/Gruppen/Doz/jjoller/ISeminare/SeminarSS2002/AISeminar/

[7] *FH-Köln*
http://www.fbi.fh-koeln.de/institut/personen/galliat/material/ws05/5u6_NeuronaleN

[8] *FH-Köln, Dr. Thomas Mandl*
http://www.uni-hildesheim.de/ mandl/Lehre/hsir/iirv_03ml.pdf

[9] *Wirtschaftsuniversität Wien*
http://wwwai.wu-wien.ac.at/Publikationen/Frisch/startalgorithmen/startalgorithmen

[10] *Neural Computing Group, Department of Computer Science and Engineering Faculty of Electrical Engineering Czech Technical University in Prague*
http://cs.felk.cvut.cz/ neurony/neocog/en/index.html

[11] *Universität Dortmund Dr. Lars Hildebrand*
http://lrb.cs.uni-dortmund.de/%7Ehildebra/GACI1-gr.html

[12] *Fachhochschule Bern, Hochschule für Informatik und Technik*
http://www.hta-bi.bfh.ch/ fornp1/neural/presentation.pdf

[13] *Universität Kassel, Hompage of FGNN, Reserarch Group Neural Networks*
http://www.neuro.informatik.uni-kassel.de/ werner/NN/Nn01.pdf

[14] *TU Darmstadt*
http://www.st.informatik.tu-darmstadt.de:8080/felzer/nn_sem.pdf

[15] *Universität Osnabrück, Institut für Informatik, Barbara Hammer*
$http://www-lehre.inf.uos.de/nn/nn.pdf$

[16] *Technisch Universität Clausthal, Institut für Informatik, Dr. Ing. Matthias Reuter*
$http://www.informatik.tu-clausthal.de/reuter/nn0405UEB1.pdf$

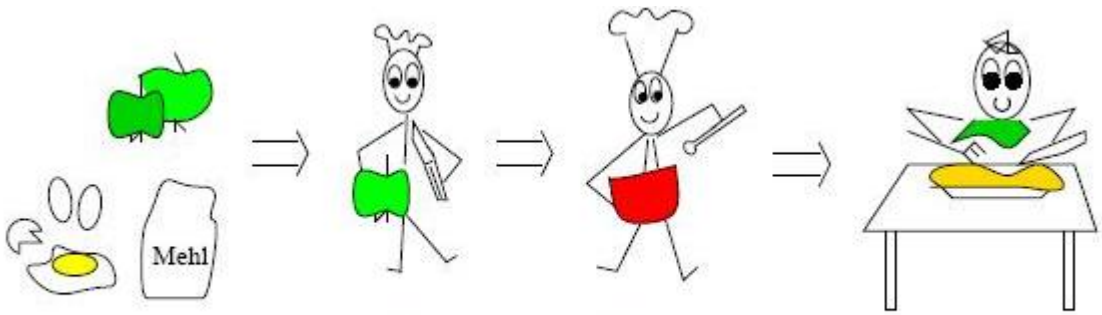
[17] *Universität Stuttgart, SNNS* $http://www.informatik.uni-stuttgart.de/ipvs/bv/projekte/snns/snns.html$

[18] *Universität Magdeburg, "Handschriftenerkennung mit dem Computer"* $http://graf350.urz.uni-magdeburg.de/dornheim/Germand/Projects/HSK/HSK-Documentation.pdf$

[19] *Optimierung*
Gerdts, Matthias
$http://www.math.uni-hamburg.de/home/gerts$
WS 2004/2005

[20] *Institut for Information Systems and Computer Media*
$http://www.iicm.edu/greif/node10.html$

[21] *Bayesian Learning*
$http://www.faqa.org/ai-faq/neural-nets/part3/section-7.html$



"Feed-forward Netz"

ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Bayreuth, den 10. Mai 2005 .....

Lisa Sammer