Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik VI

Dominik Scherer

GPU-beschleunigte Objekterkennung mit neuronalen Konvolutionsnetzen

8. Juni 2009

Diplomarbeit

Erstgutachter: Prof. Dr. Sven Behnke

Zweitgutachter: Prof. Dr. Joachim K. Anlauf

Abstract

The recognition of visual objects within a natural image is a comparetively easy task for the human brain. However, despite the increasing power of modern processors and recent advances in machine vision this is still a huge challenge for computer programs. The recognition of handwritten digits is considered an easy subset of this problem and multi-layered convolutional neural networks like LeNet-5 are considered the state of the art solution for this problem. Up until very recently systems of this kind could not be adapted for the recognition of complex objects in realistic environments due to the computational power which is required to process such high dimensional data.

Due to the availability of Nvidias CUDA interface it is now possible to harness the tremendous computational power of modern graphics cards for scientific calculations. The parallel hardware of such GPUs has been exploited to accelerate a huge convolutional neural network for this diploma thesis. With this parallel implementation a speed gain ranging from 95× - 115× compared to a serial CPU program was achieved by optimizing the memory accesses and using the full capacity of the multiprocessors.

The convolutional neural network has been applied to several datasets to evaluate the recognition rate. For the test set of the MNIST database of handwritten digits a recognition error of 1.38 % was achieved. On the NORB dataset 11.07 % of the objects were classified incorrectly. The results on the considerably harder dataset of the annual *PASCAL Visual Object Classes Challenge* have shown that the topology of the network as well as the lerning algorithm require further improvements.

The insights gained by the accelerated parallel application developed in this diploma thesis can be considered a first step towards further research on convolutional neural networks for large natural images.

Zusammenfassung

Die für das menschliche Gehirn vergleichsweise einfache Aufgabe, Objekte in natürlichen Bildern zu erkennen, stellt trotz stetig wachsender Rechenleistung moderner Prozessoren, sowie beachtlicher Fortschritte im Bereich des maschinellen Sehens eine große Herausforderung für Computer dar. Bei der Erkennung von handschriftlichen Ziffern als einer einfachen Teilmenge dieses Problems zählen mehrschichtige neuronale Konvolutionsnetze, wie LeNet-5, zu den erfolgreichsten Verfahren. Auf die Erkennung von komplexeren Objekten in realistischen Szenen waren diese Systeme aufgrund der erforderlichen Rechenleistung zur Verarbeitung großer Eingabedaten bisher nicht übertragbar.

Mit der leicht zugänglichen CUDA-Schnittstelle von Nvidia lässt sich die enorme Rechenleistung moderner Grafikkarten nun auch für eine Vielzahl von rechenintensiven Anwendungen nutzen. Im Rahmen dieser Diplomarbeit wurde zur Beschleunigung eines großen neuronalen Konvolutionsnetzes daher diese parallele Hardware eingesetzt.

Mittels der parallelen Implementierung ist es gelungen, das Trainieren dieses Netzes gegenüber der Laufzeit eines seriellen Programms um den Faktor 95 bis 115 zu beschleunigen. Dazu war es notwendig, sowohl die Speicherzugriffe zu optimieren als auch die Multiprozessoren effizient auszulasten.

Es wurde die Erkennungsleistung eines trainierten Konvolutionsnetzes auf verschiedenen Datensätzen untersucht. Auf der Testmenge der MNIST-Datenbank handschriftlicher Ziffern konnte die Fehlerquote auf 1,38 % reduziert werden. Die Objekte des NORB-Datensatzes wurden nur zu 11,07 % falsch klassifiziert. An den Ergebnissen auf dem schwierigen Datensatz der jährlich ausgeschriebenen *PASCAL Visual Object Classes Challenge* hat sich gezeigt, dass sowohl bei der Netzstruktur, als auch beim Lernverfahren noch Optimierungsbedarf besteht.

Insgesamt stellen die in dieser Diplomarbeit gewonnenen Erkenntnisse zur parallelen Beschleunigung einen wichtigen ersten Schritt zur Erforschung neuronaler Konvolutionsnetze auf großen natürlichen Bildern dar.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Problembeschreibung . . . 1
1.2 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
2.1 Neuronale Netze . . . 3
2.1.1 Das Perzeptron . . . 4
2.1.2 Das Multilayer-Perzeptron . . . 5
2.1.3 Lernverfahren Backpropagation . . . 7
2.1.4 Erweiterungen von Backpropagation . . . 11
2.2 Konvolutionsnetze . . . 13
2.2.1 Motivation . . . 14
2.2.2 Funktionsweise am Beispiel von LeNet-5 . . . 15
2.2.3 Eigenschaften von Konvolutionsnetzen . . . 17
2.3 Datensätze zur Objekterkennung . . . 18
2.3.1 MNIST-Datensatz handgeschriebener Ziffern . . . 19
2.3.2 NORB-Datensatz . . . 20
2.3.3 PASCAL VOC Challenge 2008 . . . 21
2.3.4 Andere Datensätze . . . 24
2.4 Parallele Hardware . . . 27
2.4.1 Simulation neuronaler Netze auf paralleler Hardware . . . 27
2.4.2 Grafikkarten zur Beschleunigung von Berechnungen . . . 28
2.4.3 Das CUDA Programmiermodell . . . 29
2.4.4 Die CUDA Hardware-Architektur . . . 31
2.4.5 Einschränkungen von CUDA und Optimierungsmöglichkeiten . . . 34

3 Verwandte Arbeiten **37**
3.1 Verfahren zur Objekterkennung in der PASCAL 2008 Visual Object Challenge 37
3.2 Anwendungen von Konvolutionsnetzen . . . 39
3.3 Beschleunigung mit Grafikkartenhardware . . . 43
3.4 Zusammenfassung und Einordnung . . . 44

4 Entwurf eines Konvolutionsnetzes **45**
4.1 Netztopologie . . . 45
4.1.1 Hierarchische Konvolutionsschichten . . . 46
4.1.2 Lokale Filter . . . 46
4.1.3 Vollverknüpfte Schichten . . . 48
4.2 Kapazität des Netzes . . . 48
4.3 Lernverfahren . . . 49
4.3.1 Aktivierungsfunktion . . . 49
4.3.2 Gewichtsinitialisierungen . . . 49
4.3.3 Gradientenabstieg . . . 51
4.3.4 Arbeitsmenge . . . 52
4.3.5 Gewichtetes Selektionsschema . . . 53

4.4 Ein- und Ausgabe . . . 54
4.4.1 Eingabe . . . 54
4.4.2 Ausgabe . . . 56

5 Implementierung **57**
5.1 Framework . . . 57
5.1.1 CPU-Version . . . 57
5.1.2 Grafische Benutzeroberfläche . . . 58
5.2 Parallelisierungshierarchie . . . 59
5.2.1 Grobkörnige Parallelisierung . . . 60
5.2.2 Feinkörnige Parallelisierung . . . 61
5.3 Vorwärtspropagierung . . . 62
5.3.1 Naiver Ansatz . . . 62
5.3.2 Konfiguration des Forward-Kernels . . . 64
5.3.3 Konvolutionsalgorithmus . . . 64
5.3.4 Anmerkungen zum Forward-Kernel . . . 67
5.4 Strukturierung des Speichers . . . 68
5.4.1 Aktivitäten und Fehlersignale . . . 69
5.4.2 Temporärer Zwischenspeicher zur Berechnung der Fehlersignale . . . 70
5.4.3 Konvolutionsfilter . . . 71
5.5 Rückwärtspropagierung . . . 72
5.5.1 Akkumulation der Fehlersignale . . . 72
5.5.2 Konfiguration des Backprop-Kernels . . . 73
5.5.3 Backpropagation-Algorithmus . . . 74
5.5.4 Anmerkungen zum Backprop-Kernel . . . 77
5.6 Gewichtsanpassungen . . . 78
5.7 Vollverknüpfte Schichten . . . 78
5.8 Sonstige Kernel . . . 79
5.9 Anmerkungen zur parallelen Implementierung . . . 80
5.9.1 Behandlung von Einschränkungen und Spezialfällen . . . 80
5.9.2 Schwierigkeiten bei der Implementierung . . . 82

6 Experimente und Ergebnisse **83**
6.1 Erweiterungen des Lernverfahrens . . . 83
6.1.1 Arbeitsmenge . . . 83
6.1.2 Gewichtsinitialisierung und -anpassung . . . 85
6.2 Experimente mit unterschiedlichen Datensätzen . . . 89
6.2.1 Lernen bildartiger Abbildungen . . . 89
6.2.2 Erkennung handschriftlicher Ziffern . . . 91
6.2.3 Objektklassifizierung auf dem NORB-Datensatz . . . 93
6.2.4 Objektklassifizierung auf dem PASCAL-Datensatz . . . 97
6.3 Laufzeitanalyse . . . 100
6.3.1 Kernellaufzeiten . . . 100
6.3.2 Gesamtlaufzeit . . . 102

7 Zusammenfassung **105**
7.1 Diskussion der Ergebnisse . . . 105
7.2 Identifikation von Problemquellen . . . 106
7.3 Offene Fragen und Ausblick . . . 107

1 Einleitung

Das Auge ist für den Menschen das wichtigste Sinnesorgan. Im Laufe der ersten Lebensjahre lernen wir, Gegenstände anhand ihrer Form und Farbe zu unterscheiden und Personen anhand von Gesichtszügen, Haut- und Haarfarbe zu identifizieren. Jeder alltägliche Handgriff wird durch die visuelle Wahrnehmung koordiniert und gesteuert. Innerhalb von Sekundenbruchteilen ist es uns möglich, uns neue Personen oder Gegenstände einzuprägen und diese auch Wochen später wiederzuerkennen.

Trotz der stetig wachsenden Leistung moderner Prozessoren und der großen Fortschritte im Bereich des maschinellen Sehens in den letzten Jahren stellt die Erkennung von Objekten in ihrer natürlichen Umgebung für Computer nach wie vor eine enorme Herausforderung dar. Heutige Computersysteme zur visuellen Wahrnehmung beschränken sich daher häufig auf sehr einfache Spezialfälle. Briefe werden zum Beispiel schon seit vielen Jahren mit Hilfe der automatischen Erkennung handgeschriebener Postleitzahlen sortiert. In modernen Digitalkameras sind Verfahren eingebaut, die zur Fokussierung des Bildes automatisch Gesichter lokalisieren.

Doch auch eine Vielzahl weiterer Anwendungsgebiete könnte von einer robusten Objekterkennung profitieren, die innerhalb eines breiten Aufgabenspektrums einsetzbar ist. Ein solches visuelles Wahrnehmungssystem könnte insbesondere in der Robotik dazu beitragen, dass sich Roboter selbstständig in der dem Menschen adäquaten Umgebung zurechtfinden.

1.1 Problembeschreibung

Ziel dieser Arbeit ist es, ein lernfähiges System zu entwickeln, das unterschiedliche Objekte in natürlichen Bildern zu erkennen vermag. Vorgabe und Ausgangspunkt sind dazu Bilder, in denen diese Objekte von Menschen markiert wurden. Ohne weiteres Vorwissen des Programmierers soll das System die Unterscheidung verschiedener Objekte aus diesen Beispielen lernen.

Ein weit verbreitetes biologisch motivertes Modell für allgemeine Klassifizierungsaufgaben sind neuronale Netze. Auf einfachen Teilproblemen wie etwa der Erkennung von handschriftlichen Ziffern zählen dabei sogenannte Konvolutionsnetze zu den erfolgreichsten Verfahren. Sie sind inspiriert durch die Art und Weise der Verarbeitung von visuellen Eindrücken im menschlichen Gehirn.

Bisher wurden diese Netze in der Regel jedoch nur auf kleine Bilder wie z.B. einzelne handgeschriebene Ziffern angewandt. Bei komplexeren Daten war das Training solcher Netze bisher nicht praktikabel, da selbst die Rechenleistung moderner Computer zur Verarbeitung von hochauflösenden Fotos nicht ausreicht. Ob derartige Netze sich also zur Erkennung von komplexen Objekten in natürlichen Szenen eignen, ist noch nicht geklärt.

Moderne Grafikkarten verfügen über mehrere Dutzend Rechenwerke, die erst seit kurzem durch die Verfügbarkeit von Programmierschnittstellen wie CUDA nicht nur zur Berechnung von 3D-Grafik, sondern z.B. auch für wissenschaftliche Berechnungen nutzbar sind. Andere Anwendungen haben gezeigt, dass mit einer effizienten Auslastung dieser parallelen Multiprozessoren eine Beschleunigung um den Faktor 50 bis 200 möglich ist.

Diese Hardware soll im Rahmen der vorliegenden Diplomarbeit dazu genutzt werden, um die Berechnung von Konvolutionsnetzen auf größeren Datenmengen zu beschleunigen. Entscheidend ist dabei, eine Netzstruktur zu finden, die verschiedene Parallelisierungsmöglichkeiten ausnutzt und Speicherzugriffe optimiert. Wenn dies gelingt, lässt sich die Berechnungszeit für das Lernverfahren eines solchen neuronalen Netzes von einem halben Jahr auf wenige Tage reduzieren.

Das in dieser Diplomarbeit implementierte Netz soll anhand verschiedener Datensätze mit den Ergebnissen anderer Verfahren verglichen werden. Dabei steht jedoch nicht die Erkennungsleistung im Vordergrund, sondern zunächst vor allem die Frage, ob es gelungen ist, das Netz gegenüber einer seriellen Implementierung zu beschleunigen. Denn nur ein erheblicher Geschwindigkeitsvorteil rechtfertigt den Mehraufwand einer parallelen Implementierung.

1.2 Aufbau der Arbeit

Nach einer kurzen Einführung in das Problem der Objekterkennung in natürlichen Bildern folgt in Kapitel 2 ein Überblick über verschiedene für das Verständnis dieser Arbeit notwendige Grundlagen. Zunächst wird in Abschnitt 2.1 die Funktionsweise der biologisch motivierten neuronalen Netze erklärt und die Probleme beim Training derselben werden erläutert. Danach wird in Abschnitt 2.2 auf die spezielleren Konvolutionsnetze eingegangen. In Abschnitt 2.3 werden unterschiedliche Datensätze zur Objekterkennung vorgestellt, anhand derer die Erkennungsleistung verschiedener Klassifikatoren verglichen werden kann. Abschnitt 2.4 schließt die Grundlagen mit einer Erläuterung der Grafikkartenarchitektur CUDA ab.

Aus der Fülle der Arbeiten, die sich mit Objekterkennung in natürlichen Bildern beschäftigen, werden ausgewählte Veröffentlichungen in Kapitel 3 vorgestellt, um darzustellen, wie sich diese Diplomarbeit in den wissenschaftlichen Kontext einordnen lässt. In diesem Rahmen werden auch relevante Arbeiten beleuchtet, in denen Berechnungen durch parallele GPU-Hardware beschleunigt werden konnten.

Kapitel 4 beschreibt die Struktur des in dieser Arbeit entworfenen neuronalen Konvolutionsnetzes, analysiert dessen Generalisierungsfähigkeit und erläutert, welche Lernverfahren eingesetzt wurden, um das Netz zu trainieren.

Interessante Aspekte der parallelen Implementierung dieser Netzwerkarchitektur werden in Kapitel 5 herausgegriffen und näher erläutert. Dabei soll dieses Kapitel nicht den Charakter einer Programmdokumentation haben, sondern es wird hier herausgearbeitet, welche Hardwaremöglichkeiten zur Beschleunigung ausgereizt wurden.

Unterschiedliche mit diesem Netz durchgeführte Experimente werden in Kapitel 6 erläutert und deren Ergebnisse beschrieben. Des Weiteren werden die Auswirkungen von verschiedenen Netzparametern und von unterschiedlichen Modifikationen des Lernverfahrens näher analysiert. Insbesondere wird in diesem Kapitel ein Vergleich der Laufzeit zwischen der parallelen GPU-Implementierung und einer äquivalenten seriellen CPU-Implementierung durchgeführt.

Im abschließenden Kapitel 7 werden die Ergebnisse bewertet und Möglichkeiten aufgezeigt, das hier implementierte Netz weiterzuentwickeln.

2 Grundlagen

In diesem Kapitel wird in Abschnitt 2.1 zunächst die grundlegende Funktionsweise von einfachen neuronalen Strukturen wie Multilayer Perzeptrons erklärt. Mit Backpropagation wird in diesem Zusammenhang ein typisches Lernverfahren zum Trainieren solcher neuronalen Netze eingeführt. In Abschnitt 2.2 wird danach auf eine speziellere neuronale Architektur eingegangen, nämlich auf die in dieser Arbeit eingesetzten Konvolutionsnetze.

Außerdem werden in Abschnitt 2.3 verschiedene Bilddatensätze vorgestellt, die zum Testen von Verfahren zur Objekterkennung geeignet sind. Insbesondere werden auch die Aufgabenstellungen der PASCAL VOC Challenge behandelt. Abschnitt 2.4 erläutert zum Abschluss die zur Beschleunigung verwendete Hardware – Grafikkarten auf Basis von Nvidias CUDA-Architektur – und die damit verbundenen Einschränkungen.

2.1 Neuronale Netze

Unter dem Begriff der künstlichen neuronalen Netze fasst man informationsverarbeitende Systeme zusammen, die von Struktur und Architektur des Gehirns und des Nervensystems von Säugetieren inspiriert sind. Grundelemente sind dabei relativ einfach arbeitende Neuronen, die mehrere Eingabesignale akkumulieren und ab einem bestimmten Schwellenwert ein Ausgabesignal senden. Von diesen informationsverarbeitenden Neuronen sind viele zu einem komplexen Netz miteinander verknüpft.

Zentrale Idee ist es, die Funktionsweise der Neuronen und damit auch die Funktionsweise des gesamten Netzes iterativ aus Trainingsbeispielen zu erlernen. Der Vorteil eines solchen sogenannten *überwachten Lernverfahrens* im Gegensatz zu anderen Methoden der künstlichen Intelligenz (KI) ist, dass kaum Vorwissen über das zu lösende Problem benötigt wird. Ein Paradebeispiel der klassischen KI ist der Schachcomputer *Deep Blue* [CHH02], der in der Lage ist, die Großmeister dieser Disziplin zu schlagen. Dennoch wird seine Leistung maßgeblich durch das von Menschen einprogrammierte Wissen über Regelwerk, gute Eröffnungszüge, Positionsbewertungen usw. bestimmt.

Gegenüber einem solchen statischen Programm, das zwar schnell rechnen kann, aber nur eine konkrete Aufgabe erfüllt, verfügt ein neuronales Netz über die Fähigkeit, unterschiedliche Problemklassen zu lernen. Zudem kann es generalisieren, nämlich von der Lösung einer bestimmten Probleminstanz auf andere, ähnlich geartete Instanzen schließen. Die Gesetzmäßigkeiten der zu erlernenden Abbildung, also die zugrunde liegende Funktion muss dafür nicht bekannt sein.

Ein Beispiel hierfür wäre ein System zur Vorhersage der Windstärke, wie es für Betreiber von Windkraftanlagen von Interesse ist: Die aktuelle Windstärke hängt sicherlich von einer Vielzahl von Faktoren ab, beispielsweise von der Temperatur, vom Luftdruck, von der Luftfeuchtigkeit, der Tages- und Jahreszeit, sowie der Windstärkehistorie der letzten Stunden. Anhand dieser Parameter, der *Inputs*, ist ein erfahrener Experte in der Lage, solide Vorhersagen der Windstärke, also des *Outputs*, zu treffen. Es handelt sich dabei aber um eine komplizierte nichtlineare Funktion, deren geschlossene Formel sich nicht angeben lässt und deren Eingabedimension nicht vollständig ermittelt werden kann.

Neuronale Netze sind also gerade für Anwendungen interessant, bei denen nur wenig explizites systematisches Wissen über das zu lösende Problem vorliegt und bei denen die Eingaben unvollständig oder verrauscht sind. Unter anderem kommen sie daher in folgenden Bereichen zum Einsatz:

- Regelungstechnik (z.B. zur Temperaturregelung oder in der Fahrzeugtechnik)
- Zeitreihenanalyse (z.B. zur Wettervorhersage oder Prognose des Verkehrsaufkommens)
- Mustererkennung und Bildverarbeitung (z.B. zur Texterkennung oder Gesichtsanalyse)
- Agentensteuerung (z.B. in der Robotik oder in Simulationen)

Im Laufe der letzten Jahrzehnte wurden verschiedene Typen von künstlichen neuronalen Netzen entwickelt, die sich vor allem durch unterschiedliche Netztopologien, Neuronenarten und Verbindungsarten voneinander abgrenzen. Zu den bekanntesten zählen Multilayer-Perzeptrons, Radiale-Basisfunktionen-Netze, Self-Organizing-Maps, Hopfield-Netze, rückgekoppelte Netze und das Neocognitron (siehe [Zel94]). Die Wahl einer geeigneten Netzarchitektur hängt von der zu lösenden Aufgabe ab, und auch die Dimensionierung eines Netzes kann zum Problem werden. Zu große Netze mit zu viel Speicherkapazität neigen dazu, die Trainingsbeispiele lediglich „auswendig" zu lernen (engl. *overfitting*). Wenn das trainierte Netz in solch einem Fall nicht mehr über die zugrunde liegende Abbildungsfunktion abstrahiert führt dies dazu, dass sich die Ergebnisse nicht auf neue Daten übertragen lassen.

Spricht man von künstlichen neuronalen Netzen, so sind damit oft Multilayer-Perzeptrons gemeint, die auch in der Praxis am häufigsten zum Einsatz kommen. Dieses von Frank Rosenblatt als Perzeptron [Ros58] entwickelte und seitdem erweiterte Modell und das häufig zu dessen Training eingesetzte Lernverfahren *Backpropagation of Error* [RHW86] werden in den folgenden Kapiteln beispielhaft erläutert.

2.1.1 Das Perzeptron

Das erstmals 1958 von Frank Rosenblatt beschriebene Perzeptron [Ros58] umfasst inzwischen eine ganze Familie neuronaler Modelle. Das einstufige Perzeptron (siehe Abbildung 2.1) besteht aus einer Schicht von n Eingabezellen, die über gewichtete Verbindungen mit einem einzelnen Ausgabeneuron verbunden sind. Die Eingaben o_i dieser Neuronen werden mit w_{ij} gewichtet und zur Netzeingabe net_j aufsummiert:

$$net_j = \sum_{i=1}^{n} o_i w_{ij} \tag{2.1}$$

Wenn die Netzeingabe einen gewissen Schwellenwert θ_j überschreitet, ist das Neuron aktiv, ansonsten ist es inaktiv. Für die Aktivität a_j des Neurons gilt also

$$a_j = \begin{cases} 1, & \text{falls } net_j \geq \theta_j \\ 0, & \text{sonst.} \end{cases} \tag{2.2}$$

Der Schwellenwert θ_j lässt sich durch ein einzelnes zusätzliches Eingabeneuron mit fester Aktivität $o_0 = 1$ und einem Gewicht w_{0j}, dem sogenannten *Bias*, repräsentieren. Das Perzeptron stellt somit einen simplen binären Klassifikator dar, dessen Gewichtsvektor $\mathbf{w}$ eine separierende Hyperebene beschreibt, die den Raum der Eingabesignale $\mathbf{x} \in \mathbb{R}$ in die zwei Klassen $\mathbf{x} \cdot \mathbf{w} \geq 0$ und $\mathbf{x} \cdot \mathbf{w} < 0$ teilt.

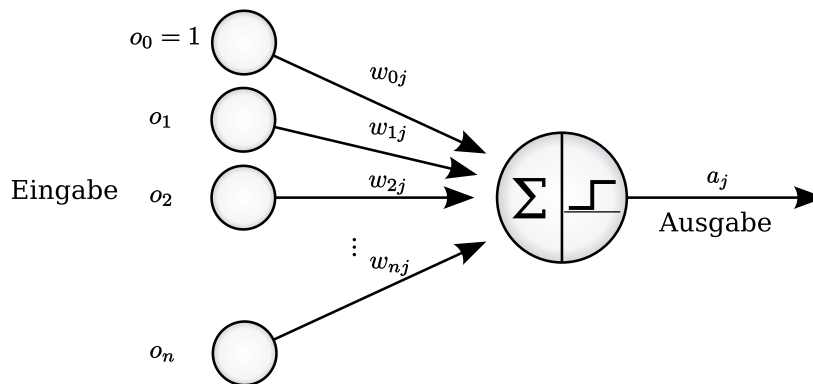


Abbildung 2.1: Einschichtiges Perzeptron mit n Eingaben o_i. Auf die gewichtete Summe $\sum_{i=0}^{n} w_{ij} o_i$ wird eine Schwellenwertfunktion angewandt um die Ausgabe a_j zu erhalten.

Bereits 1962 hat Rosenblatt das Perzeptron-Konvergenz-Theorem [Ros62] bewiesen, das besagt, dass „der Lernalgorithmus des Perzeptrons in endlicher Zeit konvergiert, d.h. das Perzeptron in endlicher Zeit alles lernen kann, was es repräsentieren kann.“ Die große Einschränkung ist hier die Repräsentierbarkeit, denn ein einstufiges Perzeptron kann nur die linear separierbaren Funktionen repräsentieren und scheitert daher selbst an einfachen Klassifikationsaufgaben, wie Minsky und Papert am Beispiel des XOR-Problems in [MP69] dargelegt haben.

Widner [Was89] hat in seinen theoretischen Untersuchungen gezeigt, dass unter allen möglichen Binärfunktionen der Prozentsatz der linear separierbaren Funktionen mit wachsender Anzahl der Eingabeneuronen stark abnimmt. So sind lediglich 0,002% aller binären Funktionen mit 5 Eingaben linear separierbar. Für viele praktische Anwendungen eignen sich Perzeptrons daher nicht. Deshalb hat sich aus diesem Modell das Multilayer-Perzeptron entwickelt, das über zusätzliche Schichten mit Neuronen verfügt.

2.1.2 Das Multilayer-Perzeptron

Erst in den 1980er Jahren haben sich durch die Entdeckung einer praktikablen Lernregel, nämlich des Backpropagation-Algorithmus (siehe Kapitel 2.1.3), mehrschichtige Perzeptrons durchgesetzt. Ein *Multilayer-Perzeptron* (MLP) besteht aus einer Vielzahl von Neuronen, die – wie in Abbildung 2.2 dargestellt – in mehreren Schichten angeordnet sind. Es verfügt über eine *Eingabeschicht* (engl. *input layer*), eine oder mehrere *verdeckte Schichten* (engl. *hidden layer*) und eine *Ausgabeschicht* (engl. *output layer*). Die verdeckten Schichten und die Ausgabeschicht bestehen aus informationsverarbeitenden, trainierbaren Neuronen, die ähnlich wie die Perzeptrons funktionieren. Die Eingabeschicht dagegen leitet ihre Aktivität ohne weitere Verarbeitung an die nächste Schicht weiter.

Die Neuronen einer Schicht sind nur mit denen der vorherigen Schicht verbunden. Eine an der Eingabeschicht angelegte Aktivität wird nach vorne propagiert. Man spricht in diesem Fall von einem vorwärtsgerichteten Netzwerk (engl. *feedforward net*). Ein Netzwerk, bei dem sämtliche Neuronen einer Schicht mit jedem Neuron der vorherigen Schicht verknüpft sind, nennt man *vollverknüpft*.

Ein Neuron j erhält als Netzeingabe net_j die gewichtete Summe aller mit ihm verknüpften Neuronen. Im Gegensatz zum Perzeptron wird die Ausgabe eines Neurons jedoch nicht durch

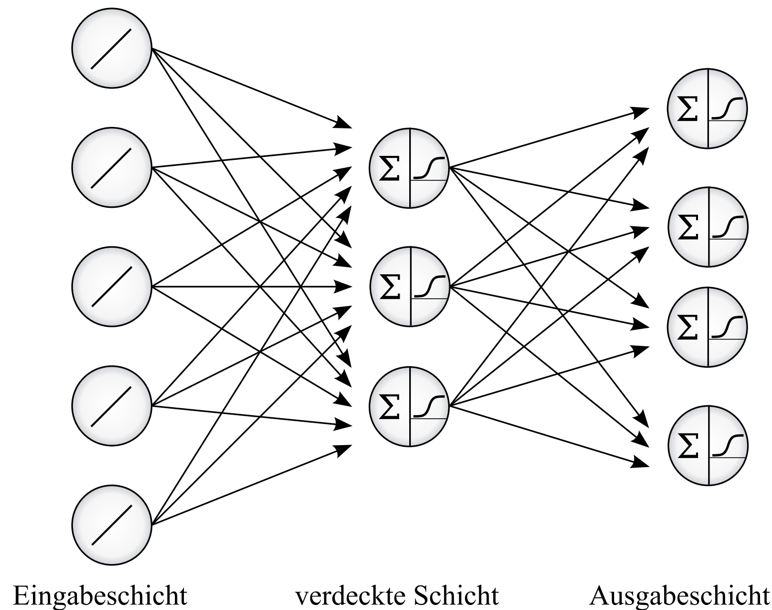


Abbildung 2.2: Zweischichtiges Multilayer-Perzeptron. Die verdeckte Schicht und die Ausgabeschicht haben trainierbare Gewichte.

die binäre Schwellenwertfunktion berechnet, sondern es kommen andere, differenzierbare *Aktivierungsfunktionen* zum Einsatz. In der Regel implementiert jedes Neuron eines MLP dieselbe Funktion. Einige typische Funktionen sind

$$f_{act}(x) = x \qquad \text{Linearfunktion} \qquad (2.3)$$

$$f_{act}(x) = \frac{1}{1+e^{-x}} \qquad \text{Logistische Funktion} \qquad (2.4)$$

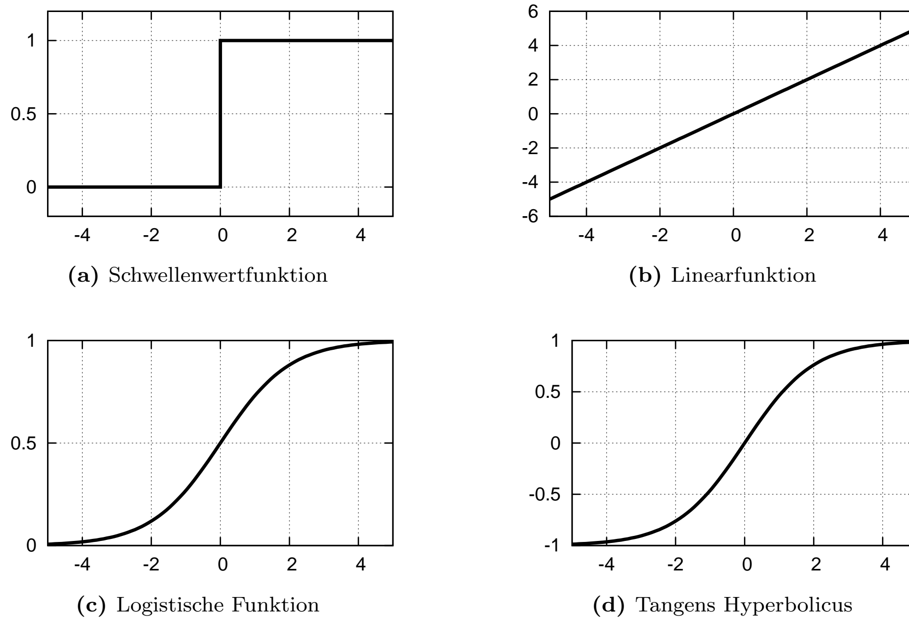
$$f_{act}(x) = \tanh(x) = \frac{1-e^{-x}}{1+e^{-x}} \qquad \text{Tangens Hyperbolicus} \qquad (2.5)$$

In Abbildung 2.3 sind die verschiedenen Aktivierungsfunktionen zusammen mit der im Perzeptron verwendeten Schwellenwertfunktion dargestellt. Sei w_{ij} das Gewicht zwischen einem Neuron i und einem Neuron j. Für die Aktivierung a_j des Neurons j, das mit n Eingabeneuronen verknüpft ist, gilt also:

$$a_j = f_{act}\left(\sum_{i=0}^{n} a_i \cdot w_{ij}\right) \qquad (2.6)$$

Im Gegensatz zum einstufigen Perzeptron lassen sich mit einem wie in Abbildung 2.2 dargestellten zweistufigen[1] MLP konvexe Polygone klassifizieren. Ein Netz mit mehr als drei Schichten kann sogar als universeller nichtlinearer Funktionsapproximator verwendet werden, sofern die Aktivierungsfunktion nichtlinear ist. Zusätzliche Schichten fügen dem Netz keine weiteren Fähigkeiten hinzu, und in vielen Fällen ist selbst die Berechnungskomplexität eines zweistufigen Perzeptrons bereits ausreichend.

[1]In dieser Arbeit wird ein Netz als zweistufig bezeichnet, wenn es über eine Eingabe-, eine verdeckte und



(a) Schwellenwertfunktion

(b) Linearfunktion

(c) Logistische Funktion

(d) Tangens Hyperbolicus

Abbildung 2.3: Verschiedene Aktivierungsfunktionen

Welche Funktion ein Multilayer-Perzeptron berechnet, hängt von den Gewichten der Verbindungen ab, die als Parameter der Funktion angesehen werden können. Geeignete Gewichte zu bestimmen, ist keine triviale Aufgabe, sondern oft nur durch Lernverfahren approximativ möglich. Das häufig angewandte Lernverfahren *Backpropagation of Error* wird im folgenden Abschnitt vorgestellt.

2.1.3 Lernverfahren Backpropagation

In der Regel werden die Parameter eines künstlichen neuronalen Netzes, also seine Gewichte, anhand von Trainingsdaten bestehend aus den Eingabewerten und der gewünschten Ausgabe erlernt. Da Trainieren eines neuronalen Netzes ein hochdimensionales, nichtlineares Optimierungsproblem ist, ist eine analytische Berechnung der optimalen Parameter in der Regel nicht praktikabel oder sogar unmöglich. Stattdessen wird meistens ein iterativer Ansatz verfolgt.

Als universell einsetzbar für die meisten Feedforward-Netze, insbesondere Multilayer-Perzeptrons, hat sich das Gradientenabstiegsverfahren *Backpropagation of Error* etabliert. Dieses Verfahren zur Fehlerminimierung wurde erstmals 1974 von Paul Werbos [Wer74] vorgestellt, wurde jedoch erst 1986 von der Wissenschaft [RHW86] wahrgenommen und führte zur immer noch anhaltenden Popularität der neuronalen Netze.

Backpropagation of Error ist ein überwachtes Lernverfahren, d.h. es setzt voraus, dass Trainingsmuster vorhanden sind, die dem neuronalen Netz „gezeigt“ werden können, um daraus zu lernen. Wünschenswert ist dabei, dass nicht nur die gezeigten Beispiele gelernt werden, sondern dass das Programm danach in der Lage ist, über die Trainingsmenge hinaus

eine Ausgabeschicht verfügt. In der Literatur ist diese Bezeichnung nicht einheitlich, daher wird hier der Argumentation gefolgt, dass nur die verdeckte und die Ausgabeschicht veränderliche Gewichte haben und informationstragend sind.

zu extrapolieren und auch zu völlig neuen Mustern derselben Problemklasse die richtige Ausgabe zu liefern.

Hierzu muss zunächst eine Fehlerfunktion E gewählt werden, die angibt, wie stark die tatsächliche Ausgabe von der gewünschten Ausgabe abweicht. Die verwendete Fehlerfunktion sei definiert als

$$E_p = \frac{1}{2}\sum_j (t_j - o_j)^2, \tag{2.7}$$

wobei t_j die gewünschte Ausgabe (*teacher*) des Neurons j und o_j seine tatsächliche Ausgabe für ein Muster p bezeichnet. Beliebige andere Fehlerfunktionen können auch zum Einsatz kommen (siehe 2.1.4), sofern sie stetig und differenzierbar sind.

Man bezeichnet einen Durchlauf aller Muster als eine *Lernepoche* und unterscheidet dabei zwei grundlegend verschiedene Möglichkeiten, Gewichtsänderungen (*updates*) durchzuführen:

Online-Training (stochastisches Lernen): Dem Netz wird ein Muster präsentiert, und der Fehler wird durch Vergleich mit der gewünschten Ausgabe berechnet. Anhand dessen werden die Gewichte so angepasst, dass der Fehler für dieses Muster reduziert wird.

Offline-Training (*batch learning*): Zunächst werden sämtliche Muster der Trainingsmenge durch das Netz propagiert und deren Fehler aufsummiert. Erst danach werden die Gewichte angepasst.

Da die Gewichtsanpassungen beim Offline-Training anhand des tatsächlichen Gradienten erfolgen, konvergiert das Lernverfahren vollständig gegen ein (möglicherweise nur lokales) Minimum. Das Training lässt sich zudem durch die Anwendung von Lernverfahren zweiter Ordnung (siehe [Zel94]) verfeinern, bei denen die geschätzte Krümmung der Fehleroberfläche berücksichtigt wird.

Beim Online-Training erfolgt die Anpassung der Gewichte dagegen lediglich auf Grundlage eines geschätzten Gradienten. Da diese Schätzung in der Regel verrauscht ist, kann es vorkommen, dass die Gewichte sich nicht entlang des tatsächlichen Gradienten bewegen. Im Gegensatz zum Offline-Training erfolgt aber viel häufiger, nämlich nach jedem Muster, eine Anpassung der Gewichte. Daher konvergiert stochastisches Lernen wesentlich schneller, insbesondere bei großen Datenmengen, die viele redundante Daten enthalten. Der verrauschte Gradient kann sogar vorteilhaft sein, da hiermit lokale suboptimale Minima beim Gradientenabstieg übersprungen werden können.

Sofern es möglich ist, wird in der Praxis aufgrund der wesentlich schnelleren Konvergenz meist Online-Training durchgeführt. Daher beziehen sich die Formeln im folgenden jeweils auf ein einziges Gewichtsupdate beim Online-Training.

Das Lernverfahren wird in zwei Schritten ausgeführt: Beim Vorwärtsschritt wird zunächst das Muster an der Eingabeschicht angelegt und dessen Aktivitäten durch das Netz propagiert. Danach – bei der Rückwärtspropagierung – wird für jede Schicht beginnend bei der Ausgabeschicht der Fehler jedes Neurons berechnet, und die Gewichte w_{ij} werden geringfügig um Δw_{ij} angepasst.

$$w_{ij} = w_{ij} + \Delta w_{ij} \tag{2.8}$$

Dabei soll der steilste Abstieg im Raum der Gewichte gewählt werden. Die Gewichtsanpassung Δw_{ij} soll also entgegen dem Gradienten erfolgen:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} \tag{2.9}$$

Wie stark ein Gewichtsupdate ausfallen soll, wird dabei über die *Lernrate* η kontrolliert, die entscheidenden Einfluss auf die Konvergenz und die Geschwindigkeit des Lernprozesses haben kann, wie im nächsten Abschnitt gezeigt wird.

Durch Anwendung der Kettenregel, wie in [Zel94] beschrieben, leitet sich aus der partiellen Ableitung der Fehlerfunktion die Delta-Regel

$$\Delta w_{ij} = \eta \cdot o_i \cdot \delta_j \tag{2.10}$$

für das Gewicht her, das von Neuron i zu Neuron j führt. Darin bezeichnet δ_j das Fehlersignal, welches sich je nach Schicht unterschiedlich berechnet. Für Neuronen der Ausgabeschicht gilt

$$\delta_j = f'_{act}(net_j) \cdot (t_j - o_j). \tag{2.11}$$

Für Neuronen in verdeckten Schichten ergibt es sich aus der gewichteten Summe der Fehler der Neuronen der Nachfolgeschicht:

$$\delta_j = f'_{act}(net_j) \cdot \sum_k \delta_k w_{jk} \tag{2.12}$$

Bei Verwendung der logistischen Aktivierungsfunktion oder des Tangens Hyperbolicus lassen sich die Fehlersignale sehr effizient berechnen, da sich die Ableitungen als Funktion der Ausgaben o_j ausdrücken lassen:

$$\begin{aligned} f'_{log}(net_j) &= f_{log}(net_j) \cdot (1 - f_{log}(net_j)) &&= o_j \cdot (1 - o_j) \\ f'_{tanh}(net_j) &= 1 - f^2_{tanh}(net_j) &&= 1 - o_j^2 \end{aligned}$$

Auf einfache Weise lassen sich mit diesem Verfahren rückwärtig, ausgehend von der Ausgabeschicht, für jedes Neuron das Fehlersignal δ_j und somit auch die Gewichtsänderungen Δw_{ij} für jedes Gewicht w_{ij} berechnen. Der Backpropagation-Algorithmus mit Online-Update unter Verwendung der logistischen Aktivierungsfunktion ist zusammenfassend in Algorithmus 1 dargestellt. Einige Schwierigkeiten dieses Gradientenabstiegsverfahrens werden im folgenden Abschnitt behandelt.

Probleme mit Backpropagation

Obwohl der Backpropagation-Algorithmus einfach zu implementieren ist, kann seine Anwendung eine Vielzahl von Problemen verursachen. Wie auch bei anderen Gradientenabstiegsverfahren wird zu jedem Zeitschritt nur eine relativ kleine lokale Umgebung der Fehleroberfläche bei der Suche nach einem Minimum berücksichtigt. Nach Zell [Zel94] werden im Folgenden einige damit verbundene Schwierigkeiten vorgestellt und auf ausgewählte konkrete Lösungen wird im Abschnitt 2.1.4 eingegangen.

Suboptimale lokale Minima Durch den iterativen Lernprozess kann ein lokales Minimum erreicht werden, das ein schlechteres Ergebnis liefert, als das globale Minimum. Da der Gradient am Scheitelpunkt dieses Minimums Null ist, wird der Algorithmus hier terminieren. Dieses Problem tritt verstärkt bei höherdimensionalen Netzen mit vielen Verbindungen auf, da hier die Fehleroberfläche besonders zerklüftet ist. Die Wahl einer kleineren Lernrate kann zwar helfen solche Minima zu vermeiden, eine allgemeingültige Lösung dieses Problems gibt es aber nicht. Ein Ansatz, die Methoden des *Simulated Annealing* auf Backpropagation anzuwenden, ist der *SARPROP* Algorithmus [TG98], bei dem zufälliges Rauschen und gelegentliche Neustarts helfen sollen, aus lokalen Minima zu springen.

Algorithmus 1 Backpropagation of Error

```
Wähle eine positive kleine Lernrate η
Initialisiere die Gewichte auf zufällige, betragsmäßig kleine Werte
for Epochen n = 1 to N do
  for Muster p = 1 to P do
    propagiere Muster x_p durch das Netz
    for Ausgabeneuronen k = 1 to K do
      berechne das Fehlersignal
      δ_k ⇐ o_j · (1 − o_j) · (t_j − o_j)
      berechne die Gewichtsänderungen der Gewichte zu den Ausgabeneuronen
      Δw_jk ⇐ η · δ_k · o_j
    end for
    for verdeckte Neuronen j = 1 to J do
      berechne das Fehlersignal
      δ_j ⇐ o_j · (1 − o_j) · Σ_{k=1}^{K} δ_k · w_jk
      berechne die Gewichtsänderungen der verdeckten Gewichte
      Δw_ij ⇐ η · δ_j · o_i
    end for
    for alle Gewichte do
      führe die Gewichtsupdates aus
      w ⇐ w + Δw
    end for
  end for
end for
```

Flache Plateaus Auch zu flache Regionen der Fehleroberfläche können ein Problem darstellen, da hier die Ableitung nahe bei Null sein kann. Die Gewichtsänderungen stagnieren dann, und viele Epochen sind nötig, um solche Plateaus wieder zu verlassen. Insbesondere wenn die Aktivität eines Neurons sich den Asymptoten der Aktivierungsfunktion annähert – die Funktion also gesättigt ist – kann dies auftreten. Vermeiden lässt sich dieses Phänomen unter anderem durch eine additive Konstante in der Aktivierungsfunktion (*Flat Spot Elimination* [Fah88]) oder durch Einfügen eines Momentum-Terms, wie in [RHW86] vorgeschlagen. Dieser Term berücksichtigt anteilig die Gewichtsänderung der letzten Epoche, um den „Schwung“ des vorherigen Abstiegs auszunutzen.

Oszillation in steilen Tälern An Stellen, wo die Fehleroberfläche besonders steil und somit der Gradient besonders groß ist, kann es vorkommen, dass ein lokales Minimum übersprungen wird und der Gradient auf der anderen Seite des Minimums in die entgegengesetzte Richtung zeigt. Möglicherweise kommt es dadurch zu Oszillationen und das lokale Minimum wird nie erreicht. Neben dem zuletzt erwähnten Momentum-Term kann auch der in Kapitel 2.1.4 vorgestellte Rprop-Algorithmus dieses Problem abmildern.

Verlassen guter Minima Zu hohe Lernraten und auch ein zu großer Gradient können bewirken, dass ein gutes lokales Minimum durch eine zu starke Gewichtsanpassung wieder verlassen wird und Backpropagation stattdessen gegen eine schlechtere Lösung konvergiert. Dies ist insbesondere bei engen, steilen Tälern möglich. Bei reinem Backpropagation tritt das Problem in der Praxis sehr selten auf, einige Modifikationen, wie z.B. der Momentum Term oder auch Rprop, können es jedoch verstärken.

Schlechte Wahl der Lernrate Die Wahl der Lernrate η hat große Auswirkungen darauf, wie stark die bisher genannten Probleme auftreten können. Eine zu große Lernrate verursacht starke Sprünge auf der Fehleroberfläche und Oszillationen. Außerdem können gute lokale Minima verlassen werden, und das Netz kann divergieren. In Kombination mit Online-Training kann es bei einer zu großen Lernrate vorkommen, dass die Gewichte sich zu sehr auf die anfangs gesehenen Muster spezialisieren und die späteren Muster vom Netz kaum gelernt werden. Bei einer zu kleinen Lernrate hingegen konvergiert das Verfahren zu langsam, so dass der Zeitaufwand in der Praxis nicht akzeptabel ist. Es existieren verschiedene Erweiterungen des Lernalgorithmus, die die Lernrate dynamisch anpassen. Neben Rprop seien hier noch die stochastische, diagonale Levenberg-Marquardt-Methode [LBOM98] sowie die Delta-Bar-Delta-Regel [Jac87] erwähnt.

Kleiner Gradient in tiefen Netzen In einem Netz mit vielen Schichten – einem tiefen Netz – schrumpft der propagierte Fehler und somit auch der Gradient von Schicht zu Schicht. Dies hat zur Folge, dass die Gewichte zur Eingabeschicht hin wesentlich langsamer angepasst werden. Eine Möglichkeit dies zu vermeiden ist es, in tieferen Schichten eine größere Lernrate einzusetzen, oder, wie bei Rprop, die Magnitude des Gradienten nicht in das Gewichtsupdate einfließen zu lassen.

2.1.4 Erweiterungen von Backpropagation

Die bei der Verwendung von Backpropagation auftretenden Probleme haben dazu geführt, dass eine Reihe von optionalen Erweiterungen entwickelt wurden. Einige davon versuchen, dem negativen Einfluss einer schlecht gewählten Lernrate entgegenzuwirken, die Generalisierung des Netzes zu verbessern (z.B. Optimal Brain Damage [LDS+90]), oder den Lernprozeß zu beschleunigen (z.B. Quickprop [Fah88]). Welche Erweiterungen sinnvoll sind, ist sowohl von der Problemstellung, als auch von der Netztopologie abhängig. Daher werden hier nur die in dieser Arbeit angewandten Erweiterungen vorgestellt.

Resilient Backpropagation

Mit Resilient Backpropagation (kurz *Rprop*) haben Martin Riedmiller und Heinrich Braun 1992 in [RB92] einen Lernalgorithmus vorgestellt, bei dem die Wahl der oft problematischen Lernrate η entfällt und der zudem auch noch außerordentlich schnell konvergiert.

Grundidee des Verfahrens ist, dass jedem Gewicht w_{ij} eine eigene Lernrate $\Delta_{ij}^{(t)}$ zugeordnet wird, die jeden Zeitschritt t angepasst wird. Hierzu wird der Gradient der aktuellen Epoche $\frac{\partial E^{(t)}}{\partial w_{ij}}$ mit dem Gradienten der vorherigen Epoche verglichen. Wenn beide in dieselbe Richtung zeigen, kann die Lernrate um einen Faktor η^+ erhöht werden. Unterscheidet sich das Vorzeichen der Gradienten jedoch, so bedeutet das, dass ein lokales Minimum übersprungen wurde und die Lernrate um η^- reduziert werden muss:

$$\Delta_{ij}^{(t)} = \begin{cases} \Delta_{ij}^{(t-1)} \cdot \eta^+, & \text{falls } \frac{\partial E^{(t-1)}}{\partial w_{ij}} \cdot \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ \Delta_{ij}^{(t-1)} \cdot \eta^-, & \text{falls } \frac{\partial E^{(t-1)}}{\partial w_{ij}} \cdot \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ \Delta_{ij}^{(t-1)}, & \text{sonst} \end{cases} \tag{2.13}$$

$$\text{mit } 0 < \eta^- < 1 < \eta^+.$$

Zusätzlich bleibt die Lernrate unverändert, wenn sie im letzten Schritt reduziert wurde. Wie im vorherigen Kapitel gezeigt, kann die Größe des Gradientens negativen Einfluss auf den

Lernprozeß nehmen, daher bestimmt Rprop lediglich die Richtung eines Updates aus dem Vorzeichen der partiellen Ableitung. Die Gewichtsänderung $\Delta w_{ij}^{(t)}$ wird anhand der Regel

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)}, & \text{falls } \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ +\Delta_{ij}^{(t)}, & \text{falls } \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ 0, & \text{sonst} \end{cases} \tag{2.14}$$

bestimmt. Die Regeln dieses adaptiven Verfahrens können effizient implementiert werden (siehe Algorithmus 2); es wird lediglich etwas mehr Speicher für die Lernraten und die Gradienten der letzten Epoche benötigt.

Auch die Wahl der Parameter ist bei Rprop weitgehend unkritisch. Experimentell hat sich herausgestellt, dass $\eta^+ = 1{,}2$ und $\eta^- = 0{,}5$ in den meisten Fällen gute Adaptionsfaktoren sind. Zudem sollte man das Wachstum der Lernraten nach oben beschränken, wofür Riedmiller in [RB94] eine obere Schranke von $\Delta_{max} = 50$ vorschlägt. In Sonderfällen, z.B. bei stark zerklüfteten Fehleroberflächen, ist es sinnvoll, diesen Parameter niedriger zu wählen. Um numerischen Problemen bei der Berechung vorzubeugen, kann zudem eine minimale Lernrate Δ_{min} in Abhängigkeit der Berechnungsgenauigkeit festgelegt werden. Des Weiteren müssen die Lernraten mit einem positiven, nichtnegativen Wert initialisiert werden, z.B. $\Delta_0 = 0{,}1$, dessen Wahl aber kaum Einfluss auf die Konvergenz des Verfahrens hat.

Rprop kann die Gewichtsänderungen erst nach einer vollständigen Trainingsepoche ausführen, da es einen stabilen Gradienten benötigt. Trotzdem ist Rprop was die Trainingszeit angeht auch Online-Lernverfahren in der Regel deutlich überlegen. Insbesondere in tiefen Netzen ist das Wachtum der Gewichte besser verteilt. Denn auch die Gewichte bei der Eingabeschicht, wo der Gradient kleine Werte annimmt, haben dieselbe Chance zu wachsen, wie Gewichte der Ausgabeschicht.

Durch die teils extrem großen Lernraten kann es bei Rprop vorkommen, dass ein gutes lokales Minimum übersprungen wird. Auch in der Praxis hat sich herausgestellt, dass Netze mit Rprop leichter übertrainiert werden können und dann nicht mehr gut generalisieren. Eine Möglichkeit dem entgegenzuwirken ist der im nächsten Abschnitt vorgestellte *Weight-Decay-Term*.

Weight Decay

Die Generalisierungsfähigkeit eines neuronalen Netzes hängt zum großen Teil von einer ausgeglichenen Balance zwischen Netzgröße und der Komplexität der Daten ab. Bei einem zu großen Netz findet eine Überanpassung (engl. *overfitting*) statt, ist das Netz aber zu klein dimensioniert, dann lernt es überhaupt nicht. Die von Paul Werbos in [Wer88] vorgestellte Modifikation von Backpropagation verzichtet jedoch darauf, die Netztopologie anzupassen, sondern beschränkt stattdessen das Wachstum der Gewichte.

Die Motivation hinter dem als *Weight Decay* (Gewichtsabnahme) bezeichnetem Verfahren ist, dass zu große Gewichte neurobiologisch unplausibel sind. Da die Fehleroberfläche bei großen Gewichten zerklüfteter ist, kann es im Lernprozeß häufiger zu Oszillationen und Sprüngen kommen. Um einen kleinen Absolutbetrag der Gewichte zu erreichen, wird daher ein Bestrafungsterm auf die Fehlerfunktion addiert. Dazu eignet sich z.B. die Summe der quadratischen Gewichte, was folgende modifizierte Fehlerfunktion ergibt:

$$E = E' + \lambda \cdot \frac{1}{2} \sum_{i,j} (w_{ij})^2, \tag{2.15}$$

Algorithmus 2 Rprop (Resilient Backpropagation)

Initialisiere die Lernraten $\Delta_{ij}(t)$ mit Δ_0
Initialisiere die Gradienten mit $\frac{\partial E}{\partial w_{ij}}(t-1) = 0$
repeat
  Berechne den Gradienten $\frac{\partial E}{\partial w}(t)$
  for alle Gewichte w_{ij} **do**
    if $(\frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) > 0)$ **then** //Gradient unverändert
      $\Delta_{ij}(t) \Leftarrow$ **minimum**$(\Delta_{ij}(t-1) \cdot \eta^{+}, \Delta_{max})$
      $\Delta w_{ij}(t) \Leftarrow -$**sign**$(\frac{\partial E}{\partial w_{ij}}(t) \cdot \Delta_{ij}(t))$
      $\frac{\partial E}{\partial w_{ij}}(t-1) \Leftarrow \frac{\partial E}{\partial w_{ij}}(t)$
    else if $(\frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) < 0)$ **then** //Gradient geändert
      $\Delta_{ij}(t) \Leftarrow$ **maximum**$(\Delta_{ij}(t-1) \cdot \eta^{-}, \Delta_{min})$
      $\Delta w_{ij}(t) \Leftarrow 0$
      $\frac{\partial E}{\partial w_{ij}}(t-1) \Leftarrow 0$
    else if $(\frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) = 0)$ **then** //Gradient Null
      $\Delta w_{ij}(t) \Leftarrow -$**sign**$(\frac{\partial E}{\partial w_{ij}}(t) \cdot \Delta_{ij}(t))$
      $\frac{\partial E}{\partial w_{ij}}(t-1) \Leftarrow \frac{\partial E}{\partial w_{ij}}(t)$
    end if
    führe Gewichtsupdate aus
    $w_{ij}(t+1) \Leftarrow w_{ij}(t) + \Delta w_{ij}(t)$
  end for
until Fehler konvergiert

dabei bezeichnet λ den *Decay-Faktor*, mit dem die Stärke der Dämpfung angegeben werden kann. Mit Bildung der partiellen Ableitung nach den Gewichten gilt dann für Gewichtsänderungen

$$\Delta w_{ij}(t+1) = \eta \cdot (o_i \cdot \delta_j - \lambda \cdot w_{ij}(t)) \,. \tag{2.16}$$

Der Decay-Faktor sollte klein gewählt werden, in [KH92] wurden mit Werten zwischen 0,0001 und 0,00005 gute Ergebnisse erzielt. Einen geeigneten Faktor zu ermitteln ist aber ähnlich schwierig wie eine optimale Lernrate zu finden. Am besten wäre es, für jedes Gewicht einen separaten Faktor zu bestimmen, zumindest sollte man aber in jeder Schicht eine eigene Konstante festlegen.

Bei geeigneter Wahl der Parameter unterdrückt dieses Verfahren irrelevante Gewichte. Daher kann man das Verfahren auch zur Minimierung eines Netzes verwenden, indem man Gewichte, die unter einem gewissen Schwellenwert liegen, komplett entfernt. Zudem wird die Initialisierung der Gewichte weniger wichtig und insbesondere bei kleinen Trainingsmengen führt das zu einer besseren Generalisierungsleistung.

2.2 Konvolutionsnetze

Für eine Vielzahl von Klassifikationsaufgaben, unter anderem auch zur Bilderkennung, werden neuronale Netze bereits eingesetzt. Die klassische Herangehensweise ist dabei, anhand eines manuell festgelegten Merkmalsextraktors zunächst die relevanten Daten aus der hochdimensionalen Eingabe zu gewinnen. Der so erzeugte, niedrig-dimensionale Merkmalsvektor

dient dann als Eingabe für ein Multilayer-Perzeptron, welches die eigentliche Klassifikationsaufgabe übernimmt. Solche empirisch gewonnenen Merkmale stellen jedoch nicht immer eine geeignete Repräsentation der Daten dar. Interessanter und auch biologisch plausibler wäre es daher, die Rohdaten, also z.B. Bilder, direkt als Eingabe eines neuronalen Netzes zu verwenden.

2.2.1 Motivation

Vollverknüpfte, vorwärtsgerichtete Netze wie Multilayer-Perzeptrons sind gleich in mehrerlei Hinsicht zum Extrahieren von Merkmalen nur schlecht geeignet.

Zum einen stellt die Größe eines MLPs bei hoher Eingabedimension ein Problem dar. Eingabedimensionen von einigen 10.000 Werten sind bei Bilddaten nicht unüblich, was aber dazu führt, dass einige 100.000 Gewichte in den verdeckten Schichten trainiert werden müssen. Die Komplexität des Netzes steht dann in keinem Verhältnis zur häufig vergleichsweise geringen Anzahl an Trainingsbeispielen, was zu einer Überanpassung des Netzes führt.

Ein weiteres großes Defizit von Multilayer-Perzeptrons ist, dass sie die Topologie der Eingabe völlig ignorieren. Die z.B. bei Bilddaten starke Korrelation zwischen benachbarten Pixeln wird von traditionellen Merkmalsextraktoren ausgenutzt, geht aber bei einem vollverknüpften Netz verloren. Zudem ist ein trainiertes MLP auch nicht invariant gegenüber Translationen der Eingabe. Um das gleiche Muster an verschiedenen Stellen der Eingabe zu erkennen, müsste es unterschiedliche Gewichte für dieselbe Aufgabe ausbilden.

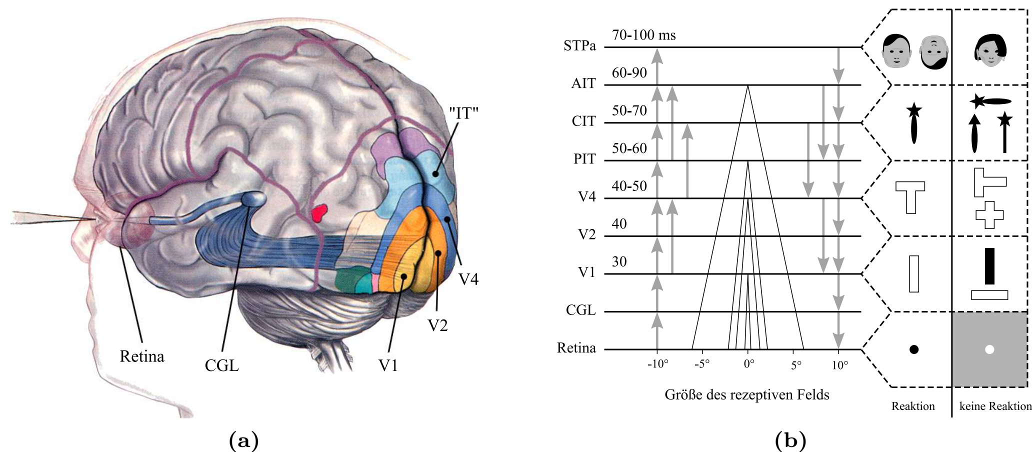


Abbildung 2.4: Visuelle Wahrnehmung im menschlichen Gehirn. (a) Die unterschiedlichen Areale im visuellen Cortex reagieren auf zunehmend komplexere Formen (aus [Log99], bearbeitet). (b) In höheren Schichten des visuellen Systems wird ein größeres rezeptives Feld der Eingabe wahrgenommen. (aus [Wis04], bearbeitet)

Wie auch schon bei der Entwicklung des Perzeptrons hat die Analyse des biologischen Nervensystems ein neues wissenschaftliches Modell begründet. Denn im visuellen Cortex der Großhirnrinde von Menschen und Säugetieren wurden lokal sensitive Neuronen entdeckt, die ähnlich wie Merkmalsextraktoren arbeiten. So reagieren die Zellen des primären visuellen Cortex (V1) selektiv auf Linien und Kanten spezieller Orientierung (siehe Abb. 2.4). Die nächste Schicht von Nervenzellen, V2, erhält diese Merkmale als Eingaben und reagiert auf geringfügig komplexere Figuren wie Ecken, Linienschnitte oder Kreise. Untersuchungen deuten darauf hin, dass diese Informationsverarbeitung in den höheren Schichten fortgeführt

wird, so dass deren Nervenzellen auf komplexe Formen, etwa Buchstaben und Gesichter, reagieren.

Als einer der ersten Versuche diese Form der Informationsverarbeitung künstlich nachzubilden, kann das von Kunihiko Fukushima entwickelte *Neocognitron* (siehe u.a. [Fuk86]) angesehen werden. Sein schichtweise aufgebautes hierarchisches Netzwerk, eignet sich zur skalierungs- und translationsinvarianten Erkennung von visuellen Mustern, insbesondere von handgeschriebenen Ziffern. Die Schichten bestehen abwechselnd aus S-Zellen, die Muster extrahieren, und C-Zellen, die die Auflösung verringern. Aufgrund einer Vielzahl von problemspezifischen Parametern ist es jedoch leider nur schwer zu trainieren.

Das Neocognitron bildete den Anfang einer Reihe von Modellen, die sich an der Informationsverarbeitung des visuellen Cortex orientieren. Ein wichtiger Durchbruch war dabei das von LeCun *et al.* in [LBD+90] vorgestellte Netzwerk, da es das erste Modell darstellt, welches sich effizient mit Backpropagation trainieren lässt. Weitere neuronale Architekturen, die Merkmale mit Hilfe von Faltungen extrahieren, sind das HMAX-Modell [RP99] und die neuronale Abstraktionspyramide [Beh03]. Zusammenfassend werden derartige Modelle oftmals als *Convolutional Neural Networks* (neuronale Konvolutionsnetze) bezeichnet. Das namensgebende Modell von LeCun, das auch die Basis für viele Weiterentwicklungen bildet, wird exemplarisch im Folgenden erläutert.

2.2.2 Funktionsweise am Beispiel von LeNet-5

Das erstmals 1989 von LeCun *et al.* vorgestellte Konvolutionsnetz zur Erkennung handschriftlicher Zeichen wurde seitdem mehrmals überarbeitet. Die wohl bekannteste Version dieses Netzes ist in Abbildung 2.5 dargestellt und trägt den Namen *LeNet-5*. Das vorwärtsgerichtete Netz erhält als Eingabe ein 32×32 Pixel großes Graustufenbild eines handgeschriebenen Zeichens und soll dies einem von 96 ASCII-Zeichen zuordnen. Es besteht aus einer Eingabeschicht und sieben informationsverarbeitenden Schichten (engl. *layers*), die wiederum in mehrere Merkmalskarten (engl. *feature maps*) unterteilt sind. Unterschiedliche, zunehmend komplexere Merkmale sollen aus der vorherigen Schicht extrahiert werden. Nicht jede Schicht führt jedoch die gleiche Art von Operation aus, sondern es gibt zwei verschiedene Typen: *Konvolutionsschichten* und *Subsampling-Schichten*.

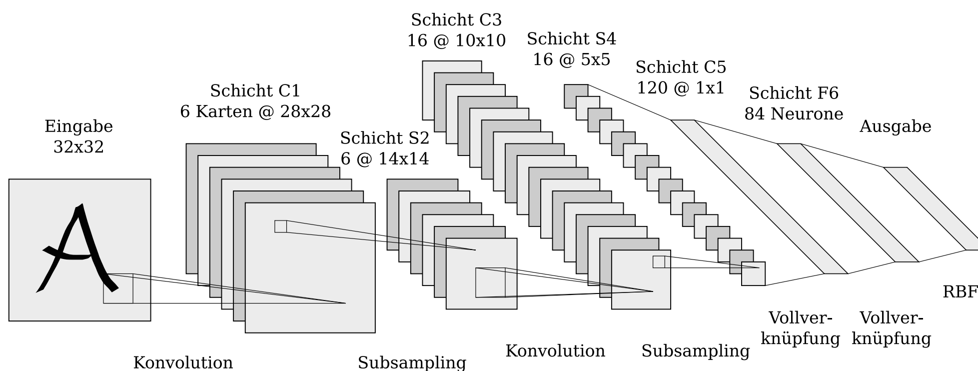


Abbildung 2.5: Architektur des Konvolutionsnetzes *LeNet-5* zur Erkennung handschriftlicher Zeichen. Abwechselnd werden die Eingaben gefaltet und herunterskaliert. (nach [LBBH98])

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Tabelle 2.1: Schema der Verknüpfungen zwischen Schicht C2 und S3. Ein Kreuz bedeutet, dass eine Merkmalskarte der Schicht S2 (Zeilen) auf eine Karte der Schicht C3 (Spalten) abgebildet wird.

Konvolutionsschichten

Die *C-Schicht* führt Faltungsoperationen auf der Eingabekarte aus und generiert daraus unterschiedliche neue Merkmalskarten gleicher Auflösung. Die erste Schicht C1 besteht z.B. aus 28×28 Neuronen, die jeweils mit einer 5×5 Umgebung von Neuronen in der vorherigen Schicht verbunden sind, so dass sich die rezeptiven Felder benachbarter Neuronen überlappen. Ähnlich wie bei einem Perzeptron wird für jedes Pixel dieser Schicht die gewichtete Summe seiner 25 Eingaben berechnet und ein Bias hinzuaddiert. Auf das Ergebnis wird eine nichtlineare Aktivierungsfunktion angewandt.

Da sich die Aktivität jedes Zielneurons aus einem kleinen Fenster der vorherigen Schicht berechnet, können die Gewichte dieser Verknüpfung als Kern einer diskreten zweidimensionalen Faltung betrachtet werden. Eine Verschiebung der Eingabe hat daher zur Folge, dass auch die Ausgabe entsprechend verschoben ist; diese merkmalsextrahierende Operation ist also translationsinvariant.

Subsampling-Schichten

Die Aufgabe einer *S-Schicht* ist es, die räumliche Auflösung der Eingabe zu reduzieren. Dazu betrachtet jedes Neuron eine 2×2-Umgebung der vorherigen Merkmalskarte, errechnet den Durchschnitt dieser vier Eingaben und multipliziert ihn mit einem einzigen trainierbaren Koeffizienten. Auch in dieser Schicht wird zuletzt eine Aktivierungsfunktion angewandt. Im Gegensatz zu den Konvolutionsschichten überlappen sich die rezeptiven Felder von benachbarten Neuronen in dieser Schicht nicht. Daher reduziert sich die Auflösung in beiden Dimensionen um die Hälfte.

Die abnehmende Größe der Merkmalskarten wird dadurch kompensiert, dass deren Anzahl in tieferen Schichten zunimmt. Aus der einen Karte der Eingabeschicht extrahiert die Schicht C1 sechs verschiedene Merkmalskarten. C3 beinhaltet bereits 16 Merkmalskarten, die sich aus unterschiedlichen Kombinationen der sechs Eingabekarten der vorherigen Schicht zusammensetzen. Die Schichten sind nach dem in Tabelle 2.1 angegebenen Schema verknüpft, um Symmetrien in der Netzstruktur zu brechen und das Netz zur Generierung von unterschiedlichen Merkmalen zu zwingen. In der letzten Konvolutionsschicht C5 sind alle 120 Karten wieder mit jeder der 16 Eingabekarten verbunden. Jedoch haben die Karten nur noch eine Auflösung von einem Pixel, sie besteht also effektiv aus 120 Neuronen. Auf C5 folgt eine normale, vollverknüpfte Schicht mit 84 Einheiten.

Die Ausgabeschicht stellt eine Besonderheit dar und hebt LeNet-5 von seinen Vorgängerversionen ab. Sie besteht aus einem Radialen-Basisfunktionen-Neuron (RBF) für jede Ausgabeklasse und erhält die 84 Eingaben der vorherigen Schicht. Die Ausgabe y_i einer RBF-

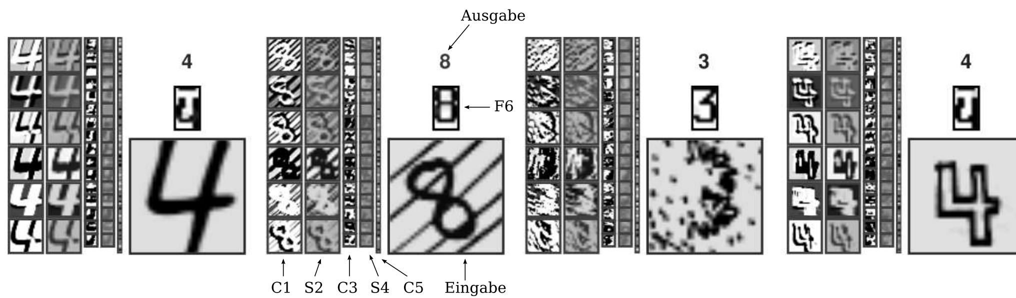


Abbildung 2.6: Die Aktivitäten der Merkmalskarten des Konvolutionsnetzes bei Eingabe verschiedener Muster. Selbst stark verrauschte Muster können noch korrekt erkannt werden. (aus [LBBH98])

Einheit berechnet sich aus der euklidischen Distanz zwischen dem Eingabevektor x_j und dem Gewichtsvektor w_{ij}:

$$y_i = \sum_{j=1}^{84} (x_j - w_{ij})^2. \tag{2.17}$$

Dazu werden die Gewichte, also die Zentren der Radialen-Basisfunktionen, zunächst auf feste Werte mit -1 oder $+1$ initialisiert und so gewählt, dass sie in einer 7×12 Matrix ein stilisiertes Bild des entsprechenden Zeichens wiedergeben. Diese Darstellung wurde bewusst so gewählt, da sich viele Zeichen, etwa die Ziffer 0 und der Buchstabe *O* sehr ähnlich sehen. Zum Beispiel im Zuge einer linguistischen Nachbearbeitung kann eine solche Ausgabe hilfreich sein, um aus nicht eindeutig erkannten Buchstaben eines Wortes den richtigen auszuwählen.

Aufgrund der RBF-Neuronen haben LeCun *et al.* außerdem eine besondere Fehlerfunktion verwendet, nämlich

$$E(W) = \frac{1}{P} \sum_{p=1}^{P} (y_{pD^p} + \log(e^{-j} + \sum_i e^{-y_{pi}})), \tag{2.18}$$

wobei P die Anzahl der Trainingsmuster bezeichnet, D_p die gewünschte Ausgabeklasse und y_{pi} die Aktivierung des Ausgabeneurons i für das Muster p ist. Diese Fehlerfunktion E zu minimieren bedeutet, beide Terme zu minimieren. Dabei bezeichnet der erste Term den euklidischen Abstand der gewünschten Ausgabe zum 84-dimensionalen RBF-Zentrum. Der zweite Term erreicht, dass die Zentren der RBF-Neuronen möglichst weit voneinander entfernt sind, um die triviale Lösung zu verhindern, nämlich dass alle Zentren ineinander liegen.

Mit dieser Netzstruktur konnte auf der Testmenge der MNIST-Datenbank eine sehr gute Fehlerquote von bis zu 0,8% nicht erkannten Mustern erreicht werden. Zum damaligen Zeitpunkt waren dies die besten auf dieser Datenbank erreichten Ergebnisse (siehe Abschnitt 2.3.1.)

2.2.3 Eigenschaften von Konvolutionsnetzen

LeCun *et al.* [LB95] identifizieren drei grundlegende Eigenschaften, die Konvolutionsnetze charaktierisieren: lokale rezeptive Felder, gekoppelte Gewichte (engl. *shared weights*) und räumliches oder temporales Subsampling. Diese topologischen Merkmale ermöglichen es, ein Netz effizient für Aufgaben der Objekterkennung zu trainieren.

Lokale rezeptive Felder

Jedes Neuron des Netzes erhält nur Eingaben einer kleinen Umgebung der vorherigen Schicht. In der Konvolutionsschicht C1 besteht das rezeptive Feld jedes Neurons zum Beispiel nur aus einer 5×5-Nachbarschaft in der Eingabeschicht. Diese gegenüber einer Vollverknüpfung reduzierte Anzahl von Verbindungen hat zum Vorteil, dass weniger Rechenoperationen benötigt werden. Elementare visuelle Merkmale können damit an jeder Stelle im Bild extrahiert werden, und bei Verschiebungen oder Verzerrungen der Eingabe ändert sich nur die Position der hervorstehenden Merkmale.

Gekoppelte Gewichte

Wenn unterschiedliche Verbindungen dasselbe Gewicht verwenden, spricht man von gekoppelten Gewichten (engl. *shared weights*). In Konvolutionsnetzen werden innerhalb einer Merkmalskarte an jeder Position dieselben Gewichte verwendet. Zum einen ermöglicht das, die gleiche Operation an jeder Position des Bildes auszuführen, zum anderen wird dadurch eine gewisse Translationsinvarianz und Robustheit gegenüber Verzerrungen erreicht. Im Beispiel von LeNet-5 werden in der Schicht C3 pro Eingabekarte und Merkmalskarte lediglich 26 Koeffizienten (25 Gewichte zzgl. eines Biasgewichts) benötigt. Dass ein solches Netz trotz seiner Größe nur relativ wenige Parameter besitzt, führt zum einen zu einer guten Generalisierungsleistung und reduziert zum anderen die Dimension des Optimierungsproblems.

Subsampling

Die Motivation für die Verwendung von Subsampling-Schichten ist, dass die exakte Position eines erkannten Merkmals weniger relevant ist, als vielmehr die grobe Lage der Merkmale relativ zueinander. Da unterschiedliche Muster derselben Klasse leicht variieren, wäre es für die Erkennung sogar nachteilig, wenn sich das Verfahren zu sehr auf die exakte Position eines Merkmals stützen würde. Die Sensitivität der Ausgabe gegenüber Verschiebungen und Distortionen soll durch eine Reduzierung der räumlichen Auflösung verringert werden. Die sinkende Genauigkeit wird in diesem Modell durch eine ansteigende Reichhaltigkeit der Repräsentation kompensiert. Daher erhöht sich im Gegenzug die Anzahl der Merkmalskarten in der Verarbeitungshierarchie.

Konvolutionsnetze kommen im Gegensatz zu vollverknüpften Netzen mit wesentlich weniger Parametern aus und nutzen zudem das Vorwissen über die Zweidimensionalität der Eingabe aus. Auch bei kleinen Trainingsmengen können sie daher mit Gradientenabstiegsverfahren für Mustererkennungsaufgaben trainiert werden. Ihre Netztopologie muss jedoch auf die konkrete Aufgabe sorgfältig abgestimmt werden.

2.3 Datensätze zur Objekterkennung

Menschen lernen anhand von Interaktionen mit ihrer Umgebung, verschiedene Objekte voneinander zu unterscheiden. Ein neuronales Netz muss diese Fähigkeit aus dem Trainingsdatensatz ableiten können. Dafür ist es einerseits nötig, dass hinreichend viele Trainingsbeispiele vorhanden sind und diese das volle Spektrum der Problemklasse widerspiegeln, andererseits müssen die in den Mustern dargestellten Objekte bei überwachten Lernverfahren markiert sein. Im Folgenden werden einige Datensätze vorgestellt, mit denen neuronale Netze zur Objekterkennung trainiert werden können.

2.3.1 MNIST-Datensatz handgeschriebener Ziffern

Die frei verfügbare MNIST-Datenbank handschriftlicher Ziffern[2] wird häufig als Standardtest zur Evaluierung von Verfahren der Mustererkennung verwendet. Sie besteht aus einer Trainingsmenge von 60.000 Graustufenbildern mit einer Auflösung von 28×28 Pixeln und einer gleichartigen Testmenge mit 10.000 Bildern.

Der Datensatz setzt sich zusammen aus Teilen der ursprünglich als Testmenge gedachten *Special Database 1* und der als Trainingsmenge gedachten *Special Database 3* des amerikanischen *National Institute of Standards and Technology* (*NIST*). Die Muster im Datensatz SD-3 wurden unter Büroangestellten erhoben, welche eine deutlich klarere und sauberere Handschrift hatten als die Schüler, aus deren handgeschriebenen Ziffern der Datensatz SD-1 besteht. Da eine derart starke Korrelation innerhalb der Trainings- und der Testmenge die Ergebnisse eines Lernverfahrens verfälschen kann, wurden beide Datensätze gemischt. Die Trainingsmenge der modifizierten NIST-Datenbank (*MNIST*) besteht aus jeweils 30.000 Bildern eines Datensatzes; für die Testmenge wurden jeweils 5.000 Bilder ausgewählt.

Unter Beibehaltung des Seitenverhältnisses wurden die ursprünglichen Binärbilder auf 20×20 Pixel größennormalisiert und aufgrund der damit verbundenen Glättung in Graustufenbilder konvertiert. Anschließend wurden die Ziffern anhand ihres Masseschwerpunkts innerhalb eines 28×28 Pixel großen Fensters zentriert. Einige Beispiele der Trainingsmenge sind in Abbildung 2.7 dargestellt.

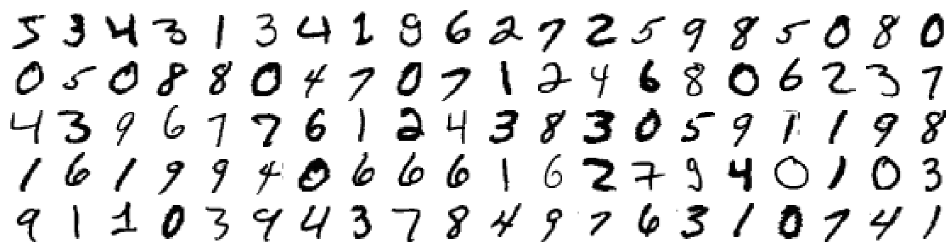


Abbildung 2.7: Einige Ziffern aus der Trainingsmenge der MNIST-Datenbank

Am Beispiel der MNIST-Datenbank haben LeCun *et al.* [LBBH98] unterschiedliche Verfahren zur Mustererkennung verglichen, siehe Tabelle 2.2. Sämtliche Verfahren wurden ausschließlich anhand der Muster der Trainingsmenge trainiert. Ein vollverknüpftes Multilayer-Perzeptron erreichte auf der Testmenge eine Fehlerrate von 2,95%. Hierfür wurde ein relativ großes vorwärtsgerichtetes dreischichtiges Netz mit 28×28-1000-150-10 Neuronen verwendet. Die besten Ergebnisse erreichten hingegen neuronale Konvolutionsnetze. Bereits 1998 erzielte LeNet-5 nach 20 Trainingsepochen einen Testfehler von 0,95%. Schon damals zeigte sich aber auch, dass der Lernerfolg hauptsächlich von der Größe und Qualität der Trainingsmenge abhängt. Durch affine Transformationen (Translationen, Skalierungen und Scherungen) wurden aus der ursprünglichen Trainingsmenge 540.000 weitere, künstliche Trainingsmuster erzeugt. Mit diesem vergrößerten Datensatz konnte das Konvolutionsnetz auf einen Testfehler von 0,8% trainiert werden.

Als besonders effektiv zum Erzeugen größerer Trainingsmengen hat sich das Verfahren der elastischen Verzerrungen erwiesen. Konvolutionsnetze, die mit solchen künstlich vergrößerten Trainingsmengen trainiert wurden, konnten die bis heute besten Ergebnisse auf der Testmenge vorweisen. Trotz einer relativ simplen Netzarchitektur berichten Simard *et*

[2] `http://yann.lecun.com/exdb/mnist/`

Algorithmus	Distortionen	Fehler	Referenz
K-NN, euklidisch (L2)	keine	5.00%	[LBBH98]
2-schichtiges MLP (28×28-1000-10)	keine	4.50%	
2-schichtiges MLP (28×28-1000-10)	affine	3.80%	
40 PCA + quadratischer Klassifizierer	keine	3.30%	
3-schichtiges MLP (28×28-500-150-10)	keine	2.95%	
3-schichtiges MLP (28×28-500-150-10)	affine	2.45%	
LeNet-1	keine	1.70%	
LeNet-4	keine	1.10%	
SVM	entzerrt	1.10%	
LeNet-5	keine	0.95%	
LeNet-5	affine	0.80%	
Konvolutionsnetz + SVM	keine	0.83%	[LSB07]
Konvolutionsnetz + SVM	affine	0.54%	
Einfaches Konvolutionsnetz mit Cross-Entropy	affine	0.60%	[SSP03]
Einfaches Konvolutionsnetz mit Cross-Entropy	elastische	**0.40%**	
Großes Konvolutionsnetz, vortrainiert	keine	0.60%	[RPCL06]
Großes Konvolutionsnetz, vortrainiert	elastische	**0.39%**	

Tabelle 2.2: Ergebnisse einiger ausgewählter Lernverfahren zur Erkennung handschriftlicher Ziffern. Mit einer durch Distortionen künstlich vergrößerten Trainingsmenge lässt sich der Fehler auf der Testmenge teils deutlich reduzieren.

Abbildung 2.8: Einige Beispiele des NORB-Datensatzes. Die obere Zeile enthält Bilder der Trainingsmenge, die untere Bilder der Testmenge.

al. in [SSP03] von 0,40% Testfehler. Mit einem etwas größeren Netz und insbesondere durch unüberwachtes Vortrainieren der untersten Konvolutionsschichten erreichen Ranzato *et al.* [RPCL06] sogar 0,39%.

Die Erkennung handschriftlicher Ziffern stellt nach heutigem Stand der Technik für Verfahren der Mustererkennung kaum mehr eine Herausforderung dar. Ähnlich große Datenbanken mit komplexeren Mustern und gut markierten Objekten sind derzeit jedoch noch nicht verfügbar.

2.3.2 NORB-Datensatz

Der NORB-Datensatz [LHB04] wurde entworfen, um Klassifikatoren vergleichbar zu machen, die nur anhand der Form Objekte erkennen.

Die Bilder des Datensatzes (siehe Abbildung 2.8) zeigen je 10 verschiedene Spielzeugfiguren aus den fünf Kategorien *animal*, *human*, *airplane*, *truck* und *car*. Diese Figuren wurden weiß angestrichen und vor neutralem Hintergrund aufgenommen, damit weder Farbe noch

Textur als Klassifizierungsmerkmale dienen können. Die einzig zuverlässige Information ist somit die Form des Objekts. Bei der Generierung der Bilder wurden unter anderem Blickwinkel, Skalierung, Beleuchtung und Rotation variabel gewählt.

Die Objekte wurden auf einem Drehteller platziert und mit zwei in festem Abstand von 7,5 cm entfernten Kameras aufgenommen. Die so entstandenen Stereobildpaare wurden in Graustufenbilder konvertiert und liegen in einer Auflösung von jeweils 96×96 Pixeln vor.

Für die Trainingsmenge wurden fünf verschiedene Objekte jeder Kategorie aus neun Höhenlagen, mit 18 Drehwinkeln und mit sechs verschiedenen Beleuchtungsquellen aufgenommen. Der Trainingsdatensatz besteht somit aus 972 Bildpaaren pro Objekt, also aus insgesamt 24.300 Bildpaaren. Für den gleichgroßen Testdatensatz wurden fünf andere Objekte jeder Kategorie gewählt.

Die besten Erkennungsraten auf dieser Datenmenge erzielen Konvolutionsnetze (6,6% Testfehler) und Support Vector Machines (12,6% Testfehler). Aus dem hier beschriebenen Datensatz wurden synthetisch weitere, teils schwierigere Datensätze generiert, indem unter anderem der Hintergrund durch zufällig ausgewählte Texturen ersetzt wurde.

2.3.3 PASCAL VOC Challenge 2008

Das europäische Forschungsprojekt PASCAL (*Pattern Analysis, Statistical Modelling and Computational Learning*) zur Förderung der Kooperationen in den Bereichen Maschinellen Lernens, Statistik und Optimierung schreibt alljährlich einen Wettbewerb zum Thema Objekterkennung aus. Dazu wird ein Datensatz mit natürlichen Bildern veröffentlicht, in denen Objekte bestimmter Kategorien markiert sind.

Auf zwei Teilaufgaben der aktuellen *Visual Object Classes Challenge 2008* [EVGW+] wird in dieser Arbeit eingegangen: die Objektklassifizierung und die Objektlokalisierung. In beiden Wettbewerben stehen zunächst Trainingsbeispiele für überwachte Lernverfahren zur Verfügung, dessen Ergebnisse sich anschließend auf einer zunächst unbekannten Testmenge vergleichen lassen.

Der Datensatz

Insgesamt 4.340 Bilder mit 10.363 markierten Objekten werden zur Objektklassifizierung und -lokalisierung bereitgestellt. Davon sind 2.113 Bilder als Trainings- und 2.227 Bilder als Validierungsmenge designiert, trotzdem dürfen aber beide beliebig verwendet werden. Die markierten Objekte sind jeweils einer der 20 verschiedenen in Tabelle 2.3 angegebenen Objektkategorien zugeordnet. Die Testmenge mit 4.133 weiteren Bildern weist etwa die gleiche Klassenverteilung auf. Sie dient im Rahmen des Wettbewerbs zum Vergleich der Erkennungsraten verschiedener Verfahren, daher sind die Annotationen zu den Bildern nicht öffentlich verfügbar.

Der Datensatz wurde erstellt, indem zunächst von der Webseite Flickr[3]anhand fester Suchbegriffe eine große Anzahl natürlicher Fotos heruntergeladen wurde. Nach fest vorgegebenen Regeln wurden daraufhin manuell die Objekte in den Bildern markiert und einer eindeutigen Kategorie zugeordnet. Für jedes Objekt ist zum einen das achsparallele umgebende Rechteck (engl. *bounding box*) angegeben, zum anderen sind verschiedene Hinweise vorhanden. So wird angegeben, aus welcher Sicht das Objekt zu sehen ist (*frontal*, *rear*, *left* oder *right*), ob das Objekt über die Bildgrenzen hinaus geht (*truncated*), ob es durch einen anderen Gegenstand verdeckt ist (*occluded*) oder ob es selbst für menschliche Betrachter nur sehr schwer

[3] `http://www.flickr.com`

Klasse	Vorkommen		Klasse	Vorkommen
Aeroplane	316 (3,0%)		Diningtable	110 (1,1%)
Bicycle	269 (2,6%)		Dog	477 (4,6%)
Bird	476 (4,6%)		Horse	285 (2,8%)
Boat	336 (3,2%)		Motorbike	272 (2,6%)
Bottle	457 (4,4%)		Person	4168 (40,2%)
Bus	129 (1,2%)		Pottedplant	361 (3,5%)
Car	840 (8,1%)		Sheep	145 (1,4%)
Cat	378 (3,6%)		Sofa	151 (1,5%)
Chair	623 (6,0%)		Train	166 (1,6%)
Cow	130 (1,3%)		Tvmonitor	274 (2,6%)

Tabelle 2.3: Klassenverteilung der 10.363 Objekte in der kombinierten Trainings- und Validierungsmenge

und durch Einbeziehung eines größeren Kontexts identifizierbar ist (*difficult*). Die maximale Auflösung der Bilder beträgt in der größten Dimension ca. 500 Pixel.

Die A-priori-Verteilung verschiedener Objektklassen fällt sehr unterschiedlich aus. Am häufigsten tritt die Klasse *person* mit 4.168 Beispielen in den Bildern auf. Alle anderen Klassen kommen zwischen 110 mal (*diningtable*) und 840 mal (*car*) vor. Zudem können sich die Objekte innerhalb der relativ grob gefassten Kategorien sehr stark unterscheiden, wie auch anhand der Beispielbilder in Abbildung 2.9 zu sehen ist.

Objektklassifizierung

Die Aufgabe bei der Objektklassifizierung besteht darin festzustellen, welche Objekte in einem Bild vorkommen. Für jede der 20 Objektklassen soll die Wahrscheinlichkeit angegeben werden, dass eine Instanz dieser Kategorie abgebildet ist. Hierbei ist es unerheblich, an welchen Positionen im Bild ein solches Objekt vorkommt und wie häufig es auftritt.

Zur Bewertung eines Klassifikators werden für jede Klasse die Trefferquote und die Genauigkeit in Form einer Precision/Recall-Kurve herangezogen (siehe Abb. 2.10). Aus dieser Kurve lässt sich die durchschnittliche Genauigkeit (engl. *average precision, AP*) des Klassifikators ermitteln, die nur dann hoch ausfällt, wenn sowohl die Trefferquote als auch die Genauigkeit hoch sind.

Objektlokalisierung

Bei der Objektlokalisierung sollen außer der Klassenzugehörigkeit noch die Koordinaten jedes klassifizierten Objekts durch Angabe eines achsparallelen Rechtecks ermittelt werden. Außer der durchschnittlichen Genauigkeit wird als zusätzliches Gütemaß daher die Überschneidungsfläche A_o des detektierten Rechtecks mit dem vorgegebenen Rechteck verwendet (siehe Abb. 2.11), die sich aus

$$A_o = \frac{|B_p \cap B_{gt}|}{|B_p \cup B_{gt}|} \tag{2.19}$$

berechnet. Dabei bezeichnet B_p das vorhergesagte Rechteck (*predicted*) und B_{gt} das tatsächliche das Objekt umschließende Rechteck (*ground truth*). Wenn diese Schnittmenge 50% überschreitet, gilt das Objekt als korrekt erkannt.

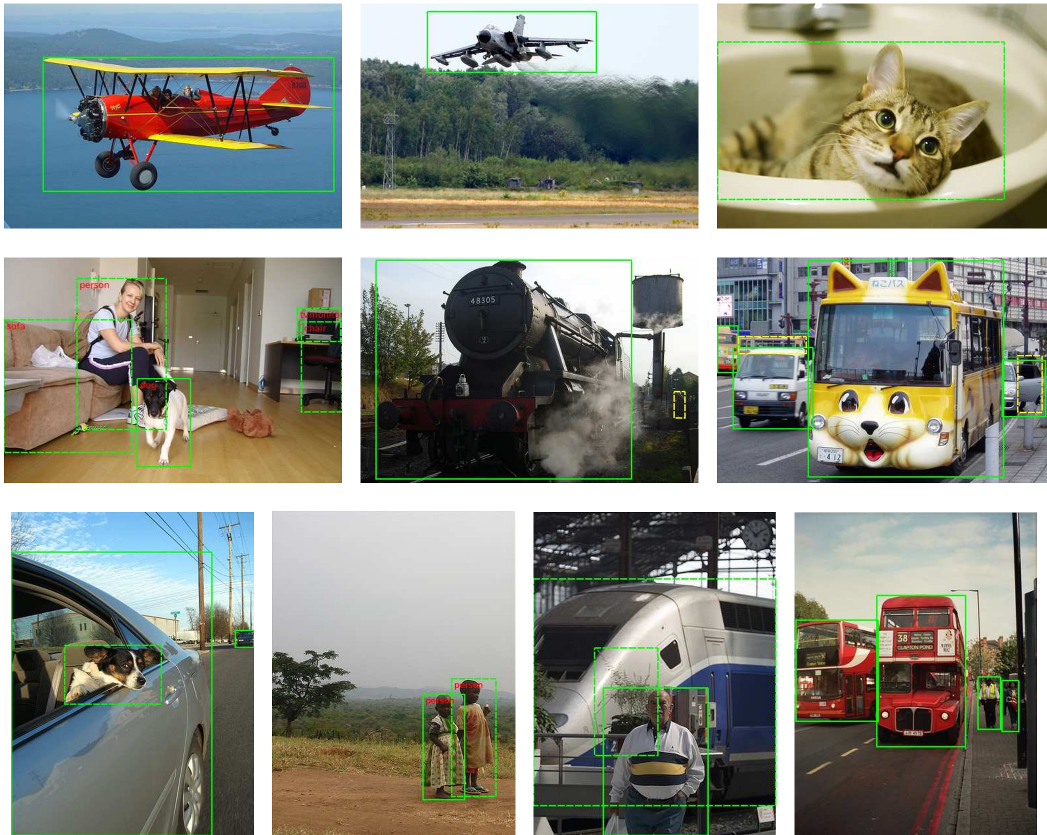


Abbildung 2.9: Einige Beispielbilder aus dem Datensatz der PASCAL VOC 2008. (aus [EVGW+], bearbeitet)

Vor- und Nachteile

Aus mehreren Gründen wurde der Datensatz der PASCAL Challenge für diese Arbeit ausgewählt. Zum einen lässt sich ein mit diesem Datensatz trainierter Klassifikator sehr gut mit anderen dem aktuellen Stand der Forschung entsprechenden Verfahren vergleichen. Durch ein einheitliches Gütemaß lässt sich der Unterschied zwischen verschiedenen Verfahren leicht quantifizieren.

Zum anderen sind die Bilder genauso groß oder nur unwesentlich kleiner als in praktischen Anwendungen. Die in den Bildern dargestellten Szenen entstammen allesamt typischen alltäglichen Umgebungen und beschränken sich nicht auf ein spezielles Szenario.

Im Gegensatz zu anderen Datensätze, wo einige Objekte sehr präzise und andere nur sehr grob umrandet sind, sind die Annotationen in der PASCAL Datenbank sehr konsistent. Diese Homogenität der Markierungen ist sicherlich auch dem Umstand zu verdanken, dass zu den Objekten nur *bounding boxes* und keine umrandenden Polygone gegeben sind.

Die Anzahl der verschiedenen Objektklassen ist hinreichend groß, um auch komplizierte Klassifikatoren zu testen. Wie die Ergebnisse auf der MNIST-Datenbank gezeigt haben, ist auch die Quantität der Trainingsbeispiele ein wichtiger Faktor für den Lernerfolg eines Verfahrens. Die gut 4.000 Bilder im PASCAL VOC 2008 sind daher unter Umständen nicht

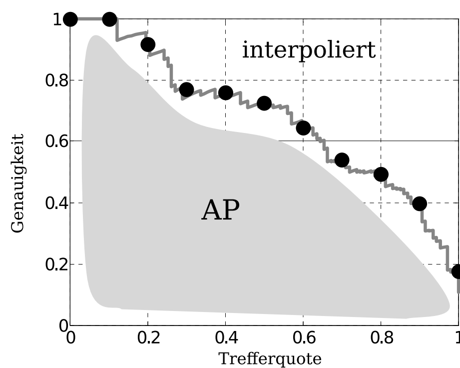


Abbildung 2.10: Precision/Recall-Kurve. Die durchschnittliche Genauigkeit (AP) errechnet sich aus der Fläche unter der Kurve. (aus [EVGW+], bearbeitet)

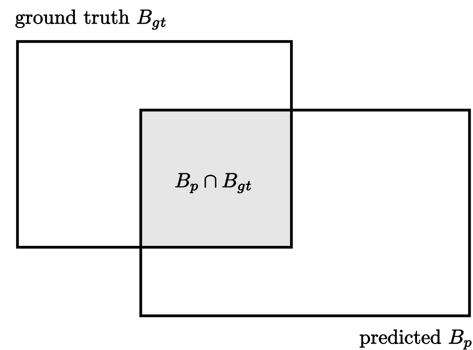


Abbildung 2.11: Die Überschneidungsfläche zwischen tatsächlichem Objekt und dem vorhergesagten Rechteck wird zur Bewertung der Objektlokalisierung verwendet. (aus [EVGW+], bearbeitet)

ausreichend, um einen gut generalisierenden Klassifikator zu trainieren. Sorgfältig berücksichtigt werden muss zudem die starke Dominanz der Objektklasse *person*.

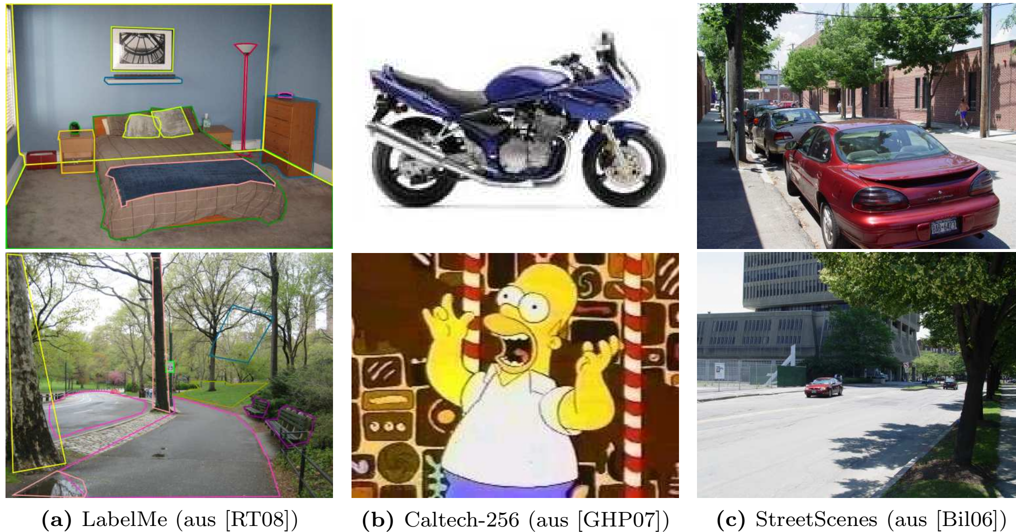
2.3.4 Andere Datensätze

Mehrere Datensätze zur Objekterkennung sind frei verfügbar. Dabei konzentrieren sich viele nur auf sehr spezielle Kategorien wie z.B. Gesichter, Personen, Autos oder Fußgänger. Einige Datensätze, die ein breiteres Spektrum an Objektklassen abdecken, werden hier vorgestellt. Typische Beispielbilder aus diesen Datensätzen sind in Abbildung 2.12 zu sehen.

Caltech-101 und Caltech-256

Häufig werden Verfahren zur Objekterkennung nur auf ein konkretes Problem zugeschnitten und können auf andere Daten dann nicht übertragen werden. Der *Caltech-101* Datensatz wurde daher 2003 am *California Institute of Technology* als unabhängige Datenmenge zusammengestellt, anhand derer sich unterschiedliche Verfahren messen können. Jedes der 9.144 Bilder ist genau einer von 101 Objektkategorien oder einer speziellen Hintergrundkategorie zugeordnet. Zusätzlich ist zu jedem Objekt seine polygonale Umrandung angegeben. Kritisiert wird an diesem Datensatz, dass er zum einen für einige Kategorien nur sehr wenige Bilder enthält und zum anderen zu wenige Kategorien umfasst. Daher wurde er 2007 durch den umfangreicheren Caltech-256 Datensatz abgelöst.

Die insgesamt 30.608 Bilder im Datensatz *Caltech-256* [GHP07] werden 256 Objektkategorien und einer Hintergrundkategorie zugeordnet. Sie verteilen sich auf mindestens 80 Bilder pro Kategorie; somit sind die Kategorien also deutlich größer als im Vorgängerdatensatz. Mittels Anfragen an Bildsuchmaschinen im Internet wurden die Bilder zusammengetragen, um anschließend manuell zu bewerten, ob sie auf die gesuchte Kategorie zutreffen. Durch diese Vorgehensweise handelt es sich bei den Bildern häufig nicht um natürliche Szenen, sondern z.B. um freigestellte Produktfotos oder Zeichnungen. Da die Objekte meistens im Bild zentriert sind und somit deren Position bekannt ist, eignet sich der Datensatz nicht zur Objektlokalisierung. In die Datenbank wurden zudem bewusst einige nur schwer unterscheidbare Objektkategorien – zum Beispiel „Kröte“ und „Frosch“ – aufgenommen.



(a) LabelMe (aus [RT08]) (b) Caltech-256 (aus [GHP07]) (c) StreetScenes (aus [Bil06])

Abbildung 2.12: Beispielbilder aus unterschiedlichen Datensätzen zur Mustererkennung (LabelMe mit Annotationen)

LabelMe

Einen anderen Ansatz verfolgen die Autoren der LabelMe-Datenbank [RT08], die vom *MIT Computer Science and Artificial Intelligence Laboratory* initiiert wurde. Ziel dieses Projekts ist es, für Forschungsvorhaben in den Bereichen Objekterkennung und Computergrafik einen dynamischen und möglichst umfangreichen Datensatz bereit zu stellen. Der Schwerpunkt liegt dabei nicht darauf, verschiedene Verfahren vergleichbar zu machen, sondern die Forschung im Bereich des maschinellen Sehens voranzutreiben.

Sämtliche Bilder sind über die Webseite des Projekts[4] abrufbar, und die Annotationen sowie Markierungen der Objekte können von jedem Benutzer durch ein Web-Interface bearbeitet werden. Mit Stand März 2009 umfasst der ständig anwachsende Datenbestand 176.361 Bilder unterschiedlicher Auflösung, die größtenteils von den Autoren selbst aufgenommen wurden. Die Objekte werden durch eine polygonale Umrandung markiert und mit einer frei wählbaren Kategorie beschriftet – strenge Richtlinien für die Annotationen gibt es nicht. Bislang wurden in 51.845 dieser Bilder insgesamt 339.840 Objekte aus 4.418 Kategorien markiert.

Kein anderer frei verfügbarer Datensatz verfügt über derart viele Bilder und markierte Objekte. Dass im LabelMe-Datensatz deutlich mehr Kategorien zur Verfügung stehen als in anderen Datensätzen birgt sowohl Vor- als auch Nachteile. Viele der Kategorien überlappen sich, zum Beispiel könnte ein und dasselbe Objekt als „Person“, als „Fußgänger“ oder als „Mann“ markiert sein. Ohne eine hierarchische Ordnung der Kategorien sind viele der Objekte daher für einen Großteil der Lernverfahren unbrauchbar. Aufgrund fehlender Qualitätskontrolle ist der Datensatz gleich in mehrerlei Hinsicht inhomogen. Nicht nur die Objektbeschreibungen variieren stark in ihrer Genauigkeit, sondern auch die Komplexität der markierenden Polygone ist sehr unterschiedlich. Zudem ist nur ein Bruchteil der Objekte überhaupt markiert, was angesichts der Menge an Bildern auch nicht verwundert.

[4] `http://labelme.csail.mit.edu/`

CBCL StreetScenes

Der mit LabelMe kompatible Datensatz *CBCL StreetScenes* wurde am *Center for Biological and Computational Learning* des MIT im Rahmen einer Doktorarbeit [Bil06] erstellt. Enthalten sind nicht nur Bilder und deren Annotationen, sondern auch Programmcode, um verschiedene Daten zu extrahieren und um Lernverfahren zu evaluieren. Der Datensatz wurde mit dem Ziel entworfen, Systeme zur Objekterkennung zu trainieren, zu testen und deren Erkennungsleistung zu vergleichen. Man hat sich bei dem Entwurf ausschließlich auf Straßenansichten konzentriert, weil in solchen Umgebungen bestimmte Objektkategorien mit einer gewissen Zuverlässigkeit vorkommen. Weitere Vorteile sind, dass solche Aufnahmen häufig sehr ähnliche Strukturen aufweisen und dass ein direkter realer Bezug – etwa mit Anwendungen in den Bereichen Überwachung und Verkehrssicherheit – besteht.

Bei der Erstellung dieses Datensatzes wurde besonders darauf geachtet, dass sowohl die Bilder als auch die Markierungen der Objekte von einheitlicher Qualität sind. Er enthält 3.547 Bilder mit einer Auflösung von 1280×960 Pixeln, die alle tagsüber mit dem gleichen Kameramodell und den selben Einstellungen aufgenommen wurden. Die insgesamt mehr als 25.000 Objekte wurden anhand ihres Umrisses markiert und neun relativ eindeutigen Kategorien (z.B. *Fußgänger*, *Straße*, *Himmel*) zugewiesen. Nur hinreichend große und gut sichtbare Objekte sind markiert. Durch die Homogenität der Szenen und die weit gefassten Kategorien gelingt es, einen Großteil der Pixel in den Bildern mit Annotationen abzudecken.

Ein Nachteil der Datenbank ist, dass die Qualität der polygonalen Umrandungen variiert, je nachdem welche Person das jeweilige Bild annotiert hat. Es kann auch problematisch sein, dass stark verdeckte Objekte – wie beispielsweise eine Straße mit hohem Verkehrsaufkommen – nicht markiert sind.

2.4 Parallele Hardware

Parallelrechner können Operationen auf mehreren verteilten Prozessoren gleichzeitig ausführen. Um diese Rechenleistung effizient ausnutzen zu können, muss sich ein großes Problem in sehr viele kleinere Probleme unterteilen lassen, die gleichzeitig gelöst werden können. Besonders gut kann man Probleme parallelisieren, die eine hohe *arithmetische Dichte* aufweisen, bei denen also das Verhältnis zwischen der Anzahl an arithmetischen Operationen und an Speicheroperationen groß ist.

Verschiedene Möglichkeiten, neuronale Netze parallel zu implementieren, werden in diesem Abschnitt erläutert. Danach werden moderne Grafikkarten als leistungsstarke und kostengünstige Hardware für parallele Berechnungen vorgestellt. Die von Nvidia für ihre GeForce-Serie entworfene und in dieser Arbeit zur Beschleunigung der Objekterkennung eingesetzte CUDA-Architektur wird im Detail besprochen.

2.4.1 Simulation neuronaler Netze auf paralleler Hardware

Biologisches Vorbild künstlicher neuronaler Netze sind die parallel arbeitenden Nervenzellen im Gehirn. Aufgrund dieser inhärenten Parallelität neuronaler Netze eignet sich nach verbreiteter Meinung insbesondere parallele Hardware zu deren Implementierung. Denn genauso wie ein neuronales Netz über eine große Anzahl von miteinander verknüpften einfachen parallel arbeitenden Neuronen verfügt, so besteht ein Parallelrechner aus Tausenden von einfachen Prozessoren, die untereinander kommunizieren können. Ein SIMD-Rechner (*single instruction, multiple data*) vereint meist 2^{12} bis 2^{16} Prozessoren, welche die gleichen Instruktionen synchron auf unterschiedlichen Daten ausführen.

Die naheliegende Idee, neuronale Netze auf paralleler Hardware zu implementieren, wurde daher vor allem seit den 1980er Jahren verfolgt und führte zur Entwicklung von Neurochips und Neurocomputern mit massiv parallelen Spezialprozessoren. Aber auch universell einsetzbare SIMD-Parallelrechner wurden erfolgreich zur Beschleunigung künstlicher neuronaler Netze eingesetzt. Um vorwärtsgerichtete neuronale Netze auf solche Architekturen effizient zu übertragen, unterscheidet Zell [Zel94] verschiedene Arten der Parallelität:

- **Trainingsmusterparallelität**: Jeder Prozessor speichert ein einziges Trainingsmuster sowie eine lokale Kopie des gesamten Netzes. Nachdem auf allen Prozessoren simultan die lokalen Gewichtsänderungen für unterschiedliche Muster berechnet wurden, müssen diese zentral aufsummiert werden.
- **Kantenparallelität**: Alle Kanten zwischen zwei Schichten werden parallel berechnet, somit wird auf jeden Prozessor genau eine Kante abgebildet.
- **Neuronenparallelität**: Die Parallelisierung erfolgt über Neuronen einer Schicht. Jeder Prozessor berechnet die Aktivität eines einzigen Neurons. Dabei können die eingehenden Kanten sequentiell abgearbeitet werden.
- **Ebenenparallelität**: Die Aktivierungen mehrerer Schichten werden parallel berechnet. Diese Parallelisierung ist vor allem bei Netzen mit vielen Schichten sinnvoll.
- **Phasenparallelität**: Für verschiedene Trainingsmuster werden sowohl Vorwärts- als auch Rückwärtspropagierung gleichzeitig durchgeführt. Zusammen mit der Ebenenparallelisierung kann damit eine Art Pipelining der Propagierungen im Netz durchgeführt werden.

Die oben genannten Formen der Parallelität sind sortiert nach höchstmöglichem Grad der Parallelisierung, wobei die Trainingsmusterparallelität potentiell am effizientesten ist. Kombinationen dieser Ansätze liefern noch höhere Grade an Parallelität.

2.4.2 Grafikkarten zur Beschleunigung von Berechnungen

Viele Bereiche der wissenschaftliche Forschung verdanken ihre Fortschritte den in den letzten Jahrzehnten immer weiter steigenden Taktfrequenzen der Hauptprozessoren (engl. *central processing unit, CPU*). Seit die Hersteller seit ca. 2003 an die physikalischen Grenzen der Fertigung gestossen sind, fallen die Geschwindigkeitssprünge deutlich kleiner aus.

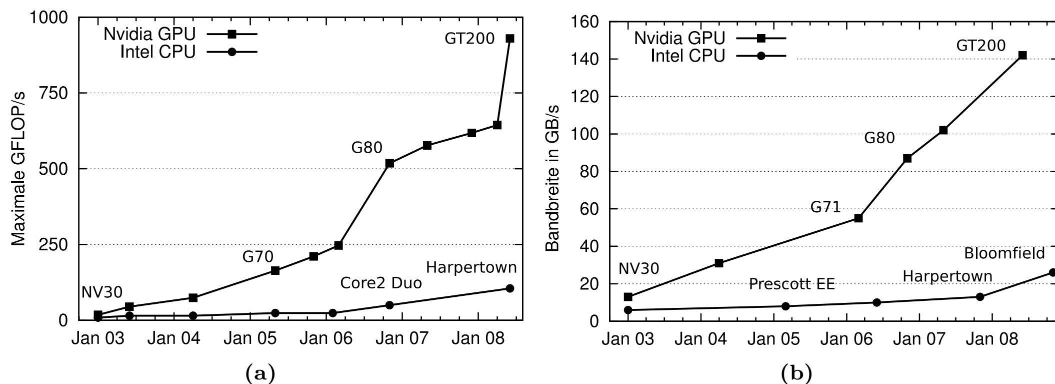


Abbildung 2.13: Vergleich zwischen Nvidia Grafikkarten und Intel Prozessoren. (a) Maximale Gleitkommaberechnungen pro Sekunde. (b) Speicherbandbreite in Gigabyte pro Sekunde. (aus [Nvi08c], bearbeitet)

Sowohl was die Rechenleistung angeht als auch bezüglich der Speicherbandbreite machen Mehrkernprozessoren und insbesondere Grafikprozessoren (engl. *graphics processing unit, GPU*) weiterhin enorme Fortschritte (siehe Abb. 2.13). Die Gründe für die große Diskrepanz zwischen CPU und GPU liegen in deren völlig unterschiedlichen Architekturen. Traditionell führte eine CPU immer nur eine Instruktion nach der anderen aus, seit dem *Pentium* dann auch mehrere unterschiedliche Instruktionen parallel. Dagegen führen GPUs ein und denselben Befehlssatz gleichzeitig auf mehreren Daten aus; man spricht von *Datenparallelisierung.* Einem GPU stehen somit mehr Transistoren für die Recheneinheiten (engl. *arithmetic logic unit, ALU*) zur Verfügung, da weniger Kapazität für Kontrollmechanismen und zum Zwischenspeichern (engl. *caching*) der Daten benötigt wird (siehe Abb. 2.14). Ein weiterer Grund liegt darin, dass Grafikkarten sehr effizient auf den Speicher zugreifen. Zur Berechnung eines Bildes werden stets benachbarte Texel gelesen und anschließend benachbarte Pixel geschrieben, so dass kaum caching nötig ist und nahezu die maximale theoretische Speicherbandbreite ausgenutzt werden kann.

Aufgrund der starken Spezialisierung der Hardware haben sich universelle Berechnungen auf Grafikkarten (engl. *general-purpose computing on graphics processing units*, GPGPU) nur langsam durchgesetzt. Ein wichtiger Fortschritt war dabei die 2003 erschienene *NV30* von Nvidia, die erstmals Berechnungen mit 32 Bit Gleitkommazahlen ermöglichte. Durch deren programmierbare Pixel- und Vertexshader waren zwar sogar Matrixberechnungen möglich, allerdings konnte auf diese Recheneinheiten nur mittels Grafik-Schnittstellen (*DirectX*, *OpenGL*) zugegriffen werden.

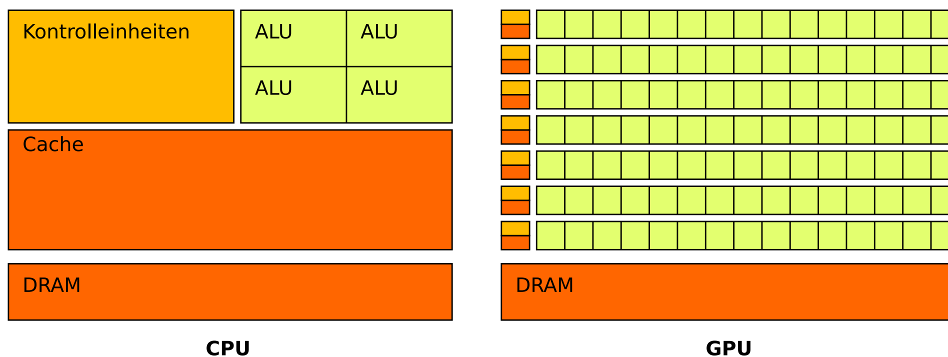


Abbildung 2.14: Auf der GPU werden weniger Transistoren zur Steuerung des Kontrollflusses verwendet und mehr für arithmetische Berechnungen. (aus [Nvi08c], bearbeitet)

An der Stanford University wurde daher der Compiler BrookGPU entwickelt, der ein Bindeglied zwischen der Programmiersprache C und den Grafikkartentreibern darstellte. Erst dadurch wurden die Grafikkartenhersteller Nvidia und AMD auf die Möglichkeit aufmerksam, mit allgemeinen Berechnungen auf GPUs neue Märkte zu erschließen. Seitdem wurden GPUs flexibler, so dass auch Berechnungen mit ganzzahligen und doppelwertigen Daten auf Grafikkarten sowie beliebige Schreib- und Lesezugriffe auf die Speicherbereiche möglich sind. Da Grafikkarten ebenso wie Supercomputer auf der SPMD-Architektur (*single program, multiple data*) aufbauen, lassen sich viele bestehende wissenschaftliche Programme auf GPUs übertragen. Auch bei Heimcomputern ist abzusehen, dass sich Systeme mit vielen Prozessorkernen in Zukunft durchsetzen werden und das Paradigma der Parallelprogrammierung an Bedeutung gewinnen wird.

Derzeit stellen mit AMD und Nvidia die beiden größten Grafikkartenhersteller jeweils ihre eigenen Entwicklungsumgebungen für GPGPU-Anwendungen zur Verfügung. Als offener Standard soll in Zukunft zudem *OpenCL*[5] eine herstellerübergreifende Programmierplattform für CPUs, GPUs und DSPs bieten. Mit Abstand am weitesten fortgeschritten und breitflächig unterstützt wird aber derzeit noch die auf Karten von Nvidia beschränkte Entwicklungsumgebung CUDA.

Im folgenden wird zunächst das Softwaremodell der Programmierschnittstelle CUDA vorgestellt und danach die Hardwarearchitektur von Nvidias Grafikkarten erläutert. Diese strikte Trennung macht in Anbetracht dessen Sinn, dass das Programmiermodell auch auf andere Hardware übertragbar ist (z.B. auf Mehrkernprozessoren wie von Stratton *et al.* [SSH08] gezeigt).

2.4.3 Das CUDA Programmiermodell

Mit der GeForce-8-Serie führte Nvidia im November 2006 eine Programmierschnittstelle ein, die universelle Berechnungen auf Grafikkarten ermöglicht. Diese CUDA (*Compute Unified Device Architecture*) genannte Architektur [Nvi08c] ist darauf ausgelegt, auf unterschiedliche Anzahlen von Prozessorkernen zu skalieren. Der Zugriff auf die Hardware erfolgt mittels „C for CUDA“, einer Erweiterung der für viele Entwickler vertrauten Programmiersprache C. Im CUDA-Modell wird die Grafikkarte (*Device*) als physikalisch getrennter Co-Prozessor zur CPU (*Host*) behandelt.

[5] `http://www.khronos.org/opencl/`

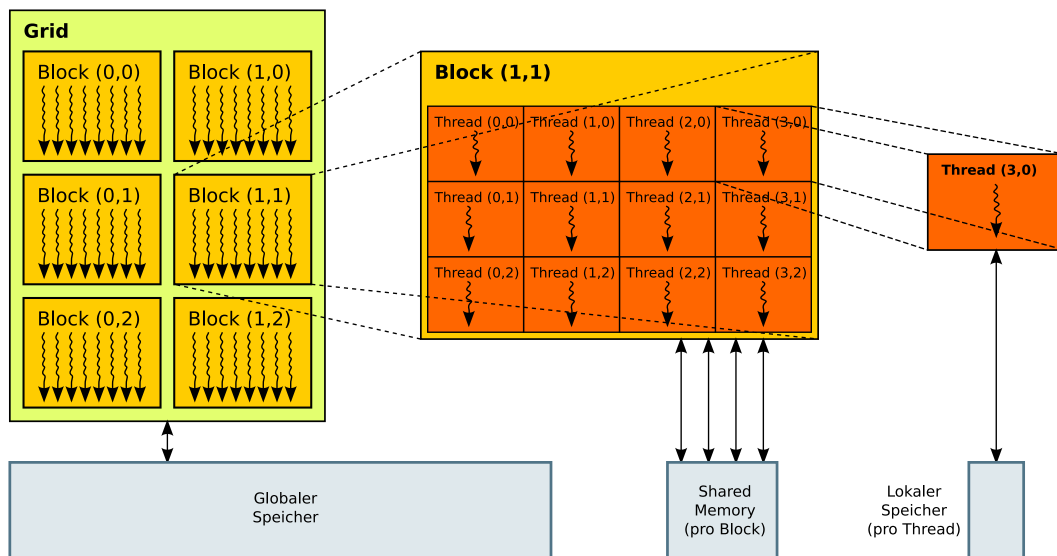


Abbildung 2.15: Threadhierarchie und Speicherhierarchie im CUDA Programmiermodell (nach [Nvi08c])

Kernels und Threadhierarchie

Berechnungen auf der Grafikkarte werden durch das Ausführen von sogenannten *Kernels* initiiert. Dabei handelt es sich um spezielle C-Funktionen, die beim Aufruf N-mal parallel von N verschiedenen *Threads* ausgeführt werden. Jeder Thread erhält dazu eine eindeutige Identifikationsnummer, die im Kernel über die Variable `threadIdx` abrufbar ist. Dadurch ist es z.B. möglich, dass ein Kernel einen N-elementigen Vektor bearbeitet und jeder Thread genau ein Element dieses Vektors berechnet.

Um Zugriffe auf Vektoren, Matrizen oder Tensoren zu vereinfachen, werden Threads in ein-, zwei- oder dreidimensionalen Blöcken angeordnet (siehe Abb. 2.15). Innerhalb eines Blocks können die Threads über gemeinsame Speicherbereiche miteinander kommunizieren, außerdem kann mittels Synchronisationspunkten garantiert werden, dass alle Threads an derselben Stelle der Ausführungsreihenfolge angekommen sind. Um schnelle Speicherzugriffe und kurze Synchronisationszeiten zu garantieren, werden alle Threads eines Blocks parallel auf demselben Prozessorkern ausgeführt. Eine kompakte Bündelung der Threads wird erreicht, indem die Größe eines Blocks auf 512 Threads beschränkt wird.

Ein Kernel kann jedoch von beliebig vielen Blöcken ausgeführt werden, die wiederum in einem ein- oder zweidimensionalen *Grid* angeordnet sind. Vorraussetzung ist, dass die verschiedenen Blöcke in beliebiger Reihenfolge, sequentiell oder auch simultan, ausführbar sind. Das ermöglicht es dem Treiber, die Ausführung der Blöcke an die Systemressourcen anzupassen. Wenn viele Prozessorkerne zur Verfügung stehen, können alle Blöcke parallel ausgeführt werden, ansonsten müssen zumindest einige Blöcke sequentiell abgearbeitet werden. Somit lässt sich ein Programm nicht nur auf verschiedene aktuelle Systeme skalieren, sondern auch auf zukünftige Generationen mit noch mehr Prozessorkernen.

Diese Abstraktionen führen dazu, dass Algorithmen für CUDA auf zweierlei Ebenen parallelisiert werden müssen. Ein Problem muss zunächst in grobe, unabhängig voneinander lösbare Aufgaben eingeteilt werden, die blockweise parallel berechnet werden. Innerhalb ei-

nes Blocks erfolgt dann eine feinere Unterteilung in Subprobleme, die kooperativ parallel gelöst werden.

Speicherhierarchie

Einem Thread stehen unterschiedliche, hierarchisch angeordnete Speicherbereiche zur Verfügung, die unter anderem auch dazu dienen, mit anderen Threads und mit dem Hauptprozessor zu kommunizieren.

Jeder einzelne Thread verfügt über einen kleinen lokalen Speicher mit geringer Latenz, der für andere Threads nicht sichtbar ist. Innerhalb eines Blocks können alle Threads auf einen größeren gemeinsamen Speicher (*Shared Memory*) zugreifen. Dieser ermöglicht die Kommunikation zwischen den Threads und hat eine ähnlich geringe Latenz wie CPU-interner L1-Cache. Sein Inhalt ist nach Ausführung des Blocks jedoch nicht mehr verfügbar.

Der größere globale Speicher (*Global Memory*) ist im Gegensatz dazu von allen Blöcken zugänglich und sogar zwischen unterschiedlichen Kernelaufrufen persistent. Für spezielle Zwecke stehen allen Blöcken zudem noch zwei weitere, nur lesbare Speicherbereiche zur Verfügung: der Texturspeicher (*Texture Memory*) und der Konstantenspeicher (*Constant Memory*).

Spracherweiterungen

Mittels einiger unkomplizierter Erweiterungen zur Programmiersprache C kann auf die Funktionen der GPU zugegriffen werden. Beim Kompilieren extrahiert der *nvcc*-Compilertreiber zunächst die für die CPU bestimmten Teile des Quellcodes und lässt sie vom Standardcompiler (z.B. *gcc* unter Unix) verarbeiten. Der für die GPU relevante Quelltext wird dagegen erst in Assembler-ähnlichen PTX-Code umgewandelt und anschließend in ein Binärformat kompiliert, das auf unterschiedlichen GPUs lauffähig ist. Einige wichtige Deklarationen und Schlüsselwörter werden im folgenden vorgestellt.

Kernels, also Funktionen, die auf der GPU ausgeführt werden, muss das Schlüsselwort `__global__` vorangestellt werden. Damit wird dem Compiler mitgeteilt, den Code für das Device zu kompilieren. Für Kernelfunktionen gelten einige Einschränkungen: Sie können keine variable Anzahl von Parametern erhalten und rekursive Aufrufe sind nicht zulässig.

Kernelaufrufe werden immer vom Host initiiert. Dazu müssen beim Funktionsaufruf in eckigen Klammern `<<< ... >>>` die Grid- und Blockgröße als zusätzliche Parameter übergeben werden.

Variablen können durch Angabe des Vermerks `__device__`, `__constant__` oder `__shared__` explizit in einem bestimmten Speicherbereich auf dem Device abgelegt werden.

Synchronisation der Threads eines Blocks erfolgt innerhalb einer Kernelfunktion durch den Aufruf des Befehls `__synchthreads()`.

2.4.4 Die CUDA Hardware-Architektur

Grafikkarten ab der GeForce-8-Generation ermöglichen es, CUDA-Programme auszuführen. Da sich die Spezifikationen verschiedener Karten stark unterscheiden, beschränkt sich der folgende Abschnitt auf das in dieser Arbeit verwendete Modell GeForce GTX 285. Die Spezifikationen dieser derzeit im Heimanwenderbereich leistungsfähigsten Karte werden in Tabelle 2.4 einigen anderen Modellen gegenübergestellt.

	8800 Ultra	**9800 GTX+**	**GTX 285**
Veröffentlichung:	05/2007	07/2008	01/2009
Multiprozessoren:	16	16	30
Prozessorkerne:	128	128	240
Globaler Speicher:	768 MB	512 MB	1024 MB
Konstantenspeicher:	64 KB	64 KB	64 KB
Shared Memory pro Block:	16 KB	16 KB	16 KB
Maximale Speicherbandbreite:	103,7 GB/s	70,4 GB/s	159,0 GB/s
Theoretische Geschwindigkeit:	576 GFLOPs	705 GFLOPs	1063 GFLOPs
Register per block:	8192	8192	16384
Chipsatz:	G80	G92b	G200b
Compute Capability:	1.0	1.1	1.3

Tabelle 2.4: Ausgewählte High-End-Modelle unterschiedlicher GeForce-Serien.

Multiprozessoren

Die GeForce GTX 285 verfügt über 30 Multiprozessoren. Jeder dieser Multiprozessoren besteht aus acht skalaren Prozessorkernen, zwei Berechnungseinheiten für nicht-algebraische Funktionen, einer Multithread-Anweisungseinheit und einem auf dem Chip integriertem Speicher (*shared memory*), siehe Abb. 2.16. Das CUDA-Programmiermodell wurde im Hinblick auf eine optimale Ausnutzung der Architektur aktueller Grafikkarten entwickelt. Beim Ausführen eines Kernels werden die Blöcke eines Grids auf die freien Multiprozessoren verteilt. Dabei werden alle Threads eines Blocks simultan auf diesem Multiprozessor ausgeführt. Sobald ein Block terminiert, werden weitere Blöcke auf dem freien Multiprozessor gestartet.

Die Kontrolle über Erstellung, Ausführung und Verwaltung simultaner Threads ist hardwareseitig implementiert. Vorteilhaft ist, dass diese intrinsische Synchronisierung der Threads nur eine Anweisung benötigt und dass sowohl bei der Threaderstellung als auch beim Scheduling keine Rechenkapazität der arithmetischen Einheiten beansprucht werden muss.

SIMT-Architektur

Damit Hunderte von Threads mehrere verschiedene Programme ausführen können, implementieren die Multiprozessoren eine von Nvidia als SIMT (*single-instruction, multiple-thread*) bezeichnete Architektur. Ein Multiprozessor bildet jeden Thread auf genau einen der acht Prozessorkerne ab, so dass jeder Thread unabhängig voneinander ausgeführt wird. Dabei verwaltet die SIMT-Einheit des Multiprozessors immer 32 aufeinanderfolgende Threads auf einmal, die zu einem sogenannten *Warp* zusammengefasst werden. Alle Threads eines Warps starten gleichzeitig, können jedoch unterschiedliche Ausführungspfade einschlagen. Da innerhalb eines Warps immer nur eine gemeinsame Instruktion gleichzeitig ausgeführt wird, kann die maximale Effizienz erreicht werden, wenn alle 32 Threads die selben Befehle ausführen. Bei datenabhängigen Verzweigungen muss jeder mögliche Pfad seriell ausgeführt werden.

Speicher

Wie in Abb. 2.16 illustriert, verfügt jeder Multiprozessor über mehrere, verschieden schnelle und große Speicherbereiche. Einem Multiprozessor stehen einige wenige, dafür aber sehr schnelle Register zur Verfügung, die einzelnen Threads zugewiesen werden.

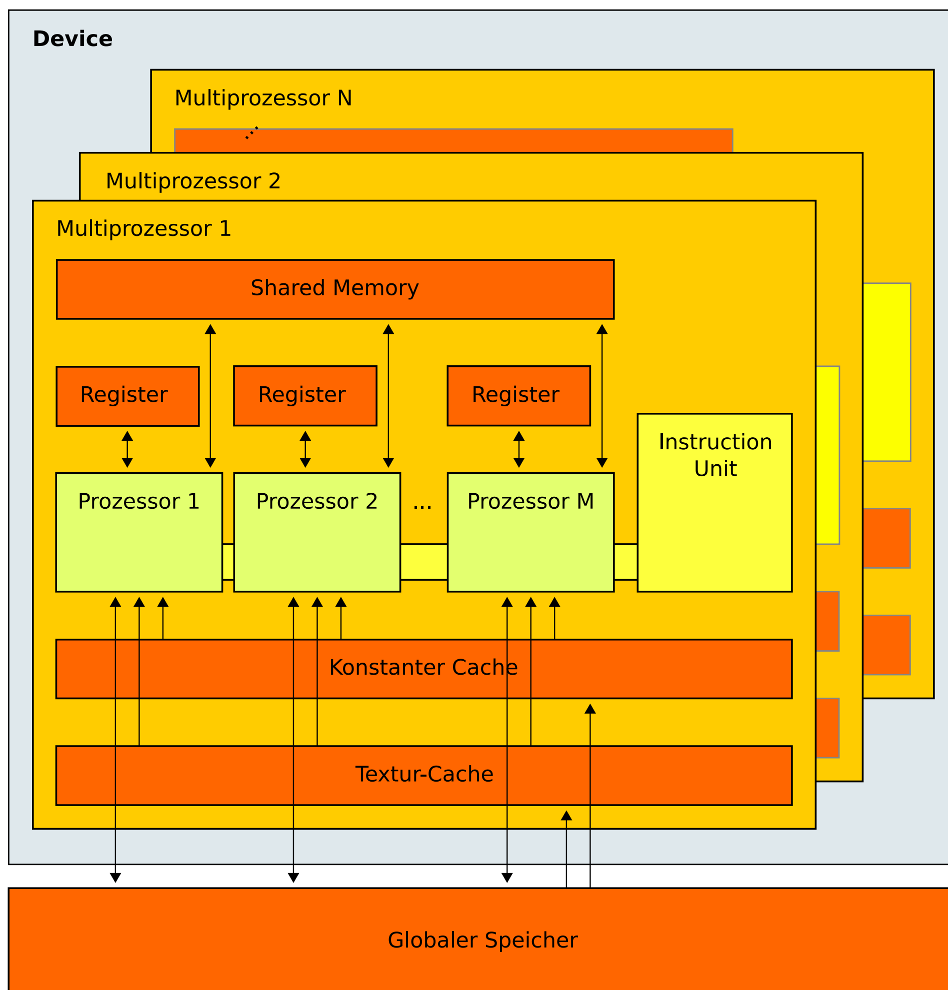


Abbildung 2.16: SIMT-Architektur: Mehrere Multiprozessoren mit auf dem Chip integriertem Shared Memory (aus [Nvi08c], bearbeitet)

Fast genauso schnell, aber etwas flexibler ist der 16 KB große *Shared Memory*, auf den alle Threads eines Blocks zugreifen können. Die größte Kapazität auf der Grafikkarte hat mit 1 GB der *Device Memory*, über den sämtliche Multiprozessoren verfügen können. Da dieser Speicher jedoch eine relativ hohe Latenz hat, verfügt jeder Multiprozessor über einen 64 KB großen *Konstantenspeicher*, auf den nur lesend zugegriffen werden kann. Eine Besonderheit stellt außerdem der *Texturspeicher* dar, der für zweidimensionale Zugriffe optimiert ist und dessen Adressierung über eine spezielle Textureinheit gesteuert wird. Vom *Host* kann lediglich auf den Device Memory zugegriffen werden, alle anderen Speicherbereiche sind nur auf der GPU verfügbar.

Die Berechnungsgeschwindigkeit wird für viele Algorithmen durch den zur Verfügung stehenden Speicher begrenzt. Denn die Anzahl der Blöcke, die ein Multiprozessor gleichzeitig verarbeiten kann, hängt vor allem von zwei Faktoren ab: Zum einen davon, wie viele Register ein Thread benötigt und zum anderen von der Größe des Shared Memorys, den ein Block

braucht. Denn sowohl Register, als auch Shared Memory müssen zwischen allen Threads der aktiven Blöcke aufgeteilt werden.

2.4.5 Einschränkungen von CUDA und Optimierungsmöglichkeiten

Um eine maximale Auslastung der Hardware in Bezug auf Rechenleistung und Speichertransferraten zu erreichen, müssen beim Abbilden eines Algorithmus auf die CUDA-Architektur einige Einschränkungen in Betracht gezogen werden. Mit neueren Hardwarerevisionen (*Compute Capability* genannt) werden diese Restriktionen schrittweise gelockert, daher wird hier ausschließlich auf die in der GeForce GTX 285 eingesetzte, aktuellste Compute Capability 1.3 Bezug genommen.

Datentransfer zwischen Host und Device

Bei der GeForce GTX 285 ist die Bandbreite für Speichertransfers innerhalb der GPU vom Hersteller mit 159 GB/s angegeben, eigene Messungen haben dagegen einen Durchsatz von 131 GB/s ergeben. Da diese Bandbreite auf sämtliche Threads verteilt werden muss, sollte eines der Entwurfsziele sein, diese Datentransfers zu minimieren.

Um Größenordnungen langsamer ist der Transfer zwischen dem Hauptspeicher der CPU und dem globalen Speicher der GPU. Es ist daher sinnvoll, möglichst viele Berechnungen, z.B. zur Vorverarbeitung der Daten, auf die GPU auszulagern, auch wenn diese sich nur schlecht parallelisieren lassen. Da der Overhead bei einer Speichertransaktion relativ groß ist, sollte man viele kleine Transaktionen zudem vermeiden und die Datenpakete stattdessen zu einer Transaktion zusammenfassen.

Zusammenhängende Zugriffe auf Global Memory

Damit Threads mit maximaler Bandbreite auf den globalen Speicher zugreifen können, müssen verschiedene Zugriffsregeln beachtet werden.

Variablen können mittels einer einzigen Instruktion vom globalen Speicher direkt in die Register eines Threads übertragen werden, wenn sie genau 4, 8 oder 16 Byte groß sind und deren Adresse ein Vielfaches der Variablengröße ist (*address alignment*).

Um die Latenzen zusätzlich zu verringenrn, können Zugriffe aus mehreren parallelen Threads heraus auf unterschiedliche Adressen im globalen Speicher zu einer einzigen Speicherinstruktion koalesziert werden. Dazu müssen alle 16 Threads eines *Half-Warps* auf dasselbe zusammenhängende Speichersegment zugreifen, das je nach Variablengröße 32, 64 oder 128 Byte groß ist. Im Gegensatz zu älteren Grafikkartenrevisionen ist es dabei nicht mehr maßgeblich, ob aufeinanderfolgende Threads auf sequentielle Adressen zugreifen. Fallen die Adressen jedoch in n verschiedene Speicherbereiche, so müssen n einzelne Transaktionen initiiert werden.

Bankkonflikte beim Zugriff auf Shared Memory

Auch auf den Shared Memory kann gleichzeitig aus allen 16 Threads eines Half-Warps zugegriffen werden. Der 16 KB große Speicher ist dazu in genau 16 gleichgroße Module, die sogenannten Speicherbänke (engl. *memory banks*) unterteilt. Eine Optimierung der Speicherbandbreite ist möglich, wenn Adressen unterschiedlicher Bänke simultan ausgelesen und beschrieben werden. Ein Konflikt ergibt sich hingegen, wenn zwei oder mehr Threads versuchen, auf dieselbe Speicherbank zuzugreifen. Ist dies der Fall, so müssen die in Konflikt stehenden Speichertransaktionen seriell ausgeführt werden.

Besondere Beachtung erfordert die Zuordnung der Speicheradressen zu den Speicherbänken. Denn eine Speicherbank ist kein zusammenhängender Adressbereich; vielmehr sind aufeinanderfolgende Bitfolgen von 32 Bit Länge aufeinanderfolgenden Speicherbänken zugeordnet. Es kann also vorkommen, dass ein Zugriff auf Variablen, die größer oder kleiner als 32 Bit sind, Bankkonflikte verursacht.

Ein Spezialfall tritt ein, wenn alle Threads auf dieselbe Speicherbank zugreifen. In diesem Fall werden die Daten an alle Threads gesendet (engl. *broadcast*) und es treten keine Konflikte auf.

Divergenzen bei bedingten Anweisungen

Bei der Verwendung bedingter Anweisungen (`if`, `switch`, `do`, `for` oder `while`) in einer Kernelfunktion ist zu beachten, dass Threads innerhalb eines Warps denselben Ausführungspfad abarbeiten sollten. Ist dies nicht der Fall, dann muss jede Alternative seriell ausgeführt werden, während andere Threads warten müssen. Die Anzahl der für diesen Warp benötigten Instruktionen kann dadurch erheblich größer ausfallen.

Probleme mit divergenten Threads innerhalb eines Warps können aus zwei unterschiedlichen Gründen auftreten. Zum einen dann, wenn bedingte Anweisungen von Daten abhängig sind, die für jeden Thread unterschiedlich sind, zum anderen dann, wenn eine Bedingung von der Thread ID abhängt. Letzterer Fall lässt sich jedoch häufig vermeiden, indem man Threads mit unterschiedlichen Instruktionspfaden geschickt auf verschiedene Warps aufteilt.

Gridgröße und Blockgröße

Ein Multiprozessor kann Speicherlatenzen verbergen, indem er während der Wartezeit die Instruktionen eines anderen Blocks ausführt. Um eine gute Auslastung der Prozessorkerne zu gewährleisten ist es daher sinnvoll, deutlich mehr Blöcke zu verwenden, als Multiprozessoren vorhanden sind.

Zu beachten ist jedoch, dass sich alle aktiven Blöcke die Register und den Shared Memory eines Multiprozessors teilen müssen. Je mehr Blöcke auf einem Prozessor ausgeführt werden, desto weniger Register und desto weniger Shared Memory stehen einem Block also zur Verfügung.

Ähnliche Kompromisse müssen auch beim Festlegen der Anzahl der Threads pro Block eingegangen werden. Zunächst einmal sollte die Anzahl der Threads ein Vielfaches der Warpgröße (32) betragen, damit kein Warp seine Kapazitäten verschwendet. Um etwaige Registerkonflikte zu vermeiden, wird sogar ein Vielfaches von 64 empfohlen. Begrenzt wird die Anzahl der Threads pro Block zum einen durch die hardwareseitige Obergrenze von 512, zum anderen durch die geringe Anzahl an Registern. Denn auch die Threads innerhalb eines Blocks müssen sich die diesem Block zugewiesenem Register teilen.

Die prozentuale Auslastung eines Multiprozessors (engl. *multiprocessor occupancy*) lässt sich aus dem Verhältnis zwischen den aktiven Warps und der maximalen theoretischen Anzahl aktiver Warps pro Multiprozessor berechnen.

3 Verwandte Arbeiten

Nicht zuletzt durch die Verfügbarkeit geeigneter Datensätze hat die Forschung im Bereich der visuellen Objekterkennung in den letzten Jahren große Fortschritte gemacht. Exemplarisch werden in diesem Kapitel in Hinblick auf die PASCAL VOC 2008 einige Arbeiten dieses breiten Forschungsgebiets vorgestellt.

Neuronale Netze werden nicht mehr nur als Klassifikatoren eingesetzt, sondern auch zur Synthetisierung eigener Merkmalsextraktoren angewandt. Daher soll hier zum einen ein Überblick über unterschiedliche Einsatzgebiete neuronaler Konvolutionsnetze in der Wissenschaft gegeben werden, zum anderen werden Ansätze aus der jüngsten Vergangenheit vorgestellt, bei denen neuronale Netze erfolgreich mittels GPU-Hardware beschleunigt wurden.

3.1 Verfahren zur Objekterkennung in der PASCAL 2008 Visual Object Challenge

Die PASCAL VOC Challenge [EVGW$^+$] gilt häufig als der derzeit anspruchsvollste Datensatz zur Objekterkennung. Dementsprechend können die darauf erfolgreich angewandten Verfahren als State-of-the-Art in diesem Gebiet angesehen werden.

An der Klassifizierungsaufgabe der PASCAL Visual Object Challenge 2008 nahmen 18 Teams teil, für die Lokalisierungsaufgabe gab es sieben Beiträge. Wie aus den in Abbildung 3.1a zusammengefassten Ergebnissen hervorgeht, hängt die Erkennungsrate auf dem PASCAL-Datensatz auch von der zu erkennenden Klasse ab. Die bei allen Methoden überdurchschnittliche Erkennungsleistung für die Klasse *person* liegt in der hohen A-priori-Wahrscheinlichkeit dieser Klasse begründet. Insgesamt stellten sich Kontext und Umgebung

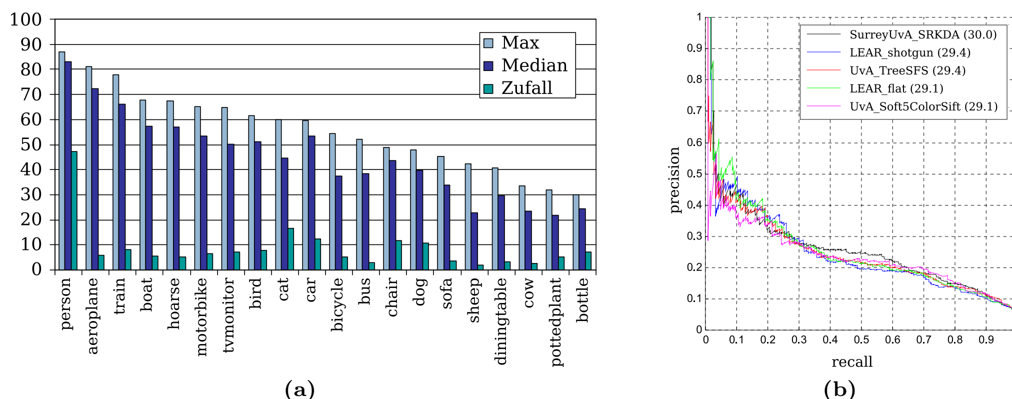


Abbildung 3.1: Ergebnisse der Objektklassifizierung in der PASCAL VOC 2008. (a) Durchschnittliche Genauigkeit für sämtliche Objektklassen. (b) Precision/Recall-Kurve verschiedener Teilnehmer für die Klasse *bottle*. (aus [EVGW$^+$], bearbeitet)

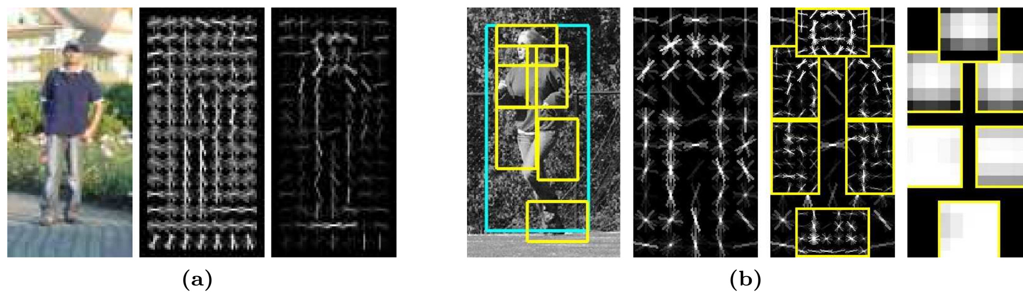


Abbildung 3.2: Histogram-of-Gradients. (a) Eingabebild, orientierte Gradienten und gewichtete Gradienten. (b) Modell deformierbarer Teile für die Klasse *person*. (aus [DT05, FMR08], bearbeitet)

eines Objekts als wichtigere Klassifizierungsmerkmale heraus als die für Menschen entscheidenden Kriterien wie Form oder Farbe eines Objekts.

Sämtliche erfolgreiche Verfahren extrahieren zunächst verschiedene Merkmale aus den Bildern und führen anschließend anhand dieser dimensionsreduzierten Daten eine Klassifikation durch. Zur Objektlokalisierung wird häufig zunächst ein Klassifikator für Objekte fester Größe trainiert, der anschließend über verschiedene Skalierungen und Positionen des Bildes geschoben wird.

Beim erfolgreichsten dieser *Sliding-Window*-Ansätze von Harzallah *et al.* [HSJG08] dienen unterschiedliche Bildrepräsentationen als Eingabe für einen Klassifikator. Zum einen die von Dalal und Triggs [DT05] zur Erkennung von Fußgängern entwickelten *Histograms-of-Gradients* (HoG), zum anderen über verschiedene Skalen auf einem dichten Gitternetz extrahierte SIFT-Merkmale. Die so gewonnenen Merkmale werden mit einem *k-means-Clustering* diskretisiert, um daraus mit der *Bag-of-Features*-Methode [LSP06] eine ungeordnete Bildrepräsentation zu erhalten, die als Eingabe für eine nicht-lineare *Support Vector Machine* (SVM) mit χ^2-Kernel dient. Insbesondere bei starren Objektklassen erzielte das Verfahren sowohl bei der Objektklassifizierung (bei 7 der 20 Klassen) als auch bei der Objektlokalisierung (bei 11 der 20 Klassen) Bestwerte.

Das *Deformable Parts Model* von Felzenszwalb *et al.* [FMR08, FGMR08] repräsentiert Objekte als eine klassenspezifische Konfiguration deformierbarer Teile. Als Bildrepräsentation werden HoG verwendet, die für das Grundgerüst in niedriger und für die einzelnen Teile in hoher Auflösung berechnet werden. Sowohl die Position der verschiedenen Objektteile in Relation zum Grundgerüst als auch die Gewichte der HoG werden als latente Variablen durch eine generalisierende SVM ermittelt.

Bei mehreren Objektklassen lernt das Verfahren eine aussagekräftige Objekthierarchie; z.B. entspricht bei der Klasse *person* ein Objektteil dem Kopf. Um mit dieser teilweise deformationsinvarianten Objektrepräsentation skalierungsinvariant Objekte zu lokalisieren, erfolgt die Suche auf einer Bildpyramide.

Hoiem, Divvala und Hays erforschen aufbauend auf dem Verfahren von Felzenszwalb *et al.*, in wie weit sich die Ergebnisse durch Einbeziehung kontextueller Informationen verbessern lassen. Dazu stellen Divvala et al. [DHH+09] vier verschiedene Bewertungsfunktionen auf, deren Gewichtung gelernt wird. Neben dem Objektdetektor fließen dabei unter anderem die globale Bildstatistik (*gist feature*) [SI07], geometrische Informationen [HE08] und Wahrscheinlichkeiten über die Größe und Position des Objekts ein. Die Lokalisierung wird anschließend zusätzlich durch Informationen aus der Segmentierung verbessert. Auf einem Großteil der Klassen konnte damit die Erkennungsrate verbessert werden, auf einigen Klas-

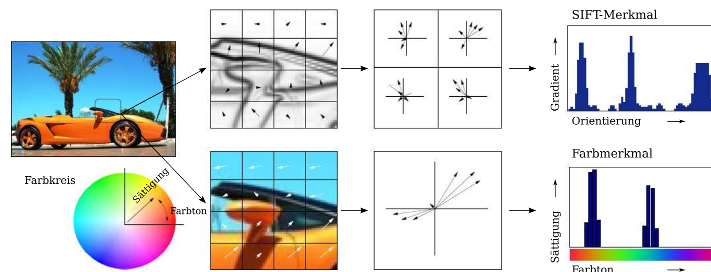


Abbildung 3.3: Bildrepräsentation aus Gradienten und Farbinformationen (aus [MSHvdW07], bearbeitet)

sen führte es jedoch zu deutlichen Verschlechterungen.

Experimentelle Ergebnisse von van de Sande *et al.* [vdSGS08] haben gezeigt, dass auf den translations-, rotations- und skalierungsinvarianten SIFT-Merkmalen [Low04] basierende Farbdeskriptoren (siehe Abb. 3.3) wie OpponentSIFT und W-SIFT robust gegenüber Veränderungen von Lichtintensität und Farbe sind. Der Beitrag der Universitäten Amsterdam und Surrey [EVGW$^+$] zeigt, dass durch diese zusätzliche Beleuchtungsinvarianz auf der PASCAL-Datenbank eine Verbesserung der Erkennungsleistung um 8% im Vegleich zu herkömmlichen SIFT-Merkmalen erreicht werden kann.

Um die Klassifikation gegenüber SVMs zu beschleunigen, wandten die Autoren zudem die für nicht-lineare Daten geeignete *Kernel Discriminent Analysis* [MRW$^+$99] an. Mit einer zur Vermeidung von Overfitting angepassten Variante mit spektralen Regressionen [CHH07] gelang es, acht der 20 Objektklassen besser zu identifizieren als die anderen Teilnehmer des Wettbewerbs.

3.2 Anwendungen von Konvolutionsnetzen

Neuronale Konvolutionsnetze (siehe Abschnitt 2.2) gelten für das Problem der Erkennung handgeschriebener Ziffern als State-of-the-Art [RPCL06, SSP03]. Bei diesem vergleichsweise einfachen Problem hat sich gezeigt, dass Gradientenabstiegsverfahren zum Training sehr viele Muster benötigen. Auf den bisher verfügbaren, relativ kleinen Datensätzen mit natürlichen Bildern dominieren auch deshalb derzeit noch Verfahren, die systematisch konkrete Merkmale extrahieren und anschließend klassifizieren. Auf spezifischen Teilproblemen mit natürlichen Bildern konnten in den letzten Jahren dennoch Fortschritte gemacht werden.

Rekonstruktion und Bildverarbeitung

Ziel der Bildrekonstruktion ist es, die für viele Anwendungen mangelhafte Qualität von Rohdaten zu verbessern. Eine Vorverarbeitung der Bilddaten kann aus verschiedenen Gründen notwendig sein, etwa wenn die Daten verrauscht sind oder eine zu geringe Auflösung haben, wenn wichtige Bildteile verdeckt sind oder durch schlechte Lichtverhältnisse der Kontrast zu gering ist.

Behnke hat gezeigt, dass solche Aufgaben der Bildrekonstruktion mit der hierarchischen *neuronalen Abstraktionspyramide* [Beh01, Beh03] gelöst werden können. Der maßgebliche Unterschied dieses Modells zu Konvolutionsnetzen nach LeCun besteht darin, dass auch laterale und rekurrente Verbindungen zwischen Schichten zugelassen sind und folglich als

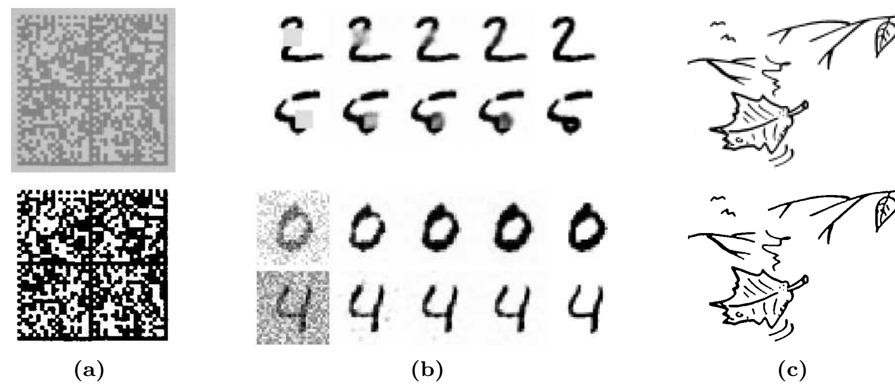


Abbildung 3.4: Mit einer neuronalen Abstraktionspyramide durchgeführte Bildrekonstuktionen. (a) Binarisierung von Matrixcodes (b) Entfernen von Rauschen und Verdeckungen (c) Superresolution (aus [Beh03], bearbeitet)

Lernverfahren *Backpropagation-Through-Time* [WP90] in Kombination mit RPROP (siehe Kapitel 2.1.4) angewandt wird.

In verschiedenen Testszenarien wurde die Performance dieses Modells evaluiert. Um die Erkennungsraten auf Matrixcodes zu verbessern, wird eine neuronale Abstraktionspyramide trainiert, die stark degenerierte Codes mit schwachem Kontrast binarisiert. Gegenüber einem herkömmlichen Verfahren mit adaptiver Anpassung der Schwellenwerte konnte damit die Erkennungsrate von 64,6% auf 99,9% gesteigert werden. Einige weitere in Abbildung 3.4 gezeigte Anwendungen dieses Modells sind die Rekonstruktion teilweise verdeckter Ziffern und stark verrauschter Ziffernsequenzen oder die Erhöhung der Auflösung eines Bildes (engl. *super-resolution*).

Browne und Ghidary [BG03] setzen Konvolutionsnetze nach dem Vorbild von LeNet zur Bildanalyse ein. Anhand der Bilder einer Infrarotkamera soll ein autonomer Roboter Risse und andere Schäden in Abwasserkanälen aufspüren und charakterisieren. Wechselnde Lichtverhältnisse, sehr unterschiedliche Typen von Rissen und uneindeutige Strukturen wie z.B. das Verbindungsstück zweier Rohre erschweren diese Detektionsaufgabe (siehe Abb.3.5).

Das von den Autoren verwendete Konvolutionsnetz besteht neben den Ein- und Ausgabeschichten aus drei verdeckten Schichten mit vier, drei und zwei Merkmalskarten. Da keine

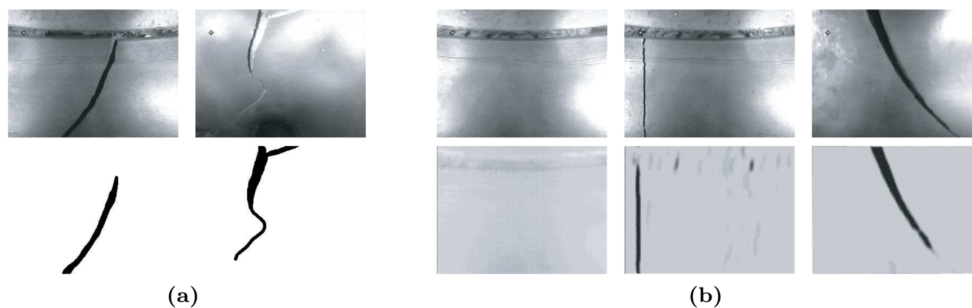


Abbildung 3.5: Erkennung von Rissen in Abwasserkanälen. (a) Trainingsbeispiele mit manuell markierten Rissen. (b) Ausgabe des trainierten Netzes für unterschiedlich komplizierte Eingabebilder. (aus [BG03], bearbeitet)

Subsamplingschichten zum Einsatz kommen, wird die 68×68 Pixel große Eingabe durch sukzessive Anwendung von 5×5 Filtern auf eine Ausgabe von 50×50 Pixeln abgebildet. Das Training dieses Netzes mit 6000 ausgewählten Bildausschnitten benötigte für 10.000 Trainingsepochen ca. zwei Stunden und erreichte damit eine Fehlerrate von 7% falsch erkannten Pixeln auf der Testmenge.

Gesichtserkennung

Große Fortschritte wurden in den letzten Jahren bei der Lokalisierung und Klassifizierung von Gesichtern in natürlichen Bildern erzielt, die inzwischen bei einer Vielzahl von praxisnahen Anwendungen zum Einsatz kommen, z.B. bei der Benutzeridentifizierung, bei der Videoüberwachung oder in Mensch-Maschine-Interaktionssystemen.

Eine Komponente eines bereits 1997 von Lewrence *et al.* [LGTB97] vorgestellten größeren Systems zur Gesichtsklassifizierung ist ein Konvolutionsnetz. Als Eingabe des Netzes dient eine mit einer *selbstorganisierenden Karte* [Koh90] erzeugte quantisierte und dimensionsreduzierte Repräsentation des Gesichts. Auf der ORL-Datenbank mit je fünf Trainings- und Testbildern von 40 verschiedenen Personen konnte ein Testfehler von 3,8% erzielt werden.

Die Bestimmung der Augenposition in Bildern der BioID-Datenbank führt Behnke mit der rekurrenten *neuronalen Abstraktionspyramide* [Beh05] durch und erreicht eine Fehlerrate von 1,3%. In synthetisch erzeugten Sequenzen aus bewegten Bildern können damit die Augen in 97,4% der Fälle verfolgt werden. Auch der *Convolutional Face Finder* (CFF) von Garcia und Delakis [GD04] lokalisiert Gesichter, wird aber in der Lernphase nicht mit ganzen Bildern, sonder mit 32×36 Pixel groß ausgeschnittenen Gesichtern trainiert. Die Lokalisierung in diesem LeNet-ähnlichen Modell erfolgt in verschiedenen Skalen auf beliebig dimensionierten Eingabebildern. Das System ist robust gegenüber Rotationen um 20 Grad, Gesichtsdrehungen um bis zu 60 Grad sowie gegenüber unterschiedlichen Gesichtsposen und Beleuchtungen.

Die von Osadchy, LeCun und Miller vorgestellte Variante von LeNet [OLM07] ermittelt nicht nur die Position von Gesichtern, sondern simultan auch deren Roll-, Nick- und Gierwinkel sowie die Skalierung. Eingabedaten dieses Netzes sind in der Lernphase 32×32 Pixel große Bildausschnitte; Ausgaberaum ist eine 6-dimensionale Mannigfaltigkeit. Aufgrund der Translationsinvarianz der Konvolutionen ist ähnlich wie beim CFF eine Lokalisierung auf beliebig großen Bildern möglich. Skalierungsinvarianz wird durch die Anwendung auf ei-

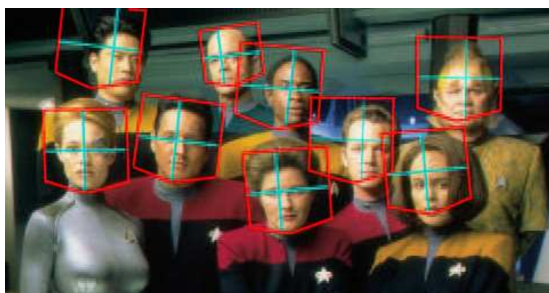


Abbildung 3.6: Ergebnisse der Gesichtserkennung. Die Polygone geben den geschätzten Roll- und Nickwinkel an, das Fadenkreuz den Gierwinkel. (aus [OLM07])

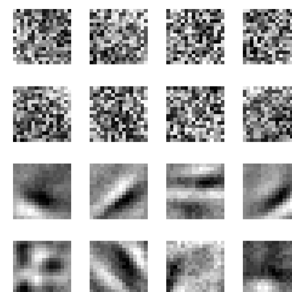


Abbildung 3.7: Oben: Einige Filter eines trainierten Konvolutionsnetzes. Unten: Identisches Netz mit Kernelregularisierung. (aus [YXG08], bearbeitet)

ne Bildpyramide erreicht, zudem ist das System sehr robust gegenüber Rotationen um die o.g. Drehwinkel. Experimentelle Ergebnisse haben gezeigt, dass die Erkennungsleistung bei gleichzeitiger Schätzung von Koordinaten und Gesichtspose durch Synergieeffekte steigt.

Stefan Duffner führt in seiner Dissertation [Duf07] keine Lokalisierung durch, sondern analysiert mit einem Konvolutionsnetz verschiedene Eigenschaften von Gesichtern. Auf zentrierten und auf 46×56 Pixel skalierten Auschnitten der BioID-Datenbank konnten bei der Bestimmung von zehn markanten Gesichtsmerkmalen (wie z.B. Augen und Mundwinkel) 87% der Punkte korrekt identifiziert werden. Bei der Klassifizierung nach Geschlecht auf der FERET-Datenbank konnten 94,7% der Muster korrekt klassifiziert werden, bei der Identifizierung von Personen auf der ORL-Datenbank wurden 93% korrekt erkannt.

Allgemeine Objekterkennung

Der *Convolutional Object Finder* von Garcia und Duffner [EZW+06] ist ein Konvolutionsnetz zur allgemeinen Objekterkennung, das im Rahmen der PASCAL Challenge 2005 eingesetzt wurde, um Objekte einer bestimmten Klasse zu lokalisieren. Ähnlich wie der *Convolutional Face Finder* wird das Netz auf Eingaben fester Größe (z.B. 52×30 Pixel für die Klasse *motorbikes*) trainiert und beschränkt sich somit für jede Objektklasse auf ein festes Seitenverhältnis. Skalierungsinvarianz bei der Lokalisierung wird durch die Eingabe eines Bildes in verschiedenen Größen erreicht.

Neben der Erkennung von handschriftlichen Ziffern und der Gesichtsanalyse wenden Yu, Xu und Gong [YXG08] ein durch anwendungsspezifische Kernel reguliertes Konvolutionsnetz zur Objekterkennung auf den Caltech-101-Datensatz an. Mit einem nichtlinearen *Spatial-Pyramid-Matching*-Kernel [LSP06] konnte die Erkennungsleistung drastisch von 43,6% auf 67,4% gesteigert werden. Abbildung 3.7 verdeutlicht die Auswirkung dieser Vorwissen ausnutzenden Regularisierung. Gegenüber einer SVM zeichnet sich dieses Modell vor allem durch eine höhere Geschwindigkeit in der Objekterkennungsphase aus.

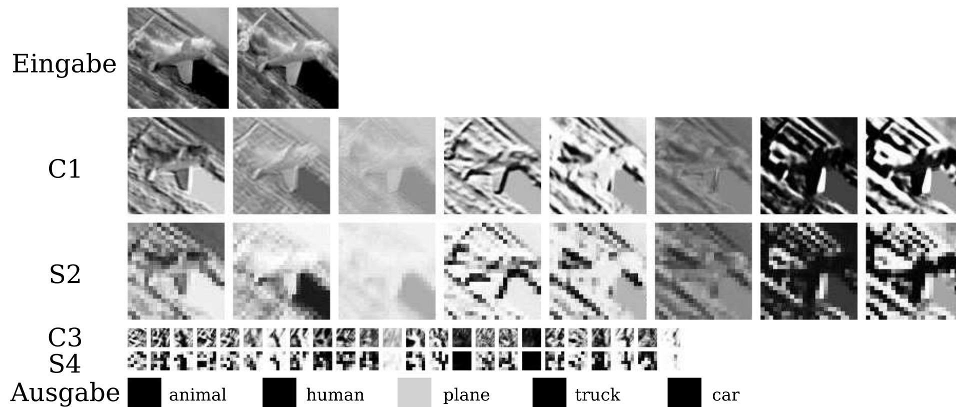


Abbildung 3.8: Aktivitäten der Merkmalskarten auf verschiedenen Schichten von LeCuns Konvolutionsnetz für ein Eingabemuster des NORB-Datensatzes. (aus [LHB04], bearbeitet)

LeCun, Huang und Bottou [LHB04] wendeten ein 6-schichtiges Konvolutionsnetz auf den semi-synthetischen NORB-Datensatz (siehe Abschnitt 2.3.2) an. Auf der vereinfachten Version dieses Datensatzes mit uniformem Hintergrund erreichte das Netz einen Testfehler von 6,6 %, im Vergleich zu einem Testfehler von 12,6 % einer SVM mit gaußschem Kernel. Auf

der schwierigeren Variante mit texturiertem Hintergrund, für die keine Vergleichswerte vorliegen, konnte mit dem Konvolutionsnetz ein Testfehler von 16,7 % erreicht werden. Die Erkennungsleistung sinkt drastisch auf 39,9 %, wenn dem Netz jeweils nur eines der Stereobilder zur Verfügung steht.

3.3 Beschleunigung mit Grafikkartenhardware

Seit mit der CUDA-Architektur von Nvidia eine leicht zugängliche Schnittstelle zur Programmierung von Grafikkarten verfügbar ist, sind zahlreiche kommerzielle und wissenschaftliche Anwendungen für diese Plattform adaptiert worden. Es können z.B. physikalische Simulationen, medizinische Analysen, Kryptographie-Algorithmen und Verfahren zur Videodekodierung können um mehr als eine Größenordnung beschleunigt werden. Die wenigen im Rahmen dieser Arbeit relevanten Implementierungen aus den Bereichen Objekterkennung, neuronale Netze und Bildverarbeitung werden in diesem Abschnitt vorgestellt.

Multilayer-Perzeptrons

Die inhärente Parallelität neuronaler Netze wurde in wissenschaftlichen Arbeiten noch nicht umfassend auf GPU-Multiprozessoren ausgenutzt. Lahaber *et al.* [LAN08] untersuchten, ob sich Multilayer-Perzeptrons auf CUDA-Architekturen beschleunigen lassen. Die Berechnungen eines MLP lassen sich größtenteils mit den durch die CUBLAS-Bilbliothek [Nvi08a] beschleunigten Matrixmultiplikationen und Vektoradditionen darstellen. Das Training eines zweischichtigen Netzes mit 484 Eingabeneuronen, 128 verdeckten Neuronen und 10 Ausgabeneuronen konnte damit auf einer GeForce 8800 GTX gegenüber einer ähnlichen Matlab-Implementierung auf einem 3 GHz Pentium IV um den Faktor 90 bis 110 beschleunigt werden.

Auch Jang *et al.* [JPJ08] implementierten zur Identifizierung von Text in einem Bild ein vollverknüpftes MLP mit CUBLAS. Da die Vorverarbeitung der Eingabe auf der CPU das System ausbremste, wurden diese Berechnungen zudem mittels OpenMP[1] über mehrere CPU-Kerne parallelisiert. Auf einem 2,4 GHz Intel Core 2 Quad Q6600 konnte diese Anwendung mit einer GeForce 8800 GTX um den Faktor 20 beschleunigt werden.

Objekterkennung

Das dem Neocognitron ähnliche hierarchische HMAX-Modell von Riesenhuber und Poggio [RP99] wurde am MIT auf paralleler Hardware implementiert. Dabei hat Chikkerur [Chi08] ausgenutzt, dass in den sogenannten einfachen Schichten unterschiedliche Filter auf dieselbe Eingabe angewandt werden. Durch Parallelisierung über diese Filter lief der Algorithmus auf einem System mit GeForce 8800 GTX ca. zehn mal schneller als eine äquivalente CPU-Implementierung auf beiden Kernen einer 3 GHz Dual Core CPU.

Wojek *et al.* [WDSS08] implementierten zur Objekterkennung ein Modell, das zunächst die *Histograms-of-Gradients* [DT05] auf einer GeForce 8 Ultra extrahiert. Zur Vermeidung des bei Speichertransfers auftretenden Overheads wurde zudem auch die Klassifikation mit einer SVM auf Grafikkartenhardware realisiert. Der erzielte Beschleunigungsfaktor von 34 ermöglicht die Detektion in Echtzeit mit 34 Bildern pro Sekunde bei einer Auflösung von 320×240 Pixeln.

[1] `http://www.openmp.org`

Konvolutionen

Für viele Anwendungen der Bildverarbeitung werden berechnungsintensive zweidimensionale Konvolutionsfilter benötigt. Verschiedene Möglichkeiten, separierbare Filter effizient auf CUDA-kompatibler Hardware zu implementieren, erläutert eine Fallstudie von Nvidia [NP07]. Unter Ausnutzung des Shared Memory kann demnach bei einer Faltung mit einem 5×5-Filter ein Durchsatz von bis zu 1800 Megapixeln pro Sekunde erreicht werden.

Chellapilla, Puri und Simard [CPS06] implementierten ein Konvolutionsnetz zur Erkennung handschriftlicher Zeichen auf einer GeForce 7800 Ultra. Da diese Grafikkarten-Serie nicht auf der CUDA-Architektur basiert, wurden in einem GPGPU-Framework (siehe Kapitel 2.4.2) die Pixel-Shader programmiert. Zur Beschleunigung wurden zum einen die Schleifen zur Berechnung der Konvolutionen aufgerollt (engl. *loop unrolling*), zum anderen wurde die zweidimensionale Speicherstruktur ausgenutzt. Mit diesen Techniken konnte gegenüber einer CPU-Implementierung auf einem 2,8 GHz Pentium IV die Berechnung einer vollständigen Trainingsepoche um bis zum 4,11-fachen beschleunigt werden.

3.4 Zusammenfassung und Einordnung

Auf niedrigdimensionalen Problemen wie der Erkennung handschriftlicher Ziffern hat sich bereits erwiesen, dass neuronale Konvolutionsnetze sowohl bei der Merkmalsextraktion als auch zur Klassifikation anderen State-of-the-art-Verfahren mindestens ebenbürtig wenn nicht sogar überlegen sind (siehe auch Kapitel 2.3.1). Zur Klassifizierung von Objekten in natürlichen Bildern liefern derzeit Verfahren, die manuell extrahierte Merkmale mittels Support Vector Machines klassifizieren die besten Ergebnisse.

Die Gründe dafür, dass Konvolutionsnetze auf natürlichen Bildern bisher nur in Spezialfällen wie zur Gesichtsanalyse eingesetzt werden, liegen auch darin, dass Faltungsoperationen auf großen Eingaben sehr rechenintensiv sind. Andererseits können gerade diese Operationen parallel auf dem gesamten Eingabebild durchgeführt werden und sie eignen sich daher für eine Implementierung auf paralleler Hardware. Andere derartig massiv parallele Algorithmen konnten mit der CUDA-Schnittstelle auf Multiprozessor-GPUs um ein bis zwei Größenordnungen beschleunigen.

In dieser Diplomarbeit wird daher versucht, die inhärente Parallelität von neuronalen Konvolutionsnetzen auszunutzen, um deren Berechnung auf parallelen GPU-Multiprozessoren zu beschleunigen. Ziel ist es, bekannte Architekturen und Lernverfahren durch diese zusätzliche Rechenleistung auf deutlich größeren Bildern als bisher zu erproben.

4 Entwurf eines Konvolutionsnetzes

Zwei Zielsetzungen verfolgt diese Diplomarbeit. Zum einen soll ein neuronales Netz entwickelt werden, das in der Lage ist, Objekte in natürlichen Bildern zu identifizieren, zum anderen soll ergründet werden, ob und in welchem Maße sich ein solches Netz durch parallele Hardware beschleunigen lässt.

Dieses Kapitel befasst sich mit dem ersten Ziel, nämlich dem Entwurf eines neuronalen Konvolutionsnetzes zur Objekterkennung. Um die Rechenleistung der GPU-Multiprozessoren optimal auszunutzen, berücksichtigt die in Abschnitt 4.1 erläuterte Netztopologie bereits die Einschränkungen der CUDA-Schnittstelle. In Abschnitt 4.2 wird die Kapazität dieser Architektur in Hinblick auf die Generalisierungsleistung analysiert. Bei tiefen neuronalen Netzen, also Netzen mit vielen nichtlinearen Schichten, stellt vor allem die Initialisierung der Gewichte und das Traineren derselben eine Herausforderung dar. Auch die durch Einschränkungen der Hardware notwendig gewordenen Erweiterungen des Lernverfahrens werden in Abschnitt 4.3 vorgestellt. Abschließend werden in Abschnitt 4.4 verschiedene Codierungen der Ein- und Ausgabe diskutiert.

4.1 Netztopologie

Grundlage des in dieser Arbeit implementierten Netzes bildet die Struktur von LeNet-5 (siehe Abschnitt 2.2.2). Dass derartige Netze bei der Erkennung handschriftlicher Ziffern auf der MNIST-Datenbank (Abschnitt 2.3.1) die mit Abstand erfolgreichsten Verfahren sind

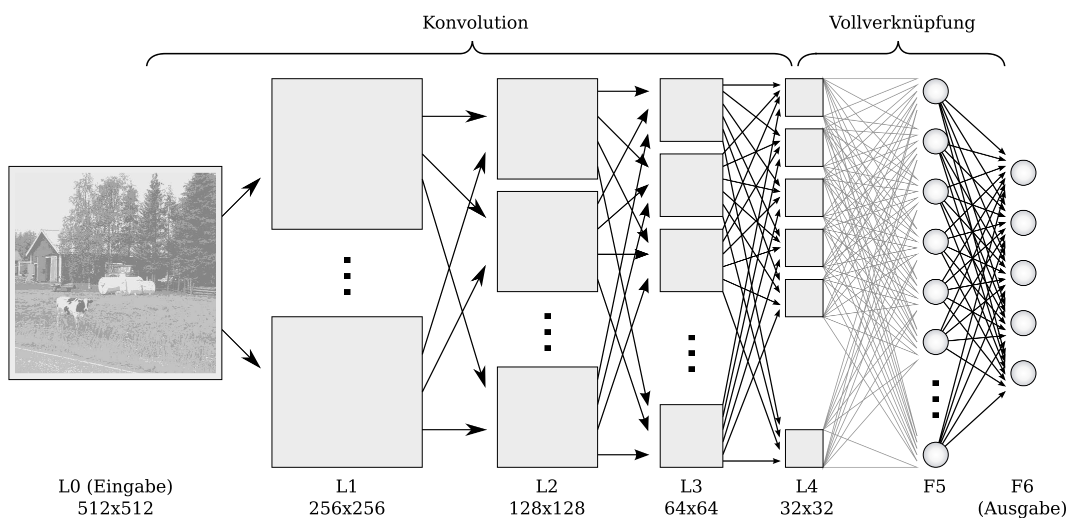


Abbildung 4.1: Struktur des Konvolutionsnetzes. Die unteren Konvolutionsschichten sind lokal mit Faltungskernen als Gewichten verknüpft. An diese Schichten schließen sich vollverknüpfte Schichten an, die sämtliche Aktivitäten der vorherigen Schicht als Eingabe erhalten.

[RPCL06], legt die Vermutung nahe, dass mit größeren Konvolutionsnetzen eine robuste Objekterkennung in natürlichen Bildern möglich ist. Der Grundgedanke dieser hierarchischen vorwärtsgerichteten Architektur ist es, auf den unteren Ebenen aus Bildern hoher Auflösung einfache Merkmale zu extrahieren und diese dann sukzessive zu komplexeren Merkmalen gröberer Auflösung zu kombinieren.

Abhängig von der Größe der Eingabebilder und der Problemstellung ist das Netz konfigurierbar. Exemplarisch wird in diesem Kapitel das in Abbildung 4.1 gezeigte sechsschichtige Netz mit einer Eingabegröße von 512×512 Pixeln beschrieben. Außer der Eingabeschicht enthält das Netz zwei verschiedene Typen von Schichten, in denen Informationsverarbeitung stattfindet: die sogenannten tieferen, bildartigen Konvolutionsschichten (mit Ln bezeichnet) und die vollverknüpften Schichten (mit Fn bezeichnet).

4.1.1 Hierarchische Konvolutionsschichten

Eine Konvolutionsschicht enthält mehrere zweidimensionale bildartige Merkmalskarten, die unterschiedliche Repräsentationen des Eingabebildes enthalten. Bedingt durch die in Kapitel 5 beschriebene effiziente Implementierung sind Merkmalskarten in diesem Modell immer quadratisch und deren Seitenlänge ist eine Zweierpotenz. Jede Konvolutionsschicht führt gleichzeitig eine Faltung und ein Subsampling durch, so dass sich die Seitenlänge der Merkmalskarten von einer Schicht zur nächsten um den Faktor zwei verringert. Der mit dieser reduzierten räumlichen Auflösung einhergehende Informationsverlust soll durch eine zur Ausgabeschicht hin zunehmende Anzahl an Merkmalskarten kompensiert werden.

In dem von LeCun *et al.* [LBBH98] vorgeschlagenen und weit verbreiteten Modell sind – wie in Tabelle 2.1 auf Seite 16 beschrieben – nur ausgewählte Merkmalskarten miteinander verknüpft. Im Gegensatz dazu werden in dem hier vorgestellten Konvolutionsnetz sämtliche Merkmalskarten zweier benachbarter Schichten miteinander verknüpft. Diese Entwurfsentscheidung vereinfacht nicht nur die Implementierung, sondern ermöglicht erst die einheitliche parallele Berechnung unterschiedlicher Verknüpfungen. Für die Implementierung ist zudem wichtig, dass der Propagierungsfunktion weniger Parameter übergeben werden müssen und somit der Bedarf an benötigten Registern reduziert wird.

Von LeCun *et al.* wird angeführt, dass mit einer spärlichen Verknüpfung Symmetrien gebrochen werden können und die Diversität der Merkmalskarten gefördert wird. Dass aber auch mit vollverknüpften Konvolutionsschichten ein mindestens ähnlich guter Lernerfolg erzielt werden kann, zeigt die zum State-of-the-art zählende Arbeit von Simard *et al.* [SSP03]. Trotzdem soll der Einfluss spärlicher Verknüpfungen in dieser Arbeit untersucht werden, indem diese durch geeignete Gewichtsinitialisierungen simuliert werden.

4.1.2 Lokale Filter

Einen weiteren Gegensatz dieses Modells zu LeNet-5 stellen die Verbindungen zwischen zwei Merkmalskarten dar, weil in jeder Schicht sowohl Konvolutionen, als auch Subsampling von demselben Filter ausgeführt werden (siehe Abbildung 4.2). Sei $a_i(x,y)$ die Aktivität der Merkmalskarte i an der Position (x,y). Es seien w_{ij} der Filter, der die Merkmalskarte i auf die Merkmalskarte j der nächsten Schicht abbildet, $w_{ij}(u,v)$ das Element dieses Filters an Position (u,v) und b_{ij} das zugehörige Biasgewicht. Die Netzeingabe net_j der Merkmalskarte j wird über alle I Merkmalskarten der vorherigen Schicht summiert:

$$net_j(x,y) = \sum_{i=0}^{I} \left(\left(\sum_{(u,v)} w_{ij}(u,v) \cdot a_i(2x+u,2y+v) \right) + b_{ij} \right) \tag{4.1}$$

Diese Berechnung wird mit denselben Gewichten für jede Position (x,y) der Ausgabekarte j durchgeführt, so dass sich die rezeptiven Felder zweier benachbarter Ausgabeneuronen überlappen. Die Aktivität $a_j(x,y)$ der Ausgabekarte ergibt sich durch Anwendung der nichtlinearen Aktivierungsfunktion:

$$a_j(x,y) = f_{act}(net_j(x,y)) \tag{4.2}$$

Der Grund für ein gleichzeitiges Durchführen von Konvolution und Subsampling liegt vor allem in den Hardwareeinschränkungen der GPUs. In reinen Konvolutionsschichten wie bei LeCun *et al.* findet nämlich keine Dimensionsreduzierung statt, so dass diese Merkmalskarten viermal größer sind als die der darauf folgenden Subsampling-Schicht. Entsprechend wird zur Bereithaltung der Aktivitäten solcher Schichten erheblich mehr Speicher benötigt, der jedoch in aktuellen CUDA-Grafikkarten extrem begrenzt ist. Sowohl Simard *et al.* [SSP03], als auch Behnke [Beh03] verwenden in ihren Modellen eine solche Kombination von Subsampling und Konvolution und berichten nicht von negativen Effekten.

Abweichend von Simard und analog zu Behnke wird in dieser Arbeit ein Faltungskern mit gerader Seitenlänge gewählt. Motiviert wird diese Wahl durch die in Abbildung 4.3 dargestellte Tatsache, dass bei Subsampling-Kernen mit einer ungeraden Anzahl von Elementen die Neuronen einer Schicht unterschiedlich viele ausgehende Verbindungen haben. Bei gerader Seitenlänge der Filter wird hingegen mit Ausnahme der Randneuronen jede Aktivität gleich häufig in der nächsten Schicht weiterverarbeitet.

Konkret bestehen die trainierbaren Konvolutionsfilter in diesem Modell auf allen Schichten aus 8×8 Gewichten sowie einem Biasgewicht und überlappen sich somit zu 75%. Ein Neuron der Schicht L1 erhält also Eingaben von 64 Neuronen der vorherigen Schicht. In der Schicht L2 wird ein Neuron bereits von $22 \times 22 = 484$ Neuronen der Eingabeschicht beeinflusst. Das sogenannte *rezeptive Feld* eines Neurons ist also ähnlich wie im visuellen Cortex des Menschen (siehe Abbildung 2.4 auf Seite 14) in höheren Schichten größer. Bei einem 8×8 Filter erhält ein Bereich aus $n \times n$ Neuronen Eingaben aus $(2n+6) \times (2n+6)$ Neuronen der darunterliegenden Schicht. Damit ergibt sich für die Breite r_n des rezeptiven Feldes eines Neurons in Schicht Ln:

$$r_n = 2^n + (2^n - 1) \cdot 6 \tag{4.3}$$

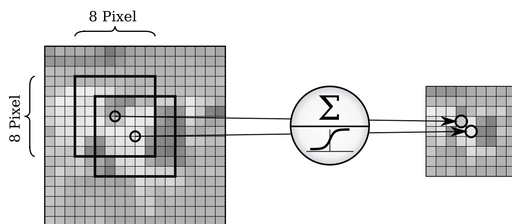


Abbildung 4.2: Verarbeitung zwischen zwei Merkmalskarten. Auf die Eingabe werden in regelmäßigen Abständen zunächst ein 8×8 Filter angewandt dessen Resultat als Eingabe für die Aktivierungsfunktion dient.

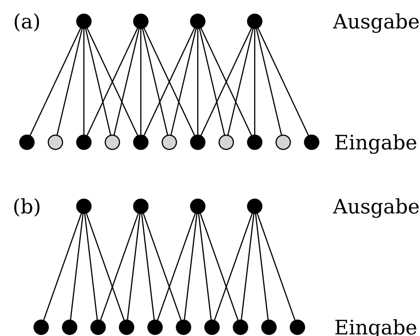


Abbildung 4.3: Auswirkung unterschiedlicher Filtergrößen. (a) Bei ungeraden Filtergrößen haben einige Eingabeneuronen (hellgrau) weniger ausgehende Verbindungen. (b) Bei geraden Filtergrößen ist die Anzahl für alle Eingabeneuronen gleich.

Schicht	Größe	Karten	Neuronen	Filter	Verknüpfungen	Parameter
L0	512×512	4	1.048.576	-	-	-
L1	256×256	8	524.288	32	136.314.880	2.080
L2	128×128	16	262.144	128	136.314.880	8.320
L3	64×64	32	131.072	512	136.314.880	33.280
L4	32×32	64	65.536	2048	136.314.880	133.120
Schicht	Neuronen			Verknüpfungen		Parameter
F5	100			6.553.600		6.553.600
F6	10			1.000		1.000
Summe	2.031.726			551.814.120		6.731.400

Tabelle 4.1: Kapazität des neuronalen Konvolutionsnetzes. Obwohl ein Großteil der Verknüpfungen zwischen Neuronen in den Konvolutionsschichten liegt, verfügen die vollverknüpften Schichten über wesentlich mehr Parameter.

Somit deckt das rezeptive Feld eines Neurons der Schicht L4 maximal 106×106 Pixel ab, also 4,29% der Eingabekarte. Große Objekte werden somit auf mehrere Pixel der Merkmalskarten in Schicht L4 abgebildet, kleinere Objekte können durch ein einziges Pixel repräsentiert werden.

Ein weiteres Kriterium für die Wahl der Filtergröße ist der verfügbare Speicherplatz. Um einen schnellen Zugriff auf die Gewichte zu ermöglichen, sollen sie im nur 64 KB großen Konstantenspeicher der GPU abgelegt werden. Ein Filter benötigt bei Verwendung von Gleitkommazahlen mit einfacher Genauigkeit (engl. *single precision float*) $(8\times8+1)\times4$ Byte $= 260$ Byte Speicherplatz. Somit reicht der Konstantenspeicher für ca. 250 dieser Filter und muss nur selten ausgetauscht werden.

Wie sich in Kapitel 5 zeigen wird, haben 8×8 große Filter außerdem den Vorteil, dass die 64 Filterelemente von genau zwei Threadwarps verarbeitet werden können, was nicht unerheblich zur Beschleunigung der Laufzeit beiträgt.

4.1.3 Vollverknüpfte Schichten

Die Merkmalskarten der letzten Konvolutionsschicht haben in diesem Modell mindestens eine Größe von 32×32 Pixeln. Diese Einschränkung liegt in der in Kapitel 5.3 beschriebenen optimierten Implementierung des Propagierungskernels begründet. Auf diese letzte bildartige Schicht folgen zwei vollverknüpfte Schichten, welche die Daten weiter abstrahieren und komprimieren, aber die Topologie des Eingangsbildes nicht mehr erhalten.

Die Neuronen dieser Schicht sind wie in einem Multilayer-Perzeptron (siehe 2.1.2) aufgebaut und erhalten als Eingabe sämtliche Pixel aller Merkmalskarten aus L4. Die Verarbeitung in der Ausgabeschicht erfolgt analog.

4.2 Kapazität des Netzes

Gekoppelte Gewichte (siehe Kapitel 2.2) reduzieren die Anzahl der freien Parameter und tragen somit zur Generalisierungsfähigkeit des neuronalen Netzes bei. Dieser positive Einfluss auf die Kapazität des Netzes soll im Folgenden anhand einer konkreten Instanz des hier vorgestellten Modells erläutert werden.

Als Eingabe erhält dieses Netz vier verschiedene Repräsentationen des Bildes, somit besteht die erste Schicht aus vier Merkmalskarten. Die Anzahl der Merkmalskarten verdoppelt

sich sukzessive pro Schicht, so dass die letzte Konvolutionsschicht L4 schließlich aus 64 Merkmalskarten besteht. Auf diese Schicht folgt eine vollverknüpfte Schicht mit 100 Neuronen und schließlich eine Ausgabeschicht mit zehn Neuronen. Tabelle 4.1 stellt die Anzahl der Neuronen, die Anzahl der Verknüpfungen und die Anzahl der freien Parameter für jede Schicht in Relation zueinander dar.

Obwohl die Neuronen zwischen zwei Schichten nur lokal verknüpft sind, verfügt dieses Netz in den Konvolutionsschichten über mehr als eine halbe Milliarde Verknüpfungen. Durch gekoppelte Gewichte liegt die Anzahl der freien Parameter dagegen nur bei einigen Zehntausenden und bleibt somit im Rahmen des Trainings handhabbar. Positiver Effekt der gekoppelten Gewichte ist nicht nur ein erheblich niedrigerer Speicherplatzbedarf, sondern durch die reduzierte Kapazität wird zudem die Generalisierungsleistung des Netzes gefördert [LeC89]. Obwohl 98,8% der Verknüpfungen des gesamten Netzes in den Konvolutionsschichten liegen, umfasst dieser Teil nur 2,6% der trainierbaren Parameter.

4.3 Lernverfahren

Die Gewichte des Konvolutionsnetzes werden mit dem Gradientenabstiegsverfahren Backpropagation of Error (siehe Abschnitt 2.1.3) trainiert. Unter anderem die gekoppelten Gewichte des Konvolutionsnetzes machen jedoch eine Anpassung des Lernalgorithmus erforderlich. Durch den begrenzten Speicher der GPU-Hardware und durch die Parallelisierung sind zudem weder reines Offline-Training noch reines Online-Training möglich, daher werden in diesem Abschnitt Modifikationen des Lernverfahrens vorgestellt.

4.3.1 Aktivierungsfunktion

Durch die geeignete Wahl einer Aktivierungsfunktion erhält das Netz die Möglichkeit, nichtlineare Abbildungen zu lernen. Hier wird der ursprungssymmetrische nichtlineare Tangens Hyperbolicus (siehe Kapitel 2.1.3) verwendet, weil die Ausgaben dieser Funktion (also die Eingaben der nächsten Schicht) einen Mittelwert nahe Null haben. Dies führt in Kombination mit normalisierten Eingaben häufig zu einer schnelleren Konvergenz des Lernverfahrens.

Sowohl am Ursprung der Aktivierungsfunktion als auch an den Asymptoten treten sehr kleine Gradienten auf. Um solche flachen Stellen auf der Fehleroberfläche zu vermeiden, wird die von LeCun *et al.* [LBOM98] empfohlene skalierte Funktion

$$f_{act}(x) = 1{,}7159 \tanh(\frac{2}{3}x) \tag{4.4}$$

verwendet. Wenn als Zielwerte -1 und +1 gewählt werden, fallen die Aktivitäten in einen Wertebereich mit hohem Gradienten und größtmöglichem nichtlinearen Anteil und beschleunigen somit das Lernverfahren.

4.3.2 Gewichtsinitialisierungen

Da die Anfangswerte der Gewichte signifikante Auswirkungen auf den Lernprozess haben können, sollen in dieser Arbeit unterschiedliche Strategien zur Initialisierung der Gewichte, insbesondere der Gewichte in den Konvolutionsschichten, evaluiert werden. Um Symmetrien zu brechen haben alle Initialisierungen grundsätzlich einen additiven oder multiplikativen randomisierten Anteil, unterscheiden sich ansonsten aber erheblich.

Randomisierte Initialisierung Bei einer rein zufälligen Initialisierung sollten die Gewichte so gewählt werden, dass Aktivitäten hauptsächlich in den linearen Bereich der sigmoiden Funktion fallen [LBOM98]. Sowohl zu kleine als auch zu große Gewichte neigen dazu, sehr kleine Gradienten zu verursachen und somit die Konvergenz zu verlangsamen. Um eine Standardabweichung von $\sigma_y = 1$ der gewichteten Summe der Aktivitäten zu erzielen, werden die Gewichte aus einer Zufallsverteilung mit Mittelwert Null und einer Standardabweichung von

$$\sigma_w = \frac{1}{\sqrt{c_{in} \times n}} \tag{4.5}$$

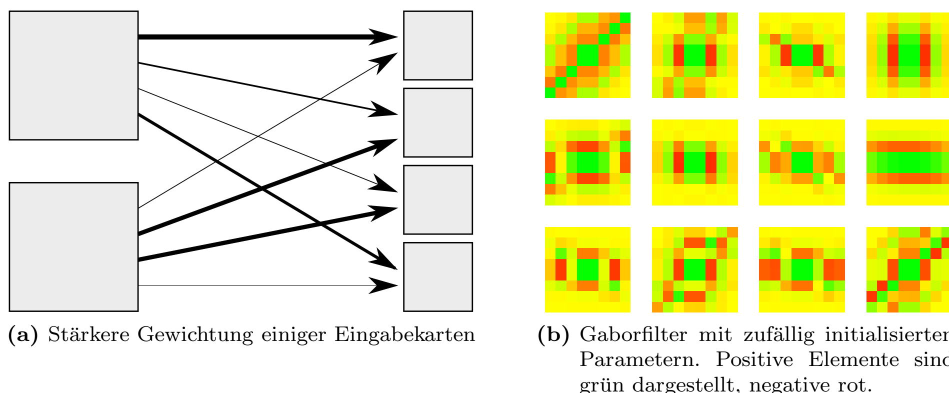
gewählt. Dabei bezeichnet c_{in} die Anzahl der eingehenden Merkmalskarten und $n = 65$ die Filtergröße.

Dominante Filter Unterschiedliche Merkmalsextraktoren innerhalb einer Schicht werden in anderen Konvolutionsnetzen meistens durch unsymmetrische spärliche Verknüpfungen zwischen den Merkmalskarten forciert [LBBH98]. Aufgrund der in dieser Arbeit implementierungsbedingten Vollverknüpfung soll derselbe symmetriebrechende Effekt erzielt werden, indem die Gewichte einiger weniger Filter deutlich größer initialisiert werden. Eine Merkmalskarte wird somit nur von wenigen Karten der vorherigen Schicht maßgeblich beeinflusst (siehe Abbildung 4.4a). Die Wahl der auf diese Art verstärkten Verbindungen erfolgt zufällig.

Gaborfilter Inspiriert durch die Tatsache, dass die einfachen Zellen im menschlichen visuellen Cortex auf Gabor-ähnliche Reize reagieren, werden Gaborfilter [Gab46] in der Bildverarbeitung zur Kantendetektion, zur Texturunterscheidung und auch zur Objekterkennung [ML06] eingesetzt. Ein Gaborfilter kann durch folgende Formel beschrieben werden:

$$G(x,y) = \exp\left(-\frac{(X^2 + \gamma^2 Y^2)}{2\sigma^2}\right) \cos\left(\frac{2\pi}{\lambda} X\right) \tag{4.6}$$

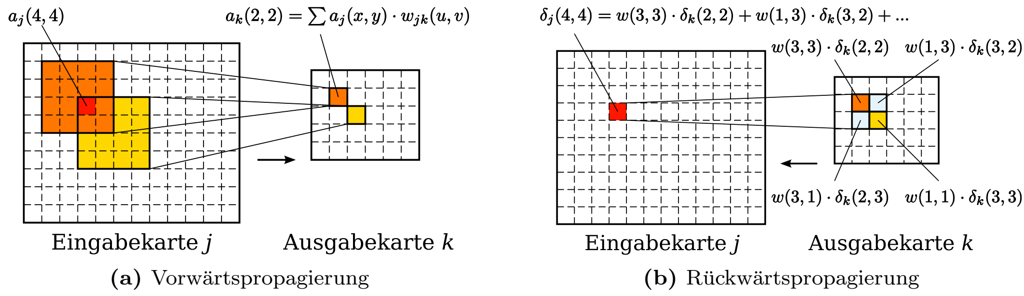
mit $X = x\cos\theta - y\sin\theta$ und $Y = x\sin\theta + y\cos\theta$. Der Orientierungswinkel θ, das Seitenverhältnis γ, die effektive Breite σ und die Wellenlänge λ lassen sich frei wählen. Einige auf diese Weise durch zufällige Wahl der Parameter initialisierte Filter sind in Abbildung 4.4b dargestellt.



(a) Stärkere Gewichtung einiger Eingabekarten

(b) Gaborfilter mit zufällig initialisierten Parametern. Positive Elemente sind grün dargestellt, negative rot.

Abbildung 4.4: Möglichkeiten zur Initialisierung der Gewichte



(a) Vorwärtspropagierung **(b)** Rückwärtspropagierung

Abbildung 4.5: Beispiel eines 4×4 Filters. (a) An der Berechnung der Aktivität eines Neurons der Ausgabekarte j sind alle Elemente des Filters beteiligt. (b) Bei der Rückwärtspropagierung hängt das Fehlersignal auf der Eingabekarte i nur von vier Elementen ab.

4.3.3 Gradientenabstieg

Für das Gradientenabstiegsverfahren wird als zu minimierende Fehlerfunktion der mittlere quadratische Fehler (engl. *mean squared error*, MSE) verwendet:

$$E = \frac{1}{2} \sum_{p=1}^{P} \sum_{c=1}^{C} (t_c^p - o_c^p)^2 \tag{4.7}$$

Darin bezeichnet P die Anzahl der Muster, C die Anzahl der Ausgabeklassen, t_c^p die gewünschte Ausgabe für das Muster p und o_c^p die tatsächliche Ausgabe. Das Fehlersignal δ_j eines Neurons in der Ausgabeschicht und in den vollverknüpften Schichten berechnet sich wie in einem Multilayer-Perzeptron (siehe Abschnitt 2.1.3). Zur Vereinfachung der Notation entfällt der Index p im Folgenden.

Die Berechnung des Fehlersignals in den Konvolutionsschichten ist aufgrund der dimensionsreduzierenden Filter komplizierter. Das Fehlersignal ${\delta_j}^{(l)}(x,y)$ an Position (x,y) einer Merkmalskarte der Schicht l hängt von den Fehlersignalen ${\delta_k}^{(l+1)}(x,y)$ aller Merkmalskarten der Schicht $l+1$ ab. Abbildung 4.5 veranschaulicht die an der für ein Neuron der Eingabekarte j beteiligten Neuronen der Ausgabekarte k.

In dem hier vereinfachten Beispiel eines 4×4 Filters hängt die Aktivität eines Neurons der Karte k von 16 Neuronen der Karte j ab. Umgekehrt hängt das Fehlersignal eines Neurons der Karte j jedoch nur von vier Neuronen der Karte k ab. Aufgrund des Subsamplings werden unterschiedliche Filterelemente in die Berechnung einbezogen, je nachdem ob das zu berechnende Neuron in einer geraden oder ungeraden Zeile bzw. Spalte der Eingabekarte liegt. Diese Zuweisung eines Filterelements (u,v) zur korrespondierenden Position in der Ausgabekarte sei durch eine Funktion $\xi_{(x,y)}(u,v)$ gegeben. Dann berechnet sich das Fehlersignal auf der Eingabekarte j als Summe über alle Ausgabekarten K durch

$$\delta_j^{(l)}(x,y) = \sum_{k=1}^{K} \sum_{(u,v)} \delta_k^{(l+1)}(\xi_{(x,y)}(u,v)) \cdot w_{jk}(u,v) \cdot f'_{act}(net_j^{(l)}(x,y)), \tag{4.8}$$

wobei K die Anzahl der ausgehenden Merkmalskarten bezeichnet.

Aufgrund der gekoppelten Gewichte wird die partielle Ableitung des Fehlers bezüglich eines Gewichts $w_{jk}(u,v)$ nicht nur über alle Muster p aufsummiert, sondern auch über alle Bildpositionen (x,y):

$$\frac{\partial E}{\partial w_{jk}(u,v)} = \sum_{p=1}^{P} \sum_{(x,y)} \left(-\delta_k^{(l+1)}(x,y) \cdot a_j^{(l)}(2 \cdot x + u, 2 \cdot y + v) \right) \tag{4.9}$$

Der so ermittelte Gradient wird bei Backpropagation of Error zur Aktualisierung der Gewichte verwendet:

$$\Delta w_{ji}(u,v) = -\eta \cdot \frac{\partial E}{\partial w_{jk}(u,v)} \tag{4.10}$$

Auf zufällig initialisierten tiefen Netzen mit mehr als drei verdeckten Schichten konvergieren stochastische Lernverfahren nur langsam und finden häufig nur schlechte Lösungen [LBLL09]. Ein Grund dafür ist, dass die partielle Ableitung der Fehlerfunktion auf tieferen Schichten häufig deutlich kleiner ausfällt [RB92], als zur Ausgabeschicht hin. Daher ist es sinnvoll, zumindest eine Lernrate für jede Schicht zu verwenden, besser noch eine individuelle Lernrate für jedes Gewicht. Bei gekoppelten Gewichten ist es zudem notwendig eine Anpassung der Lernrate abhängig von der Anzahl der Verknüpfungen, die dieses Gewicht nutzen, vorzunehmen [LBOM98].

Um eine empirische Optimierung dieser Parameter zu vermeiden, wird daher Resilient Backpropagation (Rprop, siehe Kapitel 2.1.4) eingesetzt, so dass jedes Gewicht eine eigene, lokal adaptive Lernrate erhält. Für Rprop ist es notwendig, für jedes Gewicht zusätzlich eine eigene Lernrate sowie die Gradienten der letzten Epoche zu speichern. Aufgrund der bei einem Konvolutionsnetz im Vergleich zu den Verknüpfungen geringen Anzahl an Gewichten hält sich der zusätzliche Speicherbedarf jedoch in Grenzen.

4.3.4 Arbeitsmenge

Da der Gradient beim Online-Training in der Regel von Muster zu Muster stark fluktuiert, wird Rprop meistens in Kombination mit Offline-Training eingesetzt. Auch zur effizienten parallelen Implementierung ist es sinnvoll, mehrere oder sogar alle Trainingsmuster simultan zu verarbeiten. Durch den begrenzten globalen Speicher von CUDA-fähigen Grafikkarten (siehe Kapitel 2.4.4) ist es bei den meisten größeren Datensätzen jedoch nur möglich, einen Bruchteil der Trainingsmenge zu verarbeiten, und somit ist ein Batch-Lernverfahren ausgeschlossen.

Insbesondere für eine grobkörnige Parallelisierung ist es daher interessant, mit sogenannten Mini-Batches [WM03] zu trainieren. Dazu wird der Trainingsdatensatz in gleichgroße Teilmengen unterteilt, und eine Gewichtsaktualisierung erfolgt nach Akkumulierung des Gradienten über alle Muster eines solchen Mini-Batches. Auch bei dieser Variante, die deutlich schneller als Offline-Training konvergiert, ist der Gradient für Rprop jedoch zu instabil. Zudem ist in Hinblick auf eine effiziente, speichertransferminimierende Programmierung der GPU-Multiprozessoren ein Austausch sämtlicher Trainingsmuster im globalen Speicher nicht praktikabel.

Um also häufige große Speichertransfers zu vermeiden und gleichzeitig mit einem stabilen Gradienten zu trainieren, wird auf einer zufällig gewählten kleinen Teilmenge der Trainingsmenge trainiert und ein geringer Anteil dieser Muster jede Epoche ausgetauscht. Die Größe dieser *Arbeitsmenge* kann so gewählt werden, dass alle Muster in den globalen Speicher passen. Eine Regulierung der benötigten Speicherbandbreite erfolgt dann durch eine Anpassung des Prozentsatzes auszutauschender Muster. Bei experimentellen Ergebnissen von Behnke [Beh01] hat sich diese Kombination von Rprop mit einer austauschbaren Arbeitsmenge als

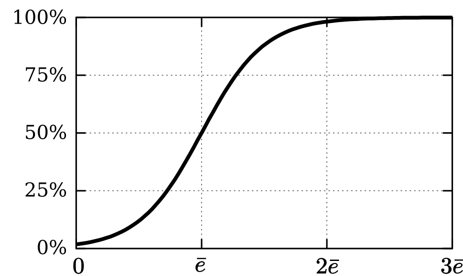


Abbildung 4.6: Gewichtetes Selektionsschema. Ein Muster, dessen Fehler e_p genau dem arithmetischen Mittel $\bar{e}$ entspricht, wird mit einer Wahrscheinlichkeit von 50% in die Arbeitsmenge aufgenommen. Bei doppelt so großem Fehler liegt die Wahrscheinlichkeit bei 98,2%; bei einem Fehler von Null liegt sie bei 1,8%.

stabil erwiesen und zudem eine Beschleunigung um zwei Größenordnungen gegenüber reinem Offline-Training ermöglicht.

Probleme können beim Training auf einer kleinen Arbeitsmenge auftreten, wenn die Lernrate so groß gewählt ist, dass die Gewichte sich zu stark an die aktuellen Muster anpassen. Insbesondere durch Rprop wird dieses Problem zusätzlich verstärkt, so dass die Gewichte überproportional wachsen. Eine Möglichkeit dies zu vermeiden und eine bessere Generalisierung zu erzwingen ist es, einen größeren Anteil der Menge auszutauschen, wodurch sich aber wiederum die benötigte Speicherbandbreite erhöht.

Um dem entgegenzuwirken, wird in dieser Arbeit daher eine Regularisierung mit Weight Decay (siehe Kapitel 2.1.4) durchgeführt. Durch die unterschiedliche Dimensionierung der Schichten muss der Decay-Faktor $d^{(l)}$ für die Schicht l in Abhängigkeit der zu einem Neuron eingehenden Verbindungen gewählt werden und wird dazu auf

$$d^{(l)} = c_{in}^{(l)} \cdot n \cdot \lambda \tag{4.11}$$

gesetzt. Dabei bezeichnet $c_{in}^{(l)}$ die Anzahl der eingehenden Merkmalskarten, $n = 65$ die Filtergröße, und der Parameter λ gibt die experimentell zu bestimmende Magnitude des Gewichtsverfalls an.

4.3.5 Gewichtetes Selektionsschema

In der Arbeitsmenge steht jede Epoche dem Lernverfahren nur ein kleiner Bruchteil der gesamten Trainingsmenge zur Verfügung, und somit wird jedes Muster nur zu wenigen Zeitpunkten des Trainings berücksichtigt. Viele Datensätze enthalten jedoch eine große Anzahl sehr ähnlicher, redundanter Muster. Wenn eine ganze Musterklasse oder eine bestimmte Variation eines Musters im Datensatz unterrepräsentiert ist, konvergiert das Trainingsverfahren bezüglich dieser Muster also nur sehr langsam und generalisiert schlechter.

In dieser Arbeit wird daher eine Methode untersucht, die Muster, welche potentiell den größten Informationsgehalt für den Lernalgorithmus haben, bevorzugt in die Arbeitsmenge aufnimmt. Ähnlich wie bei Adaboost [FS95] wird ein Trainingsmuster p anhand seines quadratischen Fehlers e_p gewichtet. Sei $\bar{e}$ das arithmetische Mittel der Fehler aller Muster, dann wird die Wahrscheinlichkeit x_p, dass dieses Muster ausgewählt wird, folgendermaßen bestimmt:

$$x_p = \left(1 + \exp\left(-\frac{4 \cdot e_p}{\bar{e}} + 4\right)\right)^{-1}. \tag{4.12}$$

Aus der Kurve in Abbildung 4.6 wird ersichtlich, dass ein Muster, dessen Fehler deutlich über dem arithmetischen Mittel $\overline{e}$ liegt, mit erheblich größerer Wahrscheinlichkeit in die Arbeitsmenge aufgenommen wird. Die Gefahr, gut gelernte Muster wieder zu verlernen, wird bei diesem *gewichteten Selektionsschema* reduziert, weil auch solche Muster mit geringerer Wahrscheinlichkeit wieder in die Arbeitsmenge aufgenommen werden.

Desaströse Auswirkungen hat diese Methode bei Datensätzen mit falsch markierten Mustern, da diese fehlerhafte Klassifizierung dann auch noch verstärkt wird. Solche Muster können aber leicht anhand ihres großen Fehlers identifiziert werden und manuell vom Training ausgeschlossen werden. Ein weiterer Nebeneffekt ist, dass die Verteilung der Eingabemuster möglicherweise verzerrt wird.

4.4 Ein- und Ausgabe

Bei der Auswahl geeigneter Eingabedaten für ein neuronales Netz wird häufig eine Dimensionsreduzierung durchgeführt, um die Berechnungskomplexität zu verringern. Umgekehrt kann die Konvergenz des Lernverfahrens aber auch beschleunigt werden, indem zusätzliche aus den Daten gewonnenene Merkmale als Eingabe verwendet werden. Im Rahmen dieser Arbeit soll daher der Einfluss manueller Vorverarbeitungen der Eingabe evaluiert werden.

4.4.1 Eingabe

Die Anzahl der benötigten Merkmalskarten auf der Eingabeschicht hängt davon ab, welche Repräsentation der Daten gewählt wird. Bei einfachen Datensätzen, wie z.B. der MNIST-Datenbank, reicht es, das Graustufenbild eines Trainingsmusters als einzige Merkmalskarte zu verwenden.

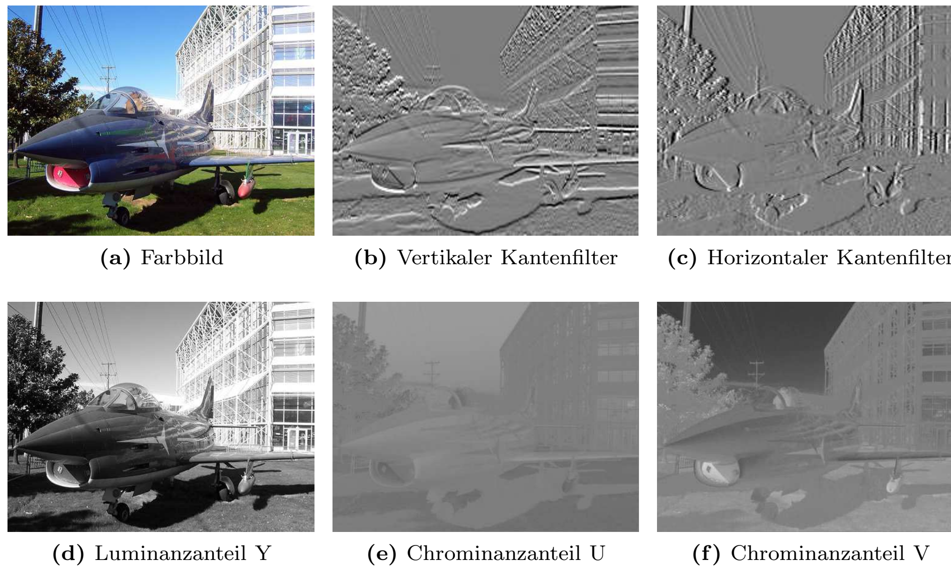
Repräsentation der Eingaben

Sofern Farbinformationen vorhanden sind, können diese wie Geusebroek *et al.* [GvdBSG01] gezeigt haben zur Beleuchtungsinvarianz und Robustheit gegenüber Schatten beitragen und somit die Klassifikation zusätzlich erleichtern. Digitale Bilder sind häufig in die drei RGB-Farbkanäle zerlegt, was eine Verwendung dieser drei Komponenten als Merkmalskarten nahelegt. Zusätzlich soll in dieser Diplomarbeit ergründet werden, ob eine andere Repräsentation – wie etwa der Y'UV-Farbraum – besser zur Objektklassifikation geeignet ist.

Siagian und Itti [SI07] haben gezeigt, dass nicht nur Intensität und Farbkanäle zur Klassifikation von Szenen geeignet sind, sondern auch die lokale Orientierung des Bildes. Mit sehr einfachen Filtern wie z.B. $[-1,0,1]$ detektierte hochfrequente Kanten verschiedener Orientierung werden daher als zusätzliche Eingabe verwendet. Abbildung 4.7 zeigt für ein farbiges Bild sowohl die Zerlegung in verschiedene Farbkanäle als auch die aus dem Intensitätsbild erzeugten Kantenbilder.

Platzierung der Eingaben und Randbehandlung

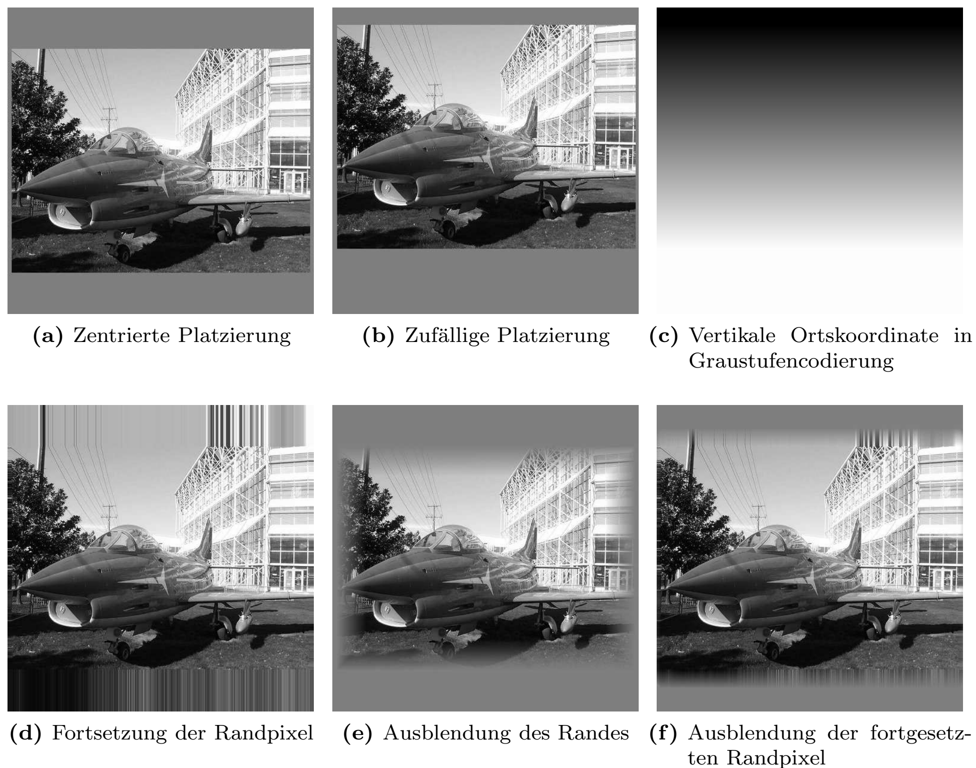
Die Seitenlänge für Merkmalskarten der letzten Konvolutionsschicht beträgt mindestens 32 Pixel, wie in Kapitel 5 begründet wird. Um zusätzliche Parameter für Breite und Höhe der Merkmalskarten zu vermeiden, wird die Größe unabhängig vom konkreten Muster oder Datensatz festgelegt. Die letzte Konvolutionsschicht ist immer 32×32 Pixel groß, daher ergibt sich durch das Subsampling um den Faktor zwei für Merkmalskarten der Eingabeschicht eine Größe von 512×512 Pixeln.



(a) Farbbild (b) Vertikaler Kantenfilter (c) Horizontaler Kantenfilter

(d) Luminanzanteil Y (e) Chrominanzanteil U (f) Chrominanzanteil V

Abbildung 4.7: Verschiedene Vorverarbeitungen des mehrfabigen Eingabebildes



(a) Zentrierte Platzierung (b) Zufällige Platzierung (c) Vertikale Ortskoordinate in Graustufencodierung

(d) Fortsetzung der Randpixel (e) Ausblendung des Randes (f) Ausblendung der fortgesetzten Randpixel

Abbildung 4.8: Alternative Möglichkeiten, mit einem für die Eingabeebene zu klein dimensionierten Bild umzugehen.

Trotz fester Größe der Eingabeschicht variiert die Auflösung der Trainingsbilder zum Teil erheblich. Zu große Bilder werden daher unter Beibehaltung des Seitenverhältnisses zunächst skaliert, so dass sie in das Eingabefeld passen. Anschließend werden die Bilder zentriert im Eingabefeld platziert und nicht belegte Pixel werden mit Null initialisiert. Dadurch treten an den Bildrändern Diskontinuitäten auf, denen auf verschiedene Art entgegengewirkt werden kann (siehe Abbildung 4.8). Eine Möglichkeit ist es, die Bilder zu den Rändern hin durch eine Reduzierung des Kontrasts auszublenden. Behnke [Beh05] berichtet, mit dieser Randbehandlung zwar gute Ergebnisse erzielt zu haben, jedoch verringert sich dadurch die effektiv nutzbare Bildfläche.

Eine Methode, nicht vorhandene Daten außerhalb des Bildes zu approximieren, ist es, die einem Punkt nächstgelegenen Werte des Bildrandes zu verwenden (siehe Abb. 4.8d). Bildkanten, die orthogonal zum Bildrand liegen, werden damit häufig sinnvoll fortgesetzt, aber insbesondere durch Rauschen am Bildrand können störende hochfrequente künstliche Kanten entstehen. Eine Kombination von fortgesetzten Randpixeln, die mit einer Kontrastreduzierung langsam ausgeblendet werden, soll daher die Diskontinuitäten an den Bildrändern entfernen, ohne aber Bildinhalt zu verlieren oder neue Diskontinuitäten einzuführen. (Abb. 4.8f).

Um größere Variationen der Eingabemuster zu erzeugen und das Lernverfahren robust gegenüber Translationen zu gestalten, kann man das Bild anstatt es zu zentrieren an einer wechselnden zufälligen Position in der Eingabekarte platzieren. Bei solchen Verschiebungen verliert das Netz die Information darüber, wo im Bild oben und unten ist. Grade die vertikale Position kann aber ein wichtiger Indikator zur Erkennung bestimmter Objekte sein: Beispielsweise befinden sich Flugzeuge häufiger in der oberen Bildhälfte. Diese Ortskoordinate wird dem Netz in Form einer eigenen Merkmalskarte zugänglich gemacht. Die vertikale Lage wird darin durch Pixelwerte zwischen -1 und +1 beschrieben.

Eingaben auf unterschiedlichen Schichten

Eine weitere Option zur Beschleunigung des Trainingsprozesses ist die Eingabe von Daten in verschiedenen Auflösungen. Einige Merkmalskarten der höheren Schichten dienen somit nur als Eingabekarte und sind nicht mit den Karten der vorherigen Schicht verbunden. Diese Vorgehensweise ist zwar nicht biologisch motiviert, erspart dem Netz aber, eine Skalierung des Bildes selbst zu lernen. Auch auf diesen niedrigdimensionalen Eingaben wird eine Kantendetektion zur Erzeugung weiterer Merkmale durchgeführt, wodurch eine Repräsentationen der niederfrequenten Bildanteile generiert wird.

4.4.2 Ausgabe

Je nach Klassifikationsaufgabe wird eine unterschiedliche Dimensionierung der Ausgabeschicht gewählt. In der Regel wird jede zu erkennende Objektklasse durch genau ein Ausgabeneuron repräsentiert. Die Zielwerte (der Teacher) werden aus dem Intervall $[-1, +1]$ gewählt und fallen somit in den Dynamikbereich des Tangens Hyperbolicus, was eine Sättigung der Aktivierungsfunktion vermeiden soll und somit die Konvergenz beschleunigt [LBOM98].

Wenn die Eingabe wie im Falle der MNIST-Datenbank nur genau ein Objekt enthalten kann, wird eine 1-aus-N-Codierung (engl. *one-hot encoding*) verwendet. Ein Neuron, welches die korrekte Objektklasse repräsentiert, ist also aktiv (Wert $+1$), alle anderen Neuronen sind inaktiv (Wert -1). Kann das Eingabebild wie beim PASCAL-Datensatz mehrere Objekte enthalten, dann wird eine M-aus-N-Codierung verwendet; jedes zu den vorhandenen Objekten korrespondierende Neuron ist also aktiv.

5 Implementierung

Einige Einschränkungen der CUDA-Schnittstelle wurden sowohl beim Entwurf der Netztopologie des Konvolutionsnetzes als auch bei der Auswahl des Lernverfahrens wie in Kapitel 4 beschrieben bereits berücksichtigt. In diesem Kapitel wird im Detail erläutert, wie diese Netzarchitektur auf GPU-Multiprozessoren parallel implementiert werden kann und welche Optimierungen zur Beschleunigung beitragen.

Zunächst erfolgt in Abschnitt 5.1 ein grober Überblick über den Funktionsumfang der verschiedenen in dieser Arbeit entwickelten Softwarekomponenten. Die Grundstruktur des Programms wird im Abschnitt 5.2 erläutert und verschiedene Möglichkeiten der Parallelisierung werden aufgezeigt. Die umfangreichen und speziell auf die CUDA-Architektur zugeschnittenen Kernelfunktionen zur Vorwärtspropagierung (Abschnitt 5.3), Rückwärtspropagierung (Abschnitt 5.5) und Gewichtsanpassung (Abschnitt 5.6) werden detailliert beschrieben. In Abschnitt 5.4 wird erläutert, wie die verschiedenen Speicherbereiche dabei effizient genutzt werden. Die Berechnungen der vollverknüpften Schichten des Konvolutionsnetzes werden in Abschnitt 5.7 kurz angerissen. Das Kapitel schließt mit einem Überblick über die zusätzlich implementierten Hilfskernel (Abschnitt 5.8) und einer Diskussion der Implementierung (Abschnitt 5.9).

5.1 Framework

Das in Kapitel 4 vorgestellte Modell eines Konvolutionsnetzes wurde im Rahmen dieser Diplomarbeit in zwei verschiedenen Varianten implementiert. Die CPU-Version nutzt keine speziellen Hardware-Optimierungen und ist auf jedem herkömmlichen PC lauffähig. Hauptaugenmerk lag auf der GPU-Version, die sämtliche Berechnungen des neuronalen Konvolutionsnetzes parallel auf GPU-Multiprozessoren ausführt. Die CPU-Version nutzt dagegen nur einen Kern des Hauptprozessors und ist somit deutlich langsamer.

5.1.1 CPU-Version

Der überwiegende Teil des Programmcodes wird sowohl von der CPU-Version als auch von der GPU-Version genutzt. Dabei handelt es sich vor allem um nicht laufzeitkritsche Unterroutinen wie z.B. zum Laden eines Datensatzes aus dem externen Speicher. Beide Versionen verwenden zudem dieselbe grafische Benutzeroberfläche (engl. *graphical user interface*, GUI). Der erhebliche Mehraufwand, den die zusätzliche Implementierung einer solchen langsameren CPU-Version verursacht, ist aus den folgenden Gründen gerechtfertigt.

Debugging Fehler in den GPU-seitig ausgeführten Kernelfunktionen sind oft nur sehr schwer aufzuspüren, da bisher kein Debugger zur Verfügung steht. Selbst einfaches Debugging durch die Ausgabe von Variablenwerten in der Kommandozeile ist nicht möglich. Debugging ist nur im sogenannten Emulationsmodus möglich, in dem CUDA-Programme auf der CPU ausgeführt werden. Durch diese sequentielle Ausführung treten jedoch viele Probleme nicht auf, wie beispielsweise gleichzeitige Schreibzugriffe. Zudem ist

der Emulationsmodus für komplexere Programme in der Regel zu langsam. Zur Fehlersuche können die Ausgaben der CPU-Version und der GPU-Version miteinander verglichen und somit die Ergebnisse kontrolliert werden.

Laufzeitanalyse Eines der Ziele dieser Arbeit ist es insbesondere, zu ergründen, ob der Mehraufwand einer parallelen Implementierung durch die Beschleunigung gegenüber einer seriellen Version gerechtfertigt ist. Der Emulationsmodus ist durch das Nachbilden der Funktionsweise der GPU-Hardware deutlich langsamer und somit für realistische Laufzeitanalysen nicht geeignet. Eine explizite CPU-Implementierung ermöglicht somit praxisnahe Laufzeitvergleiche.

Wiederverwendbarkeit Während das Trainieren eines neuronalen Netzes häufig sehr viel Rechenzeit in Anspruch nimmt, kann ein einzelnes Muster in Sekundenbruchteilen durch das Netz propagiert werden. Mit einem fertig trainerten Netz können also auch auf Systemen ohne leistungsstarke Grafikkarte in Echtzeit Muster klassifiziert werden. Beispielsweise werden auf mobilen autonomen Robotern, die häufig nur mit minimalistischer Hardware ausgestattet sind, Systeme zur Objekterkennung benötigt. In solchen Fällen könnte eine CPU-Version eingesetzt werden, welche die vortrainierten Gewichte der GPU-Version verwendet.

Eine detaillierte Beschreibung der CPU-Implementierung erfolgt hier nicht. Die Berechnungen werden in dieser Version in mehreren ineinander verschachtelten Schleifen seriell ausgeführt. Speziell optimierte CPU-Instruktionen wie z.B. der SSE-Befehlssatz werden nicht verwendet. Die CPU-Version wird mit *gcc* und der geschwindigkeitsoptimierenden Option `-O3` kompiliert, und die Laufzeitmessungen werden auf einem der Prozessorkerne eines *Intel Core i7 940* mit einer Taktfrequenz von 2,93 GHz durchgeführt.

5.1.2 Grafische Benutzeroberfläche

Die Entwicklung erfolgte unter Ubuntu Linux 8.04 in C/C++ mit der in Abschnitt 2.4.3 beschriebenen Erweiterung *C for CUDA*. Das Programm lässt sich in zwei verschiedenen Modi starten: entweder mit grafischer Oberfläche oder mit ausschließlich textueller Ausgabe über die Kommandozeile. Beide Varianten führen dieselben Berechnungen durch; im nichtgrafischen Modus werden aber sämtliche unnötigen Datentransfers zwischen Host und Device vermieden. Dieser Modus lässt zudem kaum Benutzerinteraktionen zu und ist in erster Linie für zeitintensive Berechnungen gedacht.

Die mit der freien Qt-Bibliothek[1] umgesetzte grafische Benutzeroberfläche ist in Abbildung 5.1 zu sehen. Zur Analyse des Trainingsprozesses und zur Fehlersuche werden sämtliche beim Training des Konvolutionsnetzes relevanten Daten grafisch aufbereitet:

- Aktivitäten der Merkmalskarten: $a_i(x,y)$
- Fehlersignale der Merkmalskarten: $\delta_i(x,y)$
- Konvolutionsfilter: $w_{ij}(u,v)$
- Gradienten der Gewichte: $\partial E/\partial w_{ij}(u,v)$
- Lernraten der Gewichte bei Rprop: $\eta_i(u,v)$
- Gewichte und Gradienten der vollverknüpften Schichten: w_{ij} , $\partial E/\partial w_{ij}$
- Aktivitäten und Fehlersignale der vollverknüpften Schichten: a_i , δ_i
- Ausgabe des Netzes: o_i
- gewünschte Ausgabe (Teacher): t_i

[1] `http://www.qtsoftware.com/`

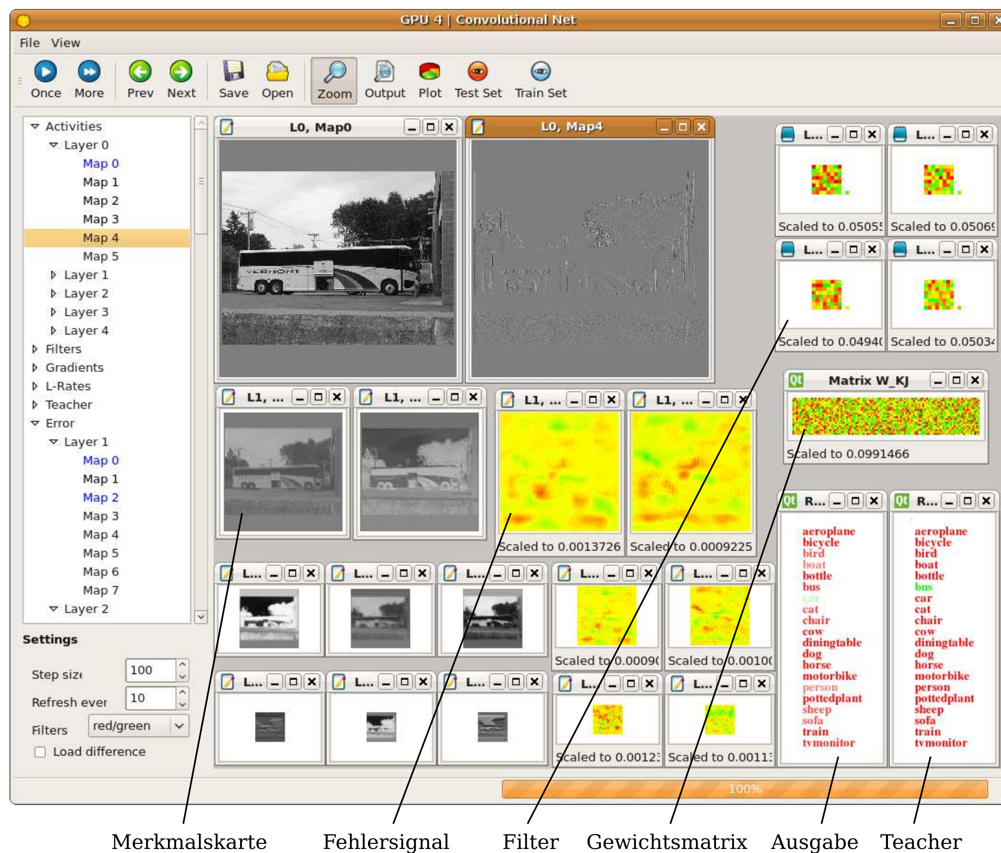


Abbildung 5.1: Grafische Benutzeroberfläche eines Konvolutionsnetzes mit einem Eingabebild der PASCAL-Datenbank. Auf der linken Seite können Merkmalskarten, Filter, Gewichte, usw. einzeln ausgewählt werden, die auf der rechten Seite dann in eigenen Fenstern angezeigt werden. Grün stellt positive Werte dar, rot negative.

Mit der Software können eine oder beliebig viele Trainingsepochen ausgeführt werden, Gewichte geladen oder gespeichert werden, und man kann zwischen der Evaluierung von Trainings- oder Testdatensatz wählen.

5.2 Parallelisierungshierarchie

In der CPU-Version erfolgen die Berechnungen des Lernverfahrens in ineinander verschachtelten Schleifen über neun verschiedene Variablen: Epochen, Trainingsmuster, Schichten, Quellkarten einer Schicht, Zielkarten der nächsten Schicht, Zeilen und Spalten der Merkmalskarte, sowie über Zeilen und Spalten des Filters. Eine serielle Ausführung der Vorwärtspropagierung im Konvolutionsnetz ist in Algorithmus 3 angegeben.

Die Epochen müssen zwangsläufig sequentiell ausgeführt werden, aber die restlichen Variablen lassen sich potentiell parallelisieren. Im Folgenden erfolgt die Analyse der verschiedenen Parallelisierungsmöglichkeiten in Hinblick auf die Vorwärtspropagierung. In Abschnitt 5.5 wird gezeigt, dass diese Überlegungen auch auf die Rückwärtspropagierung zutreffen.

Algorithmus 3 Serielle Vorwärtspropagierung

```
for Muster p = 1 to P do
  for Schichten l = 1 to L do
    for Quellkarten j = 1 to J der Schicht l + 1 do
      for Zielkarten i = 1 to I der Schicht l do
        for alle Positionen (x,y) der Zielkarte j do
          wende den Konvolutionsfilter w_ij auf die Quellkarte i an:
          for alle Elemente (u,v) des Filters w_ij do
            net_j(x,y) += net_i(2·x + u, 2·y + v) · w_ij(u,v)
          end for
        end for
      end for
      berechne die Aktivität für jede Position (x,y) aus der Netzaktivität:
      a_j = f_act(net_j)
    end for
  end for
end for
```

Wie in Abschnitt 2.4.4 erläutert, muss die Parallelisierung eines Algorithmus für die CUDA-Architektur auf zwei verschiedenen Ebenen durchgeführt werden. Zunächst muss ein Problem grobkörnig so unterteilt werden, dass die Unterprobleme unabhängig voneinander und in beliebiger Reihenfolge gelöst werden können. Diese disjunkten Teilprobleme werden dann an verschiedene Multiprozessoren zur Bearbeitung delegiert.

Ein Teilproblem muss weiter feinkörnig parallelisiert werden, indem es auf viele simultane Threads verteilt wird, die miteinander kooperieren können. Sowohl feinkörnig als auch grobkörnig bieten sich verschiedene Möglichkeiten, die serielle Vorwärtspropagierung zu parallelisieren. Diese werden im Folgenden analysiert, und es wird begründet, warum die in Abschnitt 5.3 beschriebene konkrete Parallelisierung gewählt wurde.

5.2.1 Grobkörnige Parallelisierung

Zur parallelen Ausführung auf verschiedenen Multiprozessoren werden in CUDA mehrere Threads zu einem Block zusammengefasst. Da jeder Threadblock auf genau einen Multiprozessor abgebildet wird, müssen die Blöcke unabhängig voneinander und in beliebiger Reihenfolge ausführbar sein.

Ein Datenaustausch zwischen Threads verschiedener Blöcke ist nicht möglich, und Schreibzugriffe aus verschiedenen Blöcken auf dasselbe Datenelement können unvorhersehbare Ergebnisse verursachen.

Analyse verschiedener Parallelisierungsmöglichkeiten

Für eine optimale Auslastung der Multiprozessoren ist es sinnvoll, jedem Block ein etwa gleich großes Teilproblem zuzuweisen. Eine Parallelisierung über die verschiedenen Schichten des Konvolutionsnetzes ist daher nicht praktikabel, denn zum einen verfügt jede Schicht über unterschiedlich viele Merkmalskarten und zum anderen sind diese auch noch je nach Schicht unterschiedlich groß.

Auch die Verarbeitung der Merkmalskarten derselben Schicht können nicht gleichzeitig ausgeführt werden. Denn sowohl Merkmalskarte i als auch Merkmalskarte $i + 1$ tragen zur

Berechnung einer Merkmalskarte j auf der folgenden Schicht bei. Bei gleichzeitiger Verarbeitung würde ein Konflikt entstehen, weil sowohl der für Merkmalskarte i als auch der für Merkmalskarte $i+1$ zuständige Block in dieselbe Merkmalskarte j schreiben müssten.

Um dies zu vermeiden, wäre eine grobkörnige Parallelisierung über die Ausgabe der Konvolution, also die Merkmalskarten der Schicht $l+1$ denkbar. Ein Block müsste dann sämtliche Aktivitäten aller Merkmalskarten der Schicht l laden und daraus die Aktivität der Zielkarte j der Schicht $l+1$ berechnen. Von dieser Parallelisierung wurde abgesehen, da die Merkmalskarten der Schicht l mehrfach geladen werden müssten, nämlich für jede Karte der Schicht $l+1$ mindestens einmal.

Parallelisierung über Trainingsmuster

Bei der Vorwärtspropagierung wird jedes Muster während einer Epoche mit denselben Netzparametern und Gewichten berechnet. Da diese Berechnungen völlig unabhängig voneinander sind, wurde in dieser Arbeit eine grobkörnige Parallelisierung über die Trainingsmuster vorgenommen. Jedem Block werden also die Aktivitäten eines bestimmten Musters zugewiesen.

Der grobkörnigen Trainingsmusterparallelisierung sind dennoch Grenzen gesetzt, denn die Aktivitäten sämtlicher simultan berechneten Trainingsmuster müssen im globalen Speicher der GPU abrufbar sein. Bei 32 Bit Genauigkeit benötigt eine 512×512 Pixel große Merkmalskarte auf der Eingabeschicht $515 \times 512 \cdot 4$ Byte (=1 MB) Speicherplatz. Auch die Aktivitäten der höherschichtigen Merkmalskarten müssen für die Rückwärtspropagierung im globalen Speicher abgelegt sein. Je nach Anzahl der Karten pro Schicht und der Größe der Karten beansprucht ein Muster also 10 MB oder sogar mehr, daher können im 1 GB großen globalen Speicher nur einige Dutzend Muster simultan gespeichert werden.

Ein reines Offline-Training bei dem sämtliche Muster parallel verarbeitet werden ist also nicht möglich. Stattdessen liegt nur ein Teil der Trainingsmuster – die in Abschnitt 4.3.4 beschriebene Arbeitsmenge – im globalen Speicher und wird sukzessive ausgetauscht.

Wie in Abschnitt 2.4.5 erläutert wurde, können Speicherlatenzen verborgen werden, wenn die Anzahl der aktiven Blöcke die Anzahl der verfügbaren Multiprozessoren um ein Vielfaches übersteigen. Bei einer zu kleinen Arbeitsmenge würde die GPU-Hardware somit nicht effizient ausgenutzt. Die in dieser Arbeit eingesetzte GeForce GTX 285 verfügt über 30 Multiprozessoren, demnach muss für eine optimale Ausnutzung über mindestens 30 Trainingsmuster parallelisiert werden. Ein Vielfaches dessen wäre empfehlenswert, ist aber bei großen Netzen nicht immer möglich.

5.2.2 Feinkörnige Parallelisierung

Jeder Block besteht aus bis zu 512 Threads. Diese können über den Shared Memory miteinander kommunizieren, was eine feinkörnige kooperative Parallelisierung ermöglicht. Aufgrund der begrenzten Anzahl an Threads pro Block ist es dennoch notwendig zu analysieren, welche Variablen zur Parallelisierung am besten geeignet sind.

Durch die im vorherigen Abschnitt begründete Entscheidung, jedem Threadblock ein Muster zuzuordnen, ist die Berechnung der Konvolutionen zwischen zwei Schichten noch nicht geklärt. Gegeben seien die Aktivitäten der I Quellkarten einer Schicht l, aus denen mit Hilfe der Konvolutionsfilter ${w_{ij}}^{(l)}$ die Aktivitäten der J Zielkarten der folgenden Schicht $l+1$ berechnet werden sollen. Es müssen also $I \times J$ Faltungen von $n \times n$ großen Merkmalskarten mit einem 8×8 Filter durchgeführt werden.

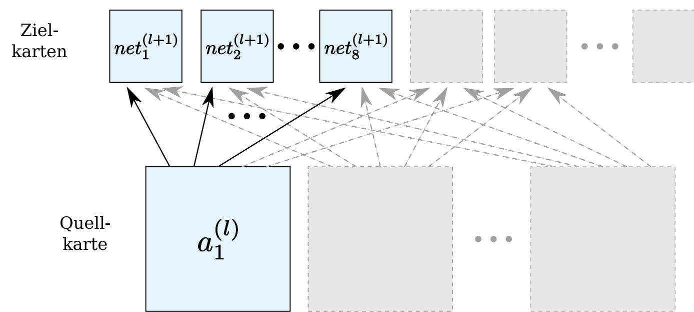


Abbildung 5.2: Der Vorwärtskernel berechnet die Konvolution einer Quellkarte der Schicht l zu acht Zielkarten der Schicht $l + 1$.

Eine Möglichkeit bestünde darin, die Faltungen nacheinander durchzuführen. Jeder Thread könnte dann jeweils ein Pixel der Zielkarte berechnen. Dieser naive Ansatz wird zunächst in Abschnitt 5.3 erläutert und dient als Grundlage für das implementierte Verfahren. Er ist jedoch suboptimal, weil die Aktivitäten einer Quellkarte für jede Faltung erneut geladen werden müssen.

Das andere Extremum wäre es, sämtliche Merkmalskarten parallel zu verarbeiten und über die Pixel der Quellkarte zu iterieren. Vorteilhaft wäre in diesem Fall, dass für jedes Pixel der Zielkarten nur ein einziger Schreibzugriff erforderlich wäre. Dazu müsste allerdings mindestens ein 8×8 Pixel großer Bildausschnitt jeder Merkmalskarte zur Verfügung stehen. Da dies nur mit häufigen Speichertransfers möglich wäre und zudem jedes Quellpixel sehr häufig geladen werden müsste, ist diese Option nicht praktikabel.

Einen Kompromiss stellt die in dieser Arbeit entworfene Kernelfunktion dar. Wie Abbildung 5.2 veranschaulicht, führt diese Funktion simultan die Faltung von einer Quellkarte auf acht verschiedene Zielkarten durch. Bei dieser Variante muss also eine Quellkarte für die Berechnung von acht Zielkarten nur einmal geladen werden. Die genaue Funktionsweise dieser Implementierung wird im folgenden Abschnitt erläutert und deren Vor- und Nachteile werden aufgezeigt.

5.3 Vorwärtspropagierung

Aufgabe des Vorwärtspropagierungskernels (*Forward-Kernel*) ist es, aus den Aktivitäten ${a_i}^{(l)}(x,y)$ der Merkmalskarten einer Eingabeschicht l die Netzeingabe ${net_j}^{(l+1)}(x,y)$ der Merkmalskarten der Ausgabeschicht $l + 1$ zu berechnen. Die Netzeingabe einer Karte j an der Position (x,y) berechnet sich durch

$$net_j^{(l+1)}(x,y) = \sum_{i=0}^{I} \sum_{(u,v)} w_{ij}(u,v) \cdot a_i(2x + u,2y + v), \tag{5.1}$$

wobei (u,v) alle Zeilen und Spalten des Konvolutionsfilters durchlaufen. Im Folgenden wird eine parallele Implementierung dieser Konvolutionen erläutert.

5.3.1 Naiver Ansatz

Zum besseren Verständnis der Gründe für den komplizierteren Forward-Kernel wird hier zunächst in Anlehnung an das Whitepaper zu Bildkonvolution von Nvidia [NP07] eine nicht

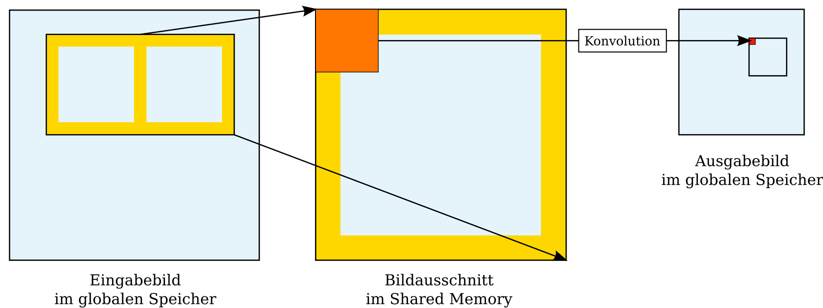


Abbildung 5.3: Naive parallele Konvolution. Links: Jeder Threadblock lädt einen Bildausschnitt aus dem globalen Speicher in den Shared Memory. Mitte: Ein Thread wendet den Konvolutionsfilter auf ein 8×8 Pixel großes Fenster an. Rechts: Anschließend schreibt der Thread die Summe in ein Pixel des Ausgabebildes. (adaptiert nach [NP07])

optimierte parallele Implementierung einer Konvolution beschrieben. Gegeben sei ein $2n \times 2n$ Pixel großes Eingabebild, das mit einem 8×8 Filter auf eine $n \times n$ Pixel große Ausgabe abgebildet werden soll.

Eine einfache Herangehensweise ist es, nur einen Block des Eingabebildes aus dem globalen Speicher in den Shared Memory der Grafikkarte zu laden. Auf diesen dann im schnellen Speicher verfügbaren Pixeldaten wird eine punktweise Multiplikation mit den Filterelementen durchgeführt. Anschließend wird das aufsummierte Skalarprodukt in das Ausgabebild im globalen Speicher geschrieben. Abbildung 5.3 illustriert diesen Ansatz. Somit verarbeitet jeder Threadblock einen Bildausschnitt und jeder Thread berechnet ein Pixel des Ausgabebildes.

An den Rändern eines Blocks ragt der Konvolutionsfilter über den eigentlich zu verwendenden Bildausschnitt hinaus. Dabei werden einige Pixel benötigt, die eigentlich zu einem benachbarten Block gehören. Diese Pixel des Randstreifens (in Abbildung 5.3 gelb dargestellt) müssen also mehrfach geladen werden, nämlich für jeden angrenzenden Block erneut. Aufgrund der Randpixel ist dieser Ansatz in mehrerlei Hinsicht nicht optimal.

Die Pixel sollen so geladen werden, dass jeder einem Pixel im Ausgabebild entsprechende Thread vier Pixel des Eingabebildes lädt. Auf den Randpixeln zentriert muss jedoch kein Konvolutionsfilter angewandt werden, so dass diese Threads während der Berechnung nicht aktiv wären. Ein nicht unerheblicher Teil der Rechenkapazität wird also verschwendet.

Da die Randpixel von mehr als einem Block geladen werden, sind zudem einige Datentransfers redundant. Angenommen ein Threadblock berechnet einen 16×16 Pixel großen Ausschnitt des Ausgabebildes, dann entspricht dies einem 38×38 Pixel bzw. 5,6 KB großen Ausschnitt des Eingabebildes. Da 420 von 1444 zu ladenden Pixeln somit Randpixel sind, muss jedes Pixel mindestens 1,4-mal geladen werden. Ein deutlich größerer Ausschnitt würde jedoch nicht in den 16 KB großen Shared Memory passen.

Dieser Ansatz ist also schon für eine einzelne Konvolution ungeeignet. Im Konvolutionsnetz soll ein Eingabebild aber sogar mehrfach gefaltet werden. Der im nächsten Abschnitt beschriebene Forward-Kernel nutzt diesen Umstand aus, um das mehrfache Laden der Eingabepixel zu reduzieren.

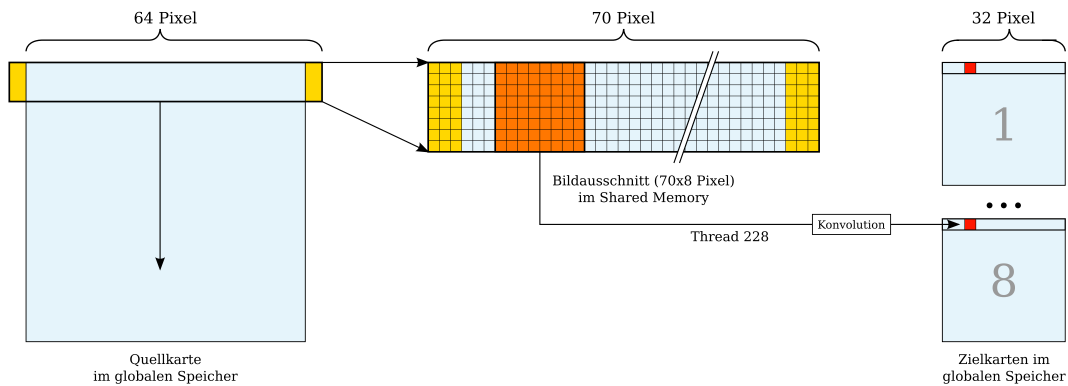


Abbildung 5.4: Konvolution mit Ringpuffer. Links: Ein Bildausschnitt der Größe 70×8 Pixel wird aus dem globalen Speicher in den Shared Memory geladen. Mitte: In diesem Ausschnitt lässt sich der 8×8 Konvolutionsfilter an 32 Positionen anwenden. Rechts: Jeder Thread schreibt das Ergebnis der Faltung an die jeweilige Position in einer von acht Zielkarten im globalen Speicher.

5.3.2 Konfiguration des Forward-Kernels

Zur Beschreibung des Forward-Kernels wird zunächst von dem speziellen Fall ausgegangen, dass die Quellkarten 64×64 Pixel groß sind. Auch die Behandlung von Randproblemen und Biasgewichten wird zunächst ausgeklammert. Am Ende dieses Kapitels werden diese Sonderfälle getrennt betrachtet.

Gegeben seien I Quellkarten der Schicht l, die durch Faltung mit $I \times J$ Konvolutionsfiltern auf J Zielkarten der Schicht $l + 1$ abgebildet werden sollen. Der Forward-Kernel betrachtet jedoch nicht wie in Gleichung 5.1 eine feste Zielkarte j und summiert dann über die Quellkarten. Stattdessen wird eine Quellkarte als fest angesehen und deren Anteil an der Netzeingabe $net_j(x,y)$ der Zielkarten berechnet. Die Gesamtsumme der Netzeingaben wird also über alle Kernelaufrufe akkumuliert.

Jeder Threadblock bearbeitet ein eigenes Trainingsmuster. Wie in Abbildung 5.2 zu sehen, führt ein Aufruf des Kernels die Konvolution von einer 64×64 Pixel großen Quellkarte i auf die acht 32×32 Pixel großen Zielkarten j bis $j+7$ aus. Um die akkumulierte Netzeingabe zu berechnen, muss der Kernel also $\lceil J/8 \rceil$-mal für jede der I Quellkarten ausgeführt werden.

Als Eingabe erhält der Forward-Kernel einen Pointer auf die Aktivitäten der i-ten Merkmalskarte der Schicht l, die im globalen Speicher abgelegt sind. Für die Ausgabe wird dem Kernel ein Pointer auf den Akkumulator für die Netzeingabe der Merkmalskarten j bis $j+7$ der Schicht $l + 1$ im globalen Speicher übergeben. Des Weiteren erhält er die acht für die Konvolution benötigten Filter.

5.3.3 Konvolutionsalgorithmus

Wie die naive Implementierung gezeigt hat, schränkt vor allem der begrenzte Shared Memory die Möglichkeiten einer parallelen Implementierung ein. Zentrale Idee dieser optimierten Implementierung ist es daher, wie im naiven Ansatz zwar nur einen Ausschnitt des Eingabebildes im Shared Memory zu halten, diesen aber nicht immer komplett auszutauschen.

Mehrfache Verwendung geladener Bildpixel

Beim Aufruf des Kernels wird zunächst ein 70×8 großer Ausschnitt der Quellkarte in den Shared Memory geladen. Dieses Fenster ist genauso hoch wie der $8{\times}8$ Konvolutionsfilter und breit genug um die gesamte Breite des Bildes zuzüglich drei Randpixeln auf jeder Seite abzudecken (siehe Abbildung 5.4). Der Speicherbedarf dieses $70 \times 8 = 560$ Pixel großen Bildausschnitts ist relativ gering – er nimmt lediglich 2240 Byte des Shared Memory ein. Mit jeweils zwei Pixeln Abstand passt der Konvolutionsfilter somit genau 32 Mal in den Bildausschnitt, so dass die gesamten 32 Pixel einer Zeile der Zielkarte erzeugt werden können.

Die Pixel einer Zeile der Zielkarte können somit von 32 Threads simultan berechnet werden. Wie in Abschnitt 2.4.5 erläutert, bringt eine solch geringe Anzahl an Threads pro Block jedoch erhebliche Performance-Einbußen mit sich. Daher werden die Daten im Shared Memory nicht nur für die Abbildung auf eine einzige Zielkarte verwendet, sondern es werden gleich acht Zielkarten parallel berechnet. Die Anzahl der Threads erhöht sich somit auf $8 \times 32 = 256$ bei gleichbleibendem Speicherbedarf des Shared Memorys. Zugleich wird damit eines der Entwurfsziele erreicht: Die Pixel der Quellkarten müssen nämlich nur einmal geladen werden, können aber für acht Zielkarten wiederverwendet werden.

Bildausschnitt als Ringpuffer

Größtes Problem der naiven Implementierung ist, dass die Randpixel für benachbarte Bildausschnitte jedes Mal erneut geladen werden müssen. Wie können diese unnötigen Speichertransfers verhindert werden, wenn der angrenzende Bildausschnitt berechnet werden soll? Die Lösung liegt darin, nicht den kompletten Inhalt des Shared Memory auszutauschen, sondern nur die neu benötigten Zeilen nachzuladen.

Die zur Berechnung der zweiten Zeile der Zielkarte benötigten Pixel der Quellkarte sind nämlich größtenteils noch im Shared Memory vorhanden. Eine Verschiebung um eine Zeile in der Zielkarte entspricht einer Verschiebung von zwei Zeilen in der Quellkarte; somit überlappen sich die rezeptiven Felder der Zeilen zu 75%.

Das Bildfenster wird daher als Ringpuffer implementiert. Wenn das Fenster von einer Zeile zur nächsten wandert, werden die ältesten Zeilen im Shared Memory entfernt und durch zwei neue Zeilen ersetzt. Es müssen also lediglich $70 \times 2 = 140$ Pixel der Quellkarte aus dem globalen Speicher neu geladen werden. Dieser Ladevorgang wird von 140 der 256 aktiven Threads erledigt, so dass jeder Thread entweder genau ein Pixel lädt oder während des Ladevorgangs ruht.

Bei einer $64{\times}64$ Pixel großen Quellkarte wird also jedes Bildelelement für acht Konvolutionen nur ein einziges Mal geladen. Durch die überlappenden Filter wird ein Datenelement für bis zu 16 Multiplikationen pro Zielkarte verwendet. Über alle acht Zielkarten kommen auf den Ladevorgang eines Datums also mindestens 128 arithmetische Operationen. Selbst unter Berücksichtigung der während des Ladevorgangs ruhenden Threads ist die arithmetische Dichte (siehe Seite 27) dieses Teils der Implementierung außerordentlich hoch.

Schritte des Forward-Kernels

Der Konvolutionsalgorithmus durchläuft also alle 32 Zeilen der Zielkarten (bzw. je zwei Zeilen der $64{\times}64$ Pixel großen Quellkarte) iterativ und parallelisiert innerhalb einer Zeile über alle 32 Spalten der Zielkarten. Während jeder Iteration werden drei Schritte ausgeführt: Laden der Quellpixel aus dem globalen Speicher, Berechnung der Konvolution und Speichern des Skalarprodukts in den Zielkarten.

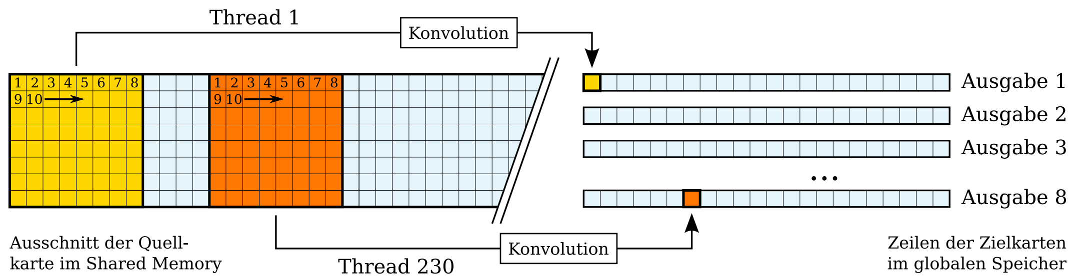


Abbildung 5.5: Jeder Thread berechnet die Konvolution an einer von 32 Positionen für eine der acht Zielkarten. Dazu iteriert er über alle 64 Elemente des Filters und multipliziert ein Filterelement mit der korrespondierenden Aktivität der Quellkarte.

Laden der Quellpixel Als erstes werden die Pixel der nächsten zwei Zeilen aus dem globalen Speicher in den 70 Pixel breiten Ringpuffer geladen. 140 Threads eines Blocks führen diese Ladeoperation durch, die restlichen 116 Threads sind derweil inaktiv. Die Bildpixel sind zeilenweise als 32 Bit Variablen von Typ `float` abgelegt, also kann dieser Ladevorgang für die 32 aufeinanderfolgenden Threads eines Warps als eine zusammenhängende Speichertransaktion mit der maximal möglichen Geschwindigkeit erfolgen (siehe Abschnitt 2.4.5).

Die 116 Threads, die keine Ladeoperationen durchführen, werden so weit wie möglich zu eigenständigen Warps zusammengefasst und haben somit kaum negative Auswirkungen auf die Ladegeschwindigkeit.

Berechnung der Konvolution Den Hauptteil der Kernellaufzeit macht die Berechnung des Skalarprodukts zwischen Bildausschnitt und Konvolutionsfiltern aus, daher werden hierzu alle 256 Threads beansprucht. Abbildung 5.5 zeigt, wie jeder Thread in der selben Reihenfolge über alle 64 Elemente des Filters iteriert und das Skalarprodukt $net_j = \sum_{(u,v)} w(u,v) \cdot a_i$ zunächst in einem Register akkumuliert. Da die Threads mit je zwei Pixel Abstand auf den Shared Memory zugreifen, entsteht ein zweifacher Bankkonflikt innerhalb eines Warps. Je 32 Threads berechnen unterschiedliche Spalten derselben Zielkarte, somit verwenden z.B. Thread 1 und Thread 33 simultan dasselbe Datenelement aus dem Shared Memory. Dennoch kommt es zu keinen Performanceeinbußen durch Bankkonflikte, da die Threads unterschiedlichen Warps angehören.

Beim Zugriff auf die im Konstantenspeicher abgelegten Filterelemente verhält es sich genau umgekehrt: Alle Threads eines Warps verwenden denselben Filter w_{ij}, da sie eine bestimmte Ausgabekarte j berechnen. Da die Threads in einer festen Reihenfolge über die Filterelemente iterieren, wird 32 Mal simultan dasselbe Filterelement benötigt. Im Gegensatz zum Shared Memory unterliegt der Konstantenspeicher aber keinen Zugriffsbeschränkungen, sondern ein mehrfacher Zugriff auf dieselben Daten wirkt sich positiv auf die Zugriffszeiten aus, da weniger Cache-Misses auftreten.

Speichern des Skalarprodukts Nachdem über alle 64 Filterelemente iteriert wurde, hat jeder Thread das Skalarprodukt in einem Register akkumuliert. Je 32 Threads eines Warps haben die Netzeingabe für eine 32 Pixel breite Zeile einer Zielkarte berechnet. Diese zusammenhängenden Pixeldaten können nun koaleszierend im globalen Speicher abgelegt werden, so dass bei diesem Schreibzugriff keinerlei Zugriffskonflikte entstehen. Weder zwischen Threads verschiedener Warps noch zwischen unterschiedlichen

	Seriell	Parallel
Host	Epochen Schichten Quellkarten (Zielkarten)	Muster (min. 30)
Device	Bildzeilen (Bildspalten) Filterzeilen Filterspalten	Bildspalten (je 64) Zielkarten (je 8)

Abbildung 5.6: Parallelisierung der Iterationsvariablen. Sowohl auf dem Host als auch innerhalb des Kernels werden einige Variablen seriell durchlaufen. In Klammern: Blöcke von 8 Zielkarten bzw. 64 Bildspalten werden seriell iteriert; nur innerhalb dieser Blöcke wird parallelisiert.

Blöcken kommt es zu Konflikten, denn unterschiedliche Warps schreiben in unterschiedliche Zielkarten und unterschiedliche Blöcke verarbeiten unterschiedliche Muster, die unterschiedliche Speicherbereiche haben.

Nachdem der Kernel auf sämtlichen 32 Zeilen der Zielkarten diese Berechnungsschritte ausgeführt hat, ist der Anteil der Quellkarte an der Netzeingabe der acht Ausgabekarten vollständig berechnet.

5.3.4 Anmerkungen zum Forward-Kernel

Der oben beschriebene Forward-Kernel nutzt viele der Optimierungsmöglichkeiten der CUDA-Schnittstelle. Im folgenden Abschnitt werden die Überlegungen dazu näher beleuchtet und einige Vor- und Nachteile der Implementierung erläutert.

Serielle und parallele Variablen

Aufgrund verschiedener Faktoren ist es nicht möglich, über alle Variablen der Vorwärtspropagierung zu parallelisieren. Unter anderem spielen dabei die Begrenzung der Threads, der geringe Speicher und die Einschränkungen der CUDA-Architektur eine Rolle. Deshalb werden sowohl auf CPU-Seite als auch innerhalb eines Threads einige Variablen seriell iteriert. Abbildung 5.6 stellt die serielle und parallele Ausführung der neun Iterationsvariablen gegenüber.

Je kleiner die Muster eines Datensatzes, desto mehr Muster können im globalen Speicher abgelegt werden, und desto stärker kann über die Trainingsmuster parallelisiert werden. Ab ca. 60 parallelen Mustern sind die Multiprozessoren optimal ausgereizt. Eine andere Parallelisierung, zum Beispiel über die Quellkarten der Konvolution ist nicht möglich, da es zu Konflikten kommen würde, wenn verschiedene Blöcke auf dieselbe Ausgabekarte schrieben.

In Hinblick auf die in Abschnitt 2.4 erwähnten unterschiedlichen Formen der Parallelisierung wird bei dieser Implementierung des Konvolutionsnetzes die Trainingsmusterparallelität am stärksten ausgenutzt. Es findet jedoch auch eine Parallelisierung über die Neuronen statt, die hier den Pixeln der Merkmalskarten entsprechen. In gewisser Weise, nämlich in Bezug auf die Verbindungen zwischen der Quellkarte und den acht Zielkarten, kann man auch von Kantenparallelität (siehe Abschnitt 2.4.1) sprechen.

Größe der Fenster und der Merkmalskarten

Der Ringpuffer für den Bildausschnitt wurde bewusst relativ klein gewählt und verbraucht für die $70 \times 8 = 560$ Pixel nur 2240 Byte des Shared Memory. In Hinblick auf den 16 KB großen Speicher können somit theoretisch bis zu sieben Threadblöcke auf einem Multiprozessor aktiv sein. Praktisch wird diese Zahl auf höchstens vier Threadblöcke begrenzt, da jeder Thread mindestens 16 Register beansprucht.

Ein größerer Bildausschnitt erscheint zunächst vorteilhaft, da somit mehr Pixel der Zielkarten gleichzeitig berechnet werden könnten. Dies wäre aber nur mit mehr Threads möglich, was bei gleichbleibender Anzahl an Registern pro Thread jedoch zur Folge hätte, dass die Auslastung der Multiprozessoren wieder sinken würde. Statt vier Threadblöcken könnten nur noch drei Threadblöcke von einem Multiprozessor simultan verarbeitet werden.

Ein 70 Pixel breites Fenster der Quellkarte entspricht einem 32 Pixel breiten Streifen der Zielkarte. Das hat zur Folge, dass die Merkmalskarten der letzten Konvolutionsschicht ebenfalls 32 Pixel breit sein müssen. Wenn dagegen ein größerer Bildausschnitt gewählt werden würde, müsste das Netz insgesamt mit größeren Merkmalskarten arbeiten, was negative Auswirkungen auf die Abstraktionsfähigkeit des Netzes hätte.

Die Berechnung von kleineren z.B. 16×16 Pixel großen Merkmalskarten ist mit diesem Kernel nur möglich, wenn die Hälfte der Threads inaktiv ist und wäre somit ineffizient. In der Beschränkung auf Dimensionen, die ein Vielfaches von 32 Pixeln betragen liegt der Vorteil, dass ein Warp in der CUDA-Architektur genau 32 Threads umfasst. Somit werden bei den rechenintensiven Operationen alle 32 Threads eines Warps ausgelastet und können zusammenhängend auf die verschiedenen Speicherbereiche zugreifen.

Abwägung zwischen Lade- und Speicheroperationen

Der Kernel wurde so entworfen, dass ein häufiges Laden der Quellkarten aus dem globalen Speicher vermieden wird. Stattdessen werden für möglichst viele Berechnungen die Daten aus dem Shared Memory verwendet. Aufgrund der überlappenden Filter wird ein in den Shared Memory geladenes Pixel bis zu 16 Mal für jede der acht Zielkarten verwendet, insgesamt also bis zu 128 Mal.

Eine Alternative zu dem hier implementierten Kernel wäre es, nicht mehrere Zielkarten gleichzeitig zu betrachten, sondern nur eine einzige. Parallelisiert werden könnte dann über mehrere Quellkarten. Diese Herangehensweise wurde aus mehreren Gründen verworfen. Betrachtet man nur die Konvolution einer Quellkarte auf eine Zielkarte, dann wird jeder Pixel der Quellkarte 16 Mal gelesen, jeder Pixel der Zielkarte aber nur einmal geschrieben. Wenn simultan die Abbildung mehrerer Quellkarten berechnet würde, müssten auch mehrere Bildausschnitte in den Shared Memory geladen werden, was aufgrund dessen geringer Größe kaum möglich ist.

Außerdem lassen sich schreibende Zugriffe auf die Zielkarten im Gegensatz zu den Lesezugriffen auf den Quellkarten nicht parallel durchführen. Für einen Schreibvorgang werden deutlich mehr Berechnungen durchgeführt, als für einen Lesevorgang erforderlich sind.

5.4 Strukturierung des Speichers

Die Verwaltung und Strukturierung der unterschiedlich großen und schnellen Speicherbereiche der CUDA-Architektur kann die Effizienz einer Implementierung gravierend beeinflussen. Durch eine geschickte Abstimmung der Threadreihenfolge und der Anordnung der

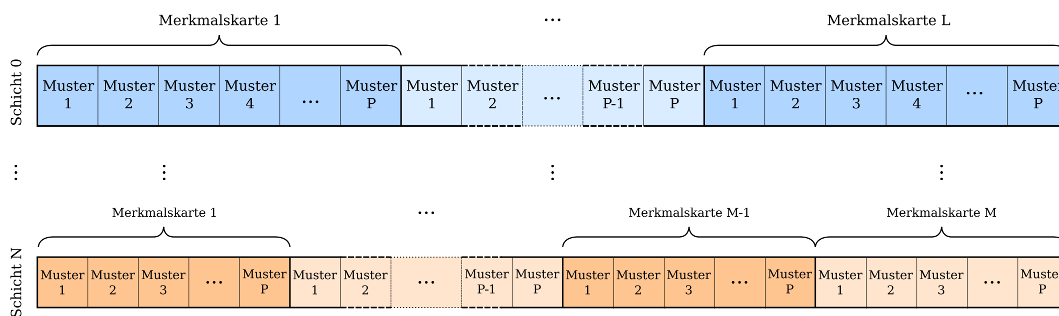


Abbildung 5.7: Anordnung der Pixeldaten im Speicher. Die Unterteilung erfolgt primär nach Schichten, sekundär nach Merkmalskarten und tertiär nach Mustern. Die Daten eines Musters p in einer Merkmalskarte i der Schicht l sind zeilenweise angeordnet.

Daten können zusammenhängende Speicherzugriffe erzwungen werden, die konfliktfrei und deutlich schneller sind.

Ziel der Implementierung ist es zudem, Speichertransfers so gut wie möglich zu minimieren, da zum einen die Bandbreite zwischen Host und Device beschränkt ist und zum anderen spezielle Beschränkungen für Zugriffe auf die Daten im globalen Speicher gelten (siehe Abschnitt 2.4.5). Des Weiteren sollen sämtliche schnellen Speicherbereiche, also vor allem die Register, der Shared Memory und der Konstantenspeicher, so gut wie möglich ausgenutzt werden.

5.4.1 Aktivitäten und Fehlersignale

Wie bereits in Abschnitt 5.2.1 erwähnt, schränkt die geringe Größe des globalen Speichers die Parallelisierung über mehrere Muster stark ein. Mit 1 GB globalem Speicher ist die GTX 285 eine der am besten ausgestatteten CUDA-fähigen GeForce-Grafikkarten. Dennoch fasst dieser Speicher nur die Aktivitäten für ein paar Dutzend Muster. Außer den Aktivitäten der Eingabekarten ist es für die Rückwärtspropagierung nämlich notwendig, auch sämtliche Aktivitäten der nachfolgenden Merkmalskarten im Speicher zu halten.

Bei einem Netz mit vier Eingabekarten der Größe 512×512 Pixel und jeweils 8, 16, 32 und 64 Merkmalskarten in den Schichten L1 bis L4 entspricht dies einer Datenmenge von ca. 8 MB für ein einziges Muster. Die für die Berechnung der Gewichtsanpassungen benötigten Fehlersignale einer Merkmalskarte benötigen nochmals genauso viel Speicher für jedes Muster. Durch die im nächsten Abschnitt erläuterte Zwischenspeicherung in einem temporären Puffer kann dieser Speicherbedarf jedoch stark reduziert werden. Sowohl Aktivitäten als auch Fehlersignale nutzen dieselbe, im folgenden Abschnitt beschriebene Speicheranordnung, daher wird zusammenfassend für beides auch der Begriff *Pixeldaten* verwendet.

Speicheranordnung

Die Strukturierung der Daten im Speicher ist auf CPU-Seite und auf GPU-Seite identisch. Somit ist es möglich, die benötigten Daten mit geringem Overhead blockweise mit einer Speichertransaktion zu kopieren. Bei der laufzeitoptimierten GPU-Version ohne Visualisierung (siehe Abschnitt 5.1.2) sind bis auf die Eingabeaktivitäten sämtliche Merkmalskarten und Fehlersignale nur im globalen Speicher der GPU abgelegt.

Die Anordnung im Speicher erfolgt so, dass gleichzeitig benötigte Daten in zusammenhängenden Speicherbereichen abgelegt sind. Abbildung 5.7 zeigt den schematischen Aufbau des Speichers für die Pixeldaten sämtlicher Schichten und Muster.

Eine grobe Unterteilung erfolgt zunächst nach unterschiedlichen Schichten, da verschiedene Schichten nicht parallel verarbeitet und zu jedem Zeitpunkt nur die Daten von zwei benachbarten Schichten benötigt werden. Innerhalb des Speicherbereichs einer Schicht erfolgt die Unterteilung dann nach Merkmalskarten. Für sämtliche Muster sind die Pixeldaten einer Merkmalskarte also in einem großen zusammenhängenden Block abgespeichert.

Erst innerhalb eines solchen Merkmalskarten-Blocks erfolgt eine feinere Einteilung in verschiedene Muster. Für ein Muster sind die Pixeldaten – wie bei den meisten Grafikformaten üblich – zeilenweise abgelegt.

Begründung

Auf den ersten Blick mag es als problematisch erscheinen, dass die zu einem bestimmten Muster gehörenden Daten in völlig unterschiedlichen Speicherbereichen abgelegt sind. Zum Austauschen einiger Muster der Arbeitsmenge muss jedoch nur die Eingabeschicht betrachtet werden, die typischerweise nur aus wenigen Merkmalskarten (ca. 1 bis 4) besteht. Um mehrere aufeinanderfolgende Muster auszutauschen, reichen also dementsprechend viele Speichertransaktionen. Ein Löschen der Aktivitäten der ausgetauschten Muster auf den höheren Schichten ist nicht nötig; diese können einfach überschrieben werden.

Im Lernverfahren selbst werden zu keinem Zeitpunkt sämtliche Pixeldaten eines Musters von allen Schichten und allen Merkmalskarten benötigt. Man könnte denken, dass der Multiprozessor, der ein konkretes Muster verarbeitet, alle diesem Muster zugeordneten Aktivitäten einer Schicht laden muss. Da der in Abschnitt 5.3 beschriebene Kernel jedoch immer nur eine einzige Merkmalskarte betrachtet, reicht es völlig aus, wenn die Daten dieser Merkmalskarte zusammenhängend gespeichert sind.

Intuitiv würde man den Speicher innerhalb einer Schicht nicht zu Blöcken verschiedener Merkmalskarten zusammenfassen, sondern alle Daten eines Musters blockweise zusammenfassen. Bewusst wurde aber das hier genannte Schema gewählt um die beim Kernelaufruf benötigten Parameter – und somit die benötigten Register – zu reduzieren. Dem Kernel wird nämlich ein Pointer auf das Muster 1 der ersten von acht aufeinanderfolgenden Merkmalskarten übergeben. Bei $n \times n$ großen Merkmalskarten kann auf die Pixeldaten eines konkreten Musters p mit dem Offset $(n \times n) \times p$ zugegriffen werden. Da die Anzahl der Muster P eine Konstante ist, kann mit dem Offset $(n \times n) \times (P \times i + p)$ außerdem auf die folgenden i Merkmalskarten zugegriffen werden. Im Gegensatz zur stets bekannten Anzahl der Muster ist dem Kernel die Anzahl der Merkmalskarten einer Schicht nicht bekannt. Bei primärer Sortierung nach Mustern wäre dieser zusätzliche Parameter und somit ein zusätzliches Register nötig.

Für die Visualisierung ist die Speicherstruktur weniger gut geeignet, denn meistens möchte der Benutzer auf alle relevanten Daten zu einem konkreten Muster direkt zugreifen. Da die grafische Aufbereitung allerdings nicht laufzeitkritisch ist, können dazu mehrfache Speichertransaktionen in Kauf genommen werden.

5.4.2 Temporärer Zwischenspeicher zur Berechnung der Fehlersignale

Wie bereits erwähnt, kann es vermieden werden, dass sowohl Aktivitäten als auch Fehlersignale einer Schicht gespeichert werden müssen. Bei der Rückwärtspropagierung berechnet sich das Fehlersignal ${\delta_i}^{(l)}$ einer Merkmalskarte i in Schicht l aus den Aktivitäten ${a_j}^{(l)}$ derselben Schicht und aus den Fehlersignalen ${\delta_j}^{(l+1)}$ aller Merkmalskarten j der folgenden Schicht

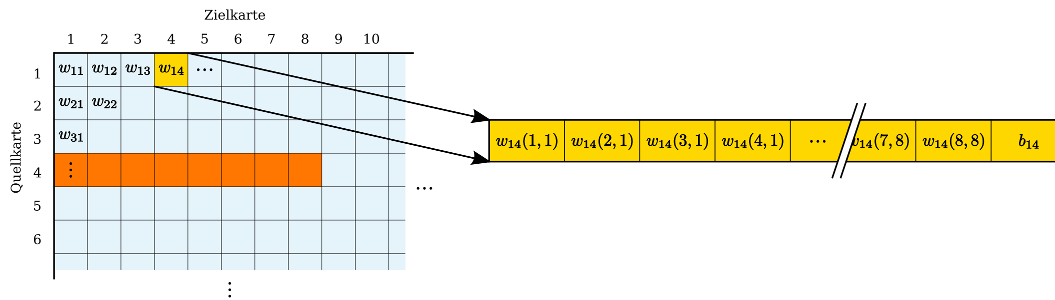


Abbildung 5.8: Speicherung der Konvolutionsfilter. Primär sind die Filter danach sortiert, welche Quellkarte sie verwenden, sekundär danach, welche Zielkarte sie erzeugen. Die Elemente eines Filters sind sequentiell mit abschließendem Biasgewicht abgelegt. Rot markiert ist ein Block von acht Filtern, der für einen Kernelaufruf benötigt wird.

$l+1$ (siehe Gleichung 4.8 auf Seite 51).

Bei sukzessiver Verarbeitung der Schichten werden die Aktivitäten der Schicht l daher nicht mehr benötigt, sobald die Fehlersignale dieser Schicht vollständig berechnet sind. Da die Fehlersignale jedoch wie in Abschnitt 5.5.1 erläutert nicht direkt berechnet werden, sondern zunächst akkumuliert werden, kann der Speicher der Aktivitäten nicht einfach durch die Fehlersignale ersetzt werden. Bei der Rückwärtspropagierung kommt daher ein temporärer Zwischenspeicher zum Einsatz, der als Akkumulator für die Fehlersignale dient.

Dieser Speicher hat dieselbe Struktur und Größe wie die restlichen Speicherbereiche, er umfasst aber nur die Daten einer einzigen Schicht. Nachdem das Fehlersignal im Akkumulator vollständig berechnet ist, werden die Aktivitäten dieser Schicht nicht mehr benötigt und können freigegeben werden. Der temporäre Zwischenspeicher kann anschließend als neuer Pixeldatenspeicher für diese Schicht weiterverwendet werden.

Der Zwischenspeicher wird immer nur für die aktuell verarbeitete Schicht benötigt und verbraucht somit maximal soviel Speicher, wie die größte Schicht im Konvolutionsnetz benötigt. Da die Fehlersignale auf der untersten Schicht nicht berechnet werden müssen, werden bei einer 512×512 Pixel großen Eingabe und acht Merkmalskarten auf Schicht L1 nur $256 \times 256 \cdot 8 \cdot 4 = 2\,\text{MB}$ pro Muster gebraucht. Gegenüber einer Speicherung der Fehlersignale aller Schichten (8 MB) ist dies also eine Ersparnis von 75%.

5.4.3 Konvolutionsfilter

Der Speicherbedarf der 8×8 Filter ist im Vergleich zu dem der Merkmalskarten relativ gering. Da die Filter aber an jeder Bildposition angewandt werden, muss häufig auf diese Werte zugegriffen werden. Es sind nur Lesezugriffe nötig, da während einer Epoche die Gewichte nicht verändert werden. Aus diesen beiden Gründen ist der 64 KB große Konstantenspeicher (siehe Abschnitt 2.4.3) prädestiniert zur Speicherung der Gewichte.

Die Konvolutionsfilter sind genau wie die Pixeldaten in eigenen, für jede Schicht getrennten Speicherbereichen abgelegt. Die Anordnung der Gewichte einer Schicht ist in Abbildung 5.8 illustriert. Bei einer Quellschicht mit I Merkmalskarten und einer darauffolgenden Zielschicht mit J Merkmalskarten, werden $I \times J$ Filter benötigt. Jeder Filter besteht aus 8×8 Gewichten und einem Biasgewicht, verbraucht also $(8 \times 8 + 1) \times 4 = 260\,\text{Byte}$.

Sie sind so im Speicher angeordnet, dass alle Filter, die als Eingabe dieselbe Merkmalskarte verwenden, hintereinander im Speicher liegen. Da ein Kernelaufruf eine Quellkarte auf

acht Zielkarten abbildet, liegen die acht dafür benötigten Filter also zusammenhängend im Speicher und können mit nur einer Speichertransaktion kopiert werden. Da der Konstantenspeicher für keine anderen Zwecke verwendet wird, ist es sogar möglich, im voraus für mehrere Kernelaufrufe bis zu 250 Filter zu kopieren.

Der Konstantenspeicher ist in der CUDA-Hardwarearchitektur als Cache realisiert. Jeder Multiprozessor verfügt über einen 8 KB großen Cache, aus dem die Daten – sofern sie einmal gecacht sind – genauso schnell wie aus Registern geladen werden können. Bereits vor dem Kernelaufruf müssen die Daten daher in den Konstantenspeicher kopiert werden und können dort nicht mehr verändert werden. Die acht Konvolutionsfilter benötigen nur 2080 Byte des Speichers, somit können sämtliche Filterelemente im Cache zur Verfügung gestellt werden, ohne dass ein Austauschen erforderlich wäre.

5.5 Rückwärtspropagierung

Der Kernel zur Rückwärtspropagierung (*Backprop-Kernel*) soll zwei Aufgaben in einem Schritt erledigen. Gegeben seien die Fehlersignale ${\delta_j}^{(l+1)}(x,y)$ der Merkmalskarten einer Schicht $l+1$ (Zielschicht). Daraus sollen die Fehlersignale ${\delta_i}^{(l)}(x,y)$ der darunterliegenden Schicht (Quellschicht) berechnet werden.

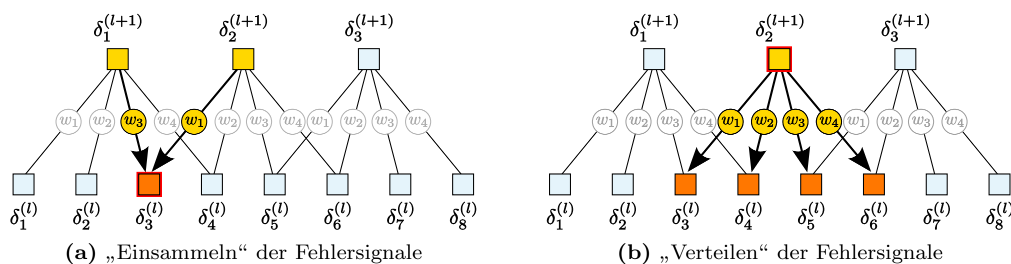
Außerdem sollen zu jedem Gewicht $w_{ij}(u,v)$ die Gradienten $\partial E / \partial w_{ij}(u,v)$ ermittelt werden, die zur Anpassung der Gewichte benötigt werden.

5.5.1 Akkumulation der Fehlersignale

Die Rückpropagierung des Fehlers ist nicht nur berechnungsintensiver, sondern erfordert bei der Implementierung eines Konvolutionsnetzes außerdem komplizierte und fehleranfällige Indexberechnungen.

In gewöhnlichen Multilayer-Perzeptrons wird der Fehler einer tieferen Schicht l meistens berechnet, indem alle Neuronen der höheren Schicht $l+1$ durchlaufen werden und die gewichteten Fehler aufsummiert werden:

$$\delta_i^{(l)} = \sum_{j=1}^{J} w_{ij}^{(l+1)} \cdot \delta_j^{(l+1)} \cdot f'_{act}(net_i^{(l)}) \tag{5.2}$$



(a) „Einsammeln“ der Fehlersignale

(b) „Verteilen“ der Fehlersignale

Abbildung 5.9: Einfaches Beispiel der Rückwärtspropagierung in einem eindimensionalen Konvolutionsnetz. Für die Berechnung der Fehlersignale auf der unteren Schicht sind zwei verschiedene Methoden möglich: (a) Beim „Einsammeln“ ist für ein Neuron der unteren Schicht nicht direkt ersichtlich, welche Gewichte und Fehlersignale zur Berechnung benötigt werden. Beim „Verteilen“ (b) wird jedes Gewicht einmal verwendet und es ist einfach zu ermitteln, welche Neuronen der unteren Schicht betroffen sind.

Man könnte sagen, dass ein Neuron der unteren Schicht l die Fehlersignale der oberen Schicht $l+1$ „einsammelt". Abbildung 5.9a zeigt diese Vorgehensweise bei einem Konvolutionsnetz anhand eines eindimensionalen Beispiels mit einem 4-elementigen Filter. Aufgrund der gekoppelten Gewichte und des Subsamplings ist die Berechnung selbst in diesem vereinfachten Fall umständlich. Benachbarte Neuronen der unteren Schicht (z.B. Neron 3 und 4) verwenden unterschiedliche Gewichte, greifen aber teilweise auf dieselben Neuronen der oberen Schicht zu. Umgekehrt verwenden einige Neuronen (z.B. Neuron 3 und 5) dieselben Gewichte aber unterschiedliche Fehlersignale der oberen Schicht.

Die Berechnungen lassen sich vereinfachen, indem man ein Neuron der oberen Schicht betrachtet und sozusagen dessen Fehlersignal an die Neuronen der unteren Schicht „verteilt" (wie in Abbildung 5.9b gezeigt). Ein Neuron der oberen Schicht durchläuft dann alle Gewichte und akkumuliert das gewichtete Fehlersignal in der unteren Schicht.

Für den zweidimensionalen Fall mit 8×8 Filtern ermöglicht diese Methode eine erhebliche Vereinfachung der Implementierung. Laut Simard *et al.* [SSP03], die diese beiden Methoden als *pulling* bzw. *pushing* bezeichnen, ist ein Verteilen der Fehlersignale generell langsamer, da die Akkumulation im Hauptspeicher und nicht wie beim Einsammeln in Registern erfolgt. Sie gehen aber davon aus, dass bei großen Konvolutionsfiltern und durch das Ausnutzen von parallelen Instruktionen der umgekehrte Fall eintritt.

In dieser Arbeit wurde daher die Variante des Verteilens der Fehlersignale gewählt. Da der Backprop-Kernel somit sozusagen die Operationen des Forward-Kernels umkehrt, können die im Forward-Kernel eingesetzten Konzepte und Speicherstrukturen in ähnlicher Form erneut verwendet werden.

5.5.2 Konfiguration des Backprop-Kernels

Auch bei der Beschreibung des Backprop-Kernels wird zunächst der Spezialfall angenommen, dass die Karten der Schicht l, für die das Fehlersignal berechnet werden soll, eine Größe von 64×64 Pixeln haben. Eine Erweiterung auf größere Karten sowie die Behandlung von Randfällen und Biasgewichten werden in Abschnitt 5.9 diskutiert.

Gegeben seien die Fehlersignale ${\delta_j}^{(l+1)}(x,y)$ der J Merkmalskarten der Zielschicht $l+1$. Der Backprop-Kernel soll diese Fehlersignale auf die I Merkmalskarten der Quellschicht l zurückpropagieren. Wie auch der Forward-Kernel bearbeitet jeder Kernelaufruf genau eine Quellkarte und acht Zielkarten. Um Gleichung 5.2 zu berechnen, wird zunächst die Summe der gewichteten Fehlersignale $\sum_{j=0}^{J} \delta_j \cdot w_{ij}$ über alle Kernelaufrufe akkumuliert, und diese wird erst nach vollständiger Berechnung mit der Ableitung der Netzeingabe $f'_{act}(net_i)$ multipliziert.

Ein Kernelaufruf berechnet also den von acht 32×32 Pixel großen Zielkarten j bis $j+7$ der Schicht $l+1$ erzeugten Anteil des Fehlersignals für eine 64×64 Pixel große Quellkarte i der Schicht l. Für die vollständige Akkumulation sind also für jede der I Quellkarten $\lceil J/8 \rceil$ Aufrufe des Kernels notwendig.

Der Backprop-Kernel erhält als Eingabeparameter einen Pointer auf die Fehlersignale der Merkmalskarten j bis $j+7$ der Schicht $l+1$, die wie in Abschnitt 5.4 beschrieben als zusammenhängender Speicher vorliegen. Außerdem wird ihm die Aktivität der Merkmalskarte i aus Schicht l übergeben sowie ein Pointer auf den Akkumulator für das Fehlersignal derselben Karte. Die acht benötigten Konvolutionsfilter werden wiederum im Konstantenspeicher zur Verfügung gestellt.

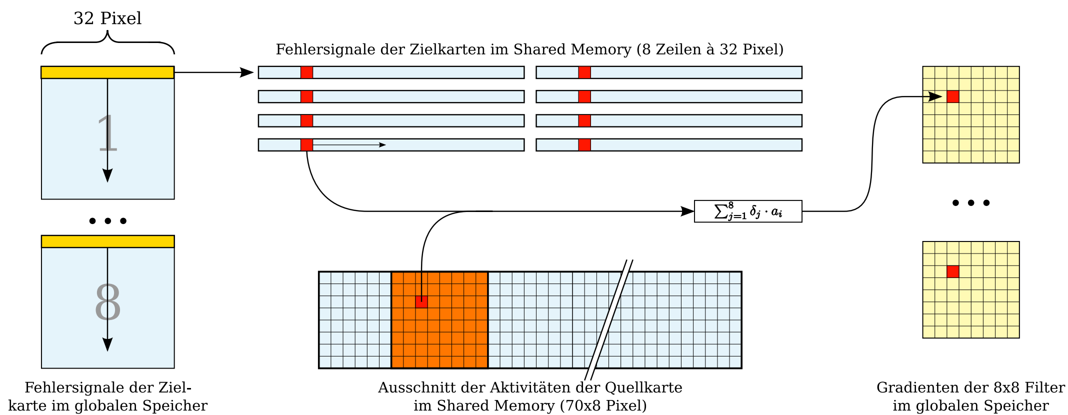


Abbildung 5.10: Berechnung der Gradienten im Backprop-Kernel. Links: Aus jeder der acht Zielkarten wird eine 32 Pixel breite Zeile der Fehlersignale in den Shared Memory geladen. Mitte: Die Teilsumme $\delta_j \cdot w_{ij}(u,v)$ des Gradienten $\partial E/\partial w_{ij}(u,v)$ wird aus den Fehlersignalen und den Aktivitäten für 8×64 Elemente berechnet. Rechts: Das Ergebnis wird im Register akkumuliert und schließlich in den globalen Speicher kopiert.

5.5.3 Backpropagation-Algorithmus

Wie im Forward-Kernel ist das Ziel der Backpropagation-Implementierung möglichst alle zur Berechnung benötigten Daten im Shared Memory zu halten und diese möglichst häufig zu verwenden. Ein einziger Ringpuffer reicht dazu nicht aus, da außer den Fehlersignalen der höheren Schicht $l + 1$ auch die Aktivitäten der tieferen Schicht l benötigt werden und die Fehlersignale der tieferen Schicht im Shared Memory aggregiert werden sollen.

Berechnung der Gewichtsgradienten

Die partiellen Ableitungen der Gewichte berechnen sich nach Gleichung 4.9 aus den Aktivitäten ${a_i}^{(l)}(x,y)$ der Quellschicht und den Fehlersignalen ${\delta_j}^{(l+1)}(x,y)$ der Zielschicht. Der Backprop-Kernel soll die Gradienten für acht Konvolutionsfilter, die jeweils aus $8 \times 8 = 64$ Gewichten bestehen, berechnen. Dies ist einer der Gründe, warum der Kernel $8 \times 64 = 512$ Threads verwendet, da somit der Gradient jedes Gewichts von einem eigenen Thread berechnet werden kann.

Abbildung 5.10 skizziert die Vorgehensweise der Gradientenberechnung. Wie beim Forward-Kernel wird ein Teil der Aktivitäten der 64×64 Pixel großen Quellkarte in einem 70×8 Pixel großen Ringpuffer im Shared Memory zwischengespeichert. Außerdem wird jeweils eine 32 Pixel breite Zeile der Fehlersignale von jeder der acht Zielkarten in den Shared Memory geladen. Anhand dieser Daten kann nun die Teilsumme der Gradienten für 32 Positionen in der Quellkarte berechnet werden.

Ein Thread behandelt ein festes Gewicht $w_{ij}(u,v)$ aus einem der acht Konvolutionsfilter. Die Teilsumme des Gradienten $\partial E/\partial w_{ij}(u,v)$ akkumuliert der Thread, indem er über alle 32 Posititonen iteriert und somit die gesamte Breite der geladenen Daten abdeckt. Die aggregierte Teilsumme wird in einem Register gespeichert und kann zum Ende des Kerneldurchlaufs im globalen Speicher akkumuliert werden.

Sowohl Lese- als auch Schreibzugriffe sind bei diesem Schema optimal. Für die Fehlersignale der Zielkarten werden $32 \times 8 \times 4 = 1024$ Byte des Shared Memory benötigt, die

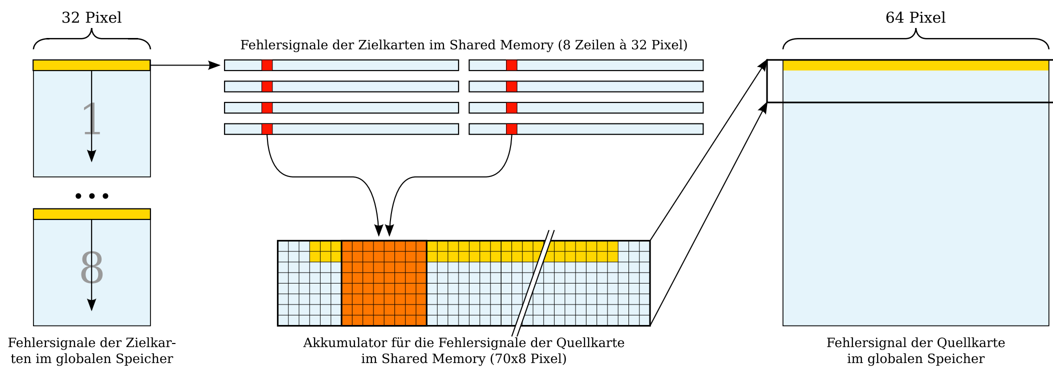


Abbildung 5.11: Aggregierung der Fehlersignale. Links: Je eine Zeile der Fehlersignale wird für acht Zielkarten in den Shared Memory geladen. Mitte: Das Fehlersignal der Zielkarten wird in den Akkumulator für die Fehlersignale der Quellkarten geschrieben. Rechts: Am Ende einer Iteration werden zwei Zeilen des Fehlersignals (gelb) aus dem Ringpuffer in den globalen Speicher geschrieben.

von jeweils einem Warp (32 Threads) zusammenhängend aus dem globalen Speicher geladen werden können. Je 64 zusammenhängende Threads (alle sind demselben Filter aber unterschiedlichen Filterelementen zugeordnet) lesen dasselbe Fehlersignal aus dem Shared Memory, was somit per schnellem Broadcast geladen werden kann (siehe Seite 35). Auch die Schreibzugriffe in den globalen Speicher sind optimal, da die 64 Gewichte (= zwei Warps) eines Filters im globalen Speicher hintereinander liegen.

Aggregierung der Fehlersignale im Ringpuffer

Die Aggregierung der Fehlersignale für ein Fenster der Eingabekarte ist in Abbildung 5.11 dargestellt. Der Kernel verwendet dazu die 32 Pixel breiten Zeilen der Fehlersignale der acht Zielkarten, die wegen der Gradientenberechnung bereits im Shared Memory liegen.

Die Fehlersignale der Eingabeschicht l sollen, wie in Abschnitt 5.5.1 beschrieben, akkumuliert werden, indem die schon geladenen Pixeldaten der Ausgabeschicht auf die Quellschicht „verteilt“ werden. Eine Aggregierung im globalen Speicher wäre nicht effizient, daher dient ein Zwischenpuffer im Shared Memory als Akkumulator. Die 32 Pixel einer Zeile der Zielkarte müssen auf ein 70×8 Pixel großes Fenster der Quellkarte verteilt werden. Ein 70×8 Pixel großer Bildausschnitt reicht also zur Akkumulation aus, wenn dieser wie bei der Vorwärtspropagierung als Ringpuffer implementiert wird. Dazu wird dieser Ringpuffer mit Null initialisiert, sodass die Fehlersignale darin sukzessive aggregiert werden können.

Die Aufteilung der 512 Threads zur Berechnung ist nicht intuitiv, sondern für günstige Speicherzugriffe optimiert und daher relativ kompliziert. Anhand von Abbildung 5.12 soll diese Aufteilung erläutert werden.

Generell soll ein Fehlersignal der Zielkarte auf die korrespondierenden 8×8 Pixel der Quellkarte verteilt werden. Die Schreibzugriffe in den 70×8 großen Ringpuffer sollen jedoch so erfolgen, dass jeweils 32 aufeinanderfolgende Threads einen zusammenhängenden Teil schreiben. Jeder der 512 Threads führt insgesamt vier Schreibzugriffe durch, und akkumuliert jedes Mal die Teilsumme $\sum_{j=1}^{8} \delta_j \cdot w_{ij}$ für je ein Pixel der acht Ausgabefehlersignale.

Als Beispiel diene Thread 1: Dieser berechnet zunächst die Summe $\sum_{j=1}^{8} \delta_j(1{,}1) \cdot w_{ij}(1{,}1)$ über dem gleichen Pixel aller acht Zielkarten j. Diese Summe akkumuliert er in der ersten Zelle des Ringpuffers der Zielkarte. Im zweiten Durchlauf führt der Thread die gleiche Be-

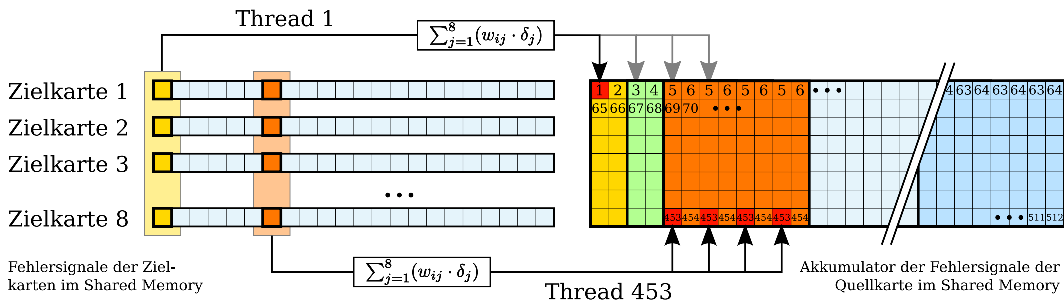


Abbildung 5.12: Im Vorwärtsschritt wurde der rechts rot markierte 8×8 Pixel große Bereich auf jedes der acht Pixel in den Zielkarten (links) abgebildet. Im Backprop-Kernel wird dieser Schritt umgekehrt. Ein Thread berechnet zunächst die gewichtete Summe aus acht Zielkarten und schreibt diese anschließend in den Akkumulator der Fehlersignale. Dieser Schritt wiederholt sich viermal und jedes Mal rückt der Thread auf dem Akkumulator zwei Pixel vor. Die Threads sind – wie hier durch die Nummerierung angedeutet – so angeordnet, dass die Schreibvorgänge zusammenhängend erfolgen können.

rechnung mit Gewicht w_{ij} durch und schreibt das Ergebnis in die dritte Spalte der ersten Zeile des Ringpuffers. Durch das Subsampling überspringt er also eine Spalte.

Dieses etwas komplizierte Schema ermöglicht es, dass alle Schreibzugriffe auf den Akkumulator zusammenhängend durchgeführt werden können.

Schritte des Backpropagation-Algorithmus

Wie auch bei der Vorwärtspropagierung durchläuft der Backpropagation-Algorithmus iterativ alle 32 Zeilen der Zielkarten. Während jeder Iteration werden vier Schritte ausgeführt: Laden der Fehlersignale in den Shared Memory, Berechnung des zurückpropagierten Fehlersignals, Speichern der Fehlersignale im globalen Speicher und die Berechnung der Gewichtsgradienten.

Laden der Fehlersignale Die Fehlersignale der Zielschicht liegen vollständig berechnet im globalen Speicher vor. In einem Schritt wird jeweils eine 32 Pixel breite Zeile aller acht Karten aus dem Shared Memory geladen. Die 32 für eine Karte zuständigen Threads bilden dabei einen Warp und lesen einen zusammenhängenden Speicherbereich für ihre Quellkarte. Insgesamt führen 256 Threads diese Leseoperation aus, während die restlichen 256 Threads ruhen. Da ruhende und aktive Threads aber nicht demselben Warp angehören, treten keine divergenten Warps auf.

Außerdem müssen die nächsten zwei Zeilen der zuvor von anderen Kernelaufrufen partiell berechneten Fehlersignale der Quellkarten geladen werden. Das Laden dieser 70×2 Pixel erfolgt in 140 Threads. Aufgrund der Randpixel treten bei diesem Ladevorgang unvermeidbare Divergenzen innerhalb mindestens eines Warps auf.

Aggregieren der Fehlersignale Das laufzeitintensive Kernstück des Backprop-Kernels ist die bereits beschriebene akkumulative Berechnung des Fehlersignals der Quellkarte. Hierzu sind alle 512 Threads aktiv und so angeordnet, dass ausschließlich zusammenhängende Schreibzugriffe auf den Ringpuffer der Fehlersignale erforderlich sind.

Auf die häufig benötigten Gewichte aus dem Konstantenspeicher kann wie beim Forward-Kernel sehr schnell zugegriffen werden, da sie vollständig im Cache vorliegen.

Speichern der Fehlersignale Nach den Berechnungen der Fehlersignale in einer Iteration stehen im Ringpuffer größtenteils nur partielle Teilsummen. Lediglich die beiden obersten Zeilen enthalten finale Werte und würden in der nächsten Iteration aus dem Ringpuffer gelöscht. Daher werden sie vorher aus dem Ringpuffer in den globalen Speicher geschrieben. Bevor dieser Schreibvorgang ausgeführt wird, wird die Teilsumme jedoch noch mit der Ableitung der Quellaktivität multipliziert: $\delta_j = f'(net_j) \cdot \sum(\delta_k \cdot w_{ij}))$. Bei diesem Schreibvorgang sind wiederum einige Threads – auch innerhalb eines Warps – untätig, da nur $70 \times 2 = 140$ Pixel geschrieben werden müssen, aber 512 Threads vorhanden sind.

Gradientenberechnung Letzter Schritt einer Zeilen-Iteration im Backprop-Kernel ist die Berechnung des Gewichtsgradienten. Jeweils 64 zusammenhängende Threads berechnen den Gradienten für jedes Element des 8×8 Filters. Jeder Thread iteriert dazu über die 32 Pixel einer Zeile der Zielkarte und akkumuliert den Gradienten in einem Register. Da die 64 Threads somit in jeder Iteration dasselbe Datenelement der Zielkarte benötigen, kann dieser Ladevorgang als Broadcast erfolgen.

Auch die Schreibzugriffe in den globalen Speicher sind optimal, da die 64 Gewichte (= zwei Warps) eines Filters im globalen Speicher hintereinander liegen.

Nach dem Durchlauf des Kernels steht in der Quellkarte das partielle akkumulierte Fehlersignal aller bisher bearbeiteten Zielkarten. Auch die Gewichtsgradienten enthalten dann den Anteil der bisher bearbeiteten Zielkarten.

5.5.4 Anmerkungen zum Backprop-Kernel

Die in Abschnitt 5.5.1 genannten Erwägungen haben dazu geführt, dass der Backprop-Kernel bei der Aggregierung der Fehlersignale dem Forward-Kernel sehr ähnlich ist. Viele der in Abschnitt 5.3.4 dargelegten Abwägungen treffen daher auch auf den Backprop-Kernel zu. Gravierendster Unterschied der beiden Kernels ist die Anzahl der Threads, die hauptsächlich durch den höheren Speicherbedarf bedingt ist.

Speicherbedarf und Blockgröße

Wie im Forward-Kernel wird ein 70×8 Pixel großes Fenster (2240 Byte) der Aktivitäten der Quellkarte für die Berechnungen benötigt. Zusätzlich müssen jedoch die Fehlersignale der Quellkarte für einen ebenso großen Bildausschnitt akkumuliert werden, da eine Akkumulation im globalen Speicher nicht effizient wäre. Daher wird ein weiterer Speicherbereich von 70×8 Pixeln Größe (2240 Byte) benötigt.

Außerdem müssen die Fehlersignale der acht Zielkarten aus dem globalen Speicher geladen werden. Da hierzu jeweils ein 32 Pixel breiter Streifen ausreicht, sind nur weitere 1024 Byte erforderlich. Insgesamt benötigt der Kernel also 5504 Byte des Shared Memory.

Ein Multiprozessor verfügt über 16 KB Shared Memory, der unter allen aktiven Threadblöcken aufgeteilt wird. Wenn jeder Threadblock 5,5 KB davon benötigt, können somit höchstens zwei Blöcke gleichzeitig auf einem Multiprozessor aktiv sein. Mit je 256 Threads wären demnach nur 50% der maximal möglichen 1024 Threads aktiv.

Um trotz des hohen Speicherbedarfs eine gute Auslastung der Prozessoren zu erreichen, wurde daher die Anzahl der Threads pro Block auf 512 erhöht.

Auslastung der Threads

Der Kernel nutzt nicht immer sämtliche 512 Threads aus, insbesondere bei einigen Lade- und Schreibvorgängen. An den Stellen mit hoher arithmetischer Dichte, nämlich bei der Rückpropagierung des Fehlersignals und bei der Berechnung der Gradienten ist es – wie in Abschnitt 5.5.3 gezeigt – trotzdem möglich, alle 512 Threads zu beschäftigen. Insbesondere die Berechnung der Gradienten der acht 8×8 Filterelemente lässt sich intuitiv auf 512 Threads verteilen.

Mit einem Aufruf des Backprop-Kernels werden sehr viele Berechnungen ausgeführt, somit hat er eine relativ lange Laufzeit (siehe Abschnitt 6.3.1), wodurch der Einfluss des Overheads beim Kernelaufruf reduziert wird. Insbesondere die langsamen Zugriffe auf den globalen Speicher wurden minimiert, da auf jedes Datenelement im globalen Speicher nur selten zugegriffen wird.

5.6 Gewichtsanpassungen

Der Backprop-Kernel hat die Gradienten von acht Konvolutionsfiltern zwar vollständig berechnet, aber für jedes der P Muster unabhängig voneinander gespeichert. Eine Aufsummierung ist parallel nicht möglich, da es keine atomare Addieroperation für Variablen vom Typen `float` gibt und gleichzeitige Schreibzugriffe auf dasselbe Datenelement inkonsistente Resultate liefern. Bevor eine Anpassung der Gewichte vorgenommen werde kann, müssen also diese P Teilergebnisse für jedes Gewicht aufsummiert werden.

Dieses Aufsummieren der Gradienten und die Gewichtsanpassungen werden von einem eigenen Kernel durchgeführt, der nicht über die Anzahl der Muster parallelisiert. Da jedes der 8×64 Filterelemente aktualisiert werden muss, wird stattdessen eine Parallelisierung mit 512 Threads durchgeführt. Mit dieser geringen Anzahl von Threads ist keine optimale Auslastung möglich, damit aber kein Multiprozessor inaktiv ist, wird jedes der 64 Elemente eines Filters von einem Block bearbeitet. Innerhalb eines Blocks wird jeweils ein anderer der acht Filter von einem Thread verarbeitet.

Für dieses Filterelement summiert der Thread zunächst alle der P Gradienten. Anschließend wird anhand dieses Gradienten abhängig von der Lernregel (Backpropagation oder Rprop) ein Update des Gewichts durchgeführt.

Im mehrerlei Hinsicht ist dieser Kernel nicht optimal. Bei Rprop führt jeder Thred unter Umständen einen anderen Anweisungspfad aus, was dazu führt, dass jeder Thread eines Warps alle Pfade abarbeiten muss. Da auf einem Multiprozessor nur acht Threads parallel laufen, wird dieser nicht hinreichend ausgenutzt. Außerdem greifen die Threads eines Blocks auf unterschiedliche Bereiche des globalen Speichers zu. Da die Gewichtsaktualisierungen aber nur einen Bruchteil der Berechnungen des Lernverfahrens ausmachen, sind diese Unzulänglichkeiten hinnehmbar.

5.7 Vollverknüpfte Schichten

Die Berechnungen der vollverknüpften Schichten können beim Offline-Training als Matrizenoperationen formuliert werden. Als Beispiel seien hier drei vollverknüpfte Schichten gegeben: eine Eingabeschicht I, eine verdeckte Schicht J und eine Ausgabeschicht K. Die Eingabeschicht soll die Werte der letzten Konvolutionsschicht enthalten und als Lernverfahren wird Backpropagation of Error mit einer Lernrate η verwendet.

Sei P die Anzahl der n_I-dimensionalen Eingabemuster, dann wird eine Eingabematrix X der Größe $P \times n_I$ gebildet, indem die Werte des Eingabemusters p in die p-te Zeile geschrieben werden. Nach demselben Schema wird für die n_K-dimensionale Ausgabeschicht die $P \times n_K$-dimensionale Teacher-Matrix T für die gewünschte Ausgabe gebildet. Die Matrizen der Aktivitäten der verdeckten Schicht (Y) und der Ausgabeschicht (Z) haben die Dimension $P \times n_J$ bzw. $P \times n_K$. Die Gewichte w_{ij} zwischen zwei Schichten I und J werden in einer $n_I \times n_J$-dimensionalen Matrix W_{IJ} abgelegt. Somit lässt sich die Propagierung der Muster von Schicht I nach J schreiben als

$$Y = f_{act}(net_J) = f_{act}(X * W_{IJ}). \tag{5.3}$$

Für die Gewichtsanpassungen der verdeckten Gewichte W_{IJ} und der Ausgabegewichte W_{JK} ergeben sich folgende Gleichungen:

$$\Delta W_{JK} = \eta \cdot Y^T * ((T - Z).f'_{act}(net_K)) \tag{5.4}$$

$$\Delta W_{IJ} = \eta \cdot X^T * \big(((T - Z).f'_{act}(net_K)) * W_{JK}^T.f'_{act}(net_J)\big) \tag{5.5}$$

Darin bezeichnet (.) das elementweise Hadamard-Produkt zweier Matrizen und $f_{act}(M)$ die Anwendung der sigmoiden Funktion auf jedes Element der Matrix M.

Mit der CUBLAS-Bibliothek [Nvi08a] können Multiplikationen und Additionen von Matrizen und Vektoren wie von Lahabar *et al.* [LAN08] beschrieben auf CUDA-Hardware beschleunigt werden. Da die Matrizen für die CUBLAS-Schnittstelle spaltenweise vorliegen müssen, ist es notwendig, die Aktivitäten aus der Konvolutionsschicht zunächst in entsprechender Reihenfolge zu speichern. Ein eigener Kernel erledigt diese Aufgabe, indem er threadweise je ein Datenelement von der Konvolutionsschicht in die Matrix kopiert. Für die Rückwärtspropagierung werden umgekehrt die Fehlersignale aus der ersten vollverknüpften Schicht in die letzte Konvolutionsschicht kopiert.

Für die elementweise Multiplikation zweier Matrizen und die elementweise Anwendung der sigmoiden Funktion stellt die CUBLAS-Bibliothek keine Routinen bereit. Daher wird für diese Berechnungen jeweils eine einfache Kernelfunktion implementiert, die in jedem Thread ein Matrixelement verarbeitet.

Da nicht nur in den Konvolutionsschichten, sondern auch in den vollverknüpften Schichten als Lernverfahren Rprop mit Weight Decay eingesetzt werden soll, wird ein weiterer Kernel benötigt. Dieser berechnet für jedes Gewicht aus den Gradienten die in Abschnitt 2.1.4 beschriebene Gewichtsanpassung. Da die Instruktionsfolge im Rprop-Algorithmus von den Gradienten abhängt, lässt es sich nicht vermeiden, dass Threads innerhalb eines Warps divergieren. Zumindest bei den Speicher- und Ladevorgängen sind aber keine Effizienzeinbußen zu verzeichnen, da diese Zugriffe auf zusammenhängende Speicherbereiche erfolgen.

Die Funktionen der CUBLAS-Bibliothek können mit Matrizen beliebiger Größe rechnen und sind zudem auch für Spezialfälle – wie z.B. das gleichzeitige Transponieren von Matrizen – ausgelegt. Es kann daher davon ausgegangen werden, dass eine auf den hier vorliegenden Spezialfall zugeschnittene Implementierung zwar aufwändiger wäre, aber die Laufzeit noch weiter verringern könnte.

5.8 Sonstige Kernel

An mehreren Stellen werden in der Implementierung kleine Kernelfunktionen eingesetzt, um die parallele Berechnung auszunutzen und um zu verhindern, dass Daten zur Weiterverarbeitung aus dem GPU-Speicher in den CPU-Speicher kopiert werden müssen. Die meisten

Kernel	Anzahl Blöcke	Anzahl Threads	Register	S.Mem (Byte)	C.Mem (Byte)	Auslas-tung
Forward-Kernel	P	256	16	2.276	33.324	100%
Backprop-Kernel *	P	512	16	6.680	33.348	100%
Backprop-Update	8	65	7	32	33.284	75%
Rprop-Update	8	65	8	72	33.296	75%
Vorverarbeitung	$n \times P$	n	11	45	24	100%
Teacher	100	256	10	36	4	100%
Aktivierungsfunktion	100	256	10	28	44	100%
Kopieren nach MLP	8192	P	7	36	8	50%
Kopieren vom MLP	8192	P	6	36	8	50%
Vergrößerung	$256 \times P$	256	9	24	12	100%

Tabelle 5.1: Daten der unterschiedlichen Kernelfunktionen der Implementierung. Die theoretische Auslastung eines Multiprozessors ergibt sich aus der Anzahl der Threads, der Anzahl benötigter Register und dem benötigten Shared Memory und kann mit dem *CUDA Occupancy Calculator* [Nvi08b] berechnet werden. P bezeichnet die Anzahl der Muster der Arbeitsmenge, n die Breite der Eingabekarten. (*) Der Backprop-Kernel verwendet zusätzlich 32 KB lokalen Speicher um einen höheren Verbrauch an Registern zu vermeiden.

dieser Kernel führen einfache Berechnungen durch und verarbeiten pro Thread ein Element einer Merkmalskarte. Ein Beispiel ist der sigmoide Kernel, der an jeder Positon (x,y) auf die Netzeingabe die Aktivierungsfunktion anwendet:

$$a_i(x,y) = f_{act}\left(net_i(x,y)\right) \tag{5.6}$$

Bei einem Netz ohne vollverknüpfte Schichten (siehe Abschnitt 6.2.1) berechnet ein Kernel das Fehlersignal $\delta_i(x,y)$ auf der Ausgabekarte j elementweise aus der Differenz zwischen Teacher $t_i(x,y)$ und Ausgabe $a_i(x,y)$:

$$\delta_i(x,y) = (t_i(x,y) - a_i(x,y)) \cdot f'_{act}(net_i(x,y)) \tag{5.7}$$

Eine weitere Kernelfunktion erzeugt zusätzliche Eingaben auf höheren Schichten, indem die Auflösung des Eingabebildes verringert wird und Kanten extrahiert werden (siehe Abschnitt 4.4.1). Tabelle 5.1 vergleicht die Daten einiger dieser Kernel, auf eine detaillierte Beschreibung wird jedoch verzichtet.

5.9 Anmerkungen zur parallelen Implementierung

5.9.1 Behandlung von Einschränkungen und Spezialfällen

Zur besseren Verständlichkeit wurde die Beschreibung der Implementierung in diesem Kapitel auf vereinfachte Spezialfälle beschränkt. Was beim Aufheben dieser Einschränkungen der Vorwärtspropagierung und der Rückwärtspropagierung beachtet werden muss, wird im Folgenden erläutert.

Größe der Quellkarten Bisher wurde von 64×64 Pixel großen Quellkarten und 32×32 Pixel großen Zielkarten ausgegangen. Da sowohl der Forward- als auch der Backprop-Kernel zeilenweise iterieren, lassen sich die Algorithmen leicht auf Karten mit mehr Zeilen erweitern. Eine Erweiterung ist auch auf breitere Merkmalskarten einfach durchführbar,

wenn deren Breite ein Vielfaches von 64 bzw. 32 beträgt. Dazu wird die Eingabekarte in jeweils 64 Pixel breite Streifen unterteilt, die der Kernel nacheinander abarbeitet (siehe Abbildung 5.13).

Im Vorwärtskernel hat dies zur Folge, dass sich die Randpixel dieser Streifen überlappen und doppelt geladen werden müssen. Davon betroffen sind bei den verwendeten 8×8 Filtern jedoch nur 10% aller Pixel. Die korrespondierenden Streifen der Zielkarten überlappen sich hingegen nicht. Somit kommt es bei den Schreibvorgängen nicht zu Konflikten. Auch im Backprop-Kernel muss diesem Sonderfall Beachtung geschenkt werden. Da die Fehlersignale jedoch im globalen Speicher akkumuliert werden, kann eine Spalte nicht vorher berechnete Werte überschreiben.

Randbehandlungen An den Bildrändern muss der Konvolutionsfilter auf Positionen zugreifen, die außerhalb des Bildes liegen (siehe Abbildung 5.14). Um Konflikte bei den Speicherzugriffen zu verhindern, werden diese Fälle mit bedingten Anweisungen abgefangen und auf den Wert Null gesetzt. An den Rändern kommt es daher zu divergenten Anweisungspfaden innerhalb eines Warps, die sich insbesondere bei sehr kleinen Merkmalskarten in der Effizienz bemerkbar machen können.

Eine Alternative wäre es, die Bilder um entsprechend viele Randpixel zu vergrößern. Da zum einen eine Vorinitialisierung dieser Pixel notwendig wäre und zum anderen mehr Speicherplatz für diese vergrößerten Merkmalskarten benötigt würde, wurde diese Methode hier nicht gewählt.

Biasgewichte Die Gewichte der Biasneurone in den Konvolutionsschichten werden in sämtlichen Kernels separat behandelt, da sich die Elemente der 8×8-Filter häufig effizient auf zwei Threadwarps, also 64 Threads, verteilen lassen. Da ein Kernel in der Regel acht Konvolutionsfilter verwaltet, müssen diese acht Biase in einem nicht voll ausgenutzten Halfwarp behandelt werden.

Initialisierungen Auch die Initialisierung verschiedener Speicherbereiche bedarf spezieller Beachtung. Im Vorwärtskernel muss der Ringpuffer für jeden Streifen zunächst mit den ersten Zeilen der Merkmalskarte initialisiert werden. Der Akkumulator im Rückwärtskernel muss vor der Verarbeitung einer Schicht mit Null initialisiert werden. Bei einem Kernelaufruf müssen dann die ersten Zeilen dieses Akkumulators in den Ringpuffer geladen werden.

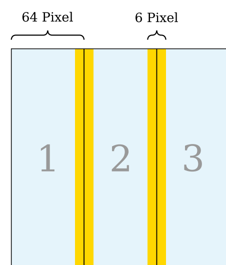


Abbildung 5.13: Aufteilung einer 192×192 Pixel großen Eingabekarte in 64 Pixel breite Streifen, die nacheinander verarbeitet werden. Gelb dargestellt sind Randpixel, die doppelt geladen werden müssen.

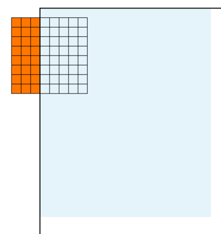


Abbildung 5.14: An den Rändern ragen die Konvolutionsfilter über den Bildrand hinaus. Diese Pixel werden auf Null gesetzt.

Eingabeschicht Auf der Eingabeschicht kann auf die Berechnung der Fehlersignale verzichtet werden. Für die Eingabeschicht ist es daher nicht notwendig, Speicher für eine temporäre Merkmalskarte (siehe Abschnitt 5.4.2) zu reservieren. Da die Eingabeschicht in der Regel den größten Speicherplatzbedarf hat, kann ein Teil des globalen Speichers eingespart werden, der somit zum Speichern der Muster zur Verfügung steht.

5.9.2 Schwierigkeiten bei der Implementierung

Die parallele Implementierung auf der CUDA-Architektur gestaltet sich deutlich komplizierter als eine serielle CPU-Implementierung.

Die Möglichkeiten zur Fehlersuche in Kernelfunktionen sind stark eingeschränkt, da ein hardwareseitiger Debugger von Nvidia erst mit der CUDA Version 2.2 nach Vollendung dieser Diplomarbeit zur Verfügung gestellt wurde. Bis dahin war es auf direktem Wege nur im sehr langsamen und unpräzisen Emulationsmodus möglich, Fehler in Kernelfunktionen nachzuvollziehen. Ein Vergleich der Ausgaben von GPU- und CPU-Implementierung ist aufgrund der nicht standardkonformen Berechnung von Gleitkommazahlen auf der GPU nur ansatzweise hilfreich.

Als große Fehlerquelle stellte sich auch die Initialisierung und Handhabung von Speicherbereichen heraus. Insbesondere bietet CUDA keine Möglichkeit, um Speicherüberläufe zu erkennen oder zu vermeiden. Bei mehreren aufeinanderfolgenden Programmaufrufen wird häufig derselbe Speicherbereich erneut reserviert, was bei nicht vorhandener oder fehlerhafter Initialisierung dazu führt, dass die sich darin noch befindlichen veralteten Daten fälschlicherweise weiterverwendet werden.

Insbesondere die Programmierung der Indexberechnungen haben sich als fehleranfällig herausgestellt. Zwar lassen sich beim Kernelaufruf die Threads eines Blocks semantisch auf ein-, zwei- oder dreidimensionale Arrays aufteilen, so dass eine einfache Indexierung in drei Dimensionen möglich ist. Dennoch ist es oftmals notwendig, manuell die Threadindizes zu berechnen, um Speicherzugriffe zu optimieren.

6 Experimente und Ergebnisse

Mit dem in dieser Diplomarbeit entworfenen Konvolutionsnetz, dessen parallele Implementierung im Kapitel 5 beschrieben wurde, sind unterschiedliche Untersuchungen durchgeführt worden. Die Ergebnisse dieser Messungen werden in diesem Kapitel vorgestellt.

In Abschnitt 6.1 wird zunächst detailliert analysiert, welchen Einfluss verschiedene Parameter und Varianten des Algorithmus auf die Konvergenz des Lernverfahrens haben. Die dabei gewonnenen Kenntnisse wurden auf unterschiedliche Datensätze angewandt, auf denen die Klassifizierungsleistung des Konvolutionsnetzes in Abschnitt 6.2 evaluiert wird. Zum Abschluss werden in Abschnitt 6.3 die Ergebnisse der Laufzeitmessungen vorgestellt, anhand derer ersichtlich wird, dass die parallele Implementierung des Verfahrens gegenüber der CPU-Version deutlich beschleunigt werden konnte.

6.1 Erweiterungen des Lernverfahrens

In Abschnitt 4.3 wurden mehrere Modifikationen des Lernverfahrens Backpropagation of Error beschrieben. Im Folgenden wird untersucht, wie sich diese Erweiterungen auf die Konvergenz des Lernalgorithmus auswirken und es wird versucht, geeignete Parametrisierungen der freien Variablen zu finden.

6.1.1 Arbeitsmenge

Aufgrund des begrenzten Speichers der Grafikkartenhardware (siehe Abschnitt 2.4.4) ist es nicht möglich, sämtliche Trainingsmuster im Offline-Training parallel zu verarbeiten. Stattdessen wird daher die in Abschnitt 4.3.4 erläuterte Arbeitsmenge eingesetzt, die nur aus einem Bruchteil der Trainingsmuster besteht und die sukzessive ausgetauscht wird. Dabei ist zum einen unklar, wieviele Muster der Arbeitsmenge man idealerweise in jeder Epoche austauscht und zum anderen mit welcher Strategie man Muster für die Arbeitsmenge auswählt. Beide Punkte werden experimentell geklärt.

Für diese Versuche wurde ein sehr kleines Netz eingesetzt, das aus vier Konvolutionsschichten besteht. Die Eingabeschicht L0 enthält nur eine Merkmalskarte der Größe 256×256 Pixel und die darauffolgenden Schichten enthalten jeweils acht Merkmalskarten, deren Größe sukzessive halbiert wird. Auf die 32×32 Pixel große letzte Konvolutionsschicht L3 folgt eine vollverknüpfte Schicht mit 100 Neuronen und eine Ausgabeschicht mit 10 Neuronen. Als Klassifikationsaufgabe dient der MNIST-Datensatz handschriftlicher Ziffern (siehe Abschnitt 2.3.1).

Die Arbeitsmenge umfasst in diesem Experiment 250 Muster der aus insgesamt 60.000 Mustern bestehenden Trainingsmenge und als Lernverfahren wurde Rprop mit dem in Abschnitt 6.1.2 ermittelten Weight-Decay-Faktor $\lambda = 0{,}0001$ eingesetzt.

Prozentsatz auszutauschender Muster

Das Lernverfahren Rprop (siehe Abschnitt 2.1.4) setzt zum Konvergieren einen stabilen Gradienten voraus. Wenn ein zu großer Anteil der Arbeitsmenge jede Epoche ausgetauscht

Austausch	1%	2%	5%	10%	20%	30%	50%	100%
Epochen	**Fehlerquote auf der Testmenge (in %)**							
1.000	15,8	14,5	11,0	10,9	9,1	14,5	17,9	21,5
5.000	14,4	14,2	9,0	8,2	4,4	10,0	15,2	20,3
10.000	12,8	13,4	8,1	7,6	3,7	9,3	14,9	20,1
20.000	9,1	14,0	7,1	6,4	3,2	8,8	13,5	20,1
50.000	8,1	13,3	6,2	5,6	3,1	7,7	12,5	19,9

Tabelle 6.1: Prozentsatz falsch klassifizierter Muster auf der Testmenge in Abhängigkeit davon, welcher Prozentsatz der Muster der Arbeitsmenge jede Epoche ausgetauscht wird. Die besten Ergebnisse werden erzielt, wenn 20% der Arbeitsmenge jede Epoche ausgetauscht werden.

wird, fluktuieren die Gradienten jedoch so stark, dass die Lernraten dauerhaft zu niedrig sind. Ein vollständiges Austauschen der Arbeitsmenge ist daher nicht sinnvoll. Umgekehrt kann es aber – wenn nur ein Bruchteil ausgetauscht wird – vorkommen, dass das Netz sich zu stark an die Muster der Arbeitsmenge anpasst und nicht über die restlichen Muster der Trainingsmenge generalisiert.

Auf dem kleinen Konvolutionsnetz wurde daher mit der MNIST-Datenbank untersucht, welcher Prozentsatz der Arbeitsmenge jede Epoche ausgetauscht werden muss, damit das Lernverfahren möglichst gut konvergiert. Für unterschiedliche Prozentsätze auszutauschender Muster wurde über viele Epochen hinweg beobachtet, wie sich der Fehler auf der Testmenge entwickelt. Tabelle 6.1 fasst diese Ergebnisse zusammen.

Erwartungsgemäß ist die Fehlerquote sowohl bei einem kleinen Prozentsatz als auch bei einem großen Prozentsatz sehr hoch. Als ideal hat sich herausgestellt, 20% der Muster jede Epoche auszutauschen. Da jede Epoche somit nur 50 neue Muster in die Arbeitsmenge aufgenommen werden, dauert es mindestens 1200 Epochen, bis das Konvolutionsnetz sämtliche Muster der Trainingsmenge einmal verarbeitet hat.

Gewichtetes Selektionsschema

Da das Netz immer nur auf den Mustern der aktuellen Arbeitsmenge lernt, ist es sinnvoll, diese so auszuwählen, dass sie den größtmöglichen Informationsgehalt haben. Daher wurden zwei verschiedene Strategien untersucht, die auswählen, welche Muster in die Arbeitsmenge aufgenommen werden.

Bei Online-Trainingsverfahren zeigt die Erfahrung, dass es sinnvoll ist, die Reihenfolge der Trainingsmuster jede Epoche zufällig zu permutieren. Auf Grundlage dieser Erkenntnisse besteht daher die erste Strategie darin, zufällig Muster der Trainingsmenge auszuwählen. Hierbei kann es im ungünstigen Fall jedoch vorkommen, dass einige wenige schwer zu lernende Muster nur sehr selten in die Arbeitsmenge aufgenommen werden und das Netz sich stattdessen auf die stark repräsentierten, einfachen Trainingsmuster spezialisiert.

Dieses Problem soll mit dem in Abschnitt 4.3.5 beschriebenen gewichteten Selektionsschema vermieden werden, indem schwer zu erlernende Trainingsmuster deutlich häufiger in die Arbeitsmenge aufgenommen werden als solche, deren quadratischer Fehler klein ist.

Beide Strategien sollen anhand des schon im letzten Abschnitt verwendeten kleinen Konvolutionsnetzes auf der MNIST-Datenbank verglichen werden. Die Arbeitsmenge besteht hier aus 250 Trainingsmustern, von denen jede Epoche 20% ausgetauscht werden.

Abbildung 6.1 vergleicht die Fehlerrate auf der Testmenge für beide Methoden. Nach 40.000 Epochen wurden bei randomisierter Auswahl 222 Muster der 10.000 Muster großen

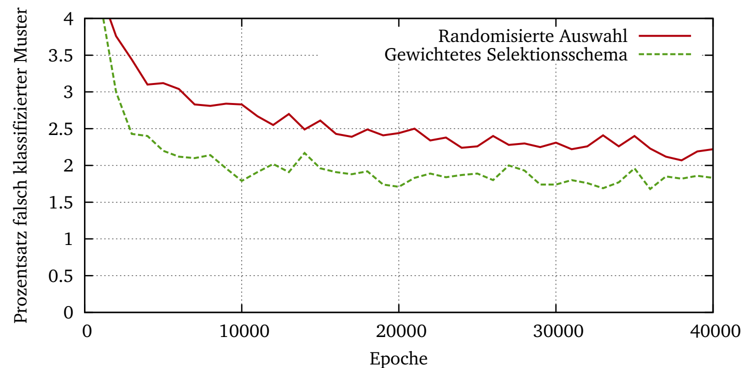


Abbildung 6.1: Beim gewichteten Selektionsschema werden Muster, die schlecht erkannt werden, mit höherer Wahrscheinlichkeit in die Arbeitsmenge aufgenommen. Diese Austauschstrategie verbessert die Fehlerquote gegenüber einem Verfahren, das zufällig Muster auswählt, deutlich.

Testmenge nicht erkannt. Mit dem gewichteten Selektionsschema wurden dagegen lediglich 183 Muster falsch klassifiziert. Gegenüber dem randomisierten Auswahlverfahren können somit ca. 20% der sonst nicht erkannten Muster korrekt klassifiziert werden.

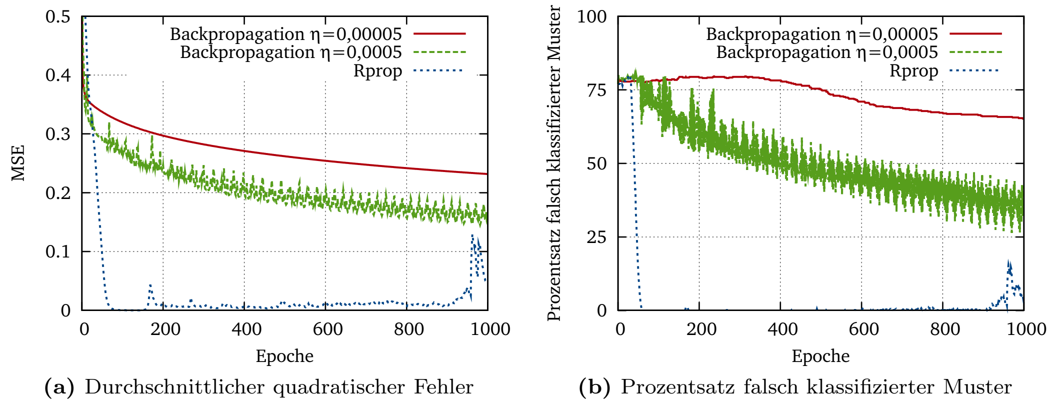
Es zeigt sich also, dass beim Training auf einer Arbeitsmenge mit dem gewichteten Selektionsschema eine erhebliche Verbesserung des Lernverfahrens möglich ist. Auch der mit Abstand niedrigste in dieser Diplomarbeit erzielte Fehler auf der Testmenge der MNIST-Datenbank wurde mit dieser Austauschstrategie erreicht (siehe Abschnitt 6.2.2). Das Konvolutionsnetz klassifizierte nach 100.000 Trainingsepochen nur noch 1,38% aller Testmuster falsch.

6.1.2 Gewichtsinitialisierung und -anpassung

In welche Richtung und wie stark die Gewichte jede Epoche angepasst werden, hängt von verschiedenen Parametern ab. Entscheided ist vor allem die Lernrate, aber auch der Weight-Decay-Faktor (siehe Abschnitt 2.1.4). Wie bereits in Abschnitt 4.3.3 erläutert, ist insbesondere die Wahl einer geeigneten Lernrate in Konvolutionsnetzen problematisch. Bei den im Folgenden beschriebenen Versuchen wird daher experimentell eine geeignete Parametrisierungen bestimmt. Auch der Einfluss der verschiedenen in Abschnitt 4.3.2 beschriebenen Initialisierungen der Gewichte auf die Erkennungsraten wird ergründet.

Für diese Experimente dient ein Konvolutionsnetz mit nur drei Konvolutionsschichten und zwei vollverknüpften Schichten, das auch auf den höheren Konvolutionsschichten Eingaben erhält. Die Eingabeschicht L0 besteht aus vier Merkmalskarten (je 128×128 Pixel), die Schicht L1 aus acht Merkmalskarten und drei Eingabekarten (je 64×64 Pixel) und die Schicht L2 aus 16 Merkmalskarten und drei Eingabekarten (je 32×32 Pixel). Darauf folgen zwei vollverknüpfte Schichten mit 20 und fünf Neuronen.

Mit diesem Netz soll eine Teilmenge des NORB-Datensatzes (siehe Abschnitt 2.3.2) gelernt werden, die aus 500 Mustern (je 100 pro Klasse) der Trainingsmenge besteht. Aufgrund der geringen Anzahl an Mustern und der kleinen Netzgröße kann auf den Einsatz einer Arbeitsmenge verzichtet werden und es wird echtes Offline-Training durchgeführt.



(a) Durchschnittlicher quadratischer Fehler **(b)** Prozentsatz falsch klassifizierter Muster

Abbildung 6.2: Beim Lernverfahren Backpropagation in einem Konvolutionsnetz ist die Wahl einer geeigneten Lernrate ein mehrdimensionales Optimierungsproblem. Erst die adaptive Anpassung gewichtsspezifischer Lernraten mit Rprop lässt das Netz gegen ein gutes Minimum konvergieren.

Lernrate

Das Ermitteln einer geeigneten Lernrate ist in Konvolutionsnetzen nicht trivial. Generell fällt in tiefen Netzen der Gradient auf den untersten Ebenen sehr klein aus, so dass hier meistens eine größere Lernrate verwendet werden muss. Auf den Konvolutionsschichten muss bei der Wahl der Lernrate zusätzlich berücksichtigt werden, dass sich mehrere Verbindungen ein Gewicht teilen.

Empirisch wurde versucht, eine Lernrate zu finden, mit der das Trainingsverfahren konvergiert. Mit festen Lernraten auf allen Schichten bzw. mit zwei getrennten Lernraten für Konvolutionsschichten und vollverknüpfte Schichten war dies nicht möglich, da das Netz schlecht konvergierte oder stark oszillierte.

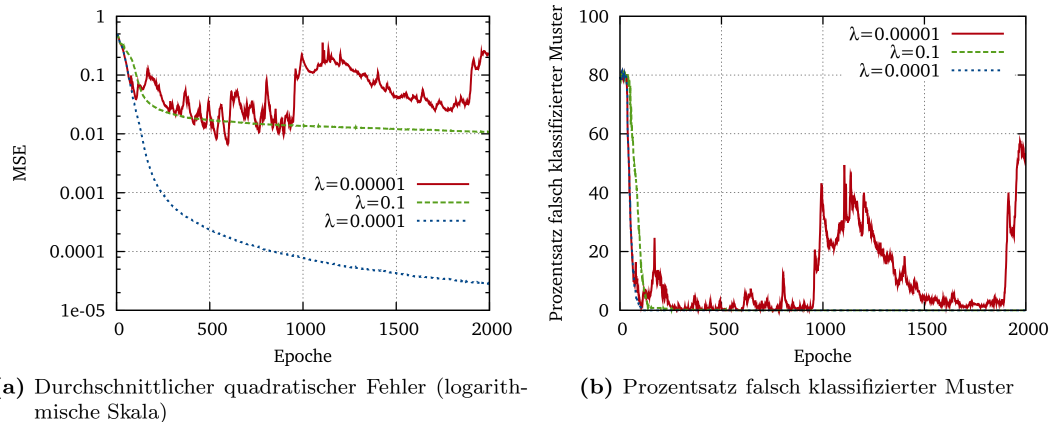
Daher wird die Lernrate für dieses Experiment sowohl in Abhängigkeit von der Anzahl c_s der Verbindungen, die dieses Gewicht teilen, als auch von der Anzahl der eingehenden Verbindungen c_{in} und der Anzahl der Muster im Offline-Training P gewählt. Für eine Konvolutionsschicht $l+1$ mit Merkmalskarten der Größe $n \times n$ gilt ${c_s}^{(l+1)} = (n/2)^2$. Sei J die Anzahl der Merkmalskarten auf Schicht l, dann gilt auf Schicht $l+1$ für die Anzahl der eingehenden Verbindungen eines Neurons: $c_{in} = J \times (8 \times 8 + 1)$.

Für jede Schicht l wurde abhängig von diesen Parametern eine eigene Lernrate $\eta^{(l)}$ auf die von LeCun *et al.* [LBOM98] vorgeschlagene Weise gewählt:

$$\eta^{(l)} = \frac{\eta}{\sqrt{P \cdot c_{in} \cdot c_s}} \tag{6.1}$$

Somit kann einzig mit dem Lernfaktor η für sämtliche Schichten die Lerngeschwindigkeit eingestellt werden.

Auch durch die Einschränkung auf diesen einen Parameter war es nicht möglich, experimentell Lernraten zu finden, die das Netz gegen ein gutes Minimum konvergieren lassen. Abbildung 6.2 zeigt die Lernkurven für die beiden repräsentativen Werte $\eta = 0{,}0001$ und $\eta = 0{,}0005$, die jedoch beide ungeeignet sind. Bei der größeren Lernrate oszilliert das Netz schon nach wenigen Epochen stark. Bei der kleineren Lernrate konvergiert es gegen einen relativ großen quadratischen Fehler und erreicht nur eine Erkennnungsrate von ca. 60%.



(a) Durchschnittlicher quadratischer Fehler (logarithmische Skala)

(b) Prozentsatz falsch klassifizierter Muster

Abbildung 6.3: Auswirkungen unterschiedlicher Weight-Decay-Faktoren λ auf den Lernerfolg auf 500 Mustern der Trainingsmenge des NORB-Datensatzes. Bei zu großem λ werden die Gewichte zu klein gehalten und das Netz konvergiert schlecht. Ein kleiner Wert führt jedoch zu starken Sprüngen auf der Fehleroberfläche. Als Anhaltspunkt für einen guten Wert konnte $\lambda = 0{,}000$ ermittelt werden.

Zusätzlich wurde das Lernverfahren Rprop (siehe Abschnitt 2.1.4) getestet, welches adaptiv für jedes Gewicht eine eigene Lernrate wählt. Nach nur wenigen Epochen kann das Netz damit die 500 Trainingsmuster lernen. Anhand der Lernkurve sieht man jedoch auch, dass das Netz ab einem gewissen Punkt wieder einige Muster verlernt. Dies ist darauf zurückzuführen, dass die Gewichte mit der Zeit zu groß werden. Zum effizienten Einsatz von Rprop ist daher eine Regularisierung der Gewichte notwendig, die im nächsten Abschnitt erörtert wird.

Weight Decay

Ein wirksames Mittel zur Gewichtsregularisierung bei Rprop ist die in Abschnitt 2.1.4 vorgestellte Methode Weight Decay. Entscheidend für den Lernerfolg ist dabei die Wahl des Decay-Faktors λ, mit dem indirekt gesteuert werden kann, wie groß die Gewichte werden sollen. Ist dieser Faktor zu klein gewählt, ergeben sich dieselben Probleme wie ohne Weight Decay, nämlich dass die Gewichte zu stark wachsen. Wird der Faktor zu groß gewählt, dann generalisiert das Netz zu stark und lernt gar nicht oder nur die A-priori-Wahrscheinlichkeiten der Klassen.

Um einen geeigneten Decay-Faktor zu finden, wurde ein ähnliches Netz verwendet wie im letzten Abschnitt. Es wurde jedoch auf die zusätzlichen Eingabekarten der höheren Schichten verzichtet, damit zwangsläufig alle Schichten am Lernverfahren beteiligt sind. Das Netz wurde erneut auf nur 500 Mustern der Trainingsmenge des NORB-Datensatzes mit Rprop trainiert.

Je mehr Eingaben ein Neuron erhält, desto kleiner sollen diese Gewichte ausfallen. Daher wird der Decay-Faktor auf Schicht l abhängig von der Anzahl der eingehenden Verbindungen c_{in} gewählt:

$$\lambda^{(l)} = \lambda \cdot c_{in} \tag{6.2}$$

Somit kann mit der Variable λ für sämtliche Schichten festgelegt werden, wie stark das Weight Decay ausfallen soll. Für einige der experimentell untersuchten Parametrisierungen

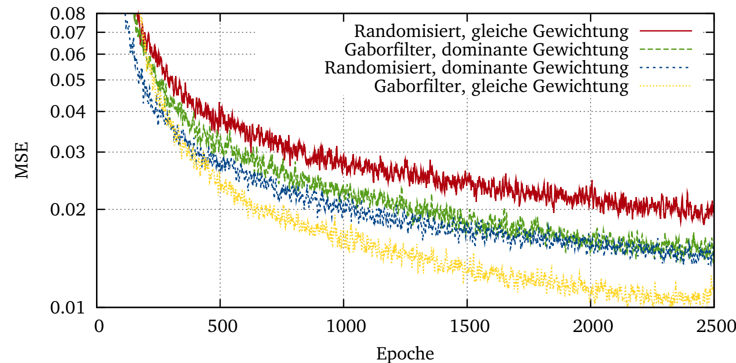


Abbildung 6.4: Entwicklung des durchschnittlichen quadratischer Fehlers auf der Arbeitsmenge für eine kleine Teilmenge des NORB-Datensatzes. Die Initialisierung der Gewichte der Konvolutionsschichten wurde unterschiedlich gewählt, alle anderen Lernparameter sind gleich gewählt. Bei einer Initialisierung mit Gaborfiltern konvergiert das Verfahren deutlich schneller als mit anderen Initialisierungen.

dieser Variablen zeigt Abbildung 6.3 die Ergebnisse des Lernverfahrens. Man sieht deutlich, dass bei zu kleiner Wahl des Parameters ($\lambda = 0{,}00001$) keine Regularisierung stattfindet und das Netz – wie schon im vorigen Abschnitt gesehen – bereits gelernte Muster wieder verlernt. Bei $\lambda = 0{,}1$ bleibt die Fehlerquote dauerhaft bei 0%, der durchschnittliche quadratische Fehler konvergiert aber gegen einen hohen Wert. Als ideal hat sich in diesem Experiment die Wahl $\lambda = 0{,}0001$ herausgestellt.

Der hier ermittelte Decay-Faktor ist nur als Richtwert anzusehen, da sich nicht pauschal für alle Netze ein idealer Wert angeben lässt. Abhängig von der Anzahl der Schichten und Merkmalskarten, der Größe der Arbeitsmenge, dem Prozentsatz auszutauschender Muster und weiteren Faktoren ist zur Optimierung der Erkennungsleistung in der Regel eine individuelle Anpassung notwendig. Auch von der in diesem Experiment nicht untersuchten Generalisierungsleistung auf der Testmenge kann diese Parameterwahl abhängen.

Gewichtsinitialisierung

Zur Untersuchung verschiedener Gewichtsinitialisierungen (siehe Abschnitt 4.3.2) wurde erneut das in den vorherigen Abschnitten verwendete Konvolutionsnetz eingesetzt. Als Trainingsmenge diente eine 2000 Muster große Untermenge des NORB-Datensatzes. Das Training erfolgte jedoch nicht direkt auf dieser Menge, sondern auf einer Arbeitsmenge aus 500 Mustern, von denen 100 jede Epoche ausgetauscht wurden.

Die Filter wurden für diesen Versuch wie in Abschnitt 4.3.2 beschrieben entweder mit Zufallswerten oder als Gaborfilter mit randomisierten Parametern initialisiert. Außerdem können die Filter einer Schicht $l+1$ durch einen Faktor γ_{ij} gewichtet werden. Bei der Wahl $\gamma_i = 1$ fließen alle Merkmalskarten i der Schicht l mit derselben Gewichtung in die Berechnung der Merkmalskarte j ein. Alternativ kann eine Merkmalskarte i die Merkmalskarte j der darauffolgenden Schicht dominieren, indem deren Gewichtung mit $\gamma = I \ \cdot 0{,}9$ gewählt wird. Alle anderen Merkmalskarten erhalten dann die Gewichtung $\gamma = 0{,}1$. Durch die zufällige Auswahl einer Karte i für jede Karte j soll die Symmetrie gebrochen und eine spärliche Verknüpfung von Merkmalskarten wie in dem Modell von LeCun *et al.* (siehe Seite 16) simuliert werden.

(a) Skaliert (b) Geglättet (c) Vertikaler Gradient (d) Horizontaler Gradient

(e) Invertiert (f) Dilation (g) Erosion (h) Laplace

Abbildung 6.5: Eingabemuster zum Lernen bildartiger Abbildungen (512×512).

Abbildung 6.6: Die gewünschten Ausgabebilder des Netzes (64×64 Pixel) wurden durch acht verschiedene Bildverarbeitungsoperationen manuell erstellt.

Aus der Kombination dieser Methoden ergeben sich vier Möglichkeiten, die Gewichte zu initialisieren. Abbildung 6.4 zeigt die Lernkurve auf der Arbeitsmenge für die unterschiedlichen Kombinationen. Daraus wird ersichtlich, dass sowohl die Initialisierung mit Gaborfiltern als auch die spärliche Verknüpfung von Merkmalskarten symmetriebrechende Wirkung haben, was deutlich zum Lernerfolg beiträgt.

Obwohl dominante Verknüpfungen ebenso wie Gaborfilter den Lernerfolg verbessern, kann mit einer Kombination dieser beiden Methoden erstaunlicherweise jedoch keine Verbesserung erzielt werden.

6.2 Experimente mit unterschiedlichen Datensätzen

Das in dieser Diplomarbeit entwickelte Konvolutionsnetz wurde auf unterschiedliche Datensätze angewandt, um den Lernerfolg dieser Architektur mit anderen Verfahren zu vergleichen. Je nach Datensatz wurden dazu eine andere Netzarchitektur und verschiedene Lernparameter gewählt. Die Ergebnisse in diesem Abschnitt beziehen sich auf das jeweilige Netz, welches den besten Lernerfolg erzielen konnte.

6.2.1 Lernen bildartiger Abbildungen

Im ersten Experiment soll das Konvolutionsnetz relativ simple zweidimensionale Faltungsfilter lernen, die zunächst ein Eingabebild herunterskalieren und darauf typische Bildverarbeitungsoperationen wie z.B. Kantenextraktionen durchführen. Dieses Experiment dient zum einen als erster Indikator für die Leistungsfähigkeit der in Abschnitt 4.1 beschriebenen Netzarchitektur, zum anderen zur Verifizierung der Implementierung der Konvolutionsschichten.

Datensatz

Die zu lernende Datenmenge ist sehr klein und besteht nur aus je vier Mustern für Trainings- und Testmenge. Eingabe sind 512×512 Pixel große, aus Fotos ausgeschnittene Graustufen-

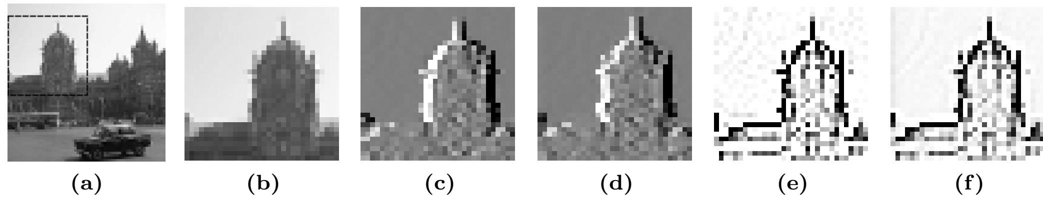


Abbildung 6.7: Vergleich zwischen Teacher und Ausgabe eines für bildartige Abbildungen trainierten Netzes nach 5000 Epochen. (a) Eingabebild der Testmenge. (b) Vergrößerter Ausschnitt des Eingabebildes. (c) Teacher des horizontalen Gradientenfilters. (d) Gelernte Ausgabe für den horizontalen Gradientenfilter. (e) Teacher des Laplace-Filters. (f) Gelernte Ausgabe für den Laplace-Filter.

bilder, aus denen manuell mit dem Bildverarbeitungsprogramm Gimp[1] acht verschiedene 64×64 Pixel große Ausgabebilder erzeugt wurden (siehe Abbildung 6.6).

Für die ersten beiden Ausgaben wurde das Bild jeweils skaliert und dabei einmal nicht interpoliert und einmal mit kubischer Interpolation geglättet. Die nächsten beiden Ausgabebilder entstanden, indem ein einfacher horizontaler bzw. vertikaler Gradientenfilter mit Kern $[-1,0,1]$ auf das nicht interpolierte Bild angewandt wurde. Für eine weitere Ausgabe wurden lediglich die Pixel des skalierten Bildes invertiert. Etwas komplexer sind die durch Anwendung von morphologischen Filtern erzeugten Bilder, in denen helle (Dilation) bzw. dunkle (Erosion) Strukturen verstärkt werden. Schwierigste Abbildung dieses Datensatzes soll der Laplace-Filter sein, der auf das geglättete Ausgabebild angewandt wurde und Kanten aus dem Bild extrahiert.

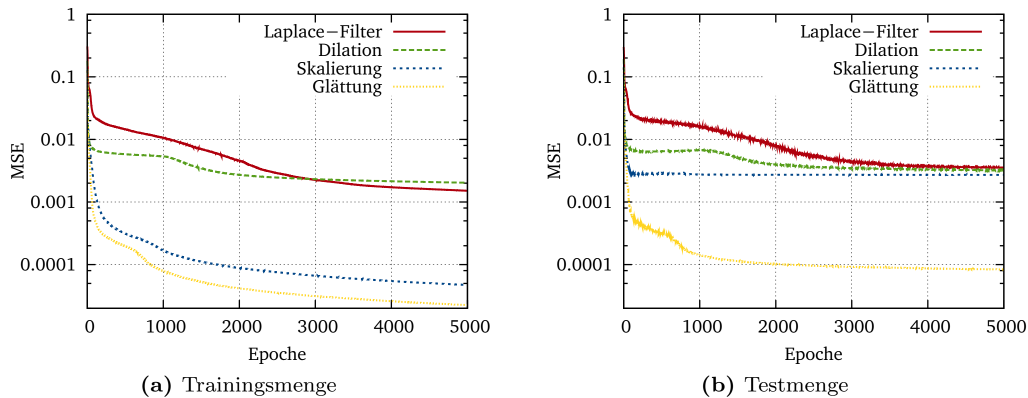
Für ein vollverknüpftes Netz wäre ein Datensatz von vier Mustern deutlich zu klein. Aufgrund der gekoppelten Gewichte stellt jedoch jeder Bildpunkt der 64×64 Pixel großen Ausgabekarte ein Trainingsmuster der zu lernenden Abbildung dar. Bei Verwendung eines Netzes mit geringer Kapazität sollte es also möglich sein, diese Abbildungen zu lernen, ohne dass ein Overfitting der Trainingsmenge stattfindet.

Netzstruktur

Da die Ausgabe des Konvolutionsnetzes zweidimensionale Bilddaten sein sollen, unterscheidet sich das hier verwendete Netz von der in Abschnitt 4.1 beschriebenen Architektur dadurch, dass es keine vollverknüpften Schichten enthält. Das Netz besteht also aus einer einzigen Eingabekarte von 512×512 Pixeln und drei weiteren Schichten mit Merkmalskarten von 256×256, 128×128 und 64×64 Pixeln. Bis auf die Eingabeschicht besteht jede Schicht aus der durch den Kernel zur Vorwärtspropagierung (siehe Abschnitt 5.3) bedingten Mindestanzahl von acht Merkmalskarten. Da nicht alle Abbildungen gleichzeitig, sondern jeweils nur eine gelernt werden soll, wird die gewünschte Ausgabe in jede der acht Ausgabekarten kopiert.

In der ersten Schicht L1 werden also acht Konvolutionsfilter mit je 65 Gewichten eingesetzt, auf den anderen beiden Schichten L2 und L3 sind es jeweils 64 Filter. Insgesamt verfügt das Netz daher nur über 8840 trainierbare Gewichte, die wie in Abschnitt 4.3.2 beschrieben zufällig initialisiert wurden. Als Lernverfahren wurde Rprop (siehe Abschnitt 2.1.4) ohne jegliche Erweiterungen als Offline-Training auf die vier Trainingsmuster angewandt.

[1] `http://www.gimp.org/`



(a) Trainingsmenge

(b) Testmenge

Abbildung 6.8: Logarithmisch skalierter durchschnittlicher quadratischer Fehler auf Trainings- und Testmenge für vier der gelernten Abbildungen.

Ergebnisse

Das Konvolutionsnetz konnte jede dieser Abbildungen mit für das menschliche Auge hinreichender Genauigkeit aus der Trainingsmenge lernen. Abbildung 6.7 zeigt anhand eines Bildausschnitts der Testmenge für den horizontalen Gradientenfilter und den Laplace-Kantenextraktor, dass der Unterschied zwischen gewünschter und tatsächlicher Ausgabe nur gering ist.

Wie man anhand des in Abbildung 6.8 gezeigten durchschnittlichen quadratischen Fehlers erkennen kann, ist das skalierte und geglättete Bild am leichtesten zu erlernen. Die rechnerisch simplere nicht interpolierte Abbildung lässt sich dagegen schlechter lernen. Dass diese Abbildung für das Konvolutionsnetz deutlich schwieriger ist, lässt sich dadurch erklären, dass jeder Filter nur ein konkretes Pixel der Eingabe weiterpropagieren darf. Wenn durch die Initialisierung fälschlicherweise das benachbarte Pixel gewählt wurde, konvergiert das Netz gegen ein lokales Minimum.

Auffällig ist auch, dass die Lernkurven für einige der Abbildungen – insbesondere die Dilation – anfänglich auf einem Plateau verharren und erst danach wieder deutliche Fortschritte machen. Obwohl hier der agressive Rprop-Algorithmus ohne Regularisierung der Gewichte angewandt wurde und trotz der relativ kleinen Trainingsmenge ist kein Overfittung auf der Testmenge festzustellen, was die Generalisierungsleistung von Konvolutionsnetzen verdeutlicht.

Als weiterer Test wurde versucht, alle acht oben genannten Abbildungen auf einmal zu lernen, indem jede der acht Ausgabekarten den Teacher einer anderen Abbildung enthält. Da die Anzahl der trainierbaren Parameter gegenüber den vorherigen Experimenten gleich bleibt, stellt es sich erwartungsgemäß als deutlich schwieriger heraus, diese teils sehr unterschiedlichen Abbildungen gleichzeitig zu lernen. Nach etwa drei Mal so vielen Epochen konnte jedoch derselbe Trainingserfolg erzielt werden.

6.2.2 Erkennung handschriftlicher Ziffern

Um die Erkennungsleistung des implementierten Konvolutionsnetzes auch mit anderen Verfahren zur Objekterkennungn vergleichbar zu machen, wurde die MNIST-Datenbank handschriftlicher Ziffern (siehe Abschnitt 2.3.1) als Benchmark gewählt. Es wurde nämlich schon

in anderen Arbeiten gründlich erforscht, welche Erkennungsleistungen mit unterschiedlich parametrisierten Klassifikatoren möglich sind. Die im Folgenden vorgestellten Ergebnisse beziehen sich auf das Netz, mit dem die beste Erkennungsleistung erreicht wurde. Welchen Einfluss Modifikationen des Lernverfahrens auf das Training haben, wurde bereits in Abschnitt 6.1 erläutert.

Netzstruktur und Datensatz

Wie bereits in Abschnitt 5.9.1 erwähnt, beträgt die minimale Größe von Merkmalskarten in den Konvolutionsschichten 32×32 Pixel. Damit das Netz auch in den Konvolutionsschichten genügend trainierbare Parameter zur Verfügung hat, werden weitere Schichten mit größeren Merkmalskarten benötigt. Für dieses Experiment wurde die Anzahl der Schichten auf vier festgelegt. Somit haben die Merkmalskarten der Eingabeschicht L0 eine Größe von 256×256 Pixeln, die drei darauffolgenden Schichten sind 128×128 (L1), 64×64 (L2) und 32×32 (L3) Pixel groß.

Die Muster der MNIST-Datenbank liegen als Graustufenbilder mit einer Auflösung von 28×28 Pixeln vor, daher muss eine geeignete Möglichkeit gefunden werden, diese auf die deutlich größere Eingabeschicht abzubilden. Es kommt nicht in Betracht, die kleinen Muster zentriert in der Eingabeschicht zu platzieren, weil durch das Subsampling nur wenige Pixel der Ausgabeschicht das Muster in ihrem rezeptiven Feld erfassen. Stattdessen werden die Muster auf 256×256 Pixel skaliert und mit Hilfe des Texturspeichers der CUDA-Grafikkarten linear interpoliert.

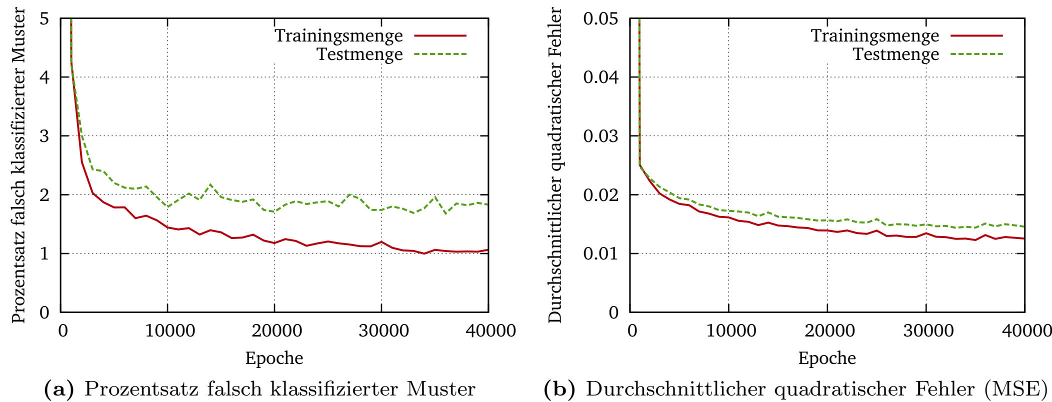
Die Pixelwerte der Eingabemuster wurden normalisiert, so dass der Hintergrund (weiß) einen Wert von $-0{,}1$ hatte und der Vordergrund (schwarz) einen Wert von 1,175. Für die verwendete Aktivierungsfunktion, den Tangens Hyperbolicus, ergeben sich damit ein idealer Durchschnittswert der Muster von ca. Null und eine Varianz von ca. Eins.

Bei den in Abschnitt 6.1 beschriebenen Versuchen stellte sich heraus, dass ein Netz mit dem Minimalwert von jeweils acht Merkmalskarten auf den Schichten L1, L2 und L3 zum Lernen der MNIST-Datenmenge ausreichend groß ist. Auch die Größe der beiden vollverknüpften Schichten wurde empirisch gewählt. Die Schicht F4 erhält als Eingabe $8 \cdot 32 \times 32 = 8192$ Werte der letzten Konvolutionsschicht und verfügt über 100 verdeckte Neuronen. Die Ausgabeschicht besteht aus zehn Neuronen, die mit einer 1-aus-N-Codierung angeben, welche Ziffer erkannt wurde.

Initialisierung und Lernverfahren

Der Vorteil des hier eingesetzten kleinen Netzes ist, dass ein Trainingsmuster inklusive der Aktivitäten auf allen Schichten nur wenig Speicherplatz benötigt. Somit kann die Arbeitsmenge (siehe Abschnitt 4.3.4) mit 250 Mustern relativ groß gewählt werden. Bei den Experimenten mit Rprop stellte sich heraus, dass beim Training zwar die Arbeitsmenge gut gelernt wurde, das Netz jedoch auf den restlichen Mustern der Trainingsmenge deutlich schlechtere Ergebnisse erzielte. Um diesem Auswendiglernen der wenigen Muster der Arbeitsmenge entgegenzuwirken, wird daher jede Epoche ein relativ großer Teil der Arbeitsmenge, nämlich 50 Muster, ausgetauscht. Die Muster werden nach dem in Abschnitt 4.3.5 beschriebenen gewichteten Selektionsschema in die Arbeitsmenge aufgenommen, so dass Muster, die einen überdurchschnittlich großen Fehler verursachen, bevorzugt werden.

Es hat sich außerdem gezeigt, dass Rprop dazu neigt, betragsmäßig sehr große Gewichte in den Konvolutionsschichten zu verursachen, was zur Folge hat, dass die Aktivitäten der Merkmalskarten nur den maximalen oder den minimalen Wert annehmen. In diesem Fall



(a) Prozentsatz falsch klassifizierter Muster **(b)** Durchschnittlicher quadratischer Fehler (MSE)

Abbildung 6.9: Lernkurven des Konvolutionsnetzes auf Trainings- und Testmenge der MNIST-Datenbank handschriftlicher Ziffern.

ist die Aktivierungsfunktion gesättigt und der Gradient fällt so klein aus, dass auch die Gewichtsanpassungen sehr klein sind (siehe Abschnitt 2.1.3). Daher wurde zur Regularisierung der Gewichte – wie in Abschnitt 6.1.2 beschrieben – Weight Decay eingesetzt.

Auch bei diesem Netz wurden die Gewichte mit einer Zufallsverteilung initialisiert, die von der Anzahl der auf dasselbe Neuron eingehenden Gewichte abhängt. Da die Merkmalskarten auf Schicht L1 nur Eingaben aus einer Merkmalskarte erhalten, wurden die Gewichte hier also zwischen −0,1240 und 0,1240 gewählt. Auf den höheren Konvolutonsschichten liegen die Gewichte zwischen −0,0439 und 0,0439.

Ergebnisse

Durch das Austauschen der Arbeitsmenge konvergiert das Lernverfahren deutlich langsamer als beim Online-Training. Auch nach 100.000 Trainingsepochen fluktuiert der durchschnitttliche quadratische Fehler sowohl auf der Trainings- als auch auf der Testmenge noch leicht (siehe Abbildung 6.9). Ein Overfitting kann dennoch nicht festgestellt werden. Abbildung 6.10 zeigt für zwei Trainingsbeispiele die Aktivitäten der Merkmalskarten des trainierten Netzes.

Bei der Variante mit randomisiertem Austauschen der Arbeitsmenge hat das Netz nur 1,65% der Testmuster falsch klassifiziert. Mit dem gewichteten Selektionsschema wurde eine weitere deutliche Verbesserung erreicht, so dass der Testfehler des so trainierten Netzes nur 1,38% beträgt. Abbildung 6.11 vergleicht diese Erkennungsrate mit den anderen in Abschnitt 2.3.1 beschriebenen Verfahren, die wie hier nicht die Eingabe modifizieren. Dabei wird ersichtlich, dass dieser Wert deutlich besser ist als die besten mit einem MLP erzielten Ergebnisse (2,95%). Die hier verwendete Netzarchitektur ist auch besser als ein einfaches Konvolutionsnetz (1,7%), aber deutlich schlechter als spezialisierte Konvolutionsnetze (0,6% bis 1,10%).

6.2.3 Objektklassifizierung auf dem NORB-Datensatz

Auch der NORB-Datensatz (siehe Abschnitt 2.3.2) eignet sich zur Evaluierung von Objekterkennungsverfahren, weil Vergleichswerte für unterschiedliche Klassifikatoren vorliegen. Im

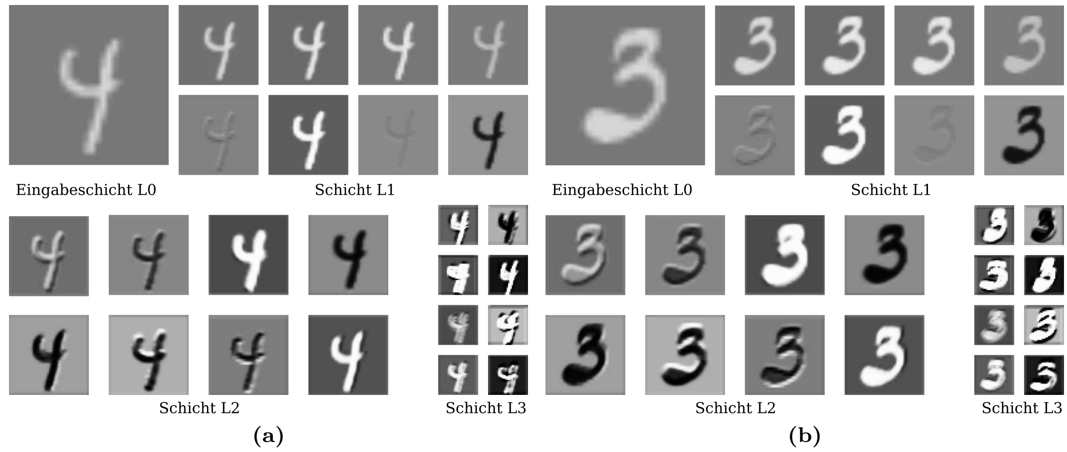


Abbildung 6.10: Die Aktivitäten der Merkmalskarten des trainierten Konvolutionsnetzes für zwei verschiedene Muster der MNIST-Datenbank. Die Merkmalskarten der Schicht L2 und L3 sind um den Faktor 2 vergrößert dargestellt.

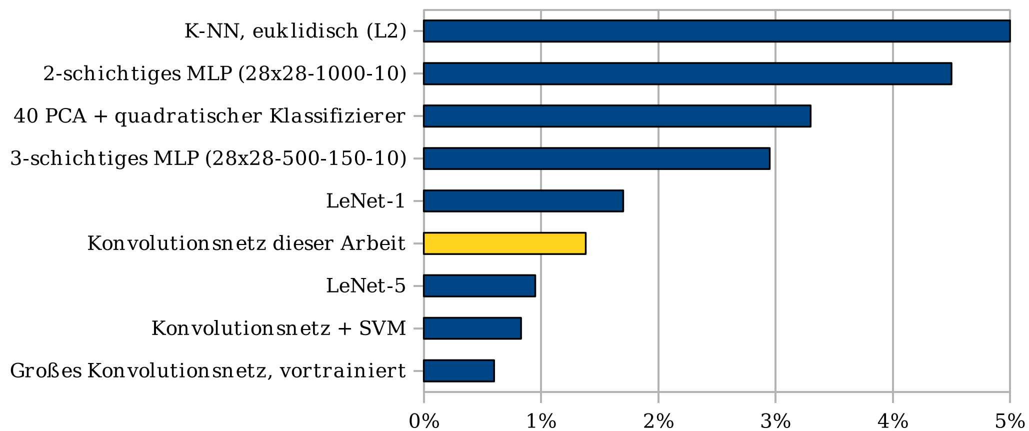


Abbildung 6.11: Vergleich der Fehlerquote verschiedener Klassifizierungsverfahren auf der Testmenge der MNIST-Datenbank. Dargestellt sind ausschließlich Verfahren, die unmodifizierte Eingabebilder zum Training verwenden.

Klasse	animal	human	airplane	truck	car
animal	0,923	0,037	0,038	0,000	0,001
human	0,145	0,846	0,009	0,000	0,000
airplane	0,059	0,000	0,922	0,001	0,018
truck	0,038	0,000	0,001	0,945	0,016
car	0,087	0,000	0,007	0,095	0,810

Tabelle 6.2: Konfusionsmatrix für das trainierte Konvolutionsnetz auf der Testmenge der NORB-Datenbank. Eine Zeile gibt an, wie hoch die Wahrscheinlichkeit ist, dass das Netz ein Objekt der gegebenen Klasse den fünf Kategorien zuordnet. Am schwierigsten sind Objekte der Klasse *car* zu klassifizieren, die relativ häufig fehlerhaft als *animal* oder *truck* erkannt werden.

Gegensatz zu komplizierteren Datensätzen sind die Objekte hier unabhängig von Beleuchtung, Umgebung, Farbe und Textur.

Netzstruktur

Die Eingaben des NORB-Datensatzes liegen als Stereopaar von Graustufenbildern mit einer Auflösung von 96×96 Pixeln vor. Naheliegend wäre es also, die Merkmalskarten der Eingabeschicht der Größe 128×128 Pixel zu wählen. Da die Mindestgröße von Merkmalskarten im Konvolutionsnetz aber 32×32 beträgt, wären somit nur zwei informationsverarbeitende Konvolutionsschichten möglich. Daher wird die Eingabe stattdessen in 256×256 Pixel großen Eingabekarten zentriert und die Bildränder werden wie in Abschnitt 4.4.1 beschrieben fortgesetzt und ausgeblendet.

Das Netz erhält als zusätzliche Eingabe auch auf höheren Konvolutionsschichten eines der beiden Bilder in passender Skalierung sowie das zugehörige Kantenbild in horizontaler und vertikaler Richtung. Die Eingabeschicht L0 besteht somit aus vier Eingabekarten, die Schicht L1 aus acht Merkmalskarten und drei Eingabekarten (128×128 Pixel) und die Schichten L2 und L3 aus jeweils 16 Merkmalskarten und drei Eingabekarten (64×64 Pixel bzw. 32×32 Pixel). Für die vollverknüpfte Schicht hat sich experimentell herausgestellt, dass das Netz mit wenigen Neuronen gut generalisiert, daher werden 20 Neuronen in Schicht F4 eingesetzt und die fünf Neuronen der Ausgabeschicht codieren die fünf möglichen Ausgabeklassen.

Initialisierung und Lernverfahren

Auch auf dem NORB-Datensatz wird als Lernverfahren Rprop verwendet. Da bei Verwendung einer Arbeitsmenge jedoch keine Netzparameter gefunden werden konnten, mit denen die Objekte auf der Trainingsmenge gut erlernbar waren, wurde stattdessen echtes Offline-Training auf dem gesamten 24.300 Muster umfassenden Trainingsdatensatz durchgeführt. Die Gewichte wurden mit Gaborfiltern initialisiert, die wie in Abschnitt 6.1.2 beschrieben durch dominante Verbindungen zusätzlich gewichtet wurden. Eine Regularisierung der Gewichte wurde mit Weight Decay mit einem Faktor von $\lambda = 0{,}0001$ vorgenommen.

Ergebnisse

Nach bereits 100 Trainingsepochen erreichte das Netz auf der Trainingsmenge einen Fehler von 0%. Insgesamt wurde es über 1000 Epochen trainiert, was bei der nicht optimierten GUI-Version des Programms ca. 96 Stunden Laufzeit in Anspruch genommen hat. Mit den so

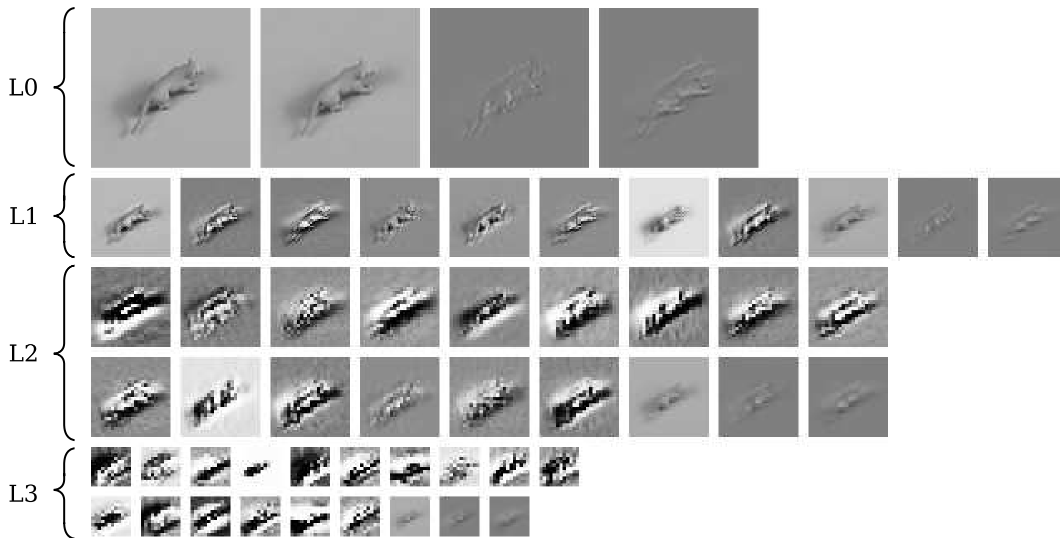


Abbildung 6.12: Die Aktivitäten der Merkmalskarten des Konvolutionsnetzes für eines der Trainingsmuster des NORB-Datensatzes. Die Merkmalskarten der Schicht L2 und L3 sind um den Faktor 2 vergrößert dargestellt.

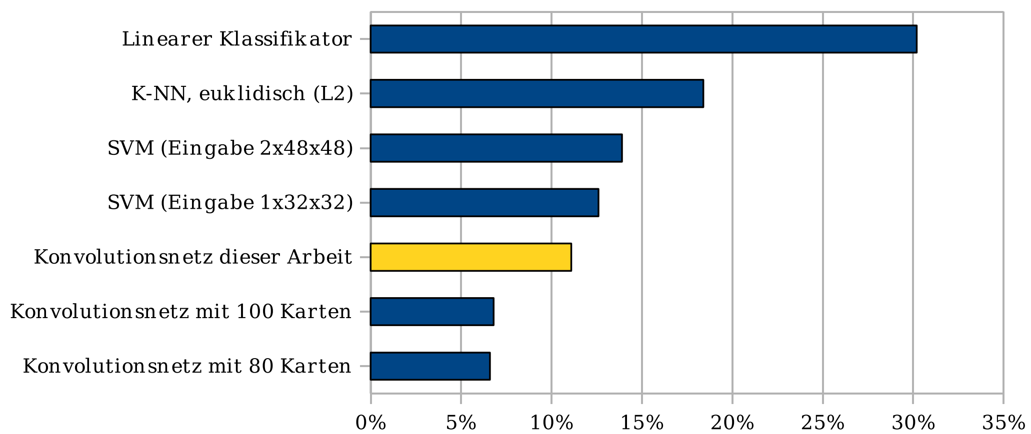


Abbildung 6.13: Vergleich der Fehlerquote verschiedener Klassifizierungsverfahren auf der Testmenge der NORB-Datenbank.

gelernten Gewichten konnten nur 11,07% der Testmuster nicht korrekt klassifiziert werden. Dabei zeigte sich, dass einige Klassen für das Netz deutlich schwieriger zu lernen sind. Aus der Konfusionsmatrix in Tabelle 6.2 zeigt sich, dass über 94,5% der Objekte der Klasse *truck* korrekt erkennt werden, von der Klasse *car* jedoch lediglich 81,0%. Abbildung 6.12 zeigt die Aktivitäten der Merkmalskarten des trainierten Netzes.

Zu den Ergebnissen anderer Verfahren auf dem NORB-Datensatz ist eine Aufschlüsselung nach Klassen nicht verfügbar. Anhand des Vergleichs der Erkennungsleistung in Abbildung 6.13 ist ersichtlich, dass das in dieser Arbeit beschriebene Konvolutionsnetz deutlich besser ist als ein linearer Klassifikator und eine K-Nearest-Neighbors-Klassifikation und geringfügig besser als eine SVM. Die von LeCun *et al.* verwendeten, auf diese Problemstellung spezialisierten Konvolutionsnetze sind mit einer Fehlerquote von 6,6% dagegen deutlich besser [LHB04]. Mit einem Online-Lernverfahren mit adaptiven gewichtsspezifischen Lernraten wurde dieses Netz jedoch 250.000 Epochen lang traininert.

6.2.4 Objektklassifizierung auf dem PASCAL-Datensatz

Der Datensatz der jährlich ausgeschriebenen PASCAL Challenge (siehe Abschnitt 2.3.3) gilt als einer der schwersten Datensätze zur Objekterkennung. Kaum eines der bisher auf diesem Datensatz erfolgreich angewandten Verfahren (siehe Abschnitt 3.1) verwendet neuronale Netze zur Klassifizierung. Aufgrund der geringen Anzahl an Trainingsmustern stellt dieser Datensatz für Lernverfahren wie Backpropagation of Error eine Herausforderung dar.

Netzstruktur und Datensatz

Da die Bilder der PASCAL-Datenbank eine maximale Auflösung von 500×500 Pixel haben, wurden die Merkmalskarten der Eingabeschicht L0 auf 512×512 Pixel festgelegt. Somit haben die Merkmalskarten der darauffolgenden Schichten eine Größe von 256×256 Pixeln (L1), 128×128 Pixeln (L2), 64×64 Pixeln (L3) und 32×32 Pixeln (L4).

Die drei Y'UV-Kanäle eines mehrfarbigen Bildes dienen als jeweils eine Eingabekarte. Bei Bildern, die kleiner als die Eingabekarte sind, werden – wie in Abschnitt 4.4 beschrieben – die Randpixel fortgesetzt und langsam ausgeblendet. Weitere Eingaben sind die Ortscodierung sowie vertikale und horizontale Kantenfilter, die auf dem Luminanzkanal berechnet werden. Auf oberster Schicht erhält das Netz somit sechs unterschiedliche Eingaben. Auch die höheren Schichten enthalten jeweils drei Eingabekarten, nämlich den skalierten Y'-Kanal und die darauf angewandten horizontalen und vertikalen Kantenfilter.

Die Anzahl der Merkmalskarten auf jeder Schicht wurde so gewählt, dass trotz hohen Speicherbedarfs 30 Trainingsmuster parallel berechnet werden können. Neben den jeweils drei Eingabekarten pro Schicht enthält das Netz acht Merkmalskarten in Schicht L1, 16 Merkmalskarten in Schicht L2 und jeweils 32 Merkmalskarten in den Schichten L3 und L4.

Das Konvolutionsnetz wurde ausschließlich auf der 2.113 Bilder umfassenden Trainingsmenge trainiert und die Validierungsmenge diente zur Evaluierung der Ergebnisse. Um eine größere Variabilität der Trainingsmuster zu erreichen, wurden die Bilder jede Epoche zufällig um ±32 Pixel in horizontaler und vertikaler Richtung vom Bildmittelpunkt verschoben in den Merkmalskarten platziert.

Initialisierung und Lernverfahren

Auch bei diesem Netz wurde Rprop mit echtem Offline-Training als Lernverfahren eingesetzt, da das Konvolutionsnetz bei sämtlichen Experimenten mit einer Arbeitsmenge gegen

Klasse	AP
Aeroplane	39,0
Bicycle	8,4
Bird	18,7
Boat	29,5
Bottle	9,3
Bus	3,8
Car	15,4
Cat	11,6
Chair	13,6
Cow	2,1

Klasse	AP
Diningtable	4,8
Dog	12,3
Horse	7,0
Motorbike	7,8
Person	65,6
Pottedplant	6,5
Sheep	9,4
Sofa	14,0
Train	9,2
Tvmonitor	9,3

Tabelle 6.3: Durchschnittliche Genauigkeit (*average precision*, AP) des trainierten Konvolutionsnetzes für jede der 20 Objektklassen der Validierungsmenge.

schlechtere Werte konvergierte. Um zu große Gewichte zu verhindern und die Generalisierungsleistung auf der Validierungsmenge zu verbessern, wurde Weight Decay mit einem relativ großen Faktor von $\lambda = 0{,}001$ eingesetzt.

Die Gewichte wurden mit Gaborfiltern initialisiert. Da bei den vorausgegangenen Experimenten zur Ermittlung guter Netzparameter zu beobachten war, dass der Fehler oszillierte und einige Lernraten das im Rprop-Algorithmus festgelegte Minimum annahmen, wurde dieser Wert mit $\Delta_{min} = 10^{-16}$ deutlich kleiner als üblich gewählt.

Ergebnisse

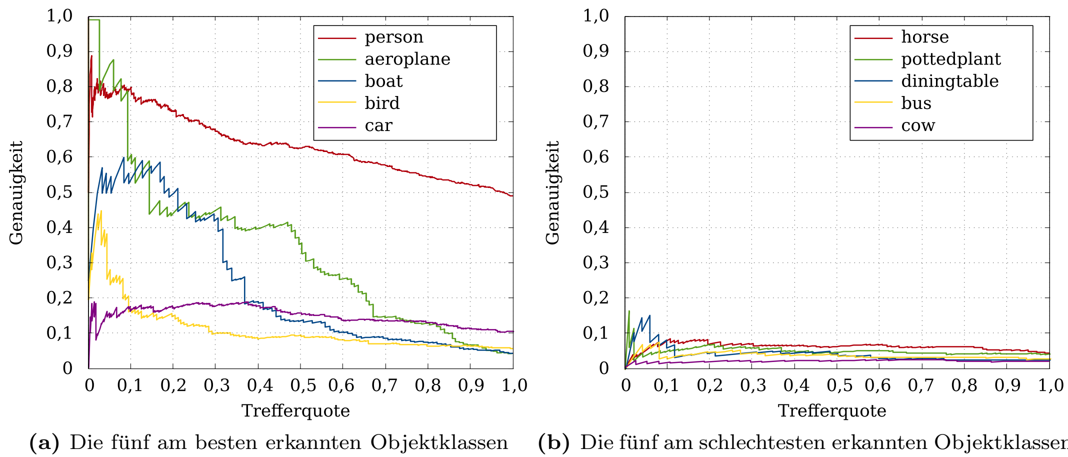
Da die Objektmarkierungen und Annotationen für die Testmenge des Datensatzes nicht öffentlich zugänglich sind, sondern nur im Rahmen des PASCAL Wettbewerbs von den Veranstaltern zur Fehlermessung verwendet werden, musste hier zur Evaluierung der Ergebnisse die deutlich kleinere Validierungsmenge verwendet werden. Ein direkter Vergleich mit den anderen Teilnehmern der PASCAL VOC 2008 ist somit nicht möglich, da zu deren Verfahren lediglich die Ergebnisse auf der Testmenge bekannt sind.

Das Offline-Trainingsverfahren wurde ca. 1000 Epochen mit allen 2221 Mustern der Trainingsmenge durchgeführt. Anschließend betrug der quadratische Fehler auf der Traingsmenge 0,0703 und auf der Validierungsmenge 0,5631. Die durchschnittliche Genauigkeit (*average precision*, siehe Abschnitt 2.3.3) auf der Validierungsmenge ist für jede der 20 Objektklassen in Tabelle 6.3 angegeben.

Wie auch bei den anderen in Abschnitt 3.1 vorgestellten Verfahren ist die Erkennungsleistung insbesondere auf der Klasse *person* überdurchschnittlich hoch. Dies kann mit der großen A-priori-Wahrscheinlichkeit dieser Klasse erklärt werden. Weitere gut erkannte Klassen sind *aeroplane* und *boat*, die möglicherweise anhand des Kontexts (häufig ein blauer Hintergrund) erkannt werden. Abbildung 6.14 stellt die Precision/Recall-Kurven für die am besten und am schlechtesten erkannten Objektklassen dar.

In Abbildung 6.15 werden diese auf der Validierungsmenge gemessenen Werte mit den auf der Testmenge erzielten Ergebnissen anderer Verfahren verglichen. Da die Klassenverteilung in beiden Mengen ähnlich ist, kann dieser Vergleich zumindest als Anhaltspunkt für die Leistung des hier implementierten Systems dienen.

Aus dem Vergleich wird ersichtlich, dass die Erkennungsleistung des Konvolutionsnetzes noch weit hinter anderen Verfahren zurückliegt.



(a) Die fünf am besten erkannten Objektklassen **(b)** Die fünf am schlechtesten erkannten Objektklassen

Abbildung 6.14: Precision/Recall-Kurven für einige der Objektklassen des PASCAL-Datensatzes. Die Kurven zeigen die Ergebnisse auf der Validierungsmenge für ein Konvolutionsnetz, das auf der Trainingsmenge trainiert wurde.

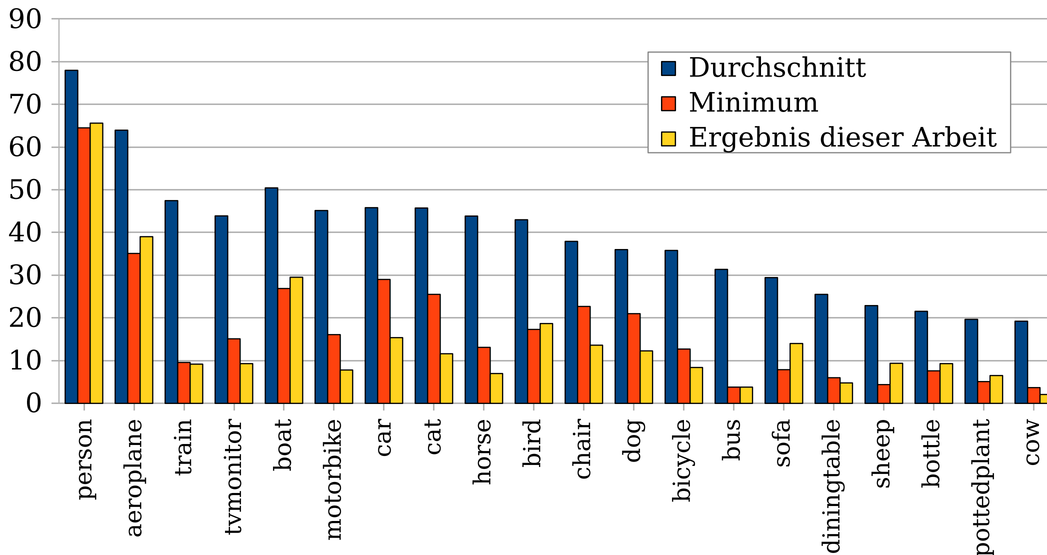


Abbildung 6.15: Vergleich der durchschnittlichen Genauigkeit (AP) des in dieser Arbeit implementierten Konvolutionsnetzes mit den Ergebnissen der PASCAL 2008 VOC. Für das Konvolutionsnetz wurde die AP auf der Validierungsmenge gemessen, bei allen anderen Verfahren auf der Testmenge. Gezeigt werden für jede Klasse der durchschnittliche Wert aller 18 Wettbewerbsteilnehmer sowie das Ergebnis des jeweils schlechtesten Teilnehmers.

6.3 Laufzeitanalyse

Eine maßgeblich zu klärende Frage dieser Diplomarbeit war es, ob sich ein neuronales Konvolutionsnetz mit paralleler Grafikkartenhardware so gut beschleunigen lässt, dass der Implementierungsaufwand gerechtfertigt wird. Wie in Abschnitt 3.3 für andere wissenschaftliche Anwendungen gezeigt wurde, kann ein geeigneter Algorithmus um bis zu zwei Größenordnungen beschleunigt werden. Im Folgenden soll daher ermittelt werden, wie viel schneller parallele Implementierung des Konvolutionsnetzes (siehe Kapitel 5) gegenüber einer CPU-Version ist.

Sämtliche Laufzeitmessungen wurden auf einem System mit einem *Intel Core i7 940* (2,93 GHz) und der CUDA-fähigen Grafikkarte *GeForce GTX 285* durchgeführt. Die CPU-Version wurde mit der laufzeitoptimierenden Option `-O3` compiliert, verwendet keinerlei Parallelisierungen und läuft auf nur einem der Prozessorkerne.

6.3.1 Kernellaufzeiten

Kernstück der Implementierung sind die in den Abschnitten 5.3 und 5.5 beschriebenen Kernelfunktionen zur Vorwärtspropagierung und zur Rückwärtspropagierung. Ohne eine Optimierung dieser Funktionen würde die Gesamtlaufzeit des Konvolutionsnetzes dramatisch steigen. Daher werden die Laufzeiten dieser Funktionen detailliert analysiert.

Durchgeführte Messungen

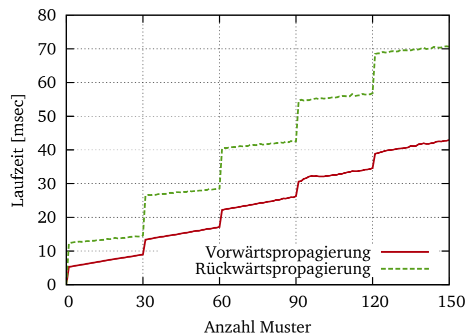
Um die Laufzeiten der optimierten Implementierung unabhängig von der Netzstruktur zu analysieren, wird im Folgenden zunächst nur die Berechnung zwischen zwei Schichten untersucht. Sowohl bei der Vorwärtspropagierung als auch bei der Rückwärtspropagierung wurden für diese Messungen sämtliche Berechnungen und Speichertransaktionen durchgeführt, die auch zur Berechnung des gesamten Konvolutionsnetzes notwendig sind. Nicht in den Messungen berücksichtigt ist das Austauschen der Trainingsmuster.

Vorwärtspropagierung Die Kernelfunktion zur Vorwärtspropagierung (*Forward-Kernel*) wird auf einer Merkmalskarte der Größe 256×256 für P Muster ausgeführt. Die Ausgabe erfolgt in acht Merkmalskarten der Größe 128×128, so dass acht Konvolutionen mit einem 8×8 Filter durchgeführt werden. Anschließend wird in einem separaten Kernel die Aktivierungsfunktion auf die Netzeingabe jeder dieser Merkmalskarten angewandt, um die Aktivitäten dieser Schicht zu erhalten.

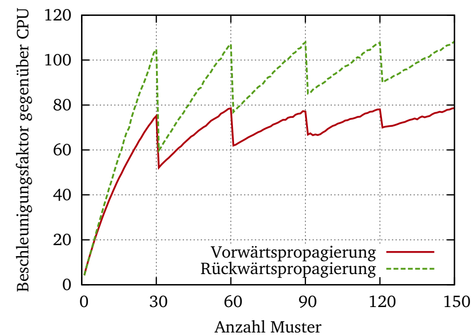
Rückwärtspropagierung Bei einem Schritt der Rückwärtspropagierung wird der Backprop-Kernel aufgerufen, der die Gradienten aller Gewichte aus den Fehlersignalen berechnet. Gleichzeitig wird der Fehler von acht 128×128 Pixel großen Merkmalskarten auf eine Merkmalskarte der Größe 256×256 zurückpropagiert. Von einem weiteren Kernel wird anschließend die Aktualisierung der Gewichte durchgeführt.

Laufzeit in Abhängigkeit der Anzahl parallel berechneter Muster

Wie in Abschnitt 5.2.1 beschrieben, parallelisiert die CPU-Implementierung in erster Linie über die Anzahl der simultan berechneten Trainingsmuster. Wenn viele Muster gleichzeitig trainiert werden können ist es also zu erwarten, dass ein hoher Beschleunigungsfaktor gegenüber der CPU-Implementierung erreicht wird.



(a) Laufzeit von Vorwärts- und Rückwärtspropagierung der GPU-Implementierung in Abhängigkeit von den parallel berechneten Mustern



(b) Beschleunigungsfaktor gegenüber einer äquivalenten CPU-Implementierung in Abhängigkeit von den parallel berechneten Mustern

Abbildung 6.16: Die Laufzeit der GPU-Implementierung hängt stark davon ab, wieviele Muster parallel berechnet werden können. Für diese Messung wurden die Berechnungen zwischen einer 256×256 Pixel großen Quellkarte und acht 128×128 Pixel großen Zielkarten verglichen.

	256×256			512×512		
	CPU	GPU	Faktor	CPU	GPU	Faktor
Vorwärtspropagierung	1358,99	17,18	79 ×	5645,51	71,03	79 ×
Rückwärtspropagierung	1599,35	20,30	79 ×	6411,79	80,36	80 ×
R. mit Fehlersignalberechnung	3142,23	28,67	110 ×	12525,28	113,74	110 ×

Tabelle 6.4: Laufzeiten von CPU- und GPU-Version (in Millisekunden) für die Vorwärts- und Rückwärtspropagierung von 60 Mustern bei unterschiedlich großen Merkmalskarten.

Abbildung 6.16a zeigt die Laufzeiten der Vorwärts- und Rückwärtspropagierung der GPU-Implementierung in Abhängigkeit von den parallel berechneten Trainingsmustern. Insbesondere bei der Rückwärtspropagierung ist zu beobachten, dass die Laufzeit trotz steigender Anzahl an Mustern zunächst konstant bleibt und beim Übergang von 30 auf 31 Muster plötzlich ansteigt. Dieser Sprung in der Laufzeit wiederholt sich alle 30 Muster und ist auch bei der Rückwärtspropagierung zu beobachten.

Ein Blick auf die Spezifikationen der eingesetzten Grafikkarte erklärt dieses Phänomen: Die *GeForce GTX 285* verfügt über 30 Multiprozessoren und in der Implementierung wird jeweils ein Muster von einem Multiprozessor verarbeitet. Auf die Laufzeit hat es also kaum Einfluss, ob einer oder 30 Multiprozessoren gleichzeitig Berechnungen durchführen. Wenn jedoch 31 Muster verarbeitet werden sollen, muss das letzte Muster zunächst warten, bis einer der Multiprozessoren wieder frei wird; die Laufzeit verdoppelt sich also.

Bei der Vorwärtspropagierung fallen diese Sprünge weniger stark aus und es fällt deutlich auf, dass die Laufzeit intervallweise linear ansteigt. Der Grund hierfür liegt in dem Kernel, der die rechenzeitintensive Aktivierungsfunktion berechnet. Er lastet alle Multiprozessoren gleichermaßen aus, indem jedem Multiprozessor ein gewisser Anteil der zu berechnenden Pixel zugewiesen wird.

Beschleunigungsfaktor

Eine effiziente Auslastung der Multiprozessoren ist somit nur möglich, wenn die Anzahl der Muster ein Vielfaches von 30 ist. In Gegensatz dazu steigt die Laufzeit der CPU-Version linear mit der Anzahl der Muster. Tabelle 6.4 vergleicht die Laufzeiten der beiden Versionen für die Verarbeitung von 60 Mustern. Der *Beschleunigungsfaktor* gibt das Verhältnis von CPU-Geschwindigkeit zu GPU-Geschwindigkeit an. Daraus wird ersichtlich, dass die parallele Implementierung unabhängig von der Größe der Eingabekarte bei der Vorwärtspropagierung ca. 80 Mal schneller ist.

Die Beschleunigung der Rückwärtspropagierung hängt von einem weiteren Faktor ab. Für Eingabekarten ist es wie in Abschnitt 5.4.2 beschrieben nicht notwendig, die Fehlersignale zu berechnen, sondern es reicht aus, nur die Gewichtsanpassung durchzuführen. Auf diesen weniger rechenintensiven Eingabeschichten wird ein Beschleunigungsfaktor von ca. 80 erreicht. Ein Konvolutionsnetz verfügt in der Regel jedoch über wesentlich mehr reguläre Schichten, und hier kann die Rückwärtspropagierung um den Faktor 110 beschleunigt werden.

Abbildung 6.16b stellt den Beschleunigungsfaktor in Abhängigkeit von der Anzahl parallel berechneter Muster dar. Im ungünstigsten Fall fällt die Beschleunigung demnach deutlich niedriger aus. In der Regel stellt es jedoch keine große Einschränkung dar vorauszusetzen, dass 30 Muster (oder ein Vielfaches davon) parallel berechnet werden sollen. Im Folgenden wird daher für die Laufzeitmessungen angenommen, dass über 60 Muster parallelisiert wird.

6.3.2 Gesamtlaufzeit

Die Gesamtlaufzeit einer Epoche des Lernverfahrens setzt sich nicht nur aus den Berechnungen aller Konvolutionsschichten zusammen, sondern hängt auch von Speichertransfers und den Berechnungen der vollverknüpften Schichten ab. Bei einer bandbreitenbeschränkten Anwendung sind es gerade die Speichertransfers, die eine Begrenzung der Laufzeit darstellen.

Eine Epoche des Lernverfahrens der GPU-Implementierung des Konvolutionsnetzes kann grob in folgende sieben Schritte unterteilt werden:

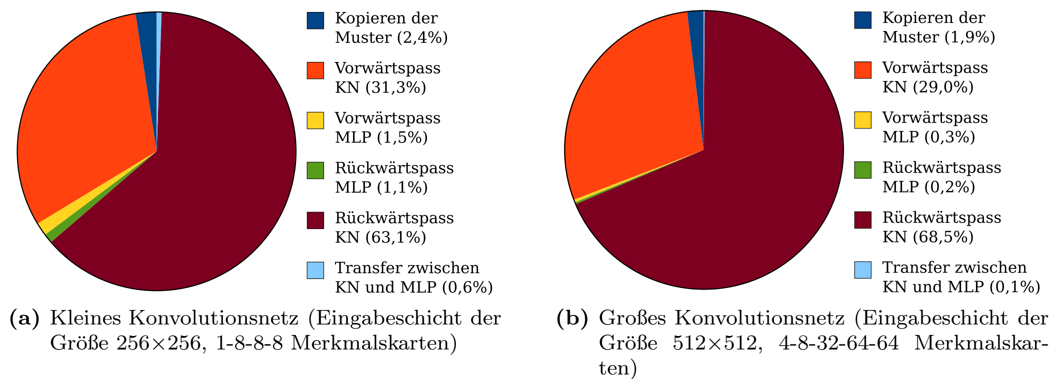
Kopieren der Trainingsmuster Die Aktivitäten der Trainingsmuster werden vom Hauptspeicher der CPU in den globalen Speicher der GPU kopiert. Für die hier durchgeführte Laufzeitanalyse wird vom schlimmsten Fall ausgegangen, nämlich dass sämtliche Muster der Arbeitsmenge ausgetauscht werden müssen.

Vorwärtspropagierung in den Konvolutionsschichten Für jede Konvolutionsschicht wird der Forward-Kernel auf sämtlichen Merkmalskarten ausgeführt. Aus den so berechneten Netzeingaben werden durch Anwendung der Aktivierungsfunktion die Aktivitäten der Merkmalskarten ermittelt.

Kopieren der Aktivitäten Da Konvolutionsschichten und vollverknüpfte Schichten unterschiedliche Datenstrukturen verwenden, werden die Aktivitäten der letzten Konvolutionsschicht in die Eingabematrix des vollverknüpften Netzes kopiert.

Vorwärtspropagierung in den vollverknüpften Schichten Mit Hilfe von Matrizenmultiplikationen werden die Aktivitäten in vollverknüpften Schichten berechnet. Anschließend enthält die Ausgabeschicht das Ergebnis der Klassifikation.

Rückwärtspropagierung in den vollverknüpften Schichten Zunächst wird der Fehler der Ausgabe durch Vergleich mit dem Teacher ermittelt. Anschließend werden die Fehlersignale über sämtliche vollverknüpften Schichten zurückpropagiert, die Gradienten berechnet und deren Gewichte angepasst.



(a) Kleines Konvolutionsnetz (Eingabeschicht der Größe 256×256, 1-8-8-8 Merkmalskarten)

(b) Großes Konvolutionsnetz (Eingabeschicht der Größe 512×512, 4-8-32-64-64 Merkmalskarten)

Abbildung 6.17: Die Größe des Netzes hat kaum einen Einfluss auf die relative Laufzeit der unterschiedlichen Operationen. Das Kopieren der Muster nimmt bei einer größeren Anzahl von Eingabekarten – wie in (b) – zwar mehr Zeit in Anspruch, dies wird aber durch längere Berechnungszeiten relativiert.

Kopieren der Fehlersignale Das Fehlersignal der ersten vollverknüpften Schicht wird in die letzte Konvolutionsschicht kopiert.

Rückwärtspropagierung in den Konvolutionsschichten Der Backprop-Kernel berechnet die Gradienten für sämtliche Filter und das Fehlersignal aller Merkmalskarten der Konvolutionsschichten. Ein weiterer Kernel führt die Aktualisierung der Gewichte durch. Für die Eingabeschicht entfällt die Berechnung der Fehlersignale.

Abbildung 6.17 schlüsselt die anteiligen Laufzeiten dieser Operationen für zwei repräsentative, verschieden große Netze auf. Das Kopieren der Fehlersignale und Aktivitäten zwischen der ersten Schicht des vollverknüpften Netzes (MLP) und der letzten Schicht des Konvolutionsnetzes (KN) sind als ein Posten zusammengefasst.

Erwartungsgemäß beansprucht die Rückwärtspropagierung des Fehlersignals (inklusive der Gewichtsanpassungen) in den Konvolutionsschichten fast zwei Drittel der Laufzeit. Da für die Vorwärtspropagierung deutlich weniger arithmetische Operationen notwendig sind, können diese Berechnungen etwa doppelt so schnell erfolgen. Bei einem kleineren Netz fallen zwar die Berechnungen der vollverknüpften Schichten etwas stärker ins Gewicht als bei einem großen Netz, was sich jedoch kaum auf die Gesamtlaufzeit auswirkt.

Die Aktivitäten der Eingabekarten können mit einer Geschwindigkeit von ca. 4,5 GB/s vom Hauptspeicher der CPU in den Speicher der Grafikkarte kopiert werden. Auch wenn wie bei den hier durchgeführten Messungen sämtliche Muster jede Epoche ausgetauscht werden, machen diese Speichertransfers nur einen Bruchteil der Laufzeit aus.

Vergleich zur CPU-Implementierung

In der CPU-Implementierung fallen keine laufzeitintensiven Speichertransfers an und Speicherzugriffe können durch hierarchisches Caching beschleunigt werden. Da die Geschwindigkeit von parallelen Implementierungen häufig durch die Speicherbandbreite begrenzt ist, müssen diese Operationen bei einem realistischen Vergleich von CPU- und GPU-Implementierung berücksichtigt werden.

In Tabelle 6.5 sind für unterschiedlich große Netze die Laufzeiten für eine vollständige Trainingsepoche angegeben. Die GPU-Geschwindigkeiten berücksichtigen dabei sämtliche

Eingabegröße	Merkmalskarten	Neuronen	CPU [ms]	GPU [ms]	Faktor
256×256	1-8-8-8	100-10	14045,19	140,67	99,8 ×
256×256	2-8-16-32	100-10	44241,95	382,51	115,7 ×
256×256	4-16-64-64	100-10	278009,54	2583,1	107,6 ×
512×512	1-8-8-8-8	100-10	53495,24	560,76	95,4 ×
512×512	2-8-16-32-64	100-10	225109,95	2045,1	110,1 ×
512×512	4-8-32-64-64	100-10	545803,28	5142,95	106,1 ×

Tabelle 6.5: Laufzeiten einer vollständigen Lernepoche für 60 Trainingsmuster inklusive aller Speichertransfers. Die Größe des Netzes hat nur geringen Einfluss auf den Beschleunigungsfaktor.

oben genannten Speichertransfers. Außerdem werden die Ergebnisse der Klassifizierung vom globalen Speicher der Grafikkarte in den Hauptspeicher der CPU kopiert.

Bei den untersuchten Netzen handelt es sich um einen repräsentativen Querschnitt über alle realistisch einsetzbaren Netzgrößen. Anhand der Daten zeigt sich, dass die absolute Laufzeit sowohl von der Anzahl der Merkmalskarten als auch von deren Größe abhängt. Auf den Beschleunigungsfaktor haben hingegen weder die Anzahl noch die Größe der Merkmalskarten einen signifikanten Einfluss. In sämtlichen Fällen liegt der Beschleunigungsfaktor zwischen 95 und 115.

Durch diese erhebliche Beschleunigung ist es möglich, ein großes Konvolutionsnetz innerhalb von ca. 24 Stunden auf Grafikkarten zu trainieren, wofür eine äquivalente CPU-Implementierung mehrere Monate benötigen würde.

7 Zusammenfassung

Systeme zur visuellen Objekterkennung beschränken sich bislang entweder auf Spezialfälle wie etwa die Gesichtserkennung oder sie können nur kleine Eingabebilder in praktikabler Laufzeit verarbeiten. Trotz vielversprechender Ergebnisse mit neuronalen Konvolutionsnetzen auf niedrigdimensionalen Datensätzen wie der MNIST-Datenbank wurden diese Netze insbesondere aufgrund von Rechenzeitbeschränkungen bislang nicht zur Objekterkennung in größeren Bildern eingesetzt.

Auf Basis dieser Netzarchitekturen wurde in Kapitel 4 ein neuronales Konvolutionsnetz zur Erkennung von Objekten in natürlichen Bildern entworfen. Da auch die Rechenleistung moderner CPUs nicht ausreicht, um ein solches Konvolutionsnetz in wenigen Tagen zu trainieren, wurde es mit der CUDA-Schnittstelle auf Grafikkartenhardware implementiert. Die enorme Rechenleistung der parallelen Multiprozessoren dieser GPUs kann jedoch nur optimal genutzt werden, wenn die Algorithmen auf diese Hardwarearchitektur übertragen werden und deren restriktiver interner Speicher effizient genutzt wird.

7.1 Diskussion der Ergebnisse

Beschleunigung

In dieser Diplomarbeit konnte nachgewiesen werden, dass sich das Training neuronaler Konvolutionsnetze mit parallelen GPU-Multiprozessoren erheblich beschleunigen lässt. In Kapitel 5 wurde eine Möglichkeit beschrieben, die Berechnung von Konvolutionsfiltern trotz des begrenzten Shared Memorys feinkörnig zu parallelisieren. Wie die Laufzeitanalysen in Kapitel 6 gezeigt haben, hängt die Beschleunigung jedoch stark davon ab, wie die Trainingsmuster auf die Multiprozessoren der Grafikkarte verteilt werden können. Voraussetzung für eine optimale Beschleunigung gegenüber einer seriellen Implementierung ist, dass die Anzahl der parallel zu berechnenden Trainingsmuster ein Vielfaches der Multiprozessoren der Grafikkarte beträgt. Zwar ist dies eine Einschränkung, für die meisten Klassifikationsaufgaben kann diese Wahl aber so flexibel gehandhabt werden, dass sie kein Problem darstellt.

Unter dieser Vorraussetzung kann das Trainingsverfahren je nach Größe und Struktur des Konvolutionsnetzes um den Faktor 95 bis 115 gegenüber einer auf einem Prozessorkern ausgeführten seriellen CPU-Version beschleunigt werden. Somit hat diese Arbeit gezeigt, dass sich die inhärente Parallelität neuronaler Konvolutionsnetze wie erhofft sehr gut auf paralleler Hardware zur Beschleunigung ausnutzen lässt.

Erkennungsleistung

Sowohl das Lernverfahren als auch die Topologie des hier eingesetzten Konvolutionsnetzes weisen jedoch noch Defizite auf, die sich darin zeigen, dass die Erkennungsleistung auf mehreren Testproblemen teilweise deutlich hinter den Ergebnissen anderer Implementierungen zurückliegt. Auf der Testmenge der MNIST-Datenbank handschriftlicher Ziffern konnte eine gegenüber vergleichbaren Verfahren nur unwesentlich schlechtere Fehlerquote von 1,38% erzielt werden. Bei der Klassifizierung von Objekten des NORB-Datensatzes können mit

dem parallel implementierten Konvolutionsnetz nur 11,07% der Testmuster nicht korrekt erkannt werden. Anhand der Erkennungsleistung auf dem Datensatz der PASCAL Challenge hat sich deutlich gezeigt, dass das hier implementierte Konvolutionsnetz weit hinter den State-of-the-Art-Verfahren zurückliegt.

7.2 Identifikation von Problemquellen

Die auf schwierigen Datensätzen geringe Erkennungsleistung des hier verwendeten Konvolutionsnetzes kann maßgeblich auf zwei Gründe zurückgeführt werden: Zum einen kann rückblickend gesagt werden, dass die konkrete Topologie dieses Konvolutionsnetzes für Klassifikationsaufgaben suboptimal ist, zum anderen ist mit Rprop noch nicht das ideale Lernverfahren für tiefe Konvolutionsnetze gefunden.

Translationsinvarianz

Dass die von Konvolutionsnetzen durchgeführten Faltungen translationsinvariant sind, wird häufig als einer der großen Vorteile dieser Netzarchitektur genannt. Die für diese Arbeit gewählte Netzstruktur nutzt diese Eigenschaft jedoch kaum, da die letzte Konvolutionsschicht implementierungsbedingt mindestens 32×32 Pixel groß sein muss. Die unmittelbar darauf folgende vollverknüpfte Schicht ist daher nicht translationsinvariant und muss somit explizit für jede Position dieser dimensionsreduzierten Merkmalskarte lernen, ein Merkmal zu erkennen.

Eine bessere Abstrahierung der zweidimensionalen Daten hätte möglicherweise mit weiteren Konvolutionsschichten erreicht werden können, die wie in den Entwürfen von Osadchy *et al.* [OLM07] und Simard *et al.* [SSP03] die räumliche Auflösung auf 1×1 Pixel große Merkmalskarten reduzieren.

Da hierzu weitere Kernelfunktionen notwendig sind, konnte dies im Rahmen der Diplomarbeit nicht untersucht werden. Die vorgestellten Parallelisierungsalgorithmen lassen sich jedoch in ähnlicher Weise auf kleinere Merkmalskarten übertragen, so dass anzunehmen ist, dass sich auch deren Berechnung ähnlich gut beschleunigen lässt.

Parameter des Lernverfahrens

Eine weitere Ursache für die beim Training des Netzes aufgetretenen Probleme ist das Lernverfahren Rprop in Kombination mit der austauschbaren Arbeitsmenge (siehe Abschnitt 4.3.4).

Da Rprop zum Konvergieren einen stabilen Gradienten benötigt, war es nicht möglich, Online-Training mit Mini-Batches durchzuführen. Die Alternative, nämlich das Trainingsverfahren auf einer austauschbaren Arbeitsmenge durchzuführen, konvergierte jedoch ebenso wie echtes Offline-Training nur schlecht.

Anstatt wie erhofft mit Rprop die Wahl der Lernraten zu erleichtern, musste das Lernverfahren in beiden Fällen erweitert werden. So war z.B. eine Regularisierung der Gewichte mit Weight Decay erforderlich, wodurch weitere zu optimierende Parameter eingeführt werden mussten.

Obwohl individuelle Lernraten grade bei gekoppelten Gewichten in tiefen Netzen sinnvoll sind, hat sich im Laufe der Diplomarbeit gezeigt, dass Rprop zum Training eines Konvolutionsnetzes mit Mini-Batches problematisch sein kann. Als Alternative könnte eine Gewichtsanpassung mit dem Levenberg-Marquardt-Algorithmus [LBBH98] untersucht werden. Dieses Lernverfahren ist auch für Mini-Batch-Traning geeignet, da es in regelmäßigen Abständen

die Lernraten für jedes Gewicht durch Approximierung der zweiten Ableitung der Fehlerfunktion anpasst.

7.3 Offene Fragen und Ausblick

Mit den in dieser Diplomarbeit entwickelten und implementierten parallelen Algorithmen konnte das Lernverfahren eines Konvolutionsnetzes gegenüber einer seriellen Implementierung signifikant beschleunigt werden. Diese Ergebnisse lassen sich auch auf andere Konvolutionsnetze und auf ähnliche parallele Hardwarearchitekturen übertragen.

Forschungsbedarf besteht bezüglich der Frage, welches die optimale Topologie eines neuronalen Konvolutionsnetzen zum Erlernen von Klassifikatoren zur Objekterkennung in natürlichen Bildern ist. Da echtes Online-Training aufgrund der Parallelisierung über die Trainingsmuster nicht möglich ist, muss zudem ein Lernverfahren mit adaptiven Lernraten gefunden werden, welches auch auf Mini-Batches gut konvergiert.

Die in dieser Diplomarbeit gewonnenen Erkenntnisse bezüglich Netzarchitektur und Lernverfahren können als Ausgangspunkt für die im Folgenden skizzierten möglichen Erweiterungen dienen.

Spärliche Verknüpfungen Wie die Versuche mit dominanten Verknüpfungen (siehe Abschnitt 6.1.2) gezeigt haben, konvergiert das Lernverfahren deutlich schneller, wenn nicht alle Merkmalskarten zweier benachbarter Schichten miteinander verknüpft sind. Mit einer spärlichen Verknüpfung der Merkmalskarten würde aber nicht nur die Konvergenz verbessert, sondern es könnte auch erheblich an Laufzeit eingespart werden.

Ohne die Beschleunigung des Netzes zu verringern, wäre eine entsprechende Modifikation der vorliegenden Implementierung möglich.

Gewichtsinitialisierung Tiefe neuronale Netze sind deutlich schwieriger zu trainieren, als flache Architekturen mit ein oder zwei verdeckten Schichten. Bei den Versuchen mit unterschiedlichen Gewichtsinitialisierungen in Abschnitt 6.1.2 hat sich gezeigt, dass eine geeignete Wahl der Anfangswerte die Erkennungsraten erheblich verbessern kann.

Eine relativ neue Möglichkeit gute Anfangsgewichte für den Gradientenabstieg zu finden, ist das unüberwachte schichtweise Vortrainieren mit sogenannten Autoencodern [HOT06, VLBM08]. Dabei werden die Schichten des Netzes sukzessive darauf trainiert, das künstlich generierte Rauschen aus einem Eingabebild zu entfernen. Versuche haben gezeigt, dass die damit gewonnenen Informationen über die strukturellen Eigenschaften eines Datensatzes die Ergebnisse des darauffolgenden überwachten Lernens deutlich verbessern können.

Gewichtsregularisierung Die Regularisierung der Gewichte kann insbesondere bei Datensätzen mit wenig Trainingsmustern zu einer besseren Generalisierungsleistung führen. Die in dieser Arbeit durchgeführte Regularisierung jedes einzelnen Gewichts mit Weight Decay ist jedoch in den Konvolutionsschichten nicht ideal.

Konsequenter wäre es, die Gewichte der Konvolutionsschicht abhängig von der Summe aller Elemente eines Filters zu regularisieren. Eine mögliche Regularisierung für Konvolutionsnetze, die zudem Vorwissen über das zu lernende Problem ausnutzt, wurde von Yu *et al.* [YXG08] vorgeschlagen, die damit über gute Ergebnisse auch auf kleinen Datensätzen berichten.

Erweiterung zur Objektlokalisierung Mit verschiedenen Erweiterungen könnte das implementierte Konvolutionsnetz auch zur Objektlokalisierung angewandt werden. Dazu könnten z.B. sogenannte *Space Displacement Neural Networks* [LBBH98] eingesetzt werden, die die Translationsinvarianz der Konvolutionsfilter ausnutzen und daher gegenüber den Trainingsmustern auf deutlich größeren Bildern einzusetzen wären. Zur Gesichtserkennung [GD04, OLM07] wurden solche Netze mehrfach erfolgreich angewandt.

Möglich wäre es außerdem, wie in der neuronalen Abstraktionspyramide [Beh05], rekurrente Verbindungen von höheren zu tieferen Schichten zuzulassen. Damit könnte die Ausgabe auf den tieferen, bildartigen Schichten erfolgen. Zu überdenken wäre hierbei, ob der begrenzte Grafikkartenspeicher für Lernverfahren wie Backpropagation-through-Time ausreichend ist.

Mit den hier vorgestellten Lösungsansätzen zeichnen sich mögliche Wege zur Beantwortung der noch offenen Fragen ab. Ob mit diesen Konzepten ein lernendes System zur Objekterkennung in natürlichen Umgebungen erstellt werden kann, ist im Zuge weiterer Arbeiten zu prüfen.

Literaturverzeichnis

[Beh01] Sven Behnke. Learning iterative image reconstruction in the Neural Abstraction Pyramid. *International Journal of Computational Intelligence and Applications*, 1:427–438, 2001.

[Beh03] Sven Behnke. *Hierarchical Neural Networks for Image Interpretation*, volume 2766 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, June 2003.

[Beh05] Sven Behnke. Face Localization and Tracking in the Neural Abstraction Pyramid. *Neural Computing and Applications*, 14(2):97–103, July 2005.

[BG03] Matthew Browne and Saeed Shiry Ghidary. Convolutional neural networks for image processing: an application in robot vision. In *Australian Joint Conference on Artificial Intelligence*, pages 641–652, December 2003.

[Bil06] Stanley M. Bileschi. *StreetScenes: Towards Scene Understanding in Still Images*. PhD thesis, Massachusetts Institute of Technology, May 2006.

[CHH02] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57 – 83, 2002.

[CHH07] Deng Cai, Xiaofei He, and Jiawei Han. Efficient Kernel Discriminant Analysis via Spectral Regression. In *ICDM*, pages 427–432. IEEE Computer Society, 2007.

[Chi08] Sharat Chikkerur. CUDA implementation of a biologically inspired object recognition system, 2008.

[CPS06] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule, France, October 2006. Université de Rennes, Suvisoft.

[DHH+09] Santosh K. Divvala, Derek Hoiem, James H. Hays, Alexei A. Efros, and Martial Hebert. An Empirical Study of Context in Object Detection. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2009.

[DT05] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. In *CVPR*, pages 886–893, 2005.

[Duf07] Stefan Duffner. *Face Image Analysis With Convolutional Neural Networks*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2007.

[EVGW+] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2008 (VOC2008) Results. `http://www.pascal-network.org/challenges/VOC/voc2008/workshop/index.html`. Zuletzt besucht am 30.03.2009.

[EZW+06] Mark Everingham, Andrew Zisserman, Christopher K. I. Williams, Luc van Gool, Moray Allan, Christopher M. Bishop, Olivier Chapelle, Navneet Dalal, Thomas Deselaers, Gyuri Dorko, Stefan Duffner, Jan Eichhorn, Jason D. R. Farquhar, Mario Fritz, Christophe Garcia, Tom Griffiths, Frederic Jurie, Daniel Keysers, Markus Koskela, Jorma Laaksonen, Diane Larlus, Bastian Leibe, Hongying Meng, Hermann Ney, Bernt Schiele, Cordelia Schmid, Edgar Seemann, John Shawe-Taylor, Amos Storkey, Sandor Szedmak, Bill Triggs, Ilkay Ulusoy, Ville Viitaniemi, and Jianguo Zhang. The 2005 PASCAL Visual Object Classes Challenge. In *Machine Learning Challenges. Evaluating Predictive Uncertainty, Visual Object Classification, and Recognising Tectual Entailment (PASCAL Workshop 05)*, number 3944 in Lecture Notes in Artificial Intelligence, pages 117–176, Southampton, UK, 2006.

[Fah88] Scott E. Fahlman. An Empirical Study of Learning Speed in Backpropagation Networks. Technical Report CMU-CS-88-162, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1988.

[FGMR08] Pedro F. Felzenszwalb, Ross Girshick, David A. McAllester, and Deva Ramanan. Discriminatively Trained Mixtures of Deformable Part Models, 2008.

[FMR08] Pedro F. Felzenszwalb, David A. McAllester, and Deva Ramanan. A discriminatively trained, multiscale, deformable part model. In *CVPR*. IEEE Computer Society, 2008.

[FS95] Yoav Freund and Robert E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. In *EuroCOLT '95: Proceedings of the Second European Conference on Computational Learning Theory*, pages 23–37, London, UK, 1995. Springer-Verlag.

[Fuk86] Kunihiko Fukushima. A neural network model for selective attention in visual pattern recognition. *Biological Cybernetics*, 55(1):5–15, October 1986.

[Gab46] Dennis Gabor. Theory of communications. *Journal of the Institution of Electrical Engineers*, 93:429–457, 1946.

[GD04] Christophe Garcia and Manolis Delakis. Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(11):1408–1423, 2004.

[GHP07] G. Griffin, A. Holub, and P. Perona. Caltech-256 Object Category Dataset. Technical Report 7694, California Institute of Technology, 2007.

[GvdBSG01] J.-M. Geusebroek, R. van den Boomgaard, A.W.M. Smeulders, and H. Geerts. Color invariance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(12):1338–1350, December 2001.

[HE08] J. Hays and A.A. Efros. IM2GPS: Estimating geographic information from a single image. In *IEEE Conference on Computer Vision and Pattern Recognition, 2008*, pages 1–8, 2008.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, September 2006.

[HSJG08] Hedi Harzallah, Cordelia Schmid, Frederic Jurie, and Adrien Gaidon. Classification aided two stage localization, October 2008. PASCAL Visual Object Classes Challenge Workshop, in conjunction with ECCV.

[Jac87] Robert A. Jacobs. Increased Rates of Convergence Through Learning Rate Adaptation. Technical report, University of Massachusetts, Amherst, MA, USA, 1987.

[JPJ08] H. Jang, A. Park, and K. Jung. Neural Network Implementation Using CUDA and OpenMP. *Computing: Techniques and Applications, 2008. DICTA'08. Digital Image*, pages 155–161, 2008.

[KH92] Anders Krogh and John A. Hertz. A Simple Weight Decay Can Improve Generalization. In *Advances in Neural Information Processing Systems*, volume 4, pages 950–957. Morgan Kaufmann, 1992.

[Koh90] Teuvo Kohonen. The Self-organizing Map. *Proceedings of the IEEE*, 78(9):1464–1480, September 1990.

[LAN08] Sheetal Lahabar, Pinky Agrawal, and P. J. Narayanan. High Performance Pattern Recognition on GPU. In *Proceedings of National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics*, pages 154–159, January 2008.

[LB95] Y. LeCun and Y. Bengio. Convolutional Networks for Images, Speech, and Time-Series. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press, 1995.

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[LBD+90] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In David Touretzky, editor, *Advances in Neural Information Processing Systems 2 (NIPS*89)*, Denver, CO, 1990. Morgan Kaufman.

[LBLL09] Hugo Larochelle, Yoshua Bengio, Jerome Louradour, and Pascal Lamblin. Exploring Strategies for Training Deep Neural Networks. *The Journal of Machine Learning Research*, pages 1–40, 2009.

[LBOM98] Y. LeCun, L. Bottou, G. Orr, and K. Müller. Efficient BackProp. In *Neural Networks: Tricks of the trade*. Springer, 1998.

[LDS+90] Y. LeCun, J. S. Denker, S. Solla, R. E. Howard, and L. D. Jackel. Optimal Brain Damage. In David Touretzky, editor, *Advances in Neural Information Processing Systems 2 (NIPS*89)*, Denver, CO, 1990. Morgan Kaufman.

[LeC89] Y. LeCun. Generalization and Network Design Strategies. Technical Report CRG-TR-89-4, Department of Computer Science, University of Toronto, 1989.

[LGTB97] Steve Lawrence, C. Lee Giles, Ah Chung Tsoi, and Andrew D. Back. Face Recognition: A Convolutional Neural Network Approach. *IEEE Transactions on Neural Networks*, 8:98–113, 1997.

[LHB04] Y. LeCun, F. Huang, and L. Bottou. Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. In *Proceedings of CVPR'04*. IEEE Press, 2004.

[Log99] N. Logothetis. Vision: A window on consciousness. *Scientific American*, November 1999.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[LSB07] Fabien Lauer, Ching Y. Suen, and Gérard Bloch. A trainable feature extractor for handwritten digit recognition. *Pattern Recognition*, 40(6):1816–1824, 2007.

[LSP06] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. In *CVPR (2)*, pages 2169–2178. IEEE Computer Society, 2006.

[ML06] J. Mutch and D.G. Lowe. Multiclass Object Recognition with Sparse, Localized Features. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 11–18, June 2006.

[MP69] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[MRW+99] Sebastian Mika, Gunnar Rätsch, Jason Weston, Bernhard Schölkopf, and Klaus-Robert Müller. Fisher Discriminant Analysis With Kernels, 1999.

[MSHvdW07] Marcin Marszałek, Cordelia Schmid, Hedi Harzallah, and Joost van de Weijer. Learning Object Representations for Visual Object Class Recognition, October 2007. Visual Recognition Challange workshop, in conjunction with ICCV.

[NP07] Nvidia Corporation and Victor Podlozhnyuk. Image Convolution with CUDA, 2007.

[Nvi08a] Nvidia Corporation. CUDA CUBLAS Library 2.1, September 2008.

[Nvi08b] Nvidia Corporation. CUDA Occupancy Calculator 1.5. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, 2008. Zuletzt besucht am 22.05.2009.

[Nvi08c] Nvidia Corporation. CUDA Programming Guide 2.1, August 2008.

[OLM07] M. Osadchy, Y. LeCun, and M. Miller. Synergistic Face Detection and Pose Estimation with Energy-Based Models. *Journal of Machine Learning Research*, 8:1197–1215, May 2007.

[RB92] Martin Riedmiller and Heinrich Braun. RPROP – A fast adaptive learning algorithm. In *Proceedings of the Int. Symposium on Computer and Information Science VII*, 1992.

[RB94] Martin Riedmiller and Heinrich Braun. RPROP – Description and Implementation Details. Technical report, University of Karlsruhe, January 1994.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*, pages 318–362, 1986.

[Ros58] Frank Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psych. Rev.*, 65:386–407, 1958. (Reprinted in *Neurocomputing* (MIT Press, 1988)).

[Ros62] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, 1962.

[RP99] M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2:1019–1025, 1999.

[RPCL06] Marc'Aurelio Ranzato, Christopher Poultney, Sumit Chopra, and Yann LeCun. Efficient Learning of Sparse Representations with an Energy-Based Model. In J. Platt et al., editor, *Advances in Neural Information Processing Systems (NIPS 2006)*. MIT Press, 2006.

[RT08] Bryan C. Russell and Antonio Torralba. LabelMe: a database and web-based tool for image annotation. *International Journal of Computer Vision*, 77:157–173, 2008.

[SI07] C. Siagian and L. Itti. Rapid biologically-inspired scene classification using features shared with visual attention. *IEEE transactions on pattern analysis and machine intelligence*, 29(2):300, 2007.

[SSH08] John Stratton, Sam Stone, and Wen-mei Hwu. MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[SSP03] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. Best Practice for Convolutional Neural Networks Applied to Visual Document Analysis. In *International Conference on Document Analysis and Recogntion (ICDAR), IEEE Computer Society, Los Alamitos*, pages 958–962, 2003.

[TG98] Nicholas K. Treadgold and Tamas D. Gedeon. Simulated Annealing and Weight Decay in Adaptive Learning: The SARPROP Algorithm. *IEEE-NN*, 9(4):662, July 1998.

[vdSGS08] Koen E. A. van de Sande, Theo Gevers, and Cees G. M. Snoek. Evaluation of Color Descriptors for Object and Scene Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, Anchorage, Alaska, USA, June 2008.

[VLBM08] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 1096–1103. ACM, August 2008.

[Was89] Philip D. Wasserman. *Neural Computing: Theory and Practice*. Van Nostrand Reinhold, New York, 1989.

[WDSS08] C. Wojek, G. Dorkó, A. Schulz, and B. Schiele. Sliding-Windows for Rapid Object Class Localization: A Parallel Technique. *Lecture Notes in Computer Science*, 5096:71–81, 2008.

[Wer74] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, Cambridge, MA, 1974.

[Wer88] P. Werbos. Backpropagation: Past and Future. In *IEEE International Conference on Neural Networks*, volume 1, pages 343–353, July 1988.

[Wis04] Laurenz Wiskott. How Does Our Visual System Achieve Shift and Size Invariance? *Problems in Systems Neuroscience*, 2004.

[WM03] D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429 – 1451, 2003.

[WP90] Ronald J. Williams and Jing Peng. An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories. *Neural Computation*, 2:490–501, 1990.

[YXG08] Kai Yu, Wei Xu, and Yihong Gong. Deep Learning with Kernel Regularization for Visual Recognition. In Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou, editors, *Neural Information Processing Systems Conference*. MIT Press, December 2008.

[Zel94] Andreas Zell. *Simulation Neuronaler Netze*. Oldenbourg, September 1994.

Erklärung

Hiermit erkläre ich gemäß § 19 Abs. 7 der Diplomprüfungsordnung vom 15. August 1998, dass ich meine Diplomarbeit selbständig erstellt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Alfter, den 8. Juni 2009 Dominik Scherer