Parallele Implementierung hierarchischer neuronaler Netze mit lokaler Konnektivität zur Objekterkennung in natürlichen Bildern

Diplomarbeit

Vorgelegt von Rafael Uetz
am 8. Juni 2009

Erstgutachter: Prof. Dr. Sven Behnke
Zweitgutachter: Prof. Dr. Joachim K. Anlauf

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik VI
Autonome Intelligente Systeme

Zusammenfassung

In dieser Arbeit wird ein biologisch motiviertes Objekterkennungssystem entworfen, implementiert und evaluiert. Das System basiert auf einem künstlichen neuronalen Netz mit mehreren, hierarchisch aufgebauten Schichten und lokalen Verknüpfungen. Performanzkritische Funktionen des Systems sind mit Hilfe des CUDA-Frameworks implementiert, so dass sie parallel auf einer CUDA-kompatiblen Grafikkarte ausgeführt werden können. Dadurch wird gegenüber einer Implementierung ohne CUDA ein Geschwindigkeitsgewinn von bis zu Faktor 82 erzielt.

Die Erkennungsleistung des Systems wird mit drei verschiedenen Datensätzen evaluiert. Beim MNIST-Datensatz handgeschriebener Ziffern wird ein Testfehler (das ist der Anteil falsch klassifizierter Muster der Testmenge) von 1,74 % erzielt. Beim NORB-Datensatz, welcher fünf verschiedene Objektklassen in unterschiedlichen Blickwinkeln und Beleuchtungsverhältnissen enthält, wird ein Testfehler von 5,28 % erreicht. Diese Ergebnisse sind ähnlich gut wie aktuelle „State of the Art“-Verfahren im Bereich der Objekterkennung, zum Beispiel Support Vector Machines und Konvolutionsnetze.

Durch die hohe Ausführungsgeschwindigkeit des Systems können erheblich größere Datenmengen verarbeitet werden, als zuvor in der Literatur zu finden waren. Es wird deshalb eine große Trainings- und Testmenge aus dem LabelMe-Datensatz extrahiert. Dieser Datensatz enthält natürliche Bilder mit darin annotierten Objekten und stellt eine große Herausforderung für Objekterkennungssysteme dar. Die erzeugte Menge besteht aus 86.574 Mustern und enthält 12 verschiedene Objektklassen sowie eine Kategorie mit Negativbeispielen. Es werden mehrere Messungen mit der erzeugten Mustermenge vorgestellt, um verschiedene Parameter miteinander zu vergleichen. Zur Erkennung von Objekten in beliebig großen Eingabebildern wird außerdem ein Sliding-Window-Algorithmus entwickelt und evaluiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das Sehen ist der vermutlich wichtigste Sinn des Menschen. Völlig ohne Anstrengung gelingt es uns, selbst komplexe Szenen und Objekte innerhalb kürzester Zeit zu erfassen. Erst bei näherer Betrachtung wird klar, welch faszinierende Leistung unser Sehsystem dabei erbringt. So hat sich die automatische Erkennung von Objekten mit dem Computer – trotz großer Fortschritte und vieler neu gewonnener Erkenntnisse in den letzten Jahren – als enorme Herausforderung erwiesen, die zum aktuellen Zeitpunkt keinesfalls zufriedenstellend gelöst ist.

Systeme zur Objekterkennung werden allerdings in vielen Bereichen immer häufiger benötigt. Beispiele dafür sind die autonome Robotik, kameragestützte Sicherheitssysteme, Fahrerassistenzsysteme sowie automatische Sortierung von Fotos nach darauf abgebildeten Personen. Zum aktuellen Zeitpunkt kommen dabei häufig Speziallösungen zum Einsatz, die nur unter bestimmten Voraussetzungen an Beleuchtungsverhältnisse, Blickwinkel oder Existenz von Verdeckungen funktionieren und nicht auf andere Situationen oder Objektklassen übertragbar sind. Ein bekanntes Beispiel dafür ist der Algorithmus von Viola und Jones [65], der zuverlässig und schnell Gesichter in Bildern entdeckt. Er ist allerdings nur bedingt auf andere Objektklassen übertragbar und funktioniert nicht mehr richtig, wenn beispielsweise die Stirn durch ein anderes Objekt verdeckt ist.

Aufgrund dieser Einschränkungen stieg in den letzten Jahren das Interesse an sogenannter *allgemeiner Objekterkennung*. Ziel hierbei ist es, beliebige Objekte unter realistischen Bedingungen korrekt zu erkennen. Die verschiedenen Objektklassen werden dabei aus Beispielen gelernt, statt manuelles Vorwissen über einzelne Klassen einfließen zu lassen. Inspiriert durch das menschliche Sehsystem entstanden verschiedene biologisch motivierte Modelle, die derzeit zu den „State of the Art"-Verfahren im Bereich der allgemeinen Objekterkennung gehören.

1.1 Ziele der Arbeit

Das Ziel der vorliegenden Arbeit ist der Entwurf und die Implementierung eines Systems zur allgemeinen Objekterkennung, mit dem beliebige Objekte (zum Beispiel Autos und Personen) zuverlässig in *natürlichen Bildern* (das sind Fotografien von real existierenden Szenen) gefunden werden können.

Weiterhin soll das System in der Lage sein, sowohl in der Trainings- als auch in der Anwendungsphase sehr große Datenmengen mit hoher Geschwindigkeit zu verarbeiten, so dass nicht nur „Spielzeugprobleme", sondern große Bilddatensätze zur Evaluation herangezogen werden können. Besonderer Wert wird dabei auf die Auswahl der Trainings- und Testmengen gelegt. Wie in [50] beschrieben, lassen unrealistische oder zu einfache Bilddatensätze keine sinnvolle Bewertung allgemeiner Objekterkennungsverfahren zu. Aus diesem Grund wird der LabelMe-Datensatz [43] verwendet, der eine Vielzahl von Objekten in natürlicher Umgebung und unter realistischen Bedingungen (Lichtverhältnisse, Blickwinkel, Verdeckungen et cetera) enthält.

Um die genannten Ziele zu erreichen, wird ein biologisch motiviertes Modell zur Objekterkennung verwendet, genauer gesagt ein hierarchisch aufgebautes künstliches neuronales Netz. Dabei wird versucht, eine gute Abwägung zwischen biologischer Plausibilität und hoher Ausführungsgeschwindigkeit zu finden. So wird zum Beispiel im Gegensatz zu einigen anderen

Abbildung 1.1: Objekterkennung in natürlichen Bildern. Die erkannten Objektklassen werden durch verschiedene Farben markiert. Bildquelle: [42]

Modellen auf die Anwendung berechnungsintensiver Gabor-Filter [44] verzichtet. Des Weiteren verfügt das neuronale Netz im Gegensatz zu mehrschichtigen Perzeptrons [56; 67] nicht über eine vollständige, sondern über eine lokale Vernetzung zwischen aufeinanderfolgenden Schichten.

Die Implementierung erfolgt mit Hilfe des *CUDA-Frameworks* (Compute Unified Device Architecture) von NVIDIA, mit dem Teile des Programms parallel auf der Grafikkarte ausgeführt werden können. Aktuelle Grafikkarten bieten im Vergleich zu aktuellen CPUs (Central Processing Units) eine drastisch höhere Rechenleistung sowie eine höhere Speicherbandbreite, wenn sie effizient genutzt werden. Das vorgeschlagene Modell eignet sich hervorragend für die CUDA-Implementierung, weil es sich – wie das biologische Vorbild – durch feinkörnige Parallelität und lokale Verbindungen auszeichnet.

1.2 Inhalt und Struktur

Kapitel 2 vermittelt zunächst alle für diese Arbeit benötigten Grundlagen. Es wird nacheinander auf das biologische Sehsystem, künstliche neuronale Netze, Datensätze natürlicher Bilder und das CUDA-Framework eingegangen. Anschließend erfolgt in Kapitel 3 eine Vorstellung und Diskussion von Arbeiten, die sich mit verwandten Problemstellungen beschäftigen. Danach wird in Kapitel 4 das im weiteren Verlauf verwendete Modell zur Objekterkennung vorgestellt und Designentscheidungen werden begründet. Kapitel 5 beschreibt die Implementierung des Modells und geht dabei insbesondere auf die sogenannten Kernel-Funktionen ein, die auf der Grafikkarte ausgeführt werden und eine wesentlich höhere Verarbeitungsgeschwindigkeit ermöglichen. In Kapitel 6 wird das implementierte System dann anhand von verschiedenen Messungen evaluiert. Als erste Trainings- und Testmenge dient der MNIST-Datensatz handgeschriebener Ziffern [37], da für ihn eine Vielzahl von Vergleichswerten aus anderen Arbeiten existiert. Anschließend werden Messungen mit Hilfe des NORB- und des LabelMe-Datensatzes durchgeführt, um die Tauglichkeit des Systems für allgemeine Objekterkennungsaufgaben zu überprüfen. Kapitel 7 enthält abschließend eine Zusammenfassung und Bewertung der Ergebnisse und einen Ausblick auf mögliche nachfolgende Arbeiten.

2 Grundlagen

In diesem Kapitel werden die für den Rest der Arbeit benötigten Grundlagen vermittelt und ein Überblick über den aktuellen Stand der Forschung gegeben.

2.1 Biologische Grundlagen der visuellen Wahrnehmung

Das Sehen ist – bezogen auf die geleistete Informationsverarbeitung – der komplexeste Sinn des Menschen [30]. Um diesen Sinn besser zu verstehen, muss man zunächst die Informationsverarbeitung auf einer sehr viel tieferen Ebene betrachten. Abschnitt 2.1.1 erklärt deshalb die grundlegende Funktionsweise von informationsverarbeitenden Nervenzellen – den Neuronen. Außerdem wird das abstrahierte Modell eingeführt, das für die später behandelten künstlichen neuronalen Netze genutzt wird. In Abschnitt 2.1.2 wird dann auf das Sehsystem des Menschen eingegangen.

2.1.1 Neuronen und neuronale Netze

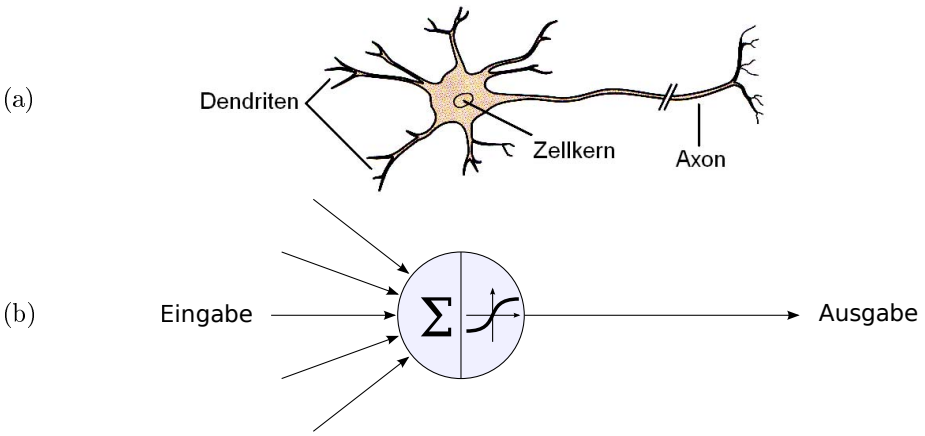


Abbildung 2.1: (a) Vereinfachte Darstellung eines biologischen Neurons. Quelle: [30], bearbeitet.
(b) Abstrahiertes Neuronenmodell

Alle Lebewesen sind aus *Zellen* aufgebaut. Die Zelle ist die unterste Strukturebene, die sämtliche Eigenschaften des Lebens besitzt [6]. Es gibt viele verschiedene Arten von Zellen, die auf unterschiedliche Aufgaben spezialisiert sind. Eine dieser Arten ist die in Abbildung 2.1 (a) dargestellte *Nervenzelle*, die auch *Neuron* genannt wird. Sie ist auf die Weiterleitung und Verarbeitung von Informationen in Form von elektrischen Signalen spezialisiert [22]. Dazu besitzt sie Ausläufer, die *Dendriten* genannt werden. Diese sind an speziellen Kontaktstellen, den *Synapsen*, mit anderen Nerven- oder Sinneszellen verbunden und reagieren auf elektrische Signale von diesen. Wenn ein gewisser Schwellwert von ankommenden elektrischen Signalen überschritten wird, löst die Nervenzelle ein sogenanntes *Aktionspotenzial* aus. Dabei wird ein elektrischer Impuls aktiv durch das *Axon* weitergeleitet. Die Ausläufer des Axons (*Endbäumchen*) docken dann wieder an andere Nervenzellen an, die ihrerseits auf

das Signal reagieren. Die auf diese Weise entstehende Vernetzung von Nervenzellen nennt man *neuronales Netz*.

Abbildung 2.1 (b) zeigt ein abstrahiertes Modell der Funktionsweise eines Neurons. Das Neuron bekommt eine oder mehrere Eingaben, die zu einer sogenannten *Netzeingabe* aufkumuliert werden. Dabei können verschiedene Eingänge unterschiedlich gewichtet sein. Das Aktionspotenzial ergibt sich dann als nichtlineare Funktion der Netzeingabe und bildet die Ausgabe des Neurons.

Dieses stark vereinfachte Modell bildet die Grundlage für viele Modelle künstlicher neuronaler Netze. Unterschiede dieser Modelle bestehen unter anderem darin, wie die Eingaben kumuliert werden und wie die zeitliche Komponente behandelt wird. In Abschnitt 2.2 werden einige Modelle künstlicher neuronaler Netze vorgestellt.

Neuronale Netze arbeiten massiv parallel und können bereits mit wenigen Neuronen ein sehr komplexes Verhalten aufweisen. Das menschliche Gehirn besitzt rund 10^{11} Neuronen [67] und ist damit zu enormen Leistungen wie zum Beispiel der visuellen Wahrnehmung fähig, die im nächsten Abschnitt behandelt wird.

2.1.2 Das menschliche Sehsystem

Das menschliche Gehirn ist in viele verschiedene Funktionsbereiche aufgeteilt [30]. Diese Aufteilung ist allerdings nicht disjunkt und kann sich auch über die Zeit verändern. Bei einer Schädigung von bestimmten Teilen des Gehirns (zum Beispiel nach einem Unfall) können andere Teile die verlorenen Bereiche partiell ersetzen. Weiterhin bestehen zahlreiche Verbindungen zwischen verschiedenen Bereichen des Gehirns, allerdings nicht so viele wie innerhalb eines Funktionsbereichs.

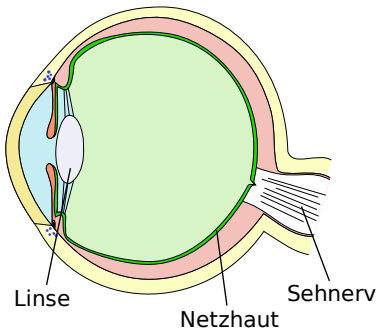


Abbildung 2.2: Aufbau des Auges. Quelle: Wikimedia Commons (http://de.wikipedia.org/w/index.php?title=Datei:Eye_scheme.svg), bearbeitet

Informationsverarbeitung im Auge

Die visuelle Informationsverarbeitung beginnt bereits im Auge. Das durch die *Linse* gebündelte Licht fällt auf die *Netzhaut* (auch *Retina* genannt), siehe Abbildung 2.2. Dort befinden sich verschiedene Arten von lichtempfindlichen Zellen, die auf einfallendes Licht mit elektrischen Signalen reagieren und *Photorezeptoren* genannt werden. Es gibt zwei Gruppen von Photorezeptoren: *Stäbchen* können nur Helligkeit unterscheiden, während *Zapfen* für das Farbsehen verantwortlich sind. Zapfen unterteilen sich wiederum in drei Arten, die für verschiedene Wellenlängenbereiche empfindlich sind; diese entsprechen den Farben Rot, Grün und Blau. Die Signale der Photorezeptoren werden durch eine Zwischenschicht an sogenannte *Ganglienzellen* weitergegeben. Jede Ganglienzelle wird von mehreren Photorezeptoren be-

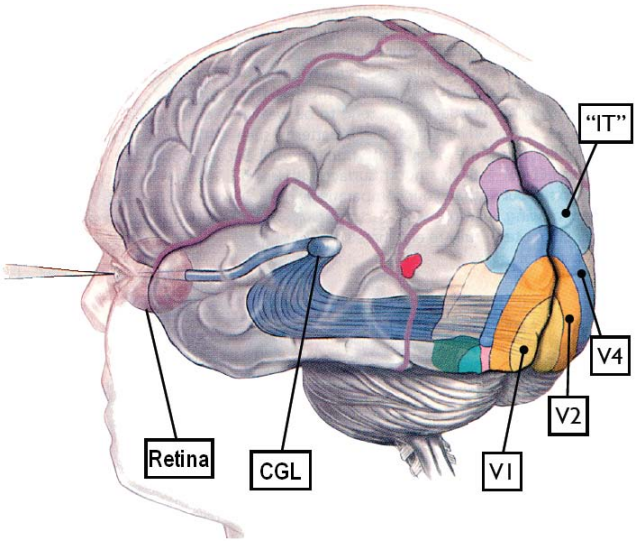


Abbildung 2.3: Hauptbereiche der Verarbeitung visueller Informationen im Gehirn. Quelle: [40], bearbeitet

einflusst, die räumlich dicht beieinander liegen. Dieser Bereich wird *rezeptives Feld* der Zelle genannt und hat in etwa eine runde Form.

Ganglienzellen sind Kontrastdetektoren. Der innere Bereich des rezeptiven Feldes wirkt sich entgegengesetzt zum äußeren Bereich auf das Aktionspotenzial der Zelle aus, so dass die größte Erregung auftritt, wenn ein starker Kontrast zwischen dem inneren und dem äußeren Bereich des rezeptiven Feldes herrscht. Man spricht hierbei auch von *differenzieller Codierung*. Es werden zwei Hauptarten unterschieden: Bei *On-Center-Ganglienzellen* wirkt das Zentrum erregend und der äußere Bereich hemmend auf das Aktionspotenzial, bei *Off-Center-Ganglienzellen* ist es umgekehrt. Beide Typen sind in ähnlicher Anzahl vorhanden. Abgesehen von dieser Unterscheidung gibt es verschiedene Unterarten von Ganglienzellen, wobei in jedem Bereich der Netzhaut alle Arten vorhanden sind. Insbesondere sind hier *M-Zellen* und *P-Zellen* zu unterscheiden. Die M-Zellen reagieren bereits auf sehr feine Helligkeitsunterschiede, sind aber nicht in der Lage, Farbunterschiede zu erkennen. P-Zellen verhalten sich genau umgekehrt. Außerdem weisen P-Zellen kleinere rezeptive Felder, also eine höhere räumliche Auflösung auf. Im Zentrum ihres rezeptiven Feldes befinden sich andere Zapfen als im äußeren Bereich. Somit wird eigentlich nicht auf Farben, sondern auf Farbdifferenzen – genauer gesagt auf Rot-Grün- und Blau-Gelb-Differenzen – reagiert. Die gelbe Farbwahrnehmung ergibt sich dabei aus rot- und grünempfindlichen Zapfen zusammen.

Die etwa eine Million Axone aller Ganglienzellen bilden zusammen den *Sehnerv*, der das Auge mit dem Gehirn verbindet.

Der Corpus Geniculatum Laterale

Von den Axonen des Sehnervs führen etwa neunzig Prozent zum *Corpus Geniculatum Laterale* (*CGL*) im Zwischenhirn, siehe Abbildung 2.3. Die restlichen Axone führen beispielsweise in Bereiche zur Steuerung des Pupillenmuskels und der Augenbewegung, sind aber für die visuelle Wahrnehmung fast nicht von Bedeutung.

Die Verbindungen der Ganglienzellen der Retina zum CGL sind *retinotop*. Das bedeutet, dass die räumliche Struktur (Topologie) der Netzhaut auf dem CGL erhalten bleibt. Benachbarte Gebiete der Retina werden also auf benachbarte Gebiete des CGL abgebildet.

Der CGL besteht aus sechs Neuronenschichten. Jeweils drei Schichten erhalten ihre Eingaben ausschließlich vom linken beziehungsweise rechten Auge. Jedes Neuron erhält seine Eingaben außerdem nur von wenigen Ganglienzellen. Wie Hubel und Wiesel in [25] zeigen konnten, besitzen die Neuronen des CGL ein ähnliches rezeptives Feld wie die Neuronen der Retina; die differenzielle Codierung bleibt erhalten. Auch die Trennung von M- und P-Zellen bleibt erhalten und führt in zwei parallelen Nervenbahnen, den *M-* und *P-Bahnen*, weiter zum *primären visuellen Cortex*, der den nächsten Bereich der visuellen Informationsverarbeitung darstellt. Die genaue Funktion des CGL ist noch nicht geklärt, da er neben den Eingaben des Sehnervs auch viele Eingaben aus anderen Regionen des Gehirns bekommt, darunter viele aus dem visuellen Cortex. Es wird vermutet, dass so der Fluss von Informationen zum visuellen Cortex gesteuert wird [30].

Der primäre visuelle Cortex

(a) (b)

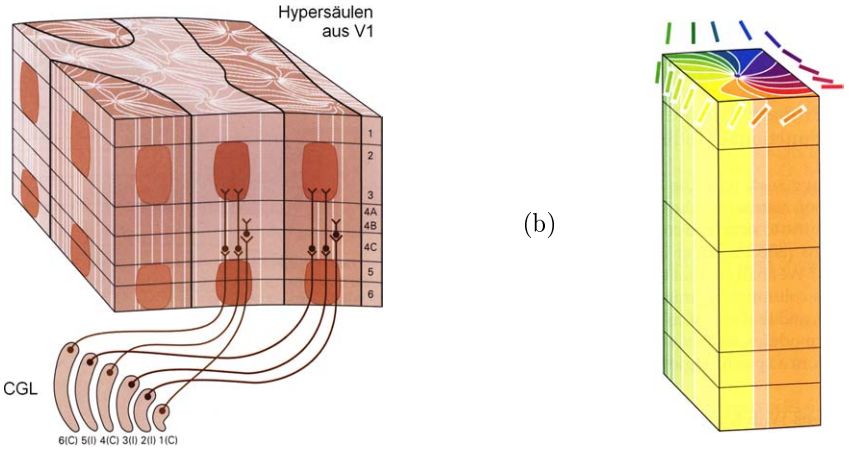


Abbildung 2.4: (a) Schematische Darstellung eines Teils des primären visuellen Cortex mit markierten Schichten und Verbindungen vom CGL. Quelle: [30], bearbeitet. (b) Abstrahiertes Modell einer Hypersäule. Die verschiedenen Farben geben die Orientierung des rezeptiven Feldes der einfachen Zellen an der jeweiligen Stelle an. Quelle: [30], bearbeitet

Der primäre visuelle Cortex (englisch auch *Visual Area 1* oder *V1*) ist wie der CGL retinotop. Benachbarte Bereiche entsprechen also immer noch benachbarten Bereichen der Netzhaut und somit des Sehfeldes. In die Tiefe existieren sechs Schichten, wobei untereinanderliegende Bereiche immer den gleichen Bereich des Sehfeldes verarbeiten. Diese „Säulen", die für die Verarbeitung eines bestimmten räumlichen Bereichs zuständig sind, werden *Hypersäulen* (englisch: *Hypercolumns*) genannt. Fast alle Verbindungen vom CGL enden in Schicht vier, wobei Verbindungen der M-Bahn in anderen Teilschichten als Verbindungen der P-Bahn enden. Die bereits im Sehnerv vorhandene Trennung beider Signalwege wird also hier fortgeführt. Die Struktur der Hypersäulen ist in Abbildung 2.4 (a) dargestellt.

Im primären visuellen Cortex existieren zwei Haupttypen von Zellen: *einfache* und *komplexe Zellen* (englisch: *simple* und *complex cells*). Einfache Zellen sind Kantendetektoren. Ihre Erregung ist am größten, wenn an einer bestimmten Stelle im Sehfeld ein Liniensegment in einer bestimmten Orientierung auftritt. Dieses Verhalten erreichen sie, indem sie Eingaben mehrerer Zellen mit rundem rezeptiven Feld erhalten, die in einer Linie angeordnet sind (siehe Abbildung 2.5). Komplexe Zellen erhalten ihre Eingaben größtenteils von mehreren einfachen Zellen. Sie können dadurch auf komplexere Merkmale des Sehfeldes reagieren.

Jede Hypersäule enthält einfache Zellen aller möglichen Orientierungen, siehe Abbildung

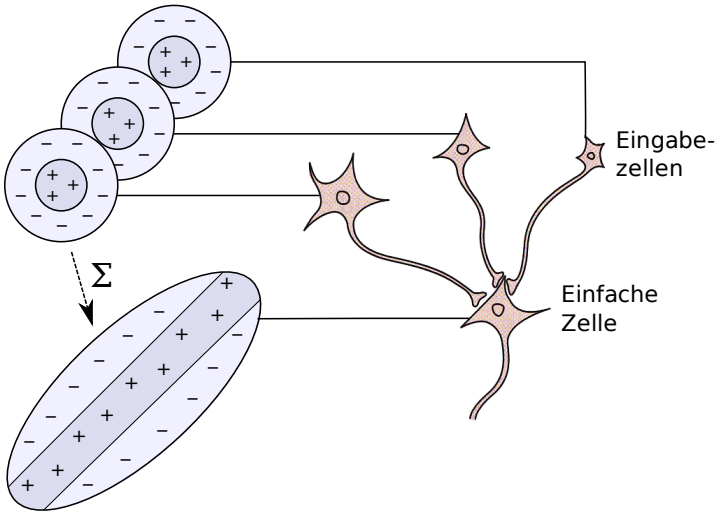


Abbildung 2.5: Funktionsweise von einfachen Zellen in V1. Quelle: [30], bearbeitet

2.4 (b). Somit können innerhalb jedes kleinen Bereichs des Sehfeldes Kanten in allen Orientierungen unterschieden werden. Zur Weiterverarbeitung von Farbkontrasten existieren sogenannte *Blobs*, die keine bestimmte Orientierung aufweisen. Alle Hypersäulen sind nahezu identisch aufgebaut und können damit als eine Art „Modul" zur Verarbeitung eines räumlichen Bereichs aufgefasst werden. Es existieren neben den Verbindungen innerhalb einer Hypersäule auch Verbindungen zwischen verschiedenen Hypersäulen, die vermutlich zur Verarbeitung von Kontextinformation außerhalb des eigenen rezeptiven Feldes genutzt werden.

Tiefere Bereiche der visuellen Informationsverarbeitung

Nach dem primären visuellen Cortex verläuft die Weiterverarbeitung der visuellen Informationen in zwei räumlich getrennten Bahnen – der *dorsalen* und der *ventralen Bahn*. Im Gegensatz zu den M- und P-Bahnen verlaufen diese nun nicht mehr nebeneinander, sondern zum Teil durch unterschiedliche Bereiche des Gehirns. Die dorsale Bahn entspricht im Wesentlichen der Fortführung der M-Bahn. Sie verarbeitet hauptsächlich Bewegungs- und Tiefeninformationen. Die ventrale Bahn kann dementsprechend als Fortführung der P-Bahn gesehen werden. Hier werden hauptsächlich Form und Farbe verarbeitet. Es wird vermutet, dass die ventrale Bahn damit essenziell für Objekterkennung verantwortlich ist. Die Eigenschaften und Aufgaben der Bahnen lassen sich jedoch nicht so stark voneinander abgrenzen wie bei den M- und P-Bahnen, es existieren darüber hinaus sehr viele Verbindungen zwischen den Bahnen und mehrere Bereiche wie zum Beispiel *V2* (*Visual Area 2*), die von beiden Bahnen durchlaufen werden. Außerdem gibt es wesentlich mehr bidirektionale Verbindungen zwischen den Bereichen, so dass man anders als bei den aufeinanderfolgenden Bereichen Netzhaut, CGL und V1 nicht mehr von einer hauptsächlich vorwärtsgericheten Verarbeitung sprechen kann.

Insgesamt lässt sich sagen, dass die rezeptiven Felder von Zellen bezogen auf das Sehfeld bei fortschreitender Verarbeitung auf den Sehbahnen meist größer werden auf komplexere Merkmale reagieren. So konnte zum Beispiel gezeigt werden, dass es im Bereich *IT* (*Inferior Temporal Cortex*) Zellen gibt, die ausschließlich auf gesichtsartige Formen reagieren [31].

2.2 Künstliche neuronale Netze zur Muster- und Objekterkennung

Inspiriert durch die biologischen Erkenntnisse über das Nervensystem entstanden abstrahierte Modelle für neuronale Netze, die als *künstliche neuronale Netze* bezeichnet werden. Häufig können diese in Software umgesetzt und damit evaluiert und praktisch genutzt werden. Nachfolgend werden die Eigenschaften solcher Modelle beschrieben. Abschnitt 2.2.2 führt dann das für diese Arbeit wesentliche Modell des *mehrschichtigen Perzeptrons* ein. In Abschnitt 2.2.3 wird ein Überblick über darauf aufbauende sowie andere bekannte Modelle künstlicher neuronaler Netze gegeben.

2.2.1 Eigenschaften künstlicher neuronaler Netze

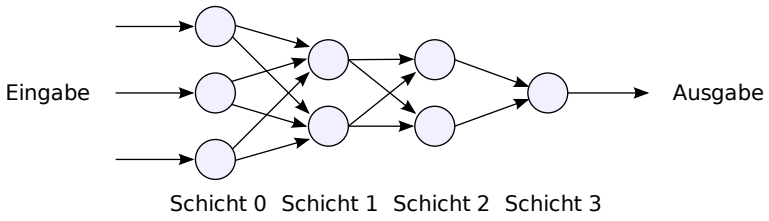


Abbildung 2.6: Vorwärtsgerichtetes neuronales Netz

Die Umsetzung neuronaler Netze in Software dient nicht nur dem besseren Verständnis des Nervensystems, sondern bietet auch für die Informatik äußerst interessante Möglichkeiten. Das Grundkonzept ist die Verbindung vieler gleichartiger und sehr einfach aufgebauter „Zellen“ zu einem komplexen Netzwerk. Dieses Konzept wird *Konnektionismus* genannt. Im Vergleich zu klassischen Algorithmen weisen neuronale Netze insbesondere folgende Eigenschaften auf:

- *Lernfähigkeit.* Die Parameter des Modells werden meist nicht manuell festgelegt, sondern durch ein Lernverfahren anhand von Trainingsbeispielen gelernt.

- *Robustheit.* Durch geschickt gewählte Lernmethoden und Trainingsdaten können neuronale Netze im Allgemeinen gut generalisieren, das heißt sie liefern auch bei verrauschten oder unbekannten Eingaben sinnvolle und erwünschte Ausgaben. Durch die verteilte Wissensrepräsentation (zum Beispiel in Form von vielen gelernten Verbindungsgewichten) weisen neuronale Netze außerdem eine gewisse Läsionstoleranz auf, das heißt sie verhalten sich bei Fehlern im Netz oder bei Entfernung von Teilen des Netzes „gutartig“ und verlieren nur einen Teil ihrer Funktionalität.

- *Parallelität.* Neuronale Netze arbeiten grundsätzlich massiv parallel. Dadurch können sie auf parallel arbeitender Hardware eine sehr hohe Performanz erreichen und komplexe Aufgaben in kurzer Zeit lösen.

Je nach zugrundeliegendem Modell unterscheidet sich die Topologie eines neuronalen Netzes. Meist sind die Neuronen in aufsteigend nummerierte *Schichten* unterteilt. In diesem Zusammenhang werden zwei grundlegende Topologien unterschieden: *vorwärtsgerichtete* Netze besitzen ausschließlich Verbindungen von Schicht n zu Schicht m, $m > n$, *rekurrente* oder *rückgekoppelte* Netze können zusätzlich Verbindungen innerhalb einer Schicht oder zu Schichten $m < n$ enthalten. Meist ist $m = n + 1$ für vorwärtsgerichtete beziehungsweise $m = n$

oder $m = n - 1$ für rekurrente Verbindungen, ansonsten spricht man von Netzen mit *abkürzenden Verbindungen* (englisch: *shortcut connections*). Abbildung 2.6 zeigt die Struktur eines vorwärtsgerichteten Netzes, die Pfeile geben hierbei die Richtung des Datenflusses an. Die Eingabe erfolgt in Schicht 0 und die Ausgabe in Schicht 3.

Neuronale Netze eignen sich sehr gut für Klassifikations- beziehungsweise Mustererkennungs-Aufgaben. Dazu wird das Netz mit einer großen Anzahl bekannter Muster trainiert. Nach erfolgreichem Training ist es dann in der Lage, auch ähnliche, bisher unbekannte oder verrauschte Muster richtig zu klassifizieren. Ein typisches Beispiel für die Anwendung neuronaler Netze ist die Erkennung von handgeschriebenen Ziffern, die auch in späteren Kapiteln dieser Arbeit noch behandelt wird.

2.2.2 Das mehrschichtige Perzeptron

Das Perzeptron ist das wahrscheinlich bekannteste Modell eines neuronalen Netzes. Es wurde von Frank Rosenblatt erstmals 1958 in [56] vorgestellt. Das dort eingeführte Modell unterscheidet sich allerdings in einigen Punkten vom heute gebräuchlichen. Die ursprüngliche Form des Perzeptrons – das *zweischichtige Perzeptron* – wird hier nicht gesondert behandelt, es wird stattdessen sofort das allgemeinere *mehrschichtige Perzeptron* eingeführt. Da das Perzeptron nicht eindeutig definiert ist, können sich einige der nachfolgend genannten Eigenschaften, je nach verwendeter Literatur, unterscheiden. Für detailliertere Informationen zur Geschichte des Perzeptrons sei an dieser Stelle auf [67] verwiesen.

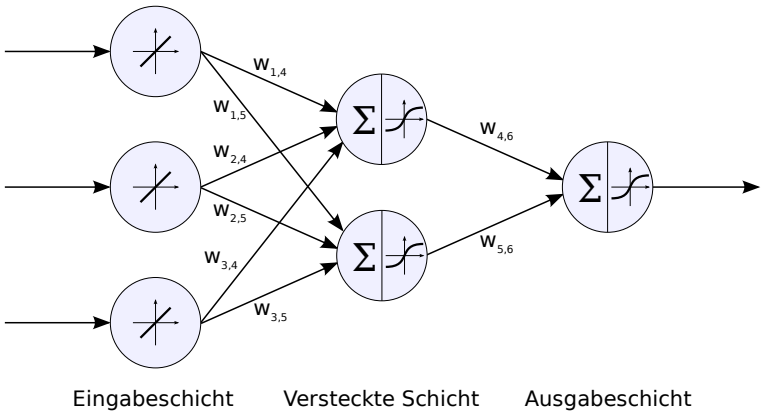


Abbildung 2.7: Dreischichtiges Perzeptron mit insgesamt sechs Neuronen

Aufbau und Funktionsweise

Ein (*mehrschichtiges*) *Perzeptron* ist ein vorwärtsgerichtetes, mindestens zweischichtiges neuronales Netz ohne abkürzende Verbindungen. Die erste Schicht heißt *Eingabeschicht*, die letzte *Ausgabeschicht*. Dazwischenliegende Schichten werden als *versteckte Schichten* bezeichnet (siehe Abbildung 2.7). Dementsprechend heißen die Neuronen der Eingabeschicht *Eingabeneuronen*, die der versteckten Schicht *versteckte Neuronen* und die der Ausgabeschicht *Ausgabeneuronen*. Jedes Neuron verfügt über einen reellen internen Zustand, der *Netzeingabe* genannt wird und mit net_j bezeichnet wird, wobei j der Index des Neurons ist. Die Eingabe in das Netz erfolgt, indem die Netzeingaben der Eingabeneuronen auf die gewünschten Daten gesetzt werden. Als Beispiel kann man bei der Erkennung von handgeschriebenen Ziffern ein Eingabeneuron für jeden Pixel des Bereichs verwenden, in dem die Ziffern erkannt werden sollen (siehe Abbildung 2.8). Weiterhin hat jedes Neuron einen reellen

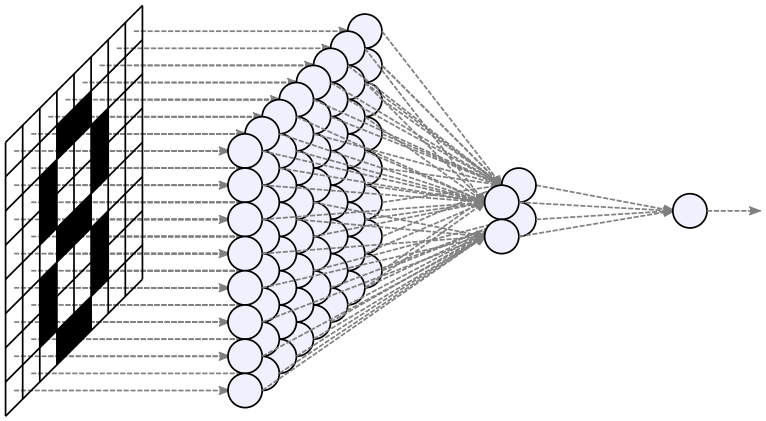


Abbildung 2.8: Eingabe eines Musters in ein mehrschichtiges Perzeptron

Ausgabewert, der *Aktivierung* genannt wird. Dieser Wert entspricht dem Aktionspotenzial von biologischen Neuronen. Er wird mit a_j bezeichnet und ergibt sich durch Berechnung der *Transfer-* oder *Aktivierungsfunktion* $f_j : \mathbb{R} \to \mathbb{R}$ aus der Netzeingabe, es gilt also

$$a_j = f_j(net_j).$$

Üblicherweise wird als Aktivierungsfunktion in der Eingabeschicht die Identität

$$f_{Eingabe}(net_j) = net_j$$

benutzt, während in allen anderen Schichten eine sigmoide (und somit nichtlineare) Funktion wie zum Beispiel die Fermi-Funktion oder der Tangens Hyperbolicus verwendet werden:

$$f_{Fermi}(net_j) = \frac{1}{1 + \exp(-net_j)} \quad \text{bzw.} \quad f_{tanh}(net_j) = \tanh(net_j).$$

Perzeptrons sind üblicherweise *vollverknüpft*. Das bedeutet, dass jedes Neuron der Schicht n Verbindungen zu jedem Neuron der Schicht $n+1$ besitzt. Jede Verbindung zwischen zwei Neuronen hat ein relles *Verbindungsgewicht*, das in Abbildung 2.7 mit $w_{i,j}$ bezeichnet ist. Dabei ist i der Index des Quellneurons und j der Index des Zielneurons.

Das „Durchrechnen“ eines Perzeptrons nennt man *Vorwärtspropagierung* (englisch: *forward propagation* oder *forward pass*). Dabei werden in diskreten Zeitschritten alle Schichten, beginnend mit der zweiten, nacheinander bearbeitet. Pro Schritt werden für alle Neuronen der aktuellen Schicht die Netzeingaben berechnet. Dies geschieht, indem über alle eingehenden Verbindungen eines Neurons eine Summe über die Aktivierungen der jeweiligen Quellneuronen gebildet wird. Jede Aktivierung wird dabei mit dem entsprechenden Verbindungsgewicht multipliziert, so dass Verbindungen mit betragsmäßig größerem Gewicht einen höheren Einfluss auf die Netzeingabe haben. Zusätzlich wird ein Schwellwert addiert, der *Bias* heißt und mit θ_j bezeichnet wird. Als Formel ausgedrückt berechnet sich die Netzeingabe eines Neurons mit dem Index j als

$$net_j = \sum_{i \in I} (a_i \cdot w_{i,j}) + \theta_j, \tag{2.1}$$

mit $I = \{\text{Indizes aller Neuronen, die Verbindungen zu Neuron j haben}\}$.

Anschließend wird die Aktivierung des aktuellen Neurons durch Anwendung der Aktivierungsfunktion auf die Netzeingabe berechnet. Diese Aktivierungen werden dann im nächsten Schritt zur Berechnung der Netzeingabe für die Neuronen der nächsten Schicht verwendet und so weiter. Die Ausgabe des Netzes bilden schließlich die Aktivierungen der Ausgabeneuronen.

Theoretische Fähigkeiten mehrschichtiger Perzeptrons

Die beschriebene Funktionsweise sagt a priori wenig über die theoretischen Fähigkeiten von Perzeptrons aus. Grundsätzlich leistet ein Perzeptron mit N Eingabeneuronen und M Ausgabeneuronen eine nichtlineare Abbildung $\mathbb{R}^N \to \mathbb{R}^M$. Die Fähigkeiten unterscheiden sich jedoch in Abhängigkeit von der Anzahl der Schichten und der Anzahl Neuronen pro Schicht. Im Folgenden wird angenommen, dass jedes Neuron (abgesehen von den Eingabeneuronen) nur die Ausgaben 0 und 1 erzeugen kann, also als Aktivierungsfunktion eine *binäre Schwellwertfunktion*

$$a_j = f_j(net_j) = \begin{cases} 0, & \text{falls } net_j < 0 \\ 1, & \text{falls } net_j \geq 0 \end{cases} \tag{2.2}$$

besitzt. Dies ist zwar eine Einschränkung zum Beispiel gegenüber der stetigen Ausgabe der Fermifunktion im Intervall $[0, 1]$, ist aber für folgendes Beispiel anschaulicher und lässt sich leicht auf den allgemeinen Fall übertragen.

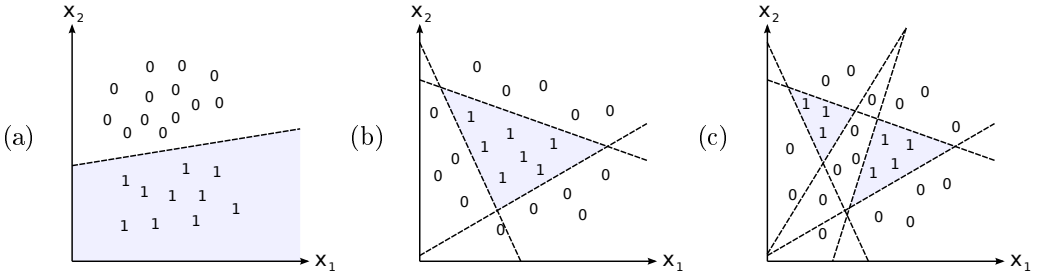


Abbildung 2.9: Klassifizierungsfähigkeiten eines Ausgabeneurons von (a) zweischichtigen, (b) dreischichtigen und (c) vierschichtigen Perzeptrons für eine zweidimensionale Eingabe. Jede Ziffer repräsentiert eine Eingabe in das Netz, ihr Wert repräsentiert die entsprechende Ausgabe nach der Vorwärtspropagierung

Für Klassifikationsaufgaben werden häufig Perzeptrons mit mehreren Ausgabeneuronen verwendet. Jedes Ausgabeneuron repräsentiert eine Klasse und soll den Wert 1 annehmen, wenn das Eingabemuster zur Klasse gehört, ansonsten soll der Wert 0 angenommen werden. Ein Beispiel dafür ist die Erkennung von handgeschriebenen Ziffern, bei der für jede Ziffer ein Ausgabeneuron existiert. Wie Minsky und Papert in [41] erstmals zeigten, kann ein Ausgabeneuron eines zweischichtigen Perzeptrons lediglich *linear separieren*. Das bedeutet, dass ein Ausgabeneuron j die beiden Eingabemengen, die eine Aktivierung auslösen ($a_j = 1$) beziehungsweise nicht auslösen ($a_j = 0$) bei einem N-dimensionalem Eingaberaum nur durch eine $(N-1)$-dimensionale Hyperebene trennen kann. Dies kann man im Fall einer zweidimensionalen Eingabe (also zwei Eingabeneuronen) leicht durch Umstellung der Aktivierungsungleichung 2.2 mit eingesetzter Netzeingabe (Gleichung 2.1) sehen:

$$a_1 w_{1,j} + a_2 w_{2,j} + \theta_j \geq 0 \quad \Leftrightarrow \quad a_2 \geq \frac{1}{w_{2,j}} \left(\theta_j - a_1 w_{1,j}\right).$$

Es ergibt sich eine lineare Funktion, die im zweidimensionalen Fall als Geradengleichung aufgefasst werden kann, siehe dazu Abbildung 2.9 (a).

Bei dreischichtigen Perzeptrons kann folglich bereits jedes Neuron der versteckten Schicht eine lineare Separierung leisten. Die Neuronen der Ausgabeschicht können die so entstandenen Hyperebenen dann durch entsprechende Verbindungsgewichte zu einem konvexen Polygon zusammenfügen, siehe Abbildung 2.9 (b). Das Polygon hat dabei höchstens so viele Kanten, wie das Ausgabeneuron Verbindungen zu versteckten Neuronen hat. Im Falle einer Vollverknüpfung entspricht das genau der Anzahl Neuronen in der versteckten Schicht.

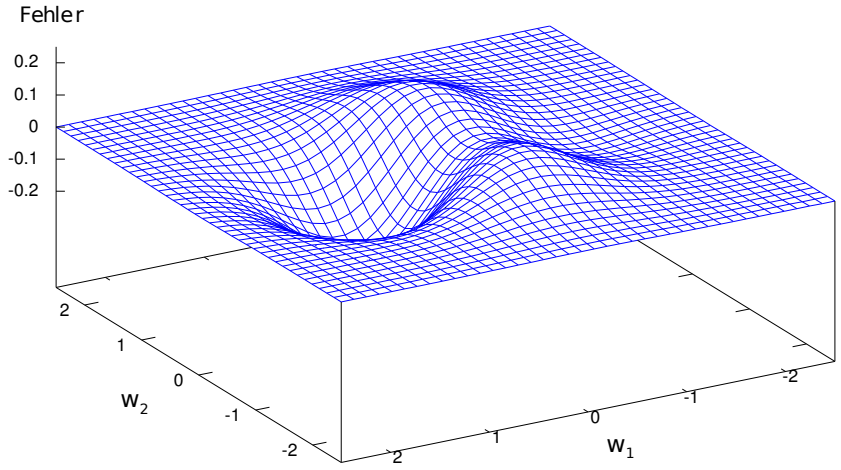


Abbildung 2.10: Beispiel einer Fehleroberfläche eines Perzeptrons mit zwei Verbindungsgewichten

Vierschichtige Perzeptrons können schließlich in der Ausgabeschicht auch Schnitte und Vereinigungen konvexer Polygone bilden, siehe Abbildung 2.9 (c). Bei ausreichender Anzahl Neuronen pro Schicht sind sie berechnungsuniversell, das heißt sie können jede beliebige Klassifizierung beziehungsweise Abbildung realisieren. Perzeptrons mit mehr als vier Schichten haben theoretisch die gleichen Fähigkeiten. In der Praxis können sie komplexe Aufgaben aber häufig besser lösen, da bei einer höheren Anzahl Schichten deutlich weniger Neuronen pro Schicht benötigt werden.

Allgemeines zu Lernverfahren

Im letzten Abschnitt wurde gezeigt, dass vier- oder mehrschichtige Perzeptrons in der Theorie jede beliebige Abbildung leisten können. Diese hängt von allen Parametern des Perzeptrons ab: der Anzahl Schichten, der Anzahl Neuronen, der Aktivierungsfunktion sowie der Verbindungen und Gewichte. Theoretisch könnte man all diese Parameter durch Lernen verändern, in der Praxis wird allerdings normalerweise nur durch Veränderung der Verbindungsgewichte gelernt.

Das Lernen kann *überwacht* oder *unüberwacht* erfolgen. Für beide Methoden benötigt man eine große Anzahl gültiger Eingaben für das Netz. Diese Eingaben werden im Folgenden *Trainingsmuster* genannt. Beim überwachten Lernen muss weiterhin die erwünschte Ausgabe des Netzes für jedes Trainingsmuster – der sogenannte *Teacher* – bekannt sein. Für das Beispiel der Ziffernerkennung hieße das, dass zum Beispiel mehrere tausend Trainingsmuster mit je einer handgeschriebenen Ziffer vorhanden sind und für jedes Muster bekannt ist, um welche Ziffer es sich handelt. Beim unüberwachten Lernen wird hingegen ausschließlich aus den Trainingsmustern selbst gelernt. Diese Variante wird allerdings im Folgenden nicht weiter behandelt.

Das bekannteste und auch in dieser Arbeit genutzte überwachte Lernverfahren für mehrschichtige Perzeptrons ist *Backpropagation of Error*.

Das Lernverfahren „Backpropagation of Error“

Beim Lernverfahren „Backpropagation of Error“ [57] werden alle Trainingsmuster nacheinander in das Netz eingegeben und vorwärtspropagiert. Für jedes Muster wird dabei die Abweichung der aktuellen von der gewünschten Ausgabe ermittelt. Dazu wird eine Summe der quadratischen Abweichungen von Ausgabe und Teacher über alle Ausgabeneuronen

berechnet:

$$E^p = \frac{1}{2} \sum_{m=1}^{M} (t_m^p - a_m^p)^2.$$

Dabei ist M die Anzahl der Ausgabeneuronen, a_m^p die Aktivität des Ausgabeneurons m nach der Vorwärtspropagierung von Muster p und t_m^p der Teacher des Neurons m für das Muster p. E^p wird *Einzelfehler* des Trainingsmusters p genannt. Die Multiplikation mit $\frac{1}{2}$ dient lediglich einer eleganteren Ableitung des Einzelfehlers, ist aber sonst nicht von Bedeutung.

Ein gesamter Durchlauf aller Trainingsmuster wird als *Epoche* bezeichnet. Häufig werden die einzelnen Trainingsmuster nach jeder Epoche pseudozufällig umsortiert, um das Training robuster zu machen. Nach jeder Epoche kann der sogenannte *Gesamtfehler* F angegeben werden, welcher der Summe aller Einzelfehler der Epoche entspricht, es gilt also

$$F = \sum_{p=1}^{P} E^p,$$

wobei P die Anzahl der Trainingsmuster ist.

Da der Gesamtfehler von allen Gewichten abhängt, kann man ihn sich als Fehleroberfläche im Raum der Gewichte vorstellen. Für Perzeptrons mit nur zwei Gewichten ist eine solche in Abbildung 2.10 skizziert. Die meisten Perzeptrons besitzen allerdings mehrere tausend oder sogar millionen Gewichte, so dass eine Visualisierung für solche Netze unmöglich ist. Jeder mögliche Zustand der Gewichte wird durch einen Punkt auf der Fehleroberfläche repräsentiert. Das Ziel des Lernverfahrens ist es nun, den Gesamtfehler durch Anpassung der Verbindungsgewichte zu minimieren. Am Beispiel von Abbildung 2.10 hieße das, dass sich der erwünschte Zustand der Gewichte genau in einem globalen Minimum der Fehleroberfläche befindet. Aufgrund der hohen Dimensionalität und der Nichtlinearität ist diese Minimierung aber im Allgemeinen nicht geschlossen möglich, stattdessen wird der Fehler durch *Gradientenabstieg* sukzessive verringert. Dazu wird der Einzelfehler jedes Trainingsmusters nach jedem Gewicht abgeleitet. Der daraus resultierende Vektor im Gewichtsraum heißt *Gradient*. Er zeigt in die Richtung des steilsten Anstiegs der Ableitung, seine Länge entspricht der Steigung. Um den Fehler zu verringern, muss das Gewicht in die entgegengesetzte Richtung des Gradienten verändert werden. Bei Backpropagation of Error ist diese Veränderung immer proportional zur Steigung, es gilt

$$\Delta w_{ij} = \eta \cdot \frac{\partial E^p(w_{ij})}{\partial w_{ij}}.$$

Hierbei ist Δw_{ij} die Änderung des Verbindungsgewichts von Neuron i zu Neuron j und $\eta \in \mathbb{R}$ die sogenannte *Lernrate*. Sie darf nicht zu groß gewählt werden, da sonst eventuell Minima übersprungen werden. Zu kleine Lernraten führen im Gegensatz dazu, dass das Training deutlich länger dauert, also mehr Epochen für ein vergleichbares Ergebnis benötigt werden. Sinnvolle Werte für die Lernrate liegen oft im Bereich $[0{,}01; 0{,}1]$.

Auf die Formeln zur Berechnung der Ableitung wird an dieser Stelle verzichtet. Es sei allerdings erwähnt, dass der an den Ausgabeneuronen ermittelte Einzelfehler sukzessive durch alle Schichten zurückpropagiert wird, woraus sich der Name des Lernverfahrens ergibt. Eine ausführliche Herleitung findet sich zum Beispiel in [67].

Die Anpassung der Gewichte kann entweder nach jedem Muster (*Online-Learning*) oder nur nach jeder Epoche (*Batch-Learning*) erfolgen. Die Schrittweite der Gewichtsänderung sollte bei beiden Verfahren in der gleichen Größenordnung liegen, um eine robuste Erkennung von Fehlerminima zu ermöglichen. Aus diesem Grund ist das Lernen bei Gewichtsanpassung nach jedem Muster deutlich schneller, weil Gewichtsänderungen häufiger erfolgen.

Es ist allerdings formal nicht sichergestellt, dass auf diese Weise tatsächlich der Gesamtfehler minimiert wird, es werden stattdessen nur die Einzelfehler schrittweise minimiert. Im ungünstigsten Fall könnte jedes Muster die Gewichtsänderungen des letzten Musters wieder rückgängig machen, wodurch sich gar kein Lerneffekt einstellen würde. Die praktische Erfahrung zeigt allerdings, dass Online-Learning gut funktioniert und häufig deutlich schneller einen zufriedenstellenden Lernerfolg erzielt als Batch-Learning. Einen Kompromiss aus beiden Methoden stellen *Mini-Batches* dar. Bei dieser Variante wird die Gewichtsänderung immer nach einer festen Anzahl von Trainingsmustern, zum Beispiel 16 Stück, durchgeführt.

2.2.3 Kurzüberblick über andere Modelle

Neben dem Perzeptron existiert eine Fülle weiterer Modelle neuronaler Netze, von denen einige in diesem Abschnitt kurz vorgestellt werden. Die Auswahl erfolgte dabei nach dem Gesichtspunkt, welche Modelle sich grundsätzlich gut für Objekterkennungsaufgaben eignen.

Neocognitron

Das *Neocognitron* ist ein mehrschichtiges, vorwärtsgerichtetes neuronales Netz ohne abkürzende Verbindungen, das speziell zur Muster- beziehungsweise Objekterkennung entworfen wurde. Es wurde von Fukushima erstmals 1980 in [19] vorgestellt. Das Modell ist der Informationsverarbeitung des menschlichen Sehsystems bis zum primären visuellen Cortex nachempfunden (vergleiche Abschnitt 2.1.2).

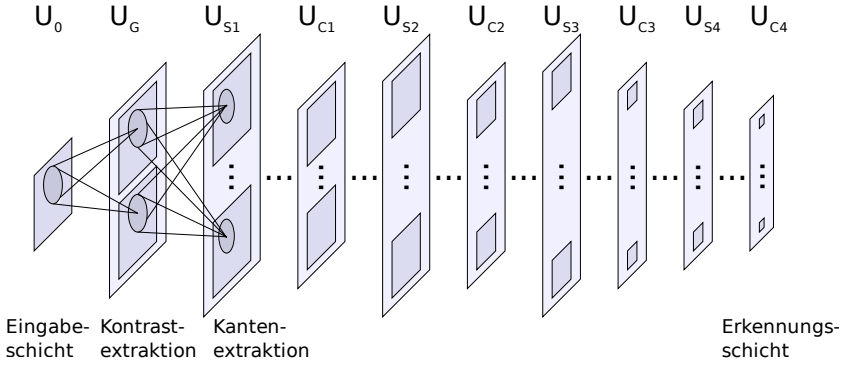


Abbildung 2.11: Struktur des Neocognitrons. Nach [19]

Die Eingabeschicht repräsentiert die Photorezeptoren der Netzhaut, in der zweiten Schicht werden Kontraste extrahiert. Danach wechseln sich *S-* und *C-Schichten* ab. Beide enthalten jeweils mehrere retinotope Flächen von Zellen (englisch: *cell planes*), die durch ein räumlich beschränktes rezeptives Feld zu den Flächen der vorangehenden Schicht verbunden sind. Die Zellen in S-Schichten sind einfachen Zellen des primären visuellen Cortex nachempfunden und dienen als Merkmalsextraktoren. Ihre Eingabeverbindungen können durch Lernen verändert werden. Die Zellen der C-Schichten sind komplexen Zellen nachempfunden. Sie vereinigen die Eingaben mehrerer räumlich nah beieinander liegender S-Zellen, um eine gewisse Translationsinvarianz zu erreichen. Ihre Verbindungen sind fest und werden nicht durch Lernen verändert.

Sowohl bei S- als auch bei C-Schichten sind die Verbindungsgewichte pro Zellfläche *gekoppelt* (englisch: *shared weights*). Das bedeutet, dass pro Fläche nur ein Satz von Gewichten existiert, der für jede Zelle angewandt wird. Mit anderen Worten gewichtet also jede Zelle Eingaben an den gleichen Stellen ihres rezeptiven Feldes identisch, nur die Eingaben sind aufgrund der verschiedenen rezeptiven Felder der Zellen unterschiedlich. Im später folgenden

Kapitel 4 werden die Eigenschaften gekoppelter Gewichte noch einmal genauer diskutiert und mit nicht gekoppelten Gewichten verglichen.

Das Training eines Neocognitrons kann überwacht oder unüberwacht erfolgen. Für beide Arten existieren verschiedene Trainingsvarianten. Außerdem gibt es verschiedene Erweiterungen des Neocognitrons, zum Beispiel rekurrente Verbindungen [20].

HMAX-Modell

Das HMAX-Modell von Riesenhuber und Poggio [54] ist ein ebenfalls biologisch motiviertes Modell, das die ventrale Bahn des menschlichen Sehsystems nachbildet. Wie beim Neocognitron handelt es sich auch hier um ein mehrschichtiges, vorwärtsgerichtetes Netz. Besonderer Wert wurde auf biologisch plausible, experimentell ermittelte Parameter gelegt. So sind zum Beispiel die Größe beziehungsweise der Winkel rezeptiver Felder so gewählt, dass sie sich mit biologischen Forschungsergebnissen decken.

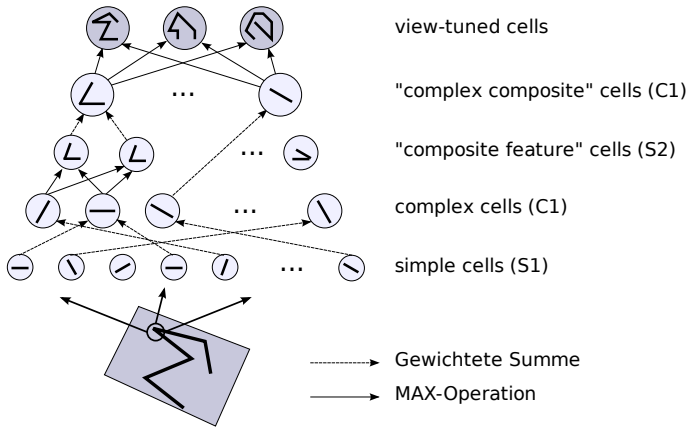


Abbildung 2.12: Struktur des HMAX-Modells. Nach [54]

Das Netz besteht aus je zwei sich abwechselnden Schichten von S- und C-Zellen, gefolgt von einer Ausgabeschicht (siehe Abbildung 2.12). Jede S1-Zelle reagiert durch Anwendung eines zweidimensionalen Filters (zweite Ableitung der Gaußfunktion) auf Balken einer bestimmten Orientierung im Eingabebild. Vergleichbar mit den in Abschnitt 2.1.2 vorgestellten Hypersäulen exisitieren für jede Position des Eingabebildes mehrere S1-Zellen mit verschiedenen Filterorientierungen und -größen.

C1-Zellen dienen als „Pooling"-Zellen, das heißt, sie fassen die Aktivierungen mehrerer benachbarter S1-Zellen, deren Filter alle die gleiche Orientierung besitzen, zusammen. Damit wird eine gewisse Invarianz gegenüber Translationen des Eingabemusters erreicht. Die Aktivierung der C1-Zellen ergibt sich dabei nicht wie zum Beispiel beim Perzeptron durch eine gewichtete Summe, sondern durch die Auswahl der größten Eingabe aller verbundenen S1-Zellen (*MAX-Operation*). Die Autoren begründen dies damit, dass bei Verwendung einer gewichteten Summe als Netzeingabe der Fall auftreten kann, dass alle zu einer C1-Zelle verbundenen C2-Zellen eine geringe Aktivierung aufweisen, die aber in der Summe genauso groß sind wie die einer einzelnen Zelle, die stark auf ein vorhandenes Merkmal reagiert. Dadurch ist die Spezifität der Eingabe nicht gegeben, es kann also anhand der Ausgabe der C1-Zelle nicht festgestellt werden, ob ein Merkmal im entsprechenden Bereich tatsächlich stark ausgeprägt ist.

In der S2-Schicht werden Ausgaben benachbarter C1-Zellen zu komplexeren Merkmalen kombiniert. An jeder Position existieren Zellen für alle möglichen Kombinationen von Orientierungen. In der danach folgenden C2-Schicht werden Ausgaben benachbarter S2-Zellen

dann analog zur C1-Schicht durch MAX-Operationen zusammengefasst.

Beim Lernen werden ausschließlich die Verbindungsgewichte der Ausgabeschicht verändert. Jedes Neuron dieser Schicht soll eine *Ansicht* (englisch: *view*) eines bestimmten Objektes repäsentieren, die sich aus der Kombination der Ausgaben mehrerer C2-Zellen ergibt.

Konvolutionsnetze am Beispiel von LeNet-5

Die grobe Struktur von *Konvolutionsnetzen* ist den beiden bereits vorgestellten Modellen ähnlich. Auch hier ist die Eingabe zweidimensional und es gibt mehrere Schichten. Jede Schicht unterteilt sich in ein oder mehrere gleich große Flächen von Zellen. Jede Zellfläche verfügt über retinotope Verbindungen zu ein oder mehreren Zellflächen der vorhergehenden Schicht, die rezeptiven Felder aller Zellen sind dabei alle quadratisch und entsprechen jeweils einem gleich großen Bereich der Quellfläche. Die Gewichte sind innerhalb einer Verbindung gekoppelt. Da somit jede Zelle einer Zellfläche auf die gleichen Merkmale innerhalb ihres rezeptiven Feldes reagiert, werden die Zellflächen bei Konvolutionsnetzen *Merkmalskarten* (englisch: *Feature Maps*) genannt. Mathematisch entspricht die Vorwärtspropagierung von einer zur nächsten Schicht somit einer Konvolution (Faltung) mit einem quadratischen Filter pro Verbindung zweier Zellflächen.

Bei Konvolutionsnetzen können im Gegensatz zu den vorherigen Modellen alle Vebindungsgewichte durch Lernen, zum Beispiel mit Backpropagation of Error, verändert werden.

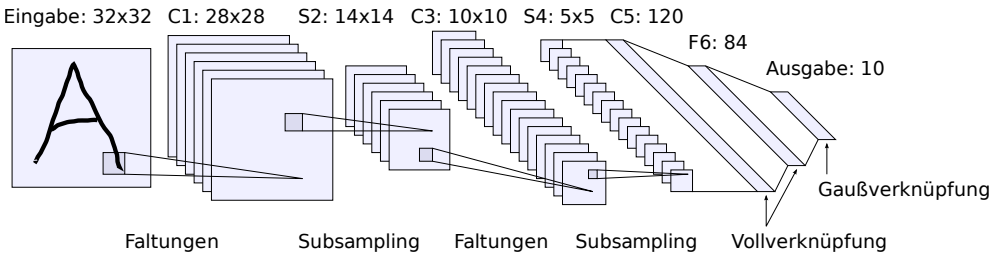


Abbildung 2.13: Struktur des LeNet-5-Konvolutionsnetzes. Nach [36]

Ein bekanntes Beispiel für Konvolutionsnetze ist das von LeCun entwickelte *LeNet-5* [36], das in Abbildung 2.13 dargestellt ist. Das Netz besteht aus insgesamt sieben Schichten, wobei die Eingabeschicht nicht mitgezählt wird. Es ist von der Dimensionierung auf die Erkennung von handgeschriebenen Ziffern ausgelegt, eignet sich aber grundsätzlich auch für alle anderen bildartigen Eingabedaten. Die ersten vier Schichten sind jeweils aufeinanderfolgende Konvolutions- und Subsamplingschichten. Die Konvolutionsschichten verhalten sich wie oben beschrieben, die Subsamplingschichten repräsentieren komplexe Zellen des primären visuellen Cortex. Sie dienen der Zusammenfassung von jeweils vier benachbarten Quellzellen einer Zellfläche zu einer Zielzelle, womit die Translationsinvazianz verbessert werden soll. Dabei verfügt jede Zellfläche der Subsamplingschicht über einen Koeffizienten, mit dem die vier Eingabezellen multipliziert werden. Dieser wird wie die Gewichte der Konvolutionsschichten durch Backpropagation of Error gelernt. Zusätzlich wird beim Subsampling wie auch bei der Konvolution ein ebenfalls gelernter Schwellwert (Bias) addiert.

Die ersten beiden Schichten verfügen jeweils über sechs Merkmalskarten, Schicht drei und vier über je sechzehn. Die Verknüpfung von Schicht zwei zu Schicht drei ist nicht vollverknüpft, sondern nach einer Verknüpfungstabelle aufgebaut. Jede Zielkarte hat Verbindungen zu drei bis sechs Quellkarten. Im Gegensatz zu einer Vollverknüpfung soll das Netz damit besser in der Lage sein, für jede Zielkarte unterschiedliche Merkmale zu extrahieren.

Schicht fünf und sechs sind komplett vollverknüpft zur jeweils vorhergehenden Schicht. Schicht sieben nutzt Zellen mit *radialen Basisfunktionen* (*RBF*) [51]. Diese Zellen ermit-

teln den euklidischen Abstand zwischen dem Eingabevektor und einem Parametervektor mit gleicher Dimensionalität, hier 84. Die Ausgabe der Zelle wird dann berechnet, indem eine radiale Basisfunktion auf diese Differenz angewandt wird. Bei LeNet-5 wird als solche die Gaußfunktion verwendet. Der Parametervektor jeder RBF-Zelle wird anfangs auf ein stilisiertes Bild der zu erkennenden Ziffer gesetzt, wobei die 84 Dimensionen als zweidimensionales, 7×12 Pixel großes Bild behandelt werden. Im Gegensatz zu einer 1-aus-N-Codierung mit herkömmlichen Zellen (gewichtete Summe und sigmoide Aktivierungsfunktion) sind die RBF-Zellen laut [36] leichter zu trainieren und können Eingaben, die keiner Ziffer entsprechen, besser zurückweisen, da sie nur in einem relativ kleinen Bereich ihres Eingaberaums eine hohe Aktivierung aufweisen.

Neuronale Abstraktionspyramide

Die *neuronale Abstraktionspyramide* wurde von Behnke in [2] vorgestellt. Neben dem hierarchischen Aufbau in mehreren zweidimensionalen Schichten mit jeweils einem oder mehreren Merkmalsfeldern (englisch: *Feature Arrays*) besitzt dieses Modell als einziges der hier vorgestellten die Möglichkeit rekurrenter Verbindungen zu einer vorhergehenden Schicht oder innerhalb der gleichen Schicht. Damit sollen Mehrdeutigkeiten in einem Bild mit Hilfe der lokalen Umgebung durch Vorwärts- und Rückwärtspropagierung sowie laterale Verbindungen (das heißt innerhalb einer Schicht) aufgelöst werden. Dadurch wird die anfängliche Interpretation des Eingabebildes iterativ verbessert. Abbildung 2.14 zeigt die Struktur des Modells.

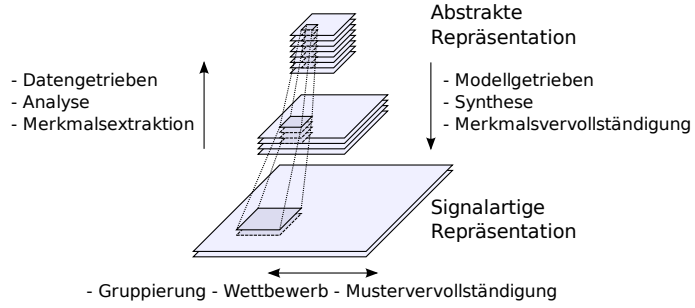


Abbildung 2.14: Struktur der neuronalen Abstraktionspyramide. Nach [1]

Die Höhe und Breite der Merkmalsfelder und somit die Anzahl der darin enthaltenen Zellen verringert sich mit jeder Schicht, um den Abstraktionsgrad zu erhöhen und die Abhängigkeit vom Ort zu reduzieren. Analog dazu wird die Anzahl der Merkmalsfelder pro Schicht erhöht, um die Mächtigkeit der repräsentierten Merkmale zu gewährleisten.

Gleiche Positionen in verschiedenen Merkmalsfeldern einer Schicht werden als Hypersäulen betrachtet; benachbarte Positionen einer Schicht bilden Hyper-Nachbarschaften. Verbindungen von einer Schicht zu einer anderen sind retinotop, das heißt, das Zentrum des rezeptiven Feldes einer Zelle liegt an der relativ gesehen gleichen Position wie die Zelle selbst. Laterale Verbindungen bestehen ausschließlich zu Zellen der Hyper-Nachbarschaft. Jede Verbindung von einem zu einem anderen Merkmalsfeld besitzt sogenannte *Projektionszellen* (englisch: *Projection Units*), die verschiedene Aktivierungsfunktionen aufweisen können. Die Ausgaben aller Projektionszellen wiederum bilden die Netzeingabe für die *Ausgabezellen* des Merkmalsfeldes, die über eine eigene Aktivierungsfunktion verfügen. Meist wird für Projektionszellen die Identität als Aktivierungsfunktion genutzt, dies entspricht dann einer gewichteten Summe aller eingehenden Verbindungen als Netzeingabe der Ausgabezellen. Jede Verbindung besitzt jeweils eigene, gekoppelte Gewichte.

Eingaben und Ausgaben können in jeder Schicht erfolgen, alternativ kann eine explizite Ausgabeschicht in Form einer 1-aus-N-Codierung hinzugefügt werden.

2.3 Vorstellung verschiedener Trainings- und Testdatensätze

Im vorhergehenden Abschnitt wurden verschiedene Modelle beschrieben, die zur Erkennung von Objekten in natürlichen Bildern oder anderen zweidimensionalen Eingabemustern verwendet werden können. Wie bereits erläutert, muss bei jedem Modell zunächst eine Menge von Parametern gelernt werden, damit Objekte anschließend wie gewünscht erkannt beziehungsweise klassifiziert werden können. Dieses Training erfordert eine große Anzahl an Mustern, weil das Modell nur anhand ausreichend vieler Beispiele die Gemeinsamkeiten und Unterschiede der verschiedenen Objektklassen lernen kann.

In diesem Abschnitt werden deshalb mehrere Datensätze vorgestellt, die sich als Trainings- und Testmenge für Objekterkennungssysteme eignen. Neben den Datensätzen natürlicher Bilder, die in 2.3.3 und 2.3.4 beschrieben sind, werden in dieser Arbeit auch handgeschriebene Ziffern (2.3.1) und nicht-natürliche Bilder (2.3.2) zur Evaluation verwendet.

2.3.1 Der MNIST-Datensatz

Der von LeCun und Cortes veröffentlichte *MNIST-Datensatz* [37] enthält als Trainingsmenge 60.000 handgeschriebene Ziffern in Graustufen mit je 28×28 Pixeln. Die Testmenge besteht aus 10.000 Ziffern im gleichen Format. Alle Ziffern sind nach Größe normalisiert und nach Schwerpunkt zentriert. Abbildung 2.15 (a) zeigt einige Beispielmuster der Trainingsmenge.

Der Datensatz wird in der vorliegenden Arbeit verwendet, um die grundsätzliche Funktionsweise des Objekterkennungssystems zu evaluieren. Er ist dafür gut geeignet, weil er zum einen relativ klein ist (und somit auch ohne Grafikkarte ein schnelles Training erlaubt) und zum anderen weil für ihn sehr viele Ergebnisse anderer Klassifizierungsmethoden verfügbar sind, die einen direkten Vergleich erlauben.

2.3.2 Der NORB-Datensatz

In [38] stellen LeCun et al. den *NORB-Datensatz* vor, der speziell zur Evaluation von allgemeinen Objekterkennungsverfahren erstellt wurde. Der Datensatz enthält Stereobilder von 50 verschiedenen Spielzeugen der fünf Kategorien Mensch, vierbeiniges Tier, Auto, LKW und Flugzeug. Jedes Objekt wurde jeweils aus mehreren unterschiedlichen Winkeln, Entfernungen und unter verschiedenen Lichtverhältnissen aufgenommen. Außerdem existieren verschiedene Varianten des Datensatzes. In der vorliegenden Arbeit wurde die Variante mit unveränderten Hintergründen verwendet, die jeweils aus 24.300 Trainings- und Testbildern besteht. Abbildung 2.15 (b) zeigt einige Beispielmuster.

Aufgrund der geringen Anzahl von Objektklassen und den unnatürlichen Bedingungen bei der Erstellung ist dieser Datensatz, ähnlich wie MNIST, für die Evaluation des in dieser Arbeit vorgestellten Objekterkennungssystems nicht ausreichend. Vorteilhaft ist jedoch, dass wie beim MNIST-Datensatz mehrere Ergebnisse anderer Klassifizierungsmethoden zum direkten Vergleich zur Verfügung stehen.

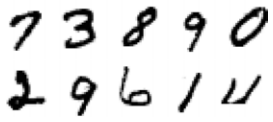
(a)  (b) 

Abbildung 2.15: Beispielmuster aus dem (a) MNIST- und (b) NORB-Datensatz. Quelle: [37] bzw. [38]

Abbildung 2.16: Ausschnitt aus dem Online-Annotationswerkzeug von LabelMe mit einem Beispielbild und mehreren darin markierten Objekten. Quelle: [43]

2.3.3 Der LabelMe-Datensatz

LabelMe [58] ist ein umfangreicher Bilddatensatz, der aus über 175.000 Fotos natürlicher Szenen besteht. Die Fotos unterteilen sich in 260 verschiedene Serien, zum Beispiel Straßenaufnahmen verschiedener Städte, Büro- und Universitätsgebäude sowie Landschaften.

LabelMe entstand unter der Leitung von Russel und Torralba am Massachusetts Institute of Technology (MIT). Die mit der Schaffung des Datensatzes verfolgten Ziele sind neben einer möglichst hohen Anzahl von Bildern insbesondere (siehe [58]):

- Eine große Anzahl verschiedener Objektklassen
- Viele unterschiedliche Instanzen pro Objektklasse
- Einbettung der Objekte in natürlicher Umgebung statt ausgeschnittener Objekte wie in einigen anderen Datensätzen
- Viele verschiedene natürliche Umgebungen.

Annotation und Format von LabelMe

Das Besondere am LabelMe-Datensatz ist, dass jeder an der Annotierung von Objekten mitwirken kann. Dazu steht ein Online-Annotationswerkzeug auf der Internetseite des Projekts [43] bereit, mit dem die Ränder von Objekten markiert werden können. Das so entstandene geschlossene Polygon wird dann zusammen mit einem frei wählbaren Objektnamen (zum Beispiel „person walking“) in einer zum jeweiligen Bild gehörenden Annotationsdatei im XML-Format gespeichert. Die Bilder selbst liegen im JPG-Format vor. Neben der Annotation bereits bestehender Fotos ist auch das Hochladen neuer Bildserien möglich.

Die meisten Bildserien enthalten ausschließlich hochaufgelöste Fotos mit mehr als einem Megapixel. Es gibt allerdings auch einige Serien, bei denen mehrere Bilder pro Sekunde mit einer Webcam aufgezeichnet wurden, so dass insgesamt mehrere tausend Bilder mit niedrigerer Auflösung (640×480 oder 320×240 Pixel) aus der gleichen Kameraposition entstanden sind.

Abbildung 2.17: Ausgeschnittene Beispielobjekte aus dem LabelMe-Datensatz. Quelle: [43]

Abbildung 2.16 zeigt einen Ausschnitt des Annotationswerkzeugs mit einem Beispielbild und mehreren annotierten Objekten. Abbildung 2.17 zeigt ausgeschnittene Objekte der Klassen „face“ (a), „car“ (b), „stopsign“ (c) und „streetlamp“ (d). Die Ausschnitte zeigen jeweils das umschließende Rechteck der Objekte (*Bounding Box*) ohne zusätzlichen Rand.

Auf jedem Bild des Datensatzes sind durchschnittlich ca. 1,8 Objekte annotiert. Insgesamt gibt es 4.418 veschiedene Objektklassen, von denen jedoch viele nur eine Instanz enthalten. Abbildung 2.18 zeigt deutlich, dass es nur wenige Klassen mit sehr vielen, aber sehr viele Klassen mit wenigen Instanzen gibt. Man beachte die logarithmische X- und Y-Skala! Tabelle 2.1 fasst die wichtigsten Eigenschaften des LabelMe-Datensatzes zusammen.

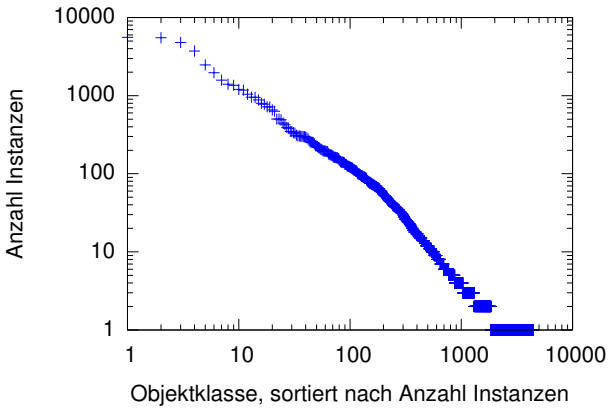


Abbildung 2.18: Anzahl Instanzen aller Objektklassen des LabelMe-Datensatzes

Die MATLAB-Toolbox zum Bearbeiten des Datensatzes

Für die Arbeit mit dem Datensatz haben die Autoren von LabelMe eine Sammlung von Skripten für das Mathematikprogramm MATLAB[1] veröffentlicht. Diese *MATLAB Toolbox* beinhaltet Funktionen unter anderem zum Durchsuchen des Datensatzes und zum Anzeigen, Ausschneiden sowie zur Annotation von Objekten.

[1] `http://www.mathworks.de/products/matlab/`

Eigenschaften des LabelMe-Datensatzes	
Anzahl Fotos	175.634
Davon annotiert (mindestens ein Objekt)	49.973
Anzahl Objekte	324.201
Anzahl Objektklassen	4.418
Anzahl Objektklassen mit mind. 50 Instanzen	215
Anzahl Objektklassen mit mind. 100 Instanzen	120
Durchschnittliche Auflösung der Fotos (horizontal × vertikal)	934×675 Pixel
Standardabweichung der Fotoauflösung	764×584 Pixel
Durchschnittliche Auflösung der Objekte (horizontal × vertikal)	247×219 Pixel
Standardabweichung der Objektauflösung	423×276 Pixel
Gesamtgröße	ca. 30 GB
Davon Annotationsdateien	ca. 183 MB

Tabelle 2.1: Eigenschaften des LabelMe-Datensatzes, Stand 24. Oktober 2008

2.3.4 Weitere Datensätze natürlicher Bilder

Neben LabelMe existieren noch andere Datensätze natürlicher Bilder, die sich als Trainings- und Testmengen für allgemeine Objekterkennung eignen. Dieser Abschnitt stellt einige dieser Datensätze vor und erläutert deren Gemeinsamkeiten und Unterschiede.

Caltech 101

Der *Caltech-101-Datensatz* [14] entstand im September 2003 am California Institute of Technology. Zusammengestellt wurde er von Li, Andreetto und Ranzato. Der Datensatz besteht aus 9.146 Bildern (Fotos, Zeichnungen und am Computer erstellte Grafiken) von Objekten aus insgesamt 101 Kategorien, zum Beispiel Delfine, Fußbälle und Sonnenblumen. Zusätzlich existiert eine Kategorie für Negativbeispiele („clutter“). Die Bilder wurden mit Hilfe einer Suchmaschine aus dem Internet gesammelt, unpassende Treffer wurden manuell aussortiert. Sie liegen im JPG-Format vor, zu jedem Bild existiert eine Annotationsdatei im MATLAB-Format. Letztere enthält sowohl die Ecken einer Bounding Box für das Objekt als auch eine genaue Umrandung in Form eines Polygons. Auf jedem Bild ist jeweils nur ein zentriertes Objekt enthalten. Zusätzlich haben alle Objekte gleicher Kategorie die gleiche seitliche Ausrichtung und Rotation, so dass zum Beispiel alle abgebildeten Krokodile den Kopf rechts und den Schwanz links auf dem Bild haben. Pro Kategorie existieren etwa 40 bis 800 Instanzen, die Bilder haben eine Auslösung von etwa 300×200 Pixeln. Die Annotationen können mit Hilfe eines MATLAB-Skriptes betrachtet werden. Der Datensatz wurde in mehreren Veröffentlichungen zum Thema Objekterkennung verwendet.

Caltech 256

Caltech 256 [21] ist der Nachfolger von Caltech 101 und wurde 2006 veröffentlicht. Der Datensatz besteht aus 256 Objektkategorien und ebenfalls einer zusätzlichen „clutter“-Kategorie. Insgesamt sind 30.607 Bildern enthalten, von jeder Kategorie sind es mindestens 80. Wie beim Vorgänger Caltech 101 wurden die Bilder mit Hilfe von Suchmaschinen zusammengetragen, jedoch wurde auf Anpassung der seitlichen Ausrichtung sowie der Rotation verzichtet. Beim direkten Vergleich in [21] liegen die Erkennungsraten bei Caltech 256 deutlich unter denen von Caltech 101. Von den Autoren wird Caltech 256 als größere Herausforderung für Objekterkennungssysteme gegenüber dem Vorgänger bezeichnet.

Abbildung 2.19: Beispielbilder der vorgestellten Bilddatensätze. (a) Caltech-101. Quelle: [15] (b) Caltech-256. Quelle: [21] (c) Pascal VOC 2008. Quelle: [13]

Pascal-Challenge

Die *Pascal Visual Object Classes Challenge* (*VOC*) [13] ist ein seit 2005 jährlich stattfindender Wettbewerb. Das Ziel ist die Erkennung von Objektklassen und -positionen in natürlichen Bildern, genauer gesagt besteht die Hauptaufgabe darin, zum einen eine korrekte Entscheidung zu treffen, ob ein Bild eine Instanz einer bestimmten Objektklasse enthält und zum anderen die Bestimmung der Objektgrenzen mit einer Bounding Box. Als weitergehende Herausforderung können Bilder pixelgenau segmentiert werden und einzelne Körperteile von Personen (Kopf, Hände und Füße) mit Bounding Boxes markiert werden.

Die VOC 2008 enthält 10.057 Bilder, die jeweils ein oder mehrere Objekte aus insgesamt 20 verschiedenen Objektklassen enthalten. Auch bei diesem Datensatz existiert zu jeder Bilddatei eine Annotationsdatei mit Angabe der Objektklasse(n) und der Bounding Box(es).

Weitere Datensätze

Neben den oben vorgestellten Datensätzen existieren noch viele weitere, die sich für das Training von Objekterkennungssystemen eignen. Die meisten enthalten jedoch deutlich weniger Bilder, Objektklassen oder Instanzen. Einige Datensätze wurden auch für spezielle Aufgaben wie zum Beispiel Gesichtserkennung erstellt und eignen sich daher weniger für allgemeine Objekterkennungsaufgaben. Eine Auswahl an Links zu weiteren Datensätzen ist im Internet unter [10] zu finden.

Vergleich der vorgestellten Datensätze natürlicher Bilder

In diesem Absatz werden die vorgestellten Datensätze natürlicher Bilder kurz verglichen und es wird begründet, warum für diese Arbeit der LabelMe-Datensatz ausgewählt wurde. Tabelle 2.2 zeigt noch einmal eine Zusammenfassung der Eigenschaften der oben vorgestellten Datensätze.

Datensatz	Anzahl Bilder	Anzahl Klassen	Annotationstyp
LabelMe	175.634	4418	Polygone
Caltech 101	9.146	102	Polygone
Caltech 256	30.607	257	Polygone
Pascal VOC 2008	10.057	20	Bounding Boxes

Tabelle 2.2: Übersicht der Eigenschaften der vorgestellten Bilddatensätze

Unabhängig von den in der Tabelle dargestellten Eigenschaften gibt es zwischen Caltech 101 und Caltech 256 auf der einen Seite und LabelMe und der Pascal VOC auf der anderen Seite mehrere wesentliche Unterschiede.

Bei ersteren ist pro Bild nur ein Objekt enthalten. Dieses ist üblicherweise ähnlich groß wie das gesamte Bild, so dass nur wenig Hintergrund beziehungsweise Umgebung des Objekts (im Folgenden auch *Kontext* genannt) sichtbar ist. Weiterhin sind Objekte oft ausgeschnitten und haben einen einfarbigen oder künstlich veränderten Hintergrund, weil sie größtenteils von Internetseiten stammen.

Bei LabelMe und Pascal hingegen sind die Objekte immer in einem natürlichen Kontext enthalten und unterscheiden sich zusätzlich in Größe (sowohl relativ zur Bildgröße als auch absolut) und Position im Bild.

Diese Unterschiede sprechen für LabelMe und die Pascal VOC als Trainings- und Testmengen für ein biologisch motiviertes Objekterkennungssystem, da dieses in der Lage sein soll, Objekte in ihrer natürlichen Umgebung möglichst gut zu erkennen. LabelMe im Speziellen bietet darüber hinaus den Vorteil, dass sehr viele Objektklassen existieren. Dadurch kann gut überprüft werden, ob das Objekterkennungssystem gut mit der Anzahl trainierter Klassen skaliert.

Auf der anderen Seite hat LabelMe auch einige Nachteile. Erstens ist die Qualität der Annotationen nur schwer überprüfbar, da jeder mit Hilfe des Online-Annotationstools selber Annotationen erstellen kann. Zweitens kann der Datensatz wegen der großen Vielseitigkeit (Objektgröße, -position, -bezeichnung und -kontext) unter Umständen zu schwierig für aktuelle Objekterkennungsysteme sein. Des Weiteren existieren bisher keine direkt vergleichbaren wissenschaftlichen Arbeiten, die diesen Datensatz als Trainings- und Testmenge benutzen und somit eine Beurteilung der erreichten Erkennungsleistung erlauben.

2.4 Das CUDA-Framework

In diesem Abschnitt wird das vom Grafikkartenhersteller *NVIDIA Corporation*[2] entwickelte und Anfang 2007 erstmals veröffentlichte *CUDA-Framework* (Compute Unified Device Architecture) beschrieben, mit dessen Hilfe Teile von selbst entwickelten Programmen auf CUDA-kompatiblen Grafikkarten von NVIDIA ausgeführt werden können. Dazu wird ein C/C++-Programm um spezielle Funktionen erweitert, die mit dem zu CUDA gehörenden Compiler kompiliert werden und bei Aufruf im Programm von der CUDA-Laufzeitbibliothek und dem Grafikkartentreiber automatisch auf die Grafikkarte kopiert und dort ausgeführt werden. Zusätzlich stehen etliche API-Funktionen zur Verfügung, mit denen zum Beispiel Daten vom Arbeitsspeicher des Computers in den Arbeitsspeicher der Grafikkarte und zurück kopiert werden können, um entweder die Eingabe für die Berechnung auf der Grafikkarte zur Verfügung zu stellen oder die Ausgabe zur weiteren Bearbeitung durch das Hauptprogramm zurück zu kopieren. Außerdem gibt es CUDA-Bibliotheken, die häufig vorkommende Berechnungen auf der Grafikkarte ausführen können. Eine dieser Bibliotheken ist *CUBLAS* (Compute Unified Basic Linear Algebra Subprograms). Diese bietet Funktionen für typische

[2] `http://www.nvidia.com`

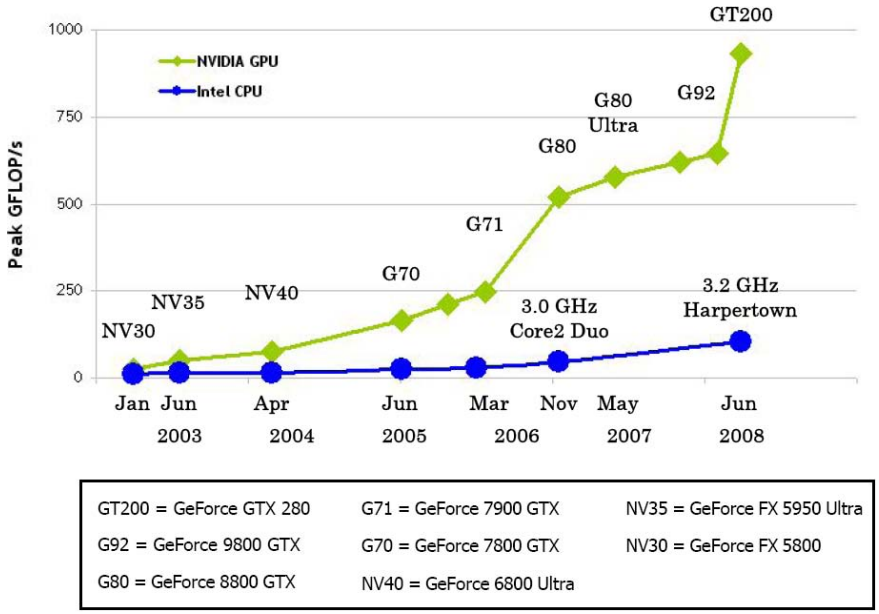


Abbildung 2.20: Entwicklung der Rechenleistung in GFLOPS von NVIDIA-GPUs gegenüber Intel-CPUs. Quelle: [47]

Aufgabenstellungen der linearen Algebra, zum Beispiel Matrixmultiplikation oder Berechnung eines Skalarproduktes.

Das CUDA-Framework wird in dieser Arbeit genutzt, um die Ausführungsgeschwindigkeit des verwendeten neuronalen Netzes wesentlich zu beschleunigen. Das dadurch angestrebte Ziel ist die Verarbeitung größerer Eingaben, größerer Netze (also höhere Anzahl Schichten beziehungsweise Gewichte) sowie Verarbeitung von mehr Trainingsbeispielen, als bei einem ohne CUDA-Unterstützung ausgeführten Programm möglich wäre. Die Begründung für diese Annahme ist die erheblich höhere theoretische Rechenleistung einer aktuellen CUDA-kompatiblen Grafikkarte im Vergleich zu einer aktuellen CPU. Dieser Sachverhalt ist in Abbildung 2.20 anhand der maximal möglichen GFLOPS (*Giga Floating Point Operations per Second*, also Fließkommaberechnungen pro Sekunde) und in Abbildung 2.21 anhand der maximalen Speicherbandbreite dargestellt. Die Abkürzung GPU in der Abbildung steht für *Graphics Processing Unit*, also den Prozessor der (CUDA-)Grafikkarte. Entsprechend steht CPU (*Central Processing Unit*) für den Hauptprozessor des Computers.

Der hohe GFLOPS-Wert der GPUs wird hauptsächlich durch den hohen Parallelisierungsgrad bei der Berechnung erreicht. Diese Möglichkeit zur Parallelisierung selbst entwickelter Programme wird in den folgenden Abschnitten näher beschrieben. In Abschnitt 2.4.1 werden zunächst die wichtigsten Begriffe der CUDA-Nomenklatur definiert. Im darauf folgenden Abschnitt 2.4.2 wird die Hardware CUDA-kompatibler Grafikkarten beschrieben, während Abschnitt 2.4.3 dann die Software-Umsetzung des Frameworks und die C/C++-Erweiterungen, mit deren Hilfe eigene CUDA-Programme entwickelt werden können, erläutert. Zum besseren Verständnis zeigt Abschnitt 2.4.4 ein kurzes CUDA-Beispielprogramm und erklärt den Quellcode. In Abschnitt 2.4.5 wird dann beschrieben, welche Richtlinien beim Programmieren mit CUDA eingehalten werden sollten, damit die resultierenden Programme die Hardware effizient ausnutzen und damit möglichst performant sind.

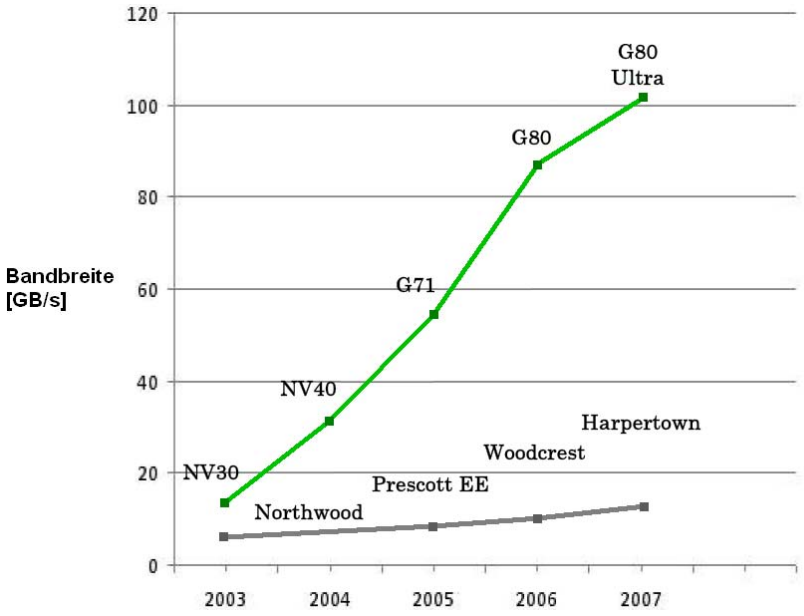


Abbildung 2.21: Entwicklung der Speicherbandbreite von NVIDIA-Grafikkarten gegenüber Intel-CPUs. Quelle: [47]

2.4.1 CUDA-Nomenklatur

In diesem Abschnitt werden die Hauptbegriffe des CUDA-Frameworks definiert. Dabei werden, wie auch in späteren Abschnitten, die englischen Originalbegriffe größtenteils beibehalten, solange die Übersetzung ins Deutsche nicht offensichtlich ist.

Device und Host

Eine vom CUDA-Framework verwendete Grafikkarte wird als *Device* bezeichnet. Der Computer, auf dessen CPU das Hauptprogramm läuft, wird *Host* genannt. Somit wird zum Beispiel der Hauptspeicher der Grafikkarte als Device-Speicher beziehungsweise Device-Memory bezeichnet, während der Hauptspeicher des Computers Host-Speicher beziehungsweise Host-Memory heißt.

Kernels, Threads, Blocks und Grids

In CUDA-Programmen können spezielle Funktionen definiert werden, die bei Aufruf auf die Grafikkarte kopiert und dort ausgeführt werden. Diese Funktionen werden *Kernels* genannt. Im Gegensatz zu normalen Funktionen kann ein Kernel massiv parallel ausgeführt werden. Die einzelnen Instanzen eines laufenden Kernels werden *Threads* genannt. Die Adressierung der Threads kann in bis zu drei Dimensionen erfolgen. Mehrere Threads werden in sogenannten *(Thread) Blocks* (oder auch *Blöcken*) zusammengefasst, die in bis zu zwei Dimensionen adressiert werden können. Jeder Thread Block enthält die gleiche Anzahl Threads. Alle Blocks zusammen, also alle Instanzen des Kernels, werden als *Grid* bezeichnet. Jeder Thread kann zur Laufzeit seinen Threadindex und seinen Blockindex sowie die Gesamtgröße des Blocks und des Grids als drei- beziehungsweise zweidimensionalen Vektor abfragen. Mit diesen Indizes wird dann üblicherweise festgelegt, welche Datenelemente ein Thread bearbeiten soll. Die Aufteilung von Kernel-Instanzen in Threads und Blocks ist in Abbildung 2.22 dargestellt.

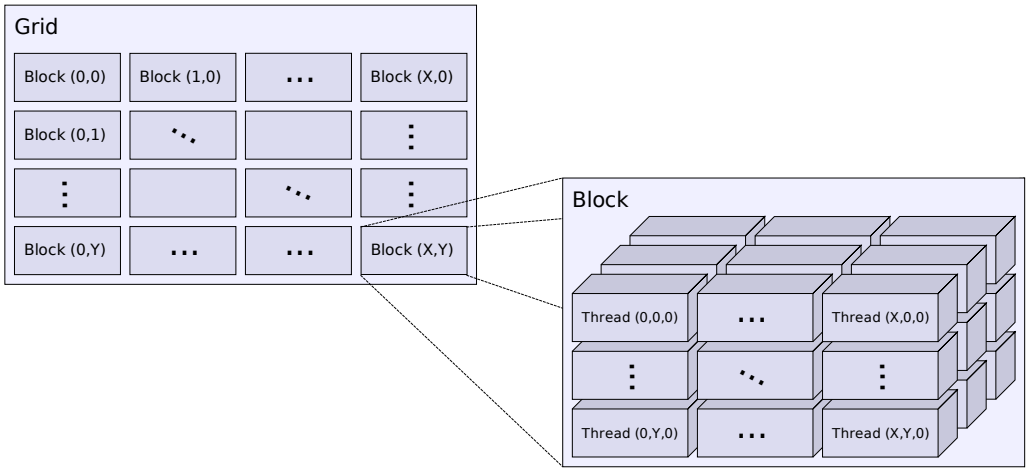


Abbildung 2.22: Aufteilung von Kernel-Instanzen in Threads und Blocks

2.4.2 Hardware-Architektur

NVIDIA-Grafikkarten wurden jahrelang auf ihre ursprüngliche Hauptaufgabe optimiert – die schnelle Berechnung aufwändiger 3D-Grafiken. Aus diesem Grund ist die Hardware speziell auf massive Parallelisierung und einen hohen GFLOPS-Wert ausgelegt. Durch die Einführung von CUDA stehen diese Fähigkeiten nun zur Berechnung beliebiger Probleme zur Verfügung. Die wesentlichen Elemente der Grafikkarten-Hardware werden nachfolgend erläutert.

Multiprozessoren und skalare Prozessoren

Das Kernstück einer CUDA-kompatiblen Grafikkarte bildet die GPU. Sie gliedert sich in mehrere identisch aufgebaute *Multiprozessoren*. Jeder der Multiprozessoren besitzt die gleiche Anzahl an *skalaren Prozessoren* (im Folgenden meist einfach *Prozessor* genannt) sowie mehrere Arten von lokalem, schnellem Speicher beziehungsweise Cache. Jeder Thread eines Kernels wird auf genau einem Prozessor ausgeführt. Alle Threads eines Thread Blocks werden auf einem bestimmten Multiprozessor ausgeführt. Enthält der Thread Block mehr Kernels als der Multiprozessor Prozessoren besitzt, werden die Threads nacheinander auf den zur Verfügung stehenden Prozessoren des Multiprozessors ausgeführt. Anders herum können bei ausreichenden Hardware-Ressourcen mehrere Thread Blocks gleichzeitig auf einem Multiprozessor ausgeführt werden, ansonsten werden sie auf alle zur Verfügung stehenden Multiprozessoren aufgeteilt und gegebenenfalls nacheinander ausgeführt. Durch diese Vorgehensweise ist die Ausführung von CUDA-Programmen auf unterschiedlichen Grafikkarten-Modellen möglich, die sich in der Anzahl der Multiprozessoren unterscheiden. Als Beispiel besitzt die NVIDIA GTX 280 30 Multiprozessoren, während es bei der GTX 260 je nach Modell 24 oder 27 sind.

SIMT-Architektur

Das beschriebene Prinzip nennt NVIDIA *SIMT* (*Single Instruction, Multiple Thread*). Im Gegensatz zur bekannten SIMD-Architektur (Single Instruction, Multiple Data) soll damit hervorgehoben werden, dass jeder Thread unabhängig von den anderen ausgeführt wird, das heißt, er besitzt einen eigenen Instruktionszeiger und eigene Register. Somit können verschiedene Threads verschiedene Anweisungen ausführen.

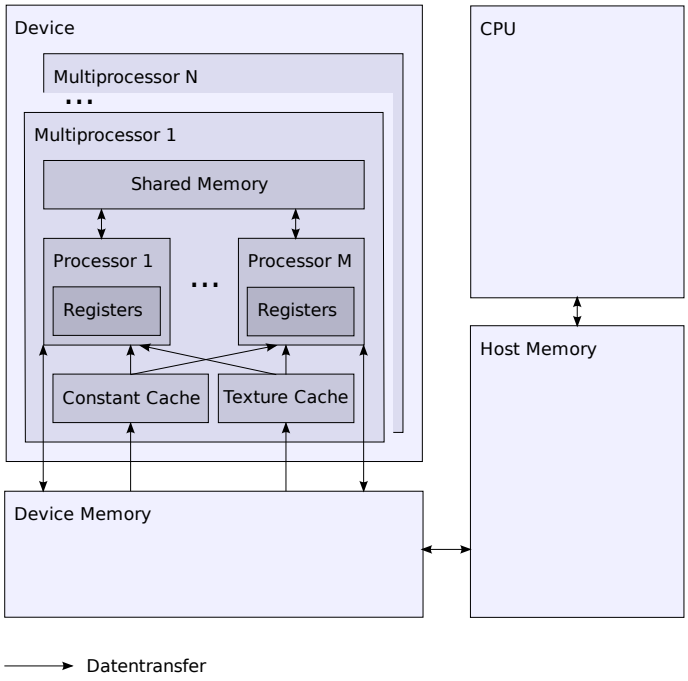


Abbildung 2.23: Hardware-Struktur von CUDA-kompatiblen Grafikkarten

Speicher und Caches

Zum Laden und Speichern von Daten stehen verschiedene Arten von Speicher sowie Caches zur Verfügung. Grundsätzlich wird zwischen*On-Chip-Speicher* der Multiprozessoren und *globalem Speicher* unterschieden.

Der globale Speicher, auch Device-Speicher oder Device-Memory genannt, ist der größte Speicher der Grafikkarte und vergleichbar mit dem Arbeitsspeicher eines Computers. Er ist nicht Teil der GPU und weist deshalb eine deutlich höhere Latenz beim Lesen und Schreiben von Daten gegenüber dem On-Chip-Speicher der Multiprozessoren auf. Außerdem müssen gewisse Zugriffsmuster eingehalten werden, um die maximale Speicherbandbreite auszunutzen (siehe dazu Abschnitt 2.4.5). Zwischen verschiedenen Thread Blocks eines Grids kann ausschließlich über globalen Speicher kommuniziert werden, da wie im vorherigen Abschnitt beschrieben nicht garantiert werden kann, dass bestimmte Thread Blocks auf bestimmten Multiprozessoren ausgeführt werden.

Im Gegensatz zum globalen Speicher sind die On-Chip-Speicher feste Bestandteile der Multiprozessoren. Jeder Multiprozessor verfügt über folgende Speicher und Caches:

- *Register*. Register werden unter allen gleichzeitig auf einem Multiprozessor ausgeführten Threads aufgeteilt. Auf sie kann ausschließlich vom ihnen zugewiesenen Thread zugegriffen werden. In Registern werden üblicherweise die meisten lokalen Variablen eines Threads abgelegt.

- *Shared Memory*. Dieser Speicherbereich dient hauptsächlich zum schnellen Datenaustausch zwischen verschiedenen Threads eines Thread Blocks. Da wie oben beschrieben garantiert ist, dass alle Threads eines Thread Blocks auf dem gleichen Multiprozessor ausgeführt werden, können sie alle auf den gleichen Shared-Memory-Bereich zugreifen. Threads aus unterschiedlichen Thread Blocks können aus oben genannten Gründen grundsätzlich nicht über Shared Memory kommunizieren.

- *Konstantencache.* Im globalen Speicher der Grafikkarte existiert ein dedizierter Bereich für die Speicherung konstanter Daten (der *Konstantenspeicher*), der nur vom Host aus beschrieben werden kann. Bei Abruf dieser Daten durch einen Thread dient der Konstantencache als Zwischenspeicher und bietet somit eine geringere Latenz, falls Daten innerhalb eines Thread Blocks mehrmals abgerufen werden.

- *Texturcache.* Neben dem direkten Lesezugriff auf den globalen Speicher können sogenannte *Texturen* definiert werden, die für bestimmte Zugriffsmuster mehrere Vorteile gegenüber dem direkten Speicherzugriff aufweisen. Einer dieser Vorteile ist die Zwischenspeicherung (Caching) gelesener Daten, für die der Texturcache zur Verfügung steht. Neben der geringeren Latenz bei mehrfachen Zugriffen auf das gleiche Element innerhalb eines Thread Blocks werden außerdem auch benachbarte Datenelemente aus dem globalen Speicher automatisch in den Cache geladen. Dadurch kann die Gesamtlatenz bei der Ausführung eines Thread Blocks zum Teil deutlich reduziert werden. Welche Datenelemente als benachbart gelten, hängt von der Definition der Textur ab, ist allerdings kaum dokumentiert und muss daher experimentell ermittelt werden.

Abbildung 2.23 zeigt die beschriebene Hardware-Struktur als Diagramm. In Tabelle 2.3 sind konkrete technische Daten für das Grafikkartenmodell NVIDIA GeForce GTX 285 angegeben, um eine bessere Vorstellung der genannten Begriffe zu vermitteln.

2.4.3 Software-Architektur und C/C++-Spracherweiterungen

CUDA-Programme können zum aktuellen Zeitpunkt nur mit den Programmiersprachen C und C++ entwickelt werden. Kernel-Funktionen können ausschließlich in C geschrieben werden. Zukünftig sollen allerdings auch weitere Programmiersprachen wie Fortran, OpenCL und Direct3D 11 Compute unterstützt werden [47].

Um die Programmierspache C/C++ um die Fähigkeiten von CUDA zu erweitern, wurden einige CUDA-spezifische Syntaxerweiterungen und neue Schlüsselwörter eingeführt, die später in diesem Abschnitt beschrieben werden.

Kompilierung mit NVCC

Für die Kompilierung des gesamten C/C++-Programms inklusive CUDA-Code kann der NVIDIA-Compilertreiber *NVCC* verwendet werden. Dieser trennt automatisch den Host-Code (der auf dem Host ausgeführt werden soll) vom Device-Code (der auf der Grafikkarte ausgeführt werden soll) und ruft für den Host-Code dann den Standard-C/C++-Compiler auf. Der Device-Code wird mit Hilfe von weiteren NVIDIA-Tools zuerst in einen Assembler-ähnlichen Code im Textformat übersetzt (sogenannter *PTX-Code*) und dann in Maschinencode für die Grafikkarte kompiliert. Anschließend ruft NVCC einen Linker auf, so dass die Ausgabe eine reguläre ausführbare Datei ist, die auf dem Host läuft und den Device-Code enthält. Durch die Verwendung von NVCC können CUDA-Programme also wie herkömmliche C/C++-Programme entwickelt und kompiliert werden; Kernel-Funktionen und normaler C/C++-Code können in einer Quellcodedatei gemischt enthalten sein.

CUDA-spezifische C/C++-Spacherweiterungen

CUDA-Programme besitzen eine besondere Syntax zur Definition und zum Aufruf von Kernel-Funktionen.

Eine Kernel-Funktion wird bei der Definition mit dem Schlüsselwort **__global__** gekennzeichnet und hat immer den Rückgabewert **void**:

Technische Daten der NVIDIA GeForce GTX 285	
Anzahl Multiprozessoren	30
Anzahl Prozessoren pro Multiprozessor	8
Prozessor-Takt	1476 MHz
Globaler Speicher	1 GB
Takt des globalen Speichers	1242 MHz
Shared Memory pro Multiprozessor	16.384 Byte
Anzahl Register pro Multiprozessor (je 4 Byte)	16.384
Maximale Speicherbandbreite (globaler Speicher)	159 GB/s
Maximale Anzahl Threads pro Block	512

Tabelle 2.3: Technische Daten der NVIDIA-Grafikkarte GeForce GTX 285. Quelle: [48] und Augabe des CUDA-SDK-Programms deviceQuery

```
__global__ void myKernel(/*Parameters*/)
{
    // Kernel code
}
```

Beim Aufruf eines Kernels wird eine CUDA-spezifische Syntax mit drei spitzen Klammern verwendet, um zusätzliche Parameter zu übergeben:

```
myKernel<<<gridDim, blockDim, sharedMemSize, stream>>>(/*Params*/);
```

gridDim und blockDim geben an, wie oft der Kernel parallel gestartet werden soll. Dabei ist gridDim ein zweidimensionaler Vektor, der die Anzahl Blöcke angibt und blockDim ein dreidimensionaler Vektor, der die Anzahl Threads pro Block angibt (vergleiche Abbildung 2.22).

Mit sharedMemSize kann optional angegeben werden, wie viel Shared Memory pro Block dynamisch alloziert wird. Dies ist vorteilhaft, wenn erst zur Ausführungszeit bekannt ist, wie viele Daten jeder Block verarbeiten soll. Ebenfalls optional ist der Parameter stream, der verwendet werden kann, um mehrere Aktionen auf der Grafikkarte gleichzeitig auszuführen. So kann zum Beispiel eine Kopie von einem Speicherbereich in einen anderen stattfinden, während ein Kernel ausgeführt wird. Diese Parallelisierung passiert aber nur dann, wenn die entsprechenden API- beziehungsweise Kernel-Aufrufe mit unterschiedlichen Stream-Instanzen aufgerufen wurden. Wird der Parameter stream nicht angegeben, werden alle API- und Kernelaufrufe synchron verarbeitet, was im Normalfall auch für die Korrektheit des Programms erforderlich ist.

Neben den gerade beschriebenen Spracherweiterungen zur Definition und zum Aufruf von Kernels existieren CUDA-spezifische Schlüsselwörter zur Angabe des Speicherbereichs einer Variablen. Mit **__device__** wird eine Variable deklariert, die im globalen Speicher abgelegt ist, **__constant__** deklariert Variablen im Konstantenspeicher und **__shared__** deklariert Variablen im Shared Memory eines Thread Blocks. Beispiel:

```
__device__ float myFloatOnDevice;
```

Mit **__device__** können außerdem Funktionen definiert werden, die auf der Grafikkarte ausgeführt werden. Diese sind allerdings im Gegensatz zu Kernels nicht direkt vom Host aufrufbar, sondern nur aus Kernels. Als **__shared__** können Variablen aus offensichtlichen Gründen nur innerhalb von Kernels deklariert werden.

CUDA-API

Zusätzlich zu den Spracherweiterungen für Kernels und Speicherallokation gibt es eine Fülle von CUDA-spezifischen API-Funktionen, die verschiedenste Aufgaben erfüllen. Aus diesem Grund soll an dieser Stelle nur ein Beispiel genannt werden, nämlich die sehr häufig verwendete API-Funktion zum Kopieren von Speicherbereichen:

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum
    cudaMemcpyKind kind, cudaStream_t stream);
```

Ähnlich wie bei der C-Funktion memcpy wird hier mit dst die Zieladresse, mit src die Quelladresse und mit count die Anzahl zu kopierender Bytes angegeben. Zusätzlich wird mit kind angegeben, zu welchen Speicherbereichen dst und src gehören. Es kann von globalem Speicher in Host-Speicher, von Host-Speicher in globalen Speicher und von globalem Speicher in globalen Speicher kopiert werden. Der Parameter stream verhält sich wie bereits weiter oben beschrieben. Als Rückgabewert erhält man CUDA_SUCCESS bei Erfolg, andernfalls einen entsprechenden Fehlercode. Der Aufruf ist synchron.

Ausführung und Debugging

CUDA-Programme können auf jedem System mit mindestens einer CUDA-kompatiblen Grafikkarte ausgeführt werden. Es ist nicht erforderlich, das Programm für unterschiedliche Grafikkartenmodelle neu zu kompilieren.

Wird vom Hauptprogramm ein Kernel gestartet, so wird dieser grundsätzlich asynchron ausgeführt, das heißt das Hauptprogramm läuft weiter, auch wenn die Ausführung des Kernels noch nicht beendet ist. Mit der API-Funktion cudaThreadSynchronize kann allerdings jederzeit eine Synchronisierung durchgeführt werden.

Debugging von CUDA-Programmen ist seit Version 2.1 unter Linux mit dem Debugger gdb möglich. Außerdem lassen sich mit dem *CUDA-Profiler* statistische Daten über Ausführungszeit, Speicherzugriffe und Performanz von Kernel- und API-Aufrufen nachvollziehen.

2.4.4 Ein kurzes CUDA-Beispielprogramm

Listing 1 zeigt ein komplettes, lauffähiges CUDA-Beispielprogramm. Das Programm alloziert ein Array mit 1024 Elementen vom Typ **int** auf dem Host und auf dem Device. Dann wird ein Kernel mit zwei Thread Blocks mit jeweils 32×16 Threads ausgeführt, also insgesamt 1024 Threads. Im Kernel wird der Index des aktuellen Threads so berechnet, dass alle Indizes von 0 bis 1023 während der Ausführung genau einmal vorkommen. Dazu werden die Variablen blockDim, blockIdx und threadIdx verwendet, die innerhalb von Kernels immer zur Verfügung stehen und Auskunft über die Größe der Thread Blocks, den Index des aktuellen Thread Blocks und den Index des aktuellen Threads innerhalb des Thread Blocks geben. Dann wird der Wert jedes Array-Elements auf seinen Index gesetzt.

Nach dem Kernelaufruf muss ein Aufruf der CUDA-API-Funktion cudaThreadSynchronize erfolgen, da Kernels wie bereits beschrieben asynchron gestartet werden. Nach dem Aufruf ist sichergestellt, dass der Kernel beendet ist und alle Schreiboperationen auf dem globalen Speicher ausgeführt wurden.

Abschließend wird das vom Kernel geschriebene Array auf den Host kopiert und dort ausgegeben. Als Ausgabe ergibt sich „0 1 2 3 4 5 ... 1023“.

2.4.5 Richtlinien für performante CUDA-Programme

Die Entwicklung von CUDA-Programmen erfordert im Vergleich zur Entwicklung von herkömmlichen Programmen eine verstärkte Auseinandersetzung mit der verwendeten Hardware. Um eine Performanz nahe am theoretischen Maximum der Rechenleistung oder der

Listing 1 Ein kurzes CUDA-Beispielprogramm

```
1  #include <cutil.h>
2  #include <stdio.h>
3
4  __global__ void myKernel(int* array)
5  {
6      int index =
7          (blockDim.y * blockDim.x * blockIdx.x) +
8          (blockDim.x * threadIdx.y) +
9          threadIdx.x;
10
11     array[index] = index;
12 }
13
14 int main(int argc, char* argv[])
15 {
16     // Allocate array on host
17     int myArray[1024];
18
19     // Allocate array on device
20     int* myArrayGPU;
21     cudaMalloc((void**)&myArrayGPU, 1024*sizeof(int));
22
23     // Fill array on device (32*16*2=1024 threads in total)
24     dim3 blockDim(32,16);
25     dim3 gridDim(2);
26     myKernel<<<gridDim,blockDim>>>(myArrayGPU);
27     cudaThreadSynchronize();
28
29     // Copy array from device to host
30     cudaMemcpy(myArray, myArrayGPU, 1024*sizeof(int),
           cudaMemcpyDeviceToHost);
31
32     // Print array
33     for (int i=0; i<1024; i++)
34         printf("%d ", myArray[i]);
35
36     return 0;
37 }
```

Speicherbandbreite der Grafikkarte zu erreichen, müssen die Möglichkeiten und Beschränkungen der verschiedenen Hardwarekomponenten bedacht und Flaschenhälse bei der Programmierung vermieden werden.

Dieser Abschnitt beschreibt die wichtigsten Richtlinien zur Entwicklung performanter CUDA-Programme. Aus Platzgründen können allerdings nicht alle relevanten Richtlinien an dieser Stelle genannt werden. Spezielle Fragestellungen werden deshalb erst für konkrete Situationen in späteren Kapiteln dieser Arbeit behandelt.

Divergent Branches

Obwohl aktuelle CUDA-Grafikkarten nur 8 Skalarprozessoren pro Multiprozessor besitzen und somit nur 8 Threads echt parallel ausführen können, gelten einige spezielle Bedingungen für größere Gruppen von Threads. Die Instruktionseinheit der Multiprozessoren ist auf aufeinanderfolgende Gruppen von je 32 Threads optimiert. Eine solche Gruppe wird *Warp* genannt. Dementsprechend ist ein *Half-Warp* eine Gruppe von 16 aufeinanderfolgenden Threads.

Innerhalb eines Warps wird zu einem bestimmten Zeitpunkt immer nur eine bestimmte Anweisung ausgeführt. Bei Abzweigungen („if-then“- bzw. „if-then-else“-Konstrukte) in Kernels sollte deshalb vermieden werden, dass die Threads eines Warps unterschiedliche Ausführungspfade haben, da sonst die Laufzeit des Warps der Summe aller vorkommenden Abzweigungen entspricht. Diese Situation wird als *Divergent Branch* bezeichnet. Durch Verwendung des CUDA-Profilers kann leicht überprüft werden, ob Divergent Branches während der Ausführung eines Kernels aufgetreten sind.

Register, Shared Memory und Local Memory

Als schneller On-Chip-Speicher stehen auf den Multiprozessoren Register und Shared Memory zur Verfügung. Falls Daten nur innerhalb eines Threads benötigt werden, sind Register gegenüber Shared Memory zu bevorzugen, da keine bestimmten Zugriffsmuster eingehalten werden müssen. Der Shared Memory ist in 16 *Bänke* unterteilt, wobei jeweils das n-te Byte des Shared Memory zur (n mod 16)-ten Bank gehört. Jede Bank kann pro Anweisung innerhalb eines Half-Warps nur eine Schreib- oder Leseoperation ausführen. Solange innerhalb eines Half-Warps auf jede Bank höchstens einmal zugegriffen wird, ist das Lesen oder Schreiben daher verzögerungsfrei möglich. Andernfalls wird automatisch eine Serialisierung ausgeführt, das heißt die Schreib-/Leseoperationen werden nacheinander ausgeführt, wodurch sich die Ausführungszeit des Half-Warps verlängert. Eine Ausnahme ist gegeben, wenn alle Threads eines Half-Warps auf die gleich Bank zugreifen. Für diesem Fall existiert ein Broadcast-Mechanismus, so dass sich die Ausführungszeit nicht erhöht. Das Auftreten von Serialisierungen während der Kernel-Ausführung lässt sich mit dem CUDA-Profiler überprüfen.

Wenn ein Kernel mehr Shared Memory anfordert als die Hardware der Multiprozessoren erlaubt, so schlägt der Start des Kernels fehl. Wenn ein Kernel hingegen mehr als die zur Verfügung stehende Anzahl Register pro Multiprozessor benötigen würde, werden vom Compiler automatisch so viele Variablen in den globalen Speicher ausgelagert, dass genau die maximale Anzahl Register ausgenutzt wird. Die Speicherbereiche des globalen Speichers, in dem sich die ausgelagerten lokalen Variablen befinden, werden *Local Memory* genannt. Da der globale Speicher eine drastisch höhere Latenz als der On-Chip-Speicher aufweist, sollte die Auslagerung von Registern im Allgemeinen vermieden werden. Mit der NVCC-Option `--ptxas-options=-v` lässt sich die Auslagerung meist schnell erkennen. In Einzelfällen kann diese aber auch in späteren Phasen der Kompilierung erfolgen, was wiederrum mit dem CUDA-Profiler überprüft werden kann.

Globaler Speicher

Der globale Speicher ist der Hauptspeicher der Grafikkarte und kann, anders als die oben beschriebenen On-Chip-Speicher, zum Datenaustausch mit dem Host genutzt werden. Er weist allerdings eine um etwa Faktor 100 höhere Latenz auf. Deshalb sollten die On-Chip-Speicher möglichst effektiv genutzt werden. Ein typisches Muster bei der Entwicklung von Kernels ist es, am Anfang Daten aus dem globalen Speicher in Register oder in den Shared Memory zu laden, dort zu bearbeiten und am Ende wieder in den globalen Speicher zu schreiben. Dadurch werden Zugriffe auf den globalen Speicher minimiert. Ähnlich wie beim Shared Memory müssen auch beim globalen Speicher bestimmte Zugriffsmuster eingehalten werden, um die theoretische Speicherbandbreite auszunutzen. Dafür ist es entscheidend, ob gleichzeitige Lese-/Schreibzugriffe der Threads eines Half-Warps *zusammenhängend* (englisch: *coalesced*) durchgeführt werden können. Zusammenhängende Zugriffe werden gemeinsam in einer *Speichertransaktion* durchgeführt, während nicht zusammenhängende Zugriffe je eine eigene Transaktion erfordern. Da eine Transaktion unabhängig von der Anzahl übertragener Daten immer die gleiche Latenz aufweist, ergeben sich deutliche Geschwindigkeitsunterschiede zwischen zusammenhängenden und nicht zusammenhägenden Zugriffen (teilweise mehr als Faktor 10). Die Speicherzugriffe einer Anweisung in allen Threads eines Half-Warps sind unter folgenden Bedingungen zusammenhängend:

1. Beliebig viele Threads des Half-Warps müssen auf je ein Datum der gleichen Größe entweder alle lesend oder alle schreibend zugreifen. Zulässige Größen sind 32, 64 und 128 Bit.

2. Die Daten müssen alle im gleichen *Segment* des globalen Speichers liegen. Die Größe eines Segments beträgt 32 Byte bei Zugriff auf 8-Bit-Daten, 64 Byte bei Zugriff auf 16-Bit-Daten und 128 Byte bei Zugriff auf 32- oder 64-Bit-Daten. Ein Segment beginnt immer bei einer Speicheradresse, die ein Vielfaches der Segmentgröße ist.

Für ältere CUDA-Grafikkarten (vor dem GT200-Chipsatz) gelten strengere Bedingungen für zusammenhängende Speicherugriffe, die bei Interesse in [47], Abschnitt 5.1.2.1, nachgelesen werden können. Zugriffe, die nicht in der gleichen Anweisung innerhalb eines Half-Warps auftreten, sind grundsätzlich nicht zusammenhängend.

Konstantenspeicher und Texturspeicher

Auf Seite 27 wurden bereits die Vorzüge von Konstantenspeicher und Texturspeicher gegenüber direkten Zugriffen auf den globalen Speicher erläutert. Insbesondere der Texturspeicher kann auf sehr einfache Weise die Performanz bei Lesezugriffen deutlich erhöhen, da jeder beliebige Bereich im globalen Speicher mit nur einem API-Aufruf als Textur deklariert werden kann. Insbesondere bei nicht zusammenhängenden Zugriffen sollte der Texturspeicher in Betracht gezogen werden, aber selbst bei zusammenhängenden Zugriffen erreicht er in der Praxis oft eine höhere Geschwindigkeit. Problematisch kann allerdings die Verwendung von mehreren Texturreferenzen in einem Kernel sein. In der Praxis hat sich gezeigt, dass die Ausführungsgeschwindigkeit eines Kernels bei Verwendung von zwei Texturen teils langsamer ist als bei der Verwendung von nur einer Textur und direktem Zugriff auf den zweiten Speicherbereich. Dieser Fall trat sogar bei nicht zusammenhängenden Zugriffen auf. Vermutlich wird der Texturcache also teilweise nicht optimal genutzt.

Der Konstantenspeicher ist in der Praxis meist weniger vorteilhaft als das Deklarieren von Texturen, da er lediglich 64 kB groß ist. In diesen dedizierten Bereich des globalen Speichers müssen konstante Daten explizit per API-Aufruf kopiert werden, was längere Zeit in Anspruch nimmt als das Deklarieren einer Textur, deren Daten bereits im globalen Speicher liegen. Falls allerdings bereits eine Textur im Kernel verwendet wird, kann wie im oben

genannten Fall die zusätzliche Benutzung des Konstantenspeichers schneller als die Verwendung einer weiteren Textur oder der direkte Zugriff auf den globalen Speicher sein.

Datenaustausch zwischen Host und Device

Die Speicherbandbreite beim Kopieren von Host zu Device beziehungsweise andersherum liegt deutlich unter der Speicherbandbreite des globalen Speichers der Grafikkarte, wodurch an dieser Stelle leicht Flaschenhälse entstehen können. Mit dem zum CUDA-SDK gehörenden Programm *bandwidthTest* kann die Bandbreite direkt verglichen werden. Mit den Standardoptionen ergibt sich auf dem in dieser Arbeit verwendeten System beim Kopieren von Host zu Device eine Bandbreite von etwa 5,2 GB/s, andersherum etwa 4,4 GB/s. Kopien von Device zu Device haben hingegen eine Bandbreite von etwa 118,0 GB/s und sind damit um mehr als Faktor 20 schneller.

Eine Verbesserung der Host-Device-Bandbreite lässt sich erreichen, wenn der verwendete Hostspeicher als *pinned* deklariert wird. Er kann dann nicht ausgelagert werden und befindet sich an einer festen Adresse im Host-Arbeitsspeicher, wodurch er schneller adressiert werden kann. Mit Pinned-Speicher ergibt sich eine Host-zu-Device-Bandbreite von 5,6 GB/s und eine Device-zu-Host-Bandbreite von 5,5 GB/s. Diese Messungen wurden mit den Optionen `--memory=pageable` und `--memory=pinned` durchgeführt.

Effiziente Ausnutzung von Multiprozessoren

Um eine Performanz nahe der theoretischen Maximalleistung zu erreichen, sollten alle Rechenwerke möglichst durchgängig ausgelastet sein.

Zum einen heißt das, dass pro Kernel genügend Thread Blocks gestartet werden müssen, so dass alle Multiprozessoren der Grafikkarte gleichzeitig genutzt werden. Die Anzahl der Thread Blocks kann extrem groß werden und stellt deshalb kaum eine Einschränkung dar. Bei der in dieser Arbeit verwendeten GeForce GTX 285 beträgt die maximale Gridgröße zum Beispiel 65.535×65.535 Thread Blocks.

Zum anderen müssen die einzelnen Multiprozessoren selbst effizient ausgelastet werden, so dass keine Flaschenhälse durch zu geringe Hardware-Ressourcen entstehen. Aus diesem Grund sollten möglichst viele Threads parallel auf einem Multiprozessor ausgeführt werden. Durch eine hohe Anzahl Threads kann die Hardware Latenzen beim Zugriff auf langsame Speicher gut verstecken, indem während der Wartezeit bereits Anweisungen anderer Threads ausgeführt werden. Die Threads müssen dabei nicht unbedingt zu einem Thread Block gehören, da bei ausreichenden Hardware-Ressourcen auch mehrere Thread Blocks pro Multiprozessor parallel laufen können. Die hohe Anzahl Threads ist nur erreichbar, wenn pro Thread nicht zu viele Register benötigt werden. Außerdem können mehrere Thread Blocks nur dann gleichzeitig auf einem Multiprozessor ausgeführt werden, wenn sie zusammen nicht mehr als den zur Verfügung stehenden Shared Memory benötigen.

Um die Auslastung (englisch: *Occupancy*) der Multiprozessoren zu berechnen, bietet NVIDIA den *CUDA Occupancy Calculator* [46] an. Dabei handelt es sich um eine Tabelle im Microsoft-Excel-Format, die die technischen Daten CUDA-kompatibler Grafikkarten enthält und Flaschenhälse bei Eingabe der benötigten Kernel-Ressourcen visualisiert.

Abwägung der genannten Faktoren

Die in diesem Abschnitt genannten Anforderungen zur Maximierung der Ausführungsgeschwindigkeit können meistens nicht alle optimal erfüllt werden. Oft müssen Kompromisse gemacht werden, wobei in der Praxis häufig erst durch Messungen die beste Alternative bestimmt werden kann. Im später folgenden Kapitel über die Implementierung wird konkret auf einige dieser Fälle eingegangen.

3 Verwandte Arbeiten

Im vorhergehenden Kapitel wurden bereits einige Modelle neuronaler Netze eingeführt. Dieses Kapitel stellt nun speziellere und größtenteils aktuellere Veröffentlichungen vor, die sich – zumindest teilweise – mit ähnlichen Problemstellungen wie die vorliegende Arbeit beschäftigen.

Das Kapitel ist in vier Abschnitte unterteilt: In 3.1 werden Arbeiten vorgestellt, die „State of the Art"-Ergebnisse im Bereich der Handschrifterkennung erreicht haben. Abschnitt 3.2 behandelt analog dazu die Objekterkennung in natürlichen Bildern. Abschnitt 3.3 stellt Arbeiten vor, in denen neuronale Netze auf Grafikkarten implementiert wurden. Bei diesen Arbeiten liegt der Fokus auf einer hohen Ausführungsgeschwindigkeit; insbesondere sind dort keine neuartigen Ansätze oder „State of the Art"-Ergebnisse zu finden. In Abschnitt 3.4 werden die Ergebnisse der vorgestellten Arbeiten diskutiert.

3.1 Erkennung von handgeschriebenen Ziffern

Die automatische Erkennung handgeschriebener Zeichen oder Ziffern ist eine äußerst praxisrelevante Anwendung für neuronale Netze. So können zum Beispiel die Adressen auf Briefen oder der Inhalt von Überweisungsträgern automatisch in hoher Geschwindigkeit verarbeitet werden. Da diese Anwendung allerdings nicht den Schwerpunkt der vorliegenden Arbeit darstellt, werden hier nur kurz die wichtigsten Vorgehensweisen und Ergebnisse beschrieben.

LeCun et al. nutzen in [36] das bereits im Grundlagenkapitel vorgestellte Konvolutionsnetz LeNet in verschiedenen Varianten zur Erkennung handgeschriebener Ziffern. Als Trainings- und Testmenge dient der MNIST-Datensatz.

In der Arbeit wird das Konvolutionsnetz systematisch mit anderen Klassifizierungsansätzen verglichen, darunter lineare Klassifizierung (entspricht einem einschichtigen Perzeptron mit Identität als Aktivierungsfunktion), k-Nearest-Neighbor [11], drei- und vierschichtige Perzeptrons und Support Vector Machines (SVMs) [5; 63]. Das beste Ergebnis erzielt ein modifiziertes LeNet4-Konvolutionsnetz, welches nur geringe Unterschiede zu LeNet5 aufweist. Der Prozentsatz falsch erkannter Muster auf der Testmenge (*Testfehler*) nach dem Training mit der Trainingsmenge beträgt 0,7%. Mit mehrschichtigen Perzeptrons, wie sie in Abschnitt 2.2.2 beschrieben sind, wurde ohne besondere Anpassungen ein Testfehler von 2,95% erreicht.

Simard et al. [61] konnten den Testfehler durch Vergrößerung der Trainingsmenge mit sogenannten *elastischen Verzerrungen* (englisch: *elastic distortions*) auf 0,4% mit Konvolutionsnetzen und 0,7% mit mehrschichtigen Perzeptrons senken, siehe Abbildung 3.1. Ranzato et al. [53] erreichen mit 0,39% Testfehler das aktuell beste Ergebnis durch Verwendung eines Konvolutionsnetzes mit speziell initialisierten Filtern, die eine übervollständige, spärliche Repräsentation des Datensatzes bilden (siehe Abbildung 3.2). Übervollständig bedeutet dabei, dass mehr Filter verwendet werden, als zur Repräsentation der Datenmenge nötig sind.

Abbildung 3.1: Links Originalmuster, rechts elastisch verzerrte Muster. Quelle: [61], bearbeitet

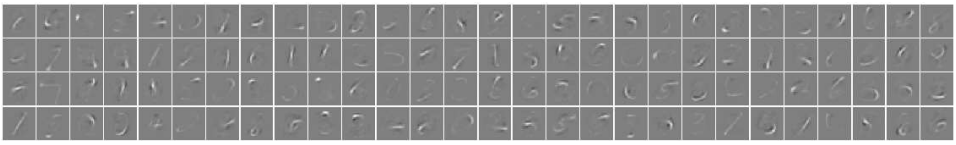


Abbildung 3.2: Teilmenge der spärlichen Filter für den MNIST-Datensatz. Quelle: [53]

Spärlich bedeutet, dass bei jedem Eingabemuster immer nur eine geringe Anzahl der vorhandenen Filter eine deutlich von Null abweichende Ausgabe erzeugt.

3.2 Objekterkennung in natürlichen Bildern

Die allgemeine Objekterkennung, also die Erkennung von beliebigen Objekten in natürlichen Bildern, stellt hohe Anforderungen an die verwendeten Modelle und Lernverfahren. Viele Faktoren, darunter Lichtverhältnisse, Blickwinkel und Verdeckungen führen zu sehr unterschiedlichen Instanzen der Objektklassen.

Zwar existieren viele hinreichend gute Algorithmen für speziellere Probleme wie die Lokalisierung von Gesichtern (siehe zum Beispiel [65]), diese lassen sich aber im Allgemeinen nicht auf die Erkennung beliebiger Objekte übertragen. Aus diesem Grund werden hier nur Arbeiten vorgestellt, die sich mit allgemeiner Objekterkennung beschäftigen.

Flache und hierarchische Ansätze

Modelle zur Objekterkennung lassen sich in *flache* und *hierarchische* Modelle unterteilen. Flache Modelle extrahieren bestimmte Merkmale wie zum Beispiel Kanten aus einem Bild und verwenden diese dann als Eingabe eines nicht-hierarchischen Klassifikators wie Support Vector Machines. Hierarchische Ansätze (auch *tiefe* Ansätze genannt) hingegen führen die Merkmalsextrakation oder die Klassifikation in mehreren hierarchischen Schritten durch. Beispiele dafür sind die in Abschnitt 2.2.3 vorgestellten Modelle sowie das mehrschichtige Perzeptron. Hierarchische Ansätze sind häufig stärker am biologischen Sehsystem orientiert, siehe Abschnitt 2.1.2.

Riesenhuber und Poggio geben in [55] einen Überblick über verschiedene Ansätze und stellen fest, dass hierarchische Ansätze bei allgemeiner Objekterkennung zum aktuellen Zeitpunkt mindestens vergleichbar gute Erkennungsraten erreichen wie flache. Weiterhin werden zum einen vorwärtsgerichtete von rekurrenten Ansätzen abgegrenzt und zum anderen sogenannte *objektzentrierte* (englisch: *object-centered*) von *ansichtszentrierten* (englisch: *view-centered*) Ansätzen unterschieden. Bei objektzentrierten Ansätzen wird versucht, ein vom jeweiligen Blickwinkel unabhängiges Modell des Objektes aus dem Bild zu extrahieren und mit zuvor gelernten Modellen von Objektklassen zu vergleichen (siehe zum Beispiel den „Recognition-by-Components“-Ansatz von Biederman [3]). Ansichtsbasierte Modelle hingegen lernen einzelne Ansichten von Objekten, so dass zum Beispiel jedes Neuron nur auf eingeschränkte Blickwinkel bestimmter Objektklassen mit hoher Aktivierung reagiert. Es wird betont, dass vorwärtsgerichtete, ansichtsbasierte Ansätze eine hohe Übereinstimmung mit den ersten Stufen des menschlichen Sehsystems aufweisen (vergleiche Abschnitt 2.1.2). Rekurrenz ist nach Meinung von Riesenhuber und Poggio für die schnelle Erkennung von bekannten, leicht erkennbaren Objekten vernachlässigbar. Diese These wird dadurch gestützt, dass dieser Vorgang beim Menschen in etwa so lange dauert, wie der Signalfluss vom Auge zum inferotemporalen Cortex [55].

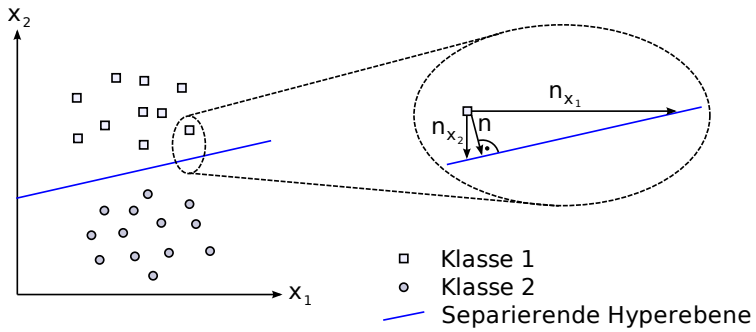


Abbildung 3.3: Darstellung der Klassifizierung mit einer Support Vector Machine (SVM) mit zwei Merkmalen x_1 und x_2 und zwei verschiedenen Klassen. Die separierende Hyperebene wird so berechnet, dass sie einen maximalen Abstand von den jeweils am nächsten aneinanderliegenden Instanzen (*Stützvektoren*) beider Klassen hat. Die Merkmale werden danach gewichtet, wie groß ihr Anteil am Normalenvektor n zur separierenden Hyperebene ist. In diesem Fall ist der Anteil von n_{x_2} über alle Instanzen gemittelt größer als der von n_{x_1} und wird deshalb höher bewertet. Je nach Schwellwert würde das Verfahren aus [44] also das Merkmal x_1 verwerfen und nur noch x_2 verwenden, was in diesem Beispiel auch sinnvoll wäre.

HMAX-basierte Modelle

Auf den Aussagen des vorhergehenden Abschnitts aufbauend schlagen Serre et al. in [60] ein zweistufiges Modell zur allgemeinen Objekterkennung vor:

Im ersten Schritt erfolgt eine vorwärtsgerichtete, ansichtsbasierte Merkmalsextraktion, die dem bereits vorgestellten HMAX-Modell [54] sehr ähnlich ist. Im Unterschied zu diesem werden hier allerdings in der Trainingsphase alle Muster der Trainingsmenge einmal vorwärtspropagiert und dabei viele Teile (englisch: *patches*) unterschiedlicher Position und Größe zufällig aus der C1-Schicht ausgewählt und ausgeschnitten. Jedes dieser Teile bildet dann nach dem Training das Zentrum einer RBF-Zelle der S2-Schicht. Diese Art des unüberwachten Lernens generiert automatisch S2-Zellen, die bestimmte Merkmale von Teilen der Trainingsmuster repräsentieren.

Als zweiter Schritt werden die Ausgaben der C2-Zellen, die als Pooling-Zellen für die S2-Schicht fungieren, als Eingabe für Support Vector Machines beziehungsweise *AdaBoost* [16] verwendet, welche als Klassifikatoren dienen. Mit dieser Struktur wurden sehr gute Ergebnisse auf dem Caltech-101-Datensatz und dem MIT-CBCL-Datensatz [42] erreicht. Auf dem Caltech-101-Datensatz wurden beispielsweise 42% der Testmuster korrekt klassifiziert, wenn jeweils 30 Muster aus jeder der 102 Klassen als Trainingsmenge und der Rest als Testmenge verwendet wurden.

Mutch und Lowe verbessern in [44] obiges Modell durch eine *spärliche Codierung*. Diese wird durch drei Maßnahmen erreicht: Erstens werden die während der Trainingsphase extrahierten Teile spärlicher gespeichert: Statt pro Position die Aktivierung aller vier Filter (die vier unterschiedliche Orientierungen der zugrundeliegenden Gaborfilter darstellen) zu speichern, wird nur die Aktivierung und der Index des Filters mit der größten Aktivierung gespeichert. Diese Vorgehensweise setzt das Maximumsprinzip des HMAX-Modells fort. Zweitens wird *laterale Hemmung* eingesetzt, um nicht-dominante Ausgaben der S1- und C1- Schicht zu unterdrücken. Laterale Hemmung bedeutet hier, dass in einem jeweils lokalen Umfeld alle Aktivierungen unterdrückt werden, die kleiner als ein bestimmter Anteil – festgelegt durch einen Schwellwert – der aktuell betrachteten Aktivierung sind. Drittens wird ein Teil der gelernten S2-Zellen wieder verworfen. Dies geschieht, indem zunächst eine

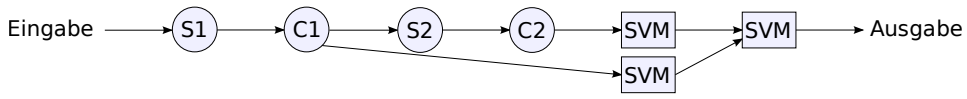


Abbildung 3.4: Prinzip der semantischen Verkettung. Die Ausgabe der C1-Schicht und der C2-Schicht werden jeweils von einer SVM klassifiziert. Die Ausgabe dieser beiden SVMs wird wiederum durch eine weitere SVM klassifiziert und bildet die finale Ausgabe. Nach [66]

SVM mit den Merkmalen aller S2-Zellen trainiert wird. Da eine SVM Klassen durch eine Hyperebene separiert, können die Merkmale danach gewichtet werden, wie gut sie sich für die Separierung eignen, siehe Abbildung 3.3. Die S2-Zellen mit den für die Klassifizierung am wenigsten hilfreichen Merkmalen werden entfernt. Durch diese drei Verbesserungen wurden auf dem Caltech-101-Datensatz bei gleicher Meßmethode wie oben mit 30 Trainingsmustern pro Klasse 56% der Testmuster korrekt klassifiziert, bei 15 Trainingsmustern pro Klasse betrug die Erkennungsrate 51%.

Serre et al. geben in [59], aufbauend auf [66], eine noch bessere Erkennungsrate von 55% bei 15 Trainingsbeispielen an. Dabei wird das bereits beschriebene Modell aus [60] (HMAX und SVM) verwendet, allerdings wird je ein Klassifikator (also eine SVM) für die Ausgabe der C1- und der C2-Schicht verwendet. Die Ausgabe beider Klassifikatoren dient dann wieder als Eingabe für den finalen Klassifikator, siehe Abbildung 3.4. Diese Methode nennen die Autoren *semantische Verkettung* (englisch: *semantic concatenation*).

In [59] wird außerdem ein „Sliding Window"-Ansatz evaluiert, der dem in dieser Arbeit gewählten Ansatz sehr ähnlich ist. Als Datensatz wurde hier CBCL StreetScenes [42] verwendet. Zum Trainieren wurden Instanzen der Klassen Auto (*car*), Fußgänger (*pedestrian*) und Fahrrad (*bicycle*) sowie zufällig gewählte Teile, die sich mit keiner Instanz der genannten Klassen überschneiden, ausgeschnitten sowie auf 128×128 Pixel skaliert und in Graustufen umgewandelt. Die Klassifikation erfolgte mit *gentleBoost* [17], wobei verschiedene Merkmale als Eingaben für den Klassifikator miteinander verglichen wurden: (1) die Ausgaben der C1-Schicht, (2) die Ausgaben der C2-Schicht, (3) das normalisierte Graustufenbild, (4) *Histograms of Gradients* (*HoG*) [12], (5) *Local Patch Correlation* [62] und (6) der teilebasierte Ansatz aus [39]. Die Ergebnisse des Vergleichs sind in Abbildung 3.5 zu finden (aus Platzgründen sind nur die Erkennungsraten für Autos und Fußgänger dargestellt). Bei der Erkennung von Autos schneiden die HMAX-Merkmale am besten ab, bei Fußgängern hingegen der HoG-Ansatz.

Bei letzterem wird das Bild zunächst lokal normalisiert und in überlappende *Zellen* aufgeteilt. Innerhalb jeder Zelle werden dann Gradienten (Kanten) verschiedener Orientierung an allen Positionen berechnet. Gradienten gleicher Orientierung werden pro Zelle zu einem Wert zusammengefasst, um eine gewisse Translationsinvarianz zu erreichen. Die Parameter des Modells wurden speziell auf die Erkennung von Fußgängern optimiert, siehe dazu [12].

Bewertung von Bilddatensätzen

Pinto et al. diskutieren in [50], wie gut sich die am häufigsten verwendeten Datensätze natürlicher Bilder als Benchmark für allgemeine Objekterkennungsverfahren eignen. Sie bemängeln, dass häufig genutzte Bilddatensätze wie Caltech-101 nur wenig Varianz bei Blickwinkel, Objektgröße und -position aufweisen. Sie belegen dies, indem sie mit einem relativ einfachen, flachen Modell Erkennungsraten auf Caltech-101 erreichen, die vergleichbar mit den zuvor genannten State-of-the-Art-Verfahren sind.

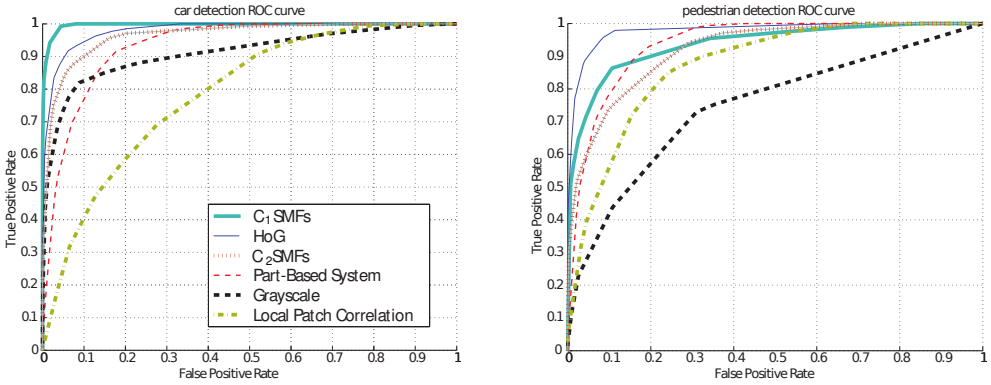


Abbildung 3.5: Erkennungsraten für Autos und Personen bei Verwendung der in [59] beschriebenen Merkmale für die SVM-Klassifikation. Quelle: [59]

Messungen mit dem NORB-Datensatz

LeCun et al. führen in [38] Messungen mit dem NORB-Datensatz durch. Als Klassifizierungsverfahren werden K-Nearest-Neighbor (KNN), SVMs und Konvolutionsnetze miteinander verglichen. Mit 6,6% Fehlerrate auf der Testmenge liegt das Konvolutionsnetz deutlich vor der besten SVM mit 12,6% und KNN mit 16,6%. Wegen der hohen Ausführungszeit müssen bei den Verfahren KNN und SVM allerdings die Trainingsmengen verkleinert werden. Das Konvolutionsnetz wird hingegen neben der ursprünglichen Trainingsmenge mit 24.300 Mustern auch mit größeren und schwierigeren Mengen trainiert, indem zufällige Hintergründe mit einfachen Texturen sowie Ausschnitten aus natürlichen Bildern eingefügt werden. Hier erreicht das Konvolutionsnetz je nach verwendeter Trainingsmenge Fehlerraten zwischen 7,1% und 39,9%.

Einfluss hierarchischer Merkmalsextraktion

Oberhoff und Kolesnik [49] nutzen ein HMAX-ähnliches Modell mit jeweils drei S- und C-Schichten und unüberwachtem Lernen der Merkmale zur Objekterkennung. Als Trainings- und Testmenge dienen Personen und Autos aus dem LabelMe-Datensatz. Um den Einfluss der hierarchischen Merkmalsextraktion zu testen, wird ein Klassifikator nacheinander mit den Ausgaben der drei C-Schichten trainiert und die Ergebnisse verglichen. Wie Tabelle 3.1 zeigt, steigen die Erkennungsraten bei Verwendung höherer C-Schichten als Eingabe für den Klassifikator.

Schicht	C1	C2	C3
Autos	68,7 %	74,7 %	90,3 %
Personen	57,6 %	76,5 %	84,7 %
Sonstige	50,4 %	61,3 %	72,2 %

Tabelle 3.1: Erkennungsraten auf Teilen des LabelMe-Datensatzes aus [49]. Die Spaltenüberschrift gibt die Schicht an, deren Ausgaben als Eingaben für den Klassifikator verwendet wurden. Es ist deutlich zu erkennen, dass Ausgaben aus höheren Schichten zu besseren Erkennungsraten führen.

3.3 Implementierung von neuronalen Netzen auf Grafikkarten

In diesem Abschnitt werden Arbeiten vorgestellt, die sich mit der Implementierung neuronaler Netze auf Grafikkarten beschäftigen. Mit Ausnahme der ersten stammen alle Veröffentlichungen aus dem Jahr 2008 und nutzen das CUDA-Framework.

Beschleunigung von Konvolutionsnetzen mit CPU-BLAS-Bibliothek und GPU

In der Arbeit von Chellapilla et al. aus dem Jahr 2006 [7] werden drei Techniken zur Beschleunigung von Konvolutionsnetzen evaluiert. Erstens wird durch Entfalten und Duplizieren der Daten in den Konvolutionsschichten sowohl die Vorwärts- als auch die Rückwärtspropagierung so verändert, dass beides als Matrixmultiplikation berechnet werden kann. Diese Technik bringt allein allerdings noch keinen Geschwindigkeitsvorteil. Zweitens wird für die genannten Matrixoperationen eine für die verwendete CPU optimierte BLAS-Bibliothek (Basic Linear Algebra Subroutines) von Intel genutzt, womit ein Beschleunigungsfaktor zwischen 2,4 und 3 gegenüber der naiven CPU-Implementierung erreicht wird. Drittens wurde das *Pixel-Shader*-Modell verwendet, um die für die Konvolutionen benötigten Matrixoperationen auf der Grafikkarte statt auf der CPU durchzuführen. Damit wurde ein Beschleunigungsfaktor zwischen 3,11 und 4,11 gegenüber der naiven CPU-Version erreicht. Für die Messungen wurde eine NVIDIA GeForce 7800 Ultra verwendet. Die verwendete CPU wird nicht genannt.

HMAX-Implementierung mit CUDA

In einer Projektgruppenausarbeitung [8] beschreibt Chikkerur die Implementierung des HMAX-Modells (vergleiche Abschnitt 2.2.3) als MATLAB-, Multithreading- und CUDA-Variante.

Das HMAX-Modell ist sehr berechnungsintensiv, da eine große Anzahl von Filtern auf alle Bereiche des Eingabebildes angewandt werden muss. Hier wurden beispielsweise in Schicht S1 64 verschiedene Filter verwendet (16 Größen mit je 4 Orientierungen). Auf der anderen Seite ist das Modell bereits im Ansatz massiv parallel, es werden für jeden Bereich der Eingabe die gleichen Filter genutzt.

Zur Parallelisierung auf der CPU wird ein Thread für jeden Filter der Schichten S1 und S2 und analog dazu ein Thread pro komplexer Zelle (MAX-Operation über alle Eingaben) in den Schichten C1 und C2 verwendet. Bei der CUDA-Implementierung wird für S1 und S2 ein Block pro Filter verwendet, wobei jeder Thread innerhalb des Blocks je eine Zeile der Eingabe bearbeitet. Die Berechnung von C1 nutzt einen Block pro Filtergröße. Auch hier bearbeitet jeder Thread eine Zeile. C2 wird auf der CPU berechnet, da sich eine Berechnung auf GPU laut dem Autor wegen des entstehenden Overheads nicht lohnt.

Von den drei verglichenen Implementierungen ist die CUDA-Variante mit Abstand am schnellsten. Sie erreicht gegenüber der MATLAB-Variante Geschwindigkeitsgewinne zwischen Faktor 11,3 und 25,2, je nach betrachteter Schicht des HMAX-Modells. Bei der Multithread-CPU-Variante beträgt der Faktor zwischen 1,5 und 8,2 gegenüber der MATLAB-Variante. Vergleicht man die Multithreaded- mit der CUDA-Variante, ist letztere um Faktor 1,7 bis 11,2 schneller. Für die Messung wurde eine Dualcore-CPU mit 3,0 GHz verwendet, als Grafikkarte diente eine NVIDIA GeForce 8800 GTX.

Parzen-Fenster-Methode und mehrschichtige Perzeptrons mit CUDA

Lahabar et al. beschreiben in [34] die Implementierung der *Parzen-Fenster-Methode* und mehrschichtiger Perzeptrons auf CUDA-Grafikkarten.

Die Parzen-Fenster-Methode, auch *Kernel-Dichte-Schätzung* (englisch: *Kernel Density Estimation*) genannt, ist eine statistische Methode zur nicht-parametrischen Approximation der Wahrscheinlichkeitsdichtefunktion einer Zufallsvariable. Sie berechnet aus N d-dimensionalen Vektoren einer Stichprobe $\mathbf{x}_1, \ldots, \mathbf{x}_N$ eine geglättete Schätzung der Wahrscheinlichkeitsdichtefunktion der Zufallsvariable:

$$p_N(\mathbf{x}) = \frac{1}{Nh_N} \sum_{i=1}^{N} \varphi\left(\frac{\mathbf{x} - \mathbf{x}_i}{h_N}\right).$$

Dabei ist h der Glättungsparameter und φ eine sogenannte *Kernelfunktion*, aus der die Zielfunktion p_N zusammengesetzt wird. Häufig wird dafür die Gaußfunktion verwendet. Die Berechnung lässt sich im Wesentlichen mit zwei Matrixmultiplikationen ausdrücken, wobei die Matrizen je nach Stichprobengröße und Dimension der Stichprobenvektoren sehr groß werden können. Für die Berechnung auf der Grafikkarte wird neben zwei Matrixmultiplikationen ein (CUDA-)Kernel benötigt, der auf jedes Element einer Matrix die gewählte Kernelfunktion φ anwendet. Der maximal erreichte Beschleunigungsfaktor gegenüber einer MATLAB-Implementierung beträgt etwa 336 bei 4000 16-dimensionalen Stichprobenelementen.

Des Weiteren wurde ein mehrschichtiges Perzeptron mit Vorwärts- und Rückwärtspropagierung (Backpropagation of Error) implementiert. Da das Netz im Batch-Modus trainiert wurde, konnten die zeitaufwändigsten Operationen als Matrixmultiplikationen ausgedrückt werden. Die GPU-Implementierung erfolgte auf zwei verschiedene Weisen: Erstens ohne Nutzung des CUDA-Frameworks durch Aufruf von Shadern und zweitens unter Verwendung der CUBLAS-Bibliothek. Als Datensatz wurden 22×22 Pixel große handgeschriebene Zeichen aus dem NIST-Datensatz[1] verwendet, womit sich eine Eingabedimension von 484 ergibt. Außerdem enthielt das Netz eine versteckte Schicht mit 128 Neuronen und eine Ausgabeschicht mit 10 Neuronen (eins für jede Ziffer). Die CUBLAS-Variante erreicht eine um maximal Faktor 110 höhere Geschwindigkeit als eine von der Funktionsweise her äquivalente MATLAB-Variante. Die Shader-Variante ist deutlich langsamer als CUDA/CUBLAS, erreicht aber immer noch einen Faktor von maximal 50 gegenüber MATLAB. Außerdem wurde noch mit dem CPU-Programm FANN (Fast Artificial Neural Network) verglichen, das allerdings langsamer als MATLAB lief.

Für die Messungen wurde eine 3 GHz Pentium-4-CPU und eine NVIDIA GeForce 8800 GTX als Grafikkarte verwendet.

Schriftlokalisierung in Bildern mit OpenMP und CUDA

In [29] beschreiben Jang et al. die Implementierung eines Systems zur Lokalisierung von Schrift in Bildern. Das Ziel ist die Binarisierung von beliebigen Eingabebildern wie zum Beispiel Seiten einer Zeitschrift, so dass Text schwarz erscheint und alles andere weiß. Die Implementierung erfolgte zweigeteilt: Die Vorverarbeitung wurde mit *OpenMP*[2] (*Open Multi-Processing*) auf der CPU parallelisiert, um den verwendeten Mehrkernprozessor effizient auszunutzen. Als zweiter Schritt wurden die extrahierten Merkmale durch ein vortrainiertes mehrschichtiges Perzeptron propagiert. Dieser Schritt wurde mit Hilfe von CUDA auf der Grafikkarte ausgeführt. Der erreichte Beschleunigungsfaktor beträgt 20 gegenüber einer CPU-Variante, die lediglich einen Kern verwendet. Wenn die Merkmalsextraktion ebenfalls auf der GPU ausgeführt wird, beträgt der Faktor lediglich 5, da sehr viele Daten aufgrund des beschränkten Grafikkartenspeichers in mehreren Schritten vom Host kopiert werden müssen.

[1] `http://www.nist.gov`
[2] `http://www.openmp.org`

Gesichtserkennung mit einem Neocognitron und CUDA

Poli et al. beschreiben in [52] die Implementierung eines Neocognitrons (siehe Abschnitt 2.2.3) mit CUDA zur Gesichtserkennung. Die Lernphase wird ausschließlich auf der CPU ausgeführt, lediglich die Vorwärtspropagierung erfolgt auf der Grafikkarte. Dabei repräsentiert jeder Thread ein Neuron, jeder Block eine Zellfläche und jede Schicht ein Grid, also einen Kernelaufruf. Die Laufzeiten wurden mit denen einer CPU-Variante verglichen, die auf einem und auf acht CPU-Kernen getestet wurde. Im ersten Fall betrug der Geschwindigkeitsgewinn der CUDA-Variante Faktor 255,319, im zweiten Fall 79,787.

3.4 Diskussion

In den vorgestellten Arbeiten zur allgemeinen Objekterkennung werden sowohl flache [50] als auch hierarchische Modelle [38; 59] genutzt. Mehrere Arbeiten kombinieren auch eine hierarchische Merkmalsextraktion mit einer flachen Klassifizierung per SVM [60; 44; 49].

Die Ergebnisse legen nahe, dass grundsätzlich sowohl flache als auch hierarchische Ansätze für allgemeine Objekterkennung geeignet sind und zum aktuellen Zeitpunkt vergleichbar gute Erkennungsraten erzielen. Einige flache Ansätze, darunter die in den vorgestellten Arbeiten zumeist genutzten SVMs, skalieren allerdings schlechter mit größeren Datenmengen [38]. Der hierarchische Aufbau des menschlichen Sehsystems legt außerdem nahe, dass hierarchische Ansätze das Potenzial haben, auch enorm große Datenmengen effizient zu verarbeiten.

Die Erkenntnisse aus [50] zeigen, dass auch die Wahl des Datensatzes eine entscheidende Rolle bei der Bewertung von Objekterkennungsverfahren spielt. Datensätze, die zu wenig Varianz in Blickwinkel, Objektgröße oder -position aufweisen, sind nicht gut geeignet, um die Erkennungsleistung verschiedener Modelle realistisch zu bewerten.

Während speziellere Probleme wie die Erkennung handgeschriebener Ziffern praktisch gelöst sind – Computer erreichen hier bessere Ergebnisse als Menschen – besteht im Bereich der allgemeinen Objekterkennung noch erheblicher Forschungsbedarf in mehrere Richtungen:

- Training und Bewertung anhand von realistischeren Datensätzen
- Stärkere Ausrichtung am Vorbild des biologischen Sehsystems, das heißt zum Beispiel lokale statt globale Merkmalsextraktion, um massiv parallele Verarbeitung zu ermöglichen
- Verarbeitung größerer Datenmengen beziehungsweise höhere Verarbeitungsgeschwindigkeit bis hin zur Echtzeitanwendung auch in komplexen Szenarien

Das Ziel der vorliegenden Arbeit ist es, den genannten Punkten durch Verwendung des LabelMe-Datensatzes und der parallelen Implementierung des biologisch motivierten Modells ein Stück näher zu kommen. Im nächsten Kapitel wird das dafür verwendete Modell vorgestellt.

4 Entwurf des Objekterkennungssystems

In diesem Kapitel wird das im weiteren Verlauf verwendete Modell zur Objekterkennung vorgestellt. Es werden die benötigten Begriffe definiert und die Entscheidungen für das Design des Modells begründet.

4.1 Ein künstliches neuronales Netz als Grundlage

Die Grundstruktur des Modells ist ein vorwärtsgerichtetes künstliches neuronales Netz. Wie aus dem vorhergehenden Kapitel hervorgeht, hat diese Struktur grundsätzlich das Potenzial, State-of-the-Art-Ergebnisse im Bereich der Objekterkennung zu erreichen. Vergleichbar gute Ergebnisse werden derzeit zwar auch von Support Vector Machines erzielt, allerdings sprechen mehrere Gründe gegen eine Verwendung von SVMs in der vorliegenden Arbeit: Erstens wurde in [38] gezeigt, dass die Ausführungszeit von SVMs bei sehr großen Eingaben deutlich stärker ansteigt als die von vorwärtsgerichteten neuronalen Netzen. Eine hohe Geschwindigkeit stellt jedoch eines der wesentlichen Ziele dieser Arbeit dar. Zweitens sind SVMs nicht biologisch motiviert. Der Autor der vorliegenden Arbeit ist der Meinung, dass biologisch motivierte Ansätze ein sehr großes Potenzial besitzen, wie man an der Leistungsfähigkeit des menschlichen Sehsystems erkennen kann. Drittens eignet sich ein neuronales Netz grundsätzlich sehr gut für eine Implementierung mit CUDA, da es aus vielen gleichartigen Neuronen aufgebaut ist und sich daher auf natürliche Weise feinkörnig parallelisieren lässt.

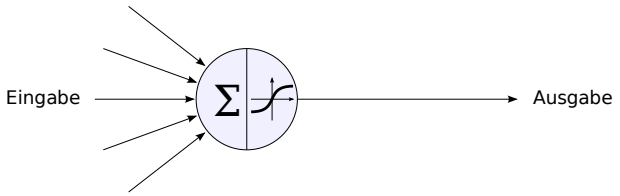


Abbildung 4.1: Neuronenmodell des Perzeptrons, das auch für das hier vorgestellte Modell verwendet wird. Die Netzeingabe ist durch das $\sum$-Symbol dargestellt.

Den Grundbaustein für das Netz bilden Neuronen, die denen des Perzeptrons gleichen (siehe Abbildung 4.1). Dieses Neuronenmodell hat die vorteilhafte Eigenschaft, dass die Netzeingabe

$$net_j = \sum_{i \in I} (a_i \cdot w_{i,j}) + \theta_j$$

eines Neurons j auf der Grafikkarte effizient berechnet werden kann, da ein MAD-Befehl (Multiply-Add) exitiert, der Multiplikation (von Aktivität und Gewicht) und Addition (zur bisherigen Netzeingabe) gemeinsam in einer Instruktion durchführt. Als Aktivierungsfunktion $f_j(net_j)$ kann zwischen dem Tangens Hyperbolicus $tanh(net_j)$ und der Fermi-Funktion $1/(1 + \exp(-net_j))$ gewählt werden.

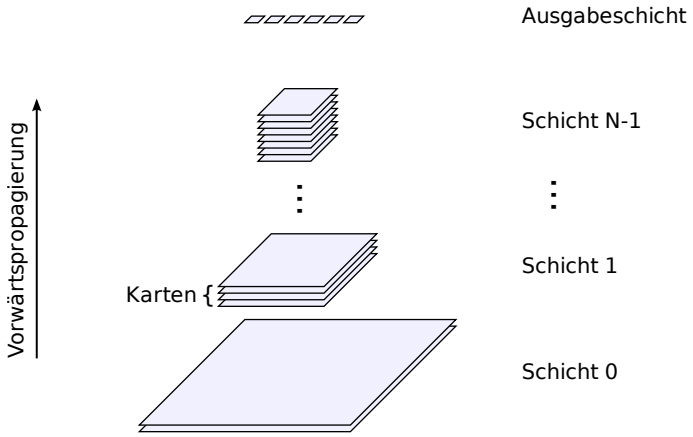


Abbildung 4.2: Modell des hier beschriebenen neuronalen Netzes

4.1.1 Hierarchischer Aufbau

Das neuronale Netz ist aus N *Schichten* aufgebaut, die von 0 bis $N-1$ indexiert sind. Jede Schicht besteht aus mindestens einer *Karte*. Jede Karte besteht aus Neuronen in zweidimensionaler, quadratischer Anordnung. Karten einer Schicht haben immer die gleiche Größe. Zusätzlich existiert eine eindimensionale *Ausgabeschicht* aus M Neuronen. Man beachte, dass die Anzahl Schichten N des Netzes stets ohne die Ausgabeschicht angegeben wird. Um die Schichten 0 bis $N-1$ sprachlich von der Ausgabeschicht abzugrenzen, werden diese im Folgenden auch als *reguläre Schichten* und deren Karten als *reguläre Karten* bezeichnet. Abbildung 4.2 veranschaulicht das Modell.

Wie bei der neuronalen Abstraktionspyramide aus Abschnitt 2.2.3 werden die Karten mit wachsendem Schichtindex kleiner, um die Ortsinvarianz sukzessive zu erhöhen. Konkret wurde in der vorliegenden Arbeit eine Halbierung der Seitenlänge mit jeder weiteren Schicht gewählt, weil sich die Bearbeitung einer Zweierpotenz von Neuronen generell besser mit CUDA implementieren lässt als andere Zahlen. Bestehen also beispielsweise Karten in Schicht 0 aus 256×256 Neuronen, so bestehen Karten in Schicht 1 aus 128×128 Neuronen.

4.1.2 Lokale Verbindungen

Für $n \in \{1, N-1\}$ kann jede Karte aus Schicht n mit jeder Karte aus Schicht $n-1$ verbunden sein. Solche Verbindungen zwischen zwei Karten werden im Folgenden als *Kartenverbindungen* bezeichnet, um sie der Eindeutigkeit halber von Verbindungen zwischen einzelnen Neuronen der beiden Karten innerhalb einer Kartenverbindung abzugrenzen. Im Kontext der Vorwärtspropagierung heißt die Karte aus Schicht $n-1$ *Quellkarte*, die aus Schicht n *Zielkarte*. Entsprechend werden auch Neuronen dieser Karten als *Quell-* und *Zielneuronen* bezeichnet. Im Kontext der Rückwärtspropagierung sind die Bezeichnungen umgekehrt. Da der Kontext an einigen Stellen nicht eindeutig ist, werden die beiden Karten einer Kartenverbindung im Folgenden auch einfach als größere und kleinere Karte oder als Karte mit dem kleineren beziehungsweise größeren Index bezeichnet.

Innerhalb einer Kartenverbindung sind die Neuronen der größeren Karte (Quellneuronen) und die der kleineren Karte (Zielneuronen) lokal verbunden. Das bedeutet, dass ein Zielneuron ausschließlich zu Quellneuronen verbunden ist, die relativ zur Kartengröße an der gleichen Stelle auf der Quellkarte liegen. Ist die Zielkarte (bezogen auf die Seitenlänge) halb so groß wie die Quellkarte, so wäre jedes Zielneuron an der Position (x, y) mit vier Quellneuronen an den Positionen $(2x, 2y)$, $(2x+1, 2y)$, $(2x, 2y+1)$ und $(2x+1, 2y+1)$ verbunden.

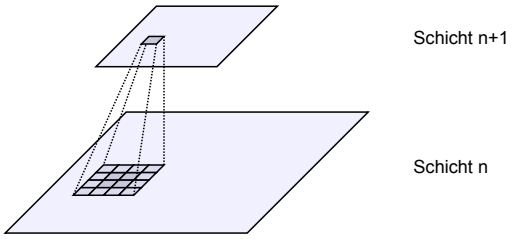


Abbildung 4.3: Darstellung einer Kartenverbindung. Das Zielneuron einer Karte aus Schicht $n+1$ erhält Eingaben von insgesamt 16 Quellneuronen einer Karte aus Schicht n. Die äußeren Quellneuronen werden als Rand bezeichnet.

Zusätzlich werden auch noch Neuronen in einem Randbereich der Breite r um die zuvor genannten Quellneuronen zum entsprechenden Zielneuron verbunden (siehe Abbildung 4.3). Dadurch überlappen sich die rezeptiven Felder der Zielneuronen. Im Rahmen dieser Arbeit wurde als Randbreite $r = 1$ festgelegt.

Die lokalen Verbindungen sind durch die retinotope Struktur im primären visuellen Cortex motiviert (vergleiche Abschnitt 2.1.2). Außerdem dient als Motivation, dass bei Konvolutionsnetzen, welche im Bereich der Handschrift- und Objekterkennung sehr gute Ergebnisse erzielt haben, ebenfalls nur ein räumlich beschränkter Bereich (entsprechend der Größe des Konvolutionsfilters) die Netzeingabe des Zielneurons bildet.

Im Gegensatz zu Konvolutionsnetzen werden im hier vorgestellten Modell aber keine gekoppelten Gewichte eingesetzt, das heißt, jede Verbindung zwischen zwei Neuronen hat ein eigenes Verbindungsgewicht. Der Grund dafür ist zum einen, dass im menschlichen Sehsystem ebenfalls keine gekoppelten Gewichte vorhanden sind. Zum anderen sind dem Autor bisher keine Arbeiten bekannt, die lokale, nicht gekoppelte Gewichte in dieser Größenordnung verwenden, so dass eventuell neue Erkenntnisse gewonnen werden können. Da das hier vorgestellte System leistungsfähig genug ist, um mit der großen Menge an Verbindungsgewichten umzugehen, wurden deshalb nicht-gekoppelte Gewichte gewählt.

Die Ausgabeschicht des neuronalen Netzes ist nicht lokal, sondern wie beim mehrschichtigen Perzeptron vollständig mit Schicht $N-1$ verknüpft, damit das rezeptive Feld eines Ausgabeneurons den gesamten Bereich der Quellkarte – und damit auch automatisch den gesamten Bereich aller Karten in allen Schichten – abdeckt. Jede Karte der Schicht $N-1$ ist mit jeweils allen Ausgabeneuronen verbunden. Eine solche Verbindung heißt *Ausgabeverbindung*, um sie sprachlich von Kartenverbindungen abzugrenzen.

4.1.3 Randbehandlung

Ein Sonderfall sind Neuronen, deren rezeptives Feld über die Quellkarte hinausreicht, so dass einige der potenziellen Quellneuronen gar nicht existieren. Eingaben dieser Positionen werden grundsätzlich auf Null gesetzt. Damit erhalten Neuronen am Rand einer Zielkarte im Allgemeinen betragsmäßig kleinere Netzeingaben. Dieses Verhalten ist erwünscht, da auf diese Weise negative Einflüsse durch Randeffekte reduziert werden. Ein Beispiel dafür ist die fehlerhafte Berechnung von Kanten (siehe Abschnitt 4.3.2) am Rand einer Karte.

4.2 Verwendung eines Sliding Windows

Alle Schichten beziehungsweise Karten des neuronalen Netzes haben eine feste Größe. Trainingsmuster der zu lernenden Objektklassen werden während der Trainingsphase zentriert

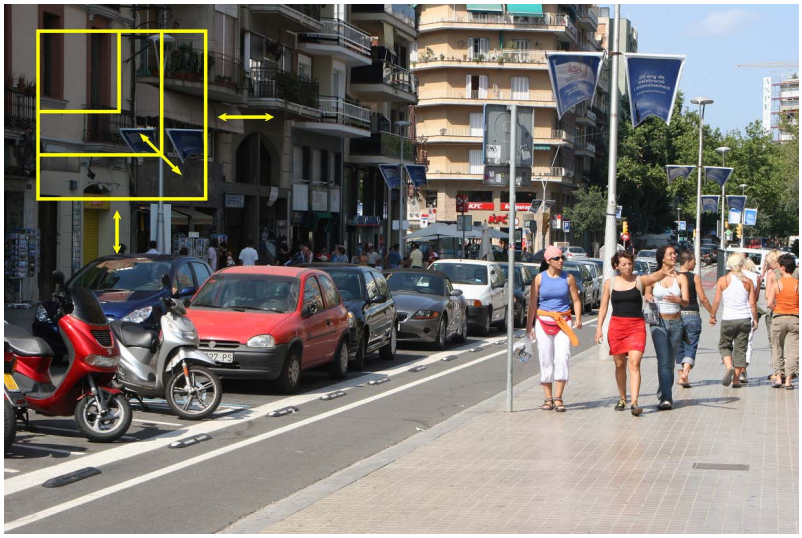


Abbildung 4.4: Veranschaulichung der „Sliding Window"-Technik. Das Fenster, das als Eingabe für das neuronale Netz dient, wird in einem festgelegten Raster nacheinander auf alle möglichen x- und y-Positionen gelegt. Dieses Vorgehen wird für verschiedene Skalen (also Größen des Fensters) wiederholt. Bildquelle: [43], bearbeitet

und skaliert eingegeben. Diese Vorgehensweise erleichtert es dem Netz, Gemeinsamkeiten und Unterschiede von Objekten zu lernen. Allerdings kann dadurch in der Anwendungsphase nicht das ganze Bild in das Netz eingegeben werden. Stattdessen wird ein *gleitendes Fenster* oder *Sliding Window* verwendet, so dass nacheinander alle möglichen Ausschnitte in x- und y-Richtung in einem festgelegten Raster und in verschiedenen Skalen in das Netz eingegeben werden, um Objekte im Bild zu finden (siehe Abbildung 4.4).

Diese Technik ist trotz der Einfachheit mit der biologischen Motivation des Modells vereinbar, da auch der Mensch immer nur einen kleinen Ausschnitt des gesamten Blickfelds aktiv wahrnimmt. Die Vergößerung durch verschiedene Skalierungen des Fensters ist dagegen nicht biologisch plausibel, soll dem System aber die Erkennung unterschiedlich großer Objekte vereinfachen. Eine explizite Invarianz gegenüber Rotationen ist – wie bei den ersten Stufen des menschlichen Sehsystems – nicht vorgesehen.

Der Sliding-Window-Ansatz bietet die Möglichkeit, in nachfolgenden Arbeiten durch Aufmerksamkeitsmodelle wie in [28] verbessert zu werden, so dass ähnlich wie bei der aufeinanderfolgenden Fixierung des Auges auf bestimmte Punkte in einer Szene bestimmte Bereiche bevorzugt untersucht werden. Alternativ kann zum Beispiel bei Videos ein Objekt, wenn es einmal im Bild gefunden wurde, leichter verfolgt werden, wenn gewisse Annahmen über die Bewegungsgeschwindigkeit und -richtung zutreffen.

Ein Nachteil des Ansatzes ist der hohe Rechenaufwand. Die Berechnung ist allerdings mit CUDA sehr gut parallelisierbar. Da die Technik nur in der Anwendungs- und nicht in der Trainingsphase verwendet wird, wird die Lerngeschwindigkeit des Netzes nicht beeinflusst.

4.3 Eingabe in das Netz

Eingaben können grundsätzlich in jeder Karte einer regulären Schicht erfolgen. Dazu werden die Aktivierungen der Neuronen auf die gewünschte Eingabe gesetzt. Die Karte heißt dann *Eingabekarte*. Eingabekarten können offensichtlich nicht Ziel von Kartenverbindungen sein, da die Eingabewerte sonst bei der Vorwärtspropagierung überschrieben würden. Bei

Verwendung des Tangens Hyperbolicus als Aktivierungsfunktion sind Eingaben im Bereich $[-1, 1]$ sinnvoll, aber nicht zwingend nötig. Bei Verwendung der Fermi-Funktion ergibt sich entsprechend ein Wertebereich von $[0, 1]$.

Die direkte Eingabe in höheren Schichten ist biologisch nicht plausibel, kann dem Netz allerdings die Erkennung von Strukturen erleichtern, die größer als das rezeptive Feld eines Neurons in Schicht 1 sind. In Kapitel 6 werden entsprechende Vergleichsmessungen durchgeführt.

Nachfolgend werden zwei Eingaben für das neuronale Netz beschrieben, die sehr schnell aus dem Trainingsbild extrahiert werden können. Die beschriebene Trennung zwischen Farb- und Helligkeitsinformation findet sich auch im menschlichen Sehsystem bis hin zum visuellen Cortex [30]. Da eine hohe Ausführungsgeschwindigkeit ein wesentliches Ziel des vorgestellten Systems darstellt, wird auf berechnungsaufwändige Merkmalsextraktion wie zum Beispiel Gabor-Filter verzichtet. Insbesondere werden nicht-lokale Merkmale vermieden, da sie biologisch nicht plausibel sind und sich außerdem mit CUDA nicht effizient implementieren lassen. Der Grund dafür ist, dass lokale Speicherzugriffe meist schneller durchgeführt werden können (siehe dazu Kapitel 5).

4.3.1 Farbkanäle und Hauptkomponentenanalyse

In den meisten Arbeiten zum Thema Objekterkennung werden natürliche Bilder vor der Weiterverarbeitung in Graustufen umgewandelt. Dieser Schritt verringert den Rechenaufwand und wird teilweise damit begründet, dass der Mensch Objekte in Graustufenbildern fast genauso gut erkennt wie in Farbbildern. Dennoch bietet Farbe zusätzliche Informationen, die selbstverständlich auch vom menschlichen Sehsystem genutzt werden. Aus diesem Grund werden Farbinformationen im hier beschriebenen Modell nicht verworfen.

Farbinformationen sind in Bildern des LabelMe-Datensatzes sowie in allen anderen Bilddatensätzen, die das JPG-Format benutzen, in Form je eines Rot-, Grün- und Blaukanals (RGB) gespeichert. Ein naheliegender Ansatz wäre, diese direkt als drei Karten in das neuronale Netz einzugeben. Allerdings sind alle drei Kanäle stark korreliert (siehe Abbildung 4.5 oben). In [35] beschreiben LeCun et al., dass stark korrelierte Eingaben bei neuronalen Netzen vermieden werden sollten, da sie sich negativ auf den Lernerfolg des Gradientenabstiegverfahrens auswirken können.

Eine Möglichkeit zur Dekorrelation der Daten stellt die *Hauptkomponentenanalyse* (englisch: *Principal Component Analysis*, *PCA*) dar. Das Verfahren berechnet aus einer großen Stichprobe n-dimensionaler Vektoren – in diesem Fall ist $n = 3$, nämlich Rot-, Grün- und Blaukanal – eine neue orthonormale Basis mit der Eigenschaft, dass der erste Basisvektor in Richtung der größten Varianz der Stichprobe zeigt. Er wird *erste Hauptkomponente* genannt. Für eine Stichprobe, bei der die $n = 3$ Dimensionen aller Elemente jeweils den gleichen Wert haben (und somit alle Dimensionen der Stichprobe eine paarweise Korrelation von 1 aufweisen), ergäbe sich zum Beispiel

$$\overrightarrow{p_1} = \pm \begin{pmatrix} \sqrt{\frac{1}{3}} \\ \sqrt{\frac{1}{3}} \\ \sqrt{\frac{1}{3}} \end{pmatrix}$$

als erste Hauptkomponente, weil nur in dieser Richtung die Varianz ungleich Null ist und $\|\overrightarrow{p_1}\| = 1$ gilt. Die weiteren Hauptkomponenten beziehungsweise Basisvektoren ergänzen den ersten zu einer orthonormalen Basis und werden so sortiert, dass die Varianz in der entsprechenden Richtung jedes Basisvektors abnimmt.

Durch die Anwendung von PCA lässt sich die paarweise Korrelation der drei Eingabedimensionen deutlich verringern. Zum Test wurde eine große Stichprobe aus dem LabelMe-

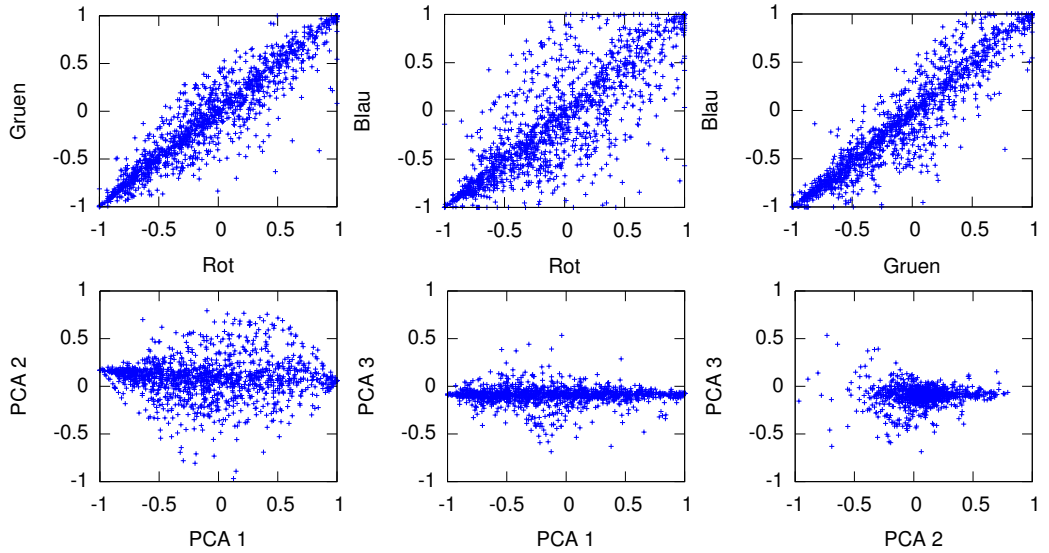


Abbildung 4.5: Vergleich der Korrelation von RGB- und PCA-Kanälen. Die obere Zeile zeigt die starke Korrelation von je zwei RGB-Kanälen. In der unteren Zeile sind je zwei der Kanäle gegeneinander aufgetragen, die durch die Hauptkomponentenanalyse (PCA) aus den RGB-Kanälen berechnet wurden. Der Wertebereich ist bei allen Stichproben auf $[-1, 1]$ skaliert.

Datensatz extrahiert, indem je ein zufälliges Bild aus jeder der 260 Bildserien ausgewählt wurde. Anschließend wurden die RGB-Werte aller Pixel auf der Diagonalen durch jedes dieser Bilder gespeichert – also an Position $(1, 1), (2, 2), \ldots, (m, m)$ mit $m = min(x_{max}, y_{max})$, wobei x_{max} der Breite des Bildes in Pixeln und y_{max} der Höhe entspricht. Die resultierende Stichprobe besteht aus 266.292 Pixeln. Die Hauptkomponentenanalyse wurde mit der MATLAB-Funktion `princomp` durchgeführt und ergab die neuen Basisvektoren

$$\overrightarrow{p_1} = \begin{pmatrix} 0.5521 \\ 0.5780 \\ 0.6009 \end{pmatrix} \quad \overrightarrow{p_2} = \begin{pmatrix} -0.7207 \\ -0.0316 \\ 0.6926 \end{pmatrix} \quad \overrightarrow{p_3} = \begin{pmatrix} 0.4193 \\ -0.8154 \\ 0.3991 \end{pmatrix}.$$

Die Umwandlung von RGB-Werten in die PCA-Kanäle entspricht mathematisch einer Basistransformation. Da es sich bei der RGB-Basis um die kanonische Basis handelt, erfolgt die Umrechnung durch einfache Linearkombination mit

$$\begin{aligned} PCA_1 &= p_{1,1}R + p_{1,2}G + p_{1,3}B, \\ PCA_2 &= p_{2,1}R + p_{2,2}G + p_{2,3}B \text{ und} \\ PCA_3 &= p_{3,1}R + p_{3,2}G + p_{3,3}B. \end{aligned}$$

PCA_1, PCA_2 und PCA_3 werden jeweils auf $[-1, 1]$ skaliert und dann als drei Karten in das Netz eingegeben.

Interessanterweise repräsentiert $\overrightarrow{p_1}$ in etwa ein Graustufenbild, während $\overrightarrow{p_2}$ und $\overrightarrow{p_3}$ Farbdifferenzen codieren. Dies entspricht vom Prinzip her der Codierung des menschlichen Sehsystems, bei der ebenfalls ein Helligkeitskanal und zwei Farbdifferenzkanäle existieren [30].

In Tabelle 4.1 werden die Korrelationen der berechneten PCA-Kanäle mit denen der ursprünglichen RGB-Codierung verglichen. Zusätzlich wurde, motiviert durch [30], eine differenzielle Codierung mit den Kanälen Helligkeit $(R + G + B)$, Rot-Grün-Differenz $(R - G)$

Farbraum	**Korrelation von je 2 Kanälen**			**Durchschn. Korrel.**
	1 zu 2	1 zu 3	2 zu 3	
RGB	93 %	81 %	93 %	89 %
Differenziell	75 %	11 %	12 %	33 %
PCA	0 %	0 %	0 %	0 %

Tabelle 4.1: Paarweise Korrelation der drei Kanäle bei RGB-, PCA- und differenzieller Codierung auf der im Text genannten Stichprobe aus dem LabelMe-Datensatz. Die durchschnittliche Korrelation gibt das arithmetische Mittel der drei paarweisen Korrelationen an. Bei PCA liegen die Werte in der Größenordnung 10^{-15}.

und Blau-Gelb-Differenz $(0.5(R+G)-B)$ getestet. Auch hierbei ist die Korrelation deutlich geringer als bei der RGB-Codierung, aber wesentlich höher als bei der PCA-Codierung. Abbildung 4.5 zeigt die paarweise Korrelation der Kanäle auf der genannten Stichprobe bei RGB- und PCA-Codierung.

Zur Validierung der Ergebnisse wurden die Messungen mit einer größeren Stichprobe wiederholt, bei der die Diagonale *jedes* Bildes aus dem Datensatz verwendet wurde. Die resultierenden Hauptkomponenten wichen nur unwesentlich von den oben genannten ab. Größere Stichproben waren aufgrund des hohen Speicher- und Rechenzeitbedarfs nicht praktikabel. Weiterhin wurde als alternatives Verfahren *Independent Component Analysis* (*ICA*) [26] angewandt, das Ergebnis war allerdings geringfügig schlechter als bei PCA, weshalb diese Methode nicht weiter verfolgt wurde. Zur Berechnung wurde das `fastica`-Paket für MATLAB [33] benutzt.

4.3.2 Kantenfilter

Als weitere Eingabe dienen Gradienten in vier verschiedenen Orientierungen, mit anderen Worten Kantenfilter. Die Berechnung der Gradienten $g_1(x,y),\ldots,g_4(x,y)$ an der Position (x,y) erfolgt für jeden Pixel $p(x,y)$ mit

$$
\begin{aligned}
g_1(x,y) &= abs(gray(p(x,y)) - gray(p(x,y+1))),\\
g_2(x,y) &= abs(gray(p(x,y)) - gray(p(x+1,y))),\\
g_3(x,y) &= abs(gray(p(x+1,y)) - gray(p(x,y+1))) \text{ und}\\
g_4(x,y) &= abs(gray(p(x,y)) - gray(p(x+1,y+1))),
\end{aligned}
$$

wobei *gray* den Grauwert eines Pixels $(R+G+B)$ und *abs* den Absolutbetrag angibt. Auch hier werden – wie bei den PCA-Kanälen – die Eingaben auf $[-1,1]$ skaliert. Abbildung 4.6 zeigt die vier Eingabebilder, die sich durch obige Vorgehensweise aus einem Bild des LabelMe-Datensatzes ergeben.

Die Berechnung von Gradienten aus Graustufenbildern ist durch die Hypersäulen im primären visuellen Cortex motiviert, in denen ebenfalls orientierte Helligkeitsgradienten „berechnet“ werden.

4.3.3 Eingabe in mehreren Schichten

Sowohl PCA-Kanäle als auch Kantenfilter können in mehreren Schichten gleichzeitig eingegeben werden. Dazu wird das ursprüngliche Eingabebild entsprechend verkleinert, bevor PCA-Kanäle und Kanten als Eingabekarten berechnet werden. Bei Halbierung der Kartengröße in jeder Schicht genügt es, jeweils den Durchschnitt der RGB-Werte von vier Pixeln zu berechnen, um daraus den Wert des verkleinerten Pixels zu erhalten. Für die CUDA-Implementierung ist dieses Vorgehen güstig, da für die Kantenberechnung sowieso die Werte

Abbildung 4.6: Eingabecodierung mit PCA-Kanälen und Gradienten. (a) zeigt das Originalbild, (b)–(d) die PCA-Kanäle. Darunter sind (e) vertikale, (f) horizontale, (g) 45-Grad- und (h) 135-Grad-Kanten abgebildet. Bei den PCA-Kanälen repräsentiert Schwarz negative, Grau neutrale und Weiß positive Werte. Bei den Kanten repräsentiert Weiß vom Betrag her kleine, Schwarz vom Betrag her große Gradienten. Bildquelle: [43]

der drei Nachbarpixel geladen werden müssen. Somit kann das ursprüngliche Eingabebild sukzessive von Schicht zu Schicht verkleinert werden, während gleichzeitig die gewünschten Merkmale extrahiert werden.

Die Eingabe auf mehreren (oder sogar allen) Schichten bietet insbesondere für die Kantenextraktion Vorteile, da durch die sukzessive Verkleinerung des Bildes bei gleichbleibender Filtergröße auch Gradienten niedriger räumlicher Frequenz gefunden werden.

4.4 Ausgabe als 1-aus-N-Codierung

Die Ausgabe des neuronalen Netzes erfolgt in der Ausgabeschicht. Für jede trainierte Objektklasse existiert darin ein Ausgabeneuron, das Werte nahe 1 annehmen soll, wenn die repräsentierte Klasse in der Eingabe erkannt wurde. Ansonsten soll das Neuron Werte nahe -1 annehmen. Diese Codierung nennt sich *1-aus-N-Codierung* (weil höchstens eins von N Neuronen eine hohe Aktivierung aufweist) und bietet den Vorteil, dass die Anzahl der trainierten Klassen problemlos geändert werden kann. Da die Ausgabe einen reellen Wert im Bereich $[-1, 1]$ annimmt, kann außerdem der Schwellwert für die Erkennung beliebig festgelegt werden.

4.5 Überwachtes Lernen mit Backpropagation of Error

Zum Lernen während der Trainingsphase wird das Verfahren Backpropagation of Error verwendet. Die Lernrate steigt von der Ausgabeschicht zu Schicht 0 hin exponentiell an, um den kleiner werdenden zurückpropagierten Fehler zu kompensieren. Konkret berechnet sich die Lernrate für Verbindungen von Schicht $n-1$ zu Schicht n, $n \in \{1, \ldots, N-1\}$, als

$$\eta_n = \eta \cdot \lambda^{N-n}, \quad \lambda \geq 1{,}0,$$

wobei η die initiale Lernrate und N die Anzahl der Schichten exklusive der Ausgabeschicht bezeichnet. Für Verbindungen zwischen Schicht $N-1$ und der Ausgabeschicht (also Ausgabeverbindungen) wird als Lernrate

$$\eta_{Ausgabe} = \eta \cdot \lambda_{Ausgabe}, \quad \lambda_{Ausgabe} \ll 1{,}0$$

verwendet, weil durch die Vollverknüpfung eine kleinere Lernrate als bei den lokal verknüpften Kartenverbindungen benötigt wird.

4.5.1 Verwendung von Mini-Batches

Während der Trainingsphase wird eine Strategie verwendet, die zwischen Online-Learning und Batch-Learning liegt: *Mini-Batch-Learning*. Dabei werden je 16 Trainingsmuster parallel zuerst vorwärts und dann rückwärts propagiert. Die berechneten Gewichtsänderungen werden über die 16 Muster aufkumuliert und am Ende nach Multiplikation mit der Lernrate zu den aktuellen Gewichten addiert.

Der Lernerfolg bei der Verwendung von Mini-Batches ergibt sich bei herkömmlichem Backpropagation of Error beinahe so schnell wie bei Online-Learning und damit erheblich schneller als bei Batch-Learning (siehe dazu Abschnitt 6.3). Gegenüber Online-Learning bietet es den Vorteil, dass die Gewichte nur einmal pro Mini-Batch aus dem globalen Speicher geladen und in diesen geschrieben werden müssen, was die Laufzeit der CUDA-Implementierung drastisch beschleunigt. Weiterhin ist das Lernen numerisch stabiler, da zuerst 16 Gewichtsänderungen von ungefähr gleicher Größenordnung addiert werden, anstatt die Änderungen einzeln zu den betragsmäßig meist deutlich größeren Gewichten zu addieren. Die Größe der Mini-Batches ergibt sich durch Abwägungen bei der Implementierung, die in Kapitel 5 beschrieben sind.

4.5.2 Initialisierung der Gewichte

Bei der anfänglichen Wahl der Verbindungsgewichte aller Schichten sollte vermieden werden, dass Neuronen von vorne herein Aktivierungen im gesättigten Bereich aufweisen (beim Tangens Hyperbolicus als Aktivierungsfunktion wären das Werte nahe ± 1). Andernfalls ist die Steigung der Aktivierungsfunktion so gering, dass nur minimale Gewichtsveränderungen und somit beinahe kein Lernen stattfinden. Aus diesem Grund werden alle Gewichte so initialisiert, dass die Netzeingabe jedes Neurons auf einen sinnvollen Bereich beschränkt ist. Dazu wird für jede Karte k berechnet, wieviele eingehende Verbindungen jedes Neuron der Karte von Neuronen anderer Karten bekommt. Dieser Wert wird im Folgenden als $numinputs_k$ bezeichnet.

Ein Beispiel: Wenn jede Schicht die halbe Seitenlänge wie die vorhergehende Schicht aufweist und eine Karte k in Schicht n über genau eine Kartenverbindung zu einer Karte aus Schicht $n-1$ verfügt, so besitzt jedes Neuron aus k mindestens vier eingehende Verbindungen. Bei einem zusätzlichen Rand mit Breite 1 (also einer Überlappung der rezeptiven Felder um ein Neuron in jede Richtung) beträgt die Anzahl eingehender Verbindungen pro Neuron 16. Diese Situation ist in Abbildung 4.3 dargestellt. Bei mehreren Kartenverbindungen oder breiterem Rand erhöht sich dieser Wert entsprechend.

Gewichte eingehender Verbindungen einer Karte k werden deshalb pseudozufällig und gleichverteilt aus einem Intervall $[min_k, max_k]$ gewählt, wobei

$$max_k = \frac{NET\_MAX_f}{numinputs_k},$$
$$min_k = -max_k.$$

NET_MAX_f ist dabei eine Konstante, die von der gewählten Aktivierungsfunktion $f(\cdot)$ abhängt. Als Werte können beispielsweise

$$NET\_MAX_{tanh} = 10{,}0 \text{ und}$$
$$NET\_MAX_{Fermi} = 20{,}0$$

gewählt werden. Die maximal mögliche Netzeingabe eines Neurons beträgt somit offensichtlich $\pm NET\_MAX_f$. Da die Verbindungsgewichte aber ein arithmetisches Mittel von 0 haben, sind die resultierenden Netzeingaben im Allgemeinen betragsmäßig deutlich kleiner.

5 Implementierung

Dieses Kapitel beschreibt die Implementierung des Objekterkennungsystems in „Top-Down"-Reihenfolge. Abschnitt 5.1 gibt zunächst einen Überblick über die Struktur des Programms. In Abschnitt 5.2 wird dann die grafische Benutzeroberfläche vorgestellt, die der Visualisierung des neuronalen Netzes dient. Die restlichen Abschnitte erläutern detailliert die Implementierung performanzkritischer Funktionen: 5.3 beschreibt, wie die Daten des neuronalen Netzes im Speicher abgelegt werden, 5.5 und 5.6 gehen nach einigen Vorbemerkungen in 5.4 auf die CUDA-Implementierung der Rückwärts- beziehungsweise Vorwärtspropagierung ein.

5.1 Übersicht der Programmstruktur

Das Objekterkennungssystem wurde unter Linux als C/C++-Programm mit CUDA-Erweiterungen implementiert. Die Bestandteile des Programms lassen sich grob nach Funktionalität einordnen:

1. Grundfunktionen des in Kapitel 4 vorgestellten neuronalen Netzes. Dazu zählen Allokation und Initialisierung der Schichten, Karten und Kartenverbindungen inklusive Gewichtsinitialisierung, Vorwärts- und Rückwärtpropagierung, Setzen und Abfragen von Parametern sowie Hilfsfunktionen wie das Laden und Speichern von Gewichten oder die Synchronisierung des Netzes zwischen Host und Device.

2. Durchführung von Messungen, das heißt, Laden von Datensätzen, Trainieren des Netzes mit der Trainingsmenge und Testen mit der Testmenge.

3. Benutzeroberfläche und Visualisierung. Hier werden Befehle wie das Starten des Trainings verarbeitet. Weiterhin werden zum Beispiel die entsprechenden Fenster aktualisiert, wenn sich die Daten des neuronalen Netzes ändern (diese Funktionalität ist durch ein *Beobachter-Entwurfsmuster* [4] umgesetzt).

4. Hilfsfunktionen für Debugging und Fehlerbehandlung.

Die nachfolgenden Abschnitte dieses Kapitels widmen sich ausschließlich dem ersten Punkt. Anmerkungen zu Punkt zwei finden sich an den entsprechenden Stellen in Kapitel 6, weil sich die verwendeten Funktionen und Kernels je nach Datensatz unterscheiden. Auf die Punkte drei und vier wird nicht weiter eingegangen.

5.1.1 Grundfunktionen des neuronalen Netzes

Die Grundfunktionen des neuronalen Netzes sind auf wenige Klassen aufgeteilt, die in Abbildung 5.1 als UML-Diagramm dargestellt sind.

Eine Instanz der Hauptklasse `NeuralNetwork` kann mit Hilfe der Funktion `createLayer` beliebig viele Schichten erzeugen und zur internen Liste von Schichten `layers` hinzufügen. Mit `createOutputLayer` wird hingegen genau eine Ausgabeschicht erzeugt. Analog dazu sind, wie im Diagramm erkennbar, weitere `create...`-Funktionen in den anderen Klassen vorhanden, um hierarchisch Karten und Kartenverbindungen beziehungsweise vollverknüpfte Ausgabeverbindungen zu erzeugen.

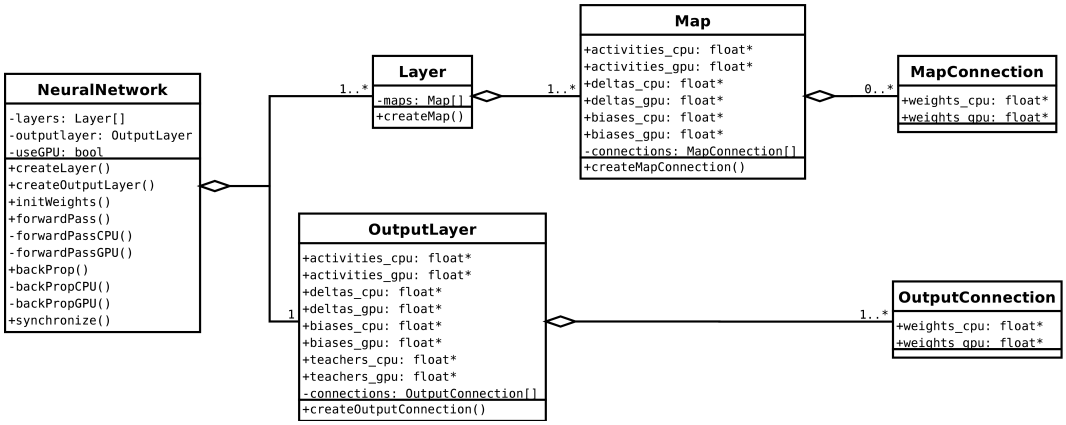


Abbildung 5.1: UML-Klassendiagramm der Hauptklassen des neuronalen Netzes. Es sind nur die für das Verständnis wichtigen Attribute und Operationen dargestellt. Funktionsparameter sind aus Gründen der Übersichtlichkeit ebenfalls nicht angegeben.

Jede Instanz einer Karte (`Map`) enthält verschiedene Attribute für jeweils alle ihre Neuronen, nämlich Aktivierungen, Fehlerwerte (Deltas) und Biases. Die Instanz der Ausgabeschicht (`OutputLayer`) enthält zusätzlich einen Teacher-Wert für jedes Neuron. Wenn die Grafikkarte zur Beschleunigung der Berechnung eingesetzt wird (`NeuralNetwork::useGPU` ist wahr), dann werden all diese Daten sowohl im Host- als auch im Devicespeicher abgelegt und können mit Hilfe der `synchronize`-Funktion synchronisiert werden. Sonst wird kein Speicher auf der Grafikkarte alloziert. Der Zugriff auf die verschiedenen Attribute der Neuronen einer Karte beziehungsweise der Ausgabeschicht erfolgt entgegen des Konzepts der Kapselung direkt über einen `float`-Zeiger, damit Schreib- und Leseoperationen effizient ausgeführt werden können. Gleiches gilt für Verbindungsgewichte, die bei Instanzen von `MapConnection` und `OutputConnection` enthalten sind.

Mit der Funktion `NeuralNetwork::initWeights` werden alle Verbindungsgewichte des Netzes wie in Kapitel 4 beschrieben initialisiert.

5.1.2 Performanzkritische Funktionen als CPU- und GPU-Variante

Eine besondere Bedeutung kommt den Funktionen `forwardPass` und `backProp` zu. Diese sind die einzigen Funktionen, die während der Trainings- und Testphase für jedes Muster aufgerufen werden und damit – neben der Trainings- oder Testfunktion – in höchstem Maße perfomanzkritisch sind. Mit anderen Worten hängt die Geschwindigkeit des neuronalen Netzes, abgesehen von der Anwendung des Sliding Windows und der konstanten Initialisierungszeit, fast ausschließlich von diesen beiden Funktionen ab. Aus diesem Grund wurden beide Funktionen mit CUDA implementiert, um durch die Nutzung der Grafikkarte eine höhere Ausführungsgeschwindigkeit zu erreichen.

Zur Verifikation der Korrektheit und zum Geschwindigkeitsvergleich wurden beide Funktionen auch als reine CPU-Variante – also ohne Verwendung von CUDA – implementiert. Bei Aufruf von `forwardPass` oder `backProp` wird abhängig vom Wert des Attributs `useGPU` dann `forwardPassCPU` oder `forwardPassGPU` beziehungsweise `backPropCPU` oder `backPropGPU` aufgerufen. Bei den CPU-Funktionen handelt es sich um „naive“ Implementierungen, die jeweils nur einen CPU-Kern nutzen und keine erweiterten Befehlssätze wie zum Beispiel MMX oder SSE verwenden. Durch Optimierungen wie zum Beispiel eine Parallelisierung über mehrere Kerne ließe sich die Geschwindigkeit wahrscheinlich um ein Mehrfaches steigern. Ziel der vorliegenden Arbeit ist es jedoch, mit der CUDA-Implementierung eine Geschwindig-

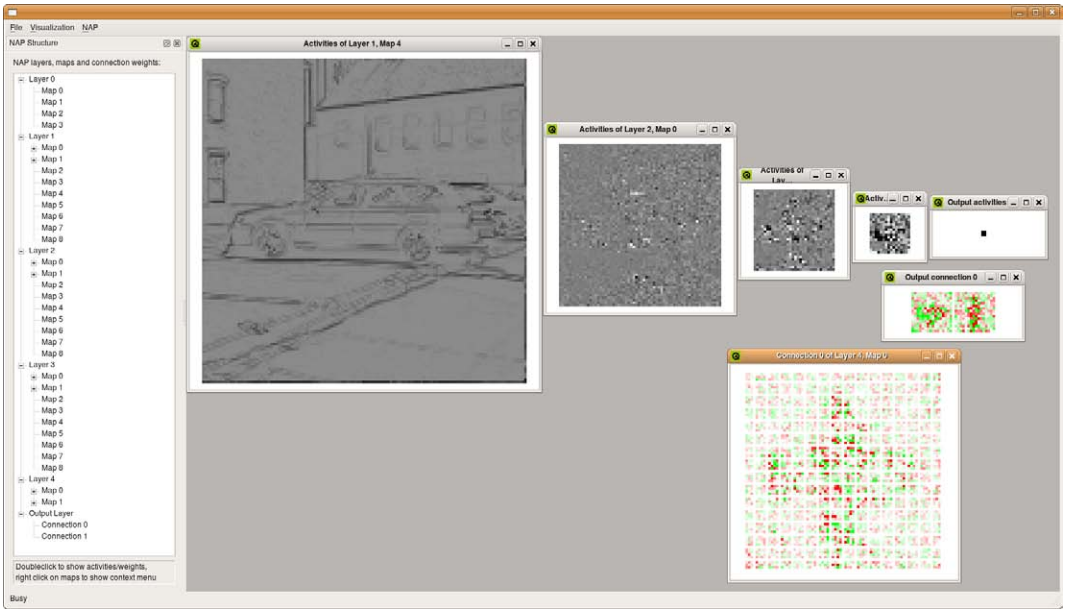


Abbildung 5.2: Grafische Benutzeroberfläche des Objekterkennungssystems. Auf der linken Seite befindet sich die Baumansicht der Schichten, Karten und Kartenverbindungen des Netzes. Im MDI-Bereich sind Aktivierungen (Graustufen) und Gewichte (Rot/Grün) verschiedener Schichten zu sehen.

keit zu erreichen, die wiederum um ein Vielfaches höher ist als die durch Optimierung der CPU-Variante erreichbare. Ein direkter Vergleich sowohl der Geschwindigkeit als auch der Ergebnisse beider Varianten ist in Kapitel 6 zu finden.

5.2 Grafische Benutzeroberfläche

Die grafische Benutzeroberfläche des Objekterkennungssystems nutzt das Qt-Framework von Trolltech[1]. Das Hauptfenster, welches in Abbildung 5.2 dargestellt ist, besteht aus einer Menüleiste, einer Baumansicht des neuronalen Netzes und einem MDI-Bereich (Multiple Document Interface). In letzterem können Aktivierungen, Fehlerwerte, Gewichte und Teacherwerte der aktuellen Trainings- oder Testmenge angezeigt werden.

Die Baumansicht enthält auf oberster Ebene die Schichten des Netzes inklusive der Ausgabeschicht. Auf der Ebene darunter befinden sich, außer bei der Ausgabeschicht, die zugehörigen Karten. Wiederum darunter befinden sich die eingehenden Kartenverbindungen jeder Karte beziehungsweise der Ausgabeschicht.

Durch Doppel- und Rechtsklick auf Karten lassen sich Aktivierungen und Fehler sowie bei der Ausgabeschicht zusätzlich Teacher-Werte anzeigen. Diese Visualisierungen werden in einem eigenen Fenster im MDI-Bereich geöffnet und lassen sich beliebig vergrößern oder verkleinern. Gleiches gilt für Gewichte, die durch Doppelklick auf Kartenverbindungen geöffnet werden.

In der Menüleiste befinden sich unter anderem Befehle zum Starten des Trainings, zum Speichern und Laden der Gewichte des neuronalen Netzes sowie zum Öffnen von Bildern, die mit der Sliding-Window-Technik auf Objekte untersucht werden sollen.

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

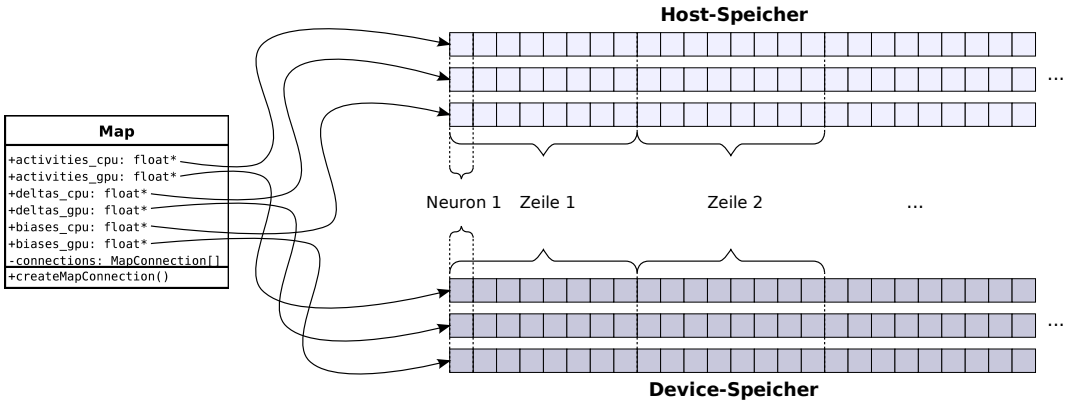


Abbildung 5.3: Speicherstruktur von Aktivierungen, Fehlern (Deltas) und Biases. Diese Variablen werden jeweils für alle Neuronen einer Karte zusammenhängend gespeichert. In der Ausgabeschicht existieren zusätzlich Teacher-Werte, die nach dem gleichen Schema im Speicher abgelegt sind. Bei Eingabekarten existieren nur Aktivierungen.

5.3 Speicherstruktur des neuronalen Netzes

Dieser Abschnitt beschreibt, wie die Parameter des neuronalen Netzes (beispielsweise Aktivierungen und Verbindungsgewichte einzelner Neuronen) im Host- beziehungsweise Devicespeicher abgelegt sind. Die Struktur wurde so gewählt, dass alle globalen Speicherzugriffe innerhalb der später beschriebenen Vorwärts- und Rückwärtspropagierungs-Kernels zusammenhängend erfolgen (siehe Abschnitt 2.4.5). Die Struktur der im Speicher abgelegten Daten ist für die GPU- und CPU-Variante jeweils gleich, um eine einfache und schnelle Synchronisation zwischen Host und Device zu ermöglichen.

5.3.1 Aktivierungen, Fehler, Biases und Teacher-Werte

Alle Neuronen einer Karte besitzen jeweils die Attribute Aktivierung, Fehler und Bias-Gewicht. Neuronen der Ausgabeschicht verfügen zusätzlich über einen Teacher-Wert. Eine Ausnahme bilden Neuronen in Eingabekarten, die nur das Attribut Aktivierung besitzen. Die Netzeingabe eines Neurons erfordert keine eigene Variable, stattdessen wird diese direkt in der Aktivierungs-Variable aufkumuliert, anschließend wird die Aktivierungsfunktion darauf angewandt.

Alle genannten Attribute werden durch eine Gleitkommazahl repräsentiert. Im Rahmen dieser Arbeit wird dafür der Datentyp `float` benutzt. Die Verwendung des Datentyps `double` ist zwar mit aktuellen CUDA-Grafikkarten ebenfalls möglich, benötigt aber doppelt so viel Speicher. Da die Speicherbandbreite – hauptsächlich wegen der hohen Anzahl an Verbindungsgewichten – einen Flaschenhals bei der Vorwärts- und Rückwärtspropagierung darstellt, wird deshalb ausschließlich der Datentyp `float` verwendet. Darüber hinaus ist auch das Rechnen mit Variablen vom Typ `double` deutlich langsamer, was sich allerdings in späteren Generationen von CUDA-Grafikkarten ändern soll.

Jedes Attribut ist für alle Neuronen einer Karte zusammenhängend und zeilenweise im Speicher abgelegt, siehe dazu Abbildung 5.3. Besitzt eine Karte zum Beispiel eine Seitenlänge von 256 und damit insgesamt $256 \cdot 256 = 65.536$ Neuronen, so zeigt `activities_cpu` auf einen $65.536 \cdot 4$ Byte großen Speicherbereich des Hosts (die Größe einer `float`-Variable beträgt 4 Byte). Falls `useGPU` wahr ist, zeigt zusätzlich `activities_gpu` auf einen ebenso großen Bereich des Devicespeichers. Die Speicherbereiche *verschiedener* Attribute stehen in keiner Beziehung zueinander, werden also alle einzeln alloziert.

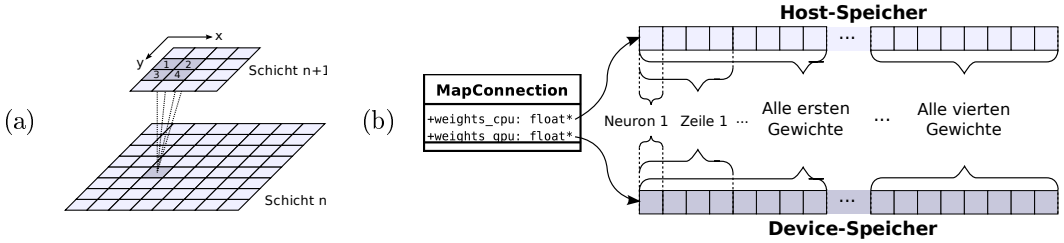


Abbildung 5.4: Speicherstruktur von Verbindungsgewichten (exklusive Biasgewichte). Zur Beschreibung siehe Abschnitt 5.3.2.

5.3.2 Verbindungsgewichte von Kartenverbindungen

Die Verbindungsgewichte einer Kartenverbindung werden ebenfalls zusammenhängend als `float`-Array gespeichert. Die Struktur ist hier allerdings weniger offensichtlich. Wie bereits in Kapitel 4 erwähnt, wurde für die CUDA-Implementierung festgelegt, dass jede Karte der Schicht $n+1$ genau die halbe Seitenlänge wie die Karten der Schicht n aufweist. Außerdem ist der Rand (also die Überlappung des rezeptiven Feldes) auf $r = 1$ festgelegt, siehe dazu Abbildung 4.3.

Daraus resultiert, dass jedes Neuron der Schicht $n+1$ zu 16 Neuronen der Schicht n verbunden ist (eine Ausnahme sind Neuronen am Rand oder in einer Ecke der Karte, die lediglich 12 beziehungsweise 9 Verbindungen aufweisen). Anders herum betrachtet besitzt jedes Neuron der Schicht n (bis auf die am Rand oder in einer Ecke) genau vier Verbindungen zu Neuronen der Schicht $n+1$, siehe Abbildung 5.4 (a).

Der Speicherbereich der Verbindungsgewichte ist dementsprechend in vier gleich große, direkt hintereinanderliegende Abschnitte unterteilt. Im ersten Abschnitt sind für alle Neuronen der Schicht n aufeinanderfolgend jeweils die Verbindungsgewichte zum entsprechenden Zielneuron mit dem kleinsten x- und y-Index gespeichert. Bei der Darstellung in Abbildung 5.4 (a) entspricht dieser Index jeweils der Verbindung zum linken oberen der vier Zielneuronen. Im zweiten der vier Abschnitte ist dann jedes zweite Verbindungsgewicht für alle Quellneuronen enthalten, und so weiter. Innerhalb jedes Abschnitts sind die Verbindungsgewichte so abgelegt, dass die zugehörigen Quellneuronen zeilenweise durchlaufen werden.

Die erläuterte Speicherstruktur ist in Abbildung 5.4 (b) dargestellt.

5.3.3 Verbindungsgewichte von Ausgabeverbindungen

Da Ausgabeverbindungen vollverknüpft sind, wird für sie keine so spezielle Speicherstruktur wie im letzten Abschnitt benötigt. Stattdessen sind die Verbindungsgewichte in einer Matrixform gespeichert, wie sie häufig auch bei Implementierungen von mehrschichtigen Perzeptrons zu finden ist. Jede Ausgabeverbindung verfügt dazu über eine Gewichtsmatrix der Form

$$W_k = \begin{pmatrix} w_k(1,1) & w_k(2,1) & \cdots & w_k(I,1) \\ w_k(1,2) & w_k(2,2) & \cdots & w_k(I,2) \\ \vdots & & & \vdots \\ w_k(1,J) & w_k(2,J) & \cdots & w_k(I,J) \end{pmatrix}. \tag{5.1}$$

Dabei ist k der Index einer Karte aus Schicht $N-1$, I ist die Anzahl Neuronen dieser Karte und J die Anzahl Neuronen in der Ausgabeschicht.

Die Speicherung dieser Matrix erfolgt, anders als in C/C++ üblich, nicht zeilenweise, sondern spaltenweise. Der Grund dafür ist, dass zur Berechnung der Vollverknüpfung die CUBLAS-Bibliothek [45] verwendet wird, die grundsätzlich spaltenweise gespeicherte Matrizen verarbeitet.

5.4 Vorbemerkungen zur parallelen Implementierung von Algorithmen mit CUDA

Bevor in den nächsten Abschnitten die konkrete Implementierung der Rückwärts- und Vorwärtspropagierung vorgestellt wird, werden an dieser Stelle noch einige generelle Anmerkungen zur möglichen Vorgehensweise bei der parallelen Implementierung von Algorithmen mit CUDA gemacht. Im Gegensatz zu Abschnitt 2.4 werden dem Leser hier anhand von Erfahrungen des Autors die Herausforderungen beim Entwurf von CUDA-Programmen vermittelt, so dass die nachfolgenden Abschnitte besser verständlich sind.

Grundsätzlich existieren verschiedene Möglichkeiten, einen Algorithmus mit CUDA zu parallelisieren. Die vermutlich häufigste Vorgehensweise ist die Parallelisierung von Schleifen, die im Algorithmus vorkommen. Dabei können jeweils einzelne oder mehrere Durchläufe auf einzelne oder mehrere Threads oder Blocks aufgeteilt werden. Die Threads können wiederum in drei Dimensionen aufgeteilt werden, die Blöcke in zwei Dimensionen. Voraussetzung für die Parallelisierung ist, dass die Berechnungen innerhalb der Schleife unabhängig voneinander ausgeführt werden können. Falls im Algorithmus mehrere Schleifen ineinander geschachtelt sind, kann eventuell über mehrere gleichzeitig parallelisiert werden.

Häufig muss die Aufteilung in Threads und Blöcke anders gewählt werden, als es der Algorithmus a priori nahelegt, weil entweder die Hardware-Ressourcen nicht ausreichen (siehe Tabelle 5.1) oder eine zu geringe Anzahl an Threads und Blöcken zu einer schlechten Auslastung der Grafikkarte (*Occupancy*) und somit zu einer schlechten Performanz führt.

Weitere Möglichkeiten, Schleifen im Algorithmus auf CUDA-Programme abzubilden, sind Schleifen innerhalb eines Kernels (und somit innerhalb jedes Threads) oder Kernelaufrufe in Schleifen.

Eigenschaft	**Maximale Größe**
Blockdimension (x×y×z)	512×512×64
Anzahl Threads pro Block insgesamt	512
Griddimension (x×y)	65.535×65.535

Tabelle 5.1: Maximale Block- und Griddimensionen bei aktuellen CUDA-Grafikkarten. Quelle: [47]

Eine zusätzliche Herausforderung ist die effiziente Verwendung des Speichers. Zugriffe auf den globalen Speicher sind um Größenordnungen langsamer als auf die On-Chip-Speicher. Sie stellen aber die einzige Möglichkeit dar, zwischen Blöcken zu kommunizieren oder große Datenmengen zu lesen und zu schreiben. Es sollte unbedingt darauf geachtet werden, dass Zugriffe zusammenhängend ausgeführt werden, da die Latenz linear in der Anzahl übertragener Segmente steigt. Bei reinen Lesezugriffen können Textur- und Konstantencache einen deutlichen Geschwindigkeitsgewinn gegenüber direkten Zugriffen auf den globalen Speicher bringen, wenn ihre Eigenschaften beachtet werden.

Für die Kommunikation zwischen den Threads eines Blocks bietet sich der Shared Memory an. Bei diesem muss darauf geachtet werden, keine Bankkonflikte beim Zugriff auszulösen, da die Ausführung einer Anweisung bei n Bankkonflikten innerhalb eines Half-Warps $(n+1)$-mal so lange dauert wie ohne Bankkonflikte. Daten, die nur innerhalb eines Threads benötigt werden, sollten in Registern abgelegt sein. Es hat sich herausgestellt, dass diese gegenüber Shared Memory oft deutlich schneller sind, obwohl in [47] angegeben ist, dass der Zugriff auf beide Speicherarten gleich schnell ist.

Tabelle 5.2 fasst die Eigenschaften der zur Verfügung stehenden Speicher zusammen.

Art des Speichers	Anzahl/Größe	On-Chip-Cache
Register	16.384 pro MP	—
Shared Memory	16 KB pro MP	—
Konstantenspeicher	64 KB	8 KB pro MP
Globaler Speicher/Texturspeicher	1 GB	Textur: 6–8 KB pro MP

Tabelle 5.2: Zur Verfügung stehende Speicher (MP=Multiprozessor). Quelle: [47]

5.5 CUDA-Implementierung der Rückwärtspropagierung

Im Rahmen der vorliegenden Arbeit wurde der Entwurf der Rückwärtspropagierung vor dem der Vorwärtspropagierung durchgeführt, da erstere mehr Speicherzugriffe und mehr Rechenoperationen erfordert und damit performanzkritischer ist. Aus diesem Grund wird sie hier auch als erstes beschrieben.

5.5.1 Analyse der Rückwärtspropagierung

Zur sukzessiven Minimierung des Gesamtfehlers (siehe Abschnitt 2.2.2) werden in der Trainingsphase alle Trainingsmuster nacheinander jeweils zuerst vorwärts und dann rückwärts propagiert. Die Rückwärtspropagierung dient dabei der Veränderung aller Verbindungsgewichte, so dass der Einzelfehler des aktuellen Musters per Gradientenabstieg verringert wird.

Berechnung des Fehlers

Dazu wird, beginnend bei der Ausgabeschicht, pro Neuron ein Wert berechnet, der als *Fehler* oder *Delta-Wert* bezeichnet wird. Für Neuronen der Ausgabeschicht berechnet sich dieser Fehler als

$$\delta(x) = (t(x) - a(x)) \cdot tanh'(net(x)).$$

Die Nomenklatur zu dieser und den nächsten Formeln ist in Tabelle 5.3 angegeben. Nachdem der Fehler für alle Neuronen einer Schicht $n \geq 1$ berechnet ist, kann er für die Neuronen der Schicht $n-1$ berechnet werden. Diese „Rückwärtspropagierung des Fehlers“ verleiht dem Lernverfahren seinen Namen. Der Fehler der Neuronen einer Karte i in der Schicht $n-1$ hängt dabei von allen Karten $j \in J$ aus Schicht n ab, die durch eine Kartenverbindung mit

Datenelement	Variable/Indexierung
Reguläre Schichten des Netzes	$n_0, \ldots, n_{N-1}$
Aktivierung eines Ausgabeneurons	$a(x)$
Netzeingabe eines Ausgabeneurons	$net(x)$
Fehler eines Ausgabeneurons	$\delta(x)$
Teacher eines Ausgabeneurons	$t(x)$
Aktivierung eines Neurons der Karte i	$a_i(x, y)$
Netzeingabe eines Neurons der Karte i	$net_i(x, y)$
Fehler eines Neurons der Karte i	$\delta_i(x, y)$
Verbindungsgewicht zwischen Karte i und j	$w_{ij}((x, y), (x', y'))$
Lernrate für Schicht n	η_n

Tabelle 5.3: Nomenklatur für die Beschreibung der Rückwärtspropagierung. Neuronen aller regulären Karten werden mit ihrer x- und y-Position auf der Karte indexiert. Ausgabeneuronen werden eindimensional mit dem Index x adressiert.

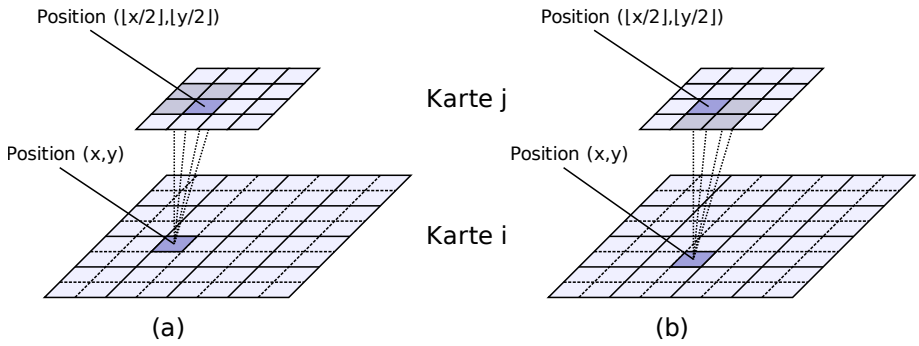


Abbildung 5.5: Veranschaulichung der Rückwärtspropagierung des Fehlers. Das markierte Neuron in Karte i ist Bestandteil der rezeptiven Felder der markierten Neuronen von Karte j (vergleiche dazu auch Abbildung 4.3).

i verbunden sind:

$$\delta_i(x,y) = tanh'(net_i(x,y)) \cdot \sum_{j \in J} \left(\sum_{(x',y') \in C_{ij}^B(x,y)} \delta_j(x',y') \cdot w_{ij}((x,y),(x',y')) \right). \tag{5.2}$$

Dabei ist $C_{ij}^B(x,y)$ die *Verbindungsfunktion* für die Rückwärtspropagierung (C für „Connectivity Function“, B für „Backpropagation“). Sie gibt die Menge der Positionen aller Neuronen der Karte j an, die durch eine Karten- oder Ausgabeverbindung mit dem Neuron (x,y) aus Karte i verbunden sind.

Sei nun j die Ausgabeschicht und i eine Karte der Schicht $N-1$. Dann ist

$$C_{ij}^B(x,y) = \{(0),(1),\ldots,(M)\},$$

wobei M die Anzahl von Ausgabeneuronen ist. Das Neuron (x,y) der Karte i ist also zu allen Ausgabeneuronen verbunden. Für alle Kartenverbindungen zwischen zwei regulären Karten i und j gilt

$$C_{ij}^B(x,y) = \left\{ \left(\left\lfloor \frac{x}{2} \right\rfloor + \alpha, \left\lfloor \frac{y}{2} \right\rfloor + \beta \right) \right\}, \tag{5.3}$$

mit

$$\alpha \in \begin{cases} \{-1,0\} & \text{, falls } x \bmod 2 = 0 \\ \{0,1\} & \text{, falls } x \bmod 2 = 1 \end{cases} \quad \text{und} \quad \beta \in \begin{cases} \{-1,0\} & \text{, falls } y \bmod 2 = 0 \\ \{0,1\} & \text{, falls } y \bmod 2 = 1 \end{cases}.$$

$C_{ij}^B(x,y)$ gibt also die Neuronen zurück, deren rezeptive Felder das Neuron (x,y) einschließen. Abbildung 5.5 veranschaulicht die Formel an zwei Beispielen.

Berechnung der Gewichsveränderung

Zur Verkleinerung des Einzelfehlers durch Gradientenabstieg werden für alle Karten- und Ausgabeverbindungen die Gewichte angepasst. Für eine Verbindung von Karte i und Karte j, wobei j sich in der Schicht $n \geq 1$ und i in der Schicht $n-1$ befindet, werden dafür die Fehler aus Schicht j und die Aktivierungen aus Schicht i benötigt:

$$\Delta w_{ij}((x,y),(x',y')) = \eta_n \cdot \delta_j(x',y') \cdot a_i(x,y).$$

Aufgrund der Verwendung von Mini-Batches in der hier beschriebenen Implementierung erfolgt die Änderung der Verbindungsgewichte zwischen zwei Schichten nicht sofort, stattdessen werden alle Gewichtsänderungen Δw_{ij} zunächst über 16 Muster akkumuliert und danach erst zu den Gewichten w_{ij} addiert. Die Begründung dafür wird in Abschnitt 5.5.2 gegeben.

Konkrete Größenangaben für benötigte Daten

Tabelle 5.4 zeigt alle Variablen, die zur Ausführung einer Rückwärtspropagierung für eine Kartenverbindung benötigt werden. Die Quellkarte hat hier eine Größe von 256×256 Neuronen, die Zielkarte 128×128.

Daten	Größe in Byte
Fehler (Deltas) der Karte j	$128 \cdot 128 \cdot 4 = 65.536$
Biasgewichte der Karte j	$128 \cdot 128 \cdot 4 = 65.536$
Fehler (Deltas) der Karte i	$256 \cdot 256 \cdot 4 = 262.144$
Aktivierungen der Karte i	$256 \cdot 256 \cdot 4 = 262.144$
Verbindungsgewichte (exklusive Biases)	$128 \cdot 128 \cdot 16 \cdot 4 = 1.048.576$

Tabelle 5.4: Konkrete Größe der benötigten Daten bei einer Kartenverbindung von einer Karte i der Größe 256×256 zu einer Karte j der Größe 128×128

5.5.2 Entwurf der Parallelisierungsmethode

Keine der Variablen aus Tabelle 5.4 passt komplett in einen schnellen Speicher oder einen Cache eines Multiprozessors. Da der globale Speicher aber um Größenordungen langsamer ist als die schnellen Speicher, ist ein Verzicht auf letztere nicht sinnvoll. Somit muss die Berechnung einer Kartenverbindung auf jeden Fall so aufgeteilt werden, dass verschiedene räumliche Teile der Kartenverbindung entweder nacheinander auf einem Multiprozessor oder parallel auf mehreren Multiprozessoren berechnet werden. Die erstgenannte Variante ist allerdings nicht sinnvoll, weil im Normalfall nicht genügend Kartenverbindungen zwischen zwei Schichten existieren, um alle Multiprozessoren effizient auszulasten. Eine Parallelisierung über mehrere Schichten kann nicht erfolgen, weil die Fehler bei der Rückwärtspropagierung Schicht für Schicht ausgerechnet werden müssen. Des Weiteren ergeben sich folgende Überlegungen:

Hohe Anzahl Threads und Blöcke

Um eine hohe Auslastung (*Occupancy*) der Grafikkarte sicherzustellen, müssen möglichst zu jeder Zeit genügend Threads und Blöcke ausgeführt werden. Da eine Parallelisierung über Kartenverbindungen wie oben erläutert höchstens eingeschränkt möglich ist, bietet sich eine Parallelisierung über die Neuronen der Quell- oder Zielkarte an. Eine weitere Möglichkeit wäre die zusätzliche Parallelisierung über sehr viele Muster. Dann würden aber die verarbeiteten Batches deutlich größer werden, weil die Aktualisierung der (enorm vielen) Verbindungsgewichte lediglich über den globalen Speicher erfolgen kann und damit verhältnismäßig langsam ist. Darüber hinaus können Synchronisationsprobleme entstehen, weil verschiedene Blocks im Allgemeinen nicht schreibend auf die gleichen Speicheradressen zugreifen dürfen (dieses Problem liesse sich allerdings mit *Atomic*-Funktionen [47] lösen). Fazit: Eine Parallelisierung über Neuronen (der Quell- und/oder der Zielkarte) und eventuell noch weiteren Dimensionen ist die einzige Lösung, die eine ausreichende Anzahl Threads und Blöcke garantiert.

Gewichte mehrmals verwenden

Wie in Tabelle 5.4 zu sehen ist, machen die Verbindungsgewichte vom Speicherplatzbedarf den größten Anteil der benötigten Daten bei der Rückwärtspropagierung aus. Außerdem müssen sie während der Rückwärtspropagierung sowohl gelesen als auch geschrieben werden. Es ist daher naheliegend, die Gewichte möglichst effizient zu nutzen, das heißt, sie

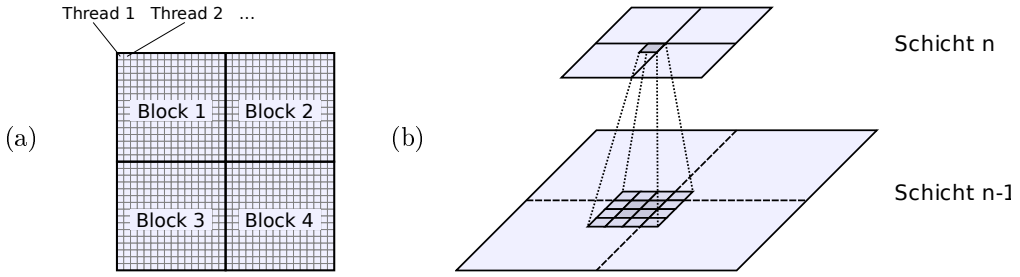


Abbildung 5.6: Parallelisierungsmethode der Rückwärtspropagierung. (a) Jeder Thread repräsentiert ein Neuron der größeren Karte der Kartenverbindung. Jeder Block repräsentiert ein Quadrat von 16×16 Neuronen. (b) Das rezeptive Feld eines Neurons der kleineren Karte kann auf mehrere Blöcke aufgeteilt sein. Bei einer Parallelisierung über Neuronen der kleineren Karte käme es deshalb zu Synchronisationsproblemen zwischen den Blöcken, weil Die Fehler von Neuronen am Rand der größeren Karte dann von mehreren Blöcken verändert werden müssten.

möglichst selten zu laden und zu speichern und dazwischen in einem schnellen Speicher zu halten. In der Praxis haben sich Register als die schnellste Speicherart erwiesen. Ideal ist daher die Verwendung von Registern für die Verbindungsgewichte. Dieses Argument legt nahe, dass einzelne Threads Neuronen einer bestimmten Position repräsentieren und mehrere Muster eines Mini-Batches *nacheinander* verarbeiten. Somit müssen die Gewichte nur einmal geladen und einmal gespeichert werden, da jedes Neuron über eigene Gewichte verfügt, auf die von keinem anderen Neuron zugegriffen werden muss.

Parallelisierung über die Neuronen der größeren Karte

Für Kartenverbindungen zwischen zwei Schichten n und $n-1$ kann die Rückwärtspropagierung aus zwei Richtungen betrachtet werden. Die eine Sichtweise ist, dass alle Karten der Schicht n ihren Fehler weiter auf die jeweils verbundenen Karten der Schicht $n-1$ propagieren. Die andere Sichtweise ist, dass alle Karten der Schicht $n-1$ ihre eigenen Fehler berechnen, indem sie die Fehler aller verbundenen Karten der Schicht n abfragen. Abhängig von der Sichtweise wäre es also möglich, eine Parallelisierung entweder über die Neuronen der Karte der aktuellen Quellschicht n oder der aktuellen Zielschicht $n-1$ zu implementieren. Die erste Möglichkeit ist allerdings nicht gut realisierbar, weil Probleme beim Schreiben der berechneten Fehler der Zielkarte entstehen: Da wie bereits begründet eine Aufteilung einzelner Kartenverbindungen in mehrere Threads und Blöcke notwendig ist, müssten aufgrund des Randes der lokalen Verknüpfungen gewisse Fehlerwerte zwischen verschiedenen Blöcken synchronisiert werden (siehe dazu Abbildung 5.6 (b)). Das ist aber nicht möglich, weil zur Entwicklungszeit keine Annahmen darüber getroffen werden können, wann verschiedene Blöcke ausgeführt werden. Aus diesem Grund ist eine Parallelisierung über die Neuronen der Zielkarte besser geeignet. Hier entstehen blockübergreifende Zugriffe nur für die Fehler der Quellkarte (vergleiche Abbildung 5.5). Diese müssen aber nur gelesen und nicht geschrieben werden, so dass es keine Synchronisationsprobleme gibt.

Eine sinnvolle Parallelisierungsmöglichkeit ist eine räumliche Einteilung der Zielkarte in Rechtecke, wie es in Abbildung 5.6 (a) dargestellt ist. Da Variablen wie Aktivierung und Fehler für die Neuronen einer Karte jeweils zeilenweise im Speicher abgelegt sind, ist auf diese Weise ein zusammenhängender Lese- und Schreibzugriff auf alle Variablen der Karte möglich. Die Breite eines solchen Blocks sollte mindestens 16 Neuronen betragen. Damit kann die maximale Speicherbandbreite genutzt werden, da alle Threads eines Half-Warps einen zusammenhängenden Speicherzugriff ausführen.

Einschränkungen durch die lokale Verknüpfung

Bei der beschriebenen Parallelisierung über die Neuronen der größeren Karte tritt das Problem auf, dass Fehler der kleineren Karte aufgrund der lokalen Verknüpfung mit Rand mehrmals gelesen werden müssen. Die größere Karte besitzt die doppelte Seitenlänge und damit viermal so viele Neuronen wie die kleinere Karte. Da jedes Neuron der größeren Karte zu vier Neuronen der kleineren Karte verbunden ist, wird jeder Fehlerwert der kleineren Karte insgesamt 16 mal gelesen und sollte daher in einem schneller Speicher abgelegt sein.

Zur Lösung existieren zwei Alternativen: Zum einen könnten alle Fehlerwerte zunächst in den Shared Memory geladen werden, zum anderen kann der Texturspeicher und -cache verwendet werden. Beide Ansätze wurden verglichen. Letzterer stellte sich dabei als deutlich schneller heraus und wird deshalb verwendet.

Formulierung der resultierenden Parallelisierungsmethode

Die Parallelisierung erfolgt bei der Rückwärtspropagierung über die Neuronen der größeren Karte einer Kartenverbindung. Verschiedene Kartenverbindungen werden in unterschiedlichen Kernelaufrufen bearbeitet. Jeder Kernelaufruf und somit jeder Thread bearbeitet statt eines einzelnen Musters einen Mini-Batch aus 16 Mustern. Auf diese Weise müssen die Verbindungsgewichte pro Batch nur einmal global geladen und gespeichert werden. Im Gegensatz zu Online-Learning ergibt sich daraus ein drastischer Geschwindigkeitsvorteil.

Die Batches könnten selbstverständlich auch größer gewählt werden. Pro Muster muss allerdings ein komplettes Abbild aller Karten im Speicher der Grafikkarte abgelegt sein, da die Rückwärtspropagierung jeweils alle von der Vorwärtspropagierung berechneten Aktivierungen beziehungsweise Netzeingaben benötigt. Bei großen Netzen ergibt sich dadurch ein hoher Speicherbedarf von mehreren hundert MB. Außerdem haben größere Batches den Nachteil, dass in der Praxis meist mehr Epochen trainiert werden muss, um gleich gute Ergebnisse zu erzielen. Deshalb wurde als Abwägung zwischen Größe und Effizienzgewinn eine Batchgröße von 16 Mustern gewählt.

Bei der Ausführung des Rückwärtspropagierungs-Kernels wird die größere Karte der aktuellen Kartenverbindung in Quadrate von je 16×16 Neuronen eingeteilt. Jedes dieser Quadrate wird in einem Block mit dementsprechend 256 Threads (einer pro Neuron) bearbeitet. Die Größe der Quadrate wurde gewählt, weil damit Karten bis zu einer Minimalgröße von 16×16 Neuronen bearbeitet werden können. Kleinere Karten (abgesehen von der Ausgabeschicht) sind nicht vorgesehen. Zwar könnte man die Quadrate trotzdem kleiner wählen oder alternativ auch nicht-quadratische Rechtecke verwenden, dies bringt aber geringfügige Nachteile mit sich, weil Fehlerwerte der kleineren Karte aufgrund des anteilig größeren Randbereichs von Blöcken häufiger in den Texturcache geladen werden müssen. Größere Quadrate sind grundsätzlich nicht möglich, weil die Anzahl Threads eines Blocks auf 512 beschränkt ist.

Die Wahl der Quadratgröße führt dazu, dass die Seitenlänge aller Karten ein Vielfaches von 16 sein muss. Diese Forderung ist erfüllt, weil die kleinste Karte vor der Ausgabeschicht immer eine Größe von 16×16 hat und sich die Seitenlänge in jeder darunterliegenden Schicht verdoppelt.

Listing 2 ordnet die Rückwärtspropagierung in den Kontext der gesamten Trainingsphase ein und gibt ihren groben Ablauf an. Die Vorwärtspropagierung `forwardPassGPU` wird später in Abschnitt 5.6 beschrieben.

Bewertung der Methode

Die beschriebene Methode garantiert einen nahezu optimalen Speicherzugriff auf alle benötigten Variablen bei der Verarbeitung einer Kartenverbindung und minimiert damit den stärksten Flaschenhals der Rückwärtspropagierung – die benötigte Speicherbandbreite.

Listing 2 Übersicht der Trainingsphase, Teil 1 von 2

```
1  function trainingGPU()
2      for each Mini-Batch do
3          forwardPassGPU(); // processes 16 patterns at once
4          backPropGPU();    // processes 16 patterns at once
5      end for
6  end function
7
8  function backPropGPU()
9      process output layer; // not described in detail here
10     for layer = N-1 to 1 do
11         for each map in layer do
12             for each downward connection in map do
13                 backPropKernel(connection);
14             end for
15         end for
16     end for
17 end function
18
19 kernel backPropKernel(connection)
20     load weights from global memory into registers;
21     for pattern = 1 to 16 do
22         load source deltas from texture memory into registers;
23         calculate and save target delta to global memory;
24         calculate weight changes;
25     end for
26     apply weight changes and save weights to global memory;
27 end kernel
```

Ein Nachteil der Methode ist allerdings, dass ein Kernelaufruf immer nur eine Kartenverbindung bearbeitet. Bei den kleinstmöglichen Karten (16×16 Neuronen) wird dabei ein Kernel mit nur einem Block ausgeführt. Bei mehreren Kartenverbindungen zwischen zwei Schichten n und $n-1$ bedeutet das außerdem, dass Fehlerwerte von Karten der Schicht $n-1$ mehrmals aktualisiert und damit auch mehrmals gelesen und geschrieben werden müssen, falls sie mit mehrern Karten der Schicht n verbunden sind (Abbildung 5.7 links). Weiterhin müssen so auch Fehlerwerte der Schicht n mehrmals gelesen werden, wenn Karten dort mit mehreren Karten aus Schicht $n-1$ verbunden sind (Abbildung 5.7 rechts). Eine theoretische Möglichkeit zur Behebung dieses Nachteils wäre die gleichzeitige Bearbeitung von mehreren Kartenverbindungen, die eine gemeinsame Quell- oder Zielkarte besitzen, in einem Kernelaufruf. Diese könnte als Schleife um alle Anweisungen des in Listing 2 gezeigten Codes des Kernels `backPropKernel` implementiert werden. Praktisch ist die Umsetzung allerdings nicht möglich, weil der schnelle On-Chip-Speicher deutlich zu klein ist, um neben den Gewichten einer Kartenverbindung auch noch alle benötigten Fehlerwerte der 16 Muster zu halten. Diese müssten also ohnehin für jede Kartenverbindung neu geladen werden. Bei der aktuellen Implementierung ist außerdem die Anzahl verwendeter Register voll ausgeschöpft, so dass keine weitere Schleife innerhalb des Kernels möglich ist, ohne dass Register vom Compiler in den langsameren lokalen Speicher ausgelagert werden (siehe Abschnitt 2.4.5). Die Speicherung von Gewichten oder Fehlerwerten im Shared Memory statt in den Registern führt allerdings zu einer deutlich höheren Ausführungszeit, die den zeitlichen Overhead

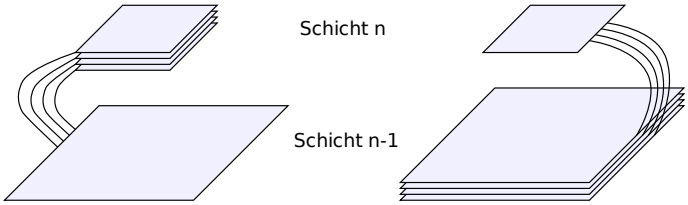


Abbildung 5.7: Mögliche Kartenverbindungen

beim Starten eines Kernels nicht kompensiert. Somit ist die vorgestellte Lösung die schnellste unter allen getesteten.

5.5.3 Beschreibung der backPropGPU-Funktion

Dieser Abschnitt beschreibt detaillierter die bereits in Abschnitt 5.1.2 eingeführte und in Listing 2 skizzierte `backPropGPU`-Funktion. Es handelt sich dabei um die Host-Funktion, welche die gesamte, CUDA-beschleunigte Rückwärtspropagierung für einen Mini-Batch, also 16 Muster, durchführt. Der Pseudocode der Funktion ist in Anhang A, Listing 11, angegeben.

Die Funktion `backPropGPU` ist in zwei Teile aufgeteilt. Der erste Teil berechnet die Rückwärtspropagierung für die vollverknüpften Ausgabeverbindungen, der zweite Teil berechnet die Rückwärtspropagierung für alle Kartenverbindungen.

Der erste Teil durchläuft in einer Schleife alle Ausgabeverbindungen. Im Normalfall existiert genau eine Ausgabeverbindung von jeder Karte der Schicht $N-1$ (also der letzten regulären Schicht) zur Ausgabeschicht. Jeder Schleifendurchlauf berechnet sowohl die Gewichtsveränderungen der aktuellen Verbindung als auch die Fehlerwerte der verbundenen Karte aus Schicht $N-1$ (die dafür benötigten Fehlerwerte der Ausgabeneuronen werden bereits am Ende der `forwardPassGPU`-Funktion berechnet, siehe dazu Abschnitt 5.6.3). Beide Berechnungen werden als Matrixmultiplikation mit Hilfe der CUBLAS-Bibliothek [45] durchgeführt. Dazu wird die Funktion `cublasSgemm` verwendet. Die Fehlermatrix $D_i = W^T \cdot D_j$ der Karte i wird berechnet, indem die transponierte Gewichtsmatrix W^T der aktuellen Ausgabeverbindung (siehe Gleichung 5.1) mit der Fehlermatrix D_j der Ausgabeschicht j multipliziert wird:

$$\underbrace{\begin{pmatrix} \delta_i^{p_1}(1) & \cdots & \delta_i^{p_{16}}(1) \\ \delta_i^{p_1}(2) & \cdots & \delta_i^{p_{16}}(2) \\ \vdots & & \vdots \\ \delta_i^{p_1}(I) & \cdots & \delta_i^{p_{16}}(I) \end{pmatrix}}_{D_i} = \underbrace{\begin{pmatrix} w_{ij}(1,1) & \cdots & w_{ij}(1,J) \\ w_{ij}(2,1) & \cdots & w_{ij}(2,J) \\ \vdots & & \vdots \\ w_{ij}(I,1) & \cdots & w_{ij}(I,J) \end{pmatrix}}_{W^T} \cdot \underbrace{\begin{pmatrix} \delta_j^{p_1}(1) & \cdots & \delta_j^{p_{16}}(1) \\ \delta_j^{p_1}(2) & \cdots & \delta_j^{p_{16}}(2) \\ \vdots & & \vdots \\ \delta_j^{p_1}(J) & \cdots & \delta_j^{p_{16}}(J) \end{pmatrix}}_{D_j}$$

Dabei ist beispielsweise $\delta_i^{p_{16}}(1)$ der Fehler der Karte i für das Muster 16 und das Neuron 1 (zeilenweise gezählt – eine zweidimensionale Indexierung ist hier zu unübersichtlich). $\delta_j^{p_{16}}(1)$ ist analog dazu der Fehler der Ausgabeschicht für das Muster 16 und das Ausgabeneuron 1. $w_{ij}(2,1)$ ist das Verbindungsgewicht zwischen Neuron 2 von Karte i und Ausgabeneuron 1.

Diese Form der Multiplikation bietet sich an, weil die Speicherstruktur der Fehlerwerte (vergleiche Abschnitt 5.3) genau dem von `cublasSgemm` erwarteten Format entspricht. Die Gewichtsmatrix muss zwar hier transponiert werden, dafür kann sie aber bei der später beschriebenen Vorwärtspropagierung unverändert verwendet werden. Dies ist vorteilhaft, weil in der Test- und Anwendungsphase ausschließlich die Vorwärtspropagierung benötigt wird. In der Trainingsphase werden Vorwärts- und Rückwärtspropagierung gleich oft aufgerufen.

Anschließend müssen die berechneten Fehler noch mit der Ableitung der Netzeingabe von Karte i multipliziert werden. Dies geschieht mit dem Kernel `deltaKernel`, welcher mit

Listing 3 Code der `deltaKernel`-Funktion

```
1  __global__ void deltaKernel(float* deltas, float* activities)
2  {
3      const unsigned int idx =
4          blockDim.x * blockDim.x * blockIdx.x +
5          blockDim.x * threadIdx.y +
6          threadIdx.x;
7
8      float act = activities[idx];
9      deltas[idx] *= 1.0f - act*act;
10 }
```

einem Thread für jeden Fehlerwert eines Musters und einem Block für jedes Muster ausgeführt wird. Statt der (nicht gespeicherten) Netzeingabe können bei Verwendung des Tangens Hyperbolicus als Transferfunktion die Aktivierungen zur Berechnung genutzt werden, da

$$\forall x \in \mathbb{R} \quad \tanh'(x) = 1 - \tanh^2(x). \tag{5.4}$$

Listing 3 zeigt den Code des Kernels. Die vorkommenden Lese- und Schreibzugriffe auf den globalen Speicher werden zusammenhängend ausgeführt, so dass der Kernel die maximale Speicherbandbreite ausnutzt.

Die Berechnung der Gewichtsänderungen von Ausgabeverbindungen erfolgt ebenfalls als Matrixmultiplikation mit `cublasSgemm`. Dazu werden die Fehlerwerte D_j der Ausgabeschicht mit den transponierten Aktivierungen A_i^T der Karte i multipliziert. Die resultierende Matrix wird skalar mit der Lernrate η multipliziert und das Ergebnis dann zur bisherigen Gewichtsmatrix W addiert. Insgesamt ergibt sich

$$\underbrace{\begin{pmatrix} w_{ij}(1,1) & \cdots & w_{ij}(1,I) \\ w_{ij}(2,1) & \cdots & w_{ij}(2,I) \\ \vdots & & \vdots \\ w_{ij}(J,1) & \cdots & w_{ij}(J,I) \end{pmatrix}}_{W} += \eta \cdot \underbrace{\begin{pmatrix} \delta_j^{p_1}(1) & \cdots & \delta_j^{p_{16}}(1) \\ \delta_j^{p_1}(2) & \cdots & \delta_j^{p_{16}}(2) \\ \vdots & & \vdots \\ \delta_j^{p_1}(J) & \cdots & \delta_j^{p_{16}}(J) \end{pmatrix}}_{D_j} \cdot \underbrace{\begin{pmatrix} a_i^{p_1}(1) & \cdots & a_i^{p_1}(I) \\ a_i^{p_2}(1) & \cdots & a_i^{p_2}(I) \\ \vdots & & \vdots \\ a_i^{p_{16}}(1) & \cdots & a_i^{p_{16}}(I) \end{pmatrix}}_{A_i^T}.$$

Im zweiten Teil der `backPropGPU`-Funktion findet die Rückwärtpropagierung für alle regulären Schichten statt. Die Schichten werden dabei rückwärts von $N-1$ bis 1 durchlaufen. Damit ist sichergestellt, dass Fehlerwerte einer Schicht n bereits aktualisiert sind, bevor die Berechnung in Schicht $n-1$ beginnt. Für jede Schicht werden alle Kartenverbindungen durchlaufen, deren kleinere Karte in der aktuellen Schicht liegt. Aus diesem Grund muss Schicht 0 nicht in der Schleife behandelt werden.

Für jede Kartenverbindung wird der Kernel `backPropKernel` ausgeführt, wobei jeder Block wie zuvor beschrieben 16×16 Neuronen repräsentiert. Es gibt so viele Blöcke, dass alle Neuronen der Quellkarte bearbeitet werden (vergleiche Abbildung 5.6 (a)). Der Kernel erhält als Parameter

- Die Gewichte der aktuellen Kartenverbindung
- Die Biasgewichte der aktuellen Karte, falls es sich um die erste bearbeitete Kartenverbindung dieser Karte handelt. Ansonsten wird ein Nullzeiger übergeben, der dem Kernel angibt, dass die Biases dieser Karte bereits aktualisiert wurden.
- Die Aktivierungen der Quellkarte

- Die Fehlerwerte der Quellkarte. Diese werden benötigt, falls bereits zuvor bearbeitete Kartenverbindungen zu der aktuellen Quellkarte verbunden waren und somit bereits ein Fehler zurückpropagiert wurde. Die Fehlerwerte der aktuellen Verbindung werden dann gemäß Gleichung 5.2 zu den bereits vorher berechneten addiert. Erst bei der letzten Kartenverbindung zur aktuellen Quellkarte wird der Fehler dann mit Hilfe von Formel 5.4 mit der Ableitung der Netzeingabe multipliziert. Auf diese Weise müssen die Aktivierungen der Quellkarte nur bei einer statt bei allen Verbindungen aus dem globalen Speicher geladen werden.
- Die Lernrate
- Eine `bool`-Variable, die angibt, ob es sich um die letzte zu bearbeitende Karte der aktuellen Schicht handelt. In diesem Fall wird vom Kernel, wie gerade beschrieben, die Ableitung der Netzeingabe mit den berechneten Fehlerwerten multipliziert. Es sei angemerkt, dass die im Code dargestellte Überprüfung, ob es sich um die letzte Verbindung handelt, nur dann korrekt funktioniert, wenn alle Karten der Schicht $n \in \{1, \ldots, N-1\}$ zu allen Karten der Schicht $n-1$ verbunden sind. Bei einer Aufhebung dieser Einschränkung ließe sich die Variable aber ebenfalls leicht berechnen.

5.5.4 Details zur Implementierung des backProp-Kernels

Die Listings 9 und 10 in Anhang A zeigen den gesamten, kommentierten Code des Rückwärtspropagierungs-Kernels. Da die Funktionsweise bereits ausführlich erläutert wurde, werden in diesem Abschnitt nur einige technische Details der Implementierung herausgegriffen.

Berechnung der Quellneuronen In den Zeilen 11 und 12 wird zunächst berechnet, welches Neuron der Zielkarte der aktuelle Thread repräsentiert. Die Zeilen 16 und 17 berechnen dann analog zu Formel 5.3 die Position der vier Neuronen, die mit dem repräsentierten Neuron verbunden sind.

Randbehandlung Neuronen am Rand der Zielkarte bilden eine Ausnahme, weil bis zu drei der berechneten Quellneuronen gar nicht existieren – sie lägen außerhalb der Quellkarte. Diese Neuronen werden in Zeile 32–36 ermittelt. Es hat sich als performanteste Lösung herausgestellt, diese Erkenntnis zunächst zu ignorieren und somit in der Hauptschleife über die 16 Muster des Mini-Batches keine Sonderbehandlung für nicht existierende Neuronen zu benutzen. Es existieren also auch Gewichte zu diesen Neuronen. Da die Fehlerwerte aller Quellneuronen aus dem Texturspeicher geladen werden, können für nicht existierende Neuronen undefinierte Werte zurückgegeben werden (der Kernel kann aber nicht abstürzen). Damit die berechneten Fehler trotzdem korrekt sind, müssen die Verbindungsgewichte zu diesen Neuronen den Wert 0 haben. Dies wird durch die Initialisierung sichergestellt (Funktion `NeuralNetwork::initWeights()`).

Laden und Speichern der Gewichte Aufgrund der in Abschnitt 5.3 beschriebenen Speicherstruktur können die Verbindungsgewichte (exklusive Biasgewichte) alle zusammenhängend aus dem globalen Speicher geladen (Zeile 20–24) und auch wieder dort gespeichert werden, nachdem die berechneten Gewichtsänderungen aller 16 Muster des Mini-Batches addiert wurden (Zeile 92–99). Gewichte zu nicht existierenden Neuronen dürfen aus den oben genannten Gründen nicht verändert werden, daher erfolgt beim Speichern eine Fallunterscheidung.

Biasgewichte werden nur dann verändert, wenn `biases` $\neq 0$ gilt, siehe dazu Abschnitt 5.5.3. Da es viermal so viele Threads wie Biasgewichte gibt (die Zielkarte besitzt viermal so viele Neuronen wie die Quellkarte), müssen nur die Threads ein Biasgewicht verarbeiten, für die

$x \bmod 2 = 0$ und $y \bmod 2 = 0$ gelten (Zeile 28–29), wobei x und y die Position des Neurons auf der Zielkarte darstellen. Diese Behandlung führt im Übrigen nicht zu Divergent Branches (siehe Abschnitt 2.4.5). Außerdem werden Lese- und Schreibzugriffe hier ebenfalls zusammenhängend und somit optimal ausgeführt.

5.6 CUDA-Implementierung der Vorwärtspropagierung

In diesem Abschnitt werden Analyse, Entwurf und Implementierung der Vorwärtspropagierung beschrieben.

5.6.1 Analyse der Vorwärtspropagierung

Die Vorwärtspropagierung ist insgesamt weniger komplex als die Rückwärtspropagierung. Es müssen für jede Kartenverbindung einer Quellkarte i zu einer Zielkarte j nur jeweils die Aktivierungen beider Karten sowie die Verbindungsgewichte (inklusive Biases) gelesen werden. Geschrieben werden nur die Aktivierungen der Zielkarte. Für eine Zielkarte j, die zu Quellkarten $i \in I$ verbunden ist, gilt mit der Nomenklatur von Tabelle 5.3:

$$net_j(x,y) = \sum_{i \in I} \left(\sum_{(x',y') \in C_{ij}^F(x,y)} w_{ij}((x',y'),(x,y)) \cdot a_i(x',y') \right)$$

und

$$a_j(x,y) = \tanh(net_j(x,y)).$$

Falls j die Ausgabeschicht ist, entfällt der Index y in beiden Formeln, weil die Neuronen der Ausgabeschicht eindimensional adressiert werden. $C_{ij}^F(x,y)$ ist die Verbindunsfunktion für die Vorwärtspropgierung (vergleiche Formel 5.3 in Abschnitt 5.5.1). Sie gibt die Menge der Positionen von Neuronen der Quellkarte i an, die sich im rezeptiven Feld des Neurons (x,y) aus Karte j befinden. Für Ausgabeverbindungen (also wenn j die Ausgabeschicht ist) ist offensichtlich jedes Neuron der Karte i zu jedem Ausgabeneuron x verbunden, es gilt also

$$C_{ij}^F(x) = \{(x',y') \mid x',y' \in \{0,\ldots,S_i - 1\}\},$$

wobei S_i die Seitenlänge der Karte i bezeichnet. Für alle Kartenverbindungen (bei denen lokale Verknüpfungen verwendet werden) ergibt sich

$$C_{ij}^F(x,y) = \{(2x+\alpha, 2y+\beta) \mid \alpha,\beta \in \{-1,0,1,2\}\}.$$

5.6.2 Entwurf der Parallelisierungsmethode

Für die Parallelisierung der Vorwärtspropagierung gelten die meisten Überlegungen aus Abschnitt 5.5.2 ebenfalls. Auch hier soll eine hohe Auslastung erreicht werden, so dass eine Aufteilung einzelner Karten und/oder Kartenverbindungen in mehrere Threads und Blöcke unausweichlich ist. Auch sollten Verbindungsgewichte in einem schnellen Speicher abgelegt sein und mehrmals verwendet werden, da sie den größten Teil der benötigten Daten ausmachen, nämlich 16-mal so viel wie die Aktivierungen der Zielkarte und 4-mal so viel wie die der Quellkarte. Weiterhin sind folgende Überlegungen von Bedeutung:

Parallelisierung über Neuronen der kleineren Karte

Während die Aktivierungen der größeren Karte nur gelesen werden, müssen die Aktivierungen der kleineren Karte bei der Vorwärtspropagierung gelesen und geschrieben werden.

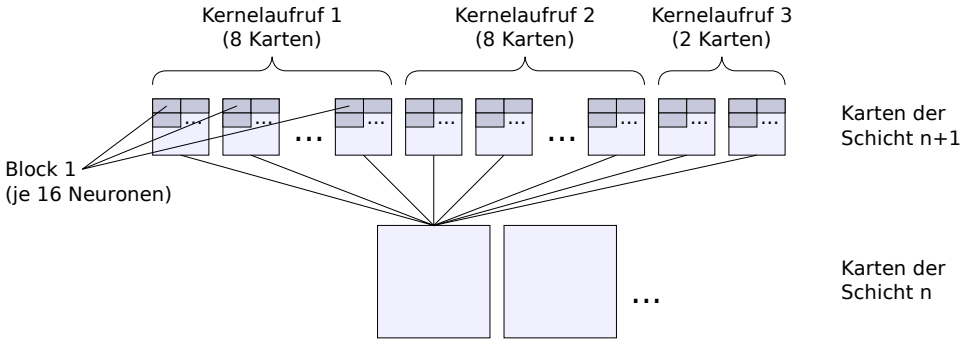


Abbildung 5.8: Funktionsweise des Vorwärtspropagierungs-Kernels. Jeder Kernelaufruf bearbeitet bis zu acht Karten der Schicht $n+1$, die mit einer bestimmten Karte aus Schicht n verbunden sind. In diesem Beispiel sind 17 Karten zur aktuellen Quellkarte der Schicht n verbunden, so dass insgesamt 3 Kernelaufrufe durchgeführt werden. Jeder Block bearbeitet dabei 16 Neuronen in einer Zeile jeweils von allen Karten. Des Weiteren werden alle 16 Muster eines Mini-Batches parallel bearbeitet, was hier nicht dargestellt ist.

Dies legt die Vorgehensweise nahe, dass jeder Thread eine Position der kleineren Karte repräsentiert. Wenn alle 16 Threads eines Half-Warps dann die Aktivierungen nebeneinanderliegender Neuronen einer Zeile lesen oder schreiben, ist ein zusammenhängender Zugriff auf den globalen Speicher sichergestellt.

Verbindungsgewichte im Shared Memory

Bei der Parallelisierung über die Neuronen der kleineren Karte tritt ein Problem auf: Wenn jeder Thread in je einer Anweisung die 16 Verbindungsgewichte lädt, die das durch ihn repräsentierte Neuron mit Neuronen der größeren Karte verbinden, dann kann der Zugriff auf den globalen Speicher nicht zusammenhängend erfolgen. Der Grund dafür ist, dass die Gewichte so im Speicher abgelegt sind, dass sie bei einer Parallelisierung über Neuronen der größeren Karte zusammenhängend gelesen werden können (vergleiche dazu die Implementierung der Rückwärtspropagierung). Zur Lösung dieses Problems müssen die Gewichte zunächst so in den Shared Memory geladen werden, dass zusammenhängende Lesezugriffe entstehen und sie später ohne Bankkonflikte aus dem Shared Memory gelesen werden können.

Quellaktivierungen als Textur

Den zweitgrößten Flaschenhals stellt das Laden der Aktivierungen der größeren Karte dar. Jedes Neuron der kleineren Karte verfügt über 16 Verbindungen pro Kartenverbindung und benötigt somit 16 Aktivierungswerte der größeren Karte. Da sich innerhalb eines Half-Warps ein lokales, aber nicht zusammenhängendes Zugriffsmuster ergibt, bietet sich der Texturcache zum Laden der Quellaktivierungen an.

Parallele Bearbeitung mehrerer Zielkarten

Eine weitere Performanzverbesserung lässt sich erreichen, wenn neben den Verbindungsgewichten auch die geladenen Quellaktivierungen mehrfach verwendet werden. Dies lässt sich erreichen, indem innerhalb eines Threads beziehungsweise Blocks nicht nur alle Muster des aktuellen Mini-Batches bearbeitet werden, sondern zusätzlich auch mehrere Zielkarten (also mehrere Karten, die zu einer bestimmten größeren Karten verbunden sind, siehe Abbildung 5.7 links).

Listing 4 Übersicht der Trainingsphase, Teil 2 von 2

```
1  function forwardPassGPU()
2      for layer = 0 to N-2 do
3          for each map in layer do
4              process all upward connections of map, up to 8 at once
5              with each call of forwardPassKernel(connections);
6          end for
7      end for
8      process output layer; // not described in detail here
9  end function
10
11 kernel forwardPassKernel(connections)
12     load weights of all connections into shared memory;
13     determine how many loop iterations must be done to process all 16 patterns
14     (this number increases with the number of parallely processed connections);
15     for each required loop do
16         load source activities via texture memory;
17         load target activity;
18         calculate new target activity and save to global memory;
19     end for
20 end kernel
```

Formulierung der resultierenden Parallelisierungsmethode

Jeder gestartete Kernel bearbeitet Kartenverbindungen von einer Quellkarte der Schicht $n \in \{0, \ldots, N-2\}$ zu maximal acht verbundenen Zielkarten der Schicht $n+1$. Jeder Block bearbeitet dabei für alle Zielkarten und für alle Muster des Mini-Batches jeweils die gleichen 16 in einer Zeile nebeneinanderliegenden Neuronen. Die Methode ist in Abbildung 5.8 dargestellt und in Listing 4 als Pseudocode formuliert (siehe dazu auch Listing 2).

5.6.3 Beschreibung der forwardPassGPU-Funktion

Die `forwardPassGPU`-Funktion (Listing 6 in Anhang A) setzt die im vorhergehenden Abschnitt beschriebene Funktionalität um. Wie die `backPropGPU`-Funktion gliedert sie sich in zwei Teile, wobei der erste Teil alle Kartenverbindungen und der zweite Teil alle Ausgabeverbindungen bearbeitet.

Bearbeitung der Kartenverbindungen

Im ersten Teil werden in einer Schleife alle Schichten von 0 bis $N-2$ durchlaufen. Innerhalb jedes Schleifendurchlaufs werden alle Karten der aktuellen Schicht n hintereinander bearbeitet. Für jede Karte wird dabei ermittelt, zu wie vielen Karten der Schicht $n+1$ sie verbunden ist. Bis zu acht dieser Verbindungen werden dann mit einem Aufruf von `forwardPassKernel` gleichzeitig bearbeitet.

Die Speicheradressen der Verbindungsgewichte sowie der Aktivierungen, Biases und Fehlerwerte werden zuvor in eine Struktur im Konstantenspeicher kopiert. Diese Vorgehensweise ist im Gegensatz zur Übergabe der Adressen als Parameter für die Kernelfunktion übersichtlicher und ebenfalls sehr schnell.

Die Fehlerwerte werden für die Vorwärtspropagierung nicht benötigt. Sie werden durch den Kernel auf Null zurückgesetzt, weil sonst der nächste Aufruf des Rückwärtspropagierungs-

Kernels fehlerhafte Werte berechnen würde. In der Anwendungsphase lässt sich dieser Schritt mit einem Parameter deaktivieren.

Jeder Block des Kernels besteht aus 512 Threads, also der maximal möglichen Anzahl. Dies ergibt sich, weil sich die drei Dimensionen `threadDim.x`, `threadDim.y` und `threadDim.z` stets zu 512 addieren. Die erste Dimension, `threadDim.x`, repräsentiert die 16 in einer Zeile liegenden Neuronenpositionen. `threadDim.y` gibt die Anzahl der in diesem Kernelaufruf bearbeiteten Zielkarten an. Aufgrund des Kerneldesigns sind hier nur die Werte 2, 4 und 8 möglich. Für die Anzahl K von Kartenverbindungen von einer Karte der Schicht n zu Karten der Schicht $n+1$ muss also $K \bmod 8 \in \{0, 2, 4\}$ gelten.

Es sei angemerkt, dass ein weiterer Kernel implementiert wurde, für den diese Einschränkung nicht gilt. Dieser bearbeitet, genau wie der Kernel der Rückwärtspropagierung, mit jedem Aufruf nur eine Kartenverbindung. Die durchgeführten Messungen wurden jedoch alle mit dem zuerst beschriebenen Kernel durchgeführt, da dieser schneller ist und ohnehin immer mindestens zwei Karten pro Schicht verwendet wurden.

Die dritte Thread-Dimension, `threadDim.z`, wird für die Parallelisierung über die 16 Muster des aktuellen Mini-Batches genutzt. Je nach Anzahl bearbeiteter Karten können damit 4, 8 oder alle 16 Muster auf Threads aufgeteilt werden, da sonst die Maximalgröße von 512 Threads überschritten wird. Um trotzdem alle 16 Muster zu bearbeiten, existiert im Kernel eine Schleife, die dementsprechend 4, 2 oder 1 Iteration(en) ausführt.

Bearbeitung der Ausgabeverbindungen

Im zweiten Teil von `backPropGPU` werden die Ausgabeverbindungen bearbeitet. Dazu werden zunächst alle Aktivierungen der Ausgabeneuronen mit ihrem Biasgewicht initialisiert. Dies geschieht durch einen einfachen Kernel, der dem `deltaKernel` aus Listing 3 ähnelt und deswegen nicht weiter beschrieben wird. Anschließend werden in einer Schleife über alle Ausgabeverbindungen die endgültigen Netzeingaben mit Hilfe von Matrixmultiplikationen berechnet. Die Aktivierungsmatrizen sind dabei von der gleichen Form wie die Fehlermatrizen in Abschnitt 5.5.3. Als letzter Schritt werden mit Hilfe eines weiteren einfachen Kernels die Aktivierungsfunktionen auf die Netzeingaben der Ausgabeneuronen angewandt. Gleichzeitig werden deren Fehlerwerte berechnet und die Biasgewichte angepasst. Es wird also schon ein Teil der Rückwärtspropagierung durchgeführt, weil die Kombination aus Vorwärts- und Rückwärtspropagierung auf diese Weise insgesamt schneller ist. In der Anwendungsphase lässt sich dieser Schritt deaktivieren.

5.6.4 Details zur Implementierung des forwardPass-Kernels

Die Listings 7 und 8 in Anhang A zeigen den gesamten, kommentierten Code des Vorwärtspropagierungs-Kernels. Wie bereits bei der Beschreibung des Rückwärtspropagierungs-Kernels werden auch hier nur einige technische Details herausgegriffen, da die Funktionsweise bereits erläutert wurde.

Laden der Gewichte Die Codezeilen 11–45 dienen dem Laden der Verbindungsgewichte in den Shared Memory. Bereits beim Aufruf des Kernels wird so viel Shared Memory dynamisch alloziert, dass alle Gewichte geladen werden können. Bei K gleichzeitig verarbeiteten Zielkarten ergibt das $16 \cdot 16 \cdot K \cdot 4$ Bytes Shared Memory (16 Neuronen pro Karte mit je 16 Gewichten und je 4 Byte), bei $K = 8$ Zielkarten also maximal 8.192 Byte. Da 512 Threads existieren, können in jeder Iteration der Schleife die Gewichte von zwei Karten geladen werden. Dazu wird zunächst für jeden Thread berechnet, das wievielte (1–16) Gewicht des aktuellen Neurons (festgelegt durch `threadIdx.x`) geladen werden soll. Damit kann dann berechnet werden, in welchem der vier Abschnitte des Gewichtsarrays (siehe Abschnitt 5.3.2)

sich das Gewicht im Speicher befindet. Außerdem wird die Position auf der Quellkarte berechnet, zu der das gewünschte Gewicht verbindet (Zeile 32–33). Mit diesen Informationen lässt sich dann die Position des Gewichts im globalen Speicher berechnen. Alle Gewichte werden so in den Shared Memory geschrieben, dass sowohl beim Schreiben als auch beim Lesen Bankkonflikte vermieden werden. Gewichte, die zu nicht existierenden Neuronen verbinden (die außerhalb der Quellkarte liegen würden), werden mit Null initialisiert und haben damit keinen Einfluss auf die Aktivierung der Zielneuronen.

Laden der Quellaktivierungen Die Schleife in den Zeilen 60–105 durchläuft alle Muster des Mini-Batches. Je mehr Karten gleichzeitig verarbeitet werden, umso mehr Iterationen werden benötigt, weil insgesamt nur 512 Threads zur Verfügung stehen.

In jeder Iteration werden zuerst mit Hilfe des Texturcaches die 16 Quellaktivierungen geladen, die zur Berechnung der Aktivierung des aktuellen Neurons benötigt werden. Die Nutzung des Texturcaches ist hier sehr wichtig, weil sonst aufgrund der Überlappung der rezeptiven Felder viele Aktivierungen mehrmals aus dem globalen Speicher geladen würden.

Das Laden der Gewichte befindet sich in einer zusätzlichen Schleife über alle *gleichzeitig* verarbeiteten Muster. Auf diese Weise wird die Optimierung des Texturcaches auf lokale Zugriffe besser ausgenutzt, so dass der Ladevorgang wesentlich schneller ist. Alle 16 Quellaktivierungen werden in Registern gespeichert.

5.7 Implementierung des Sliding Windows

Wie bereits in Abschnitt 4.2 beschrieben, wird zur Erkennung von Objekten in beliebig großen Bildern eine Sliding-Window-Technik angewandt. Da die Fenstergröße durch die Größe der Eingabekarten des verwendeten Objekterkennungssystems fest vorgegeben ist, wird zur Erkennung verschieden großer Objekte das Eingabebild sukzessive skaliert. Die Skalierung wird immer mit dem gleichen Faktor $\texttt{scalefactor} \in (0, 1)$ durchgeführt, bis ein festgelegter Schwellwert unterschritten wird.

Im Rahmen dieser Arbeit wird $\texttt{scalefactor} = 0{,}9$ verwendet, das heißt, das Eingabebild wird in jeder Skalierungsstufe auf 90 % seiner aktuellen Größe verkleinert. Wenn die größere Dimension des Bildes (entweder Höhe oder Breite) 160 Pixel unterschreitet, wird der Algorithmus abgebrochen. Dieser Wert wird verwendet, weil Objekte in den Trainingsmustern des LabelMe-Datensatzes stets auf 160 Pixel in ihrer größeren Dimension skaliert sind (siehe dazu Abschnitt 6.7.1).

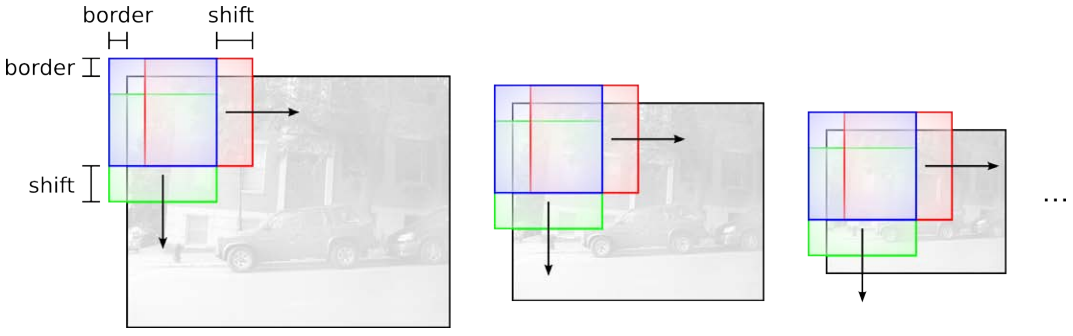


Abbildung 5.9: Funktionsweise der Sliding-Window-Technik. Auf jedes der sukzessive verkleinerten Eingabebilder werden Fenster einer festen Größe in einem festen Raster gelegt, die als Eingabe für das Objekterkennungssystem dienen. Das erste Fenster wird bei $x = y = -\texttt{border}$ platziert, alle weiteren werden dann jeweils `shift` Pixel nach rechts beziehungsweise unten verschoben.

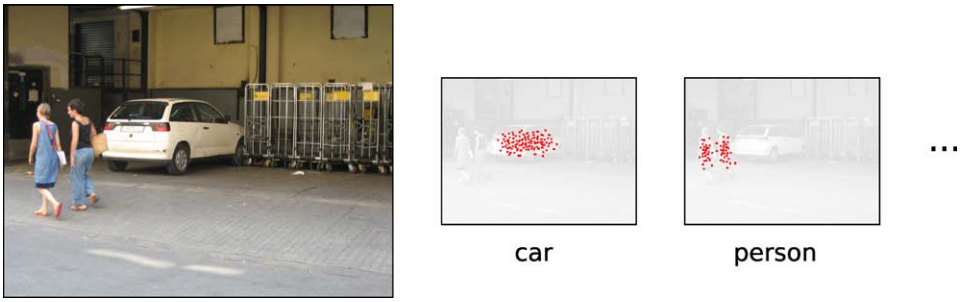


Abbildung 5.10: Darstellung eines Eingabebildes (links) und der erwünschten Ausgabe des Sliding-Window-Algorithmus in Form von Ausgabematrizen für jede Objektklasse (rechts). Die von Null verschiedenen Ausgabewerte sind in rot dargestellt. Bildquelle: [43]

Auf jeder Skalierungsstufe werden Fenster nacheinander auf verschiedene, durch ein Raster festgelegte Positionen des skalierten Eingabebildes gelegt. Diese Fenster bilden dann die Eingabe des Objekterkennungssystems. Die Abstände der Fenster zueinander sind, unabhängig von der Skalierung, stets gleich und werden durch den Parameter `shift` festgelegt.

Ein weiterer Parameter ist `border`. Dieser legt fest, wie weit das Fenster maximal über den Rand des Eingabebildes hinausragen darf. Werte größer Null sind dabei sinnvoll, damit auch Objekte am Bildrand erkannt werden können, die sonst in keinem Fenster zentriert wären.

Die beschriebene Vorgehensweise ist in Abbildung 5.9 dargestellt.

Ausgabecodierung

Als Ausgabe existiert für jede Objektklasse $m \in \{0, \ldots, M-1\}$ eine *Ausgabematrix*, die $\frac{1}{r}$-mal so groß ist wie das Eingabebild, $r \geq 1$. Bei einer Bildgröße von $x \times y$ Pixeln hat die Matrix $\lceil \frac{1}{r}x \rceil$ Spalten und $\lceil \frac{1}{r}y \rceil$ Zeilen. Die Werte aller Matrizen werden zu Beginn mit Null initialisiert.

Nach der Vorwärtspropagierung eines jeden Fensters wird für alle M Ausgabematrizen ein *Ausgabewert* $A_m(x', y')$ an der Matrixposition (x', y') gespeichert. Diese Position liegt in der Matrix relativ gesehen an der gleichen Stelle wie der Mittelpunkt des aktuellen Fensters (x, y) zur Gesamtgröße des Eingabebildes. Es ist also $x' = \lfloor \frac{1}{r}x \rfloor$ und $y' = \lfloor \frac{1}{r}y \rfloor$. Der Ausgabewert wurde im Rahmen dieser Arbeit folgendermaßen berechnet:

$$A_m^t(x', y') = \begin{cases} \min\left(\ 1,\ A_m^{t-1}(x', y') + (a^t(m) - 0{,}95) \cdot 10\ \right) & \text{, falls } a^t(m) \geq 0{,}95 \\ A_m^{t-1}(x', y') & \text{, sonst.} \end{cases}$$

Dabei steht t für den aktuellen Zeitschritt, $t-1$ gibt den Zeitpunkt nach Berechnung des vorhergehenden Fensters an. $a^t(m)$ steht für die Aktivierung des Ausgabeneurons m (für das aktuelle Fenster). Der Parameter t wird hier angegeben, weil durch die Anwendung des Sliding Windows in verschiedenen Skalierungen jeder Ausgabewert im Allgemeinen mehrmals aktualisiert wird. Außerdem können Ausgabewerte mehrerer Fenster bei großem r auf die gleiche Matrixposition fallen.

Die obige Formel besagt, dass der Ausgabewert erhöht wird, wenn die Aktivierung des Ausgabeneurons der aktuell betrachteten Objektklasse einen Schwellwert von 0,95 überschreitet. Dieser Wert hat sich experimentell als sinnvoll herausgestellt, um eine gute Abwägung zwischen False Positives und False Negatives zu erreichen. Die Erhöhung beträgt maximal $(1{,}0 - 0{,}95) \cdot 10 = 0{,}5$ pro vorwärtspropagiertem Fenster. Insgesamt kann der Ausgabewert nicht größer als 1 werden.

Abbildung 5.10 zeigt beispielhafte Ausgabematrizen.

Listing 5 Pseudocode des parallelisierten Sliding-Window-Algorithmus

```
1  function slidingWindow(image)
2      do
3          x = y = -border;
4          while (y+256 <= imageheight+border)
5              while (x+256 <= imagewidth+border)
6                  do forward pass for up to 16 windows in parallel
7                  (4 rows and 4 columns);
8                  for each processed window do
9                      for each object class do
10                         update value of the output matrix of this class;
11                     end for
12                 end for
13                 x += 4*shift;
14             end while
15             y += 4*shift;
16             x = -border;
17         end while
18         scale down image by scalefactor;
19     while (larger dimension of image > 160)
20 end function
```

Implementierungsdetails

Listing 5 zeigt den gesamten Sliding-Window-Algorithmus als Pseudocode. Durch die Verwendung von CUDA können, ähnlich wie in der Trainingsphase des Objekterkennungssystems, jeweils 16 Fenster parallel bearbeitet werden. Diese ergeben sich durch vier aufeinanderfolgende Fensterpositionen jeweils in x- und y-Richtung. Wenn ein oder mehrere der 16 Fenster über den zulässigen Rand hinausreichen, werden deren Ausgaben nicht weiterverarbeitet.

Die sukzessive Skalierung der Eingabebilder geschieht nicht auf der Grafikkarte, sondern auf dem Host. Dazu wird die Qt-Funktion `QImage::scaledToWidth` verwendet, die mit dem Parameter `Qt::SmoothTransformation` aufgerufen wird. Die Implementierung einer qualitativ hochwertigen Skalierung auf der Grafikkarte wäre im Rahmen dieser Arbeit zu zeitaufwändig gewesen.

Um den Sliding-Window-Algorithmus auch bei kleinen Eingabebildern anwenden zu können, werden diese auf eine Größe von 2048 Pixeln in ihrer größeren Dimension (entweder Höhe oder Breite) skaliert, falls sie ursprünglich kleiner sind.

Für jede Skalierungsstufe werden zunächst so viele um jeweils Faktor zwei verkleinerte Kopien des aktuellen (skalierten) Eingabebildes erzeugt, wie Eingabeschichten oberhalb von Schicht 0 existieren. Diese Verkleinerung geschieht mit Hilfe eines einfachen Kernels auf der Grafikkarte[2]. Jedes der Bilder wird dann in ein CUDA-Array [47] kopiert, weil diese es im Gegensatz zu konventionell alloziertem Speicher erlauben, zweidimensionale Texturreferenzen zu nutzen. Dadurch ist keine manuelle Randbehandlung erforderlich, weil bei Zugriffen außerhalb des gültigen Bereichs immer der Wert Null zurückgeben wird.

[2] Die Verkleinerung um Faktor zwei ist einfacher als eine beliebige Skalierung, weil sich die RGB-Werte eines Zielpixels aus dem arithmetischen Mittel der RGB-Werte von genau vier Quellpixeln ergeben. Bei einem beliebigen Skalierungsfaktor müssen zusätzlich die Koeffizienten für eine bilineare Filterung berechnet werden. Des Weiteren wird unter Umständen eine vorherige Tiefpassfilterung benötigt, um Alias-Effekte zu vermeiden.

6 Messungen und Ergebnisse

Dieses Kapitel stellt die mit dem Objekterkennungssystem durchgeführten Messungen und deren Ergebnisse vor.

6.1 Testumgebung

Für alle Messungen in diesem Kapitel wurde ein PC mit 64 bit Linux-System (Ubuntu 8.04), einer Intel Core i7 940 CPU und 12 GB Arbeitsspeicher verwendet. In dem System sind zwei NVIDIA-Grafikkarten eingebaut. Die erste, eine GeForce GTX 285, diente als dediziertes Device für alle ausgeführten CUDA-Programme. Die zweite wurde lediglich zur Bildschirmausgabe verwendet. Auf diese Weise stand der vollständige globale Speicher der ersten Karte zur Verfügung und die gemessenen Zeiten wurden nicht durch gleichzeitige Berechnungen für die Grafikausgabe verfälscht.

Das Objekterkennungssystem wurde stets im Release-Modus mit der Option `-O2` compiliert (dies ist die Standard-Einstellung von Qt-Programmen). Stichproben mit `-O3` ergaben eine Beschleunigung der CPU-Varianten im einstelligen Prozentbereich.

6.2 Durchführung der Messungen

Für alle vorgestellten Messungen wurden disjunkte Trainings- und Testmengen verwendet. Beim MNIST- und beim NORB-Datensatz sind diese Mengen festgelegt. Für den LabelMe-Datensatz wird die Erstellung der Mengen gesondert beschrieben.

Für jede Messung wurden pro Epoche folgende Werte berechnet:

- Zeit in Millisekunden für die Bearbeitung der Trainingsmenge einer Epoche. Nicht eingeschlossen sind Bearbeitung der Testmenge sowie Visualisierung und Speichern der Gewichte auf Festplatte.
- Gesamtfehler (siehe Abschnitt 2.2.2) auf der Trainingsmenge.
- Anteil falsch klassifizierter Muster der Trainingsmenge, im Folgenden *Trainingsfehler* genannt. Die Angabe erfolgt in Prozent.
- Gesamtfehler auf der Testmenge.
- Anteil falsch klassifizierter Muster der Testmenge (*Testfehler*).

Eine Epoche bedeutet, dass alle Muster der Trainingsmenge einmal jeweils vorwärts- und rückwärtspropagiert wurden. Die resultierenden Log-Dateien haben sind von der Form

```
1   7441  0.067423  16.375 0.040429  10.360
2   7848  0.037454   9.637 0.032561   8.290
3   7821  0.032137   8.355 0.028773   7.350
...
```

und zeigen neben der gerade abgeschlossenen Epoche (erste Spalte) die beschriebenen Werte in der oben genannten Reihenfolge. Üblicherweise ist der Gesamtfehler der Trainings- beziehungsweise Testmenge stark mit dem prozentualen Trainings- beziehungsweise Testfehler korreliert, weswegen nachfolgend immer nur einer der beiden Werte als Plot gezeigt wird.

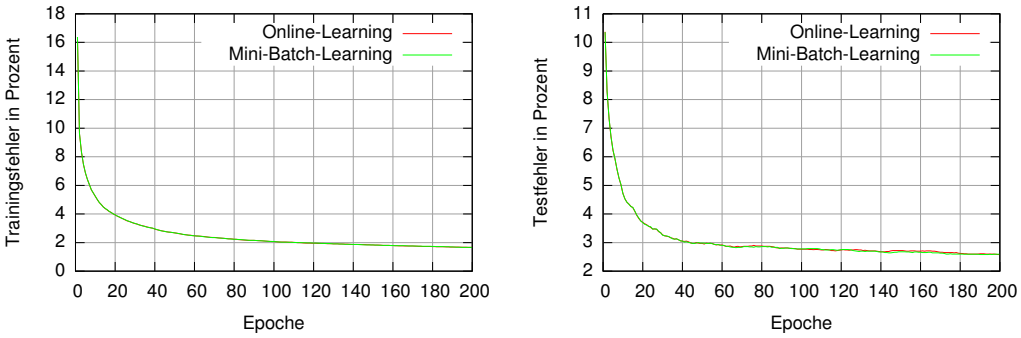


Abbildung 6.1: Vergleich zwischen Online- und Mini-Batch-Learning anhand des MNIST-Datensatzes. Die beiden Kurven liegen beinahe übereinander.

6.3 Vergleich von Online- und Mini-Batch-Learning

Im vorhergehenden Kapitel wurde beschrieben, dass die GPU-Variante zur besseren Ausnutzung des schnellen On-Chip-Speichers immer 16 Muster parallel verarbeitet und die kumulierten Gewichtsänderungen erst danach speichert. Die CPU-Variante hingegen wurde so implementiert, dass sowohl Mini-Batch- als auch Online-Learning möglich ist. Auf diese Weise konnte anhand zweier Messung mit der CPU-Variante überprüft werden, ob sich die Ergebnisse signifikant voneinander unterscheiden.

Als Trainings- und Testmenge für diese Messung wurde der MNIST-Datensatz verwendet (siehe Abschnitt 2.3.1). Pro Epoche wurden alle 60.000 Trainingsmuster nacheinander in das Netz eingegeben und jeweils vorwärts- und rückwärtspropagiert. Die Reihenfolge der Trainingsmuster wurde nicht verändert. Die Eingabe der 28×28 Pixel großen Muster erfolgte zentriert in der 32×32 Pixel großen Karte in Schicht 0, die Eingabewerte wurden auf das Intervall $[-0{,}2; 1{,}0]$ skaliert, wobei $-0{,}2$ dem Hintergrund entspricht. Durch diese Skalierung beträgt die durchschnittliche Eingabe etwa Null. Die Ausgabeschicht enthielt ein Neuron pro möglicher Ziffer, also 10 insgesamt. Als Teacher-Werte wurden 1,0 (aktuelles Muster zeigt die Ziffer) und $-1{,}0$ (Muster zeigt eine andere Ziffer) gewählt. Alle Karten zweier aufeinanderfolgender Schichten waren jeweils miteinander verbunden. Bei der Online-Variante erfolgte die Gewichtsanpassung nach jedem Muster, bei der Mini-Batch-Variante nach jedem 16-ten Muster (wie bei der GPU-Implementierung). Die Struktur des neuronalen Netzes ist in Tabelle 6.1 angegeben.

Schicht	Anzahl Karten	Kartengröße
0	1	32×32
1	2	16×16
Ausgabe	1	10

Tabelle 6.1: Dimensionen des neuronalen Netzes zum Vergleich zwischen Online- und Mini-Batch-Learning

Die resultierenden Lernkurven sind in Abbildung 6.1 dargestellt. Beide Kurven sind beinahe identisch, die Abweichungen sind kaum erkennbar. Auch die Gesamtfehler auf Trainings- und Testmenge weichen nur minimal voneinander ab (hier nicht dargestellt). Dies legt die Vermutung nahe, dass die Verwendung von 16er-Mini-Batches keinen negativen Einfluss auf den Lernerfolg hat. Der Unterschied zwischen Online-Learning und Mini-Batches wurde deshalb nicht weiter untersucht.

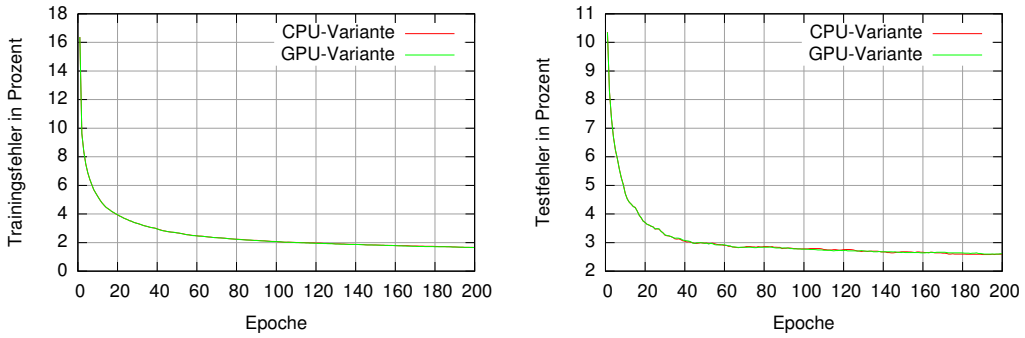


Abbildung 6.2: Vergleich zwischen CPU- und GPU-Variante anhand des MNIST-Datensatzes. Auch hier liegen beide Kurven beinahe übereinander.

6.4 Vergleich von CPU- und GPU-Version

Als nächstes wurde untersucht, inwiefern sich die Ergebnisse der CPU- und der GPU-Variante voneinander unterscheiden. Dieser Schritt ist wichtig, um potenzielle Fehler in der Implementierung aufzudecken und die Vergleichbarkeit beider Varianten – insbesondere für die nachfolgende Geschwindigkeitsmessung – zu gewährleisten.

6.4.1 Numerische Unterschiede

Die Gleitkommaarithmetik ist bei CUDA-kompatiblen GPUs im Gegensatz zu aktuellen Intel-CPUs nicht vollständig kompatibel zur IEEE-Norm 754 [27]. Die Abweichungen sind in [47], Abschnitt A.2, aufgelistet. Selbst bei exakt gleicher Berechnung auf CPU und GPU können sich die Ergebnisse deshalb voneinander unterscheiden. Außerdem sind die beiden Implementierungen des neuronalen Netzes wegen der zugrundeliegenden Hardware sehr unterschiedlich, so dass durch verlustbehaftete Rechnungen mit einfacher Genauigkeit ohnehin abweichende Ergebnisse entstehen können.

Bei neuronalen Netzen können diese Unterschiede kritisch sein. Die Fehlerwerte beispielsweise werden durch große gewichtete Summen berechnet und durch mehrere Schichten propagiert, so dass sich kleine Unterschiede in der Berechnung im Endergebnis unter ungünstigen Umständen deutlich auswirken können. Kritisch ist auch die Anpassung der Gewichte, bei der Gleitkommazahlen von sehr unterschiedlicher Größenordnung miteinander addiert werden.

Aus diesen Gründen ist es notwendig, die Ergebnisse von CPU- und GPU-Variante anhand von Messungen miteinander zu vergleichen. Zu diesem Zweck wurde die Messung aus Abschnitt 6.3 mit der GPU-Variante wiederholt. Die Ergebnisse wurden dann mit der Mini-Batch-CPU-Variante verglichen. Die resultierenden Lernkurven sind in Abbildung 6.2 dargestellt. Auch hier sind beinahe keine Unterschiede zu erkennen, der Lernerfolg ist also fast identisch.

Um die Unterschiede dennoch genauer zu untersuchen, wurden die Differenzen zwischen den gelernten Gewichten von CPU- und GPU-Variante nach einer und nach 200 Epochen berechnet. Die Ergebnisse sind in Tabelle 6.2 dargestellt. Es gibt demnach nennenswerte Unterschiede zwischen den Gewichten beider Varianten. Das arithmetische Mittel der Absolutbeträge der Differenzen beträgt nach einer Epoche etwa 1%, nach 200 Epochen etwa 3% des arithmetischen Mittels der Absolutbeträge der Gewichte. Die betragsmäßig größten Differenzen (Minimum und Maximum) befinden sich sogar in der gleichen Größenordnung wie die Gewichte selbst.

Gewichte			
Epoche	**Arithm. Mittel**	**Minimum**	**Maximum**
1	0,201425	-1,275289	1,495490
200	0,486083	-6,932257	5,909699

Gewichtsdifferenzen			
Epoche	**Arithm. Mittel**	**Minimum**	**Maximum**
1	0,002519	-0,054198	0,057355
200	0,014952	-1,274424	1,217698

Tabelle 6.2: Gelernte Gewichte der CPU-Variante im Vergleich zu den Differenzen zwischen den gelernten Gewichten der CPU- und GPU-Variante. Das arithmetische Mittel wurde jeweils über die Absolutbeträge der Gewichte beziehungsweise Gewichtsdifferenzen gebildet.

Mit Hilfe der Gewichtsvisualisierung der grafischen Benutzeroberfläche wurde deshalb überprüft, ob große Differenzen systematisch an bestimmten Stellen auftreten, um auf diese Weise Fehler in der Implementierung zu lokalisieren. Da die größten Differenzen aber unregelmäßig über alle Karten und Positionen verteilt auftreten, sind die Unterschiede vermutlich durch die abweichende Gleitkommaarithmetik und Implementierung begründet.

Der Einfluss der Gewichtsdifferenzen wurde noch in zwei weiteren Messungen mit größeren Netzen überprüft. Auch dort ergaben sich nur minimale Unterschiede im Lernerfolg nach mehreren hundert Epochen.

6.4.2 Ausführungsgeschwindigkeit

Das Hauptziel bei der parallelen Implementierung mit CUDA ist eine um ein Vielfaches höhere Ausführungsgeschwindigkeit im Vergleich zur „naiven“ CPU-Implementierung. Es wurden deshalb Messungen durchgeführt, die die Geschwindigkeit der CPU- und GPU-Version direkt miteinander vergleichen. Es sei an dieser Stelle noch einmal angemerkt, dass bei der CPU-Implementierung lediglich ein Thread und somit ein Kern der CPU genutzt wird. Außerdem wurden keine Optimierungen wie ASM, MMX oder SSE verwendet. Mit entsprechendem Aufwand ließe sich die Implementierung daher eventuell um ein Mehrfaches beschleunigen. Es ist deshalb wichtig, einen so deutlichen Geschwindigkeitsgewinn zu erreichen, dass sich der höhere Implementierungsaufwand für die CUDA-Variante lohnt.

Vergleich der Ausführungsgeschwindigkeit bei verschiedenen Netzstrukturen

Der Geschwindigkeitsunterschied zwischen CPU- und GPU-Implementierung ist unter Anderem abhängig von der Netzstruktur, also der Anzahl und Größe der Karten und deren Verbindungen zueinander. Insbesondere bei Netzen mit kleinen Karten fallen die Overhead-Zeiten zum Starten der Kernels und zum Kopieren von Speicherbereichen mehr ins Gewicht, wodurch sich kein so hoher Geschwindigkeitsgewinn gegenüber der CPU-Variante ergibt. Es wurden deshalb drei verschieden große Netze getestet.

Um die Netzstruktur prägnant anzugeben, wird an dieser Stelle eine neue Nomenklatur eingeführt: Von Schicht 0 bis zur Ausgabeschicht werden nacheinander die Anzahl der Karten und deren Seitenlänge, getrennt durch ein „×“, angegeben. Da alle Karten quadratisch sind, genügt die Angabe der Seitenlänge. Schichten werden durch „–“ getrennt. Wenn in einer Schicht Eingabekarten vorhanden sind, wird deren Anzahl in Klammern an die Gesamtzahl der Karten angehängt. Das Netz aus Tabelle 6.1 hat demnach die Bezeichnung 1(1)×32–2×16–10 (bei der Ausgabeschicht wird die Angabe der Kartenanzahl nicht benötigt). Falls nicht anders angegeben, sind alle Karten zweier aufeinanderfolgender Schichten miteinander

Netzstruktur	Trainings-zeit CPU	Trainings-zeit GPU	Faktor
1(1)×32–2×16–10	7.368 ms	952 ms	7,74
1(1)×256–2×128–2×64–2×32–2×16–10	575.347 ms	7.834 ms	73,44
1(1)×256–2×128–4×64–8×32–16×16–10	1.348.293 ms	30.967 ms	43,54
4(4)×256–9(7)×128–9(7)×64–9(7)×32–2×16–10	2.255.758 ms	27.445 ms	82,19

Tabelle 6.3: Ergebnisse der Geschwindigkeitsmessung. Dargestellt ist die Laufzeit für eine MNIST-Epoche (Zeit gemittelt über 10 Epochen). Die Netzstruktur aus der letzten Zeile wurde (mit veränderter Anzahl von Ausgabeneuronen) für die meisten der nachfolgenden Messungen mit dem LabelMe-Datensatz verwendet.

verbunden. Eine Ausnahme bilden Eingabekarten, weil sie grundsätzlich nicht zu Karten einer tieferen Ebene (also mit kleinerem Index) verbunden sind.

Tabelle 6.3 zeigt die Geschwindigkeitsmessungen für verschiedene Netzstrukturen. Die angegebene Zeit bezieht sich auf die Durchführung einer kompletten Trainingsepoche des MNIST-Datensatzes mit 60.000 Mustern. Die Eingabe erfolgte bei allen Messungen lediglich in der ersten Eingabekarte von Schicht 0, weil nur die Ausführungszeit, nicht aber die Erkennungsleistung gemessen wurde. Beim Vergleich von Messung 1 und 2 ist zu sehen, dass der Geschwindigkeitsgewinn bei dem großen Netz deutlich höher ist als bei dem kleinen. Bei Messung 3 ist der Faktor wiederum geringer. Das liegt daran, dass der Rückwärtspropagierungs-Kernel kleinere Karten nicht so effizient bearbeitet wie größere (siehe dazu Abschnitt 5.5.2).

Anzahl Gewichtsaktualisierungen pro Sekunde

Aus der Anzahl von Verbindungsgewichten und der gemessenen Zeit lässt sich berechnen, wie viele Gewichtsaktualisierungen das Netz pro Sekunde durchführt. Dieser Wert wird hier als *WUPS-Wert* (*Weight Updates Per Second*) bezeichnet. Analog dazu steht *GWUPS* für *Giga Weight Updates Per Second*, also $1\,\mathrm{GWUPS} = 10^9\,\mathrm{WUPS}$. Beispielsweise verfügt die Netzstruktur in Zeile vier von Tabelle 6.3 über insgesamt 3.650.560 Gewichte. Mit der Musteranzahl von 60.000 und der Zeit von 27,445 Sekunden ergeben sich daraus 7,98 GWUPS für die GPU-Variante.

Diese Zahl lässt erkennen, dass die Ausführungszeit durch die Speicherbandbreite beschränkt ist (und vermutlich nicht durch die mögliche Anzahl von Rechenoperationen pro Sekunde). Da jedes Gewicht vier Byte groß ist und mindestens einmal für Vorwärts- und Rückwärtspropagierung geladen werden muss, ergibt sich eine genutzte Speicherbandbreite von etwa 64 GB pro Sekunde allein für die Gewichte. Zusammen mit Aktivierungen, Fehlerwerten, Kopieren der Eingabemuster und dem vorhandenen Overhead kommt man somit recht nah an die maximale Speicherbandbreite von etwa 130 GB/s.

Aufschlüsselung der Trainingszeit der GPU-Variante

Neben der Gesamtlaufzeit für eine Trainingsepoche wurde ermittelt, welche Zeit einzelne Funktionen innerhalb einer solchen Epoche in Anspruch nehmen. Dazu wurde die Trainingsfunktion für den LabelMe-Datensatz verwendet. An dieser Stelle wird nur die Laufzeit dieser Funktion ausgewertet. Die Erkennungsleistung wird später in Abschnitt 6.7 behandelt.

Die Trainingsfunktion kopiert pro Schleifendurchlauf jeweils 16 Trainingsmuster (Bilder) der Größe 256×256 Pixel und die zugehörigen Teacher-Werte vom Host auf das Device. Dann werden die Eingabekarten aus den Bildern berechnet, nämlich die in Abschnitt 4.3 beschriebenen vier Kanten- und drei PCA-Kanäle. Anschließend werden Vorwärts- und Rückwärtspropagierung durchgeführt, dann wird die Ausgabe zurück zum Host kopiert, um dort den Trainingsfehler zu berechnen.

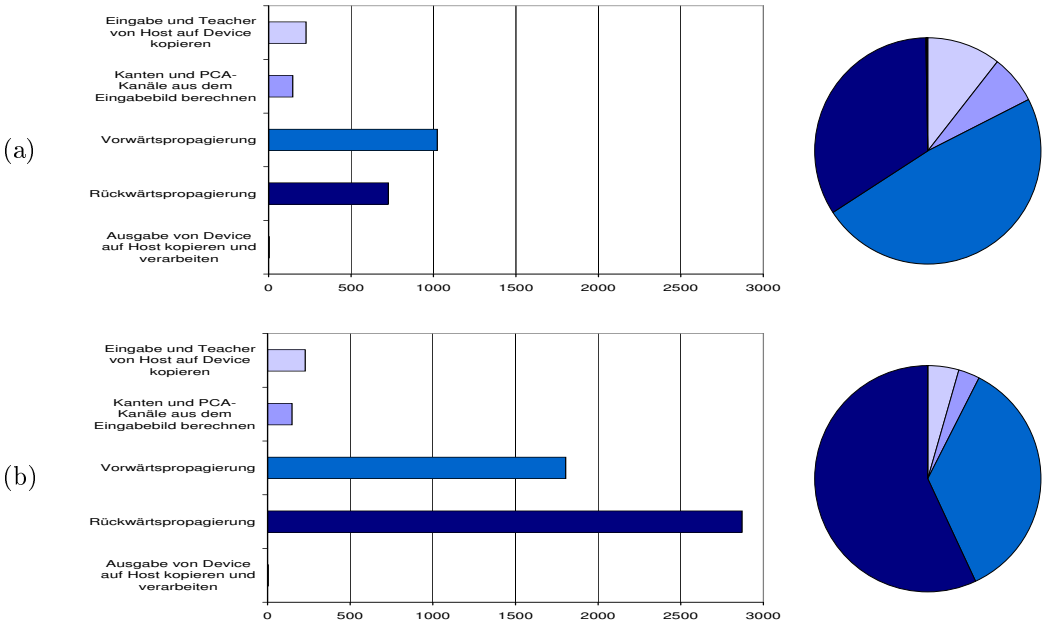


Abbildung 6.3: Laufzeiten der einzelnen Schritte der LabelMe-Trainingsfunktion. Das Training wurde mit 4000 Mustern durchgeführt. Die Zahlen auf der x-Achse geben die Zeit in Millisekunden an, die der entsprechende Schritt pro Epoche benötigt. (a) wurde mit der Netzstruktur Nummer 4 aus Tabelle 6.3 gemessen, (b) mit einem 4(4)×256–9(7)×128–11(7)×64–15(7)×32–16×16–10-Netz, also mit gleicher Anzahl Eingabekarten und sich jeweils verdoppelnder Anzahl anderer Karten in den Schichten 1–4.

Die Laufzeiten dieser Schritte sind in Abbildung 6.3 dargestellt. Auffällig ist, dass die Vorwärtspropagierung in (a) länger als die Rückwärtspropagierung dauert. Ein Grund dafür ist die für die Rückwärtspropagierung optimierte Speicherstruktur der Gewichte im Devicespeicher, die bei der Vorwärtspropagierung erst in den Shared Memory geladen werden müssen (welcher in der Praxis trotz vermiedener Bankkonflikte langsamer als die ausschließliche Verwendung von Registern ist). Außerdem dauert der Zugriff auf die Aktivierungen der Quellkarte relativ lange. Dieses Problem konnte nicht vollständig geklärt werden, denn eigentlich sollte die Lokalität der Lesezugriffe durch die Nutzung des Texturcaches einen schnellen Speichertransfer ermöglichen, auch wenn die Zugriffe nicht zusammenhängend sind.

In (b) wurde die Laufzeit eines größeren Netzes gemessen. Durch die größere Anzahl von Karten auf den höheren Ebenen wird die Vorwärtspropagierung hier im Verhältnis schneller, weil der Kernel wie in Kapitel 5 beschrieben mehrere Zielkarten parallel bearbeiten kann. Dabei wird der Texturcache effizienter genutzt, weil jeweils alle Threads innerhalb eines Blocks, die die gleiche x- und y-Position auf einer der Zielkarten repräsentieren, auf die gleichen Quellaktivierungen zugreifen. Das oben genannte Problem fällt somit weniger ins Gewicht.

6.5 Messungen mit dem MNIST-Datensatz

In den bisher vorgestellten Messungen dieses Kapitels lag der Fokus auf dem Vergleich zwischen CPU- und GPU-Variante und der erreichten Ausführungsgeschwindigkeit. In den folgenden Abschnitten werden die Erkennungsleistungen anhand von drei verschiedenen Datensätzen evaluiert.

Parameter	Wert	Beschreibung
η	0,01	Lernrate aller regulären Schichten
$\eta_{Ausgabe}$	0,0001	Lernrate der Ausgabeschicht
λ	2,0	Faktor für anwachsende Lernrate in tieferen Schichten (siehe Abschnitt 4.5).
NET_MAX	10,0	Maximale Netzeingabe nach Gewichtsinitialisierung (siehe Abschnitt 4.5.2)
Biasinitialisierung	$[-0{,}01; 0{,}01]$	Intervall der zufälligen Biasinitailisierung
Wertebereich der Eingabe	$[-0{,}2; 1{,}0]$	Intervall der Eingabe. $-0{,}2$ entspricht dem Hintergrund.
Teacherwerte	$\{-1, 1\}$	Teacherwerte der Ausgabeneuronen. 1, falls das aktuelle Muster die repräsentierte Ziffer darstellt, -1 sonst.

Tabelle 6.4: Parameter bei der Messung verschiedener Netzstrukturen mit dem MNIST-Datensatz

Dieser Abschnitt stellt Messungen mit dem MNIST-Datensatz vor und vergleicht die Ergebnisse mit denen anderer Ansätze, um einen Anhaltspunkt für die Qualität der Klassifizierung zu erhalten.

6.5.1 Vergleich verschiedener Netzstrukturen

Neben weiteren Parametern wie Lernrate, Gewichtsinitialisierung und Ein- und Ausgabecodierung hat die Netzstruktur einen großen Einfluss auf den Lernerfolg. Da es unmöglich ist, alle Parameterkombinationen zu testen, wurden zunächst Messungen durchgeführt, bei denen nur die Netzstruktur verändert wurde. Alle anderen Parameter wurden währenddessen nicht verändert und so gewählt, wie es sich nach Meinung des Autors in zahlreichen Experimenten zuvor als sinnvoll herausgestellt hatte. Diese Parameter sind in Tabelle 6.4 angegeben.

Die Eingabe erfolgte immer in Schicht 0. Die Trainingsmuster wurden dabei auf die Größe der Eingabekarte hochskaliert. Die Reihenfolge der Trainingsmuster war immer gleich, wurde also nicht randomisiert. Für die Messung wurden die ersten drei Netzstrukturen aus Tabelle 6.3 verwendet, weil es sich dabei um ein sehr kleines, ein mittelgroßes und ein sehr großes Netz handelt. Die resultierenden Lernkurven sind in Abbildung 6.4 dargestellt, Tabelle 6.5 zeigt die Ergebnisse nach 200 Epochen.

Erwartungsgemäß fällt der Trainingsfehler bei den größeren Netzen durchgängig geringer aus. Aber auch der Testfehler ist beim größten Netz am geringsten, wenn auch beinahe gleich dem des mittleren Netzes. Dies ist nicht offensichtlich, weil größere Netze eher als kleine zum Auswendiglernen der Trainingsmenge neigen und somit das Ergebnis auf der Testmenge durch mangelnde Generalisierung verschlechtert werden kann.

Netz	Trainingsfehler nach 200 Epochen	Testfehler nach 200 Epochen	Bester Testfehler
Klein	1,66 %	2,59 %	2,58 %
Mittel	0,18 %	1,88 %	1,82 %
Groß	0,16 %	1,80 %	1,74 %

Tabelle 6.5: Trainings- und Testfehler auf dem MNIST-Datensatz bei unterschiedlichen Netzstrukturen. Die Netze „klein“, „mittel“ und „groß“ entsprechen den Strukturen aus Zeile 1–3 in Tabelle 6.3. Der beste Testfehler ist der geringste Testfehler, der innerhalb der ersten 200 Epochen gemessen wurde.

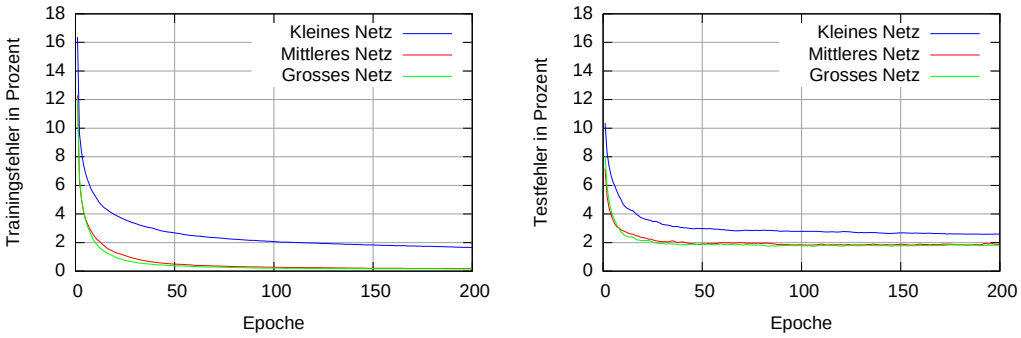


Abbildung 6.4: Trainings- und Testfehler auf dem MNIST-Datensatz bei unterschiedlichen Netzstrukturen. Zur Beschreibung siehe die Unterschrift von Tabelle 6.5.

6.5.2 Einfluss von anderen Parametern

Anhand des mittelgroßen Netzes aus dem letzten Abschnitt wurde die Messung mit veränderten Parametern wiederholt. Dabei ergaben sich folgende Erkenntnisse:

Lernrate Neben der in Tabelle 6.4 angegebenen Lernrate von $\eta = 0{,}01$ wurden die Lernraten 0,1 und 0,001 getestet. Entsprechend wurde dabei auch $\eta_{Ausgabe}$ auf 0,001 beziehungsweise 0,00001 geändert. Erwartungsgemäß dauerte das Training bis zum Erreichen vergleichbarer Ergebnisse kürzer beziehungsweise länger als mit der urspünglichen Lernrate. Die Erkennungsleistung wurde mit $\eta = 0{,}1$ drastisch schlechter, mit $\eta = 0{,}001$ war sie nach einer deutlich höheren Anzahl Epochen fast identisch. Die urspüngliche Lernrate wurde daher in allen folgenden Messungen beibehalten.

NET_MAX Statt mit NET_MAX = 10,0 wurde je eine Messung mit den Werten 1,0 und 100,0 durchgeführt. Bei dem kleineren Wert entstand während der ersten 20 Epochen ein Plateau, bei dem sich die Erkennungsrate auf Trainings- und Testmenge nicht veränderte. Dies ist nachvollziehbar, weil bei betragsmäßig sehr kleiner Gewichtsinitialisierung zunächst alle Muster ähnliche Ausgaben erzeugen und außerdem die Fehlerwerte sehr klein sind. Dadurch ist auch die Änderung der Gewichte klein. Im weiteren Verlauf der Messung ergab sich eine deutlich schlechtere Generalisierungsleistung als bei der größeren Gewichtsinitialisierung. Eine Erklärung hierfür ist dem Autor nicht bekannt.

Bei der Messung mit NET_MAX = 100,0 war die Erkennungsleistung durchgängig deutlich schlechter. Dies könnte daran liegen, dass es in diesem Fall schwieriger ist, gute Minima der Fehleroberfläche zu finden, weil die hohe Zufallsinitialisierung eventuell weit von diesen entfernt ist.

Eingabeskalierung Statt die Eingabemuster auf die Größe der Eingabekarte zu skalieren, wurden die Eingaben zentriert und der restliche Bereich auf $-0{,}2$ gesetzt. Die Erkennungsleistung wurde dabei schlechter. Mit dem mittelgroßen Netz wurde dadurch in etwa die Erkennungsleistung des kleinen Netzes erzielt.

Die in Tabelle 6.4 angegebenen Parameter wurden aufgrund der gerade vorgestellten Ergebnisse in allen nachfolgenden Messungen beibehalten. Sowohl beim NORB- als auch beim LabelMe-Datensatz wurde stichprobenartig untersucht, ob sich eine Veränderung der Parameter positiv auswirkt. Es ergaben sich jedoch keine signifikanten Verbesserungen.

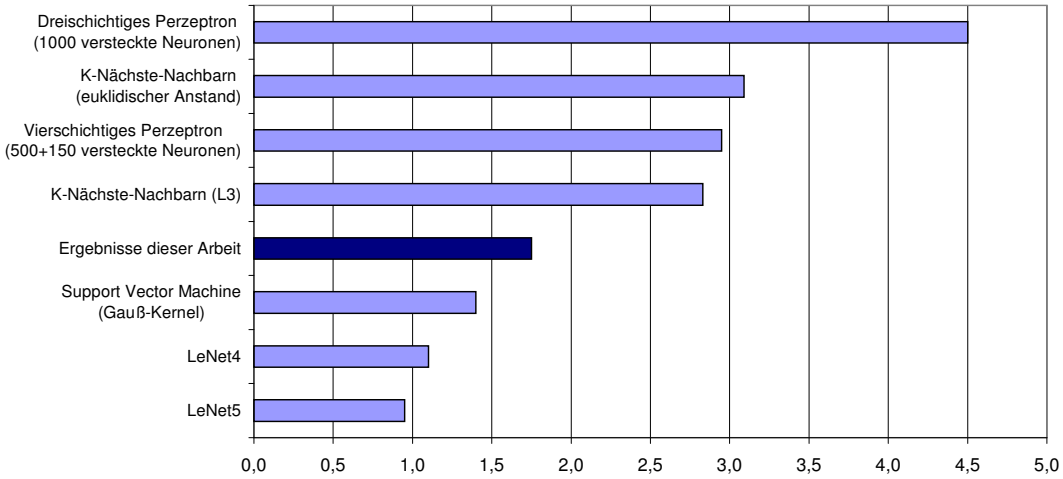


Abbildung 6.5: Vergleich des prozentualen Testfehlers auf dem MNIST-Datensatz bei verschiedenen Klassifizierungsmethoden. Quelle der Ergebnisse: [37].

6.5.3 Vergleich mit anderen Ansätzen

In Abbildung 6.5 ist der beste erreichte Testfehler auf dem MNIST-Datensatz im Vergleich zu den Testfehlern anderer Klassifizierungsmethoden dargestellt. Auf der MNIST-Webseite [37] sind noch viele weitere Ergebnisse zu finden. An dieser Stelle wurden nur Ansätze ausgewählt, die keinerlei Vorverarbeitung der Eingabe oder unüberwachtes Vortraining nutzen. Ansätze mit Vorverarbeitung (zum Beispiel elastische Verzerrungen [61]) oder Vortraining (zum Beispiel in [53]) liefern meist bessere Erkennungsleistungen (siehe Abschnitt 3.1). Diese Techniken wurden allerdings im Rahmen der vorliegenden Arbeit nicht angewandt.

6.6 Messungen mit dem NORB-Datensatz

Neben MNIST existiert mit NORB (vorgestellt in Abschnitt 2.3.2) ein weiterer Datensatz, für den gut vergleichbare Ergebnisse verschiedener Klassifizierungsmethoden verfügbar sind. Für die folgenden Messungen wurde der Small-NORB-Datensatz [24] verwendet, der in [38] als „norm-unif“ bezeichnet wird. Es wurden drei Messungen mit unterschiedlichen Eingabekarten durchgeführt:

1. Zwei Eingabekarten, die zentriert jeweils eines der stereoskopischen Bilder enthielten
2. Wie 1, aber mit zufälligen Verschiebungen von ± 16 Pixeln in x- und y-Richtung während der Trainingsphase
3. Wie 2, aber zusätzlich mit je zwei Kanteneingaben pro Blickwinkel. Diese Variante ist in Abbildung 6.7 dargestellt.

Als Netzstruktur wurde $N(N)\times 128\text{–}2\times 64\text{–}2\times 32\text{–}2\times 16\text{–}5$ gewählt, mit $N = 2$ bei Messung eins und zwei und $N = 6$ bei Messung drei. Tabelle 6.6 zeigt die Ergebnisse der Messungen, in Abbildung 6.6 werden diese mit den Ergebnissen aus [38] verglichen.

Es ist deutlich zu erkennen, dass sowohl die Vergrößerung der Trainingsmenge durch zufällige Verschiebungen als auch die zusätzlichen Kanteneingaben die Erkennungsraten auf der Testmenge verbessern. Da die Trainingsmenge bei Messung eins bereits nach wenigen Epochen komplett auswendig gelernt wurde, ist diese Verbesserung nicht auf die insgesamt erhöhte Größe des Netzes zurückzuführen.

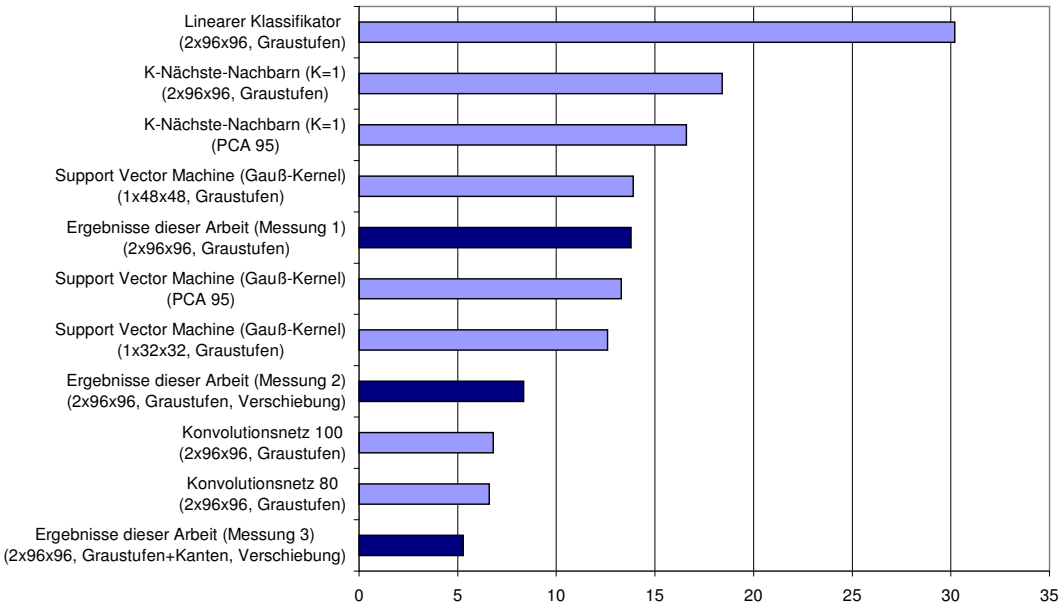


Abbildung 6.6: Vergleich des prozentualen Testfehlers auf dem NORB-Datensatz bei verschiedenen Klassifizierungsmethoden. Quelle der Ergebnisse: [38].

Messung	Trainings-fehler	Test-fehler	Bester Testfehler	Anzahl Epochen
1	0,00 %	13,79 %	13,78 %	1000
2	0,83 %	8,35 %	6,72 %	2200
3	0,07 %	6,41 %	5,28 %	5800

Tabelle 6.6: Ergebnisse der drei beschriebenen Messungen mit dem NORB-Datensatz. Wegen Fluktuationen bei Trainings- und Testfehler wurden diese beide Werte jeweils über die letzten 10 Epochen gemittelt. Der beste Testfehler ist der geringste, der innerhalb der gesamten Messung ermittelt wurde. Die Messungen wurden jeweils beendet, wenn keine nennenswerte Änderung des Testfehlers über mehrere hundert Epochen mehr stattfand (daher die unterschiedliche Anzahl von Epochen).

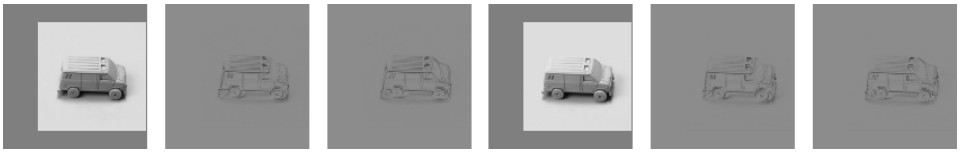


Abbildung 6.7: Eingabekarten der dritten NORB-Messung mit einem zufällig verschobenen Trainingsmuster. Der Wertebereich der Eingabe beträgt -1 (weiß) bis 1 (schwarz).

(a)  (b)  (c) 

Abbildung 6.8: Extraktion von Mustern für die Trainings- und Testmenge. (a) zeigt die quadratische Bounding Box, die mit den exakten Grenzen des Annotationspolygons erzeugt wurde. Bei (b) wurden an jeder Seite 30 % Kontext angefügt. Weil ein Teil der Bounding Box außerhalb des Bildes liegt, wird er grau gefärbt. Bei (c) wurde die Bounding Box so verschoben, dass der Schwerpunkt des Annotiationspolygons genau zentriert ist. Dadurch verschiebt sich der Bildausschnitt leicht nach links unten. Bildquelle: [58]

6.7 Messungen mit dem LabelMe-Datensatz

Im Gegensatz zu den beiden vorhergehenden Datensätzen gibt es bei LabelMe keine festgelegte Trainings- oder Testmenge, sondern nur Bilder mit darin annotierten Objekten. Abschnitt 6.7.1 erläutert deshalb zunächst, wie die Trainings- und Testmenge erstellt wurde. Die nachfolgenden Abschnitte beschreiben dann die damit durchgeführten Messungen. Abschnitt 6.7.5 geht abschließend auf die Ergebnisse der Sliding-Window-Technik ein.

6.7.1 Erstellung von Trainings- und Testmenge

Die erstellte Trainings- und Testmenge besteht aus insgesamt 86.574 Mustern, wobei jedes Muster mindestens ein annotiertes Objekt enthält und als JPG-Grafik gespeichert ist. Es gibt keine feste Trennung zwischen Trainings- und Testmustern. Stattdessen kann bei jeder Messung entschieden werden, welche Muster für die Trainings- und welche für die Testphase verwendet werden. Aus diesem Grund wird die gesamte Menge der Muster im Folgenden einfach *Mustermenge* genannt.

Festlegung von Format und Inhalt der Muster

Die Größe jedes Musters beträgt 256×256 Pixel. Dementsprechend sind auch Karten in Schicht 0 des neuronalen Netzes genauso groß. Dieser Wert wurde gewählt, weil er eine gute Abwägung zwischen der Ausführungsgeschwindigkeit des Netzes und dem Detailreichtum des Inhalts darstellt.

Jedes Muster enthält ein Objekt aus dem Datensatz, das nach dem Schwerpunkt seines Annotationspolygons zentriert und auf eine Größe von 160 Pixeln in seiner größeren Dimension – entweder Höhe oder Breite – skaliert ist (siehe dazu Abbildung 6.8). Da mehr als 79 % aller Objekte des LabelMe-Datensatzes in mindestens einer Dimension so groß sind, wurden keine kleineren Objekte für die Erstellung der Mustermenge verwendet, weil diese teilweise stark vergrößert werden müssten.

Durch die Schwerpunktberechnung wird vermieden, dass flächenmäßig kleine Teile des Annotationspolygons (wie zum Beispiel ein ausgestreckter Arm einer Person) dazu führen, dass der Hauptteil des dargestellten Objekts (in diesem Fall der restliche Körper) zu weit von der Mitte des Musters entfernt ist. Die Zentrierung nach Schwerpunkt kommt dem näher, was Menschen vermutlich als Bounding Box wählen würden.

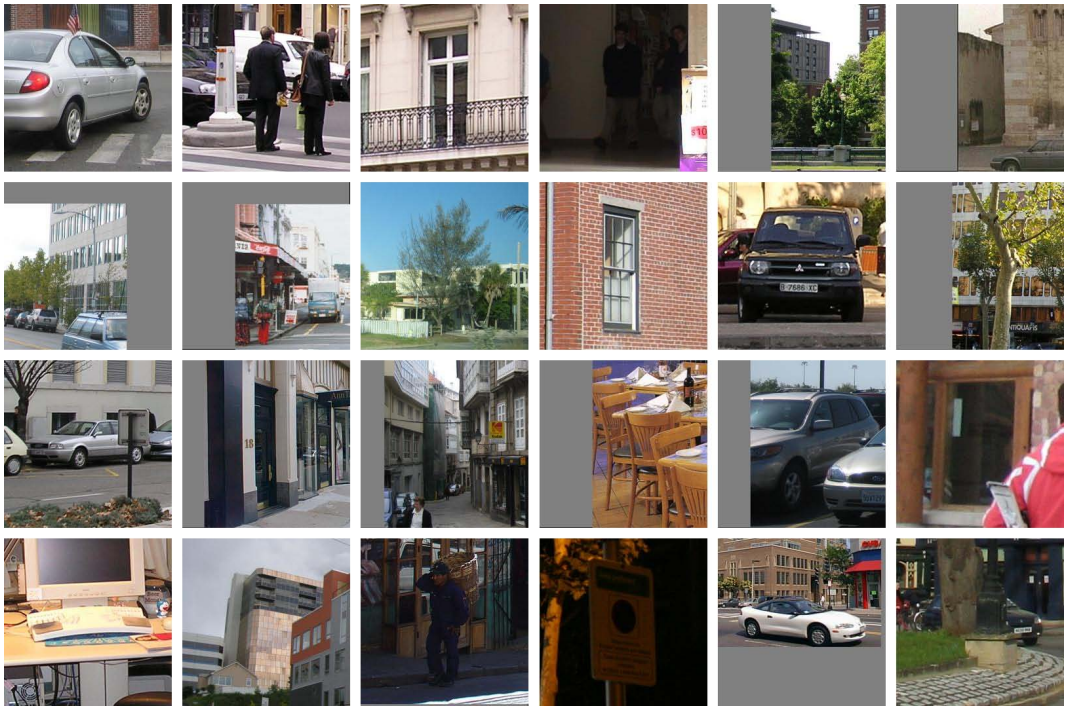


Abbildung 6.9: Zufällig ausgewählte Muster der erzeugten Mustermenge

#	**Klasse**	**Anzahl**	**Anteil**
1	person	9.129	22,04 %
2	car	6.863	16,57 %
3	building	6.793	16,40 %
4	window	6.759	16,32 %
5	tree	5.227	12,62 %
6	sign	1.639	3,96 %
7	door	1.552	3,75 %
8	bookshelf	1.049	2,53 %
9	chair	781	1,89 %
10	table	670	1,62 %
11	keyboard	625	1,51 %
12	head	332	0,80 %
	Summe	41.419	100,00 %
(13)	clutter	45.155	

Tabelle 6.7: Klassen der erstellten LabelMe-Mustermenge. Die letzte Spalte gibt den prozentualen Anteil an der Gesamtzahl der Objektklassen (exklusive der „clutter"-Klasse) an.

Die Teile der Bounding Box, die außerhalb des Bildes liegen, werden grau gefärbt (RGB 0x808080). Diese Farbe entspricht für die verwendete Eingabecodierung mit einem Wertebereich von $[-1, 1]$ bei den drei PCA-Kanälen ungefähr dem Wert 0.

Auswahl der Objektklassen

Von den 20 häufigsten Objektklassen des LabelMe-Datensatzes[1] wurden 12 für die Mustermenge ausgewählt. Klassen wie „sky“, „road“ und „sidewalk“ wurden nicht verwendet, da sie keine klar abgrenzbaren Objekte darstellen. Statt „person walking“ wurde die Oberklasse „person“ gewählt. Tabelle 6.7 zeigt alle Klassen und die Anzahl ihrer Instanzen in der Mustermenge.

Zusätzlich wurden zufällige, quadratische Ausschnitte aus allen Bildern des Datensatzes ausgewählt, die sich mit keinem Annotationspolygon einer Instanz der 12 Objektklassen überschneiden. Die resultierende Klasse heißt „clutter“ (auf deutsch in etwa „Durcheinander“). Die Größe der Ausschnitte (vor der Skalierung auf 256×256 Pixel) wurde für jedes Bild zufällig zwischen 64×64 und 1024×1024 Pixel gewählt. Da sich große Ausschnitte mit höherer Wahrscheinlichkeit mit Instanzen einer der 12 Objektklassen überschneiden, gibt es mehr kleinere Ausschnitte in der „clutter“-Klasse.

Erstellung der Mustermenge mit MATLAB

Die Erstellung der Mustermenge erfolgte mit MATLAB. Zum Durchsuchen des Datensatzes nach den Instanzen der gewünschten Objektklassen wurde die `LMdbquery`-Funktion der in Abschnitt 2.3.3 beschriebenen LabelMe-Toolbox für MATLAB verwendet.

Anschließend wurden jeweils 1000 Muster als JPG-Grafiken in einem Verzeichnis gespeichert, weil beim verwendeten Dateisystem FAT der Lese- und Schreibzugriff deutlich langsamer wird, wenn ein Verzeichnis mehrere tausend Dateien enthält. Vor dem Speichern wurde die Reihenfolge der Bilder, die Instanzen der gesuchten Objektklassen enthalten, zufällig permutiert. Abbildung 6.9 zeigt einige zufällig ausgewählte Muster der Menge.

Die Teacherwerte der Mustermenge wurden in der Binärdatei `annotation.dat` gespeichert. Der Aufbau dieser Datei ist in Tabelle 6.8 dargestellt. Als Teacherwert für das extrahierte Objekt (welches zentriert im Muster zu sehen ist) wird 1,0 verwendet. Für alle weiteren Objektklassen wird überprüft, ob es eine Instanz gibt, die sich teilweise oder ganz mit dem Muster überschneidet[2]. In diesem Fall wird als Teacherwert 0,0 verwendet, ansonsten −1,0.

Offset	Datentyp	Wert	Beschreibung
0000	integer	`0x0014B315`	Magic Number
0004	integer	12	Anzahl Objektklassen
0008	float	$[-1, 1]$	Teacher der ersten Klasse des ersten Musters
...	...	...	...
0056	float	$[-1, 1]$	Teacher der zwölften Klasse des ersten Musters
0060	float	$[-1, 1]$	Teacher der ersten Klasse des zweiten Musters
...	...	...	...

Tabelle 6.8: Aufbau der erzeugten Annotationsdatei. Nach dem Header folgen jeweils 12 `float`-Teacherwerte für jedes Muster.

[1] `http://people.csail.mit.edu/torralba/research/LabelMe/objstats/statsobjects_all.html`

[2] Ein Beispielprogramm für die Überprüfung auf Überlappung befindet sich auf dem beigelegten Datenträger.

6.7.2 Einfluss der Eingabe auf den Lernerfolg

In diesem Abschnitt werden verschiedene Arten der Eingabe anhand von Messungen miteinander verglichen.

Vergleich von Kanten- und PCA-Kanälen

Als erster Schritt wurde untersucht, ob die Eingabe von Kanten und den drei PCA-Kanälen (vergleiche Abschnitt 4.3) bessere Erkennungsleistungen auf der Testmenge bringt als die Eingabe von Kanten oder PCA-Kanälen allein beziehungsweise dem ersten PCA-Kanal allein (ähnlich einem Graustufenbild).

Die im Folgenden verwendete Netzstruktur ist 0×256–2×128–2×64–2×32–2×16–10, *zuzüglich* der jeweiligen Eingabekarten. Als Trainingsmenge wurden 40.000 Muster der im vorhergehenden Abschnitt beschriebenen LabelMe-Mustermenge verwendet. 20.000 davon waren aus der „clutter"-Kategorie, die anderen 20.000 enthielten je mindestens eine der 12 Objektklassen. Als Testmenge wurden 10.000 weitere Muster verwendet, davon ebenfalls 50 % „clutter". Während der Trainingsphase wurden alle Trainingsmuster zufällig um ±12 Pixel in x- und y-Richtung verschoben, um eine bessere Generalisierung zu erreichen. Für jedes vorwärtspropagierte Muster wird zunächst überprüft, welches Ausgabeneuron die höchste Aktivierung besitzt. Wenn diese Aktivierung einen bestimmten Schwellwert überschreitet, wird das Ergebnis als gefundenes Objekt gewertet. Bei einer Aktivierung unter dem Schwellwert wird das Muster als „clutter" gewertet. Der Schwellwert wurde in den nachfolgenden Messungen auf 0,5 festgelegt.

Tabelle 6.9 zeigt die verglichenen Eingaben und gibt die Ergebnisse nach jeweils 500 Epochen an. Die zugehörigen Lernkurven für die Testmenge sind in Abbildung 6.10 (links) dargestellt.

Aufgrund der Tatsache, dass Farbunterschiede vom menschlichen Sehsystem mit geringerer Auflösung als Helligkeitsunterschiede wahrgenommen werden [30], wurde bei Messung vier in Schicht null auf die Eingabe der PCA-Kanäle verzichtet.

Die schlechteste Erkennungsrate auf der Testmenge ergibt sich, wenn nur der erste PCA-Kanal eingegeben wird. Bei allen drei PCA-Kanälen ist das Ergebnis bereits signifikant besser, allerdings noch nicht so gut wie bei der Eingabe der vier Kantenkanäle. Am besten ist die Kombination aus beidem.

Es sei angemerkt, dass die Netze aus Tabelle 6.9 in ansteigender Reihenfolge aufgrund der höheren Anzahl von Eingabekarten auch jeweils über mehr Verbindungsgewichte verfügen. Dies könnte theoretisch auch ein Grund für die unterschiedliche Erkennungsleistung sein. Zur Untersuchung wurde eine weitere Messung durchgeführt, bei der genauso viele Eingabekarten wie bei der vierten Messung verwendet wurden, aber jede Eingabekarte den ersten PCA-Kanal enthielt. Der Testfehler war während der gesamten Messung (über 500 Epochen)

#	**Eingabe in Schicht 0**	**Eingabe in Schicht 1–3**	**Trainingsfehler nach 500 Epochen**	**Testfehler nach 500 Epochen**
1	PCA 1	PCA 1	24,18 %	26,33 %
2	PCA 1–3	PCA 1–3	16,93 %	23,22 %
3	4×Kanten	4×Kanten	16,59 %	20,13 %
4	4×Kanten	4×Kanten u. PCA 1–3	9,52 %	17,65 %

Tabelle 6.9: Einfluss der Eingabecodierung auf den Lernerfolg. Die ersten beiden Spalten geben die jeweils verwendeten Eingabekarten an. 4×Kanten steht für vier Eingabekarten mit jeweils einem der in Abschnitt 4.3.2 beschriebenen Kantenkanäle. Aufgrund der Fluktuationen des Testfehlers sind die angegebenen Ergebnisse jeweils über die letzten fünf Epochen gemittelt.

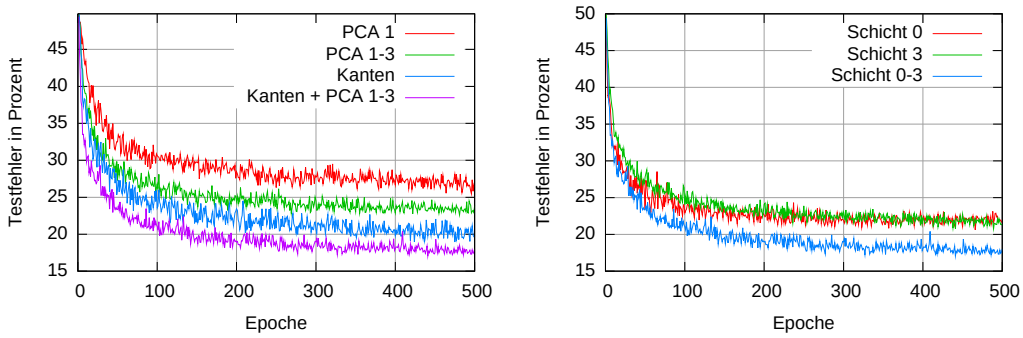


Abbildung 6.10: Links: Testfehler zu Tabelle 6.9. Rechts: Testfehler bei alleiniger Eingabe von Kanten- und PCA-Kanälen in Schicht 0 und Schicht 3 sowie bei Eingabe in Schicht 0–3.

beinahe identisch wie der der ersten Messung. Somit hat die unterschiedliche Anzahl von Verbindungsgewichten vermutlich keinen Einfluss auf die obigen Messergebnisse.

Alle nachfolgenden Messungen wurden mit der in Messung vier verwendeten Eingabe aus Kanten und den drei PCA-Kanälen durchgeführt.

Vergleich von verschiedenen Eingabeschichten

Als zweiter Schritt wurde untersucht, wie sich der Lernerfolg bei Eingabe in den Schichten 0–3 (wie in der obigen Messung) im Gegensatz zu einer Eingabe verhält, die nur in Schicht 0 oder nur in Schicht 3 erfolgt. Der resultierende Testfehler ist in Abbildung 6.10 rechts angegeben.

Die alleinige Eingabe der Kanten- und PCA-Kanäle in Schicht 0 liefert ähnliche Ergebnisse wie die Eingabe in Schicht 3. Beide Ergebnisse sind aber signifikant schlechter als die Eingabe auf den Schichten 0–3. Nachfolgend wurden deshalb alle Messungen mit Eingaben auf Schicht 0–3 durchgeführt.

6.7.3 Gelernte Gewichte

Abbildung 6.11 zeigt Visualisierungen von mehreren Kartenverbindungen. Grundlage für diese Visualisierungen bilden die in Messung vier des vorhergehenden Abschnitts gelernten Verbindungsgewichte. Dargestellt sind einige der Kartenverbindungen zwischen Schicht 2 und 3. Da die Karten in Schicht 3 eine Größe von 32×32 Neuronen haben, sind auf jeder Abbildung 32×32 Blöcke dargestellt. Jeder dieser Blöcke repräsentiert das rezeptive Feld eines Zielneurons. Dieses Feld hat, wie in Abschnitt 4.1.2 beschrieben, eine Größe von 4×4 Neuronen, die als einzelne Pixel in der Visualisierung zu erkennen sind. Zwischen den Blöcken ist zur besseren Unterscheidbarkeit jeweils ein kleiner Freiraum vorhanden. Die eigentlich vorhandene Überlappung der rezeptiven Felder kann in der Visualisierung nicht dargestellt werden.

Die gelernten Verbindungsgewichte weisen je nach Typ der Quellkarte deutliche Unterschiede auf. Diese Unterschiede treten, trotz verschiedener Parameter, reproduzierbar über alle Messungen hinweg auf und sehen in allen Schichten des Netzes ähnlich aus.

Abbildung 6.11 (a) zeigt eine Kartenverbindung zu einer Karte aus Schicht 2, die keine Eingabekarte ist. Die Struktur der Gewichte ist hier sehr fein, es lassen sich keine Muster erkennen. Teil (b) zeigt die Verbindung zu einer Eingabekarte, die einen Kantenkanal enthält. Die Gewichtstruktur ist hier grober. Vermutlich wird damit eine gewisse Translationsinvarianz erreicht, da Zielneuronen auch bei kleinen Verschiebungen des Eingabebildes ähnliche Aktivierungen aufweisen.

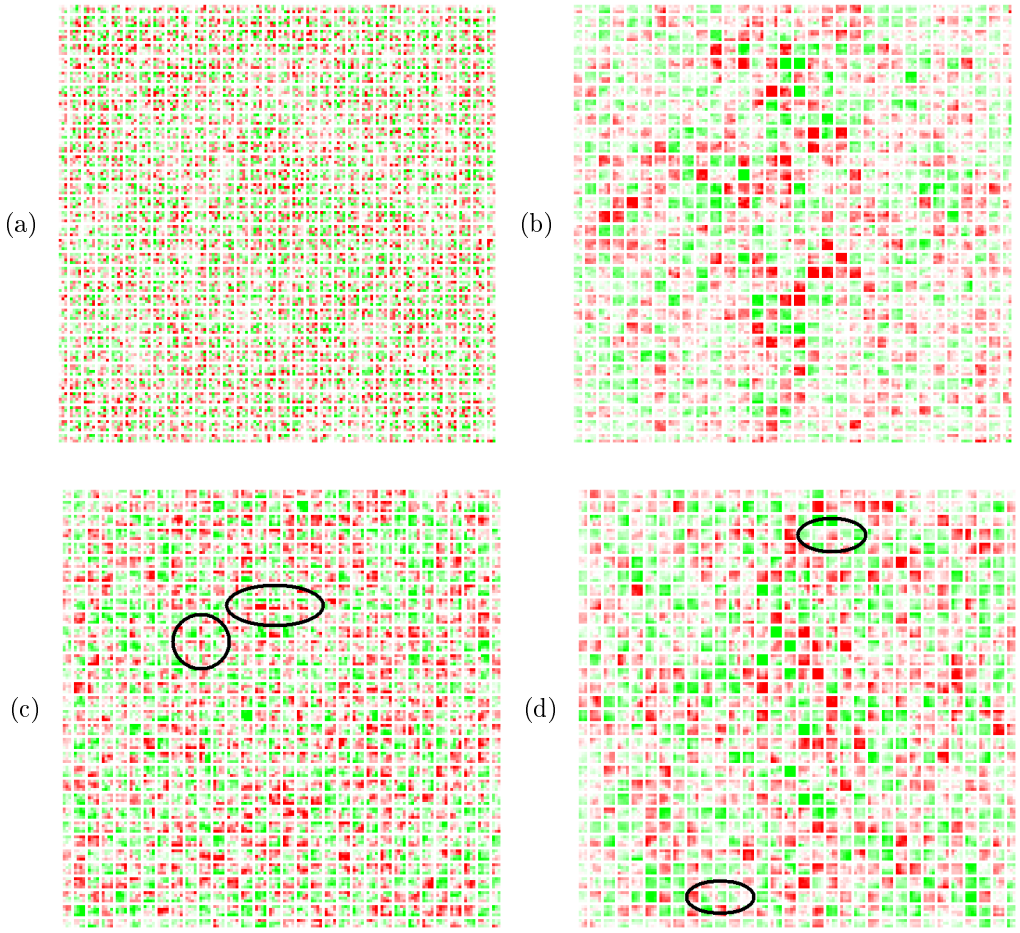


Abbildung 6.11: Visualisierung einiger Kartenverbindungen zwischen den Schichten 2 und 3 von Messung vier aus Abschnitt 6.7.2. Abbildung (a) zeigt Gewichte einer Verbindung zu einer Karte, die keine Eingabekarte ist. Die restlichen Abbildungen zeigen Gewichte von Verbindungen zu je einer Eingabekarte mit (b) Kanten-, (c) PCA1- und (d) PCA2-Kanal. Grün repräsentiert positive, rot negative Gewichte, weiß repräsentiert Gewichte nahe Null.

Besonders interessant sind die Abbildungen (c) und (d), die Verbindungen zu einer Eingabekarte mit PCA1- beziehungsweise PCA2-Kanal darstellen. Bei beiden Verbindungen haben sich klar erkennbare Kantenfilter gebildet. Einige besonders deutliche Beispiele sind schwarz umkreist. Tendenziell weisen die Filter in (c) härtere Übergänge auf als die in (d). Eine mögliche Interpretation dafür ist, dass aus Graustufenbildern feinere Details gewonnen werden können als aus Farbdifferenzbildern.

6.7.4 Weitere Messungen

Es wurden weitere Messungen durchgeführt, um die Auswirkungen verschiedener Parameter zu testen. Diese Messungen werden hier kurz vorgestellt.

Größere Trainingsmenge Der beschränkende Faktor für die Größe der Trainings- und Testmenge war im Rahmen dieser Arbeit die Größe des Host-Arbeitsspeichers (12 GB). Damit

konnten in etwa 42.000 Trainingsmuster der Größe 256×256 Pixel im Speicher gehalten werden. Bei mehr Mustern wurden während der gesamten Trainingsphase Teile der Mustermenge vom Betriebssystem auf die Festplatte ausgelagert (*Swapping*), wodurch das Training um bis zu Faktor 650 langsamer wurde.

Eine mit 80.000 Trainings- und 2.000 Testmustern (jeweils 50 % „clutter“) durchgeführte Messung lief deshalb nur 16 Epochen lang, wies aber danach bereits signifikant bessere Ergebnisse auf als eine halb so große Trainingsmenge nach doppelt so vielen Epochen (20,95 % statt 25,21 % Testfehler). Dieses Ergebnis lässt vermuten, dass größere Trainingsmengen zu besseren Erkennungsleistungen führen. Die beiden Ergebnisse sind allerdings mit Vorsicht zu betrachten, da die Testmenge mit lediglich 2.000 Mustern relativ klein und somit nicht so repräsentativ ist wie die zuvor verwendete Testmenge mit 10.000 Mustern.

Andere Netzstruktur Des Weiteren wurde die vierte Messung aus Tabelle 6.9 mit einem größeren Netz der Struktur 4(4)×256–9(7)×128–11(7)×64–15(7)×32–16×16–10 wiederholt. Dieses Netz hat genauso viele Eingabekarten, aber eine sich verdoppelnde Anzahl der sonstigen Karten in jeder Schicht. Die Ergebnisse dieser Messung waren nur geringfügig besser als die bereits vorgestellten. Aufgrund der höheren Ausführungsgeschwindigkeit wurden die nachfolgenden Messungen deshalb wieder mit dem kleineren Netz durchgeführt.

Verwendung von Weight Decay Beim Verfahren *Weight Decay* [32] wird der Gesamtfehler um einen Term erweitert, der mit der Summe des Betrages aller Verbindungsgewichte ansteigt. Umgesetzt wird dieses Verfahren durch die Multiplikation jedes Gewichts mit einer Zahl $1 - \varepsilon$, $\varepsilon > 0$, vor jeder Gewichtsaktualisierung. Das Verfahren wird häufig eingesetzt, um betragsmäßig zu großen Gewichten entgegenzuwirken.

Stichprobenhafte Messungen ergaben durchweg schlechtere Testfehler bei der Verwendung von Weight Decay. Dabei wurden Messungen mit $\varepsilon = 0,001$, $\varepsilon = 0,0001$ und $\varepsilon = 0,00001$ durchgeführt. Noch größere Werte führten dazu, dass der Testfehler wesentlich schlechter wurde, weil alle Gewichte Werte nahe Null aufwiesen. Auf der andere Seite konnte bei deutlich kleineren Werten kein nennenswerter Unterschied zur Messung ohne Weight Decay festgestellt werden.

Anderer Schwellwert für die Objekterkennung Für die bisher vorgestellten Messungen wurde ein Schwellwert von 0,5 verwendet, ab dem Muster als eine der 12 Objektklassen gewertet wurden. Die Veränderung dieses Schwellwerts wirkt sich auf Trainings- und Testfehler aus, weil sich dadurch die Anzahl der als „clutter“ klassifizierten Muster ändert.

Bei stichprobenhaften Messungen ergaben sich bei kleinen Schwellwerten (0,0 und -0,3) geringere Testfehler als bei dem oben verwendeten Wert von 0,5. Bei einer Messung mit 40.000 Trainings- und 2.000 Testmustern (je 50 % „clutter“) und einem Schwellwert von -0,3 betrug der Testfehler nach 500 Epochen 13,29 % (gemittelt über fünf Epochen).

Für die nachfolgend beschriebene Anwendung des Sliding Windows haben sich allerdings höhere Schwellwerte als bessere Abwägung zwischen False Positives und False Negatives herausgestellt (vergleiche dazu Abschnitt 5.7).

6.7.5 Ergebnisse des Sliding-Window-Algorithmus

Die Ausgaben des Sliding-Window-Algorithmus lassen sich nur sehr schwer automatisiert bewerten. Der Grund dafür ist, dass viele Objekte in Bildern des LabelMe-Datensatzes nicht annotiert sind. So würden bei einer automatischen Auswertung zum Beispiel alle positiven Ausgaben auf einem nicht annotierten Auto als Fehler gewertet werden. Aus diesem Grund werden in dieser Arbeit nur einige Ausgaben als Beispiele gezeigt.

Abbildung 6.12: Ausgabe des Sliding-Window-Algorithmus für ein Beispielbild. Bildquelle: [43]

Bei der Anwendung des Sliding Windows wurden als Parameter `shift` = 24 und `border` = 128 gewählt. Der Wert für `shift` ergibt sich daraus, dass während der Trainingsphase eine zufällige Verschiebung von ±12 Pixeln für jedes Trainingsmuster verwendet wurde. Der Wert von `border` wurde gewählt, um auch noch Objekte erkennen zu können, die sich genau zur Hälfte im Bild befinden.

Durch eine bisher nicht geklärte Eigenschaft der Texturfunktion `tex2D` [47] musste `border` jedoch auf 0 gesetzt werden, weil sonst alle Fenster, die über das Eingabebild hinausragten, komplett ungültige Werte aufwiesen. Durch dieses Problem werden bei den vorgestellten Ausgaben Objekte am Rand des Eingabebildes nicht korrekt erkannt.

Abbildung 6.12 zeigt ein Eingabebild und die daraus erzeugten Ausgabematrizen. Jeweils sechs Klassen wurden in einer Abbildung zusammengefasst. Jede Klasse wird durch eine eigene Farbe repräsentiert. Die Ausgabematrizen sind $\frac{1}{16}$-mal so groß wie das Eingabebild. Nach der Nomenklatur aus Abschnitt 5.7 ist also $r = 16$. Die Pixel der Ausgabematrix sind geglättet, weil sie auf die Größe des Eingabebildes hochskaliert wurden.

Die Ausführungszeit des Sliding-Window-Algorithmus hängt von der Größe des Eingabebildes ab. Bei einer Größe von 2048×1536 Pixeln bearbeitet der Algorithmus beispielsweise 25 Skalierungsstufen (vergleiche Abschnitt 5.7) und benötigt dafür insgesamt 6,4 Sekunden.

Weitere Beispiele für Ausgaben des Algorithmus sind in Anhang B zu finden.

7 Zusammenfassung, Diskussion und Ausblick

In der vorliegenden Arbeit wurde ein biologisch motiviertes System zur Objekterkennung entworfen und mit Hilfe des CUDA-Frameworks parallel implementiert. Anschließend wurde das System mit mehreren geeigneten Datensätzen evaluiert. Nachfolgend sind zunächst die wichtigsten Schritte des Entwurfs, der Implementierung und der Evaluation zusammengefasst. Danach folgt eine Diskussion der Ergebnisse und ein Ausblick auf mögliche weiterführende Arbeiten.

7.1 Zusammenfassung

Die Grundlage des vorgestellten Objekterkennungssystems bildet ein künstliches neuronales Netz. Dieses ist aus mehreren, hierarchisch angeordneten Schichten aufgebaut, die jeweils aus einer oder mehreren Karten bestehen. Verbindungen zwischen den Karten sind vorwärtsgerichtet und lokal verknüpft. Als Eingabe für das Netz dienen Kanten- und PCA-Kanäle, die aus dem ursprünglichen Eingabemuster berechnet werden. Eingaben können in mehreren Schichten erfolgen. Für das Lernen wird das Verfahren Backpropagation of Error verwendet.

Die Implementierung des Systems erfolgte so, dass performanzkritische Funktionen (Vorwärtspropagierung, Rückwärtspropagierung sowie Teile der Trainingsfunktionen für verschiedene Datensätze) entweder auf CPU oder mit Hilfe von CUDA auf der Grafikkarte ausgeführt werden können. Auf diese Weise konnten die beiden Varianten direkt miteinander verglichen werden. Die Speicherstruktur aller für diese Funktionen benötigten Variablen wurde so gewählt, dass auf den Devicespeicher stets zusammenhängend zugegriffen werden kann. Bei der Entwicklung der CUDA-Kernels wurde darauf geachtet, eine möglichst hohe Auslastung der GPU zu erreichen und die verschiedenen Arten von Speicher effizient zu nutzen. Zur Erkennung von Objekten in beliebig großen Eingabebildern wurde ein Sliding-Window-Algorithmus entwickelt, der das Eingabebild nacheinander in mehreren Skalierungsstufen bearbeitet.

Anschließend wurden die Ausführungszeit sowie die Erkennungsleistung anhand von drei verschiedenen Datensätzen evaluiert. Es zeigte sich, dass der Lernerfolg auf dem MNIST-Datensatz handgeschriebener Ziffern trotz unterschiedlicher Gleitkommaarithmetik bei CPU- und GPU-Variante beinahe identisch ist. Ein Vergleich der Ausführungszeit ergab, dass die GPU-Variante je nach verwendeter Netzstruktur um bis zu Faktor 82 schneller ist als die CPU-Variante. Dabei werden bei der GPU-Variante während der Trainingsphase rund $8 \cdot 10^9$ Gewichtsaktualisierungen pro Sekunde ausgeführt, was einer genutzten Speicherbandbreite von etwa 100 GB/s entspricht.

Als nächstes wurde die Erkennungsleistung des entworfenen Systems untersucht. Beim MNIST-Datensatz erfolgte die Eingabe lediglich in einer Karte, ohne Berechnung von Kanten- und PCA-Kanälen. Es wurde ein Testfehler von 1,74 % erreicht. Beim NORB-Datensatz wurde ein Testfehler von 5,28 % erreicht. Der dritte verwendete Datensatz ist LabelMe. Da hierfür keine festgelegten Trainings- und Testmengen existieren, wurde eine Mustermenge aus 12 Objektklassen sowie einer Kategorie mit Negativbeispielen erstellt. Die Menge besteht aus insgesamt 86.574 Mustern. Mit dieser Menge wurden verschiedene Messungen durchgeführt. Es zeigte sich, dass mit der Kombination von Kanten- und PCA-Kanälen als Eingabe

signifikant bessere Ergebnisse erzielt werden als mit nur einer dieser Eingaben oder einem Graustufenbild als Eingabe. Auch die Eingabe auf mehreren Schichten führte zu besseren Erkennungsleistungen auf der Testmenge.

Abschließend wurde der Sliding-Window-Algorithmus getestet. Die Erkennung von Objektklassen, welche viele Instanzen in der Trainingsmenge besitzen, funktionierte hierbei wesentlich besser als die der anderen Objektklassen.

7.2 Diskussion und Ausblick

Ein wesentliches Ziel dieser Arbeit war es, durch den mit der CUDA-Implementierung erreichten Geschwindigkeitsgewinn Trainings- und Testmengen in einer bisher nur schwer zu realisierenden Größenordnung zu verarbeiten. Mit der bis zu 82-fachen Ausführungsgeschwindigkeit gegenüber der CPU-Implementierung wurde dieses Ziel erreicht. Messungen, die mit der CUDA-Variante in einer Nacht ausgeführt werden können, würden mit der CPU-Variante beinahe einen Monat in Anspruch nehmen. Durch die schnell fortschreitende GPU-Entwicklung ist zu erwarten, dass sich dieser Faktor in Zukunft sogar noch erhöht.

Ein weiterer Beitrag dieser Arbeit ist die gewonnene Erfahrung bei der parallelen Implementierung neuronaler Netze. Da der Trend auch bei der CPU-Entwicklung zu immer mehr Kernen anstatt zu höherer Taktrate geht, gewinnen parallelisierte Algorithmen auch dort an Bedeutung.

Im Gegensatz zu seriell ablaufenden Programmen erfordern der Entwurf und die Implementierung parallel arbeitender Programme im Allgemeinen einen höheren Entwicklungsaufwand. Dies trifft auch für das Objekterkennungssystem in dieser Arbeit zu. Zum einen liegt das an der generellen Herausforderung, die der Entwurf parallel ausgeführter Programme mit sich bringt (Synchronisation, gute Auslastung der Hardwareressourcen et cetera). Zum anderen erfordert speziell CUDA eine relativ lange Einarbeitungszeit, weil die meisten Ressourcen manuell verwaltet werden müssen (zum Beispiel Prozessoren und Shared Memory). Hinzu kommt, dass die Dokumentation des Frameworks an einigen Stellen noch nicht ausgereift ist, was den Entwicklungsaufwand zusätzlich erhöht. Der Autor ist jedoch der Meinung, dass der erhöhte Aufwand auch zum aktuellen Zeitpunkt schon wegen der erreichten Ausführungszeit gerechtfertigt ist.

Der im Rahmen dieser Arbeit erstellte Datensatz natürlicher Bilder bietet eine Alternative zu den in Abschnitt 2.3 vorgestellten Datensätzen. Gegenüber NORB, Caltech-101 und Caltech-256 zeichnet er sich dadurch aus, dass alle Objekte aus natürlichen Bildern stammen und dementsprechend eine Vielfalt realistischer Lichtverhältnisse, Blickwinkel und Umgebungen aufweisen. Außerdem sind teilweise mehrere Objekte in einem Muster vorhanden und es treten zusätzlich erschwerte Bedingungen wie Verdeckungen auf. Der auf dem NORB-Datensatz erreichte Testfehler von 5,28 % im Vergleich zum Testfehler auf der erzeugten LabelMe-Mustermenge zeigt, dass letzterer eine große Herausforderung darstellt. Dies gilt zwar auch für den Pascal-Datensatz, dieser verfügt aber nach Meinung des Autors über zu wenige Muster, um ein Objekterkennungsystem so zu trainieren, dass es beispielsweise Personen ähnlich gut wie ein menschlicher Betrachter erkennen kann.

Die Anzahl der Muster ist aber auch in der erzeugten Mustermenge problematisch: So werden die am häufigsten vertretenen Klassen wie Autos und Personen relativ zuverlässig und unter vielen verschiedenen Blickwinkeln und Beleuchtungverhältnissen erkannt. Objektklassen mit weniger Instanzen (zum Beispiel Stühle, Tische und Köpfe) werden drastisch schlechter erkannt. Bei Anwendung des Sliding-Window-Ansatzes ist die Erkennungsleistung auf diesen Klassen nur wenig besser als Zufallswahrscheinlichkeit.

Dieser Punkt führt zur Frage, wie die Erkennungsleistung des entworfenen Systems im Vergleich zu bereits existierenden Ansätzen zu bewerten ist. Die direkten Vergleiche der Testfehler auf dem MNIST- und dem NORB-Datensatz (Abschnitt 6.5 und 6.6) legen nahe,

dass das System mit etablierten Methoden wie SVMs und Konvolutionsnetzen mithalten kann. Durch die hohe Anzahl der in dieser Arbeit durchgeführten Messungen war nur wenig Zeit vorhanden, um die Ergebnisse jeweils „auf das letzte Prozent“ zu optimieren. Der Autor vermutet daher, dass das vorgestellte Objekterkennungssystem auch mit geringen Veränderungen noch Potenzial für Verbesserungen aufweist.

Eine mögliche Verbesserung stellt ein unüberwachtes Vortraining mit sogenannten *Autoencodern* [23] dar. Dabei wird das Netz vor dem überwachten Training schichtweise darauf trainiert, mit einer Karte aus Schicht $n \in \{1, \ldots, N-1\}$ den Inhalt einer Karte aus Schicht $n-1$ möglichst genau zu reproduzieren und dabei eventuell sogar zu entrauschen (*Denoising Autoencoder* [64]). Eine ebenfalls denkbare Erweiterung wäre die Verwendung von rekurrenten Kartenverbindungen in Kombination mit dem Lernverfahren *Unfolding in Time* [57]. Seitens der Eingabecodierung könnten lokale Normalisierung oder aufwändigere Kanten- und Texturfilter die Erkennungsleistung verbessern.

Nicht zufriedenstellend ist bisher die Ausgabe des Sliding-Window-Algorithmus. Zur besseren Markierung von gefundenen Objekten sollte hier ein Clustering-Verfahren wie der *Mean-Shift-Algorithmus* [9; 18] angewandt werden. Dieses sollte zum einen Ausreißer unter den Ausgaben verwerfen und zum anderen Ausgaben verschiedener Skalierungsstufen unterscheiden. Damit könnten zum Beispiel mehrere kleine Objekte besser von einem großen unterschieden werden.

Um die Objekterkennung in großen Bildern oder Videos zu beschleunigen, könnte das System außerdem um ein Aufmerksamkeitsmodell wie zum Beispiel [28] erweitert werden, so dass der Sliding-Window-Algorithmus nicht immer auf das gesamte Bild angewandt werden muss, beziehungsweise „interessante“ Stellen bevorzugt untersucht werden.

A Codelisting der Vorwärts- und Rückwärtspropagierungs-Funktionen

Listing 6 Pseudocode der `forwardPassGPU`-Funktion

```
1  // Forward pass for regular layers
2  for layer = 0 to N-2 do
3      for map = 0 to M-1 do
4          let numconns = number of upward connections of map;
5          let numkernelcalls = numconns/8 + ((numconns%8!=0) ? 1 : 0);
6          let targetwidth = width of map / 2;
7          let isfirstconn = (map==0), islastconn = (map==M-1);
8
9          Bind activities of map to texture tex_sourceacts;
10
11         for k = 0 to numkernelcalls do
12             let firstindex = k * 8;
13             let lastindex  = (numconns - (k * 8)) % 8;
14
15             for c = firstindex to lastindex do
16                 Copy pointers to the weights of the c-th upward connection
17                 and pointers to the activities, biases and deltas of its target maps
18                 to structure c_data in constant memory;
19             end for
20
21             let num = lastindex - firstindex + 1;
22             let sharedmem = num * 256 * sizeof(float*);
23             dim3 blockDim(16, num, 32/num);
24             dim3 gridDim(targetwidth/16, targetwidth);
25             forwardKernel<<<gridDim, blockDim, sharedmem>>>
26                 (isfirstconn, islastconn);
27         end for
28     end for
29 end for
30
31 Set activities of all output neurons to their bias values;
32
33 // Forward pass for (fully connected) output layer
34 for each OutputConnection oc do
35     let W = weights of oc;
36     let A_i = activities of source map of oc;
37     let A_j = activities of output layer;
38     A_j += W · A_i;
39 end for
40
41 Calculate non-linearity, deltas and biases of output layer;
```

Listing 7 Code der `forwardPassKernel`-Funktion, Teil 1 von 2

```
1  __global__ void forwardPassKernel(bool firstconn, bool lastconn)
2  {
3      // Calculate target and source map width
4      const unsigned int tw = gridDim.y;
5      const unsigned int sw = tw * 2;
6
7      // Calculate x and y position of target neuron
8      const unsigned int x = 16*blockIdx.x + threadIdx.x;
9      const unsigned int y = blockIdx.y;
10
11     // Load weights of all target maps into shared memory...
12     extern __shared__ float s_weights[];
13     // Calculate continuous index of this thread (0,...,511)
14     const unsigned int idx = blockDim.y*blockDim.x*threadIdx.z +
15         blockDim.x*threadIdx.y + threadIdx.x;
16     // Determine if this thread is in the first or second half
17     // of the 512 threads of this block
18     const unsigned int half = (idx >= 256) ? 1 : 0;
19     // Determine which of the 16 weights of the current neuron
20     // shall be loaded
21     const unsigned int weightidx = (idx / 16) % 16;
22     // Loop to get all weights.
23     // As we have 512 threads and 256 weights per target map,
24     // we need to go through the loop (number of maps / 2) times.
25     // Note that the number of maps is always even.
26     for (unsigned int i=0; i<blockDim.y; i+=2)
27     {
28         // Determine address of the weights array in global memory
29         const float* weights = c_data[i+half];
30
31         // Determine position of the desired weight
32         const int X = 2*x - 1 + (weightidx % 4);
33         const int Y = 2*y - 1 + (weightidx / 4);
34         const unsigned int section = 3 - (2 * int(weightidx / 8) +
35             ((weightidx / 2) % 2)); // is in [0,1,2,3]
36         const unsigned int sourceidx = sw*sw*section + sw*Y + X;
37
38         // Determine target position in shared memory
39         const unsigned int targetidx =
40             256*(i+half) + 16*weightidx + threadIdx.x;
41
42         // Load weights. Set out-of-range weights to zero.
43         s_weights[targetidx] = (X>=0 && Y>=0 && X<sw && Y<sw) ?
44             weights[sourceidx] : 0.0f;
45     }
46
47     // Make sure all weights are available in shared memory
48     __syncthreads();
49
50     // Get addresses of biases, target activities and target deltas
51     float* biases       = c_data[1*8 + threadIdx.y];
52     float* targetacts   = c_data[2*8 + threadIdx.y];
53     float* targetdeltas = c_data[3*8 + threadIdx.y];
```

Listing 8 Code der `forwardPassKernel`-Funktion, Teil 2 von 2

```
54      // Loop to process all 16 patterns.
55      // The more target maps are processed in parallel,
56      // the more loop iterations are required
57      // because there are just 512 threads.
58      float targetact;
59      float4 sa1, sa2, sa3, sa4; // 16 source activities
60      for (unsigned int i=0; i<blockDim.y/2; i++)
61      {
62          // Determine which pattern will be processed
63          const unsigned int patternidx = blockDim.z*i + threadIdx.z;
64
65          // Load all 16 source activities of the current neuron
66          for (unsigned int j=0; j<blockDim.z; j++)
67          {
68              if (j == threadIdx.z)
69              {
70                  sa1.x = tex1Dfetch(tex_sourceacts,
71                      sw*sw*patternidx + sw*(2*y-1) + (2*x-1));
72                  sa1.y = tex1Dfetch(tex_sourceacts,
73                      sw*sw*patternidx + sw*(2*y-1) + (2*x  ));
74                  ...
75                  sa4.w = tex1Dfetch(tex_sourceacts,
76                      sw*sw*patternidx + sw*(2*y+2) + (2*x+2));
77              }
78          }
79
80          // If this is the first processed connection of the current
81          // map, initialize target acts with their bias, else load
82          // the current value. Additionally, reset the delta values
83          // to zero if this is the first connection.
84          if (firstconn)
85          {
86              targetdeltas[tw*tw*patternidx + tw*y + x] = 0.0f;
87              targetact = biases[tw*y + x];
88          }
89          else
90          {
91              targetact = targetacts[tw*tw*patternidx + tw*y + x];
92          }
93
94          // Calculate new target activity
95          const unsigned int patoffset = 256 * threadIdx.y;
96          targetact +=
97                s_weights[patoffset +  0*16 + threadIdx.x] * sa1.x
98              + s_weights[patoffset +  1*16 + threadIdx.x] * sa1.y
99              + ...
100             + s_weights[patoffset + 15*16 + threadIdx.x] * sa4.w;
101
102         // Write target activity back to global memory
103         targetacts[tw*tw*patternidx + tw*y + x] =
104             (lastconn ? tanh(targetact) : targetact);
105     }
106 }
```

Listing 9 Code der `backPropKernel`-Funktion, Teil 1 von 2

```
1  __global__ void backpropKernel(
2      float* weights, float* biases,
3      float* targetacts, float* targetdeltas,
4      float eta, bool calcderivative)
5  {
6      // Calculate target and source map width
7      const unsigned int sw = gridDim.x *  8; // source map width
8      const unsigned int tw = sw * 2;         // target map width
9
10     // Calculate target neuron position
11     const unsigned int x  = 16*blockIdx.x + threadIdx.x;
12     const unsigned int y  = 16*blockIdx.y + threadIdx.y;
13
14     // Calc. pos. of the upper left one of the four source neurons.
15     // Note that up to three source neurons may be out of range.
16     const int X = x/2 + x%2 - 1;
17     const int Y = y/2 + y%2 - 1;
18
19     // Load weights
20     float4 myweights;
21     myweights.x = weights[tw*tw*0 + tw*y + x];
22     myweights.y = weights[tw*tw*1 + tw*y + x];
23     myweights.z = weights[tw*tw*2 + tw*y + x];
24     myweights.w = weights[tw*tw*3 + tw*y + x];
25
26     // Load biases
27     float dbias = 0.0f;
28     const bool changebias =
29         (biases && (threadIdx.x%2 == 0) && (threadIdx.y%2 == 0));
30
31     // Check which of the source neuron positions are out of range
32     int4 outofrange;
33     outofrange.x = (x==0    || y==0   );
34     outofrange.y = (x==tw-1 || y==0   );
35     outofrange.z = (x==0    || y==tw-1);
36     outofrange.w = (x==tw-1 || y==tw-1);
37
38     // Declare variable for weight changes and initialize it to zero
39     float4 dweights = make_float4(0.0f, 0.0f, 0.0f, 0.0f);
40
41     // Process the 16 patterns of the Mini-Batch consecutively
42     float4 sourcedelta;
43     float targetact;
44     for (unsigned int i=0; i<16; i++)
45     {
46         // Load target activity
47         const unsigned int targetneuronindex = tw*tw*i + tw*y + x;
48         targetact = targetacts[targetneuronindex];
49
50         // Load source deltas
51         sourcedelta.x =
52             tex1Dfetch(tex_sourcedeltas, sw*sw*i + sw*(Y  ) + X  );
```

Listing 10 Code der `backPropKernel`-Funktion, Teil 2 von 2

```
53          sourcedelta.y =
54              tex1Dfetch(tex_sourcedeltas, sw*sw*i + sw*(Y   ) + X+1);
55          sourcedelta.z =
56              tex1Dfetch(tex_sourcedeltas, sw*sw*i + sw*(Y+1) + X   );
57          sourcedelta.w =
58              tex1Dfetch(tex_sourcedeltas, sw*sw*i + sw*(Y+1) + X+1);
59
60          // Change neuron deltas. Note that out-of-range weights are
61          // guaranteed to be zero so they do not influence the result
62          if (targetdeltas)
63          {
64              float targetdelta = targetdeltas[targetneuronindex];
65
66              targetdelta +=
67                    myweights.x * sourcedelta.x
68                  + myweights.y * sourcedelta.y
69                  + myweights.z * sourcedelta.z
70                  + myweights.w * sourcedelta.w;
71
72              if (calcderivative)
73                  targetdelta *= (1.0f - targetact*targetact);
74
75              targetdeltas[targetneuronindex] = targetdelta;
76          }
77
78          // Change bias
79          if (changebias)
80              dbias += sourcedelta.w;
81
82          // Change weights
83          dweights.x += sourcedelta.x * targetact;
84          dweights.y += sourcedelta.y * targetact;
85          dweights.z += sourcedelta.z * targetact;
86          dweights.w += sourcedelta.w * targetact;
87      }
88
89      // Write other weights back to global memory if source
90      // position is not out of range (else they are currently
91      // zero and must not be changed, see above)
92      if (!outofrange.x)
93          weights[tw*tw*0 + tw*y + x] = myweights.x + dweights.x*eta;
94      if (!outofrange.y)
95          weights[tw*tw*1 + tw*y + x] = myweights.y + dweights.y*eta;
96      if (!outofrange.z)
97          weights[tw*tw*2 + tw*y + x] = myweights.z + dweights.z*eta;
98      if (!outofrange.w)
99          weights[tw*tw*3 + tw*y + x] = myweights.w + dweights.w*eta;
100
101     // Write biases back to global memory
102     if (changebias)
103         biases[sw*y/2 + x/2] += eta * dbias;
104 }
```

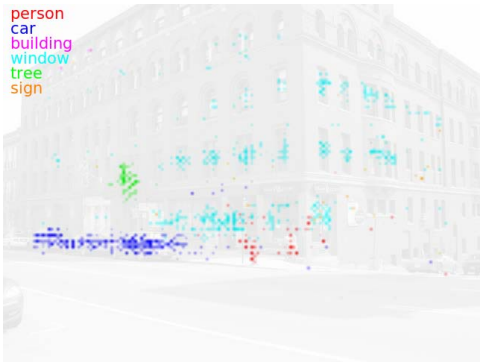
Listing 11 Pseudocode der `backPropGPU`-Funktion

```
1  // Backpropagation for (fully connected) output layer
2  for each OutputConnection oc do
3      let sourcemap = source map of oc;
4      let sourcewidth = width of sourcemap;
5
6      let W = weights of oc;
7      let D_j = deltas of outputlayer;
8      let D_i = deltas of sourcemap;
9      let A_i = activities of sourcemap;
10
11     // Calculate neuron deltas of the target feature map
12     D_i = W^T · D_j;
13
14     // Multiply neuron deltas by derivative of net input
15     dim3 blockDim(sourcewidth, sourcewidth);
16     dim3 gridDim(16);
17     deltaKernel<<<gridDim, blockDim>>>(D_i, A_i);
18
19     // Calculate and apply weight changes
20     W += η · (D_j · A_i^T);
21 end for
22
23 // Backpropagation for all regular layers
24 for layer = N-1 to 1 do
25     for map = 0 to M-1 do
26
27         Bind source deltas of map to texture tex_sourcedeltas;
28
29         for connection = 0 to K-1 do
30
31             let sourcemap = source map of connection;
32             let sourcewidth = width of sourcemap;
33             let biases = ((connection==0) ? biases of map : 0);
34             let calcderivative = ((map==M-1) ? true : false);
35
36             // Run backProp kernel
37             dim3 blockDim(16, 16);
38             dim3 gridDim(sourcewidth/16, sourcewidth/16);
39             backPropKernel<<<gridDim, blockDim>>>(
40                 weights of connection,
41                 biases,
42                 activities of sourcemap,
43                 deltas of sourcemap,
44                 η,
45                 calcderivative);
46         end for
47     end for
48 end for
```

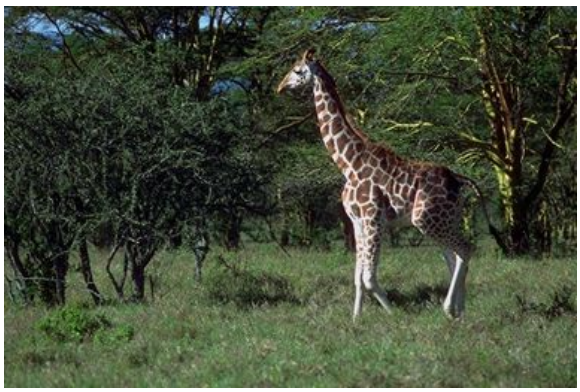
B Beispiele für Ausgaben des Sliding-Window-Algorithmus

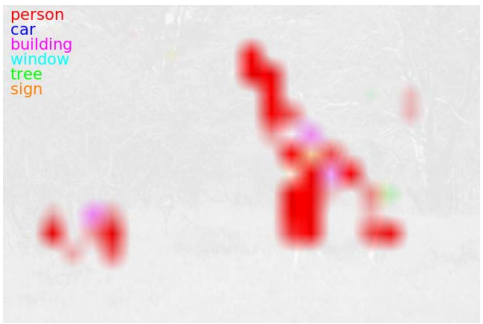
An dieser Stelle werden Ausgaben des Sliding-Window-Algorithmus für verschiedene Eingabebilder gezeigt. Die ersten fünf Bilder sind Teil des LabelMe-Datensatzes, die letzten beiden stammen aus dem CBCL-StreetScenes-Datensatz [42]. Die Bilder wurden so gewählt, dass sowohl subjektiv gute als auch schlechte Ausgaben gezeigt werden.

Die Größe der Ausgabematrix unterscheidet sich von Bild zu Bild, weshalb die Ausgabepixel unterschiedlich groß sind.

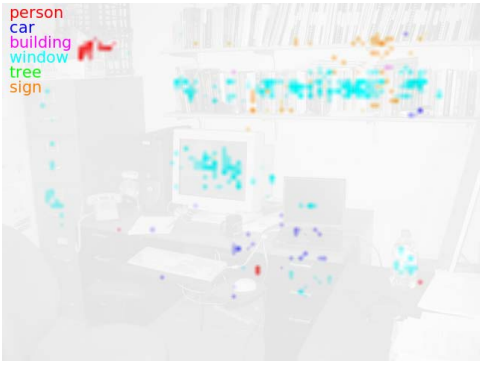

person
car
building
window
tree
sign

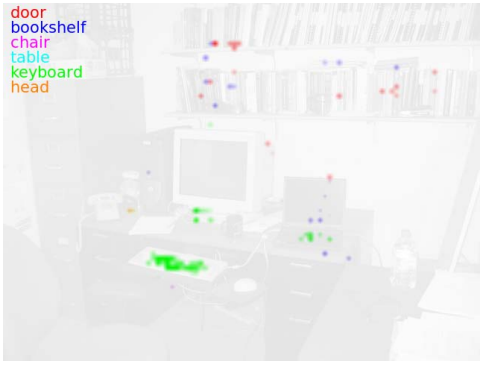

door
bookshelf
chair
table
keyboard
head

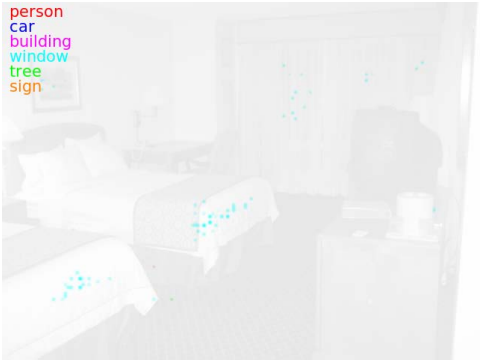


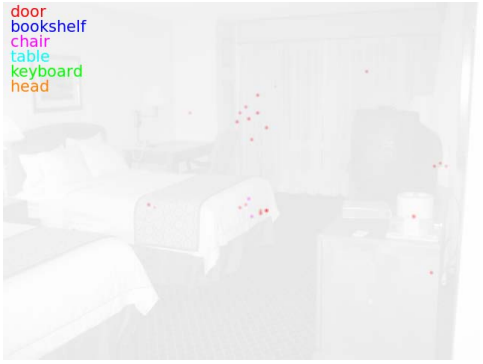

person
car
building
window
tree
sign

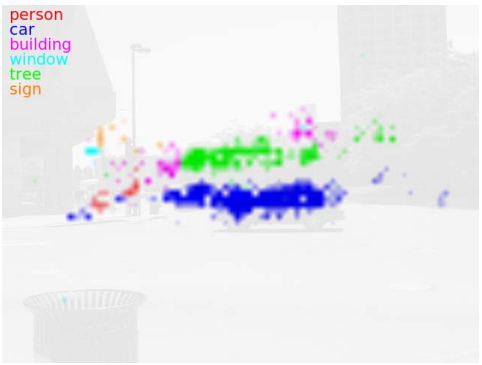

door
bookshelf
chair
table
keyboard
head


person
car
building
window
tree
sign

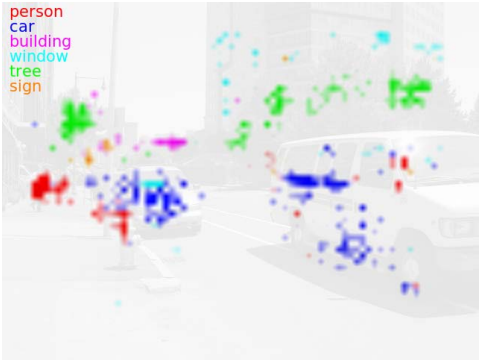

door
bookshelf
chair
table
keyboard
head

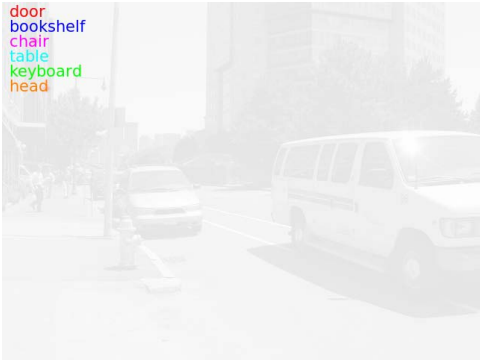

person
car
building
window
tree
sign


door
bookshelf
chair
table
keyboard
head


person
car
building
window
tree
sign


door
bookshelf
chair
table
keyboard
head


person
car
building
window
tree
sign


door
bookshelf
chair
table
keyboard
head

Literaturverzeichnis

[1] Behnke, Sven: Discovering hierarchical speech features using convolutional non-negative matrix factorization. In: *Proceedings of International Joint Conference on Neural Networks (IJCNN)* Bd. 4, 2003, S. 2758–2763

[2] Behnke, Sven: *Hierarchical Neural Networks for Image Interpretation*, Freie Universität Berlin, Diss., 2003

[3] Biederman, I.: Recognition-by-components: A theory of human image understanding. In: *Psychological review* 94 (1987), Nr. 2, S. 115–147

[4] Brügge, Bernd ; Dutoit, Allen H.: *Objektorientierte Softwaretechnik*. Pearson Studium, 2004. – ISBN 3–8273–7261–5

[5] Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. In: *Data mining and knowledge discovery* 2 (1998), Nr. 2, S. 121–167

[6] Campbell, Neil A. ; Reece, Jane B.: *Biologie*. 6. Auflage. Spektrum Akademischer Verlag, 2003. – ISBN 3–8274–1352–4

[7] Chellapilla, Kumar ; Puri, Sidd ; Simard, Patrice: High Performance Convolutional Neural Networks for Document Processing / Microsoft Research. 2006. – Forschungsbericht

[8] Chikkerur, Sharat: *NVIDIA CUDA Implementation of a Hierarchical Object Recognition Algorithm*. 2008. – `http://beowulf.lcs.mit.edu/18.337-2008/projects/reports/Carver_6.338FinalProjectWriteupv2.pdf`, zuletzt besucht am 31.03.2009

[9] Comaniciu, D. ; Meer, P.: Mean shift: a robust approach toward feature space analysis. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24 (2002), Nr. 5, S. 603–619

[10] Computational and Biological Learning Laboratory, New York University: *CBLL: Links*. 2008. – `http://www.cs.nyu.edu/~yann/links.html`, zuletzt besucht am 12.01.2009

[11] Cover, T. ; Hart, P.: Nearest neighbor pattern classification. In: *IEEE Transactions on Information Theory* 13 (1967), Nr. 1, S. 21–27

[12] Dalal, Navneet ; Triggs, Bill: Histograms of Oriented Gradients for Human Detection. In: *CVPR*, 2005, S. 886–893

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

[14] Fei-Fei, Li ; Fergus, Rob ; Perona, Pietro: Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories. In: *Computer Vision and Image Understanding* In Press, Corrected Proof

[15] Fei-Fei, Li ; Fergus, Rob ; Perona, Pietro: *Caltech101*. 2006. – http://www.vision.caltech.edu/Image_Datasets/Caltech101/, zuletzt besucht am 30.03.2009

[16] Freund, Y. ; Schapire, R.: A short introduction to boosting. In: *Journal of Japanese Society for Artificial Intelligence* 14 (1999), S. 771–780

[17] Friedman, J. ; Hastie, T. ; Tibshirani, R.: Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). In: *The Annals of Statistics* 28 (2000), Nr. 2, S. 337–407

[18] Fukunaga, K. ; Hostetler, L.: The estimation of the gradient of a density function, with applications in pattern recognition. In: *IEEE Transactions on Information Theory* 21 (1975), Nr. 1, S. 32–40

[19] Fukushima, Kunihiko: Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. In: *Biological cybernetics* 36 (1980), Nr. 4, S. 193–202

[20] Fukushima, Kunihiko: Recognition of partly occluded patterns: a neural network model. In: *Biological Cybernetics* 84 (2001), S. 251–259

[21] Griffin, G. ; Holub, A. ; Perona, P.: Caltech-256 Object Category Dataset / California Institute of Technology. Version: 2007. http://authors.library.caltech.edu/7694. 2007 (7694). – Forschungsbericht

[22] Hanser, Hartwig (Hrsg.) ; Scholtyssek, Christine (Hrsg.): *Lexikon der Neurowissenschaft*. Spektrum Akademischer Verlag, 2000. – ISBN 3–8274–0452–5

[23] Hinton, G. E. ; Salakhutdinov, R. R.: Reducing the Dimensionality of Data with Neural Networks. In: *Science* 313 (2006)

[24] Huang, Fu J. ; LeCun, Yann: *The Small NORB Dataset, V1.0*. 2005. – http://www.cs.nyu.edu/~ylclab/data/norb-v1.0-small/, zuletzt besucht am 27.05.2009

[25] Hubel, D. H. ; Wiesel, T. N.: Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. In: *J. Physiol* 160 (1962), S. 106–154

[26] Hyvärinen, Aapo: *What is independent Component Analysis?*. – http://www.cs.helsinki.fi/u/ahyvarin/whatisica.shtml, zuletzt besucht am 24.04.2009

[27] Institute of Electrical and Electronics Engineers, Inc (IEEE): *IEEE Standard for Binary Floating-Point Arithmetic*. 1985. – Download auf http://754r.ucbtest.org/standards/754.pdf

[28] Itti, Laurent ; Niebur, Ernst: A Model of Saliency-Based Visual Attention for Rapid Scene Analysis. In: *IEEE Transaction on Pattern Analysis and Machine Intelligence* Bd. 20, 1998, S. 1254–1259

[29] Jang, Honghoon ; Park, Anjin ; Jung, Keechul: Neural Network Implementation Using CUDA and OpenMP. In: *DICTA '08: Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, IEEE Computer Society, 2008, S. 155–161

[30] Kandel, Eric R. ; Schwartz, James H. ; Jessel, Thomas M.: *Principles of Neural Science*. Fourth Edition. McGraw-Hill, 2000. – ISBN 0–07–112000–9

[31] Kobatake, E. ; Tanaka, K.: Neuronal selectivities to complex object features in the ventral visual pathway of the macaque cerebral cortex. In: *J Neurophys* 71 (1994), S. 856–867

[32] Krogh, Anders ; Hertz, John A.: A Simple Weight Decay Can Improve Generalization. In: *Advances in Neural Information Processing Systems 4* (1995), S. 950–957

[33] Laboratory of Computer and Information Science, Helsinki University of Technology: *FastICA*. – http://www.cis.hut.fi/projects/ica/fastica/, zuletzt besucht am 24.04.2009

[34] Lahabar, Sheetal ; Agrawal, Pinky ; Narayanan, P.J.: High Performance Pattern Recognition on GPU. In: *Proceedings of National Conference on Computer Vision Pattern Recognition Image Processing and Graphics (NCVPRIPG'08)*, 2008, S. 154–159

[35] LeCun, Y. ; Bottou, L. ; Orr, G. ; Müller, K.: Efficient BackProp. In: Orr, G. (Hrsg.) ; K., Muller (Hrsg.): *Neural Networks: Tricks of the trade*, Springer, 1998

[36] Lecun, Yann ; Bottou, Léon ; Bengio, Yoshua ; Haffner, Patrick: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE*, 1998, S. 2278–2324

[37] LeCun, Yann ; Cortes, Corinna: *MNIST handwritten digit database*. 2008. – http://yann.lecun.com/exdb/mnist/, zuletzt besucht am 26.05.2009

[38] Lecun, Yann ; Huang, Fu J. ; Bottou, Léon: Learning methods for generic object recognition with invariance to pose and lighting. In: *Proceedings of CVPR'04*, IEEE Press, 2004

[39] Leibe, B. ; Leonardis, A. ; Schiele, B.: Combined object categorization and segmentation with an implicit shape model. In: *Workshop on Statistical Learning in Computer Vision, ECCV*, 2004, S. 17–32

[40] Lewicki, Michael S.: *Sensory Coding and Hierarchical Representations*. Neural Information Processing Systems Conference (NIPS) 2007 Tutorial, 2007. – Download auf http://nips.cc/Conferences/2007/Program/event.php?ID=577, zuletzt besucht am 17.03.2009

[41] Minsky, M. ; Papert, S.: *Perceptrons*. MIT Press, 1969

[42] MIT CBCL: *CBCL Code/Datasets*. 2000. – http://cbcl.mit.edu/software-datasets/index.html, zuletzt besucht am 10.04.2009

[43] MIT Computer Science and Artificial Intelligence Laboratory: *LabelMe: The open annotation tool*. 2009. – http://labelme.csail.mit.edu, zuletzt besucht am 30.03.2009

[44] Mutch, J. ; Lowe, D.G.: Multiclass object recognition with sparse, localized features. In: *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* Bd. 1, 2006, S. 11–18

[45] NVIDIA Corporation: *CUDA CUBLAS Library*. 9 2008. – Download auf http://www.nvidia.com/object/cuda_develop.html

[46] NVIDIA Corporation: *NVIDIA CUDA GPU Occupancy Calculator Version 1.5*. 2008. – http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls, zuletzt besucht am 31.03.2009

[47] NVIDIA Corporation: *NVIDIA CUDA Programming Guide Version 2.1*. 12 2008. – Download auf http://www.nvidia.com/object/cuda_develop.html

[48] NVIDIA Corporation: *GeForce GTX 280 – Specifications.* 2009. – `http://www.nvidia.com/object/product_geforce_gtx_285_us.html`, zuletzt besucht am 06.03.2009

[49] Oberhoff, D. ; Kolesnik, M.: Unsupervised shape learning in a neuromorphic hierarchy. In: *Pattern Recognition and Image Analysis* 18 (2008), Nr. 2, S. 314–322

[50] Pinto, N. ; Cox, D.D. ; DiCarlo, J.J.: Why is real-world visual object recognition hard. In: *PLoS Computational Biology* 4 (2008), Nr. 1, S. 151–156

[51] Poggio, Tomaso ; Girosi, Frederico: A theory of networks for approximation and learning. In: *A.I. Memo* 1140 (1989)

[52] Poli, Gustavo ; Saito, José H. ; Mari, Joao F. ; Zorzan, Marcelo R.: Processing Neocognitron of Face Recognition on High Performance Environment Based on GPU with CUDA Architecture. In: *SBAC-PAD '08: Proceeding of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, IEEE Computer Society, 2008, S. 81–88

[53] Ranzato, M.A. ; Poultney, C. ; Chopra, S. ; LeCun, Y.: Efficient learning of sparse representations with an energy-based model. In: *Advances in Neural Information Processing Systems* 19 (2006)

[54] Riesenhuber, M. ; Poggio, T.: Hierarchical Models of Object Recognition in Cortex. In: *Nature Neuroscience* 2 (1999), S. 1019–1025. – `http://maxlab.neuro.georgetown.edu/hmax.html`, zuletzt besucht am 27.03.2009

[55] Riesenhuber, Maximilian ; Poggio, Tomaso: Computational models of object recognition in cortex: A review / Artificial Intelligence Laboratory and Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology. 2000. – Forschungsbericht

[56] Rosenblatt, Frank: The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. In: *Psychological Review* 65 (1958), Nr. 6

[57] Rumelhart, D.E ; Hinton, G. E. ; Williams, R. J.: Learning internal representations by error propagation. In: *Parallel distributed processing: Explorations in the microstructure of cognition* 1 (1986), S. 318–362

[58] Russell, B. C. ; Torralba, A. ; Murphy, K.P. ; Freeman, W. T.: LabelMe: a database and web-based tool for image annotation. In: *International Journal of Computer Vision* 77 (2008), Nr. 1–3, S. 157–173

[59] Serre, T. ; Wolf, L. ; Bileschi, S. ; Riesenhuber, M. ; Poggio, T.: Robust object recognition with cortex-like mechanisms. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29 (2007), Nr. 3, S. 411

[60] Serre, T. ; Wolf, L. ; Poggio, T.: Object recognition with features inspired by visual cortex. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition 2005* Bd. 2, 2005

[61] Simard, PY ; Steinkraus, D. ; Platt, JC: Best practices for convolutional neural networks applied to visual document analysis. In: *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*, 2003, S. 958–963

[62] Torralba, A. ; Murphy, KP ; Freeman, WT: Sharing features: efficient boosting procedures for multiclass object detection. In: *Proceedings of Computer Vision and Pattern Recognition (CVPR) 2004* Bd. 2, IEEE

[63] Vapnik, V.N.: *The nature of statistical learning theory*. Springer, 2000

[64] Vincent, Pascal ; Larochelle, Hugo ; Bengio, Yoshua ; Manzagol, Pierre-Antoine: Extracting and composing robust features with denoising autoencoders. In: *ICML '08: Proceedings of the 25th international conference on Machine learning*, ACM, 2008, S. 1096–1103

[65] Viola, P. ; Jones, M.J.: Robust real-time face detection. In: *International Journal of Computer Vision* 57 (2004), Nr. 2, S. 137–154

[66] Wolf, L. ; Bileschi, S. ; Meyers, E.: *Perception strategies in hierarchical vision systems*. 2006

[67] Zell, Andreas: *Simulation neuronaler Netze*. 2., unveränderter Nachdruck. Oldenbourg, 1997. – ISBN 3–486–24350–0

Index

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe und dabei keine anderen Hilfsmittel als die in der Arbeit angegebenen benutzt habe. Zitate, die wörtlich oder sinngemäß aus anderen Arbeiten übernommen wurden, habe ich als solche kenntlich gemacht.

Bonn, den 8. Juni 2009

Rafael Uetz