Rheinische Friedrich-Wilhelms-Universität Bonn

Institut für Informatik
– AG Autonome Intelligente Systeme –

Diplomarbeit

im Studiengang Informatik

Lokalisierung von komplexen Bildmerkmalen mit Konvolutionsnetzen

eingereicht von: Markus Gerhards

eingereicht am: 21. September 2010

Betreuer: Prof. Dr. Sven Behnke
Prof. Dr. Joachim K. Anlauf

Inhaltsverzeichnis

1 Ziele und Aufgabenstellung

1.1 Problemstellung

1.2 Zielsetzung

Im Rahmen dieser Diplomarbeit soll mit einem Konvolutionsnetz gelernt werden, in gegebenen Bildern komplexe Bildmerkmale, wie beispielsweise Augen zu lokalisieren. Das Modell soll schätzen können, an welcher 2D-Position im Bild die Merkmale auftreten. Aus diesen gewonnenen Informationen können dann in einem weiteren Schritt wie in der Arbeit zur Gestenerkennung von Axenbeck et al. [ABBB08] Merkmale generiert werden, aus denen robust Bewegungen erkannt werden. Als Grundlage dafür dient die Arbeit von Sven Behnke, in der das Konzept der Neuronalen Abstraktionspyramide vorstellt. Mit diesem Modell hat Behnke bereits erfolgreich Augen in dem umfangreichen Testdatensatz mit Graustufenbildern „BioID“ markieren können.

Obwohl die Arbeit sich vor allem mit dem Problem der Lokalisation beschäftigt, soll das zu entwickelnde Framework für die Generierung von Konvolutionsnetzen auch den Entwurf rekurreter Netze ermöglichen. Zusätzlich soll das Modell mit Hilfe einer CUDA-fähigen GPU parallelisiert werden, um auch die Verarbeitung von größeren Bildern in Echtzeit zu ermöglichen.

1.3 Aufbau der Arbeit

In Kapitel 2 wird zunächst ein kurzer Überblick über die in dieser Arbeit als grundlegendes Wissen vorausgesetzten Themenbereiche gegeben. Hier werden *Neuronale Netze* biologisch motiviert und ihre technische Umsetzung beschrieben. Über die Experimente von Hubel und Wiesel, die herausfanden, dass der visuelle Cortex von Säugetieren *rezeptive Felder* hat, werden *Konvolutionsnetze* motiviert. Anschliessend wird kurz beschrieben, welche allgemeinen Techniken sich durchgesetzt haben. Draufhin wird die Funktion von *Gradientenabstiegsverfahren* als Trainingsverfahren für neuronale Netze – auch für rekurrente Netze – erläutert. Der letzte Abschnitt in Kapitel 2 beschreibt *CUDA*, die Programmierschnittstelle für NVIDIAs GPUS.

In Kapitel 3 werden einige Arbeiten aus dem Bereich der Gesichtslokalisierung vorgestellt.

Kapitel 4 beschäftigt sich mit dem Entwurf und der Implementierung des Netzes. Es werden zunächst allgemeine Konzepte und Überlegungen vorgestellt, bevor die Implementierung der Neuronen und Verbindungsschichten erläutert wird.

Kapitel 5 beschäftigt sich mit den durchgeführten Experimenten. Hier werden zunächst Experimente vorgestellt, mit denen die Funktionsweise der Komponenten getestet werden kann. Anschliessend werden die Experimente zur Detektion von Augen sowohl mit rein vorwärtsgerichteten Netzen, als auch mit der eigentlichen Arbeit, den rekurrenten Netzen, vorgestellt.

In Kapitel 6 werden die Ergebnisse noch einmal zusammengefasst und Ideen diskutiert, die in Zukunft noch umgesetzt werden könnten.

2 Grundlagen

2.1 Zusammenfassung

In diesem Kapitel werden die Grundlagen für die in dieser Arbeit verwendeten Techniken besprochen. Erst werden die verwendeten Elemente biologisch motiviert. Ein kurzer Überblick über künstliche neuronale Konvolutions-Netze führt auch die im Folgenden verwendeten Notation ein. Anschließend wird die CUDA-Plattform der Firma NVIDIA zur Programmierung von parallelen Multiprozessoren vorgestellt.

2.2 Neuronale Konvolutions-Netze

2.2.1 Biologische Nervenzellen

Einzelne Nervenzellen dienen als Grundbaustein für komplexe neuronale Netze in natürlichen Organismen. Auch wenn es viele verschiedene Arten von Nervenzellen gibt, liegt ihr Aufbau ein gemeinsames Schema zugrunde, welches sie als Klasse von anderen Zellarten unterscheidet.

Im Folgenden sollen kurz die an der Informationsverarbeitung beteiligten Teile einer Nervenzelle dargestellt werden. Eine tiefere Einführung in die Funktionsweise von Nervenzellen, die auch als Grundlage für diesen Abschnitt dient, findet sich in [KSJ96].

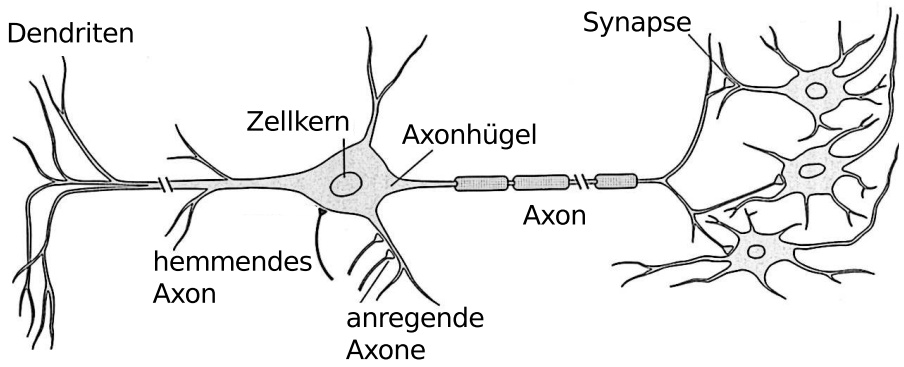


Abbildung 2.1: Schematische Darstellung eines Neurons. Quelle: [KSJ96], bearbeitet

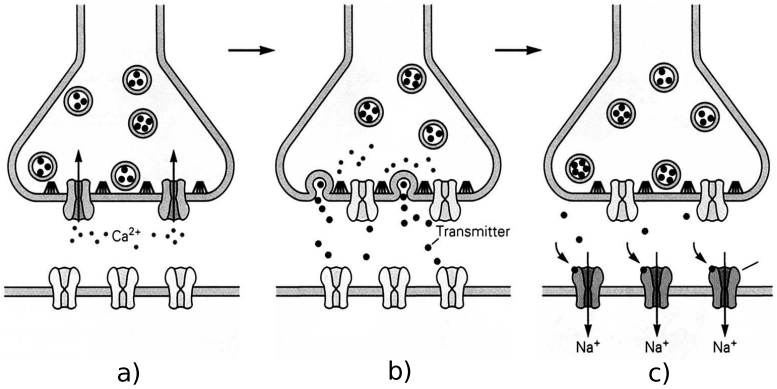


Abbildung 2.2: Informationsverarbeitung am synaptischen Spalt: a) Ein Aktionspotential erreicht das präsynaptische Ende (oben) und lässt positiv geladene Ca^{2+}-Ionen einströmen, b) Die einströmenden $Ca^{2+} - Ionen$ sorgen dafür, dass Vesikel mit Neurotransmitter an der Membran der Synapse andocken und die Transmitter freisetzen. c) Die Neurotransmitter bewirken eine Öffnung der Ionenkanäle auf der postsynaptischen Seite, die dazu führt, dass posititve Na^{+}-Ionen einströmen. Quelle: [KSJ96], bearbeitet

Neben dem Zellkern besteht die Nervenzelle (Abbildung 2.1) aus zwei weiteren Teilen: Dem Axon und den Dendriten. Eine Nervenzelle hat viele Dendriten, die weit verzweigte Bäume bilden. Sie nehmen die Informationen von anderen Zellen auf und leiten sie zum Axonhügel weiter. Dort wird das elektrische Aktionspotential – das Signal – der Zelle ausgelöst. Es handelt sich dabei um einen elektrischen Impuls von etwa 1 ms Dauer und einer Amplitude von etwa 100 Millivolt.

Das Axon der Nervenzelle ist ein einzelner Fortsatz, der vom Axonhügel ausgeht, und die Aufgabe hat, dieses Potential über Entfernungen von bis zu zwei Metern zu transportieren. Das Axon kann dabei den Impuls über die gesamte Entfernung erhalten. An seinem Ende verzweigt sich das Axon und bildet Synapsen aus, die Kontakt zu anderen Zellen herstellen. Die Nervenzelle, die ein Signal überträgt, heißt *präsynaptisch*; diejenige, die ein Signal empfängt, wird als *postsynaptisch* bezeichnet. Die prä- und postsynaptischen Zellen sind nicht direkt miteinander verbunden. Zwischen ihnen befindet sich der *synaptische Spalt*.

Das elektrische Signal wird am synaptischen Spalt (vgl. Abbildung 2.2) in ein chemisches Signal umgewandelt. Dies geschieht in mehreren Schritten. Erreicht das Aktionspotential das Ende der Synapse der präsynaptischen Nervenzelle, so führt das zunächst dazu, dass sich spannungsgesteuerte Ca^{2+}-Kanäle öffnen und positiv geladenen Ca^{2+}-Ionen in die Zelle einströmen. Dieser Mechanismus führt dazu, dass sich in der Synapse einge-

lagerte Vesikel mit Neurotransmittern mit der Zellmembran verschmelzen und dabei die Transmitter in den synaptischen Spalt freisetzen.

Für diese Neurotransmittermoleküle gibt es Rezeptoren auf der postsynaptischen Seite des Spaltes, welche die Neurotransmitter binden. Diese Rezeptoren öffnen, wenn die Transmitter an sie gebunden werden, Ionenkanäle, die ebenfalls positiv geladenen Na^+-Ionen einströmen lassen. Deren Vorhandensein innerhalb der aufnehmenden Nervenzelle de-polarisiert deren elektrische Ladung und führt zu einer Reizung. Das übertragene Signal kann die Aktivität der postsynaptischen Zelle anregen oder hemmen. Daher spricht man von *exitatorischen* (anregenden) und *inhibierenden* (hemmenden) Verbindungen.

Interessant ist das Zusammenspiel von elektrischer und chemischer Leitung der Signale. Bei näherer Betrachtung stellen beide Verfahren allerdings eigene interessante Eigenschaften zur Verfügung. Die elektrische Übertragung ist im Vergleich zur chemischen Übertragung schneller. Die chemische Übertragung kann hingegen die Reizung verstärken, da nur ein Signal an der Synapse ankommen muss, um die Ausschüttung der Neurotransmitter zu ermöglichen.

Bereits 1949 postulierte Donald Hebb eine Lernregel für die Verbindung zwischen Nervenzellen:

> Wenn ein Axon der Zelle A [...] Zelle B erregt und wiederholt und dauerhaft zur Erzeugung von Aktionspotentialen in Zelle B beisteuert, so resultiert dies in Wachstumsprozessen oder metabolischen Veränderungen in einer oder beiden Zellen, die bewirken, dass die Effizienz von Zelle A in Bezug auf die Erzeugung von Aktionspotenzialen in B größer wird. [Heb49]

Neuere Untersuchungen, wie beispielsweise von Bi und Poo [BP01], untermauern diese Hypothese. Sie bringen diese Lernprozesse vor allem mit den chemischen Wechselwirkungen in Verbindung und betonen die Bedeutung der zeitlichen und räumlichen Bedeutung beim Lernen: Die Synapsen fungieren als gerichtete Verbindungen und werden verstärkt, wenn erst die präsynaptische und dann die postsynaptische Seite aktiv waren.

2.2.2 Künstliche Neuronale Netze

Künstliche Neuronale Netze versuchen die Funktionsweise ihres biologischen Vorbildes in vereinfachter Form nachzubilden. Eine tiefere Einführung in künstliche neuronale Netze findet sich unter anderem in [Roj96], [Hay94],[Zel94] und [Kri07].

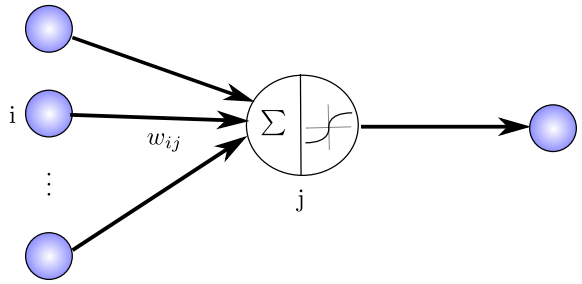


Abbildung 2.3: Schematische Darstellung der Verbindung von Neuronen und des Aufbaus eines Neurons j: Zunächst werden die eingehenden Gewichte aufsummiert, anschließend wird eine Aktivierungsfunktion auf die Summe angewendet.

Eine grundlegende Klasse von neuronalen Netzen ist das Perzeptron. Es gibt eine Eingabeschicht mit Neuronen und eine Ausgabeschicht mit Neuronen. Die Neuronen in den Schichten haben sowohl eingehende, als auch ausgehende Verbindungen. Beim Perzeptron gehen die Verbindungen von einem Neuron der Eingabeschicht l_i zur Ausgabeschicht l_{i+1}. Solche Netze werden als *vorwärts gerichtete Netze* bezeichnet. Prinzipiell wäre es jedoch auch möglich Verbindungen zwischen beliebigen Neuronen zuzulassen. In diesem Fall würden sich dann zyklische Abhängigkeiten ergeben. Solche Netze heißen *rekurrent*.

Die künstlichen Neurone sollen verschiedene Aspekte der in Abschnitt 2.2.1 vorgestellten biologischen Nervenzellen nachbilden. So produziert ein Neuron i eine *Ausgabe*, welche die Aktivierung repräsentiert und im Folgenden mit o_i bezeichnet wird. Ein Neuron i, dessen Ausgabe als Eingabe für ein Neuron j dient wird als *Eingabeneuron* für Neuron j bezeichnet. Die Ausgabe des Eingabeneurons wird mit einem *Verbindungsgewicht* w_{ij}, von einem Neuron i zu einem Neuron j, multipliziert, was sich an die Vorgänge an den Synapsen der biologischen Vorbilder anlehnt. Das Produkt aus den Ausgaben und den Verbindungsgewichten wird im Neuron aufsummiert und dient dann als *Netzeingabe*:

$$net_j = \sum_{i \in Eingabeneurone} w_{ij} o_i.$$

Damit stellt net_j also die eingehende Reizung des Neurons j dar. Die Ausgabe wird dann in Anlehnung an die Informationsverarbeitung im Axonhügel durch eine nicht-lineare *Aktivierungsfunktion* f_{act} geschickt:

$$o_j = f_{act}(net_j)$$

Häufig wird f_{act} so gewählt, dass der Bildbereich in dem Intervall $[0, 1]$ oder $[-1, 1]$ liegt. Gängige Aktivierungsfunktionen (vgl. Abbildung 2.4) sind

- die *Stufenfunktion*:

$$f_{act}(x) = \begin{cases} 0, & \text{falls } x < 0 \\ 1, & \text{sonst} \end{cases} \tag{2.1}$$

- die *Linearfunktion*:

$$f_{act}(x) = x \tag{2.2}$$

- die *Logistische Funktion*:

$$f_{act}(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

- der *Tangens Hyperbolicus*:

$$f_{act}(x) = \tanh(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{2.4}$$

Im Zusammenhang mit gradientenbasierten Lernverfahren (siehe den folgenden Abschnitt 2.2.5) ist es jedoch erforderlich, dass die Aktivierungsfunktion stetig differenzierbar ist. Daher findet die Stufenfunktion nur in einschichtigen Netzen Verwendung. In diesem Zusammenhang sind die Sigmoidfunktion und der Tangens Hyperbolicus vorteilhaft, da sie sich einfach ableiten lassen. Für die Sigmoidfunktion gilt:

$$f'_{sigm}(x) = f_{sigm}(x) \cdot (1 - f_{sigm}(x))$$

und für den Tangens Hyperbolicus:

$$f'_{tanh}(x) = 1 - f^2_{tanh}(x) = 1 - \tanh(x)^2$$

Das Perzeptron kann als Funktionsapproximator genutzt werden, der den Verlauf unbekannter Funktionen aus verschiedenen Stützstellen nachbildet. In der Literatur werden Perzeptrone – insbesondere auch Multi-Layer-Perzeptrone, also Perzeptrone mit mehreren Schichten – auch als Klassifizierer beschrieben. Auch bei diesem Einsatzgebiet lernt ein Perzeptron eine Funktion, welche den Eingaberaum in Klassen teilt: die Trennfunktion. Genauer gesagt, kann ein Perzeptron mit n Eingabeneurone eine $(n-1)$-dimensionale Hyperebene lernen, die die Eingaben separiert. Ein einschichtiges Perzeptron kann also linear separierbare Mengen – also solche, die durch eine Hyperebene trennbar sind – klassifizieren.

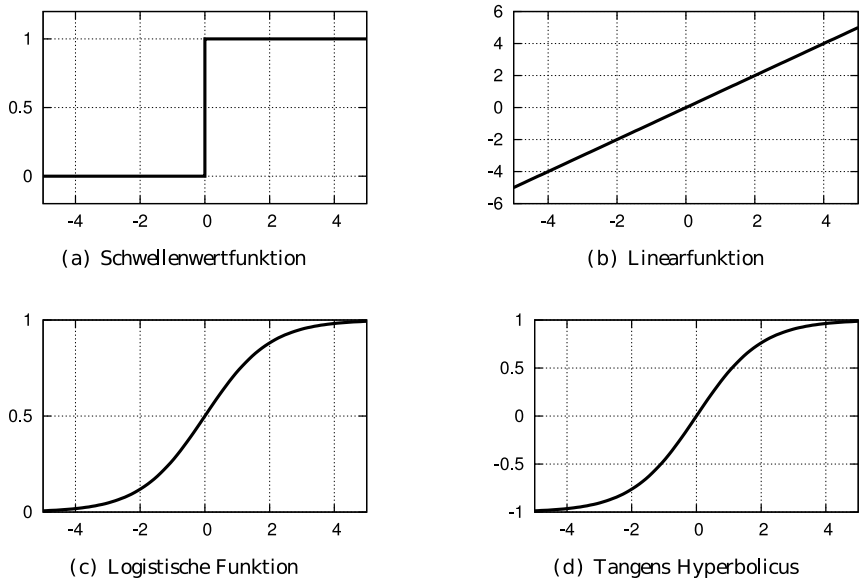


Abbildung 2.4: Verschiedene gängige Aktivierungsfunktionen f_{act} eines Neurons. Quelle: [SMB10a]

Mittels mehrerer Schichten können die entstehenden Halbräume mit einander geschnitten und somit komplexere Trennfunktionen gelernt werden. Ein dreischichtiges MLP kann bereits mehrere trennende Hyperebenen und somit konvexe Polygone in einem Eingaberaum lernen.

2.2.3 Informationsverarbeitung im visuellen Kortex

Das Konzept für Konvolutionsnetze wurde auch durch die Erkenntnisse der Modelle entwickelt, die für die Verarbeitung visueller Informationen bei Säugetieren gewonnen wurden. Sie sind durch zwei für diese Arbeit wesentlichen Prinzipien gestützt:

1. Die Verarbeitung erfolgt hierarchisch und
2. die Verarbeitung erfolgt auf lokalen rezeptiven Feldern.

Um dies zu verdeutlichen wird im Folgenden kurz erläutert, wo diese Eigenschaften in den Modellen vom Sehsystem der Säugetiere zu finden sind. Auch diese Beschreibung orientiert sich an [KSJ96] und den Erkenntnissen, die Hubel und Wiesel experimentell bereits 1959 gewonnen haben [HW59].

Die Informationsverarbeitung der visuellen Reize beginnt bereits in der Retina, wenn das Licht durch die Linse im Augeninneren auftrifft. Die Retina lässt sich in drei Bereiche einteilen:

- einer lichtempfindlichen Schicht, die die einfallenden Lichtreize in Nervenimpulse umwandelt.
- eine Zwischenschicht, die aus horizontalen und vertikalen Verbindungsneuronen besteht und
- der Ganglienzellschicht, die die Information an den Sehnerv weiterleitet.

Die lichtempfindliche Schicht des Auges besteht aus Nervenzellen, die die Aufgabe haben, Lichtreize in Nervensignale umzuwandeln – den Stäbchen (für das *Hell-Dunkel-Sehen*) und den Zapfen (für das *Farbsehen*). Diese Nervenimpulse werden über die Ganglienzellen als Folge von Aktionspotentialen an das Gehirn geleitet. Bevor die Lichtreize aber an die Ganglienzellen weitergeleitet werden, passieren sie eine Schicht von *Interneuronen*, die aus drei Klassen von Neuronen bestehen: den Bipolar-, den Horizontal- und den Amakrinzellen (siehe Abbildung 2.5).

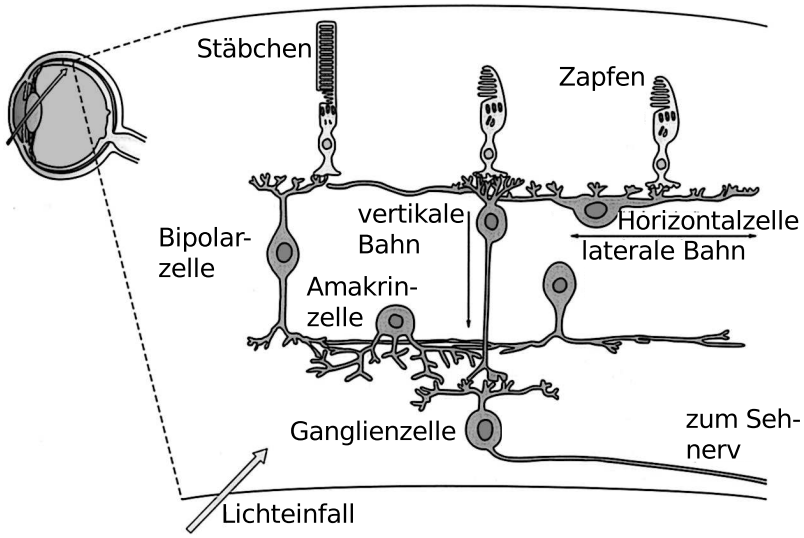


Abbildung 2.5: Aufbau der Netzhaut (Retina): Quelle: [KSJ96]

Über diese Zellen haben die Ganglienzellen nicht nur Zugriff auf die direkt benachbarten lichtempfindlichen Zellen, sondern auf eine größere Nachbarschaft. Diese Nachbarschaft wird als *rezeptives Feld* der Ganglienzelle bezeichnet. Zwei Eigenschaften zeichnen die rezeptiven Felder aus. Zum Einen sind sie kreisförmig und zum Anderen teilt sich das Feld in einen runden Zentrumsbereich und einen ringförmigen Aussenbereich auf. Werden das Zentrum und sein Umfeld unterschiedlich belichtet, so ist die Reaktion der Ganglienzelle, zu der das rezeptive Feld gehört, am stärksten. Daher können die rezeptiven Felder lokale Kontrastunterschiede besonders gut wahrnehmen. Auch im visuellen Cortex fanden Wiesel und Hubel spezialisierte Nervenzellen, die auf bestimmte Reize reagieren während sie andere ignorieren. Das von ihnen entwickelt Modell des visuellen Cortex des Menschen

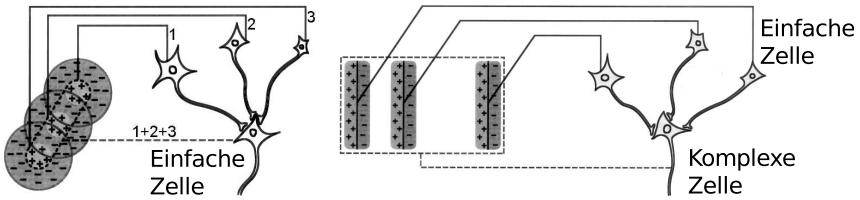


Abbildung 2.6: Einfache und komplexe Zellen im visuellen Kortex: Quelle: [KSJ96]

besteht aus sechs Schichten, denen die Erkennung von einfachen Formen zugeschrieben wird. Auch dort arbeiten Zellen mit rezeptiven Feldern. Hier liegen diese allerdings nicht auf dem direkten Lichteinfall, sondern auf den rezeptiven Feldern, die von der Retina durch das *Corpus geniculum laterale* in einer (Eingangs-)Schicht des Visuellen Cortex ankommen. Anhand ihrer Antworten auf visuelle Stimuli haben Hubel und Wiesel die corticalen Zellen in zwei verschiedene Typen gegliedert (siehe Abildung 2.6).

Die *einfachen* Zellen haben längliche und lineare rezeptive Felder die größer sind als diejenigen in der Retina. Ihre Eingabe stammt von der der Eingabeschicht des visuellen Cortex. Sie sind zudem lageabhängig und reagieren besonders gut auf Reize, die in der richtigen Orientierung an der richtigen Stelle auftreffen. Ihre rezeptiven Felder eignen sich zur Detektion komplexerer Stimuli, wie Balken oder Kanten, die in einem bestimmten Winkel liegen. Die *komplexen Zellen* haben größere rezeptive Felder und beziehen ihre Eingabe von mehreren einfachen Zellen. Sie haben keine genaue On- und Off-Zonen und reagieren positionsinvariant auf bestimmte Reize. Man geht insgesamt davon aus, dass das Sehsystem von der Retina bis zum visuellen Cortex eine Hierarchie von Erkennungsmerkmalen besitzt (siehe Abbildung 2.7). Auf den unteren Schichten des Systems werden lokal einfache Strukturen erkannt. Je höher die Schicht desto komplexere Formen werden erkannt und desto größer wird das rezeptive Feld.

Ein Modell, welches sich an die oben skizzierte Informationsverarbeitung anlehnt, findet sich im Neocognitron von Fukushima (siehe den folgenden Abschnitt 2.2.4).

2.2.4 Konvolutionsnetze

Konvolutionsnetze wurden durch die Prinzipien der Informationsverarbeitung im visuellen Cortex inspiriert. In den vergangenen Jahren wurden sie erfolgreich auf Aufgaben der Objektklassifikation und Gesichtsdetektion, wie beispielsweise zur Buchstaben- oder Zahlenerkennung unter geringer Rotation angewandt (vergleiche Abschnitt 3). Zu den Vorteilen der Konvolutionsnetze zählen, dass sie im Gegensatz zu anderen Netzen robust

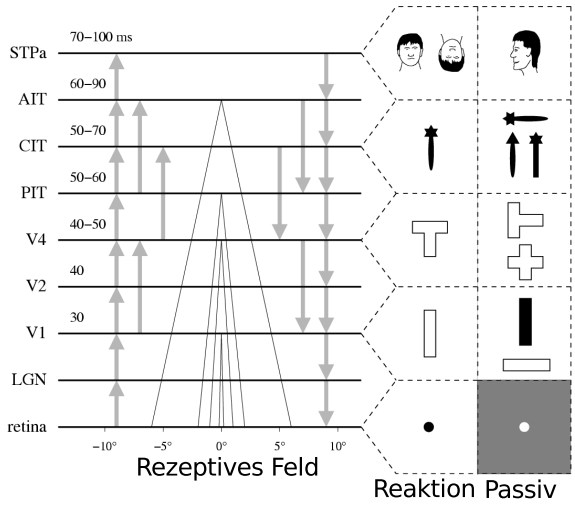


Abbildung 2.7: Aufbau des Auges: Quelle: [Wis05], bearbeitet

gegenüber Translationen, Skalierungen, Deformierungen, Beleuchtung und teilweise sogar gegen Verdeckungen sind.

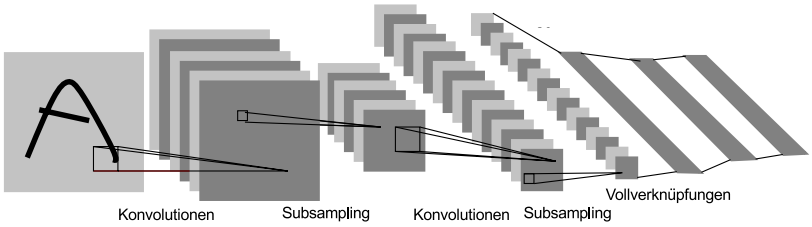


Abbildung 2.8: Struktur eines Konvolutionsnetzes am Beispiel des LeNet-5 (vgl. Abschnitt 3.1.2). nach:[LCJB+89]

Konvolutionsnetze sind schichtweise aufgebaut (vgl. Abbildung 2.8). Jede Schicht hat eine topologische Struktur, in der jedes Neuron eine feste Position hat, die mit einer Position in dem Eingabebild korrespondiert. Zwischen diesen Schichten gibt es mehrere Arten von Verbindungen: Konvolutionen, Subsampling oder Vollverknüpfungen. Entsprechend der Operation, die auf die Eingabeneurone bzw. die Eingabeschicht angewandt werden, spricht die Literatur von *Konvolutionsschichten* oder *Subsamplingschichten*.

2.2.4.1 Vollverknüpfte Schichten

Vollverknüpfte Schichten funktionieren nach dem Prinzip des Perzeptrons: Alle Neuronen der Zielschicht sind mit allen Neuronen der Eingabeschicht verbunden. Diese Verknüpfung

zu berechnen ist entsprechend aufwendig. Daher werden sie bei Konvolutionsnetzen in der Regel nur dort angewendet, wo die Auflösung durch mehrfaches Subsampling bereits drastisch reduziert wurde: an der Spitze der Pyramide.

Die Funktion der Schicht ist ebenfalls analog zu der Funktion des Perzeptrons. Bereits gewonnene Merkmale sollen integriert werden und klassifisziert werden. Im Rahmen der Objekterkennung soll beispielsweise entschieden werden, ob ein Objekt in dem Eingabebild zu sehen ist, oder nicht. Der Grad der Aktivierung kann zusätzlich als Grad der Sicherheit der Schätzung gewonnen werden, so dass das Ergebnis nicht durch einen Binärwert, sondern durch eine reelle Zahl $a \in [0, 1]$ oder $a \in [-1, 1]$ repräsentiert wird. Die kleinste Zahl bedeutet dabei in der Regel, dass das gesuchte Objekt ziemlich sicher nicht im Bild ist. Die größte Zahl zeigt an, dass das gesuchte Objekt ziemlich sicher im Bild zu finden ist.

2.2.4.2 Konvolutionsschichten

Die Schichten, die eine Faltungsoperation auf ihren Eingabeschichten realisieren, heißen Konvolutionsschichten. Der Rechenaufwand einer Konvolutionsschicht ist dadurch beherrschbar, dass jedes Neuron der Zielschicht ein gleichgroßes rezeptives Feld hat. Alle Felder nutzen die gleichen Gewichte (*weight sharing*), so dass die Anwendung der Gewichte als Faltung realisiert werden kann. Jede Konvolutionsschicht hat eine Eingabeschicht und N Filter der Größe $(k \times k)$. Die Eingabeschicht enthält ein oder mehrere Eingabebilder der Größe $(m \times n)$. Sie werden mit jedem der N Filter zu N Ausgabebildern der Größe $(m - k + 1) \times (n - k + 1)$ gefaltet. Jedes Neuron (i, j) der Konvolutionsschicht ist dazu mit einem Neuron der Eingabeschicht an der Position$(i + \lfloor (k/2) \rfloor, j + \lfloor (k/2) \rfloor)$ und einer Nachbarschaft, dem rezeptiven Feld, assoziert. Die Größe des rezeptiven Feldes wird durch den verwendeten Filter bestimmt. Der Filter besteht aus einer Menge von Filtergewichten, die in einer rechteckigen Struktur angeordnet sind:

$$W = \begin{matrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1} & w_{k2} & \dots & w_{kk} \end{matrix}$$

Eine Neuronenaktivierung wird berechnet, indem in einem Bereich um die Position des Zielpixels alle Aktivierungswerte $a_{x,y}$ mit einem Filtergewicht multipliziert werden.

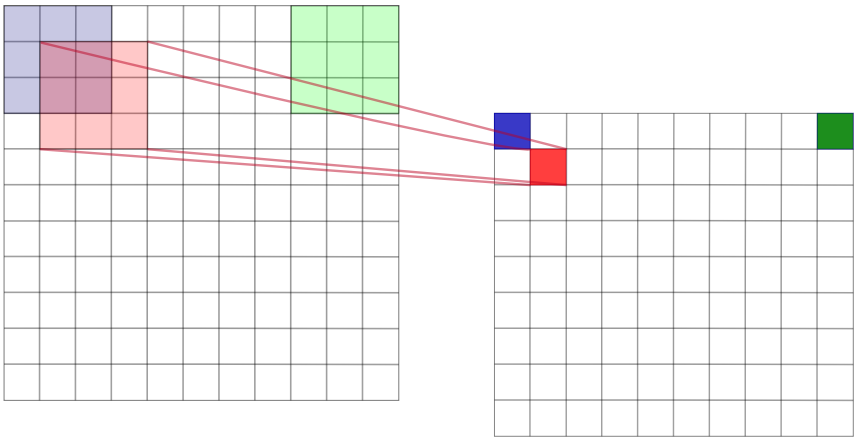


Abbildung 2.9: Die Faltung am Beispiel eines 11×11 Pixel großen Bildes und eines 3×3 Pixel großen Filters. In die Berechnung des Pixelwertes im neuen Bild der Größe 9×9 Pixel wird für einen Pixel jeweils die Pixel der unmittelbaren Umgebung in Filtergröße im Ursprungsbild mit den Filterkoeffizienten multipliziert und summiert.

Anschließend werden die Produkte sowie ein Biaswert aufaddiert, um dann als Argument für eine Aktivierungsfunktion genutzt zu werden:

$$a_{x,y} = f_{act}(b + \sum_{i=x-\frac{n}{2}}^{x+\frac{n}{2}} \sum_{j=y-\frac{n}{2}}^{y+\frac{n}{2}} w_{ij} a_{x-1,y-j})$$

Dieser Vorgang ist in Abbildung 2.9 visualisiert. Die Filterkernwerte und ihre Anordnung repräsentieren die Beschaffenheit des Reizes, also eines Musters in einem Eingabebild, auf das die Zelle besonders gut reagiert. Eine Konvolutionsschicht mit N Filtern versucht daher N verschiedene Merkmale auf der Eingabeschicht zu detektieren. Da die Merkmalsdetektoren auf alle Positionen des Bildes angewandt werden tragen sie zur Translationsinvarianz des Netzes bei.

Ein weiterer Vorteil in der Verwendung gemeinsamer Filtergewichte ist, dass erheblich weniger Gewichte durch das neuronale Netz gelernt werden müssen. In der Vergangenheit war so die Nutzung komplexerer und tieferer Netze erst möglich geworden.

2.2.4.3 Subsamplingschichten

Die Subsampling-Schichten eines Konvolutionsnetzes tragen zur Robustheit gegenüber Skalierung und Rotation bei. Zudem verringern sie die Anzahl der Gewichte, die in den folgenden Schichten gelernt werden müssen. Hier wird die Auflösung der Bildrepräsentation verkleinert. In der Regel wird ein Subsamplingfaktor $s = 2$ verwendet. Das bedeutet,

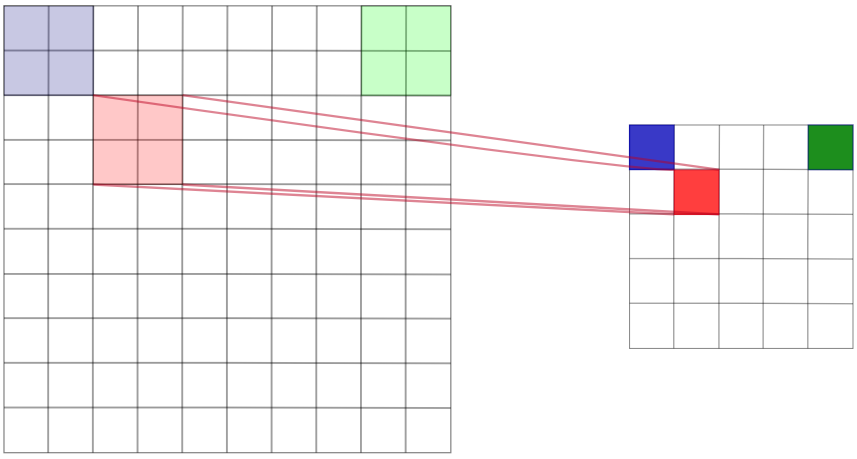


Abbildung 2.10: Das Subsampling am Beispiel eines 10×10 Pixel großen Bildes und einem Subsamplingfaktor von 2. In die Berechnung des Pixelwertes im neuen Bild der Größe 5×5 Pixel wird für einen Pixel eine Funktion auf die Pixel im rezeptiven Feld der Zelle angewandt. Dies könnte beispielsweise das gewichtete Mittel oder die Maximum-Funktion sein.

dass das Bild in jeder Dimension um den Faktor 2 verkleinert wird. Jeder Ausgabewert einer Subsampling-Schicht berechnet sich aus einer Funktion, die auf einem nicht überlappenden Feld der Größe $s \times s$ auf der Eingabekarte der Subsamplingschicht, also einer Konvolutionsschicht, angewendet wird. Häufig wird der Durchschnitt der Aktivitäten berechnet, die in dem Eingabefeld der Größe $s \times s$ liegen (vergleiche Abbildung 2.10. In einigen Arbeiten wird in Anlehnung an die Konvolutionsschichten noch ein lernbarer Parameter β eingeführt, der mit dem Durchschnitt multipliziert wird. Zusätzlich wird noch ein Bias addiert. Ebenfalls in Anlehnung an die Konvolutionsschichten wird dieses Ergebnis als Argument einer Transferfunktion benutzt. Die Aktivierung a_j eines solchen Neurons berechnet sich dann durch

$$a_j = \tanh(\beta \sum_{s \times s} a_i^{s \times s} + b)$$

Detektierte Merkmale einer unteren Schicht werden so über ihre überproportionale Aktivierung auch an die nächste Schicht weiter gegeben.

2.2.5 Gradientenabstiegsverfahren zum Training neuronaler Netze

Sowohl Konvolutionsnetze als auch Multilayer-Perzeptrons werden häufig mit Gradientenabstiegsverfahren trainiert. Dabei handelt es sich um eine Klasse von Lernverfahren, bei denen versucht wird, eine festgelegte Fehlerfunktion zu minimieren. Dem Namen

entsprechend wird dies versucht, in dem man den Gradienten der Fehlerfunktion berechnet und entgegen der Richtung seines steilsten Anstieges absteigt:

$$\Delta W = -\eta \nabla E(W) \tag{2.5}$$

Der Gewichtsvektor W enthält dabei alle Gewichte, die von allen n Neuronen der Quellschicht zu allen m Neuronen der Zielschicht gehen:

$$W = \begin{pmatrix} w_{11} \\ w_{12} \\ \dots \\ w_{nm} \end{pmatrix} \tag{2.6}$$

Als freie Parameter der Fehlerfunktion dienen hier nur die Gewichte. Es muß daher versucht werden, duch eine Veränderung des Gewichtsvektors[1] ΔW die Fehlerfunktion E zu minimieren. Dazu wählt man den Betrag der partiellen Ableitungen $\frac{\partial E}{\partial w}, w \in \{w_{11}, \dots, w_{nm}\}$ und multipliziert diese Änderung mit einer Schrittweite η, die als *Lernrate* bezeichnet wird.

Für ein einzelnes Gewicht w_{ij} gilt dabei:

$$\Delta w_{ij} = -\eta \frac{\partial}{\partial w_{ij}} E(W) \tag{2.7}$$

Als Fehlerfunktion wird häufig der *quadratische Fehler* über alle Ausgabeneurone $L_{Ausgabe}$ verwendet:

$$E_m = \frac{1}{2} \sum_{j \in L_{Ausgabe}} (t_j - o_j)^2 \tag{2.8}$$

verwendet. Der Fehler E_m eines Musters m setzt sich zusammen aus der Summe der Differenzen zwischen der gewünschten Ausgabe der Neurone (t_j) mit den tatsächlichen Aktivierungen der Neurone (o_j). Soll mehr als ein einzelnes Muster gelernt werden, so setzt sich der Gesamtfehler zusammen aus

$$E = \sum_{m \in Muster} E_m$$

zusammen.

[1] Der Nablaoperator ∇ ist ein Vektor, dessen Komponenten die partiellen Ableitungsoperatoren sind. Durch ∇E erhält man daher einen Vektor, in dem E nach allen freien Parametern w partiell abgeleitet wurde.

2.2.5.1 Backpropagation of Error

Ein häufig verwendetes Gradientenverfahren wurde 1974 von Paul Werbos [Wer74] vorgestellt. Das Verfahren ist unter dem Namen *Backpropagation of Error* bekannt und beschreibt, wie man ein grundlegendes Problem beim Training mehrstufiger Netze lösen kann. Soll ein neuronales Netz beispielsweise eine Klassifizierungsaufgabe lösen, so kann man die Ausgabe so kodieren, dass die Aktivierung eines Neurons i der Ausgabeschicht angibt, dass die Eingabe der Klasse i angehört. Diese Kodierung wird als 1-aus-N-Kodierung bezeichnet. Sind die Eingabe, die gewünschte Ausgabe und die tatsächliche Ausgabe für die Ausgabeschicht bekannt, so lässt sich duch die Differenz zwischen der gewünschten Ausgabe und der tatsächlichen Ausgabe ein Fehler bestimmen und zum Training nutzen.

Die beim Perzeptron[2] häufig genutzte Delta-Regel beruht auf diesem Zusammenhang:

$$\Delta w_{ij} = -\eta o_i \delta_j = -\eta o_i (t_j - o_j) \tag{2.9}$$

Die Änderung des Gewichts der Verbindung von Neuron i in Schicht L zu Neuron j in der Schicht $L+1$, die zugleich Ausgabeschicht ist, ist also proportional zum Ausgabefehler und der Aktivierung des Eingabe-Neurons i. Diese Regel ist auch biologisch motivierbar, wie in Abschnitt 2.2.1 die Hebb'sche Regel andeutet. Wie dort wird auch in der Delta-Regel die Verbindung – also das Gewicht – zwischen zwei Neuronen stark verändert, wenn sie gleichzeitig aktiv waren und deren Ausgabe stark vom gewünschten Ergebnis abweicht.

Für Neurone der verdeckten Schichten ist allerdings nicht offensichtlich, welchen Zustand sie annehmen sollen. Daher ist es erforderlich, die Fehlergradienten der Ausgabeschicht in geeigneter Weise durch das Netz zurück zu propagieren. Zu diesem Zweck wurde durch Umformung des Gradienten $\partial E / \partial w_{ij}$ eine verallgemeinerte Form der Delta-Regel entwickelt:

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

wobei

$$\delta_j = \begin{cases} f'_{act}(net_j) \cdot (t_j - o_j), & \text{falls } j \text{ Output-Neuron} \\ f'_{act}(net_j) \cdot \sum_{k \in L+2} w_{jk} \delta_k, & \text{sonst} \end{cases} \tag{2.11}$$

Die Fehler der Ausgabeschicht werden in Richtung der Eingabeschicht zurück propagiert. Es wird allerdings nicht der gesamte Fehler an jedes Neuron weiter gegeben, sondern nur ein Teil des Fehlers der Folgeschicht, der proportional zum Beitrag der Ausgabe des inneren Neurons ist.

[2] Dabei wird beim Perzeptron eine lineare Transferfunktion $f_{act}(x) = x$ angenommen, deren Ableitung $\frac{\partial f}{\partial x} = 1$ ist.

Beim Backpropagation of Error-Verfahren werden die Muster dem Netz präsentiert, also in das Netz eingegeben und vorwärts propagiert, und anschließend die Fehler berechnet. Je nachdem, ob die Berechnung der Gewichtsänderung direkt oder nach der Präsentation aller Muster – einer *Epoche* – geschieht, spricht man von einem *online*- oder einem *offline*-Training. In der Regel konvergiert das Online-Verfahren trotz wenig stabiler Gradienten schneller als das Offline-Training. Allerdings erfordern einige Erweiterungen stabile Gradienten, so dass hier zumindest mehrere Gradientenberechnungen zusammengefasst werden müssen. Ein Beispiel für ein solches Verfahren ist das weiter unten beschriebene Resilient-Propagation-Verfahren.

Das beschriebene Verfahren hat einige Probleme auf ungünstigen Fehleroberflächen. Diese werden in Abbildung 2.11 illustriert und im Folgenden kurz erläutert:

Überwindung flacher Plateaus der Fehleroberfläche Da bei Gradientenverfahren die Größe der Gewichtsänderung von dem Betrag des Gradienten abhängig ist, führen flache Plateaus zur Stagnation des Lernvorganges. Als schwierig gilt es in der Praxis zu erkennen, ob man sich in einem Minimum oder einem flachen Plateau befindet.

Für dieses Problem gibt es allerdings einfache Modifikationen des Gradientenverfahrens, wie das Momentum-Term-Verfahren, die helfen können diese flachen Plateaus zu überwinden (siehe [RHW86]).

Nebenminima halten das Verfahren fest Ein Problem neuronaler Netze ist, dass die Fehleroberfläche mit wachsender Größe des Netzes – also mit mehr Verbindungen – mehr Nebenminima aufweist [HN90]. In diesen Nebenminima können Gradientenabstiegsverfahren hängen bleiben, da der Gradient hier immer in Richtung des Nebenminimums zeigt und dort gegen Null geht. Neben dem Versuch, mit kleinen Lernraten an solchen Nebenminima vorbei zu kommen, gibt es keine allgemein gültige Lösung für dieses Problem.

Hin- und Herspringen in steilen Klüften An Stellen in der Fehleroberfläche, an denen sich steile Schluchten befinden, kann der Gradient so groß werden, dass das Minimum übersprungen wird. Da auf der anderen Seite des Minimums der Gradient wieder in die andere Richtung zeigt, spring das Verfahren wieder in die Region, aus der es gekommen ist. Auch hier kann das oben erwähnte Momentum-Verfahren eingesetzt werden, um das Problem zu umgehen.

Verlassen oder Überspringen guter Minima Steile Klüfte in der Fehleroberfläche führen noch zu einem weiteren Problem. Sind die Klüfte zu dem eng, kann es vorkommen, dass der Betrag des Gradienten so groß ist, dass das Verfahren

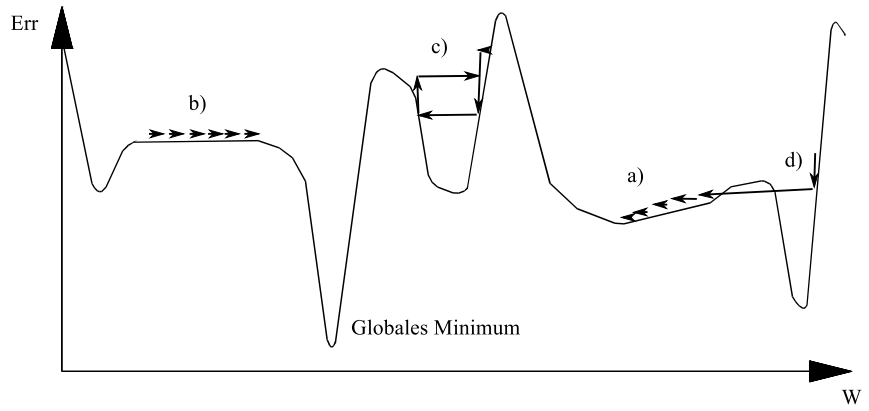


Abbildung 2.11: Probleme, die bei Gradientenabstiegsverfahren auftreten können: a) Es werden schlechte Nebenminima gefunden, b) Das Überwinden flacher Plateaus dauert lange, c) Oszillationen in steilen Schluchten, d) gute Minima werden übersprungen; Quelle: [Kri07], bearbeitet

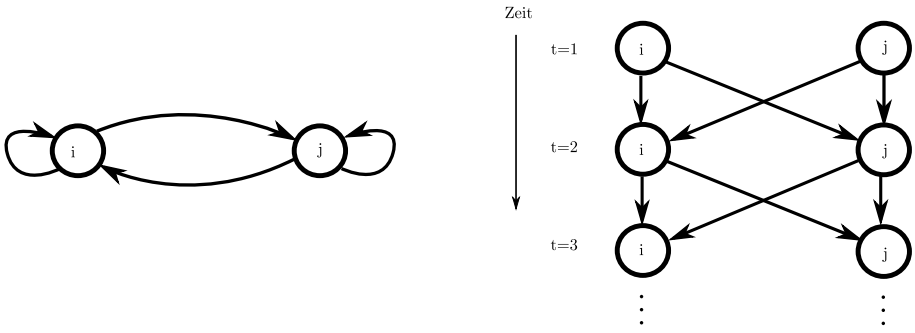
das Minimum übergeht. Unglücklicherweise begünstigt die Verwendung des Momentum-Term-Verfahrens dieses Problem.

Schlechte Wahl der Lernrate Die Wahl der Lernrate beeinflusst maßgeblich das Verhalten des Algorithmus. Wählt man eine zu kleine Lernrate, erhöht sich der Zeitaufwand für den Lernvorgang. Wird eine zu große Lernrate gewählt, so springt dieses Verfahren auf der Fehleroberfläche hin und her. Grundsätzlich hängt auch hier eine gute Wahl von der Problemstellung und der Topologie der Fehleroberfläche ab.

Es erscheint allerdings sinnvoll, die Lernrate im Laufe des Verfahrens anzupassen. Zu Beginn kann nicht davon ausgegangen werden, dass sich der (zufällig initialisierte) Gewichtsvektor in der Nähe eines Minimums befindet. Zudem müssen unter Umständen auch größere Distanzen zurückgelegt werden. Im weiteren Verlauf nähert sich das Verfahren dann einem Minimum, von dem es möglichst nicht wieder weit weg springen soll.

2.2.5.2 Training rekurrenter Netze

Das Training rekurrenter Netze gestaltet sich schwieriger als das vorwärtsgerichteter Netze. Die Propagierung des Fehlers muss hier in alle Verbindungsrichtungen erfolgen. Eines der meistverwendeten Verfahren ist das von Werbos [Wer90] vorgestellte *Backpropagation through time* (BPTT). Die Idee dabei ist, dass das Netz über die Zeit, also über n diskrete Zeitschritte, entfaltet wird. Für jeden Zeitschritt wird eine Kopie des Netzes angelegt und die rekurrenten Verbindungen in dem Netz mit den passenden Zeitindezes aufgelöst (siehe Abbildung 2.12). So wird aus einem rekurrenten Netz ein vorwärts gerichtetes Netz. Dieses



a) b)

Abbildung 2.12: Die Idee von Backpropagation trough time: a) Ein einfaches neuronales Netz, bestehend aus zwei Knoten i und j; b) Das Netz aus a) wurde zeitlich entfaltet und jeweils pro Zeitschritt eine Kopie der Neurone i und j angelegt. Da im Zeitschritt t nur Verbindungen in den Zeitschritt $t+1$ zugelassen sind, entsteht ein vorwärtsgerichtetes Netz.

lässt sich trainieren, in dem man einen Vorwärtpass und die Rückwärtspropagierung der Fehlersignale durchführt. Allerdings müssen die Gewichtswerte über die Zeit konsistent gehalten werden. So muss sichergestellt werden, dass $w_{ij}(t) = w_{ij}(t+1) = \ldots = w_{ij}(n)$.

Soll das rekurrente Netz über die Zeit Lehreingaben erhalten, so muss beachtet werden, dass sich der Fehler δ_j^t nun aus einer Lehrereingabe e_j^t und einem Fehler δ_l^{t+1} von oben zusammensetzt:

$$\delta_j^t = \frac{\partial f_{act}}{\partial o_j^t}(e_j^t + \sum_l w_{kl}\delta_l^{t+1}) \tag{2.12}$$

2.2.6 Modifikationen von Backpropagation of Error

Im Folgenden werden kurz die in dieser Arbeit genutzten Erweiterungen vorgestellt.

2.2.6.1 Resilient Propagation

Resilient Propagation – oder Rprop – ist ein von Martin Riedmiller und Heinrich Braun ([RB93]) entwickeltes Offline-Lernverfahren, bei dem die problematische Wahl der Lernrate entfällt. Dafür wird für jedes Gewicht eine eigene Lernrate eingeführt, die abhängig von der Entwicklung der Gradienten geändert wird. Allerdings basieren die Änderungen nicht – wie bisher – auf dem Betrag des Fehlergradienten, sondern nur auf dessen

Vorzeichen. In Abhängigkeit der Vorzeichen der Gradienten im letzten und im aktuellen Zeitschritt wird die Lernrate eines Gewichts Δ_{ij} erhöht oder erniedrigt:

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

Dabei ist $0 < \eta^- < 1 < \eta^+$ und S steht hier für den Gradienten selbst: $S(t) = \partial E / \partial w_{ij}$.

Wiederverwendet wird die bereits bekannte Gleichtung für das Gewichtsupdate:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \tag{2.14}$$

Die Gewichtsänderungen $\Delta w_{ij}(t)$ hängen aber nun ebenfalls von den Vorzeichen der Steigungen der letzten beiden Zeitschritte ab:

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t), & \text{falls } S(t-1)S(t) > 0 \wedge S(t) > 0 \\ \Delta_{ij}(t), & \text{falls } S(t-1)S(t) > 0 \wedge S(t) < 0 \\ -\Delta w_{ij}(t-1), & \text{falls } S(t-1)S(t) < 0 \\ -sgn(S(t)) \cdot \Delta_{ij}(t), & \text{sonst} \end{cases} \tag{2.15}$$

Sind die Vorzeichen gleich, so wird die Gewichtsänderung entweder addiert oder subtrahiert – je nach Vorzeichen der Steigung. Sind die Vorzeichen der Steigungen unterschiedlich so war die Gewichtsänderung zu groß und ein Minimum wurde durch die Änderung des Gewichtes übersprungen. In diesem Fall muss die letzte Gewichtsänderung zurück genommen und die nächste reduziert werden. Gleichzeitig muss im dritten Fall die Steigung $S(t) = 0$ gesetzt werden, so dass sich im folgenden Schritt das Vorzeichen nicht wieder ändert.

Als Richtwerte für die Wahl der konstanten Parameter haben sich in Experimenten $\eta^+ = 1{,}2$ und $\eta^- = 0{,}5$ als für viele Fälle gut geeignete Werte herausgestellt. Die Lernraten sollten beschränkt werden, wofür Riedmiller selbst eine obere Schranke von $\Delta_{max} = 50$ und eine untere Schranke von $\Delta_{min} = 10^{-6}$ vorschlägt [RB94]. Zu den Lernraten wird vorgeschlagen, diese mit einem kleinen positiven Wert, z.B. $\forall i, \forall j : \Delta_{ij}(0) = 0{,}1$ zu initialisieren. Die Wahl der Lernraten-Initialisierung gilt allerdings als unkritisch, da diese rasch adaptiert werden.

Im Vergleich zu Backpropagation, SuperSAB [Jac88] und Quickprop [Fah88] waren die Trainingszeiten von Rprop beim Training eines 10-5-10-Encoders und eines 12-2-12-

Encoders deutlich kürzer. Zudem liegt der Vorteil des Verfahrens in einer relativ einfachen Lernregel, die sich leicht implementieren läßt.

In [Rie93] weist Riedmiller allerdings darauf hin, dass Netze, die mit RProp trainiert wurden, nicht die gleiche Generalisierung leisten, wie solche, die mit Backpropagation trainiert werden, da sie schneller übertrainiert werden können. Daher wurde vorgeschlagen, das Verfahren um eine *weight decay*-Komponente zu erweitern.

2.2.6.2 Weight Decay

Das unter dem Namen *Gewichtsabnahme* (weight decay) bekannt gewordene Verfahren geht auch auf Paul Werbos [Wer88] zurück. Ausgangspunkt der Überlegungen zu diesem Verfahren ist, dass große Gewichte neurobiologisch unwahrscheinlich erschienen. Daher handelt es sich bei dem Gewichtsabnahme-Verfahren um eine Modifikation des normalen Backpropagation-Lernverfahrens, bei dem große Gewichte durch einen zusätzlichen Term in der Fehlerfunktionbestraft werden:

$$E^{neu} = E^{alt} + \frac{\lambda}{2} \sum_{i,j} (w_{ij})^2 \tag{2.16}$$

Nach der Bildung eines Gradienten für das Gewicht w_{ij} ergibt sich eine Gewichtsänderungsregel, die zugleich den Betrag der Gewichte minimiert:

$$\Delta w_{ij}(t+1) = -\eta o_i(t) \delta_j(t) - \lambda w_{ij}(t) \tag{2.17}$$

Während es wünschenswert ist, kleine Gewichte zu bekommen, die eine bessere Generalisierungsleistung ermöglichen, können zu große Werte des Parameters λ leicht das Lernen behindern. In [KH92] empfehlen Krogh und Hertz nach Experimenten für den Decay-Parameter λ Werte zwischen $\lambda_{min} = 0{,}00005$ und $\lambda_{max} = 0{,}0001$.

2.3 Die Computing Unified Device Architecture CUDA

2.3.1 CUDA

Um ein rekurrentes neuronales Netz mit Bildern in der Größenordnung von 256×256 Pixeln und größer trainieren zu können, bietet sich der Einsatz paralleler Hardware wie beispielsweise einer Grafikkarte an. Genau wie bei den Berechnungen einzelner Bildpunkte eines darzustellenden Bildes – dem *Rendern* – ist auch die Berechnung der Aktivierungen

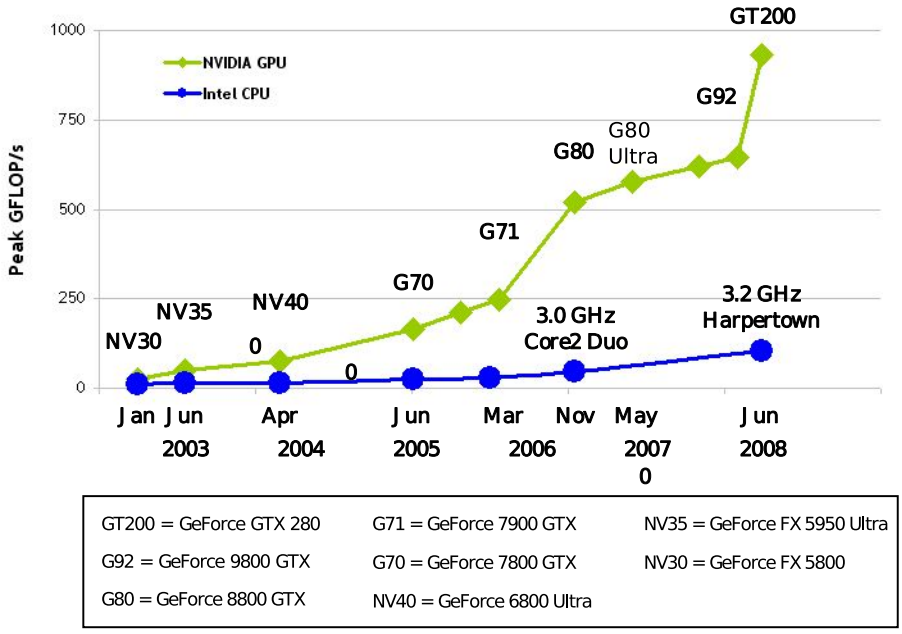


Abbildung 2.13: Die Entwicklung der Rechenleistung in GFLOPS von GPUs der Firma NVIDIA im Vergleich mit CPUs der Firma Intel (nach [Cor10])

der Neuronen parallel durchführbar. Im Vergleich zur Berechnung auf der CPU lassen sich so enorme Geschwindigkeitsvorteile erzielen. Abbildung 2.13 verdeutlicht diesen Zusammenhang anhand der theoretisch zur Verfügung stehenden Rechenleistung jeweils für GPUs der Firma NVIDIA und CPUs der Firma Intel. Während eine CPU immer nur eine Instruktion nach der anderen ausführt, führen GPUs die selbe Instruktion auf verschiedenen Daten parallel aus. Daher spricht mann auch von *Datenparallelität*. Die GPU ist daher gut dafür geeignet, Algorithmen mit hoher algorithmischer Dichte, also dem Verhältnis von Berechnungsoperationen zu Ladeoperationen auf Daten, zu implementieren.

Die Verwendung der Grafikkarte für allgemeine Berechnungen wurde durch die Veröffentlichung des CUDA (Compute Unified Device Architecture) Frameworks in 2006 erheblich gefördert. Zuvor konnte man sich mit der Programmierung von Shadern behelfen, die allerdings ursprünglich lediglich für die Berechnung der Farbwerte einzelner Pixel entwickelt wurden. Zwar stand auch für die Shaderprogrammierung eine C-ähnliche Sprache zur Verfügung. Allerdings waren die Shader nicht für allgemeine Berechnungen gedacht und die Nutzung und insbesondere die Fehlersuche waren sehr umständlich. Zudem war es nicht möglich Daten zwischen einzelnen Shader-Durchläufen auszutauschen.

Die CUDA-Schnittstelle wurde dagegen speziell dafür entwickelt parallelisierbare Teile der Software auf der Grafikkarte zu berechnen. Dazu erweitert der CUDA-Standard die Programmiersprache C/C++ um spezielle Funktionen und Schlüsselwörter, die markieren, welche Teile des Programms auf der Grafikkarte berechnet werden sollen. Diese werden

	Grafikkartengeneration		
Eigenschaft	8800 Ultra	9800 GTX+	GTX 285
Veröffentlichung	05/2007	07/2008	01/2009
Compute Capability	1.0	1.1	1.3
Multiprozessoren	16	16	30
Prozessoren (gesamt)	128	128	240
Globaler Speicher	768 MB	512 MB	1024 MB
Konstantenspeicher	64 KB	64 KB	64 KB
Shared Memory/ Block	16 KB	16 KB	16 KB
Max. Speicherbandbreite	103,7 GB/s	70,4 GB/s	159,0 GB/s
Max. Geschwindigkeit	576 GFLOPs	705 GFLOPs	1063 GFLOPs
Register per block	8192	8192	16384

Tabelle 2.1: Vergleich der Eigenschaften ausgewählter CUDA-Grafikkarten (nach [SMB10a])

von dem mitgelieferten Compiler kompiliert und bei Programmstart von der CUDA-Laufzeitbibliothek und dem Grafikkartentreiber auf die Grafikkarte kopiert und dort ausgeführt. Zusätzlich stellt die CUDA-API verschiedene Funktionen zur Verfügung um zur Laufzeit Daten auf die Grafikkarte oder zurück zu kopieren.

In naher Zukunft soll mit dem OpenCL-Standard eine herstellerübergreifende Programmierplattform für CPUs und GPUs biebten [Gro10].

2.3.1.1 Eigenschaften von CUDA-Hardware

In diesem Abschnitt folgt eine kurze Beschreibung der Eigenschaften der CUDA-fähigen Hardware mit der Rechenfähigkeit (*compute capability*) 1.3, die in dieser Arbeit genutzt wurde (nach [Cor10]). Auf dieser Hardware wird das im folgenden Abschnitt erläuterte Programmiermodell ausgeführt.

Das Herzstück jeder CUDA-fähigen Hardware ist ein skalierbares Feld von mehrprozessfähigen Streaming Multiprocessors (SMs). Jeder dieser Multiprozessoren (vgl. Abbildung 2.14 und Tabelle 2.1) besteht aus

- acht skalaren Prozessoren (SP),
- zwei Modulen zur Berechnung transzendenter Funktionen,
- einer Befehlseinheit, die mehrere Prozesse steuern kann, und
- lokalem Speicher der folgenden Arten:
  - einem Satz 32-bit Register pro skalarem Prozessor

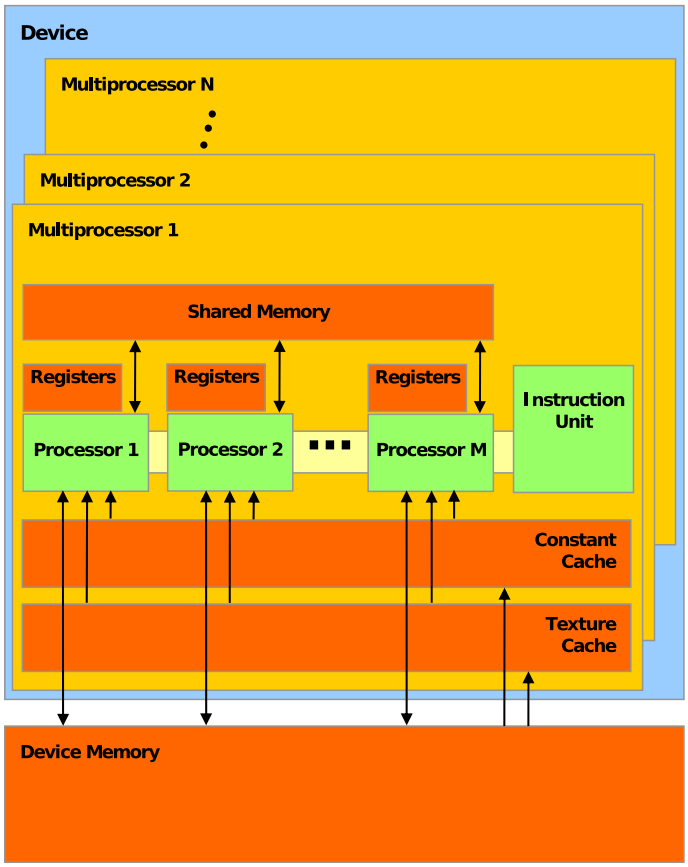


Abbildung 2.14: Das CUDA-Hardware-Modell (nach [Cor10])

- einem schnellen gemeinsamen lokalen Speicher (shared memory) auf den alle skalaren Prozessoren eines MS zugreifen können
- einen Cache für Konstanten, der nur ausgelesen werden kann und
- einem Textur-Cache, der Daten aus den globalen Texturspeicher zwischenspeichert und so das Laden bereits gelesener Texturen deutlich beschleunigt.

Der Multiprozessor übernimmt die Verwaltung seiner acht Prozessoren und betreibt auf ihnen Prozesse (*Threads*). Dadurch, dass diese Verwaltung hardwareseitig implementiert ist, geht keine Rechenzeit in den arithmetischen Rechenwerken verloren. Jede Grafikkarte hat zudem noch einen großen globalen Speicher, auf den alle Multiprozessoren und die CPU-Seite zugreifen können. Daten, die auf der GPU verarbeitet werden sollen werden in der Regel erst einmal dorthin kopiert. Einem Thread stehen zusätzlich unterschiedliche, hierarchisch angeordnete Speicherbereiche zur Verfügung. Deren Einsatz spielt bei der Optimierung der Performance eine wichtige Rolle, da die Zugriffszeiten stark differieren.

Jeder Prozess, den ein Multiprozessor übergeben bekommt, wird auf auf einen Skalaren Prozessor abgebildet. Dieser stellt ihm mehrere Register mit geringer Latenz zur Verfügung, die allerdings für andere Prozesse nicht sichtbar sind. Über ein Sprachkonstrukt

im Programmiermodell kann bei einem Kernelaufruf die Restriktion angegeben werden, dass eine Gruppe von Prozessen nur im selben Multiprozessor verarbeitet werden darf. Diese Gruppe heißt ein *Block*. Alle Prozesse eines Blockes können auf den gleichen, gemeinsamen lokalen Speicher (den shared memory) zugreifen. Über diesen lokalen Speicher können die Prozesse miteinander gemeinsame Daten verarbeiten. Zudem ist dessen Zugriffsgeschwindigkeit wesentlich schneller, als die des globalen Speichers. Der Inhalt des lokalen Speichers ist nach Ausführung des Blocks jedoch nicht mehr verfügbar. Daneben gibt es noch den Texturspeicher und den Konstantenspeicher, die nach Starten einer Kernelfunktion nur noch lesbar sind. Damit Prozesse auf den gemeinsamen lokalen Speicher zugreifen können, muss sicher gestellt sein, dass die Prozesse auf dem selben Multiprozessor abgearbeitet werden.

Werden der GPU mehrere Blöcke übergeben, so geschieht dass in Form eines *Grids*. Dazu werden die Blöcke der Reihe nach auf die Multiprozessoren verteilt, wie in Abbildung 2.15 dargestellt. Ist ein Block mit all seinen Prozessen abgearbeitet wird der nächste Block geladen. Dieser Mechanismus ermöglicht es dem Treiber die Ausführung der Blöcke an die Ressourcen der Grafikkarte anzupassen. Je mehr Prozessorkerne zur Verfügung stehen, desto mehr Blöcke können parallel ausgeführt werden. Das CUDA-Programm passt sich somit automatisch an andere – insbesondere auch aktuellere, mit mehr Kernen ausgestattete – Grafikkarten angepasst werden und skaliert die Leistung automatisch.

Die Abarbeitung eines Prozesses geschieht intern weitgehend unabhängig von den anderen Prozessen des gleichen Multiprozessors. NVIDIA führt dazu eine neue Architektur ein, die sie *SIMT* (Single Instruction, Multiple Thread) nennen. Sie lehnt sich an die bekannte *SIMD*-Architektur – *Single Instruction, Multiple Data* – an, bei der ein Befehl auf mehrere Datenelemente angewandt wird. Davon abweichend wird allerdings durch Verzweigungen zugelassen, dass nicht für jedes Datenelement jede Anweisung ausgeführt wird.

Der Multiprozessor bekommt eine Gruppe von 32 Prozessen – sogenanntes *Warp* übergeben. Dieser Warp besteht aus ein gemeinsames Programm, welches als *Kernel* bezeichnet wird, und für das die Prozesse ausgeführt werden. Alle Prozesse haben einen gemeinsamen Programmspeicher und einen jeweils eigenen Befehlszeiger. Das erlaubt die parallele Abarbeitung des gleichen Programmes, während unterschiedliche Sprünge und Verzweigungen innerhalb des Programms erlaubt sind. Bei der Abarbeitung von datenbedingten Verzweigungen, werden alle Zweige seriell abgearbeitet. Prozesse, die in einem abgearbeiteten Zweig nicht aktiv sind, werden schlafen gelegt. Ist die Verzweigung abgeschlossen, stellt der Multiprozessor wieder einen einheitlichen Befehlszeigerstand her. Daher wirken sich große und tiefe Verzweigungen negativ auf die Laufzeit aus.

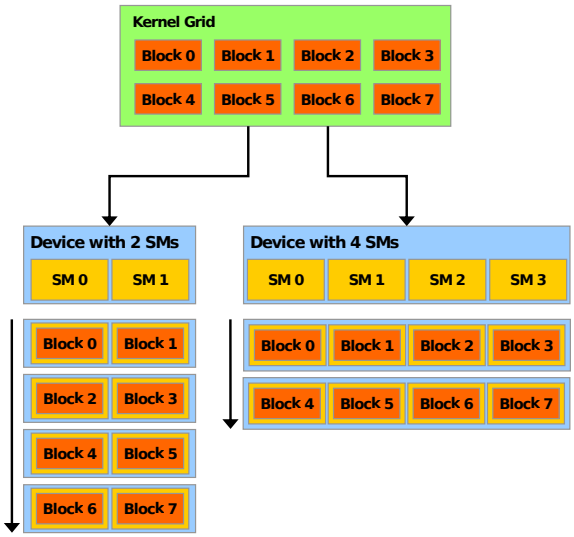


Abbildung 2.15: Ausführung mehrerer Threads bei unterschiedlicher Anzahl von Multiprozessoren (nach [Cor10])

Der Vorteil dieser Architektur ist dennoch, dass es möglich ist für einzelne Datenelemente auf Prozessebene zu spezifizieren, welche Anweisungen ausgeführt werden. So kann der Programmierer einerseits Code für die parallele Verarbeitung einzelner unabhängiger Elemente schreiben. Andererseits ermöglicht das Modell die Entwicklung koordiniert arbeitender Prozesse auf zusammenhängenden Daten.

Wie eingangs erwähnt, bezieht sich diese Beschreibung auf Grafikkarten der 1.3-Generation. Das neuere Modell mit der Rechenfähigkeit 2.0 läuft auf GPUs der Fermi-Generation, die allerdings zu Beginn der Arbeit noch nicht verfügbar waren. In der Fermi-Generation wurde das Hardware-Modell modifiziert. Insbesondere wurde die Anzahl der Prozessoren von 8 auf 32 pro Multiprozessor erhöht, die durch zwei Warp-Scheduler verwaltet werden. Zusätzlich haben sich Änderungen am Speicherzugriff ergeben. Nach einigen Experimenten konnten allerdings nicht alle Funktionen mit dem Umstieg auf eine Fermi-Karte beschleunigt werden. Daher wurde auf eine Verwendung dieser Karte in dieser Arbeit verzichtet.

2.3.1.2 Das CUDA-Programmiermodell

Es gibt zwei Programmier-Schnittstellen für CUDA-fähige Hardware. Zum einen die Treiber-Schnittstelle (*Driver API*) und die hochsprachige Schnittstelle *CUDA for C*. An dieser Stelle wird das Modell *CUDA for C* erläutert, welches den Standart C um einige Sprach-Konstrukte erweitert. Diese Schnittstelle wird von der in dieser Arbeit genutzten CUV-Bibliothek genutzt. Bestandteile der Bibliothek, die im Rahmen dieser Arbeit zur

CUV-Bibliothek hinzugefügt wurden, sind ebenfalls mit dieser Schnittstelle entwickelt worden. Die Schnittstelle CUDA for C erweitert die Sprache C/C++ um Schlüsselwörter mit denen die Konzepte der CUDA Hardware abgebildet werden.

Das grundlegende Element in der Entwicklung paralleler Datenverarbungsalgorithmen sind Kernelfunktionen. Eine Kernelfunktion ist eine spezielle C-Funktion, welche auf den skalaren Prozessoren ausgeführt wird. Der Funktionsdefinition eines Kernels wird zur Kennzeichnung das Schlüsselwort

```
__global__
```

vorangestellt.

Beim Aufruf der Kernelfunktion wird in drei spitzen Klammern eine Ausführungsumgebung (*execution environment*) festgelegt:

```
foo<<<dimGrid, dimBlock, N,C>>>(Parameterliste)
```

Darin wird spezifiziert, wie viele Prozesse ausgeführt werden. Jeder skalare Prozessor führt dann eine Instanz dieser Kernelfunktion in sogenannten Prozessen (*Threads*) aus. Neben der Anzahl der Prozesse wird auch eine Struktur festgelegt, in der die Prozesse ausgeführt werden. Dazu lassen sich Prozesse dreidimensional in einem Block anordnen: Jeder Thread bekommt zur Laufzeit eine eindeutige Identifikationsnummer, die in der dreidimensionalen Struktur `threadIdx` abgelegt ist. Diese Identifikationsnummer kann als Position innerhalb einer zwei- oder dreidimensionalen Matrix interpretiert werden. Innerhalb der Kernelfunktion kann über die Struktur `threadIdx` bestimmt werden, welcher Thread gerade ausgeführt wird. Punktweise Additionen von Vektoren oder Matrizen können nun beispielsweise parallelisiert werden, indem jeder Prozess ein Element des Vektors oder der Matrix berechnet. Die Prozesse können also in ein-, zwei- oder dreidimensionale Anordnungen gebracht werden, was eine intuitive Abbildung auf bis zu dreidimensionale diskrete Strukturen ermöglicht.

Alle Prozesse eines Blockes werden auf dem gleichen Multiprozessor ausgeführt und können daher auf einen gemeinsamen Speicherbereich (*shared memory*) zugreifen. Da der Prozessorspeicher begrenzt ist, können jedoch nicht beliebig viele Prozesse in einem Block ausgeführt werden. Jeder Block kann daher auch nur bis zu 512 Prozesse enthalten. Wenn mehr Prozesse benötigt werden, müssen weitere Blöcke erstellt werden. Diese können in einem bis zu zweidimensionalen Gitter angeordnet werden. Die Anzahl der Blöcke in einem Gitter wiederum kann in jeder Dimension auf bis zu 65536 steigen.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
```

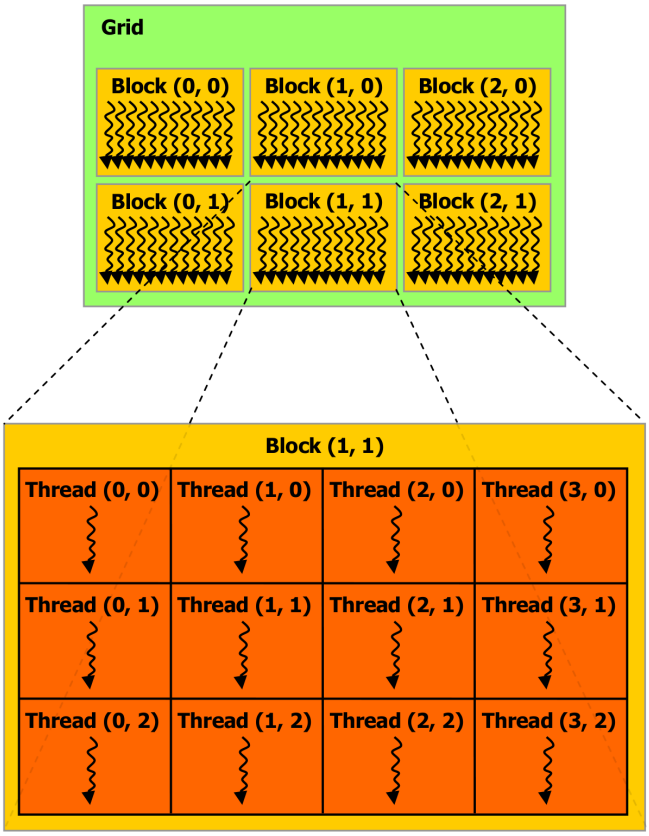


Abbildung 2.16: Ein Grid besteht aus einer zweidimensionalen Anordnung von Blöcken; ein Block besteht aus einer bis zu dreidimensionalen Anordnung von Threads.

```
                        float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Innerhalb eines Prozesses deklarierte Variablen werden, genau wie die Parameter der Kernelfunktion, in Registern abgelegt. Alternativ dazu kann der Speicherbereich, in dem die Variablen angelegt werden sollen durch das Voranstellen der Schlüsselwörter `__device__` (globaler Speicher), `__constant__` (Konstanten-Speicher) oder `__shared__` (lokaler Speicher) direkt angegeben werden.

2.3.2 Optimierung von CUDA-Kernelfunktionen

Die folgenden Ansätze stehen stellvertretend für die an verschiedenen Stellen immer wieder auffindbaren Regeln, um Kernelfunktionen zu optimieren:

- Wenige Register pro Prozess – erlaubt es dem Multiprozessor mehr Prozesse zu laden
- Wenige Speicherzugriffe
- (Temporäre/lokale) Laufzeitvariable in Register speichern
- Verzichte auf lokale Felder wie int `a[3]={1,2,3}` – benutze lieber `a0=1;a1=...` etc.
- Kleine Kernel: Splitte große Kernel in kleinere auf – diese benötigen dann wahrscheinlich weniger Register)
- Speichere Daten als Textur, wann möglich: Zugriffe auf den Texturspeicher werden gecached – globale Speicherzugriffe nicht.
- In Verzweigungen sollten sich möglichst alle Prozesse gleich verhalten.
- Vermeide Schleifen, die nur eine kleine Zahl von Prozessen durchläuft während die anderen warten müssen
- Eine komplexe Rechnung ist meist schneller, als eine große Lookup-Tabelle
- Ein eigener Chaching-Mechanismus, der den shared memory nutzt, ist nicht unbedingt von Vorteil
- Zugriffe auf den globalen Speicher sollen zusammenhängend sein
- Vermeide Bank-Konflikte beim Zugriff auf den shared memory
- Kleine Lookup-Tabellen sollten im shared memory gespeichert werden.

Die aufgeführten Regeln liegen oft leider im Widerspruch zueinander. Will man beispielsweise temporäre Variablen in Registern unterbringen wird die Regel verletzt, man solle möglichst wenige Register verwenden. Oft hilft daher nur die Trial-and-Error-Methode um die beste Beschleunigung zu erzielen.

2.3.3 Die Bibiothek CUV-Bibliothek

Die CUV-Bibliothek ist ein C++-Framework für Linux-Plattformen, welches eine einfache und intuitive Nutzung der CUDA-Schnittstelle für Arbeit mit Matrizen und Verktoren ermöglichen soll. Sie wurde an der Universität Bonn entwickelt – siehe [SMB10b] und [fIV10].

Die Bibliothek stellt Klassen zur Verfügung die Matrizen und Vektoren einerseits im RAM (*Host-Matrizen*) sowie im globalen Speicher der Grafikkarte (*Device-Matrizen*) speichert. Ebenfalls bereit gestellt werden grundlegende und speziellere Matrix-Operationen, wie sie im Zusammenhang mit der Entwicklung von Neuronalen Konvolutionsnetzen verwendet werden.

Teil des Projektes war die Entwicklung einer Schnittstelle zur Skriptprache Python. Um Konvolutionen durchzuführen wurde bei der Entwicklung auf den Quellcode für 2-D CUDA-Konvolutionen von Alex Krizhevsky zurückgegriffen. Dieser Code ist besonders schnell, da er für viele verschiedenen Einzelfälle implementiert wurde.

Ein weiterer Teil der Bibliothek stellt die Funktionalität als Module für die den CUDA-Funktionen bereit.

Wie in MATLAB sind die Objekte, die in der Bibliothek verwendet werden, allesamt Matrizen oder Vektoren, die Werte eines beliebigen Typs, beispielsweise Ganzzahlen oder Fließkommazahlen, enthalten. Zudem kann bei der Instanzierung eines Objektes angegeben werden, ob das Objekt im CPU-Speicher oder im GPU-Speicher angelegt werden soll. Die Matrizen können zudem spalten- oder zeilenorientiert angelegt werden. Dabei ist zu beachten, dass nicht alle durch die Bibkiothek zur Verfügung gestellten Funktionen mit Objekten beider Speicheranordnungen funktionieren. Das gilt insbesondere für die Faltungsfunktionen, welche von Alex Krizhevsky [Kri10] übernommen wurden.

Die CUV-Bibliothek stellt mit Hilfe der Bibliothek PyUblas komfortable Funktionen bereit, um CUV-Matrizen und -Vektoren in Numpy-Objekte zu transormieren und umgekehrt. Die Numpy-Bibliothek stellt ihrerseits eine Reihe von Funktionen für wissenschaftliche Berechnungen zur Verfügung, die allerdings nicht auf der GPU ausgeführt werden. Das Zusammenspiel beider Bibliotheken ermöglicht daher die Nutzung eines breit gefächerten Feldes numerischer und algebraischer Funktionen, die zum Teil auf der Grafikkarte ausgeführt werden können.

3 Verwandte Arbeiten

Im Folgenden wird ein kurzer Überblick über Arbeiten gegeben, die den Aufgabenstellungen, die dieser Arbeit zu Grunde liegen, verwandt sind. Zunächst werden grundlegende Arbeiten zu Konvolutionsnetzen vorgestellt. Anschließend werden grundlegende Techniken der Gesichts- und Handlokalisierung beschrieben. Da die Gesichtserkennung in den vergangen zwanzig Jahren Gegenstand intensiver Forschung war, wird die Darstellung auf die grundlegenden Strömungen beschränkt und auf zusammenfassende Darstellungen verwiesen.

3.1 Konvolutionsnetze

3.1.1 Neocognitron

Als eines der ersten Konvolutionsnetze wurde zwischen 1980 und 1989 das Neocognitron von Kunihiko Fukushima entwickelt [Fuk80], [FMI88] und [Fuk89]. Seine Stärke liegt in der Erkennung visueller Muster. Erfolgreich angewandt wurde es bei der Erkennung handgeschriebener Zeichen.

Das Necognitron ist ein in hierarchischen Schichten aufgebautes Neuronales Netz. Jede Schicht besteht wiederum aus mehreren Karten. Dabei bilden zwei aufeinanderfolgende spezialisierte Schichten eine Verarbeitungsstufe:

- eine Schicht von *S-Zellen* mit trainierbaren Gewichten für ein rezeptives Feld, deren Aufgabe die Merkmalsextraktion aus der Schicht darunter ist und eine
- eine Schicht von *C-Zellen* mit festen Verbindungen, die aktiviert werden, falls eine der Zellen in ihrem rezeptiven Feld aktiviert ist, und deren Aufgabe es ist, die Erkennung von Mustern positions- und skalierungsinvarianter zu machen.

Die Gewichte der S-Zellen einer Karte sind dabei für alle Neurone gleich und positiv (exitatorisch). Die Aktivierung einer S-Zelle berechnet sich aus den gewichteten Eingaben der Vorgängerschicht und einer hemmenden Verbindung. Im Gegensatz zu den trainierbaren Gewichten, die von den C-Zellen kommen, sind die hemmenden Gewichte fest. Auf das Ergebnis wird dann eine stückweise lineare Aktivierungsfunktion f_{act} angewandt:

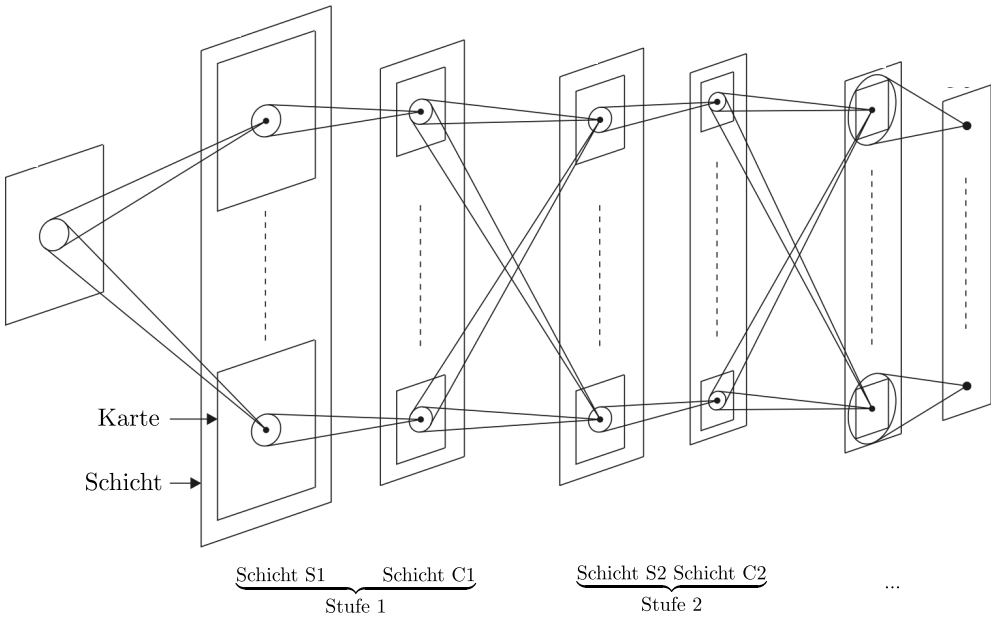


Abbildung 3.1: Die Struktur eines Neokognitrons. Nach: [Fuk89]

$$f_{act}(x) = \begin{cases} 1, & \text{falls } x \geq 0 \\ 0, & \text{sonst} \end{cases} \tag{3.1}$$

Als Lernalgorithmus für das Neocognitron wurde zunächst ein *unüberwachtes Wettbewerbslernverfahren* vorgeschlagen, bei dem die Verbindungen der Zelle einer Karte verstärkt werden, bei denen ein Stimulus die stärkste Reaktion hervoruft. Auch die hemmende Verbindung wird – wenn auch weniger stark – verstärkt. Damit ist gesichert, dass eine Karte hohe Aktivitäten nur bei dem Stimulus entwickelt, auf den sie trainiert wird.

Dieses unüberwachte Lernverfahren hatte allerdings den Nachteil, dass das Training sehr lange dauerte und die Unterscheidung ähnlicher Muster schwierig zu trainieren war. Daher wurde in [FMI88] ein *überwachtes Lernverfahren* eingeführt. Bei diesem Verfahren mussten aber für ein Eingabemuster nicht nur eine gewünschte Ausgabe, sondern auch die gewünschten Teilmuster für die Karten der Zwischenebenen bereitgestellt werden. Damit wurde das Neocognitron dann schichtweise trainiert.

Das Neocognitron modellierte als erstes Verfahren wichtige Teile der menschlichen visuellen Kognition wie die Verarbeitung durch rezeptive Felder, Muster-Sensibilität und die Verringerung der Ortsauflösung in höheren Hierarchie-Ebenen. Es wurde erfolgreich bei der Klassifizierung handgeschriebener Buchstaben eingesetzt und stellte sich als relativ skalierungs- und translationsinvariant heraus. Allerdings müssen im Entwicklungs- und Trainingsverfahren eine hohe Anzahl von Konstanten bestimmt werden, von deren richtiger Wahl die Güte des Erkennungssystems abhängt. Zusätzlich ist es sehr aufwändig

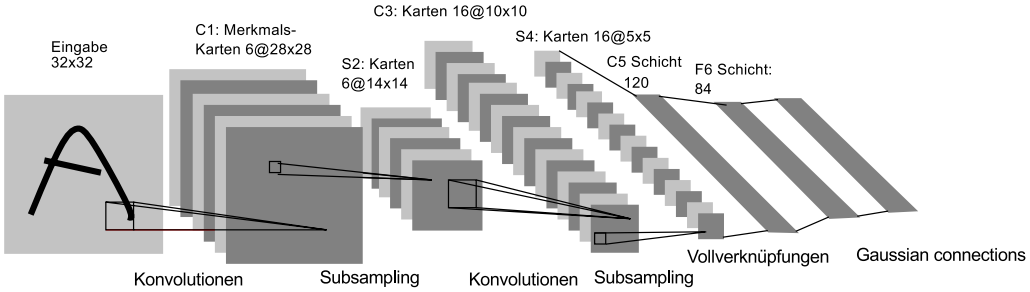


Abbildung 3.2: Die Struktur des LeNet. Nach: [LHBB99], bearbeitet

einen Trainingsdatensatz zu erstellen, da nicht nur Ein- und Ausgabe, sondern auch die Teilmuster, die erkannt werden sollen, definiert werden müssen. Dies waren wohl die Hauptgründe, die dazu führten, dass das Neocognitron sich über die Zeit nicht durchsetzen konnte.

3.1.2 LeNet

Von Yann LeCun und anderen stammt ein weiteres mehrfach überarbeitetes Konvolutionsnetz: das LeNet [LCJB+89]. In Abbildung 3.2 ist das LeNet-5 dargestellt. Ursprünglich wurde es zur Erkennung von Zeichen entwickelt, in [OCM07] wurde eine Variante des Netzes auch erfolgreich zur Detektion von Gesichtern, inklusive der Erkennung der Neigungswinkel verwendet. Das LeNet-5 verarbeitet Eingaben der Größe 32×32 und besteht aus sieben Schichten mit jeweils mehreren Karten. Die ersten vier Schichten realisieren dabei alternierend Faltungen und Subsampling. Letzteres halbiert die Auflösung in beiden Dimensionen. Abweichend zum in Abschnitt 2.2.4 beschriebenen Pooling besitzt hier auch die Subsampling-Schicht einen Bias β mit dem alle Pixelintensitäten in dem rezeptiven 2×2-Feld des Subsampling-Neurons multipliziert werden. Damit existiert hier auch in der Subsamplingschicht ein lernbarer Parameter.

Die Faltungs-Verknüpfung von Schicht zwei zu Schicht drei ist gemäß einer Verknüpfungstabelle aufgebaut. Jede Zielkarte aus Schicht drei hat Verbindungen zu drei bis sechs Quellkarten. Dieses Schema soll es ermöglichen, einfacher für jede Zielkarte unterschiedliche Merkmale zu extrahieren. Während die vorletzten beiden Schichten vollverknüpft sind, besteht die letzte Schicht aus radialen Basisfunktionen. LeCun empfiehlt diese in [LBOM98], da sie besser zu trainieren sind. Ein wesentlicher Punkt dabei ist, dass sie nur in einem räumlich begrenzten Bereich um ihren Eingaberaum eine hohe Aktivierung aufweisen. Falsche Eingaben, die weit von allen Zentren entfernt liegen, lassen sich so leichter zurückweisen.

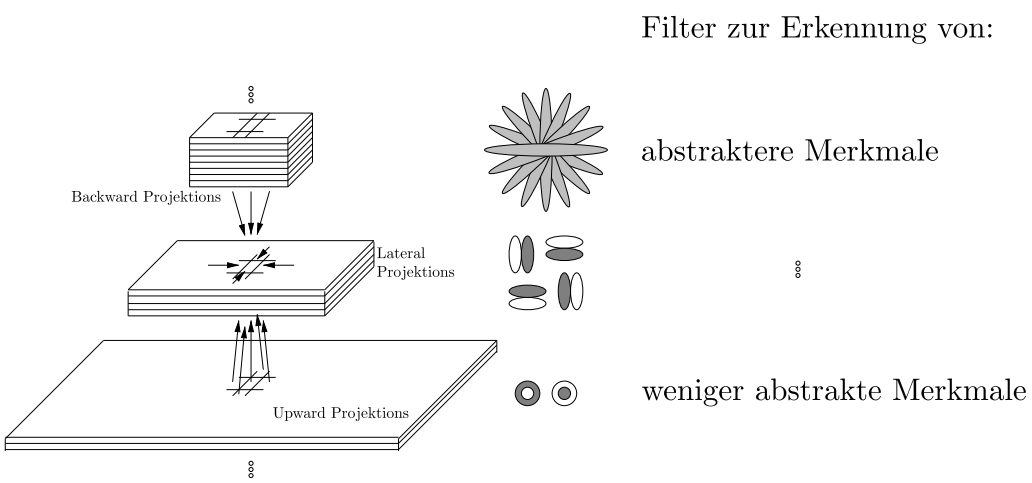


Abbildung 3.3: Struktur einer Neuronalen Abstraktionspyramide. Nach: [Beh01]

3.1.3 Die Neuronale Abstraktionspyramide

Die von Behnke in [Beh01] vorgestellte Neuronale Abstraktionspyramide ist ein hierarchisch geordnetes rekurrentes Neuronales Netz. Es besteht aus mehreren miteinander verknüpften Schichten. Jede Schicht enthält wiederum verschiedene Gitter, die diskrete Zellen enthalten.

Die Zellen bestehen jeweils aus einer Einheit zur Informationsverarbeitung. Diese Einheiten können Neuronen oder sogar kleine neuronale Netze sein. In seiner erfolgreichen Anwendung zur Erkennung von Augen verwendet Behnke einfache Neuronen. Die Gitter sollen im Folgenden Neuronenkarten, oder einfacher Karten genannt werden. Höhere Schichten der Pyramide haben weniger Neuronenzellen pro Karte, dafür aber mehr Karten. So wird die Ortsauflösung reduziert und weiter entfernte Merkmale rücken geometrisch näher zusammen. Auf der anderen Seite wird der damit einhergehende Informationsverlust zumindest teilweise durch die Erhöhung der Schichten-Anzahl ausgeglichen. Abbildung 3.3 illustriert den Aufbau der Verbindungen innerhalb der Pyramide. Die Position (i, j) in allen Karten einer Schicht l wird zu einer Hyper-Spalte zusammengefasst. Jede Zelle dieser Spalte ist in ihrer Karte lokal verknüpft. Das heißt, es bestehen nur wenige Verknüpfungen mit direkten Nachbarn. Die Nachbarschaft aller Zellen einer Hyper-Spalte werden zur Hyper-Nachbarschaft zusammengefasst. Mittels lateraler Verbindungen bekommen die Zellen der Schicht l auch Informationen aus ihrer Hypernachbarschaft übermittelt.

Auch zwischen den Schichten gibt es Verbindungen. Vorwärts gerichtete Verbindungen zur Zelle (i, j) haben ihren Ursprung in der Hyper-Nachbarschaft in der nächst niedrigeren Schicht an der Position $(2i, 2j)$. Rückwärts gerichtete Verbindungen kommen aus der Hyper-Nachbarschaft der nächst höheren Schicht an der Position $(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$.

Behnke führte verschiedene Versuche durch und konnte mit der Neuronalen Abstaktionspyramide verschiedene Aufgabenstellungen aus dem Bereich der Objekterkennung lösen. In einer Anwendung wurde erfolgreich die Detektion von Augen in einem Bild gelernt. Um Augen an einer beliebigen Position im Bild finden zu können wurden die Kartenverbindungen durch gemeinsam genutzte Gewichte einer festgelegten Nachbarschaft als Konvolutionen modelliert.

3.2 Gesichtserkennung

Die Gesichtserkennung stellt einem Algorithmus oder Detektor die Aufgabe herauszufinden, ob in einem Bild mit willkürlichem Hintergrund kein, ein oder mehrere Gesichter zu sehen sind. Diese Anforderungen sollen auch unter den erschwerten Bedingungen erfüllt werden, dass die dargestellten Gesichter verformt oder rotiert sein können. Erste Anläufe wurden bereits in den 1970er Jahren gemacht, beispielsweise von Sagai, Nagao und Kanade [SNK72]. Seitdem wurden für die Detektion und Lokalisierung von Gesichtern eine Vielzahl von Ansätzen vorgeschlagen. Einen größeren Überblick geben die Arbeiten von Hjelmås und Low [HL01], von Yang, Kriegman und Ahuja [YKA02] sowie von Zhang und Zhang [ZZ10].

Es lassen sich zwei grundlegende Klassen von Ansätzen unterscheiden, die in unterschiedlicher Weise untersucht wurden. Bei der ersten Klasse wird versucht, die Detektion von Gesichtern durch die Detektion und Komposition von Merkmalen, wie Augen, Mund oder Nase umzusetzen. Bei der anderen Klasse versucht direkt ganze Gesichter aus Bildern heraus zu erkennen.

3.2.1 Merkmalsbasierte Methoden

Die merkmalsbasierten Methoden lassen sich weiter unterteilen. Gemein ist ihnen, dass sie versuchen Regeln über die Zusammensetzung von Merkmalen zu nutzen, die Menschen als Gesicht erkennen. Erste Arbeiten, wie die bereits oben erwähnte Arbeit von Sagai et al. stellen die Erkennung von Gesichtern auf eine grammatikalische Basis: Es gilt Elemente (Merkmale) zu finden, welche in einen Zusammenhang zueinander (Grammatik) zu bringen sind. Stimmt ein Bild in der Zusammensetzung der Merkmale überein, so kann man von einem gültigen Gesicht sprechen. Welche Art von Merkmalen hierzu verwendet werden variiert zwischen einfachen Merkmalen, wie Kanten oder Helligkeitsstufen, über komplexere Merkmale, wie Augen oder Nasen, bis hin zu *active shape* Modellen, die ein

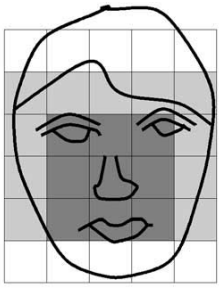


Abbildung 3.4: Beispiel einer Verteilung der Grauwerte in einem potentiellen Gesicht. Die Gitterstruktur deutet eine niedrige Auflösung an. Nach [YH94]

Abstandsmaß mitbringen, welches für gegebene Formen angibt, wie kongruent diese zu einem Merkmal sind.

Der Vorteil der in dieser Arbeit genutzten Methode gegenüber dieser Gruppe von Ansätzen ist im Wesentlichen, dass die Merkmale nicht durch den Entwickler festgelegt werden müssen. Auch das Konvolutionsnetz extrahiert Merkmale, allerdings kann es aus dem Trainingsdatensatz heraus lernen, welche Merkmale zu extrahieren sind.

3.2.1.1 Mermale der niedrigsten Stufe

Die einfachsten Merkmale, die zur Verfügung stehen sind die Pixel-Intensitäten der Bilder. Einige Ansätze, beispielsweise [KP97] oder [YH94], versuchen mit eigenen Regelsystemem direkt aus den Grauwerten Gesichtsdetektion zu betreiben.

Yang und Huang [YH94] verwenden ein hierarchisches System mit 3 Regel-Ebenen, welches auf eine Bild-Pyramide mit ebenfalls mehreren Ebenen angewandt wird. Die Bilder in den aufsteigenden Ebenen der Pyramide werden durch Mittelung und Subsampling immer grober aufgelöst. Auf der obersten Regelebene wird nach grobkörnigen Mustern in der obersten Pyramidenebene gesucht. Dort werden einfache allgemeine Regeln angewandt, wie etwa: „Im Zentrum des Gesichtes befinden sich vier Zellen mit gleicher Farbintensität“. Abbildung 3.4 verdeutlicht diese Regel. Auf dem Weg nach unten in Regelwerk und Pyramide werden solche Kandidaten durch Histogram-Angleichungen und Kanten-Erkennung gefiltert. Schließlich werden auf der untersten Ebene detaillierte Merkmale, wie Augen, Mund oder Ohren gesucht. Mit diesem Ansatz fanden sie 50 Gesichter in einem Testdatensatz mit 60 Gesichtern. Allerdings gab es bei 28 Bildern einen falschen Alarm.

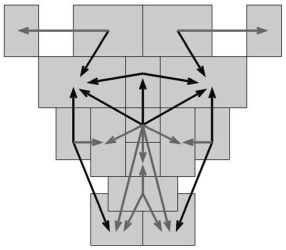


Abbildung 3.5: Beispiel einer Verteilung der Grauwerte in einem potentiellen Gesicht. Nach [YKA02]

3.2.1.2 Komplexere Merkmale

Andere Ansätze, beispielsweise [HMLYC97] und [HW99], versuchen komplexere Gesichtsmerkmale, wie Augen oder Mund, die Hautfarbe oder typische Gesichts-Texturen in den präsentierten Bildern zu finden. Die Methoden dieser Klasse gehen davon aus, dass diese Merkmale über verschiedene Sichtwinkel, Kopfposen oder Beleuchtungsbedingungen hinweg existieren. Meist werden zunächst Kanten detektiert und versucht, diese zu Merkmalen zusammen zu fassen.

Han et al. [HMLYC97] konnten in Experimenten mit 130 Gesichtern auf 122 Bildern demonstrieren, dass sie eine Trefferrate von 94 Prozent erreichen. Dazu benutzen Sie eine morphologische Technik, die aus Pixeln Segmente entwickelt, die wahrscheinlich Augen sein könnten. Dazu wird versucht, gefundene Pixelregionen zu ergänzen oder abzuschneiden, so dass die Merkmale Augen oder Augenbrauen darstellen können. In einem weiteren Schritt wird versucht die gefundenen vermeintlichen Merkmale anhand ihrer geometrischen Position als Gesichtsregion zu klassifizieren. Als Letztes wird versucht, die Kandidaten mit einem trainierten Neuronalen Netz zu verifizieren.

3.2.1.3 Schablonenbasierte Methoden

Schablonenbasierte Methoden benutzen gespeicherte Vorlagen von Gesichtern oder von Gesichtsteilen und versuchen die Korrelation zwischen gespeicherter Vorlage und Bild für die Erkennung zu nutzen. Yang, Kriegman und Ahuja berichten, dass diese Technik nur eingeschränkt nutzbar ist, da sie nicht gut mit Veränderungen in der Skalierung, der Pose oder der Form zurecht kommt. Um Skalierungs- und Forminvarianz verbessern zu können, wurde in einigen Arbeiten vorgeschlagen Modifikationen an den Schablonen zu erlauben. Sinha hat in [Sin94] und [Sin95] einen Ansatz vorgeschlagen, der mit kleinen räumlichen Verhältnis-Mustern arbeitet. Seine Idee war, dass auch wenn die Helligkeit in Fotos oder Videobildern mit Gesichtern unterschiedlich ist, die relative Helligkeit zwischen

den Gesichtsregionen ungefähr gleich bleibt. Daher definierte er für Gesichtsmerkmale wie Augen, Wangen und Stirn paarweise direktionale Helligkeitsverhältnisse. Diese Verhältnisse wurden dann zu einer Schablone zusammengesetzt, so dass ein Gesicht dann erkannt wird, wenn alle Verhältnisse räumlich verteilt stimmen.

Yuille et al. haben in [YHC92] *verformbare Schablonen* genutzt um Gesichtsmerkmale auch in verzerrten Formen finden zu können. Verformbare Schablonen sind parameterisiert. Um die richtigen Parameter für die Schablone zu finden, wird eine Energiefunktion minimiert, die tiefschichtige Merkmale, wie Kanten im Eingabebild, zu einem komplexen Merkmal, wie einem Auge, verbinden könnte.

In der Regel ist es für die merkmalsbasierten Methoden schwerer, mit Rotationen und Verformungen umzugehen. Viele Ansätze müssen zunächst versuchen, Kanten zu detektieren und diese zu Merkmalen zusammen zu setzen. Kanten im Bild können aber leicht durch Effekte wie Beleuchtung, Rauschen oder Verdeckungen beeinflusst werden. Schatten etwa führen zu Kanten, wo eigentlich gar keine geometrischen Merkmale sind. Ein weiterer Nachteil dieser Methoden ist, dass die Regeln, die beim Entwurf weitgehend festgelegt werden müssen, wesentlich zur Genauigkeit des Detektors beitragen. Dieses *a priori*-Wissen hängt aber nur von der Einschätzung des Entwicklers ab.

3.2.2 Bildhafte Methoden

Im Gegensatz zu den schablonenbasierten Methoden, lernen die bildhaften Methoden die Merkmale aus einer Reihe von präsentierten Gesichtern – und *Nicht-Gesichtern* – selbst. Kong et al. beschreiben sie als die dominanten Methoden seit Beginn der 1990er Jahre.

Methoden, die diesen Ansatz verfolgen, detektieren in der Regel nur. Um eine Lokalisierung zu erhalten muss ein weiterer Ansatz verwendet werden. Beispielsweise kann die *Sliding Window*-Technik näheren Aufschluss über die Position des Gesichts im Bild geben.

3.2.2.1 Eigenfaces und Eigenfeatures

Kirby and Sirovich entwickelten in [SK87] eine Methode unter Verwendung der Hauptkomponentenanalyse (PCA), um effektiv Gesichter zu repräsentieren. Ausgehend von einer Menge von verschiedenen Bildern mit Gesichern, wird mittels der Hauptkomponentenanalyse eine Menge von Eigenvektoren der Kovarianzmatrix dieser Menge ermittelt. Diese Eigenvektoren spannen dann einen Unterraum - den *Face-Space*, auf. Jedes Gesicht aus

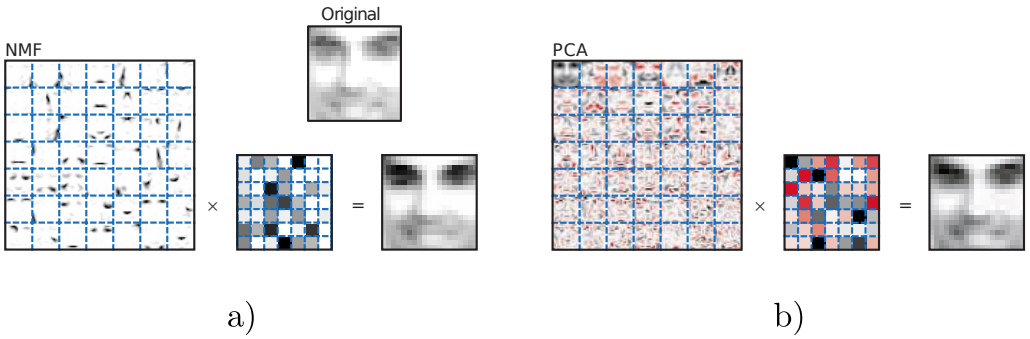


Abbildung 3.6: Vergleich der Basis-Merkmale bei PCA und NMF, aus denen Gesichter kombiniert werden können. Quelle [LS99]

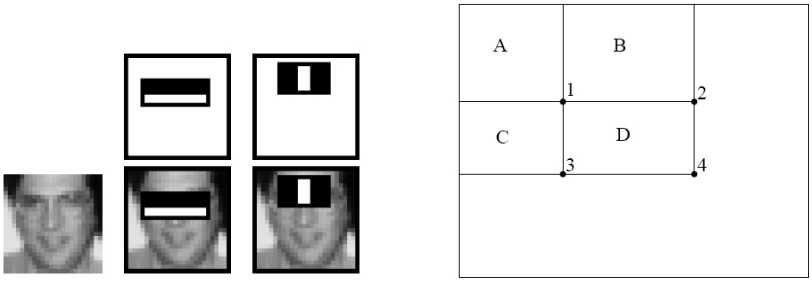
der Trainingsmenge kann dann durch eine Linearkombination der größten Eigenvektoren – den Eigenfaces – dargestellt werden.

Hier werden gegenüber den oben beschriebenen Ansätzen den Vorteil, dass die verwendeten Merkmale gelernt werden. Das in dieser Arbeit verwendete Konvolutionsnetz ist darüber hinaus auch in der Lage Merkmale verschiedener Komplexität zu kombinieren.

Turk and Pentland [TP91] haben diese Methode später auch zur Lokalisierung und Detektion von Gesichtern verwendet. Dazu nutzen sie aus, dass die Rekonstruktion der Bilder aus ihren Hauptkomponeten immer eine Approximation ist. Ein Fehler zwischen dem Originalbild und dem aus der Projektion rekonstruierten Bild kann dann berechnet werden. Abschliessend kann entschieden werden, ob der Fehler zu groß ist, um noch ein Gesicht darstellen zu können.

Moghaddam und Pentland [MP94] haben diese Technik dann auf einzelne Gesichtsmerkmale angewandt. So bekamen sie statt eines Eigenface-Unterraums mehrere Unterräume, beispielsweise für *Eigen-Eyes* (Augen) und *Eigen-Mouths* (Mund). Mit einem Sliding-Window-Ansatz werden verschiedene Bereiche des Bildes in verschiedenen Skalierungen extrahiert und in die Merkmalsräume projeziert und rekonstruiert. Wenn dann alle oder einige Merkmale erkannt werden, wird ein Gesicht erkannt. Mit diesem Verfahren konnten Moghaddam und Pentland die Erkennung deutlich robuster machen.

Lee und Seung stellen in [LS99] die Methode der nicht-negativen Matrixfaktorisierung (NFM) vor. Hier sind die Faktoren – im Gegensatz zur PCA – alle nicht negativ. Daher sind die resultierenden Merkmale wie Bausteine, aus denen ein Gesicht zusammengesetzt werden kann. So sind die Merkmale im Vergleich zu PCA leichter interpretierbar (vgl. Abbildung 3.6), was neben der einfachen Berechenbarbeit einer der wesentlichen Vorteile der Methode ist. Abbildung illustriert diesen Vergleich.



(a) **(b)**

Abbildung 3.7: (a) Viola und Jones Merkmale werden auf den Bildern angewendet. (b) Das Integralbild von Viola und Jones: Die Pixel an den Positionen 1, 2, 3 und 4 enthalten die Summen aller Pixel im Rechteckt zwischen (0,0) und dem jeweiligen Punkt. Der Inhalt des Rechtecks aus D lässt sich daher aus D-B-C+A berechnen.

3.2.2.2 Haar-Klassifizierer

Viola und Jones haben in [VJ01] einen Ansatz vorgeschlagen, der schnell ist und die Verarbeitung von Videodaten in Echtzeit ermöglicht. Zudem ist der Ansatz nicht auf die Detektion von Gesichtern beschränkt.

Sie entwickelten eine Kaskade, basierend auf den AdaBoost-Ansatz. Kleine, einfach zu berechnende Klassifikatoren basierend auf einer Menge von Haar-ähnlichen Merkmalen werden in einer Kaskade nacheinander abgearbeitet. Die Klassifikatoren werden der gelernten Bedeutung nach angeordnet, so dass wichtige vorne stehen. Nur wenn ein Klassifikator auf „Gesicht“ entscheidet geht die Prüfung in die nächste Stufe der Kaskade. Regionen ohne Gesichter können so schnell durchlaufen werden, da die komplette Kaskade erst durchlaufen werden muss, wenn tatsächlich ein Gesicht erkannt wird. Eine Bildrepräsentation, die sich Integralimage nennt, beschleunigt diesen Prozess noch.

Im Gegensatz zu den in Konvolutionsnetzen verwendeten Merkmale, sind die hier verwendeten Merkmale einfacher Natur (vgl. Abbildung 3.7). Sie beschreiben großflächige Verteilungen von hoher und niedriger Pixelintensität. So wird im Viola & Jones Detektor in der Regel das Verhältnis von dunklen zu hellen Flächen in größeren Blöcken untersucht.

3.2.2.3 Neuronale Netze und Multi-Layer-Perzeptrons

Formuliert man die Problemstellung der Gesichtserkennung als Klassifikationsproblem, so liegt der Einsatz neuronaler Netze nahe. Dieser Ansicht folgend wurden auch einige An-

sätze entwickelt, die auf neuronalen Netzen basieren. Vorteilhaft ist dabei, dass neuronale Netze die für die Unterscheidung zwischen Gesichtern und Nicht-Gesichtern wichtigen Merkmale selbst lernen können. Der Einfluss des Entwicklers ist auf die Zusammenstellung eines sinnvollen Trainingsdatensatz und die Festlegung der Netzstruktur beschränkt. Allerding ist bei allen vollverknüpfen Netzen die Trainingsdauer und auch die Verarbeitungsdauer ein entscheidendes Ausschlusskriterium.

Agui et al. haben 1992 ein hierarchisches Netz in [AKNN92] vorgeschlagen. Es bestand in der ersten Hierarchiebene aus zwei parallelen Subnetzen. Eines dieser Subnetze bekommt als Eingabe das normale Bild. Das andere bekommt eine mit einem 3×3 Sobelfilter vorverarbeitete Kopie des Eingabebildes. Die Ausgaben dieser Subnetze und zusätzliche Merkmale, wie das Verhältnis zwischen der Anzahl weißer Pixel und der Gesamtzahl der Pixel, dienen als Input für die nächste Ebene. Deren Ausgabe zeigt dann an, ob das Netz ein Gesicht gefunden hat oder nicht.

Ein Nachteil dieses Ansatzes war, dass die Gesichter alle die selbe Größe haben mussten, damit sie erkannt werden. Diesen Nachteil versuchten Soulie et al. in [FS93] durch Wavelet-Transformationen zu umgehen. Dazu nutzen sie ein Time-Delay-Netzwerk mit einer Eingabegröße von 20×25 Pixel. Mit einer berichteten Falsch-Negativ-Rate von $2{,}7\%$ Prozent und einer Falsch-Positiv-Rate von $0{,}5$ Prozent erzielten sie gute Resultate.

Feraud und Bernier zeigten ebenfalls gute Ergebnisse. Sie näherten mit einem 5-schichtigen neuronalen Netz eine nichtlineare Hauptkomponenten-Analyse (NLPCA) an. Die grundlegende Idee dabei ist dass das Netzwerk die Daten kodieren und wieder Dekodieren muss. Die mittlere Schicht bildet den Flaschenhals mit so vielen Neuronen, wie Basisvektoren gesucht werden. In ihren Experimenten verwendeten Feraud und Bernier ein Netzwerk um nach frontalen Gesichtern zu suchen. Ein zweites war darauf trainiert um 60 Grad gedrehte Gesichter zu erkennen. Ein weiteres Netzwerk (*gating network*) wird darauf trainiert die Entscheidungen der Detektor-Netzwerke richtig zu interpretieren. Mit dieser Architektur gelang ihnen ein ähnlich gutes Ergebnis [YKA02], wie bei dem im Folgenden beschriebenen weit verbreiteten Ansatz. Burel und Carol [BC94] sowie von Vaillant et al. [VCLC94] entwickelten eine Architektur, die ebenfalls von Rowley et al. [HR96] in abgewandelter Form veröffentlicht wurde (vgl. Abbildung 3.8). Allen Ansätzen sind zwei Hauptkomponenten gemein:

Ein Modul zur Gesichtsmustererkennung mit mehreren neuronalen Netzen. Ein Netzwerk bekommt dabei einen Bildausschnitt der Größe 20 Pixel mal 20 Pixel präsentiert (sliding window). In seiner Ausgabe $v \in \{-1, 1\}$ gibt das Netzwerk einen Tipp ab, ob ein Gesicht im Bildausschnitt vorhanden ist ($v = 1$), oder nicht ($v = -1$).

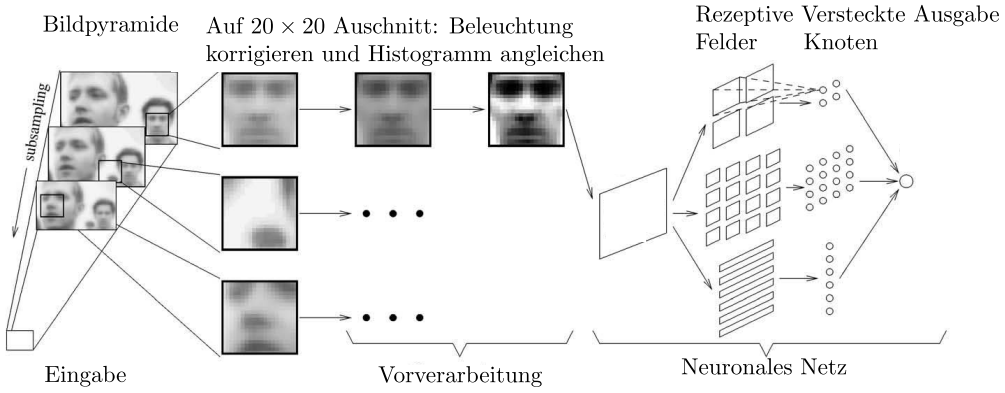


Abbildung 3.8: Das Erkennungssytem nach Rowley: Ausschnitte des Bildes werden alle vorverarbeitet in ein großes neuronales Netz eingegeben. Quelle: [HR96], bearbeitet

Ein Entscheidungsmodul welches letztlich die Entscheidung darüber trifft, ob ein Gesicht im Bild auftritt oder nicht. Diese Entscheidung soll die Tipps der Module zur Gesichtsmustererkennung integrieren. Dazu wurden simple logische Operatoren oder auch Abstimmungsverfahren genutzt.

Um das Problem der Skalierung zu lösen wurde die Auflösung des Eingabebildes sukzessive reduziert (subsampling). In jeder Auflösungsstufe werden wieder Abtastungsfenster über das Bild geschoben und den Neuronalen Netzen präsentiert. Damit erreichten die Autoren eine Entdeckungsrate von bis zu $92{,}3$ Prozent.

Das Problem bei der Verwendung vollverknüpfter neuronaler Netze ist der hohe Rechenaufwand insbesondere im Training. Gerade bei tieferen Architekturen die mehr als fünf Schichten haben, führt der Rechenaufwand auf größeren Bildern zu nicht mehr vertretbaren Trainingszeiten (siehe dazu auch [Ben09]). Dieses Problem tritt bei Konvolutionsnetzen durch die geteilten Gewichte weniger stark auf, weshalb sie im Bereich der Klassifiezierung auf Bildern gerne verwendetet werden. Aktuelle Arbeiten versuchen das Trainingszeitproblem zu lösen, in dem die Berechnungen durch die Verwendung von GPUs beschleunigt werden. Auf dem gut untersuchten Datensatz MNIST [YLH98] konnte so gerade eine der besten Ergebnisse erzielt werden. Ciresan et al. [CMGS10] erreichten eine Fehlerrate von $0{,}35$ Prozent mit einem sechs-schichtigen Neuronalen Netz.

3.2.2.4 Konvolutionsnetze

Die allgemeine Objekterkennung mittels Neuronaler Netze ist ebenfalls ein verbreiteter Ansatz. Auch hier werden zunächst Merkmale extrahiert, um die Dimension des Merkmalsraumes zu verkleinern. Anhand dieser gewonnenen Merkmalsvektoren wird dann

die Klassifikation durch ein Neuronales Netz vorgenommen. Solche Ansätze sind auch für die Erkennung von Gesichtern vorgeschlagen worden, beispielsweise in [HMLYC97]. Allerdings setzt eine solche Lösung voraus, dass man die räumlich und zeitlich invariante Merkmale kennt und einfach ausrechnen kann. Sollen diese erst gelernt werden, werden große und komplexe Netze notwendig. Bengio beschreibt in [Ben09], dass Ansätze die komplexe Objekterkennung auf Bildern mit vollverknüpften Neuronalen Netzen mit mehr als vier Schichten leisten bis 2006 als schwer trainierbar galten. Da bei Konvolutionsnetzen die Anzahl der freien Parameter durch das weight-sharing deutlich kleiner ist, waren diese bis Mitte des Jahrzehnts beliebter.

Eine frühe Arbeit, die ein Konvolutionsnetz zur Gesichtserkennung einsetzt ist die Arbeit von Lewrence et al. aus 1997 [LGTB97]. Dieses System reduziert zunächst die Dimension des Eingaberaumes, in dem es mit einer selbstorganisierenden Karte die Eingabemuster quantisiert. Damit wird auch die Robustheit gegen Rauschen gestärkt. Das Ergebnis wird als Eingabe für ein Konvolutionsnetz verwendet, mit dem sie zumindest partielle Invarianz gegen Translation, Rotation, Skalierung und Deformation erreichen. Im Vergleich mit den 5,3 % einer PCA basierten Methode erreichten sie mit 3,8 % eine niedrigere Fehlerrate auf der ORL-Datenbank mit je fünf Trainings- und Testbildern von 40 verschiedenen Personen.

Im wesentlichen wurden Konvolutionsnetze für Klassifizierungsaufgaben verwendet. Datensätze, wie beispielsweise der NORB-Datensatz [YL04] stellen die Aufgabe, Objekte in Bildern anhand der Form erkennen. Die Bilder des Datensatzes zeigen je eine Spielzeugfigur aus den fünf Kategorien animal, human, airplane, truck und car. Dazu lernen die Konvolutionsnetze Merkmale, und Kombinationen von Merkmalen. Auf der letzten Schicht des Netzes sind dann Neurone, deren Aktivität hoch ist, wenn die zugehörige Klasse erkannt wurde. Das Ziel dieser Arbeit ist es hingegen ein Netz zu entwickeln, welches gleichzeitig zur Art des Körperteils auch die Position der Körperteile in einer Neuronenkarte zu lernen.

Mit seiner rekurrenten Abstraktionspyramide (NAP) – siehe Abschnitt 3.1.3 – erreichte Behnke eine Fehlerrate von $1{,}5$ Prozent. Er verwendete für seine Versuche herunterskalierte Bilder der Größe 48×36 aus dem BioID Datensatz. Dieser enthält 1521 Bilder von 23 Personen mit unterschiedlichem Geschlecht, Alter und verschiedener Hautfarbe. Behnke unterteilte in seinem Versuch die Bilddatenbank in 1000 Trainingsbilder und 521 Testbilder.

Diese Architektur kommt der in dieser Arbeit entworfenen Architektur und Aufgabenstellung am nächsten. Der Netzaufbau der NAP realisert durch die Anwendung von geteilten Gewichten prinzipiell auch Faltungen. In Abweichung von dieser Architektur soll einerseits die Erkenntnis von Scherer et. al. [SMB10a] eingearbeitet werden, dass Pooling bessere

Resultate erzielt. Zudem soll ohne die Beschränkung der Aufteilung in anregende und hemmende Verbindungen aufgehoben werden und das Netz beschleunigt werden, so dass größere Bilder und Filter verarbeitet werden können.

3.3 Handerkennung

Die Suche nach Händen im Bild wird durch die Beweglichkeit und durch viele verschiedene Formen der Hände erschwert. Kölsch und Turk [KT04] und Ong und Bowden [OB04] erzielten mit einem Kaskadenansatz und Haar-ähnlichen Merkmalen gute Ergebnisse. Kölsch und Turk zeigten darüber hinaus, dass es Handformen gibt, die sehr schwer zu detektieren sind. Sie argumentierten aber auch, dass eine Hand nicht in jedem Frame erkannt werden müsse. Unter Einbeziehung des zeitlich-räumlichen Kontextes lasse sich die Hand dennoch gut verfolgen.

Auch Axenbeck verwendet in [ABBB08] einen Viola-Jones-basierten Detektor. Dieser wurde darauf trainiert die rechte Hand zu erkennen. Dazu wurden viele Bilder mit verschiedenen Handkonturen die zentriert im Bild waren zum Training benutzt. Um die linke Hand zu detektieren, wurde das Bild einfach gespiegelt. Da Axenbeck die Handerkennung im Rahmen der Gestenerkennung einsetzte, wurden die Positionen, an denen die Hände sein können, von vornherein durch die letzte Position und den Geschwindigkeitsvektor der Hand beschränkt. Eine auf diese Positionen beschränkte Sliding-Window-Technik ermöglichte die Lokalisierung.

3.4 Neuronale Netze auf Grafikkarten

Die Firma NVIDIA veröffentlichte in ihren White Papern eine Arbeit von Podlozhnyuk [Pod07], in der die Beschleunigung von Faltungen auf Grafikkarten untersucht wurde. Die Fallstudie beschreibt in verschiedenen Schritten, wie eine Optimierung der Laufzeit der Faltungsoperation bei separierbaren Filtern entworfen werden kann. Podlonzhnyuk erreichte mit einem 5×5-Filter ein Durchsatz von bis zu 1800 Megapixeln pro Sekunde.

Rafael Uetz implementierte in seiner Arbeit [Uet09] ein lokal verknüpftes hierarchisches Neuronales Netz. Performanzkritische Funktionen des Systems wurden unter Verwendung des mit Hilfe des CUDA-Frameworks implementiert. Dadurch erreichte Uetz eine maximale Beschleunigung um den Faktor 82 gegenüber einer Single-Core CPU-Implementierung. So wurde es möglich mit deutlich größeren Datenmengen zu arbeiten,

beispielhaft mit einer Menge bestehend aus 86.574 Mustern unterteilt in 12 verschiedene Objektklassen. Dominik Scherer unternahm in 2009 den Versuch, ein Konvolutionsnetz zu beschleunigen [Sch09]. Ihm ist es gelungen, das Trainieren dieses Netzes bis auf den Faktor 115 zu beschleunigen.

Im Vergleich zu diesen Arbeiten, die durch die Verarbeitung kleiner Batches (*Mini-Batches*) wesentliche Geschwindigkeitsvorteile erzielen, muss in dieser Arbeit durch die Rekurrenz ein anderer Weg zur Beschleunigung gewählt werden. Mehr dazu findet sich in Abschnitt 4.1.2.

4 Entwurf und Implementierung der Komponenten der Neuronalen Pyramide

4.1 Allgemeine Konzepte

4.1.1 Struktur des Netzes

4.1.1.1 Neuronen- und Axonschichten

Um Modular zu bleiben und Bestandteile der Pyramide austauschen oder wie im Fall des BPTT wieder verwenden zu können, wurden Schichten und ihre Verbindungen getrennt. So gibt es Neuronenschichten N^l und Axonschichten A^l, wobei l den Index der Schicht bezeichnet. Der Index $l = 0$ bezeichnet die unterste Schicht des Netzes.

Die Neuronenschichten speichern die Aktivierung N_A^l und Fehlersignale N_E^l. Die Aktivierungen N_A^l bestehen genauer aus den Aktivierungen aller $m \in \mathbb{N}, m > 0$ Neuronenkarten der Schicht. Um die Aktivierung eines Neurons an der Position (i, j) der Neuronenkarte m in der Schicht l explizit anzusprechen wird die Notation $N_A^{l,m}(i, j)$ verwendet. Im Folgenden bezeichne L die Anzahl der Schichten im Netz und $M(l)$, mit $l \in \{0, \ldots, L-1\}$ die Anzahl der Karten in Schicht l. Benachbarte Neuronenschichten N^l und N^{l+1} sind nicht direkt miteinander verbunden, sondern über Axonschichten $A^{l \to l+1}$. Dieser Zusammenhang ist in Abbildung 4.1 dargestellt. Die Neuronenschichten speichern eine Menge von eingehenden und eine Menge von ausgehenden Verbindungen. Eine Axonschicht implementiert eine der möglichen Verbindungen:

- Konvolutionsverbindungen $A_C^{l \to l+1}$
- Poolingverbindungen $A_P^{l \to l+1}$
- Supersamplingverbindungen $A_S^{l \to l+1}$
- Voll verknüpfte Verbindungen $A_V^{l \to l+1}$
- Konvolutions- und Poolingverbindungen $A_{CP}^{l \to l+1}$
- Supersampling- und Konvolutionsverbindungen $A_{SC}^{l \to l+1}$

Soll eine Neuronenschicht mehrere Axonschichten haben, was insbesondere bei der Anlage eines rekurrenten Netzes der Fall ist, muss beachtet werden, dass nicht alle Kombinationen zulässig sind. Es können nur Verbindungen implementiert werden, deren Eingangs- bzw.

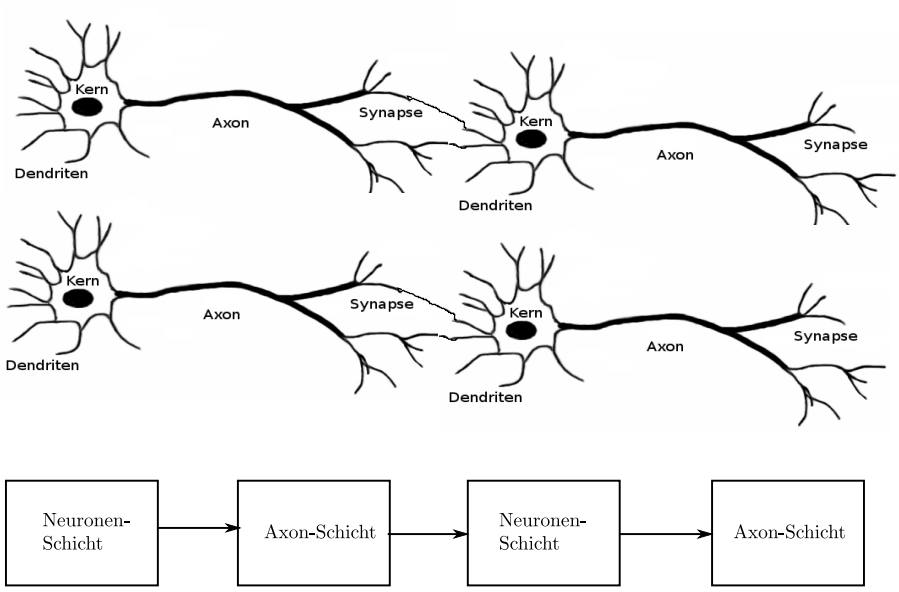


Abbildung 4.1: Oben: schematische Darstellung zweier verbundener Neuronen; unten: Neuronenschichten stehen für die Aktivierung eines Neurons, die Axonschichten für die Übertragung des Signals

Ausgabegrößen gleich sind. Ist beispielsweise zwischen zwei Neuronenschichten N^l und N^{l+1} eine Poolingschicht $A_P^{l \to l+1}$, die die Eingabe verkleinert, so kann in der Rückrichtung keine Konvolutionsschicht $A_C^{l \to l+1}$, sondern nur eine Supersamplingschicht $A_S^{l \to l+1}$ angelegt werden. Ansonsten würden die Ausgabe- und Eingabegrößen der Schichten nicht übereinstimmen.

Die Axonschichten speichern neben den Konvolutionsgewichten $A_{C,W}^{l \to l+1}$ auch den Gewichtsgradienten und andere für das Lernverfahren benötigte Parameter, wie beispielsweise die Lernraten. Soll ein einzelner Faltungsfilter von einer Karte i in Schicht l zu einer Karte j in Schicht $l+1$ angesprochen werden, so wird dies im Folgenden mit $A_{C,W}^{l \to l+1, i \to j}$ notiert.

Die topologische Struktur des Netzes wird in einer Netzklasse gespeichert. Diese sollte immer von der Klasse `NetBase` oder `GuiAdapter` abgeleitet werden, da diese die benötigten Grundfunktionalitäten zur Verfügung stellen. Die Klasse `NetBase` ist dafür zuständig ein Netz aus gegebenen Parametern zu bauen. Sie stellt auch nützliche Funktionen, beispielsweise zur Ausgabe von Matrizen, zur Verfügung. Beispielsweise kann der Klasse eine Folge von Vorwärtsverbindungen übergeben werden und es konstruiert automatisch die richtigen rekurrente Verbindungen. Die Klasse `GuiAdapter` stellt darüber hinaus auch die Funktionen zur Verfügung, die von der Grafischen Benutzeroberfläche genutzt werden.

4.1.1.2 Entfaltung des Netzes

Der BPTT-Algorithmus (siehe Abschnitt 2.2.5.2) sieht vor, dass das rekurrente Netz über n Zeitschritte entfaltet wird. So wurde entschieden, die rekurrente Pyramide direkt als zeitlich entfaltetes neuronales Netz zu implementieren. Dazu müssen die Neuronen- und Axonschichten pro Zeitschritt kopiert werden. Für Konvolutions- und Vollverknüpfungsschichten gilt allerdings eine Besonderheit: Da sie in allen Zeitschritten die gleichen Gewichte verwenden sollen, werden diese nicht mit kopiert, sondern in einem eigenen Gewichtsobjekt gespeichert. Jede Kopie der Verbindung greift daher auf ein gemeinsames Gewichtsobjekt zu und nutzt dieses für die Faltung. Die Axonschichten speichern daher die topologische Information – welche Schicht mit welcher Verknüpft ist – während die Informationen zur Merkmalsextraktion in ein Gewichtsobjekt ausgelagert sind.

Daher erhält die Notation der Neuronenschichten in zeitlich entfalteten Netzen zusätzlich noch einen Zeitindex $t : N_t^l$, ebenso wird die Aktivierung im Zeitschritt t als $N_{A,t}^l$ und der entsprechende Fehler t wird als $N_{E,t}^l$ notiert. Im Fall des vorwärts gerichteten Netzes entfällt der Zeitindex nach dem Komma.

Jede Axonschicht $A^{l \to l+1}$ hat ein oder mehrere Quell-Schichten $N_{t_i}^l$ und ein oder mehrere Zielschichten $N_{t_j}^{l+1}$ mit $t_i, t_j \in \{1, \ldots, T\}$. Ist $T > 1$, so handelt es sich um ein zeitlich entfaltetes Netz und es werden nur Verbindungen von einem Zeitindex t_i zu einem Zeitindex $t_j = t_i + 1$ zugelassen. Ist T hingegen 1, so handelt es sich nicht um ein zeitlich entfaltetes Netz. Dann werden nur Verbindungen zwischen gleichen Zeitschritten zugelassen. In Abbildung 4.2 wird dargestellt, wie die Komponenten in einem entfalteten Netz zusammenwirken. In horizontaler Richtung sind die Schichten N^l in aufsteigender Folge angeordnet. In vertikaler Richtung steigt die Zeit auf. Die Neuronenschichten sind dabei durch Rechtecke mit spitzen Ecken dargestellt. Axonschichten haben in der Abbildung abgerundete Ecken. Durchgehende Pfeile gehören zu aufwärts gerichteten Axonverbindungen, gestrichelte Pfeile zu lateralen Axonverbindungen und gepunktete Pfeile gehören zu nach unten gerichteten Verbindungen. In Abbildung 4.3 wird deutlich, dass zwischen den temporalen Kopien einer Axonschicht ein gemeinsames Gewichtsobjekt geteilt wird. Dieses Gewichtsobjekt wird in den Instanzen der Axonschichten zur Faltung genutzt. So wird sichergestellt, dass die Gewichte über alle Zeitschritte gleich sind. Zusätzlich wird im Trainingsverfahren der Fehler, der bei allen Aufrufen der Axonobjekte, entsteht in diesem Gewichtsobjekt gesammelt und zur Aktualisierung der Gewichte verwendet.

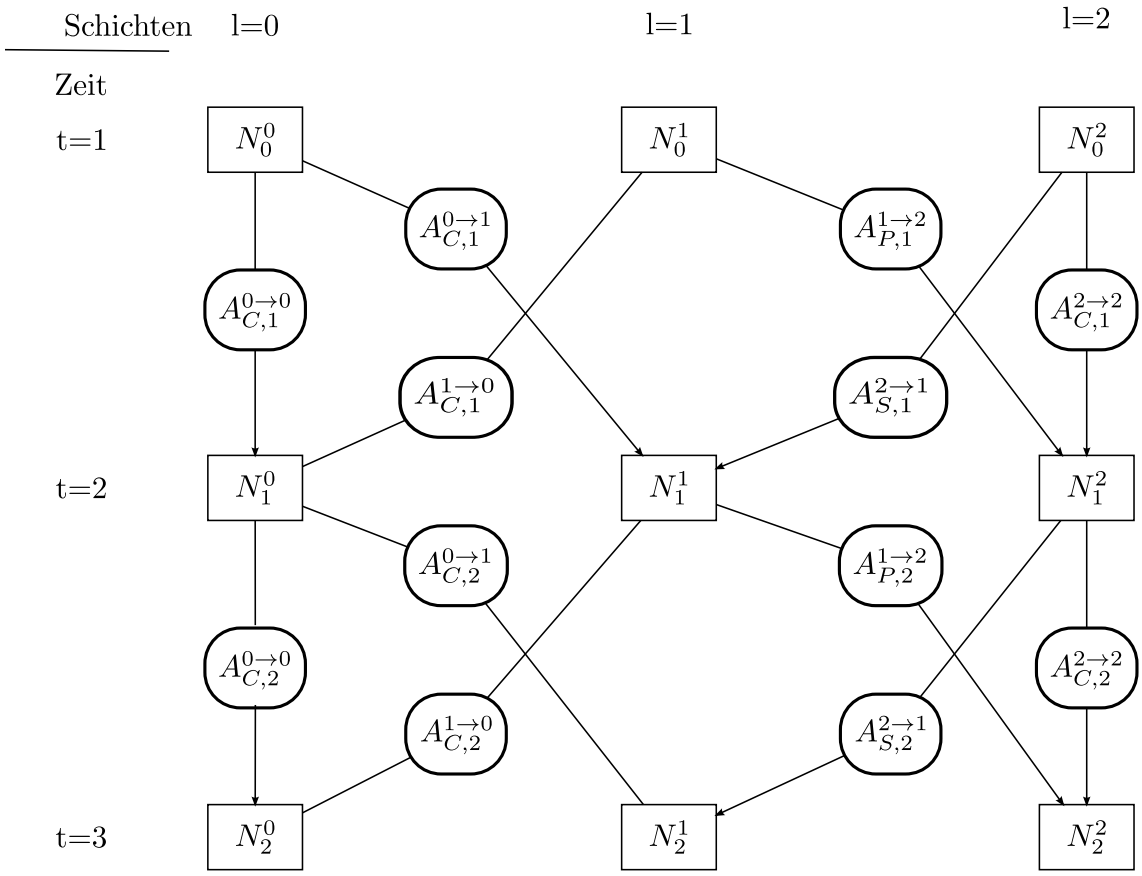


Abbildung 4.2: Schematische Darstellung der Verknüpfungen in dem entfalteten Netz. Rechtecke repräsentieren die Neuronen-Schichten, Rechtecke mit abgerundeten Ecken repräsentieren die Axonschichten. Das Beispiel zeigt ein Netz mit drei Neuronenschichten und zwei Axonschichten. Die erste Axonschicht ist eine Faltungsverbindung, die zweite Axonschicht ist eine Maxpooling-Verbindung.

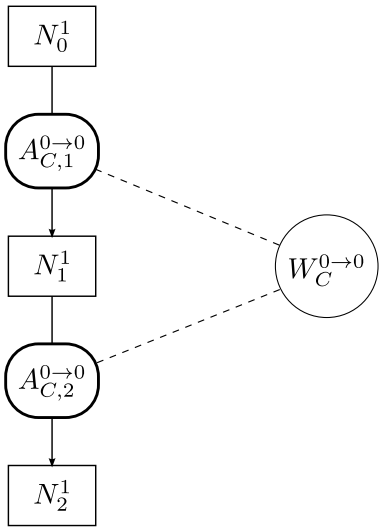


Abbildung 4.3: Schematische Darstellung der Teilung des Gewichtsobjektes $W_C^{0\to0}$ zwischen zwei Axonschichten $A_{C,1}^{0\to0}$ und $A_{C,2}^{0\to0}$. Die beiden Axonschichten wurden hier wieder für die Zeitschritte kopiert. Durch die Verwendung des gleichen Gewichtsobjektes ist sichergestellt, dass in jedem Zeitschritt die gleichen Gewichte genutzt werden.

4.1.1.3 Objektmodell

Hier werden kurz die Klassen besprochen, durch welche das Netz repräsentiert wird. Diese werden in Abbildung 4.4 dargestellt. In der Klasse `Layer` werden grundlegende Helferklassen angelegt. Davon erben die Klassen `WeightLayer` und `NeuralLayer`, welche die Axon- und Neuronen-Schichten repräsentieren. Die Sohnklassen von `WeightLayer` implementieren dann konkrete Verbindungen:

- Die Klasse `ConvWeightLayer` implementiert eine Konvolutionsverbindung,
- Die Klasse `MaxPoolWeightLayer` implementiert ein 2×2-MaxPooling,
- Die Klasse `SupersamplingWeightLayer` implementiert ein Supersampling um den Faktor 2.
- Die Klasse `ConvPoolWeightLayer` implementiert eine Verbindung, bei der erst gefaltet und anschliessend gepoolt wird.
- Die Klasse `PoolConvWeightLayer` implementiert eine Verbindung, bei der erst gepoolt und anschliessend gefaltet wird.
- Die Klasse `BiasWeightLayer` implementiert eine Verbindung von einer imaginären Neuronenkarte, deren Aktivität immer bei eins liegt. Pro Karte in der Zielschicht kann die Verbindungsschicht ein Gewicht lernen.

4.1.2 Durch Rekurrenz induzierte Einschränkungen

In dieser Arbeit soll ein rekurrentes Konvolutionsnetz auf einer CUDA-fähigen Grafikkarte beschleunigt werden. Da der globale Speicher der Grafikkarte mit etw 1 GB relativ klein ist, war zu Beginn zunächst zu klären,

- ob lokale Verknüpfungen für die Erkennung nutzbar waren, oder nur Konvolutionsoperationen in Frage kommen und
- ob das Netz über mehrere Muster parallelisiert werden sollte, oder über ein Muster oder eine Sequenz.

Da das Entwurfsziel war, dass das Netz auf einer GPU beschleunigt werden sollte, um in Echtzeit eine Sequenz von Bildern zu verarbeiten, wurde zunächst Speicherbedarf ermittelt. Dazu wurde zunächst von einem rekurrent lokal verknüpften Neuronalen Netz mit drei Verbindungsschichten, die gleichzeitig auch ein Max-Pooling durchführen, ausgegangen. In Tabelle 4.1 wird das Ergebnis für vier verschiedene Fälle gezeigt.

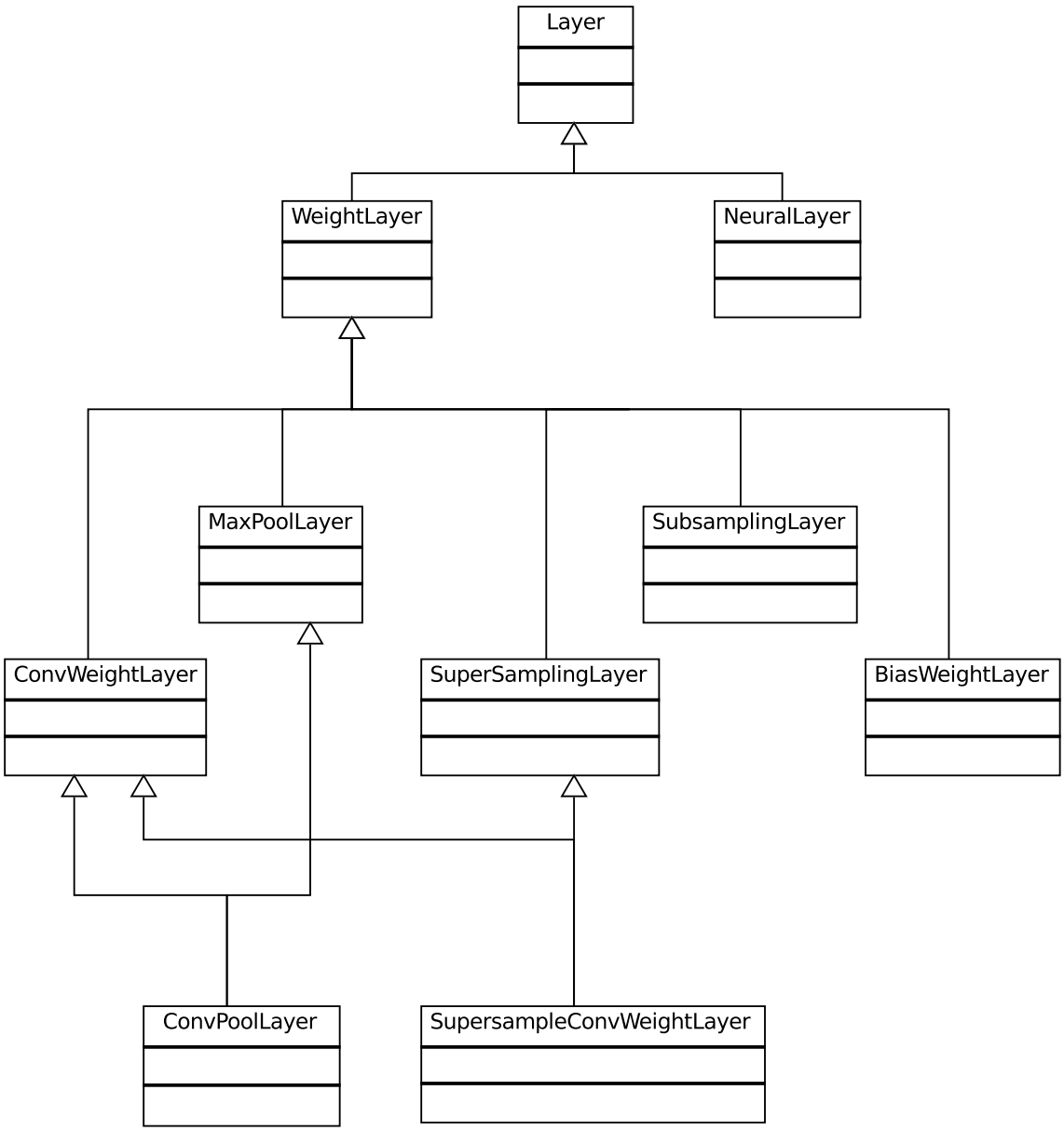


Abbildung 4.4: UML-Diagramm der in dieser Arbeit entworfenen Software. Dargestellt sind nur die wichtigsten Klassen.

Um den benötigten Platz zu berechnen wurde davon ausgegangen, dass alle benötigten Parameter vom Datentyp `float` sind, welcher vier Bytes Speicherplatz belegt. Die Größe der Eingabeschicht ergibt sich einfach aus der Anzahl der benötigten Neuronen. Damit ein Bild in eine Karte der Eingabeschicht geladen werden kann, muss diese so groß sein, wie das Eingabebild. Da in dieser Arbeit nur quadratische Bilder verwendet werden, reicht die Bildbreite zur Beschreibung der Bildgröße aus.

Im Folgenden wird die Bildbreite in der Eingabeschicht mit w_0, in der ersten Schicht mit w_1 und fortlaufend bezeichnet. Ebenso wird die Anzahl der Karten mit m_i bezeichnet, wobei $i \in N_+$ die Schicht angibt, deren Karten referenziert werden sollen. Die Größe der Eingabeschicht $N_A(0)$ berechnet sich daher aus

$$N_A(0) = w_0 \cdot w_0 \cdot m_0$$

In den folgenden Schichten wird im Fall der lokalen Verknüpfung davon ausgegangen, dass in den Schichten zweistufig arbeiten. In der ersten Stufe wird zunächst eine Eingangsaktivierung berechnet, die genauso groß ist, wie die Aktivierung der vorangehenden Schicht, aber so viele Karten hat, wie Schicht i. Anschliessend wird durch Maxpooling die Auflösung verringert. Daher müssen die Aktivierungen der beiden Stufen gespeichert werden:

$$N_A(i) = t \cdot w_{i-1}^2 \cdot m_i + tm_i \cdot w_i^2, i > 0$$

Um die Aktivierung durch die rezeptiven Felder zu berechnen, werden Gewichte verwendet. Diese Gewichte existieren im lokal verknüpften Netz für jedes Neuron. Jedes Neuron der verdeckten Schicht $i, i > 0$ besitzt je ein rezeptives Feld in den Schichten $i-1, i$ und $i+1$. Es wird angenommen, dass die lateralen rezeptiven Felder kleiner sind, als die aufwärts und abwärts gerichteten rezeptiven Felder. Entsprechend berechnet sich die Zahl der Gewichte von Schicht i $N_W(i)$ durch

$$N_W(i) = w_{i-1}^2 \cdot m_{i-1} \cdot m_i \cdot fs^2 + w_i^2 \cdot m_{i-1} \cdot m_i \cdot fs^2 + m_i^2 \cdot w_i^2 \cdot lfs$$

wobei lfs für die laterale Filterbreite und fs für die übrigen Filterbreiten steht.

Lediglich das in Fall 4 gezeigte Netz würde demnach überhaupt in den Speicher der Grafikkarte passen. Da die Übertragung der Daten auf die Grafikkarte aber als laufzeitkritische Operation angesehen werden muss, wurde als Entwurfsziel festgelegt, dass alle Daten des Netzes auf die Grafikkarte passen sollten. Diese Überlegungen legen nahe, dass auf lokale Verknüpfungen in dieser Arbeit verzichtet werden sollte.

In Tabelle 4.2 wurde untersucht, wie groß ein rekurrentes Konvolutionsnetz wird. Hier passen auch die Netze mit großen rezeptiven Feldern – also mit großen Filtern – in den globalen Speicher der Grafikkarte. Das liegt daran, dass die Faltungsgewichte von allen Neuronen der Zielschicht geteilt werden. Im Gegensatz dazu gibt es in einem lokal verknüpften Netz für jedes Neuron der Zielschicht einen Satz an Filtergewichten. Dementsprechend berechnet sich die Größe der Gewichte für ein Konvolutionsnetz $N_W(i)$ durch

$$N_W(i) = m_{i-1} \cdot m_i \cdot fs^2 + m_{i-1} \cdot m_i \cdot fs^2 + m_i^2 \cdot lfs$$

Fall 1 ist vergleichbar mit Fall 1 in Tabelle 4.1. Man erkennt leicht, dass die Platzersparnis bei dem Konvolutionsnetz im Wesentlichen durch die Reduktion der freien Parameter, also der Gewichte, entsteht. Während im Fall der lokalen Verknüpfung mehr als 3,5 Milliarden Parameter pro Schicht zu speichern sind, sind es bei einem Konvolutionsnetz nur etwa eine Million Parameter.

Fall 3 zeigt, dass auch große Filtergrößen realisierbar sind. Diese Filtergrößen decken ähnlich große Bildbereiche ab, wie die Filter in Behnkes Neuronalen Abstraktionspyramide [Beh01]. Sein Netz arbeitete auf Eingabebiltern der Größe 48×36 Pixeln und verwendete Filter der Größe 3×3 und 4×4.

Allerdings wird auch deutlich, dass sich ein Minibatch-Lernen, bei dem zwischen 8 und 16 Muster parallel verarbeitet werden, nicht anbietet, da der Platz lediglich für zwei Muster gleichzeitig reicht. Um das Netz zu Beschleunigen wurde die Verarbeitung über die alle Neuronen einer Schicht parallelisiert. Alle Karten einer Neuronenschicht werden also parallel verarbeitet. Daran wurde auch das Speicherformat angepasst. Zudem eröffnet diese Entscheidung die Möglichkeit das komplette Trainingsset im globalen Speicher der Grafikkarte zu halten, da dieses ebenfalls etwa 400 Megabyte groß ist.

4.1.3 Eingesetzte Techniken und Bibliotheken

4.1.3.1 Python und C++

Die vorliegende Arbeit wurde zu Teilen in der Skriptsprache Python und zu Teilen in der Programmiersprache C++ umgesetzt. Python wurde Anfang der 1990er Jahre entwickelt. Ziel war es, eine einfach strukturierte Skriptsprache bereit zu stellen, die schnell erlernt werden kann. Dies wurde durch den Einsatz einer im Vergleich mit anderen Programmiersprachen reduzierte Syntax mit einem großen Funktionsumfang erreicht. Diese Ausrichtung hat der Sprache nach dem Eindruck des Autors zu großer Beliebtheit verholfen. Heute

Netzparameter	Fall 1	Fall 2	Fall 3	Fall 4
Filtergröße	15.0	5.0	5.0	5.0
laterale Filtergröße	11.0	3.0	3.0	3.0
Bildgröße	384.0	384.0	192.0	128.0
Zeitschritte	10	10	10	10
Speicherbedarf für die Eingabeschicht				
Eingangsgröße	384 * 384	384 * 384	192 * 192	128 * 128
Eingabekarten	6	6	6	6
Aktivierungen	33 MB	33 MB	8 MB	3 MB
Speicher für Schicht 1				
Eingangsgröße (vor Pooling)	384 * 384	384 * 384	192 * 192	128 * 128
Karten (vor Pooling)	6	6	6	6
Ausgangsgröße (nach Pooling)	192 * 192	192 * 192	96 * 96	64 * 64
Karten (nach Pooling)	12	12	12	12
Anzahl Gewichte	3.628.302.336	379.551.744	94.887.936	42.172.416
Gewichte	13.840 MB	1.447 MB	361 MB	160 MB
Aktivierungen	84 MB	84 MB	21 MB	9 MB
Fehlerwerte	84 MB	84 MB	21 MB	9 MB
Speicher für Schicht 2				
Eingangsgröße (vor Pooling)	192 * 192	192 * 192	96 * 96	64 * 64
Karten (vor Pooling)	12	12	12	12
Ausgangsgröße (nach Pooling)	96 * 96	96 * 96	48 * 48	32 * 32
Karten (nach Pooling)	24	24	24	24
Anzahl Gewichte	3.628.302.336	379.551.744	94.887.936	42.172.416
Gewichte	13.840 MB	1.447 MB	361 MB	160 MB
Aktivierungen	42 MB	42 MB	10 MB	4 MB
Fehlerwerte	42 MB	42 MB	10 MB	4 MB
Speicher für Schicht 3				
Eingangsgröße (vor Pooling)	96 * 96	96 * 96	48 * 48	32 * 32
Karten (vor Pooling)	24	24	24	24
Ausgangsgröße (nach Pooling)	48 * 48	48 * 48	24 * 24	16*16
Karten (nach Pooling)	48	48	48	48
Anzahl Gewichte	3.628.302.336	379.551.744	94.887.936	42.172.416
Gewichte	13.840 MB	1.447 MB	361 MB	160 MB
Aktivierungen	21 MB	21 MB	5 MB	2 MB
Fehlerwerte	21 MB	21 MB	5 MB	2 MB
Sonstiger Speicher				
Gewichtsgradienten	3.460 MB	1.447 MB	361 MB	160 MB
Summe				
Belegter Speicher	55.687 MB	6.120 MB	1.524 MB	673 MB
Freier Speicher	-54.724 MB	-5.152 MB	-561 MB	289 MB

Tabelle 4.1: Vergleich der Speicherbedarfe verschiedener Netze mit lokaler Verknüpfung für die Verarbeitung von nur einem Muster gleichzeitig. Die letzten beiden Zeilen der Tabelle zeigen auf, wie viel Speicher das Netz belegt und wie viel Speicherplatz noch frei wäre, wenn alle Gewichte im globalen, ein Gigabyte großen Speicher der Grafikkarte abgelegt sind. Die Tabelle zeigt, dass die die Verwendung lokaler Verknüpfungen den Speicheraufwand für die Grafikkarte zu groß macht.

Netzparameter	Fall 1	Fall 2	Fall 3
Filtergröße	15.0	21.0	39.0
laterale Filtergröße	11.0	17.0	29.0
Bildgröße	384.0	384.0	384.0
Zeitschritte	10	10	10
Speicherbedarf für die Eingabeschicht			
Eingangsgröße	384 * 384	384 * 384	384 * 384
Eingabekarten	6	6	6
Aktivierungen	33 MB	33 MB	33 MB
Speicher für Schicht 1			
Eingangsgröße (vor Pooling)	384 * 384	384 * 384	384 * 384
Karten (vor Pooling)	6	6	6
Ausgangsgrößse (nach Pooling)	192 * 192	192 * 192	192 * 192
Karten (nach Pooling)	12	12	12
Anzahl Gewichte	49.824	105.120	340.128
Gewichte	0.8 MB	1,6 MB	5,3 MB
Aktivierungen	84 MB	84 MB	84 MB
Fehlerwerte	84 MB	84 MB	84 MB
Speicher für Schicht 2			
Eingangsgröße (vor Pooling)	192 * 192	192 * 192	192 * 192
Karten (vor Pooling)	12	12	12
Ausgangsgröße (nach Pooling)	96 * 96	96 * 96	96 * 96
Karten (nach Pooling)	24	24	24
Anzahl Gewichte	199.296	420.480	1.360.512
Gewichte	3 MB	6,5 MB	21,2 MB
Aktivierungen	42 MB	42 MB	42 MB
Fehlerwerte	42 MB	42 MB	42 MB
Speicher für Schicht 3			
Eingangsgröße (vor Pooling)	96 * 96	96 * 96	48 * 48
Karten (vor Pooling)	24	24	24
Ausgangsgröße (nach Pooling)	48 * 48	48 * 48	24 * 24
Karten (nach Pooling)	48	48	48
Anzahl Gewichte	797.184	1.681.920	5.442.048
Gewichte	12 MB	26,2 MB	85 MB
Aktivierungen	21 MB	21 MB	21 MB
Fehlerwerte	21 MB	21 MB	21 MB
Summe			
Used memory	342 MB	359 MB	435 MB
Free memory	620 MB	603 MB	527 MB

Tabelle 4.2: Vergleich der Speicherbedarfe verschiedener Konvolutionsnetze für die Verarbeitung von nur einem Muster gleichzeitig unter Verwendung des Resilient Propagation-Algorithmus, der mehr Speicher benötigt, als das normale Backpropagation. Die letzten beiden Zeilen der Tabelle zeigen auf, wie viel Speicher das Netz belegt und wie viel Speicherplatz noch frei wäre, wenn alle Parameter im globalen, ein Gigabyte großen Speicher der Grafikkarte abgelegt sind.

wird die Skriptsprache durch die viele Benutzer weiterentwickelt – ein Prozess der durch die Python Software Foundation moderiert wird.

Nach der Erfahrung des Verfassers, der im Rahmen dieser Arbeit erstmals mit Python gearbeitet hat, lässt sich die Skriptsprache tatsächlich leicht erlernen und eignet sich für die rasche Umsetzung kleiner bis mittlerer Projekte. Mit wenig Codezeilen kann vergleichbar viel Funktionalität umgesetzt werden. Da Python eine objektorientierte Sprache ist, unterstützt sie auch Polymorphie und Vererbung. Ebenfalls positiv hervorheben lässt sich, dass die Skriptsprache mit Hilfe des Moduls PyDev[1] gut in die Entwicklungsumgebung Eclipse[2] integrieren lässt. Neben den gängigen Hilfsmitteln, wie Syntax-Highlighting und Auto-Completion wird dem Entwickler auch ein anspruchsvolles Debugging ermöglicht.

Die Programmiersprache C++ [Str97] wurde von Bjarne Stroustrup hingegen mit dem Ziel entwickelt, Simulationsprojekte mit minimalem Speicherplatz und Zeibedarf zu realisieren. Die Programmiersprache ist geeignet um schnelle, maschinennahe Programme zu entwickeln. Mit den *C Develpment Tools* lässt sich auch ein Plugin für Ecplipse finden, welches die Entwicklung von C/C++ Programmen unterstützt – inklusive eines voll funktionsfähigen Debuggers. Die Nutzung der Programmiersprache C, die als Grundlage für die Programmierschnittstelle C für CUDA dient, wird ebenfalls von C++-Compilern verstanden und überstetzt.

Im Zusammenspiel sollen die Vorteile beider Welten ausgenutzt werden. Während arbeitsintensive Teile der Software mit einer C/C++-Bibliothek implementiert werden sollen, sollen die Strukturen des Netztes, die grafische Benutzeroberfläche und Hilfsfunktionen in Python realisiert werden, um schnelle Änderungen zu ermöglichen.

4.1.3.2 Die CUV-Faltungsroutienen

Die CUV-Bibliothek (siehe Abschnitt 2.3.3) stellt optimierte Faltungsroutienen auf der Basis der Arbeit von Alex Krizhevsky [Kri10] zur Verfügung. Vorraussetzung ist, dass sowohl die Bilder, als auch die Faltungsfilter in einer zeilenorientierten Matrix vorliegen. Krizhevsky erntwickelte mit Hilfe von templatebasierter Metaprogrammierung einige Templates, die in Kernels für spezielle Varianten instantiiert werden können. So gelingt es beispielsweise den lokalen Speicher möglichst optimal einzusetzen.

Es werden in dieser Arbeit zwei dieser Funktionen genutzt, welche im Folgenden kurz vorgestellt werden. Die Funktion `convolve()` faltet jedes der m Bilder in der Bildeingabe-

[1] Erhältlich auf http://pydev.sourceforge.net/
[2] Erhältlich auf http://www.eclipse.org

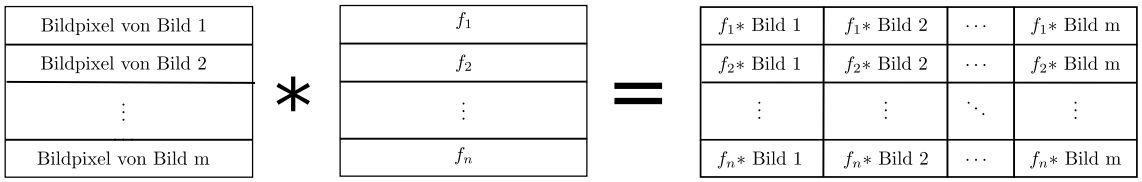


Abbildung 4.5: Die Funktion convolve()

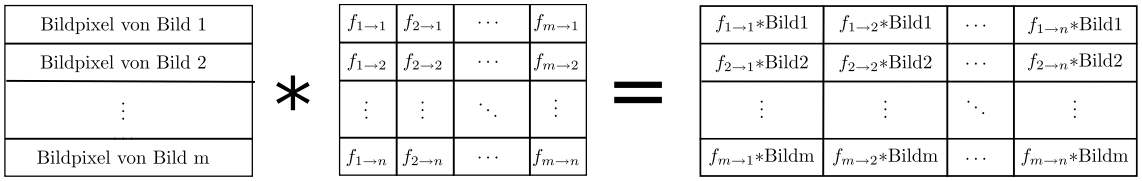


Abbildung 4.6: Die Funktion convolve2()

matrix mit den gleichen n Filtern. Die Bilder müssen dabei quadratische Größe k haben und zeilenweise vorliegen, so dass die Bildmatrix die Dimensionen $m \times k^2$ hat.

Die Filter müssen ebenfalls quadratisch sein, eine ungerade Filtergröße h haben und zeilenweise in einer Filtermatrix der Größe $n \times k^2$ abgelegt sein. Abbildung 4.5 visualisiert den Aufbau der Variablen, die der Funktion zu übergeben sind. Das Ergebnis der Funktion `convolve()` beteht aus einer Matrix mit den Dimensionen $n \times m \cdot (k - (\lfloor h/2 \rfloor))^2$. Der Grund für die Verkleinerung liegt darin, dass der Filter bündig mit dem ersten Bild angelegt wird. Das hat den Vorteil, dass die die Problematik der Randbehandlung wegfällt, führt auf der anderen Seite allerdings dazu, dass man das Eingabebild möglicherweise am Rand um die halbe Filterbreite auffüllen muss – dieses Auffüllen wird *padding* genannt. In einer Zeile sind dann hintereinander die Ergebnisse der Faltungen aller Bilder mit dem ersten Filter f_1 abgelegt. In der zweiten Zeile finden sich dann alle Faltungsergebnisse der Konvolutionen mit dem zweiten Faltungsfilter. Die Funktion `convolve2()` stellt an die Bildeingabematrix die gleichen Anforderungen, wie die Funktion `convolve()`. Der Unterschied zur vorgenannten Funktion ist, dass die m Bilder hier nicht mit den gleichen Filtern, sondern jedes Bild mit einer eigenen Menge von jeweils n Filtern gefaltet wird. So besteht eine Zeile in der Filtermatrix aus $m \cdot h^2$ Gewichten. Sie stehen für die ersten Filter, mit der die m Bilder gefaltet werden. Die Ergebnismatrix enthält in jeder Zeile alle Faltungsergebnisse des ersten Bildes mit allen Filtern hintereinander. Daher hat die Ergebnismatrix die Dimensionen $m \times n \cdots (k - (\lfloor h/2 \rfloor))^2$. Abbildung 4.6 visualisiert die Matrizen, die der Funktion `convolve2()` als Parameter zu übergeben sind.

Im Rahmen dieser Arbeit wurden dieser Bibliothek zwei wesentliche Erweiterungen hinzugefügt. Zunächst einmal wurde eine Funktion `rotate_filter()` implementiert, welche die Aufgabe hat die Filter um $180\,\mathrm{deg}$ zu rotieren. Sie findet ihren Einsatz in Abschnitt 4.10.

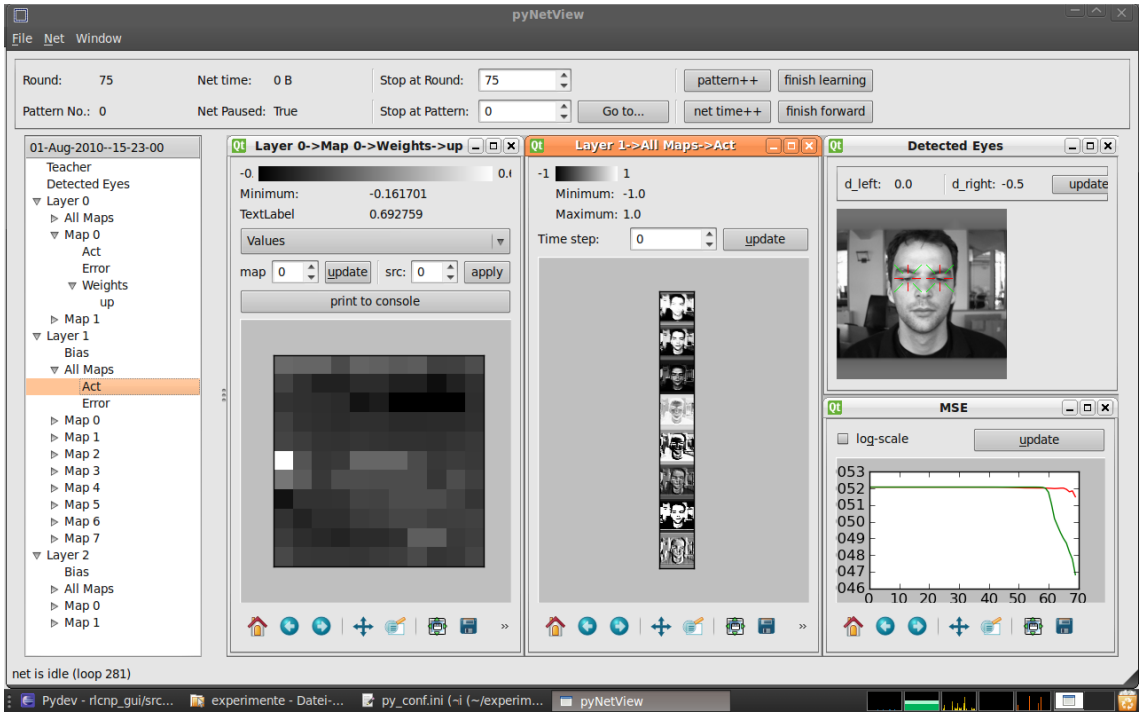


Abbildung 4.7: Die grafische Benutzeroberfläche, mit der die Pyramide überwacht werden kann.

Die zweite Erweiterung ist die Funktion `calc_error_blob()`. Ihre Aufgabe ist es für einen gegebenen (Teacher-)Punkt P_t und eine gebene Aktivierungskarte A eine Ausgabekarte zu berechnen. In dieser Ausgabekarte ist für jedem Pixel (x, y) die Differenz zwischen einer Gaußglocke um den Punkt P_t und der Aktivierung $A(x, y)$ an der Stelle (x,y) gespeichert.

4.1.4 Die Grafische Benutzeroberfläche

Um den Zustand des Netzes zu visualisieren wurde eine grafische Benutzeroberfläche implementiert, die im Folgenden nach dem englischen Fachterminus *Graphic User Interface* kuz GUI genannt wird. Abbildung 4.7 zeigt die GUI beim Training eines Netzes. Die GUI wurde mit Hilfe der Bibliotheken *PyQt* [Com10], der Python-Bibliothek zum Zugirff auf das Qt-Framework [Nok10], und *Python Remote Objecs – PYRO* [dJ] umgesetzt. Letztere ermöglicht es, das neuronale Netz und die GUI jeweils in einem eigenen Thread auszuführen. Dazu nutzt Pyro einen Pyro-Server, der Client-Anfragen von PYRO-Client-Objekten entgegennimmt und beantwortet. Die GUI muss daher nach dem Aufruf erst mit dem laufenden Pyro-Server, der in seinem Thread auf ein laufendes Netzobjekt zugreifen kann, verbunden werden.

In der GUI können alle wesentlichen Daten über den Zustand des Netzes angezeigt werden. Dazu zählen:

Die Struktur des Netzes ist in der Baumansicht abgebildet. Durch Aufklappen der Baumobjekte können die jeweils darunter liegenden Objekte sichtbar gemacht werden. Sofern es eine Ansicht zu den Baumobekten gibt, wird diese Ansicht in der Multiple-Document-Area geöffnet.

Die Aktivierungen und Fehler der Karten sind als Ansicht verfügbar. Klickt man sich in der Baumansicht zu einer Karte durch, werden die entsprechenden Baumobjekte für Aktivierung und Fehler darunter sichtbar. Ein Klick darauf öffnet ein Fenster, in dem die gewählte Karte im gewählten Zeitschritt visualisiert wird. Die Ausgabe ist vom kleinsten Aktivierungswert (schwarz) bis zum größten Aktivierungswert (weiß) skaliert. Über die Schaltfläche *update* wird die Anzeige aktualisiert.

Die Gewichte lassen sich ebenfalls in einer eigenen Ansicht öffnen. Das Fenster visualisiert wie oben zwischen dem niedrigsten Wert (schwarz) und dem größten Wert (weiß) jede Filterposition. Das Drop-Down-Feld bietet die Möglichkeit die Anzeige zwischen den aktuellen Gewichtswerten, dem Gradienten, den Vorzeichen der letzten Gradienten (interessant für den RPROP-Algorithmus) und den aktuellen Lernraten hin und her zu wechseln. Die Schaltfläche *update* aktualisiert die Ansicht mit den aktuell gewählten Einstellungen. Um das Ergebnis der Faltung der Quellkarte eines Filters mit den aktuellen Gewichtswerten isoliert zu betrachten – also auch vor Anwendung der Transferfunktion – kann über die Schaltfläche *apply* die Faltung durchgeführt und das Ergebnis angezeit werden.

Der mittlere Quadratische Fehler und auch der durchschnittliche Lokalisierungsfehler (d_eye) sind über das Menü erreichbar: *Net->Show MSE*. In dem Zeichenfenster werden die Daten, die das Netz in die Datei `mse.log` im Verzeichnis des Experimentes schreibt, visualisiert. Über die Auswahlfelder kann die Darstellung von MSE oder Lokalisierungsfehler ausgeschaltet werden.

Die Visualisierung der Lokalisierung kann in der Baumansicht angewählt werden. Nach einem Klick auf das Baumobjekt öffnet sich ein Fenster, in dem für das aktuell verarbeitete Muster der Lokalisierungsfehler für das rechte und linke Auge angegeben sind. Die Werte entsprechen der Distanz zwischen dem lokalisierten Zentrum des Auges und dem tatsächlichen Zentrum. Sie werden durch die euklidische Norm ermittelt. Die tatsächlichen Positionswerte sind jeweils durch ein rotes Kreuz mit senkrechten und waagerechten Balken markiert. Durch grüne Kreuze mit

schrägen Balken werden die lokalisierten Augenpositionen markiert. Ein Klick auf die Schaltfläche *update* aktualisiert die Ansicht.

In der oberen Informationsleiste werden Daten zum aktuellen Zustand des Netzes angezeigt. Über die Auswahlboxen *Stop at round* und *Stop at pattern* kann eingestellt werden in welcher Trainingsrunde (also welcher Epoche) und bei welchem Muster (Pattern) das Netz pausieren soll. Ein Klick auf die Schaltfläche *Go to...* sendet diese Information an das Netz.

4.2 Informationsverarbeitung in der Pyramide

Es wurden verschiedene Verknüpfungen zwischen Neuronenschichten mit je eigenen Aufgaben in der Informationsverarbeitung modelliert. Diese werden im Folgenden kurz mit ihrer Funktionsweise erläutert. Die Idee hinter der Kapselung der Verbindungen in eigenen Objekten ist es, diese Verbindungen austauschbar zu gestalten.

Im Vorwärtspass werden die Neuronenschichten in zeitlicher Abfolge neu berechnet, so dass Schichten mit kleinerem Zeitindex t zuerst berechnet werden. Wie bereits beschrieben sind die Neuronenschichten mit Axonschichten verbunden, die ihrerseits mit der nächsten Neuronenschicht verbunden sind. Der Aktivierungsanteil eines Layers kann daher berechnet werden, in dem die Funktion `getForwardActivities()` der Axonschichten aufgerufen wird. In dieser werden die jeweils notwendigen Berechnungen durchgeführt. Das Ergebnis ist eine Matrix, in der zeilenweise die Aktivierungsanteile für die Karten der Zielschicht stehen.

4.2.1 Neuronenschichten

Die Knotenschichten speichern die Aktivierungen der Neuronenkarten für jede Schicht. Zusätzlich speichern die Neuronenkarten im Trainingsmodus noch die Fehlersignale.

Eine Neuronenschicht besteht im zugrundeliegenden Modell nicht nur aus einer Menge von Neuronen, sondern ist in Merkmalskarten unterteilt. Die Merkmalskarten bestehen aus Zusammenhängenden Neuronen, deren Aktivität dann besonders hoch wird, wenn das Merkmal, welches sie vertreten, im zu verarbeitenden Bild vorkommt. Es sei an dieser Stelle darauf hingewiesen, dass die Struktur der Merkmale allerdings nicht in der Neuronenschicht, sondern in den Axonschichten gespeichert und gelernt wird, die zu der Karte hinführen. Eine hohe Aktivität eines Neurons an der Position (x, y) in einer Karte

einer Neuronenschicht deutet vielmehr daraufhin, dass ein Merkmal im rezeptiven Feld der Zelle (x, y) gefunden wurde. Vor allem wird hier also das Wissen darüber gespeichert, ob ein Merkmal an einer topologischen Stelle präsent war, oder nicht. Weiterhin muss bei der Interpretation beachtet werden, dass ein Neuron der Neuronenkarten über mehrere Merkmale integriert. Gibt es nicht nur eine Quellkarte, so tragen alle Merkmalsextraktoren zur Aktivierung bei. Eine Transferfunktion mit Sättigung sollte dann durch geeignete Skalierung der Filtergewichte dafür sorgen können, dass ein Neuron entweder eine *UND-Verknüpfung* oder eine *ODER-Verknüpfung* bildet. Für ein Vorhandensein aller Merkmale müssten dann relativ kleine Filterwerte und für nur eines oder weniger Merkmale müssten relativ große Werte gelernt werden.

In der vorliegenden Arbeit wurde ein Modell verwendet, welches aus den eigentlich zweidimensionalen Neuronenkarten ein eindimensionales Neuronenfeld macht. Diese Felder werden dann entlang der zweiten Dimension angeordnet. Besteht eine Schicht also beispielsweise aus zwei Karten $N_A^{l,0}$ und $N_A^{l,1}$ der Form

$$N_A^{l,0} = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6, \\ 7 & 8 & 9 \end{matrix}$$

und

$$N_A^{l,1} = \begin{matrix} a & b & c \\ d & e & f, \\ g & h & i \end{matrix}$$

so wird die Schicht als Matrix N_A^l abgespeichert:

$$N_A^l = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ a & b & c & d & e & f & g & h & i \end{pmatrix}$$

Die Neuronenschichten speichern die nicht nur ihre aktuelle Aktivierung, sondern auch das Fehlersignal eines Durchgangs in der gleichen Weise.

Behnke beschreibt in [Beh01], dass die Trainingsleistung verbessert werden kann, indem nicht nur am Ende der Trainingszeit sondern schon in den Zwischenschritten dem Netz ein Feedback gegeben wird. Da das primäre Ziel ist, dass nach T Zeitschritten der Zielzustand möglichst gut erreicht und nicht mehr verlassen wird, sollen frühe Zwischenimpulse weniger Einfluss auf das Lernen haben, als späte Trainingsimpulse. So werden die Gaußglocken zeitlich linear mit $0 < w_t <= 1$ gewichtet. Daher wurde in der Implementierung die Möglichkeit vorgesehen, dass Neuronenschichten zu jeder Zeit und in jeder

Schicht eine Teacher- oder Eingabe-Karte übergeben bekommen kann. Wird eine oder mehrere Eingaben im Vorwärtsschritt übergeben, so überschreibt diese die Werte aus den eingehnden Verbindungen. Werden Teacherkarten übergeben, so überschreiben diese den berechneten Fehler der

4.2.1.1 Der Vorwärtspass

In der Literatur – beispielsweise in [Bou06] – wird der Weg, um die Aktivierung einer Karte m_j in einer Schicht l in einem vorwärts gerichteten Konvolutionsnetz zu aktualisieren, wie folgt beschrieben. Die aktualisierte Aktivirung berechnet sich durch die Addition Aktivierungsbeiträge der eingehenden Schichten auf und wenden eine Transferfunktion auf die Summe an. In der in dieser Arbeit genutzten Notation sieht dieser Ablauf so aus:

$$N_{A,t+1}^{l,m_j} = f_{act}\left(\sum_{m_i \in M(l)} N_{A,t}^{l,m_i} * A_{C,W}^{l \to l+1, m_i \to m_j} + b_{m_j}^{l+1}\right) \tag{4.1}$$

Da die Neuronenschicht die Quellkarten nicht kennen, sondern die eingehenden Aktivitäten über die Axonschichten integieren müssen, wird auch der Ablauf aus Formel 4.1 modifiziert. Es wird nun nicht mehr über die Quellkarten summiert, sondern über alle Axonschichten, die eine Verbindung zu Schicht l haben:

$$N_{A,t+1}^{l,m_j} = f_{act}\left(\sum_{l_i : \exists : A_{C,W}^{l_i \to l+1}} N_{A,t}^{l,m_i} * A_{C,W}^{l_i \to l+1, m_i \to m_j} + b_{m_j}^{l+1}\right) \tag{4.2}$$

Diese Pull-Architektur wurde gewählt, da mehrere eingehende Axon-Schichten pro Neuronenschicht existieren und die Beiträge der eingehenden Schichten erst aufsummiert werden müssen. Weiter wurde verallgemeinert, dass die Art der Operation den Neuronenschichten nicht bekannt sein muss. So ist lediglich klar, dass die Operation, die eine Verbindungsschicht zur Schicht $l+1$ ausführt, eine Aktivierungs-Matrix $X(A^{l_i \to l+1, m_i \to m_j})$ ausgibt, die genauso groß ist, wie die Größe der Karte N_A^{l,m_j}. Das Fehlen des Index C, W deutet an, dass so auch andere Axonschichten, wie beispielsweise Poolingschichten, ohne Sonderbehandlung in den Aktualisierungsschritt eingebaut werden können. Abermals wird daher der Ablauf angepasst und Formel 4.2 ändert sich in

$$N_{A,t+1}^{l,m_j} = f_{act}\left(\sum_{l_i : \exists : A_{C,W}^{l_i \to l+1}} X\left(A^{l_i \to l+1, m_i \to m_j}\right) + b_{m_j}^{l+1}\right) \tag{4.3}$$

Der Bias wird im vorliegenden Modell durch eine eigene Axonschicht repräsentiert und implementiert (vgl. auch Abbildung 4.4). Allerdings hat die Axonschicht nur eine Verbindung: Die Quellkarten werden als imaginäre Karten mit der Aktivirung $N_A^{l,Bias}(i) = 1$ für alle $i \in \{0, \ldots, M(l+1) - 1\}$, wobei $M(l)$ wieder die Anzahl der Karten in Schicht l bezeichnet. Für jede dieser Eingabewerte existiert ein Biasgewicht, welches gelernt werden kann.

Die Transferfunktion ist im Modell f_{act} zu den Neuronenschichten assoziiert. Die Neuronenschichten sind mit mehreren eingehenden Schichten verbunden und fragen in einem Aktualisierungsschritt alle diese ab und addieren sie auf. Erst danach kann die Transferfunktion angewendet werden.

Bei den komplexeren Verbindungsschichten, die gleichzeitig eine Faltungs- und eine Poolingoperation realisieren findet die Anwendung der Tranferfunktion dabei auf den ersten Blick an der falschen Stelle statt: Bei der ursprünglichen Abfolge einer Faltungs- und einer Poolingschicht wird die Transferfunktion nach der Faltung angewandt und das Ergebnis wird durch Pooling verkleinert.

Die Anwendung der Transferfunktion nach dem Pooling ist mathematisch zulässig, sofern für die Transferfunktion gilt:

$$f_{act}(x) >= f_{act}(y) \Leftrightarrow x >= y$$

Für monoton steigende Funktionen, wie die unter Abschnitt 2.2.2 genannten, ist diese Bedingung erfüllt. Daher ist diese Anwendung zulässig.

4.2.1.2 Der Rückwärtspass

Der Rückwärtspass gliedert sich in zwei Phasen. Zunächst wird das Fehlersignal von dem oder den Ausgabeschichten in Richtung der Eingabe propagiert. Dazu werden die Fehlersignale der Schichten, zu denen die Neuronenschicht eine Verbindung hat aufsummiert. Dazu werden dann, falls es in der Schicht l eine Teacher-Eingabe gibt, der Teacher hinzuaddiert (vgl. Abschnitt 2.2.5.2) und anschliessend mit der Ableitung der Aktivierungsfunktion multipliziert:

$$N_{E,t-1}^{l,m_j} = f'_{act}(net_{m_j}) \cdot \sum_{m_i \in M(l)} N_{E,t+1}^{l,m_i} + t_{m_j} \tag{4.4}$$

Dieses Fehlersignal wird iterativ vom letzten Zeitschritt bis zum ersten Zeitschritt durch das Netz propagiert. Anschliessend müssen für alle Gewichtsobjekte die Gradienten berechnet werden. Dieser Schritt ist in den jeweiligen Schichten genauer dargestellt. Für die

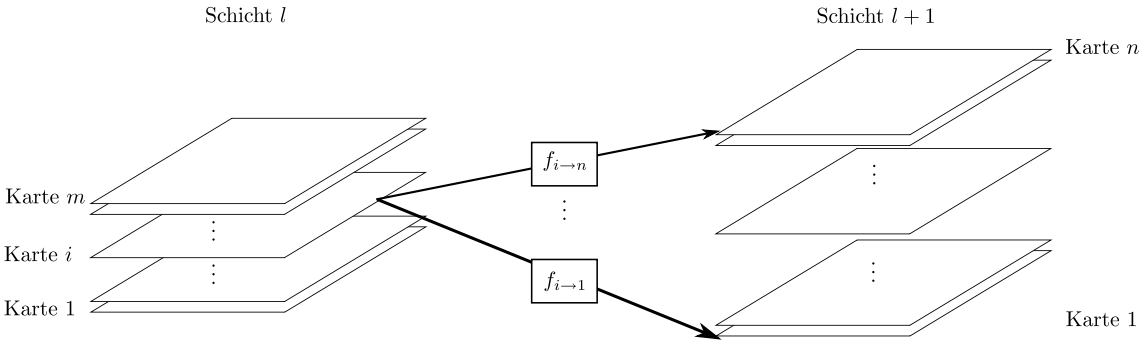


Abbildung 4.8: Es existiert für jeden der m Karten von Schicht l jeweils ein eigener Filter für jede der n Karten in Schicht $l+1$

Funktionsweise der zweiten Phase in Konvolutionsschichten siehe Abschnitt 4.2.2.2 und für vollverknüpfte Schichten siehe Abschnitt 4.2.4.3.

4.2.2 Konvolutionsschichten

Die Faltungsschichten speichern die topologische Information, welche Schichten miteinander verbunden sind. Sie haben also genau eine Quellschicht und eine Zielschicht. Zudem speichern die Faltungsschicht einen Verweis zu dem Gewichtsobjekt, welches die Filtergewichte speichert. Dieses Objekt speichert die Gewichte, die Gewichtsgradienten, die Vorzeichen der alten Gewichtsgradienten und die Lernraten für jedes Gewicht. Die letzten beiden Parameter werden gebraucht, falls der RPROP-Trainingsalgorithus angewendet werden soll. Die Faltung wird mit Hilfe der in Abschnitt 4.1.3.2 beschriebenen Faltungsfunktionen aus der CUV-Bibliothek realisiert. Diese legen die Speicherstruktur der Filter nahe.

Ist N_A^l eine die Aktivierungsmatrix aus der Quell-Schicht l, wobei die Schicht $m_l = M$ Karten habe. Jede der quadratischen Karte habe N Neurone bei einer Seitenlänge von $\sqrt{n}$. Dann hat N_A^l, wie in Abschnitt 4.2.1 die Form

$$N_A^l = \begin{matrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \dots & a_{2N} \end{matrix}$$

In jeder der M Zeilen wird je eine quaratische Karte mit insgesamt N Neuronen gespeichert. Wie in Abbildung 4.8 gezeigt ist jede Karte der Quellschicht mit allen Karten der Zielschicht k verbunden. Für jede dieser Verbindungen wird eine Konvolution mit einem eigenen Filter durchgeführt. Die Zielschicht liegt entweder über oder unter der

Quellschicht oder die Zielschicht ist gleich der Quellschicht. Daher ist $k \in \{l-1, l, l+1\}$. Die Zielschicht hat entsprechend $m_{l-1} = \frac{1}{2} \cdot M$, $m_{l+1} = 2 \cdot M$ oder $m_l = M$ Karten hat.

Die Zielschicht habe nun m_k Filter, welche von jeder Karte der Quellschicht auf eine Karte der Zielschicht falten. Es bezeichne nun f_{ij} den Filter von Karte i in Schicht l und Karte j in Schicht k. Alle Filtergewichte $w_x, x \in \{0, 1, \ldots, fs^2\}$ in f_{ij} sind, wie die Karten, ebenfalls in einer Dimension als Zeile angeordnet:

$$f^{ij} = \begin{array}{cccc} f_1^{ij} & f_2^{ij} & \ldots & f_{fs^2}^{ij} \end{array}$$

Diese Filter für eine Quellkarte werden untereinaner angeordnet. Je Karte, also je Zeile in der Quellkarten-Matrix, gibt es so viele Filter, also Zeilen wie in der Zielschicht Karten vorhanden sind.cDa die einzelnen Filter die Größe fs^2 haben hat die Filtermatrix die Dimensionen $m_k \times fs^2 \cdot m_l$. Sei $fs = 2$, so hat die Filtermatrix folgende Einträge beispielsweise aus 2×2-Filtern, dann sie die Matrix wie folgt aus:

$$A_W^{l \to l+1} = \begin{array}{cccccccccc} f_0^{11} & f_1^{11} & f_2^{11} & f_3^{11} & f_0^{21} & \ldots & f_3^{21} & \ldots & f^{m_l 1}1_0 & \ldots & f_3^{m_l 1} \\ f_0^{12} & f_1^{12} & f_2^{12} & f_3^{12} & f_0^{22} & \ldots & f_3^{22} & \ldots & f_0^{m_l 2} & \ldots & f_3^{m_l 2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ f_0^{1m_k} & f_1^{1m_k} & f_2^{1m_k} & f_3^{1m_k} & f_0^{2m_k} & \ldots & f_3^{2m_k} & \ldots & f_0^{m_l m_k} & \ldots & f_3^{m_l m_k} \end{array} \tag{4.5}$$

Dabei beschreibt f_h^{ij} das Filterpixel h des Filters von Karte i in der Quellschicht zu Karte j in der Zielschicht. Zur Vereinfachung werden im Folgenden aber nur noch die Filter geschrieben:

$$A_W^{l \to k} = \begin{array}{cccc} f_{1 \to 1} & f_{2 \to 1} & \ldots & f_{m_l \to 1} \\ f_{1 \to 2} & f_{2 \to 2} & \ldots & f_{m_l \to 2} \\ \vdots & \vdots & \ddots & \vdots \\ f_{1 \to m_k} & f_{2 \to m_k} & \ldots & f_{m_l \to m_k} \end{array}$$

Dabei deuten die Indizes $i \to j$ an, dass der Filter von Karte i in der Quellschicht zu Karte j in der Zielschicht gemeint ist.

Die Initialisierung der Filter erfolgt, wie in [LBOM98] empfohlen, in Abhängigkeit von der Anzahl der beim Zielneuron eingehenden Verbindungen. Der Initialisierungswert f_w ergibt sich hier aus:

$$f_w = 1/\sqrt{3 \cdot \sum_{C \in A(x,l)} fs(A^{C \to l})^2},$$

wobei $fs(A^{C\to l})$ die Größe des Filters der Verbindung von Schicht C zur Schicht l angibt. Die Menge $A(x,l) = \{A^{x\to l+1} | xin\mathbb{R} : \exists A^{x\to l}\}$ enthält alle eingehenden Verbindungen der Zielschicht l.

4.2.2.1 Der Vorwärtspass

Eine Konvolutionsschicht faltet die Eingabe mit einem gelernten Filter. Um eine Karte $N_{A,t}^{l,m}$ mit einem Filter $A_W^{l\to l+1}$ zu falten, wird der Filter über alle möglichen Positionen der Quellkarte i geschoben:

$$N_{A,t}^{l,j} = f_{act}(N_{A,t-q}^{l-1,i} * f_{i\to j}^{l} + b_j^l) \tag{4.6}$$

Dabei steht der Stern-Operator für die gewöhnliche zweidimensionale Faltung. Allerdings sind die Zielkarten nicht, wie nur mit einer Quellkarte verbunden. Vielmehr ist im Modell jede Zielkarte mit allen m_k Karten der Quellschicht verbunden. Daher müssen die Faltungsergebnisse noch aufaddiert werden:

$$N_{A,t}^{l,m} = f_{act}(\sum_{i\in M_j} a_i^{l-1} * f_{i\to j}^{l} + b_j^l) \tag{4.7}$$

In Abbildung 4.9 ist der algorithmische Ablauf des Vorwärtspass dargestellt. In `self.A` liegen die Aktivierungen der Quellschicht wie bereits erläutert in Zeilen. In `self.W` liegen die Filter, wie oben erläutert. Die Funktion `convolve2()` aus der CUV-Bibliothek faltet nun jede Karte i mit allen zugehörigen Filtern $f_{i\to j}, j \in \{0,1,\ldots,m_k\}$ und legt die Ergebnisse $f_{1\to j}*$ Karte 1 hintereinander in der Ergebnismatrix ab (vgl. Abschnitt 4.1.3.2. Andersherum betrachtet stellt $f_{i\to j}*$ Karte i den Beitrag der Faltung von Karte i der Quellschicht zu Karte j der Zielschicht. Diese Beiträge müssen noch aufsummiert werden. Auch dieser Schritt lässt sich mit der CUV-Bibliothek einfach implementieren. Die Ergebniskarten der Faltungsoperationen sind allerdings um den Filterradius kleiner, als die Ursprungsbilder, da der Filter hier bündig mit dem Bild angelegt wird. Um diesen unerwünschten Nebeneffekt zu vermeiden werden die Eingabekarten um den Filterrand an allen Rändern erweitert. Die neuen Pixel erhalten die Aktivität null, so dass diese keinen Beitrag zur Filtersumme leisten.

Die Ergebniskarten sind ebenfalls als Zeilenvektor abgelegt. Daher enthält eine Spalte alle Aktivierungsbeiträge von den gleichen Pixelposition von unterschiedlichen Karten der Quellschicht. Die Funktion `reduce_to_row()` summiert alle Elemente einer Spalte auf

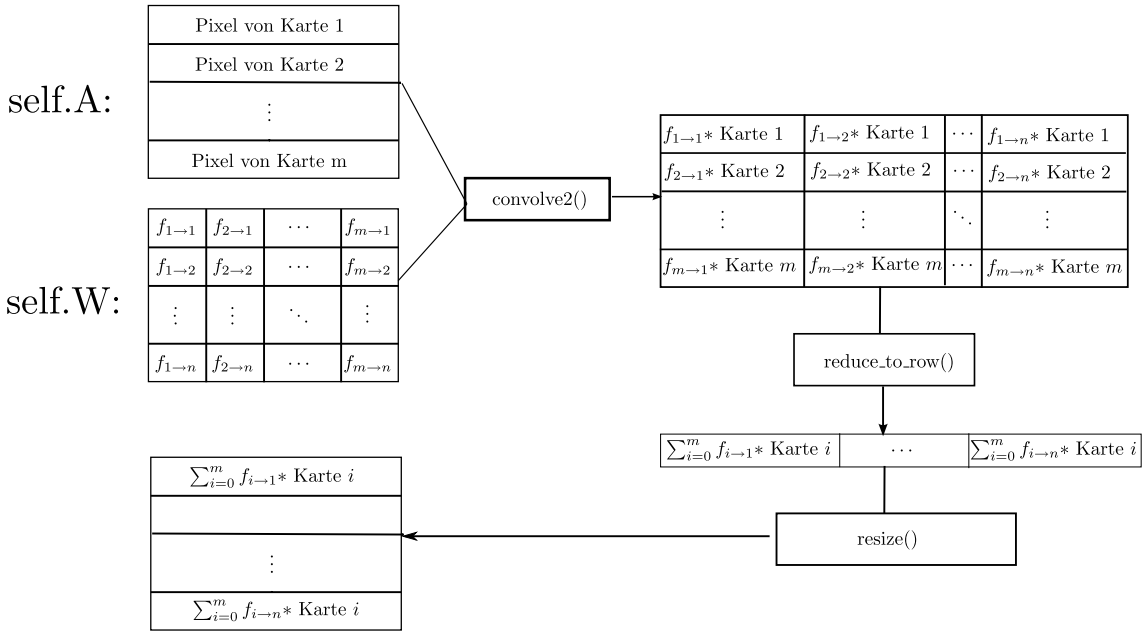


Abbildung 4.9: Schematische Darstellung des algorithmischen Ablaufes des Vorwärtspasses

und speichert das Ergebnis als Zeilenvektor. In diesem Zeilenvektor stehen nun hintereinander alle Karten der Zielschicht. Mit der CUV-Funktion `resize()` wird schließlich die Form der Matrix so geändert, dass die Karten untereinander stehen. Dieser Schritt benötigt fast keine Rechenzeit, da die Matrizen intern in einem langen Zeilenvektor abgelegt werden und zwei Variablen die Höhe und Breite der Matrix speichern. Die CUV-Funktion `resize()` muss nur diese Variablen ändern.

4.2.2.2 Rückpropagierung des Fehlersignals

Die Rückpropagierung des Fehlersignals kann ebenfalls als Konvolution aufgeschrieben und implementiert werden. Dies wird im Folgenden motiviert. Anschliessend wird beschrieben, wie der Rückwärtspass mit der CUV-Bibliothek umgesetzt werden kann.

Um die Veränderung der Gewichte des Netzes zu berechnen wird, wie in Abschnitt 2.2.5 beschrieben, die Delta-Regel für mehrschichtige Neuronale Netze verwendet:

$$\Delta w_{ij} = \nu \cdot a_i \cdot \delta_j \tag{4.8}$$

Hierbei bezeichnet w_{ij} das Gewicht von einem Neuron i in Schicht l zu einem Neuron j in Schicht $l+1$. Um dieses anzupassen wird die Aktivität des Quellneurons a_i mit dem Fehler des Zielneurons δ_j multipliziert. Es ist also erforderlich den Fehler für jede Schicht zu

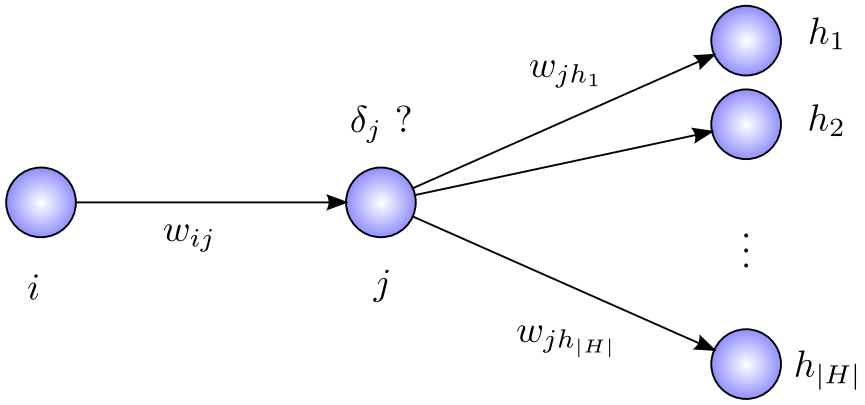


Abbildung 4.10: Visualisierung der Parameter in Formel 4.9

bestimmen, damit Gewichte der Verbindungsschichten aktualisiert werden können. Dazu werden in der normalen Backpropagation-Formel vorausgesetzt, dass die Fehler für alle Neurone $h \in H$ der übernächste Schicht $l+1$ bekannt sind. Wenn das der Fall ist, dann kann das δ_j aus Formel 4.8 berechnet werden, in dem die Ableitung der Aktivierung des Zielneurons in Schicht $l+1$ mit der Summe der gewichteten Fehler aller Zielschichten der Zielschicht multipliziert wird:

$$\delta_j = \frac{\partial f_{act}}{\partial w_{ij}} \sum_{k \in \{0,1,\ldots,|H|\}} w_{jk} \cdot \delta_{h_k} \tag{4.9}$$

Dabei bezeichnet H die Menge aller Neurone in Schicht $l+1$ mit denen das Neuron j verbunden ist. Andersherum betrachtet, summiert man die gewichteten Fehler der Neurone auf, mit denen das Gewicht j verbunden ist. Abbildung 4.10 visualisiert dies am Beispiel eines 3×3-Filters in einem Konvolutionsnetz. Man sieht dort, dass das Neuron zum ersten Mal durch den Filter abgetastet wird, wenn das Neuron an der Stelle $(x - 2 \cdot \lfloor \frac{1}{2} fs \rfloor, y - 2 \cdot \lfloor \frac{1}{2} fs \rfloor)$[3] der Zielschicht berechnet werden soll. Das Neuron j wird im weiteren Verlauf mit jedem Filtergewicht einmal multipliziert – jedes Mal zur Berechnung eines anderen Zielpixels.

Das Neuron j bekommt also die Fehlersignale von fs^2 vielen Neuronen der Zielkarte. Dies entspricht einer Faltung des Fehlerkarte des Zielbereiches mit dem Filter, der die beiden Schichten im Vorwärtspass verbindet. Allerdings muss dieser um 180 Grad rotiert werden. Da das Neuron j bei der ersten Abtastung mit dem letzten Filtergewicht multipliziert wurde, muss dieses letzte Filterelement nun mit dem Fehler an der Stelle $(x-2\cdot\lfloor \frac{1}{2} fs \rfloor, y-2\cdot\lfloor \frac{1}{2} fs \rfloor)$ der Zielkarte multipliziert werden. Bei der letzten Abtastung hingegen wurde das Neuron

[3] Die Filtergröße muss in diesem Fall zwei Mal abgezogen werden: einmal für den Abstand zum Pixel, an dem sich das Zentrum befindet, wenn j zum ersten mal abgetastet wird. Da die Zielkarte bereits um $\lfloor \frac{1}{2} fs \rfloor$ an allen Rändern geschrumpft wurde, muss dieser Betrag noch einmal abgezogen werden.

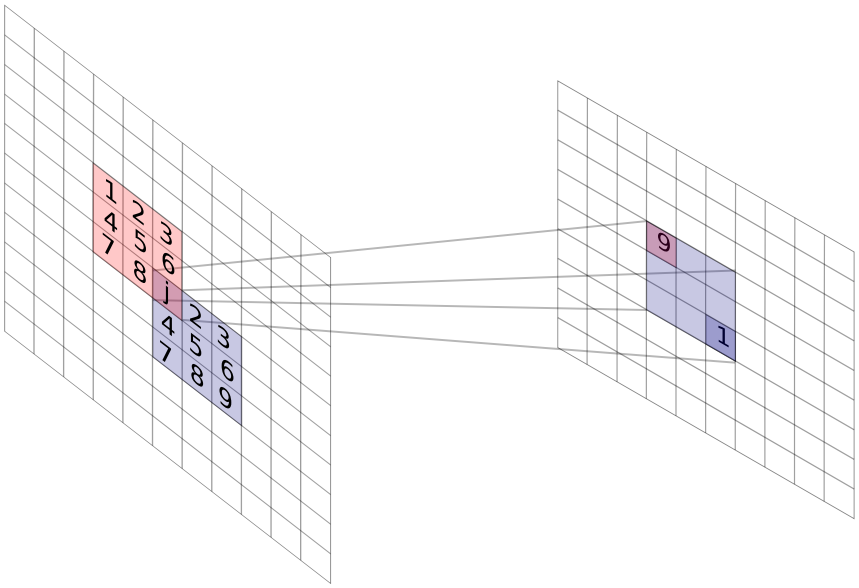


Abbildung 4.11: Visualisierung des Zusammenhangs zwischen Fehlersignal und Propagationsziel. Um den Fehler von Neuron j an der Stelle (x, y) zu bestimmen, werden die gewichteten Fehler aus der Zielkarte aufsummiert. Der schattierte Bereich in der rechten Karte deutet an, welche Neurone durch den 3×3-Konvolutionsfilter mit Neuron j verbunden sind. Zudem zeigt die Abbildung auch, welche Gewichte Neuron j mit den Neuronen in der Zielkarte verbinden. Diese Beobachtung zeigt, dass der Filter um 180 rotiert werden muss, wenn er in der Konvolution im Rückwärtspass verwendet werden soll.

mit dem ersten Filterelement multipliziert, wie Abbildung 4.11 zeigt, und muss nun im Rückwärtspass mit diesem multipliziert werden.

Die algorithmische Umsetzung unter Verwendung der CUV Bibliothek ist in Abbildung 4.12 visualisiert. Nicht dargestellt wurde der Schritt, in dem die Auffüllung des Randes um den Filterradius vorgenommen wurde. Auch hier sind die Karten an allen Rändern um den Filterradius erweitert und mit Null aufgefüllt. Der schematische Aufbau beschreibt, wie die Daten im Speicher organisiert sind und durch welche Funktionsaufrufe sie modifiziert werden. Auf die Funktionsignaturen wurde zur Vereinfachung verzichtet.

In `self.W.` sind die Filter – wie oben erläutert – abgespeichert. Im Rahmen dieser Arbeit wurde der CUV-Bibliothek die Funktion `filters_rotate()` hinzugefügrt, welche

die Filter rotiert. Diese Funktion bekommt als Patameter eine Gewichtsmatrix, wie die in Formel 4.5 angegebene

$$A_W^{l\to l+1} = \begin{matrix} f_0^{11} & f_1^{11} & f_2^{11} & f_3^{11} & f_0^{21} & \dots & f_3^{21} & \dots & f^{m_l 1}1_0 & \dots & f_3^{m_l 1} \\ f_0^{12} & f_1^{12} & f_2^{12} & f_3^{12} & f_0^{22} & \dots & f_3^{22} & \dots & f_0^{m_l 2} & \dots & f_3^{m_l 2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ f_0^{1m_k} & f_1^{1m_k} & f_2^{1m_k} & f_3^{1m_k} & f_0^{2m_k} & \dots & f_3^{2m_k} & \dots & f_0^{m_l m_k} & \dots & f_3^{m_l m_k} \end{matrix} \tag{4.10}$$

und rotiert die Filter, in dem sie die Reihenfolge der Filterelemente umkehrt:

$$A_{W,rot}^{l\to l+1} = \begin{matrix} f_3^{11} & f_2^{11} & f_1^{11} & f_0^{11} & f_3^{21} & \dots & f_0^{21} & \dots & f^{m_l 1}1_3 & \dots & f_0^{m_l 1} \\ f_3^{12} & f_2^{12} & f_1^{12} & f_0^{12} & f_3^{22} & \dots & f_0^{22} & \dots & f_3^{m_l 2} & \dots & f_0^{m_l 2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ f_3^{1m_k} & f_2^{1m_k} & f_1^{1m_k} & f_0^{1m_k} & f_3^{2m_k} & \dots & f_0^{2m_k} & \dots & f_3^{m_l m_k} & \dots & f_0^{m_l m_k} \end{matrix} \tag{4.11}$$

Um die Konvolution von den Fehlerkarten der Zielschichten auf die Quellschichten umsetzen zu können muss die Anordnung der Filter in der Filtermatrix noch transponiert werden. Dies ist notwendig, da die `convolve2()`-Funktion der CUV Bibliothek alle Filter für die erste Karte untereinander erwartet. Da nun eine Konvolution von der ursprünglichen Ziel- auf die ursprüngliche Quellschicht vorgenommen werden soll, müssen die Filter umsortiert werden. In der schematischen Zeichnung in Abbildung 4.12 kann man dies an der Anordnung der rotierten Filter in der Matrix nach Ausführung der Funktion `convolve2()`[4] erkennen. Um die Filter umzusortieren werden zwei CUV-Funktionen hintereinander ausgeführt. Die Funktion `reorder()` sortiert die Filter alle, wie in Abbildung 4.12 angezeigt untereinander, in dem sie eine Matrix von der Breite fs^2 erstellt und dort nacheinander erst die Filter der ersten Karte, dann die der zweiten Karte und so weiter ablegt. Nach der Ausführung dieser Funktion stehen also alle Filter in der richtigen Reihenfolge untereinander in der Matrix. Wieder reicht es, da die Filter intern in einem fortlaufenen eindimensionalen Speicher abgelegt sind, die Breite der Matrix mittels der Funktion `resize()` anzupassen.

Nun kann durch eine Faltung der Fehlerkarten mit den rotierten Fehlern der Zielschicht der Beitrag jeder Fehlerkarte zu den Fehlern der Quellkarte berechnet werden. Das Ergebnis jeder Faltung ist, wie auch schon beim Vorwärtspass erläutert, nacheinander in der Ergebnismatrix abgelegt. Auch hier sind wieder die Pixel aller Quellkarten untereinander angeordnet. Daher kann mit der Summation aller Zeilen `reduce_to_row()` ein Zeilen-

[4] Zur Erinnerung: Im Index der Filter bezeichnet $i \to j, i \in \{0, \dots, m_l\}, j \in \{0, \dots, m_{l+1}\}$, dass der Filter von der Karte i in der Quellschicht zu Karte j in der Zielschicht führt. Da an dieser Stelle die gleichen Filter gemeint sind, dreht sich diese Beziehung im Rückwärtspass nicht um.

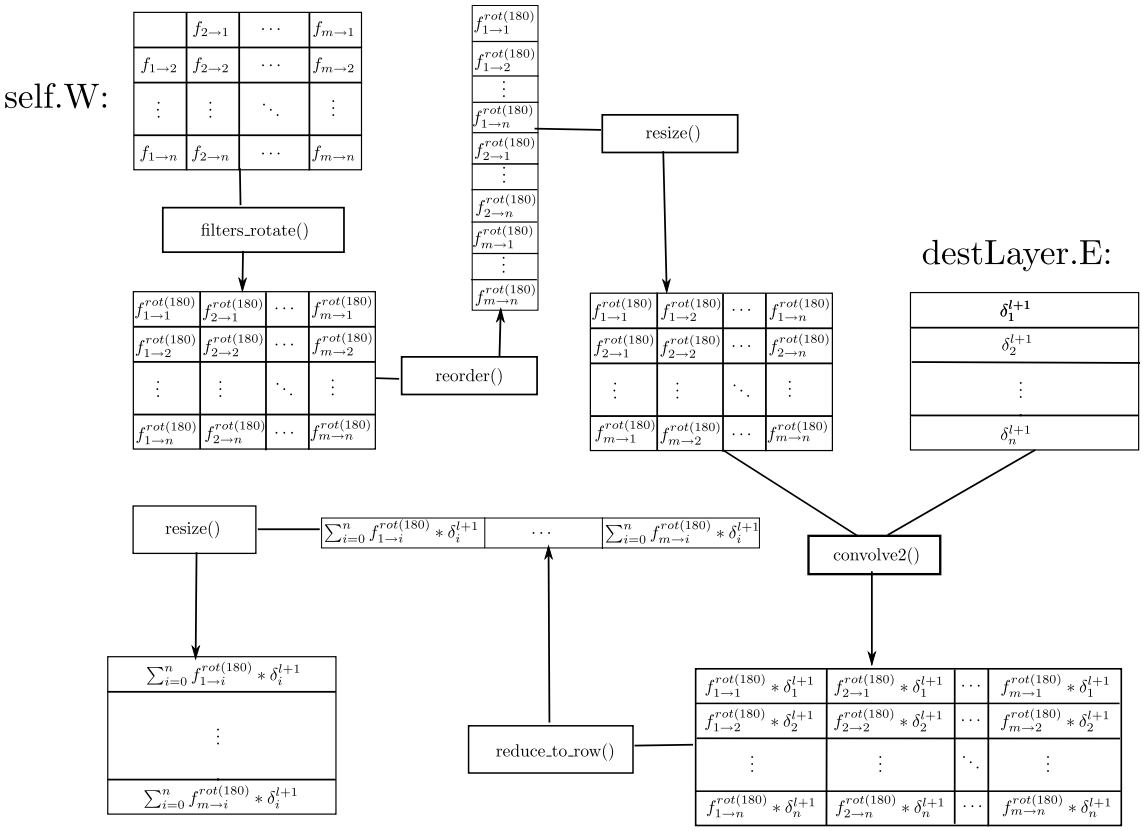


Abbildung 4.12: Schematische Darstellung des algorithmischen Ablaufes des Rückwärtspasses

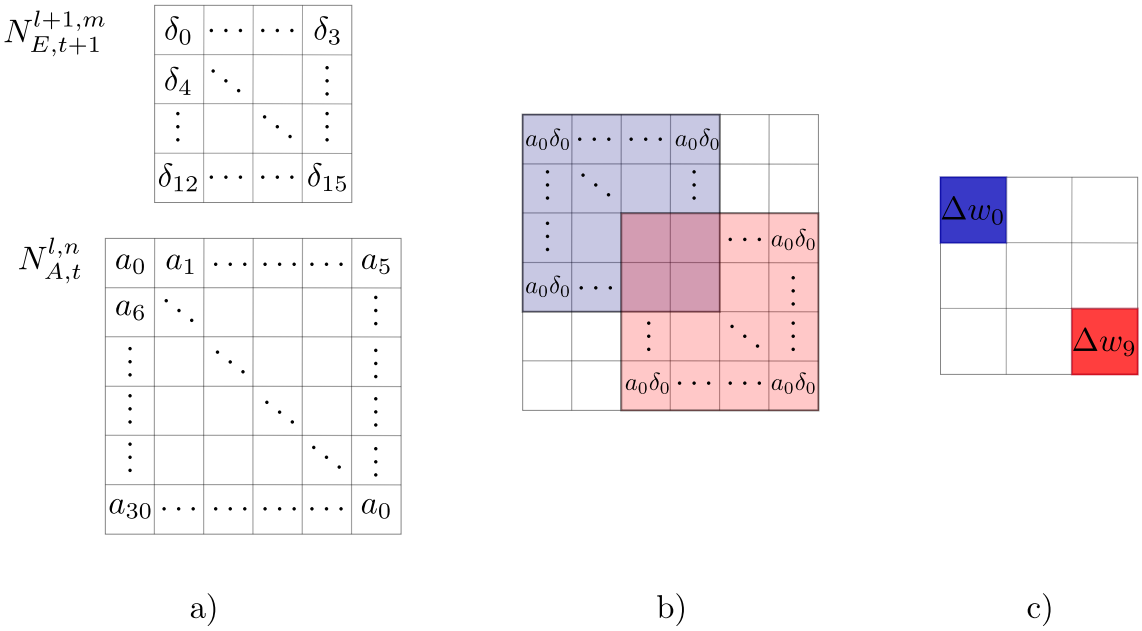


Abbildung 4.13: Die Berechnung des Gewichtsgradienten für einen 3×3-Filter $f_{n \to m}$ durch eine Faltung einer 6×6-Aktivierungskarte und einer 4×4-Fehlerkarte der Zielschicht. In a) sind oben die Fehlerkarte und unten die Aktivitäten dargestellt. Diese werden, wie in b) dargestellt miteinander gefaltet. Das Ergebis sind die in c) dargestellten Gewichtsgradienten für die Filtergewichte von $f_{n \to m}$. Um das blaue Matrix-Element in c) zu berechnen werden alle Produkte aus dem blauen Bereich in b) aufsummiert.

vektor erzeugt werden, in dem die Fehlerkarten hintereinander abgelegt sind. Auch hier reicht es, die Matrix abschliessend mit `resize()` auf die richtige Größe zu bringen.

4.2.2.3 Berechnung des Gewichtsgradienten

Die Anwendung des Gewichtsgradienten im Lernverfahren übernimmt das Gewichtsobjekt. Aber zunächst ist es erforderlich, dass der Gradient über alle Zeitschritte aufsummiert wird. Dazu werden alle Verbindungsschichten aufgerufen, ihren Beitrag auf ihr Gewichtsobjekt aufzusummieren.

Um einen Gradienten $\Delta w_i j$ zu berechnen werden die Aktivierungen der Quellschicht a_i mit dem Fehler des Neurons in der Zielschicht δ_j multipliziert:

$$\Delta w_{ij} = -\eta \cdot a_i \cdot \delta_j \tag{4.12}$$

Im Konvolutionsnetz wird ein Filtergewicht an vielen Stellen der Quellkarte angewendet.

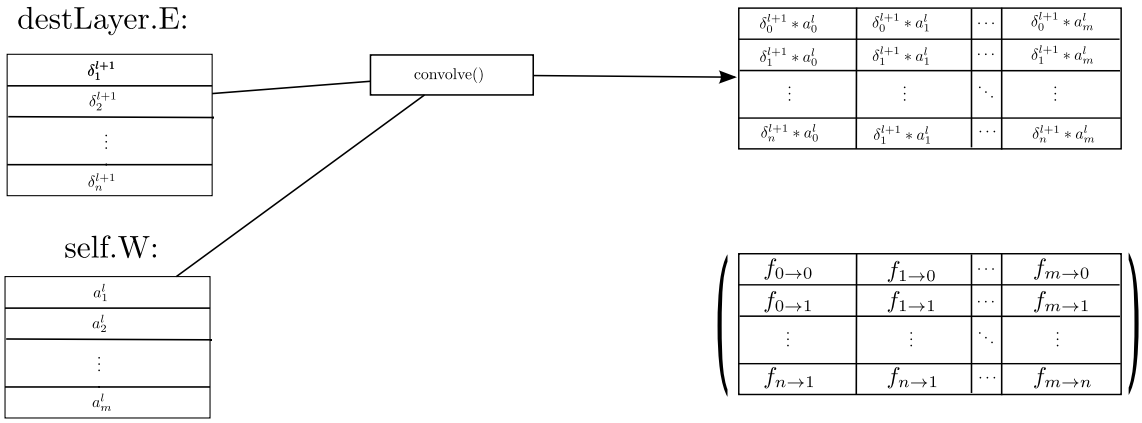


Abbildung 4.14: Schematische Darstellung des algorithmischen Ablaufes der Berechnung des Gewichtsgradienten. In Klammern ist zum Vergleich der Aufbau der Filtermatrix dargestellt.

Daher muss dieses Produkt über alle Anwendungsorte des Gewichtes aufsummiert werden und Formel 4.12 ändert sich in

$$\Delta w_{ij} = -\eta \sum_{i,j} a_i \cdot \delta_j \tag{4.13}$$

Glücklicherweise verbindet ein Filtergewicht auch hier wieder nur ein Neuron i mit einem Neuron j, so dass sich auch diese Summe als Konvolution implementieren lässt. Abbildung 4.13 illustriert diesen Vorgang anhand eines Beispiels mit einer Quell und einer Zielkarte. Wieder ist die Zielkarte um die halbe Filtergröße kleiner. Bei dieser Faltung, bei der die Aktivitäten mit dem Fehler der Zielschicht gefaltet wird, ist das Ergebnis genauso groß, wie der ursprüngliche Filter. Dabei werden automatisch alle Produkte aufsummiert, an denen das Filtergewicht im Vorwärtspass angewandt wurde. Somit lässt sich Formel 4.12 durch eine Faltung implementieren.

Auch diese Faltung wurde mit der CUV-Bibliothek umgesetzt. Die Aufgabe ist es, jede Aktivierungskarte mit allen Fehlerkarten zu falten. Würde diese Aufgabe mit der bereits mehrfach eingesetzten Funktion `convolve2()` so müssten die Fehlerkarten, die als Filter eingesetzt werden sollen, für jede Aktivierungskarte kopiert werden. Einfacher ist es, die Funktion `convolve()` zu verwenden. Diese faltet, wie in Abschnit 4.1.3.2 beschrieben, jedes der m Eingabekarten in der Bildeingabematrix mit den gleichen n Filtern. Abbildung 4.14 visualisiert das Ergebnis. Wie man sieht stehen in der Ergebnismatrix die Ergebnisse der Faltungen aller Aktivitäten mit der ersten Fehlerkarte hintereinander. Das entspricht den Gewichtsgradienten aller Filter, die zur ersten Karte der Zielschicht führen. Untereinander stehen alle Ergebnisse der Faltungen der ersten Aktivitätskarte mit

allen Fehlerkarten. Das entspricht den Gewichtsgradienten für die Filter der ersten Karte. Also sind auch die Gewichtsgradienten bereits an der richtigen Stelle.

Allerdings entspricht die Ergebnismatrix nur dem Beitrag eines Zeitschritts zum Gewichtsgradienten. Im rekurrenten Fall müssen diese Gradienten aufaddiert werden. Dies geschieht in dem das Gewichtsobjekt den Gradienten übergeben bekommt und aufaddiert.

4.2.3 Poolingschichten

In klassischen Modellen wurde die Reduzierung der Auflösung durch Subsamplingschichten erreicht (vgl. Abschnitt). Neuere Untersuchen von Scherer et al. [Sch09] und [BBLP10] zeigten allerdings, dass das Maxpooling zu schnellerer Konvergenz und besseren Ergebnissen führt, als Subsampling. Daher wurde von vornherein entschieden die Reduktion der Auflösung durch Maxpooling zu implementieren.

4.2.3.1 Der Vorwärtspass

Eine Maxpooling Verbindung $A_P^{l \to l+1}$ hat keine lernbaren Parameter. Daher ist der Vorwärtspass trivial: Die Eingabekarte wird in disjunkte Fenster der Größe $p \times p$, wobei p Pooling-Faktor heißt, eingeteilt. Aus jedem dieser Fenster wird ein Pixel der Zielschicht berechnet, in dem der maximale Wert aller Pixel in einem Fenster ausgewählt wird. Zusätzlich speichert die Schicht für jedes Fenster den Index, an dem das Maximum stand. Die Zielkarte des Pooling ist in beiden Dimensionen um den Faktor p kleiner, als die Quellkarte. In dieser Arbeit wird in Anlehnung an Behnkes Neuronale Abstraktionspyramide immer der Pooling-Faktor $p = 2$ verwendet. Abbildung 4.15 verdeutlicht diesen Vorgang.

Mit Hilfe der CUV-Funktion `max_pooling()` lassen sich mehrere Bilder nach dem beschriebenen Vorgehen poolen. Dieser Funktion kann auch eine Matrix von der Größe der Zielmatrix übergeben werden. In diese Matrix schreibt die Funktion Indices, welche bei der Propagierung des Fehlersignals noch benötigt werden.

4.2.3.2 Rückpropagierung des Fehlersignals

Da eine Maxpooling-Verbindung keine lernbaren Gewichte hat, wird hier auch keine Anpaasung der Gewichte möglich. Aufgabe der Maxpool-Schicht im Rückwärtspass ist es daher nur, das Fehlersignal auf die richtige Größe heraufzusamplen. Dazu werden wieder die im Vorwärtspass genutzten Fenster der Größe $p \times p$ herangezogen. Für jedes

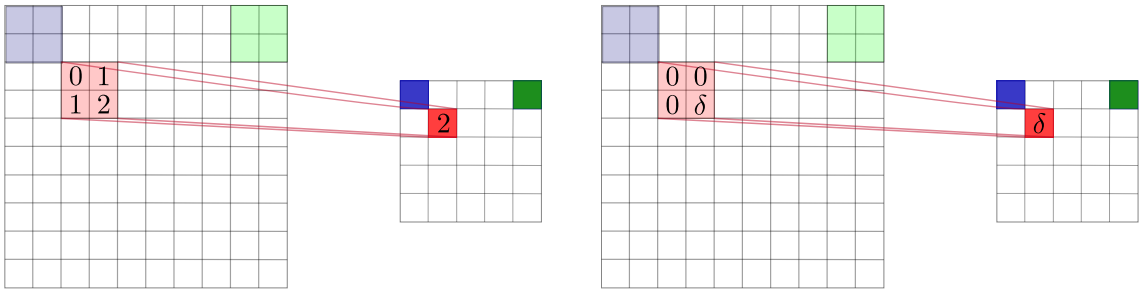


Abbildung 4.15: Das Maxpooling am Beispiel eines 10×10 Pixel großen Bildes und einem Pooling-Faktor $p = 2$. a) Im Vorwärtspass wird die maximale Aktivität übernommen und der Index, an dem diese auftrat, gespeichert. b) Rückpropagierung des Fehlersignals. Der Fehler von der Zielschicht gelangt an die Position, an der das Maximum im Vorwärspass stand.

dieser Fenster gibt es einen Pixel, dessen Fehlerwert zurück propagiert werden soll. Der Fehlerwert wird an den Pixel des Fensters geschrieben, dessen Index im Vorwärtspass für dieses Fenster gespeichert wurde. Die übrigen Positionen werden mit 0 aufgefüllt.

Dieser Vorgang lässt sich mit Hilfe der CUV-Funktion `super_to_max()` implementieren. Diese Funktion erwartet als Parameter die oben gespeicherte Matrix mit Pooling-Indices.

4.2.4 Vollverknüpfte Schichten

In den vollverknüpften Schichten funktioniert die Informationsverabreitung analog zum Modell des eines Perzeptrons. Das heißt jedes Neuron der unteren Schicht L ist mit jedem Neuron der höheren Schicht K verknüpft:

$$\forall l \in L, \forall k \in K : \exists w_{lk}$$

Die Gewichte w_{lk} werden in einer Gewichtsmatrix W mit den Dimensionen $|L| \times |K|$ gespeichert. Die Zeilenindizes $l \in \{0, \ldots, L-1\}$ bestehen aus den Indexnummern der Quellneurone. Die Spaltenindizes $k \in \{0, \ldots, K-1\}$ bestehen aus den Indexnummern der Zielneurone.

$$W = \begin{matrix} w_{11} & a_{21} & \ldots & a_{(K-1)1} \\ w_{12} & a_{22} & \ldots & a_{(K-1)2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1(L-1)} & w_{2(L-1)} & \ldots & w_{(K-1)(L-1)} \end{matrix}$$

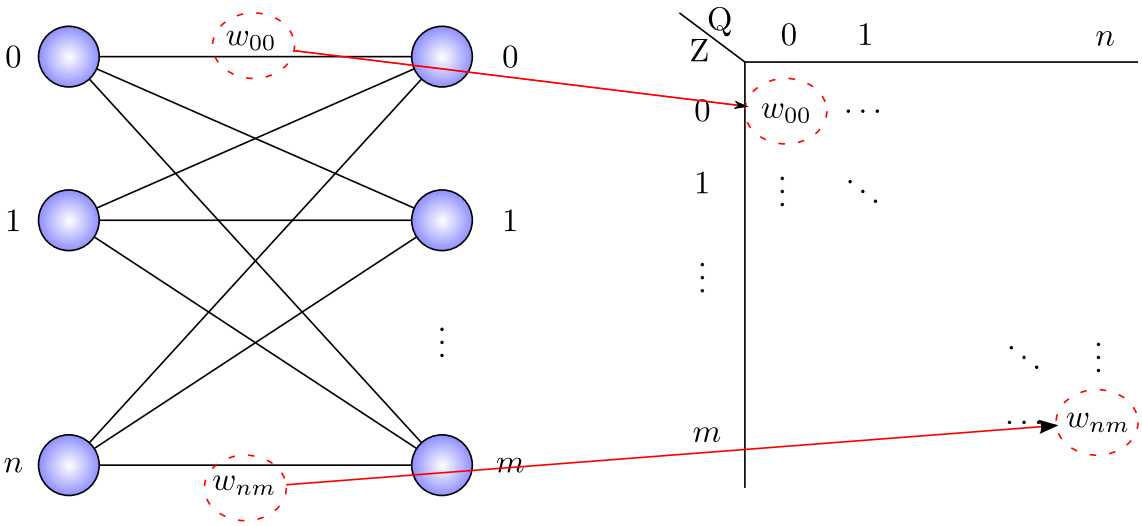


Abbildung 4.16: Speicherung der Gewichte in einer Gewichtsmatrix. Die Zeilen einer Matrix enthalten alle Gewichte, die von einem Quellneuron (Q) der Zeile zu einem Zielneuron gehen. Die Spalten enthalten alle Gewichte, die zu dem Zielneuron (Z) in der Spalte führen.

Abbildung 4.16 skizziert, wie die Gewichte aus dem Verbindungsgraphen in der Matrix angeordnet sind. Da es sich um eine Vollverknüpfung handelt spielt die Unterteilung der Neuronenschicht in Karten keine Rolle mehr und die Neuronen der Quellschicht werden als eine unstrukturierte Menge aufgefasst.

4.2.4.1 Der Vorwärtspass

Im Vorwärtspass wird die Aktivität a_j^{l+1} für ein Neuron j der Zielschicht N^{l+1} berechnet, in dem die Aktivitäten aller Neurone a_i^l der Quellschicht N^l multipliziert mit dem Gewicht von i nach j, also w_{ij}, aufsummiert und als Argument für eine Transferfunktion f_{act} verwendet werden:

$$a_j = f_{act}(net_j) = f_{act}(\sum_{a_i \in N_A^l} w_{ij} a_i)$$

Dabei ist vorteilhaft, dass die Summe $\sum_{a_i \in N_A^l} w_{ij} a_i$ sich bequem als Matrixmultiplikation schreiben lässt. Diese Matrixmultiplikation lässt sich dann leicht mit der CUV-Bibliothek auf der Grafikkarte beschleunigen. Sei nun (zur Vereinfachung der Schreibweise) die Zahl der Neurone der Zielschicht $L - 1 = N_{l+1}$ und die Zahl der Neurone der Quellschicht $K - 1 = N_l$.

Wenn nun die Gewichte zwischen Schicht N^l und N^{l+1} wie oben erläutert in einer Gewichtsmatrix

$$W_i^j = \begin{matrix} w_{11} & w_{21} & \dots & w_{N_l 1} \\ w_{12} & w_{22} & \dots & w_{2N_l} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1N_{l+1}} & w_{2N_{l+1}} & \dots & w_{N_l N_{l+1}} \end{matrix}$$

vorliegen und die Aktivitäten aller Neurone der unteren Schicht als Spaltenvektor a_l aufgeschrieben werden

$$a_l = \begin{matrix} a_1 \\ a_2 \\ \vdots \\ a_{N_l} \end{matrix}$$

, dann läßt sich die Netzeingabe net_j durch

$$net_j = W_i^{\cdot} a_l$$

berechnen.

$$net_j = \begin{pmatrix} w_{11} & w_{21} & \dots & w_{N_l 1} \\ w_{12} & w_{22} & \dots & w_{N_l 2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1N_{l+1}} & w_{2N_{l+1}} & \dots & w_{N_l N_{l+1}} \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{N_l} \end{pmatrix} = \begin{pmatrix} \sum a_i * w_{i1} \\ \sum a_i * w_{i2} \\ \vdots \\ \sum a_i * w_{iN_{l+1}} \end{pmatrix}$$

Nun muss noch punktweise die Aktivierungsfunktion angewandt werden und die Berechnung der Aktivität der nächsten Schicht ist berechnet:

$$N_A^{l+1} = f_{act} \begin{pmatrix} \sum a_i^l * w_{i1} \\ \sum a_i^l * w_{i2} \\ \vdots \\ \sum a_i^l * w_{iN_{l+1}} \end{pmatrix} = \begin{pmatrix} f_{act}(\sum a_i^l * w_{i1}) \\ f_{act}(\sum a_i^l * w_{i2}) \\ \vdots \\ f_{act}(\sum a_i^l * w_{iN_{l+1}}) \end{pmatrix} = \begin{pmatrix} a_0^{l+1} \\ a_1^{l+1} \\ \vdots \\ a_{N_{l+1}}^{l+1} \end{pmatrix}$$

Mit der CUV-Funktion `apply_binary_functor()` kann der Multiplikationsfunktor auf zwei Matrizen angewandt werden. Dieser soll dann die Gewichtsmatrix $A_{F,W}^{l \to l+1}$ mit den Aktivierungen $N_{A,t-1}^l$ multipliziert werden. Um die Aktivierungen aller Karten, die ja noch als Matrix vorliegen, in einen Spaltenvektor umzuwandeln, genügt es, auf den auf den internen Speicher zu zu greifen. Dieser ist als linearer Speicher angelegt und lässt sich direkt als CUV-Vektor ansprechen.

Mit der Funktion `apply_nullary_functor` kann ein Funktor punktweise auf alle Matrixelemente angewandt werden. So wendet die Verbindungsschicht den bei der Erzeugung der Instanz im Konstruktor übergebenen Funktor als Aktivierungsfunktion an.

4.2.4.2 Rückpropagierung des Fehlersignals

Im Rückwärtspass werden auch in den vollverknüpfen Schichten für alle Neuronen die Fehlersignale berechnet. Der eingehende Fehler am Neuron i im Rückwärtspas hängt wieder (vergleiche Abbildung 4.10) von der Summe der gewichteten Fehler der Zielschicht und der Ableitung der Aktivierungsfunktion des Neurons i ab.

Die Formel 4.9 muss allerdings um die Tatsache modifiziert werden, dass die Neuronen der Quellschicht nun mit allen Neuronen der Zielschicht verbunden sind. Also muss für jedes Neuron i der Quellschicht N^l die Summe über den Produkten aus dem Fehlersignal in der Zielschicht δ_j und dem Verbindungsgewicht gebildet werden:

$$\delta_i = f_{act}(net_i) \sum_{\delta_j \in N^{l+1}_{E,t+1}} w_{ij} \cdot \delta_j \tag{4.14}$$

Auch diese Operation lässt sich als Matrixoperation schreiben und mittels der CUV-Bibliothek beschleunigen. Dazu wird die transponie Gewichtsmatrix mit der als Spaltenvektor geschriebenen Fehlerkarte der Zielschicht multipliziert:

$$\begin{pmatrix} w_{11} & w_{12} & \dots & w_{1N_{l+1}} \\ w_{21} & w_{22} & \dots & w_{N_l 2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N_l 1} & w_{N_l 2} & \dots & w_{N_l N_{l+1}} \end{pmatrix} \cdot \begin{pmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_{N_{l+1}} \end{pmatrix} = \begin{pmatrix} \sum w_{1i} \cdot \delta_i \\ \sum w_{2i} \cdot \delta_i \\ \vdots \\ \sum w_{N_l i} \cdot \delta_i \end{pmatrix}$$

Die eingehenden Fehlersignale müssen anschliessend noch mit der Ableitung der Aktivierung multipliziert werden, um die Fehler für die Quellschicht N^l zu erhalten:

$$N^{l,t}_E = \begin{pmatrix} f_{act}(net_i) \cdot (\sum w_{1i} \cdot \delta_i) \\ f_{act}(net_i) \cdot (\sum w_{2i} \cdot \delta_i) \\ \vdots \\ f_{act}(net_i) \cdot (\sum w_{N_l i} \cdot \delta_i) \end{pmatrix} \tag{4.15}$$

In Abschnitt 4.2.4.1 wurde bereits erklärt, wie die Multiplikation umgesetzt werden kann. Um die Ableitung der Netzeingabe zu berechnen, können ebenfalls Funktoren punktweise

angewandt werden. Sollen die gängigen Funktoren, also die Ableitung des Tangens hyperbolicus oder der normalen Sigmoid-Funktion verwendet werden, so müssen zunächst die Aktivierungen in eine Hilfsvariable (also ebenfalls eine Matrix) kopiert werden, da die CUV-Implementierungen dieser Ableitungen die algorithmisch vereinfachten Varianten nutzen (vergleiche Abschnitt 2.2.2).

4.2.4.3 Berechnung des Gewichtsgradienten

Sind die Fehlersignal durch das Netz propagiert, dann aktualisieren die Gewichtsobjekte zunächst ihren Gradienten. Der Gewichtsgradient berechnet sich aus

$$\Delta w_{ij} = -\eta \cdot a_i \delta_j$$

Im letzten Abschnitt wurde besprochen, wie die Fehlersignale δ_j rekursiv berechnet werden können, so dass sie zum Zeitpunkt der Berechnung des Gewichtsgradienten bereits zur Verfügung stehen. Die Lernrate fügt die CUV-Bibliothek beim Lernvorgang ein, so dass hier nur eine Matrix aus Gradienten mit $\eta = -1$ berechnet werden muss, so dass

$$A_{\Delta w}^{l \to l+1} = \begin{pmatrix} \Delta w_{00} & \Delta w_{01} & \dots & \Delta w_{0N_{l+1}} \\ \Delta w_{10} & \Delta w_{11} & \dots & \Delta w_{1N_{l+1}} \\ \vdots & \dots & \ddots & \vdots \\ \Delta w_{N_l 0} & \Delta w_{N_l 1} & \dots & \Delta w_{N_l N_{l+1}} \end{pmatrix}$$

Diese Matrix lässt sich durch die geschickte Multiplikation der Aktivierusmatrix mit der Fehlermatrix erzeugen. Dazu werden beide Matrizen wider als Vektoren genutzt:

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N_l} \end{pmatrix} \cdot \begin{pmatrix} \delta_0 & \delta_1 & \vdots & \delta_{N_l} \end{pmatrix} = \begin{pmatrix} \Delta w_{00} & \Delta w_{01} & \dots & \Delta w_{0N_{l+1}} \\ \Delta w_{10} & \Delta w_{11} & \dots & \Delta w_{1N_{l+1}} \\ \vdots & \dots & \ddots & \vdots \\ \Delta w_{N_l 0} & \Delta w_{N_l 1} & \dots & \Delta w_{N_l N_{l+1}} \end{pmatrix} \tag{4.16}$$

Wie bereits in Abschnitt 4.2.2.3 erläutert müssen dann die Beiträge der einzelnen Zeitschritte zu dem Gesamten Gewichtsgradienten für ein Neuron i bzw. für eine Schicht N_F^l aufaddiert werden. Erst dann kann das Gewichtsobjekt seinen eigenen Lernvorgang starten.

5 Experimente

5.1 Vorbemerkung

Im Folgenden werden die durchgeführen Experimente und ihre Ergbnisse dargestellt. Um die Beschreibung der gewählten Netzarchitekturen und -parameter zu erleichern werden hier einfache Schreibweisen eingeführt.

Verbindungsschichten Die Art der aufeinander folgenden Verbindungsschichten wird durch eine Zeichenkette beschrieben, in der durch Punkte getrennt Buchstaben die Art der Schicht beschreiben. Dabei steht

- C für eine Faltungsschicht,
- M für eine Maxpoolingschicht,
- CM für eine kombinierte Faltungs- und Poolingschicht und
- F für eine Vollverknüpfung

So beschreibt beispielsweise die Zeichenkette C.M.F ein Netz mit vier Neuronenschichten, bei dem die ersten beiden Neuronenschichten durch eine Faltungsschicht, Schicht Zwei und Drei durch eine Maxpoolingschicht und die letzten beiden Schichten vollverknüpft verbunden sind.

Filtergrößen und Kartenanzahl Auch bei Angaben zu Filtergrößen und Kartenzzahlen werden als Zeichenkette geschrieben, bei denen die Angaben pro Schicht durch Punkte getrennt werden. Wird beispielsweise die Anzahl der Karten pro Schicht beschrieben steht die Zeichenkette 2.8.8.2 für ein Netz mit vier Neuronenschichten, bei denen auf der ersten und letzten Neuronenschicht je zwei Karten, auf der zweiten und dritten Schicht je acht Karten sind.

5.2 Experimente zur Verifikation der Funktionsweise

Um die Funktionsweise der Faltungsschichten zu testen wurden einige einfache Versuche mit nur einer Faltungsschicht durchgeführt. Dabei ist zu beachten, dass Problemstellungen ausgewählt werden, die durch einen Filter auch erfüllt werden können. Da die Architektur des Frameworks Schichten mit nur einer Karte nicht zulässt, wurde ein Netz mit dem

Aufbau 2.2 gewählt, wobei die Eingaben und Teacher-Werte für die zweite Karte immer auf 0 gesetzt wurden.

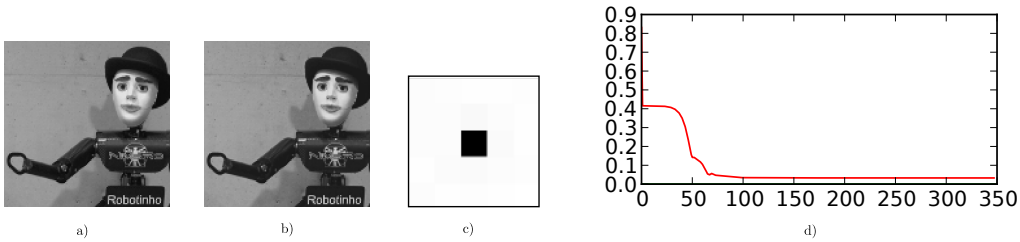


Abbildung 5.1: Ein 5×5 Faltungsfilter lernt die Identität. Weiß bezeichnet den Wert 0, schwarz den Wert 1. a) Die Eingabe, b) Die Ausgabe, c) der gelernte Filterkern, d) Die Entwicklung des mittleren quadratischen Fehlers

In Abbildung 5.1 ist das Ergebnis des Versuchs, einen 5×5 Faltungskern auf die Identität zu trainieren. Dem Netz wurde als Eingabemuster ein Bild des Kommunikationsroboters Robotinho auf der Eingabeschicht präsentiert. Anschliessend wurde das Eingabebild ebenfalls als Teacher angelegt und die Differenz zwischen der Ausgabe und dem Teacherbild genutzt, um das Netz mit Resilient Propagation zu trainieren. Obwohl die Wahl des Trainingsverfahrens bei nur einem Muster nicht optimal ist, erreichte das Netz bereits nach 60 Trainingsdurchläufen nahezu das Optimum. Optisch war zu dieser Zeit bereits kaum noch ein Unterschied zu erkennen.

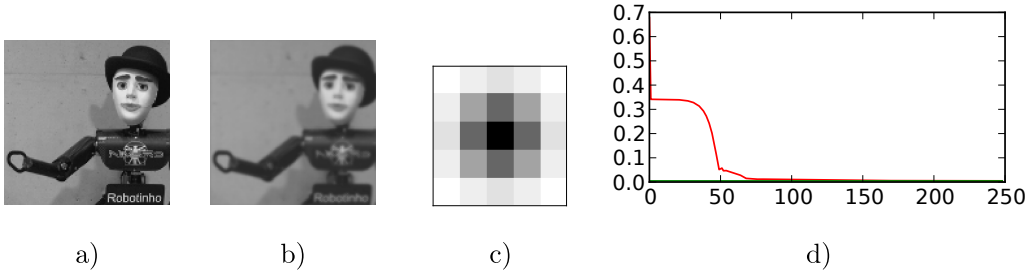


Abbildung 5.2: Ein 5×5 Faltungsfilter lernt einen Gaußkern. Weiß bezeichnet den Wert 1, schwarz den Wert -1. a) Die Eingabe, b) Die Ausgabe, c) der gelernte Filterkern, d) Die Entwicklung des mittleren quadratischen Fehlers

In einem weiteren Versuch wurde versucht, einen 5×5-großen Faltungsfilter zu trainieren, der ein Bild weichzeichnet. Dazu wurde zunächst eine Ausgabe erzeigt, in dem ein Gaußfilter in das Netz eingegeben und der Vorwärtspass ausgeführt wurde[1]. Diese Ausgabe

[1] Den Weichzeichner eines Zeichenprogramms zu verwenden hat sich als problematisch erwiesen, da diese unterschiedliche Algorithmen verwenden.

wurd dann als Teacher-Wert verwendet. Auch in diesem Test wurde nach ca. 60 Iterationen ein Minimum erreicht. Das Versuchsergebnis ist in Abbildung 5.2 dargestellt.

Um zu untersuchen, wie das Netz mit kombinierten Filtern umgeht, wurden zwei verschiedene zueinander orthogonale Kantenfilterkerne in das Netz eingegeben. Dazu wurden diesmal in der Eingabeschicht beide Karten mit je einem unterschiedlichen Eingabebild initialisiert.

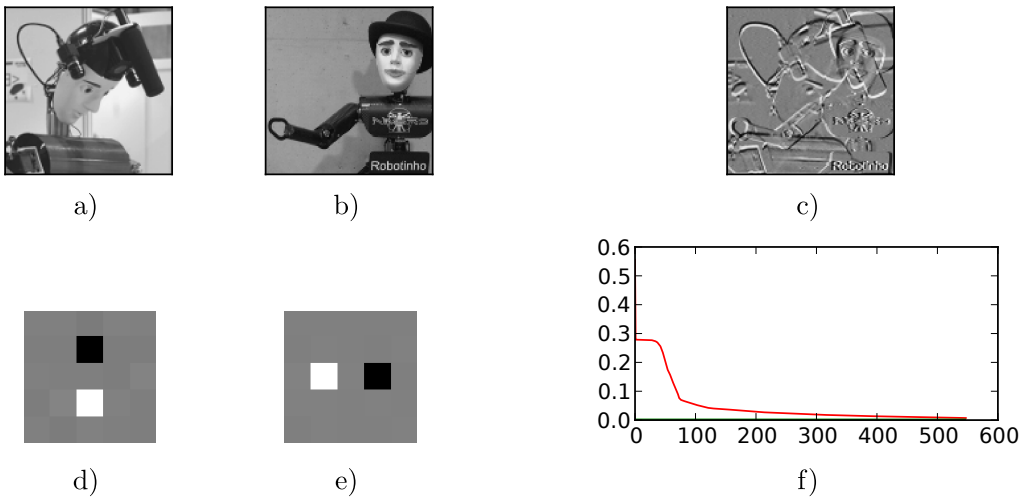


Abbildung 5.3: Ein 5×5 Faltungsfilter lernt zwei unterschiedliche Kantenfilter. Weiß bezeichnet den Wert 0, schwarz den Wert 1.

5.3 Klassifikationsleistung auf MNIST

Um die Funktionsweise des Lernverfahrens für den vorwärtsgerichteten Fall zu verifizieren, wurde untersucht, wie gut die Klassifikationsleistung eines nachgebauten LeNet-5 ist. Wie bereits beschrieben besteht das LeNet-5 aus zwei Faltungsschichten, zwei Subsamplingschichten, zwei vollverknüpften Schichten und einer Schicht radialer Basisfunktionen (vgl. Abschnit 3.1.2). Da Subsampling in der vorliegenden Arbeit nicht verwendet wurde, sind statt dessen Maxpooling-Schichten verwendet worden. Auch die letzte Schicht mit radialen Basisfunktionen stand nicht zur Verfügung und wurde daher weggelassen. Dennoch war das Ziel eine Fehlerrate von unter zwei Prozent zu erreichen.

In Anlehnung an LeNet-5 wurde die Netzstruktur also auf die folgenden Werte festgelegt. Als Verbindungsschichten wurde die Struktur CM.CM.F.F gewält, wobei die Faltungsfilter jeweils die Größe $fs = 7$ Pixel haben. LeCun verwendet die Filtergröße $fs = 6$, welche

sich durch die Randbedingung der CUV-Bibliothek, dass alle Filter eine ungerade Breite haben müssen, nicht umsetzen. Es wurde daher die nächst größere Filterbreite gewählt. Für die Neuronenschichten wurden die Kartenanzahlen 2.6.16.120.10 gewählt, wobei die letzten beiden Neuronenschichten nur Karten der Größe ein Neuron enthalten. Dies entspricht also einer Schicht mit insgesamt 120 Neuronen. Als Eingabekarte wird die erste Karte der untersten Neuronenschicht verwendet. Die zweite Karte in dieser Schicht ist leer, was bedeutet, dass die Aktivierung aller Neurone auf Null gesetzt wurde. Als Aktivierungsfunktion wird auf allen Schichten der Tangens hyperbolicus verwendet. Die Initialisierung der Gewichte erfolgte, wie in Abschnitt 4.2.2 beschrieben, in Abhängigkeit der eingehenden Verbindungen der Zielschicht. Die eingehenden Verbindungen lassen sich hier einfach durch die Anzahl der Karten der Vorgängerschicht multipliziert mit der Filtergröße ermitteln.

Der MNIST-Datensatz [YLH98] enthält 60.000 Graustufenbilder als Trainingsdatensatz und 10.000 Graustufenbilder als Testdatesatz. Alle Bilder zeigen handgeschriebene Ziffern und liegen in der Größe 28×28 Pixel vor. Der Datensatz wurde für das Training nicht vorverarbeitet, sondern nur auf den Wertebereich $[0; 1]$ normalisiert. So ist die Größe der untersten Schicht auf 28×28 Pixel festgelegt. Auf der zweiten Schicht haben die Karten die Größe 14×14 und auf der dritten Ebene 7×7.

Ziel des Klassifikationsnetzes ist es, zu entscheiden, welche Ziffer im Eingabebild zu sehen ist. Dazu wurde das Netz mit de, Backpropatagion-Algorithmus trainiert. Am Ende jeder Epoche wurde sowohl der durchschnittliche Mean-Squared-Error, als auch die Anzahl der korrekt klassifizierten Muster protokolliert. Alle zehn Epochen wird untersucht, wie gut das bislang trainierte Netz auf der Testmenge abschneidet (Validation). Es ist gelungen, das Netz so zu trainieren, dass es eine Fehlerrate von 0,83% auf der Testmenge und 0,00012% auf der Trainingsmenge erreicht wurde. Das Training wurde abgebrochen nachdem sich über 50 Epochen keine Verbesserung auf der Trainingsmenge ergeben haben. Es wurde aus Zeitgründen darauf verzichtet das Netz auszutrainieren. Das Ergebnis ist dennoch deutlich besser als die besten Ergebnisse, die mit einem Multi-Layer-Perzeptrons erzielt wurden (2,95%). Auch im Vergleich mit dem LeNet-5 ohne weitere Vorverarbeitung (0,95%) schneidet das oben beschriebene Netz besser ab. Spezialisierte Varianten, die auch Verschiebungen und Rauschen auf die Trainingsmenge anwenden, erreichen dagegen sogar Fehlerraten von 0,6%. Das jüngste berichtete Ergebnis entstand unter der Verwendung eines sechs-schichtigen vollverknüpften Neuronalen Netzes, welches von Ciresan et. al. [CMGS10] mit Hilfe einer GPU trainiert wurde. Sie erreichten sogar eine Fehlerrate von $0{,}35\%$. Dennoch legen die Ergebnisse nahe, dass die Funktionsweise des Netzes unter Verwendung des Backpropagation-Algorithmus verifiziert werden kon-

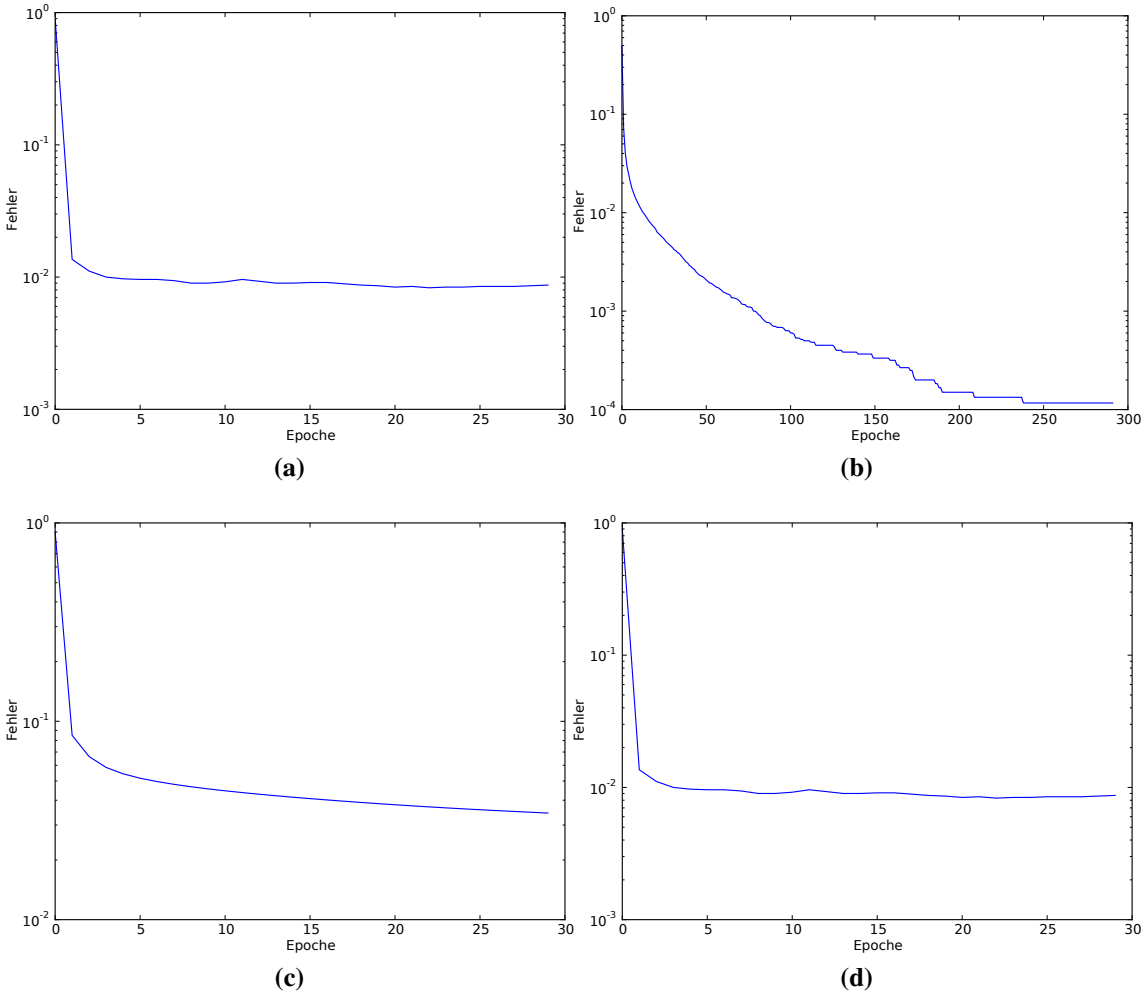


Abbildung 5.4: (a) Der durchschnittliche MSE auf der Trainingsmenge pro Epoche (b) Klassifizierungsfehler auf der Trainingsmenge pro Epoche (c) Der durchschnittliche MSE auf der Testmenge pro Epoche (d) Klassifizierungsfehler auf der Testmenge pro Epoche

nte. In Abbildung 5.4 sind die Entwicklung des mittleren quadratischen Fehlers und des Klassifikationsfehlers – oben für die Trainingsmenge und unten für die Testmenge. Die Einordnung des vergestellten Netzes im Vergleich zu anderen gängigen Ansätzen findet sich in Abbildung 5.5

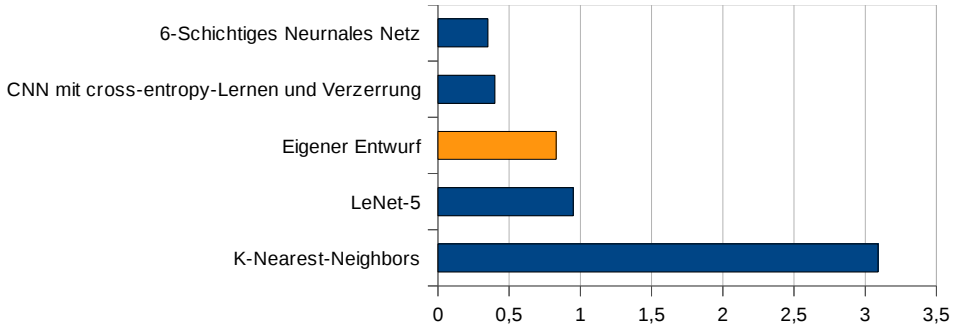


Abbildung 5.5: Vergleich verschiedener MNIST-Klassifkatoren: Die Balken geben die Größe des Klassifizierungsfehlers auf der Testmenge in Prozent an.

(a) **(b)** **(c)** **(d)**

Abbildung 5.6: Verschiedene zufällig ausgewählte Beispiele aus dem vorverarbeiteten BioID-Datensatz.

5.4 Lokalisierung von Augen

5.4.1 Der BioID-Datensatz

In dieser Arbeit wird als Datensatz zum Trainieren und Testen des Netzes die Trainingsdatenbank BioID sind 1521 Bilder Graustufenbilder der Größe 384×286 enthalten. Darauf abgebildet ist jeweils eine von insgesamt 23 in Kamerarichtung blickende Personen vor unterschiedlichen Innenraumhintergründen. Dazu sind manuell erhobene Augenpositionen als Pixelkoordinaten gegeben. Der Datensatz wurde vorverarbeitet, um das Training zu beschleunigen. Dazu mussten zunächst alle Bilder auf eine quadratische Größe von 384×384 Pixeln vergrößert werden. Dies führt dazu, dass es gleichfarbige Randbereiche im Bild gibt, deren Informationen zumindest an den Grenzbereichen zum eigentlichen BildDazu wurde zunächst der der mittlere Grauwert eines zu vergrößernden Bildes ermittelt und als Hintergrundfarbe verwendet. Anschliessend wurde ein weicher Übergang erzeugt, indem das Bild in vertikal in der Mitte eingefügt wurde und dann 30 mal mit einem Gauß-Filter geglättet wurde. Beispiele aus diesem vorverarbeiteten Datensatz finden sich in Abbildung 5.6.

5.4.2 Korrekte Lokalisation

Als Beurteilungskriterium für die Leistung der Lokalisation ist es erforderlich zu definieren, wann ein Auge als gefunden gelten kann und wann nicht. Dazu wird – analog zu [JKF01] – zunächst der euklidische Abstand zwischen den gefundenen und den tatsächlichen Augenpositionen ermittelt:

$$d_l = \|(x_l, y_l)^T - (\hat{x}_l, \hat{y}_l)^T, \| \tag{5.1}$$

$$d_r = \|(x_r, y_r)^T - (\hat{x}_r, \hat{y}_r)^T, \| \tag{5.2}$$

Dabei steht $(x_l, y_l)^T$ für den Ortsvektor des lokalisierten linken Auges und $(x_r, y_r)^T$ ist der Ortsvektor für das lokalisierte rechte Auge bei Eingabe des Musters m. Die tatsächlichen Positionen $(\hat{x}_l, \hat{y}_l)^T$ und $(\hat{x}_r, \hat{y}_r)^T$ sind dementsprechend auch Ortsvektoren. Ob die Lokalisierung der Merkmale nun genau genug erfolgt ist wird anhand des maximalen Fehlers bei der Lokalisierung bei der Merkmale im aktuellen Muster und dem Abstand zwischen den beiden Merkmalen festgelegt. Ist $d_{eye} \leq 0{,}25$, wobei

$$d_{eye} = \frac{\max(d_l, d_r)}{(\hat{x}_l, \hat{y}_l)^T - (\hat{x}_r, \hat{y}_r)^T} \tag{5.3}$$

ist, so gilt das Muster als korrekt klassifiziert. Diese Festlegung ist spezifisch für die Lokalisierung der Augen entworfen worden. Sie hilft, bei skalierten Gesichtsgrößen auch eine skalierte Akzeptanzschwelle zu berücksichtigen. Dies kann beispielsweise durch unterschiedliche Entfernungen der Gesichter zur Kamera vorkommen.

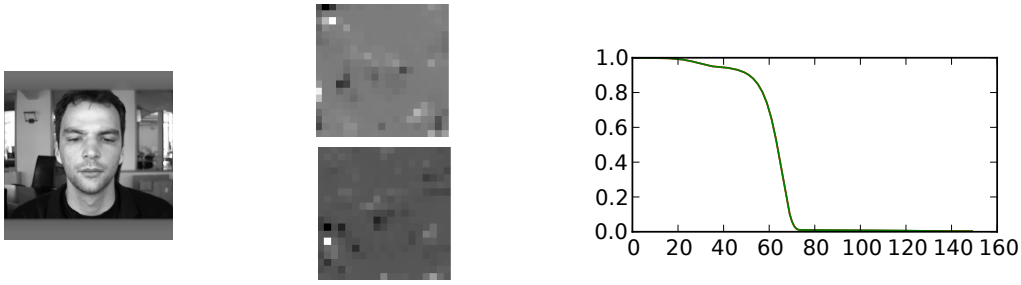
5.4.3 Experimente

Die Experimente gliederten sich in der Versuchsdurchführung in Trainingsphase, Validierungsphase und Testphase. Dazu wurde der BioID-Datensatz zufällig aufgeteilt in

- eine *Trainingsmenge* mit 1014 Bildern – das entspricht einem Drittel der Gesamtmenge
- eine *Validierungsmenge* mit 150 Bildern – das entspricht etwa 10% der Gesamtmenge
- und in eine *Testmenge* mit 357 Bildern

Die in dieser Arbeit verwendete Fehlerfunktion ist der quadratische, gemittelte Fehler (MSE):

$$E = \frac{1}{2} \sum_{m \in T_t^l} \sum_{n \in N_{t,A}^l} (t_n^m - a_n^m)^2$$



a) b) c)

Abbildung 5.7: Ein Netz mit einer Verbindungsschicht lernt zwei Filter, die jeweils das rechte bzw. linke Auge erkennen.

Dabei ist $t_n^m \in T_t^{l,k}$ der Wert auf den das Neuron n von Karte k in Schicht l trainiert werden soll, wenn Muster m in das Netz eingegeben wird.

Nach jeder 10. Trainingsepoche wird einmal der MSE auf der Validierungsmenge berechnet. Hat sich dieser gegenüber dem letzten Validierungsdurchlauf verbessert, so werden die aktuellen Gewichte gespeichert. Sonst wird weiter gelernt.

Auswendiglernen eines Bildes. Die Aufgabe einen Filter zu lernen, der in einem gegebenen Bild jeweils das rechte oder das linke Auge finden kann, wurde als Einstieg untersucht. Das Netz sollte in der Lage sein, die Schattierungen des Auges auswendig zu lernen. Da rechtes und linkes Auge normalerweise symmetrische Spiegelungen sind, dürfte auch die Entscheidung, ob es sich um ein rechtes oder linkes Auge handelt nicht allzu schwer sein.

Von diesen Überlegungen ausgehend wurde ein Experiment mit einem vorwärtsgerichteten 2.2-Netz mit einer Konvolutionsverbindung durchgeführt bei dem ein zufälliges Bild aus der Trainingsmenge genommen und als Eingabe verwendet wurde. Die zweite Karte in N_A^0 wurde an allen Positionen mit 0 gefüllt. Das Netz wurde darauf trainiert in der ersten Karte der zweiten Schicht im Zentrum des linken Auges eine 1 anzuzeigen und in der zweiten Karte der zweiten Schicht das Zentrum des rechten Auges mit einer 1 anzuzeigen. Die Teacherwerte für die Ausgabeschicht lauten unter Verwendung zweier Teacher-Zentrenpunkte also:

$$T_0^{1,k}(i,j) = \begin{cases} 1, & \text{falls } (i,j) \text{ Zentrum des Auges für Karte } k \text{ und} \\ -1, & \text{sonst} \end{cases} \tag{5.4}$$

Die gelernten Filter (siehe Abbildung 5.7) bestehen zu großen Teilen aus betragsmäßig kleinen Werten (graue Bereiche). Einige wenige Pixel nehmen betraglich große Werte an. Dabei stehen weiße Pixel für positive Werte und schwarze Pixel für negative Werte. Das Ergenis zeigt, dass es nicht erforderlich ist, die Schattierung des gesamten Auges zu lernen, sondern einige wenige Pixel ausreichen. Ein weiteres Experiment mit einem 5×5-Filter zeigte, dass diese Größe bereits ausreichte, um die Augen punktgenau zu lernen. Ein 3×3-Filter war zu einer eindeutigen Lokalisierung im verwendeten Bild allerdings nicht mehr in der Lage.

Experimente mit zwei Modellen auf allen Bildern In weiteren Experimente wurde untersucht, wie gut die Lokalisierung von Augen in einer Neuronenkarte mit einem vorwärtsgerichteten Netz funktioniert. Um den Suchraum zu beschränken wurde die Untersuchung auf zwei generelle Architekturen beschränkt. Diese werden im Folgenden kurz skizziert.

Zunächst wurde ein *Modell ohne Pooling* festgelegt. Bei der Architektur ohne Pooling wurden die Neuronenschichten mit Faltungsschichten verbunden. Da kein Pooling oder Subsampling statt findet sind die Neuronenkarten aller Schichten gleich groß. Die Zahl der Karten erhöht sich in diesem Modell um die Hälfte der Karten der vorangegangenen Schicht. Damit soll das Modell in die Lage versetzt werden komplexere Strukturen zu lernen. Das Modell ohne Pooling ist in der Lage beliebig tiefe Architekturen zu ermöglichen. Es wurde zunächst mit einem punktgenauen Zielwert trainiert, bei dem das Netz lernen soll, genau das Neuron zu aktivieren, an welchem das Auge ist und alle anderen auf Null zu setzen. Als Lehrerkarte wird also wie oben eine Karte mit den Werten aus Gleichung 5.4 verwendet.

Zudem soll ein *Modell mit Pooling* untersucht werden. In diesem Modellfolgt auf eine Faltungsschicht eine Poolingschicht. Bei jedem Pooling wird die Karte um den Faktor vier kleiner. Daher sind bei diesem Modell nicht so tiefe Architekturen möglich, wie bei dem Modell oben. Die Anzahl der Karten wird in diesem Modell, analog zu den Arbeiten von LeCun zum LeNet oder der Neuronalen Abstraktionspyramide von Behnke, nach jedem Pooling um den Faktor zwei erhöht.

In diesem Modell ändert sich die Auflösung auf der Ausgabekarte. Daher wurden die Karten subpixelgenau trainiert. Dazu wurde die Position der Augen subpixelgenau auf die Ausgabeschicht projeziert und mit einer Gaußglocke modelliert. Da die Augenpositionen als Koordinaten sowohl für das rechte Auge $eye_0 = (x_0, y_0)$, als auch für das linke Auge $eye_1 = (x_1, y_1)$ vorliegen, reicht es die Koordinaten so oft durch zwei zu teilen, wie es Schichten im Netz gibt. Bezeichne also h die Tiefe des Netzes, dann liegt das Zentrum

der Gaußglocken, die die gewünschten Aktivitäten der Ausgabekarte modellieren, bei $c_0 = (x_0/2^h, y_0/2^h)$ und $c_1 = (x_1/2^h, y_1/2^h)$. Die Teacherkarten berechnen sich dann aus:

$$T_0^{1,k}(i,j) = e^{-\left(\frac{(i-x_k)^2}{2\sigma^2}+\frac{(j-y_k)^2}{2\sigma^2}\right)}, k \in \{1,2\} \tag{5.5}$$

Da der Wert an der Spitze der Glocke auf die maximale Aktivität, also eins, gesetzt werden soll, wurde auf eine Normierung der Gaußglocke verzichtet.

Zum Vergleich der bei den Modell sollte verglichen werden, wie gut Netze mit unterschiedlich großen Filtern die Lokalisierung auf dem BioID-Datensatz lernen. Da die Tiefe im Fall des Modells mit Pooling proportional mit der Zahl der Schichten abnimmt, wurde in diesem Fall die Tiefe auf drei Schichten festgelegt. So sind die Ausgabekarten des Modells mit Pooling noch 24×24 Pixel groß. Diese Fragestellung war interessant, da in der Vergleichsarbeit von Behnke kein Maxpooling verwendet wurde. Die Ergebnisse können die Vermutung, dass Maxpooling wie bei Klassifizierungsaufgaben in [SMB10a] allerdings nicht stützen. Der theoretische Vorteil, dass das rezeptive Feld einer Zelle durch das Pooling vergrößert, konnte zumindest in der gewählten Architektur scheinbar nicht genutzt werden.

In beiden Modellen wurde die geschätzte Position wie in [Beh01] subpixelgenau bestimmt. Dazu wurde zunächst das Pixel ausgewählt, dessen Neuron die maximale Aktivierung a_{max} in der Ausgabekarte hat. Um dieses Neuron herum wurden alle Nachbarn in einem 7×7-Umfeld betrachtet, deren Aktivierung größer als ein Schwellwert $v(d)$ ist. Dabei steigt $v(d)$ mit zunnehmendem Abstand vom Zentrum und berechnet sich aus:

$$v(d) = 0,5 \cdot a_{max} \cdot d/7 \tag{5.6}$$

Von allen übrig gebliebenen Werten wurde die gewichtete Mitte berechnet, welche als Lokalisierungsergebnis angenommen wird. Somit konnte auch das Modell ohne Pooling blobs ausbilden, was auch geschehen ist (vgl. Abbildung 5.8).

Anschließend wurde untersucht, wie die Erkennungsleistung mit der Tiefe des Netzes skaliert und welche Transferfunktion auf der Ausgabeschicht besser funktioniert.

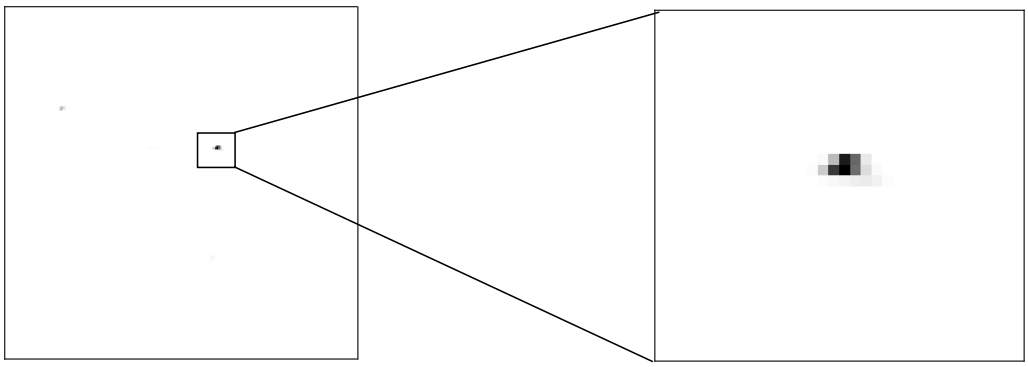


Abbildung 5.8: Der produzierte Ausgabeblob für ein linkes Auge. Links befindet sich die gesamte Neuronenkarte, rechts ein vergrößerter Ausschnitt um den eingezeichneten Bereich.

Netzkonfiguration		Resultate		
Modell	Karten	V-Fehler (%)	T-Fehler (%)	Epoche
CM	9.9.9	74,67	66,39	200
C	9.9.9	26,67	29,13	40
CM	11.11.11	74,67	57,14	230
C	11.11.11	19,33	13,45	60
CM	13.13.13	79,33	78,43	410
C	13.13.13	15,33	10,64	60
CM	15.15.15	51,33	46,22	560
C	15.15.15	10,67	20,45	40

Tabelle 5.1: Vergleich der Lokalisierungsleistung beider Modelle den beiden in Abschnitt 5.5 Lösungsstrategien. Zum Training wurden folgende Parameter verwendet: Trainingsmethode war RPROP, die Kartenzahlen waren 2.16.16.2, die Eingabegröße war 192×192 Pixel und als weight decay Parameter wurde $0{,}5$ verwendet.Angegeben sind der Fehler auf der Validierungsmenge (V) und der Testmenge (T).

Netzkonfiguration			Resultate		
Funktion	Schichten	Karten	V-Fehler (%)	T-Fehler (%)	Epoche
softmax	6	2.4.6.10.16.24.2	35,33	36,97	90
softmax	5	2.4.6.10.16.2	24,00	28,29	70
softmax	4	2.4.6.10.2	28,00	27,45	50
softmax	3	2.4.6.2	22,00	30,81	50
sigm	6	2.4.6.10.16.24.2	18,67	13,73	370
sigm	5	2.4.6.10.16.2	85,33	78,43	510
sigm	4	2.4.6.10.2	22,00	19,89	640
sigm	3	2.4.6.2	23,33	30,25	590

Tabelle 5.2: Vergleich der Ergebnisse von dem Modell ohne Pooling mit verschiedener Anzahl an Schichten und den beiden in Abschnitt 5.5 Lösungsstrategien. Zum Training wurden folgende Parameter verwendet: Trainingsmethode war RPROP, die Eingabegröße war 192×192 Pixel, die Filtergröße war jeweils 11 und als weight decay Parameter wurde $0{,}5$ verwendet. Angegeben sind der Fehler auf der Validierungsmenge (V) und der Testmenge (T).

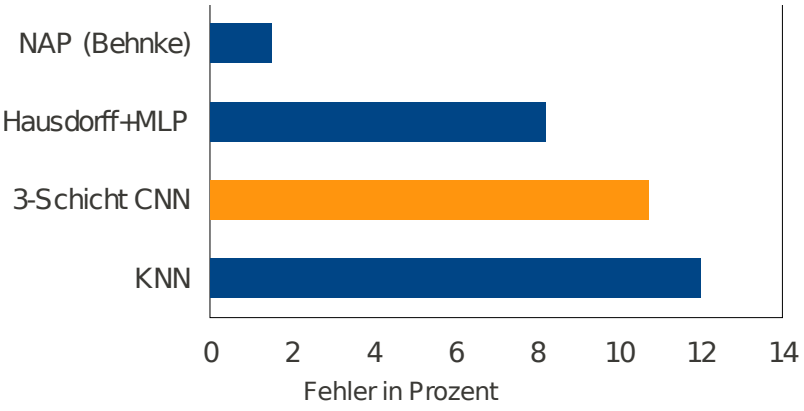


Abbildung 5.9: Vergleich der Lokalisierungsfehler verschiedener Verfahren mit dem besten gefundenen Ergebnis. Quellen: NAP [Beh01], Hausdorff+MLP [JKF01]

5.5 Das Problem der bildhaften Ausgabe

In der vorliegenden Arbeit ist das Ziel, eine Ausgabekarte zu lernen, die genauso groß ist, wie die Eingabekarte und durch ein Aktivitätsmaximum anzeigt, wo die Augen oder andere Körperteile im Bild zu finden sind. Dieser Vorgang wird in dieser Arbeit als Lokalisierung bezeichnet. Grundsätzlich soll eine Ausgabekarte nur auf ein Merkmal – also beispielsweise ein Auge – trainiert werden. Die Teacher-Karte zu einem solchen Merkmal besteht aus einer Karte, die überall wenig oder keine Aktivität hat, außer an der Stelle, an der sich das Merkmal befinet. Dort ist die Aktivität am höchsten.

Bereits bei den Experimenten mit vorwärtsgerichteten Netzen wurde deutlich, dass das Training mit einer naiven Teacherkarte, die nur einen aktivierten Pixel enthällt, keine guten Ergebnisse liefert. Da die Filter an vielen Stellen im Bild angewendet werden, sammeln sie auch die Fehler all dieser Stellen auf (vgl. Abschnitt 4.2.2.3). Offenbar wird dieser Mechanismus zum Problem, wenn ein Filter darauf trainiert werden soll, die Aktivierung nur eines einzelnen Pixels zu stärken und gleichzeitig an allen anderen Stellen die Aktivierung zu senken. In Experimenten konnte beobachtet werden, dass die Filter in diesem Fall nur lernen, dass die Aktivierung überall Null sein muss. Offenbar überlagern also die vielen, wenn auch kleinen Fehlerwerte der großen ruhigen Bereiche, den Fehler, den ein Filter macht, wenn das zu lernende Merkmal nicht angezeigt wird. Dieser Effekt wächst mit der Bildgröße.

Um diesem Problem zu begegnen, sind zwei denkbare Alternativen untersucht worden. Zunächst wurde die Wahl einer anderen Transferfunktion für die Ausgabekarten untersucht. Es wurde daher eine Ausgabefunktion gesucht, die große Fehler stärker gewichtet, als kleine Fehler. Eine bereits eingesetzte Transferfunktion, die diese Aufgabe im Rahmen der Klassifizierung leistet, ist die Softmax-Funktion:

$$f_{act}(a_j^{l+1}) = \frac{e^(a_i^l)}{\sum_{j=1}^{n} \exp(a_j^l))} \tag{5.7}$$

Hier bezeichne wieder $a_i \in N_A^{l,m}$ die Aktivierung eines Neurons i aus der Karte m der Schicht l und n bezeichne die Anzahl aller Neurone in dieser Karte. Das Ziel dieser Aktivierungsfunktion ist es, die Ausgabe der Zielneuronen als Wahrscheinlichkeitsverteilung darzustellen, so dass alle Aktivierungen einer Karte m in Schicht l zwischen Null und Eins liegen:

$$\forall a \in N_A^{l,m} : 0 \geq a \leq 1$$

und sich alle Aktivierungen sich zu eins aufaddieren:

$$\sum_{a_i \in N_A^{l,m}} a_i = 1$$

Behnke verwendet zudem in [Beh01] eine „verbesserte“ (engl. *rectifying*) Transferfunktion f_{st} wie in Formel 5.8 definiert.

$$f_{st}(x) = \frac{\log(1 + e^{\beta x})}{\beta} \tag{5.8}$$

Die Ableitung $\partial f_{st}(x)/\partial x$ ist

$$\frac{\partial f_{st}(x)}{\partial x} = \frac{e^{\beta \cdot x}}{e^{\beta \cdot x} + 1} \tag{5.9}$$

Diese Funktion wurde berücksichtigt, da Behnke berichtete, dass diese ebenfalls entworfen wurde, um die oben beschriebenen Probleme zu umgehen. Der cuv-Bibliothek wurde im Rahmen dieser Arbeit die Funktoren `tf_rect` und `tf_drect` hinzugefügt. Während `tf_rect` punktweise die Transferfunktion auf eine Matrix anwendet, in deren Zeilen die Karten als fortlaufender Vektor abgelegt sind. ist die Funktion von `tf_drect` etwas weniger intuitiv.

Sowohl in der cuv-Bibliothek, als auch im entworfenen Framework werden die Ableitungen auf den Aktivierungen berechnet und nicht auf den Netzeingaben. Grund für die Entscheidung ist, Speicherplatz auf der Grafikkarte zu sparen, indem die Netzeingaben nicht gespeichert beleiben müssen. Dazu ist es allerdings erforderlich die Formel 5.9 nicht in Abhängigkeit der Netzeingabe (in dem Fall x) zu formulieren, sondern in Abhängigkeit der Aktivierung $f_{st}(x)$.

So wurde es erforderlich eine Formulierung der Ableitung $\partial f_{st}(x)/\partial x$ herzuleiten, die es erlaubt, sie als Funktion der Aktivierung zu schreiben:

$$\partial f_{st}(x)/\partial x = f_{dst}(f_{st}(x)) = 1 - \frac{1}{f_{st}(x) \cdot e^{\beta}} \tag{5.10}$$

Diese Herleitung ergibt sich, wenn in Formel 5.9 Nenner und Zähler substituiert werden: Der Nenner kann ersetzt werden durch

$$e^{\beta x} = f_{st}(x) \cdot e^{\beta} - 1 \tag{5.11}$$

und der Zähler durch

$$e^{\beta x} + 1 = f_{st}(x) \cdot e^{\beta} \tag{5.12}$$

.

Somit ergibt sich

$$f_{dst}(f_{st}(x)) = (f_{st}(x) \cdot e^{\beta} - 1) \cdot \frac{1}{f_{st}(x) \cdot e^{\beta}} = 1 - \frac{1}{f_{st}(x) \cdot e^{\beta}} \tag{5.13}$$

Die Verwendung der Softmax- oder Rectifying-Transferfunktionen hat sich in Experimenten jedoch als problematisch herausgestellt. Da die Funktionen beide die e-Funktion im Zähler haben[2], wachsen sie sehr schnell. Die Umsetzung dieser Arbeit auf CUDA-Grafikkarten mit der Compute Capability 1.3 limitiert den zur Speicherung der Aktivierung verwendbaren Datentypen jedoch auf den Typ `float`, welcher im genutzten System maximal den Wert $3{,}40282e + 38 = \exp(100{,}235763498)$ darstellen kann. Das bedeutet, dass die Netzeingabe vor Anwendung der Exponetialfunktion nicht größer werden darf, als $100{,}235763498$. Die Verwendung des Datentyps `double` würde schon Netzeingaben bis zu $721{,}295636615$ ermöglichen.Bei den durchgeführen Experimente wurde diese Grenze mit längerer Trainingsdauer durch das stetige Anwachsen der Gewichte auch unter Verwendung von weight decay mit der Zeit nahezu immer erreicht.

Ein weiterer Nachteil ist, dass die Funktionen sich nur mit einer punktgenauen Teacherkarte trainieren lassen. Es wäre andererseits wünschenswert statt mit einem einzelnen Punkt gleich mit einer kleinen aktivierten Region zu trainieren. Denn die manuell markierten Augenpositionen haben selbst bereits eine Varianz, die nicht von der Erscheinung der Augen abhängen, sondern von der Genauigkeit, mit der die Mitte des Auges getroffen wurde.

Auf dieser Überlegung baut der zweite Ansatz um dem Problem der dominanten Mini-Fehler zu begegnen besteht darin, die Position nicht mit Hilfe eines einzelnen Pixels als Teacher-Wert zu trainieren, sondern mit Hilfe Gaußverteilung $G_T(\hat{x}, \hat{y})(x, y)$ der Aktivierungen um das Zentrum $(\hat{x}, \hat{y})$ des zu trainierenden Merkmals. Dieser Ansatz hilft dabei, die Varianz, die durch das manuelle markieren der Augen entsteht zu berücksichtigen. Hilfreich dabei ist, dass eine Lokalisation, die um ein Pixel verschoben ist, nicht direkt komplett falsch ist, wie bei der Verwendung einer punktgenauen Teacherkarte.

In der Teacherkarte, die mit der Gaußverteilung modelliert wird, sind dann um die Position der Merkmale herum weitere Pixel aktiviert, deren Intensität allerdings mit zunehmender

[2]Dies ist im Gegensatz beispielsweise zur Fermifunktion zu sehen, bei der die e-Funktion im Nenner steht und die beim Überlauf dazu führt, dass der Bruch 0 wird

Entfernung zum Zentrum abnimmt. Der Bereich, der aktiviert ist, wird im folgenden als *Blob* bezeichnet. Um den Fehler zu berechnen muss dann die Abweichung der tatsächlichen Aktivierung $N_{A,t}^{l_{T_i},m_{T_i}}(x,y)$ an Pixel (x,y) zum Zeitpunkt t von der Aktivierung dieses Pixels in $G_T(\hat{x},\hat{y})$ abgezogen werden:

$$G_{T_i(\hat{x},\hat{y})}(x,y) = w_t \cdot (e^{\frac{(x-\hat{x})^2+(y-\hat{y})^2}{2\sigma}} - N_{A,t}^{l_{T_i},m_{T_i}}(x,y)) \tag{5.14}$$

Das Training mit einem Blob in der Teacherkarte erfordert allerdings eine andere Transferfunktion zu nutzen. Schließlich lassen sich Blobs nur schlecht normieren, so dass die Grenzfunktion der Gaußkurve wieder ein einzelner aktivierter Pixel ist. Dieser Umstand führte dazu, daß wieder die sigmoide Fermifunktion als Transferfunktion auf der Ausgabeschicht verwedet wurde. Ein weiterer Vorteil dieser Wahl ist, dass das Training wieder numerisch stabil wird. Allerdings braucht das Verfahren empirisch gesehen viel länger (vgl. Tabelle 5.2) als die Verwendung des Softmax. Eine mögliche Erklärung ist, daß der Gradient beim Softmax deutlicher ist, da nur relative Unterschiede betrachtet werden. Wie oben erklärt, fallen so kleine Fehlerwerte geringer ins Gewicht.

5.6 Laufzeitanalyse

Die verwendete CUV-Bibliothek sollte es ermöglichen, die Berechnungen des CNN zu parallelisieren und damit erheblich zu beschleunigen. Daher wurde untersucht wie die gut die Laufzeit auf der Grafikkarte im Vergleich zur Laufzeit auf einem CPU-System ist. Grundsätzlich ist dabei zu berücksichtigen, dass das System im Entwurf für rekurrente Netze ausgelegt wurde.

Zur Untersuchung der Laufzeit wurde ein Experiment mit einem vorwärtsgerichteten Konvolutionsnetz mit sieben Schichten durchgeführt. Die Schichten hatten die Kartenzahlen 2.4.8.16.32.32.32.2 und jeweils die Größe 384×384 Pixel. Die Filtergröße wurde zwischen sieben und dreizehn variiert. Dabei wurde festgestellt, dass die Implementierung mit Verwendung der Grafikkarte Beschleunigungsfaktoren zwischen 60 und 103 erzielte. Abbildung 5.11 visualisiert die gemessenen Laufzeiten. Im Vergleich zur Arbeit von Scherer, der einen Beschleunigungsfaktor von bis zu 115 erreichte liegen die hier erreichten zwar leicht zurück. Allerdings ermöglicht die vorgestellte Architektur eine flexiblere Struktur.

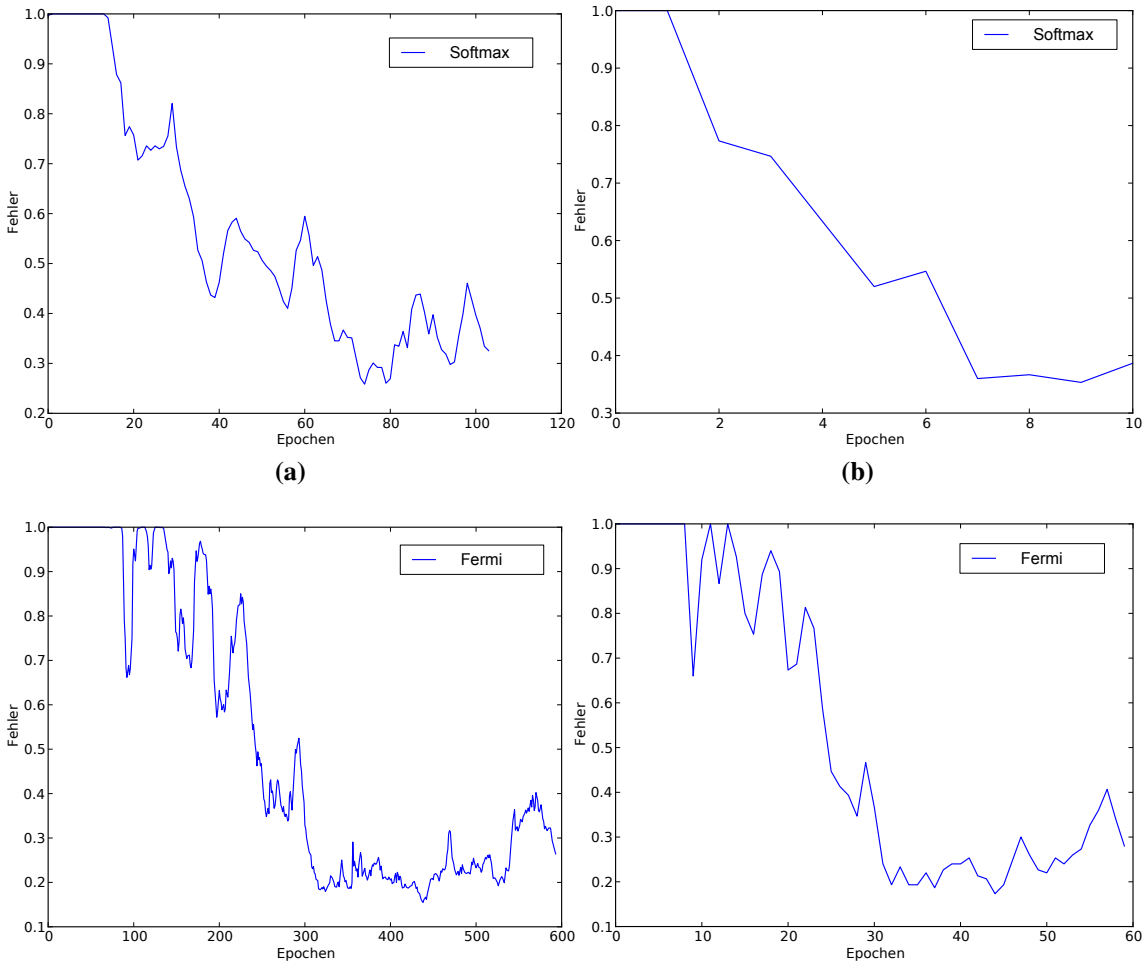


Abbildung 5.10: Die Lernkurven der 6 schichtigen Architekturen: Man sieht, dass das Training mit Softmax schneller zu Ergebnissen führt (oben). Allerdings bricht das Verfahren früh mit einem Überlauf ab.

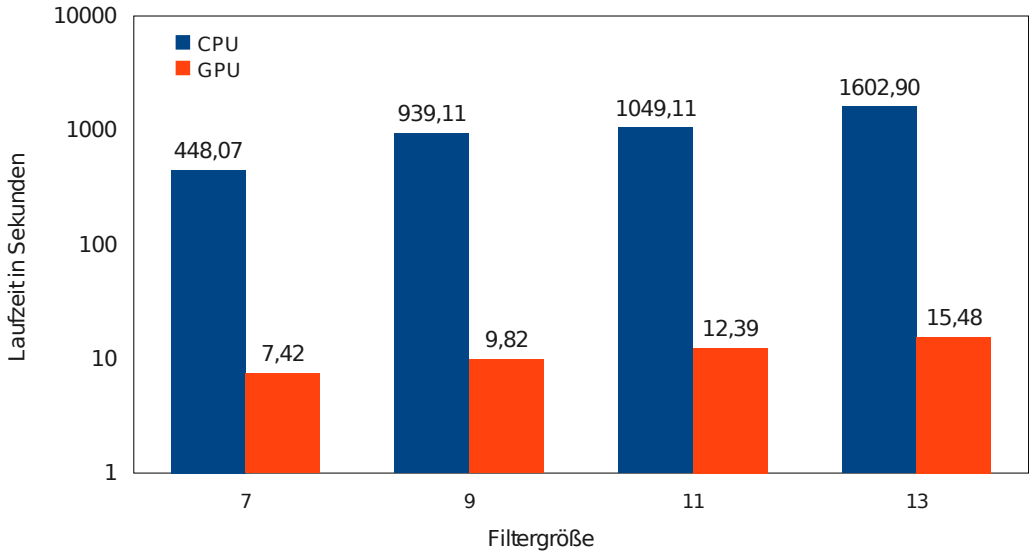


Abbildung 5.11: Vergleich der Laufzeiten zwischen der GPU- und der CPU-Variante für verschiedene Filtergrößen.

5.7 Ausblick

Im Rahmen der Arbeit sind einige Ideen und Ansätze entstanden, die aufgrund der knappen Zeit nicht mehr implementiert und untersucht werden konnten. Dennoch sollen diese hier kurz beschrieben werden.

Interessant wäre es zu untersuchen, wie anders strukturierte Filter wirken. Beispielsweise könnte man Filter verwenden, deren Auflösung radial abnimmt. In der Mitte wird dann hochaufgelöst nach Merkmalen gesucht, während im Außenbereich nur noch schwache Kontextinformationen gesammelt werden. Diese Idee ist der Anordnungsdichte der Stäbchen und Zapfen im Auge entlehnt. Auch dort gibt es einen Punkt des schärfsten Sehens an dem die Ortsabtastung deutlich höher ist als auf dem Rand.

Die Parallelisierung mit nur einem parallelen Muster macht einen erheblichen Geschwindigkeitsverlust aus. Scherer und Uetz nutzen Mini-Batches, die parallel abgearbeitet werden (vgl. Anschnitt 3.4). Wie in Abschnitt 4.1.2 beschrieben wurde, treten bei dieser Lösungsstrategie Speicherplatzprobleme auf der Grafikkarte auf. Interessant wäre es, zu versuchen, diese Beschränkung aufzuheben, in dem mittels PyRO mehrere Prozesse verwaltet werden, welche je eine Grafikkarte steuern. Da PyRO netzwerkbasiert arbeitet, wäre es sogar möglich nicht nur verschiedene Karten zu verwalten, sondern auch verschiedene Rechner.

Im Laufe der Experimente wurde festgestellt, dass die Verwendung von Softmax oder der von Behnke eingesetzten rectifying Transferfunktion als Ausgabefunktion problematisch

ist. Grund war, dass CUDA-fähige Grafikkarten mit der Compute-Capability 1.3 nur den Datentyp `float` verarbeiten können. Neuere Karten können auch den Datentyp `double` berechnen. Daher wäre es interessant zu untersuchen, ob sich die Lokalisierungsleistung bei längerem Training mit der Softmaxfunktion weiter verbessert. Dies ermöglicht auch eine bessere Vergleichbarkeit zum zweiten Ansatz zur Lösung des Problems der bildhaften Ausgabe.

Literaturverzeichnis

[ABBB08] T. Axenbeck, M. Bennewitz, S. Behnke, and W. Burgard. Recognizing complex, parameterized gestures from monocular image sequences. In *8th IEEE-RAS International Conference on Humanoid Robots, 2008. Humanoids 2008*, pages 687–692, 2008.

[AKNN92] T. Agui, Y. Kokubo, H. Nagashashi, and T. Nagao. Extraction of FaceRecognition from monochromatic photographs using neural networks. In *Proc. Second Int'l Conf. Automation, Robotics, and Computer Vision*, volume 1, pages 1881–1885, 1992.

[BBLP10] Y.L. Boureau, F. Bach, Y. LeCun, and J. Ponce. Analysis of Feature Learning and Feature Pooling for Image Recognition. *Learning Workshop, Cliff Lodge, Snowbird, Utah*, 2010.

[BC94] G. Burel and D. Carel. Detection and localization of faces on digital images. *Pattern Recognition Letters*, 15(10):963–967, 1994.

[Beh01] S. Behnke. Learning iterative image reconstruction in the neural abstraction pyramid. *International Journal of Computational Intelligence and Applications*, 1:427–438, 2001.

[Ben09] Yoshua Benigo. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, pages 1–127, 2009.

[Bou06] Jake Bouvrie. Notes on convolutional neural networks. Technical report, Massachusetts Institute of Technology, Center for Biological and Computational Learning, 2006.

[BP01] G. Bi and M. Poo. Synaptic modification by correlated activity: Hebb's postulate revisited. *Annual Review of Neuroscience*, 24(1):139–166, 2001.

[CMGS10] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.

[Com10] Riverbank Computing. PyQt 4.7.3. http://www.riverbankcomputing.co.uk/software/pyqt/, 2010.

[Cor10] NVIDIA Corporation. Nvidia cuda programming guide – version 2.3.1, 2010. Adresse: `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`, Besuch: 24.02.2010.

[dJ] Irmen de Jong. PYRO – Python Remote Objects. http://www.xs4all.nl/ irmen/pyro3/.

[Fah88] S.E. Fahlman. An empirical study of learning speed in back-propagation networks. *Computer Science Technical Report, CMU-CS-88-162, Carnegie Mellon University, Pittsburgh*, 1988.

[fIV10] Uni Bonn Institut für Informatik VI. Cuv-bibliothek (http://www.ais.uni-bonn.de/deep_learning/downloads.html), 2010. Universität Bonn, Institut für Informatik VI, Abteilung für Autonome Intelligente Systeme.

[FMI88] K. Fukushima, S. Miyake, and T. Ito. Neocognitron: A neural network model for visual pattern recognition. *Evolution, learning, and cognition*, page 233, 1988.

[FS93] B. Lamy F. Soulie, E. Viennet. Multi-modular neural network architectures: Pattern recognition applications in optical character recognition and human face recognition,. *Int'l J. Pattern Recognition and Artificial Intelligence, vol. 7, no. 4*, pages 721–755, 1993.

[Fuk80] K. Fukushima. Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193, 1980.

[Fuk89] K. Fukushima. Analysis of the process of visual pattern recognition by the neocognitron. *Neural Networks*, 2(6):413–420, 1989.

[Gro10] Krhonos Group. OpenCL – The open standard for parallel programming of heterogeneous systems, 2010. http://www.khronos.org/opencl/.

[Hay94] S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1994.

[Heb49] D. Hebb. *The Organization of Behavior*. New York: Wiley, 1949.

[HL01] E. Hjelmås and B.K. Low. Face detection: A survey. *Computer vision and image understanding*, 83(3):236–274, 2001.

[HMLYC97] C. Han, H. Mark Liao, G. Yu, and L. Chen. Fast face detection via morphology-based pre-processing. In *Image Analysis and Processing*, pages 469–476. Springer, 1997.

[HN90] R. Hecht-Nielsen. *Neurocomputing*. Addison Wesely Publishing Company, Reading, MA, 1990.

[HR96] T. Kanade H. Rowley, S. Baluja. Human face detection in visual scenes. *Advances in Neural Information Processing Systems 8, D.S. Touretzky, M.C. Mozer, and M.E. Hasselmo, eds.*, pages 875–881, 1996.

[HW59] D. H. Hubel and T. N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 148(3):574, 1959.

[HW99] E. HjelmQas and J. Wroldsen. Recognizing faces from the eyes only. *Proceedings of the 11th Scandinavian Conference on Image Analysis*, 1999.

[Jac88] R.A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

[JKF01] Oliver Jesorsky, Klaus Kirchberg, and Robert Frischholz. Robust face detection using the hausdorff distance. In Josef Bigun and Fabrizio Smeraldi, editors, *Audio- and Video-Based Biometric Person Authentication*, volume 2091 of *Lecture Notes in Computer Science*, pages 90–95. Springer Berlin / Heidelberg, 2001.

[KH92] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems 4*, pages 950–957. Morgan Kaufmann, 1992.

[KP97] C. Kotropoulos and I. Pitas. Rule-based face detection in frontal views. In *International conference on acoustics, speech, and signal processing*, pages 2537–2540, 1997.

[Kri07] David Kriesel. *Ein kleiner Überblick über Neuronale Netze*. erhältlich auf http://www.dkriesel.com, 2007.

[Kri10] Alex Krizhevsky, 2010. Adresse: `http://www.cs.utoronto.ca/~kriz/`, Besuch: 24.05.2010.

[KSJ96] E.R. Kandel, J.H. Schwartz, and T.M. Jessell. *Neurowissenschaften: eine Einführung*. Spektrum Akademischer Verlag, 1996.

[KT04] M. Kolsch and M. Turk. Robust hand detection. In *Proc. of the Sixth IEEE Int. Conf. on Automatic Face and Gesture Recognition (FG)*, 2004.

[LBOM98] Y. LeCun, L. Bottou, G. Orr, and K. Müller. Efficient backprop. *Neural networks: Tricks of the trade*, pages 546–546, 1998.

[LCJB+89] Y. Le Cun, LD Jackel, B. Boser, JS Denker, HP Graf, I. Guyon, D. Henderson, RE Howard, and W. Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.

[LGTB97] S. Lawrence, C. L. Giles, A. C Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE Transactions on Neural Networks*, 8(1):98–113, 1997.

[LHBB99] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with Gradient-Based learning. page 823. 1999.

[LS99] Daniel D. Lee and H. Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, page 401:788–791, 1999.

[MP94] B. Moghaddam and A.P. Pentland. Face recognition using view-based and modular eigenspaces. In *Proceedings of SPIE*, volume 2277, page 12, 1994.

[Nok10] Nokia. The Qt library. http://qt.nokia.com, 2010.

[OB04] E.J. Ong and R. Bowden. A boosted classifier tree for hand shape detection. In *Face and Gesture Recognition*, pages 889–894, 2004.

[OCM07] M. Osadchy, Y.L. Cun, and M.L. Miller. Synergistic face detection and pose estimation with energy-based models. *The Journal of Machine Learning Research*, 8:1215, 2007.

[Pod07] V. Podlozhnyuk. Image convolution with CUDA. *NVIDIA Corporation white paper, June*, 2097(3), 2007.

[RB93] M. Riedmiller and H. Braun. A direct adaptive method for faster back-propagation learning: The RPROP algorithm. In *Proceedings of the IEEE international conference on neural networks*, volume 1993, pages 586–591. San Francisco: IEEE, 1993.

[RB94] M. Riedmiller and H. Braun. Rprop-description and implementation details. Technical report, 1994.

[RHW86] DE Rumelhart, GE Hinton, and RJ Williams. Learning internal representations by error propagation. In *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1*, pages 318–362. MIT Press, 1986.

[Rie93] M. Riedmiller. Untersuchungen zu Konvergenz und Generalisierungsfähigkeit überwachter Lernverfahren mit dem SNNS. *Workshop SNNS-93: Simulation Neuronaler Neutze mit dem SNNS, Bericht Nummer 10/93*, pages 107 – 116, 1993.

[Roj96] R. Rojas. *Neural networks: a systematic introduction*. Springer, 1996.

[Sch09] Dominik Scherer. Gpu-beschleunigte objekterkennung mit neuronalen konvolutionsnetzen, 2009. Universität Bonn, Institut für Informatik VI, Abteilung für Autonome Intelligente Systeme.

[Sin94] P. Sinha. Object recognition via image invariants: A case study. *Investigative Ophthalmology and Visual Science, vol. 35, no. 4*, pages 1735–1740, 1994.

[Sin95] P. Sinha. Processing and recognizing 3d forms. 1995.

[SK87] L. Sirovich and M. Kirby. Low-dimensional procedure for the characterization of human faces. *Journal of the Optical Society of America A*, 4(3):519–524, 1987.

[SMB10a] D. Scherer, A. Müller, and S. Behnke. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. *20th International Conference on Artificial Neural Networks (ICANN)*, 10 2010.

[SMB10b] H. Schulz, A. Müller, and S. Behnke. Exploiting local structure in stacked Boltzmann machines. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*, 2010.

[SNK72] T. Sakai, M. Nagao, and T. Kanade. Computer analysis and classification of photographs of human faces. In *Proc. First USA-Japan Computer Conference*, volume 1, pages 2–7, 1972.

[Str97] B. Stroustrup. The C++ programming language. 1997.

[TP91] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[Uet09] Rafael Uetz. Parallele Implementierung hierarchischer neuronaler Netze mit lokaler Konnektivität zur Objekterkennung in natürlichen Bildern, 2009.

Universität Bonn, Institut für Informatik VI, Abteilung für Autonome Intelligente Systeme.

[VCLC94] R. Vaillant, Monrocq C., and Y. Le Cun. An original approach for the localisation of objects in images. *IEE Proc. Vision, Image and Signal Processing, vol. 141*, pages 245–250, 1994.

[VJ01] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, 2001.

[Wer74] P.J. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *PhD thesis*, 1974.

[Wer88] PJ Werbos. Backpropagation: Past and future. In *IEEE International Conference on Neural Networks, 1988.*, pages 343–353, 1988.

[Wer90] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 1990.

[Wis05] L. Wiskott. How Does Our Visual System Achieve Shift and Size Invariance? *23 problems in systems neuroscience*, 2005.

[YH94] G. Yang and T. S Huang. Human face detection in a complex background. *Pattern recognition*, 27(1):53–63, 1994.

[YHC92] A.L. Yuille, P.W. Hallinan, and D.S. Cohen. Feature extraction from faces using deformable templates. *International journal of computer vision*, 8(2):99–111, 1992.

[YKA02] M.H. Yang, D.J. Kriegman, and N. Ahuja. Detecting faces in images: A survey. *IEEE Transactions on Pattern analysis and Machine intelligence*, pages 34–58, 2002.

[YL04] L. Bottou Y. LeCun, F.J. Huang. Learning methods for generic object recognition with invariance to pose and lighting. *CVPR*, 2004.

[YLH98] Y. Bengio Y. LeCun, L. Bottou and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86(11):2278-2324*, 1998.

[Zel94] Andreas Zell. *Simulation Neuronaler Netze.* Oldenbourg Verlag, 1994.

[ZZ10] C. Zhang and Z. Zhang. A Survey of Recent Advances in Face Detection. Technical report, Microsoft Research, One Microsoft Way, Redmond, WA 98052, 2010.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bergisch Gladbach, 21. September 2010 .....

Markus Gerhards