Convolutional Neural Network

Studienarbeit

des Studiengangs Informationstechnik
an der dualen Hochschule Baden-Württemberg Stuttgart

von
Benjamin Riedle, Florian Schmidt, Josua Benz

4. Juni 2018

Bearbeitungszeitraum:	30. November 2017 - 4. Juni 2018
Matrikelnummer, Kurs:	9224451, 7930173, 9016181, TINF15IN
Gutachter der Dualen Hochschule:	Alfred Strey

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema "Convolutional Neural Network" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____ _____

Ort Datum Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Symbolverzeichnis

API Application Programming Interface

AVX Advanced Vector Extension

BLAS Basic Linear Algebra Subprograms

CNN Convolutional Neural Network

CPU Central Processing Unit

MIC Many Integrated Cores

MKL Math Kernel Library

MLP Multy Layer Perceptron

MMX Multi Media Extension

MNIST database Modified National Institute of Standards and Technology database

OpenMP Open Multi Procesing

PCIe Peripheral Component Interconnect express

SIMD Single Instruction, Multiple Data

SSE Streaming SIMD Extension

TBB Threading Building Blocks

1 Einleitung

Künstliche Intelligenz hält in immer mehr Bereichen des alltäglichen Lebens Einzug. Sprachassistenten auf dem Handy sind seit einigen Jahren selbst in Einsteigermodellen vorhanden und auch die Anzahl der Haushalten mit smarten Lautsprechern wächst [1]. Aktuelle Autos besitzen unter anderem einen Spurhalteassistenten und bremsen automatisch, falls ein Fußgänger die Fahrbahn betritt. Auf einem Tablett kann mit dem Finger auf den Bildschirm geschrieben werden woraufhin das Gerät die geschriebenen Buchstaben erkennt und den Text aus Unicode Zeichen schreibt.

Für einen Menschen ist es nicht schwierig, eine Zahl auf einem Bild richtig einzustufen oder zu entscheiden, ob ein Bild eine Person zeigt oder nicht. Für einen Computer ist dieser Entscheidungsvorgang jedoch sehr schwierig. W. Hilberg schreibt dazu: "Die meisten Menschen „rechnen“ nicht, sondern sie „denken“. Rechnen ist nur eine spezielle Variante des Denkens"[2]. Computer beherrschen also nur einen Teil des Denkens, in diesem Bereich sind sie jedoch sehr gut, Berechnungen mit vielen, großen Zahlen sind für einen PC sehr einfach.

Um eine künstliche Intelligenz zu erschaffen, kann deswegen nicht die klassische Programmierung verwendet werden. Stattdessen werden sogenannte Neuronale Netzwerke verwendet. Diese Bestehen aus vielen Neuronen und verhalten sich ähnlich wie Neuronen innerhalb des Gehirns von Lebewesen.

Die Neuronen in einem Gehirn empfangen Signale in Form von elektrischen Impulsen und geben diese weiter an andere Neuronen. Dieses banale Prinzip ermöglicht uns, Dinge zu erlernen und gelerntes anzuwenden. Die Anzahl der Neuronen ist hierfür entscheidend, im menschlichen Gehirn existieren laut älteren Studien etwa 100 Milliarden Neuronen (vgl. [2]). Nach aktuelleren Studien geht man heute von circa 86 Milliarden Neuronen aus [3].

Der Mensch erweitert im Laufe seines Lebens durch Interaktion mit der Umwelt sein Wissen. Genauso lernt ein Neuronales Netzwerk durch viele Trainingsdatensätze und Zyklen das gewünschte Verhalten. Wenn das Verhalten den Anforderungen entspricht, kann das Trainieren abgebrochen werden. Das trainierte Netzwerk mit den gelernten Werten kann dann für den Anwendungsfall verwendet werden. Durch diese Methode können beliebige Entscheidungsvorgänge trainiert werden, solange geeignete Trainingsdaten vorhanden sind.

Für die Bilderkennung werden spezielle, sogenannte Convolutional Neuronal Networks verwendet. Diese eignen sich Muster an, mit denen die Bilder abgescannt werden. Solch ein Netzwerk soll im Laufe dieser Arbeit entstehen, um Handgeschriebene Ziffern richtig zu erkennen. Dabei sollen drei verschiedene Implementierungen entstehen, die jeweils das Netzwerk auf eine spezielle Zielhardware optimiert.

2 Zusammenfassung

Diese Arbeit befasst sich mit der Entwicklung von Convolutional Neural Netzworks. Es wurden im Rahmen dieser Studienarbeit vier verschiedene Implementierungen entwickelt, wobei die erste Implementierung als Vorlage der drei anderen Implementierungen diente. Die erste Implementierung wurde als Gruppenarbeit von allen drei Bearbeitern der Studienarbeit erstellt. Diese hat noch keinerlei Optimierung und läuft lediglich seriell auf einem Kern der CPU ab. Für einen produktiven Einsatz ist diese Implementierung deshalb nicht geeignet.

Die anderen drei Implementierungen wurden jeweils für eine spezielle Architektur von einem der Bearbeiter optimiert. Zum einen gibt es eine Optimierung für x86-Prozessoren wie sie in Arbeitsplatz-PCs und Notebooks vorkommen. Des weiteren gibt es eine Implementierung für GPUs mit der CUDA-Programmiertechnik. Diese ist für Computer mit Nvidia-Grafikkarten optimiert und führt die Berechnungen auf der GPU anstelle der CPU aus. Eine weitere Implementierung wurde für Server mit einem Intel Xeon Phi-Coprozessor entwickelt. Dabei handelt es sich um eine PCIe-Karte mit einer CPU welche über 50 Kerne besitzt.

Zum Testen und Trainieren des Netzwerkes wurden der MNIST-Datensatz verwendet. Dabei handelt es sich um einen Datensatz welcher 65000 Bildern von handgeschriebenen Ziffern beinhaltet, wovon 55000 für das Trainieren und 10000 für das Verifizieren gedacht sind. Mit den Trainingsdatensätzen soll ein Netzwerk trainiert werden, welches die anderen Ziffern erkennt, obwohl das Netzwerk diese Bilder nie davor gesehen hat.

Am Ende erfolgt ein Vergleich zwischen den Implementierungen dieser Arbeit und TensorFlow, einer Bibliothek zum Erstellen von Neuronalen Netzwerken die Google für viele ihrer Dienste verwendet, unter anderem für den Google Translator.

3 Abstract

This thesis deals with the development of convolutional neuronal networks. Within the context of this thesis, four different implementations were developed, whereby the first implementation functions as a template of the three other implementations. The first implementation was developed together in a team. There is no optimization of any sort inside this implementation and it runs fully serial on only one core of the CPU. Therefor this implementation is not usable for productive usage

The other three implementations were optimized for a specific hardware by one of the editors. On one hand, there is a implementation for x86 processors. These are common in Office-PCs and Notebooks. In the other hand, there is a implementation for GPUs which supports the application programming interface CUDA. This one in optimized for graphic cards from Nvidia and calculates the values on the graphic processor instead of the CPU. Another implementation was designed for servers with a Intel Xeon Phi Co-processors. The Xeon Phi is a CPU-PCIe-Card with over 50 kernels inside.

For training and verification purpose, the MNIST data set was used. This in an data set with 65000 images of handwritten numbers. 55000 of these are for training purpose and 10000 for verification purpose. With the training data set a network should be conducted, whitch recognizes different numbers, allthough it has never seen them befor.

At the end of this thesis, there is a comparison between the different implementations of this piece of work and TensorFlow, a library designed to create neuronal networks which is used by Google, among others, for many services like Google Translator.

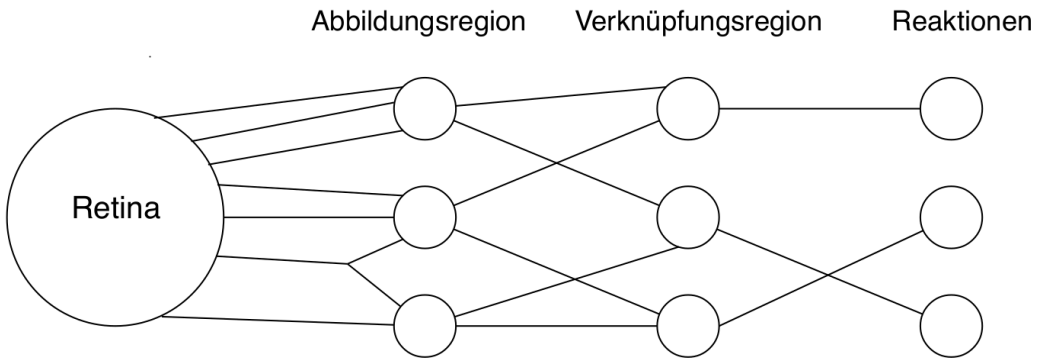
4 Klassische Neuronale Netzwerke

Neuronale Netzwerke können als Implementierung eines Gehirns auf einem Computer verstanden werden. Die Funktionsweise eines Neuronalen Netzwerkes wurde vom biologischen Vorbild abgeschaut und an die Gegebenheiten eines Computers angepasst.

Der größte Unterschied zwischen der klassischen Programmierung und der Erstellung von Neuronalen Netzwerken ist, dass bei der Erstellung eines Netzwerkes nicht die tatsächlichen Berechnungen festgelegt werden, sondern diese mit Trainingsdatensätze erst gelernt werden müssen. Der Softwareentwickler kann lediglich an Paramentern drehen und beispielsweise die Anzahl der Neuronen oder die Anzahl der Layer eines Netzwerkes festlegen. Große Firmen wie Google, Facebook oder Apple sammeln hierfür schon seit Jahren Daten von Nutzern, um diese für das Trainieren von Netzwerken zu verwenden und dadurch neue Geschäftsmodelle zu entwickeln.

4.1 Funktionsweise Perceptron

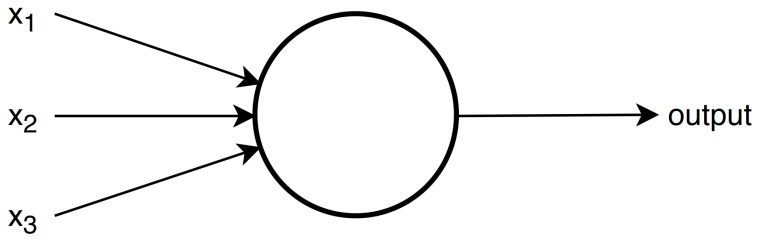
Die Idee der Perceptrons wurde von Frank Rosenblatt das erste mal 1958 in einem Paper erläutert ([4]). Der Unterschied zu noch älteren Modellen war, dass sich die Struktur eines Netzes beim Modell von Rosenblatt nicht während des Lernens ändert, sondern sich nur die Schwellwerte der einzelnen Perzeptronen ändern. Zuvor wurde versucht, den Lernvorgang eines biologischen Gehirns möglichst genau auf dem PC zu Implementieren. Später wurde das Modell von Minsky und Papert untersucht und vereinfacht. ([5])



Die Grafik zeigt ein Netz mit Perzeptronen nach Rosenblatt. Auf die Retina, dem Fachbegriff für die biologische Netzhaut eines Auges, wird ein Bild abgebildet. Die Retinapunkte sind mit der ersten Schicht von Perceptronen fest, deterministisch verbunden. Die Verbindungen zur zweiten Schicht, der Verknüpfungsschicht, sind zufällig gewählt. Ebenso sind auch die Verbindungen zwischen der zweiten und der dritten Schicht zufällig gewählt.

"Die Idee von Rosenblatt war, mit der Verknüpfungsregion bestimmte Muster in der Eingabe aus den Retinapunkten zu erkennen und die entsprechende Reaktion zu steuern. Der Lernalgorithmus muß die notwendigen Gewichte finden, um Muster aus der Retina mit Reaktionen zu assoziieren. ([5])"

Das Modell von Rosenblatt wurde im Lauf der Zeit immer weiter reduzier. Wird heute von Perceptronen gesprochen, besitzen diese mehrere binäre Inputs und einen binären Output. ([6]) In der Grafik unterhalb hat das Perceptron 3 Inputs, es kann aber ebenso weniger oder mehr Inputs besitzen.



Der Wert des Outputs ist abhängig von den Werte der Inputs. Rosenblatt führte für die Inputs Gewichte ein. Für jeden Input x_j gibt es ein bestimmtes Gewicht w_j aus dem Menge der natürlichen Zahlen. Ist die Summer der Multiplikation von Input mit dazugehörigem Gewicht größer als ein Schwellwert, ist der Output 1. Ist die Summer kleiner als der Schwellwert, ist der Output 0.

$$output = \begin{cases} 0 & falls \quad \sum_j x_j * w_j \leq Schwellwert \\ 1 & falls \quad \sum_j x_j * w_j > Schwellwert \end{cases} \tag{4.1}$$

Mit den Funktionen des Perceptrons können schon einfache Problemstellungen logisch gelöst werden. Außerdem können mit diesen Perceptronen logische Gatterschaltungen entworfen werden, da sie sich ähnlich wie digitale Bausteine verhalten. [6]

Für schwierigere Probleme, wie die Erkennung von handgeschriebenen Ziffern, sind die Perceptronen jedoch nicht mächtig genug und nur sehr eingeschränkt benutzbar. Ein weiterer Nachteil der Perceptonen ist, dass ein Netzwerk, aufgrund der verwendeten natürlichen Zahlen, nur schwer trainierbar ist. Deswegen werden heute für Neuronale Netzwerke andere Neuronen verwendet, die Grundidee dahinter ist jedoch auch nach 60 Jahren noch die gleiche.

4.2 Aktivierungen

Perceptronen bieten in ihrer Grundform die Möglichkeit, unbeschränkte Eingaben mit unbeschränkten Gewichten und zu verarbeiten. Ihre Ausgabe wiederum ist bekanntlich auf die diskreten Werte 1 oder 0 beschränkt. Um es Perceptronen zu ermöglichen, ein lernendes Netzwerk aufzubauen, ist es notwendig, dafür zu sorgen, dass die Ausgaben der Perceptronen beherrschbar bleiben. Ziel ist es, mit kleinen Veränderungen der Gewichte und Biases auch kleine Veränderungen der Outputs zu bewirken (vgl. [7]). Bei Verwendung der klassischen Perceptronen können kleine Änderungen der Weights w und Biases b nur Änderungen in der Ausgabe in Form von Sprüngen zwischen 0 und 1 und vice versa bewirken.

Um nun auch kleine Änderungen in den Outputs eines Perceptrons zu gestatten, wird die Funktion zur Bestimmung dessen nach Gleichung 4.2 zu 4.3] modifiziert.

$$output = \begin{cases} 0 & falls \quad w \cdot x + b \leq 0 \\ 1 & falls \quad w \cdot x + b > 0 \end{cases} \tag{4.2}$$

$$output = a(w \cdot x + b) \tag{4.3}$$

Hierbei stellt $a(x)$ eine sogenannte Aktivierungsfunktion dar. Aufgabe dieser Funktion ist es, die erzeugte Ausgabe kontinuierlich im Intervall $[0; 1]$ zu verteilen.
Damit ist gewährleistet, dass kleine Änderungen in Weights und Biases auch kleine Änderungen in den Outputs der Neuronen erzeugen. Es existieren einige geeignete Aktivierungsfunktionen, die die genannten Eigenschaften besitzen. In der Praxis werden auch verschiedene solcher Funktionen verwendet. Durch geschickte Wahl der Aktivierungsfunkton kann das Verhalten des Netzes für die entsprechende Aufgabe optimiert werden.
In dieser Arbeit wurden zwei unterschiedliche Aktivierungsfunktionen genutzt. Diese sollen im folgenden kurz dargestellt werden.

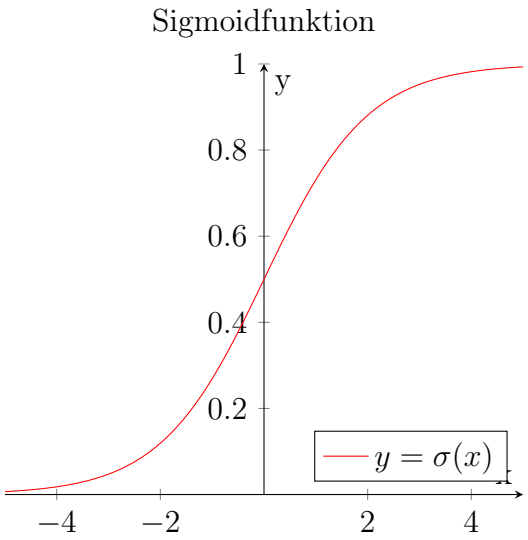


Abbildung 4.1: Funktionsgraph der Sigmoidfunktion

Sigmoidfunktion Die Sigmoidfunktion $\sigma(x)$ enstpricht den gegebenen Forderungen an eine Aktivierungsfunktion $a(x)$ und bildet damit den Definitionsbereich $\mathbb{D} =]-\infty; \infty[$ auf den Wertebereich $\mathbb{W} =]0; 1[$ ab. Der Begriff Sigmoidfunktion wird hier wie auch häufig an anderer Stelle für den Spezialfall der *logistischen Funktion* verwendet, welche nach Gleichung 4.4 definiert wird.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.4}$$

Im Fall der Neuronalen Netze wird die Sigmoidfunktion $\sigma(x)$ auf das Ergebnis der Operation $w \cdot x + b$ angewandt. Damit bestimmt sich also der Output der Perceptronen nach Gleichung **??**.

$$output = \tag{4.5}$$

Nachteile dieser Funktion sind die schnelle Annäherung der Funktion an die Endwerte 0 und 1, sowie ihr Nulldurchgang bei 0.5. Sie ist daher nur für betragsmäßig kleine Gewichte w und Biases b geeignet. Ihr Graph ist in Abbildung 4.1 gezeigt.

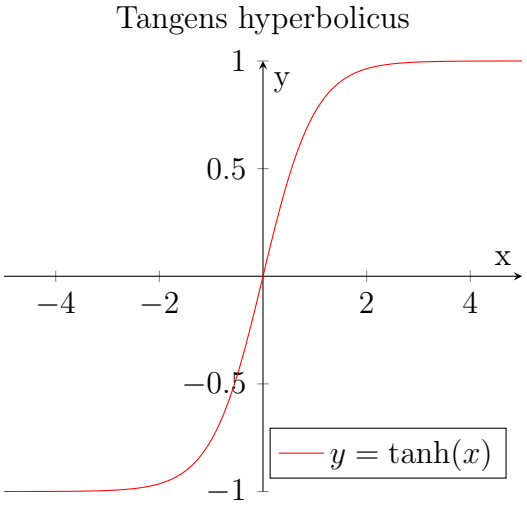


Abbildung 4.2: Funktionsgraph des Tangens hyperbolicus

Tangens hyperbolicus Die zweite in dieser Arbeit verwendete Aktivierungsfunktion ist der Tangens hyperbolicus $\tanh(x)$. Auch diese Funktion zeigt einen sigmoiden Verlauf. Sie bildet den Definitionsbereich $\mathbb{D} =]-\infty; \infty[$ allerdings auf den Wertebereich $\mathbb{W} =]-1; 1[$ ab.

Negative Werte sind nach den oben genannten Bedingungen für eine Aktivierungsfunktion jedoch nicht zulässig. Deshalb muss bei Verwendung des tanh darauf geachtet werden, die Funktion zu verschieben und negative Werte zu vermeiden. Der Vorteil in der Verwendung des Tangens hyperbolicus liegt im Vergleich zur Sigmoidfunktion im flacheren Verlauf der Funktion. Es kann durch geeignete Verschiebung eine bessere Verteilung im geforderten Wertebereich $[0; 1]$ erzielt werden. Der Graph der tanh-Funktion ist in Abbildung 4.2 dargestellt. In dieser Arbeit konnten die besseren Lernergebnisse unter Verwendung des Tangens hyperbolicus erzielt werden.

4.3 Gradient Descent

Grundidee von Neuronalen Netzen ist, das Netz mit Hilfe von Trainigsdatensätzen auf die Erkennung von Mustern zu trainieren. Eine Menge an Eingangsdaten erzeugt eine Menge an Ist-Ausgabedaten, welche eine Differenz zu gegebenen Soll-Ausgabedaten aufweist. Im Fall der in dieser Arbeit behandelten Erkennung von handschriftlichen Ziffern besteht die Eingabemenge aus einem 28×28-Pixel großen Eingabebild und die Ausgabemenge aus einer zehn Werte umfassenden Zuordnung der Eingabe zu der dargestellten Ziffer. Die Pixel des Eingabebildes werden in Graustufen im Intervall $[0; 1]$ angegeben, um den Anforderungen der Neuronen zu genügen. In der geordneten Ausgabemenge, ist jeder der möglichen Ziffern genau ein Wert zugeordnet. Stellt die Eingabe die betreffende Ziffer dar, so ist der zugeordnete Ausgabewert 1, andernfalls 0.

Das Neuronale Netz berechnet dementsprechend zehn Ausgabewerte, welche als Wahrscheinlichkeit für das Vorhandensein der jeweiligen Ziffer in der Eingabe gesehen werden kann. Somit kann für jeden Ausgabewerte eine Abweichung zum Sollwert bestimmt werden. Um nun eine Aussage über die Qualität Ausgabebestimmung des Netzes zu einem gegebenen Eingabebild in der momentanen Konfiguration von Gewichten w und Biases b zu bestimmen, muss eine Kostenfunktion definiert werden (vgl. [7]). In dieser Arbeit wurde dazu die Kreuzentropie, im weiteren Verlauf auch als Cross Entropy bezeichnet, als Kostenfunktion $C(w, b)$ verwendet. Dabei sind die Funktionsparameter zu beachten. Die Kostenfunktion von Neuronalen Netzen beschreiben im Allgemeinen eine Funktion von Gewichten w und Biases b.

Die Allgemeine Form der Kreuzentropie stellt eine Funktion $H(p, q)$ von zwei Wahrscheinlichkeitsverteilungen $p(x)$ und $q(x)$ dar.

$$H(p, q) = -\sum_{x} p(x) \log q(x) \tag{4.6}$$

Im Fall des hier beschriebenen Neuronalen Netzes ist die Cross-Entropy-Kostenfunktion $C_E(w, b)$ wie in Gleichung 4.7 dargestellt definiert.

$$C_E(w, b) = -\sum_{i=0}^{9} y_i \log x_i \tag{4.7}$$

Dabei stellt y_i den gegebenen Sollwert für eine mögliche Ziffer dar und x_i die zugehörige berechnete Wahrscheinlichkeit. Eindeutiges Ziel des Trainings des Netzes ist es offensichtlich diese Kostenfunktion zu minimieren. Eine Minimierung der Kosten bedeutet eine genauere Abbildung der Eingabemenge auf das korrekte Ausgabeergebnis.

Mit Hilfe des Gradienten ∇C_E kann nun bestimmt werden in welcher Weise die Gewichte w und die Biases b verändert werden müssen, um ein besseres Ergebnis zu erzielen. Der Gradient ∇C_E ist dabei wie ein Gleichung 4.8 definiert.

$$\nabla C_E = \begin{pmatrix} \frac{\partial C_E}{\partial w} \\ \frac{\partial C_E}{\partial b} \end{pmatrix} \tag{4.8}$$

Die Änderung der Kosten $\Delta C_E(w, b)$ bestimmt sich bei kleinen Änderungen der Gewichte Δw und kleinen Änderungen der Biases Δb nach Gleichung 4.9.

$$\Delta C_E = \frac{\partial C_E}{\partial w}\Delta w + \frac{\partial C_E}{\partial b}\Delta b \tag{4.9}$$

Wenn nun die partiellen Ableitungen $\frac{\partial C_E}{\partial w}$ und $\frac{\partial C_E}{\partial b}$ bestimmt werden, können die Kosten in kleinen Schritten dem Kostenminimum angenähert werden. Die Schrittweiten Δw und Δb werden in Neuronalen Netzen über die *Learning Rate* η bestimmt. Dieser Parameter bestimmt die Geschwindigkeit des Lernprozesses. Er sollte aber gering gehalten werden, da bei großen η ein Pendeln um das Kostenminimum entstehen kann. Die Learning Rate kann im Laufe des Lernprozesses auch dynamisch angepasst bzw. verringert werden um das Lernen zu beschleunigen und im späteren Verlauf des Lernprozesses die Genauigkeit der Anpassung zu erhöhen.

Stochastic Gradient Descent Um die Gesamtkosten des Lernprozesses zu bestimmen, muss das Neuronale Netz die Outputs aller Bilder des kompletten Trainingsdatensatzes bestimmen und die Kosten aller einzelnen Muster berechnen und summieren. Danach kann aus den gemittelten Kosten der Gradient $\nabla C(w, b)$ ermittelt werden. Bei großen Datensätzen wie dem hier verwendeten MNIST-Datensatz führt das zu einem langsamen Lernprozess, da der Datensatz komplett durchlaufen werden muss, bis eine Anpassung der Gewichte vorgenommen werden kann. Generell sind sehr viele Anpassungsschritte nötig, um hohe Genauigkeiten des Netzwerks zu erreichen. Eine Lösung für dieses Problem bietet der *Stochastic Gradient Descent*-Ansatz.

Bei Stochastic Gradient Descent nähert man den Gradienten $\nabla C(w, b)$ des Datensatzes über eine Teilmenge dessen an. Diese Teilmenge wird im Folgenden Batch genannt. Die Annahme hinter diesem Ansatz ist, dass in einem zufällig verteilten Batch der Gradient $\nabla C_j(w, b)$ eine Näherung des tatsächlichen Gradient $\nabla C(w, b)$ darstellt (vgl. [7]).

$$\nabla C = \frac{\sum_{i=0}^{N} \nabla C_i}{N} \approx \frac{\sum_{k=0}^{M} \nabla C_{j_k}}{M} = \nabla C_j \tag{4.10}$$

Gleichung 4.10 zeigt diesen Zusammenhang. Dabei stellt N die Gesamtzahl der Trainingsdaten und M die Größe eines Batches dar. Der Index j bezeichnet das jeweils vorliegende Batch. Verfolgt man nun diesen Ansatz, werden die Gewichte w und Biases b während des Durchlaufens des Trainingsdatensatzes mehrmals angepasst und der Lernprozess des Neuronalen Netzes wird deutlich beschleunigt.

4.4 Backpropagation

Im Jahr 1986 stellten David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams in ihrem Paper *Learning representations by back-propagating errors* den Algorithmus zur hier beschriebenen Backpropagation vor. Mit Hilfe dieser Methodik kann ein Zusammenhang zwischen der Änderung der Kostenfunktion $\nabla C(w, b)$, welche nur im finalen Layer exakt bestimmt werden kann, und den Änderungen der Gewichte Δw und Biases Δb der einzelnen Neuronen im gesamten Netz hergestellt werden. Dadurch wird es möglich, aus den Kosten $\nabla C(w, b)$ alle Gewichte w und Biases b im Netz anzupassen.

Der Backpropagation-Algorithmus basiert auf der Erkenntnis, dass zwischen dem Output jedes Neurons ein linearer Zusammenhang mit den Neuronen der vorhergehenden Schicht besteht (vgl. [8]). Es wird nun angenommen, jedem Neuron im Netzwerk wird eine Änderung Δz zugewiesen, wobei

$$z = w \cdot x + b \tag{4.11}$$

definiert wird. Des Weiteren wird ein δ_i für jedes Neuron nach Gleichung 4.12 definiert.

$$\delta_i = \frac{\partial C}{\partial z_i} \tag{4.12}$$

Der Backpropagation-Algorithmus stellt nun ein Verfahren zur Verfügung, welches dieses δ_i aus den Neuronen der finalen Schicht L für alle Neuronen des Netzwerks rückwirkend bestimmt. Die δ_j^L der letzten Schicht können nach Gleichung 4.13 bestimmt werden.

$$\delta_j^L = \frac{\partial C}{\partial o_j^L} \sigma'(z_j^L) \tag{4.13}$$

Dabei stellt $o_j = \sigma(z_j)$ den Output eines Neurons j dar. Die δ_j^l der weiteren Schichten l können nacheinander über Gleichung 4.14 bestimmt werden.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l) \tag{4.14}$$

Hierbei beschreibt der Operator $\circ$ das Hadamard-Produkt, welches die elementweise Multiplikation von Vektoren beschreibt. In Gleichung 4.14 beschreiben δ^l und z^l jeweils die Vektoren für die zugehörigen Schichten l und w^{l+1} die Matrix der Gewichte zwischen den Schichten l und $(l+1)$. Aufgrund der Linearität von z lassen sich nun über die $delta^l$ die partiellen Ableitungen der Kosten nach den Gewichten w und Biases b bestimmen.

$$\frac{\partial C}{\partial w^l} = a^{l-1}\delta^l \tag{4.15}$$

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{4.16}$$

Mit Hilfe der Gleichungen 4.13, 4.14, 4.15 und 4.15 lassen sich nun die partiellen Ableitungen der Kosten C nach den Gewichten w und Biases b für jedes Neuron im Netzwerk berechnen. Damit lassen sich durch die Learning Rate η alle Gewichte und Biases des Netzwerks anpassen. Somit wird ein Lernprozess durch den Backpropagation-Algorithmus gewährleistet.

5 Convolutional Neural Network

Neuronale Netze werden, wie auch in dieser Arbeit, häufig zur Bild- und Mustererkennung genutzt. Dabei steht dem Netzwerk eine zweidimensionale Matrix von Pixelinformation in Form von Farb- oder Graustufeninformationen als Eingabe zur Verfügung. Aus dieser Eingabe soll nun eine Ausgabe in Form einer Klassifizierung bestimmt werden. Eine solche Zuordnung kann mit den bereits beschriebenen klassischen Neuronalen Netzen erfolgen. Der Nachteil dieser Methode ist, dass diese Art von Neuronennetzen die innere Struktur der Bilder nicht berücksichtigen können. Beispielsweise stellt eine Verschiebung des Musters innerhalb des Eingangsbildes für solche Netze ein Problem bei der Erkennung dar. Auch eine Verkleinerung oder Vergrößerung führt zu Problemen. Solche Effekte müssen zur zuverlässigen Erkennung bereits in den Trainingsdaten berücksichtigt werden. Dies führt unweigerlich zu größeren Datensätzen, die für das Training eines zuverlässigen Netzes benötigt werden. Dies könnte durch Einbeziehung der inneren Struktur des Bildes verhindert werden. Dazu stellten Yann LeCun, Léon Bottou, Yoshua Bengio und Patrick Haffner in ihrerm Paper *Gradient-based learning applied to document recognition* von 1998 sogenannte *Convolutional Neural Networks* vor.

Die dort vorgestellte Netzstruktur erlaubt die Erkennung von Strukturen unabhängig von ihrer Position im Ausgangsbild. Im folgenden Abschnitt soll das grundlegende Prinzip von Convolutional Neural Networks (CNN) vorgestellt und erläutert werden.

5.1 Faltungsoperation

Convolutional Neural Networks beruhen auf der mathematischen Faltungsoperation (Convolution). Auf abstrakter Ebene wird des zweidimensionale Eingangsbild $X(i, j)$ mit einem zweidimensionalen gewichteten Fenster $W(k, l)$ gefaltet.

$$Y(i,j) = \sum_{p=0}^{n} \sum_{q=0}^{n} X(i-p+a, j-q+a) W(p,q) \tag{5.1}$$

Gleichung 5.1 zeigt die diskrete Faltung für den genannten Fall. Dabei bezeichnet n die Zeilen- bzw. Spaltenzahl der quadratischen Matrix W und a einen von n abhängigen Parameter, der das Fenster korrekt positioniert. Bildlich gesprochen wird hier ein gewichtetes Fenster sowohl in horizontaler als auch in vertikaler Richtung über das Ausgangsbild bewegt. An jeder Position werden die jeweiligen Werte des Ausgangsbildes mit den gewichten der Fenstermatrix multipliziert und zu einem Wert addiert. Dieser Wert fließt in Convolutional Neural Networks in ein Neuron der nächsten Netzwerkschicht.

In Abbildung 5.1 ist der Vorgang für den Faltungskern an einer Position dargestellt. Dabei werden Neuronen aus dem Eingangsbild über den Faltungskern mit den entsprechenden Gewichten auf ein Hidden Neuron abgebildet.

Führt man nun die Faltung weiter, erhält man als Ergebnis erneut ein zweidimensionales Ausgangsbild mit geringerer Seitenlänge. Abbildung 5.2 zeigt diesen Schritt. Für ein 28×28-Eingangsbild und einen 5×5-Faltungskern erhält man bei einer Schrittweite von 1 beispielsweise ein 24×24-Ausgangsbild.

Bei dieser Faltungsoperation werden logischerweise für jedes Neuron die gleichen Gewichte verwendet. Im Fall des hier gezeigten Neuronalen Netzwerkes teilen sich die Ausgangsneuronen dieser Operation auch die Biases. Mit der Aktivierungsfunktion σ bestimmt sich der Output eines Neurons damit nach Gleichung **??**.

$$output = \sigma(b + \sum_{l} \sum_{m} w_{l,m} a_{j+l,k+m}) \tag{5.2}$$

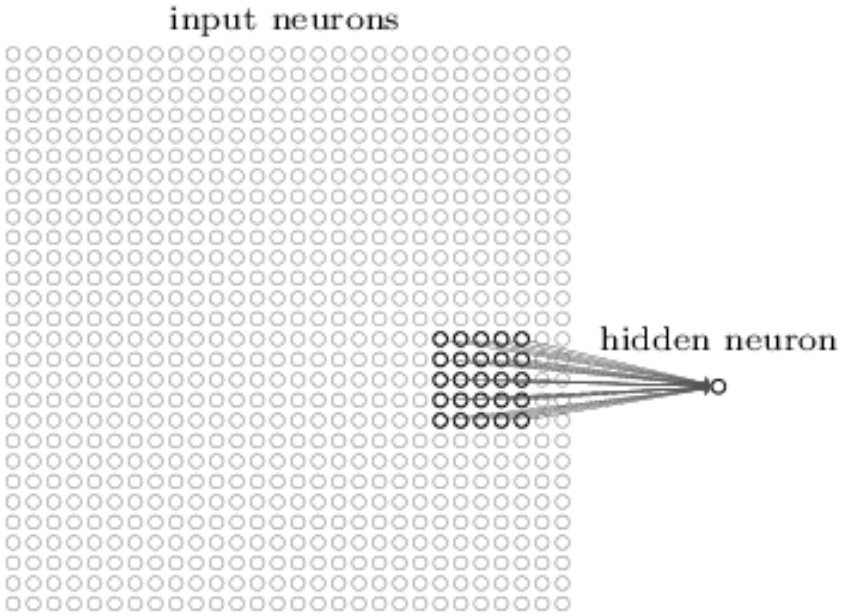


Abbildung 5.1: Berechnung eines Neurons aus einem Faltungskern (vgl. [7])

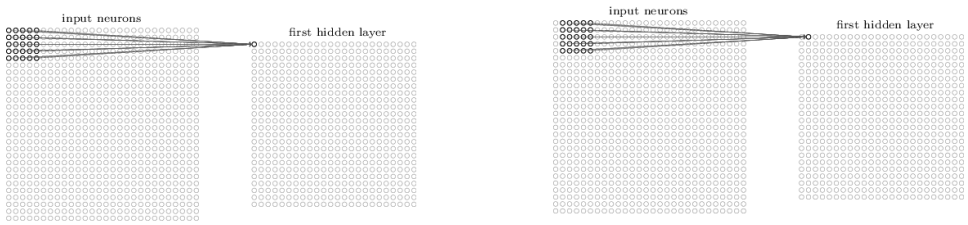


Abbildung 5.2: Faltungsoperation für zwei Ausgangsneuronen (vgl. [7])

Mit einem Satz geteilter Gewichte und Biases ist das Netz nun in der Lage, ein Strukturelement an beliebigen Positionen im Bild zu erkennen. Um es dem Netz nun zu gestatten, mehrere verschiedene Strukturen zu erkennen, vervielfacht man diesen Vorgang mit je eigenen Gewichten und Biases. Es entstehen sogenannte Featuremaps. Dieser Vorgang ist in Abbildung 5.3 dargestellt. Das Netzwerk ist somit in der Lage

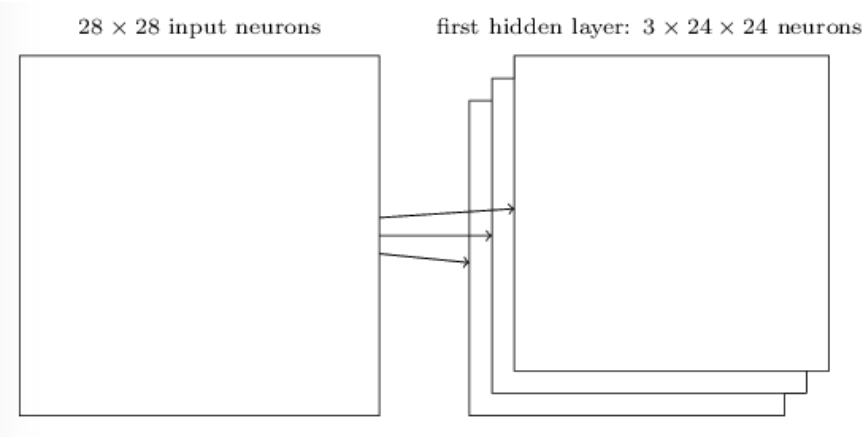


Abbildung 5.3: Darstellung verschiedener Featuremaps (vgl. [7])

verschiedene Strukturelemente an beliebigen Positionen zu erkennen. Erweitert man das Netz um weitere Schichten, kann es mit entsprechendem Training auch lernen, welche Kombinationen dieser Elemente zu einem Mustertyp gehören.

Damit erweitert sich der eingangs genannte Vorteil von CNNs. Nicht nur die Verschiebung ganzer Muster innerhalb eines Bildes kann erkannt werden, sondern auch wiederkehrende Strukturelemente können für die Erkennung anderer Muster durch entsprechende Kombination in weiteren Schichten genutzt werden.

5.2 Pooling

Die Erzeugung von vielen Featuremaps aus einer einzelnen Eingabe gehört zu den grundlegenden Eigenschaften der im vorherigen Abschnitt beschriebenen Faltungsoperation. Dies führt allerdings zu einer erheblichen Vergrößerung der Datenmenge, die von nachfolgenden Layern bearbeitet werden muss. Um das Ausmaß dieser Vergrößerung einschätzen zu können, wird an dieser Stelle die Größe des Outputs einer Faltung in Abhängigkeit von der Eingabegröße und weiterer Eigenschaften der Faltungsschicht untersucht. In den nachfolgenden Gleichungen steht S für die Größe einer einzelnen Ein- bzw. Ausgabe. Die Variablen x, y und f stehen für die Dimensionen der Ein- bzw. Ausgabe. Die Größe des Filterkernels ist durch F_x und F_y bestimmt, für die Schrittweite wird der Wert 1 angenommen.

$$\begin{aligned} S_{in} &= x_{in} y_{in} f_{in} \\ S_{out} &= x_{out} y_{out} f_{out} \\ x_{out} &= x_{in} - F_x + 1 \\ y_{out} &= y_{in} - F_y + 1 \end{aligned} \tag{5.3}$$

Unter der Bedingung, dass $x_{in} >> F_x$ und $y_{in} >> F_y$ ergibt sich die Näherung $\frac{S_{out}}{S_{in}} = \frac{f_{out}}{f_{in}}$. Die Anzahl der Neuronen einer Faltungsschicht gegenüber der des vorherigen Layers steigt bei der Faltungsoperation demzufolge annähernd proportional zur Anzahl der Featuremaps. Mit der Datenmenge im Ausgang der Faltungsschicht steigt auch die Zahl der Aktivierungen, die in den nachfolgenden Schichten bearbeitet werden müssen. Dies stellt einen beträchtlichen Mehraufwand dar und kann zu einer signifikanten Steigerung der Rechenzeit und des Speicherbedarfs führen. Um diesen Mehraufwand zu reduzieren, sind die Faltungsschichten in den meisten

Implementierungen von CNNs von sogenannten Pooling-Schichten gefolgt. Beim Pooling wird jede Featuremap in rechteckige Abschnitte unterteilt, die dann jeweils auf einen neuen Aktivierungswert abgebildet werden. Für diese Abbildung gibt es mehrere Methoden, einige davon werden im folgenden Abschnitt näher erläutert.

5.2.1 Average Pooling

Beim Average Pooling wird das arithmetische Mittel aller betrachteten Aktivierungen des Eingangs berechnet. Das Ergebnis wird direkt als Ausgang übernommen. Auf die Verwendung einer Aktivierungsfunktion wird verzichtet, da das Ergebnis durch die Verwendung des arithmetischen Mittels sowieso im für die Aktivierungsfunktion des letzten Layers üblichen Zahlenbereich liegt. Demnach gilt für die Vorwärtsberechnung eines Average Pooling Layers:

$$output_{x,y} = \frac{\sum\limits_{(m,n)\in M_x \times N_y} input_{m,n}}{|M_x \times N_y|} \tag{5.4}$$

Die Mengen M_x und N_y beinhalten dabei jeweils die x- und y-Koordinaten der Eingaben aus dem relevanten Bereich. Die Inhalte dieser Mengen hängen von x bzw. y ab, somit lässt sich jeder Position in der Ausgabe ein eigener Eingabebereich zuordnen.

5.2.2 Max Pooling

Beim Max Pooling werden die betrachteten Aktivierungen des Eingangs miteinander verglichen und der größte Wert wird unverändert im Ausgang übernommen. Auch hier wird auf die erneute Anwendung einer Aktivierungsfunktion verzichtet. Formal lässt sich Max Pooling wie folgt beschreiben:

$$output_{x,y} = max(\{input_{m,n} | m \in M_x; n \in N_y\}) \tag{5.5}$$

Die obige Gleichung wirkt zwar auf menschliche Betrachter einfach, allerdings ist sie maschinell nicht ohne Schwierigkeiten zu lösen. Der Grund dafür ist, dass zur Berechnung der max-Operation Verzweigungen nötig sind. Bei den meisten modernen CPU-Archtikturen werden diese deutlich langsamer ausgeführt als rein sequentielle Operationen.

5.3 Anpassungen der Backpropagation

Das Verhalten eines Pooling Layers weicht deutlich vom Verhalten anderer Layertypen ab. Diese Abweichung macht es erforderlich, die zur Backpropagation erforderliche Fehlerweitergabe für diesen Layertyp gesondert zu betrachten. Partielle Ableitungen der Kostenfunktion nach den Gewichten sind an dieser Stelle nicht erforderlich, da weder Average Pooling, noch Max Pooling auf Gewichte zurückgreifen. Zwar existieren auch gewichtsbehaftete Poolingverfahren, (vgl. [9]) allerdings sind diese nicht Gegenstand dieser Arbeit. Für die Backpropagation eines Pooling Layers ist es also nur erforderlich, die partielle Ableitung der Kostenfunktion nach allen Eingaben zu bestimmen. Da sich die hier behandelten Verfahren deutlich voneinander unterscheiden, werden sie im Folgenden getrennt betrachtet.

5.3.1 Average Pooling

Die Gleichung 5.6 dient als Ausgangspunkt zur Bestimmung der gesuchten partiellen Ableitung. Wie allgemein bei der Backpropagation gilt auch hier:

$$\frac{\partial C}{\partial x_i} = \sum_j \frac{\partial C}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} \tag{5.6}$$

Hier bezeichnet C das Ergebnis der Kostenfunktion, y_j alle einzelnen Aktivierungen am Ausgang des Pooling Layers und x eine beliebige Aktivierung am Eingang des Layers. Die Werte $\frac{\partial C}{\partial y_j}$ sind nach der Backpropagation in der nachfolgenden Schicht bereits bekannt. $\frac{\partial C}{\partial x_i}$ muss für jede Eingabe bestimmt werden, damit die Backpropagation im vorherigen Layer fortgesetzt werden kann. Die partiellen Ableitungn $\frac{\partial y_j}{\partial x_i}$ muss dazu noch bestimmt werden. Zur Vereinfachung wird angenommen, dass sich die rechteckigen Poolingbereiche nicht überschneiden. Dies ist bei dem in dieser Arbeit verwendeten Modell der Fall, allerdings ist es nicht allgemein erforderlich. Mit dieser Vereinfachung ergibt sich, dass es zu jedem x_i nur ein y_j gibt, das von ebendiesem x_i abhängt, für das die Ableitung also nicht 0 ist. Darüber hinaus lässt sich zu jedem x_i eindeutig ein y_j bestimmen, das weiter betrachtet werden muss. Damit reduziert sich die obige Gleichung auf:

$$\frac{\partial C}{\partial x_i} = \frac{\partial C}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} \tag{5.7}$$

Der Zusammenhang zwischen x_i und y_j ergibt sich aus der Gleichung 5.4, wobei x_i und y_j nicht den Koordinaten, sondern stattdessen den Aktivierungen $input_{m,n}$ bzw. $output_{x,y}$ entsprechen. Durch Einsetzen ergibt sich also:

$$y_j = \frac{\sum\limits_k x_k}{|M \times N|} \tag{5.8}$$

Der Betrag des Kreuzprodukts im Nenner entspricht der Anzahl der Aktivierungen des mit y_j betrachteten Clusters im Eingang. Wird nun die partielle Ableitung $\frac{\partial y_j}{\partial x_i}$ gebildet, so fallen alle Summanden außer $x_k mit k = i$ weg. Dieser Summand wird durch den konstanten Betrag der Clustergröße dividiert, somit ergibt sich für die Ableitung

$$\frac{\partial y_j}{\partial x_i} = \frac{1}{|M \times N|} \tag{5.9}$$

Durch Einsetzen in Gleichung 5.7 ergibt sich damit die finale Gleichung zur Backpropagation beim Average Pooling:

$$\frac{\partial C}{\partial x_i} = \frac{\partial C}{\partial y_j} \cdot \frac{1}{|M \times N|} \tag{5.10}$$

Wird die Eingabe also in gleich große Cluster eingeteilt, so lässt sich der Fehler aller Aktivierungen im Eingang bestimmen, indem der Fehler der zugehörigen Ausgangsaktivierung duch die konstante Clustergröße dividiert wird.

5.3.2 Max Pooling

Wie zuvor beim Average Pooling gilt auch hier die Gleichung 5.6. Außerdem wird wieder angenommen, dass sich die Cluster nicht überschneiden, womit auch Gleichung 5.7 ohne Änderungen auf Max Pooling übertragen werden kann. Die Ableitung $\frac{\partial y_j}{\partial x_i}$ muss allerdings neu bestimmt werden, da sich die beiden Verfahren bei der Vorwärtsberechnung unterscheiden. Die Vorwärtsberechnung für Max Pooling ist bereits durch Gleichung 5.5 formalisiert. Zunächst wird diese Gleichung unter Verwendung der Variablen x_i und y_j neu formuliert. Damit ergibt sich:

$$y_j = max(\{x_k | k \in (M_j \times N_j)\}) \tag{5.11}$$

Der Index k steht in dieser Gleichung für alle möglichen Positionen von Eingaben in dem für die Ausgangsaktivierung y_j relevanten Cluster. Wie sich erkennen lässt, hat nur der größte betrachtete Eingang einen Einfluss auf das Ergebnis. Dies macht eine Fallunterscheidung bei der Bestimmung der Ableitung $\frac{\partial y_j}{\partial x_i}$ notwendig:

$$\frac{\partial y_j}{\partial x_i} = \begin{cases} 1 & \text{für } x_i = max(\{x_k | k \in (M_j \times N_j)\}) \\ 0 & \text{sonst} \end{cases} \tag{5.12}$$

Dies lässt sich in die zuvor bestimmte Gleichung 5.7 einsetzen. Dadurch ergibt sich die finale Gleichung zur Backpropagation beim Max Pooling:

$$\frac{\partial C}{\partial x_i} = \begin{cases} \frac{\partial C}{\partial y_j} & \text{für } x_i = max(\{x_k | k \in (M_j \times N_j)\}) \\ 0 & \text{sonst} \end{cases} \tag{5.13}$$

Wie zu erkennen ist, wird hier das Ergebnis der Vorwärtsoperation erneut verwendet. Die Position des Maximums kann daher zwischengespeichert werden, um den Aufwand für eine erneute Berechnung einzusparen.

5.4 Vergleich

Sowohl Average Pooling, als auch Max Pooling erfüllen die Hauptanforderungen an eine Pooling-Schicht, nämlich eine Datenreduktion um einen durch die Clustergröße bestimmbaren Faktor. Unterschiede zwischen den beiden Verfahren zeigen sich in der Effizienz der Berechnung und der Qualität des Ergebnisses.

5.4.1 Effizienz

Die Gleichung der Max-Pooling-Operation beinhaltet einen Operator zur Bestimmung des Maximums aus einer Eingabemenge. Wie bereits in Abschnitt 5.2.2 erwähnt, ist die maschinelle Berechnung dieses Operators sehr ineffizient. Beim Average Pooling sind zwar mehr Rechenoperationen erforderlich, allerdings lassen sich diese mithilfe moderner CPUs und GPUs deutlich schneller berechnen. Dies ist möglich, weil viele Architekturen Möglichkeiten bereitstellen, gleichartige Operationen gleichzeitig mit einer größeren Menge von Operanden durchzuführen.

Darüber hinaus benötigt ein Max Pooling Layer zusätzlichen Speicher, um die Position des im Forward-Schritt ermittelten Maximums bis zur Backpropagation zwischenzuspeichern. Bei Prozessoren, die einen Cache besitzen, erhöht zusätzlicher Speicherbedarf die Wahrscheinlichkeit, dass im Cache abgelegte Daten überschrieben werden und erneut aus dem langsameren Arbeitsspeicher gelesen werden müssen.

Zusammenfassend lässt sich daher feststellen, dass die Berechnung von Average Pooling im Gegensatz zu Max Pooling auf den in dieser Arbeit verwendeten Architekturen performanter ausgeführt wird. Wie groß der tatsächliche Unterschied ist, hängt stark von den verwendeten Prozessoren und Implementierungen ab und lässt sich somit nicht allgemein feststellen.

5.4.2 Qualität

Die Qualität des Ergebnisses einer Pooling-Operation hängt nicht nur von der Art der Operation selbst, sondern auch von der Art und Aufgabe des Modells, sowie den Eingabedaten ab. Tensorflow verwendet in seinen Beispielen zur Ziffernerkennung Max Pooling, weil sich damit für diesen Anwendungsfall erfahrungsgemäß eine höhere Genauigkeit erzielen lässt. (vgl. [10])

Grund für das unterschiedliche Verhalten von Max Pooling und Average Pooling ist, dass sich die Ausgaben der beiden Verfahren bezüglich ihres Kontrastverhaltens unterscheiden. In den meisten Fällen geht dem Pooling Layer ein Convolution Layer voraus. Bei der Faltungsoperation wird nach bestimmten Bildabschnitten gesucht, die durch einen oder mehrere Filterkernel definiert sind. Wenn der gesuchte Bildabschnitt genau unter dem Filterkernel liegt, schlägt die Aktivierung des Perceptrons für diese Position stark aus. Bereits bei geringen Abweichungen passiert dies allerdings nicht mehr. Stattdessen stellt sich an diesen Positionen ein scheinbar zufälliges Rauschen ein. Bei der nachfolgenden Pooling-Operation sollen die starken Ausschläge möglichst erhalten bleiben, weil darin die relevantesten

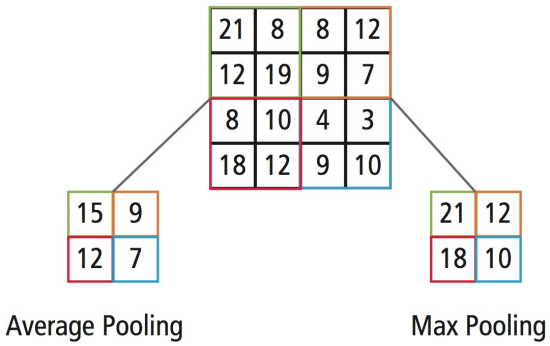


Abbildung 5.4: Vergleich von Average Pooling und Max Pooling (vgl. [11])

Informationen enthalten sind. Beim Average Pooling werden nun die Featuremaps ähnlich wie mit einem Glättungsfilter derart verarbeitet, dass der Kontrast abnimmt. Somit nähern sich die Ausschläge und das Rauschen signifikant an, wodurch das Ergebnis des Poolings für nachfolgende Layer schwerer zu verarbeiten wird. Beim Max Pooling werden positive Ausschläge unverändert weitergegeben, somit sind sie auch in nachfolgenden Layern deutlich vom Rest der Featuremap zu unterscheiden. Negative Ausschläge können beim Max Pooling dagegen nicht erkannt werden, da in diesem Fall das Aktivierungsniveau der Umgebung höher ist und als Maximum weitergegeben wird. Bei der Verwendung von Max Pooling kann das Netz demnach nicht darauf trainiert werden, relevante Informationen durch negative Ausschläge darzustellen. Es ist vom Modellaufbau und der Art des Anwendungsfalls abhängig, ob dieser Informationsverlust durch die deutlichere Erkennung von positiven Ausschlägen kompensiert werden kann. (vgl. [9])In Abbildung 5.4 wird das Verhalten der beiden Poolingverfahren durch ein kleines Zahlenbeispiel visualisiert. Aufgrund der bereits in vorhergenhenden Arbeiten festgestellten besseren Qualität der Ergebnisse bei der Klassifizierung von Ziffern aus der MNIST-Datenbank (vgl. [10]) und zur besseren Vergleichbarkeit mit anderen Implementierungen wird in dieser Arbeit Max Pooling verwendet.

6 Serielle Implementierung

Im ersten Teil dieser Arbeit soll eine rein seriellen Implementierung eines Convolutional Neural Network erstellt werden. Als Vorlage dient ein von Google veröffentlichtes Python-Skript, das die Bibliothek Tensorflow verwendet. Ziel dieser ersten Implementierung ist nicht, eine produktiv anwendbare Software zu erstellen, sondern eine in C++ geschriebene Vorlage für die später zu erstellenden optimierten Implementierungen zu schaffen.

6.1 Tensorflow

TensorFlow ist eine von Google entwickelte und veröffentlichte Bibliothek für hochperformante numerische Berechnungen. Anwendung findet TensorFlow unter anderem in den Bereichen Machine Learning und Deep Learning. Dazu gehört auch die Erkennung handgeschriebener Ziffern mithilfe eines Convolutional Neural Network. APIs zur Interaktion mit TensorFlow existieren in mehreren Programmiersprachen, am weitesten verbreitet ist Python. Die eigentlichen Berechnungen werden allerdings von einer vorkompilierten Bibliothek durchgeführt, um Overhead von interpretierten Sprachen oder mangelhafter Compileroptimierung zu vermeiden. (vgl. [12])

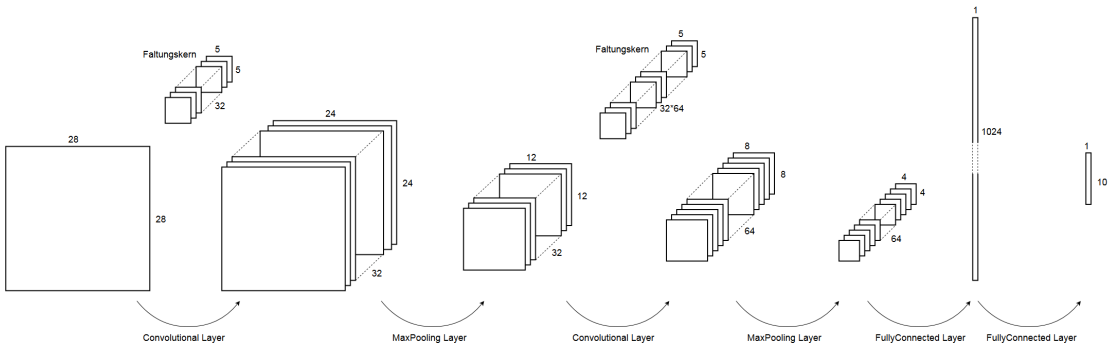


Abbildung 6.1: Schematische Darstellung des Netzaufbaus in der seriellen Implementierung

6.2 Netzaufbau

Der für die serielle Implementierung verwendete Netzaufbau wird von einem Python-Skript übernommen, das Google mit der offiziellen Dokumentation von Tensorflow veröffentlicht hat. (vgl. [10]) Die Abfolge der verwendeten Schichten, sowie einige ihrer Eigenschaften sind in Abbildung 6.1 schematisch dargestellt. Wie darin zu sehen ist, besteht das Netz aus 6 Schichten. Zwei Paare aus je einem Convolutional Layer und einem Max Pooling Layer werden von zwei Fullyconnected Layern gefolgt. Für die in der Vorlage verwendeten Filtergrößen und Featurezahlen ist bekannt, dass sie zu einer Testgenauigkeit von etwa 98 % führen sollten. (vgl. [10]) Aufgrund einiger Abweichungen von der Vorlage kann die Erwartungshaltung an die Genauigkeit allerdings nicht ohne Vorbehalte auf den Netzaufbau in dieser Arbeit übertragen werden. Grund dafür sind einige Abweichungen des Aufbaus von der Vorlage. Als Aktivierungsfunktion wird in dieser Arbeit sigmoid verwendet. Außerdem wird beim Trainieren in der Vorlage eine Technik namens Dropout Regularization eingesetzt. Diese Technik steigert die erreichbare Testgenauigkeit, indem sie einen störenden Effekt namens Overfitting vermindert. (vgl. [6])

6.3 Systembeschreibung

Das Programm der seriellen Implementierung muss folgende Aufgaben erfüllen:

Trainings- und Testdaten: Die zu verarbeitenden Bilder müssen gemeinsam mit den Labels zur Laufzeit von der Festplatte eingelesen werden.

Netzwerkmodell: Das Programm muss ein internes Modell des Netzwerks besitzen, durch das festgelegt wird, wie die Bilder zu verarbeiten sind. Außerdem sind im Netzwerkmodell die veränderlichen Gewichte angesiedelt.

Training: Beim Training müssen die Eingabedaten das Netzwerk zunächst vorwärts durchlaufen. Anschließend wird zur Fehlerberechnung eine Backpropagation durchgeführt und die Gewichtsanpassung mittels Gradient Descent vorgenommen. Der Trainingsprozess wird für alle Trainingsdaten mehrfach wiederholt. Das einmalige Trainieren mit jedem Bild der Trainingsdaten wird als Epoche bezeichnet.

Test: Zum Testen werden von den Trainingsbildern verschiedene Eingaben verwendet. Diese durchlaufen das Netzwerk zunächst in Vorwärtsrichtung. Anschließend werden die Ergebnisse mit den Labels verglichen. Die Abweichung des Ergebnisses vom erwarteten Wert gemäß des labels ist ausschlaggebend für die Kosten. Die Genauigkeit entspricht dem Anteil der Testfälle, für die die Eingabe gemäß der Erwartungen basierend auf dem Wert des Labels richtig klassifiziert wurde.

Gemäß dieser Erwartungen wird zunächst eine grobe Architektur entworfen, die in Abbildung 6.2 durch ein Klassendiagramm visualisiert wird. Die im Diagramm 6.2 dargestellten Klassen sind gemäß ihrer Aufgabenbereiche voneinander abgegrenzt. Eine nähere Erläuterung der Einsatzzwecke dieser Klassen erfolgt an dieser Stelle nach dem Bottom-Up-Ansatz:

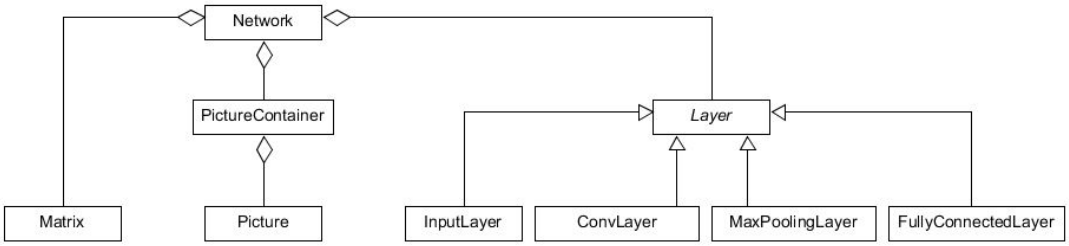


Abbildung 6.2: Klassendiagramm der seriellen Implementierung

Matrix: Diese Klasse repräsentiert eine Matrix von beliebiger Größe. Bei der Erstellung wird die Größe der Matrix an den Konstruktor übergeben, woraufhin dieser einen entsprechend großen Speicherbereich alloziert. Die Klasse stellt Operationen zur Verfügung, mit denen Matrizen addiert, multipliziert und skaliert werden können. Außerdem können sie mit zufälligen oder vorgegebenen Werten gefüllt werden.

Picture: Jede Instanz der Klasse `Picture` repräsentiert jeweils ein Eingabebild gemeinsam mit einem Vektor, der dem Wunschwert für die Ausgabe entspricht. Die Ein- und Ausgabedaten werden jeweils als Array innerhalb des Objekts gespeichert. Es lassen sich Matrix-Objekte erzeugen, die auf den Speicherbereich der Arrays innerhalb eines `Picture`-Objekts verweisen. Auf diese Weise können die im `Picture` enthaltenen Informationen leicht zur Weiterverarbeitung in Layern konvertiert werden.

PictureContainer: Der `PictureContainer` repräsentiert eine Ansammlung von `Picture`-Objekten. Es ist vorgesehen, dass sowohl für die Trainingsdaten, als auch für die Testdaten jeweils eine Instanz dieser Klasse existiert. Ein `PictureContainer` beinhaltet zudem eine Funktion, durch deren mehrfachen Aufruf über alle enthaltenen Bilder iteriert werden kann. Um den Speicher-

bedarf zu begrenzen, hält ein `PictureContainer` normalerweise 1000 `Picture` -Objekte im Arbeitsspeicher. Wenn über alle Bilder iteriert wurde, wird automatisch eine Datei von der Festplatte ausgelesen, die weitere Ein- und Ausgabedaten zum erzeugen neuer `Picture`-Objekte bereitstellt.

Layer: Diese abstrakte Klasse dient als Basis für alle Layertypen. Darin enthalten sind Informationen, die das Verhalten und die Eigenschaften einer Schicht beschreiben. Die von `Layer` abgeleiteten Klassen können zusätzliche Eigenschaften enthalten, die für den jeweiligen Layertyp spezifisch sind. Veränderliche Gewichte und Aktivierungen sind in den Objekten dieser Klasse nicht enthalten.

Network: Dies ist die zentrale Klasse des seriellen Programms. Zur Laufzeit existiert genau ein Objekt dieser Klasse. Dieses verwaltet die `PictureContainer` mit den Trainings- und Testdaten, eine Liste von Layer-Objekten zur Definition des Netzwerkmodells und mehrere Listen von `Matrix`-Objekten zur Speicherung aller veränderlichen Gewichte, Aktivierungen und Fehlergrößen. In dieser Klasse sind auch die Methoden zur Forward- und Backpropagation, zur Gewichtsanpassung und zum Trainingsablauf definiert.

main: Die Main-Funktion erzeugt ein Objekt der Klasse `Network` und initialisiert die Liste der Layer, um das Netzwerkmodell festzulegen. Anschließend ruft es die Trainings- und Testmethoden des Netzwerks auf.

6.4 Beschreibung der Problematik einer seriellen Implementierung

Prinzipiell sollte das oben beschriebene Programm bei fehlerfreier Implementierung bereits in der Lage sein, anhand der Trainingsdaten das Klassifizieren von handgeschriebenen Ziffern automatisiert zu erlernen. Eine signifikante Steigerung der Erkennungsgenauigkeit konnte jedoch auch nach längeren Testläufen nicht nachgewiesen werden. Der wahrscheinlichste Grund dafür sind unentdeckte Fehler in der Implementierung. Aber selbst mit einer korrekten Implementierung könnte die damit erkennbare Erkennungsgenauigkeit nicht in annehmbarer Zeit überprüft werden.

Bei einem Testlauf wurde mithilfe des in Visual Studio 2015 enthaltenen Diagnosetools Äpplication Timeline"(vgl. [13]) für die Bearbeitung eines Bildes (Forward- und Backpropagation) eine durchschnittliche Laufzeit von 6180 ms festgestellt. Bei dem im Testrechner verbauten Prozessor handelt es sich um einen AMD A10-8700P, der laut einem Whetstone-Benchmark unter idealen Bedingungen einen Singlethread-Durchsatz von 1,85 GFLOPS erreichen kann. (vgl. [14]) Bei konstanter Laufzeit pro verarbeitetem Bild und unter Vernachlässigung der Laufzeit für die Gewichtsanpassung und das Einlesen der Bilddaten von der Festplatte würde eine Trainingsepoche mit dem vollständigen Trainingsdatensatz von 55000 Bildern etwas länger als 94 Stunden dauern.

Mit einer weiteren Optimierung des zugrundeliegenden Programms ließe sich die Laufzeit noch deutlich verringern, allerdings stößt man mit einer rein seriellen Verarbeitung früher oder später zwangsläufig auf physikalische Grenzen: Eine Auswertung des Netzwerkmodells zur Zählung der Rechenoperationen ergibt, dass 9947718 Berechnungen mit Fließkommazahlen notwendig sind, damit ein Datensatz das Netzwerkmodell vollständig vorwärts und rückwärts durchläuft. 99,5 % davon

sind Multiplikationen, Additionen und Größenvergleiche. Aufwändigere Operationen wie Divisionen und Exponentialfunktionen machen ein halbes Prozent dieser Zahl aus. Nicht in diese Zahl eingerechnet ist der Overhead, den ein Programm zusätzlich betreiben muss, um diese Fließkommaoperationen durchführen zu können. Dazu gehört unter anderem der Kontrollfluss des Programms, also Funktionsaufrufe, Verzweigungen und Schleifen. Auch durch ungecachete Speicherzugriffe und Adressberechnungen bei Arrayzugriffen geht Zeit verloren, die nicht für das eigentliche Training des Netzwerks genutzt werden kann. Selbst unter der Annahme, dass ein serielles Programm diesen Overhead vollständig eliminieren könnte und nur noch die notwendigen Berechnungen zur Simulation des Netzwerks ausführt, ist noch ein erheblicher Aufwand von beinahe zehn Millionen Fließkommaberechnungen erforderlich. Weiter angenommen, man würde dieses Programm unter idealen Bedingungen auf einem Intel i7-7700K mit einem Singlethread-Durchsatz von 5,65 GFLOPS (vgl. [14]) laufen lassen, so würde sich theoretisch eine Verarbeitungszeit von 1,76 ms pro Bild ergeben. Eine vollständige Trainingsepoche mit 55000 Bildern (ohne Berücksichtigung des Gradient Descent und des Einlesens der Bilder) ließe sich dann in 96,84 Sekunden bewerkstelligen. Dies klingt zwar annehmbar, allerdings sind diese Werte in der Praxis nicht annähernd erreichbar. Darüber hinaus ist das in dieser Arbeit betrachtete Netzwerk sehr klein und und verarbeitet vergleichsweise kleine Eingaben mit einer Größe von 28×28 Pixeln. Moderne CNNs besitzen häufig deutlich mehr Schichten und verarbeiten größere Eingaben, womit sich auch die Zahl der Fließkommaoperationen vervielfacht.

Um die Performance über die Grenzen einer seriellen Implementierung hinaus zu steigern, ist eine parallele Ausführung der Berechnungen notwendig. Neuronale Netzwerke lassen sich prinzipiell sehr gut parallelisieren, da sich die Aktivierungen der Perceptronen innerhalb einer Schicht nicht beeinflussen und sich daher auch voneinander getrennt berechnen lassen.

7 Implementierung auf einer General Purpose x86 CPU

Unter einer General Purpose x86 CPU werden Prozessoren verstanden, die den Befehlssatz und die Architektur x86 verwenden. Diese Architektur wird von den beiden großen CPU Herstellern Intel und AMD verwendet und befindet sich somit in nahezu allen PCs und Laptops.

Heutige CPUs dieser Hersteller verstehen wesentlich mehr Instruktionen als der klassische x86-Befehlssatz. Somit bestehen heute einige Möglichkeiten, den Programmcode für diese Architektur zu optimieren. Eine dieser Optimierungsmöglichkeiten ist die Verwendung von SIMD-Instruktionen (Single Instruction, Multiple Data). Damit können mehrere Rechenoperationen gleichzeitig in speziellen Registern ausgeführt werden. Eine weitere Optimierungsmöglichkeit ergibt sich aus der Tatsache, dass CPUs heute aus mehrere CPU Kernen bestehen. Dadurch können Instruktionen gleichzeitig auf mehreren Kernen parallel ausgeführt werden.

Zur Entwicklung des Programms wurde Eclipse CDT verwendet und als Compiler gcc, ein weit verbreiteter Compiler mit einer großen Open Source Comunity. Dieser kann mit sehr vielen Flags aufgerufen werden, um den Compiliervorgang optimal einzustellen.

Ein optimiertes Neuronales Netzwerk auf der General Purpose x86 CPU wurde im Rahmen dieser Arbeit nicht erfolgreich Implementiert. Alle Funktionen eines Neuronalen Netzwerkes wurde zwar implementiert und bei einer geringen Menge von Trainingsdaten werden diese nach vielen Iterationen auch richtig erkannt. Das Erkennen von neuen Daten ist jedoch selbst nach einer langen Zeitspanne zum Trainieren nicht möglich und die Iterationen zum Trainieren des Netzwerkes sind um ein vielfaches höher als von anderen Implementierungen des selben Netzes. Dabei muss es sich um einen Fehler in der Implementierung handeln, der bis zum Ende der Zeit für die Studienarbeit nicht gefunden werden konnte. Die Suche nach dem Fehler war sehr schwer, da alle trainierten Werte des Neuronalen Netzwerk stimmen könnten und anhand der Werte nicht auf den Fehler geschlossen werden kann, was im Grundprinzip eines neuronalen Netzwerkes liegt. Die Zahlenwerte sind für einen Menschen nichtssagend und könnten alle plausibel sein. Durch die langwierige Fehlersuche wurde der Code nicht ausreichend für die Zielhardware optimiert, sodass der Code nicht performant läuft.

7.1 OpenMP (Open Multi Procesing)

Aktuelle CPUs, ob im Arbeitsrechner, Laptop, Tablet oder Smartphone, besitzen inzwischen mehrere CPU-Kerne. Dies liegt daran, dass in den letzten Jahren der Mehrgewinn neuer CPUs nicht mehr durch eine erhöhten Taktfrequenz, wie in den Jahrzehnten zuvor, sondern durch eine parallele Verarbeitung von Daten erzielt wurde. Um das volle Potenzial dieser CPUs auszunutzen ist es zwingend notwendig, Programme zu parallelisieren, sodass Programmabschnitte auf mehreren Kernen

gleichzeitig ausgeführt werden können. „Unter der Parallelisierung eines Programms versteht man, dass mehrere Teile einer Aufgabe gleichzeitig nebeneinander ausgeführt werden, um so die Gesamtaufgabe schneller als bei strikt serieller Verarbeitung zu beenden."[15]

Mit der Programmierschnittstelle OpenMP ist es relativ einfach Programme, mithilfe von Direktiven die in den Programmcode eingefügt werden, parallel ablaufen zu lassen. OpenMP ist hierbei für die Programmiersprachen C, C++ und Fortran verfügbar.

7.1.1 Prozesse und Threads

Äls Prozess bezeichnet man ein Programm, das gerade vom Betriebssystem ausgeführt wird."[15] Ein Prozess wird auch als aktive Instanz eines Programmcodes bezeichnet. Systeme, die mehrere Prozesse scheinbar gleichzeitig ausführen können, werden als Multitasking Systemen bezeichnet. Eine tatsächliche parallele Ausführung ist jedoch oft nicht möglich, stattdessen werden verschiedene Prozesse nacheinander in kurzen Intervallen ausgeführt. Der Prozess Scheduler bestimmt hierbei, welcher Prozess wann ausgeführt wird.

Es gibt verschiedene Scheduling-Strategienum um zu bestimmen, wie lange und wann ein Prozess auf der CPU ausgeführt wird. Zu den bekanntesten gehört "First-Come First-Serve", SShortest Job Firstünd Round Robin. Bei "First-Come First-Serve"werden alle Prozesse hintereinander in eine Warteschlange gestellt und müssen darauf warten, bis alle vorherigen Prozesse ausgeführt wurden. Bei der Strategie SShortest Job First"werden Prozesse, die nur kurze Ausführungszeiten benötigen, zuerst ausgeführt. Dies führt zu einer verringerten durchschnittlichen Wartezeit, Prozesse die eine lange Ausführungszeit haben brauchen aber sehr lange um ausgeführt zu werden. Beim "Round Robin"Verfahren werden lange Prozesse in mehrere Teilstücke aufgeteilt. Nacheinander werden von jedem Prozess

ein Teilstück abgearbeitet. Dadurch werden Prozesse, die viel Ausführungszeit benötigen, schneller abgearbeitet als bei SShortest Job Firstünd Prozesse mit geringer Ausführungsdauer müssen nicht so lange warten wie bei "First-Come First-Serve". Neue Betriebssysteme arbeiten mit komplexeren Scheduling-Strategien, in denen Prozesse unterschiedliche Prioritäten bekommen und sich unterschiedlich "nett"gegenüber dem Prozessor verhalten können. [16]

Ein Prozess besteht aus folgenden Komponenten:

- einer eindeutigen Prozess-ID
- dem auszuführenden Programmcode mit aktuellen Wert des Programmschritt-zählers
- den aktuellen CPU-Registerwerten
- dem Stack zur Ausführung des Programms
- einen Datenbereich mit den globale Variablen
- dem Heap mit den dynamisch angelegte Variablen

[15]

Wird der Prozess, der zurzeit bearbeitet wird, gewechselt, müssen alle zu diesem Prozess gehörenden Daten gesichert werden. Der Bereich, in dem die Daten der einzelnen Prozesse liegen, wird Prozesskontrollblock genannt.

În modernen Betriebssystemen kann ein Prozess über mehrere Ausführungsstränge oder Threads verfügen. Ein Thread kann als „Light-Version“ eines Prozesses betrachtet werden."[15] Mehrere Thread können sich einen Datenspeicherbereich, den Heap und andere Elemente eines Prozesses teilen. Dadurch kann der Wechsel zwischen Threads viel schneller erfolgen, als der Wechsel zwischen Prozessen. Des weiteren können Threads, durch den gemeinsamen Speicherbereich, auch tatsächlich

parallel, auf mehreren CPU Kernen, bearbeitet werden. OpenMP ermöglicht die einfache Nutzung dieser Technik. Es muss sich mit OpenMP nicht um die richtige Initialisierung oder das Erstellen und Beenden von Threads Gedanken gemacht werden. Stattdessen werden im Programmcode durch Compilerdirektiven einfach zu verstehende Anweisungen gegeben und dem Rest OpemMP überlassen. [15]

7.1.2 Programmiermodell

Die nebenläufige Abarbeitung von Programmteilen wird bei OpenMP durch das Fork-Join-Prinzip erzielt. Dabei erfolgt an einer bestimmten Stelle im Programmcode ein sogenannter Fork. An dieser Punkt erzeugt der Thread, der den vorherigen Programmcode ausgeführt hat, neue Threads, die den danach folgenden Code parallel ausführen. Der Thread, der die anderen Threads erzeugt hat, wird Master-Thread genannt. Am Ende des parallelen Bereichs geschieht ein Join. Bei einem Join werden alle Treads synchronisiert und beendet. Lediglich der Master-Thread bleibt erhalten und führt das weitere Programm seriell aus. Zum Ausführen des Joins, müssen alle Threads die Ausführung des Codes beendet haben.

OpenMP erlaubt es, einen Fork auch innerhalb eines parallelen Programmabschnittes zu tätigen. Der außerhalb liegende Join muss dann auf alle innere Joins warten, also bis alle innerhalb des parallelen Codes erzeugten Threads beendet wurden.

7.1.3 OpenMP-Direktiven

Im folgenden Abschnitt sind alle Programmbeispiele in C- oder C++-Syntax gehalten. Die selben Anweisungen sind auch in Fortran möglich, können sich jedoch in der Erscheinungsweise unterscheiden.

Um Funktionen der Laufzeitbibliothek zu verwenden, muss die Datei `omp.h` in das C++-Projekt includiert werden. Es ist auch möglich, Anweisungen von OpenMP ohne die Verwendung der Laufzeitbibliothek zu verwenden, jedoch nicht mit jedem Compiler. Es wird deswegen empfohlen, die Bibliothek immer in das Projekt einzubinden [15]. Um dem Conpiler gcc anzuweisen, dass er den Code unter Berücksichtigung von OpenMP übersetzen soll, muss das Compilerflag `-fopenmp` beim compilieren angegeben werden.

Mit der Compiler-Direktiven `#pragma omp parallel` kann ein Codeabschnitt parallel abgearbeitet werden. Der Codeabschnitt wird von geschwungenen Klammern umschlossen. Die Klammer muss hierbei in der sich unterhalb der Direktiven befindenden Zeile eingefügt werden. Eine Geschwungene Klammer in der selben Zeile, hinter der Compilerdirektive, ist nicht zulässig [15]. Von Compilern, die OpenMP nicht unterstützen, wird die Zeile mit der Direktive ignoriert. Da von Compilern, die OpenMP nicht unterstützen, die Bibliothek `omp.h` nicht gefunden wird, führt dies an der Stelle im Quellcode zu einem Fehler. Mit dem unterhalb stehenden Code wird die Bibliothek nur dann geladen, wenn der Compiler OpenMP unterstützt. Somit kann der gleiche Programmcode als paralleles oder als serielles Programm compiliert werden.

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

Listing 7.1: Includieren der Bibliothek omp.h

Alle Direktiven in OpenMP sind nach diesem Muster aufgebaut `#pragma omp <Direktive> [Klausel [[,] Klausel ] ...]`

7.1.4 Variablen

Variablen können entweder privat oder gemeinsam innerhalb eines parallelen Abschnitts deklariert werden.

Eine gemeinsame Variable kann von verschiedenen Threads genutzt werden, während eine private Variable für jeden Thread einen anderen Wert annehmen kann. Gemeinsame Variablen werden mit dem Parameter `shared(Liste_der_Variablen)` deklariert. In der Liste werden alle Variablen hineingeschrieben, die gemeinsam genutzt werden sollen. Die Variablen sind mit dem Wert initialisiert, die sie vor dem Fork hatten.

Private Variablen können hingegen mit `private(Liste_der_Variablen)` deklariert werden. Anders als die Gemeinsamen Variablen sind diese anfangs nicht initialisiert, mit Ausnahme von Objekten die einen Konstruktur besitzen und Indexvariablen von Schleifen. Alle anderen privaten Variablen müssen im parallelen Code initialisiert werden.

Soll der Wert, den die Variable vor Beginn der parallelen Ausführung hatte, im parallelen Abschnitt verwendet werden, muss dafür der zusätzliche Parameter `firstprivate(Liste_der_Variablen)` verwendet werden. Der letzte Wert vor Ausführung des Forks wird dann als Anfangswert der privaten Kopien verwendet.

Soll der Wert der Variable nach dem Join im weiteren Programmablauf verwendet werden, muss hierfür der zusätzliche Parameter `lastprivate(Liste_der_Variablen)` verwendet werden. Dabei wird der Wert, den die Variable bei serieller Ausführung am Ende des Abschnitts hätte als Wert für die Variable nach dem Join verwendet. Es ist auch möglich, dass eine Variable in beiden dieser zusätzlichen Parameter enthalten ist.

Eine weitere Möglichkeit der Deklaration von Variablen ergibt sich durch die Reduktionsvariablen. Diese funktionieren wie private Variablen, bis zum Zeitpunkt des Joins. Bei Reduktionsvariablen werden alle Werte der privaten Variablen durch eine Operation zusammengefasst. Bei dieser Operation kann es sich um die Addition, Subtraktion, Multiplikation, XOR, binäre und logische Und-Verknüpfung oder binäre und logische Oder-Verknüpfung handeln. Der Parameter für Reduktionsvariablen lautet `reduction( op:Liste_der_Variablen)`.

„Wird eine Variable in keiner der Deklarationen erwähnt, wird sie standardmäßig als gemeinsame Variable deklariert. Eine Ausnahme hiervon bildet die bei der Aufteilung einer Schleife auf mehrere Threads erzeugte Index-Variable der Schleife, die standardmäßig als private Variable deklariert wird. Soll ein explizites Deklarieren jeder im parallelen Bereich verwendeten Variable erzwungen werden, so wird der Direktive zusätzlich der Parameter default(none) hinzugefügt.“ (Uni München) Das Default-Verhalten lässt sich jedoch auch ändern, dafür gibt es die Datenzugriffsklausel `default(zugriffsverhalten)`. Hier kann auch der Wert `none` angegeben werden. Dann gibt es für jede Variable, für die nicht explizit das Zugriffsverhalten angegeben wurde, beim compilieren einen Fehler. Dies kann für Testzwecke in den Code eingebaut werden, um sich für jede Variable explizit Gedanken zu machen. Variablen können nur in einer Datenzugriffsklausel vorkommen, also nicht gleichzeitig privat und shared deklariert sein.

7.1.5 Parallele Abarbeitung

Parallele Abschnitte des Programms beginnen immer mit der Direktiven `#pragma omp parallel`. Der sich innerhalb der geschwungenen Klammern befindende Code wird dann vom Compiler, wenn er OpenMP unterstützt, als parallelen Code übersetzt. Der Auszuführende Code ist hierbei, ohne Angabe weiterer Direktiven, in jedem Thread derselbe.

Die Nummer des Threads, der den Aktuellen Code ausführt, kann mit `omp_get_thread_num` ermittelt werden. Die Gesamtanzahl der Threads wird bei Aufruf von `omp_get_num_threads` zurückgegeben.

Die bei der Implementierung des Neuronalen Netzwerkes am meisten verwendete Direktive von OpenMP ist `#pragma omp for [Parameter]`. Diese muss vor eine For-Schleife innerhalb eines parallelen Abschnittes gesetzt werden. Die For-Schleife wird dann parallel abgearbeitet, wobei jeder Thread nur eine Teilmenge der Iterationen abarbeitet. Eine Bedingung zur Verwendung dieser Direktiven ist, dass die Schleife in kanonischer Form vorliegt. Für die kanonische Form muss die Anzahl der Durchläufe dafür schon vor Beginn des ersten Durchlaufs berechenbar sein. Außerdem darf kein Ergebnis eines Schleifendurchlaufs zur Berechnung eines Ergebnis in einem anderen Durchlauf benötigt werden. Die Operation zum Vergleichen der Zählvariable darf nur `>, >=, <` oder texttt<= annehmen und die Zählvariable darf nicht durch eine Multiplikation oder Division erhöht oder verringert werden

Falls sich in einem parallelen Abschnitt nur eine for-Schleife befindet, kann die Direktive zusammengefasst werden. Die Anweisung heißt dann `#pragma omp parallel for`. Die geschwungenen Klammern um den parallelen Code entfallen hierbei, da die for-Schleife bereits durch geschwungene Klammern ein Codesegment umschließt. Die zwei unten stehenden Codes bewirkt somit dasselbe.

```
#pragma omp parallel
{
    #pragma omp for
    for (...) {
    }
}
```

Listing 7.2: For innerhalb eines parallelen Abschnitts

```
#pragma omp parallel for
for (...) {
}
```

Listing 7.3: Kurzschreibweise einer parallelen For-Schleife

Als Parameter für die Direktive `#pragma omp for` kann die Schedule-Strategie angegeben werden. Für eine statische Strategie, bei der eine bestimmte Anzahl an Interationen statisch auf die Threads verteilt werden, gibt es den Parameter `schedule(static, [Blockgröße])`. Die Blockgröße kann optional angegeben werden. Wird sie nicht angegeben, werden alle Iterationen durch die Anzahl der Threads geteilt und diese Anzahl verwendet. Wird die Blockgröße angegeben, darf diese nicht so klein sein, dass nicht alle Iterationen der Schleife von den Threads abgearbeitet werden. Der letzte Thread darf, wenn die Iterationsanzahl nicht gleichmäßig aufgeteilt werden kann, weniger Iterationen ausführen. Wird die Blockgröße sehr groß gewählt, sind unter Umständen nicht alle Threads mit der Abarbeitung der Schleife beschäftigt. Bei einer dynamischen Verteilung muss der Parameter `schedule(dynamic, [Blockgröße])` angegeben werden. Hierbei wird die Blockgröße kleiner gewählt und die Threads arbeiten mehrere Blöcke mit Iterationen nacheinander ab. Sollten einige Iterationen länger dauern als andere, bekommen die Threads somit weniger Blöcke zur Abarbeitung. Die verhindert, dass gegen Ende der Abarbeitung ein einziger Thread noch rechnen muss, währenddessen alle anderen Threads schon lange fertig sind. Die letzte Schedule-Strategie in OpenMP kann mit der Direktiven `schedule(guided, [Blockgröße])` verwendet werden. Dies ist eine spezielle Form der dynamischen Strategie. Hierbei werden die Blö-

cke zur Abarbeitung exponentiell immer kleiner. Iterationsblöcke, die schneller fertig werden, bekommen deshalb einen größeren zweiten Block als Iterationen, die später ihren ersten Block abgearbeitet haben. Die Unterschiede zwischen den Bearbeitungszeiten der Iterationsblöcke soll dadurch verringert werden.

Dynamische Ablaufpläne haben den Vorteil, dass die Threads auch bei unterschiedlicher Abarbeitungsdauer der Iterationen ungefähr gleichzeitig fertig werden mit ihren Berechnungen. Dennoch hat die statische Strategie ihre Daseinsberechtigung. Durch die dynamische Verwaltung entsteht ein großer Verwaltungsaufwand, da die Threads synchronisiert werden müssen und neue Iterationen den Threads zugewiesen werden müssen. Bei der statischen Strategie hingegen wird dieser Aufwand nicht benötigt. Besonders, wenn sich die Abarbeitungszeiten der Iterationen nicht verändert, kann es deswegen sinnvoll sein, auf die dynamische Strategie zu verzichten und stattdessen die statische Strategie zu verwenden.

An Ende der Schleife wird normalerweise darauf gewartet, bis alle Threads beendet werden können. Soll dies nicht geschehen, kann der Parameter `nowait` angegeben werden. Dann führen die Threads den folgendes Code aus, anstelle darauf zu warten bis alle Threads mit der Abarbeitung der parallelisierten Schleife fertig sind.

Ein weiterer Parameter zur Parallelisierung einer For-Schleife ist `ordered`. Dabei werden die Iterationen in der Reihenfolge ausgeführt, wie Sie bei paralleler Ausführung ausgeführt würden. Bei diesem Parameter kann noch ein Block mit dem Direktiv `#pragma omp ordered` unterhalb der For-Schleife stehen. Dieser wird dann auch in der Reihenfolge ausgeführt, als wäre es serieller Code.

Weitere Funktionen zur Parallelisierung mit Hilfe von OpenMP wurden für diese Arbeit nicht verwendet und werden deswegen im Folgenden nur kurz erwähnt. Für verschiedenen Bereiche, die unabhängig voneinander sind und parallel ausgeführt werden sollen, kann `#pragma imp parallel section [Parameter]` verwendet werden. Für kritische Abschnitte gibt es `#pragma omp critical [(name)]` und eine Barriere kann mit `#pragma omp barrier` erzeugt werden.

7.2 Verwendung von SIMD auf x86 Architektur

Eine weitere Optimierung ergibt sich durch die Verwendung von SIMD (Single Instruction, Multiple Data) Operationen. Neue CPUs von Intel und AMD besitzen spezielle Register, auf die SIMD Operationen angewandt werden können. Die ersten SIMD Operationen auf der x86-Architektur von Intel wurden 1997 durch die Veröffentlichung von Prozessoren mit der Multi Media Extension (MMX) eingefügt. Die MMX beinhaltete 8 Register mit einer Größe von 64 Bit, die packed Integer Werte annehmen kann. Die Register hießen `mm0` bis `mm7`. Von packed ist die Rede, wenn ein Register größer als der Datentyp den es speichert. Anstelle den Wert des Datentyps in den niedrigsten Bits zu speichern, werden mehrere Werte des Datentyps hintereinander in das Register geschrieben, um so die ganze Breite des Registers auszunutzen.

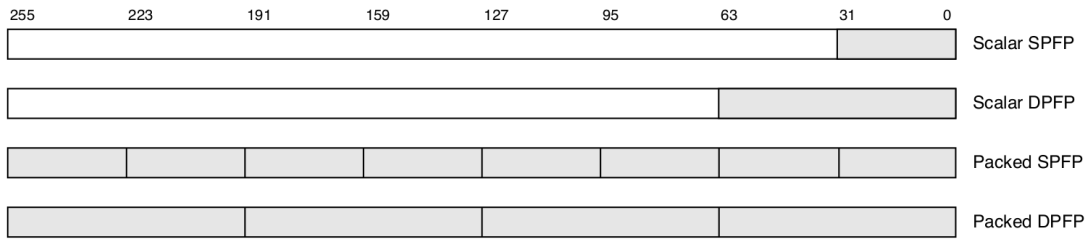


Abbildung 7.1: Anordnung von Floats und Double in einem AVX SIMD Register

Die Größe der Register wurde 1999 durch die Einführung von SSE, der Streaming SIMD Extension, verdoppelt. Der Namen der Register änderte sich zu `xmm0` bis `xmm7`. Die SSE erlaubte anfangs nur das packen von single precision floating points, was dem Datentyp Float entspricht. Die nächsten Jahre wurden mehrere Verbesserungen der SSE in neuere Generationen der CPUs verbaut, so unterstützten CPUs mit der SSE3 Hyper-Threading von SIMD Instruktionen. 2011 wurde AVX, Advanced Vector Extension, in der neuen Mikroarchitektur Sandy Bridge eingebaut. Diese fügte zu den Registern xmm0 bis mm7 die Register ymm0 bis ymm7 hinzu. Die Register können zu 8 großen Registern zusammengefasst werden, sodass die Registergröße für eine SIMD Operation 256 Bit beträgt. Auf CPUs mit 64 Bit Adressierung wächst die Anzahl der Register auf 16. 2013 wurde ein erweiterter Instruktionssatz, genannt AVX2, in die damals aktuelle Haswell Architektur eingebaut. Die neuste Entwicklung im SIMD Umfeld nennt sich AVX-512. Dabei wird die Anzahl der Register nochmal verdoppelt auf 32 Register und es werden die neuen Register zmm0 bis zmm31 hinzugefügt. Diese haben eine Breite von 256 Bit. Werden die neuen Register mit den bereits aus der AVX vorhandenen Register zusammengefasst, erhöht sich die Datenbreite auf 512 Bit. Diese Version von AVX ist jedoch bis jetzt nur in sehr teuren CPUs, die für das Server Umfeld gedacht sind, enthalten (vgl. [17]). Ein Vergleich aller Register ist in der Grafik unterhalb zu erkennen.

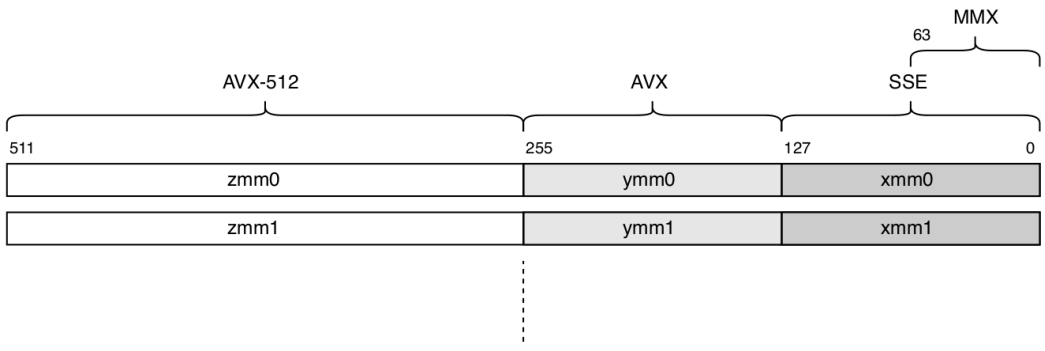


Abbildung 7.2: Übersicht der Größe von SIMD Registern der x86 Architektur

Eine für diese Arbeit interessante Funktion der SIMD Instruktionen von neuen x86 Architekturen ist die Multiplikation von Float Zahlen. Durch AVX-512 lassen sich bei einer neuen CPUs bis zu 16 Floats in einem Register unterbringen und mit einer Instruktion multiplizieren. Für die Multiplikation werden 2 Register benötigt, die mit den zu multiplizierenden Floats initialisiert sind. Desweiteren wird ein Register benötigt, in dem das Ergebnis gespeichert werden soll. Durch Aufrufen des Befehls zur Multiplikation werden das n-te Float in Register a mit dem n-ten Float in Register b multipliziert und in ein Zielregister geschrieben. Da die neuste AVX-Version AVX-512 nur in wenigen CPUs integriert ist, wird in dieser Arbeit lediglich die ältere Version AVX2 verwendet.

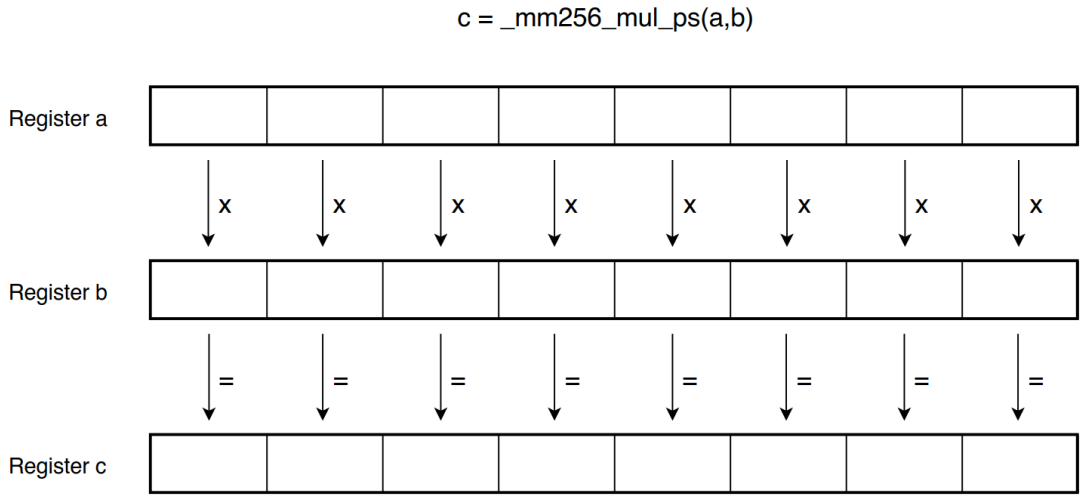


Abbildung 7.3: Multiplikation von 8 Floats durch eine SIMD Anweisung

In der Grafik werden 8 Multiplikationen von Floats mit der Instruktion `_mm256_mul_ps()` gleichzeitig ausgeführt. Die `256` gibt hierbei an, wie groß die Register sind. In diesem Fall sind die Register 256 Bit groß. Anschließend wird die Rechenart angegeben, hier eine Multiplikation (`mul`). Das `p` steht hierbei für "packed". Alternativ kann ein `s` für ßingle slotängegeben werden, dann wird le-

diglich an der hintersten Position ein Wert des Datentyps erwartet und für die Multiplikation verwendet. An letzter Stelle steht ein `s`. Dieses steht für ßingle precision", was dem Datentyp Float entspricht. Bei einem `d` an dieser Stelle wird "pouble precission", was dem Datentyp Double entspricht, angenommen.

Eine Übersicht aller SIMD-Instruktionen von Prozessoren mit aktueller x86-Architektur und findet sich unter `https://software.intel.com/sites/landingpage/IntrinsicsGuide`. Hier kann auch gesehen werden, mit welcher Erweiterung der SIMD Instruktionen ein Befehl einführt wurden ist.

Durch die Verwendung von AVX lässt sich die Zeit zum Multiplizieren von mehreren Floats, im Vergleich zur standardmäßigen Multiplikation, prinzipiell bis zu 16 mal verkürzen. Der gcc-Compiler versucht auf Optimierungsstufe 3 (Compilerflag `-O3`) for-Schleifen automatisch zu vektorisieren. Deswegen ist es nicht unbedingt vonnöten, die AVX-Register mühsam selbst zu verwenden. Durch das Compiler-Flag `-ftree-vectorizer-verbose=2` wird bei jeder Schleife ausgegeben, ob gcc den Code vektorisieren konnte und somit den Code mit SIMD-Instruktionen übersetzt hat oder nicht.

Die Instruktionen können jedoch auch explizit aufgerufen werden, sodass nicht der Compiler die Entscheidung übernimmt, wo und wie optimiert werden soll, sondern dies der Programmierer selbst Entscheiden kann.

Für das Laden von Floats in ein AVX-Register gibt es mehrere Möglichkeiten. Um Werte an expliziter Stelle des Registers zu schreiben, gibt es den Befehl `_mm256_set_ps(a8,a7,a6,a5,a4,a3,a2,a1)`. Der erste Parameter der Funktion wird hierbei an der letzten Stelle innerhalb des Registers geschrieben. Um einen Wert an alle Stellen des Registers zu schreiben, kann `_mm256_set1_ps(a1)` verwendet

werden. Mit `_mm256_load_ps(addr)` kann eine Adresse übergeben werden und die folgenden Werte an dieser Adresse werden in das Register geschrieben. Um das Ergebnis wieder als Floats anzusprechen, muss der Pointer zum Ergebnisregister als Pointer zu einem Float-Array gecastet werden.

```
float array[8]=[1,2,3,4,5,6,7,8];
_m256 a = _mm256_set_ps(8,7,6,5,4,3,2,1);
_m256 b = _mm256_load_ps(array);
_m256 res = _mm256_mul_ps(a,b);
float *f = (float*)&res;
```

Listing 7.4: Multiplikation mit AVX

An Adresse `f` stehen nun die ersten 8 Quadratzahlen, die alle mit der Instruktion `_mm256_mul_ps` auf einmal errechnet wurden.

7.3 Implementierung des Convolutional Neural Network

Zur Implementierung des Convolutional Neural Networks auf einer x86-Architektur wurde die serielle Implementierung als Grundlage verwendet. Diese wurde genauer in Kapitel Serielle Implementierung beschrieben. Zur optimierten Ausführung auf einer x86-CPU muss das Programm umgeschrieben werden. Folgende Methoden wurden zur Optimierung verwendet:

- Reduzierung der Schreibzugriffe
- Reduzierung der zufälligen Lesezugriffe
- Cache-Hierarchie ausnutzen
- SIMD-Architektur ausnutzen

7.3.1 Die Klasse Tensor

Für jegliche Inputs und Outputs aller Layer wurde die Klasse `Tensor` erstellt. Diese beinhaltet ein Array aus Floats sowie drei Integer mit den Namen `x`, `y` und `z`.

Im Konstruktor werden die Werte von `x`, `y` und `z` übergeben. Es gibt auch Konstruktoren mit nur 2 oder nur 1 Parameter, die anderen Dimensionen werden dann aus 1 gesetzt. Die Tensor-Klasse kann somit auch Matritzen oder Vektoren abbilden.

Das Array hat als Länge das Produkt aus allen 3 Dimensionen. Die Reihenfolge ist hierbei so angeordnet, wie die Leserichtung eines Buches. Es wird immer eine Zeile durchgegangen, bevor die nächste Zeile im Speicher liegt. Ist die letzte Zeile einer Seite erreicht, liegt die erste Zeile der nächsten Seite im Speicher bis der Tensor vollständig im Speicher abgebildet ist.

Die Tensor-Klasse bietet Funktionen, um die Adresse des Arrays an bestimmten Stellen zu finden. Dabei wird zuerst die z-Position und Anschließend die y-Position angegeben.

```
float *Tensor::getArray(int z, int y){
        assert(z<this->z);
        assert(y<this->y);
        return array+z*this->y*x+y*x;
}
```

Listing 7.5: Funktion zum Erhalten eines Pointers auf den Tensor

Die x-Position kann bei dieser Funktion nicht angegeben werden. Um die genaue Adresse zu erhalten, muss ein Offset in Höhe der x-Position angegeben werden. Danach zeigt der Zeiger auf die gewünschte Stelle. Durch die Verwendung eines Zeigers, kann die Funktion sowohl zum Lesen als auch zum Schreiben an dieser Position verwendet werden.

```
zeiger = tensor->getArray(z_pos, y_pos)[x_pos];
```

Listing 7.6: Verwendung der getArray()-Funktion

Die Klasse besitzt noch Funktionen, um die Größe des Tensors in die drei verschiedenen Dimensionen auszugeben. Dadurch kann mit drei verschachtelten For-Schleifen einfach durch alle Elemente iteriert werden.

7.3.2 die verschiedenen Layertypen

Es gibt drei verschiedene Layertypen. Diese sind der Convolutional Layer, der Max Pooling Layer und der Fully Connected Layer. Um diese drei verschiedenen Layer in einer gemeinsamen Liste zu speichern, wurde ein Union mit dem Namen `Layer` erstellt. Dieses Union kann einen dieser 3 Datentypen annehmen. Jede dieser Layer besitzt mehrere Zeiger auf Tensoren. Jeder Layer hat Zeiger auf Tensoren die die Activation, den Output und den Gradienten des Layers und des Layers davor darstellen. Mit Ausnahme des Max Pooling Layers gibt es noch Zeiger auf Gewichte und Biases, sowie Tensoren mit den Deltas der Gewichte und Biases. Der Zeiger auf den Output eines Layers hat den selben Wert wie der Zeiger des Inputs des folgenden Layers. Somit weden die Tensoren von mehreren Layern verwendet, einmal als Input und einmal als Output. Das Gleiche gilt für die Gradienten. Die Tensoren sind so aufgebaut, dass die x und y Dimension ein Muster darstellt und mit der z-Position die verschiedenen Muster ausgewählt werden. Alle Layer beinhalten die Methoden `generate()`, `forward()`, `backward()` und `fix_weights()`. Die `generate()` Methode benötigt als Parameter Zeiger zu den Tensoren der Activation und der Gradienten der Activation des Layers. Diese sind, ausgenommen des ersten Layers in einem Netzwerk, die Outputs der vorherigen Layer. Die `generate()` Methode berechnet aufgrund der Parameter der Methode

und der Parameter des Konstruktors des Layers die Größen alles Tensoren die vom Layer verwendet werden. Die Funktionen `forward()` und `backward()` nehmen keine Parameter an. Erstere wird aufgerufen, um die Inputs der Neuronen durch das Netzwerk zu propagieren. Letztere wird aufgerufen, um die Fehler zurück zu propagieren. Bevor die `backward()` Methode aufgerufen werden kann, muss der Fehler im letzten Layer berechnet werden und in den Tensor der Gradienten gespeichert werden. Die Formel für die Berechnung des Fehlers, bei Verwendung von Quadratic Coast als Kostenfunktion, ist in 7.1 zu sehen.

$$\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j) \tag{7.1}$$

a_j^L ist hierbei ein Output im Letzten Layer und y_j das dazu gehörende gewünschte Ergebnis. Auf die partielle Ableitung der Kostenfunktionen nach dem Output wird anschließend noch die inversen Sigmoid-Funktion angewandt.

Durch die Funktion `fix_weights()` werden die Deltas benutzt, um die Gewichte und die Biases anzupassen. Dabei wird die Batch-Größe angegeben sowie die Trainingsrate. Nach Aufruf dieser Funktion werden alle Deltas auf 0 gesetzt. Dadurch kann eine Batch von Inputs verwendet werden, da die Funktion `backward()` die Deltas von Biases und Gewichten auf den bestehenden Wert addiert.

Ein Max Pooling Layer erwartet als Parameter im Konstruktor die Schrittweite, in dem das Max Pooling stattfinden soll. Die Größe des Outputs ist der Quotient der Dimensionen des Inputs dividiert durch die Schrittweite in dieser Dimension. Die z-Dimensionen des Inputs und des Outputs sind die selben. Dasselbe gilt für die Gradienten. In der `forward()` Methode wird immer das Maximum im gewünschten Bereich gesucht und anschließend in den Output geschrieben. Die `backward` Methode leitet den Gradienten des Layers danach an die Position, die den maximalen Bereich in diesem Bereicht hatte, weiter. Die Klasse hat den Namen `MaxPoolingLayer`

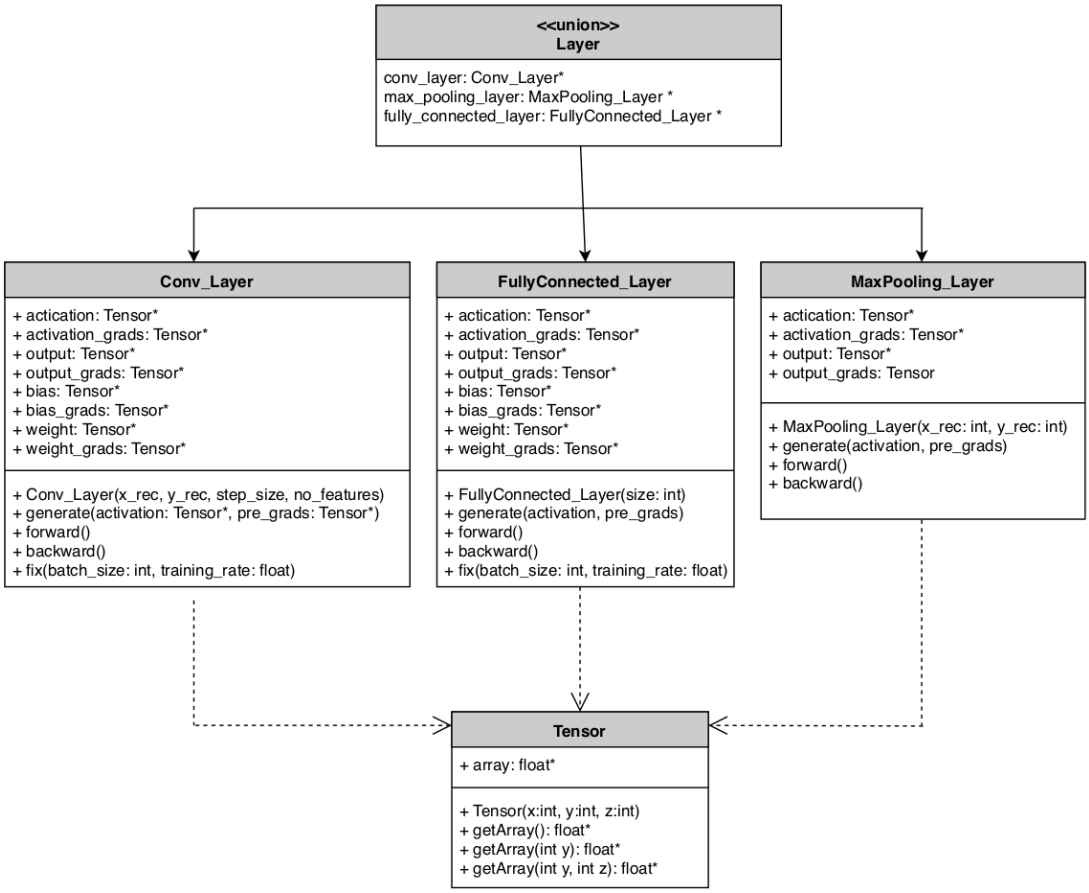


Abbildung 7.4: UML-Diagramm der Layer

Die Klasse `FullyConnectedLayer` erwartet im Konstruktor die Anzahl der Neuronen in diesem Layer. Der Output ist ein Vektor mit dieser Anzahl an Elementen. Der Vektor wird als Tensor mit allen Dimensionen, außer der x-Dimension, auf 1 abgebildet. Als Gewichts-Tensor wird die Dimension der Activation verwendet, die x-Dimension wird jedoch mit der Neuronenanzahl des Layers multipliziert. Die Gewichte für einen Output sind damit sehr einfach zu finden und es kann sich ein beliebiger Layer vor dem Fully Connected Layer befinden, ohne den Code für die verschiedenen Layertypen zu ändern. Der Bias-Tensor hat die gleiche Dimension wie der Output-Tensor.

`ConvLayer` ist der Klassennamen für den Convolutional Layer. Im Constructor wird die Größe der Faltungskerne, die Anzahl der zu ermittelnten Features und die Schrittweise zum Abfahren der Activation angegeben. Die Größe des Outputs lässt sich mit dieser Formel berechnen.

$$size_{output} = \frac{size_{input} - size_{faltungskern} + 1}{schrittweite} \tag{7.2}$$

Die z-Dimension des Outputs ist durch die Anzahl der Features gegeben. Der Tensor der Gewichte, im Convolutional Layer der Faltungskerne, entspricht in x- und y-Richtung der übergebenen Größe der Faltungskerne. Die Größe in z-Richtung entspricht dem Produkt aus der z-Dimension der Activation und der Summe an Features. Für jedes Feature ist so ein Teilstück des Gewichtstensors vorhanden. Für jedes Feature gibt es einen Eintrag in einem Tensor mit einer Dimension als Bias.

Die Klasse `InputLayer`, die es noch in der seriellen Implementierung gab, gibt es nicht mehr. Stattdessen zeigt der Pointer des ersten Layers auf einen Tensor, der direkt das Bild enthält, welches im Netzwerk analysiert werden soll.

7.3.3 Funktion der verschiedenen Tensoren

Die Grafik 7.5 zeigt das Zusammenspiel und die Funktionsweise der verschiedenen Tensors bei Verwendung der forward()- und backward()-Funktionen. Die Abkürzung "wßteht hierbei für "weight", "b"für "bias", "w_g"für "weight_gradsünd "b_g"für "bias_grads". In einem Layer sind immer Zeiger zu den Gewichten und den Biases, die sich in der Grafik vor dem Layer befinden, enthalten. In forward-Richtung werden die Gewichts- und die Bias-Tensors für die Berechnung der Activation des nächsten Layers verwendet. Im letzten Layer wird der Fehler berechnet und in den Tensor öutput_grad"geschrieben. In backward-Richtung werden nun die Fehler zurückgerechnet und dabei in jedem Layer die Tensoren "weight_gradsünd "bias_grads"geschrieben. Die Werte für die Tensoren werden auf den sich schon darin befindenden Wert addiert. Durch Aufruf der fix()-Funktion werden die Gewichte und Biases der Layer durch die Werte Tensoren "weight_gradsünd "bias_gradsängepasst. Dies ist in der Grafik als orangefarbenen Pfeil zu erkennen.

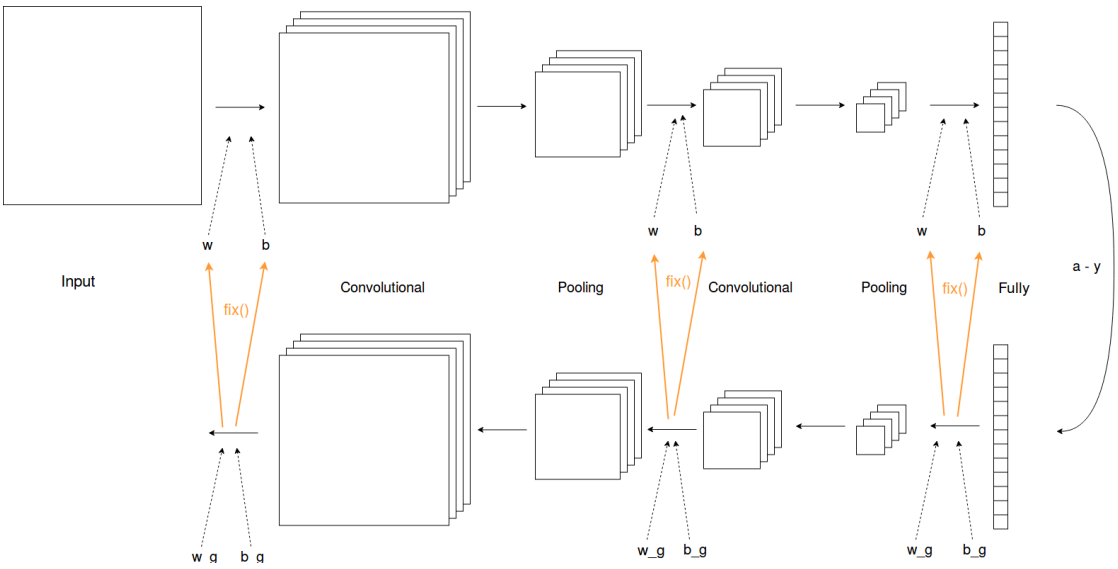


Abbildung 7.5: Zusammenspiel zwischen forward()- und backward()-Methode

7.3.4 Implementierung des Convolutional Layers

Die größten Optimierungsmöglichkeiten ergeben sich im Convolutional Layer, da dort die Anzahl der Multiplikationen am Höchsten ist. Um den Layer zu optimieren, muss zuerst die Funktionsweise aus der seriellen Implementierung analysiert werden.

Bei der seriellen Implementierung werden die benötigten Elemente der Activation für eine Faltungsoperation in einen temporären Vektor geschrieben. Dieser Vektor wird anschließend mit einer Gewichts-Matrix multipliziert. Dies stellt die Faltung mit allen Kernen für diese Elemente der Activation dar. Das Ergebnis der Multiplikation wird wiederum in einen anderen temporären Vektor gespeichert, zu welchem ein Bias Vektor elementweise addiert wird. Das Ergebnis sind die Elemente des Outputs an einer Stelle in allen Features. Die Vorgehensweise ist in der Grafik unterhalb zu sehen.

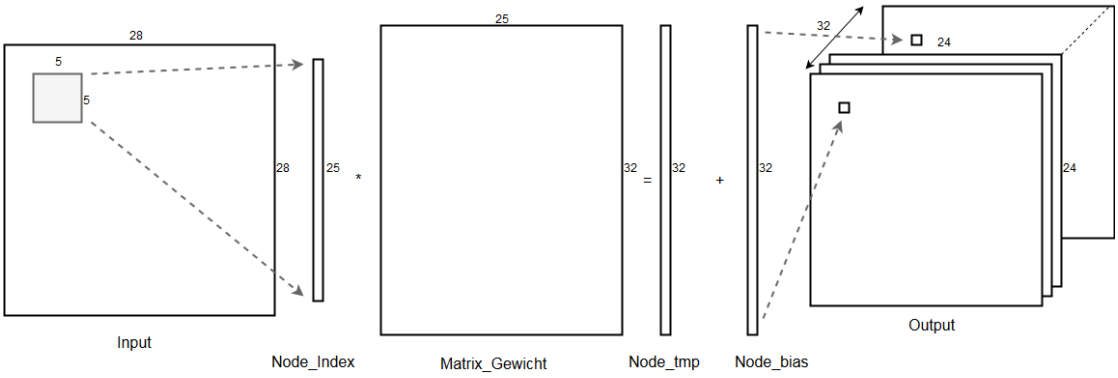


Abbildung 7.6: Ablauf des Convolutional Layers in der seriellen Implementierung

Bei dieser Vorgehensweise sind sehr viele neue Allokationen des Hauptspeichers nötig, da viel Speicherplatz kurzfristig benötigt wird. Dies lässt sich vermeiden, indem die Multiplikationen geschickt nacheinander ausführt und das Ergebnis direkt an die Zieladresse geschrieben werden.

Durchgehen der Activation

Es wurde sich für eine Methode entschieden, bei der zuerst der Output-Tensor auf 0 gesetzt wird. Anschließend wird jedes Element vom Activation-Tensor durchgegangen. Je nachdem, wo sich das Element im Activation-Tensor befindet, gibt es verschiedene Elemente der Gewichts-Matritzen, mit denen es multipliziert werden muss. Beispielweise muss das erste Element links oben nur mit dem Element links oben des Faltungskerns multipliziert werden, währenddessen Elemente in der Mitte des Input-Tensors mit allen Elementen des Faltungskerns multipliziert werden müssen. Die Ergebnisse müssen je nach Element des Faltungskerns an verschiedenen Stellen im Output geschrieben werden. Die entsprechenden Elemente des Faltungkerns werden mit zwei if-Bedingungen ermittelt. Bei diesen if-Bedingungen wird der Start- und der Endindex für beide Dimensionen eines Faltungskerns durch die Position des Elements in der Activation ermittelt. Anschließend werden diese Werte des Faltungskerns mit dem einen Element des Inputs multipliziert und die Ergebnisse an die richtige Stelle des Output-Tensors hinzugefügt.

In unten stehender Formel ist der Vorgang zu erkennen. x_pos und y_pos sind die Indizes des Faltungskerns mit dem das Element $a_{x,y,z}$ multipliziert werden muss. Alle Kombinationen von x_pos, y_pos und z_pos werden durchgegangen. Anschließend wird das nächste Element von a verwendet.

$$\begin{aligned} x\_pos &= min\_x_{a_{x,y,z}} \,..\, max\_x_{a_{x,y,z}} \\ y\_pos &= min\_y_{a_{x,y,z}} \,..\, max\_y_{a_{x,y,z}} \\ z\_pos &= z \,..\, length(w_z)\ [SW:\ length(a_z)] \\ o_{x-x\_pos,y-y\_pos,z\_pos/z} &+= a_{x,y,z} * w_{x\_pos,y\_pos,z\_pos} \end{aligned} \tag{7.3}$$

Den Programmcode ist in 7.7 zu sehen.

```
#pragma omp for firstprivate(output, activation, weight)
    lastprivate(output, activation, weight)
```

```
for(int pre_z_pos=0; pre_z_pos < activation->getZ();
    pre_z_pos++){
    for(int pre_y_pos = 0; pre_y_pos < activation->getY();
        pre_y_pos++){
        for(int pre_x_pos = 0; pre_x_pos < activation->getX()
            ;pre_x_pos++){

            int start_x_rec=0;
            int stop_x_rec=x_receptive-1;
            if(pre_x_pos < x_receptive-1) stop_x_rec =
               pre_x_pos;
            else if(pre_x_pos > activation->getX()-x_receptive
               ) start_x_rec = x_receptive + pre_x_pos -
               activation->getX();

            int start_y_rec=0;
            int stop_y_rec=y_receptive-1;
            if(pre_y_pos < y_receptive-1) stop_y_rec =
               pre_y_pos;
            else if(pre_y_pos > activation->getY()-y_receptive
               ) start_y_rec = y_receptive + pre_y_pos - activ
               [language=c++]ation->getY();

            for(int z_pos=pre_z_pos;z_pos < weight->getZ();
               z_pos+=activation->getZ()){
               for(int y_rec = start_y_rec; y_rec <=
                  stop_y_rec ; y_rec++){
```

```
                    for(int x_rec = start_x_rec; x_rec <=
                        stop_x_rec ; x_rec++){
                        output->getArray(z_pos/activation->getZ()
                            , pre_y_pos-y_rec)[pre_x_pos-x_rec] +=
                             activation->getArray(pre_z_pos,
                            pre_y_pos)[pre_x_pos] * weight->
                            getArray(z_pos,y_rec)[x_rec];
                    }
                }
            }
        }
    }
}
```

Listing 7.7: Convolution: Activation durchgehen

Durch diese Methode kann der Output-Tensor und die Gewichte im Cache liegen, die Speicherzugriffe erfolgen somit sehr schnell. Da kein neuer Speicher allokiert wird, muss nicht darauf gewartet werden, bis der vergleichsweise sehr langsame Arbeitsspeicher den Speicherplatz freigibt und die Daten in den Cache geladen werden können.

Durchgehen des Outputs

Der Convolutional Layer kann auf mehrere Arten implementiert werden. Immer müssen mehrere For-Schleifen ineinander die Faltung mit dem Faltungskern ausführen, dazu werden die einzelnen Zählvariablen als Index des Input und des Outputs verwendet. Es kann jedoch entschieden werden, welche Schleife für welchen Indes verwendet wird.

Die wohl einfachste Art der Implementierung ist die, dass die äußeren For-Schleifen den Index des Outputs bestimmen und die inneren den Index des Inputs und des Faltungskerns. Bei dieser Art kann, wenn die Schrittweite 1 beträgt, der Index des Outputs als Anfangsindex des Inputs verwendet werden. Dies kann anhand der Formel **??** abgeleitet werden. Die inneren Schleifen erzeugen einen Offset zwischen 0 und der Breite/Länge des Faltungskerns. Dieser Offset wird auf die Variable des Outputs addiert und erzeugt somit die Positionen des Inputs, die für den Output verwendet werden soll.

Eine Implementierung des Convolutional Layers nach dieser Methode ist unterhalb zu erkennen.

```
#pragma omp parallel for firstprivate(output, activation,
    weight) lastprivate(output, activation, weight)
for ( int z_pos =0; z_pos < output->getZ() ; z_pos++){
    for ( int y_pos = 0 ; y_pos < output->getY() ; y_pos++){
        for ( int x_pos = 0 ; x_pos < output->getX() ; x_pos
            ++){

            int z_stop = (z_pos+1)*activation->getZ();
            for ( int w_z_pos=z_pos*activation->getZ() ;
                w_z_pos < z_stop; w_z_pos++){
                for ( int w_y_pos = 0; w_y_pos < weight->getY
                    (); w_y_pos++){
                    for ( int w_x_pos = 0 ; w_x_pos < weight->
                        getX(); w_x_pos++){
```

```
                    output->getArray(z_pos, y_pos)[x_pos] +=
                        activation->getArray(z_pos%activation
                        ->getZ(), y_pos+w_y_pos)[x_pos+w_x_pos
                        ] * weight->getArray(w_z_pos, w_y_pos
                        )[w_x_pos] ;
                }
            }
        }


    }
  }
}
```

Listing 7.8: Convolution: Output durchgehen

Der Vorteil dieser Methode, im Vergleich zur vorherigen Methode, ist die einfache Umsetzung von SIMD Instruktionen in diesem Code. Werden AVX-Register verwendet, können immer 8 der Gewichte in ein mmx-Register geladen werden. In ein zweites Register müssen die Werte der Activation geladen werden. Anschließend können die erhaltenen Ergebnisse zusammenaddiert werden. Dies wird solange wiederholt, bis alle Multiplikationen für einen Output des Layers abgeschlossen sind. Ein weiterer Vorteil ist die Tatsache, dass die Sigmoid-Funktion direkt innerhalb der Schleife angewandt werden kann. Bei der Implementierung, in der die Activation in der äußeren Schleife durchgegangen wird, muss die Sigmoid-Funktion in einer separaten Schleife auf alle Outputs angewandt werden.

Für die explizite Verwendung der AVX-Register wurde dieser Code entwickelt, der innerhalb der äußeren drei Schleifen eingesetzt werden muss. Es werden immer acht Multiplikationen für einen Output gleichzeitig ausgeführt. Ist Anzahl der Multiplikationen nicht durch 8 teilbar, werden mit der letzten SIMD-Anweisung weniger Multiplikationen ausgeführt. Alle Ergebnisse für einen Faltungskern werden zusammenaddiert und in den Output geschrieben.

```
int z_stop = (z_pos+1)*activation->getZ();

float array [8];

for(int w_z_pos = z_pos*activation->getZ(); w_z_pos <
    z_stop; w_z_pos++){

mmx1 =  _mm256_loadu_ps(weight->getArray(w_z_pos));
int mmx_index = 0;

for(int w_y_pos = 0; w_y_pos < weight->getY(); w_y_pos++){
    for(int w_x_pos = 0; w_x_pos < weight->getX(); w_x_pos
        ++){

        array[mmx_index%8]=activation->getArray(w_z_pos%
            activation->getZ(), y_pos+w_y_pos)[x_pos+w_x_pos];
        mmx_index++;
        if(mmx_index%8 == 7 || mmx_index == weight->getSize()
            -1){
            mmx2 = _mm256_loadu_ps(array);
            mmxres = _mm256_mul_ps(mmx1, mmx2);
```

```
            float *f = (float*)&mmxres;
            for(int i=0; i < 8 && mmx_index+i < weight->
                getSize(); i++){
                output->getArray(z_pos,y_pos)[x_pos] += f[i];
            }
            mmx1 =  _mm256_loadu_ps(weight->getArray(w_z_pos)+
                mmx_index);
        }

    }
}
```

Listing 7.9: Implementierung des Convolutional Layers mit AVX

Vergleich der verschiedenen Implementierungen

Die verschiedenen Implementierungen des Convolutional Layers wurden in einem Testprogramm auf ihre Laufzeiten analysiert. Dabei wurde eine Activation mit der Größe von 24*24*32 gewählt und ein Output von 20*20*64. Die Faltungskerne haben jeweils 25 Elemente, es wird immer eine 5*5-Umgebung nach Mustern abgesucht. Als Testhardware wurde ein PC mit einer Intel Core i5-5310 CPU verwendet. Diese besitzt lediglich 2 CPU Kerne. In einem Durchlauf des Programms werden 10 Faltungsoperationen mit jeder der drei Implementierungen ausgeführt. Die Implementierungen sind einmal das Durchgehen des Outputs mit AVX, einmal das Durchgehen des Outputs ohne AVX und die Implementierung, bei der der Input durchgegangen wird. Die Ausführungszeiten werden in Sekunden gemessen. Als Optimierungsstufe wurde die Optimierungsstufe 0 und die Optimierungsstufe 3 im Compiler gcc verwendet. Dies einmal mit, und einmal ohne Verwendung von

OpenMP. Gemessen wurde mit der Funktion `clock()`. Diese Funktion gibt eine Zahl zurück, die sich bei jedem Tick auf dem Prozessor erhöht. Durch die Differenz der Zahlen vor- und nach der Ausführung eines Codes, lässt sich die Anzahl der Ticks zur Ausführung des Codes ermitteln. Die gemessene Differenz wurde durch eine Million geteilt um kleinere Zahlen zu erhalten.

Optimierungsstufe	Output AVX	Output	Input
O0	10,27	7,79	8,51
	13,26	11,44	11,45
	7,63	5,6	6,13
O3	4,44	3,64	3,49
	4,22	3,32	3,15
	4,87	4,3	3,82
O0, OpemMP	9,95	7,38	7,85
	10,29	8,29	8,15
	9,2	7,04	7,6
O3, OpenMP	4,77	4,1	4,22
	4,31	3,52	3,73
	5,73	4,98	5,332

Da die Laufzeiten teilweise sehr unterschiedlich ausfallen, wurden drei Durchläufe des Programms pro Optimierungsstufe getätigt. Es ist zu erkennen, dass die Verwendung von OpenMP aus Optimierungsstufe 0 nur zu einer sehr kleinen Geschwindigkeitsverbesserung führt. Auf Optimierungsstufe 3 verschlechtert sich das Ergebnis durch parallele Ausführung sogar. Dies liegt klar an der CPU der Testhardware, die lediglich 2 Kerne besitzt. Das Erstellen von neuen Threads ist zu aufwändig im Vergleich zum erhaltenen Performance-Gewinn. Um einen Leistungsgewinn durch Multithreading zu erzielen, muss die CPU mehrere Kerne besitzen oder das Programm besser parallelisiert werden. Eine weitere, interessante

Erkenntnis aus diesem Vergleich ist, dass die Implementierung mit expliziten AVX-Aufrufen langsamer ist als die Implementierung ohne AVX. Der Compiler schafft es, den Code besser zu optimieren als mein Versuch, die Intrinsiks zu verwenden. Die Implementierung, bei der der Input durchgegangen wird auf Optimierungsstufe 3, ist die schnellste Implementierung. Dies liegt höchstwahrscheinlich an der Ausnutzung der Cache-Hierarchie. Der Input kann bei dieser Implementierung immer im Speicher gelassen werden und wird eins nach dem anderen Durchgegangen.

Den Code tatsächlich zu optimieren, wurde im Rahmen der Studienarbeit nicht geschafft. Für das Suchen eines Fehlers beim Zurückrechnen der Biases und Gewichte wurde zu viel Zeit investiert. Die Zeit für das tatsächliche Optimieren mit SIMD und Multithreading viel deswegen zu gering aus, um den Zeitrahmen der Studienarbeit nicht komplett zu sprengen. Trotzdem wurden Messungen mit dem Programm ausgeführt, die im letzten Kapitel dieser Arbeit, Kapitel 10, beschrieben sind.

8 Implementierung auf einem Intel Xeon Phi Coprozessor

Im Gegensatz zu General Purpose CPUs lassen sich mithilfe von Coprozessoren deutlich höhere Parallelisierungsgrade erzielen. Gegenüber GPUs haben sie den Vorteil, dass die Kerne eines Coprozessors einen größeren Befehlssatz verfügen, mit dem sie bedingte Sprünge effizienter ausführen können. Mit GPUs lässt sich dagegen eine weitaus höhere Parallelität erreichen. Mit der Max Pooling-Operation ist die Fähigkeit von Coprozessoren, auch Code mit Verzweigungen parallel auszuführen, von Nutzen.

In dieser Arbeit soll auch eine Implementierung des CNNs erstellt werden, die Berechnungen an Coprozessoren auslagern kann. Ein direkter Vergleich dieser Implementierung mit Tensorflow hat nur bedingte Aussagekraft, da Tensorflow laut der offiziellen Dokumentation nur für den Einsatz auf CPUs, GPUs und TPUs optimiert ist. Coprozessoren werden von Tensorflow dagegen nicht explizit unterstützt. (vgl. [18]) Ein Vergleich zwischen dieser Implementierung und den anderen beiden in dieser Arbeit behandelten optimierten Implementierungen lässt allerdings Rückschlüsse darauf ziehen, wie der Performancegewinn unter Zuhilfenahme eines Coprozessors einzuordnen ist.

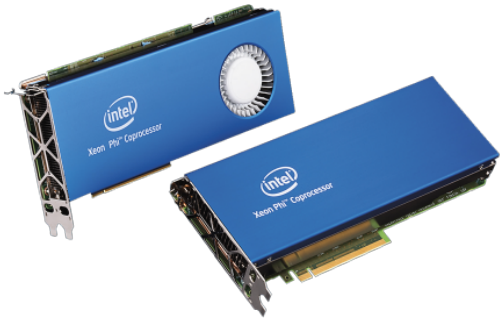


Abbildung 8.1: Eine Erweiterungskarte mit einem Intel Xeon Phi (Quelle: [20])

Die DHBW Stuttgart besitzt einen Server, der mit zwei Server-CPUs (Intel Xeon E5-2650L) und einem Coprozessor (Intel Xeon Phi) ausgestattet ist. Beim Xeon Phi handelt es sich um eine PCIe-Erweiterungskarte, auf der eine spezielle CPU mit 60 Kernen und einer Taktrate von 1 GHz, sowie ein DDR5-Arbeitsspeicher mit einer Größe von 8 GB verbaut sind. Für diese Arbeit wurde den Studenten freundlicherweise ein SSH-Zugang zur Verfügung gestellt. Die in diesem Kapitel betrachtete Implementierung soll auf dem genannten Server getestet und dementsprechend auch für dessen Prozessoren optimiert werden.

8.1 Intel Xeon Phi Coprozessor

Beim Intel Xeon Phi handelt es sich um einen Coprozessor, der in Form einer PCIe-Erweiterungskarte (siehe Abbildung 8.1) mit dem Hostsystem verbunden werden kann und numerische Operationen durch Parallelisierung über bis zu 72 (Zahl der Kerne ist modellabhängig) Rechenkerne, sowie durch Vektorisierung unterstützt. Zusätzlich zum Prozessor selbst ist auf der Erweiterungskarte ein DDR5-Arbeitsspeicher verbaut. Bei Bedarf kann der Xeon Phi ein eigenes Betriebssystem ausführen und damit Programme komplett unabhängig vom Hostsystem abarbeiten. (vgl. [19])

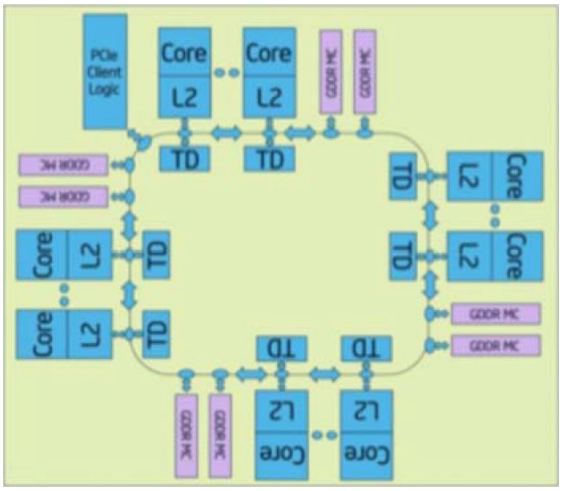


Abbildung 8.2: Mikroarchitektur Knights Corner (Quelle: [21])

8.1.1 MIC-Archtektur

Der Xeon Phi basiert auf der von Intel entwickelten MIC-Architektur. MIC steht dabei für "Many Integrated Cores". Wie in Abbildung 8.2 am Beispiel der Mikroarchitektur Knigths Corner zu sehen ist, kommunizieren die Kerne, der Arbeitsspeicher und die PCIe-Schnittstelle untereinander über einen Ringförmigen Datenbus mit einer Busbreite von 64 Bytes. (vgl. [21]) Zur Verminderung der Verlustleistung und der damit verbundenen Abwärme takten die einzelnen Rechenkerne langsamer als vergleichbare General Purpose CPUs. Der hohe Durchsatz des Coprozessors kann erreicht werden, weil Operationen auf viele Kerne verteilt und Operanden mithilfe der breiten Speicheranbindung zeitnah abgerufen werden können. (vgl. [21])

8.1.2 Anbindung an das Hostsystem

Mit dem Hostsystem ist der Xeon Phi über PCIe x16 verbunden. Dieser Bus lässt eine Bandbreite von 8 GB/s zu. Verglichen mit der Bandbreite der Speicheranbindung des Hostsystems, sowie des Datenbusses innerhalb des Xeon Phi ist diese Bandbreite sehr gering. Abbildung 8.3 vermittelt einen Eindruck über die Busbreiten innerhalb

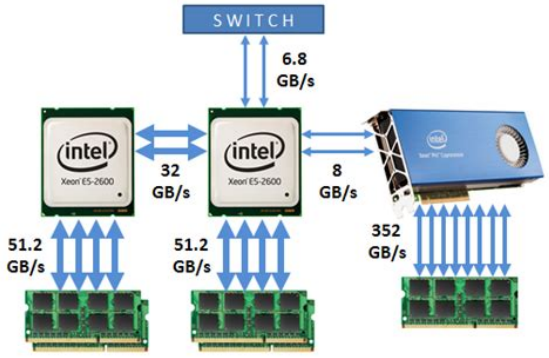


Abbildung 8.3: Busbandbreiten eines Systems mit DDR3-Arbeitsspeicher und Intel Xeon Phi (Quelle: [22])

eines Systems mit zwei Prozessoren, DDR3-Arbeitsspeichern, und einem Rechenbeschleuniger bestehend aus einem Intel Xeon Phi und DDR5-Speicher. (vgl. [22]) In Anwendungen, bei denen Daten zwischen dem Hostsystem und dem Xeon Phi geteilt werden müssen, ist die Übertragung dieser Daten ein kritischer Engpass. Der Aufwand zur Kommunikation zwischen den beiden Systemen hat großen Einfluss auf die Performance des Gesamtsystems. Der Programmierer hat dafür Sorge zu tragen, dass der Datenaustausch zwischen dem Host und dem Rechenbeschleuniger möglichst effizient verläuft. Die Beachtung folgender Faustregeln trägt zu einer hohen Effizienz des Offloads bei:

- Daten zum Coprozessor transferieren und dort behalten (vgl. [21])
- Möglichst große Rechenoperationen auslagern (vgl. [21])
- Auf dem Coprozessor bereits existierende Daten wiederverwerten (vgl. [21])

8.2 Nutzungsmodelle

Zur Planung einer Software, die einen Xeon Phi benutzen soll, gehört die Überlegung, welche Operationen sich zur Ausführung auf dem Coprozessor eignen und welche Rechenschritte auf dem Hostsystem ausgeführt werden sollten. Abhängig von dieser Verteilung sollte bestimmt werden, auf welche Weise der Coprozessor verwendet werden soll. Es kann zwischen mehreren Nutzungsmodellen gewählt werden, die bekanntesten davon werden nachfolgend vorgestellt.

8.2.1 Automatic Offload

Das für den Programmierer einfachste Nutzungsmodell ist Automatic Offload. Bei diesem Modell entscheidet das Programm zur Laufzeit, ob numerische Berechnungen auf dem Coprozessor oder dem Hostsystem ausgeführt werden sollen. Der Programmierer muss dazu keine Vorkehrungen im Quellcode treffen. Die dazu notwendige Entscheidungslogik ist bereits in einigen vorkompilierten Bibliotheken, wie der Math Kernel Library von Intel enthalten. Das bedeutet, dass sich der Programmierer bei der Nutzung von Funktionen dieser Bibliotheken keine Gedanken darüber machen muss, wann eine Auslagerung an den Coprozessor sinnvoll ist oder wie diese veranlasst wird. Ein Nachteil dieses Modells besteht allerdings darin, dass es sich nur auf dafür geeignete Bibliotheksfunktionen anwenden lässt. Zur Auslagerung von eigenem Anwendungscode lässt sich dieses Modell nicht anwenden.

8.2.2 Compiler-Assisted Offload

Für den Fall, dass aufwändige Abschnitte des Anwendungscodes auf den Coprozessor ausgelagert werden sollen, kann dies dem Compiler explizit mitgeteilt werden. In C/C++ lässt sich dies mithilfe des Pragmas `#pragma offload` bewerkstelligen. (vgl. [23]) Das nachfolgende Codebeispiel soll verdeutlichen, wie dieses Pragma anzuwenden ist.

```
float reduction(float *data, int size)
{
        float ret = 0.f;
        for (int i=0; i<size; ++i)
        {
                ret += data[i];
        }
        return ret;
}
```

Listing 8.1: Beispiel für eine Funktion, die ausgelagert werden soll (Quelle: [23])

```
float reduction(float *data, int size)
{
        float ret = 0.f;
        #pragma offload target(mic) in(data:length(size))
        for (int i=0; i<size; ++i)
        {
                ret += data[i];
        }
        return ret;
}
```

Listing 8.2: Beispielfunktion mit pragma zur Auslagerung(Quelle: [23])

Wie in den obigen Beispielen zu sehen ist, genügt bereits das Hinzufügen einer Zeile zum Quellcode, um einen zusammenhängenden Codeabschnitt (in diesem Beispiel die `for`-Schleife) an den Coprozessor auszulagern. Die Angabe `target(mic)` legt dabei fest, dass an einen Coprozessor mit der MIC-Architectur ausgelagert werden soll. Die Angabe `in(data:length(size))` teilt dem Compiler mit, dass das Array `data` mit der Größe `size` zum Coprozessor übertragen hin, aber nicht mehr zurück übertragen werden muss. Für die Variable `ret` gibt es keine explizite Angabe, daher wird sie per Default vor der Berechnung zum Coprozessor hin und anschließend zum Hostsystem zurück übertragen. (vgl. [23])

Auch wenn dieses Pragma im Quellcode angegeben ist, gibt es keine Garantie dafür, dass der betroffene Codeabschnitt tatsächlich vom Coprozessor bearbeitet wird. Laut dem C99-Standard ist es dem Compiler überlassen, wie Pragmas in einem C-Programm zu behandeln sind. Pragmas sind sogar explizit dafür vorgesehen, compilerspezifische Features zu steuern. Wenn das Programm mit einem Compiler übersetzt wird, der nicht für den Offload auf einen Xeon Phi vorgesehen ist, dann sollte er dieses Pragma ignorieren und bestenfalls mit einer Warnung auf das unbekannte Pragma hinweisen. (vgl. [24]) Selbst wenn der Compiler Offload-Code für einen Xeon-Phi erzeugt, dann muss das noch nicht zwangsläufig heißen, dass der Offload tatsächlich durchgeführt wird. Wenn der Coprozessor zur Laufzeit nicht Verfügbar ist, dann wird der auszulagernde Codeabschnitt trotzdem auf dem Hostsystem ausgeführt. (vgl. [23])

Compiler Assisted Offload ist dann hilfreich, wenn Teile des Anwendungscodes auf den Coprozessor ausgelagert werden sollen. Es liegt jedoch in der Verantwortung des Programmierers, geeignete Codeabschnitte auszuwählen und die zu übertragende Datenmenge gering zu halten.

8.2.3 Native

Eine dritte Möglichkeit besteht darin, Anwendungen nativ auf dem Coprozessor laufen zu lassen. Dazu muss die Anwendung mit unter Angabe einiger zusätzlicher Compileroptionen gebaut werden. Die dabei erstellte Binary ist dann auf dem Hostsystem selbst nicht lauffähig. Sie lässt sich allerdings auf den Xeon Phi übertragen. Dieser verfügt über ein auf dem Linux-Kernel basierendes Betriebssystem, auf dem die erzeugte Binary ausgeführt werden kann. Bei einer nativen Programmausführung muss nur einmalig die Binary zum Xeon Phi übertragen werden. Zur Laufzeit ist dann keine Kommunikation mit dem Hostsystem notwendig. Auf diese Weise kann der Flaschenhals zwischen den beiden Systemen umgangen werden. (vgl. [23])

Es ist allerdings nicht für jede Software von Vorteil, nativ auf dem Coprozessor zu laufen. Die Performance bei der Ausführung serieller Programmabschnitte nimmt bei diesem Nutzungsmodell deutlich ab, da der Xeon Phi für diese Aufgaben nicht optimiert ist. Die Performance bei der Ausführung von parallel berechenbaren Operationen nimmt nicht immer signifikant zu oder ab, denn sobald alle notwendigen Instruktionen und Operanden zur Verfügung stehen, wird die eigentliche Berechnung auf die gleiche Art ausgeführt. Was tatsächlich eingespart wird, sind die Datentransfers zwischen dem Hostsystem und dem Coprozessor. Aufgrund dieser Informationen kann eingeschätzt werden, ob sich eine native Programmausführung gegenüber einer Offload-Variante lohnt: Bei seltenen Wechseln zwischen längeren seriellen und parallelen Abschnitten sollte das Programm auf dem Hostsystem ausgeführt werden und bei Bedarf Rechenoperationen an den Xeon Phi auslagern. Wenn es erforderlich ist, das Programm in viele kurze serielle und parallele Abschnitte zu unterteilen, dann nimmt die Wahrscheinlichkeit zu, dass sich die native Ausführung der gesamten Anwendung auf dem Xeon Phi lohnt. (vgl. [23])

8.3 Compiler

Zum Übersetzen des in diesem Abschnitt zu erstellenden Programms kommt der Intel C++ Compiler zum Einsatz. Dieser wurde von Intel speziell zur Übersetzung und Optimierung von Code für Intels Prozessoren entwickelt, ist aber nicht auf diese Anwendungen beschränkt. Die erstellten Binaries sind auf allen Prozessoren ausführbar, die die Architekturen Intel64 oder IA-32 unterstützen. Darüber unterstützt der Compiler die Auslagerung von parallelisierbaren Rechenoperationen auf Intels Rechenbeschleuniger. Zu diesen gehören GPUs, die gemeinsam mit den eigentlichen CPUs in einigen Intel-Prozessoren enthalten sind, sowie Coprozessoren auf Basis der Intel MIC-Architektur, also auch der Xeon Phi. (vgl. [25])

Zur Erzeugung eines Programms, das auf mehrere Prozessoren verschiedener Architekturen verteilt werden soll, muss der Compiler heterogene Programme erzeugen. Bei der Nutzung eines Xeon-Phi und eines Nutzungsmodells, das Offload verwendet, bedeutet dies, dass auslagerbare Codeabschnitte mehrfach übersetzt werden. Erst zur Laufzeit wird entschieden, welcher Prozessor die Operation tatsächlich ausführt, daher kann auch erst dann entschieden werden, welche Variante für den ausführenden Prozessor geeignet ist. (vgl. [25])

Der Intel C++ Compiler übersetzt C/C++ standardmäßig nach den Standards C++98(ISO/IEC 14882:1998) und C90(ISO/IEC 9899:1990), unterstützt aber auch die meisten Features von C++11 und C99. Der zur Intels Compiler-Toolchain gehörende Linker kann darüber hinaus externe Objekte einbinden, die von anderen Compilern wie dem Intel Fortran Compiler oder auch demm GCC erzeugt wurden. (vgl. [25])

8.4 Multithreading

Der Intel C++ Compiler unterstützt mehrere Threading-Bibliotheken, die sich auch innerhalb eines Programms kombinieren lassen. In diesem Abschnitt werden die bekanntesten Vertreter vorgestellt und untereinander verglichen.

8.4.1 PThread

Eine grundlegende Lösung für die Parallelisierung von Prozessen wird mit der C-Bibliothek `pthread` bereitgestellt. Diese Bibliothek definiert einen Datentyp namens `pthread_t`, der alle notwendigen Informationen zur Identifizierung und Verwaltung von je einem Thread beinhaltet. Der Aufbau dieses Datentyps ist abhängig vom zugrundeliegenden Betriebssystem, daher sollte der Anwendercode nur mithilfe der von pthread definierten Funktionen darauf zugreifen. Zum Erzeugen und Starten eines Threads genügt es, die Funktion `pthread_create` mit einem Zeiger auf eine Instanz von `pthread_t`, einem Funktionspointer, einem Pointer für Funktionsargumente, sowie einem weiteren Pointer für optionale Attribute aufzurufen. Zur Identifikation dieses Threads können Zeiger auf die gleiche Instanz von `pthread_t` auch an andere Funktionen übergeben werden. Dies ist zum Beispiel dann notwendig, wenn mehrere Threads untereinander Daten austauschen oder synchronisiert werden müssen. (vgl. [26])

PThread ist auf sehr vielen Plattformen verfügbar. Da es sich dabei nur um eine Bibliothek handelt, sind keine speziellen Funktionalitäten des Compilers notwendig, um nebenläufige Programme auf Basis von pthread zu übersetzen. (vgl. [26])

Problematisch ist allerdings, dass die Verwaltung der Threads vom Anwendercode übernommen wird. Skalierbare Anwendungen, die für verschiedene Prozessoren den Aufwand einer Operation automatisch auf die optimale Zahl von Threads aufteilen, sind damit nur schwer zu realisieren. (vgl. [26])

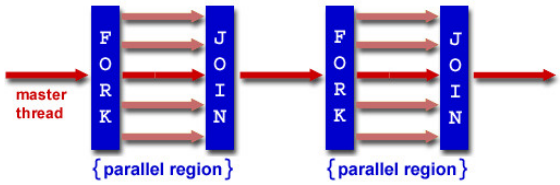


Abbildung 8.4: Das Fork-Join-Modell zur Parallelisierung einzelner Abschnitte

8.4.2 OpenMP

Für eine bessere Skalierbarkeit bei stark nebenläufigen Prozessen sorgt die Verwendung der Multithreadingbibliothek OpenMP. Voraussetzung für die sinnvolle Anwendbarkeit von OpenMP ist allerdings, dass das Programm oder zumindest Teile davon dem Fork-Join-Parllelismus entsprechen. Wie in Abbildung 8.4 zu sehen, beschreibt dieses Modell einen seriellen Programmfluss, bei dem einzelne Codeabschnitte nebenläufig ausgeführt werden. Der Übergang von einem seriellen auf einen parallelen Abschnitt wird dabei als Fork bezeichnet, der Übergang von einem parallelen auf einen seriellen Abschnitt wird Join genannt. (vgl. [27]) Der Anwendungscode muss für die Parallelisierung mittels OpenMP gegenüber einer seriellen Implementierung nur geringfügig verändert werden. Das Threadingverhalten wird dabei größtenteils durch Compileranweisungen gesteuert. Wie im Codebeispiel 8.3 zu sehen ist, genügt es ähnlich wie beim Compiler Assisted Offload (siehe Abschnitt 8.2.2) einzelnen Codeabschnitten ein `#pragma` mit der entsprechenden Anweisung voranzustellen. (vgl. [27])

```
double res[MAX]; int i;
#pragma omp parallel for
for (i=0;i< MAX; i++) {
        res[i] = huge();
}
```

Listing 8.3: Beispielfunktion mit pragma zur Nebenläufigen Ausführung einer for-Schleife(Quelle: [27])

Die in Listing 8.3 gezeigte Anweisung `#pragma omp parallel for` veranlasst den Compiler, einen Code zu erzeugen, der die Schleifendurchläufe nebenläufig ausführt. Die Anzahl der dabei verwendeten Threads lässt sich mit Umgebungsvariablen und Optionen beim Compileraufruf oder durch den Aufruf von dafür vorgesehenen Bibliotheksfunktionen im Quellcode beeinflussen. (vgl. [27]) Darüber hinaus stellt OpenMP noch eine große Zahl von weiteren Compileranweisungen und Bibliotheksfunktionen bereit. Dies ermöglicht es dem Programmierer zum Beispiel, kritische Abschnitte zu definieren, die nur von maximal einem Thread zur gleichen Zeit bearbeitet werden dürfen. Ein weiteres Beispiel sind Compileranweisungen, mit denen sich bestimmen lässt, ob bestimmte Variablen zwischen den Threads geteilt werden oder ob jeder Thread eine eigene Kopie dieser Variablen erhalten soll. (vgl. [27]) Da die erweiterten Funktionalitäten von OpenMP in diesem Teil der Arbeit nicht erforderlich sind, wird an dieser Stelle auch nicht speziell darauf eingegangen. Im Kapitel zur Optimierung für x86-CPUs wird OpenMP näher betrachtet.

Gegenüber pthread hat die Verwendung von OpenMP geringere Auswirkungen auf die Lesbarkeit und Wartbarkeit eines Programms. Es gibt allerdings Fälle, in denen das Fork-Join-Modell und damit auch OpenMP nicht sinnvoll angewendet werden können. Außerdem werden die zu OpenMP gehörenden `#pragma`-Anweisungen nicht von jedem Compiler unterstützt. Compiler, die diese Anweisungen nicht interpretieren können, sollten diese ignorieren. Dies hat zur Folge, dass der Compiler ein serielles Programm erzeugt, das zwar die gleichen Ergebnisse liefert, dabei aber nicht von Multithreading profitieren kann. (vgl. [27])

8.4.3 Intel Threading Building Blocks

Intel TBB (Threading Building Blocks) ist eine Threadingbibliothek, die Intel für die Programmiersprache C++ entwickelt hat. TBB verfolgt im Gegensatz zu anderen Modellen keinen threadbasierten, sondern einen taskbasierten Ansatz. Dies bedeutet, dass der Programmierer parallelisierbare Operationen nicht mehr in Threads, sondern in einzelne Aufgaben zerlegt, die dann von einem Taskscheduler auf mehrere Threads verteilt werden. Jedes Programm, das mithilfe von TBB zusätzliche Threads erzeugen und verwalten soll, muss zunächst einen Taskscheduler erstellen. Dazu genügt es, ein Objekt des Typs `task_scheduler_init` zu instanziieren. Listing 8.4 zeigt ein minimales Beispiel, das einen Taskscheduler erstellt und dann endet. (vgl. [28])

```
//comment
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
using namespace tbb;
int main() {
        task_scheduler_init init;
        return 0;
}
```

Listing 8.4: Minimalbeispiel zur Verwendung von Intel TBB(Quelle: [29])

Außerdem ist es möglich, die Aufgaben zu Gruppen zusammenzufassen und Eigenschaften oder Beziehungen für diese festzulegen. Auf diese Weise lassen sich auch komplexe Operationen mit kritischen und atomaren Abschnitten und Barrieren definieren. Der Scheduler verteilt die zu bearbeitenden Aufgaben dann ohne weiteres Zutun des Programmierers unter Beachtung der festgelegten Anforderungen auf

möglichst viele Threads. (vgl. [28]) Listing 8.5 zeigt beispielhaft, wie die Erstellung und Ausführung neuer Tasks aussehen kann. Es zeigt eine Funktion zur rekursiven Berechnung eines Elements der Fibonacci-Folge. In jedem Rekursionsschritt (falls die Abbruchbedingung nicht erfüllt ist) werden zwei neue Threads gestartet, die die beiden vorherigen Elemente auf die gleiche Art rekursiv berechnen. Wenn beide Threads beendet sind, endet auch die Funktion selbst.

```
{
#include "tbb/task_group.h"

using namespace tbb;

int Fib(int n) {
        if( n<2 ) {
                return n;
        } else {
                int x, y;
                task_group g;
                g.run([&]{x=Fib(n-1);}); // spawn a task
                g.run([&]{y=Fib(n-2);}); // spawn another
                    task
                g.wait();                // wait for both
                    tasks to complete
                return x+y;
        }
}
}
```

Listing 8.5: Rekursive Funktion mit TBB zur Threadverwaltung(Quelle: [28])

Wie bei OpenMP ist es auch mit TBB möglich, mit nur geringfügigen Codeänderungen Schleifen nebenläufig auszuführen. Bei TBB geschieht dies allerdings nicht mithilfe von Compileranweisungen, sondern mittels eines in der Bibliothek definierten Templates, das anstelle der `for`-Schleife verwendet wird. Als Beispiel für diese Ersetzung dienen die Listings 8.6 und 8.7. Das im zweiten Listing gezeigte Template erwartet als Parameter die Grenzen der Laufvariable und einen Lambdaausdruck mit den in der Schleife auszuführenden Operationen. (vgl. [28])

```
void SerialApplyFoo( float a[], size_t n ) {
        for( size_t i=0; i!=n; ++i ) {
                Foo(a[i]);
        }
}
```

Listing 8.6: Funktion mit serieller for-Schleife(Quelle: [28])

```
#include "tbb/tbb.h"

using namespace tbb;

void ParallelApplyFoo( float a[], size_t n ) {
        tbb::parallel_for( size_t(0), n, [&]( size_t i ) {
                Foo(a[i]);
        } );
}
```

Listing 8.7: Funktion mit parallel-for-Template(Quelle: [28])

Im Gegensatz zu OpenMP arbeitet TBB nicht mit `#pragma`-Anweisungen und ist somit nicht auf die Unterstützung des verwendeten Compilers angewiesen. Allerdings existiert TBB derzeit nur für die Programmiersprache C++. Für viele Features werden Lambda-Ausdrücke benötigt, die erst ab dem Sprachstandard C++11 definiert sind. Die Verwendung eines Taskschedulers erzeugt zwar zur Laufzeit einen geringen Overhead, bei hinreichend großen und komplexen Operationen lässt sich mit TBB allerdings trotzdem eine höhere Performance erzielen als mit OpenMP. (vgl. [29])

8.5 Intel Math Kernel Library

Wie die einführenden Kapitel gezeigt haben, lässt sich die Berechnung von künstlichen Neuronalen Netzwerken auf eine Reihe von mathematischen Standardoperationen zurückführen, die auch in anderen wissenschaftlichen Bereichen für numerische Berechnungen eingesetzt werden. Aufgrund dieses Bedarfs existieren für viele Prozessoren und Rechenbeschleuniger Bibliotheken, die diese Operationen sehr effizient und für die jeweilige Zielplattform optimiert implementieren. Intel hat für seine Prozessoren die Bibliothek MKL (Math Kernel Library) entwickelt. MKL stellt unter anderem Funktionen für Matrix- und Vektoroperationen, vektoriesierte Berechnung von mathematischen Funktionen, sowie Generierung von Zufallszahlen bereit. Darüber hinaus unterstützen viele Funktionen der MKL auch die automatische Auslagerung an Coprozessoren wie den in diesem Teil der Arbeit verwendeten Xeon Phi. Intel stellt vorkompilierte Versionen der MKL für die Architekturen IA-32, Intel64, sowie MIC zur Verfügung. Somit kann MKL nicht nur für Offload-Modelle zum Einsatz kommen, sondern auch nativ auf dem Xeon Phi verwendet werden. (vgl. [30])

8.5.1 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms (BLAS) ist ein De-facto-Standard für Bibliotheken, die Funktionen zur Berechnung von Vektor-Vektor-Operationen, Matrix-Vektor-Operationen und Matrix-Matrix-Operationen bereitstellen. Für diese Bibliotheken gibt es viele hochperformante Implementierungen, die von Hardwareherstellern speziell für deren Prozessoren optimiert wurden. Diese sind untereinander relativ leicht austauschbar, da die darin definierten Funktionen einheitlichen Namenskonventionen folgen und identische Parameterlisten besitzen. Auch in MKL ist eine BLAS-Bibliothek enthalten. (vgl. [30])

Als Beispiel für eine BLAS-Funktion sei an dieser Stelle die Funktionsdeklaration von `cblas_sgemm` in Listing 8.8 genannt. Die Funktion beschreibt eine Operation der Form $C := alpha * op(A) * op(B) + beta * C$, wobei A, B und C Matrizen sind. *alpha* und *beta* sind in dieser Gleichung Skalare. Der Operator *op* kann gibt an, dies die Matrix in dessen Argument transponiert werden kann. (vgl. [30])

```
void cblas_sgemm (const CBLAS_LAYOUT Layout, const
    CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
    const MKL_INT m, const MKL_INT n, const MKL_INT k, const
     float alpha, const float *a, const MKL_INT lda, const
    float *b, const MKL_INT ldb, const float beta, float *c,
     const MKL_INT ldc);
```

Listing 8.8: Deklaration einer BLAS-Funktion zur Matrix-Matrix-Multiplikation(Quelle: [30])

Nachfolgend werden die Funktionsparameter aus Listing 8.8 näher erläutert.

Layout: enum-Wert, der angibt, ob das Array mit den Zahlenwerten der Ergebnismatrix column-major oder row-major zu lesen ist.

transa: enum-Wert, der angibt, ob A vor der Matrix-Matrix-Multiplikation transponiert werden soll. Das übergebene Array bleibt dabei unverändert, der Funktion wird nur mitgeteilt, dass das Array entgegen der in `Layout` angegebenen Leserichtung zu lesen ist.

transb: wie `transa`, bezieht sich aber auf Matrix B.

m: Anzahl der Zeilen in Matrix C

n: Anzahl der Spalten in Matrix C

k: Anzahl der Spalten in Matrix $op(A)$ bzw. der Zeilen in Matrix $op(B)$

alpha: Wert des Skalars *alpha*

a: Pointer auf ein Array mit den Werten der Matrix A

lda: Leading Dimension der Matrix A. Dieser Parameter ist nützlich, wenn das Array `a` eine größere Matrix enthält, von der nur ein Teil betrachtet werden soll. Je nach Leserichtung der Matrix gibt dieser Wert den Abstand zwischen dem ersten Element der ersten Zeile/Spalte und dem ersten Element der zweiten Zeile/Spalte an.

b: Pointer auf ein Array mit den Werten der Matrix B

ldb: Wie `lda`, bezieht sich aber auf Matrix B

beta: Wert des Skalars *beta*

c: Pointer auf ein Array mit den Werten der Matrix C

ldc: Wie `lda`, bezieht sich aber auf Matrix C

Wie das Beispiel zeigt, müssen die Matrizen nicht in Objekte gekapselt werden. Stattdessen benötigen die BLAS-Funktionen Arrays mit Zahlenwerten und zusätzliche Informationen, wie diese als Matrizen zu interpretieren sind. Für diese Interpretation wird dem Programmierer ein großer Spielraum eingeräumt; es sind alle Kombinationen von Leserichtungen erlaubt und für jede Matrix lässt sich eine Leading Dimension angeben. Eine Ausnutzung dieser Flexibilität ermöglicht es, aufwändiges Kopieren und Umstrukturieren der zu verarbeitenden Daten zwischen den tatsächlichen Rechenoperationen weitgehend zu vermeiden. (vgl. [30])

8.5.2 Vector Mathematical Functions

Mithilfe der BLAS-Funktionen lassen sich Aufgaben aus dem Gebiet der linearen Algebra mit geringem Programmieraufwand auch für größere Datenmengen performant berechnen. Allerdings müssen in neuronalen Netzwerken auch nichtlineare Funktionen für grüßere Datenmengen berechnet werden. Notwendig ist dies speziell bei der Aktivierungsfunktion und der Ermittlung der Kosten. Für die vektorisierte Berechnung von nichtlinearen Operationen stellt MKL zusätzliche Funktionen bereit. Ein Beispiel liefert die Funktion `vsTanh( n, a, y );` Diese berechnet den Tangens Hyperbolicus von allen Elementen des Vektors `a` und speichert die Ergebnisse in Vektor `y`. Aus Sicht des Programmierers sind `a` und `y` Pointer auf Arrays der Größe `n`. Im Gegensatz zu BLAS-Routinen wird die Ausführung dieser Funktionen nicht auf mehrere CPU-Kerne verteilt. Dennoch werden die Berechnungen innerhalb eines Kerns unter Verwendung spezieller Hardware für SIMD-Berechnungen(Single Instruction Multiple Data) parallelisiert. (vgl. [30])

8.5.3 Statistical Functions

Entscheidend für die Genauigkeit, die ein CNN erreichen kann, ist die Verteilung der Gewichte im Initialzustand. Idealerweise sollte der Durchschnitt aller Gewichte bei 0 liegen, wobei die Gewichte mit geringer Varianz verteilt sein sollten, sodass sich die Featuremaps innerhalb eines Layers unterschiedlich entwickeln können. (vgl. [6])

Zur Generierung von Zufallszahlen, mit denen die Gewichte initialisiert werden können, stellt Intel in MKL eine Reihe von Zufallszahlengeneratoren zur Verfügung. (vgl. [30]) Die Anforderungen an die Geschwindigkeit sind bei der Initialisierung der Gewichte niedriger als bei der späteren Anpassung, da die Initialisierung nur einmalig zum Start des Programms erfolgt. Entscheidend für die Genauigkeit des Netzwerks nach dem Training ist, dass die Verteilung der Anfangsgewichte den oben genannten Anforderungen entspricht. Für die in diesem Teil zu erstellende Implementierung wird eine Gaußverteilung gewählt.

8.6 Grundlegende Überlegungen

Der eigentlichen Implementierung des Trainingsprogramms für das CNN gehen einige konzeptionelle Überlegungen voraus. In diesem Abschnitt werden einige grundlegende Designentscheidungen festgehalten, die Einfluss auf die Programmarchitektur und damit indirekt auf die Effizienz und Änderbarkeit des entstehenden Programms haben werden.

8.6.1 Parallelisierung

Das Trainieren des Netzwerks ist im Wesentlichen ein iterativer Prozess, der seriell abgearbeitet werden muss. Nebenläufig abarbeiten lässt sich nicht der gesamte Prozess, sondern nur einzelne Operationen innerhalb des Prozesses. Konkret bedeutet das zum Beispiel, dass zunächst die Fehler der Gewichte berechnet werden müssen, bevor auf Basis dieser Fehler die Anpassung der Gewichte vorgenommen werden kann. Beide Operationen für sich können durch Nebenläufigkeit beschleunigt werden, allerdings darf nicht mit der Bearbeitung des zweiten Schritts begonnen werden, bevor der erste Schritt abgeschlossen ist. Für diese Arbeit bedeutet dies, dass das Fork-Join-Modell zur Parallelisierung am besten geeignet ist. Viele Routinen aus der MKL-Bibliothek sind bereits so implementiert, dass sie automatisch nebenläufig ausgeführt und bei Bedarf ausgelagert werden. Für parallelisierbaren Anwendungscode wird OpenMP verwendet, da diese Bibliothek am einfachsten zu verwenden ist. Es darf angenommen werden, dass die dabei entstehenden Threads nicht auf geteilte Variablen schreiben und daher keine zusätzliche Synchronisierung notwendig ist. Somit ließe sich durch die Verwendung von TBB im Anwendungscode gegenüber OpenMP wohl kein nennenswerter Performancegewinn erzielen.

8.6.2 Programmiersprache

Um eine möglichst gute Performance zu erzielen, muss eine Sprache verwendet werden, die zu Maschinencode kompiliert werden kann. Andernfalls müsste das Programm zur Laufzeit interpretiert werden, was zusätzlichen Aufwand verursachen würde. Die verwendeten Bibliotheken OpenMP, sowie MKL unterstützen die Sprachen C, C++ und Fortran. Die Implementierung des Programms erfolgt größtenteils in C, eine Ausnahme stellt der Code dar, der die Eingabedaten und Labels des MNIST-Datensatzes von der Festplatte einliest. Dieser Code wird mit

geringen Veränderungen von der seriellen Implementierung übernommen und benötigt Funktionen, die in C++-Bibliotheken definiert sind und in C nicht direkt verwendet werden können. Die Deklarationen der in C++ geschriebenen und von C-Code aufgerufenen Funktionen werden mit dem Schlüsselwort `extern "C"` versehen, sodass sie anschließend vom Linker gebunden werden können.

Die Entscheidung für die Programmiersprache C ist mit den bevorzugten Paradigmen der Programmiersprachen C und C++ zu begründen. C ist eine strukturierte Programmiersprache, während C++ objektorierte Ansätze verfolgt. Zu diesen Ansätzen gehören auch Vererbungshierarchien und Polymorphie. Diese Ansätze erleichtern dem Programmierer zwar die Arbeit, allerdings macht die Verwendung dieser Ansätze dynamische Typbindung erforderlich. Das bedeutet, dass beim Aufruf von Funktionen, die in einer Vererbungshierarchie mehrfach implementiert wurden, zur Laufzeit entschieden wird welche Implementierung letztendlich aufgerufen wird. Die Verwendung von dynamisch gebundenen Funktionen macht den Code langsamer und wird aus diesem Grund bewusst vermieden. Auch bestimmte Design Patterns, die auf mehrfacher Delegation von Funktionsaufrufen zwischen mehreren ähnlichen Objekten basieren wird bewusst abgesehen.

Tatsächlich soll der serielle Teil des zu erstellenden Programms möglichst schnell nacheinander nebenläufige Funktionen für die eigentlichen Rechenoperationen aufrufen. Der Code, der zwischen den Aufrufen steht, sollte möglichst kurz und möglichst sequentiell sein, also keine unnötigen Verzweigungen enthalten.

8.6.3 Codegenerierung

Zu den Anforderungen gehört auch eine Änderbarkeit des verwendeten Netzwerkmodells und der Programmkonfiguration. Eine gute Änderbarkeit ist dann gegeben, wenn das Netzwerkmodell an nur einer Stelle geändert werden muss, um das Programm anzupassen. Die gebräuchlichste Möglichkeit, diese Änderbarkeit zu

erreichen, ist die Definition einer Liste von Objekten einer gemeinsamen Basisklasse, die die einzelnen Netzwerkschichten repräsentieren. Zur Veränderung des Netzwerkmodells müsste nur die Initialisierung dieser Liste geändert werden. Bei der Simulation des Netzwerkes könnte dann auf verschiedene Implementierungen von Methoden zurückgegriffen werden, die für die verschiedenen Layertypen definiert sind. Dieser Ansatz steht allerdings in Widerspruch zur Entscheidung, auf dynamische Typbindung und somit Vererbung zu verzichten. Um eine objektorientierte Beschreibung des Netzwerks zu ermöglichen und gleichzeitig alle Strukturen und Methoden statisch zu binden, wird die Implementierung des CNNs für den Xeon Phi in zwei Programme aufgeteilt:

Der erste Teil ist ein Python-Skript, das ähnlich wie bei Tensorflow ein neuronales Netzwerk anhand seiner Schichten und deren Eigenschaften beschreibt. Außerdem werden in diesem Skript einige Einstellungen festgelegt, die das Verhalten des Programms während des Trainings festlegen.

Der zweite Teil ist das eigentliche Programm zum Trainieren des Netzwerks. Dieses ist in C geschrieben und benötigt an mehreren Stellen Informationen über den Aufbau des Netzes und Details über einzelne Schichten. An diesen Stellen wird C-Code generiert, der alle notwendigen Informationen bereits enthält. Teilweise können dadurch Berechnungen schon vor der Übersetzung durchgeführt und deren Ergebnisse als Konstanten in den Code eingefügt werden. Die Verwendung von Konstanten anstatt zur Laufzeit berechneter Variablen ermöglicht es dem Compiler, den ausführbaren Code zusätzlich zu optimieren und Speicherzugriffe zu reduzieren.

Zur Codegenerierung wird das Python-Tool `cog` verwendet. Dieses Tool kann Textdateien nach speziell formatierten Zeilen durchsuchen. Diese Zeilen markieren den Anfang und das Ende eines Python-Codes. Dieser Python-Code erzeugt eine Ausgabe, die zwischen dem Ende des markierten Abschnitts und einer dritten Markierung eingefügt wird. Die Markierungen sind so gewählt, dass der Python-Code aus Sicht des C-Programms wie ein Kommentar erscheint. (vgl. [31])

Zum Ändern des Netzwerkmodells oder der Konfiguration genügt es, das entsprechende Python-Skript anzupassen und anschließend `cog` mit einigen Quellcodedateien des C-Codes aufzurufen. Der dabei entstehende Code kann anschließend in ein lauffähiges Programm übersetzt werden, das der zuvor festgelegten Konfiguration entspricht.

8.6.4 Datenorganisation

Bei der seriellen Implementierung werden zur Laufzeit mehrmals pro Trainingsdurchlauf temporäre Matrizen erzeugt, für die jeweils Speicher alliziert werden muss. Die Speicherallozierung ist ein zeitraubender Prozess, daher wird in diesem Teil der Arbeit schon früh ein Konzept zur Speicherverwaltung erarbeitet.

Für temporäre Zwischenergebnisse wird in der Initialisierungsphase des Programms ein zusammenhängender Speicherbereich reserviert und für den gesamten Rest der Laufdauer im Speicher gehalten. Funktionen, die temporäre Zwischenergebnisse ablegen müssen, können diesen Speicherbereich verwenden. Es ist allerdings sicherzustellen, dass keine Ergebnisse überschrieben werden, die später noch benötigt werden. Die Größe dieses Speichers hängt vom verwendeten Netzwerkmodell und der Batchsize ab, aus diesem Grund wird der Code zur Speicherreservierung generiert.

Beim Durchlaufen des Netzwerks (sowohl vorwärts als auch rückwärts) werden in jedem Layer Aktivierungen bzw. Fehler berechnet, die vom nachfolgenden bzw. vorhergehenden Layer als Eingaben benötigt werden. Diese Aktivierungen sollten nicht von einem Layer zum nächsten kopiert werden, denn dies wäre eine unnötige Operation. Stattdessen sollen beide Layer je einen Pointer besitzen, der auf die gleiche Speicheradresse zeigt. Diese Pointer können ohne weitere Verarbeitung zum Aufruf von MKL-Funktionen verwendet werden. Da beide Pointer auf einen gemeinsamen Speicherbereich zeigen, stellt sich die Frage, welcher Layer für diesen Speicherbereich verantwortlich ist, also diesen Initialisieren soll. Außerdem müsste dieser Layer seinen Nachbarn über die Speicheradresse informieren, was erforderlich machen würde, dass die Layer Kenntnis voneinander besitzen. Um dies zu vermeiden, wird der Speicher für sämtliche Aktivierungen als ein zusammenhhängender Speicher reserviert und die Startadresse in einer Struktur festgehalten, die das gesamte Netzwerk repräsentiert. Diese Struktur beinhaltet weitere Strukturen, die die einzelnen Layer repräsentieren. Darin finden sich Pointer, die in den reservierten Speicherbereich hineinzeigen. Sowohl die Definition der Netzwerkstruktur, als auch die Funktion zur Initialisierung müssen mit `cog` generiert werden, da die Abfolge der Layer, deren Größenangaben und die daraus ermittelbare Gesamtgröße des Aktivierungsfeldes vom Aufbau Netzwerkmodells abhängen. Jeder Aktivierung lässt sich genau ein Fehler zuordnen, der in der Backpropagation ermittelt werden kann. Daher kann in der Netzwerkstruktur ein zweiter Speicherbereich mit identischer Größe erzeugt werden, für den die Layer zusätzliche Pointer mit identischen Offsets innerhalb des Feldes erhalten.

Ähnlich wie die Aktivierungen werden auch die Gewichte und Biases aller Layer des Netzwerks in einem zusammenhängenden Array gespeichert. Wie zuvor wird auch ein zweites Array mit gleicher Größe und Einteilung für die Fehler der Aktivierungen angelegt. Obwohl für die Gewichte eindeutig feststellbar ist, zu

welchen Layern sie gehören, ist diese Einteilung von Vorteil: Bei der Durchführung des Stochastic Gradient Descent müssen die berechneten Fehler der Gewichte skaliert und von den eigentlichen Gewichten subtrahiert werden. Wenn diese jeweils in einem großen Array statt in vielen kleinen Feldern stehen, dann kann die Hardware zur Vektorisierung für den Stochastic Gradient Descent besser eingesetzt werden.

Die Trainings- und Testdaten werden während der Initialisierung vollständig geladen und während der gesamten Laufzeit im Speicher gehalten. Um unnötige Rechenschritte einzusparen, sollen die Eingabedaten nicht vor der Vorwärtspropagierung in das Aktivierungsfeld kopiert werden. Stattdessen soll der erste Layer einen Pointer erhalten, der auf die Eingabedaten verweist. Für die Vorwärts- und Rückwärtsberechnung innerhalb eines Layers werden aus diesm Grund je zwei Funktionen geschrieben. Eine der beiden Funktionen beschreibt das normale Verhalten des Layers, die andere den Sonderfall, dass es sich um den ersten Layer handelt und ein zusätzlich übergebener Pointer für die Eingabedaten verwendet werden muss. Auch wenn die Codesize und der Wartungsaufwand wegen des redundanten Codew steigen, lohnt sich diese Art der Implementierung: Welcher Layer der erste im Netzwerk ist, lässt sich schon im Schritt der Codegenerierung bestimmen. In jedem Trainingsschritt für jeden Layer und die Berechnung in beide Richtungen mithilfe von Verzweigungen die zu verwendende Eingabeadresse zu bestimmen, wäre ein signifikanter Mehraufwand zur Laufzeit des Programms.

8.6.5 Convolution und Pooling

Im vorherigen Abschnitt wurde bereits erwähnt, dass jeweils zwei benachbarte Layer gemeinsame Speicherbereiche zum Austausch von Aktivierungen und deren Fehlern verwenden. Damit die darin enthaltenen Ergebnisse richtig interpretiert werden können, benötigen diese Layer aber zusätzliche Informationen. Grundsätzlich lässt eine Anordnung von Aktivierungen im Ausgang einer Schicht als vierdimensionaler

Tensor verstehen. Die Größe des Tensors ist aufgrund des Netzwerkmodells bekannt, für die Anordnung der Zahlenwerte im Speicher wird an dieser Stelle eine Konvention festgelegt. Die Ablage der Aktivierungswerte erfolgt im Normalfall nach dem Schema BYXF. Die einzelnen Buchstaben lassen sich ähnlich wie Ziffern in einem Zahlensystem interpretieren interpretieren: Werden zwei aufeinanderfolgende Features(F) an der gleichen Position(Y-X) innerhalb des gleichen Elements einer Minibatch(B) betrachtet, dann stehen die dazugehörigen Werte direkt hintereinander im Speicher. Eine Position innerhalb eines vierdimensionalen Tensors kann somit auch als eine Zahl des Schemas BYXF interpretiert werden; diese wiederrum lässt sich als Offset innerhalb eines eindimensionalen Arrays bzw. zusammenhängenden Speicherbereichs verstehen, das alle Aktivierungen des Tensors enthält.

Es gibt allerdings eine Ausnahme: Nach einem Convolutional Layer werden die Daten in einer anderen Reihenfolge abgelegt. Grund dafür ist eine Optimierung der Faltungsoperation, die Auswirkungen auf die Struktur des Ergebnisses hat. Pooling Layer sind in dieser Implementierung darauf ausgelegt, sowohl die normale Reihenfolge nach dem Schema BYXF, als auch die Ausgabe eines Convolutional Layers als Eingabe verarbeiten zu können. Für alle anderen Layertypen gilt dies nicht, aus diesem Grund muss auf jeden Convolutional Layer ein Pooling Layer folgen. Da dies bei CNNs normalerweise sowieso der Fall ist, stellt dies in den meisten Fällen keine Einschränkung dar. Sollte es tatsächlich erforderlich sein, dass eine Faltungsoperation ohne nachfolgendes Pooling durchgeführt wird, so muss als Workaround ein Pooling Layer mit der Filtergröße 1x1 in das Netzwerkmodell eingefügt werden. Tatsächlich findet dann keine Reduktion der Datenmenge statt, aber die zu verarbeitenden Daten werden in eine Reihenfolge gebracht, die von allen nachfolgenden Layern verarbeitet werden kann. Einzelheiten zur Implementierung der Faltungsoperation und des Poolingalgorithmus befinden sich in den entsprechenden Unterkapiteln in diesem Teil der Arbeit.

8.6.6 Buildsystem

Das zu erstellende Programm soll nicht nur auf dem Rechner des Studenten, sondern auch auf dem Server der DHBW und auf weiteren Rechnern übersetzt werden können. Dies macht das ein Buildsystem erforderlich, das die Übersetzung sowohl unter Windows, als auch unter Linux und unixbasierten Systemen ermöglicht.

Der Build des ausführbaren Programms erfolgt in mehreren Schritten:

Konfiguration: Zunächst kann der Nutzer durch Modifikation der dafür vorgesehenen Pythonskripte ein Netzwerkmodell, sowie eine Konfiguration festlegen.

Codegenerierung: Nach einer Konfigurationsänderung muss ein Teil des C-Programms neu generiert werden. Dazu wird das Python-Tool `cog` benötigt. (vgl. [31]) Mit dem Befehl `cog -r @COG_filelist.txt` wird die Codegenerierung gestartet. Die ersten beiten Schritte können auch übersprungen werden, in diesem Fall wird die Konfiguration zum Zeitpunkt der Abgabe dieser Arbeit verwendet.

Build: Nach der Codegenerierung kann das Programm wie jedes andere C-Programm durch Kompilierung aller Codedateien und anschließendes Linken übersetzt werden. Der ICC-Compiler muss unter Windows anders aufgerufen werden als unter Linux, daher unterscheiden sich die Buildprozesse geringfügig. Für die Übersetzung auf Linuxsystemen existiert eine Makefile zur Automatisierung des Buildvorgangs. Wenn GNU make auf dem System installiert ist, dann lässt sich der Build mit `make all` starten. Um zwischen den Usage Modellen Native und Offload wechseln zu können, wird am Anfang der Makefile eine Variable Namens `MK_PHI_USAGE_MODEL` definiert. Eine Änderung des Wertes dieser Variable führt zu einem Build mit dem jeweiligen Nutzungsmodell. Für Windows-Systeme gibt es ein alternatives Buildsystem auf Basis von SCons.

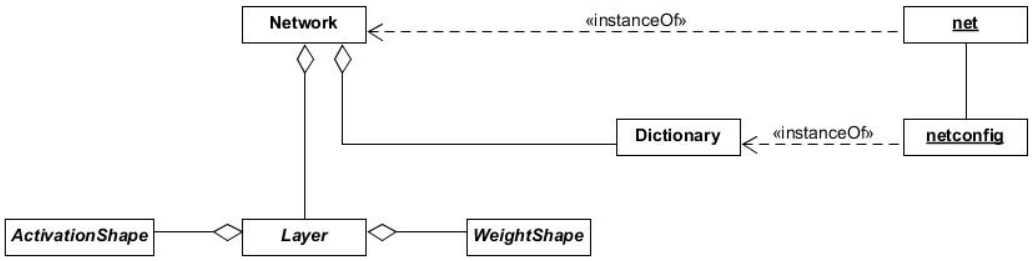


Abbildung 8.5: Klassendiagramm des Skripts zur Netzwerkbeschreibung

Ausführung: Beim Build wird die ausführbare Datei `build/program` erzeugt. Diese Datei beinhaltet das CNN-Trainingsprogramm und kann mit `./build/program` gestartet werden. Alternativ kann mit `make execute` das Programm übersetzt und nach dem erfolgreichen Build automatisch gestartet werden. Die Ausgaben des Programms selbst, sowie eines Tools zur Messung der Ausführungsdauer werden in die Datei `report.txt` geschrieben.

8.7 Skript zur Netzwerkbeschreibung

Wie in Abschnitt 8.6.3 bereits erläutert wurde, wird in dieser Implementierung zusätzlich zum eigentlichen C-Programm ein Pythonskript erstellt, das zur Modellierung des Netzwerkes dient. In diesem Abschnitt wird dieses Skript näher beschrieben.

8.7.1 Klasseneinteilung

Das Klassendiagramm aus Abbildung 8.5 soll zunächst einen groben Überblick über den Aufbau des Netzwerkmodells verschaffen. Repräsentiert wird ein CNN durch die Klasse `Network`. Objekte dieser Klasse besitzen je ein Dictionary zur Festlegung der Konfiguration und eine Liste von Objekten des abstrakten Typs `Layer`. Jeder

`Layer` besitzt zwei Referenzen auf Objekte des Typs `ActivationShape`, wobei eine für die Aktivierungen am Eingang und eine für die Aktivierungen am Ausgang steht. Darüber hinaus besitzt jeder Layer ein Objekt des abstrakten Typs `WeightShape`. `ActivationShape` beschreibt einen Tensor mit Aktivierungen, und beschreibt die Schnittstelle zwischen zwei Schichten bzw. zwischen einer Schicht und dem Ein- oder Ausgang des Netzes.

In der Datei `src/NetworkDescriptor/NetInstance.py` wird mit dem Opjekt `net` eine Instanz der Klasse Network erstellt. Zusätzlich wird aus der Datei `src /NetworkDescriptor/NetConfig.py` ein Dictionary mit den Konfigurationsdaten importiert. Alle Pythonskripte, die für die Codegenerierung verantwortlich sind, importieren das beschriebene Objekt `net` und verwenden die darin enthaltenen Daten, d.h. durch eine Änderung der Definition von `net` und anschließende Codegenerierung lässt sich das Verhalten des C-Programms beeinflussen.

8.7.2 Netzwerk

Zur Erstellung eines Objekts der Klasse `Network` wird eine `ActivationShape` zur Beschreibung des Tensors für die Eingangsdaten, sowie ein Dictionary mit der Konfiguration benötigt.

Ist das Netzwerk einmal erstellt, lassen sich der Reihe nach Schichten zum Netzwerk hinzufügen. Dazu steht für jede instanziierbare Unterklasse von Layer eine Methode zur Verfügung. Diese Methoden rufen die jeweiligen Konstruktoren der entsprechenden Typen auf und übergeben die Argumente, die sie selbst erhalten haben, sowie die `ActivationShape` für die Ausgaben der letzten Schicht des Netzwerks. Mithilfe dieser Informationen generiert der Konstruktor eine `ActivationShape`, die den Ausgangstensor einer Netzwerkschicht mit den durch die Parameter bestimm-

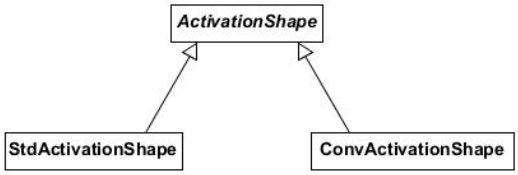


Abbildung 8.6: Unterklassen von ActivationShape

ten Eigenschaften beschreibt. Anschließend wird der neue `Layer` in die Liste der Netzwerkschichten aufgenommen, wobei die gleichzeitig erstellte `ActivationShape` zur letzten des Netzwerks wird und somit dem nächsten hizuzufügenden Layer als Eingangstensor übergeben werden kann.

Nachdem alle Schichten hinzugefügt wurden, muss die Methode `generate` aufgerufen werden. In dieser Methode werden anhand der im Netzwerkmodell enthaltenen Informationen über die Schichten und deren Schnittstellen, zusätzliche Informationen berechnet, die bei der Generierung des C-Codes benötigt werden. Dazu gehören Beispielsweise die Startpositionen für die Tensoren innerhalb des Feldes mit den Aktivierungen bzw. deren Fehlern. Diese Startpositionen entsprechen jeweils der Startposition des vorherigen Layers, erhöht um dessen Größe. Optional kann dieser Betrag zusätzlich auf das nächste Vielfache des Konfigurationsparameters `CONFIG_ARRAY_ALIGNMENT` erhöht werden. Bei der Speicherallozierung der Felder zur Laufzeit des C-Programms wird ebenfalls mit dem gleichen Alignment gearbeitet. Somit ist Sichergestellt, dass alle Adressen, die zur Laufzeit als Beginn von Matrizen verwendet werden, auf dieses Alignment eingestellt sind. Laut der Dokumentation von MKL können die Bibliothekensfunktionen effizienter arbeiten, wenn sie ein Alignment von 256 Byte aufweisen. (vgl. [30])

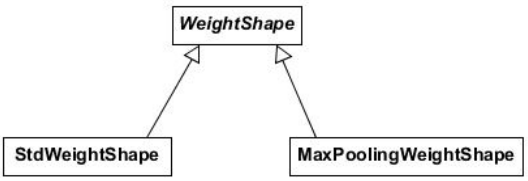


Abbildung 8.7: Unterklassen von WeightShape

8.7.3 Schnittstellen zwischen Schichten

Wie im Abschnitt 8.6.5 bereits vorweggenommen wurde, erzeugt ein Convolutional Layer eine Ausgabe, deren Größe und Reihenfolge sich von denen der anderen Schichttypen unterscheidet. Diese Besonderheit muss im Netzwerkmodell berücksichtigt werden. Aus diesem Grund besitzt die abstrakte Klasse `ActivationShape` zwei instanziierbare Unterklassen. (siehe Abbildung 8.6) Die Unterklasse `StdActivationShape` beschreibt einen Aktivierungstensor des Formats BYXF, wie er von den meisten Layertypen als Eingang verwendet und für den Ausgang erzeugt wird. `ConvActivationShape` beschreibt ein spezielles Format, das nur an der Schnittstelle von einem Convolutional Layer zu einem Pooling Layer verwendet werden kann.

Beide Unterklassen implementieren mehrere in der Oberklasse definierte get-Accessoren. Damit lassen sich die Dimensionen des Tensors in die Richtungen B, Y, X und F, sowie der tatsächliche Speicherbedarf für alle enthaltenen Werte abfragen. Außerdem sind get-Accessoren enthalten, mit denen sich die Speicherposition eines Aktivierungswertes innerhalb des Tensors anhand seiner Koordinaten ermitteln lässt.

8.7.4 Gewichtsspeicherung

Wie in Abbildung 8.7 zu sehen ist, besitzt auch die abstrakte Klasse `WeightShape` zwei instanziierbare Unterklassen. Die Unterklasse `StdWeightShape` kommt für die Layertypen `FullyConnectedLayer` und `ConvolutionalLayer` zum Einsatz. Für diese Layertypen werden Gewichte in Form einer Gewichtsmatrix und eines Biasvektors gespeichert. In Objekten des Typs `StdWeightShape` werden die Dimensionen der Matrix und des Vektors, sowie deren Gesamtgröße gespeichert. Eine weitere Unterklasse ist `MaxPoolingWeightShape`, diese wird für den Schichttyp `MaxPoolingLayer` verwendet. Max Pooling an sich ist zwar eine nicht gewichtsbehaftete Operation, daher werden im für Gewichte vorgesehenen Feld auch keine Gewichte gespeichert. Allerdings müssen für jede Ausgabe des Max Poolings mehrere Eingaben verglichen werden, die im Speicher nicht direkt hintereinander liegen. Aus diesem Grund besitzt die Netzwerkstruktur des C-Programms zwei zusätzliches Array aus Integer-Werten: Im ersten Feld werden Offsets festgehalten, mit denen die Positionen der zu vergleichenden Eingabeaktivierungen bestimmt werden können. Nach der Bestimmung des Maximums wird dessen Positionen als Offset im zweiten Array gespeichert. So kann während der Backpropagation der Fehler des Ausgangs an den dafür verantwortlichen Eingang weitergereicht werden, ohne dass die Eingaben erneut verglichen werden müssen.

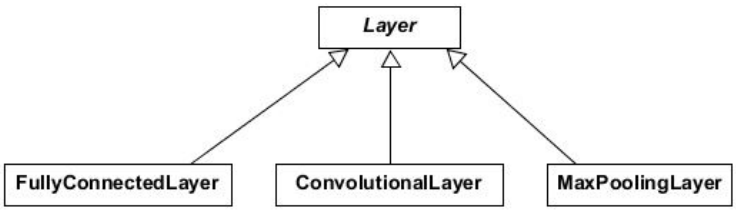


Abbildung 8.8: Unterklassen von Layer

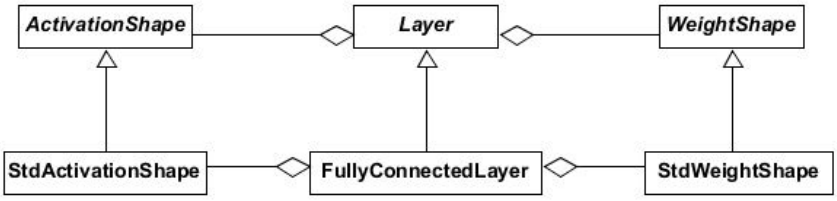


Abbildung 8.9: Layertyp Fully Connected Layer

8.7.5 Layertypen

Wie aus Abbildung 8.8 hervorgeht, wird in dieser Implementierung zwischen drei Layertypen unterschieden: `FullyConnectedLayer`, `ConvolutionalLayer` und `MaxPoolingLayer` . Alle drei Unterklassen benötigen eine andere Kombination von `WeightShape`- und `ActivationShape`-Subtypen. In den Abbildungen 8.9, 8.10 und 8.11 zeigen die Beziehungen der drei Layertypen zu den dazugehörigen `ActivationShape`- bzw. `WeightShape`-Unterklassen.

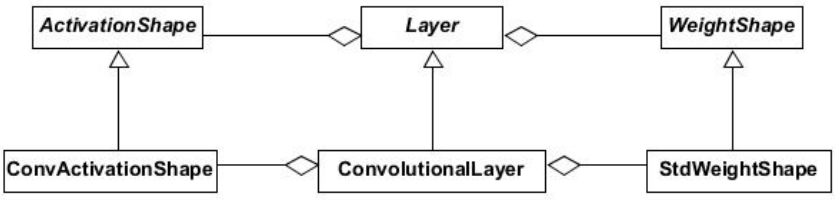


Abbildung 8.10: Convolutional Layer

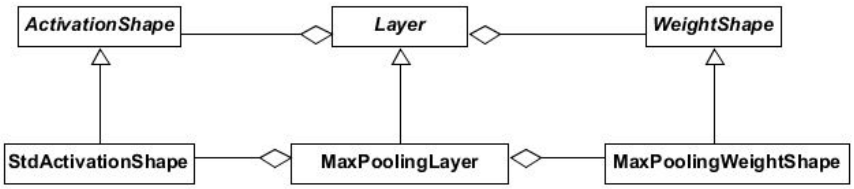


Abbildung 8.11: Max Pooling Layer

8.8 Programm zum Netzwerktraining

Das eigentliche Programm zum Training des Netzwerks wird in C geschrieben. Für einen großen Teil der Berechnungen werden die entsprechenden Funktionen der MKL-Bibliothek verwendet. Nebenläufig ausführbarer Anwendungscode wird mithilfe von OpenMP parallelisiert. Vom Netzwerkmodell oder der Konfiguration abhängiger Code wird mithilfe von cog generiert. Dieser Abschnitt befasst sich mit dem eigentlichen C-Programm, dessen Moduleinteilung und den Arbeitsweisen der wichtigsten Module.

8.8.1 Moduleinteilung

Das Diagramm in Abbildung 8.12 dient zur Veranschaulichung der Moduleinteilung und der Abhängigkeiten innerhalb des Programms.

8.8.2 Programmablauf

In der obersten Schicht ist die `main`-Funktion zu finden. Diese Funktion ist für die Initialisierung der `DataSupplier` und des Netzwerks verantwortlich und steuert den Ablauf des Trainings. Das Training an sich ist ein iterativer Prozess, in dem das Netzwerk abwechselnd mit einer konfigurierbaren Anzahl von Trainingsdatensätzen trainiert und mit einer ebenfalls konfigurierbaren Anzahl von Testdatensätzen getestet wird.

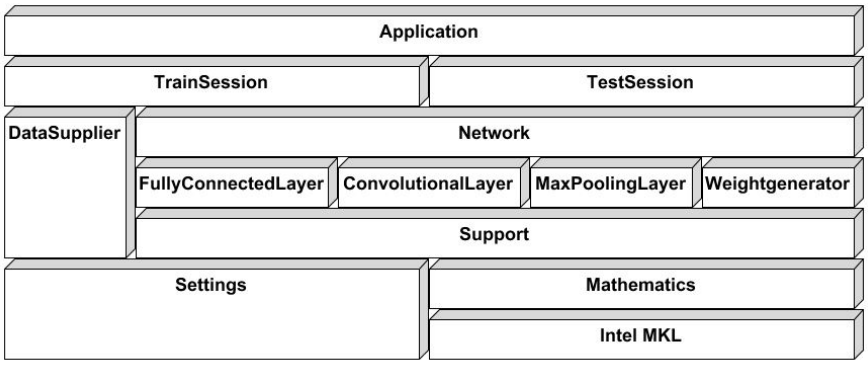


Abbildung 8.12: Moduleinteilung des Trainingsprogramms für CNNs

Für das Training und die Tests sind die Module `TrainSession` und `TestSession` verantwortlich. Diese Module definieren jeweils eine Methode, die den Ablauf zum Trainieren bzw. Testen des Netzwerks steuern. Beide Methoden benötigen jeweils einen Pointer auf eine Struktur des Typs `Network_t`, einen Pointer auf eine Struktur des Typs `DataSupply_t` und einen Integer mit der Anzahl der Minibatches, für die das Training / der Test durchgeführt werden soll. Für eine `TrainSession` muss darüber hinaus die zu verwendende Lernrate übergeben werden; für eine `TestSession` muss zusätzlich ein Pointer auf eine Struktur des Typs `TestResult_t` übergeben werden, in die die Testergebnisse geschrieben werden. In beiden Sessiontypen werden zunächst Pointer auf neue Eingabedaten von `DataSupply_t` angefordert, anschließend wird das Netzwerk mit diesen Daten vorwärts durchlaufen. Bei einer `TestSession` werden zum Schluss die durchschnittlichen Kosten pro Bild, sowie die Erkennungsgenauigkeit ermittelt und in die dafür vorgesehene Struktur geschrieben. Bei einer `TrainSession` wird auf die Ermittlung dieser Daten verzichtet, da sie für das eigentliche Training nicht erforderlich sind. Stattdessen wird die Ableitung der

Kosten nach den Aktivierungen der letzten Netzwerkschicht (die Fehler der Aktivierungen) berechnet und anschließend auf Basis dieser Werte eine Backpropagation und eine Gewichtsanpassung mithilfe des Stochastic Gradient Descent Algorithmus und der übergebenen Lernrate durchgeführt.

Die Einsparung der Kosten- und Genauigkeitsberechnung während des Trainings hat zur Folge, dass die zeitliche Entwicklung dieser Werte während des Trainings nicht ohne Weiteres mit der Vorlage aus Tensorflow verglichen werden kann. Die strikte Trennung von Training und Test hat zur Folge, dass durch eine entsprechende Konfiguration ein reines Training unter vollständigem Verzicht auf die Ermittlung dieser Daten mit höherer Effizienz durchgeführt werden kann. Für den Fall, dass der Verlauf dieser Daten während des Trainings verfolgt werden soll, ist der Test mit den dafür vorgesehenen Testdaten, die noch nicht zuvor zum Trainieren des Netzwerks verwendet wurden, etwas aussagekräftiger. Abhängig davon, wie häufig diese Daten ermittelt werden müssen und welche Aussagekraft erforderlich ist, können die Umfänge der `TrainSession` und `TestSession` unabhängig voneinander verändert werden.

8.8.3 Datenversorgung

Das Modul `DataSupply` definiert den Strukturtyp `DataSupplier_t`, sowie einige Funktionen zum Initialisieren und iterativen Durchlaufen der darin gespeicherten Ein- und Ausgabedaten. Zur Speicherung dieser Daten werden während der Initialisierung zwei zusammenhängende Speicherbereiche reserviert. Anschließend werden die Ein- und Ausgabedaten aus einem Satz von CSV-Dateien gelesen, die von der seriellen Implementierung übernommen werden, und getrennt in die beiden Speicherbereiche geschrieben. Zur der Vorwärtsberechnung des Netzwerks reicht

es aus, wenn der erste Layer als Eingabe einen Pointer verwendet, der in den entsprechenden Speicherbereich zeigt. Die Umstellung auf eine neue Minibatch erfolgt ebenfalls ohne großen Aufwand durch eine Aktualisierung des Pointers. Auch die Ausgabedaten können über einen Pointer ermittelt werden.

8.8.4 Netzwerk

Zur Repräsentation eines Netzwerks und dessen Zustand wird eine Struktur mit dem Namen `NeuronalNetwork_t` definiert. Diese Struktur beinhaltet Pointer auf gemeinsam genutzte Speicherbereiche. Dazu gehören Felder für die Aktivierungen und Gewichte, deren Fehler, ein Bereich für temporär genutzte Zwischenergebnisse, ein mit dem Wert 1.0 gefülltes Array, das für einige Rechenoperationen als Eingabe verwendet wird, und zwei Interger-Arrays zur Unterstützung der Poling-Operation. Darüber hinaus müssen die einzelnen Schichten des Netzwerks in dessen Struktur repräsentiert sein. Für jeden Layertyp existiert eine eigene Strukturdefinition, Instanzen dieser Strukturen müssen für die verwendeten Schichten in der Strukturdefinition von `NeuronalNetwork_t` angelegt werden. Der Code zum Anlegen dieser Instanzen wird unter Verwendung des Netzwerkmodells (Abschnitt 8.7) noch vor dem Übersetzen generiert.

Bei der Initialisierung des Netzwerks werden zunächst die in der Struktur `NeuronalNetwork_t` definierten Pointer initialisiert, indem je ein ausreichend großer Speicherbereich reserviert und die Anfangsadresse in den entsprechenden Pointer geschrieben wird. Anschließend erfolgt die Initialisierung der Gewichte mithilfe eines Pseudo Random Number Generators (PRGN) aus der MKL-Bibliothek. Darüber hinaus müssen auch alle in der Netzwerkstruktur enthaltenen Netzwerk-

schichten bzw. deren Strukturinstanzen initialisiert werden. Sowohl der Code zur Speicherreservierung, (Speicherbedarf ist abhängig vom Netzwerkmodell) als auch der Code zur Initialisierung der einzelnen Schichten (Durchlaufen der Layerliste, typabhängige Initialisierung) werden generiert.

Einer der in `NeuronalNetwork_t` definierten Speicherbereiche soll vorberechnete Informationen enthalten, die für die Poolingoperation verwendet werden. Wie bereits im Abschnitt 8.7.4 vorweggenommen wurde, handelt es sich dabei um ein Feld, in dem jedem Ausgang eines Poolinglayers alle dafür relevanten Eingangswerte zugeordnet werden. Die Berechnung dieser Zuordung wird ebenfalls während der Initialisierung vorgenommen, der Code dazu wird generiert. Dabei handelt es sich um eine sehr aufwändige Operation, die allerdings nur einmalig durchgeführt werden muss und mittels OpenMP-Pragmas nebenläufig ausgeführt werden kann. Eine Alternative wäre, die vollständige Zuordnung bereits im Zuge der Codegenerierung zu berechnen und als globales Array von Konstanten im Programmcode zu hinterlegen. Abhängig vom Netzwerkmodell und der Batchsize kann dies aber zu Problemen führen: In der Netzwerkkonfiguration des mit dieser Arbeit abgegebenen Quelltextes hätte das Array eine Größe von mehr als einer Million Elementen. Allein Generierung der dafür nötigen Arraydeklaration mithilfe eines seriellen Python-Skripts, das noch zur Laufzeit interpretiert werden muss, würde länger dauern als das eigentliche Training. (in einer früheren Version getestet; auf dem Versuchsrechner dauerte die Generierung länger als 30 Minuten bei einer Batchsize von 10) Zudem begrenzen einige Compiler die Dateigröße des zu übersetzenden Codes. Selbst wenn der Compiler auch derart große Dateien übersetzen kann, wird erneut viel Zeit zum Parsen des Quelltextes benötigt.

Nach Abschluss der Initialisierung wird die verwendete Instanz von `NeuronalNetwork_t` nicht mehr geändert. Der veränderliche Teil des Netzwerks, nämlich die Gewichte und Aktivierungen, sowie temporäre Daten, befindet sich innerhalb der Arrays, auf die von der Netzwerkstruktur aus verwiesen wird.

Neben der Initialisierungsfunktion definiert das Modul `Network` Funktionen zur Vorwärtsberechnung und Backpropagation, zur Berechnung der Genauigkeit und der Kostenfunktion, sowie zur Gewichtsanpassung mittels Stochastic Gradient Descent. Diese Funktionen sind relativ kurz, allerdings wird für jede dieser Funktionen generierter Code benötigt. Bei der Forward- und Backpropagation müssen die dafür vorgesehenen Funktionen der jeweiligen Schichten nach der Reihenfolge ihres Auftretens im Netzwerkmodell aufgerufen werden. Bei den Funktionen, die die Ergebnisse der Vorwärtsberechnung verarbeiten, muss die modellabhängige Größe des Ausgabevektors bekannt sein. Und für die Gewichtsanpassung muss die Anzahl aller Gewichte im Netzwerk bekannt sein.

8.8.5 Layertypen

Zwar sind die verschiedenen Layertypen in dieser Implementierung voneinander unabhängig und besitzen keine gemeinsame Basisklasse, trotzdem weisen sie gewisse Ähnlichkeiten zueinander auf. Zu jedem Layertyp wird eine Struktur definiert, diese enthält Informationen über dessen Eigenschaften, sowie Pointer zu den Startadressen des Ein- und Ausgabebereichs und Gewichte, sowie deren Fehler.

Außerdem definieren alle Layertypen je vier Funktionen, davon zwei zur Forward- und zwei zur Backpropagation, von denen wiederrum je eine für den Sonderfall vorgesehen ist, dass es sich um den ersten Layer im Netzwerk handelt und die Eingabe von einer als Parameter übergebenen Adresse gelesen werden muss.

Die Parametersignaturen dieser Funktionen sind für alle Layertypen identisch; dieser Umstand erleichtert die Codegenerierung im aufrufenden Code.

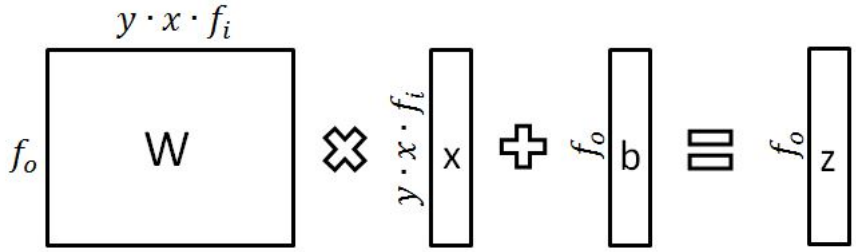


Abbildung 8.13: Bildung gewichteter Eingaben aus einem einzelnen Eingabevektor

8.8.6 Fully Connected Layer

Wie im vorherigen Abschnitt bereits angedeutet wurde, werden bei der Forward- und Backpropagation vom Netzwerkmodul die entsprechenden Funktionen nacheinander aufgerufen. Dies ist auch sinnvoll, da bei diesen Operationen jeder Layer Informationen benötigt, die bei der selben Operation im Vorgänger/Nachfolger berechnet werden. Nebenläufige Abschnitte müssen also innerhalb der entsprechenden Operationen der einzelnen Layer implementiert werden. Dabei besteht die Möglichkeit, über alle Neuronen innerhalb eines Layers zu parallelisieren/-vektorisieren. Wie in Abbildung 8.13 gezeigt wird, lässt sich die Gewichtung der Eingaben auch als Matrix-Vektor-Operation darstellen. Für jedes Neuron eines Fully Connected Layers müssen alle Aktivierungen der vorherigen Schicht auf mit je einem Gewicht multipliziert werden, anschließend wird ein Bias addiert. Stellt man die Eingangsaktivierungen, sowie die gewichteten Eingaben als Vektoren dar, so ergibt sich die symbolisch abgebildete Matrizengleichung zur Abbildung des Eingangs auf die Gewichteten Eingaben. Zusätzlich kann über die einzelnen Eingaben innerhalb einer Batch parallelisiert werden. Anstatt wie in Abbildung 8.13 die Zeilenvektoren, die jeweils ein Bild repräsentieren, einzeln abzubilden, werden sie stattdessen nebeneinander geschrieben. Dadurch ergibt sich sowohl für die Eingabe, als auch für die Ausgabe je eine Matrix, deren Breite der Batchsize entspricht. Wie bereits per Konvention festgelegt wurde, stehen die Elemente einer Batch

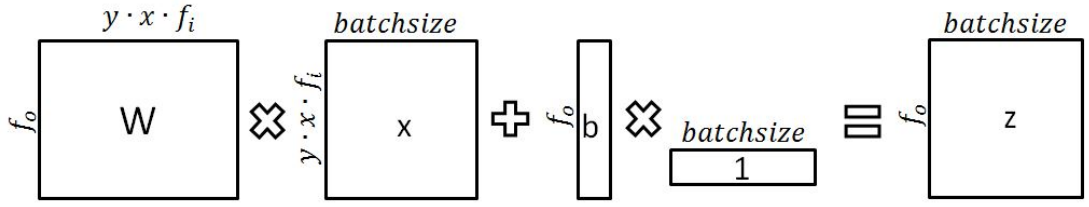


Abbildung 8.14: Bildung gewichteter Eingaben aus einer Batch von Eingabevektoren

normalerweise (immer vor und nach einem Fully Connected Layer) immer direkt hintereinander im Speicher. Daher genügt es, den Speicherbereich des Eingangs als Column-Major-Matrix zu betrachten. Die sich ergebende Abbildungsgleichung wird symbolisch in Abbildung 8.14 dargestellt. Zur Durchführung der darin beschriebenen Matrizenoperationen eignen sich die BLAS-Routinen aus der MKL-Bibliothek. Die Matrix der gewichteten Eingänge belegt ebenfalls einen zusammenhängenden Speicherbereich, die Reihenfolge der Werte entspricht bereits der gewünschten Anordnung der Aktivierungen im Ausgang. Zur Berechnung der Ausgaben muss die Aktivierungsfunktion für alle Werte dieses Speicherbereichs berechnet und an die entsprechende Position im Bereich der Ausgangsaktivierungen geschrieben werden. Für diesen Anwendungsbereich stellt Intel's MKL-Bibliothek die Vector Mathematical Functions zur Verfügung.

Wie die Forwardpropagation lässt sich auch die Backpropagation in einem Fully Connected Layer in Form von Matrixoperationen darstellen. Die Berechnung der partiellen Ableitung der Kosten nach den Gewichteten Eingaben erfolgt wie im Kapitel zur Backpropagation beschrieben. Diese partiellen Ableitungen (als Vektor dargestellt) auf die Fehler der Eingaben abgebildet werden, indem die transponierte Gewichtsmatrix von links an den Vektor multipliziert wird. Die Fehler der Bias-Werte entsprechen immer den Fehlern der gewichteten Eingaben, da die Ableitung der gewichteten Eingaben nach den entsprechenden Bias-Werten immer 1 ergibt.

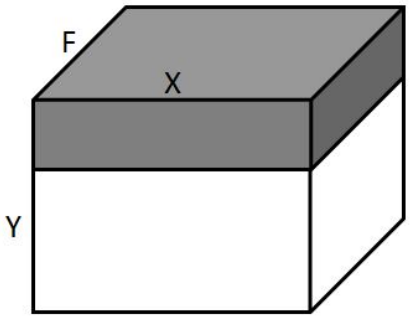


Abbildung 8.15: Räumliche Darstellung eines Filterkernels; zusammenhängender Bereich ist markiert

Zur Berechnung der Fehler in den Gewichten muss der transponierte Eingabevektor von rechts an den Vektor mit den Fehlern der gewichteten Eingaben multipliziert werden. Auch bei der Backpropagation können die Vektoren, die einzelne Eingaben repräsentieren, durch Matrizen ersetzt werden, die alle Werte einer Batch enthalten.

8.8.7 Convolutional Layer

Prinzipiell sind die Vorgänge in einem Convolutional Layer aus mathematischer Sicht denen in einem Fully Connected Layer nicht unähnlich. Im Gegensatz zum Fully Connected Layer wird nicht der ganze Layer gleichzeitig, sondern mehrfach je ein anderer Teil der Aktivierungen des Eingangs betrachtet. Dieser lässt sich aus mathematischer Sicht wie im vorherigen Abschnitt auch als Vektor darstellen und mithilfe einer Gewichtsmatrix und eines Biasvektors auf einen Vektor von gewichteten Aktivierungen abbilden, der dem Featurevektor für die aktuell betrachtete Position des Eingangs entspricht. Demzufolge lassen sich die mathematischen Vorgänge in einem Convolutional Layer auf selbige in einem Fully Connected Layer abbilden. Probleme ergeben sich allerdings bei der Implementierung. Wenn die Höhe des Filterkernels größer als 1 ist, dann liegen die darin enthaltenen Werte im

Speicher nicht mehr direkt hintereinander. Abbildung 8.15 stellt einen Filterkernel als Tensor dar. Farblich darin markiert ist ein Ausschnitt, dessen Werte im Speicher direkt nacheinander stehen. Die Zerlegung des Tensors in Schichten ist auf das für den Eingang eines Convolutional Layers verwendete Speicherschema zurückführen: Verschiedene Features für die gleiche Position werden direkt aufeinanderfolgend gespeichert; in X-Richtung benachbarte Featurevektoren stehen auch im Speicher hintereinander. Da der in Abbildung 8.15 dargestellte Tensor aber typischerweise nicht die gesamte Breite der Eingabe in X-Richtung abdeckt, gibt es Werte, die zwischen dem Ende der ersten Schicht und dem Anfang der zweiten Schicht stehen.

Wenn auf eine aufwändige Umverteilung der Eingabedaten in ein anderes Format verzichtet werden soll, muss das Programm jeden Eingabevektor in Bestandteile aufteilen, die den zusammenhängenden Schichten entsprechen, diese jeweils mit einem Teil der Gewichtsmatrix multiplizieren und anschließend die Ergebnisse aufaddieren. Dies in einzelne kleine Operationen aufzuteilen und MKL-Funktionen für diese kleinen Operationen aufzurufen würde sehr viel Overhead verursachen und eine Auslagerung an den Xeon Phi unwirtschaftlich machen. Stattdessen werden diese zu größeren Operationen zusammengefasst: Wie Abbildung 8.16 andeutet, lässt sich auch der vollständige Tensor mit allen Eingabeaktivierungen in zusammenhängende Speicherbereiche aufteilen, von denen jeder die Größe einer zuvor beschriebenen Schicht besitzt. In einem weiteren Schritt lässt sich ein derart eingeteilter Tensor auch als Column-Major-Matrix interpretieren, deren Höhe der Anzahl der Aktivierungen in einer Schicht entspricht. Diese Matrix lässt sich nun mit dem Teil der Gewichtsmatrix multiplizieren, der zur obersten Schicht gehört. Matrix-Matrix-Multiplikationen dieser Form müssen mehrfach mit unterschiedlichen Eingabedaten wiederholt werden. Bei näherer Betrachtung der Abbildung 8.16 zeigt sich, dass die Schichten sich nicht überschneiden, ihre Position also immer um die Breite einer Schicht verschiebt. Tatsächlich soll allerdings jede X-Position

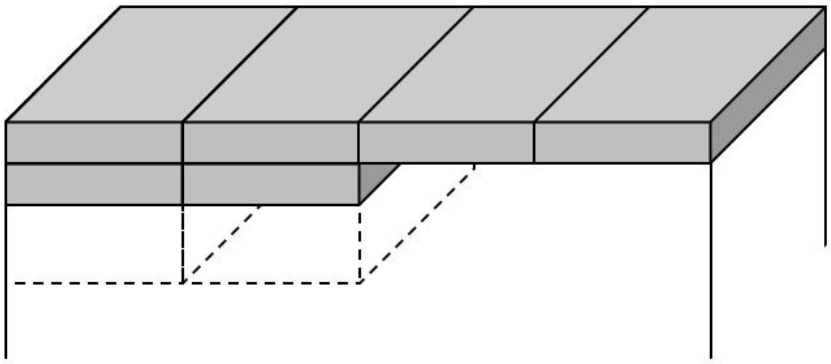


Abbildung 8.16: Zerlegung der Eingabe in zusammenhängende Bereiche

des Filterkernels über der Eingabe betrachtet werden können. Dies lässt sich erreichen, indem die oben erläuterte Matrixoperation mehrfach wiederholt wird, wobei der Start der Eingangsmatrix jeweils um die Länge eines Filtervektors verschoben wird. Darüber hinaus müssen auch für alle X-Positionen die restlichen Schichten des Filterkernels berücksichtigt werden. Dies lässt sich erreichen, indem die Startposition der Eingabematrix so weit verschoben wird, dass sie im Tensor "unter" der Startposition der ersten Schicht liegt. Außerdem muss ein anderer Teil der Gewichtsmatrix zur Abbildung verwendet werden. Die Ergebnisse können direkt auf den Ergebnisvektor der "darüberliegenden" Schicht addiert werden. Die Anzahl der x- bzw. y-Positionen, für die diese Matrix-Matrix-Multiplikation durchgeführt werden muss, entspricht der Größe des Filterkernels in den Richtungen x und y. Für die in diesem CNN verwendeten Filterkernel der Größe 5×5 werden also 25 Matrix-Matrix-Multiplikationen benötigt.

Auf diese Weise kann ein relativ hoher Parallelisierungsgrad erreicht werden, gleichzeitig entfällt ein aufwändiges Umsortieren der Eingangsdaten. Allerdings bringt diese Vorgehensweise einige Nachteile mit sich: An den Rändern der zu verarbeitenden Bilder werden unnötigerweise Positionen des Filterkernels betrachtet, die über das Bild hinausgehen. Darüber hinaus entspricht die Reihenfolge

der gewichteten Eingänge nicht der normalerweise für Aktivierungen geforderten Reihenfolge nach dem Schema BYXF. Die falsche Reihenfolge wird zur besseren Vektorisierbarkeit auch in der anschließenden Berechnung der Aktivierungsfunktion beibehalten.

Es wird davon ausgegangen, dass auf einen Convolutional Layer immer ein Max Pooling Layer folgt, der auch diese Reihenfolge von Aktivierungen in seinem Eingabebereich verarbeiten kann und eine Ausgabe nach dem Schema BYXF erzeugt.

8.8.8 Max Pooling Layer

Beim Max Pooling ergibt sich wie auch beim Convolutional Layer die Problematik, dass Aktivierungen aus einem Rechteckigen Bereich des Eingangs betrachtet werden, der im Speicher nicht zusammenhängend dargestellt wird. Die betrachteten Werte müssen miteinander verglichen werden, das Maximum jedes Bereichs muss im Ausgang übernommen werden und die Position der weitergegebenen Eingangsaktivierung muss bis zur Backpropagation zwischengespeichert werden. Hinzu kommt, dass die Poolingfilter gewöhnlich relativ klein sind. In dieser Arbeit werden Poolingfilter der Größe 2×2 verwendet, es müssen also 4 Werte miteinander verglichen werden. Durch die geringe Größe der Filter fällt auch eine mögliche Effizienzsteigerung durch Vektorisierung gering aus. Die MKL-Bibliothek stellt keine Routine bereit, mit der sich die Maxpoolingoperation effizient umsetzen lässt. Aus diesem Grund wird Anwendungscode erstellt, der die Poolingoperation ausführt. Wie auch bei den anderen Layertypen kann die Berechnung der Aktivierungen beim Max-Pooling für alle Perceptronen parallel berechnet werden, da die Perceptronen innerhalb eines Layers sich nicht gegenseitig beeinflussen. Folgende Schritte sind zur Berechnung der Aktivierung eines einzelnen Perceptrons notwendig:

Eingabepositionen ermitteln: Für jeden Ausgang gibt es mehrere Eingangsaktivierungen, die für den späteren Vergleich infrage kommen. Die Speicherpositionen, an denen diese zu finden sind, hängen nur vom Netzwerkmodell ab und bleiben während des Trainings unverändert. Daher werden diese Speicherpositionen schon während der Initialisierung des Netzwerks mithilfe eines generierten Codes einmalig berechnet und dann in einem Array gespeichert. Während der Initialisierung müssen die zu betrachtenden Positionen lediglich aus dem dafür vorgesehenen Array ausgelesen werden.

Eingangsaktivierungen vergleichen: Sind die zu betrachtenden Eingangsaktivierungen bekannt, müssen sie untereinander verglichen werden, sodass die Position und der Wert von deren Maximum bestimmt werden kann.

Ausgabe: Nachdem das Maximum ermittelt wurde, wird dessen Wert im Ausgang des Pooling Layers übernommen. Die Position wird in einem weiteren Feld zwischengespeichert und kann später zur Backpropagation verwendet werden.

Bei der Backpropagation eines Max Pooling Layers ist es erforderlich, dass der Fehler des Ausgangs zum Fehler des dafür verantwortlichen Eingangs kopiert wird. Für alle anderen Eingänge soll der Fehler den Wert 0 erhalten. Zu diesem Zweck wird zunächst der Speicherbereich aller Fehler im Eingang auf 0 initialisiert. Anschließend wird für jeden Ausgang des Pooling Layers die Position des maximalen Eingangs aus dem Zwischenspeicher ausgelesen und der Fehler des Ausgangs zur entsprechenden Speicherstelle kopiert.

8.8.9 Mathematische Funktionen

An mehreren Stellen des Programms wird auf mathematische Operationen zurückgegriffen, von denen nicht alle in der MKL-Bibliothek definiert sind. Einige Funktionen lassen sich allerdings auf Funktionen aus der MKL-Bibliothek zurückführen, so beispielsweise die zur Berechnung der Aktivierungen verwendete Sigmoidfunktion, deren Ableitung, oder Funktionen zur Berechnung der Genauigkeit und Abweichung. Implementiert werden diese Funktionen in einem eigenen Modul namens `mathematical`. Außerdem beinhaltet dieses Modul eine Datei mit Präprozessor-Makros, die während des kompilierens zu Funktionsaufrufen von MKL-Funktionen expandiert werden. Diese Makros werden zur Laufzeit anstatt der von der MKL-Bibliothek definierten Funktionsnamen verwendet. Auf die Laufzeit hat dies keinen Einfluss, da die Expandierung der Makros bereits während der Übersetzung durchgeführt wird. Diese Zwischenschicht vereinfacht spätere Programmänderungen oder Portierungen, da bei einer Ersetzung der MKL-Bibliothek durch vergleichbare Bibliotheken nur diese Zwischenschicht angepasst werden muss.

8.9 Anpassungen zur Steigerung der Genauigkeit

Nach der vollständigen Implementierung ist das Programm zwar prinzipiell in der Lage, ein neuronales Netz zur Klassifizierung handgeschriebener Ziffern zu trainieren, allerdings ist erfüllt die damit erreichbare Genauigkeit noch nicht die Erwartungen. Der Grund dafür liegt hauptsächlich in einer Vereinfachung des Netzaufbaus gegenüber der Beispielimplementierung auf Basis von Tensorflow: Die Beispielimplementierung verwendet zur Vermeidung von Overfitting eine Technik namens Dropout Regularization. Dabei wird eine zusätzliche Schicht in das Netzwerk eingebracht, bei der während des Trainings ein bestimmter Anteil der Aktivierungen des vorhergehenden Layers auf 0 gesetzt wird. Dies hat zur Folge, dass nachfolgende

Schichten darauf trainiert werden, ”stabilere Features” zu erkennen. Gleichzeitig werden mit den Aktivierungen auch deren Fehler auf 0 gesetzt, somit werden die vorhergehenden Layer in jedem Schritt nur teilweise trainiert. Der Einfachheit Halber wird sowohl in der seriellen Implementierung, als auch in den parallelen Implementierungen auf den Einsatz dieser Technik verzichtet, was auf Kosten der Erkennungsgenauigkeit bei der Evaluierung geht. Die nachfolgenden Schritte dienen dazu, mithilfe alternativer Regularisierungsverfahren die Erkennungsgenauigkeit für diese Implementierung zu steigern.

Alle in den folgenden Abschnitten vorgenommenen Anpassungen lassen sich durch Änderungen der Konfiguration und anschließende Codegenerierung ein- und ausschalten bzw. modifizieren. Daher können alle vorgestellten Testläufe bei Bedarf reproduziert werden. Da mit den Startgewichten zufällige Parameter in die Simulation mit einfließen, sind geringfügige Abweichungen von den nachfolgend genannten Ergebnissen möglich.

Die für die Testläufe verwendeten Konfigurationen erforderlichen config-Dateien, sowie die Programmausgaben und Messergebnisse (ermittelt mit `/usr/bin/time`) befinden sich im mit dieser Arbeit abgegebenen Quellcode im Verzeichnis `reports`.

8.9.1 Anpassung der Konfiguration an die Vorlage

Als Ausgangspunkt wird das Programm zunächst mit einer Konfiguration betrieben, die der Vorlage möglichst nahe kommt. Das Netzwerk wird insgesamt 2 048 000 mal trainiert, die Lernrate beträgt 0.1 %. Bei der anschließenden Evaluierung mithilfe der Testdaten ergibt sich eine finale Erkennungsgenauigkeit von 76,122 %. Bei einem zweiten Testlauf werden bereits während des Trainings Evaluierungen mit je einem Teil des Testdatensatzes (je 1024 Testbilder) durchgeführt, um so die zeitliche Entwicklung der Lernrate einschätzen zu können. Wie sich zeigt, wird diese Erkennungsgenauigkeit bereits nach weniger al 30 000 Trainingsdurchläufen

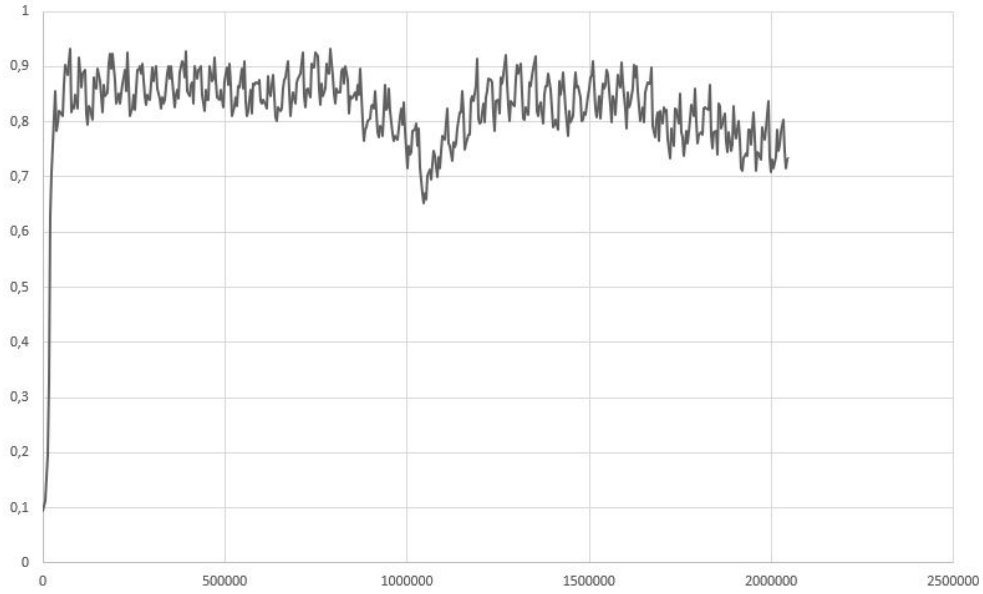


Abbildung 8.17: Verlauf der Erkennungsgenauigkeit während des Trainings mit fester Lernrate

erreicht und schwankt dann relativ stark. (vgl. Abbildung 8.17) Eine Ähnliche Entwicklung ist beim durchschnittlichen Ergebnis der Kostenfunktion pro Bild festzustellen. Allem Anschein nach findet das Netzwerk zwar ein lokales Minimum der Kosten, allerdings sind die Schritte, mit denen die Gewichte angepasst werden, zu groß für eine gute Annäherung der Gewichte auf diesen Arbeitspunkt.

8.9.2 Reduzierung der Lernrate

Es existieren viele Arbeiten, die sich mit der Wahl eines geeigneten Wertes für die Lernrate auseinandersetzen. Teilweise wird darin die Möglichkeit angesprochen, die Lernrate während des Trainings allmählich zu verringern. (vgl. [32]) Auf diese Weise wird zu Beginn eine schnelle Annäherung an ein Lokales Minimum ermöglicht. Durch die Verringerung der Lernrate wird es dem Netzwerk im weiteren Trainingsverlauf ermöglicht, sich dem optimalen Arbeitspunkt in kleineren Schritten anzunähern.

Die Main-Funktion sieht bereits vor, dass das Training regelmäßig durch kurze Evaluierungen mit einem Teil der Testdaten unterbrochen werden kann. Die Funktion wird so angepasst, dass die in während der Evaluierung festgestellten Kosten pro Bild mit dem entsprechenden Wert der letzten Evaluierung verglichen werden. Sollte sich das Ergebnis verschlechtert haben, wird die Erkennungsrate mit einem konstanten Faktor zwischen 0 und 1 multipliziert, d.h. sie nimmt bei konstanter Häufigkeit von "overshooting"-Ereignissen exponentiell ab.

Für einen erneuten Testlauf des Programms mit dynamisch angepasster Lernrate müssen zunächst die Parameter der Lernrate im Initialzustand und der Faktor zu deren Reduzierung festgelegt werden. Nach einigen experimentellen Testläufen wird der Startwert auf 0,0045 und der Reduzierungsfaktor auf 0,992 festgelegt. Ein erneuter Testlauf ergibt nach der Bearbeitung von 2 048 000 Trainingsbildern eine finale Erkennungsgenauigkeit von 81,61 %. Eine Auswertung des zeitlichen Verlaufs ergibt, dass die Genauigkeit deutlich später beginnt, signifikant zu steigen. Außerdem verläuft der Anstieg deutlich langsamer als zuvor mit einer festen Lernrate. Der in Abbildung 8.18 dargestellte Verlauf der Genauigkeit lässt vermuten, dass diese wohl auch nach dem Abbruch des Trainings noch weiter gestiegen wäre. Offensichtlich lässt sich die funktionale Entwicklung der Lernrate weiter verbessern, um bereits nach kürzerem Training eine höhere Genauigkeit zu erzielen. Eine exakte Feineinstellung der Parameter ist allerdings nicht Teil dieser Arbeit.

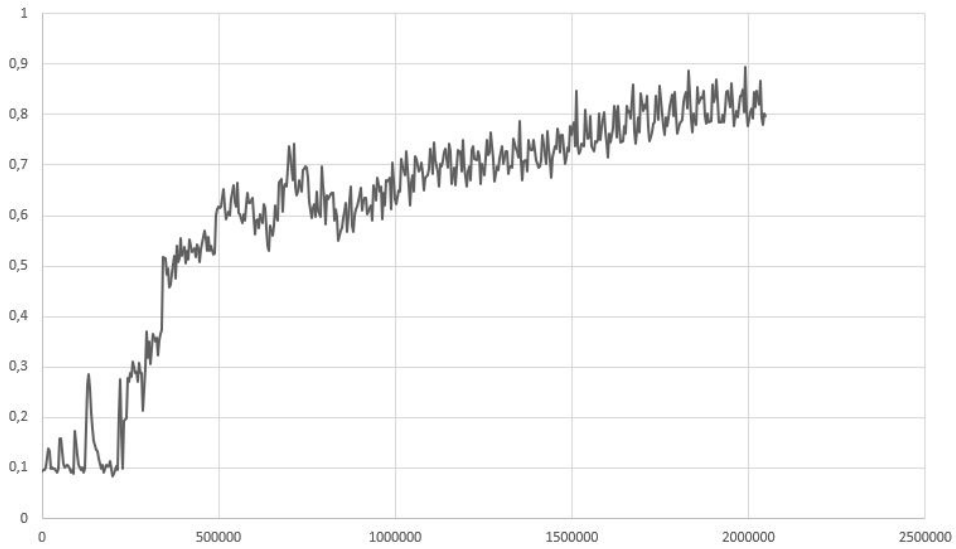


Abbildung 8.18: Verlauf der Erkennungsgenauigkeit während des Trainings mit abnehmender Lernrate

8.9.3 Fehlerreduktion beim Convolution Layer

Wie sich mit den Testergebnissen des vorhergehenden Abschnitts herausstellt, verhindert die anfangs zu hohe Lernrate eine Verbesserung der Erkennungsrate des Netzwerks. Es lässt sich annehmen, dass die hohe Lernrate ein ”overshooting” verursacht, also eine zu starke Anpassung der Gewichte, die über den angestrebten Arbeitspunkt hinausgeht. Weiterhin kann untersucht werden, welche Teile des Netzwerks vom Overshooting am meisten betroffen sind, sodass Maßnahmen zu dessen Reduzierung lokal auf diese Teile begrenzt werden können.

Innerhalb eines Fully Connected Layer werden alle Gewichte für jedes im Training durchlaufene Bild genau einmal trainiert. Bei einem Convolutional Layer werden dagegen innerhalb eines Bildes dieselben Gewichte auf mehrere Bildbereiche angewendet. Während der Backpropagation werden die Fehler der Gewichte für alle Bereiche addiert. Es kann davon ausgegangen werden, dass sich durch diese

Summierung teilweise höhere Fehlerbeträge ergeben, als beim Training mit nur einem Bildbereich der Fall wäre. Die so errechneten Fehler werden bei der Fehleranpassung mit der Lernrate skaliert und von den Gewichten abgezogen. Die größte Änderung findet somit tendenziell eher bei den Gewichten der Convolutional Layer statt, daher ist hier auch die Gefahr von Overshooting am größten.

Um diesem Effekt entgegenzuwirken, ist es sinnvoll, die Convolutional Layer mit einer niedrigeren Learning Rate als den Rest des Netzwerks zu trainieren. Wie viel kleiner die Learning Rate sein sollte, lässt sich nur schwer bestimmen, allerdings liefert die Anzahl der möglichen Positionen des Filterkernels über der Eingabe einen guten Anhaltspunkt. Die Anzahl der verschiedenen Positionen entspricht der Anzahl der Trainingsvorgänge innerhalb eines Bildes. Tatsächlich ergeben sich nicht für alle Positionen Fehler, die von Null verschieden sind. Dies ist beispielsweise der Fall, wenn in einem darauffolgenden Max Pooling Layer der Fehler einer Aktivierung auf 0 gesetzt wird, weil sich die Aktivierung nicht auf das Ergebnis des Netzwerks auswirkt. Außerdem kann angenommen werden, dass die berechneten Fehler für die einzelnen Positionen unterschiedlich gerichtet sind, was den Betrag des Gesamtfehlers reduzieren kann. Andererseits ist es sogar erwünscht, dass sich die vorderen Layer des Netzwerks langsamer anpassen, sodass die hinteren Layer sich auf die veränderten Zwischenergebnisse bei gleichen Eingaben einstellen können.

Basierend auf diesen Überlegungen wird das Programm angepasst, sodass aus logischer Sicht die Lernrate der Convolutional Layer durch die Anzahl der möglichen Filterpositionen geteilt wird. In der Implementierung werden die Fehler bereits während der Backpropagation mit dem Kehrwert dieses Faktors multipliziert, sodass bei der Gewichtsanpassung mit einer für alle Gewichte gleichwertigen Lernrate gerechnet werden kann.

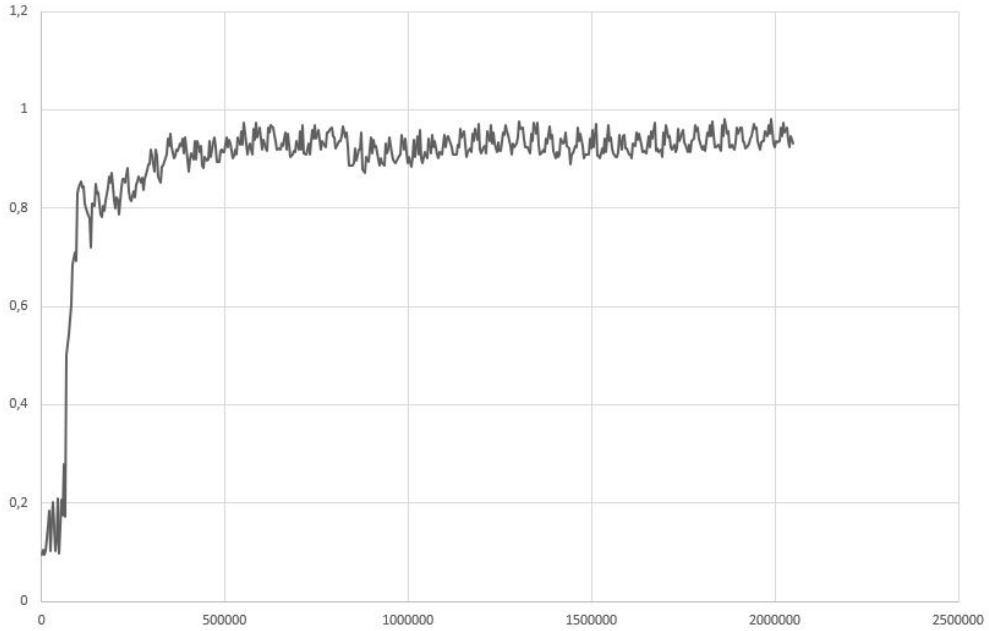


Abbildung 8.19: Verlauf der Erkennungsgenauigkeit während des Trainings mit abnehmender Lernrate und zusätzlich verringerter Lernraten der Convolutional Layer

Ein Testlauf mit dieser Anpassung (Lernrate und Reduzierungsfaktor wie im vorherigen Abschnitt) ergibt eine finale Erkennungsrate von 94,802 %. Eine Auswertung der zeitlichen Entwicklung (vgl. Abbildung 8.19) zeigt, dass die Genauigkeit sehr früh zu steigen beginnt und auch deutlich schneller ansteigt als im vorherigen Durchlauf. Nach etwa 500 000 verarbeiteten Bildern pendelt sich die Erkennungsrate knapp unter 95 % ein, wobei die Schwankungen deutlich geringer sind als beim weiter oben beschriebenen Testlauf mit statischer Lernrate. Die Laufzeit des Programms auf dem verwendeten Server beträgt für diesen Testlauf etwas mehr als 18 Minuten. Damit ist es geringfügig schneller als das Beispielprogramm auf

Basis von Tensorflow, für das auf dem gleichen System etwas mehr als 20 Minuten benötigt werden. Die erreichte Genauigkeit von 94,802 % liegt allerdings immer noch deutlich unterhalb der 98 %, die mit dem Beispielprogramm unter Verwendung der Dropout Regularization erreicht werden können.

9 Implementierung auf einer Nvidia GPU mit CUDA

Die letzte in dieser Studienarbeit betrachtete Hardwarearchitektur ist die Grafikkarte (GPU). Moderne GPUs werden nicht nur zur Berechnung der auf dem Monitor gezeigten Inhalte genutzt, sondern können aufgrund ihrer Architektur auch für komplexe mathematische bzw. numerische Berechnungen genutzt werden. Die Vorzüge der Grafikkarte als Hilfsmittel zur Implementierung eines CNN und allgemein für aufwändige Rechenoperationen soll Abschnitt 9.1 dieses Kapitels erläutert werden. Außerdem soll die Programmierung der Rechenoperationen auf GPUs am Beispiel von NVIDIAs CUDA-Umgebung gezeigt werden. Die Abschnitte 9.2 Verwendete Hardware: GTX Titan X und 9.3 Umsetzung und Parallelisierung zeigen dann die in dieser Arbeit genutzte Hardware sowie den verfolgten Implementierungsansatz für ein Convolutional Neural Network. Leider konnte die Implementierung eines funktionsfähigen und lernenden CNNs auf CUDA nicht in der vorgegebenen Zeit beendet werden. Das entstandene System ist nicht in der Lage, die Muster des MNIST-Datensatzes korrekt zu klassifizieren. Der Fehler muss dabei in der Implementierung liegen, da das verwendete Netz in anderen Implementierungen ihren Zweck erfüllt und gute Ergebnisse in der Klassifizierung erzielt.

9.1 Grafikkarten zur Berechnung

Der ursprüngliche Zweck von Grafikkarten war die Shader-Berechnung und damit waren GPUs ein dediziertes Werkzeug zur Berechnung der auf dem Bildschirm dargestellten Inhalte. Da aufwändige grafische Darstellungen auch viele Berechnungen für die steigende Zahl an Pixel bedeuten, haben sich Grafikkarten zu hochparallelen Recheneinheiten entwickelt. Ihre Möglichkeiten sind dabei im Vergleich zu general-purpose CPUs eingeschränkt, jedoch sind sie auf komplexe Matrixoperationen im *floating point* -Format ausgelegt. Dabei besitzen sie eine Vielzahl an Rechenkernen, womit es möglich wird, eine große Masse an mathematischen Operationen parallel zu bearbeiten.

Dieser Abschnitt beschäftigt sich mit der Programmierung von mathematischen Anwendungen auf Grafikkarten, im speziellen mit der CUDA-Programmierung und weist im Punkt 9.1.4 auf eine hilfreiche, von NVIDIA bereitgestellte Bibliothek hin.

9.1.1 Stärken und Schwächen der Grafikkarte

Wie bereits erwähnt, sind moderne Grafikkarten in der Lage komplexe mathematische Operationen äußerst effizient durchzuführen, da sie über eine hohe Zahl an Recheneinheiten verfügen. Aktuelle Hochleistungs-Grafikkarte besitzen mehrere Tausend Rechenwerke. Im Fall von NVIDIAs CUDA werden diese *CUDA-Cores* genannt. Der offensichtliche Vorteil der Verwendung von Grafikkarten für aufwändige Berechnungen liegt in der gegebenen hohen Parallelität (vgl. [33]). GPUs verfügen dabei aber nicht über den Instruktionsumfang gängiger CPUs. Generell kann festgestellt werden, das GPUs nicht für komplex verschachtelte Programmabläufe geeignet sind. Sie sind auf Rechenoperationen spezialisiert und eignen sich nur bedingt für andere Programmabläufe. Speziell besitzen Grafikkarten nur bedingte Möglichkeiten zur *branch prediction* (Verzweigungsvorhersage).

Außerdem benötigen GPUs immer eine CPU, die Rechenoperationen oder Programme auf der Grafikkarte startet und diese kontrolliert und koordiniert. Ein weiterer entscheidender Nachteil von modernen leistungsstarken Grafikkarten ist ihr hoher Preis. Im Vergleich zu hochwertigen CPUs sind Grafikkarten sehr teuer. Der Einsatz von GPUs für mathematische Berechnungen ist daher nur sinnvoll, wenn eine hohe Zahl an Rechenoperationen notwendig ist oder eine große Datenmenge im Sinne von Berechnungen verarbeitet werden muss. Das Problem muss sich dabei auch geeignet parallelisieren lassen, da sonst der Vorteil der Grafikkarte erlischt.

Der nächste Abschnitt 9.1.2 soll die allgemeine Architektur einer modernen NVIDIA CUDA-fähigen Grafikkarte darstellen und damit den hier erwähnten Vorteil der Parallelität unterstreichen.

9.1.2 Architektur moderner Grafikkarten

Prinzipiell sind die Architekturen der aktuellen NVIDIA Grafikkarten gleich aufgebaut. Eine Grafikkarte besteht dabei aus *Streaming Multiprocessors (SM)*, den CUDA-Cores, einem dedizierten Graphikspeicher mit entsprechenden Caches und Speicher-Controllern.

Abbildung 9.1 zeigt die erwähnten Komponenten in ihrem Zusammenspiel am Beispiel der Pascal-Architektur von NVIDIA. Diese findet beispielsweise in Grafikkarten vom Typ GTX 1080 Ti Anwendung. Von oben beginnend, ist in Abbildung 9.1 die PCIe-Schnittstelle zum Host-System gezeigt. Das Host-System bildet die bestimmende CPU. Darunter folgt die GigaThread-Engine. Weiter ist die GPU in GPCs (Graphic Processing Clusters) unterteilt. Diese besitzen je eine Raster Engine und greifen auf den selben L2-Cache zu (vgl. [34]). Die GPCs sind weiter unterteilt in je zehn SMs. Die Streaming Multiprocessors teilen die Arbeit auf in dieser Architektur 128 CUDA-Cores auf. Jede SM besitzt dabei eine separaten L1-Cache.

Abbildung 9.1: Grafikkartenarchitektur am Beispiel der Pascal-Architektur von NVIDIA ([34])

Abbildung 9.2: Aufbau eines Streaming Multiprocessors in der Pascal-Architektur ([34])

In Abbildung 9.2 ist eine detaillierte Darstellung einer SM gegeben (vgl. [34]). Ein Streaming Multiprocessor besteht dabei aus vier Blöcken, auf die sich die 128 CUDA-Cores pro SM aufteilen. In der Abbildung sind zur besseren Übersicht weniger Cores aufgetragen. Jeder dieser Blöcke besitzt einen dedizierten *Instruction Buffer*, einen *Warp Scheduler* und ein sogenanntes *Register File*. Darin sind 16384 32-Bit Register abgebildet.

Ein Warp ist in CUDA eine Einheit von 32 Threads. Die CUDA-Struktur bildet alle erzeugten Threads in solche Warps ab und verteilt diese entsprechend innerhalb eines SM. Außerdem besitzt jede SM zwei L1-Caches und acht *Texture Units*. Des weiteren besitzt jeder Streaming Multiprocessor einen dedizierten Shared-Memory-Bereich.

Nachdem nun in diesem Abschnitt auf die grundlegende Architektur von NVIDIAs Grafikkarten am Beispiel der Pascal-Architektur eingegangen wurde, soll im Abschnitt 9.1.3 auf einige Aspekte der CUDA-Bibliothek und damit der Programmierung von Anwendungen auf CUDA-fähigen Grafikkarten eingegangen werden. Die als Zielhardware für diesen Projektteil verwendete NVIDIA GTX Titan X Grafikkarte ist mit der sogenannten Maxwell Architektur ausgestattet. Diese wird in Abschnitt 9.2 näher erläutert.

9.1.3 CUDA Programmierung

Die CUDA-Programmierung basiert auf der Programmierung von sogenannten *Kernels*. Ein Kernel stellt dabei eine Routine dar, welche auf einer GPU ausgeführt werden kann.

```
__global__ void minimal_kernel(float* a, float* b, float* c,

{
        for(int i = 0; i < size; i++)
```

```
        {
            c[i] = a[i] + b[i];
        }
    }
```

Listing 9.1: Minimaler Kernel

Listing 9.1 zeigt einen simplen CUDA-Kernel. Dieses Minimalbeispiel nimmt drei Zeiger auf `floats` und eine Ganzzahl als Parameter. Er addiert dabei einfach elementweise die Inhalte der Arrays `a` und `b` und speichert das Ergebnis im Array `c`. Im momentanen Zustand wird die `for` -Schleife seriell auf der GPU ausgeführt. Zur Parallelisierung später in diesem Abschnitt mehr. Wichtig ist hier der Specifier `__global__`. Er definiert einen CUDA-Kernel. Solche Kernels können nun vom Host aus auf der Grafikkarte gestartet werden (vgl. [35]). Der Host ist in jedem Fall die CPU und der darauf laufende Teil des Programmcodes. Mit dem Specifier `__device__` können Subroutinen auf der GPU deklariert werden. Solche Funktionen dürfen nur von Kernels auf der GPU aufgerufen werden.
Es ist zu beachten, dass die übergebenen Arrays `a`, `b` und `c` auf im Speicher der GPU liegen müssen. Die Grafikkarte hat keinen direkten Zugriff auf den RAM des Systems. Es muss also immer ein Transfer der Daten vom Host über die PCIe-Schnittstelle des Systems auf das Device (Grafikkarte) und zurück erfolgen. Dazu bietet die CUDA- Library spezielle Funktionen an.

Bei der CUDA- Library handelt es sich um eine C/C++ -Bibliothek. Diese stellt verschiedene Funktionen zur Kommunikation und Kontrolle der GPU vom Host aus zur Verfügung. Wichtige Funktionen stellen dabei die Methoden `cudaMalloc(..)`, `cudaFree(..)` und `cudaMemcpy(..)` dar. Damit wird der Zugriff auf den Speicher der Grafikkarte vom Host aus geregelt.

```
float *A_host, *B_host, *C_host;
```

```
float *A_device, *B_device, *C_device;

A_host = malloc(ARRAY_SIZE_D*sizeof(float));
B_host = malloc(ARRAY_SIZE_D*sizeof(float));
C_host = malloc(ARRAY_SIZE_D*sizeof(float));

fillArrays(A_host, B_host, ARRAY_SIZE_D); /*
    Hilfsroutine, um den Arrays Werte zuzuweisen */

cudaStatus_t cuda_state;

cuda_state = cudaMalloc((void**) &A_device,
    ARRAY_SIZE_D*sizeof(float));
cuda_state = cudaMalloc((void**) &B_device,
    ARRAY_SIZE_D*sizeof(float));
cuda_state = cudaMalloc((void**) &C_device,
    ARRAY_SIZE_D*sizeof(float));

cuda_state = cudaMemcpy((void*) A_device, (void*)
    A_host, ARRAY_SIZE_D*sizeof(float),
    cudaMemcpyHostToDevice);
cuda_state = cudaMemcpy((void*) B_device, (void*)
    B_host, ARRAY_SIZE_D*sizeof(float),
    cudaMemcpyHostToDevice);

minimal_kernel<<<1,1>>>(A_device, B_device,
    C_device);
```

```
cuda_state = cudaMemcpy((void*) C_host, (void*)
    C_device, ARRAY_SIZE_D*sizeof(float),
    cudaMemcpyDeviceToHost);

cuda_state = cudaFree((void*) A_device);
cuda_state = cudaFree((void*) B_device);
cuda_state = cudaFree((void*) C_device);
```

Listing 9.2: Aufruf des minimalen Kernels mit Speicherzugriff vom Host

In Listing 9.2 ist ein Beispiel zur Nutzung des in Listing 9.1 beschriebenen Kernels zu sehen. Es werden zuerst host-seitig Pointer zu Speicheradressen auf dem Host und auf der Grafikkarte angelegt. Mit Hilfe der bekannten C Standardfunktion `malloc(..)` werden Arrays von Gleitkommazahlen auf dem Host reserviert. In einer zu definierenden Hilfsfunktion werden diese Speicherbereiche mit sinnvollen Werten beschrieben. Danach wird Speicher auf der GPU reserviert. Dazu wird die Funktion `cudaMalloc(..)` aus der CUDA-Library verwendet. Diese weist dem bereits angelegten Pointer auf dem Host eine Speicheradresse auf der Grafikkarte zu. Als Parameter fordert diese Funktion den besagten Zeiger im Format `void**` und die Größe des benötigten Speicherbereichs in Byte. Das `void**` -Format macht sowohl einen Cast als auch eine erneute Referenzierung des Pointers notwendig.
Der Rückgabewert aller hier genannten Funktionen der CUDA-Bibliothek ist vom Typ `cudaStatus_t`. Dieser informiert über den Erfolg des Funktionsaufruf. Er ist besonders hilfreich zu Debug-Zwecken. Beispielsweise werden auch fehlerhafte Aufrufe im Sinne der Übergabe von Speicheradressen auf dem Host in diesem Rückgabewert vermerkt.

Nachdem der Speicherplatz für die Arrays nun auch auf der GPU reserviert wurde, müssen nun die Werte der auf dem Host bereits belegten Arrays auf die Grafikkarte übertragen werden. Dazu steht die Funktion `cudaMemcpy(..)` zur Verfügung. Diese regelt den Zugriff auf den Speicherbereich der GPU über die PCIe-Schnittstelle. Die Methode benötigt vier Paramater. Sie fordert zwei Pointer als Ziel- und Quelladresse in genannter Reihenfolge, sowie die Anzahl der zu übertragenen Bytes. Bis zu diesem Zeitpunkt stimmt die Syntax mit der aus der C-Standardbibliothek bekannten `memcpy(..)` -Methode überein. Der vierte Paramater der CUDA-Methode legt nun die Übertragungsrichtung fest. Er bestimmt, ob vom Host zur Grafikkarte (`cudaMemcpyHostToDevice`), von der GPU zum Host (`cudaMemcpyDeviceToHost`) oder von Speicheradressen auf der Grafikkarte zu anderen dort befindlichen Speicherbereichen (`cudaMemcpyDeviceToDevice`).

Anschließend folgt der Kernel-Aufruf. Hier taucht die spezielle CUDA-Syntax zum Kernel-Aufruf auf. In CUDA müssen Kernel beim Aufruf zwei zusätzliche Parameter in drei spitzen Klammern (`«<1,1»>`) übergeben werden. Diese bestimmen die Parallelisierung des Kernels, in dem Sie die Anzahl der vom Kernel verwendeten Threads festlegen.
Threads sind dabei in CUDA in Blöcken organisiert. Der zweite angegebene Paramter in den spitzen Klammern legt dabei die Anzahl der vom Kernel zu startenden Threads pro Block fest. Der erste Parameter bestimmt die Anzahl der zu verwendenden Blöcke. Diese Blöcke sind dabei in einem sogenannten Grid organisiert (vgl. [35]).

In Abbildung 9.3 ist ein exemplarischer Aufbau eines solchen Thread-Grids gezeigt. Darin sind je 256 Threads pro Block und 4096 Blöcke im Grid abgebildet. Der zugehörige Aufruf gestaltet sich wie in Listing 9.3 gezeigt.

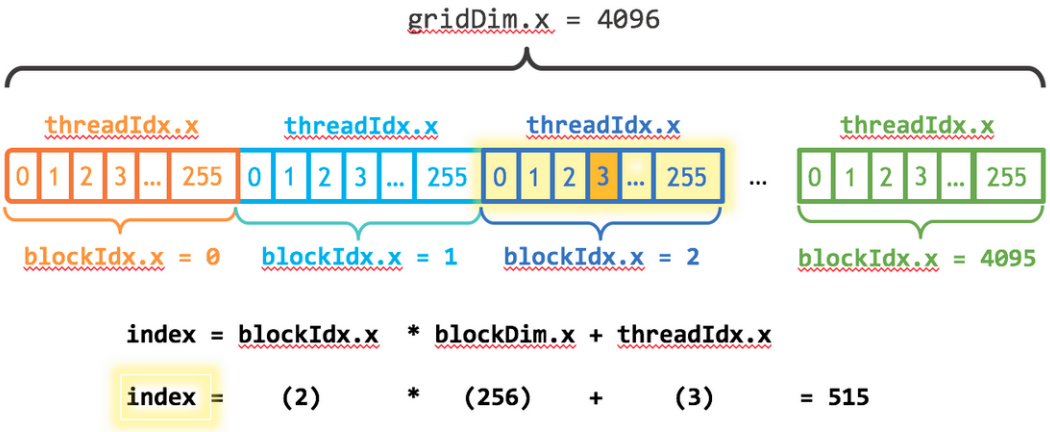


Abbildung 9.3: Exemplarische Darstellung des Aufbaus eines Thread-Grids ([35])

```
minimal_kernel<<<4096,256>>>(A_device, B_device,
    C_device);
```

Listing 9.3: Aufruf des minimalen Kernels mit 4096 Blöcken à 256 Threads

In dem dort dargestellten Beispiel würde mit der in Listing 9.1 gezeigten Implementierung des `minimal_kernel(..)` jeder der erzeugten Threads die gleichen Berechnungen redundant ausführen. Bis zu diesem Punkt ist keine konkrete Parallelisierung vorgenommen worden. Um nun die Berechnungen unter den Threads zu verteilen kann eine Modifizierung des Kernels nach Listing 9.4 vorgenommen werden.

```
__global__ void minimal_kernel(float *a, float *b,
    float *c, int size)
{
        int index = blockIdx.x * blockDim.x +
            threadIdx.x;
        int stride = blockDim.x * gridDim.x;
```

```
        for(int i = index; i < size; i+=stride)
        {
                c[i] = a[i] + b[i];
        }
}
```

Listing 9.4: Parallelisierte Variante des Kernels

Die hier verwendete Methode der Parallelisierung wird *grid-stride-loop* genannt (vgl. [36]). Damit wird die Berechnung flexibel auf die angegebenen Thread-Parameter angepasst. Jeder Thread berechnet hierbei seinen Index nach der Formel `index = blockIdx.x * blockDim.x + threadIdx.x;`. Die CUDA-Library stellt dazu die Strukturen `blockIdx`, `blockDim` und `threadIdx` zur Verfügung. Darin sind die jeweiligen Thread-Parameter des Aufrufs gespeichert. In der `for` -Schleife wird die Laufvariable dann mit dem berechneten Index initialisiert. Die Sprungweite der Laufvariable wird durch die Variable `stride` bestimmt. Diese stellt die Anzahl aller verwendeten Threads in allen Blöcken dar. Sie ist damit das Produkt aus Anzahl der Blöcke und Anzahl der Threads pro Block. Durch die Nutzung der *grid-stride-loop* wird nun jeder Wert des Ausgangsarrays `c` nur in einem Thread berechnet. Jeder Thread schreitet in der Schleife das Array mit der Anzahl aller Threads als Sprungweite ab, jeweils versetzt um seine Position im Grid.

Damit sind nun die Grundlagen der CUDA-Programmierung und der damit möglichen Parallelisierung behandelt. In Abschnitt 9.3 wird nachfolgend beschrieben, wie diese Mechanismen auf das vorliegende Problem der Parallelisierung eines CNNs angewendet wurden.

9.1.4 cuBLAS und cuRand

Die CUDA-Umgebung bietet neben der zwingend notwendigen CUDA-Runtime-Library weitere Bibliotheken zur Unterstützung an. Beispielsweise werden Bibliotheken für numerische Berechnungsmethoden oder auch die Implementierung von Deep Neural Networks angeboten. In diesem Projekt wurden zwei dieser Libraries verwendet, die cuBLAS-Bibliothek und die cuRand-Library in der Kernel-Version.

Die cuBLAS-Bibliothek stellt eine auf NVIDIA GPUs spezialisierte Implementierung der BLAS-Funktionen zur Verfügung. Daraus wurde das Skalarprodukt für Vektoren benutzt. Die cuRand-Library bietet verschiedene Methoden zur Erzeugung von Zufallszahlen mit GPU-Unterstützung an. Sie existiert sowohl als Host-Version, als auch als für Kernel geeignete Version. Sie wurde verwendet, um in einem Kernel die Initialwerte des Gewichte und Biases des Neuronalen Netzes zufällig zu bestimmen.

9.2 Verwendete Hardware: GTX Titan X

Die Implementierung und Parallelisierung des Convolutional Neural Networks in dieser Arbeit ist auf die Verwendung auf Systemen mit einer NVIDIA Geforce GTX Titan X ausgelegt. Diese stand auf einem Server der Dualen Hochschule Baden-Württemberg Stuttgart zur Verfügung. Sie ist mit 3072 CUDA-Cores ausgestattet. Dies erlaubt einen äußert hohen Grad der Parallelisierung. Außerdem verfügt die GTX Titan X über 12 GB dedizierten VRAM. Dadurch ist es Möglich, große Datenmenge ohne häufige externe Speicherzugriffe zu verarbeiten.

Die GTX Titan X ist dabei wie bereits in Abschnitt 9.1.2 Architektur moderner Grafikkarten erwähnt nach der Maxwell -Architektur aufgebaut. Auch diese Architektur stützt sich auf *Streaming Multiprocessors*, welche hier SMM genannt werden. Die GTX Titan X verfügt dabei über 24 solcher SMMs mit je 128 CUDA-Cores. Auch hier teilen sich die SMMs die gleichen Komponenten, wie bereits in Abschnitt 9.1.2 anhand der Pascal-Architektur erläutert wurde. Der wesentliche Unterschied liegt im Aufbau er SMM.

Diese Aufbau ist in Abbildung 9.4 dargestellt. Jede SMM verfügt über vier Warp-Scheduler. Damit ist es möglich, vier Warps pro SMM konkurrierend zu betreiben. Jeder Warp-Scheduler kontrolliert dabei 32 CUDA-Cores. Die L1-Caches unterscheiden sich nicht von der, der Pascal-Architektur. Auch die Register-Files sind unverändert, wodurch jedem Warp-Scheduler bzw. den darin organisierten CUDA-Cores 16384 32-Bit Register zur Verfügung stehen. Eine weitere Neuerung ist die Verlegung der *PolyMorph Engine 3.0* direkt in die SMM. In der Pascal -Architektur befand sich diese noch außerhalb der SM.

Es wurden nun alle relevanten Grundlagen der Programmierung auf CUDA sowie der Aufbau und die Struktur der verwendeten Hardware erläutert. Der nächste Abschnitt widmet sich der Implementierung des in dieser Arbeit vorgestellten Convolutional Neural Network unter Zuhilfenahme einer NVIDIA Geforce GTX Titan X Grafikkarte. Es werden der gewählte Ansatz zur Parallelisierung der Berechnungen, ein Ansatz zur Umsetzung der Faltung und die daraus resultierenden Änderungen in den weiteren Schichten des Netzes dargestellt.

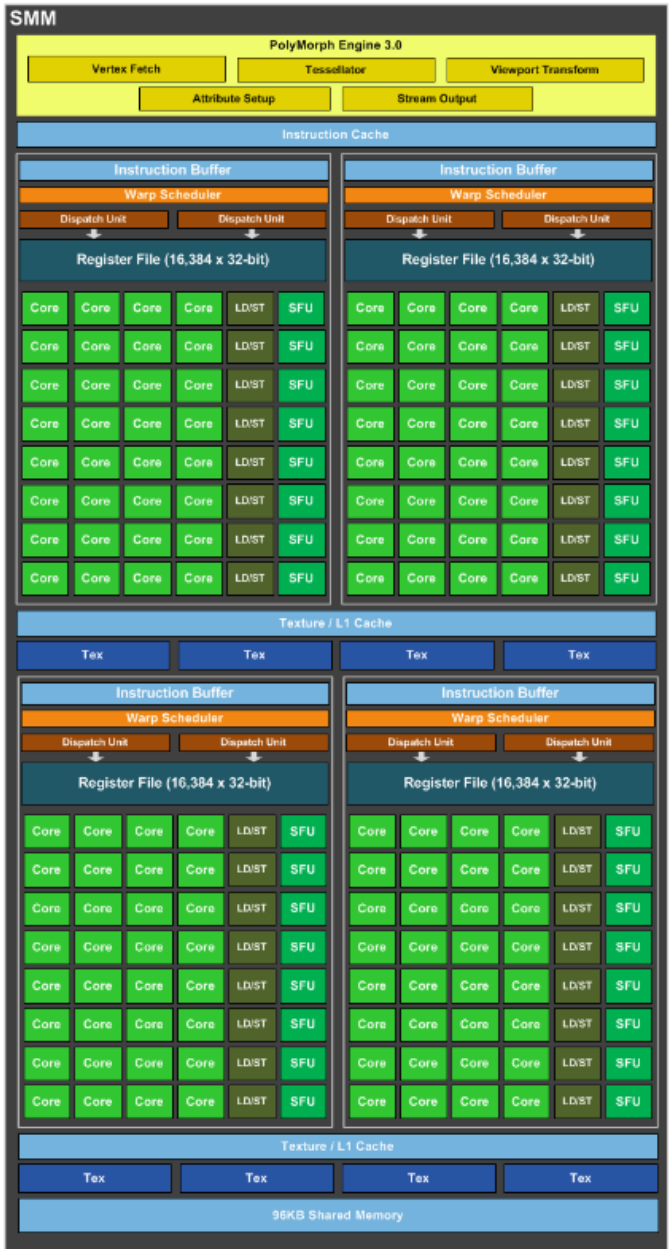


Abbildung 9.4: Aufbau einer Maxwell SMM [37]

9.3 Umsetzung und Parallelisierung

Die Stärke der Grafikkarte liegt in der effizienten Berechnung von mathematischen Operationen auf Matrizen. Folgend dieser Prämisse wurde bei der Umsetzung ein Ansatz gesucht, welcher die Berechnung der einzelnen Schichten durch geschickte Matrixoperationen abbilden kann. Es wurden im Vorfeld der Implementierung verschiedene Ansätze betrachtet. Darunter der *Winograd Small Convolution* - Algorithmus erweitert auf große Matrizen (vgl. [38]) und eine auf der Fast-Fourier-Transformation basierende Methode (vgl. [39]). Die Wahl fiel dabei aufgrund von Recherchen von bereits bestehenden Vergleichen (vgl. [40]) auf die Methode der *Unrolling based Convolution*. Damit ist es möglich, die Faltungsoperation eines Convolutional Neural Networks als einfache Matrix-Matrix-Multiplikation darzustellen. Diese Methode wird in Abschnitt 9.3.1 auf der nächsten Seite näher erläutert.

Die Unrolling-based Convolution hat nicht nur Einfluss auf die Convolutional Layer des Netzes, sonder auch auf die Pooling Layer und den Input Layer. Das in dieser Arbeit verwendete bereits in anderen Anwendungen erprobte Netz besteht aus einem Input Layer, zwei Convolutional Layern mit Faltungskernen der Größe 5×5, zwei Pooling Layern mit 2×2-Bereichen, welche die Größe des Layers jeweils halbieren, einer Fully Connected Schicht mit 1024 Nodes und einer finalen Fully Connected Schicht mit 10 Nodes. In der ersten Convolutional Schicht werden 32 Feature Maps erzeugt. In der zweiten werden 64 Feature Maps erzeugt. Eine Darstellung des verwendeten CNNs ist in Abbildung 9.5 auf der nächsten Seite abgebildet. Darin sind bereits die bei der Unrolling-based Convolution -Methode entstehenden Matrizen schematisch dargestellt. Die Abbildung zeigt in der unteren Hälfte die verwendeten Node-Matrizen, die in den jeweiligen Layern notwendig sind. Die obere Hälfte der Graphik enthält die Gewichtsmatrizen, welche zwischen den entsprechenden Layern angeordnet sind. Bei Leserichtung von links nach rechts sind

hier Matrix-Matrix-Multiplikationen der Layer mit den jeweiligen Gewichtsmatrizen zu interpretieren. Abschnitt 9.3.1 soll nun die gewählte Methode der Unrolling-based Convolution erläutern und damit Klarheit in die Darstellung der Abbildung 9.5 bringen.

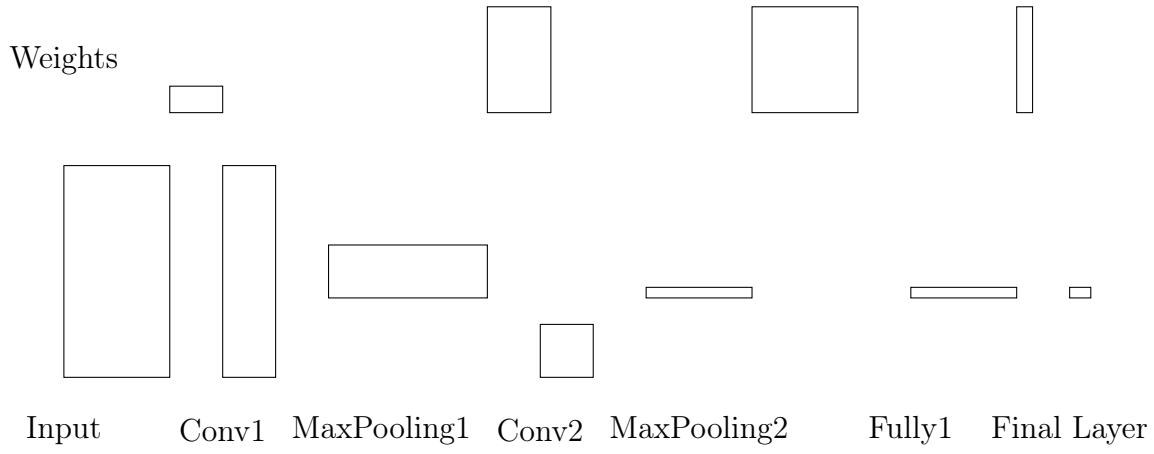


Abbildung 9.5: Darstellung der genutzten Matrizen

9.3.1 Unrolling-based Convolution

Unrolling-based Convolution für Convolutional Neural Networks wurde 2006 von Mitarbeitern der Firma Microsoft vorgestellt. Dabei handelt es sich um eine Methode um die Faltungsoperation in Convolutional Neural Networks in eine einfach zu realisierende Matrix-Matrix-Multiplikation zu überführen (vgl. [40]). Dieser Ansatz verfolgt grundlegend die Idee, die Faltungsoperation zu entfalten bzw. auszurollen. Im ersten Schritt muss der Input des betreffenden Layers umsortiert und einige Werte vervielfacht werden. Abbildung 9.6 auf der nächsten Seite zeigt die Operation am Beispiel von drei 3×3 -Input-Matrizen, sechs 2×2 -Faltungskernen und zwei Output -Features.

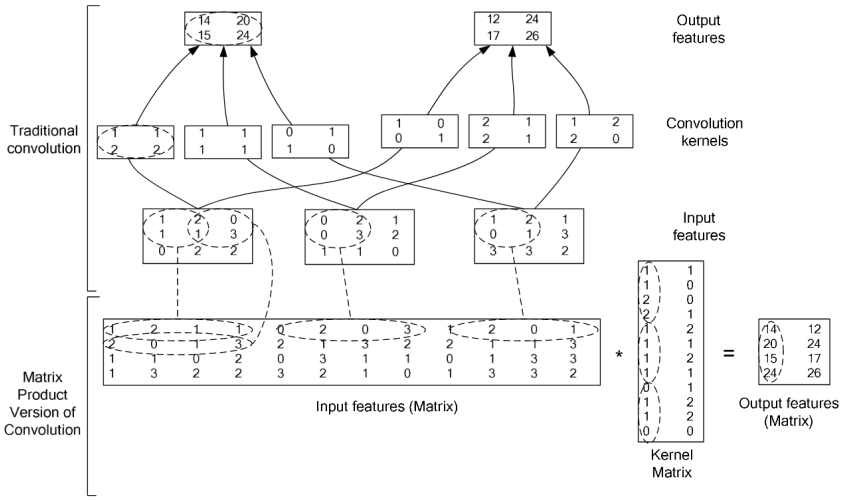


Abbildung 9.6: Exemplarische Darstellung der Unrolling-based Convolution (vgl. [40])

Das hier gezeigte Schaubild ist in zwei Bereiche unterteilt. Im oberen Teil ist die traditionelle Faltungsoperation abgebildet. Man sieht hier drei Input-Features der Größe 3×3. Darüber sind sechs Gewichtsmatrizen der Faltungskerne mit den Ausmaßen 2×2 gezeigt. Die Faltungsoperation führt dann zu zwei Output-Features der Größe 2×2.
Im unteren Teil der Graphik ist die zugehörige Unrolling-based Convolution abgebildet. Die drei Input-Features werden in eine Matrix umsortiert. Hierbei ist zu beachten, dass einige Werte der ursprünglichen Matrizen mehrfach auftreten. In dieser Abbildung ist die Zuordnung der den Faltungskernen entsprechenden Bereichen der ursprünglichen Matrizen in die neue ausgerollte Matrix anschaulich dargestellt. Auch die Faltungskerne selbst müssen in eine neue größere Matrix umsortiert werden. Dabei werden deren Gewichte pro Output-Feature in eine Spalte einer entstehenden Kernel-Matrix einsortiert. Auch dieser Schritt ist in Abbildung 9.6 überischtlich dargestellt. Wird nun die entstandene Input-Matrix mit der entstandenen Kernel-Matrix multipliziert, befinden sich in den Spalten der

daraus entstehenden Output-Matrix die Werte der Output-Features. Die Output-Matrix kann nun in einem nächsten Schritt für eine weitere Convolutional Schicht erneut nach dem nun bekannten Schema umsortiert werden. Damit ist durch diesen Ansatz jede Schicht durch eine einfache Matrix-Matrix-Multiplikation darstellbar. Die Umsortierung der Eingangswerte in den jeweiligen Schichten ist dabei nach [40] signifikant weniger aufwändig, als die klassische Faltungsoperation.

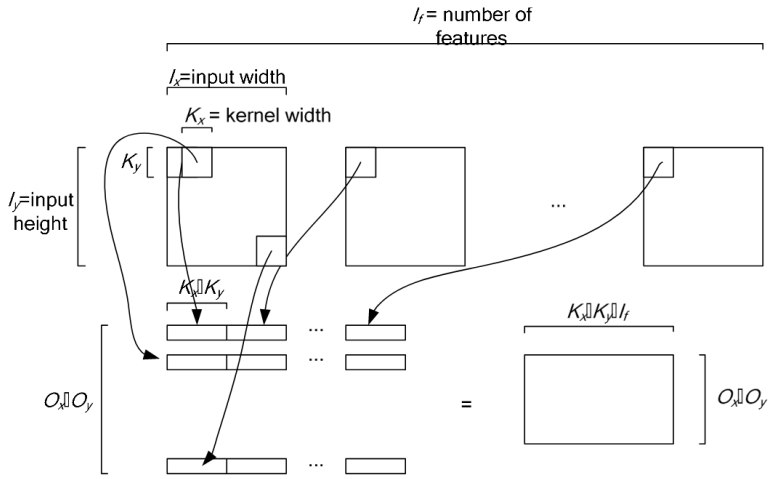


Abbildung 9.7: Darstellung des Vorgangs der Umsortierung bei Unrolling-based Convolution (vgl. [40])

Um die Größen der durch Unrolling-based Convolution entstehenden Matrizen zu bestimmen, kann das in Abbildung 9.7 gezeigte Schaubild zur anschaulichen Herleitung entsprechender Formeln dienen. Es zeigt den Vorgang der Umsortierung des Inputs eines Convolutional Layers. Dieser Input besteht aus I_f Feature Maps, deren Matrixrepräsentationen jeweils I_y Zeilen und I_x Spalten aufweisen. Außerdem sind in den Input-Feature-Maps die Faltungskerne an verschiedenen Positionen aufgetragen. Diese Faltungskerne besitzen die Abmessungen $K_y \times K_x$. Vollzieht man nun die Operation der Umsortierung erhält man die Darstellung der unteren Hälfte der Abbildung. Die den Faltungskernen zugeordneten Werte werden pro

Feature-Map jeweils in $K_x \cdot K_y$ Spalten der ausgerollten Matrix angeordnet. Es entsteht also eine Matrix mit $K_x \cdot K_y \cdot I_f$ Spalten. Die Anzahl der Zeilen dieser Matrix wird durch die Größe eines regulären Output-Features bestimmt. Diese bestimmt sich mit Schrittweite 1 durch Gleichung (9.1).

$$\begin{aligned} O_x &= I_x - K_x + 1 \\ O_y &= I_y - K_y + 1 \end{aligned} \tag{9.1}$$

Die umsortierte Matrix hat damit $O_x \cdot O_y$ Zeilen. Es ist also eine $(K_x \cdot K_y \cdot I_f) \times (O_x \cdot O_y)$ -Matrix entstanden.

Nun müssen auch die Gewichte entsprechend in einer angepassten Matrix angeordnet werden. Dabei müssen in einer Spalte dieser Matrix die den Faltungskernen in den jeweiligen Input-Features zugeordneten Gewichte verortet sein. Jede Spalte ist dabei einem Output-Feature zugeordnet.

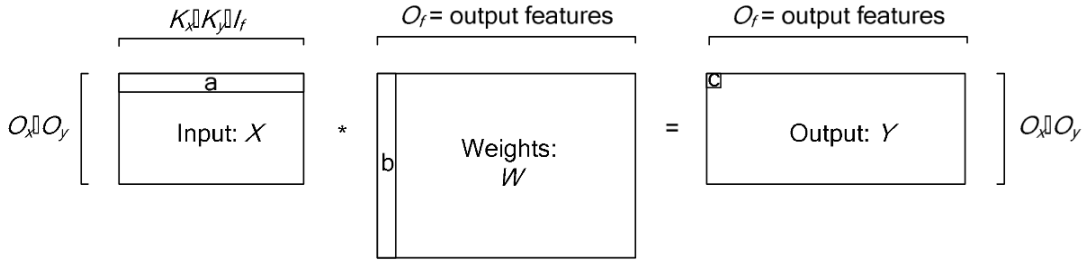


Abbildung 9.8: Veranschaulichung der sich ergebenden Matrixgrößen bei Unrolling-based Convolution (vgl. [40])

Abbildung 9.8 zeigt den Vorgang der resultierenden Matrix-Matrix-Multiplikation. Darin sind zusätzlich die Dimensionen der jeweiligen Matrizen aufgetragen. Das Ergebnis der Multiplikation liefert eine Output-Matrix mit $O_x \cdot O_y$ Zeilen und O_f Spalten. O_f bezeichnet dabei die Anzahl der Output-Features. Darin sind wie

bereits erwähnt, die jeweiligen entstehenden Feature-Maps durch die Spalten der Matrix repräsentiert. Das Skalarprodukt einer Input-Zeile a mit einer Spalte b der Gewichtsmatrix ergibt dabei ein Output-Node c, repräsentiert durch einen Wert in der Output-Matrix.

Nachdem nun die Methode der Unrolling-based Convolution betrachtet wurde, sollen in den nächsten Abschnitten die Auswirkungen dieses Ansatzes auf die verschiedenen Layer eines Convolutional Neural Networks am Beispiel des in dieser Arbeit betrachteten Netzes dargestellt werden.

9.3.2 Input-Layer

Entgegen den anderen in dieser Arbeit gezeigten Implementierungen ist mit dem gewählten Ansatz der Unrolling-based Convolution ein dedizierter Input-Layer notwendig. Dieser Layer stellt in der vorliegenden Implementierung die im Sinne der Unrolling-based Convolution ausgerollte Matrix des Eingangsbildes dar.

Dabei wird das 28×28 Pixel große Input-Muster in dem hier betrachteten Netz derart umsortiert, dass eine 576×25 -Matrix entsteht. Die Matrixdimensionen ergeben sich aus der Definition des nachfolgenden Convolutional Layers. Dieser besteht aus 5×5 -Faltungskernen, was mit einer Schrittweite von 1 zu einem $24 \times 24 = 576$ -Pixel großen Output führt. Dies findet natürlich 32 mal für jede in diesem Layer auftretende Feature Map statt. Die Anzahl der Zeilen im Input-Layer ergibt sich also durch die Größe einer Feature Map des folgenden Convolutional Layers. Die Schwierigkeit in diesem Layer lag in einem geeignet Algorithmus zur Zuordnung der Ausgangswerte in die neue ausgerollte Matrix. Listing 9.5 zeigt den in der Implementierung genutzten Algorithmus.

```
int size =  576; /*24x24*/

for(int n = index; n < size; n+=stride)
{
        int i = n/24; /*rows in original row-major picture
           */
        int j = n%24; /*columns in original picture*/

        for(int k = 0; k < 5; k++) /*convolutional kernel
           rows in original picture*/
        {
                for(int l = 0; l < 5; l++) /*convolutional
                   kernel columns in original picture*/
                {
                        arrayPtr[(k*5+l)*576+i*24+j] =
                           picturePtr[(i+k)*24+j+l];
                }
        }
}
```

Listing 9.5: Umsortierung im Input-Layer

Darin wird jeder Wert des Ausgabebildes in der Convolutional Schicht der klassischen Variante der Faltung in der äußersten Schleife abgeschritten. Anschließend die Position in diesem Originalbild in Form von Zeilen i und Spalten j bestimmt. Die beiden inneren Schleifen schreiten nun eine Zeile der neuen umsortierten Matrix ab, wobei diese befüllt mit den entsprechenden Werten des Eingabebildes befüllt. Auch hier ist eine *Grid-Stride-Loop* umgesetzt.

9.3.3 Convolutional Layer

In den Convolutional Layern des CNN mit Unrolling-based Convolution entstehen wie bereits in Abschnitt 9.3.1 auf Seite 152 detailliert beschrieben Matrizen mit x Zeilen und y Spalten, wobei x durch die Größe einer Feature Map und y durch die Anzahl der Feature Maps bestimmt wird. Damit ist die für die erste Convolutional Schicht des betrachteten Convolutional Neural Network Node-Matrix von der Größe 576×32. Der zweite Convolutional Layer benötigt eine 64×64 Node-Matrix. Hier wird eine durch den vorhergehenden ersten MaxPooling Layer entstandene 12×12 -Matrix mit einem Faltungskern der Dimension 5×5 gefaltet. Diese Faltung wird auf 64 Feature Maps angewendet.

In dieser Implementierung wurden die Gewichtsmatrizen den jeweils durch Multiplikation entstehenden Layern zugeordnet. Das bedeutet, die erste auftretende Gewichtsmatrix ist die, des ersten Convolutional Layers. Diese hat die Abmessungen 25×32. Hier sind die Gewichtswerte der Faltungskerne für jede Feature Map in den Spalten der Matrix repräsentiert. Die Gewichtsmatrix des zweiten Convolutional Layers ist von der Größe 800×64, wobei die Anzahl der Spalten erneut durch die Zahl der Feature Maps des Convolutional Layers gegeben ist. Die Anzahl der Zeilen bestimmt sich nach

$$K_x \cdot K_y \cdot I_f = 5 \cdot 5 \cdot 32 \tag{9.2}$$

und wird somit durch den vorhergehenden MaxPooling Layer bestimmt. Die in den Convolutional Layern durchgeführten Operationen beschränken sich dank der Methode der Unrolling-based Convolution auf einfache Matrix-Matrix-Multiplikationen. Der für diese Schicht vorgesehene CUDA-Kernel implementiert damit nur diese Matrix-Matrix -Multiplikation. Die notwendigen Umsortierungen werden bereits in den vorhergehenden Layern durchgeführt. Natürlich müssen hier auch die entsprechenden Biases addiert und die Aktivierungsfunktion angewendet werden.

9.3.4 MaxPooling-Layer

In den MaxPooling Layern des Convolutional Neural Network wird die Größe der vorhergehenden Convolutional Schichten durch sogenanntes *Subsampling* halbiert. Die Layer ermitteln jeweils das Maximum eines 2×2 -Bereichs. Dieser Wert wird weiter verwendet, der Rest wird verworfen. Damit ergeben sich im ersten MaxPooling Layer des hier gezeigten Neuronalen Netzes 32 Feature Maps mit einer Größe von je 12×12 Bildpunkten. In der zweiten MaxPooling Schicht befinden sich analog 64 Feature Maps mit einer Abmessung von 4×4.

In den MaxPooling Schichten des hier implementierten Netzes findet zusätzlich zum Subsampling, falls nötig, auch die Umsortierung für die nachfolgende Schicht statt. Dies ist der Fall, wenn auf die MaxPooling Schicht eine weitere Convolutional Schicht folgt. In dem hier gezeigten Beispiel betrifft das nur den ersten MaxPooling Layer. Die Node-Matrix dieses Layers hat damit die Ausmaße 64×800. Die Anzahl der Zeilen wird erneut durch die Größe der Feature Maps des nachfolgenden Convolutional Layers bestimmt. Die Anzahl der Spalten berechnet sich wie bereits gezeigt nach Gleichung (9.2) auf der vorherigen Seite.

Auch hierbei war die größte Schwierigkeit die Umsortierung der Matrix. Bei der Implementierung des MaxPooling Layers wird zuerst eine Hilfsmatrix erzeugt, welche die durch das Subsampling entstehende Matrix ohne Umsortierung darstellt. Diese enthält die entsprechenden Werte aller Feature Maps. Nach der Durchführung des Subsampling werden die Werte der Hilfsmatrix entsprechend in die eigentliche MaxPooling-Matrix einsortiert, sodass in der nachfolgenden Convolutional Schicht wieder nurmehr eine Matrix- Matrix -Multiplikation durchgeführt werden muss, um den Output dieser Schicht zu bestimmen. Für den Fall eines nachfolgenden Fully Connected Layers muss keine zusätzliche Umsortierung vorgenommen werden. Da

hier alle nach dem Subsampling verbleibenden Werte in Form eines Vektors auf die Nodes der Fully Connected Schicht abgebildet werden, kann der Output des Subsamplings ohne Modifikation verwendet werden. Die Werte liegen bereits derart im Speicher, dass sie als Vektor aufgefasst und verwendet werden können.

9.3.5 Fully Connected Layer

In dem in dieser Arbeit verwendeten Netz befinden sich am Ende des Neuronalen Netzes zwei Fully Connected Layer. Der erste hat eine Größe von 1024 Nodes. Der zweite und finale Layer besteht aus 10 Outputnodes, welche die Klassifizierung der Inputs darstellen. Die gewählte Methode der Unrolling-based Convolution hat keinen Einfluss auf das Verhalten bzw. die Implementierung dieser Layer. In den Fully Connected Schichten werden die Input-Vektoren mit den Gewichtsmatrizen multipliziert, wodurch erneut ein Ausgangsvektor entsteht. Hierauf müssen offensichtlich noch die Biases addiert und die Aktivierungsfunktion angewendet werden.

Die Gewichtsmatrix, welche dem ersten Fully Connected Layer zugeordnet ist, hat dementsprechend eine Größe von 1024×1024, um die 1024 Werte des vorhergehenden MaxPooling Layers auf die 1024 Werte des Vektors der Fully Connected Schicht abzubilden. Dem Final Layer ist somit eine 1024×10 -Gewichtsmatrix zugeordnet.

Auch die CUDA-Kernel, welche den Fully Connected Layern zugeordnet sind, beschränken sich auf Matrix -Matrix -Multiplikationen und das addieren der Biases sowie die Anwendung der Aktivierungsfunktion.

9.3.6 Parallelisierung

Durch die hohe Zahl an Rechenwerken, die der verwendeten Grafikkarte zur Verfügung stehen, ist ein hoher Grad der Parallelisierung möglich. Um die Leistungsfähigkeit der NVIDIA GTX Titan X möglichst gut ausnutzen zu können, muss das Ziel aller Überlegungen zur Parallelisierung sein, die nötigen Rechenoperation bestmöglich zu verteilen und eine große Zahl an Threads zu erzeugen, welche dann auf die CUDA-Cores verteilt werden können.

Grundsätzlich sind mehrere Ansätze zur Parallelisierung eines Convolutional Neural Networks denkbar. Eine Möglichkeit stellt eine Muster-Parallelität dar. Dabei werden die notwendigen Berechnungen für mehrere MNIST-Bilder gleichzeitig auf der Grafikkarte durchgeführt. Hier ist es beispielsweise möglich, ein ganzes Batch parallel zu berechnen. Denkt man aber daran, dass der GTX Titan X 3072 CUDA-Cores zur Verfügung stehen, wird schnell klar, dass diese Methode nicht zielführend ist. Um die Grafikkarte entsprechend auszulasten, wäre eine große Batch-Size nötig, was den Lernprozess negativ beeinflussen würde.

Ein anderer Ansatz wird durch den Modell-Parallelismus spezifiziert. Hierbei wird versucht die Rechenschritte innerhalb des Modells, also des Neuronalen Netzes, zu parallelisieren. Auch dabei sind mehrere Varianten denkbar. Beispielsweise könnten die Berechnungen in den einzelnen Schichten für jede Node parallel durchgeführt werden. Diese Methode würde offensichtlich einen hohen Grad der Parallelisierung erlauben. Implementiert man die Faltungsoperation im klassischen Sinn, ist diese Möglichkeit sicher in Betracht zu ziehen. Eine weitere denkbare Parallelisierungsart ist die Parallelisierung über Spalten der jeweils erzeugten Ausgangsmatrizen. Aufgrund der hier gewählten Implementierungsweise der Unrolling-based Convolution wurde in dieser Arbeit eben diese Art der Parallelisierung gewählt. Jedem Thread

wird dabei die Aufgabe zugeordnet, für jeweils eine Spalte der Gewichtsmatrix das Skalarprodukt mit allen Zeilen der Input-Matrix durchzuführen. Auch die Addition der Biases und die Anwendung der Aktivierungsfunktion ist diesem Task zugeordnet.

Die Parallelisierung wurde dabei mit Fokus auf die Convolutional Layer umgesetzt. Es werden in jedem Layer jeweils 64 Threads gestartet. Dies entspricht der Anzahl der Spalten bzw. der Feature Maps des zweiten Convolutional Layer. Auch die Berechnungen im Input Layer, den MaxPooling Layern und den Fully Connected Layern wird auf diese 64 Threads verteilt, um eine deterministische Anzahl an Threads zu erhalten. In der ersten Convolutional Schicht sind dabei allerdings 32 Threads ungenutzt. In den übrigen Schichten können alle 64 Threads ausgenutzt werden.

Offensichtlich ist die GTX Titan X mit 64 Threads keinesfalls ausgelastet. Um die Rechenleistung der Grafikkarte besser zu nutzen, wurde die beschriebene Modell-Parallelisierung mit einer Musterparallelisierung kombiniert. Die Software berechnet 150 Muster parallel mit ja 64 Threads. In der vorliegenden Implementierung werden 150 Threads gestartet, welche je einem Muster zugeordnet sind. Diese wiederum starten je 64 Threads zur Berechnung des Convolutional Neural Networks in oben beschriebener Weise. Dadurch werden pro Batch 9750 Threads gestartet. Die Batch-Size wurde dementsprechend angepasst und auf 150 gesetzt. Nach der Berechnung dieses Batches findet die Anwendung des *Stochastic Gradient-Descent*-Algorithmus statt. Dieser nutzt ebenfalls 64 Threads zur Anpassung der Gewichte und Biases.

9.3.7 Backpropagation

Die Methode der Unrolling-based Convolution beeinflusst in gewisser Weise auch die Anwendung des Backpropagation Algorithmus. Durch die Unrolling-based Convolution kann in der Backpropagation in den Convolutional Layern eine einfache Matrix-Matrix-Multiplikation durchgeführt werden. In den MaxPooling Schichten muss neben dem auffinden der richtigen Position des verwendeten Wertes auch die Umsortierung rückgängig gemacht werden. Dies wurde wie auch im Vorwärtszweig mit einer Hilfsmatrix gelöst. Diese wird zuerst befüllt, indem die Umsortierung rückgängig gemacht wird. Anschließend werden die richtigen Positionen der hier befindlichen Werte gefunden und befüllt, der Rest wird analog zur klassischen Variante mit Null befüllt. In den restlichen Schichten folgt auch die Implementierung mit Unrolling-based Convolution dem in Abschnitt 4.4 auf Seite 28 vorgestellten Algorithmus.
Die Parallelisierung erfolgt auch hier mit jeweils 64 Threads für ein Batch mit 150 Mustern parallel.

9.3.8 Zusammenfassung GPU-Implementierung

Der hohe Rechenaufwand, der beim Training von Neuronalen Netzen generell und speziell bei Convolutional Neural Networks auftritt, kann mit Hilfe moderner Grafikkarten in kürzester Zeit berechnet werden. Aufgrund der großen Zahl an Rechenwerken eignen sich Grafikkarten besonders gut zur Parallelisierung aufwändiger Rechenoperationen. Sie sind darauf ausgelegt, Operationen auf große Matrizen hoch parallel und effizient auszuführen. Mit der Methode der Unrolling-based Convolution kann die für die Berechnung von CNNs notwendige Faltungsoperation in eine eben solche Matrixoperation überführt werden. Damit ist es möglich die Faltung als Matrix -Matrix -Multiplikation darzustellen. Dies erfordert die Vervielfachung

und Umsortierung der Inputs des Convolutional Layers. Dadurch ergeben sich auch Änderungen in den anderen Schichten des Convolutional Neural Networks. Diese Methode kann nach erfolgreicher Implementierung beispielsweise modell-parallel implementiert werden. Dazu kann zum Beispiel über die Spalten der Matrizen der Convolutional Layer parallelisiert werden, wie in dieser Arbeit beschrieben. Zusätzlich können auch mehrere Muster parallel berechnet werden, um eine höhere Auslastung der Grafikkarte zu erzielen.

Leider war es in dieser Arbeit nicht möglich die Performance des beschriebenen Aufbaus auf der NVIDIA GTX Titan X zu bestimmen, da die Implementierung dessen nicht gelungen ist. Es ist ein lauffähiges Netz entstanden, welches nicht in der Lage war, einen Lerneffekt zu erzielen. Der Fehler muss dabei in der Implementierung liegen, da der verwendete Netzaufbau in anderen Implementierungen, sowohl in dieser Arbeit als auch bei anderen Autoren, bereits als tauglich bestätigt wurde.

10 Vergleich der unterschiedlichen Implementierungen

Zum Abschluss der Arbeit sollten die Implementierungen des CNN untereinander, sowie mit dem Beispiel auf Basis von Tensorflow, hinsichtlich ihrer Schnelligkeit (die Qualität bzw. Verwendbarkeit der Ergebnisse wird für die einzelnen Implementierungen getrennt betrachtet) verglichen werden. Um vergleichbare Ergebnisse zu erhalten, werden auf jedem Testsystem mehrere Implementierungen getestet. Für die Durchführung der Tests kommen zwei Rechner zum Einsatz:

it-phi Ein Zugang zu diesem Server wurde den Studenten für diese Arbeit freundlicherweise von der DHBW Stuttgart zur Verfügung gestellt: Er ist mit zwei Server-CPUs (Intel Xeon E5-2650L) und einem Coprozessor (Intel Xeon Phi) ausgestattet. Beim Xeon Phi handelt es sich um eine PCIe-Erweiterungskarte, auf der eine spezielle CPU mit 60 Kernen und einer Taktrate von 1 GHz, sowie ein DDR5-Arbeitsspeicher mit einer Größe von 8 GB verbaut sind.

Zweiter Testrechner Zum Testen der Implementierung für CUDA wird ein weiterer Testrechner benötigt. Dieser besitzt eine Intel-CPU (Core I7-7700HQ) sowie eine NVidia GTX1070.

Alle Implementierungen werden für die Programmtests derart angepasst, dass sie Trainingsdurchläfe und Evaluierungen für eine bestimmte Anzahl von Bildern durchführen. Idealerweise sollten Trainings- und Testläufe für je 1000, 10000 und 20000 Bilder durchgeführt werden. Tatsächlich werden nicht für jede Implementierung alle Werte bestimmt. Die serielle Implementierung wird mit je 100, 500 und 1000 Bildern getestet. Größere Datenmengen können von der seriellen Implementierung nicht in akzeptabler Zeit verarbeitet werden.

Alle getesteten Implementierungen basieren auf dem gleichen Netzaufbau. Als erstes gibt es einen Convolutional Layer mit einem Faltungskern mit 5*5 Elementen und 32 Feature Maps am Ende des Layers. Anschließend folgt ein Max Pooling Layer, der das Maximum aus 2*2 Quadraten aussucht. Anschließend folgt ein weiterer Convolutional Layer mit der selben Größe der Faltungskerne und diesmal 64 Feature Maps, sowie ein zweiter Max Pooling Layer. Am Ende folgen noch zwei Fully Connected Layer mit einer Größe von einmal 1024 Elementen und einmal 10 Elementen.

Zum Messen der Ausführungsdauer wurde das Linuxtool `time` verwendet. Dieses kann vor einem beliebigen Befehl in der Shell geschrieben werden. Ist der mitgegebene Befehl ausgeführt, zeigt `time` die vergangene zeit seit Beginn der Ausführung an. Dabei wird unterschieden in die tatsächlich verstrichene Zeit, die Zeit im User-Modus und die Zeit im Kernel-Modus.

Auf dem Server it-phi werden die Implementierungen für x86-CPUs, für die Tensorflow-Vorlage und für den Intel Xeon Phi getestet.

Das Messen der Ausführungszeit für die Implementierung auf einer x86 CPU auf dem Server it-phi ergab dabei folgende Werte:

1000 Durchläufe		20000 Durchläufe	
real	1m 53,461s	real	36m 57,646s
user	44m 55,92s	user	1082m 4,68s
sys	0m 2,30s	sys	0m 32,752s

Zu erkennen ist, dass die Zeit im User-Mode zirka um den Faktor 30 Größer ist, als die tatsächliche Ausführungszeit. Dies liegt am Multithreading des Programms. Wird während der Ausführung mit dem Befehl `top` die Auslastung des Prozessors angezeigt ist zu sehen, dass die Auslastung der CPU bei 2915% liegt. Dem Server stehen insgesamt 32 CPU-Kerne zur Verfügung, `top` rechnet dies auf einen CPU-Kern und so ist es möglich, mehr als 100% Prozessorauslastung angezeigt zu bekommen..

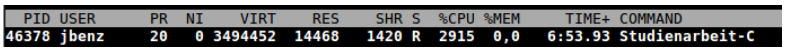


Abbildung 10.1: Eintrag mit Prozessorauslastung der x86-Implementierung

Bei der Ausführung der beiden anderen Implementierungen ergeben sich kürzere Laufzeiten für gleiche Datenmengen. Die nachfolgenden Diagramme zeigen, dass sich die Laufzeiten annähernd linear zur verarbeiteten Datenmenge verhalten. Ist der genaue Zusammenhang zwischen Datenmenge und Laufzeit bekannt, lassen sich daraus Schlüsse über die Dauer der Initialisierung und die durchschnittliche Verarbeitungszeit pro Bild ziehen.

Wie in den Diagrammen in Abbildung 10.2 abzulesen ist, benötigt die Implementierung für den Xeon Phi im Gegensatz zur Vorlage eine wesentlich größere Laufzeit zur Initialisierung des Programms. Allerdings ist die durchschnittliche Laufzeit zur Bearbeitung eines einzelnen Bildes wesentlich geringer. Beim Testdurchgang mit 20000 Trainingsbildern ist die gesamte Laufzeit der Implementierung für den Xeon Phi bereits geringer als die von der Vorlage benötigte Zeit. Die für eine einzelne Evaluierung benötigte Laufzeit ist wesentlich geringer als die Laufzeit eines

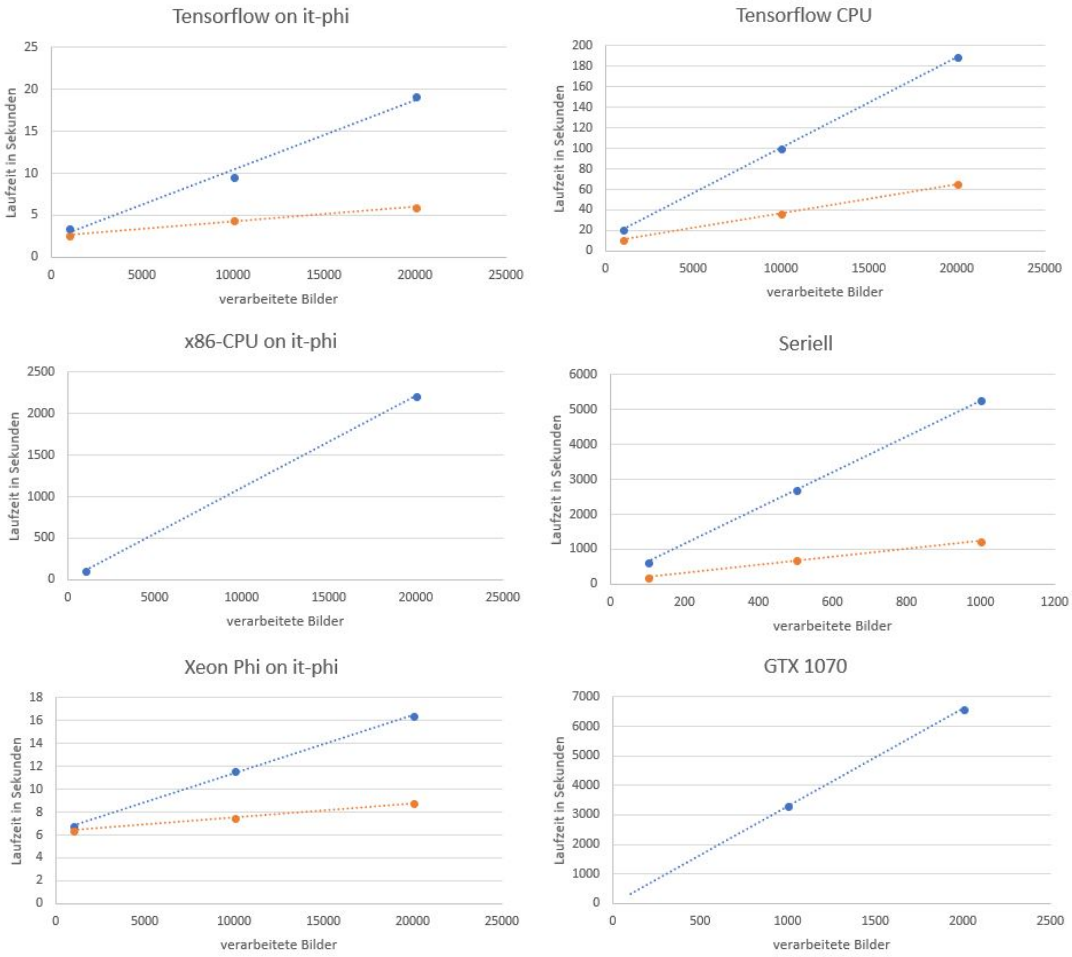


Abbildung 10.2: Ergebnisse auf dem zweiten Testrechner

Trainingsdurchgangs. In den Diagrammen ist dies anhand der deutlich langsamer steigenden Linien zu erkennen. Auch der Unterschied zwischen den Implementierungen fällt beim evaluieren wesentlich geringer aus. Daraus lässt sich folgern, dass die Implementierung für den Xeon Phi bei großen Mengen von Trainingsdaten auf diesem Server tatsächlich schneller arbeitet als Tensorflow. Der Grund dafür liegt allerdings nicht darin, dass die Implementierung für den Xeon Phi allgemein effizienter arbeiten würde. Der tatsächliche Grund ist, dass Tensorflow nicht zur Nutzung eines Coprozessors ausgelegt ist und den Xeon Phi nicht verwenden kann.

Auf dem zweiten Testrechner werden die serielle Implementierung, die Implementierung zur Nutzung von GPUs mithilfe von CUDA und zu Vergleichszwecken die Tensorflow-Vorlage (ohne Nutzung der GPU) getestet. Abbildung 10.2 zeigt Diagramme zu den für diesen Rechner ermittelten Testergebnissen. Wie am Diagramm für Tensorflow zu erkennen ist, benötigt das Referenzskript auf diesem Rechner die zehnfache Laufzeit gegenüber der Server-CPU auf dem it-phi. Die Implementierungen dieses Testlaufs lassen sich nur bedingt miteinander vergleichen. Um die Programmtests in akzeptabler Zeit abschließen zu können, werden die serielle Implementierung, sowie die Implementierung zur Nutzung von GPUs mit kleineren Datenmengen getestet. Aus dem zugehörigen Diagramm lässt sich trotzdem gut ablesen, dass die serielle Implementierung das langsamste aller hier betrachteten Programme ist. Dies entspricht auch den Erwartungen; sie ist weder zur Nutzung von Parallelisierung, noch durch Vektorisierung optimiert. Obwohl der Implementierung für CUDA im Gegensatz zu Tensorflow in der Lage ist, Rechenoperationen auf eine Grafikkarte auszulagern, ist sie deutlich langsamer als die Referenzimplementierung. Der Grund für die lange Laufzeit des CUDA-Programms liegt vermutlich in der Auslagerung von Code, der bedingte Sprünge enthält. GPUs sind zwar prinzipiell zur Ausführung von verzweigtem Code fähig, allerdings läuft dieser sehr ineffizient. Auf die schlechte Eignung von GPUs für branch prediction wird bereits in Abschnitt 9.1.1 hingewiesen. Möglicherweise ergibt sich eine weitere Verlängerung der Laufzeit durch die Verwendung von *dynamic parallelism* anstatt *konkurrierender Streams*. Eine Vergleich dieser Parallelisierungsansätze ist allerdings nicht Teil der Arbeit und wurde in deren Verlauf auch nicht durchgeführt.

Die Bewertung verschiedener Lösungsansätze hinsichtlich der Eignung für den konkreten Anwendungsfall stellt allgemein für alle in dieser Arbeit erstellten Programme die größte Herausforderung dar. Es ist davon auszugehen, dass alle erstellten Implementierungen hinsichtlich ihrer Effizienz weiter verbessert werden können.

Literatur

[1] VoiceLaby, *Absatz von Geräten mit integriertem digitalen Sprachassistenten (z.B. Amazon Echo, Google Home) weltweit in den Jahren 2015 und 2016 sowie eine Prognose für 2017 (in Millionen Stück)*, [Online; Zugriff am 17. Mai 2018]. Adresse: `https://de.statista.com/statistik/daten/studie/751649/umfrage/absatz-von-geraeten-mit-einem-digitalen-sprachassistenten-weltweit/`.

[2] V.-I. W. Hilberg IEEE, “"Künstliches Gehirnäls Nachfolger der Computers”, S. 37–40, Apr. 2014.

[3] *Wie viele Nervenzellen hat das Gehirn?*, [Online; Zugriff am 17. Mai 2018], Apr. 2015. Adresse: `https://www.helmholtz.de/gesundheit/wie-viele-nervenzellen-hat-das-gehirn/`.

[4] F. Rosenblatt, “The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain”, S. 1–22, Nov. 1958.

[5] S. 446, 1993.

[6] M. Nielsen, *Neural Networks and Deep Learning*, [Online; Zugriff am 8. Mai 2018], 2017. Adresse: `http://neuralnetworksanddeeplearning.com`.

[7] M. A. Nielson. (2015). Neural Networks and Deep Learning, Adresse: `http://neuralnetworksanddeeplearning.com/` (besucht am 26. 05. 2018).

[8] S. 533–537, 1986.

[9] D. Yu, H. Wang, P. Chen und Z. Wei, "Mixed Pooling for Convolutional Neural Networks", S. 364–375, Okt. 2014.

[10] *A Guide to TF Layers: Building a Convolutional Neural Network*, [Online; Zugriff am 23. Mai 2018], Apr. 2018. Adresse: https://www.tensorflow.org/tutorials/layers.

[11] *Using Convolutional Neural Networks for Image Recognition*, [Online; Zugriff am 23. Mai 2018], 2015. Adresse: https://ip.cadence.com/uploads/901/cnn_wp-pdf.

[12] *Deep MNIST for Experts*, [Online; Zugriff am 23. Mai 2018], Juni 2017. Adresse: https://www.tensorflow.org/versions/r1.1/get_started/mnist/pros.

[13] *Performance and Diagnostic Tools in Visual Studio 2015*, [Online; Zugriff am 24. Mai 2018], Juli 2015. Adresse: https://blogs.msdn.microsoft.com/devops/2015/07/20/performance-and-diagnostic-tools-in-visual-studio-2015/.

[14] *CPU performance*, [Online; Zugriff am 23. Mai 2018]. Adresse: https://asteroidsathome.net/boinc/cpu_list.php.

[15] 2008, korrigierter Nachdruck 2009.

[16] *Scheduling Prozess-Ablaufplanung*, [Online; Zugriff am 28. Mai 2018], 2011. Adresse: http://www.inf.fu-berlin.de/lehre/WS11/OS/slides/OS_V7_Scheduling_Teil_1.pdf.

[17] I. Corporation. (2018), Adresse: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html (besucht am 29. 05. 2018).

[18] *TensorFlow*, [Online; Zugriff am 24. Mai 2018]. Adresse: https://www.tensorflow.org/.

[19] *Intel Xeon Phi Processors*, [Online; Zugriff am 25. Mai 2018]. Adresse: https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html.

[20] *Intel® Many Integrated Core Architecture - Advanced*, [Online; Zugriff am 25. Mai 2018]. Adresse: https://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[21] *Programming Intel's Xeon Phi: A Jumpstart Introduction*, [Online; Zugriff am 25. Mai 2018], Dez. 2012. Adresse: http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160.

[22] *Levels of Communication*, [Online; Zugriff am 25. Mai 2018]. Adresse: https://cvw.cac.cornell.edu/optimization/levelscomm.

[23] "Intel® Xeon Phi™ Coprocessor DEVELOPER'S QUICK START GUIDE", S. 1–31, 2014.

[24] *The C Preprocessor*, [Online; Zugriff am 25. Mai 2018]. Adresse: https://gcc.gnu.org/onlinedocs/gcc-4.6.1/cpp/Pragmas.html.

[25] *Intel® C++ Compiler 17.0 Developer Guide and Reference*, [Online; Zugriff am 26. Mai 2018], 2018. Adresse: https://software.intel.com/sites/default/files/managed/08/ac/PDF_CPP_Compiler_UG_17_0.pdf.

[26] *POSIX Threads Programming*, [Online; Zugriff am 26. Mai 2018], Juli 2017. Adresse: https://computing.llnl.gov/tutorials/pthreads/.

[27] *A "Hands-on" Introduction to OpenMP*, [Online; Zugriff am 26. Mai 2018]. Adresse: https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf.

[28] *Intel TBB Tutorial*, [Online; Zugriff am 27. Mai 2018]. Adresse: https://www.threadingbuildingblocks.org/intel-tbb-tutorial.

[29] "Best Practice Guide Intel Xeon Phi v1.1", Feb. 2014.

[30] *Developer Reference for Intel® Math Kernel Library 2018 - C*, [Online; Zugriff am 28. Mai 2018], März 2018. Adresse: https://software.intel.com/en-us/mkl-developer-reference-c.

[31] *COG*, [Online; Zugriff am 29. Mai 2018], Jan. 2015. Adresse: https://nedbatchelder.com/code/cog/.

[32] "Sparse autoencoder", 2011, [Online; Zugriff am 31. Mai 2018]. Adresse: http://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf.

[33] N. Corp. (), Adresse: https://developer.nvidia.com/cuda-zone (besucht am 26. 05. 2018).

[34] ——, (), Adresse: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (besucht am 26. 05. 2018).

[35] M. Harris. (2017), Adresse: https://devblogs.nvidia.com/even-easier-introduction-cuda/ (besucht am 26. 05. 2018).

[36] ——, (2013), Adresse: https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (besucht am 26. 05. 2018).

[37] N. Corp. (), Adresse: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (besucht am 26. 05. 2018).

[38] I. W. Selesnick und C. S. Burrus, "Extending Winograd's small convolution algorithm to longer lengths", in *Proceedings of IEEE International Symposium on Circuits and Systems - ISCAS '94*, Bd. 2, Mai 1994, 449–452 vol.2. DOI: 10.1109/ISCAS.1994.408999.

[39] V. Podlozhnyuk. (2007), Adresse: http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/convolutionFFT2D/doc/convolutionFFT2D.pdf (besucht am 26. 05. 2018).

[40] P. S. K. Chellapilla S. Puri. (2006), Adresse: https://hal.inria.fr/inria-00112631/document (besucht am 26. 05. 2018).