technische universität
dortmund

fakultät für informatik

# Masterarbeit

## Zeit-effizientes Training von Convolutional Neural Networks

Jessica Buehler

12. März 2020

**Gutachter:**

Prof. Dr. Heinrich Müller

M.Sc. Matthias Fey

Lehrstuhl VII
Informatik
TU Dortmund

# Inhaltsverzeichnis


**1 Einleitung** **1**

1.1 Motivation und Hintergrund dieser Arbeit . . . . . 1
1.2 Aufbau der Arbeit . . . . . 1
1.3 Suchbegriffe . . . . . 1

**2 Stand der Wissenschaft** **3**

2.1 Funktionsweise eines CNNs . . . . . 3
2.2 Überblick über die gängigen Methoden . . . . . 5
2.2.1 Verringerung der für Berechnungen nötige Zeit . . . . . 5
2.2.2 Berechnung mit 16 Bit Dynamischen Festkommazahlen . . . . 7
2.3 Beschleunigung der Berechnung des Gradientenabstiegsverfahren . . . 7
2.3.1 Accelerating CNN Training by Sparsifying Activation Gradients 8
2.3.2 Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks . . . . . 8
2.3.3 Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent . . . . . 8
2.3.4 Accelerated CNN Training Through Gradient Approximation 8
2.4 Verfahren um weniger Trainingsdaten zu verwenden . . . . . 8
2.4.1 Stochastisches Pooling . . . . . 8
2.4.2 Lernen von Struktur und Stärke von CNNs . . . . . 8
2.5 Strukturelle Veränderungen zur Beschleunigung des Trainings . . . . 8
2.5.1 Pruning um Trainingszeit zu minimieren . . . . . 8
2.5.2 Net 2 Net . . . . . 11
2.5.3 Kernel rescaling . . . . . 11
2.5.4 Resource Aware Layer Replacement . . . . . 11
2.6 Weitere Herangehensweisen . . . . . 11
2.6.1 Tree CNN . . . . . 11
2.6.2 Standardization Loss . . . . . 11
2.6.3 Wavelet . . . . . 11

**3 Experimentelle Untersuchung der möglichen Strategien 13**
3.1 Experimentales Setup . . . 13
3.2 Überblick über die möglichen Strategien . . . 13
3.2.1 Zahlenformate . . . 13
3.2.2 Beschleunigung der Berechnung des Gradientenabstiegverfahren 14
3.2.3 Verfahren um weniger Trainingsdaten zu verwenden . . . 14
3.2.4 Lernen von Struktur und Stärke von CNNs . . . 14
3.2.5 Strukturelle Veränderungen . . . 14
3.2.6 andere Herangehensweisen . . . 15
3.2.7 Tensorflow vs. PyTorch . . . 15
3.3 Durchführung der Experimente . . . 15
3.3.1 Einfluss der Batch Größe . . . 15
3.3.2 Kombination von Net2Net mit PruneTrain . . . 15
3.4 Evaluation der Ergebenisse . . . 15

**4 Konklusion 17**

**I Additional information 19**

**Abbildungsverzeichnis 21**

**Algorithmenverzeichnis 23**

**Quellcodeverzeichnis 25**

**Literaturverzeichnis 27**

# Todo list


Einleitung fertigschreiben – zum Schluss . . . . . 1
Aufbau der Arbeit – erst bei fortgeschrittener Arbeit schreiben . . . . . 1
verwendete Suchbegriffe . . . . . 1
kurzer Sectioneinleitungstext über die gängigen Methoden . . . . . 5
cite . . . . . 6
subnormale Zahlen . . . . . 6
ref . . . . . 7
ref . . . . . 7
Figure: Schema . . . . . 7

# Mathematical Notation

| Notation | Meaning |
|---|---|
| $\mathbb{N}$ | Set of natural numbers $1, 2, 3, \ldots$ |
| $\mathbb{R}$ | Set of real numbers |
| $\mathbb{R}^d$ | $d$-dimensional space |
| $\mathcal{M} = \{m_1, \ldots, m_N\}$ | Set $\mathcal{M}$ of $N$ elements $m_i$ |
| $\mathbf{p}$ | Vector |
| $\mathbf{p}_i$ | Element $i$ of the vector |
| $\mathbf{v}_i^{(j)}$ | Element $i$ of the vector $j$ |
| $\mathbf{A}$ | Matrix |

# 1 Einleitung

## 1.1 Motivation und Hintergrund dieser Arbeit

Gegeben: Mehrere Datensätze aus Bildern, die in verschiedene vorgegebene Klassen klassifiziert werden sollen.
Gesucht: Effiziente Strategie um möglichst zeitsparend eine möglichst gute Klassifikationsleistung zu bekommen
Also ein Optimierungsproblem auf verschiedenen Strategien zum Trainieren von CNNs mit zwei zu Optimierenden Grössen:

- Zeit effizientes Training
- Gute Klassifikationsleistung

Gesucht ist also eine Pareto-Front
Einleitung fertigschreiben – zum Schluss

## 1.2 Aufbau der Arbeit

Aufbau der Arbeit – erst bei fortgeschrittener Arbeit schreiben

## 1.3 Suchbegriffe

verwendete Suchbegriffe

# 2 Stand der Wissenschaft

Diese Kapitel soll dem Leser eine Übersicht über den aktuellen Stand der Wissenschaft geben. Zu diesem Zweck hat dieses Kapitel zwei Teile. Im ersten Teil wird zunächst grundlegend die Funktionsweise von CNNs erläutert. Im zweiten Teil des Kapitels wird ein Überblick über die bisherigen wissenschaftlichen Erkenntnisse im Themenbereich dieser Arbeit vorgetellt.

## 2.1 Funktionsweise eines CNNs

Die Quelle für dieses Unterkapitel ist soweit nicht anders vermerkt ein Buch über „Deep Learning“ [GBC16].
CNNs sind spezielle neuronale Netze. Der Unterschied zu einem „Multilayer-Perzeptron (MLP)[1]“ ist, dass bei einem MLP jede Verbindung zwischen Neuronen und die Neuronen selber ein eigenes trainierbares Gewicht haben. Aus diesen trainierbaren Werten wird mittels einer Matrixmultiplikation mit den Eingabedaten bzw. den Daten der vorherigen Schicht die Ausgabe jedes Neurons berechnet. Im Gegensatz dazu sind CNNs neuronale Netze, die in mindestens einer ihrer Schichten die Faltung anstelle der allgemeinen Matrixmultiplikation verwenden.
Dies bedeutet, dass die Eingabedaten für ein CNN für diese Faltung geeignet sein müssen. Geeignet für die Faltung sind Eingabedaten, die gridförmig angeordnet sind. Bilddaten sind ein grosser Anwendungsbereich für CNNs.
Bei der Faltung wird auf die Eingabedaten bzw- die Daten dr vorherigen Schicht ein Kernel angewendet.
In Abbildung 2.1 ist zu sehen wie die Faltung auf einem Bild durchgeführt wird. Der Kernel wird auf jedes Teilbild mit der Grösse des Kernels angewendet. Die korrespondierenden Felder werden multipliziert und alle entstehenden Produkte werden addiert. So entsteht aus der Faltung des Kernels mit der Eingabe in die entsprechende Schicht eine Featuremap.

[1] Die Hintergründe des MLPs und allgemein neuronaler Netzwerke werden hier nicht behandelt. Für eine Einführung in neuronale Netzwerke kann aber [Hay98] herangezogen werden

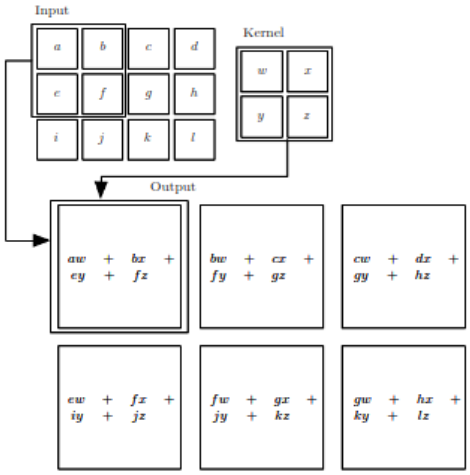



**Abbildung 2.1:** Abbildung zur Faltung [GBC16]

Mehrere dieser Kernel bilden zusammen ein Convolutional Layer. Dabei werden die Kernel wie in Abbildung 2.2 gezeigt
In einem CNN werden mehrere dieser Convolutional Layer hintereinander geschaltet, um komplexe Features erkennen zu können.
Eine beispielhafte Übersicht über die CNN-Architektur ist in Abbildung 2.2 zu sehen.

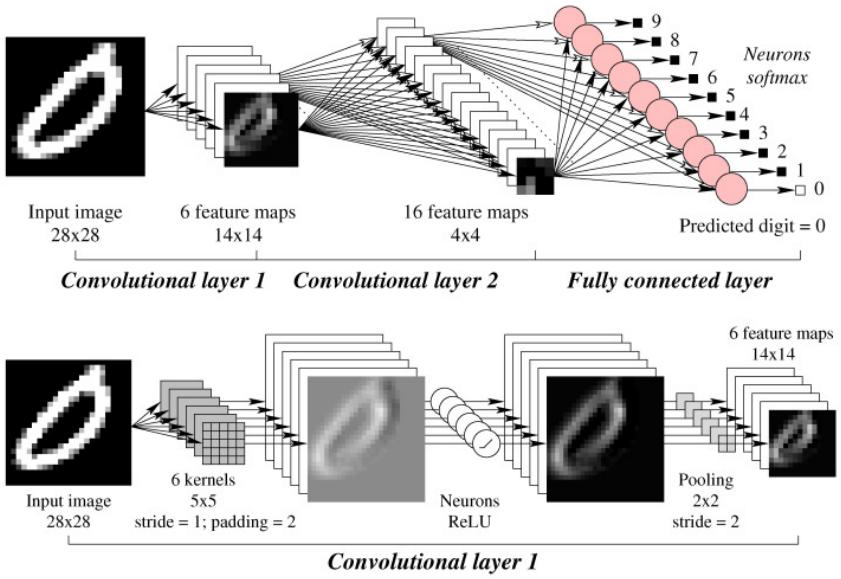



**Abbildung 2.2:** Convolutional Neural Net [CCGS16]

In Abbildung 2.2 ist zu sehen, dass ein CNN aus hintereinander geschalteten Conv-Layer besteht. Die Funktion der Conv-Layer wird nun näher betrachtet.
Wie in Abbildung 2.2 zu sehen ist, ist ein Conv-Layer die hintereinander Schaltung verschiedener Operationen:

- Faltung des Eingabebildes mit dem Kernel
- Anwendung der Aktivierungsfunktion ReLU
- Pooling

Abbildung 2.2 beinhaltet auch noch die Begriffe Stride und Padding, welche für das Verstehen dieser Arbeit zwar nicht notwendig sind, hier der Vollständigkeit halber trotzdem erklärt werden.
Padding löst das Problem, das entsteht, wenn der Mittelpunkt des Filters auf ein Pixel im Randbereich gelegt wird. Hier taucht das Problem auf, dass der Filter auch auf nicht vorhandenen Pixeln aufliegt und die Filteroperation hier somit nicht definiert ist. Padding setzt nun den Rand fort oder belegt diesen Rand mit einem festgelegten Wert, um dort eine gültige Filteroperation zu erzeugen.
Stride ist der Parameter, der bestimmt um wie viele Felder der Filter nach der Anwendung verschoben werden soll. Bei größerem Stride kann die entstehende Feature-Map verkleinert werden.
Pooling ist eine Operation, die die Größe der Feature-Map verkleinert und somit Overfitting vermeidet.
Die Fully-Connected-Layer errechnen aus den Ausgängen der Convolutional-Layer, in welche Klasse ein Objekt klassifiziert werden soll.
Die Filter, die auf die Feature Maps bzw. die Eingabebilder angewendet werden, sind trainierbar. Zusätzlich sind auch die Gewichtungen des Fully-Connected Layers trainierbar. Das heißt durch den Trainingsprozess wird versucht die Werte in der Filtermatrix und des Fully-Connected Layer so zu verändern, dass das gesamte CNN besser klassifizieren kann. Für diese Veränderung wird ein Gradientenabstiegsverfahren, welches rückwärts durch die Schichten propagiert wird, benutzt.

## 2.2 Überblick über die gängigen Methoden

kurzer Sectioneinleitungstext über die gängigen Methoden

### 2.2.1 Verringerung der für Berechnungen nötige Zeit

Die Zeit, die ein Convolutional Layer braucht um berechnet zu werden hängt ab von:

- dem verwendeten Zahlenformat

- der Filtergrösse
- der Bildgrösse
- dem verwendetem Stride und Padding

Beim Verändern der Filter- oder der Bildgrösse, um Trainingszeit zu sparen, verändert sich auch die Erkennungsleistung . Dies ist beim Verändern des verwendeten Zahlenformats nicht umbedingt gegeben. Standardformat ist eine 32 Bit Gleitkommazahl. Die einfachste Methode hier Trainingszeit zu sparen ist das Halbieren der Bitanzahl auf 16 Bit. Eine weitere Methode ist das Benutzen von 16 Bit Dynamischen Festkommazahlen. Die beiden alternativen Methoden haben unterschiedliche Anforderungen an die Ausführungsplattform. Diese Anforderungen und die Besonderheiten der beiden Verfahren werden in den folgenden zwei Unterkapiteln näher beleuchtet.

**Berechnung mit 16 Bit Gleitkomma** [? ] Die 16 Bit Gleitkommazahl unterscheidet sich nicht nur in der Länge von der 32 Bit Zahl sondern aus der unterschiedlichen Länge erwachsen Unterschiede in den darstellbaren Zahlen. In Tabelle 2.1 sind diese Unterschiede dargestellt.

**Tabelle 2.1:** Darstellbare Zahlen von 16 und 32 Bit

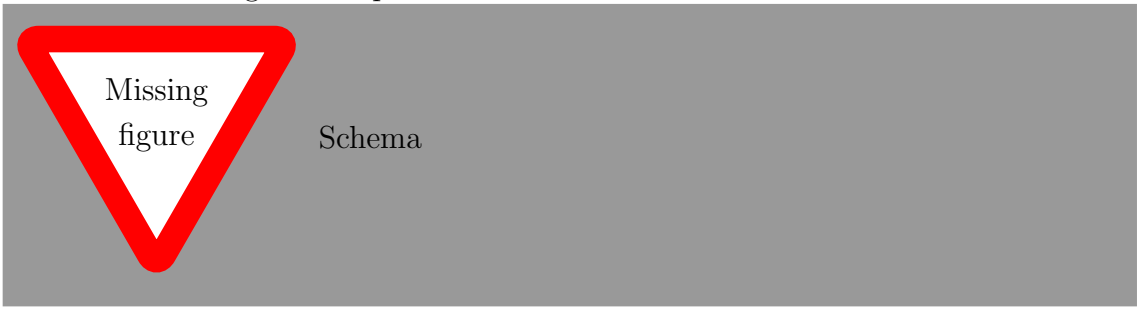
| | 16 Bit | 32 Bit |
|---|---|---|
| kleinste darstellbare positive Zahl | $0.61 \cdot 10^{-4}$ | $1.1755 \cdot 10^{-38}$ |
| grösste darstellbare positive Ganzzahl | 65504 | $3.403 \cdot 10^{38}$ |
| minimal subnormale Zahl | $2^{-24} \approxeq 5.96 \cdot 10^{-8}$ | $2^{-149}$ |

Subnormale Zalen ergeben sich, wenn der Exponent 0 ist und
subnormale Zahlen

Durch diese Unterschiede im Umfang der darstellbaren Zahlen ergibt sich ein direkter Unterschied im Training eines CNNs. Durch den Wechsel auf 16 Bit ist ein bestimmter Teil der Gradienten gleich Null.
Diese Nachteile von 16 Bit Gleitkommazahlen können durch drei Techniken abgemeildert oder sogar komplett aufgehoben werden:

- 32 Bit Mastergewichte und Updates
- Sklaierung der Loss-Funktion
- Arithmetische Präzision

Beim Trainieren von neuronalen Netzwerken mit 16 Bit Gleitkommazahlen werden die Gewichte, Aktivierungen und Gradienten im 16 Bit Format gespeichert. Die Speicherung der Gewichte als 32 Bit Mastergewichte hat zwei mögliche Erklärungen, die aber nicht immer zutreffen müssen.
Um nach einem Forward Druchlauf des Netzes die Gewichte abzudaten wird ein Gradientenabstiegsverfahren benutzt. Hierbei werden die Gradienten der Gewichte berechnet. Um für die Funktion, die das CNN approximiert einen besseren Approximationserfolg zu erlangen wird dann dieser Gradient mit der Lernrate multipliziert. Wird dieses Produkt in 16 Bit abgespeichert, so ist in viele Fällen das Produkt der beiden Zahlen gleich Null. Dies liegt an der Taqtsache, dass wie in Tabelle zu sehen ref ist die kleinste darstellbare Zahl in 16 Bit wesentlich grösser ist als in 32 Bit.
Der zweite Grund wieso man Mastergewichte brauchen könnte ist die Tatsache, dass bei grossen Gewichten die Länge der Mantisse nicht ausreicht, um sowohl das Gewicht als auch das zu addierende Update zu speichern.
Aus den beiden Gründen wird das in Abbildung gezeigte Schema zum Trainieren ref einer Schicht mit gemischt präzisen Gleitkommazahlen benutzt.




### 2.2.2 Berechnung mit 16 Bit Dynamischen Festkommazahlen

Quelle: [DMM+18]

## 2.3 Beschleunigung der Berechnung des Gradientenabstiegsverfahren

Bei der Beschleunigung der Berechnung des Gradientenabstiegsverfahren gibt es vier verschiedene publizierte Herangehensweisen:

- Accelerating CNN Training by Sparsifying Activation Gradients
- Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks

- Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent
- Accelerated CNN Training Through Gradient Approximation

### 2.3.1 Accelerating CNN Training by Sparsifying Activation Gradients

### 2.3.2 Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks

### 2.3.3 Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent

### 2.3.4 Accelerated CNN Training Through Gradient Approximation

## 2.4 Verfahren um weniger Trainingsdaten zu verwenden

### 2.4.1 Stochastisches Pooling

### 2.4.2 Lernen von Struktur und Stärke von CNNs

## 2.5 Strukturelle Veränderungen zur Beschleunigung des Trainings

### 2.5.1 Pruning um Trainingszeit zu minimieren

Pruning ist eine Technik, die entwickelt wurde, um die Inferenzzeit eines neuronalen Netzwerks zu reduzieren. Das Pruningverfahren wird auf das bereits trainierte Netz angewendet. Dabei wird entschieden, welche Gewichte nur einen minimalen Effekt auf das Klassifikationsergebnis haben um diese zu entfernen.
Aktueller Gegenstand der Forschung ist hier die Frage, ob diese kleineren Netzwerke nicht bereits ab Epoche Null trainiert werden können, um so Trainingszeit zu sparen. Dieser Ansatz wurde in verschiedenen Veröffentlichungen untersucht:

- Prune Train

- The Lottery Ticket Hypothesis

Zunächst werden die einzelen Verfahren erläutert, um sie danach miteinander zu vergleichen.

### Prune Train

Prune Train fügt einen Normalisierungsterm zur Loss-Funktion des Netzwerkes hinzu. Dies geschieht, damit der Optimierungsprozess dazu gezwungen wird möglichst kleine Gewichte zu wählen. Durch diesen Prozess wird aus dem dense Netz ein sparse Netz. Dieses sparse Netz sorgt allerdings noch nicht für weniger Zeitbedarf einer Trainingsepoche, da für ein Sparse aufwändige Datenindextechniken notwendig sind. Daher wird bei diesem Verfahren das Netz rekonfiguiert um das Modell kleiner und die Struktur wieder dense zu machen. Dabei hat Prune Train drei zentrale Optimierungsverfahren:

- eine systematische Methode zur Berechnung des group lasso Regularisierung Sanktions Koeffizienten beim Beginn des Trainings.

- Kanal union, ein Speicheraufruf kosteneffizientes und Index-freies Kanal Pruning Verfahren für moderne CNNs mit Kurzschlussverbindungen.

- Ein dynamische Mini-Batch Adjustment, dass die Größe des Mini-Batch anpasst. Dies geschieht durch beobachten des Speicherkapazitätgebrauchs einer Trainingsiteration nach jeder Pruning reconfiguration.

Der group lasso Regularisierung Sanktions Koeffizienten ist ein Hyperparameter, der einen Trade-off zwischen der Modellgröße und der Accuracy bildet. Voherige Arbeiten suchen nach einem geeignetem Sanktionsmaß, was das Einbeziehen des Prunings vom Anfang des Trainings sehr teuer macht. Unser Mechanismus kontrolliert die Group lasso Regularisierungstärke und erreicht eine hohe Modellpruningrate mit nur einem kleinen Einfluss auf die Accuracy bei nur einem Trainingsdurchlauf. Kurzschlussverbindungen werden in modernen CNNs häufig genutzt. Prunning aller genullten Kanäle solcher CNNs brauchen regelmässige Tensor Umordnung um die Kanalindizes zwischen den Schichten zu matchen. Dies vermindert die Performance. Diese Umordnung wird durch den Channel Union Algorithmus vermieden. Daher folgt eine 1.9 fache Beschleunigung des Convolutional Layetrs.
Dynmaisches Mini Batch Adjustment kompensiert die verminderte Datenparallelität aufgrund des kleineren geprunten Modells durch Erhöhung der Mini-Batch Größe.

Dies sorgt sowohl für bessere Ausnutzung der Hardware ressourcen als auch zur Reduzierung des KOmmunikation overheads durch eine Verminderte Modell Update Frequenz. Beim Erhöhen der Mini-Batch Größe wird auch die Lernrate mit demselben Verhältnis erhöht, um die Accuracy nicht zu verändern.

$$\underset{min}{W}\left(\frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i, W)) + \sum_{g=1}^{G} \lambda_g \cdot ||W_g||_2\right)$$

- $\frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i, W))$ Standard-Kreuzentropie
- $\sum_{g=1}^{G} \lambda_g \cdot ||W_g||_2$ group lasso Regulierungsterm
- $f(x_i)$ Vorhersage des Netzwerks auf Eingabe $x_i$
- $W$ Gewichte
- $l$ Verlustfunktion der Klassifikation und Grundwahrheit $y_i$
- $N$ Minibatchgröße
- $G$ Zahl von Gruppen
- $\lambda$ Verdünnungskoeffizient

$$\lambda \cdot \sum_{l=1}^{L}\left(\sum_{c_l=1}^{C_l} ||W_{c_l,:,:,:}||_2 + \sum_{k_l=1}^{K_l} ||W_{:,k_l,:,:}||_2\right)$$

Design eines speziellen Groupo Lasso Regulierers, der Gewichte jedes Kanals (Input oder Output) und jeder Schicht gruppiert. $\lambda$ wird als einziger globaler Regularisierungsfaktor gewählt, da so der Fokus auf dem Vermindern der Rechenzeit liegt und nicht auf der Modellgröße. Dies hat zur Folge, dass vorallem große Features verdünnt werden, was zu einer größeren Verminderung der Rechenleistung führt.
Um die Lasso Group Regularisierung vom Anfang des Trainings zu benutzen sollteder Koeffizient $\lambda$ sinnvoll gewählt werden. Dies sorgt für eine hohe Vorhersageaccuracy und einer hohen Pruning Rate. Um zeitintensives Hyperparametertuning zu vermeiden wird hier eine neue Methode eingeführt:

$$LPR = \frac{\lambda \sum_g^G ||W_{g,:}||}{l(y_i, f(x_i, W)) + \lambda \sum_g^G ||W_{g,:}||)}$$

Berechnet wird dies durch setzen von zufälligen Werten, mit denen die Gewichte initialisiert werden. LPR wird einmal berechnet und dann bis zum Ende weiter benutzt.
Nach jedem solchen Intervall werden Input und Outputkanäle die 0 sind gepruned. Um ein Missverhältnis zwischen den Dimensionen zu verhindern wird nur die Verbindung von 2 verdünnten Kanälen von 2 aufeinanderfolgenden Schichten gepruned. Alle Trainingsvariablen bleiben gleich. Das Reconfigurationsintervall ist ist der einzige zusätzliche Hyperparameter. Zu gross gewählt würde der Intervallparameter zu wenig Zeitverbesserung bringen. Zu klein gewählt könnte er die Lernqualität beeinflussen. 4 Matriken zur Evaluierung: Training und Inference FLOPs, gemessenen Trainingszeit, und Validierungsaccuracy.

#### The Lottery Ticket Hypothesis

### 2.5.2 Net 2 Net

### 2.5.3 Kernel rescaling

### 2.5.4 Resource Aware Layer Replacement

## 2.6 Weitere Herangehensweisen

### 2.6.1 Tree CNN

### 2.6.2 Standardization Loss

### 2.6.3 Wavelet

# 3 Experimentelle Untersuchung der möglichen Strategien

## 3.1 Experimentales Setup

Server mit 4 Graka: 2 mal Geforce GTX 1080 Ti mit CUDA Version 10.1 2 mal Geforce RTX 2080 Ti mit CUDA Version 10.1

## 3.2 Überblick über die möglichen Strategien

Welchen Strategien aus Kapitel 2 sind überhaupt durchführbar und welche sind kombinierbar? Hier werden nur die Strategien aufgeführt, welche überhaupt auf vernünftig grossen Datensätzen funktionieren und von der Technik her möglich sind. Die Strategien sind aufgeteilt in Unterkapitel.
Alle möglichen Kombinationen von Strategien sind zuviele. Daher sinnvolle Vorauswahl treffen. Bei mehreren gleichartigen/ konkurierenden Ansätze drekter Vergleich und dann den besten auswählen.

### 3.2.1 Zahlenformate

- FP16 bereits probiert
- DFP 16 without Swalp
- DFP 16 with Swalp

DFP 16 bisher nur auf CPU sinnvoll
FP16 nur auf RTX 2080 sinnvoll

### 3.2.2 Beschleunigung der Berechnung des Gradientenabstiegverfahren

#### Accelerating CNN Training by Sparsifying Activation Gradients

Funktioniert nur auf Toy-Benchmarks

#### Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks

Könnte funktionieren. Code für Lasagne: https://github.com/TimSalimans/weight_norm

#### Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent

Interessant bisher kein Code verfügbar

#### Accelerated CNN Training Through Gradient Approximation

Interessant bisher kein Code verfügbar

### 3.2.3 Verfahren um weniger Trainingsdaten zu verwenden

#### Stochastisches Pooling

Klingt sehr interessant und könnte für deutlich kleinere Trainingsdatenmenge sorgen https://github.com/Shuangfei/s3pool

### 3.2.4 Lernen von Struktur und Stärke von CNNs

bisher kein Code verfügbar.Klingt aber interessant

### 3.2.5 Strukturelle Veränderungen

#### PruneTrain

Code vorhanden

#### Net2Net

Code vorhanden

### 3.2.6 andere Herangehensweisen

### 3.2.7 Tensorflow vs. PyTorch

## 3.3 Durchführung der Experimente

Arbeite auf Grundlage des PruneTrain Codes.

### 3.3.1 Einfluss der Batch Größe

Im PruneTrain paper ist angegeben, dass die Batch Size so gross gewählt wird, dass der GPU Speicher ausgenutzt wird. Es gibt Paper, die in Frage stelen, ob eine grössere Batch Size nicht der Performance schadet [HHS17]

### 3.3.2 Kombination von Net2Net mit PruneTrain

Jedes Bildklassifizierungsproblem hat

## 3.4 Evaluation der Ergebenisse

# 4 Konklusion

# Teil I

# Additional information

# Abbildungsverzeichnis

2.1 Abbildung zur Faltung [GBC16] . . . . . 4
2.2 Convolutional Neural Net [CCGS16] . . . . . 4

# Algorithmenverzeichnis

# Quellcodeverzeichnis

# Literaturverzeichnis


[CCGS16] J.F. Couchot, R. Couturier, C. Guyeux, and M. Salomon. Steganalysis via a convolutional neural network using large convolution filters. *CoRR*, 2016.

[DMM+18] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj D. Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesús Corbal, Nikita Shustrov, Roman Dubtsov, Evarist Fomenko, and Vadim O. Pirogov. Mixed precision training of convolutional neural networks using integer operations. *CoRR*, abs/1802.00930, 2018.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[Hay98] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[HHS17] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1731–1741. Curran Associates, Inc., 2017.