

## Masterarbeit

### Zeit-effizientes Training von Convolutional Neural Networks

Jessica Buehler

5. April 2020

#### **Gutachter:**

Prof. Dr. Heinrich Müller

M.Sc. Matthias Fey

Lehrstuhl VII  
Informatik  
TU Dortmund






# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation und Hintergrund dieser Arbeit . . . . .	1
1.2	Aufbau der Arbeit . . . . .	1
1.3	Suchbegriffe . . . . .	1
<b>2</b>	<b>Stand der Wissenschaft</b>	<b>3</b>
2.1	Funktionsweise eines CNNs . . . . .	3
2.2	ResNet – eine neuere CNN-Architektur . . . . .	5
2.3	Strukturelle Methoden . . . . .	5
2.3.1	MorphNet . . . . .	6
2.4	Zeitsparende Methoden . . . . .	6
2.4.1	Verringerung der für Berechnungen nötige Zeit . . . . .	6
2.4.2	Beschleunigung der Berechnung des Gradientenabstiegsverfahren . . . . .	8
2.5	Verfahren um weniger Trainingsdaten zu verwenden . . . . .	8
2.5.1	Stochastisches Pooling . . . . .	8
2.5.2	Lernen von Struktur und Stärke von CNNs . . . . .	8
2.6	Strukturelle Veränderungen zur Beschleunigung des Trainings . . . . .	8
2.6.1	Pruning um Trainingszeit zu minimieren . . . . .	8
2.6.2	Net 2 Net . . . . .	11
2.6.3	Kernel rescaling . . . . .	11
2.6.4	Resource Aware Layer Replacement . . . . .	11
2.7	Weitere Herangehensweisen . . . . .	11
2.7.1	Tree CNN . . . . .	11
2.7.2	Standardization Loss . . . . .	11
2.7.3	Wavelet . . . . .	11
<b>3</b>	<b>Experimente – Arbeitstitel</b>	<b>13</b>
3.1	Experimentales Setup . . . . .	13
3.1.1	Hardware . . . . .	13

3.1.2	Wahl des Frameworks . . . . .	13
3.2	Untersuchung von PruneTrain . . . . .	13
3.2.1	verwendete Netzarchitektur . . . . .	13
3.2.2	Implementierung der Anpassung der Batch Größe . . . . .	14
3.2.3	Einfluss der Batchgrösse und der Lernrate auf die Verkleine- rung des Netzes . . . . .	16
3.2.4	Nachvollziehbarkeit der PruneTrain Ergebnisse . . . . .	16
3.3	PruneTrain als MorphNet . . . . .	17
3.3.1	Net 2 Net . . . . .	17
3.3.2	Morphnet . . . . .	17
3.4	Überblick über die möglichen Strategien . . . . .	18
3.4.1	Zahlenformate . . . . .	18
3.4.2	Beschleunigung der Berechnung des Gradientenabstiegverfahren	19
3.4.3	Verfahren um weniger Trainingsdaten zu verwenden . . . . .	20
3.4.4	Lernen von Struktur und Stärke von CNNs . . . . .	20
3.5	Schnelleres MorphPruneTrain . . . . .	20
<b>4</b>	<b>Evaluation</b>	<b>21</b>
<b>I</b>	<b>Additional information</b>	<b>23</b>
	<b>Abbildungsverzeichnis</b>	<b>25</b>
	<b>Algorithmenverzeichnis</b>	<b>27</b>
	<b>Quellcodeverzeichnis</b>	<b>29</b>
	<b>Literaturverzeichnis</b>	<b>31</b>

# Todo list

<input checked="" type="checkbox"/>	Einleitung fertig schreiben – zum Schluss . . . . .	1
<input checked="" type="checkbox"/>	Aufbau der Arbeit – erst bei fortgeschrittener Arbeit schreiben . . . . .	1
<input checked="" type="checkbox"/>	verwendete Suchbegriffe . . . . .	1
<input checked="" type="checkbox"/>	cite . . . . .	6
<input checked="" type="checkbox"/>	subnormale Zahlen . . . . .	6
<input checked="" type="checkbox"/>	ref . . . . .	7
<input checked="" type="checkbox"/>	ref . . . . .	7
	Figure: Schema . . . . .	7
<input checked="" type="checkbox"/>	Bottleneck -Eigenschaft . . . . .	14
<input checked="" type="checkbox"/>	Quelle . . . . .	14
<input checked="" type="checkbox"/>	Wie gut fittet die Geraden die gegebenen Punkte . . . . .	15
<input checked="" type="checkbox"/>	Überprüfe, ob dies auch bei einem anderen Datensatz funktioniert. Es sollte funktionieren, wenn die Grösse eines anderen Datensatzes ins Verhältnis zu der Grösse in Cifar10 gesetzt wird . . . . .	15
<input type="checkbox"/>	Untersuche, ob largeBatch auch auf ein PruneTrain Netzwerk anwendbar ist.	16
<input checked="" type="checkbox"/>	Untersuche, ob die Grösse des Batches beeinflusst, wie viel vom Netz ge- prunt wird . . . . .	16
<input checked="" type="checkbox"/>	Nachvollziehen der PruneTrain Ergebnisse mit dieser Implementierung . . .	16
<input checked="" type="checkbox"/>	Vergleiche Hardware . . . . .	17
<input checked="" type="checkbox"/>	Zeige wie Net2Net aus einem Netzwerk ein tieferes oder breiteres Netz macht	17
<input type="checkbox"/>	Vollziehe mit PrunTrain + Net2Net nach, ob dies funktioniert wie MorphNet	17
<input checked="" type="checkbox"/>	Mache das Netz tiefer . . . . .	17
<input type="checkbox"/>	Suche Kriterien, die entscheiden ob das Netz tiefer sein sollte . . . . .	17
<input type="checkbox"/>	Entwickle Möglichkeit dies direkter da anzuwenden, wo nicht geprunt wurde	17
<input checked="" type="checkbox"/>	Weitere Versuche, die zeigen ob die Zeiten grossen statistischen Schwan- kungen unterliegen. . . . .	19
<input type="checkbox"/>	Testen ob es funktioniert . . . . .	20
<input type="checkbox"/>	Implementieren (ist einfach) und testen . . . . .	20
<input type="checkbox"/>	Testen, wenn die anderen funktionieren . . . . .	20

	Testen . . . . .	20
	Ist zwar interessant aber ohne Code dazu wohl zu aufwändig . . . . .	20
	Funktioniert erst, wenn die anderen Experimente durchgelaufen sind . . . .	20

# Mathematical Notation

Notation	Meaning
$\mathbb{N}$	Set of natural numbers $1, 2, 3, \dots$
$\mathbb{R}$	Set of real numbers
$\mathbb{R}^d$	$d$ -dimensional space
$\mathcal{M} = \{m_1, \dots, m_N\}$	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

MorphNet ist eine Möglichkeit mittels Verkleinern und Vergrössern des Netzwerkes effizient die Struktur eines Netzwerks zu lernen. Ist dieses Konzept auch auf PruneTrain zu übertragen und so zu erweitern, dass nicht das komplette Netz erweitert wird, sondern eben nur sinnvolle Bereiche? Ist dieser Prozess mit Hilfe weiterer Methoden beschleunigbar?

Einleitung fertig schreiben – zum Schluss

## 1.2 Aufbau der Arbeit

Aufbau der Arbeit – erst bei fortgeschrittener Arbeit schreiben

## 1.3 Suchbegriffe

verwendete Suchbegriffe



## 2 Stand der Wissenschaft

Dieses 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)<sup>1</sup>“ 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.

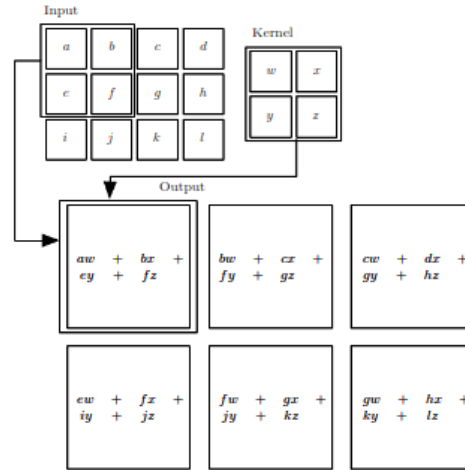
Dies bedeutet, dass die Eingabedaten für ein CNN für diese Faltung geeignet sein müssen. Geeignet für die Faltung sind Eingabedaten, die gridförmig angeordnet sind. Bilddaten sind ein grosser Anwendungsbereich für CNNs.

Bei der Faltung wird auf die Eingabedaten bzw. die Daten der 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.

---

<sup>1</sup>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]

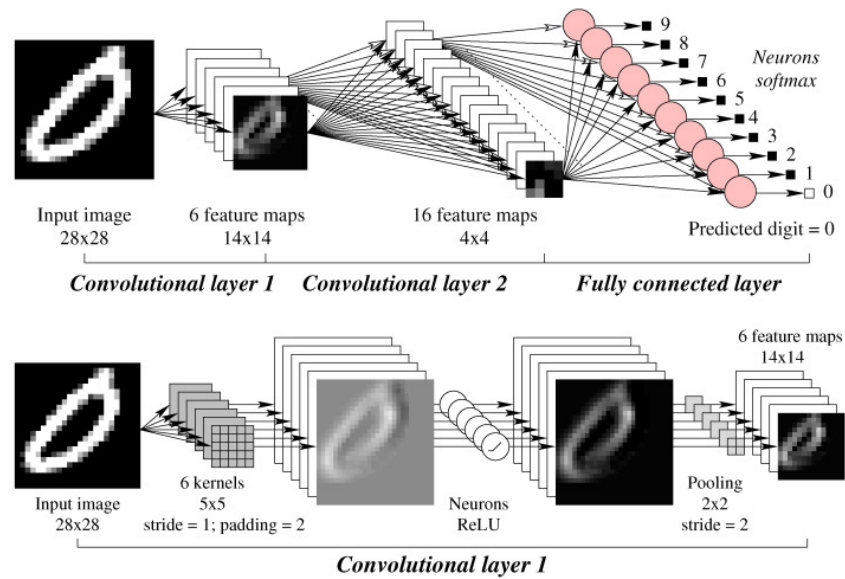
Mehrere dieser Kernel bilden zusammen ein Teil des Convolutional Layer. Dabei wird der Eingang des Layers wie in Abbildung 2.2 gezeigt auf jeden Kernel mittels der Faltung angewendet. Durch diese Faltung entstehen Erste Feature-Maps. Diese Feature-Maps werden im nächsten Schritt Komponentenweise als Eingabe für eine Aktivierungsfunktion benutzt. In Abbildung 2.2 wird ReLU als Aktivierungsfunktion benutzt<sup>2</sup>. Um Overfitting zu vermeiden kann nach dem Anwenden der Aktivierungsfunktion eine Pooling Operation eingeführt werden. Pooling verkleinert die Größe der Feature-Map verkleinert.

Der Begriff Padding aus Abbildung 2.2 enthält einen Wert, der aussagt ob und wenn ja wieviele Pixel um das eigentlich Bild gelegt werden. Dies geschieht, um dem Kernel die Möglichkeit zu geben die Pixel am Rand des Bildes(bzw. der Featuremap der vorherigen Schicht) in mehreren Teilbildern zu verarbeiten.

Beim Anwenden des Kernels auf die Eingabe kann jedes Teilbild oder weniger Teilbilder verwendet werden. Dies wird über den Parameter Stride kommuniziert. Beim Stride von Eins wird jedes Teilbild verwendet. Wird der Stride auf 2 gesetzt, so wird nach jedem verwendetem Teilbild eines ausgesetzt.

In einem CNN werden mehrere dieser Convolutional Layer hintereinander geschaltet, um komplexe Features erkennen zu können.

<sup>2</sup>Für Erklärung Relu siehe [Hay98]



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

Eine beispielhafte Übersicht über die CNN-Architektur ist in Abbildung 2.2 zu sehen. 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 ResNet – eine neuere CNN-Architektur

### 2.3 Strukturelle Methoden

In diesem Unterkapitel wird ein Überblick über die bisher erforschten Methoden des schnelleren Trainierens und des Effizienten Architekturverbesserns gegeben. In Paragraph 2.4.1 wird die Verwendung von anderen Zahlenformaten zur Zeiteinsparung beim Training gegeben. In Paragraph xxx wird erläutert, wie sich das Gradientenabstiegsverfahren beschleunigen lässt.

### 2.3.1 MorphNet

## 2.4 Zeitsparende Methoden

### 2.4.1 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 verwendeten 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 unbedingt 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	$2^{-24} \approx 5.96 \cdot 10^{-8}$	$2^{-149}$

Subnormale Zahlen 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 abgemildert oder sogar komplett aufgehoben werden:

- 32 Bit Mastergewichte und Updates
- Skalierung 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 Durchlauf des Netzes die Gewichte abzuwerten 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 vielen Fällen das Produkt der beiden Zahlen gleich Null. Dies liegt an der Tatsache, dass wie in Tabelle zu sehen ist die kleinste darstellbare Zahl in 16 Bit wesentlich grösser ist als in 32 Bit.

ref

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 einer Schicht mit gemischt präzisen Gleitkommazahlen benutzt.

ref



## 2.4.2 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

**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**

## 2.5 Verfahren um weniger Trainingsdaten zu verwenden

### 2.5.1 Stochastisches Pooling

### 2.5.2 Lernen von Struktur und Stärke von CNNs

## 2.6 Strukturelle Veränderungen zur Beschleunigung des Trainings

### 2.6.1 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 einzelnen 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 rekonfiguriert 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 Speicherkapazitätsgebrauchs 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. Vorherige 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. Pruning 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 Layetrts.

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.

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

- $\frac{1}{N} \sum_{i=1}^N l(y_i, f(x_i, W))$  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 sollter 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:

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

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 der einzige zusätzliche Hyperparameter. Zu gross gewählt würde der Intervallparameter zu wenig Zeitverbesserung bringen. Zu klein gewählt könnte er die Lernqualität beeinflussen. 4 Matrizen zur Evaluierung: Training und Inference FLOPs, gemessenen Trainingszeit, und Validierungsaccuracy.

### **The Lottery Ticket Hypothesis**

#### **2.6.2 Net 2 Net**

#### **2.6.3 Kernel rescaling**

#### **2.6.4 Resource Aware Layer Replacement**

## **2.7 Weitere Herangehensweisen**

### **2.7.1 Tree CNN**

### **2.7.2 Standardization Loss**

### **2.7.3 Wavelet**



## 3 Experimente – Arbeitstitel

Zunächst PruneTrain untersuchen.

Dann PruneTrain als MorphNet

Untersuchen welche Methoden aus 2.3 überhaupt sinnvoll auf grössere Datensätze anwendbar sind und funktionieren.

Die sinnvollen Methoden auf PruneTrain als Morphnet anwenden.

### 3.1 Experimentales Setup

#### 3.1.1 Hardware

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

#### 3.1.2 Wahl des Frameworks

Es wird mit pytorch gearbeitet, da pytorch gegenüber anderen Frameworks eine grössere Flexibilität erlaubt. Ausserdem ist eine fast vollständige Implementierung von PruneTrain in Pytorch geschrieben. Diese wird im nächsten Kapitel untersucht und soweit erweitert, dass es dem Stand im PruneTrain Paper entspricht.

### 3.2 Untersuchung von PruneTrain

#### 3.2.1 verwendete Netzarchitektur

Die PruneTrain Implementierung hat initial mehrere verschiedene Netzarchitekturen zur Auswahl:

- AlexNet
- ResNet 32/50

- vgg 8/11/13/16
- mobilenet

Schränke diese Auswahl auf ResNet ein. Gründe hierfür:

- Da die Überlegung besteht diese Netze tiefer zu machen wähle ResNet, da die Identity-Übergänge dem Netz erlauben das degradation Problem zu umgehen während das Netz noch tiefer/ breiter wird.
- Festlegung auf eine Architektur um Umfang der Arbeit zu begrenzen

Erweitere dies jedoch durch beliebige grosse ResNets. Ein ResNet ist hier durch 4 Parameter charakterisiert:

- $s$ : Anzahl an Stages, die das ResNet hat
- $[n]$ : Anzahl von Blöcken pro Stage
- $l$ : Anzahl von (Conv+Batch)-Layer pro Block
- $b$ : Boolean Parameter, der angibt ob die Blöcke im Netz die Bottleneck-Eigenschaft haben

$$\text{ResNet-Zahl} = s \cdot n \cdot l$$

Bottleneck -Eigenschaft

### 3.2.2 Implementierung der Anpassung der Batch Größe

Um die Anpassung der Batchgröße an die Verkleinerung des Netzwerkes durch das Prunen zu implementieren muss zunächst die Batchgröße des Ausgangsnetzes so gewählt werden, dass der GPU-Speicher maximal ausgelastet ist. Dies ist eine der Techniken, die bei PruneTrain für eine schnellere Trainingszeit sorgen.

Theoretisch sollte hierfür nachdem Übertragen des Modells der freie Speicher ausgelesen werden und anhand des Speicherverbrauchs eines Elements des Datensatzes berechnet werden, wie gross die Batchgröße maximal sein darf. Leider führt diese Methode nicht zum gewünschten Ergebnis. Der Grund hierfür liegt in der Speicherarchitektur von CUDA, welche ähnlich wie das Speichersystem von CPUs in Seiten organisiert ist

Quelle

	s=1		s=2		s=3		s=4	
	#Para	Batch	#Para	Batch	#Para	Batch	#Para	Batch
1	7642	14272	31034	5856	123898	2704	493946	1344
2	14650	8816	65882	3328	269722	1472	1082906	688
3	21658	6512	100730	2368	415546	1024	1671866	480
4	28666	5072	135578	1808	561370	784	2260826	368
5	35674	4208	170426	1488	707194	624	2840786	288
6	42682	3568	205274	1232	853018	528	3438746	240
7	49690	3120	240122	1072	998842	464	4027706	208
8	56698	2736	274970	944	1144666	400	4616666	176
9	63706	2464	309818	848	1290490	352	5205626	160
10	70714	2224	344666	752	1436314	320	5794586	145
11	77722	2049	379514	688	1582138	288		

. Daraus ergibt sich das Problem, dass der ausgelesene freie Speicher nicht dem tatsächlichen allokierbaren Speicher entspricht.

Diese Problem wird mit einer Methode, die für einen beliebigen Datensatz und für eine beliebige Modellgrösse die maximale Batchgröße berechnet, gelöst. Hierfür wird für einige verschiedene Modellgrößen durch eine binäre Suche die maximale Batch Size ermittelt. Gleichzeitig wird für die jeweilige Modellgrösse die Anzahl an Parametern, die das Modell hat gezählt. Diese Größen sind in Tabelle ?? eingetragen. Mit Hilfe dieser Größen wird für jede einzelne Stagegröße eine Gerade gefittet.

Wie gut fittet die Geraden die gegebenen Punkte

Überprüfe, ob dies auch bei einem anderen Datensatz funktioniert. Es sollte funktionieren, wenn die Grösse eines anderen Datensatzes ins Verhältnis zu der Grösse in Cifar10 gesetzt wird

Bei Cifar100 ist zu sehen, dass die gleiche maximale Batchgrösse verwendet werden kann. Bei gleicher Grösse der Bilder (32x32 Pixel) ändert sich hier nur die Anzahl an Parametern in der letzten Schicht.

Es stellt sich die Frage, ob das einen so grossen Einfluss auf die Ausführungszeit hat. Man sieht, dass mit steigender Batchgröße die Ausführungszeit sinkt.

Errechne zusätzlich noch ein Modell, wo abhängig von der Modellgrösse währenddem Pruning die Batchgrösse angepasst wird.

[HHS17] gibt an, dass mit grösserer Batch size die Accuracy weniger wird. Aber dort wird ein Verfahren angegeben, welches diesen Effekt entfernen kann. Da dieser Effekt da sehr deutlich gezeigt wird hier im nächsten Unterkapitel nur die Überprüfung, ob dieser Effekt auf bei geprunten Netzwerken funktioniert.

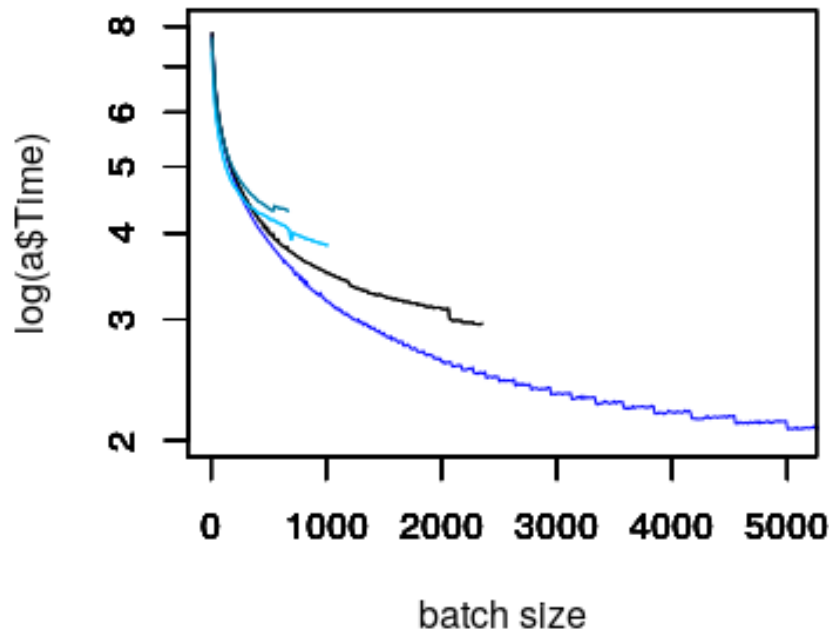


Abbildung 3.1: Batch Size vs Trainings Time über eine Epoche

### 3.2.3 Einfluss der Batchgrösse und der Lernrate auf die Verkleinerung des Netzes

Untersuche, ob largeBatch auch auf ein PruneTrain Netzwerk anwendbar ist.

Untersuche, ob die Grösse des Batches beeinflusst, wie viel vom Netz geprunt wird

### 3.2.4 Nachvollziehbarkeit der PruneTrain Ergebnisse

Nachvollziehen der PruneTrain Ergebnisse mit dieser Implementierung

Evaluation auf Cifar10, Cifar100 und auf ImageNet.

Als Netze werden Resnet32, Resnet 50 aus den Resnets verwendet.

Es werden mehrere GPUs gemeinsam benutzt. Hier nur eine einzelne 2080.

Zunächst wird überprüft weiviel Trainingflop, Trainingszeit und andere grössen durch PruneTrain ohne Anpassen der Batchgrösse gespart werden können.

Danach wird für Resnet50 und ImageNet verglichen, wie viel zusätzlich durch das



vergrössern der Batch size gewonnen werden kann.

Als nächster Schritt wird dies für Resnet50 und Cifar100 verglichen. Hier wird allerdings nicht der gesamte Speicher der Grafikkarte ausgefüllt sondern nur auf einen gleichmässigen Speicherverbrauch geachtet im Verlauf der Batch Size Anpassung. Vllt. kann hier Large Batch benutzt werden, um hier trotzdem den ganzen Speicher zu nutzen.

Als nächsten Schritt wird untersucht wie das Intervall wie häufig rekonfiguriert wird den Zusammenhang zwischen Inferenz Flop und der Validation Accuracy verändert. Die nächste Untersuchung über das Sparen von Kommunikationskosten beim Verteilten Training macht hier keinen Sinn da nur eine einzelne Graka genutzt wird.

Abschliessend wird noch evaluiert, wie die Dichte der Gewichte mit der Dichte der Kanäle nachdem Training zusammenhängen um eventuell durch spezifische Inferenzhardware weiter zusparsen.

Für das Cifar Training wird im PruneTrain Paper eine Titan XP Gpu verwendet.

Vergleiche Hardware

## 3.3 PruneTrain als MorphNet

### 3.3.1 Net 2 Net

Zeige wie Net2Net aus einem Netzwerk ein tieferes oder breiteres Netz macht

### 3.3.2 Morphnet

MorphNet macht alle Layer breiter um sie dann mit einem speziellen Regularisierer breiter zu machen. Dieser Regularisierer hat verschiedene mögliche Zielgrössen (Modelgrösse, Flops oder Inferenz-Zeit). Die Frage stellt sich hier, ob das Netz besser wird wenn alle Schichten breiter gemacht werden um später wieder geprunt zuwerden.

Vollziehe mit PrunTrain + Net2Net nach, ob dies funktioniert wie MorphNet

Weiterhin besteht die Möglichkeit das Netz nicht nur breiter zu machen sondern auch tiefer.

Mache das Netz tiefer

Suche Kriterien, die entscheiden ob das Netz tiefer sein sollte

MorphNet erwähnt, dass es Sinn macht nicht im ganzen Netz denn Wider Operator anzuwenden sonder nur da wo der Regularisierer das Netz nicht schmaller macht.

Entwickle Möglichkeit dies direkter da anzuwenden, wo nicht geprunt wurde

## 3.4 Überblick über die möglichen Strategien

Welche Strategien aus Kapitel 2 sind überhaupt durchführbar? Hier werden nur die Strategien aufgeführt, welche überhaupt auf vernünftig grossen Datensätzen funktionieren und von der Technik und dem Aufwand her möglich sind. Die Strategien sind aufgeteilt in Unterkapitel.

Alle möglichen Kombinationen von Strategien sind zuviele. Daher sinnvolle Vorauswahl treffen. Bei mehreren gleichartigen/ konkurrierenden Ansätze direkter Vergleich und dann den besten auswählen.

### 3.4.1 Zahlenformate

- FP16 bereits probiert

FP16 nur auf RTX 2080 sinnvoll Bietet nach erster Messung etwa 28 % Prozent Gewinn.

Code für dieses Verfahren liegt vor: Amp apex von Nvidia

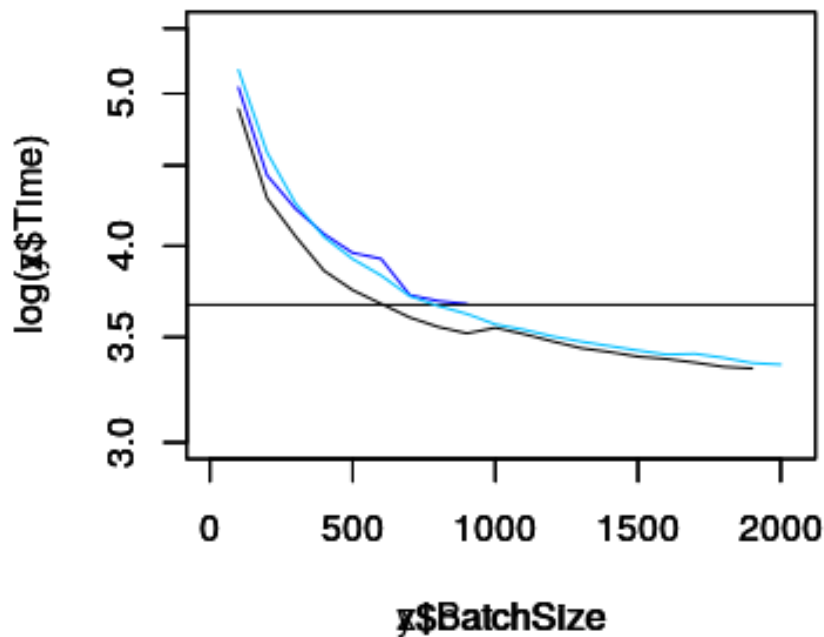
AMP bietet 3 mögliche Optimierungsstufen:

O1 Patch all Torch functions and Tensor methods to cast their inputs according to a whitelist-blacklist model. Whitelist ops (for example, Tensor Core-friendly ops like GEMMs and convolutions) are performed in FP16. Blacklist ops that benefit from FP32 precision (for example, softmax) are performed in FP32. O1 also uses dynamic loss scaling, unless overridden.

O2 casts the model weights to FP16, patches the models forward method to cast input data to FP16, keeps batchnorms in FP32, maintains FP32 master weights, updates the optimizer's paramgroups so that the optimizer.step() acts directly on the FP32 weights (followed by FP32 master weight-FP16 model weight copies if necessary), and implements dynamic loss scaling (unless overridden). Unlike O1, O2 does not patch Torch functions or Tensor methods.

O3 may not achieve the stability of the true mixed precision options O1 and O2. However, it can be useful to establish a speed baseline for your model, against which the performance of O1 and O2 can be compared. If your model uses batch normalization, to establish speed of light you can try O3 with the additional property override keepBatchnormfp32=True (which enables cudnn batchnorm, as stated earlier).

Hier nur O0, O1 und O2 dargestellt, da O3 absolut nicht mithalten kann was Performance angeht.



**Abbildung 3.2:** Vergleich Trainingszeit einer Epoche für verschiedene Optimierungsstufen von Amp Apex. DunkelBlau=O0; Schwarz = O1; Hellblau=O2

Weitere Versuche, die zeigen ob die Zeiten grossen statistischen Schwankungen unterliegen.

<https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9998-automatic-mixed-precision-in-pytorch.pdf> zeigt, dass bezüglich der Accuracy kein Verlust zu erwarten ist.

Da O2 gegenüber O1 keinen signifikanten zusätzlichen Gewinn bringt nutze O1.

### 3.4.2 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

Testen ob es funktioniert

Könnte funktionieren. Code für Lasagne: [https://github.com/TimSalimans/weight\\_norm](https://github.com/TimSalimans/weight_norm)

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

Interessant bisher kein Code verfügbar

Implementieren (ist einfach) und testen

## Accelerated CNN Training Through Gradient Approximation

Interessant bisher kein Code verfügbar

## Sind diese Verfahren theoretisch kombinierbar

Testen, wenn die anderen funktionieren

## 3.4.3 Verfahren um weniger Trainingsdaten zu verwenden

### Stochastisches Pooling

Klingt sehr interessant und könnte für deutlich kleinere Trainingsdatenmenge sorgen Oder alternativ bei gleicher Trainingsdatenmenge die Accuracy verbessern <https://github.com/Shuangfei/s3pool>

Testen

## 3.4.4 Lernen von Struktur und Stärke von CNNs

bisher kein Code verfügbar. Klingt aber interessant

Ist zwar interessant aber ohne Code dazu wohl zu aufwändig

## 3.5 Schnelleres MorphPruneTrain

FP16 + plus ein Verfahren aus dem Bereich Gradienten

Funktioniert erst, wenn die anderen Experimente durchgelaufen sind

## **4 Evaluation**



**Teil I**

**Additional information**





# Abbildungsverzeichnis

2.1	Abbildung zur Faltung [GBC16]	4
2.2	Convolutional Neural Net [CCGS16]	5
3.1	Batch Size vs Trainings Time über eine Epoche	16
3.2	Vergleich Trainingszeit einer Epoche für verschiedene Optimierungsstufen von Amp Apex. DunkelBlau=O0; Schwarz = O1; Hellblau=O2	19



# **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.
- [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.