

Masterarbeit

Zeit-effizientes Training von Convolutional Neural Networks

Jessica Buehler
29. September 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	Ziel der Arbeit	2
1.3	Ergebnisse der Arbeit	2
1.4	Aufbau der Arbeit	2
2	Stand der Wissenschaft	5
2.1	Funktionsweise eines CNN	5
2.2	ResNet – eine neuere CNN-Architektur	9
2.3	Vorgehen zur Suche nach dem Stand der Wissenschaft	11
2.4	Beschneidung des Netzes zur Beschleunigung des Trainings	13
2.5	Beschleunigung des Lernens durch Wissenstransfer	17
2.6	Automatische Architektursuche	19
2.7	Schnelles Ressourcen-beschränktes Strukturlernen tiefer Netzwerke	20
2.7.1	Definition der Nebenbedingung	22
2.7.2	Regularisierer	23
2.8	Additive Methoden	24
2.8.1	Verringerung der für Berechnungen nötige Zeit	24
2.8.2	Beschleunigung der Berechnung des Gradientenabstiegsverfahren	26
2.8.3	Verfahren zum Verwenden maximaler Batchgrößen	26
3	Überblick über die Arbeit	29
3.1	Experimentelles Setup	29
3.1.1	Hardware	29
3.1.2	Wahl des Frameworks	29
3.1.3	verwendete Netzarchitektur	30
3.1.4	Baseline Netz	31
3.2	Konzept	34

4	Untersuchung von MorphNet	37
4.1	Evaluierung der einzelnen Schritte von MorphNet	37
4.2	Evaluierung der Ergebnisse von MorphNet	39
5	Evaluation des Beschneidens des Netzes	41
5.1	Evaluation bei gleichbleibender Batchgröße	41
5.2	Experimente zur Anpassung der Batchgröße beim Beschneiden des Netzes	45
5.2.1	Berechnung der Batchgröße abhängig vom Speicherverbrauch	45
5.2.2	Evaluierung der Anpassung der Batchgröße an die Netzgröße	48
6	Evaluierung von Net2Net	51
6.1	Evaluierung des Operators für ein breiteres Netz	51
6.1.1	Verbreitern aller Phasen	52
6.2	Evaluierung des Operators für ein tieferes Netz	52
6.3	Erkunden des Modellraums	53
7	Evaluierung der Kombination von PruneTrain und Net2Net	55
8	Vergleich	57
9	Ausblick und Fazit	59
A d		61
	Abbildungsverzeichnis	64
	Literaturverzeichnis	65

Todo list

■ Was ist an dieser Formel nicht in Ordnung?	18
■ Um diese Zahl einordnen zu können: die besten zehn Accuracy-Werte liegen für Cifar 10 bei 96.62 % bis 99.37 % und für Cifar 100 bei 82,35 % bis 93,51 %. https://benchmarks.ai/	20
■ Überarbeiten, fertig schreiben; 3 Stunden	24
■ cite	24
■ subnormale Zahlen	25
■ ref	25
■ Überblick schreiben; 4 Stunden	26
■ Über Lars schreiben; 3 Stunden	26
■ nur wenn noch Zeit übrig: In Kapitel 8 werden die Ergebnisse aus 6.3 auf einem anderen Datensatz verifiziert. auch wenn noch Zeit: Prüfe wie sich die additiven Verfahren auf 6.3 auswirken Zuletzt werden noch additive Verfahren vorgetellt, welche die Trainingszeit zusätzlich mi- nimieren können. Eine dieser Verfahren, welches in Kapitel evaluiert wird, spart Zeit durch die Verwendung von gemischt präzisen Zah- lenformaten. Ein weiteres additives Verfahren in Kapitel überprüft in wiefern mit Hilfe einer adaptiven Anpassung der Lernrate die Batch- größe sinnvoll so angepasst werden kann, dass die ganze GPU genutzt werden kann.	36
■ Hier Algorithmus nochmal hinschreiben?	37
■ Die Effekte in der Accuracy vom Baseline Netz zum Rekonfigurations- intervall sind wieder mit einem Overfitting zu erklären. Die Effekte vom Rekonfigurationsintervall 2 zu 5 und zu 10 sind auch mit Zuneh- menden Overfitting je weniger geprunt wird zu erklären. Abhilfe mehr Experimente und zwar mit dem schmalen Baseline Netz	43
■ Ergebnisse	53
■ blau markieren	53

Mathematische Notation

Notation	Bedeutung
\mathbf{x}_i	Eingabe in das Netz
\mathfrak{B}	Menge der Batches des Datensatzes $\mathfrak{B} = \{\mathcal{B}_1, \dots, \mathcal{B}_k\}$
\mathcal{B}	ein Batch mit m Elemente $\mathcal{B} = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$
$\mathbf{u}_{i,j}$	Ausgabe der Faltungsschicht j wobei in das Netz \mathbf{x}_i eingegeben wurde.
$\mathbf{v}_{i,j}$	Ausgabe der Batchnormalisierungsschicht j wobei in das Netz \mathbf{x}_i eingegeben wurde.
\mathcal{W}	Gewichte der Schichten $\mathcal{W} = \{W_1, \dots, W_J\}$ geordnet nach Stelle der Schicht im Vorwärtsthroughgang.
\mathcal{W}^k	Gewichte des k -ten Trainingsdurchlaufs
$f(\mathbf{x}_i, \mathcal{W})$	Funktion, die das CNN berechnet. $f(\mathbf{x}_i, \mathcal{W})$ berechnet eine Klassenzugehörigkeit für \mathbf{x}_i
y_i	tatsächliche Klasse von \mathbf{x}_i
$l(f(\mathbf{x}_i, \mathcal{W}), y_i)$	Funktion, die das CNN berechnet

1 Einleitung

1.1 Motivation und Hintergrund dieser Arbeit

Wird die Zeit-Effizienz eines Convolutional Neural Networks betrachtet, so ist die Betrachtung mehrerer Größen möglich. Im Rahmen dieser Arbeit soll dafür die Zeit betrachtet werden, die bei einem neuen Datensatz aufgebracht werden muss, bis eine gute Netzarchitektur und die passenden Hyperparameter gefunden sind. Um diese Zeit zu verkürzen sind zwei Herangehensweisen möglich. Die erste Herangehensweise beschleunigt und automatisiert den Prozess des Findens der besten Hyperparameter, in dem das Netz nach Änderungen von bestimmten Hyperparametern weiterverwendet wird. Für dieses Weitertrainieren werden zwei Operatoren verwendet, die das Netz breiter und/oder tiefer machen können. Da ein größeres bzw. breiteres Netz anfälliger für Overfitting ist wird dem Vergrößern des Netzes ein Beschneidungsalgorithmus entgegengesetzt, der unwichtige Gewichte entfernt. Das Entfernen unwichtiger Gewichte funktioniert hier mit Hilfe eines Regularisierers, der auch ganze Kanäle entfernen kann, falls diese unwichtig für den Ausgang des Netzes sind.

Diese Technik des Vergrößern und wieder Kleiner machen wird einer ähnlichen Technik gegenübergestellt, welche das komplette Netz mit Hilfe eines Breitenmultiplikators breiter macht und mit Hilfe eines Regularisierers wieder schmaller aber nicht kleiner macht.

Dabei ist das in dieser Arbeit vorgestellte Verfahren flexibler und kann das Netz zielgerichteter vergrößern beziehungsweise tiefer machen.

Eine weitere zusätzliche Herangehensweise ist das Beschleunigen einzelner Trainingsdurchläufe, in dem das Training einer einzelnen Epoche beschleunigt wird. Zu dieser Herangehensweise werden Techniken vorgestellt, welche aber nicht praktisch evaluiert werden.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, dieses Verfahren mit einem weiteren zu vergleichen.

Diese Technik des Vergrössern und wieder Kleiner machen wird einer ähnlichen Technik gegenübergestellt, welche das komplette Netz mit Hilfe eines Breitenmultiplikators breiter macht und mit Hilfe eines Regularisierers wieder schmaller aber nicht kleiner macht.

Dabei ist das in dieser Arbeit vorgestellte Verfahren flexibler und kann das Netz zielgerichteter vergrössern beziehungsweise tiefer machen.

Die Evaluation soll ergeben, welches der beiden vorgestellten Verfahren bei einem Datensatz besser ein geeignetes Netz findet.

1.3 Ergebnisse der Arbeit

1.4 Aufbau der Arbeit

In Kapitel 2 wird zunächst der aktuelle Stand der Wissenschaft erläutert. Zu diesem Zweck werden zunächst in Unterkapitel 2.1 die Grundlagen und Funktionsweisen eines CNNs erklärt. Die in dieser Arbeit verwendete CNN-Architektur ResNet wird darauf aufbauend in Unterkapitel 2.2 beschrieben. In Kapitel 2.3 wird eine Bibliometrie zum Thema der Arbeit vorgestellt. Dies ermöglicht dem Leser nachzuvollziehen wie das Fundament auf dem diese Arbeit beruht gefunden wurde.

In Kapitel 2.4 wird das Beschneiden des Netzes vorgestellt, welches das Netz von unwichtigen Gewichten befreien soll und so das Wachstum des Netzes in Grenzen halten soll. In Kapitel 2.5 werden die Operatoren vorgestellt, welche das Netz breiter beziehungsweise tiefer machen sollen.

In Kapitel 2.6 wird ein Überblick über das Thema der automatischen Architektursuche gegeben, um das Strukturlernen des Netzes im Kontext seiner Forschungsrichtung zu betrachten. In Kapitel 2.7 wird die Vergleichsmethode vorgestellt, welche mit Hilfe von Strukturlernen versucht eine bessere Netzstruktur zu finden.

Zeitsparen: ??

Additive Methoden 2.8

Experimente:3

Setup der Experimente 3.1

PruneTrain Experimente: 5

Net2Net Experimente: 6
MorphNet: 4
Net2Net + PruneTrain: 7
Evaluation: ??
Fazit: 9

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 eines Convolutional Neural Networks (CNNs) erläutert. Im zweiten Teil des Kapitels wird ein Überblick über die bisherigen wissenschaftlichen Erkenntnisse im Themenbereich dieser Arbeit vorgeteilt.

2.1 Funktionsweise eines CNN

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 selbst 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 muss. Geeignet für die Faltung sind Eingabedaten, die gridförmig angeordnet sind. Bilddaten sind ein grosser Anwendungsbereich für CNNs.

Bei der Faltung wird auf die Eingabedaten beziehungsweise 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

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

entsprechende Schicht eine Merkmalskarte.

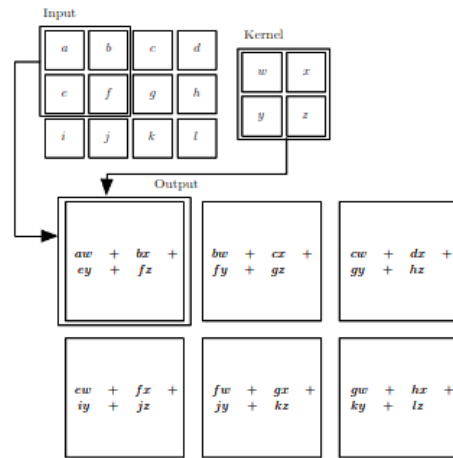


Abbildung 2.1: Abbildung zur Faltung [GBC16]

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 Merkmalskarten. Diese Merkmalskarten werden im nächsten Schritt komponentenweise als Eingabe für eine Aktivierungsfunktion benutzt. In Abbildung 2.2 wird eine Rectified Linear Unit (ReLU) als Aktivierungsfunktion benutzt².

Um die Größe der Merkmalskarte zu reduzieren kann nach dem Anwenden der Aktivierungsfunktion eine Pooling Operation eingeführt werden. Durch die Verkleinerung der Merkmalskarte wird es weniger wichtig wo genau ein Merkmal in den Daten auftaucht. Für ein Feld in der Ausgabe der Pooling Operation ist der Bereich, der von diesem Feld wahrgenommen wird grösser als ohne die Pooling Operation. Ein Nebeneffekt des Poolings ist die Vermeidung beziehungsweise Verringerung von Overfitting.

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 der Eingabe kann jedes Teilbild genutzt werden oder es können Teilbilder ausgelassen werden. Dies wird über den Parameter Stride kommuniziert. Beim Stride von Eins wird jedes Teilbild verwendet. Wird der Stride auf Zwei gesetzt, so wird nach jedem verwendetem Teilbild eines ausgelassen.

²Für Erklärung ReLU siehe [Hay98]

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

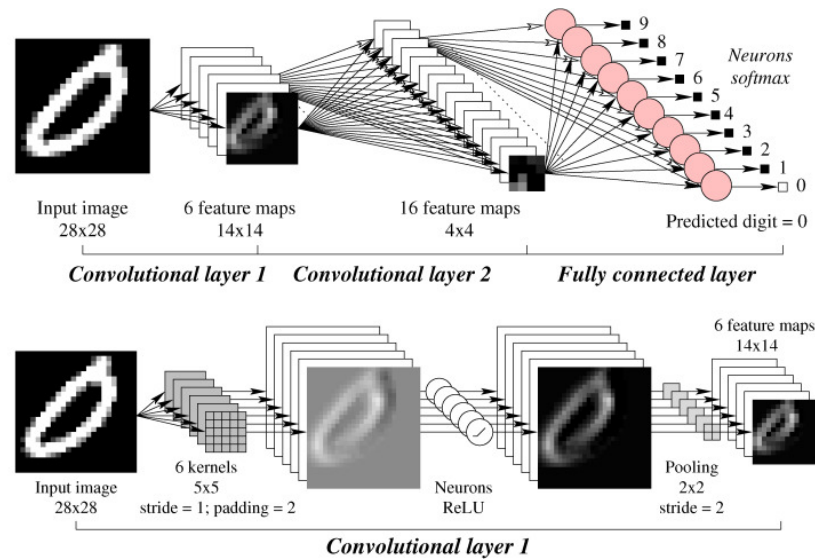


Abbildung 2.2: Convolutional Neural Net [CCGS16]

Eine beispielhafte Übersicht über die CNN-Architektur ist in Abbildung 2.2 zu sehen.

Die voll verbundene Schicht errechnet 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 der voll verbundenen Schicht trainierbar. Das heißt, durch den Trainingsprozess wird versucht die Werte in der Filtermatrix und der voll verbundenen Schicht so zu verändern, dass das gesamte CNN besser klassifizieren kann.

Die Trainingsdaten sind Daten aus dem Datensatz, die bereits klassifiziert sind. Diese Trainingsdaten werden in Batches aufgeteilt. Der Trainingsprozess beginnt mit der aufeinanderfolgenden Eingabe der Bilder \mathbf{x}_i eines Batches von Trainingsdaten in die erste Schicht. In jeder Schicht des Netzes wird mit der Eingabe aus der vorherigen Schicht weitergerechnet. Die Ausgabe einer Faltungsschicht wird $\mathbf{u}_{i,j}$ angegeben, wobei i für den Platz im jeweiligen Batch steht und j für die Nummer der entsprechenden Schicht. In der letzten, voll verbundenen Schicht ist das Ergebnis eines einzelnen Bildes die Klasse, die durch die aktuellen Belegung der Gewichte des Netzes klassifiziert wird. Die Gewichte der Schichten werden mit W_j bezeichnet, wobei j die Schicht der Gewichte angibt. Alle Gewichte des

Netzwerkes werden mit \mathcal{W} bezeichnet, wobei

$$\mathcal{W} = \{W^1, \dots, W^J\} \quad (2.1)$$

die Definition dieser Menge ist.

Diese Klassifikation ist formal eine Funktion f mit der Eingabe \mathbf{x}_i . Da das Bild \mathbf{x}_i bereits vorher klassifiziert wurde hat es ein Label y_i , welches die Klasse von \mathbf{x}_i angibt. Mit Hilfe des Labels und der Ausgabe von $f(\mathbf{x}_i, \mathcal{W})$ wird eine Verlust-Funktion $l(f(\mathbf{x}_i, \mathcal{W}), y_i)$ berechnet. Die Verlust-Funktion berechnet wie weit die tatsächliche Klasse y_i von der Ausgabe des Netzes $f(\mathbf{x}_i, \mathcal{W})$ entfernt ist. Wie wäre es mit: Die Fehlerrate Trainingsdaten bei dieser Verlustfunktion wird Trainingsfehler genannt.

Die Ableitung dieser Verlust-Funktion wird rückwärts durch das Netz propagiert und damit ein Gradient berechnet. Mittels des Gradientenabstiegsverfahrens wird die Verlust-Funktion minimiert, was dazu führt, dass das Netz die Trainingsbilder besser klassifiziert.

Im Anschluss an diesen Trainingsprozess können Bilder, die ohne zugehöriges Label in das Netz eingegeben werden, klassifiziert werden. Um die Klassifikationsleistung für unbekannte, nicht im Trainingsprozess benutzte Bilder zu testen, wird eine Menge an diesen Bildern durch das Netz klassifiziert und die Fehlerrate gemessen. Dieser Fehler wird Test-Fehler genannt.

Mit Hilfe des Trainings- und Testfehlers lässt sich die Klassifikationsleistung des Netzes beurteilen. Sind beide Fehlerarten hoch, so muss das Netz entweder noch weiter trainieren oder an der Struktur beziehungsweise den Hyperparametern muss etwas geändert werden. Ist jedoch nur der Testfehler hoch, so ist die Generalisierungsfähigkeit des Netzes nicht gut.

Eine weitere Technik, die zur Verbesserung der Generalisierungsfähigkeit führen kann ist die Batchnormalisierung. Diese Technik wird zum Ende dieses Unterkapitels betrachtet [IS15]. Beim Training eines CNNs ändert sich die Verteilung der Eingabewerte während des Trainings durch Veränderung der Gewichte der vorherigen Schicht. Dies führt zu einem langsameren Training, da es kleinere Lernraten hat und damit mehr Durchläufe braucht damit das Netz konvergiert. Dieses Phänomen wird interne Kovarianzverschiebung genannt und durch eine Normalisierung gelöst. In Algorithmus 2.1 ist zu sehen, wie dies schrittweise geschieht.

Zunächst bekommt die Batchnormalisierungsschicht die Eingabewerte $u_{i,j}$, wobei j die Nummer der entsprechenden Schicht angibt und i angibt welches Element

in der jeweiligen Batch \mathcal{B} gemeint ist. Zunächst wird aus allen Elementen des aktuellen Batches \mathcal{B} der Mittelwert $\mu_{i,\mathcal{B}}$ berechnet. Mit Hilfe dieses Wertes wird im nächsten Schritt die Varianz des Batches \mathcal{B} berechnet. Diese beiden Werte werden benutzt, um Elemente des Batches \mathcal{B} so zu normalisieren, dass der Batch einen Mittelwert von Null und eine Varianz von Eins hat. Der in dieser Formel aufgeführte Wert ϵ wird zur quadratischen Varianz addiert um zusätzliche numerische Stabilität zu gewähren.

Im Anschluss an diese Normalisierung lassen sich die Elemente des Batches durch eine weitere mit trainierbaren Gewichten versehenen Transformation verändern. Da diese Parameter γ und β durch diese Transformation Teil des Modells sind und stetig differenzierbar sind, lassen sie sich in den Trainingsprozess integrieren.

Eingabe: Werte von $\mathbf{u}_{i,j}$, j -te Schicht und i -tes Element der Menge der Trainingsdaten eines Batches \mathcal{B} , zu trainierende Parameter γ, β

Ausgabe: $\{\mathbf{v}_{(i,j)} = \text{BN}_{\gamma,\beta}(\mathbf{u}_{i,j})\}$

$$\mu_{i,\mathcal{B}} \leftarrow \frac{1}{m} \sum_{j=1}^m \mathbf{u}_{i,j}$$

$$\sigma_{i,\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{j=1}^m (\mathbf{u}_{i,j} - \mu_{i,\mathcal{B}})^2$$

$$\hat{\mathbf{u}}_{i,j} \leftarrow \frac{\mathbf{u}_{i,j} - \mu_{i,\mathcal{B}}}{\sqrt{\sigma_{i,\mathcal{B}}^2 + \epsilon}}$$

$$\mathbf{v}_{i,j} \leftarrow \gamma \hat{\mathbf{u}}_{i,j} + \beta = \text{BN}_{\gamma,\beta}(\mathbf{u}_{i,j})$$

Algorithmus 2.1: Batchnormalisierungs-Algorithmus

2.2 ResNet – eine neuere CNN-Architektur

Die wachsende Tiefe bei CNN-Architekturen geschieht mit dem Hintergrund, dass tiefere Netze größere Modellkomplexität haben. Die klassische CNN-Architektur mit hintereinander geschalteten Conv-Layern schafft es bei wachsender Tiefe des Netzes nicht, diese Komplexität in bessere Klassifikationsleistung umzusetzen. Die Quelle zu diesem Unterkapitel ist das Paper, welches wegweisend für die

Verwendung von Residualen Netzen in der Wissenschaft ist [HZRS15].

Neuere CNN-Architekturen schaffen es, dieses Problem zu vermeiden. Eine dieser neueren Architekturen ist das ResNet. Das ResNet ist ein Residualnetz, welches Kurzschlussverbindungen einführt. In Abbildung 2.3 ist zu sehen wie eine Kurzschlussverbindung aussieht. Durch die Kurzschlussverbindungen in den einzelnen Blöcken ist es für das Netzwerk einfacher, Funktionsbestandteile, die ähnlich einer Identitätsfunktion sind, zu erlernen.

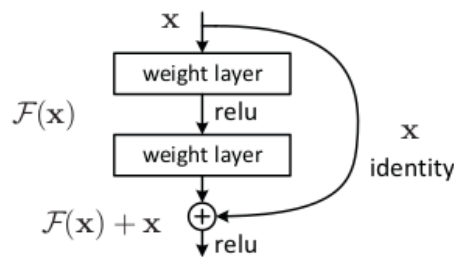


Abbildung 2.3: Abbildung der Kurzschlussverbindung [HZRS15]

Vermieden wird damit im Vergleich zu einem klassischen CNN das Problem des verschwindenden Gradienten. Bei einem klassischen CNN wird mit der letzten Schicht begonnen und der Gradient wird durch die Kettenregel bis zur ersten Schicht berechnet. Je tiefer das Netz wird, desto kleiner werden die Veränderungen des Gradienten für die ersten Schichten. Die Gewichte konvergieren dann sehr langsam bzw. teilweise gar nicht mehr in die gewünschte Richtung.

Residuale Netze vermeiden dies, indem sie aus vielen kleineren Netzen bestehen. Hier wird der Gradient nicht auf einer Linie zur Eingangsschicht zurück propagiert, sondern auch über die Kurzschlussverbindungen. So entsteht ein Netz, welches sehr viel tiefer sein kann ohne die Probleme des verschwindenden Gradienten zu haben. Durch den Wegfall dieses Problems lassen sich mit residualen Netzen bessere Trainingsfehler und Testfehlerraten erreichen.

Eine weitere Technik, die in residualen Netzen verwendet wird, ist die der Bottleneck-Blocks. Dies resultiert aus dem Problem der stark steigenden Trainingszeiten, je breiter die Blöcke sind. Ein Bottleneck-Block ist in Abbildung 2.4a abgebildet.

Die erste Schicht im Bottleneck-Block reduziert dabei die Größe der Feature-Map. Dies hat zur Folge, dass die Durchlaufzeit des mittleren Convolutional Layers geringer ist als bei einem äquivalenten Nicht-Bottleneck-Block. Der letzte Layer des Blockes stellt die Größe vor dem Block wieder her.

Ein von der Zeitkomplexität ähnlicher Block wie der Block in Abbildung 2.4a ist in Abbildung 2.4b zusehen. Wird in einem residualen Netz mit 34 Schichten

aus Blöcken wie in Abbildung 2.4b jeder Block durch einen Block wie in Abbildung 2.4a ausgetauscht, so entsteht ein 50- Layer residual Netzwerk mit einer durchschnittlich erhöhten Accuracy.

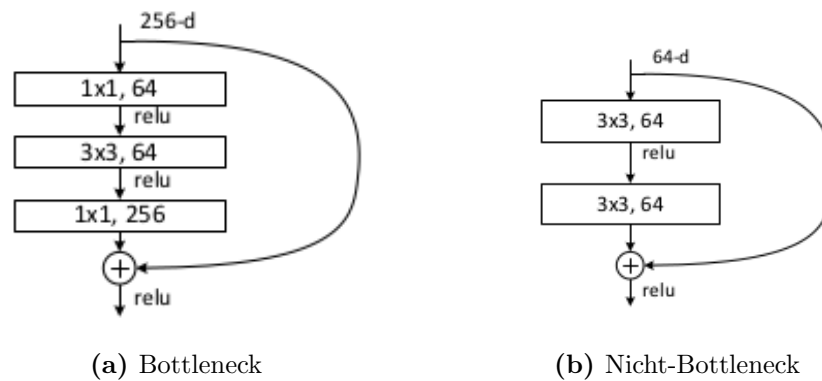


Abbildung 2.4: Vergleich zweier Residual-Netz-Blöcke [HZRS15]

2.3 Vorgehen zur Suche nach dem Stand der Wissenschaft

Eine Google-Suche nach “time efficient training convolutional neural networks” ergibt ungefähr 12 Millionen Suchergebnisse. Mit dieser Flut an Ergebnissen und vielen populär-wissenschaftlichen Einträgen ist die Suche nicht erfolgreich. Aus diesem Grund wird die Suche auf die Seite arxiv.org eingeschränkt. Diese Einschränkung macht Sinn mit dem Hintergrund, dass bereits 2017 über 60% Prozent der publizierten Paper auf arxiv.org als Preprint veröffentlicht wurden [SG17]. Diese Zahl ist seitdem weiter gestiegen, was die Zahl der veröffentlichten Paper im Bereich Machine Learning pro Tag in Abbildung 2.5 zeigt.

Auch mit der auf arxiv.org eingeschränkten Suche ist die Menge an wissenschaftlichen Veröffentlichungen weiterhin zu groß für eine einzelne wissenschaftliche Arbeit. Zunächst wird eine Vorauswahl anhand des Themas der Arbeit getroffen. Es fallen alle Veröffentlichungen weg, die auf anderen Ausführungsplattformen als GPUs arbeiten. Aufgrund des schnellen Forschungsfortschritts und der Hardware sowie Softwareentwicklung liegt der Fokus auf Veröffentlichungen nach 2016.

Die nach diesen Einschränkungen gefundenen Paper sind in einer Mindmap in Abbildung 2.6 zu sehen. Mit blauer Schrift werden die Suchbegriffe dargestellt. Die einzelnen, aufgrund dieser Suchbegriffe gefundenen Paper werden mit grüner

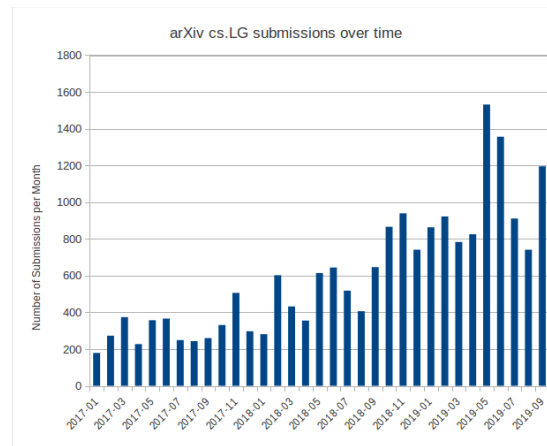


Abbildung 2.5: Tägliche Submissionen der Kategorie Machine Learning auf arxiv [oA19]

Schrift angezeigt. Mit roter Schrift werden die Paper dargestellt, die durch das Paper der vorherigen Ebene zitiert werden. Gelb hinterlegt sind Paper, die das Paper auf der vorherigen Ebene zitieren. In den weiteren Unterkapiteln werden die so gefundenen Paper vorgestellt und die verwendeten Methoden erklärt.

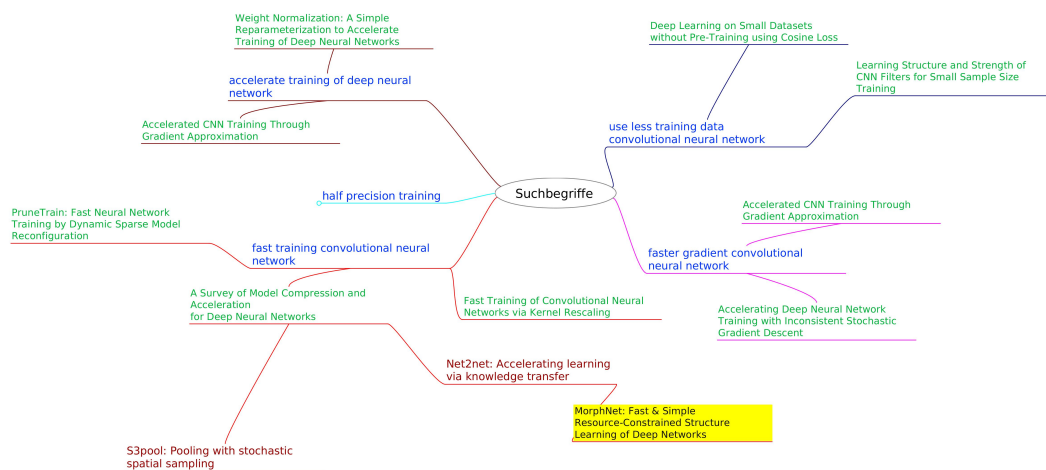


Abbildung 2.6: Mindmap zu den Suchbegriffen bezüglich des aktuellen wissenschaftlichen Stands

2.4 Beschneidung des Netzes zur Beschleunigung des Trainings

Das Beschneiden³ des Netzes ist eine Technik, die entwickelt wurde um die Inferenzzeit eines neuronalen Netzwerks zu reduzieren. Das Beschneidungsverfahren wird auf das bereits trainierte Netz angewendet. Dabei wird entschieden, welche Gewichte nur einen minimalen oder keinen Effekt auf das Klassifikationsergebnis haben, um diese zu entfernen.

Das Beschneiden des Netzes kann auch verwendet werden um die Trainingszeit zu minimieren. Diese Methode soll hier in einem Unterkapitel erläutert werden. Als Quelle für das Unterkapitel dient ein Paper, welches evaluiert, inwiefern Trainingszeit mittels Beschneiden gespart werden kann [LCZ⁺19].

Das Ziel des Beschneidens während des Trainings ist es, die Gewichte einzelner Kanäle auf Null zu setzen und zu entfernen, um mit einem kleinerem Netz in den nachfolgenden Epochen Trainingszeit zu sparen. Dazu wird zu der Verlust-Funktion des Netzwerks ein Normalisierungsterm addiert. Damit die Gewichte ganzer Kanäle möglichst unter den Schwellwert fallen, werden die Gewichte der Kanäle gemeinsam quadriert, wie in der folgenden Gleichung zu sehen ist:

$$GL(W) = \sum_{j=1}^J \left(\sum_{c_j=1}^{C_j} \|W_j(c_j, :, :, :)\|_2^2 + \sum_{k_j=1}^{K_j} \|W_j(:, k_j, :, :)\|_2^2 \right) \quad (2.2)$$

Dieser Term nennt sich Gruppen-Lasso. Der Parameter W_j stellt die Gewichte im CNN als Tensor dar. Mit j wird dargestellt, um welche Schicht es sich handelt. Die Dimensionen des Tensors sind: Ausgangskanäle \times Eingangskanäle \times Kerneldimension 1 \times Kerneldimension 2. J gibt an, über wie viele Layer der Gruppen-Lasso-Term berechnet wird. k_j ist die Laufvariable über die einzelnen Eingangskanäle und c_j über die einzelnen Ausgangskanäle. Alternativ zum Gruppen-Lasso-Regularisierer könnten hier auch andere Regularisierer, wie L1- bzw. L2-Regularisierer verwendet werden. Der Vorteil des Gruppen-Lasso-Regularisierers ist, das durch das gemeinsame Quadrieren der Gewichte einer Schicht diese gemeinsam minimiert werden.

Um das Verhältnis von Gruppen-Lasso-Term zur Verlust-Funktion dynamischer wählen zu können, werden diese nicht einfach miteinander addiert. Es wird abhängig von der Initialbelegung der Gewichte ein Parameter λ berechnet, der Gruppen-

³Beschneiden wird hier äquivalent zum Englischen „to prune“ verwendet

Lasso und Verlust-Funktion balanciert:

$$LPR(GL(\mathcal{W}), l(f(\mathbf{x}_i, \mathcal{W}), y_i)) = \frac{\lambda \cdot GL(\mathcal{W})}{l(f(\mathbf{x}_i, \mathcal{W}), y_i) + \lambda \cdot GL(\mathcal{W})} \quad (2.3)$$

Die Größe LPR ist hier zwischen Null und Eins wählbar. Je größer sie gewählt wird, desto größer ist der Anteil, der beschnitten wird. Regelmäßig werden während des Trainierens des Netzes Gewichte, die unter dem Schwellwert liegen auf Null gesetzt. Es entsteht ein nur dünn besetztes Netz. Dann wird durch ein Rekonfigurationsverfahren aus dem dünn besetzten Netz ein dicht besetztes Netz ohne die vorher nicht besetzten Kanäle. Um dieses Verfahren durchzuführen muss überprüft werden, ob mit dem Entfernen der Kanäle die Dimensionen der verschiedenen aufeinanderfolgenden Kanäle übereinstimmen. Bei einem residualen Netz muss zusätzlich darauf geachtet werden, dass die Dimensionen der Kurzschluss-Verbindungen zusammen passen.

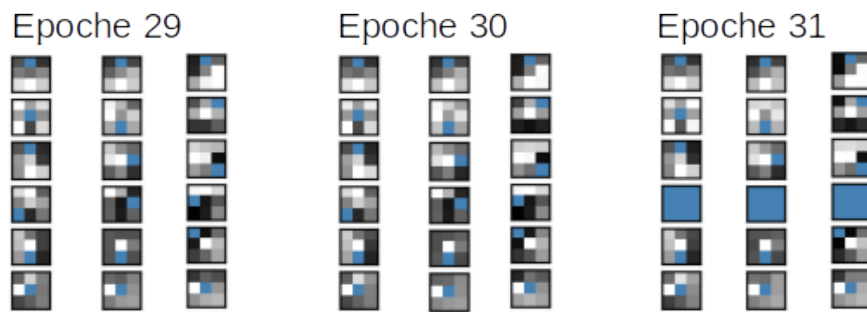


Abbildung 2.7: Beispielhafte Darstellung des Kanal-Union-Verfahrens

Zu diesem Zweck wird das Kanal-Union-Verfahren eingeführt. Beim Kanal-Union-Verfahren wird eine Liste der Layer geführt, die aufeinander abgestimmt werden müssen. Im Falle eines residualen Netzes muss zusätzlich eine Liste der zusammengehörigen Layer der Kurzschlussverbindungen geführt werden. Im nächsten Schritt werden alle Eingangs- und Ausgangskanäle, die noch Gewichte größer Null haben in einer Liste gesammelt. Mit allen Elementen dieser Liste wird nun geprüft, ob mit Hilfe von Vereinigungen Kanäle gefunden werden können, die zwar keine von Null verschiedenen Gewichte mehr haben, wegen der Dimensionalität aber trotzdem beibehalten werden müssen. Alle Kanäle die nicht unter diese Bedingung fallen, können mit Hilfe einer Rekonfiguration aus dem Netzwerk entfernt werden. In Abbildung 2.7 sind beispielhaft drei Eingangs- und sechs Ausgangskanäle dargestellt. In jedem Element des kartesischen Produkts der Menge der Eingangs- und Ausgangskanäle ist jeweils ein drei mal drei Felder großer Kernel

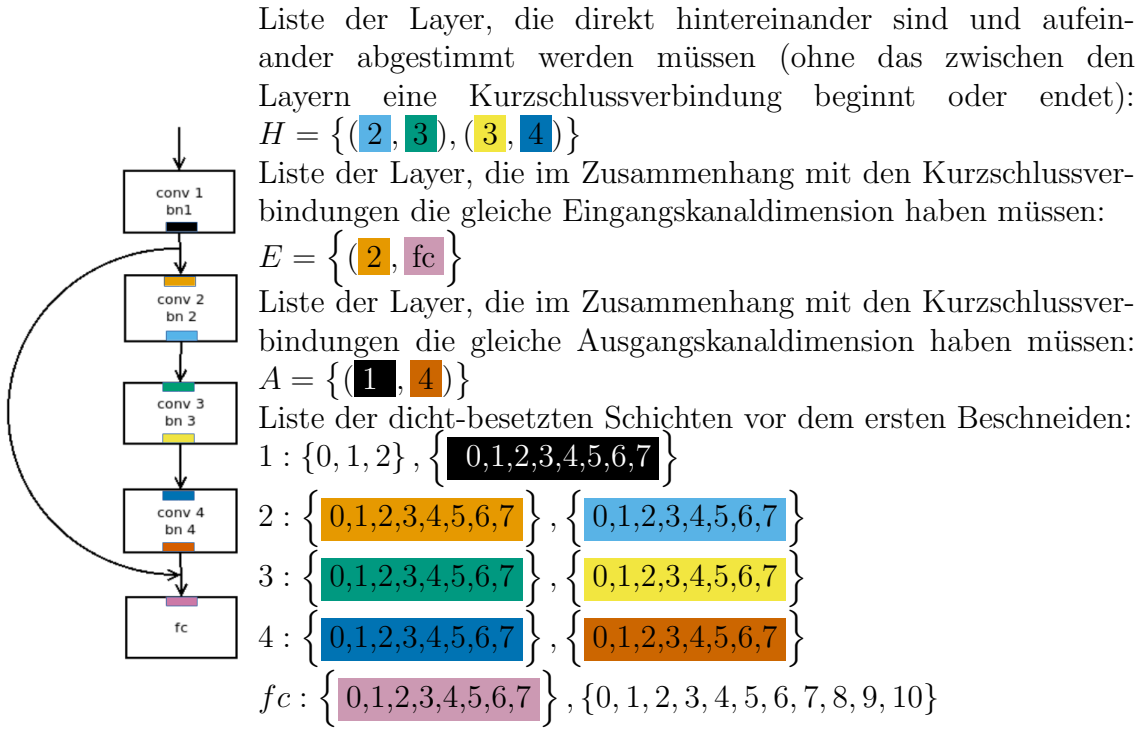
dargestellt. Die Werte der Gewichte sind blau markiert, sobald sie absolut kleiner als der gewählte Grenzwert sind. Je dunkler die nicht-blauen Gewichte sind, desto kleiner sind sie. Da zwischen den jeweiligen Epochen für jede Batch eine Anpassung der Gewichte durchgeführt wird, entstehen teilweise große Veränderungen zwischen den Epochen. Dies ist zum Beispiel von Epoche 30 zu Epoche 31 im vierten Ausgangskanal sichtbar. Hier fällt innerhalb einer Epoche der Großteil des Ausgangskanal unter den Grenzwert.

Bei einem residualen Netzwerk kann weiterhin ein ganzer Block wegfallen. In diesem Fall müssen die Kanal-Union-Listen angepasst werden, die weitere Aktion wird ohne diesen Block im um mehrere Schichten verkürzten Netzwerk weiter geführt

Da mit dem Verkleinern des Netzes nicht nur potentiell Zeit sondern auch Speicherplatz gespart wird, kann bei gleicher Speicherauslastung die Batchgröße erhöht werden. Da die verwendete Technik für die Erhöhung der Batchgröße in der Quelle nicht angegeben ist und in der verwendeten Implementierung fehlt, wurde diese nachimplementiert. Dies wird in Kapitel ?? erläutert [ptI]. Hierbei wird die Lernrate an die erhöhte Batchgröße angepasst um negative Effekte für die Accuracy abzumildern oder auszuschließen.

Damit lassen sich Netzverkleinerungsraten von etwa 50 % erreichen bei weniger als 2 % Accuracy-Verlust auf dem Datensatz Cifar10. Andere Techniken schaffen zwar zwischen 70 - 80 % Netzverkleinerungsraten brauchen jedoch wesentlich mehr Trainingszeit [FC19]. Diese großen Verkleinerungsraten sind dort sehr stark abhängig von der Initialisierung [FC19]. Das heißt, nur einzelne Initialisierungen führen zu so starken Verkleinerungsraten, was insgesamt zu einer längeren Trainingszeit führt [FC19].

Eine weitere Beschneidungstechnik arbeitet vor dem Training des Netzwerkes [TKYG20]. Damit wird das Netz abhängig von der Initialbelegung beschnitten. Es lassen sich zwar sehr große Teile der Parameter auf Null setzen, hierbei wird im Vergleich zur Beschneidungsmethode während des Trainings allerdings weder für gemeinsames Beschneiden von Kanälen gesorgt, noch wird ein Rekonfigurationsverfahren vorgestellt. Somit hat das Netz am Ende des Verfahrens zwar relativ viele auf Null gesetzte Parameter ist aber weder schneller noch kleiner.



Liste der dicht-besetzten (db) Schichten nach dem 'auf-Null-setzen' der Parameter aber vor der Rekonfiguration:

$$1 : \{0, 1, 2\}, \{0, 1, 4, 5, 6, 7\}$$

$$2 : \{0, 1, 3, 5, 6, 7\}, \{0, 1, 2, 4, 5, 6, 7\}$$

$$3 : \{0, 1, 2, 4, 5, 6, 7\}, \{0, 1, 2, 3, 5, 6, 7\}$$

$$4 : \{0, 1, 3, 4, 6, 7\}, \{0, 1, 3, 4, 5, 6, 7\}$$

$$fc : \{0, 1, 3, 4, 5, 6\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

Vorgehen des Kanal-Union-Verfahrens: Als erster Schritt wird für alle Elemente aus H die Vereinigung von Ausgangs- und Eingangskanälen berechnet und diese dann zugewiesen:

$$2, 3 : A(2) \cup E(3) = \{0, 1, 2, 4, 5, 6, 7\} \cup \{0, 1, 2, 4, 5, 6, 7\} = \{0, 1, 2, 4, 5, 6, 7\}$$

Hier wird der dünn-besetzte Kanal 3 entfernt

$$3, 4 : A(3) \cup E(4) = \{0, 1, 2, 3, 5, 6, 7\} \cup \{0, 1, 3, 4, 6, 7\} = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

Der nullwertige Eingangskanal 5 von Schicht 4 wird nicht entfernt, da der dazugehörige Ausgangskanal von Schicht 3 nicht nullwertig ist. Im nächsten Schritt werden für die Elemente an jeweils gleicher Stelle aus den Mengen A und E Vereinigungen gebildet: $A(1) \cup A(4) \cup E(2) \cup E(fc) =$

$$\{0, 1, 4, 5, 6, 7\} \cup \{0, 1, 3, 4, 5, 6, 7\} \cup \{0, 1, 3, 5, 6, 7\} \cup \{0, 1, 3, 4, 5, 6\} = \{0, 1, 3, 4, 5, 6, 7\}$$

Hier wird in den Ausgangskanälen von Schicht 1 und 2 sowie in den Eingangskanälen von Schicht 2 und fc jeweils der 2. Kanal entfernt.

2.5 Beschleunigung des Lernens durch Wissenstransfer

Beim Trainieren eines CNNs kommt es häufig vor, dass nach initialem Wählen der Tiefe beziehungsweise Breite des Netzes diese Parameter in einem weiteren Trainingslauf erhöht werden und in Folge dessen das Netzwerk komplett neu trainiert werden muss. Mit Hilfe der Quelle zu diesem Unterkapitel wurde ein Verfahren geschaffen, welches das Netz tiefer oder breiter machen kann und dabei die im ersten Trainingsdurchlauf trainierten Gewichte weiter verwendet [CGS15]. Durch diesen Wissenstransfer von einem Netz zu einem tieferen oder breiteren Netz wird eine schnellere Konvergenz des neuen Netzes erwartet. Durch die Initialisierung mit schon vorhandenen Parametern entsteht eine Transformation, die die erlernte Funktion erhält.

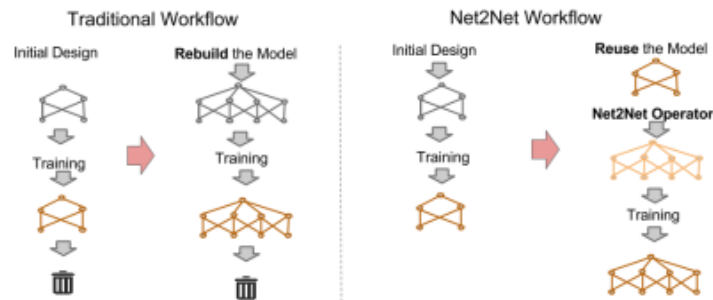


Abbildung 2.8: Traditioneller Workflow vs. Net2Net Workflow

Wie in Abbildung 2.8 abgebildet ist, lässt sich so der Arbeitsablauf zum Finden der passenden Netzstruktur anders gestalten. Der Net2Net-Operator macht hier das Netz entweder breiter (mehr Kanäle in bestimmten Schichten) oder tiefer (zusätzliche Schichten). Diese beiden Operatoren werden nun vorgestellt.

Operator für breiteres Netz

Beim Operator für ein breiteres Netz werden für eine bestimmte Schicht Ausgangskanäle und für die nachfolgende Schicht Eingangskanäle hinzugefügt. Die Schicht, der die Ausgangskanäle hinzugefügt werden, wird mit j bezeichnet und hat den Gewichtstensor \mathbf{W}_j mit der Dimensionalität von $n \times l \times d(h_{l,1}) \times d(h_{l,2})$. Die Schicht, der die Eingangskanäle hinzugefügt werden wird mit $j + 1$ bezeichnet und hat den Gewichtstensor \mathbf{W}_{j+1} mit der Dimensionalität von $m \times n \times d(h_{j+1,1}) \times d(h_{j+1,2})$. Dem Layer j werden q Kanäle hinzugefügt. Dies entspricht wie in Abbildung 2.9 abgebildet ist $q \cdot l$ zusätzlichen Filterkernen.

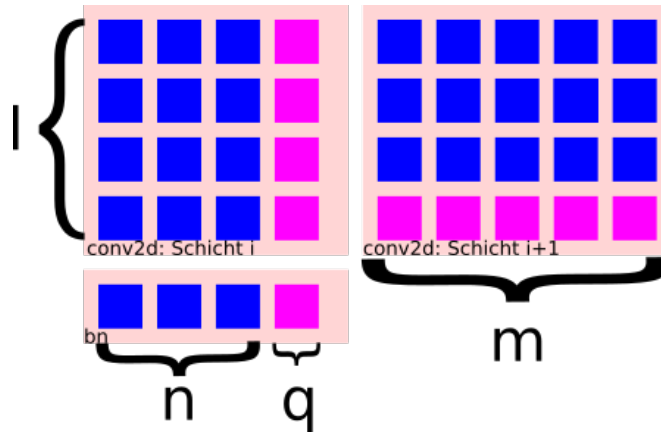


Abbildung 2.9: Übersicht über die zusätzlichen Kanäle

Für den Layer $j + 1$ sind es entsprechend $q \cdot m$ zusätzliche Kernel. Die Gewichtstensoren nach dem Anwenden des Net2Net-Operators werden mit \mathbf{U}^j und \mathbf{U}^{j+1} bezeichnet und sollen die Dimensionalität von $\mathbf{U}^j : (n + q) \times l \times d(h_{(j,1)}) \times d(h_{(j,2)})$ und $\mathbf{U}^{j+1} : m \times (n + q) \times d(h_{(j+1,1)}) \times d(h_{(j+1,2)})$ haben. Der Net2Net-Operator wird angewendet, indem zunächst eine Mapping-Funktion g definiert wird, die für eine zufällige Belegung der zusätzlichen Kernels sorgt:

$$g(j) = \begin{cases} j & , \text{ falls } j \leq n \\ k & , \text{ falls } j > n : k \text{ zufälliges Sample von } \{1, 2, \dots, n\} \end{cases} \quad (2.4)$$

Mit Hilfe dieser Mapping-Funktion werden nun die neuen Gewichtstensoren initialisiert:

Was ist an dieser Formel nicht in Ordnung?

$$\mathbf{U}_j(e, f, h_{j,1}, h_{j,2}) = \mathbf{W}_j(g(e), f, h_{j,1}, h_{j,2})$$

$$\mathbf{U}_{j+1}(e, f, h_{j+1,1}, h_{j+1,2}) = \frac{1}{|\{x | g(x) = g(a)\}|} \mathbf{W}_{j+1}(e, g(f), h_{j+1,1}, h_{j+1,2})$$

Die Funktion $g(j)$ wird dabei für jede neu hinzugekommene Schicht nur einmal ausgewertet, sodass gesamte Reihen statt einzelner Kernel kopiert werden. Sollte sich zwischen dem j -ten und $(j + 1)$ -Layer eine Batchnormalisierungsschicht befinden, so werden die Parameter dieser Schicht ebenfalls kopiert.

Um nicht mehrere exakt gleiche Kernelreihen zu haben kann außerdem noch ein Noiseanteil auf alle Gewichte addiert werden. Dies ist vor allem für den Fall wichtig, wenn der Trainingsalgorithmus keine Form der Randomisierung hat, das heißt

die gleichen Gewichtstensoren werden ermutigt, unterschiedliche Funktionen zu erlernen. Somit sind die vom ursprünglichen und neuen Netz gelernten Funktionen ähnlich aber nicht gleich.

Tieferes Netz

Der Operator für ein tieferes Netz ersetzt die Operation der j -ten Schicht $\mathbf{v}_{i,j} = \text{BN}_{\gamma,\beta}(\mathbf{v}_{i,j} * \mathbf{W}_j)$ durch die Operation von zwei Layern

$$\mathbf{v}_{i,j} = \text{BN}_{\gamma',\beta'}(\text{BN}_{\gamma,\beta}(\mathbf{v}_{i,j} * t(W_j)) * t(U_j)) \quad (2.5)$$

U wird als Identitätsmatrix initialisiert. Da zwischen den beiden Layern eine Batchnormalisierung genutzt wird, müssen die Parameter der Batchnormalisierung γ' und β' so gewählt werden, dass sie die gelernte Funktion des Netzes nicht verändern.

Diskussion der Methode

Die beiden Net2Net-Operatoren schaffen die Möglichkeit, Familien von Netzarchitekturen zu erforschen ohne jedes Mal von neuem zu lernen. Mit Hilfe der beiden Operatoren lässt sich die Komplexität des Netzes erhöhen ohne die gelernte bisherige Funktion zu vernachlässigen.

2.6 Automatische Architektursuche

Neben dem im letzten Kapitel ausführlich erläuterten Ansatz des Strukturlernens gibt es noch einige andere aktuelle Ansätze, die automatisch nach einer besseren Architektur für einen Datensatz suchen. Einige dieser Ansätze werden hier beleuchtet und es wird gezeigt, wieso das im letzten Kapitel erläuterte Verfahren im praktischen Teil weiter verwendet wird.

Beim Versuch die Hyperparameter eines Netzes sinnvoll automatisch zu wählen, entsteht ein sehr großer Suchraum. Dieser Suchraum lässt sich mit viel Aufwand absuchen [MAL⁺19]. Es entsteht ein Optimierungsproblem mit mehreren zu optimierenden Variablen bei welchem eine Pareto-Front gesucht wird [MAL⁺19]. Das Ergebnis schafft eine Verbesserung der Accuracy gegenüber bekannten Architekturen, dabei summiert sich allerdings die Trainingszeit mit 20 NVIDIA V100 Grafikkarten für Imagenet auf 2,5 Tage [MAL⁺19]. Eine weitere Methode den Suchraum zu durchsuchen sind genetisch inspirierte Suchalgorithmen [SXZ⁺20].

Dabei wird initial eine Population von Netzen gebildet [SXZ⁺20]. Nach einem Trainingsdurchgang werden diese nach ihrer Fitness (Klassifikationsleistung) selektiert [SXZ⁺20]. Im weiteren Verlauf werden jeweils zwei dieser Netze gepaart und es entsteht eine neue Generation an Netzen [SXZ⁺20]. Allerdings ist hier die Trainingszeit in einem Rahmen von 35 GPUs für einen Tag [SXZ⁺20]. Dann ist die Architektursuche allerdings komplett automatisiert und erreicht eine Accuracy von 96.78 % für Cifar 10 und 79.77 % für Cifar 100 [SXZ⁺20].

Um diese Zahl einordnen zu können: die besten zehn Accuracy-Werte liegen für Cifar 10 bei 96.62 % bis 99.37 % und für Cifar 100 bei 82,35 % bis 93,51 %. <https://benchmarks.ai/>

Das Ziel von einigen Veröffentlichungen im Themenbereich der automatischen Architektursuche ist es, diese lange Trainingszeit zu reduzieren.

Eine Möglichkeit der Reduzierung bietet sich durch Ausnutzung von domainspezifischen Eigenschaften der zu klassifizierenden Bilder. Eine Möglichkeit einer domainspezifischen Eigenschaft, die genutzt werden kann, ist, wenn die Bilder nicht klassisch mit einer Kamera, sondern mit anderen Geräten aufgenommen wurden. Diese veränderte Aufnahmeart kann dafür sorgen, dass sich der Suchraum massiv einschränken lässt und sich damit die Architektursuche beschleunigt. Als Beispiel kann hier eine Radaranlage zur Aufnahme von Bilder dienen [DZZZ20].

Eine weitere Möglichkeit die Trainingszeit zu minimieren ist es, den Suchraum deutlich zu verkleinern und die Anzahl an Durchläufen zu minimieren. Der Nachteil ist dann allerdings, dass die Wahrscheinlichkeit, ein Netz in einem globalen Optimum zu finden bezüglich des Suchraumes klein ist. Eine Methode die dies nutzt, wird im nächsten Unterkapitel vorgestellt.

2.7 Schnelles Ressourcen-beschränktes Strukturlernen tiefer Netzwerke

Im Gegensatz zu den Kapiteln 2.4 und 2.5, in denen jeweils eine Möglichkeit, ein CNN kleiner sowie größer zu machen vorgestellt wurden, geht es jetzt darum, dies zu kombinieren. Die Quelle für diese Kapitel ist, soweit nicht anders gekennzeichnet, das Paper, welches die Methode vorgestellt hat.

Die manuelle Wahl von Hyperparametern, die bestimmen wie groß und komplex ein neuronales Netz ist, braucht Erfahrung und Kunstfertigkeit. Sind die Hyperparameter falsch gewählt, so müssen diese angepasst und das Netz erneut trainiert

werden. Mit Hilfe der hier vorgestellten Methode wird die Suche nach der besten Architektur automatisiert. Dies geschieht mit Hilfe von iterativen Verkleinern und Vergrößern des Netzes. Diese Methode hat drei Vorteile:

1. Es ist auf große Netze und große Datensätze skalierbar
2. Es kann die Struktur in Bezug auf eine bestimmte Nebenbedingung (zum Beispiel Modellgröße, Anzahl an Parametern) optimieren
3. Es kann eine Struktur lernen, die die Performance erhöht

Das Ziel der Methode ist es, automatisch die beste Architektur für ein Netz zu finden. Dies umfasst die Breiten der Eingangs- und Ausgangskanäle, Größe der Kernel, die Anzahl der Schichten und die Konnektivität dieser Schichten. Im Rahmen dieser Methode wird dies auf die Breite der Ausgangskanäle eingeschränkt. Die Methode kann auf die anderen Größen erweitert werden. Allerdings ist die Einschränkung auf die Breite der Ausgangskanäle sowohl effektiv als auch simpel. Die Breite der Ausgangskanäle für alle J Schichten wird mit $\mathcal{C}_{1:J}$ bezeichnet. Der Anfangspunkt dieser Methode ist ein Netz \mathcal{W}^1 mit einer initialen Breite der Ausgangskanäle sowie fixen Filtergrößen. Die Nebenbedingung wird mit der Funktion \mathcal{F} bezeichnet. Sie optimiert entweder die Modellgröße oder die Anzahl an Flops per Inferenz. Die Methode optimiert formal gesehen also folgendes:

$$\mathcal{W}^* = \arg \min_{\mathcal{F}(\mathcal{C}_{1:J}) \leq \zeta} \min_{\mathcal{W}} l(f(\mathbf{x}_i, \mathcal{W}), y_i) \quad (2.6)$$

Das Vergrößern des Netzes basiert auf einer Lösung für die Gleichung 2.6: dem Breitenmultiplikator ω . Sei $\omega \cdot O_{1:M} = \{\lfloor \omega O_1 \rfloor, \lfloor \omega O_2 \rfloor, \dots, \lfloor \omega O_M \rfloor\}$, $\omega > 0$. Gilt $\omega > 1$, so wird das Netz vergrößert. Bei $\omega < 1$ wird das Netz verkleinert. Um die Gleichung 2.6 zu lösen, finde nun das größte ω , so dass $\mathcal{F}(\omega \cdot O_{1:M}) \leq \zeta$ gilt.

Dieser Ansatz sorgt für eine mögliche Verkleinerung und Vergrößerung des Netzes und er funktioniert gut bei einem guten initialen Netz. Ist das initialen Netz aber nicht von so guter Qualität, so hat dieser Ansatz Probleme. Grund hierfür ist wahrscheinlich ein lokales Minimum, aus welchem die Optimierungsfunktion nicht mehr herausfindet, um ein besseres lokales oder globales Minimum zu finden.

Dieser Nachteil wird durch eine Veränderung der Verlust-Funktion aufgehoben. Es wird ein Regularisierer \mathcal{G} dazu addiert, welcher misst, wie groß der Anteil eines Netzbestandteiles an $\mathcal{F}(\mathcal{C}_{1:J})$ ist, und der es damit direkt optimieren kann. Dann ist

$$\mathcal{W}^* = \arg \min_{\mathcal{F}(\mathcal{C}_{1:J}) \leq \zeta} \min_{\mathcal{W}} l(f(\mathbf{x}_i, \mathcal{W}), y_i) + \lambda \mathcal{G}(\mathcal{W}) \quad (2.7)$$

Dieser Ansatz kann die relative Größe einer Schicht ändern, hat aber den Nachteil das häufiger die zu optimierende Nebenbedingung nicht optimal maximiert wird. Die beiden Ansätze lassen sich kombinieren. Algorithmus 2.2 beschreibt den Algorithmus der bei der Kombination entsteht mit Pseudocode.

- 1: Trainiere das Netz um $\mathcal{W}^* = \underset{\mathcal{W}}{\operatorname{argmin}} l(f(\mathbf{x}_i, \mathcal{W}, y_i) + \lambda \mathcal{G}(\mathcal{W}))$ zu finden
- 2: Finde die neue Breite $\mathcal{C}'_{1:J}$, die durch 1. errechnet wurde
- 3: Finde das größte ω , so dass $\mathcal{F}(\omega \cdot \mathcal{C}_{1:J}) \leq \zeta$ gilt
- 4: Wiederhole ab 1. so häufig wie gewünscht mit $\mathcal{C}_{1:J} = \mathcal{C}'_{1:J}$

Ausgabe: $\omega \cdot \mathcal{C}_{1:J}$

Algorithmus 2.2: MorphNet Algorithmus

Dieser Algorithmus kann so oft durchlaufen werden bis entweder die Performance des Netzes gut genug ist, oder bis die letzten Durchläufe keine Veränderungen mehr hervorgebracht haben.

2.7.1 Definition der Nebenbedingung

Die Nebenbedingung \mathcal{F} lässt sich für verschiedene zu optimierende Zielgrößen definieren. Eine einfache Nebenbedingung, die Modellgröße wird hier beispielhaft erläutert. Die Größe dieser Nebenbedingung wird vor allem durch Schichten mit Matrixmultiplikation dominiert. Die Modellgröße ergibt sich durch die Größe der Tensoren der einzelnen Schichten. Da die Größe der Tensoren der einzelnen Schichten abhängig von der Anzahl der Eingangs- und Ausgangskanäle sowie der Filtergröße und nicht von der Position im Netzwerk ist, lässt sich $\mathcal{F}(\mathcal{C}_{1:J})$ auf die einzelnen Schichten zurückführen. Es gilt:

$$\mathcal{F}(\mathcal{C}_{1:J}) = \sum_{j=1}^J \mathcal{F}(j) \quad (2.8)$$

Für den Breitenmultiplikator ω gilt: $\mathcal{F}(\omega \cdot \mathcal{C}_{1:J}) = \sum_{j=1}^J \omega \cdot \mathcal{F}(j)$

Die Abhängigkeit von der Größe des jeweiligen Tensors ergibt für

$$\mathcal{F}(j) = c_j \cdot k_j \cdot d(h_{j,1}) \cdot d(h_{j,2}) \quad (2.9)$$

Da durch die Anwendung des Regularisierers einzelne Kanäle auf Null gesetzt werden und ein Netz ohne diesen Kanal möglich wäre, sollen diese Kanäle in dieser Berechnung ausgelassen werden. Daher wird die Formel um Aktivierungs-

funktionen $A_{k_l,j}$ und $B_{c_l,j}$ ergänzt die mit einer Eins angeben, dass der zugehörige Kanal nicht null ist. Eine Null als Ergebnis der Aktivierungsfunktion ergibt sich, wenn der entsprechende Kanal komplett auf Null gesetzt wurde. Dadurch lassen sich c_j und k_j aus Formel 2.9 ersetzen:

$$\mathcal{F}(j) = \left(\sum_{k=1}^{k_l} A_{k,j} \right) \cdot \left(\sum_{c=1}^{c_l} B_{c,j} \right) \cdot d(h_{j,1}) \cdot d(h_{j,2}) \quad (2.10)$$

2.7.2 Regularisierer

Beim Verkleinern des Netzes soll die Verlustfunktion l des CNN mit der Nebenbedingung $\mathcal{F}(\mathcal{C}_{1,J}) \leq \zeta$ minimiert werden. Bei der Wahl des Regularisierers muss bedacht werden, dass der Regularisierer und seine Ableitung kontinuierlich definiert sein müssen, da die Parameter im Netz durch ein Gradientenabstiegsverfahren gelernt werden. Zusätzlich kann eine Nebenbedingung nicht direkt durch ein Gradientenabstiegsverfahren gelernt werden. Daher wird \mathcal{F} in veränderter Form als Regulariser gewählt. Die Veränderung umfasst das Hinzufügen von γ , die ähnlich einer Batchnormalisierung genutzt werden:

$$\mathcal{G}(j) = \left(\sum_{k=1}^{k_l-1} A_{k_l,j} \sum_{c=1}^{c_l-1} |\gamma_{c,j}| \right) \cdot \left(\sum_{k=1}^{k_l-1} |\gamma_{k,j}| \sum_{c=1}^{c_l-1} B_{c,j} \right) \cdot d(h_{j,1}) \cdot d(h_{j,2}) \quad (2.11)$$

Mit dieser Funktion lässt sich mittels Gradientenabstieg lernen, obwohl Teile des Regularisierers nicht komplett kontinuierlich sind. γ muss dabei kontinuierlich sein. Werden die γ für einen Kanal auf Null gesetzt durch das Lernen, so ist der dazugehörige Kanal aus der Berechnung wie gewünscht ausgeschlossen. Für jeden Ein- und Ausgangskanal einer Schicht wird ein γ in den Vorwärts-Durchgang eingebaut. Diese Parameter funktionieren dann analog zu den γ aus der Batchnormalisierung, da sie kontrollieren, welcher Prozentsatz eines Kanals weitergeleitet wird.

Aus dem Regularisierer einer Schicht lässt sich mittels Addition die Regularisierung des kompletten Netzes berechnen.

$$\mathcal{G}(\mathcal{W}) = \sum_{j=1}^J \mathcal{G}(j) \quad (2.12)$$

Um die Wichtigkeit vom besseren Training des Netzes und der Regularisierung von Parametern treffen zu können wird ein Parameter λ eingeführt. So entsteht

die Verlust-Funktion

$$\mathcal{W}^* = \underset{\mathcal{W}}{\operatorname{argmin}} \ l(f(\mathbf{x}_i, \mathcal{W}), y_i) + \lambda \mathcal{G}(\mathcal{W}) \quad (2.13)$$

Dieser Regularisierer funktioniert nicht für Netze, die Kurzschlussverbindungen besitzen. Hier wird, wie beim Beschneiden des Netzes während des Trainings, ein Gruppen-Lasso verwendet. Dies stellt sicher, dass an Kurzschlussverbindungen nur so beschnitten werden kann, wie es für die Dimensionalität des Netzes zuträglich ist.

2.8 Additive Methoden

2.8.1 Verringerung der für Berechnungen nötige Zeit

Überarbeiten, fertig schreiben; 3 Stunden

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.

cite

Berechnung mit 16 Bit Gleitkomma [jee85] 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:

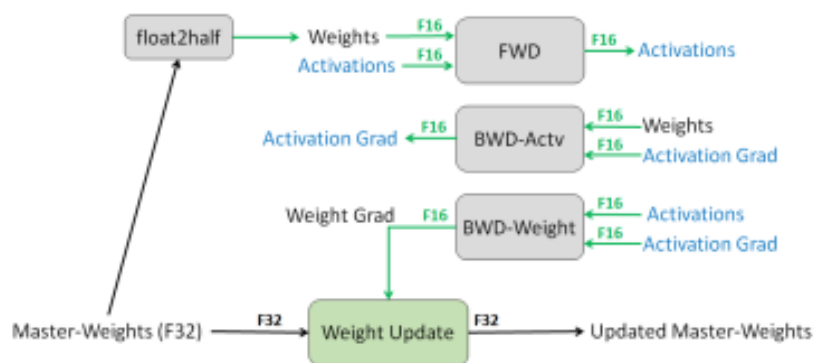
- 32 Bit Mastergewichte und Updates
- Skalierung der Loss-Funktion
- Arithmetische Präzision

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.

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.



2.8.2 Beschleunigung der Berechnung des Gradientenabstiegsverfahren

Überblick schreiben; 4 Stunden

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

2.8.3 Verfahren zum Verwenden maximaler Batchgrößen

Über Lars schreiben; 3 Stunden

Zum Training eines CNN werden die Daten in der Regel in Batches aufgeteilt. Übliche Trainingsverfahren verwenden dabei eine Batchgröße, die nur einen Bruchteil des verfügbaren Grafikspeichers belegt. Dies geschieht, da mit einer maximalen Batchgröße weniger Aktualisierungen der Gewichte pro Trainingsepoche vorgenommen werden. So entsteht beim Training mit maximaler Batchgröße eine verminderte Generalisierungsfähigkeit. Das Ergebnis eines CNNs abhängig von der Batchgröße wird anhand eines Beispieldatensatzes in Abbildung 2.10 gezeigt. Es ist zu beobachten, dass circa ab Epoche 85 eine Rangfolge von hoher Batchgröße hin zu niedriger Batchgröße in Bezug auf die Accuracy erreicht wird. Diese Rangfolge ändert sich auch bis zum Ende des Trainings nicht mehr.

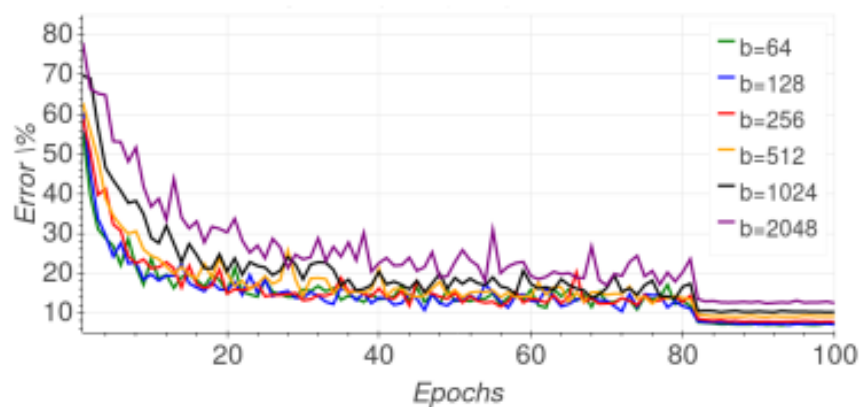


Abbildung 2.10: Validation Accuracy von CNNs anhand eines Beispieldatensatzes

Die Quelle zu diesem Kapitel untersucht dieses Problem und schlägt eine Lösung vor [?]. Für das Training mit kleinen Batchgrößen wird im Allgemeinen ein Stochastisches Gradientenabstiegsverfahren (SGD) benutzt.

Large Batch (ghost batch norm) hat leider nicht funktioniert. Dem Paper scheinen wichtige Details zu fehlen.

Eine alternative ist Lars. (LARGE BATCH TRAINING OF CONVOLUTIONAL NET - WORKS WITH LAYER - WISE ADAPTIVE RATE S CALING)

Dabei wird keine globale Lernrate mehr verwendet, sondern die Lernrate wird an die Größe der Gewichte angepasst.

Funktioniert nach einem ersten auch zusammen mit Prune Train um bei größerer Batchgröße weiterhin trotzdem eine gute Verkleinerungsrate zu bekommen.

3 Überblick über die Arbeit

Im zweiten Teil dieser Arbeit werden die in Kapitel 2 theoretisch betrachteten Methoden praktisch auf einer GPU ausgeführt und evaluiert. Der Überblick über das experimentelle Setup wird in Kapitel 3.1 vorgestellt. In Kapitel 3.2 wird das Konzept des praktischen Teils der Arbeit erläutert.

3.1 Experimentelles Setup

3.1.1 Hardware

Die Hardware umfasst einen Server mit 4 GPUs. Von diesen 4 GPUs haben 2 GPUs jeweils den gleichen Typ:

- Geforce GTX 1080 Ti
- Geforce RTX 2080 Ti

Beide GPU-Typen arbeiten mit der CUDA Version 10.1.

Während der Vorbereitung auf diese Experimente hat sich gezeigt, dass Experimente mit einer Geforce GTX 1080 Ti mit den Experimenten der Geforce RTX 2080 Ti nicht vergleichbar sind. Weiterhin lässt sich durch das Verwenden von gemischt präzisen Zahlen nur auf der Geforce RTX 2080 ein Geschwindigkeitsvorteil beim Training feststellen. Aus diesen zwei Gründen wurden alle Experimente auf der Geforce RTX 2080 Ti ausgeführt.

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. Pytorch bietet mit cudnn und cuda im Hintergrund gute Möglichkeiten die Trainingszeiten einzelner Epochen zu messen und sie so mit einander zu vergleichen.

3.1.3 verwendete Netzarchitektur

Die PruneTrain Implementierung hat initial mehrere verschiedene Netzarchitekturen zur Auswahl:

- AlexNet
- ResNet 32/50
- vgg 8/11/13/16
- mobilenet

Diese Auswahl an Netzarchitekturen ist zu umfangreich, um alle diese Architekturen auf den vorgestellten Methoden zu evaluieren. Daher wird im Rahmen dieser Arbeit nur auf ResNet gearbeitet. Diese Entscheidung liegt daran, dass Resnets durch ihre Kurzschlussverbindungen gut mit sehr tiefen Netzstrukturen umgehen können, ohne grosses Klassifikationsleistungsverluste dank Overfitting. Dies ist vorallem wichtig, wenn das Netz mit Hilfe des Operator für tieferes Netz noch tiefer gemacht werden soll. Die ResNet Struktur wird in der Implementierung so verändert, dass angegeben werden kann wie tief das Netz sein soll. Das ResNet wird hier nicht mehr nur mit einer Zahl identifiziert sondern es wird angegeben, wieviele

- s : Anzahl an Phasen, die das ResNet hat
- $N = [n_1, \dots, n_S]$: Anzahl von Blöcken pro Phase
- l : Anzahl von (Conv+Batch)-Layer pro Block
- $[k_1, \dots, k_S]$: Breite der Schichten je Phase
- b : Boolean Parameter, der angibt ob die Blöcke im Netz die Bottleneck-Eigenschaft haben

das jeweilige ResNet hat. Diese Vorgehensweise hat den Vorteil, dass für ein im Verlauf tieferes beziehungsweise breitere Netz eine Vergleichsmöglichkeit besteht. Dies bedeutet, dass das Netz welches im Verlauf entsteht auch direkt erstellt werden kann.

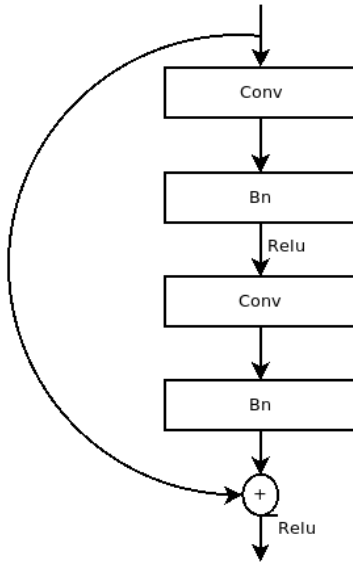


Abbildung 3.1: Basisblock

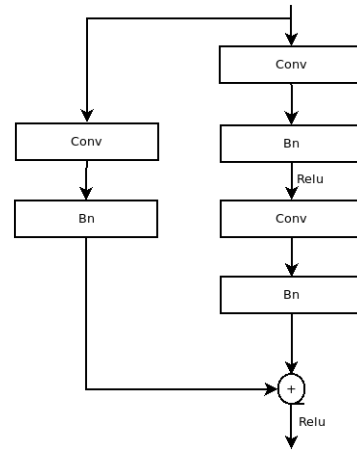


Abbildung 3.2: Übergangsblock

Abbildung 3.3: Grafische Darstellung Basis- und Übergangsblock

3.1.4 Baseline Netz

Um die Ergebnisse der Experimente in den folgenden Kapiteln einschätzen zu können wird ein ResNet, ohne Anpassungen um Trainingszeit zu sparen, trainiert. In Tabelle 3.1 ist die Struktur dieses Netze zu sehen. Das breite Baseline-Netz wird dabei für die Evaluierung des Beschneiden des Netzes verwendet. Das schmallere Baseline-Netz wird für die Evaluierung der Methoden, die das Netz breiter machen verwendet.

Das Netz hat drei Phasen ($s = 3$), wobei jeder der Phasen 5 Blöcke hat ($N = [5, 5, 5]$). Pro Basisblock sind zwei (Conv+Batch)-Schichten vorhanden ($l = 2$). Bei einem Übergangsblock, der als erster Block in einer neuen Phase bei einer Vergrößerung der Bereit beim Phasenübergang genutzt wird ist eine (Conv+Batch)-Schicht mehr vorhanden. Eine grafische Darstellung der Blöcke ist in Abbildung 3.3 zu sehen.

Evaluierung des breiteren Baseline-Netzes

Das Training wird über 180 Epochen durchgeführt. Es werden 5 Experimente durchgeführt. Dabei ergibt sich der in Abbildung 3.4a gezeigte Verlauf der Validierungs-Accuracy für Experiment vier. Bei diesem Training wurde über die gesamten 180 Epochen mit einer Lernrate von 0.1 trainiert. Mit einem Ergebnis von durchschnittlich 81.11 % über fünf Experimente ist diese Ergebnis leider nicht

Phase	Schicht/Block	breites Baseline-Netz		schmales Baseline-Netz	
		#Eingangs- kanäle	#Ausgangs- kanäle	#Eingangs- kanäle	#Ausgangs- kanäle
	Conv 1 + Bn 1	3	16	3	8
1	Basisblock	16	16	8	8
	Basisblock	16	16	8	8
	Basisblock	16	16	8	8
	Basisblock	16	16	8	8
	Basisblock	16	16	8	8
2	Übergangsblock	16	32	8	16
	Basisblock	32	32	16	16
	Basisblock	32	32	16	16
	BasisBlock	32	32	16	16
	BasisBlock	32	32	16	16
3	Übergangsblock	32	64	16	32
	Basisblock	64	64	32	32
	Basisblock	64	64	32	32
	Basisblock	64	64	32	32
	Basisblock	64	64	32	32
	Linear	64	10	32	10

Tabelle 3.1: Struktur des Netzes

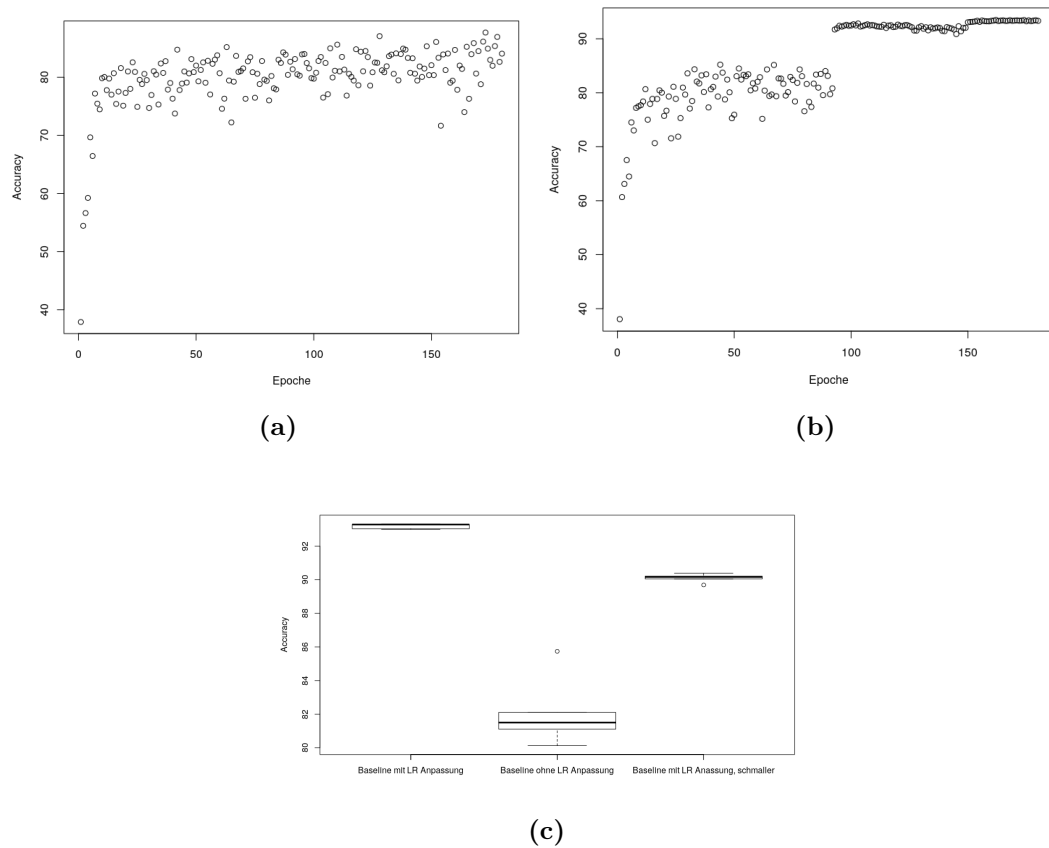


Abbildung 3.4: Vergleich zwischen (a) Baseline-Netz ohne Anpassung der Lernrate und (b) Baseline-Netz mit Anpassung der Lernrate in Epoche 93 und 150. (c) Boxplot der Accuracys

zufriedenstellend.

Eine Verkleinerung der Lernrate kann dieses Ergebnis verbessern [GBC16]. In Abbildung 3.4b wird dargestellt, wie sich der Verlauf ändert durch eine Anpassung der Lernrate bei Epoche 93 und 150. In diesen beiden Epochen wird die Lernrate jeweils auf ein Zehntel verkleinert.

In Abbildung 3.4c ist ein Boxplot dargestellt, der die Accuracy von jeweils fünf Experimenten mit oder ohne Anpassung der Lernrate vergleicht. Es ergibt sich eine deutliche Verbesserung der Accuracy des Baseline-Netzes von 81.24 % auf 91.89 % durch diese Anpassung. Aufgrund dieser Verbesserung werden die Experimente mit Anpassung der Lernrate als Grundlage zum Vergleich mit den Experimenten in den weiteren Kapiteln herangezogen. Sie werden in diesen Kapiteln nur mit Baseline-Netz bezeichnet.

Ein weiteres Vergleichskriterium neben der Accuracy sind die durchschnittli-

chen Trainingszeiten pro Epoche. Für jedes Experiment wird die durchschnittliche Dauer einer Trainingsepoche mittel des arithmetischen Mittels berechnet. Die Durchschnittswerte über alle 180 Epoche sind für die zehn Baseline Experimente sind in Tabelle 3.2 aufgelistet. Die Durchschnittswerte liegen sehr nah beieinander, der Unterschied zwischen dem grössten und dem kleinsten Durchschnittswert liegt bei 0,28. Damit ergeben sich zwischen den zehn Experimenten keine signifikante Unterschiede. Es wird daher mit Experiment vier eines der Experimente mit Anpassung der Lernrate ausgesucht um für die folgenden Experimente/ Kapitel als Vergleich zu dienen.

	Experimente ohne Anpassung der Lernrate					Experimente mit Anpassung der Lernrate				
	1	2	3	4	5	1	2	3	4	5
μ	19,49	19,53	19,50	19,48	19,84	19,57	19,56	19,53	19,53	19,62

Tabelle 3.2: Tabelle für Durchschnittswerte und Standardabweichungen der Trainingszeiten der Experimente

Evaluierung des schmalleren Baseline-Netzes

Das schmallere Baseline-Netz wird verwendet, um für MorphNet und Net2Net eine schmale Variante zu haben. Der Grund hierfür ist, dass bei einer durchschnittlichen Accuracy von 93,19 % des breiten Baseline-Netzes nicht mehr viel Raum für Verbesserungen bleibt. In Abbildung 3.4c sind die Experimente für das breite und schmale Baseline-Netz mit der Anpassung der Lernrate gegenüber gestellt. Der Unterschied vom breiten zum schmalen Netz ist ein Accuracy-Verlust von 3,1 %.

In Abbildung 3.5a ist abgebildet, wie sich die Accuracy für das schmallere Netz verhält, bei der gleichen Anpassung der Lernrate wie beim breiteren Baseline-Netzen. Der Unterschied in der Accuracy zwischen dem schmalen und breiten Baseline-Netz ist in Abbildung 3.5b abgebildet.

3.2 Konzept

In den nachfolgenden Kapiteln wird MorphNet mit einer Kombination aus PruneTrain und Net2Net verglichen. MorphNet ist eine Technik, bei der die Struktur des Netzes durch Wiederholtes Verbreitern des Netzes und Verkleinern des Netzes mittels eines Regularisierers gelernt wird. PruneTrain beschneidet das Netz

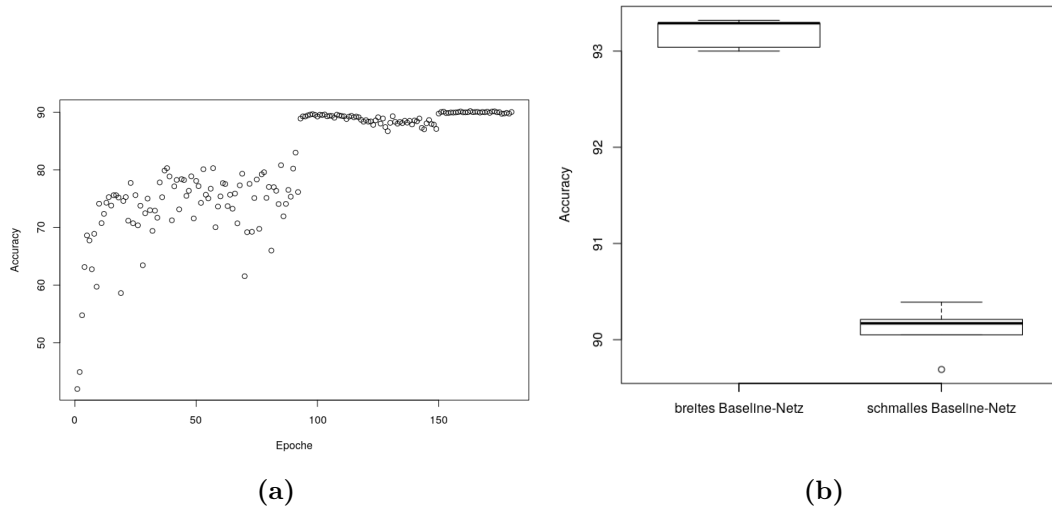


Abbildung 3.5

so, dass unwichtige Gewichte auf Null gesetzt werden mit dem Ziel ganze Kanäle auf Null zu setzen um diese zu Entfernen. Mit der Entfernung von Kanälen und falls alle Kanäle einer Schicht auf Null gesetzt wurde auch ganzen Schichten, soll Trainingszeit gespart werden bei möglichst geringem Accuracy Verlust.

Um diese beiden Methoden vergleichen zu können werden hier zunächst grundlegende Rahmenbedingungen festgelegt. Beide Methoden bekommen drei verschiedenen ausgeprägte Netzwerke mit jeweils drei Phasen:

- Netz mit $N = [3, 3, 3]$ und $K = [8, 16, 32]$
- Netz mit $N = [4, 4, 4]$ und $K = [4, 8, 16]$
- Netz mit $N = [4, 4, 4]$ und $K = [8, 16, 32]$

Dabei können mehrere Durchgänge der Netzvergrößerung durchlaufen werden. Beschränkt sind diese Durchläufe nicht direkt in der Anzahl, da hier beide Methoden verschieden lange Phasen zwischen den Netzvergrößerungsschritten haben. Um trotzdem zeitlich ein Begrenzung zu haben ist es beiden Methoden nicht erlaubt nach 4 Stunden Laufzeit das Netz nochmal zu vergrößern.

Um für die Netze eine Beschränkung zu haben sind maximal die Anzahl Parameter/Flops erlaubt, die das breite Baseline Netz hat.

MorphNet wird in Kapitel 4 evaluiert. In Kapitel 5 wird PruneTrain so evaluiert, wie es in der vorgefertigten Implementierung ¹ geschrieben wurde. In Kapitel 5.2

¹https://bitbucket.org/lph_tools/prunetrain/src/master/

wird die Erhöhung der Batchgröße bei Beschneiden des Netzes evaluiert.

In Kapitel 6 wird Net2Net evaluiert. In Kapitel 7 wird die Vorgehensweise zum Kombinieren von PruneTrain und Net2Net beschrieben sowie die Kombination evaluiert.

Der Vergleich der beiden Methoden wird in Kapitel 8 vollzogen.

nur wenn noch Zeit übrig: In Kapitel 8 werden die Ergebnisse aus 6.3 auf einem anderen Datensatz verifiziert. auch wenn noch Zeit: Prüfe wie sich die additiven Verfahren auf 6.3 auswirken Zuletzt werden noch additive Verfahren vorgetellt, welche die Trainingszeit zusätzlich minimieren können. Eine dieser Verfahren, welches in Kapitel evaluiert wird, spart Zeit durch die Verwendung von gemischt präzisen Zahlenformaten. Ein weiteres additives Verfahren in Kapitel überprüft in wiefern mit Hilfe einer adaptiven Anpassung der Lernrate die Batchgröße sinnvoll so angepasst werden kann, dass die ganze GPU genutzt werden kann.

4 Untersuchung von MorphNet

Die in Kapitel 2.7 erläuterte Methode zum “schnellen Ressourcen beschränkten Strukturlernen” (MorphNet) wird in diesem Kapitel evaluiert. In Algorithmus 2.2 wird das Vorgehen von MorphNet mittels Pseudocode dargestellt. Im ersten Schritt zur Evaluierung werden die einzelnen Schritte in diesem Algorithmus evaluiert. Im zweiten Schritt wird dann überprüft, wie gut der Algorithmus mit den in Kapitel 3.2 genannten Rahmenbedingungen abschneidet. Da für die Untersuchung der einzelnen Schritte von MorphNet die Ausführungszeit unwichtig ist wird dieser Teil auf der Geforce GTX 1080 Ti ausgeführt.

Hier Algorithmus nochmal hinschreiben?

4.1 Evaluierung der einzelnen Schritte von MorphNet

Im ersten Schritt von MorphNet wird das Netz trainiert um

$$\mathcal{W}^* = \underset{\mathcal{W}}{\operatorname{argmin}} l(f(\mathbf{x}_i, \mathcal{W}, y_i) + \lambda \mathcal{G}(\mathcal{W})) \quad (4.1)$$

zu finden. Der Regularisierer \mathcal{G} ist in dieser Formel dafür zuständig, dass die gewählte Zielgröße minimiert wird. Die zwei möglichen Zielgrößen sind die Modellgröße und Anzahl an FLOPs. Für beide Zielgrößen gilt, dass die im Regularisierer verwendete Formel nur eine vereinfachte Form der Zielgröße berechnet. Zunächst wird evaluiert, welchen Effekt der Regularisierer auf die Zielgröße hat. Dabei werden nur die ersten beiden Schritte des MorphNet-Algorithmus durchgeführt. In Abbildung 4.1b ist in Grün abgebildet, wie sich der Wert des Regularisierers für die Zielgröße FLOPs während dem Training verändert. Die blaue Kurve in Abbildung 4.1b ist der tatsächliche Verlauf der Flops über die Trainingszeit. Die blaue Kurve wird in Schritten weniger, da das Netz nur alle fünf Epochen mittels der zweiten MorphNet-Schrittes kleiner wird. Die verzögerte Reduzierung der Zielgröße liegt daran, dass erst mit einer gewissen Anzahl entfernbare Gewichte

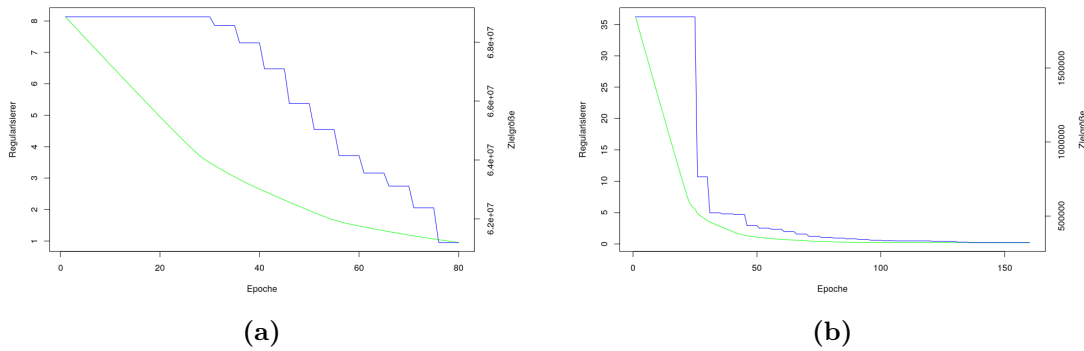


Abbildung 4.1: Vergleich Zielgröße mit Wert des Regularisierers für (a) FLOPs (b) Modellgröße

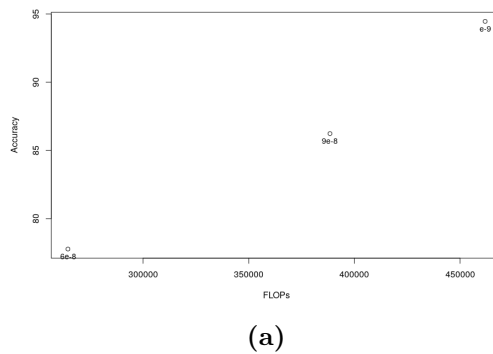


Abbildung 4.2: Effekt verschieden großer λ auf (a) FLOPs (b) Modellgröße

ttsächlich eine Änderung an den FLOPs passiert. Für die Zielgröße Modellgröße ist der Verlauf der beiden Kurven in Abbildung ?? abgebildet. Für die Zielgröße Modellgröße muss λ größer sein um einen Effekt auf die Zielgröße zu haben.

Der Effekt von verschieden großen λ wird im nächsten Schritt untersucht. Zu diesen Zweck wird ein Netzwerk mit verschiedenen Werten für λ trainiert. Auch hier werden nur die ersten beiden Schritte des MorphNet-Algorithmus mehrfach hintereinander durchgeführt. In Abbildung 4.2a ist zu sehen, wie sich der Morph-Net Algorithmus bei verschiedenen λ für die Zielgröße FLOPs verhält. Dabei wird das Netz jeweils für fünf Epochen trainiert und anschliessend wird das Netz beschnitten. Abhängig von der Größe von λ wird dem Regularisierer mehr oder weniger Gewicht gegeben. Dies sorgt für eine unterschiedlich große Verkleinerungsrate. In Abbildung ?? ist abgebildet wie sich die Netzverkleinerungsraten verändern, bei der Zielgröße Modellgröße.

Im letzten Schritt wird evaluiert, welchen Effekt die verschiedenen λ auf den gesamten MorphNet-Algorithmus haben. Auf Grund dieser Ergebnisse werden

die Parameter von MorphNet ermittelt, welches im nächsten Unterkapitel mit den Rahmenbedingungen von Kapitel 3.2 evaluiert wird.

4.2 Evaluierung der Ergebnisse von MorphNet

Die in Kapitel 3.2 vorgestellte Vorgehensweise wird hier verwendet, um MorphNet zu evaluieren. Dies bildet die Grundlage zum Vergleich in Kapitel 8.

5 Evaluation des Beschneidens des Netzes

5.1 Evaluation bei gleichbleibender Batchgröße

Die Untersuchung von PruneTrain basiert auf einer bereits vorgefertigten Implementierung [ptI]. In dieser Implementierung ist alles bis auf die Anpassung der Batchgröße an das kleiner werdende Netz enthalten. Es wird das Ergebnis der Ausführung von PruneTrain auf der Hardware mit den Ergebnissen aus der Veröffentlichung verglichen [LCZ⁺19]. Ziel der Experimente ist es zu evaluieren, wie eine Änderung der verschiedenen Hyperparameter die Trainingszeit und die Accuracy beeinflusst. Im Gegensatz zur Veröffentlichung von PruneTrain wird hier statt auf mehreren GPUs nur auf einer GPU gerechnet. Bei der Evaluierung der Einflüsse werden die veränderbaren Hyperparameter von PruneTrain einzeln verändert, um den Einfluss der einzelnen Veränderungen zu untersuchen. Die veränderbaren Hyperparameter sind:

- Lasso-Ratio 0,2
- Rekonfigurationsintervall 5
- Grenzwert 0,0001
- Lernrate 0,1

Hinter den veränderbaren Hyperparameter steht jeweils der Wert, den der Hyperparameter hat, wenn er im aktuellen Experiment nicht verändert wird, hat. Betrachte eine feste Batchgröße von 256 über 180 Epochen und vergleiche diese mit dem Baseline-Netzes aus Kapitel 3.1.4.

Einfluss von verschiedenen Lasso-Ratio Werten auf das Netz

Die Lasso-Ratio gibt an, wie stark das Netz beschnitten werden soll. In diesen Experimenten wird die Lasso-Ratio von 0,05 bis 0,25 in 0,05er Schritten verändert.

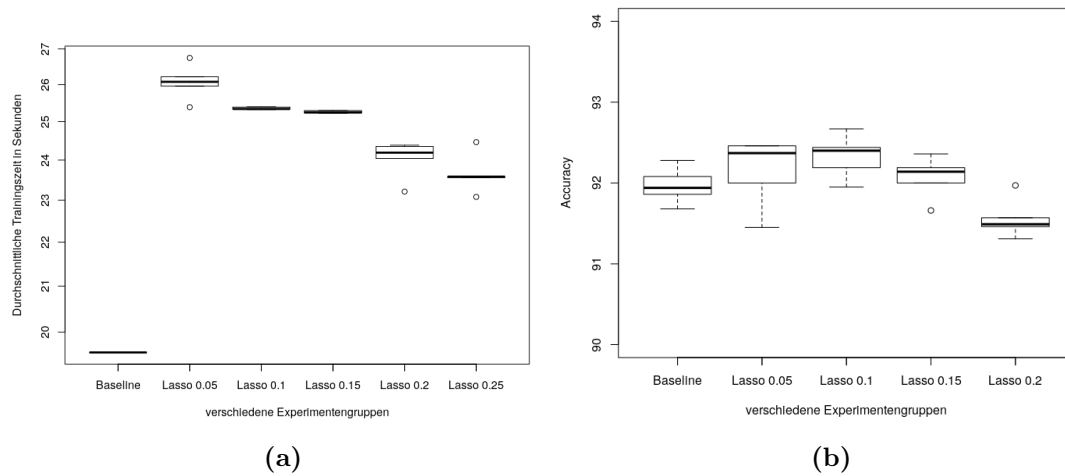


Abbildung 5.1: Lasso-Ratio Experiment: (a) Boxplot der durchschnittlichen Trainingszeit (b) Boxplot der Accuracys

In Abbildung 5.1a ist zusehen, dass mit steigender Lasso-Ratio durchschnittlich weniger Trainingszeit gebraucht wird. Für die durchschnittliche Trainingszeit einer Epoche wird das arithmetische Mittel über alle Epochen angewandt. Die Experimente werden dann nach ihrer Zugehörigkeit einsortiert und als Boxplot in Abbildung 5.1a dargestellt. Trotz der nachlassenden Trainingszeit mit steigender Lasso-Ratio ist die Trainingszeit des Baseline-Netzes signifikant schneller. Dies ist durch den Overhead erklärbar, welcher durch das Beschneiden des Netzes entsteht.

In Abbildung 5.1b ist die Accuracy der verschiedenen Experimente abgebildet. Es fällt auf, dass das Baseline-Netz im Mittel etwa schlechter ausfällt als die Experimente mit den Lasso-Ratio von 0,05 bis 0,15. Der Grund hierfür ist ein Overfitting. Das Overfitting wurde hier erkannt durch eine Trainingsaccuracy von 100,00 % für mehrere Epochen ohne ein weiter steigende Validierungsaccuracy.

Experimente zum Rekonfigurationsintervall

Als nächste Größe wird der Einfluss des Rekonfigurationsintervalls überprüft. Die entsprechenden Grafiken sind in Abbildung 5.2 zu sehen. In Abbildung 5.2a sind für die verschiedenen Experimente die Trainingszeiten pro Epoche zu sehen. Dabei werden drei verschiedene Rekonfigurationsintervalle (2,5 und 10) verglichen. In Abbildung 5.2a lässt sich für die verschiedenen Trainingszeiten der Experimente zum Rekonfigurationsintervall kaum Unterschiede erkennen. Daher ist in

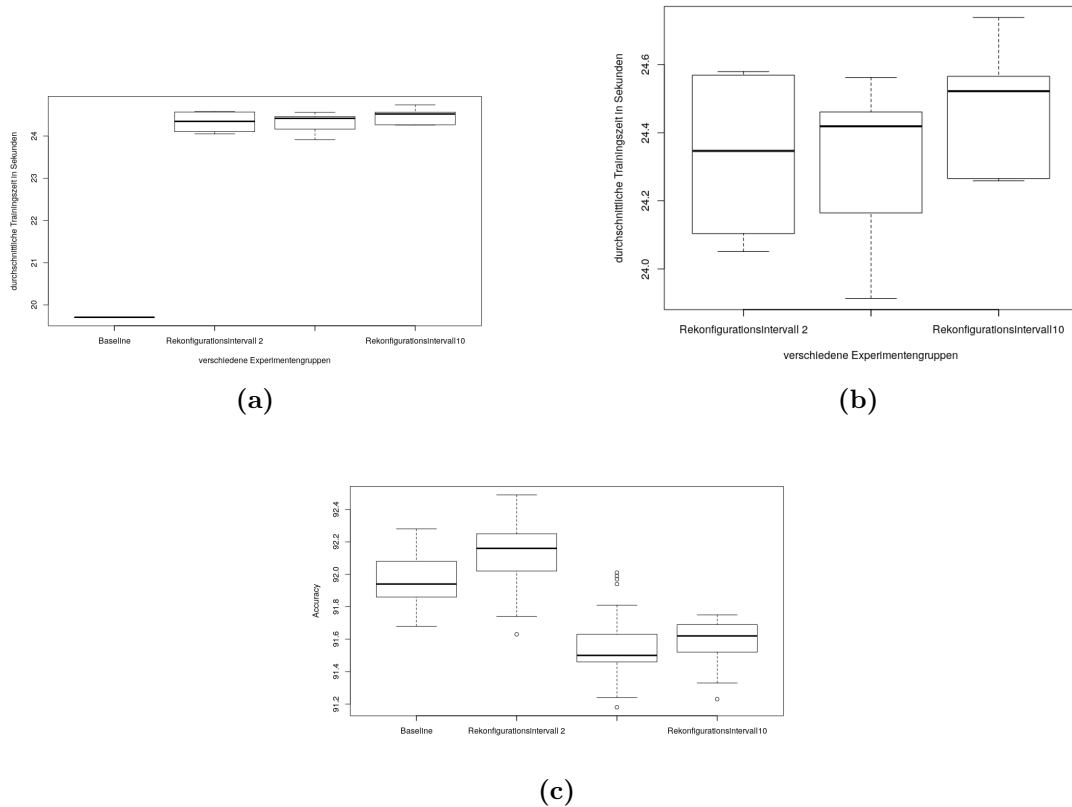


Abbildung 5.2: Experimente zum Rekonfigurationsintervall: (a) Boxplot der durchschnittlichen Trainingszeit (b) Boxplot der durchschnittlichen Trainingszeit ohne Baseline-Netz (c) Boxplot der Accuracys

Abbildung 5.2b ein Boxplot ohne die Baseline Werte abgebildet.

Mit Hilfe dieses Boxplots lässt sich erkennen, dass die durchschnittliche Trainingszeit aller Experimente mit dem Rekonfigurationsintervall steigt.

In Abbildung 5.2c ist zu sehen, wie sich die Accuracy bei diesen Experimenten verhält.

Die Effekte in der Accuracy vom Baseline Netz zum Rekonfigurationsintervall sind wieder mit einem Overfitting zu erklären. Die Effekte vom Rekonfigurationsintervall 2 zu 5 und zu 10 sind auch mit Zunehmenden Overfitting je weniger geprunt wird zu erklären. Abhilfe mehr Experimente und zwar mit dem schmalen Baseline Netz

Experimente zur Lernrate

Der Einfluss der Lernrate auf das Beschneiden des Netzes wird mit fünf verschiedenen Lernraten untersucht. Beginnend mit der Lernrate 0,2 und für jede weitere

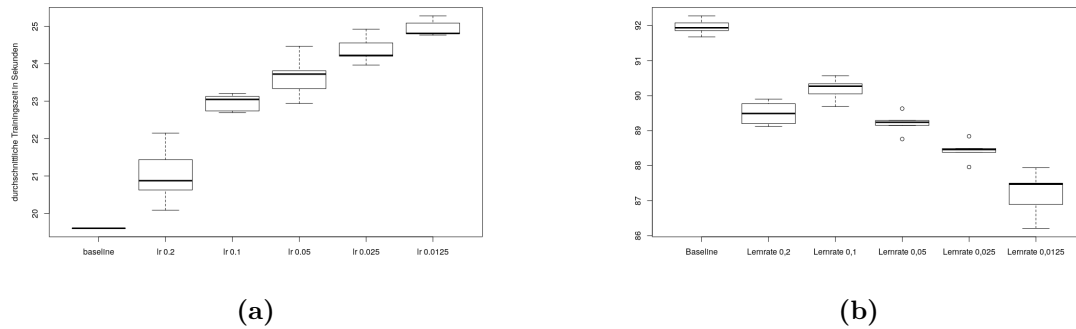


Abbildung 5.3: Experimente zu Lernrate: (a) Boxplot der durchschnittlichen Trainingszeit (b) Boxplot der durchschnittlichen Trainingszeit ohne Baseline-Netz (c) Boxplot der Accuracys

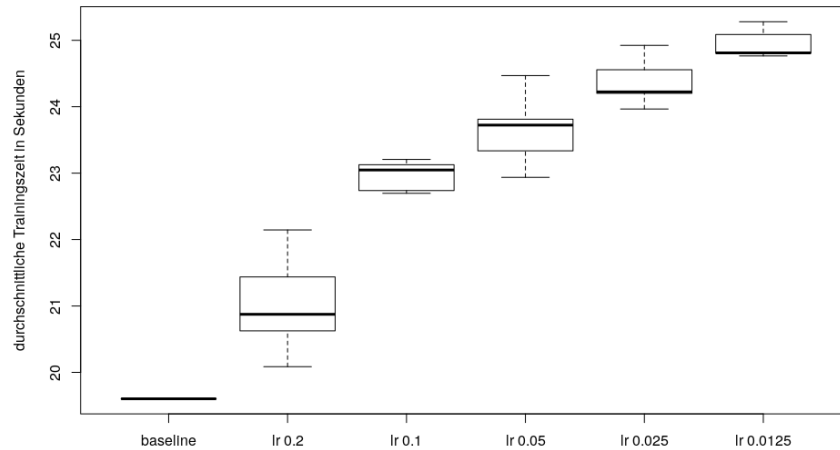
der fünf Lernraten die Hälfte der vorherigen. Die durchschnittliche Trainingszeit in Sekunden für verschiedene Lernrate ist in Abbildung 5.3 zu sehen. Es ist deutlich zu sehen, dass mit sinkender Lernrate die Trainingszeit steigt. Das bedeutet, dass mit sinkender Lernrate weniger von Netz beschnitten wird.

In Abbildung 5.3b sind die Accuracy der verschiedenen Lernrate abgebildet. Die Lernrate 0.1 schneidet hier am Besten ab. Dies kann darauf zurück geführt werden, dass bei einer größeren Lernrate weniger Minima in der Verlustfunktion gefunden werden können. Der Effekt bei wesentlich kleineren Lernraten ist, dass der Trainingsprozess zwar mit jedem Schritt in die Richtung des Minimums geht, dabei aber durch die kleine Lernrate das tatsächliche Minimum innerhalb der 180 Epochen nicht erreicht.

Experimente zum Grenzwert

Diskussion der Methode

Für die Evaluation des Beschneidens des Netzes werden in der Original-Veröffentlichung mehrere GPUs verwendet [LCZ⁺19]. Dies führt dazu, dass bereits in diesem Teil der Implementierung Trainingszeit durch verminderte Kommunikation zwischen den GPUs gespart wird. Da hier nur mit einer GPU evaluiert wird ergibt sich hier noch keine direkte Einsparung an Trainingszeit. Eine weitere Möglichkeit Trainingszeit zu sparen ergibt sich durch Erhöhen der Batchgröße bei kleiner werdendem Netz. Zu beachten ist hier, dass die Speicherauslastung gleich bleiben sollte und eine Vergleichbarkeit mit der Veröffentlichung zu gewährleisten. Diese Evaluierung wird in Kapitel 5.2 durchgeführt.



5.2 Experimente zur Anpassung der Batchgröße beim Beschneiden des Netzes

Die Anpassung der Batchgröße des Netzwerks in der Veröffentlichung arbeitet mit einer Grenze bis zu dieser der Speicher ausgelastet werden darf.

Die Berechnung der maximalen Batchgröße für eine gegebene Speichergröße und Netzarchitektur wird in Kapitel 5.2.1 beschrieben.

5.2.1 Berechnung der Batchgröße abhängig vom Speicherverbrauch

Da sich in Pytorch der freie Speicher nicht direkt auslesen lässt wird mit Hilfe von Experimenten, die auf der GPU durchgeführt werden gemessen wie sich die Speicherauslastung verhält. In Abbildung 5.4a ist zu sehen, wie sich die Speicherauslastung proportional zur Batchgröße verhält. Es ist gut zu erkennen, dass der Zusammenhang linear ist. Die Passgenauigkeit dieses Zusammenhangs kann mittels einer linearen Regression bestimmt werden. Daher wird die rote Gerade wurde einer linearen Regression berechnet. Der maximale Abstand der gemessenen Punkte zur Gerade ist 0,19 für Punkte, die unter der Gerade liegen sowie 0,67 für Punkte die über der Gerade liegen. Zusammen mit der graphischen Übereinstimmung ergibt sich klar ein linearer Zusammenhang mit kleinen Abweichungen. Durch diesen linearen Zusammenhang reicht es ein Modell zu bilden, welches für einen Wert des Speicherverbrauchs abhängig von der Netzarchitektur berechnet,

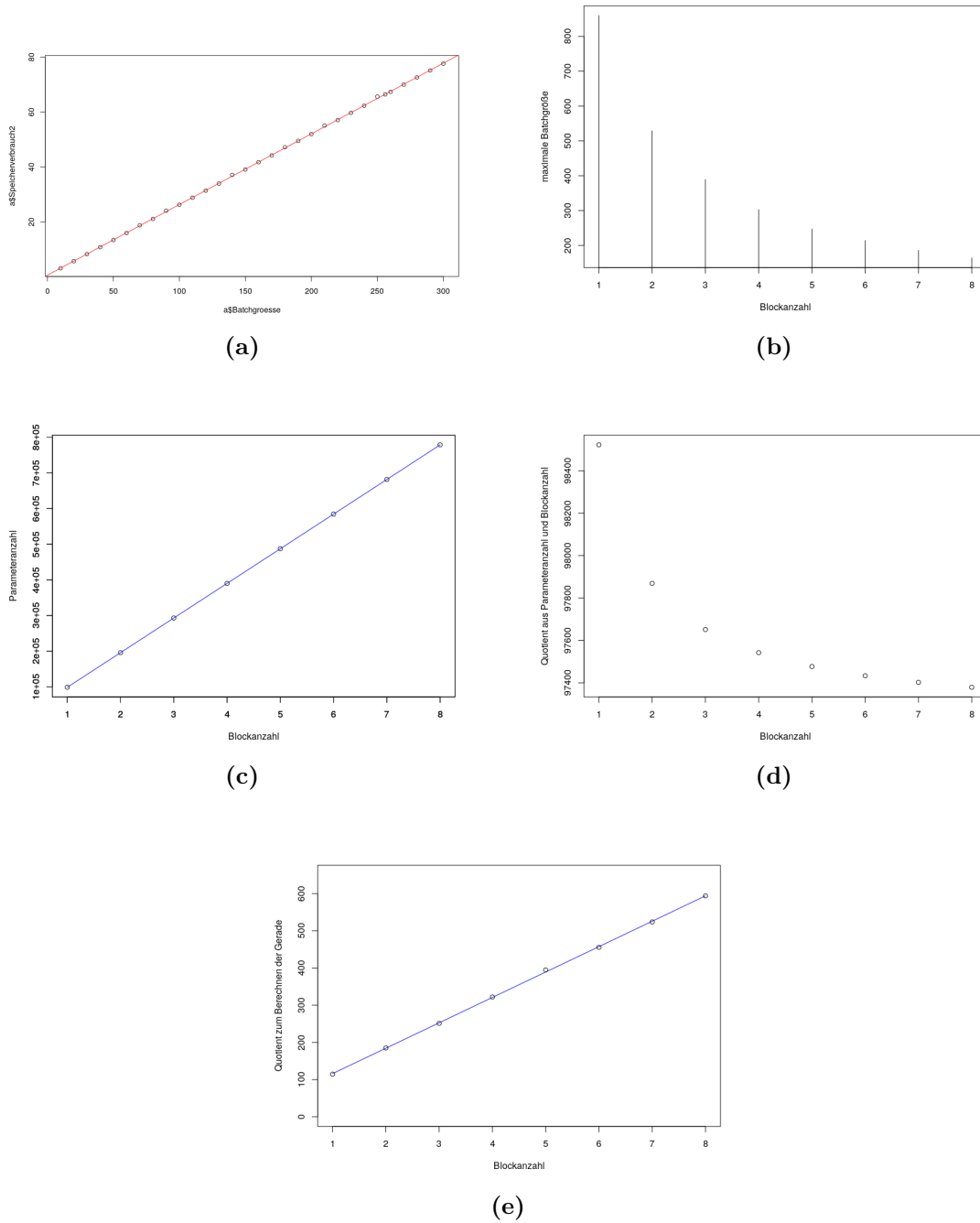


Abbildung 5.4: Darstellung des Berechnen der Geraden

wie groß die Batchgröße maximal sein darf. Zu diesem Zweck wird ein Netz mit drei Phasen betrachtet und ein Modell entwickelt mit dem sich die maximale Batchgröße berechnen lässt. Die maximale Batchgröße hängt ab von:

- der Blockanzahl
- der Phasenanzahl und
- der Anzahl an Schichten pro Block
- Breite der Schichten je Phase

In Abbildung 5.4b ist abgebildet, wie sich die Blockanzahl bei gleichbleibender Speicherauslastung auf die maximale Batchgröße auswirkt. Die Blockanzahl wirkt sich wie zu sehen ist nicht linear auf die maximale Batchgröße aus. Betrachtet man hingegen den Zusammenhang zwischen Blockanzahl und Parameteranzahl, wie in Abbildung 5.4c abgebildet, so ergibt sich hier ein linearer Zusammenhang. In Abbildung 5.4d ist der Zusammenhang zwischen Blockanzahl und dem Quotienten aus Parameteranzahl und Blockanzahl zu sehen. Der Verlauf dieser Kurve ähnelt dem Verlauf von Abbildung 5.4b. In Abbildung 5.4e wird daher der Quotient aus diesen beiden Größen gegen die Blockanzahl geplottet und eine Gerade mit linearer Regression berechnet. Um die Wahrscheinlichkeit zu prüfen, mit fälschlicherweise angenommen wird, dass in Abbildung 6.1 ein linearer Zusammenhang besteht wird ein t-Test ausgeführt. Es ergibt sich die Alternativ- und Nullhypothese:

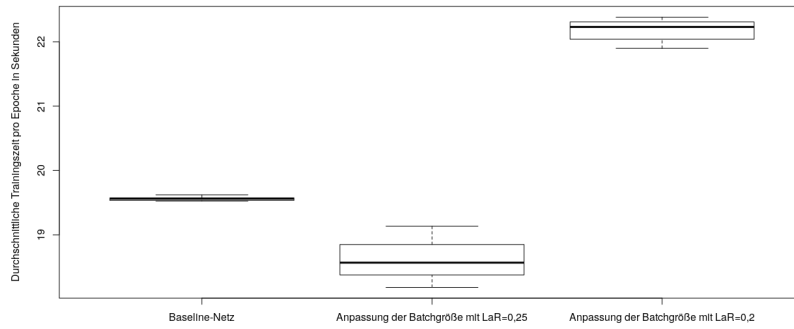
H_0 : Es besteht kein linearer Zusammenhang zwischen dem Quotienten G
und der Anzahl von Blöcken

H_1 : Es besteht ein linearer Zusammenhang zwischen dem Quotienten G
und der Anzahl von Blöcken

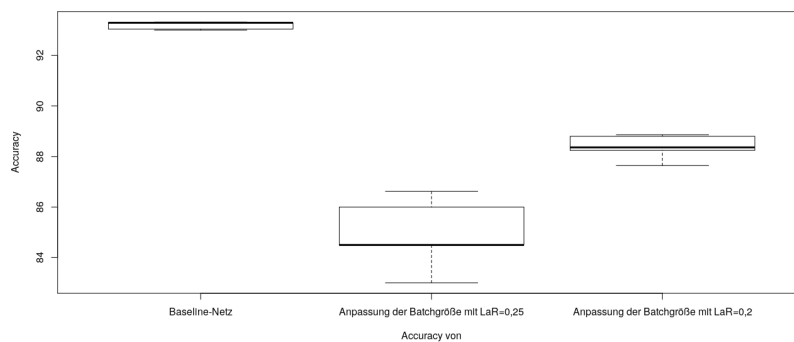
Mit einem Signifikanzniveau von $\alpha = 0,05$ und einem p -Wert von $p = 3,824 \cdot 10^{-12}$ ist die Nullhypothese abzulehnen, und die Alternativhypothese anzunehmen. Mit Hilfe dieses linearen Zusammenhangs lässt sich bei gegebener Parameteranzahl (PA) und Blockanzahl (BA) die maximale Batchgröße (BG) berechnen:

$$BG = \frac{PA}{BA(68,25 \cdot BA + 47,85)} \quad (5.1)$$

Da einzelne Punkte einen maximalen Abstand von $d_- = 2,07$ zur Geraden wurde das Ergebnis durch Multiplikation mit $0,98$ ein Sicherheitsabstand eingeführt.



(a)



(b)

Abbildung 5.5: Vergleich von (a) Durchschnittlicher Trainingszeit (b) Accuracys von PruneTrain mit Anpassung der Batchgröße und Baseline-Netz

5.2.2 Evaluierung der Anpassung der Batchgröße an die Netzgröße

Da sich für ein gegebenes Netz die Speicherauslastung abhängig von der Batchgröße ein linearer Zusammenhang ergibt, kann die Batchgröße direkt angepasst werden, sobald das beschchnittene Netz für eine Epoche trainiert hat. Es wird per Dreisatz berechnet wie groß die Batchgröße sein darf, bei gegebener maximaler Speicherauslastung. In Abbildung 5.5a ist zu sehen wie sich die durchschnittliche Trainingszeit pro Epoche entwickelt, bei Anpassung der Batchgröße. Für Abbildung 5.5a werden zwei verschieden große Lasso-Ratio Werte (LaR) (0,2 und 0,25) getestet. Bei der Lasso-Ratio von $LaR = 0,25$ ergibt sich ein Gewinn an durchschnittlicher Trainingszeit pro Epoche.

In Abbildung 5.5b ist zu sehen, wie gross der Accuracy-Verlust für das PruneTrain-Netz mit Anpassung der Batchgröße. Für PruneTrain mit einer Lasso-Ratio von

0,2 ergibt sich ein Accuracy Verlust von durchschnittlich xx,xx %. Für eine höhere Lasso-Ratio ergibt sich ein Accuracy-Verlust von xx,xx %.

6 Evaluierung von Net2Net

Die Operatoren zur Beschleunigung des Lernens durch Wissenstransfer werden in diesem Unterkapitel evaluiert. Diese Evaluierung arbeitet mit einer selbst erstellten Implementierung auf Grundlage der Veröffentlichung zum Thema Net2Net [CGS15].

Die Evaluierung umfasst drei unterschiedliche Situationen, diese Situationen sind analog zu den in der dazugehörigen Quelle [CGS15]. Die Evaluierung arbeitet mit einem ResNet, wie in Kapitel 3.1.4.

In der ersten Situation wird der Operator für ein breiteres Netz verwendet, um ein schmalleres ResNet32 zu trainieren. In der zweiten Situation wird der Operator für ein tieferes Netz benutzt um in einem der Stages des Netzes einen neuen Block einzufügen. In der dritten Situation werden beide Operatoren kombiniert. Mit der Kombination wird der Raum erkundet, der durch die verschiedenen tiefen und breiten Modelle aufgespannt wird. Die drei Situationen werden in den drei folgenden Unterkapiteln näher beschrieben.

6.1 Evaluierung des Operators für ein breiteres Netz

Evaluiert wird der Operator durch verschiedene Optionen, welcher Bereich des Netzes breiter gemacht wird:

- Alle Phasen
- Eine ganze Phase

Wie in Kapitel 2.5 beschrieben werden beim Operator für ein breiteres Netz die Gewichte für die neu hinzugefügten Gewichte aus den ursprünglichen Gewichten ausgewählt und transformiert. Um zu evaluieren wie gut diese Methode funktioniert wird sie verglichen mit dem schmalen und breiten Baseline-Netz. Um die Methode der Initialisierung der zusätzlichen Kanäle zu evaluieren wird als Vergleich ein Netz trainiert, bei welchem die zusätzlichen zusätzlichen Gewichte zufällig initialisiert werden.

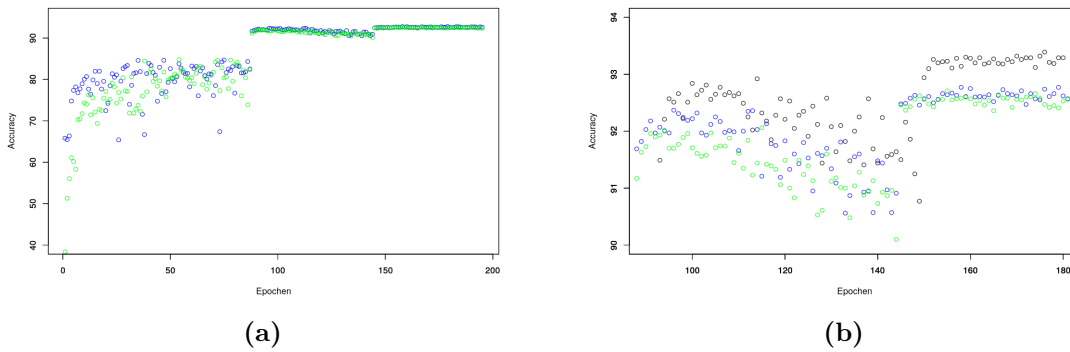


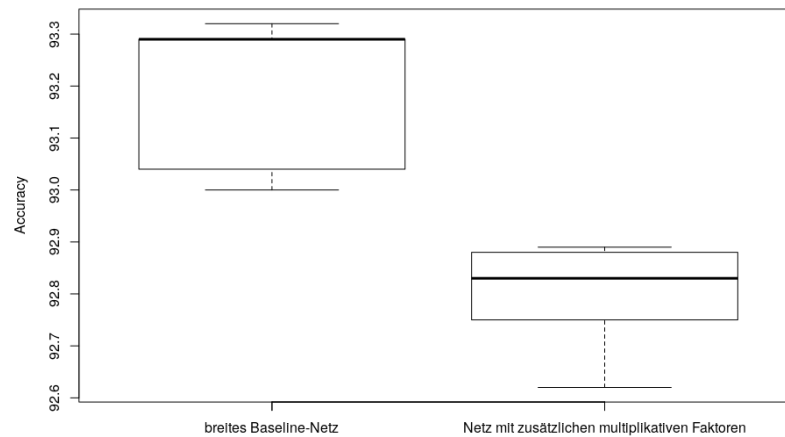
Abbildung 6.1: Vergleich der Accuracy bei verschiedenen Initialisierungsmöglichkeiten des Operators für ein breiteres mit dem Baseline Netz: (a) alle Epochen (b) Zoom auf die Epochen 90 bis 180. Blau zeigt die Accuracy der Initialisierung der zusätzlichen Gewichte mit zufälligen Werten. Grün zeigt die Initialisierung mit den Werten aus dem restlichen Tensor. Schwarz zeigt das Baseline-Netz als Vergleich.

6.1.1 Verbreitern aller Phasen

Mit Hilfe verschiedener Trainingsprotokolle wird untersucht, wie der Operator für ein breiteres Netz die Klassifikationsleistung des Netzes verändert. Zunächst wird überprüft, welches Ergebnis bei Anwendung des Operators für ein breiteres Netz auf das schmale Baseline-Netz nach 180 Epochen Training herauskommt. Nachdem Anwenden des Operators wird mit dem gleichen Trainingsprotokoll für weitere 180 Epochen trainiert. In Abbildung 6.1a ist abgebildet, wie der Verlauf der Accuracy für Net2Net mit zwei verschiedenen Möglichkeiten, die neuen Gewichte zu initialisieren, ist. In Blau dargestellt wird der Verlauf der Accuracy für die zufällige Initialisierung der neuen Gewichte durch das Verbreitern des Netzes. Es ergibt sich, wie in der größeren Abbildung 6.1b zu sehen ist eine minimale Verschlechterung durch das Anwenden des Operator für ein breiteres Netz im Vergleich zum Baseline-Netz. Um den Operator für ein breiteres Netz besser verwenden zu können wird im nächsten Schritt überprüft, ob mit einer häufigeren schrittweisen Anpassung der Lernrate und mit weniger trainierten Epochen nach Anwenden des Operator ein besseres Ergebnis möglich ist.

6.2 Evaluierung des Operators für ein tieferes Netz

Zur Evaluierung des breiteren Netzes wird zunächst wie in der Veröffentlichung jeder Block um eine Schicht erweitert. Dabei werden die zusätzlichen Schichten



(a)

Abbildung 6.2: Vergleich von

wie in Kapitel ?? beschrieben initialisiert. Als Vergleich dient das Baseline-Netz aus Kapitel 3.1.4.

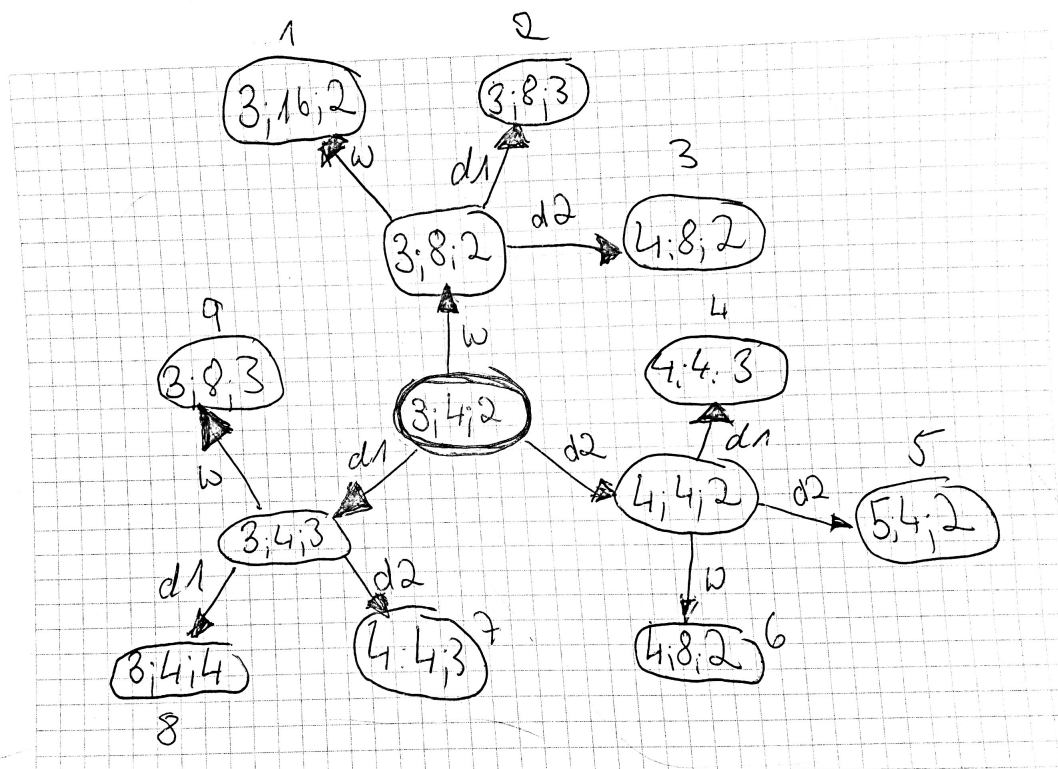
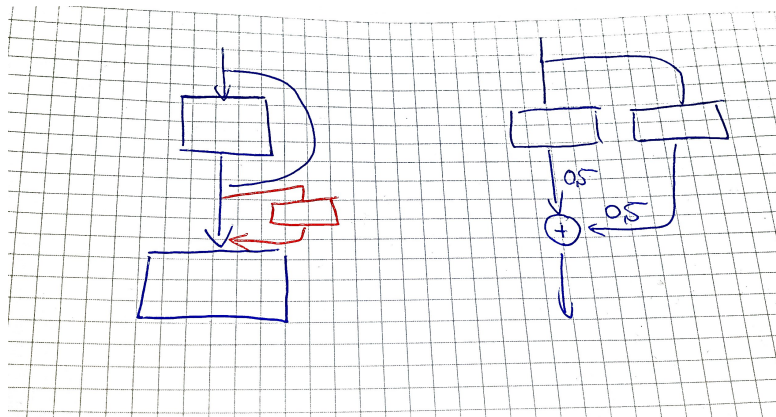
Ergebnisse

Eine weitere Verwendung des Operators für ein tieferes Netz ist die Möglichkeit einen neuen Block hinzuzufügen. Nur ein einfaches Hinzufügen würde hier zu Problemen führen. Der Grund hierfür ist Abbildung ?? dargestellt. Dabei soll an einer Verbindung, die bisher die Daten von einer Schicht zur nächsten transportiert ein neuer Block inklusive Kurzschlussverbindung entstehen. Der hinzuzufügende Block

blau markieren

blau markiert. Der neue Block soll wie die Kurzschlussverbindung die Identität berechnen. Betrachtet x als Größe der Identität. Dann wird hier mit dem neuen Block statt x das doppelte, also $2x$ berechnet. Um dieses Problem zu umgehen wird an jeder Additionsstelle für eine Kurzschlussverbindung eine Multiplikation mit $0,5$ berechnet (rechte Seite der Grafik). So wird nicht mehr x sondern $0,5x + 0,5x = x$ berechnet. Zunächst wird dieses Netz trainiert ohne einen Operator anzuwenden. In Abbildung 6.2a ist ein Boxplot abgebildet, der die Accuracy des Netzes mit zusätzlichen multiplikativen Faktoren mit dem Baseline-Netz vergleicht.

6.3 Erkunden des Modellraums



(a)

Abbildung 6.3: Anzahl an Blöcken pro Phase; Breite der ersten Schicht; Schichten pro Block

7 Evaluierung der Kombination von PruneTrain und Net2Net

Adaptive Kombination of prune and net2net

AKoPaN

Für die Kombination betrachten wir zunächst den Ausgang eines PruneTrain-Durchlaufs über 180 Epochen mit verschiedenen breiten Netzen:

4,8,16

8,16,32

16,32,64

und verschiedenen tiefen Netzen:

[3,3,3]

[4,4,4]

Gestartet wird wie bei Morphnet einmal mit

4,8,16 und mit

8,16,32 jeweils mit 3 Blöcken pro Phase analog zu Kapitel 4.

um das Größerwerden des Netzes nicht ausarten zu lassen sollte es nicht größer werden als das [5,5,5] er Netz mit 16,32,64 als maximalen Größe vorgegeben.

Drei Probleme können beim Training des Netzes auftauchen:

- Underfitting
- Overfitting
- Saturation
- Rumspringen ohne eine Tendenz zur Verbesserung

Underfitting erkennen wir durch einen immer noch recht hohen Trainingsfehler und relativ wenig Pruning.

Sowohl bei Hinzufügen von neuen Blöcken als auch beim Breiter machen des Netzes kann Overfitting passieren. Kommt es zu Overfitting ist es möglich, den Lasso-Ratio koeffizient weiter zu erhöhen, oder den Grenzwert zum Beschneiden

des Netzes zu Erhöhen. Das heisst es ist nötig, Overfitting zu erkennen. Dies wird hier durch Abstand von Trainings und Validierungsfehler. Wird der Abstand hier grösser bei einer Zunahme des Validierungsfehlers oder ist die Testaccuracy bei 100 % angekommen, wird Overfitting diagnostiziert.

Saturation Bilde aus den letzten 10 Validierungsaccuracy jeweils ein exponentiell geglättetes Mittel gewichtetes Mittel. Zeigt dieses Mittel keine Verbesserung innerhalb der nächsten 10 Epochen -> early stop und je nach Vorliegen von Under oder Overfitting weiterverfahren.

8 Vergleich

9 Ausblick und Fazit

A d

Abbildungsverzeichnis

2.1	Abbildung zur Faltung [GBC16]	6
2.2	Convolutional Neural Net [CCGS16]	7
2.3	Abbildung der Kurzschlussverbindung [HZRS15]	10
2.4	Vergleich zweier Residual-Netz-Blöcke [HZRS15]	11
2.5	Tägliche Submissionen der Category Machine Learning auf arxiv [oA19]	12
2.6	Mindmap zu den Suchbegriffen bezüglich des aktuellen wissen- schaftlichen Stands	12
2.7	Beispielhafte Darstellung des Kanal-Union-Verfahrens	14
2.8	Traditioneller Workflow vs. Net2Net Workflow	17
2.9	Übersicht über die zusätzlichen Kanäle	18
2.10	Validation Accuracy von CNNs anhand eines Beispieldatensatzes .	27
3.1	Basisblock	31
3.2	Übergangsblock	31
3.3	Grafische Darstellung Basis- und Übergangsblock	31
3.4	Vergleich zwischen (a) Baseline-Netz ohne Anpassung der Lernrate und (b) Baseline-Netz mit Anpassung der Lernrate in Epoche 93 und 150. (c) Boxplot der Accuracys	33
3.5	35
4.1	Vergleich Zielgröße mit Wert des Regularisierers für (a) FLOPs (b) Modellgröße	38
4.2	Effekt verschieden großer λ auf (a) FLOPs (b) Modellgröße . . .	38
5.1	Lasso-Ratio Experiment: (a) Boxplot der durchschnittlichen Train- ingszeit (b) Boxplot der Accuracys	42
5.2	Experimente zum Rekonfigurationsintervall: (a) Boxplot der durch- schnittlichen Trainingszeit (b) Boxplot der durchschnittlichen Train- ingszeit ohne Baseline-Netz (c) Boxplot der Accuracys	43

5.3	Experimente zu Lernrate: (a) Boxplot der durchschnittlichen Trainingszeit (b) Boxplot der durchschnittlichen Trainingszeit ohne Baseline-Netz (c) Boxplot der Accuracys	44
5.4	Darstellung des Berechnen der Geraden	46
5.5	Vergleich von (a) Durchschnittlicher Trainingszeit (b) Accuracys von PruneTrain mit Anpassung der Batchgröße und Baseline-Netz	48
6.1	Vergleich der Accuracy bei verschiedenen Initialisierungsmöglichkeiten des Operators für ein breiteres mit dem Baseline Netz: (a) alle Epochen (b) Zoom auf die Epochen 90 bis 180. Blau zeigt die Accuracy der Initialisierung der zusätzlichen Gewichte mit zufälligen Werten. Grün zeigt die Initialisierung mit den Werten aus dem restlichen Tensor. Schwarz zeigt das Baseline-Netz als Vergleich. .	52
6.2	Vergleich von	53
6.3	Anzahl an Blöcken pro Phase; Breite der ersten Schicht; Schichten pro Block	54

Literaturverzeichnis

- [CCGS16] J.F. Couchot, R. Couturier, C. Guyeux, and M. Salomon. Steganalysis via a convolutional neural network using large convolution filters. *CoRR*, 2016.
- [CGS15] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. 11 2015.
- [DZZZ20] Hongwei Dong, Bin Zou, Lamei Zhang, and Siyu Zhang. Automatic design of cnns via differentiable neural architecture search for polsar image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 2020.
- [FC19] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*, 2019.
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [Hay98] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.
- [HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 2015.
- [iee85] Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.
- [IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

- [LCZ⁺19] Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Mattan Erez, and Sujay Shanghavi. Prunetrain: Gradual structured pruning from scratch for faster neural network training. *CoRR*, abs/1901.09290, 2019.
- [MAL⁺19] Guillaume Michel, Mohammed Amine Alaoui, Alice Lebois, Amal Feriani, and Mehdi Felhi. DVOLVER: efficient pareto-optimal neural network architecture search. *CoRR*, abs/1902.01654, 2019.
- [oA19] ohne Autor. arxiv machine learning classification guide, 12 2019. Onlinequelle; Aufgerufen am 01.06.2020; <https://blogs.cornell.edu/arxiv/2019/12/05/arxiv-machine-learning-classification-guide/>.
- [ptI] Prune train implementierung. online https://bitbucket.org/lph_tools/prunetrain/src/master/ aufgerufen am 10.06.2020.
- [SG17] Charles A. Sutton and Linan Gong. Popularity of arxiv.org within computer science. *CoRR*, 2017.
- [SXZ⁺20] Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Jiancheng Lv. Automatically designing cnn architectures using the genetic algorithm for image classification. *IEEE Transactions on Cybernetics*, 2020.
- [TKYG20] Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow, 2020.