technische universität
dortmund

fakultät für informatik

Masterarbeit

Zeit-effizientes Training von Convolutional Neural Networks

Jessica Buehler

17. August 2020

Gutachter:

Prof. Dr. Heinrich Müller

M.Sc. Matthias Fey

Lehrstuhl VII
Informatik
TU Dortmund

Inhaltsverzeichnis

Todo list

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 Vorwärtsdurchgang.
$\mathcal{W}^k$	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

Trainingszeiten für Convolutional Neural Networks (CNNs) wachsen schnell mit der Komplexität des Datensatzes und der Größe des Netzes. Neben der Trainingszeit eines bestimmten Netzes kostet vorallem die eventuell nötige Anpassung der Hyperparameter des Netzes und weitere Trainingsdurchläufe Zeit. Ziel dieser Masterarbeit ist zu untersuchen in wie fern dieser Prozess beschleunigt und automatisiert werden kann. Um den Prozess des Findens der besten Hyperparameter/ Netzarchitektur zu automatisieren gibt es bereits einige Arbeiten. Viele dieser Herangehensweisen lassen sich allerdings nicht direkt auf einen unbekannten Datensatz anwenden oder der Zeitaufwand für diese Verfahren ist enorm. Ein Verfahren, dass diese Nachteile nicht hat ist das Ressourcen beschränktes Strukturlernen tiefer Netzwerke. Diese Verfahren wird in dieser Arbeit mit einer Kombination aus zwei anderen Verfahren verglichen. Diese Kombination besteht aus einem Verfahren, welches das Netz währenddem Training beschneidet und einem Verfahren, welches das Netz breiter oder tiefer machen kann.

1.2 Ziel der Arbeit

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.
Unterkapitel Suche Literatur 2.3
PruneTrain 2.4

Net2Net 2.5
MorphNet 2.7
Automatische Architektursuche 2.6
Zeitsparen: 2.8
Additive Methoden 2.9
Experimente:3
Setup der Experimente 3.1
PruneTrain Experimente: 5.1
Net2Net Experimente: 6.2
MorphNet: 4
Net2Net + PruneTrain: 6.3
Evaluation: 9
Fazit: 10

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 vorgetellt.

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 selber ein eigenes trainierbares Gewicht haben. Aus diesen trainierbaren Werten wird mittels einer Matrixmultiplikation mit den Eingabedaten bzw. den Daten der vorherigen Schicht die Ausgabe jedes Neurons berechnet. Im Gegensatz dazu sind CNNs neuronale Netze, die in mindestens einer ihrer Schichten die Faltung anstelle der allgemeinen Matrixmultiplikation verwenden.
Dies bedeutet, dass die Eingabedaten für ein CNN für diese Faltung geeignet sein müssen. Geeignet für die Faltung sind Eingabedaten, die gridförmig angeordnet sind. Bilddaten sind ein grosser Anwendungsbereich für CNNs.
Bei der Faltung wird auf die Eingabedaten 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

[1] Die 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.

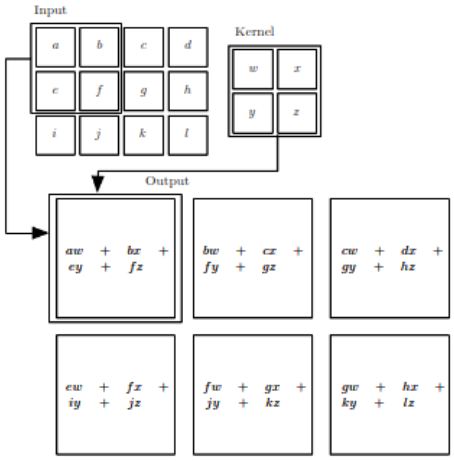


Abbildung 2.1: Abbildung zur Faltung [GBC16]

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 Merkmalskartren werden im nächsten Schritt komponentenweise als Eingabe für eine Aktivierungsfunktion benutzt. In Abbildung 2.2 wird ReLU als Aktivierungsfunktion benutzt[2]. Pooling wird verwendet umon Euch

Um die Größe der Merkmalskarte zu reduzieren kann nach dem Anwenden der Aktivierungsfunktion eine Pooling Operation eingeführt werden. Durch diee Verkleinerung der Merkmalskarte wird es weniger wichtig wo genau ein Merkmal in den Daten autaucht. 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 benutzt 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 ausgesetzt.

[2]Für Erklärung Relu siehe [Hay98]

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

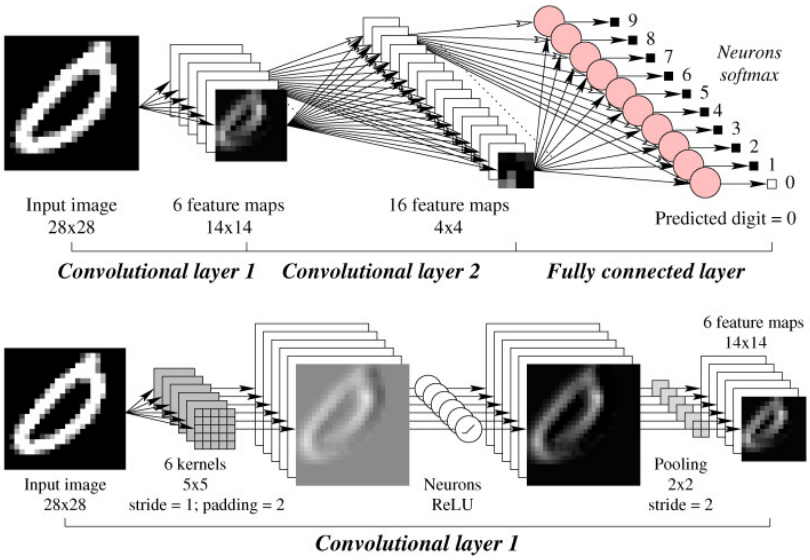


Abbildung 2.2: Convolutional Neural Net [CCGS16]

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 Patz 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

$$\mathcal{W} = \left\{W^1, \ldots W^J\right\} \tag{2.1}$$

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 das 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 $f(\mathbf{x}_i, \mathcal{W})$ entfernet ist. Wie gut die Trainingsdaten bei dieser Verlust-Funktion abschneiden wird Trainingsfehler genannt.
Die Ableitung dieser Verlust-Funktion wird rückwärts durch das Netz propagiert und damit ein Gradient berechnet. Mittels des Gradientenabstiegsverfahren wird die Verlust-Funktion minimiert, was dazu führt dass das Netz die Trainingsbilder besser klassifiziert.
Im Anschluss an diesen Trainingsprozess können Bilder, die ohne zugehörige 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 Bilder 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 allerdings nur der Testfehler hoch, so ist die Generalisierungsfähigkeit des Netzes nicht gut.
Eine weitere Technik, die zur Verbesserung der Generalisierungsfähigkeit sorgen kann ist die Batchnormalisierung. Diese Technik wird bis zum Ende dieses Unterkapitels betrachtet [IS15]. Beim Training eines CNNs ändert sich die Verteilung der Eingabewerte währenddes Trainings durch Veränderung der Gewichte der vorherigen Schicht . Dies führt zu einem langsameren Training, da es kleinere Lernraten und damit mehr Durchläufe braucht damit das Netz konvergiert. Dieses Phänomen wird interne Kovarianzeverschiebung genannt und wird durch eine Normalisierung gelöst. In Algorithmus 2.1 ist zu sehen wie dies schrittweise passiert.
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 Elemente des aktuellen Batches $\mathcal{B}$ der Mittelwert $\mu_{i,\mathcal{B}}$ berechnet. Mit Hilfe dieses Wert 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 haben. 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: $\left\{\mathbf{v}_{(i,j)} = \mathrm{BN}_{\gamma,\beta}(u_{i.j}\right\}$

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

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

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

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

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 Verwen-

dung von Residualen Netzen in der Wissenschaft ist [HZRS15].
Neuere CNN-Architekturen schaffen es dieses Problem zu vermeiden. Ein 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 Idenditätsfunktionen sind, zu lernen.

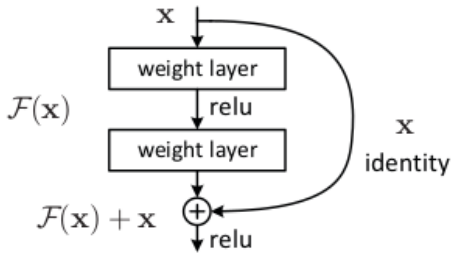
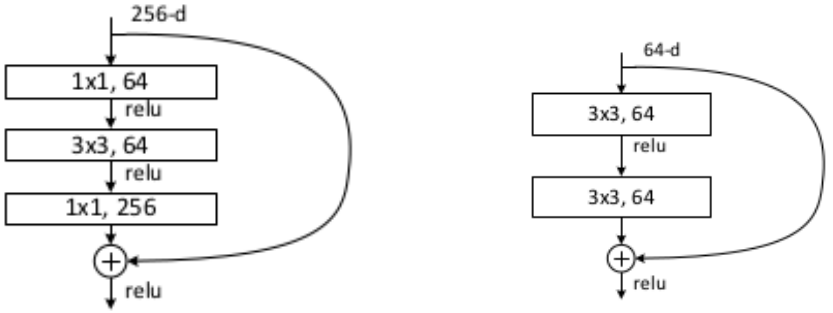


Abbildung 2.3: Abbildung der Kurzschlussverbindung [HZRS15]

Vermieden wird damit im Vergleich zu einem klassischen CNN das Problem des verschwindenden Gradientens. 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 gewünschten 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 das Wegfallen dieses Problems lassen sich mit Residualen Netzen bessere Traningsfehler 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. Das 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.



(a) Bottleneck **(b)** Nicht-Bottleneck

Abbildung 2.4: Vergleich zweier Residual Netz Blöcke [HZRS15]

2.3 Vorgehen zur Suche nachdem 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 arvix.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

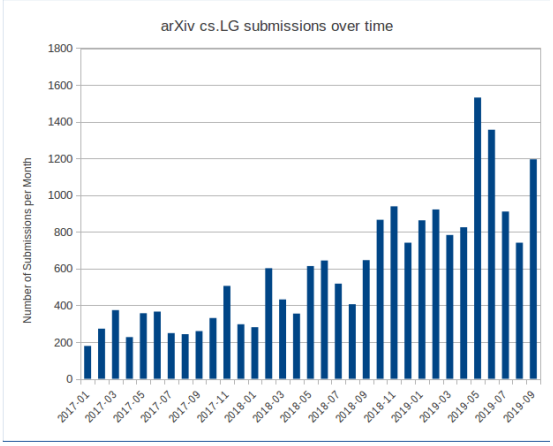


Abbildung 2.5: Tägliche Submissionen der Kategory Machine Learning auf arxiv [oA19]

Schrift gezeigt. 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 vorheigen Ebene zitieren. In den weiteren Unterkapiteln werden die so gefundenen Paper vorgestellt und die verwendeten Methoden erklärt.

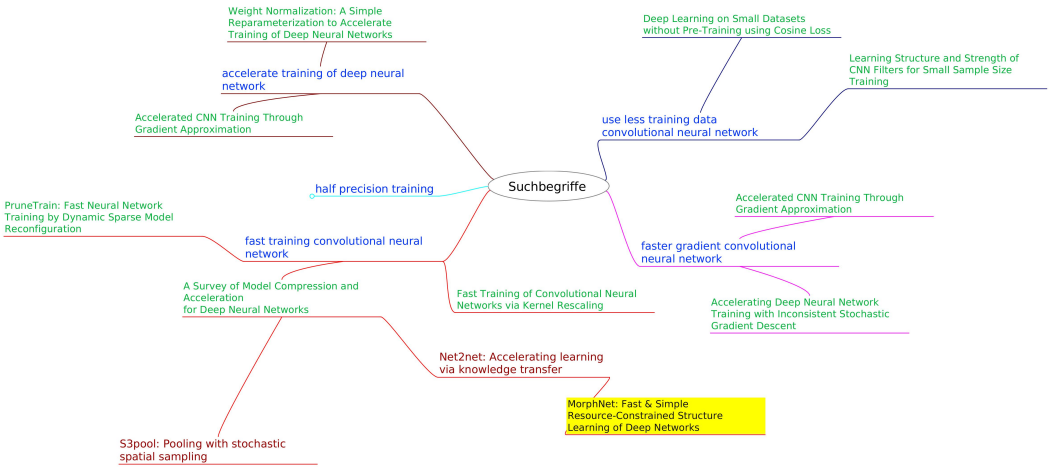


Abbildung 2.6: Mindmap zu den Suchbegriffen bezüglich des aktuellen wissenschaftlichen Stands

2.4 Beschneidung des Netzes zur Beschleunigung des Training

Das Beschneiden[3] 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 in diesem Unterkapitel erläutert werden. Als Quelle für dieses Unterkapitel dient ein Paper, welches evaluiert inwiefern Trainingszeit mittels Beschneiden gespart werden kann [LCZ+19].

Das Ziel des Beschneidens während dem Training 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 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:

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

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 ×Eingangskanäle ×Kerneldimension 1 ×Kerneldimension 2. J gibt an über wie viele Layer der Gruppen-Lasso Term berechnet wird. k_j ist die Laufvariable über die einzelnen Eingangskänale 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 aufeinander addiert. Es wird abhängig von der Initialbelegung der Gewichte ein Parameter λ berechnet, der Gruppen-

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

Lasso und Verlust-Funktion balanciert:

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

Die Größe LPR ist hier echt zwischen Null und Eins wählbar. Je größer sie gewählt wird, desto größer ist der Anteil, der beschnitten wird. Regelmäßig werden während dem Trainieren 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.

Grafik zum Verfahren

Zu diesem Zweck wird das Kanal-Union Verfahren eingeführt. Beim Kanal-Union Verfahren wird ein Liste mit den Layern geführt, die aufeinander abgestimmt werden müssen. Im Falle eines residualen Netzes muss zusätzlich eine Liste über die zusammengehörigen Layer der Kurzschlussverbindungen geführt werden. Im nächsten Schritt werden alle Eingangs- und Ausgangskanäle, die noch Gewichte größer Null haben in einer Liste gesammelt. Auf 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.

Noch genauer formulieren

Bei einem residualen Netzwerk ist es weiterhin möglich, dass ein ganzer Block wegfällt. In diesem Fall müssen die Kanal-Union Listen angepasst werden und es wird mit einem ohne diesen Block um mehrere Layer verkürztes Netzwerk weitergemacht.

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 und wird in Kapitel **??** erläutert [ptI]. Hierbei wird

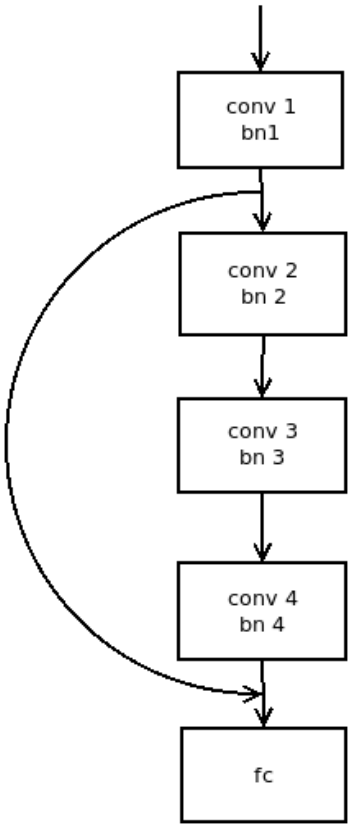


Abbildung 2.7: Beispielnetz

Liste der Layer, die direkt hintereinander sind und aufeinander abgestimmt werden müssen (ohne das zwischen den Layern eine Kurzschlussverbindung enstammt oder endet): $H = \{(2,3),(3,4)\}$
Liste der Layer, die im Zusammenhang mit den Kurzschlussverbindungen die gleiche Eingangskanaldimension haben müssen: $E = \{(2, fc\}$
Liste der Layer, die im Zusammenhang mit den Kurzschlussverbindungen die gleiche Ausgangskanaldimension haben müssen: $A = \{(1,4)\}$
Liste der dicht-besetzten Schichten vor dem ersten Beschneiden:
$1 : \{0,1,2\}, \{0,1,2,3,4,5,6,7\}$
$2 : \{0,1,2,3,4,5,6,7\}, \{0,1,2,3,4,5,6,7\}$
$3 : \{0,1,2,3,4,5,6,7\}, \{0,1,2,3,4,5,6,7\}$
$4 : \{0,1,2,3,4,5,6,7\}, \{0,1,2,3,4,5,6,7\}$
$fc : \{0,1,2,3,4,5,6\}, \{0,1,2,3,4,5,6,7,8,9,10\}$

Liste der dicht-besetzten (db) Schichten nach dem auf Null setzen der Parameter aber vor der Rekonfiguration:
$1 : \{0,1,2\}, \{0,1,4,5,6,7\}$
$2 : \{0,1,3,5,6,7\}, \{0,1,2,4,5,6,7\}$
$3 : \{0,1,2,4,5,6,7\}, \{0,1,2,3,4,5,6,7\}$
$4 : \{0,1,2,3,4,6,7\}, \{0,1,3,4,5,6,7\}$
$fc : \{0,1,3,4,5,6\}, \{0,1,2,3,4,5,6,7,8,9,10\}$

Vorgehen des Kanal Union Verfahren: Als ersten 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
$3,4 : A(3) \cup E(4) = \{0,1,2,3,4,5,6,7\} \cup : \{0,1,2,3,4,6,7\} = \{0,1,2,3,4,5,6,7\}$
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 wir in den Ausgangskanälen von Schicht 1 und 2 sowie in den Eingangskanälen von Schicht 2 und fc jeweils der 2. Kanal entfernt.

Abbildung 2.8: Beispiel für das Kanal-Union Verfahren

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 aber 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[**?**]. 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 währenddem Training 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.

2.5 Beschleunigung des Lernens durch Wissenstransfer

Beim Trainieren eines CNNs kommt es häufig vor, dass nach initialem Wählen der Tiefe bzw. 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.

Wie in Abbildung 2.9 abgebildet ist lässt sich so der Workflow zum Finden der passenden Netzstruktur anders gestalten. Der Net2Net Operator macht dabei das Netz entweder breiter (mehr Kanäle in bestimmten Schichten) oder tiefer (zusätzliche Schichten). Diese beiden Operatoren werden nun vorgestellt.

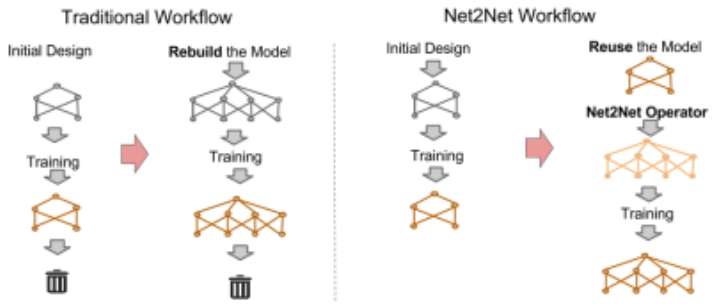


Abbildung 2.9: Traditioneller Workflow vs. Net2Net Workflow

2.5.1 Operator für breiteres Netz

Bei dem Net2Net-Operator, der das Netz breiter macht werden für eine bestimmte Schicht Ausgangskanäle und für die nachfolgende Schicht Eingangskanäle hinzugefügt. Die Schicht, der 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 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 2.10 abgebildet ist $q \cdot l$ zusätzlichen Filterkerneln.

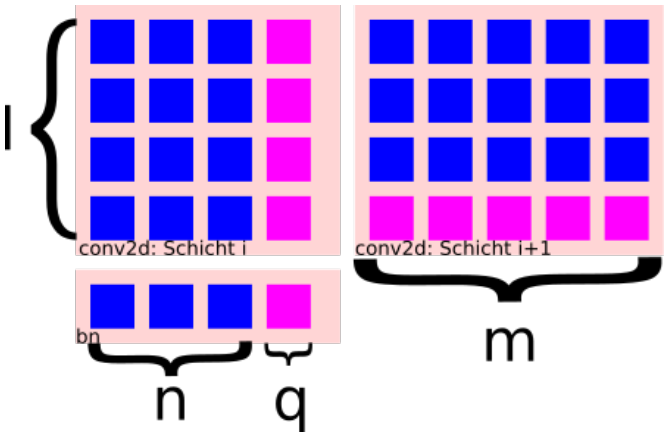


Abbildung 2.10: Übersicht über die zusätzlichen Kanäle todo: Grafik schön machen

Für das Layer $j+1$ sind es entsprechend $q \cdot m$ zusätzliche Kernel. Die Gewichtstensoren nachdem 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

ein zufällige Belegung der zusätzlichen Kernels sorgt:

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

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

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

Die Funktion $g(j)$ wird dabei für jede neu hinzugekommene Schicht nur einmal ausgewertet, sodass gesamte Reihen statt einzelne 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 $\mathbf{v}_{i,j} = \mathrm{BN}_{\gamma,\beta}(\mathbf{v}_{i,j} * \mathbf{W}_j)$ durch die Operation von zwei Layern

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

$\mathbf{U}$ 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 letzen 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.
Bei dem 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 Ergbnis 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 [**?**]. Dabei wird initial ein Population von Netzen gebildet [**?**]. Nach einem Trainingsdurchgang werden diese nach ihrer Fitness (Klassifikationsleistung) selektiert [**?**]. Im weiteren Verlauf werden jeweils zwei dieser Netze gepaart und es entsteht eine neue Generation an Netzen [**?**]. Allerdings ist hier die Trainingszeit in einem Rahmen von 17 GPUs für einen Tag [**?**]. Dabei ist die Architktursuche hier nur partiell automatisiert [**?**].
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. 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 [GEN+18].
Die manuelle Wahl von Hyperparametern, die bestimmen wie groß und komplex ein neuronales Netz ist, braucht eine gewisse Kunstfertigkeit. Sind die Hyperparameter falsch gewählt, so müssen diese angepasst und das Netz erneut trainiert werden. Mit Hilfe der hier vorgestellten Methode wird diese 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 Größen für die 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:

$$\mathcal{W}^* = \underset{\mathcal{F}(\mathcal{C}_{1:J}) \leq \zeta}{\arg\min} \min_{\mathcal{W}} l(f(\mathbf{x_i}, \mathcal{W}), y_i) \tag{2.6}$$

Das Vergrößern des Netzes basiert auf einer Lösung für die Gleichung 2.6: dem Breitenmultiplikator ω. Sei $\omega \cdot O_{1:M} = \{\lfloor \omega O_1 \rfloor, \lfloor \omega O_2 \rfloor, \ldots, \lfloor \omega O_M \rfloor\}, \omega > 0$. 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 größ der Anteil eines Netzbestandteiles an $\mathcal{F}(\mathcal{C}_{1:J})$ misst und es damit direkt optimieren kann. Dann ist

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

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. Es entsteht folgender Algorithmus:

1: Trainiere das Netz um $\mathcal{W}^* = \underset{\mathcal{W}}{argmin} l(f(\mathbf{x_i}, \mathcal{W}), y_i) + \lambda \mathcal{G}(\mathcal{W})$ 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 Nebenbedingungen, die Modellgröße wird hier beispielhaft erläutert. Die Größe dieser Nebenbedingung wird vorallem 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 Ausgangskänäle sowie der

Filtergrösse ist und nicht von der Position im Netzwerk ist, lässt sich $\mathcal{F}(\mathcal{C}_{1:J})$ auf die einzelnen Schichten zurückführen. Es gilt:

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

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

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

Da auf durch die Anwendung des Regularisierers einzelne Kanäle auf Null gesetzt werden und ein Netz ohne diesen Kanal möglich wären sollen diese Kanäle in dieser Berechnung ausgelassen werden. Daher wird die Formel um Aktivierungsfunktionen $A_{k_l,j}$ 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:

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

2.7.2 Regularsierer

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:

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

Mit dieser Funktion lässt mittels Gradientenabstieg lernen, obwohl Teile des Regularisieres nicht komplett kontinuierlich sind. γ muss dabei kontiuierlich sein. Werden die γ für einen Kanal auf Null gesetzt durch das Lernen, so ist der dazu-

gehörige Kanal aus der Berechnung wie gewünscht ausgeschlossen. Für jedes Ein- und Ausgangskanal einer Schicht wird ein γ in den Vorwärtsdurchgang eingebaut. Diese Parameter funktionieren dann analog zu den γ aus der Batchnormalisierung, da sie kontrollieren wieviel Prozent eines Kanals weitergeleitet wird.
Aus dem Regularisierer einer Schicht lässt sich mittels Addition die Regularisierung des kompletten Netzes berechnen.

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

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

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

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

2.8 Zeitsparende Nethoden

2.8.1 Verringerung der für Berechnungen nötige Zeit

Überarbeiten, fertigschreiben; 3 Stunden

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

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

Beim Verändern der Filter- oder der Bildgrösse, um Trainingszeit zu sparen, verändert sich auch die Erkennungsleistung . Dies ist beim Verändern des verwen-

cite

deten Zahlenformats nicht umbedingt gegeben. Standardformat ist eine 32 Bit Gleitkommazahl. Die einfachste Methode hier Trainingszeit zu sparen ist das Halbieren der Bitanzahl auf 16 Bit. Eine weitere Methode ist das Benutzen von 16 Bit Dynamischen Festkommazahlen. Die beiden alternativen Methoden haben unterschiedliche Anforderungen an die Ausführungsplattform. Diese Anforderungen und die Besonderheiten der beiden Verfahren werden in den folgenden zwei Unterkapiteln näher beleuchtet.

Berechnung mit 16 Bit Gleitkomma [iee85] Die 16 Bit Gleitkommazahl unterscheidet sich nicht nur in der Länge von der 32 Bit Zahl sondern aus der unterschiedlichen Länge erwachsen Unterschiede in den darstellbaren Zahlen. In Tabelle 2.1 sind diese Unterschiede dargestellt.

Tabelle 2.1: Darstellbare Zahlen von 16 und 32 Bit

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

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

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

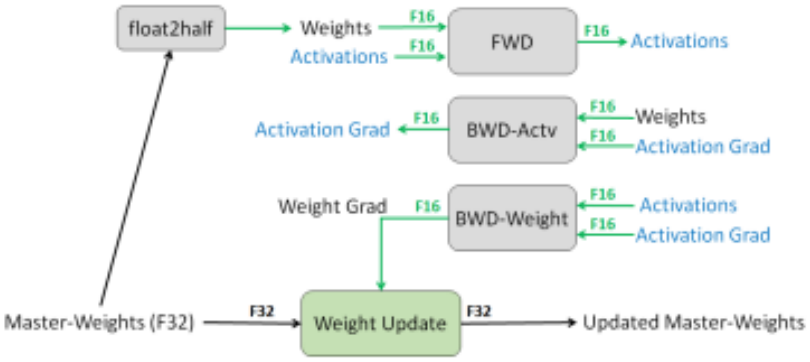
- 32 Bit Mastergewichte und Updates
- Sklaierung der Loss-Funktion
- Arithmetische Präzision

Beim Trainieren von neuronalen Netzwerken mit 16 Bit Gleitkommazahlen werden die Gewichte, Aktivierungen und Gradienten im 16 Bit Format gespeichert. Die Speicherung der Gewichte als 32 Bit Mastergewichte hat zwei mögliche Erklärungen, die aber nicht immer zutreffen müssen.
Um nach einem Forward Druchlauf des Netzes die Gewichte abzudaten wird ein Gradientenabstiegsverfahren benutzt. Hierbei werden die Gradienten der Gewichte berechnet. Um für die Funktion, die das CNN approximiert einen besseren

Approximationserfolg zu erlangen wird dann dieser Gradient mit der Lernrate multipliziert. Wird dieses Produkt in 16 Bit abgespeichert, so ist in viele Fällen das Produkt der beiden Zahlen gleich Null. Dies liegt an der Taqtsache, dass wie in Tabelle ref 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

2.9 Additive Methoden

2.9.1 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 2.11 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.

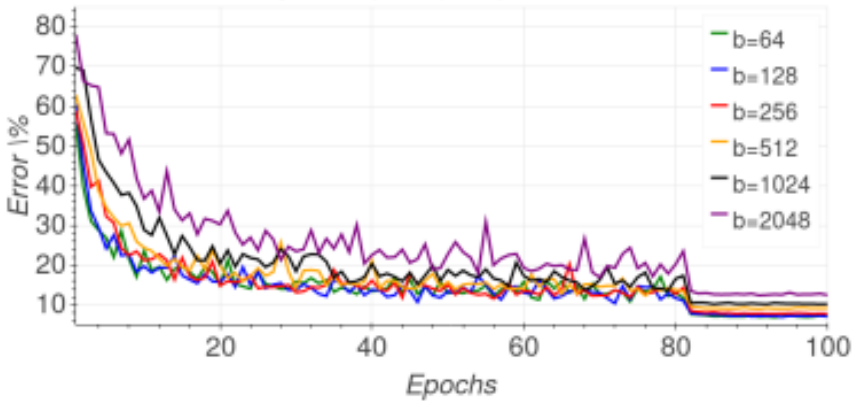


Abbildung 2.11: Validation Accuracy von CNNs anhand eines Beispieldatensatzes

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 Detail s zu fehlen.
Eine alternative ist Lars. (LARGE BATCH TRAINING OF CONVOLUTIONAL NET - WORKS WITH LAYER - WISE A DAPTIVE R ATE 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 Konzeptionelle Übersicht – Arbeitstitel

In diesem Kapitel werden verschiedenen Methoden, die in Kapitel 2 Vorgestellt wurden, auf einer Grafikkarte ausgeführt, evaluiert und die Ergebnisse verglichen. In Kapitel 3.1 werden zunächst das experimentelle Setup vorgestellt. In Kapitel 5.1 wird das Beschneiden des Netzes zunächst so getestet, wie es in der vorgefertigten Implementierung hinterlegt ist [ptI]. In Kapitel **??** wird dann untersucht wie sich das Beschneiden des Netzes mit zusätzlicher Anpassung der Batchgrösse verhält.
Die beiden Operatorn, die das Netz tiefer beziehungsweise breiter machen werden in Kapitel 6.2 durch Experimente evaluiert. In Kapitel 6.3 werden dann die Operatoren für ein breiteres Netz beziehungsweise ein tieferes Netz mit der Technik des Beschneidens der Netze kombiniert, um ein automatisches Suchen einer besseren Architektur zu ermöglichen. Diese Verbindung wird dann mit dem Schnellen Ressourcen beschränktem Strukturlernen tiefer Netze verglichen. Das Strukturlernen wird dafür zunächst in Kapitel 4 evaluiert. Der Vergleich der Ergebnisse wird dann in Kapitel 7 gemacht.
Zuletzt werden noch additive Verfahren vorgetellt, welche die Trainingszeit zusätzlich minimieren können. Eine dieser Verfahren, welches in Kapitel 8.0.1 evaluiert wird, spart Zeit durch die Verwendung von gemischt präzisen Zahlenformaten. Ein weiteres additives Verfahren in Kapitel 8.0.2 überprüft in wiefern mit Hilfe einer adaptiven Anpassung der Lernrate die Batchgröße sinvoll so angepasst werden kann, dass die ganze GPU genutzt werden kann.

3.1 Experimentales Setup

In diesem Kapitel wird das experimentelle Setup vorgestellt, um die Ergebnisse einfach nachvollziehen zu können.

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 Flexibiltät 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 unmgehen 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 angeben 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
- $[n_1, \ldots, n_S]$: Anzahl von Blöcken pro Phase
- l: Anzahl von (Conv+Batch)-Layer pro Block
- $[k_1, \ldots, k_S]$: Breite der Schichten je Phase
- b: Boolean Parameter, der angibt ob die Blöcke im Netz die Bootleneck-Eigenschaft haben

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

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 netz hat drei Phasen, wobei jeder der Phasen 5 Blöcke hat. Pro Block sind zwei (Conv+Batch)-Schichten vorhanden. Die Breite der Schichten sind $[16, 32, 64]$

3.2 Konzept

Phase	Schicht/Block	#Eingangskanäle	#Ausgangskanäle
	Conv 1 + Bn 1	3	16
1	Basisblock	16	16
	Basisblock	16	16
	Basisblock	16	16
	Basisblock	16	16
	Basisblock	16	16
2	Übergangsblock	16	32
	Basisblock	32	32
	Basisblock	32	32
	BasisBlock	32	32
	BasisBlock	32	32
3	Übergangsblock	32	64
	Basisblock	64	64
	Basisblock	64	64
	Basisblock	64	64
	Basisblock	64	64
	Linear	64	10

Tabelle 3.1: Struktur des Netzes

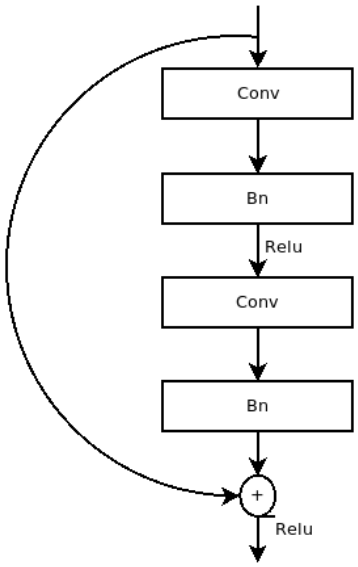


Abbildung 3.1: Basisblock

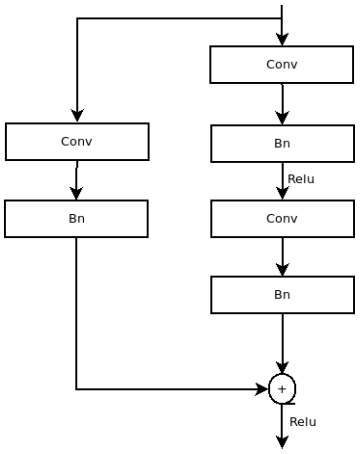


Abbildung 3.2: Übergangsblock

4 Untersuchung von MorphNet

untersuche zunächst ob für die beiden Constraints Model Size und Flop regularisieren auch diese Zielgrösse kleiner wird.
2 Experimente Zeit ??
Erprobe wie unterschiedliche gammas auf Schritt 2 wirken
5 x 5 Experimente
Danach wird evaluiert wie aus einem ResNet 32 mit Anfangsbreite ein grösseres Netz wird. Erlaubt sind maximal die Flops/bzw. Modelgrösse des 16 breiten Netzwerks mit 3 Iterationen

5 PruneTrain

5.1 Untersuchung von PruneTrain

Ergebnisse aus neuen Experimenten verarbeiten; wenn Experimente fertig; 5 Stunden

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 dem Prune Train Paper verglichen. Im ersten Abschnitt wird zunächst betrachtet, wie sich die Trainingszeiten bei den veränderbaren Hyperparametern von PruneTrain verändern. Im zweiten Abschnitt wird betrachtet, wie sich die Accuracy im Verlauf der Epochen verhält.
verwendete Kenngrößen:

- 1 GPU
- Cifar10
- 180 Epochen

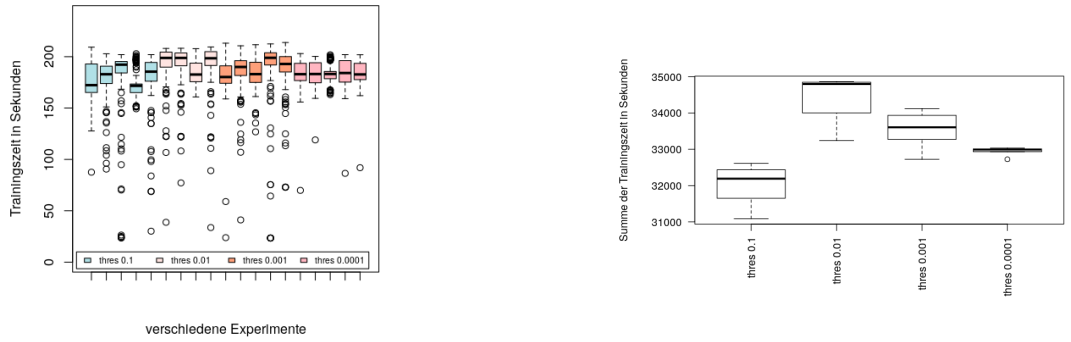
Variable Größen, die in verschiedenen Experimenten geändert werden:

- Lasso-Ratio
- Threshold experimente
- Lernrate experimente
- Rekonfigurationsinterval fertig

Um die Experimente mit den unterschiedlich großen Kenngrößen vergleichen zu können wird jeweils eine Größe geändert und der Einfluss dieser Größe auf die Trainingszeit betrachtet. Betrachte zunächst eine feste Batchgröße von 256 über alle 180 Epochen und vergleiche diese mit mehreren Durchläufen des Baseline-Netzes.

Experimente zur Lernrate

Die Trainingszeiten in Sekunden pro Epoche für verschiedene Lernrate ist in Abbildung 5.1 zu sehen. Unterschiede zwischen den Experimente sind in Abbildung 5.1a zunächst nicht direkt sichtbar. Um diese besser sichtbar zu machen wird für jedes Experimente die Summeder Trainingszeiten über die Epochen gebildet. In Abbildung 5.1b ist zu beobachten, dass beim größten Threshold die geringste summierte Trainingszeit zusammenkommt. Da bei einem höheren Threshold im Laufe des Trainings mehr Gewichte unter den Grenzwert fallen und somit entfernt werden. Liegen diese Gewichte sinnvoll, so können ganze Kanäle entfernt werden.



(a) verschiedene Experimente **(b)** Summe verschiedener Experimentengruppen

Abbildung 5.1: Boxplot der verschiedenen Grenzwerte

In Abbildung 5.2 ist zu sehen, dass der gleiche Effekt sich auch bei Accuracy zeigt. Die Accuracy für den größten Grenzwert ist am niedrigsten.

Experimente zum Rekonfigurationsintervall

Als nächste Größe wird der Einfluss des Rekonfigurationsintervalls überprüft. Die entsprechenden Grafiken sind in Abbildung 5.3 abgebildet. In Abbildung 5.3a sind für die verschiedenen Experimente die Trainingszeiten pro Epoche zu sehen. Dabei werden drei verschiedene Rekonfigurationsintervalle (2,5 und 10) verglichen.

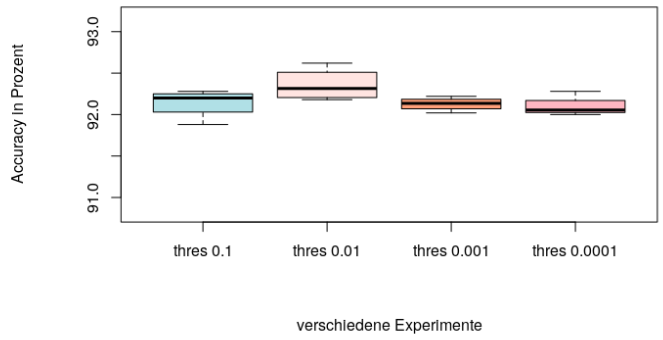
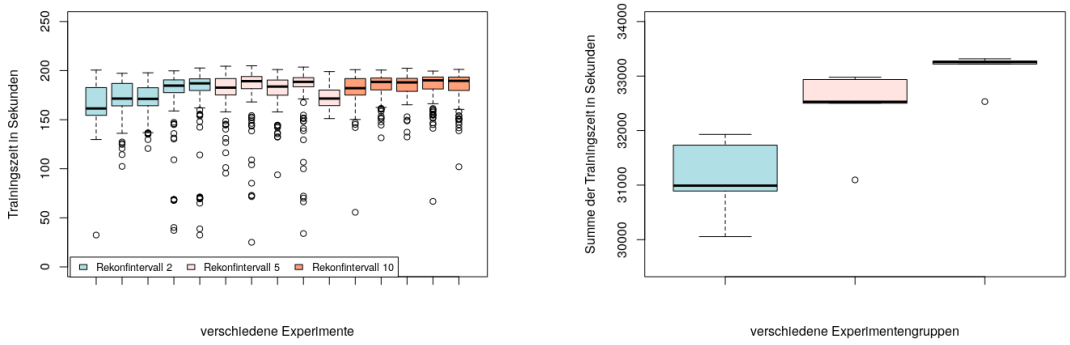


Abbildung 5.2: Accuracy von verschiedenen Experimentengruppen des Grenzwerts

In Abbildung 5.3a lässt sich für die verschiedenen Experimente keine großen Unterschiede sehen. Werden die Zeiten der jeweiligen Experimente addiert und in einem Boxplot dargestellt entsteht Abbildung 5.3b. In dieser Abbildung ist deutlich zu sehen, dass mit steigendem Rekonfigurationsintervall auch die Summe der Trainingszeiten pro Epoche steigt.



(a) verschiedene Experimente

(b) Summe verschiedener Experimentengruppen

Abbildung 5.3: Boxplot der Rekonfigurationsintervalle

Dies bedeutet, dass der Overhead des Beschneidungsverfahrens geringer ist als der Gewinn durch das Verkleinern des Netzes. In Abbildung **??** ist zu sehen, dass dieser Gewinn an Trainingszeit in Abbildung 5.3b mit einem Verlust an geringem Verlust an Accuracy einhergeht.

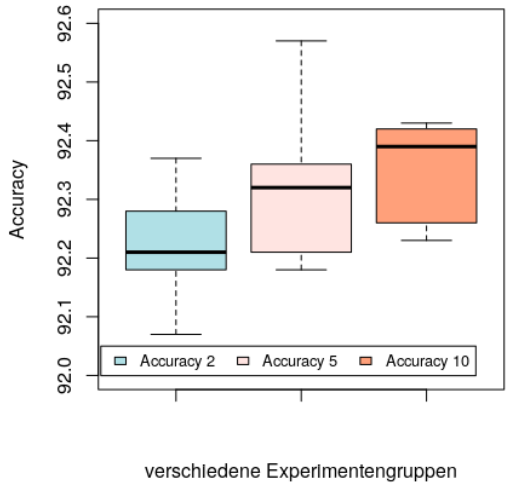
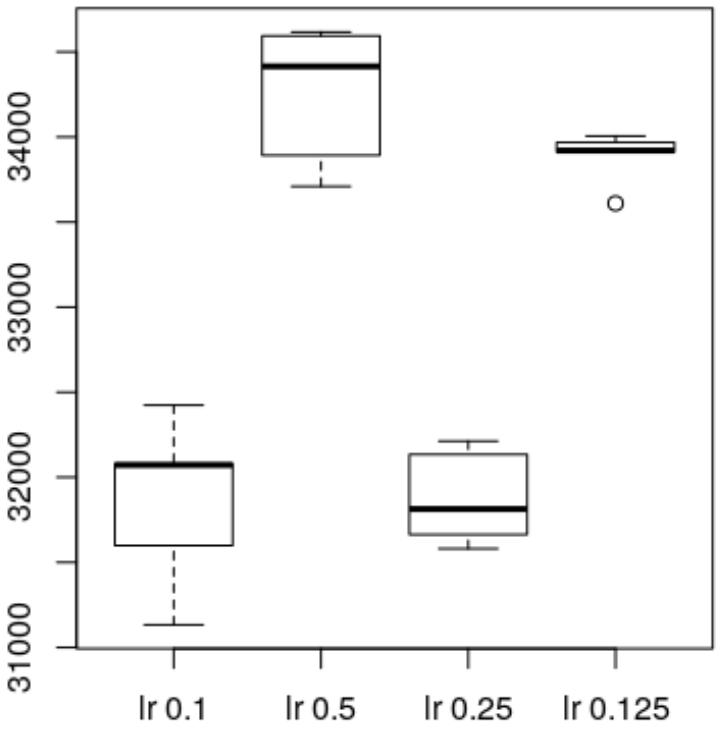


Abbildung 5.4: Accuracy von verschiedenen Experimentengruppen des Rekonfigurationsintervall

Experimente zum Grenzwert

In Abbildung **??** ist nicht wirklich was zu sehen,
Es ergibt sich, dass sich mit diesen bisherigen Experimenten keine Zeit sparen lässt. Im Gegenteil, der PruneTrain Ansatz braucht mehr Zeit. Dies steht im Widerspruch zum PruneTrain Paper. Dieser Widerspruch lässt sich durch die Verwendung von mehreren GPUs zur Evaluation im PruneTrain Paper erklären. Mit einem schmalleren Netz müssen weniger Daten zwischen den GPUs ausgetauscht werden. Es wird Kommunikationszeit gespart.
Für die Evaluation des Beschneiden des Netzes werden in der Original-Veröffentlichung mehrere GPUs verwendet. Dies führt dazu, dass bereits in diesem Teil der Implementierung Trainingszeit durch verminderte Kommunikation zwischen den Grafikkarten gespart wird. Da hier nur mit einer GPU evaluiert wird ergibt sich hier noch keine direkte Einsparung an Trainingszeit. Die Einsparung ergibt sich erst durch Erhöhen der Batchgröße bei kleiner werdendem Netz. Zu beachten ist hier, dass die Speicherauslastung gleich bleiben muss. Diese Evaluierung wird in Kapitel 6.1 durchgeführt.


34000
33000
32000
31000
lr 0.1
lr 0.5
lr 0.25
lr 0.125

6 Untersuchung der eigenen Implementierungen

6.1 Experimente zur Anpassung der Batchgröße beim Beschneiden des Netzes

Um die Batchgrösse mit der Netzverkleinerung anzupassen muss zunächst die maximale SPeicheraulastung und damit die initiale Batchgröße festgelegt werden. In Abbildung 6.1 ist zu sehen, wie sich die Speicherauslastung in Abhängigkeit zur Batchgröße verhält.

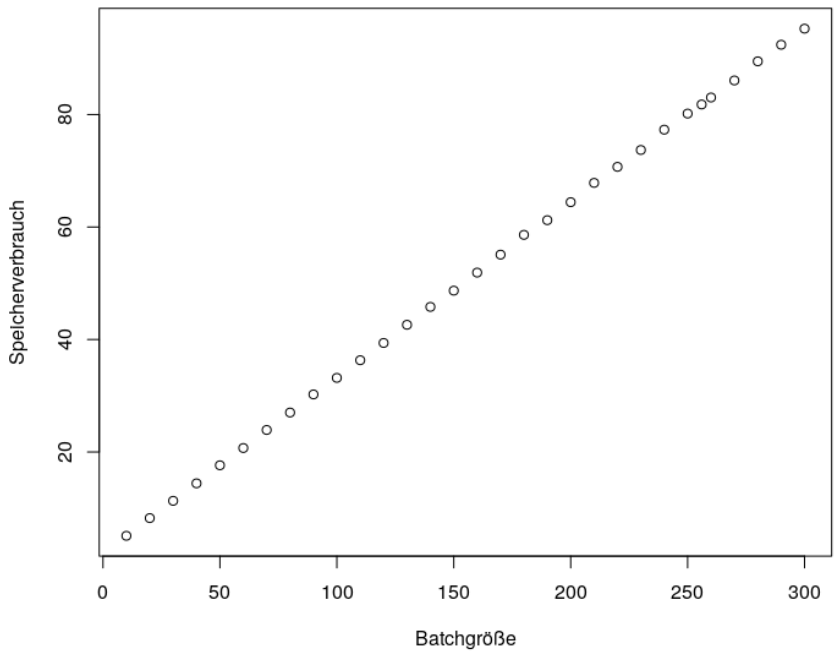


Abbildung 6.1: Zusammenhang Batchgröße und Speicherverbrauch

Dafür wird zunächst untersucht wie sich die Speichernutzung verändert bei unter-

	s=1		s=2		s=3	
	#Para	Batch	#Para	Batch	#Para	Batch
1	5306	506	24090	506		
2	9978	435	47322	435	434	
3	14650	382	70554	3382		
4	19322	339	93786	339	390170	301
5	23994	304	117018	339	487386	256
6	28666	277	140250	256	584602	223
7	33338	256	163482	229	681818	198
8	38010	236	186714	204	779034	175
9	42682	218	209946	184	876250	161
10	47354	205	233178	171	973466	147

schiedlich großen Batchgrößen aber fester Netzgröße. Genutzt wird das Basisnetz aus Kapitel 3.1.4.

20. In Abbildung **??**

Gleichzeitig wird für die jeweilige Modellgrösse die Anzahl an Parametern, die das Modell hat gezählt. Diese Größen sind in Tabelle **??** eingetragen.

Mit Hilfe dieser Grössen wird für jede einzelne Stagegröße eine Gerade gefittet. Diese gefittete Gerade wird mittels t-Test darauf überprüft wie wahrscheinlich beim Fitten der Gerade ein Fehler 1. Art auftritt.

Hierfür werden folgende Hypothesen aufgestellt:

Da der p-Wert für diese Gerade bei $p = 2,911e^{-16}$ und damit weit unter de Signifikanzniveau von $\alpha = 0,05$ kann die H_0 Hypothese abgelehnt werden und die Alternativhypothese angenommen werden.

Dies bestätigt statistisch eine hohe Wahrscheinlichkeit, dass die gefittete Gerade richtig ist.

In Abbildung 6.4 ist zu sehen, dass die Parameteranzahl in Zusammenhang mit der Anzahl an Blöcken linear steigt. Wird das Netz kleiner, so kann anhand der Gerade abhängig von der Parameteranzahl die neue Batchgröße errechnet werden.

beispielrechnung

In Abbildung ist abgebildet. wie sich die Trainingszeiten verändern, wenn die Batchgrösse angepasst wird.

ref

Veränderung der Accuracy durch PruneTrain

Durch das Pruning währenddem Training wird die Accuracy kleiner. In Abbildung 6.2 ist zu sehen wie sich im Accuracy im Verlauf der Epochen verändert.

In Abbildung 6.3 sind die Epochen 90 bis 180 näher herangezoomt.

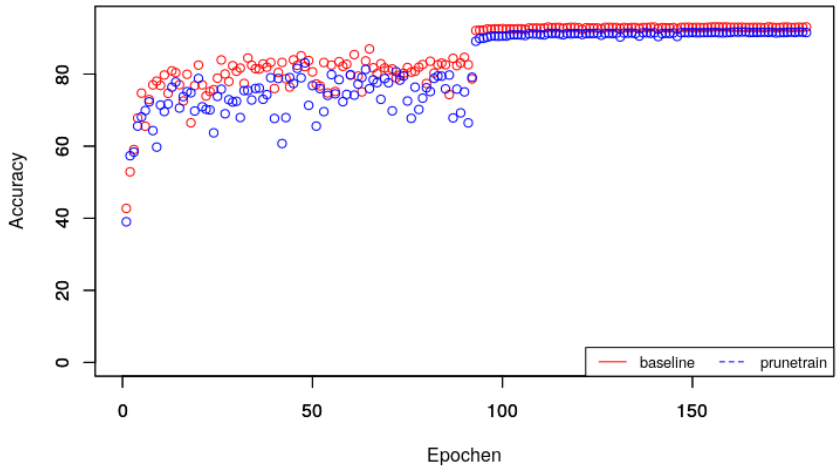


Abbildung 6.2: Veränderung der Accuracy während der Epochen

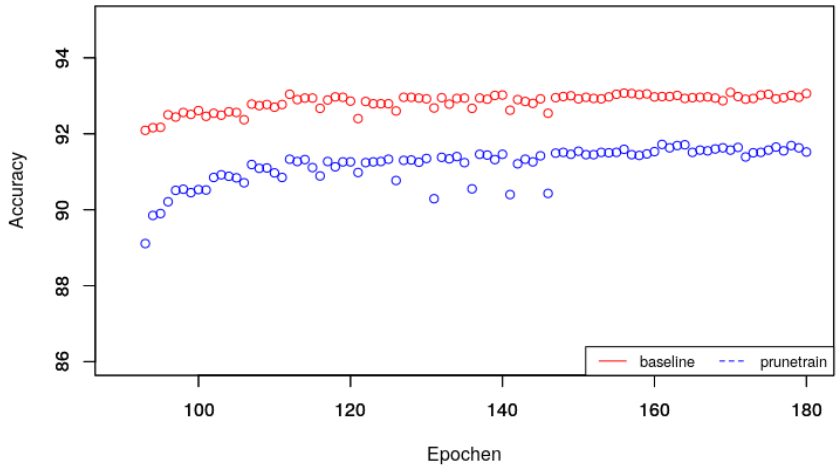


Abbildung 6.3: Zoom der Veränderung

Man sieht eine geringere Accuracy von PruneTrain im Vergleich zum Baseline. Diese Verringerung der Accuracy lässt sich durch ein Aussetzen des Verkleinern des Netzes in den letzten Epochen vermindern.

Die Frage die sich hier stellt ist, ob diese Verminderung der Accuracy am Ende der dieser Arbeit noch ins Gewicht fällt. Wenn mit Hilfe einer Kombination von MorphNet und PruneTrain ein bessere Architektur gefunden wird kann diese Architektur auch direkt mit Hilfe des äquivalenten Baseline Netzes berechnet werden.

6.1.1 Einfluss der Batchgröße auf PruneTrain

Das heisst es ist nötig, zu wissen wie gross die Batches maximal sein dürfen um keinen Out of Memory Error zu provozieren. Zusätzlich kann dann berechnet werden, inwieweit die Batchgrösse weiter angehoben werden kann bei kleiner werdendem Netz
Um die Anpassung der Batchgröße an die Verkleinerung des Netzwerkes durch das Prunen zu implmentieren muss zunächst die Batchgröße des Ausgangsnetzes so gewählt werden, dass der GPU-Speicher maximal ausgelastet ist.
Theoretisch sollte hierfür nachdem Übertragen des Modells der freie Speicher ausglesen werden und anhand des Speicherverbrauchs eines Elements des Datensatzes berechnet werden, wie gross die Batchgröße maximal sein darf. Leider führt diese Methode nicht zum gewünschten Ergebnis, da der ausgelesene freie Speicher nicht dem tratsächlich allokierbaren Speicher entspricht. Der Grund hierfür ist ein Fragmentierungsproblem. Verschiedene freie Blöcke können nicht zu einem grossen allokierbaren Block zusammengefügt werden.
Quelle
.
Diese Problem wird mit einer Methode, die für einen beliebigen Datensatz und für eine beliebige Modellgrösse die maximale Batchgröße berechnet, gelöst.

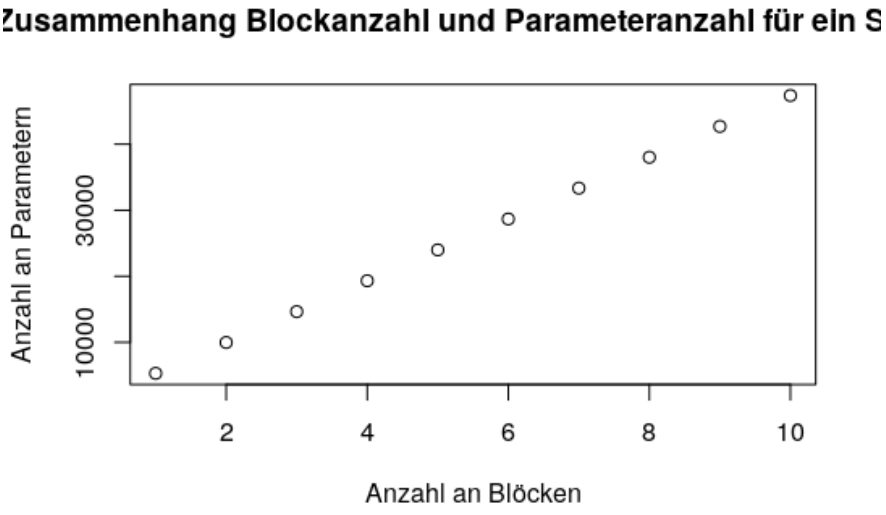


Abbildung 6.4: Batch Size vs Trainings Time über eine Epoche

Die maximal mögliche Batchgrösse in Abbildung 6.5 sinkt im Gegensatz dazu stärker als linear bei mehr Blöcken im Netz. Dies liegt darin begründet, dass

für ein grösseres Netz mehr Werte zwischengespeichert werden müssen, was den Speicherbedarf erhöht.

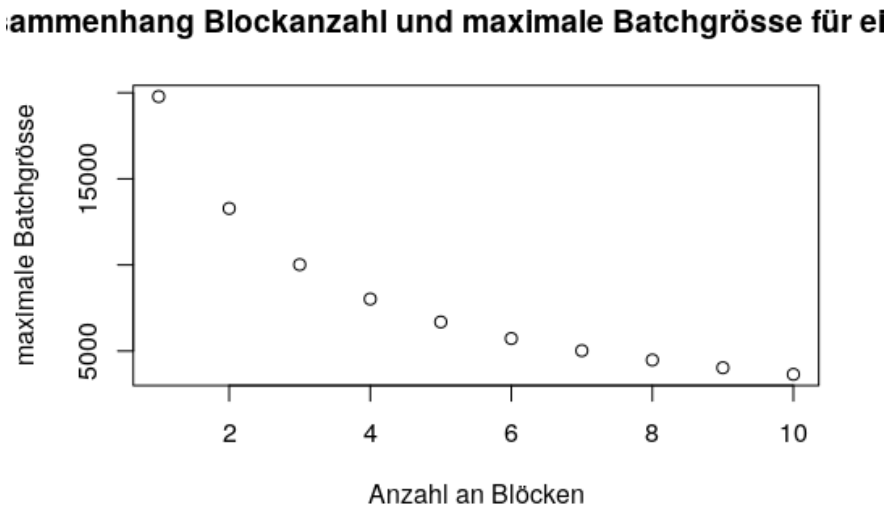


Abbildung 6.5: Batch Size vs Trainings Time über eine Epoche

Gesucht ist ein idealerweise linearer Zusammenhang zwischen der Parameteranzahl und der Batchgrösse. Um diesen herzustellen wird die Parameteranzahl durch die Batchanzahl geteilt. Das Ergebnis hiervon ist in Abbildung 6.6 zu sehen.
Da diese Kurve ähnlich der Batchsize-Kurve aussieht wird die Hypothese untersucht, ob hier ein linearer Zusammenhang besteht. Zu diesem Zweck wird die Batchgrösse durch das Ergebnis geteilt.
Augenscheinlich liegt hier ein linearer Zusammenhang vor. Daher wird hier eine Gerade gefittet. Es entsteht die Abbildung 6.7.
Die gefittete Gerade hat die Gleichung:

$$f(x) = 0.11 \cdot x + 0.15$$

Hier muss noch das Fitten des Modells und der t-Test erklärt werden

Tabelle

In Tabelle werden die Werte für die anderen Stages zusammengefasst. Zu sehen ist, dass für jeden Stage die gefittete Gerade ähnlich im t-Test abschneidet.
Als nächsten Schritt wird untersucht wie das Intervall wie häufig rekonfiguriert wird den Zusammenhang zwischen Inferenz Flop und der Validation Accuracy verändert.
Die nächste Untersuchung über das Sparen von Kommunikationskosten beim Verteilten Training macht hier keinen Sinn da nur eine einzelne Graka genutzt wird.

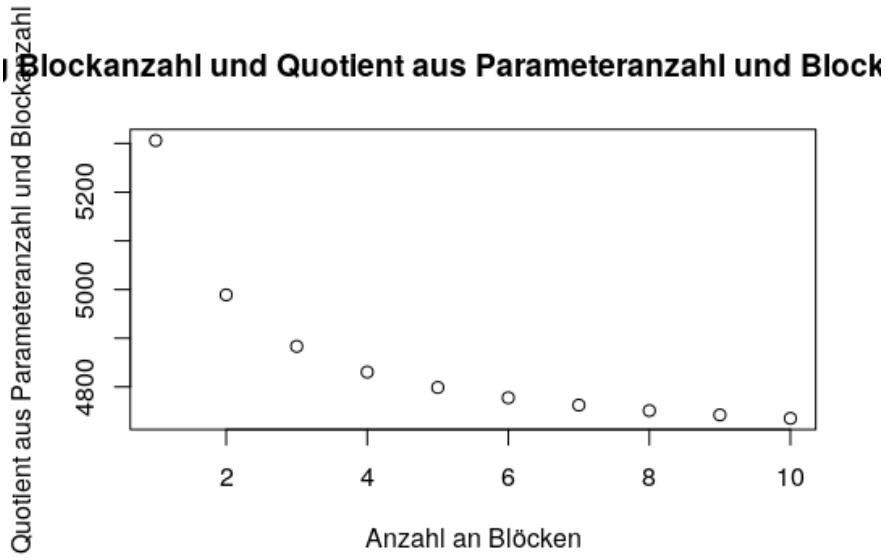


Abbildung 6.6: Batch Size vs Trainings Time über eine Epoche

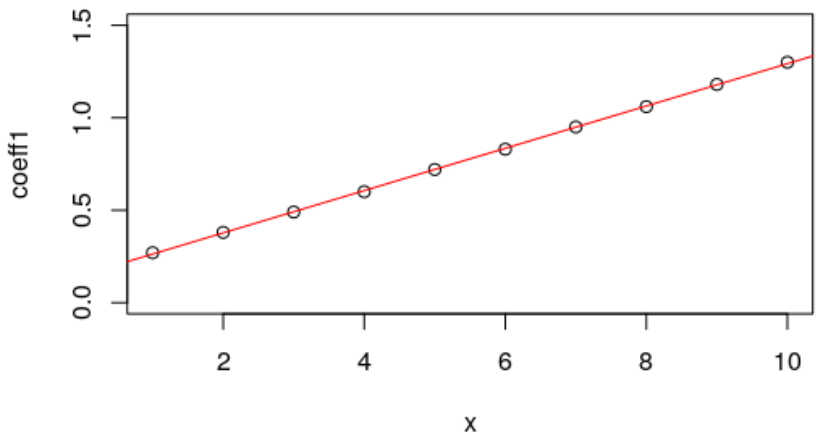


Abbildung 6.7: Batch Size vs Trainings Time über eine Epoche

Abschliessend wird noch evaluiert, wie die Dichte der Gewichte mit der Dichte der Kanäle nachdem Training zusammenhängen um eventuell durch spezifische Inferenzhardware weiter zusparen.
Bei grösserer Batchgrösse wird auch das Netz schneller. Dies ist in Abildung 6.8 für das ResNet und die verwendete Hardware abgebildet.
Wie zu sehen ist, wird die Trainingszeit pro Epoche mit grösserer Batchgrösse kleiner. Die höhere Batchgrösse sorgt neben der geringeren Trainingszeit auch für weniger Gewichtsupdates. Dies führt zu einer geringeren Generalisationsfähigkeit

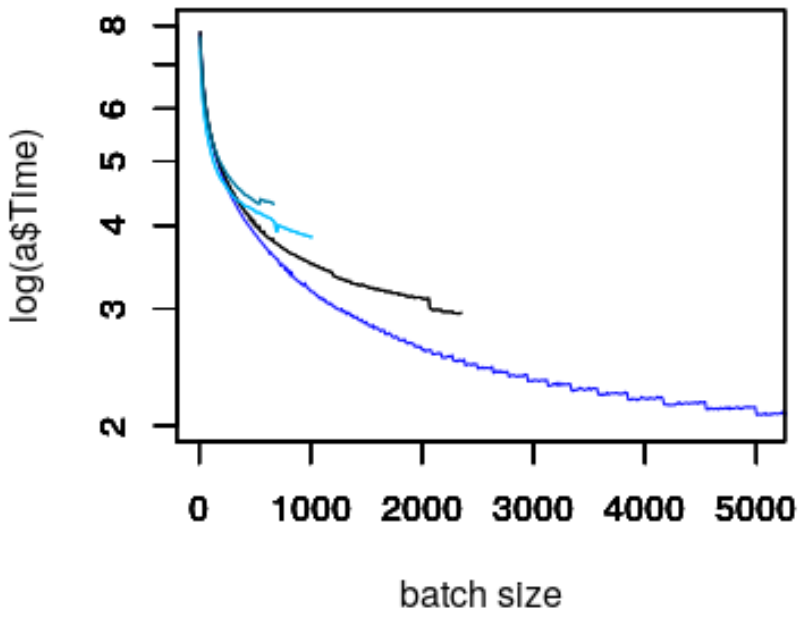


Abbildung 6.8: Batch Size vs Trainings Time über eine Epoche

und damit zu einer geringeren Klassifikationsleistung [HHS17]. Um diesen Verlust an Klassifikationsleistung auszugleichen gibt es die Möglichkeit die Lernrate anzupassen und eine andere Batch Normalisation zu verwenden [HHS17]. Diese Technik funktioniert laut dem Paper „Train longer, generalize better: closing the generalization gap in large batch training of neural networks“ bereits auf residualen Netzen wie sie in dieser Arbeit verwendet werden [HHS17]. Vorallem bleibt die Einsparung bei der Trainingszeit durch diese Technik intakt [HHS17].

Ist dieser Effekt auf PruneTrain übertragbar?

Eine grössere Batchsize sorgt auf jeden Fall für signifikant weniger Verkleinerung des Netzes.

Die Frage die sich hier stellt ist, ob mit Hilfe von largeBatch bei maximaler Batchsize die Verkleinerungsrate steigt

6.2 Untersuchung von Net2Net

Funktioniert; Experimente müssen noch durchgeführt werden; dann ca. 1 Tag für Evaluierung und Fertigstellung)

Die Operatoren zur Beschleunigung des Lernens durch Wissenstransfer werden in diesem Unterkapiel evaluiert. Diese Evaluierung arbeitet selbst erstellten Implementierung.
Die Evaluierung umfasst drei unterschiedliche Situationen, diese Situation sind analog zu den in der dazugehörigen Quelle [CGS15]. Die Evaluierung arbeitet mit einem ResNet, welches auf Cifar10 trainiert wird. 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 kombiniiert. Die drei Situationen werden in den drei folgenden Unterkapiteln näher beschrieben.

6.2.1 Evaluierung des Operators für ein breiteres Netz

Evaluiert wird der operator durch verschiedene Optionen, welche Bereich des Netzes breiter gemacht wird:

- Eine ganze Phase
- Alle Phasen

Wie in Kapitel 2.5 beschrieben werden beim Operator für ein breiteres Netz die Gewichte für die neu hinzugefügten Gewichte aus den ursprünglichen Gewichten berechnet. Um zu evaluieren wie gut diese Methode funktioniert wird sie verglichen mit einer Baseline-Methode, bei der die neu hinzugekommenen Gewichte zufällig initialisiert werden.

6.2.2 Evaluierung des Operators für ein tieferes Netz

6.3 PruneTrain + Net2Net

Anpassungen am Coe nötig aber nicht sehr umfangreich: ca 1 Tage; dann auf der Graka laufen lassen; ca. 3 Tage um zu schreiben +evaluieren

7 Vergleich

8 Additive Verfahren

8.0.1 Zahlenformate

Text fertig schreiben; etwa 4 Stunden

- FP16 bereits probiert

FP16 nur auf RTX 2080 sinnvoll Bietet nach erster Messung etwa 28 % Prozent Gewinn.
Code für dieses Verfahren liegt vor: Amp apex von Nvidia
AMP bietet 3 mögliche Optimierungsstufen:
O1 Patch all Torch functions and Tensor methods to cast their inputs according to a whitelist-blacklist model. Whitelist ops (for example, Tensor Core-friendly ops like GEMMs and convolutions) are performed in FP16. Blacklist ops that benefit from FP32 precision (for example, softmax) are performed in FP32. O1 also uses dynamic loss scaling, unless overridden.
02 casts the model weights to FP16, patches the models forward method to cast input data to FP16, keeps batchnorms in FP32, maintains FP32 master weights, updates the optimizer's paramgroups so that the optimizer.step() acts directly on the FP32 weights (followed by FP32 master weight-FP16 model weight copies if necessary), and implements dynamic loss scaling (unless overridden). Unlike O1, O2 does not patch Torch functions or Tensor methods.
O3 may not achieve the stability of the true mixed precision options O1 and O2. However, it can be useful to establish a speed baseline for your model, against which the performance of O1 and O2 can be compared. If your model uses batch normalization, to establish speed of light you can try O3 with the additional property override keepBatchnormfp32=True (which enables cudnn batchnorm, as stated earlier).
Hier nur O0, O1 und O2 dargestellt, da O3 absolut nicht mithalten kann was Performance angeht.

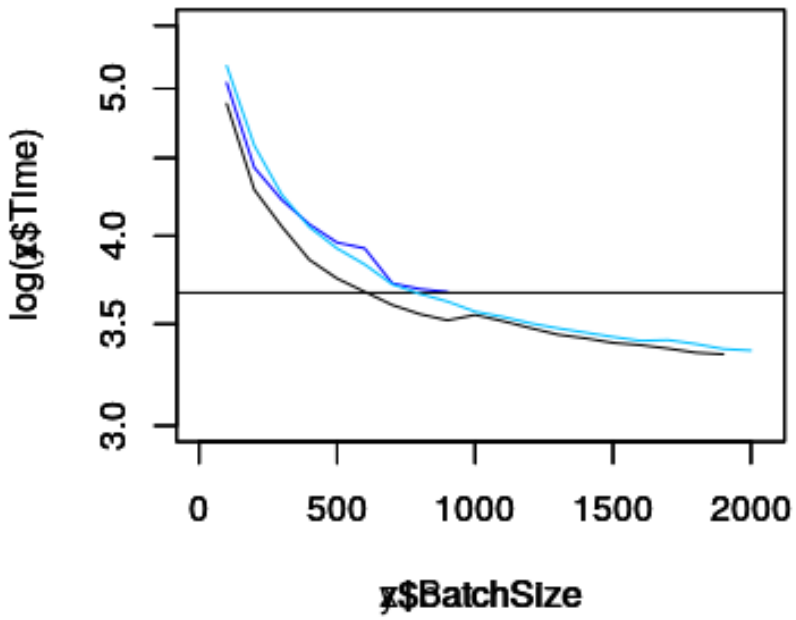


Abbildung 8.1: Vergleich Trainingszeit einer Epoche für verschiedene Optimierungsstufen von Amp Apex. DunkelBlau=O0; Schwarz = O1; Hellblau=O2

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

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

8.0.2 LARS

Experimente fast fertig (3x mal auf einer Graka für 50 min); dann etwa 3 Stunden fürText + Evaluierung

Es stellt sich die Frage, ob das einen so grossen Einfluss auf die Ausführungszeit hat.

Man sieht, dass mit steigender Batchgröße die Ausführungszeit sinkt.

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

8.0.3 Beschleunigung der Berechnung des Gradientenabstiegverfahren

ab hier löschen

Accelerating CNN Training by Sparsifying Activation Gradients funktioniert nur auf Toy-Benchmarks

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

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

Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent

Interessant bisher kein Code verfügbar

Accelerated CNN Training Through Gradient Approximation

Interessant bisher kein Code verfügbar

9 Evaluation

Hier werden die Ergebnissedes vorherigen Kapitels mit einem anderen bisher unbekannten Datensatz evluiert.

kein Implmentierungsaufwand; auf Grafikkarte: baseline + Prune-Train+Net2Net

10 Ausblick und Fazit

A d

Abbildungsverzeichnis

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`.

[GEN+18] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T. Yang, and E. Choi. Morphnet: Fast simple resource-constrained structure learning of deep networks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

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

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

[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.