technische universität
dortmund

fakultät für informatik

Masterarbeit

Zeit-effizientes Training von Convolutional Neural Networks

Jessica Buehler

15. Juni 2020

Gutachter:

Prof. Dr. Heinrich Müller

M.Sc. Matthias Fey

Lehrstuhl VII
Informatik
TU Dortmund

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Hintergrund dieser Arbeit 1

1.2 Aufbau der Arbeit 1

2 Stand der Wissenschaft **3**

2.1 Funktionsweise von CNNs 3

2.2 ResNet – eine neuere CNN-Architektur 5

2.3 Vorgehen zur Suche nachdem Stand der Wissenschaft 7

2.4 Beschneidung des Netzes zur Beschleunigung des Training 8

2.5 Beschleunigung des Lernens durch Wissenstransfer 11

2.5.1 Operator für breiteres Netz 11

2.6 Schnelles Ressourcen beschränktes Strukturlernen tiefer Netzwerke 13

2.7 Automatische Architektursuche 15

2.8 Zeitsparende Nethoden 15

2.8.1 Verringerung der für Berechnungen nötige Zeit 15

2.8.2 Beschleunigung der Berechnung des Gradientenabstiegsverfahren 17

2.9 Additive Methoden 17

2.9.1 Ghost Batch Normalization 18

3 Experimente – Arbeitstitel **19**

3.1 Experimentales Setup 19

3.1.1 Hardware 19

3.1.2 Wahl des Frameworks 19

3.1.3 verwendete Netzarchitektur 20

3.2 Untersuchung von PruneTrain 21

3.2.1 Evaluation von PruneTrain 21

3.2.2 Einfluss der Batchgröße auf PruneTrain 26

3.2.3 Einfluss der Batchgrösse und der Lernrate auf die Verkleinerung des Netzes 30

3.3 Untersuchung von Net2Net 31
3.4 Untersuchung von MorphNet 31
3.5 PruneTrain + Net2Net 31
3.6 Additive Verfahren 31
3.6.1 Zahlenformate 31
3.6.2 Beschleunigung der Berechnung des Gradientenabstiegverfahren 33

4 Evaluation **35**

A d **37**

Abbildungsverzeichnis **39**

Literaturverzeichnis **41**

Todo list

Einleitung fertigschreiben – zum Schluss 1
Aufbau der Arbeit – erst bei fortgeschrittener Arbeit schreiben 1
Grafik zum Verfahren 10
Beispiel für Channel Union? 10
cite 15
subnormale Zahlen 16
ref 16
Die Ausreisser von baseline ohne erzwungene Synchro kann ich mir nicht erklären. 20
Braucht es hier einen statistischen Test, um zu zeigen dass diese Ergebnisse signifikant unterschiedlich sind? 20
Bottleneck -Eigenschaft 21
ref 21
ref 22
beispielrechnung 24
ref 24
Quelle 26
Tabelle mit neuen Zahlen updaten, da mit kaputter Grafikkarte berechnet 26
Hier muss noch das Fitten des Modells und der t-Test erklärt werden . 28
Tabelle 28
t-Test um statistisch zu zeigen, dass das signifikant ist 30
Untersuche, ob largeBatch auch auf ein PruneTrain Netzwerk anwendbar ist. 30
Untersuche, ob die Grösse des Batches beeinflusst, wie viel vom Netz geprunt wird 30
Weitere Versuche, die zeigen ob die Zeiten grossen statistischen Schwankungen unterliegen. 32
Testen ob es funktioniert 33
Implementieren (ist einfach) und testen 33

Mathematical Notation

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

1 Einleitung

1.1 Motivation und Hintergrund dieser Arbeit

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

Einleitung fertigschreiben – zum Schluss

1.2 Aufbau der Arbeit

Aufbau der Arbeit – erst bei fortgeschrittener Arbeit schreiben

2 Stand der Wissenschaft

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

2.1 Funktionsweise von CNNs

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

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

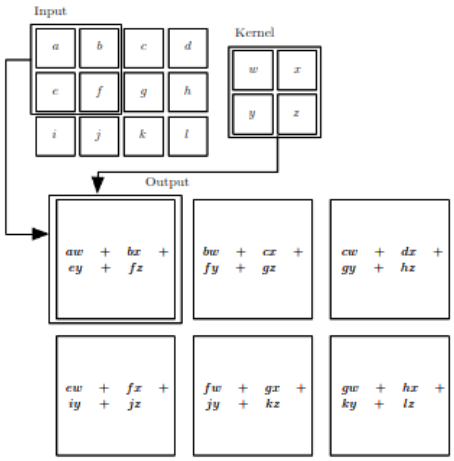


Abbildung 2.1: Abbildung zur Faltung [GBC16]

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

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

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

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

[2]Für Erklärung Relu siehe [Hay98]

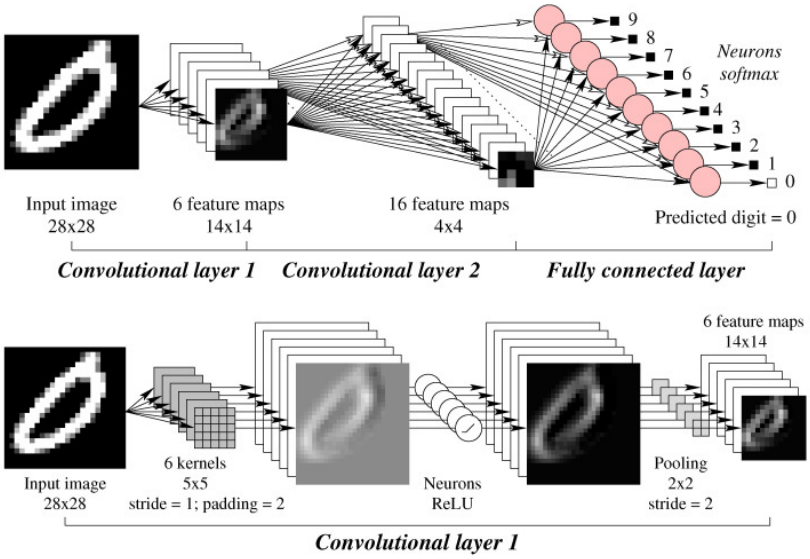


Abbildung 2.2: Convolutional Neural Net [CCGS16]

Eine beispielhafte Übersicht über die CNN-Architektur ist in Abbildung 2.2 zu sehen.
Die Fully-Connected-Layer errechnen aus den Ausgängen der Convolutional-Layer, in welche Klasse ein Objekt klassifiziert werden soll.
Die Filter, die auf die Feature Maps bzw. die Eingabebilder angewendet werden, sind trainierbar. Zusätzlich sind auch die Gewichtungen des Fully-Connected Layers trainierbar. Das heißt durch den Trainingsprozess wird versucht die Werte in der Filtermatrix und des Fully-Connected Layer so zu verändern, dass das gesamte CNN besser klassifizieren kann. Für diese Veränderung wird ein Gradientenabstiegsverfahren, welches rückwärts durch die Schichten propagiert wird, benutzt.

2.2 ResNet – eine neuere CNN-Architektur

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. 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.

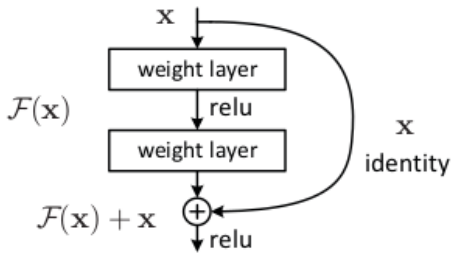
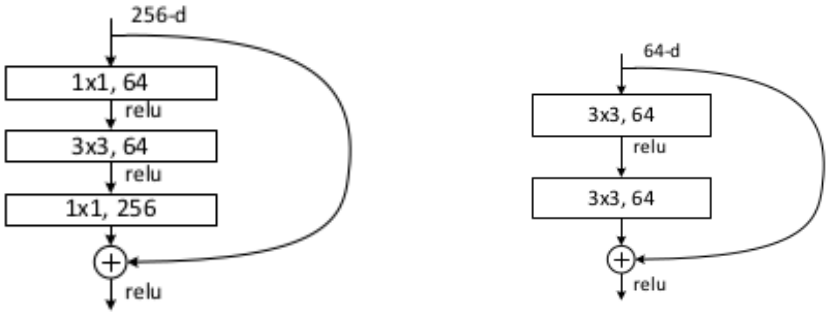


Abbildung 2.3: Abbildung der Kurzschlussverbindung [HZRS15]

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 34-Layer residualen Netzwerk 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.

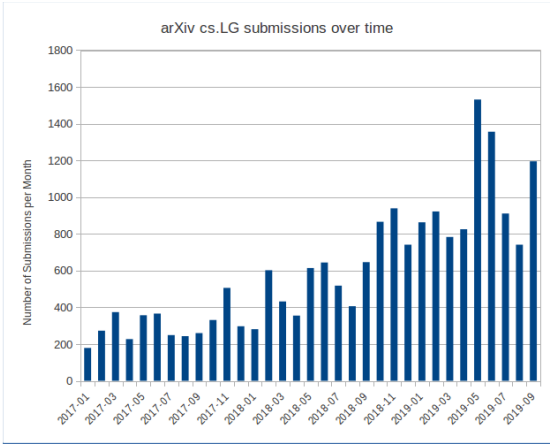


Abbildung 2.5: Tägliche Submissionen der Kategory Machine Learning auf arxiv [oA19]

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 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.

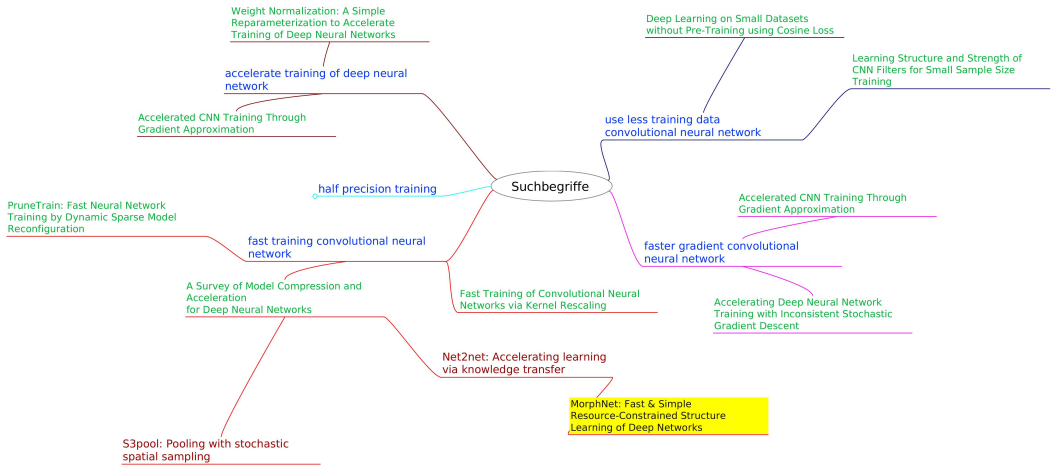


Abbildung 2.6: Mindmap zu den Suchbegriffen bezüglich des aktuellen wissenschaftlichen Stands

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.

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

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:

$$GL(\bigcup_{l=1}^{L} W^{(l)}_{:,:,:,:}) = \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} ||W^{(l)}_{c_l,:,:,:}||_2 + \sum_{k_l=1}^{K_l} ||W^{(l)}_{:,k_l,:,:}||_2 \right) \tag{2.1}$$

Dieser Term nennt sich Gruppen-Lasso. Der Parameter $W^{(l)}_{:,:,:,:}$ stellt die Gewichte im CNN als Tensor dar. Mit l wird dargestellt um welches Layer es sich handelt. Die Dimensionen des Tensors sind: Ausgangskanäle ×Eingangskanäle ×Kerneldimension 1 ×Kerneldimension 2. L gibt an über wie viele Layer der Gruppen-Lasso Term berechnet wird. k_l ist die Laufvariable über die einzelnen Eingangskänale und c_l 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-Lasso und Verlust-Funktion balanciert:

$$LPR(GL(\bigcup_{l=1}^{L} W^{(l)}_{:,:,:,:}), l(y_i, f(x_i, W))) = \frac{\lambda \cdot GL(\bigcup_{l=1}^{L} W^{(l)}_{:,:,:,:})}{l(y_i, f(x_i, W)) + \lambda \cdot GL(\bigcup_{l=1}^{L} W^{(l)}_{:,:,:,:})} \tag{2.2}$$

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.

Beispiel für Channel Union?

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. 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. 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].

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.

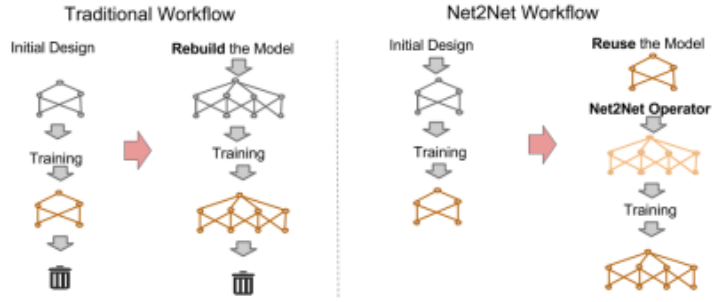


Abbildung 2.7: Traditioneller Workflow vs. Net2Net Workflow

Wie in Abbildung 2.7 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.

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 i bezeichnet und hat den Gewichtstensor $\mathbf{W}^i$ mit der Dimensionalität von $c_n \times k_l \times f_1 \times f_2$. Die Schicht, der Eingangskanäle hinzugefügt werden wird mit $i+1$ bezeichnet und hat den Gewichtstensor $\mathbf{W}^{i+1}$ mit der Dimensionalität von $c_m \times k_n \times f_3 \times f_4$. Dem Layer i werden q Kanäle hinzuefügt. Dies entspricht wie in Abbildung **??** abgebildet ist $q \cdot l$ zusätzlichen Filterkerneln.

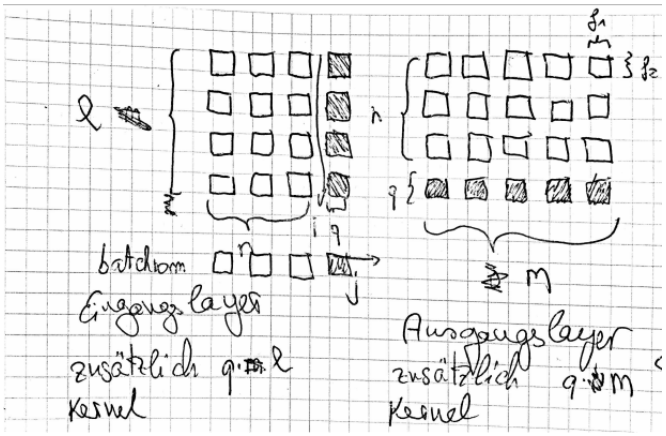


Abbildung 2.8: Übersicht über die zusätzlichen Kanäle todo: Grafik schön machen

Für das Layer $i+1$ sind es entsprechend $q \cdot m$ zusätzliche Kernel. Die Gewichtstensoren nachdem Anwenden des Net2Net Operators werden mit $\mathbf{U}^i$ und $\mathbf{U}^{i+1}$ bezeichnet und sollen die Dimensionalität von $\mathbf{U}^i : c_{n+q} \times k_l \times : \times :$ und $\mathbf{U}^{i+1} : c_m \times k_{n+q} \times : \times :$ haben. Der Net2Net Operator wird angewendet, indem zunächst eine Mapping Funktion g definiert wird die für ein zufällige Belegung der zusätzlichen Kernels sorgt:

$$g_j = \begin{cases} j & , \text{ falls } j \leq n \\ k & , \text{ falls } j > n : \ k \text{ zufälliges Sample von } \{1, 2, \ldots, n\} \end{cases} \tag{2.3}$$

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

$$\mathbf{U}^i_{j,k,f_1,f_2} = \mathbf{W}^i_{g(j),k,f_1,f_2} \quad \mathbf{U}^{i+1}_{h,j,f_3,f_4} = \frac{1}{|\{x|g(x) = g(j)\}|} \mathbf{W}^{i+1}_{h,g(j),f_3,f_4} \tag{2.4}$$

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 i-ten und $(i+1)$-Layer eine Batchnormalisierung befinden, so werden die Parameter dieser Schicht ebenfalls kopiert.

Um nicht mehrere exakt gleiche Kernelreihen zu haben kann außerdem noch ein Noiseanteil auf alle neuen Gewichte addiert werden. Dies ist vor allem für den Fall wichtig, wenn der Trainingsalgorithmus keine Form der Randomiesierung hat, die gleichen Gewichtstensoren ermutigt unterschiedliche Funktionen zu erlernen. Somit sind die vom ursprünglichen und neuen Netz gelernten Funktionen ähnlich aber nicht gleich.

Tieferes Netz

Der Net2Net- Operator für ein tieferes Netz ersetzt die Operation der i-ten Schicht $\mathbf{h}^{(i)} = \phi\left(\mathbf{h}^{(i-1)t}\mathbf{W}^{(i)}\right)$ durch die Operation von zwei Layern $\mathbf{h}^{(i)} = \phi(\mathbf{U}^{(i)t}\phi(\mathbf{W}^{(i)t}\mathbf{h}^{(i-1}))$. $\mathbf{U}$ wird als Identitätsmatrix initialisiert. Um die geforderte Erhaltung der gelernten Funktion zu erhalten muss für die Aktivierungsfunktion ϕ und alle Vektoren $\mathbf{v}$ die Gleichung $\phi(\mathbf{I}\phi(\mathbf{v})) = \phi(\mathbf{v})$). Die Aktivierungsfunktion ReLu erfüllt diese Anforderung.
Wird zwischen den beiden Layern eine Batchnormalisierung genutzt, so müssen die Parameter der Batchnormalisierung so gewählt werden, dass sie die gelernte Funktion des Netzes nicht verändert.

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 Schnelles Ressourcen beschränktes Strukturlernen tiefer Netzwerke

Im Gegensatz zu den beiden vorherigen Kapiteln, 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 Datensets skalierbar
2. Es kann die Struktur in Bezug auf eine bestimmte Nebenbedingung (z.Bsp. Modellgröße, Nummer von 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 simple. Die Breite der Ausgangskanäle für alle M Schichten wird mit $O_{1:M}$ bezeichnet.

Der Anfangspunkt dieser Methode ist ein Netz $O^0_{1:M}$ mit einer initialen Breite der Ausgangskanäle sowie fixen Größen für die Filtergrößen, die Netzwerktopologie. 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:

$$O^*_{1:M} = \underset{\mathcal{F}(O_{1:M}) \leq \zeta}{\arg\min} \min_{\theta} \mathcal{L}(\theta) \tag{2.5}$$

Das Vergrößern des Netzes basiert auf einer Lösung für die Gleichung 2.5: 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 2.5 zu lösen finde nun das größte ω, so dass $\mathcal{F}(\omega \cdot O_{1:M}) \leq \zeta$ gilt.

Dieser Ansatz sorgt zwar für eine mögliche Verkleinerung und Vergrößerung des Netzes und er funktioniert gut bei einem guten initialen Netz. Ist das initialen Netz nicht von so guter Qualität, so hat dieser Ansatz Probleme. Dieser Nachteil wird durch eine Veränderung der Verlust-Funktion aufgehoben. Es wird ein Regularisierer dazu addiert, welcher die Kontribution eines Netzbestandteiles zu $\mathcal{F}(O_{1:M})$ misst und es damit direkt optimieren kann. Dann ist

$$O^*_{1:M} = \underset{\mathcal{F}(O_{1:M}) \leq \zeta}{\arg\min} \min_{\theta} \mathcal{L}(\theta) + \lambda \mathcal{G}(\theta) \tag{2.6}$$

Dieser Ansatz kann die relative Größe eines Layers ändern, hat aber den Nachteil das häufiger die zu optimierende Nebenbedingung nicht maximal maximiert wird.

Die beiden Ansätze lassen sich kombinieren. Es entsteht folgender 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.

1: Trainiere das Netz um $\theta^* = \underset{\theta}{argmin}\mathcal{L}(\theta) + \lambda\mathcal{G}(\theta)$ zu finden
2: Finde die neue Breite $O'_{1:M}$, die durch 1. errechnet wurde
3: Finde das größte ω, so dass $\mathcal{F}(\omega \cdot O_{1:M}) \leq \zeta$ gilt
4: Wiederhole ab 1. so häufig wie gewünscht mit $O_{1:M} = O'_{1:M}$
Ausgabe: $\omega \cdot O_{1:M}$

Algorithmus 2.1: MorphNet Algorithmus

2.7 Automatische Architektursuche

2.8 Zeitsparende Nethoden

2.8.1 Verringerung der für Berechnungen nötige Zeit

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

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

cite

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

Berechnung mit 16 Bit Gleitkomma [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

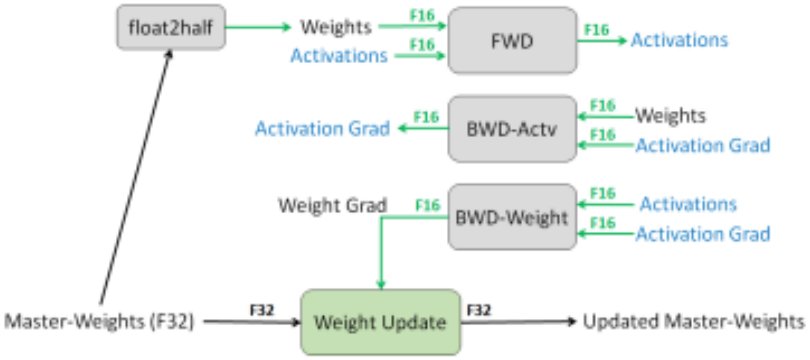
	16 Bit	32 Bit
kleinste darstellbare positive Zahl	$0.61 \cdot 10^{-4}$	$1.1755 \cdot 10^{-38}$
grösste darstellbare positive Ganzzahl	65504	$3.403 \cdot 10^{38}$
minimal subnormale Zahl	$2^{-24} \approx 5.96 \cdot 10^{-8}$	2^{-149}

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

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

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

Beim Trainieren von neuronalen Netzwerken mit 16 Bit Gleitkommazahlen werden die Gewichte, Aktivierungen und Gradienten im 16 Bit Format gespeichert. Die Speicherung der Gewichte als 32 Bit Mastergewichte hat zwei mögliche Erklärungen, die aber nicht immer zutreffen müssen.
Um nach einem Forward Druchlauf des Netzes die Gewichte abzudaten wird ein Gradientenabstiegsverfahren benutzt. Hierbei werden die Gradienten der Gewichte berechnet. Um für die Funktion, die das CNN approximiert einen besseren Approximationserfolg zu erlangen wird dann dieser Gradient mit der Lernrate multipliziert. Wird dieses Produkt in 16 Bit abgespeichert, so ist in viele Fällen das Produkt der beiden Zahlen gleich Null. Dies liegt an der Taqtsache, dass wie in Tabelle zu sehen ist die kleinste darstellbare Zahl in 16 Bit wesentlich grösser ref
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

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

Die Methoden in diesem Kapitel beeinflussen die Trainingszeit nicht direkt, sondern helfen die Folgen anderer Verfahren abzumildern.

2.9.1 Ghost Batch Normalization

3 Experimente – Arbeitstitel

Zunächst PruneTrain untersuchen.
Dann PruneTrain als MorphNet
Untersuchen welche Methoden aus 2.3 überhaupt sinnvoll auf grössere Datensätze anwendebar sind und funktionieren.
Die sinnvollen Methoden auf PruneTrain als Morphnet anwenden.

3.1 Experimentales Setup

3.1.1 Hardware

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

3.1.2 Wahl des Frameworks

Es wird mit pytorch gearbeitet, da pytorch gegenüber anderen Frameworks eine grössere 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. In Abbildung 3.1 sind mehrere Ausführungen eines Netzes mit Pytorch auf der GPU zu sehen. Auffallend ist dabei, dass in den ersten fünf Durchläufen die Trainingszeiten einer Epoche eine grosse Varianz haben.
Diese Varianz ist darauf zurückzuführen, dass Operationen in Pytorch standardmässig asynchron auf der GPU ausgeführt werden [oA]. Das bedeutet, dass bei einem Funktionsaufruf, der die GPU nutzt die Operationen hintereinander in eine Warteschlange auf der GPU eingereiht werden [oA] . Nicht notwendigerweise werden diese Operationen sofort ausgeführt. Dies erlaubt zwar mehr parallele

Berechnungen erhöht aber durch die inkludierte Wartezeit die gemessene Trainingszeit einer Epoche.

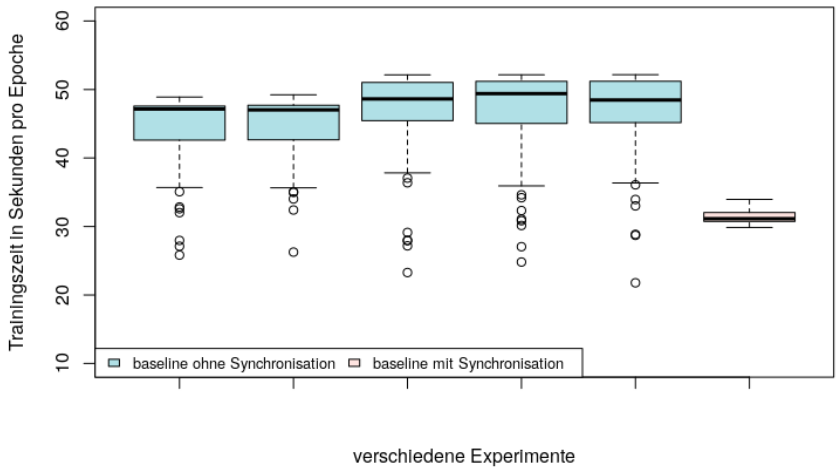


Abbildung 3.1: Vergleich asynchroner mit synchroner Ausführung von pytorch Operationen

Die sechste Messung in Abbildung 3.1 zeigt einen Durchlauf des gleichen Netzes mit erzwungener Synchronizität.

Die Ausreisser von baseline ohne erzwungene Synchro kann ich mir nicht erklären.

Braucht es hier einen statistischen Test, um zu zeigen dass diese Ergebnisse signifikant unterschiedlich sind?

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

Schränke diese Auswahl auf ResNet ein. Gründe hierfür:

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

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

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

Bottleneck -Eigenschaft

3.2 Untersuchung von PruneTrain

Die Untersuchung von PruneTrain basiert auf einer bereits vorgefertigten Implementierung [ptI].

3.2.1 Evaluation von PruneTrain

Hier 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.

ref

Als Netzwerk wird ein Res-Net 32 verwendet. In Tabelle ist die Netzstruktur aufgeführt.
verwendete Kenngrößen:

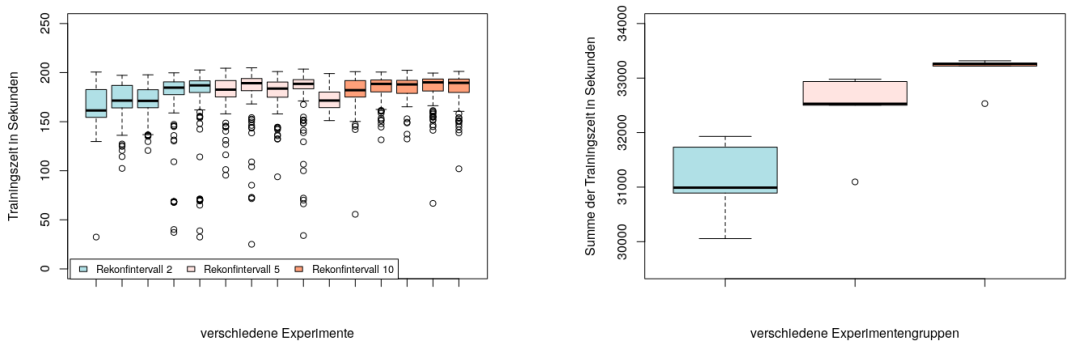
- 1 GPU
- Cifar10
- 180 Epochen

Veränderung der Trainingszeit durch PruneTrain

Zunächst wird hierfür nur das Prune Train ohne Anpassung der Batchgrösse betrachtet.
Variable Größen, die in verschiedenen Experimenten geändert werden:

- Lernrate
  - unterschiedlich große Lernraten
  - Anpassung der Lernrate währenddem Training
- Rekonfigurationsinterval
- Threshold
- Lasso-Ratio
- Batchgröße:
  - unterschiedliche Batchgrößen verschiedener Durchläufen
  - Anpassung der Batchgröße währenddem Training

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.
Die Trainingszeiten in Sekunden pro Epoche für verschiedene Lernrate ist in Abbildung zu sehen. Dabei ist zu sehen, dass unterschiedliche Lernraten einen großen ref
Einfluss auf die Verkleinerungsrate des Netzes haben. Der Grund hierfür ist, dass bei unterschiedlich großen Lernrate die Wahrscheinlichkeit, dass eine gewisse Anzahl von Gewichten unter den Treshold fällt unterschiedlich gross ist. Ist die Lernrate zu klein braucht es zu viele Epochen um bei einer zufälligen Initialisierung unter den Treshold zu gelangen. Ist die Lernrate zu groß, wird das Gewicht zwar kleiner ist danach aber wohlmöglich im negativen größer als der Treshold.
Als nächste Größe wird der Einfluss des Rekonfigurationsintervalls überprüft. Die entsprechenden Grafiken sind in Abbildung **??** abgebildet. In Abbildung 3.2a sind für die verschiedenen Experimente die Trainingszeiten pro Epoche zu sehen. Dabei werden drei verschiedene Rekonfigurationsintervalle (2,5 und 10) verglichen. In Abbildung 3.2a lässt sich für die verschiedenen Experimente keine großen Unterschiede sehen. Werden die Zeiten der jeweiligen Experimente addiert und in



(a) verschiedene Experimente

(b) Summe verschiedener Experimentengruppen

Abbildung 3.2: Boxplot der Rekonfigurationsintervalle

einem Boxplot dargestellt entsteht Abbildung 3.2b. In dieser Abbildung ist deutlich zu sehen, dass mit steigendem Rekonfigurationsintervall auch die Summe der Trainingszeiten pro Epoche steigt. Dies bedeutet, dass der Overhead des Pruningverfahrens geringer ist als der Gewinn durch das Verkleinern des Netzes. In Abbildung **??** ist zu sehen, dass dieser Gewinn an Trainingszeit in Abbildung 3.2b mit einem Verlust an geringem Verlust an Accuracy einhergeht.

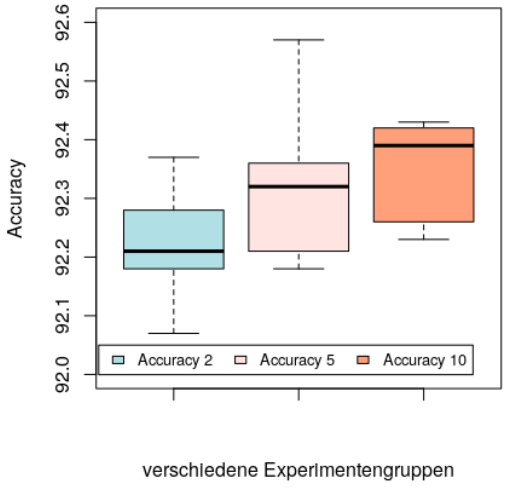


Abbildung 3.3: Accuracy von verschiedenen Experimentengruppen des Rekonfigurationsintervall

Wird der Treshold vergrössert, so wird das Netz schneller kompakter. Allerdings wird durch das kompaktere Netz auch die Accuracy geringer. Wird der Treshold

verkleinert, so wird das Netz nicht so schnell kompakter es verliert aber auch weniger Accuracy.
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.
Im Prune Train Paper wird als nächste zum Einsparen von Trainingszeit mit dem Verkleinern des Netzes die Batchgröße erhöht. Dabei soll währenddem dem Vorgang die Speicherauslastung die gleiche sein.
Als Anfangsbatchgrösse wird 265 gewählt. Für kleiner und grössere Netze wird nun mit Hilfe einer binären Suche erfasst wie groß die Batchgrösse sein darf, um eine gleiche maximale Speicherauslastung zu haben.
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 3.6 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 3.4 ist zu sehen wie sich im Accuracy im Verlauf der Epochen verändert.
In Abbildung 3.5 sind die Epochen 90 bis 180 näher herangezoomt.

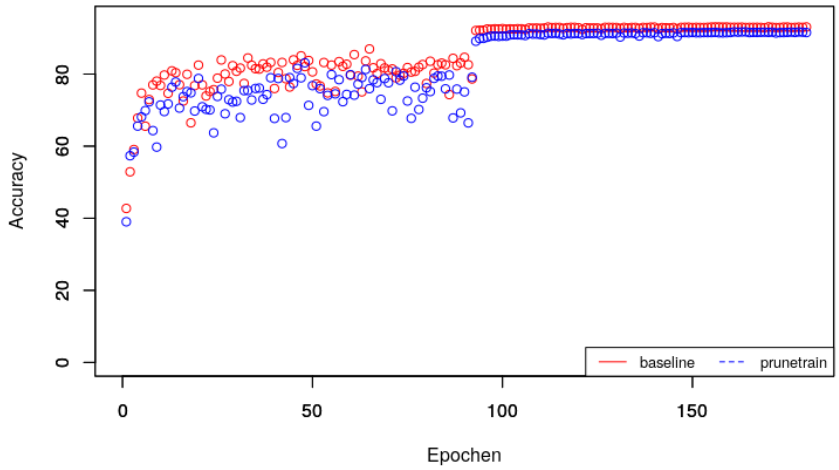


Abbildung 3.4: Veränderung der Accuracy während der Epochen

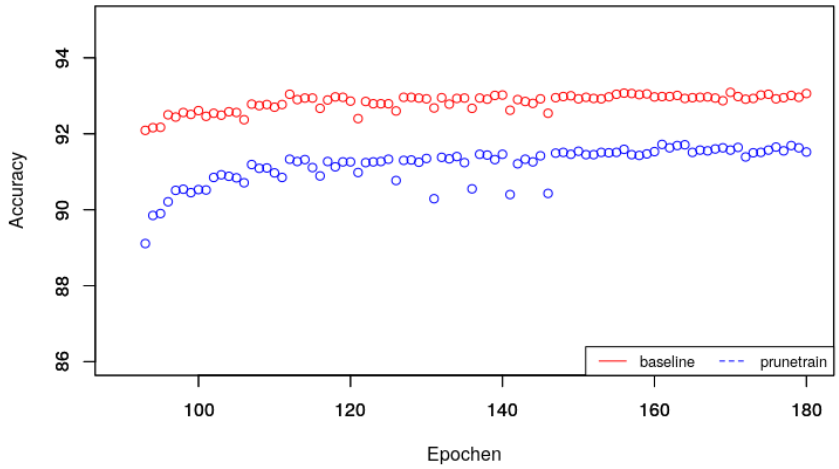


Abbildung 3.5: Zoom der Veränderung

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.

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

3.2.2 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.

Tabelle mit neuen Zahlen updaten, da mit kaputter Grafikkarte berechnet

Die maximal mögliche Batchgrösse in Abbildung 3.7 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.

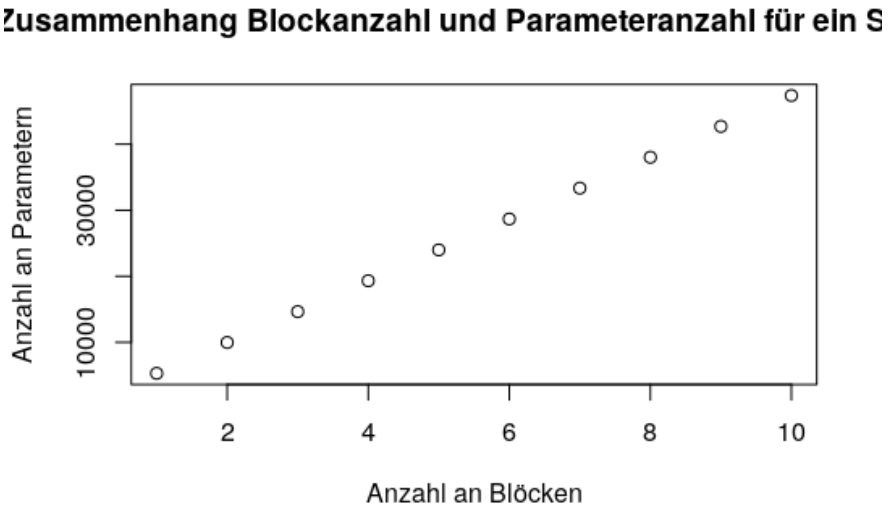


Abbildung 3.6: Batch Size vs Trainings Time über eine Epoche

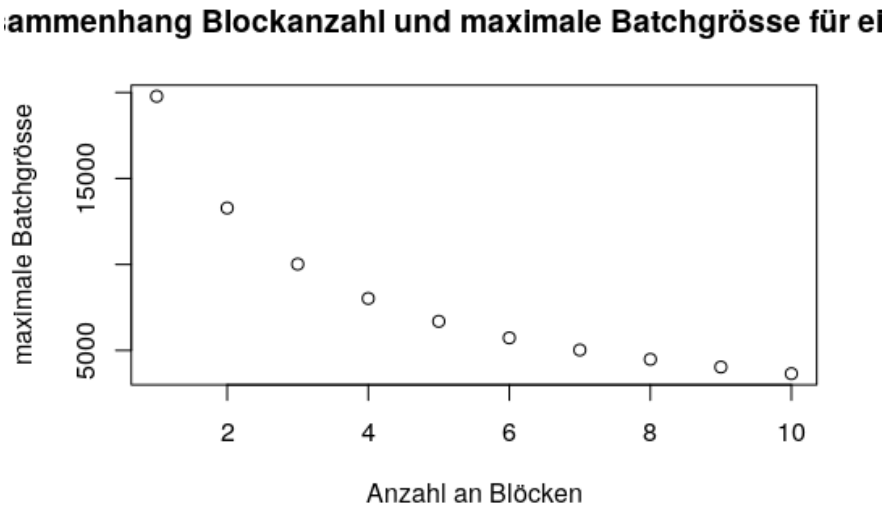


Abbildung 3.7: Batch Size vs Trainings Time über eine Epoche

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 3.8 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 3.9.

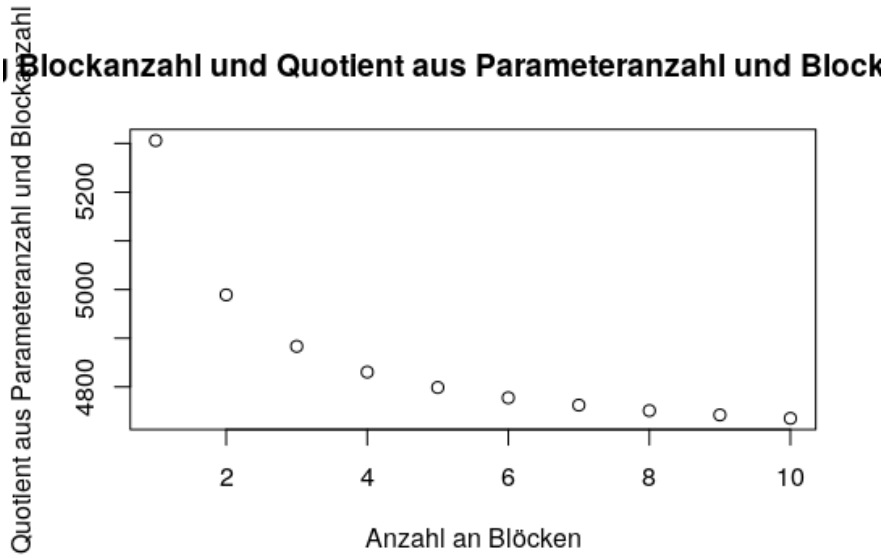


Abbildung 3.8: Batch Size vs Trainings Time über eine Epoche

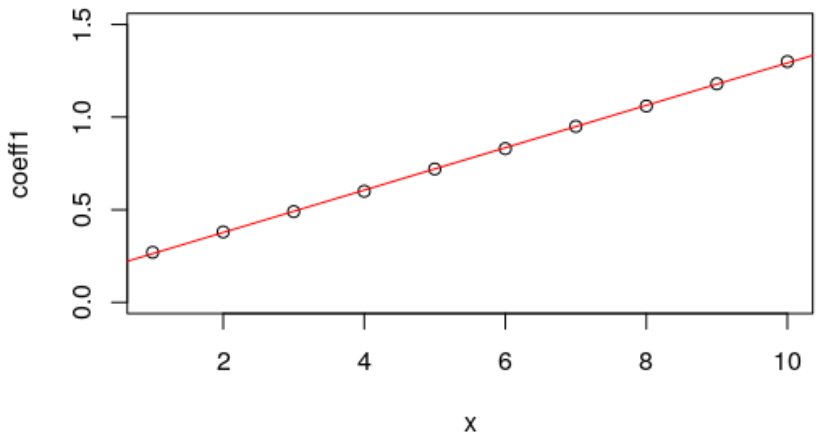


Abbildung 3.9: Batch Size vs Trainings Time über eine Epoche

Die gefittete Gerade hat die Gleichung:

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

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

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

Tabelle

wird den Zusammenhang zwischen Inferenz Flop und der Validation Accuracy verändert.
Die nächste Untersuchung über das Sparen von Kommunikationskosten beim Verteilten Training macht hier keinen Sinn da nur eine einzelne Graka genutzt wird.
Abschliessend wird noch evaluiert, wie die Dichte der Gewichte mit der Dichte der Kanäle nachdem Training zusammenhängen um eventuell durch spezifische Inferenzhardware weiter zusparen.
Bei grösserer Batchgrösse wird auch das Netz schneller. Dies ist in Abildung 3.10 für das ResNet und die verwendete Hardware abgebildet.

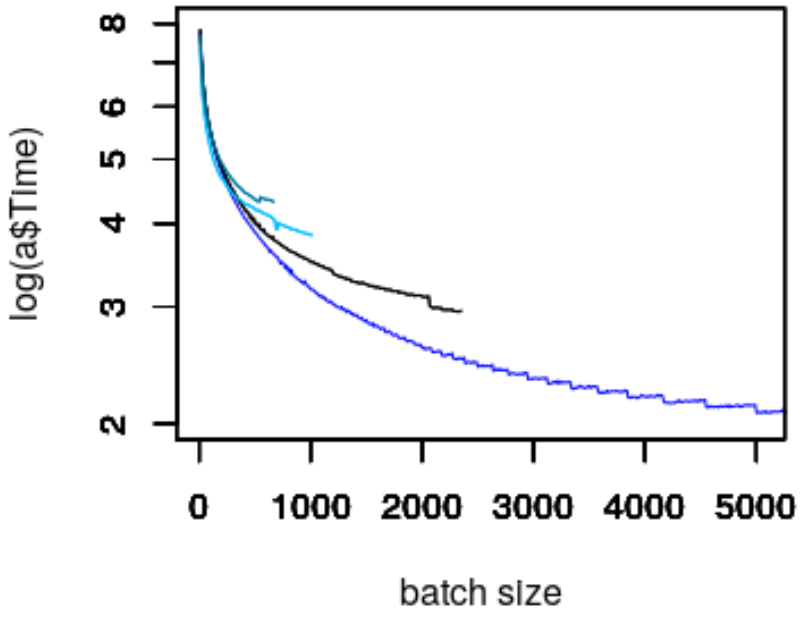


Abbildung 3.10: Batch Size vs Trainings Time über eine Epoche

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 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.

t-Test um statistisch zu zeigen, dass das signifikant ist

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

Ghost Batch Norm und LR Anpassung

Untersuchung

Im nächsten Schritt wird untersucht, ob diese Vorteile auch für den PruneTrain Vorgang genutzt werden kann. Dafür wird zunächst untersucht, welchen Effekt eine grössere Batchgrösse auf PruneTrain hat.

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

Man sieht, dass mit steigender Batchgröße die Ausführungszeit sinkt.

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

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

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

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

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

3.3 Untersuchung von Net2Net

3.4 Untersuchung von MorphNet

MorphNet macht alle Layer breiter um sie dann mit einem speziellen Regularisierer breiter zu machen. Dieser Regularisierer hat verschiedene mögliche Zielgrössen (Modelgrösse, Flops oder Inferenz-Zeit). Die Frage stellt sich hier, ob das Netz besser wird wenn alle Schichten breiter gemacht werden um später wieder geprunt zuwerden.
Weiterhin besteht die Möglichkeit das Netz nicht nur breiter zu machen sondern auch tiefer.
MorphNet erwähnt, dass es Sinn macht nicht im ganzen Netz denn Wider Operator anzuwenden sonder nur da wo der Regularisierer das Netz nicht schmaller macht.

3.5 PruneTrain + Net2Net

3.6 Additive Verfahren

3.6.1 Zahlenformate

- FP16 bereits probiert

FP16 nur auf RTX 2080 sinnvoll Bietet nach erster Messung etwa 28 % Prozent Gewinn.
Code für dieses Verfahren liegt vor: Amp apex von Nvidia
AMP bietet 3 mögliche Optimierungsstufen:
O1 Patch all Torch functions and Tensor methods to cast their inputs according to a whitelist-blacklist model. Whitelist ops (for example, Tensor Core-friendly ops like GEMMs and convolutions) are performed in FP16. Blacklist ops that benefit from FP32 precision (for example, softmax) are performed in FP32. O1 also uses dynamic loss scaling, unless overridden.
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.

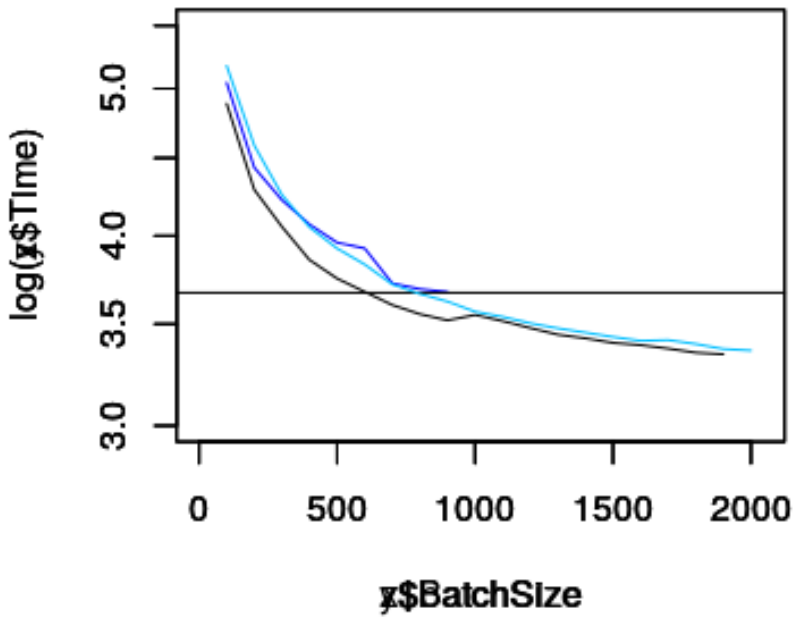


Abbildung 3.11: Vergleich Trainingszeit einer Epoche für verschiedene Optimierungsstufen von Amp Apex. DunkelBlau=O0; Schwarz = O1; Hellblau=O2

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

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

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

3.6.2 Beschleunigung der Berechnung des Gradientenabstiegverfahren

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

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

Testen ob es funktioniert

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

Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent

Interessant bisher kein Code verfügbar

Implementieren (ist einfach) und testen

Accelerated CNN Training Through Gradient Approximation

Interessant bisher kein Code verfügbar

4 Evaluation

A d

Abbildungsverzeichnis

2.1 Abbildung zur Faltung [GBC16] 4
2.2 Convolutional Neural Net [CCGS16] 5
2.3 Abbildung der Kurzschlussverbindung [HZRS15] 6
2.4 Vergleich zweier Residual Netz Blöcke [HZRS15] 7
2.5 Tägliche Submissionen der Kategory Machine Learning auf arxiv [oA19] 7
2.6 Mindmap zu den Suchbegriffen bezüglich des aktuellen wissenschaftlichen Stands 8
2.7 Traditioneller Workflow vs. Net2Net Workflow 11
2.8 Übersicht über die zusätzlichen Kanäle todo: Grafik schön machen 12

3.1 Vergleich asynchroner mit synchroner Ausführung von pytorch Operationen 20
3.2 Boxplot der Rekonfigurationsintervalle 23
3.3 Accuracy von verschiedenen Experimentengruppen des Rekonfigurationsintervall 23
3.4 Veränderung der Accuracy während der Epochen 25
3.5 Zoom der Veränderung 25
3.6 Batch Size vs Trainings Time über eine Epoche 27
3.7 Batch Size vs Trainings Time über eine Epoche 27
3.8 Batch Size vs Trainings Time über eine Epoche 28
3.9 Batch Size vs Trainings Time über eine Epoche 28
3.10 Batch Size vs Trainings Time über eine Epoche 29
3.11 Vergleich Trainingszeit einer Epoche für verschiedene Optimierungsstufen von Amp Apex. DunkelBlau=O0; Schwarz = O1; Hellblau=O2 32

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.

[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.

[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.

[oA] ohne Autor. Cuda semantice – pytorch 1.5.0 documentation. online `https://pytorch.org/docs/stable/notes/cuda.html` aufgerufen am 10.06.2020.

[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.