Karlsruhe Institute of Technology

Verfahren zur Reduktion von neuronalen Netzen

Analyse und Automatisierung

Masterarbeit
von

Tobias Viehmann

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	PD Dr. Victor Pankratius
Zweitgutachter:	Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter:	PD Dr. Victor Pankratius

Bearbeitungszeit: 15.07.2019 – 14.01.2020

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 14.01.2020

. . . .

(**Tobias Viehmann**)

Publikationsgenehmigung

Melder der Publikation

Hildegard Sauer

Institut für Programmstrukturen und Datenorganisation (IPD)
Lehrstuhl für Programmiersysteme
Leiter Prof. Dr. Walter F. Tichy

+49 721 608-43934
hildegard.sauer@kit.edu

Erklärung des Verfassers

Ich räume dem Karlsruher Institut für Technologie (KIT) dauerhaft ein einfaches Nutzungsrecht für die Bereitstellung einer elektronischen Fassung meiner Publikation auf dem zentralen Dokumentenserver des KIT ein.

Ich bin Inhaber aller Rechte an dem Werk; Ansprüche Dritter sind davon nicht berührt.

Bei etwaigen Forderungen Dritter stelle ich das KIT frei.

Eventuelle Mitautoren sind mit diesen Regelungen einverstanden.

Der Betreuer der Arbeit ist mit der Veröffentlichung einverstanden.

Art der Abschlussarbeit:	Masterarbeit
Titel:	Verfahren zur Reduktion von Neuronalen Netzen
Datum:	14.01.2020
Name:	Tobias Viehmann

Karlsruhe, 14.01.2020

. . . .

(**Tobias Viehmann**)

Inhaltsverzeichnis

1 Einführung 1
1.1 Ziele dieser Arbeit 2
1.2 Abgrenzung 3
1.3 Struktur der Arbeit 4

2 Hintergrund 5
2.1 Neuronale Netze 5
2.1.1 Perzeptron 5
2.1.2 Architektur 7
2.1.3 Forward-Propagation 11
2.1.4 Training 12
2.1.5 Convolutional Neural Networks 15

3 Verwandte Arbeiten 19
3.1 Matrizen- und Tensorzerlegung 19
3.2 Pruning 20
3.2.1 Pruning von Verbindungen 21
3.2.2 Pruning von Neuronen 23
3.2.3 Pruning von Filtern 25
3.3 Quantisierung 26

4 Reduktionsstrategien 29
4.1 Motivation 29
4.2 Grundlagen 30
4.2.1 Weitere Metriken 30
4.2.2 Architekturen 31
4.3 Pruning von Verbindungen 35
4.4 Pruning von Neuronen / Filtern 39

5 SNARE 49
5.1 Aufbau 49
5.2 Identifikation von relevanten Schichten 50
5.2.1 Vorteile der Betrachtung ganzer Netzwerke 50

5.3 Bewertung von Schichten 52
5.3.1 Regularisierung 54
5.3.2 Operationen 55
5.4 Pruning und Retraining 56
5.4.1 Implementierungsdetails 57
5.4.2 Resultate 59

6 Abschluss 61
6.1 Zusammenfassung 61
6.2 Erweiterungen SNARE 62
6.3 Ausblick 62

Abkürzungsverzeichnis 69

Abbildungsverzeichnis 71

Tabellenverzeichnis 73

1 Einführung

Zahlreiche Erfolge der letzten Jahre im Bereich des maschinellen Lernens lassen sich auf künstliche neuronale Netze zurückführen. Spracherkennung, maschinelle Übersetzung, Gesichtserkennung, automatische Bildfolgenbeschreibung sind nur einige wenige Anwendungsgebiete von neuronalen Netzen.

2012 lieferten Krizhevsky et al. einen entscheidenden Beitrag zur heutigen Popularität von neuronalen Netzen [KSH12]. Das nach Krizhevsky benannte *AlexNet* erzielte im *ILSVRC-2012*, einem Wettbewerb im Bereich der automatischen Bildklassifizierung, über 10% geringere Fehlerraten als seine Konkurrenten [KSH12]. Diese Leistung ist jedoch erst durch das rechenintensive Training von über 60 Milionen Parametern möglich geworden.

Dieser enorme Bedarf an Rechenzeit und Speicherplatz ist ein allgemeines Problem neuronaler Netze. Modernere Architekturen wie *VGGNet* umfassen dabei sogar noch mehr Schichten und Parameter, sodass nicht nur die Trainingszeiten Tage und Wochen andauern können, sondern auch eine einzelne Klassifizierung mehrere Milliarden Berechnungsschritte benötigt. Dies hat zur Folge, dass der Einsatz von derartigen Netzen an ein Mindestmaß von Hardwarevoraussetzungen gekoppelt ist. Dies erschwert die Portierung auf Geräte mit geringer Leistung wie beispielsweise Smartphones oder eingebettete Systeme erheblich. Ein Ansatz, um die gewünschten Netze auf diesen Platformen verfügbar zu machen, ist es, die bereits trainierten Netze zu komprimieren.

In den vergangenen Jahren ist die Komprimierung von neuronalen Netzen Gegenstand verschiedener Arbeiten geworden und hat sich zu einem umfangreichen Forschungsgebiet entwickelt. Eine beliebte Strategie ist es, unter Berücksichtigung der Genauigkeit redundante oder irrelevante Parameter zu entfernen. Anschließend wird in wenigen Epochen neu trainiert, um entstandene Verluste zu kompensieren [Mol+16; DCP17]. Ergebnisse aus [Mol+16; DCP17] oder [Han+15] verdeutlichen, dass damit Kompressionsraten über den Faktor 10 hinaus, umsetzbar sind.

1.1 Ziele dieser Arbeit

Das erste Ziel der vorliegenden Arbeit ist es, einen Einblick in aktuelle Techniken zur Komprimierung von neuronalen Netzen zu geben. Besonders das Pruning von Neuronen oder Filtern steht dabei im Vordergrund. Es werden Verfahren aus der Literatur beschrieben und an gängigen Architekturen evaluiert. Eigen entwickelte Modifikationen werden dabei ebenfalls betrachtet. Zusätzlich wird anhand dieser Praxisbeispiele demonstriert, inwiefern die Kompressionsraten mit Kombinationen und Ergänzungen der Verfahren weiter verbessert werden können.

Der zweite Beitrag dieser Arbeit ist die Vorstellung eines neuartigen Verfahrens mit dem Namen *Scorebased Neural Architecture REduction* (SNARE), welches ein trainiertes Netz automatisiert analysiert und komprimiert. In einer einzelnen Phase des Vorgangs bewertet, reduziert und trainiert es die Schichten eines gegebenen Eingabenetzes. Um die Kompressionsraten weiter zu verbessern, wiederholt sich dieser Prozess bis eine festgelegte Fehlerrate überschritten wird. Abbildung 1.1 verdeutlicht die generelle Vorgehensweise einer Iteration.

Zuvor wurde bereits erwähnt, dass komprimierte Netze häufig erneut trainiert werden, um Verluste auszugleichen. Diese Arbeit stellt abschließend einen Ansatz vor, nicht das gesamte Netz, sondern Teilmengen an Schichten erneut zu trainieren. Die benötigten Trainingsdaten lassen sich bei der Anwendung des unkomprimierten Netzes auf den ursprünglichen Trainingsdaten als Zwischenergebnisse extrahieren. Der entscheidende Vorteil ist eine deutliche Einsparung der Rechenzeit im Vergleich zu dem Trainieren des gesamten Netzes. Die damit einhergehenden Schwierigkeiten und Probleme werden ebenfalls aufgezeigt.

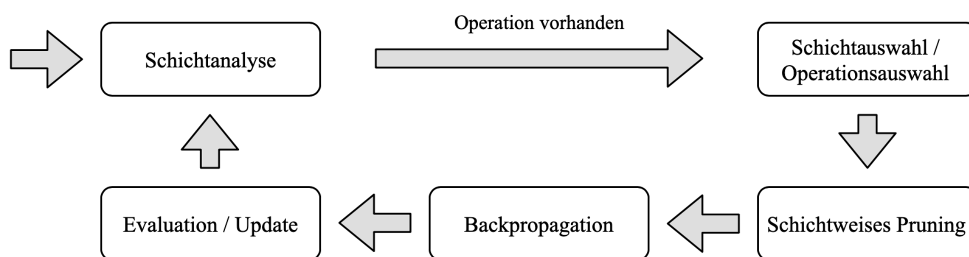


Abbildung 1.1: Vorgehensweise einer Iteration von SNARE

1.2 Abgrenzung

Es gibt einige unterschiedliche Forschungsbereiche und Ansätze, die das Erzeugen neuronaler Netze mit wenigen Parametern oder die Komprimierung bestehender Netze umfassen. Dieser Abschnitt dient dazu, verwandte Bereiche vorzustellen, die in der vorliegenden Arbeit jedoch nicht weiter betrachtet werden:

Verbesserte Trainingsverfahren: Aus den in [Han+15], [Lin+16] erzielten Kompressionsraten lässt sich folgern, dass ein trainiertes neuronales Netz oftmals große Mengen an irrelevanten Informationen enthält. Gleichzeitig werden meist Netze mit großer Kapazität wie *VGGNet* trainiert, da sie bessere Resultate erzielen als kleinere Netze. Dies führt zu der Forschungsfrage, ob man mit verbesserten Trainingsverfahren direkt ein kleines Netz trainieren kann, statt erst ein großes Netz zu trainieren und es zu komprimieren. Lokale Optima und Überanpassung sind weitere Probleme aktueller Gradientenabstiegsverfahren [GBC16], [CLG01]. Sogenannte *Dropout-Layer* [Sri+14], Regularisierungen wie frühzeitiges Beenden [CLG01] oder der Einsatz von genetischen Algorithmen [Fis+07] sind potenzielle Maßnahmen, diese Probleme zu lösen.

Besondere Netze mit wenigen Parametern: Andere Forschungsgruppen beschäftigen sich mit dem Bau kleiner neuronaler Netze mit besonderen Architekturen, die mit aktuellen Trainingsverfahren bereits ausreichende Ergebnisse liefern. Iandola et al. haben mit *SqueezeNet* ein neuronales Netz vorgestellt, welches um den Faktor 50 weniger Parameter als *AlexNet* erhält und dennoch die gleiche Genauigkeit erreicht [Ian+16]. Frankle et al. stellen in [FC18] die Lotterie-Hypothese auf und unterstützen diese mit Beispielen. Die Hypothese besagt, dass ein zufällig initialisiertes, dichtes neuronales Netz Teilnetze beinhaltet, welche durch ein isoliertes Training die gleiche Testgenauigkeit wie das Gesamtnetz erreichen können. Diese besonderen Teilnetze mit effektiver Initialisierung werden schließlich als die gewinnenden Lose bezeichnet. Der Erfolg dieser Methode hängt davon ab, ob durch zukünftige Arbeiten diese Hypothese in weiteren Kontexten belegt und die Suche nach den gewinnenden Losen verbessert werden kann.

Komprimierungen zur Laufzeit: Die Arbeit von Lin et al. verfolgt das Ziel, neuronale Netze zur Laufzeit zu komprimieren [Lin+17]. Dazu wird ein Entscheidungsprozess mit Hilfe von *Reinforcement Learning* für jede Schicht modelliert. Basierend auf dem aktuellen Bild sowie den Zwischenergebnissen der Schichten bestimmt dieser Prozess die jeweils beste Strategie. Darüber hinaus ist das vorgestellte Framework in der Lage, Rechenressourcen individuell zu vergeben.

1.3 Struktur der Arbeit

Die vorliegende Arbeit gliedert sich in 6 Kapitel. Das nachfolgende Kapitel dient dazu, die grundlegende Terminologie und die Konzepte zu vermitteln, welche für das weitere Verstehen notwendig sind. Im 3. Kapitel werden verwandte Arbeiten erläutert, deren Techniken und Inhalte später in der Arbeit wieder aufgegriffen werden. Das 4. Kapitel umfasst die Evaluation von Pruningverfahren aus der Literatur an unterschiedlichen neuronalen Netzen. Eigene Variationen werden dabei ebenfalls untersucht. Im 5. Kapitel wird dargelegt, wie die bis dahin beschriebenen Kenntnisse in einem Verfahren auf ganze Netzwerke übertragen werden können und zu welchen Herausforderungen eine Automatisierung des gesamten Prozesses führt. Das abschließende Kapitel enthält eine Zusammenfassung der Arbeit sowie einen Ausblick in künftige Arbeiten.

2 Hintergrund

Dieses Kapitel dient dazu, dass grundlegende Wissen zum Verständnis der Arbeit zu vermitteln. Darüber hinaus werden Definitionen und Notationen aufgeführt, welche in nachfolgenden Kapiteln genutzt werden.

2.1 Neuronale Netze

Das grundlegende Ziel eines (tiefen) künstlichen neuronalen Netzes ist es, eine Funktion f^* zu approximieren [GBC16]. Das Vorgehen ist dabei dem menschlichen Gehirn nachempfunden. Verknüpfte Neuronen werden verwendet, um Informationen zu speichern, zu verarbeiten und weiterzugeben.

Die Ein- und Ausgaben eines künstlichen neuronalen Netzes können sich je nach Anwendungsgebiet und Aufgabe erheblich unterscheiden. Mögliche Eingaben sind beispielsweise Pixel eines Bildes, digitalisierte Sprachaufnahmen oder auch personenbezogene Daten.

Die in dieser Arbeit verwendete verkürzte Schreibweise neuronale Netze bezieht sich immer auf künstliche neuronale Netze.

2.1.1 Perzeptron

Neuronale Netze bestehen aus einer Vielzahl gleich aufgebauter Neuronen. Ein **Perzeptron** ist ein einfaches Beispiel für ein solches Neuron. Als eigenständiges Element kann dieses auch als binärer Klassifizier eingesetzt werden. Abbildung 2.1 zeigt den Aufbau eines Perzeptrons. Daran wird auch die Berechnung der Ausgabe deutlich. Es erhält als Eingabe aus n eingehenden Verbindungen jeweils einen reellen Wert, welcher mit einem zugehörigen Gewicht multipliziert wird. Diese Ergebnisse werden anschließend aufaddiert und ein zusätzlicher **Bias-Wert** b wird addiert. Abschließend bildet die sogenannte **Heaviside-Funktion** oder auch Stufenfunktion den resultierenden Wert auf das binäre Klassifizierungsergebnis des Perzeptrons ab. Formel 2.1 fasst die Berechnung zusammen, wobei $\mathbf{w} \cdot \mathbf{x}$ dem Skalarprodukt aus Gewichts- und Eingabevektor entspricht.

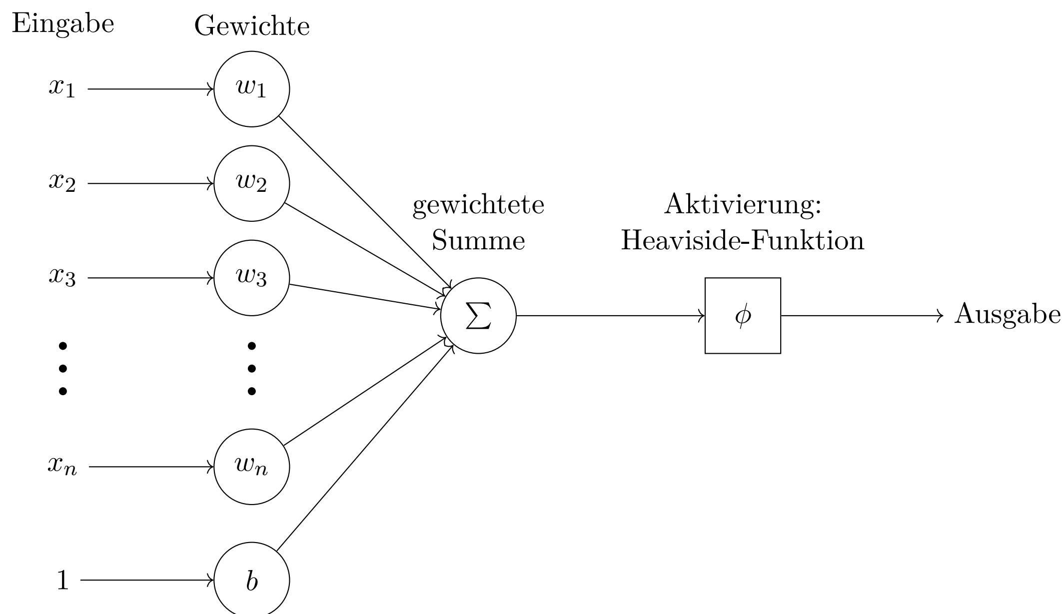


Abbildung 2.1: Aufbau eines Perzeptrons

$$a(\mathbf{x}) = \begin{cases} 1, & \text{wenn } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0, & \text{ansonsten} \end{cases} \tag{2.1}$$

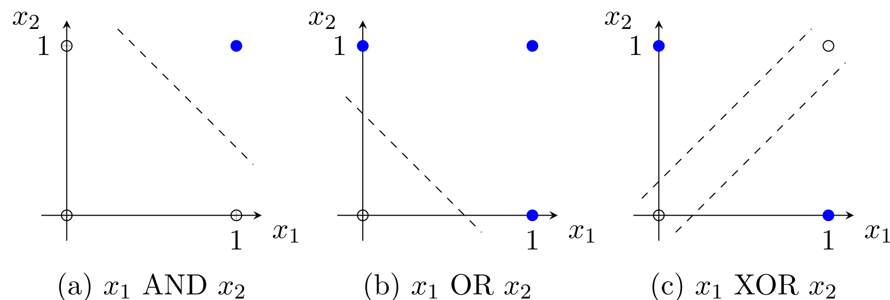
Lernverfahren Perzeptron

Damit das Perzeptron zur linearen Klassifizierung eingesetzt werden kann, gilt es passende Werte für die eingabeunabhängigen Parameter zu bestimmen. Diese umfassen den Gewichtsvektor $\mathbf{w}$ und den Bias-Wert b. Bereits 1958 beschrieb Rosenberg mit der Vorstellung des Perzeptrons einen Algorithmus, um dieses zu trainieren [Ros58]. Dabei werden wiederholt beliebige Elemente des Eingaberaums gezogen und die Gewichte angepasst, falls das Element fehlerhaft klassifiziert wurde. Bei einem linear separierbaren Eingaberaum konvergiert dies zu einer korrekten Lösung. Da dieses Verfahren nicht direkt auf weitere neuronale Netze übertragbar ist, wird an dieser Stelle für eine detailliertere Beschreibung und den Konvergenzbeweis auf [Kal17] verwiesen.

Grenzen Perzeptron

Allgemein ist das vorgestellte Perzeptron auf linear separierbare Datenmengen beschränkt. Abbildung 2.2 zeigt, dass die booleschen Operatoren **AND** und **OR** diese Eigenschaft aufweisen. Weiterhin verdeutlicht die Abbildung, dass die **XOR-Funktion** nicht linear separierbar ist und somit nicht durch ein einzelnes Perzeptron lernbar ist. Dagegen können mehrere Perzeptronen, die hintereinander angeordnet sind, die XOR-

Funktion vollständig adaptieren. Abbildung 2.2 zeigt ein **Multilayer-Perzeptron**, welches die einzelnen Perzeptronen in **Schichten** anordnet. Die Ausgaben einer Schicht sind dabei mit den Eingaben der folgenden Schicht verknüpft. Die Begriffe Multilayer-Perzeptron und neuronales Netz werden häufig als Synonym betrachtet.



(a) x_1 AND x_2 (b) x_1 OR x_2 (c) x_1 XOR x_2

Abbildung 2.2: Separierbarkeit von booleschen Operatoren

2.1.2 Architektur

Für den Einsatz eines neuronalen Netzes muss zunächst der Aufbau und die Struktur des Netzes definiert werden. Die Architektur eines Netzes und die damit verbundenen Designentscheidungen gehen dabei weit über eine Festlegung der Neuronenanzahl hinaus. Eine geeignete Architektur für eine Anwendung zu finden, ist oftmals ein schwieriger Prozess und weiterhin wichtiger Bestandteil der Forschung. Dies wird unter anderem daran deutlich, dass fortlaufend verbesserte Architekturen für den ILSVRC[1], einem Wettbewerb zur Bildklassifizierung und Objekterkennung, veröffentlicht werden [KSH12; SZ14; Zop+17].

Die nachfolgende Liste stellt wichtige Faktoren bei der Modellierung einer Architektur vor, die anwendungsspezifisch variiert werden können. In der Literatur heißen diese Faktoren **Hyperparameter**. Diese umfassen aber auch zusätzliche Faktoren zur Beeinflussung des Trainingsprozesses. Arbeiten wie [FH19] oder [Ber+11] befassen sich mit der Optimierung dieser Faktoren, sogenanntes **Hyperparameter-Tuning**.

- Ein- und Ausgabe des Netzes
- Anzahl / Auswahl der verdeckten Schichten
- Auswahl Neuronen bzw. Aktivierungsfunktionen

1 Large Scale Visual Recognition Challenge

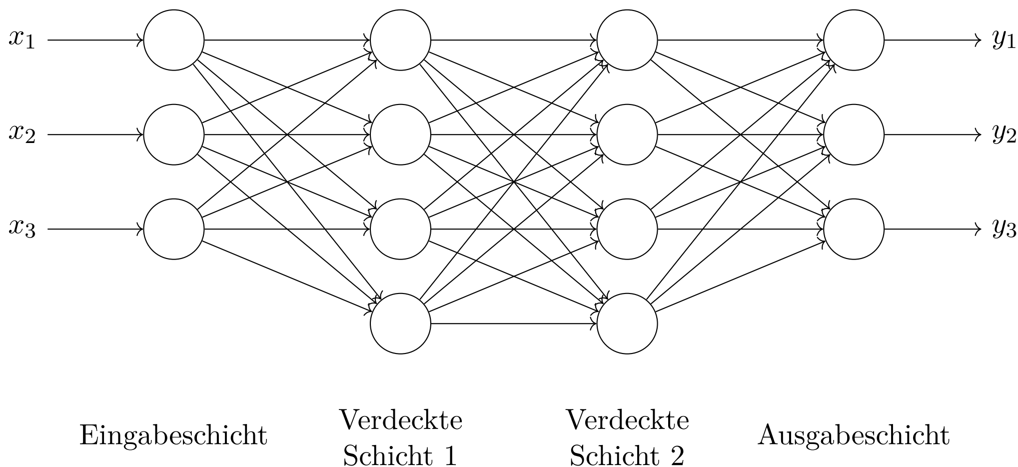


Abbildung 2.3: Illustration eines neuronalen Netzes mit zwei verdeckten Schichten

Ein- und Ausgabe des Netzes

Die Anzahl der Eingabeneuronen wird durch die Ausgabe der **Vorverarbeitung** festgelegt. Eine typische Vorverarbeitung bei bildbasierten Daten besteht darin, die Bilder auf ein gewünschtes Seitenverhältnis zuzuschneiden und anschließend auf eine Einheitsgröße zu skalieren. Um die Eingabedaten zu verringern, kann zusätzlich eine Dimensionsreduktion der Farbkanäle auf Graustufen durchgeführt werden. Aufgrund des Informationsverlustes können dann aber einige auf Farbmuster basierende Merkmale nicht mehr erlernt werden. Im Bereich der automatischen Spracherkennung dagegen werden aufwändigere Techniken eingesetzt. Huang et al. zeigen eine mögliche Pipeline an Vorverarbeitungsschritten [HAH01].

Die Ausgabe hängt ausschließlich von der Aufgabe des Netzes ab. Wenn das Netz zur **Klassifikation** eingesetzt wird, kann die Ausgabeschicht bei n verschiedenen Klassen aus n binären Ausgabeneuronen bestehen. Die dafür benötigte Heaviside-Aktivierung ist in Abschnitt 2.1.1 bereits beschrieben worden. Durch Umwandlung der Aktivierung in eine sigmoide Funktion kann jedes Neuron auch individuelle Klassenwahrscheinlichkeiten erlernen (Abbildung 2.4c). Formel 2.2 zeigt eine weitere häufig eingesetzte Aktivierungsfunktion, die **Softmax-Funktion**. Auch dabei wird jedem Ausgabeneuron i eine Klasse zugeordnet. Anstatt eine einzelne Wahrscheinlichkeit auszugeben, wird hierbei das Ergebnis der Exponentialfunktion über alle Ausgabeneuronen normalisiert. Dadurch entsteht als Gesamtausgabe eine Wahrscheinlichkeitsverteilung über alle Klassen.

$$softmax(z)_i = \frac{\mathrm{e}_i^z}{\sum\limits_{k=1}^{n} \mathrm{e}_k^z} \tag{2.2}$$

Anzahl / Auswahl der verdeckten Schichten

Verdeckte Schichten (*engl. hidden layer*) werden alle Schichten zwischen Ein- und Ausgabeschicht bezeichnet. Die Verbindungen zwischen den Schichten definieren den Informationsfluss in einem neuronalen Netz. **Feedforward-Netze** bezeichnet man die Netze, in denen die Informationen in eine Richtung fließen. Die Graphstruktur, die von den einzelnen Schichten dabei aufgespannt wird, ist azyklisch. Eine weitere Variante an neuronalen Netzen heißen **rekurrente Netze**. Diese können zyklische Verbindungen in frühere Schichten haben und somit zuvor berechnete Ausgaben als Eingabe erhalten. Insbesondere bei der Verarbeitung von zeitlichen Abfolgen werden rekurrente neuronale Netze eingesetzt. Detailliertere Erläuterungen dazu befinden sich in [GBC16].

Die Anzahl an verdeckten Schichten wird häufig als die **Tiefe** eines neuronalen Netzes bezeichnet. Im Gegensatz zu der Ein- und Ausgabe lässt sich eine optimale Anzahl an verdeckten Schichten und Neuronen nicht an der Aufgabe des Netzes ableiten. Nach dem *Universal Approximation Theorem* ist in der Theorie bereits eine Schicht ausreichend. Goodfellow fasst dies so zusammen:

> „A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.“[GBC16]

Die von Goodfellow angesprochenen Probleme führten dazu, dass bereits frühe Neuronale Netze wie das LeNet-5 [Lec+98] aus mehreren Schichten bestehen. Szegedy et al. haben mit GoogleNet gezeigt, dass auch Netze mit einer Tiefe von 22 effektiv trainiert werden können [Sze+14]. Andererseits existieren auch kompakte Netze wie SqueezeNet, die eine hohe Genauigkeit aufweisen können [Ian+16]. Allgemein gibt es keine optimale Vorgehensweise bei der Festlegung der verdeckten Schichten und als Teil des Hyperparameter-Tunings ist dies weiterhin Gegenstand der Forschung [GR18].

Im Zusammenhang mit der Anzahl Neuronen pro Schicht sowie der Gesamtzahl an Neuronen kann **Underfitting** und **Overfitting** auftreten. Underfitting tritt genau dann auf, wenn die Neuronenanzahl und somit die Kapazität des Netzes nicht ausreicht, um alle relevanten Muster zu erlernen. Overfitting dagegen meint, dass ein neuronales Netz viele Neuronen hat und durch angewendete Trainingsverfahren zu stark an die Trainingsdaten angepasst ist. Dabei erlernt das Netz spezifische Muster, welche sich nicht verallgemeinern lassen. Es erzielt zwar dadurch auf den Trainingsdaten gute Ergebnisse, schneidet jedoch im praktischen Einsatz schlecht ab. Lösungsansätze zur Vermeidung von Overfitting sind sogenannte *Dropout-Layer* [Sri+14], Regularisierung [GBC16], Kreuzvalidierung [KV95] oder ein frühzeitiges Beenden des Trainings [KV95]. Als Orientierungshilfe für eine geeignete Anzahl an Neuronen in verdeckten Schichten

veröffentlichte Heaton die Faustregel, eine Zahl zwischen der Ein- und Ausgabegröße zu wählen [Hea08].

Auswahl Neuronen bzw. Aktivierungsfunktionen

Das in Abschnitt 2.1.1 vorgestellte Perzeptron nutzt die Heaviside-Funktion für eine binäre Ausgabe. Ein entscheidender Nachteil der Stufenfunktionen ist, dass diese weder stetig noch differenzierbar ist. Beides sind allerdings Voraussetzungen für gradientenbasierte Lernverfahren. Eine naheliegende Alternative ist die Identitätsfunktion $\phi(z) = z$. Lineare Funktionen und damit auch die Identitätsfunktion sind jedoch genauso ungeeignet. Dies wird bei der Betrachtung der Tiefe eines derartigen Netzs deutlich. Ein Neuron mit linearer Aktivierungsfunktion ist eine simple Linearkombination über alle Eingaben. Verknüpfte Schichten sind somit also verkettete Linearkombinationen, die sich zu einer einzelnen Schicht zusammenfassen lassen. Um komplexe Abbildungen zwischen Ein- und Ausgaben modellieren zu können, nutzen neuronale Netze deshalb nichtlineare Funktionen. Die Sigmoidfunktion erfüllt diese Eigenschaft und kann als geglättete Stufenfunktion aufgefasst werden. Der entscheidende Vorteil ist jedoch der eingeschränkte Wertebereich zwischen 0 und 1. Dadurch können Wahrscheinlichkeiten in der Ausgabe leicht realisiert werden. Weiterhin hat die Funktion eine normalisierende Wirkung, sodass keine beliebig hohen Aktivierungen vorkommen können.

$$\phi_{sig}(z) = \frac{1}{1 + \mathrm{e}^{-z}} \tag{2.3}$$

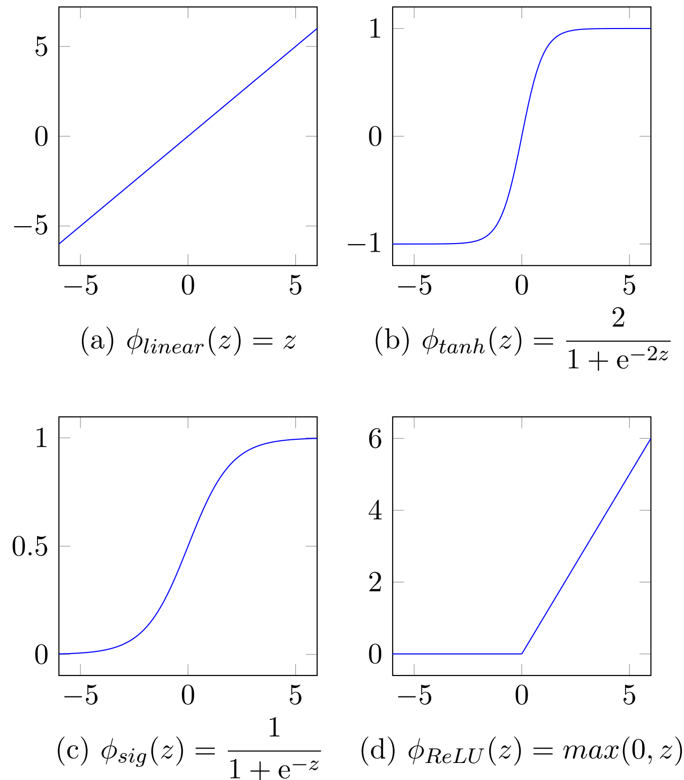
Die tanh-Funktion ist eine Alternative zur Sigmoidfunktion. Der Wertebereich $[-1, 1]$ ermöglicht, dass auch negative Aktivierungen erlernt werden können. Zusätzlich erleichtert die Ähnlichkeit zur Identitätsfunktion um den Nullpunkt das Training mit tanh [GBC16]. Beide genannten Funktionen haben jedoch bei Extremwerten das Problem, dass die Gradienten sehr klein werden (*engl. vanishing gradients*), was Auswirkungen auf den Lernprozess hat [GBC16].

$$\phi_{tanh}(z) = \frac{2}{1 + \mathrm{e}^{-2z}} \tag{2.4}$$

Eine weitere Aktivierungsfunktion ist ReLU (*Rectified Linear Unit*). Bei Eingaben größer oder gleich 0 verhält sich ReLU wie die Identitätsfunktion. Die Nichtlinearität entsteht dadurch, dass alle negativen Werte auf 0 abgebildet werden. ReLU ist aktueller Stand der Technik und wird in nahezu jeder modernen Architektur eingesetzt.

$$\phi_{ReLU}(z) = max(0, z) \tag{2.5}$$

Abbildung 2.4 zeigt einen Vergleich der Aktivierungsfunktionen.



(a) $\phi_{linear}(z) = z$ (b) $\phi_{tanh}(z) = \frac{2}{1+e^{-2z}}$

(c) $\phi_{sig}(z) = \frac{1}{1+e^{-z}}$ (d) $\phi_{ReLU}(z) = max(0, z)$

Abbildung 2.4: Aktivierungsfunktionen im Vergleich

2.1.3 Forward-Propagation

Der Prozess wie ein neuronales Netz $\mathcal{N}$ mit Architektur $\mathcal{A}$ eine Ausgabe generiert, heißt **Forward-Propagation**. Der Name leitet sich davon ab, dass die Informationen in einem Feedforward-Netz von der Eingabe über die verdeckten Schichten bis hin zu der Ausgabe fließen. Beginnend bei der Eingabe können die Aktivierungen der nachfolgenden Schichten nacheinander berechnet werden.

Formel 2.6 zeigt zunächst die Aktivierung eines allgemeinen Neurons mit Eingabevektor $\mathbf{x}$, Gewichtsvektor $\mathbf{w}$, Bias-Wert b und Aktivierungsfunktion ϕ.

$$a = \phi(\mathbf{w} \cdot \mathbf{x} + b) \tag{2.6}$$

Diese Definition lässt sich nun auf die Schichten des Netzes übertragen (Formel 2.7). Der Eingabevektor einer Schicht entspricht dabei den Aktivierungen der vorhergehenden Schicht $\mathbf{a}^{l-1}$. Die Gewichte aller Neuronen der Schicht werden zu einer Gewichtsmatrix $\mathbf{W}^l$ und die Bias-Werte zu dem Vektor $\mathbf{b}^l$ zusammengefasst. Die Aktivierungsfunktion $\phi(\mathbf{z})$ wird nun elementweise auf dem Vektor $\mathbf{z}$ angewendet.

$$\mathbf{a}^l = \phi(\underbrace{\mathbf{W}^l \mathbf{a}^{l-1} + b^l}_{\mathbf{z}}) \tag{2.7}$$

Das Ergebnis der Forward-Propagation bzw. die Ausgabe des neuronalen Netzes $\mathcal{N}(\mathbf{x})$ ist die Aktivierung der letzten Schicht $\mathbf{a}^L$.

2.1.4 Training

Ein neuronales Netz enthält durch die Gewichte und Bias-Werte viele **Parameter**, die beliebige Werte annehmen können. Das Ziel des Lernprozesses ist es, eine möglichst optimale Konfiguration mit Hilfe einer Trainingsdatenmenge X zu finden. Da tiefe neuronale Netze mehrere Millionen Parameter haben können, ist der Suchraum der Konfigurationen zu groß, um diese auszutesten. Stattdessen berechnen moderne Verfahren den Einfluss jedes Parameters auf eine globale **Kostenfunktion**. Basierend auf den Gradienten werden bei einer fehlerhaften Ausgabe die Parameter des gesamten Netzes angepasst. Parameter, welche stärker zu dem Fehler beitragen, werden dementsprechend stärker korrigiert.

Zur Berechnung der einzelnen Gradienten wird der **Backpropagation-Algorithmus** angewendet [R+88]. Ein klassisches Verfahren zur Anpassung der Parameter ist **Stochastic Gradient Descent**, welches in den folgenden Abschnitten beschrieben wird. Die verwendeten Notationen orientieren sich an der Arbeit von Nielsen [Nie15].

Kostenfunktion

Die Berechnungen aus Abschnitt 2.1.3 können dafür verwendet werden, Ausgaben für die Trainingsdaten X zu generieren. Um die Ergebnisse zu evaluieren, müssen die Trainingsdaten annotiert sein. Dies bedeutet, dass für jeden Eingabevektor $\mathbf{x}_i$ eine erwartete Ausgabe $\mathbf{y}_i$ vorhanden sein muss. Bei Klassifikationen entspricht dies der Referenzklasse und wird **Label** genannt.

Aus diesen Informationen kann eine Kostenfunktion C aufgestellt werden, die den mittleren Fehler des neuronalen Netzes über einer Trainingsdatenmenge angibt. Nielsen zeigt Vorteile von verschiedenen Kostenfunktionen auf [Nie15]. Ein Beispiel für eine

solche Funktion ist Formel 2.8, die mittlere quadratische Fehlerfunktion (*engl. mean-squared-error*). $\|\cdot\|$ ist die L1-Norm eines Vektors.

$$C = \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{y}_i - \mathcal{N}(\mathbf{x}_i)\|^2 \ , \quad n = |X| \tag{2.8}$$

Backpropagation

Aus der Kostenfunktion können nun die Gradienten bezüglich der Parameter berechnet werden. Für konkrete Parameter ist dies die Veränderung der Gesamtkostenfunktion durch die Veränderung eines einzelnen Gewichtes $\frac{\partial C}{\partial w}$ oder Bias-Wertes $\frac{\partial C}{\partial b}$. Die Berechnung kann durch die Kettenregel zerteilt werden.

$$\frac{\partial C}{\partial w_{kj}^l} = \underbrace{\frac{\partial C}{\partial z_k^l}}_{(1)} \underbrace{\frac{\partial z_k^l}{\partial w_{kj}^l}}_{(2)} = \frac{\partial C}{\partial z_k^l} a_j^{l-1} \tag{2.9}$$

w_{kj}^l beschreibt dabei das jte Gewicht im kten Neuron der Schicht l. Term (2) ist äquivalent zu a_j^{l-1}, da w_{kj}^l die Gewichtung der eingehenden Verbindung a_j^{l-1} ist und bei einer Veränderung alle weiteren Gewichte konstant bleiben. Term (1) ist der lokale Fehleranteil von z des kten Neurons und wird in Formel 2.10 als δ_k^l definiert.

$$\delta_k^l = \frac{\partial C}{\partial z_k^l} = \frac{\partial C}{\partial a_k^l} \frac{\partial a_k^l}{\partial z_k^l} = \frac{\partial C}{\partial a_k^l} \ \phi'(z_k^l) \tag{2.10}$$

$\frac{\partial C}{\partial a_k^l}$ ist der Fehleranteil aller nach l folgenden Schichten. Daraus folgt für die Berechnung, dass für jede Schicht l Fehleranteile aller Schichten $l^+ > l$ bereits berechnet sein müssen. Beginnend bei der Ausgabeschicht ergibt sich ein rekursives Schema. Dies ist Grund für den Namen **Backpropagation-Algorithmus**.

$$\begin{aligned} \frac{\partial C}{\partial a_k^l} &= \sum_m \frac{\partial C}{\partial z_m^{l+1}} \frac{\partial z_m^{l+1}}{\partial a_k^l} \\ &= \sum_m \frac{\partial C}{\partial z_m^{l+1}} \ w_{mk}^{l+1} \\ &= \sum_m \delta_m^{l+1} \ w_{mk}^{l+1} \end{aligned} \tag{2.11}$$

Als Startpunkt der Rekursion dient δ_k^L. Der letzte fehlende Term ist wieder $\frac{\partial C}{\partial a_k^L}$, welcher die Veränderung der Kostenfunktion bei Veränderung der Ausgabe beschreibt. Bei

gegebener Kostenfunktion kann dieser mit Hilfe der Ableitung direkt bestimmt werden. Abschließend ergeben sich die Gradienten für w^l_kj (Formel 2.12) und b^l_k (Formel 2.13).

$$\frac{\partial C}{\partial w^l_{kj}} = \delta^l_k a^{l-1}_j \tag{2.12}$$

$$\frac{\partial C}{\partial b^l_k} = \frac{\partial C}{\partial z^l_k} \underbrace{\frac{\partial z^l_k}{\partial b^l_k}}_{1} = \delta^l_k \tag{2.13}$$

Stochastic Gradient Descent

Bei gegebener Trainigsdatenmenge lässt sich durch den Backpropagation-Algorithmus ein Gradientenvektor ∇C der Kostenfunktion über alle Parameter λ aufstellen. Dieser beschreibt die Richtung des stärksten Anstiegs der Kostenfunktion. Ziel eines Gradientenabstiegsverfahren ist es, mit diesem Vektor die Kostenfunktion zu minimieren. Dazu werden alle Parameter in Richtung des negativen Gradientenvektors angepasst. Jedoch bewegt sich die Kostenfunktion dadurch auf ein lokales Minimum zu. In der Praxis hat sich dies aber als ausreichend erwiesen [PG17].

$$\lambda' = \lambda - \eta \nabla C \tag{2.14}$$

Die **Lernrate** η bestimmt die Schrittgröße der Parameteranpassung, die in jeder Trainingsphase durchgeführt wird. Kleine Lernraten führen zu kleinen Schritten im Lernprozess, was sich negativ auf die Laufzeit des Trainings auswirken kann. Große Schritte dagegen können ein Überspringen des gesuchten Minimums bewirken. Eine mögliche Strategie ist es, mit einer großen Lernrate anzufangen und diese im Laufe des Trainingsprozesses zu verkleinern.

Das gesamte Trainingsverfahren lässt sich in folgenden Schritte zusammenfassen:

1. **Initialisierung**: Alle Parameter werden mit zufälligen Werten initialisiert. In der Literatur sind auch weitere Strategien zur Initialisierung zu finden [GBC16].

2. **Forward Propagation**: In diesem Schritt werden die Ausgaben des Netzes für die Trainingsdatenmenge bestimmt.

3. **Kostenfunktion berechnen**: Die Fehler des vorherigen Schrittes werden mit Referenzausgaben durch eine gewählte Kostenfunktion bestimmt und gemittelt.

4. **Backpropagation**: Die Gradienten der Kostenfunktion werden berechnet.

5. **Parameteranpassung**: Formel 2.14 wird angewendet.

6. Wiederhole Schritt 2-5 bis das Netz die gewünschte Genauigkeit erreicht oder eine bestimmte Anzahl an Iterationen (*engl. epochs*) abgeschlossen ist.

Die Berechnung der gemittelten Kostenfunktion über alle verfügbaren Trainingsdaten ist bei Datensätzen mit Millionen von Beispielen zu aufwändig. Da ein solcher Datensatz viele ähnliche Daten enthält bietet es sich an, eine kleine Stichprobe zu wählen. Die Größe dieses Bündels (*engl. Batch*) kann beliebig groß ausgewählt werden. *Stochastic gradient descent* nutzt pro Iteration nur ein einzelnen Trainingsbeispiel.

2.1.5 Convolutional Neural Networks

Eine besondere Variante der neuronalen Netze sind *Convolutional Neural Network*s (CNNs) . Für Probleme im Bereich der Bildverarbeitung erzielten CNNs bedeutende Erfolge [KSH12; SZ14; Zop+17], werden aber auch in Anwendungen der natürlichen Sprachverarbeitung [L+95] oder Dokumentenanalyse [S+03] eingesetzt.

Eine Schwäche von vollverbundenen neuronalen Netzen ist, dass diese sensitiv gegenüber Translationen sind. Deutlich wird dies beispielsweise an der Objekterkennung in Bildern. Vollverbundene Netze sind nicht in der Lage, die Merkmale von Objekten unabhängig von ihrer Position zu erlernen. Das Lernverfahren ist somit auf Trainingsdaten angewiesen, die Beispiele für alle Objekte in allen möglichen Positionen beinhalten. Dies ist in der Praxis zu umfangreich und nicht umsetzbar [GBC16].

Convolutional Layers

Die Hauptbestandteile einer CNN-Architektur sind Convolutional Layer. Die zugrundeliegende mathematische Operation ist die **Faltung**. Gefaltet wird dabei die Eingabe der Schicht mit einer festgelegten Anzahl an **Filtern** mit fester Größe. Abbildung 2.5 zeigt die Faltung bei zweidimensionaler Eingabe. In der Abbildung ist zu erkennen wie der Filter über die gesamte Eingabe geschoben wird und in jedem Schritt eine einzelnes Resultat produziert. Die Filtergröße k gibt dabei den Abstand um den aktuellen Punkt an, in welchem Werte in die Berechnung einbezogen werden.

Ziel der Schicht ist es, räumliche Merkmale mit Hilfe der Filter zu extrahieren. Die erlernten Filterwerte geben an, wie hoch jeder Wert in der Umgebung eines Eingabepunkts dafür gewichtet werden soll. Aus diesem Grund nennt man die Resultate der Faltung **Feature Maps**. Formel 2.15 fasst die Rechenoperationen zusammen, die für einen einzelnen Wert einer Feature Map *out* durchgeführt werden. *in* ist die Eingabematrix,

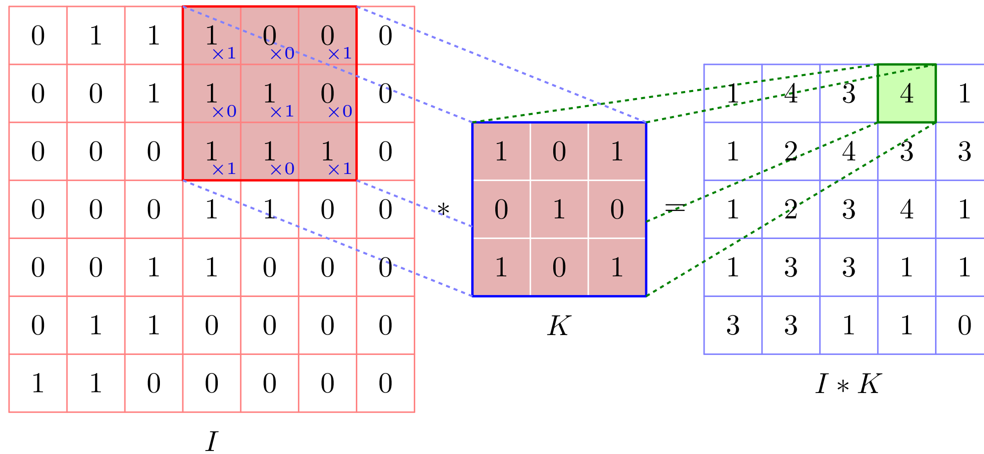


Abbildung 2.5: 2D-Faltung einer Eingabe I mit Kernel K

h die Filtermatrix. Zur Veranschaulichung ist eine zweidimensionale Eingabe gewählt. Dies lässt sich jedoch auf eine beliebige Anzahl an Dimensionen erweitern.

$$out[x, y] = \sum_{j=-k}^{k} \sum_{i=-k}^{k} h[i,j] * in[x + i, y + j] \qquad (2.15)$$

Da die Faltoperation eine lineare Transformation ist, ist wie bei einer vollverbundenen Schicht abschließend eine Aktivierung notwendig. Beispiele für Aktivierungen befinden sich in Abschnitt 2.1.2. Bei modernen Architekturen ist ReLU verbreitet.

Pooling Layer

Ein Convolutional Layer kann kleine räumliche Merkmale unabhängig ihrer Position erlernen. Dennoch wirkt sich die genaue Position des erkannten Merkmals auf die berechnete Feature Map aus. Ein Merkmal, was in der Eingabe um einen Wert verschoben wurde, wird in der zugehörigen Feature Map um den gleichen Wert verschoben sein. Pooling Layer führen dazu, dass die Feature Maps invariant gegenüber kleinen Translationen werden [GBC16].

Die durchgeführte Operation ist vergleichbar mit der Faltung. Anstatt einen gewichteten Filter anzuwenden, wird bei **Max Pooling** das Maximum über der Nachbarschaft gebildet. Kleine Verschiebungen in der Nachbarschaft verändern hierbei das Ergebnis der Ausgabe nicht.

Darüber hinaus wird Pooling noch verwendet, um räumliche Umgebungen zusammenzufassen (*engl. downsampling*). Dies verringert einerseits die Größe der Feature Maps und spart damit in den darauffolgenden Schichten Rechenzeit. Andererseits macht es

erkannte Merkmale ebenfalls robuster gegenüber räumlichen Verschiebungen. Abbildung 2.6 zeigt die Anwendung von Max Pooling auf einer zweidimensionalen Eingabe.

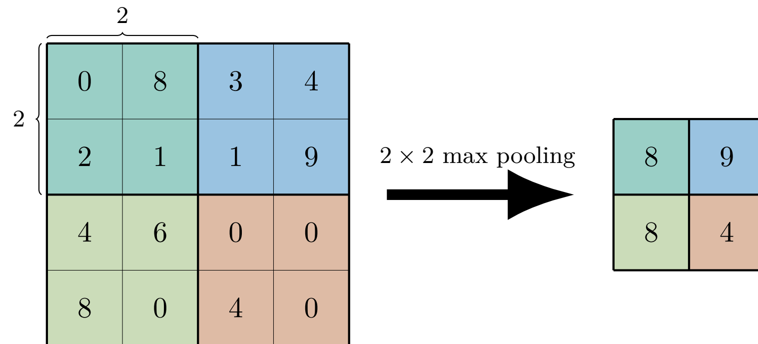


Abbildung 2.6: Max Pooling der Größe 2x2 mit Downsampling

3 Verwandte Arbeiten

Die Reduktion von neuronalen Netzen ist ein breites Forschungsgebiet mit unterschiedlichen Ansätzen. Ziel dieses Kapitels ist es, einen Einblick in verwandte Arbeiten zu geben. Ein besonderer Fokus liegt dabei auf dem Entfernen von Neuronen und Verbindungen, da dies in den nachfolgenden Kapiteln im Vordergrund steht.

3.1 Matrizen- und Tensorzerlegung

Einige mathematische Konzepte hinter neuronalen Netze finden ihren Ursprung in der linearen Algebra. Dies zeigt beispielsweise die Inferenz von vollverbundenen Schichten, die in der Praxis als Matrixmultiplikation realisiert wird. So ist es naheliegend, Ansätze zur Faktorisierung von Matrizen aus der linearen Algebra zu nutzen, um ein neuronales Netz zu komprimieren oder um Berechnungen einzusparen.

Ein mögliches Konzept, Matrizen in Faktoren zu zerlegen, ist die Singulärwertzerlegung (*engl. Singular Value Decomposition* (SVD)). Dabei wird eine beliebige Matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$ in drei Faktoren zerlegt:

$$\mathbf{A} = \mathbf{USV}^T \tag{3.1}$$

$\mathbf{U}$ und $\mathbf{V}$ sind orthonormale Matrizen und $\mathbf{S} \in \mathbb{R}^{n \times m}$ eine mit Nullen erweiterte Diagonalmatrix, welche die Singulärwerte σ_i enthält. Für nähere Erläuterungen und ein Verfahren zur Berechnung der einzelnen Matrizen wird auf ergänzende Literatur wie [YTT11] oder [RV15] verwiesen.

Im Kontext von neuronalen Netzen ist eine weitere Eigenschaft von SVD relevant. Mit SVD kann aus einer Matrix $\mathbf{A}$ eine Approximation $\mathbf{A}_r = \mathbf{U}_r\mathbf{S}_r\mathbf{V}_r^T$ mit niedrigerem Rang r bestimmt werden. Wenn dabei die r größten Singulärwerte erhalten bleiben, ist die L_2-Norm der Differenz $\mathbf{A} - \mathbf{A}_r$ gegenüber beliebigen Matrizen $\mathbf{B}_r$ mit Rang r minimal (Formel 3.2) [BHH19].

$$\|\mathbf{A} - \mathbf{A}_r\|_2 \leq \|\mathbf{A} - \mathbf{B}_r\|_2 \tag{3.2}$$

Abbildung 3.1 zeigt wie Yaguchi et al. die Approximation bei der Verkleinerung von neuronalen Netzen nutzen. Die einzelnen Schichten des Netzes werden jeweils durch SVD in zwei Matrizen faktorisiert. Dadurch entsteht zwischen allen ursprünglichen Schichten eine weitere Schicht der Größe r ohne Aktivierungsfunktion. Der benötigte Parameterbedarf sinkt von mn Parametern auf $(m+n) * R$.

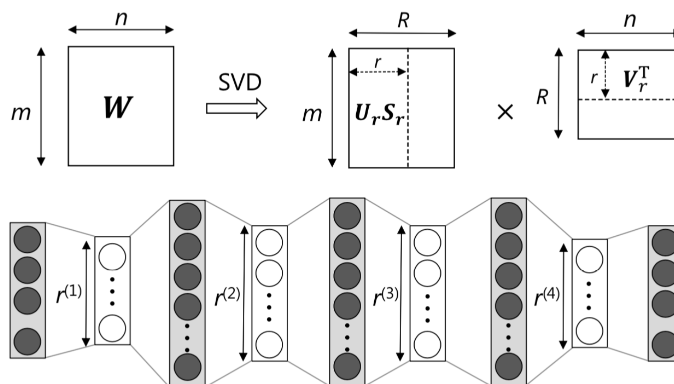


Abbildung 3.1: Einsatz von SVD in neuronalen Netzen aus [Yag+19]

Lin et al. wenden SVD und erneutes Training an, um die vollverbundenen Schichten in *AlexNet* und *VGG19* zu komprimieren [Lin+16]. Denton et al. gelang es mit SVD, Convolutional Schichten in einem CNN um den Faktor 2-3x zu beschleuningen [Den+14].

Ein entscheidender Nachteil von SVD ist, dass es sich nicht direkt auf höherdimensionale Tensoren übertragen lässt [Lee+19]. Erst mit Adaptionen wie in [Den+14] ist SVD auch auf Convolutional Schichten anwendbar. Kim et al. nutzten stattdessen die Tucker-Zerlegung für eine Approximation von Convolutional Schichten [Kim+15]. Ein weiterer Beitrag der Autoren ist, dass das vorgestellte Verfahren auch zur Kompimierung ganzer Netze und nicht nur einzelner Schichten wie in [Den+14] oder [Lin+16] eingesetzt werden kann.

3.2 Pruning

Eine häufig eingesetze Technik zur Reduktion von neuronalen Netzes ist das Ausdünnen (engl. *Pruning*) eines bereits trainierten Netzes. Li et al. teilen dortige Arbeiten in zwei Kategorien ein: Einerseits das Prunen von Verbindungen, andererseits das Prunen ganzer Neuronen [Li+18]. Darüber hinaus lässt sich noch eine 3. Kategorie definie-

ren, die Arbeiten zur Ausdünnung von Eingaben umfassen. Derartige Techniken sind beispielsweise in [KML04] oder [AK11] zu finden.

3.2.1 Pruning von Verbindungen

In den vergangenen 30 Jahren sind zahlreiche Methoden konstruiert worden, welche unrelevante Verbindungen zwischen Neuronen identifizieren und entfernen.

Wertebasiertes Pruning: Ein simples, häufig eingesetzes Verfahren ist es, Parameter basierend auf ihren jeweiligen Werten zu entfernen [LDS90]. Die zugrundeliegende Hypothese ist, dass Parameter mit niedrigen Werten auch einen geringen Einfluss auf die Ausgabe des Netzes haben. Die Entfernung solcher Parameter habe deshalb eine weniger starke Auswirkung auf die Akkuratheit des Netzes als die Entfernung anderer Parameter mit höheren Werten.

Han et al. stellen eine Methode vor, welche wertebasiertes Pruning einbindet. Das Verfahren besteht aus mehreren Schritten [Han+15]:

1. In einer initialen Trainingsphase wird das gesamte Netz trainiert.
2. Die trainierten Werte bilden die Wissensgrundlage der Beurteilung, ob eine Verbindung relevant ist. Basierend auf den Werten wird jede Verbindung unterhalb eines Schwellwertes verworfen.
3. Im abschließenden Schritt werden die übrigen Verbindungen mit reduzierter Lernrate neu trainiert, um die Fehler der entfernten Parameter zu kompensieren.

Dadurch erreichten Han et al. eine Reduktion der Parameteranzahl von *AlexNet* um den Faktor 5. Weitere Verbesserungen erzielten sie, indem die letzten beiden Schritte iterativ wiederholt wurden. Insgesamt erzielten sie somit eine Reduktion von *AlexNet* um den Faktor 9 und von *VGG16* um den Faktor 13.

Anstatt geprunte Verbindungen direkt zu verwerfen, verfolgen Guo et al. einen anderen Ansatz [GYC16]. Neben der Gewichtsmatrix symbolisiert eine Binärmatrix für jede Verbindung den Status, ob diese geprunt wird oder nicht. Während des gesamten vorgestellten Verfahrens bleiben alle ursprünglichen Verbindungen in der Gewichtsmatrix enthalten und werden in einem angepassten Gradientenabstiegsverfahren ebenfalls mit trainiert. So kann es vorkommen, dass sich der Wert einer zuvor geprunten Verbindung durch das Training erhöht und in einem *splicing*-Schritt wieder eingefügt wird. Ohne Genauigkeitsverlust gelang dadurch eine Parameterreduktion von *AlexNet* um den Faktor 17,7. Bei einer *LeNet-5*-ähnlichen Architektur reduzierten die Autoren die Parameteranzahl um den Faktor 108.

Eine Schwierigkeit bei der Umsetzung von wertebasiertem Pruning besteht darin, einen geeigneten Schwellwert zu finden [Li+18]. Dieser dient dabei als Entscheidungsgrenze. Verbindungen, deren Werte unter diese Grenze fallen, werden entfernt. Ein hoher Schwellwert führt dazu, dass zu viele Neuronen abgeschnitten werden und die Genauigkeit des Netzes nachlässt. Ein Verfahren dagegen, welches einen zu niedrigen Schwellwert verwendet, schneidet zu wenige Parameter ab und erreicht suboptimale Kompressionsraten. Außerdem zeigen Arbeiten wie [Li+18] oder [GYC16], dass der optimale Wert für jede Schicht im Netz variieren kann.

Li et al. beschreiben in [Li+18] eine Methode, diese Schwellwertsuche zu automatisieren. Dazu betrachten diese das Finden eines geeigneten Schwellwertes als restriktives Optimierungsproblem. Grundlegendes Ziel der Optimierung ist es, einen Parametersatz $\mathbf{W}^*$ zu finden, der die Anzahl der Verbindungen minimiert. Gleichzeitig gilt die Einschränkung, dass die Genauigkeit des Netzes nicht unter eine festgelegte Schranke δ fallen darf.

$$\mathbf{W}^* = \operatorname*{arg\,min}_{\mathbf{W}' \subseteq \mathbf{W}} |\mathbf{W}'| \qquad \text{s.t.} \quad f(\mathbf{W}) - f(\mathbf{W}') \leq \delta \tag{3.3}$$

$\mathbf{W}'$ als Teilmenge von $\mathbf{W}$ bezeichne dabei den Parametersatz eines geprunten Netzes, $|\mathbf{W}|$ die Größe von $\mathbf{W}$ und f ist eine Evaluationsfunktion, welche mit Hilfe eines Datensatzes die Genauigkeit bei gegebenem $\mathbf{W}$ misst.

Bewertungsbasiertes Pruning: Bisher sind Verfahren vorgestellt worden, welche die Bedeutung einer Verbindung lediglich anhand des Wertes beurteilen. Bereits 1990 ist mit *Optimal Brain Damage* (OBD) ein Verfahren demonstriert worden, welches andere Bewertungskriterien nutzt [LDS90]. Darin ist die Bedeutung eines Parameters als die Veränderung der Kostenfuntion definiert, wenn dieser Parameter entfernt wird. Um nicht wiederholt die Kostenfunktion C evaluieren zu müssen, approximieren LeCun et al. die Veränderung δU des Parametervektors lokal durch eine Taylorreihe.

$$\begin{gathered}\Delta C = \underbrace{\frac{\partial C}{\partial u_i}\delta u_i}_{(1)} + \underbrace{\frac{1}{2}\sum_i h_{ii}\delta u_i^2}_{(2)} + \underbrace{\frac{1}{2}\sum_{i \neq j} h_{ij}\delta u_i \delta u_j}_{(3)} + \underbrace{O(\|\delta U\|^3)}_{(4)} \\ \text{mit} \quad h_{ij} = \frac{\partial^2 C}{\partial u_i \partial u_j}\end{gathered} \tag{3.4}$$

u_i bezeichnen dabei die einzelnen Komponenten von U, also einzelne Parameter. h_{ij} sind die Einträge einer Hesse-Matrix von U in Bezug auf die Kostenfunktion C. Um die erforderlichen Berechnungsschritte zu reduzieren, treffen LeCun et al. einige Annahmen über die Terme (1) bis (4). Die erste Annahme ist, dass OBD immer nach der

Konvergenz des Trainings ausgeführt wird. Somit befindet sich die Kostenfunktion C in einem lokalen/globalen Minimum, sodass die Gradienten aus Term (1) den Wert 0 haben. Weiterhin wird eine Unabhängigkeitsbehauptung aufgestellt, dass das Löschen eines Parameters keine Auswirkungen auf das Entfernen eines anderen Parameters hat. Demnach wäre Term (3) ebenfalls 0. Schließlich werden die Terme höherer Ordnung aus (4) vernachlässigt, sodass nur (2) als Bewertungskriterium eingesetzt wird. Die Bewertung eines einzelnen Parameters ist demnach $s_i = h_{ii}u_i^2/2$, da beim Löschen $\delta u_i^2 = u_i^2$ gilt.

Im Pruningprozess verläuft OBD in iterativen Phasen bis die gewünschte Komprimierung erreicht wird. Zu Beginn jeder Phase wird zunächst trainiert. Anschließend wird die Bewertung s_i für jeden Parameter berechnet und die Parameter mit der niedrigsten Bewertung aus dem Netzwerk entfernt.

In den vergangenen Jahrzehnten sind zahlreiche Erweiterungen zu OBD entstanden. Hassibi et al. demonstrieren mit *Optimal Brain Surgeon* (OBS) ein Verfahren, welches alle Ableitungen zweiter Ordnung verwendet [HS93]. Somit ist die Approximation der Zielfunktion im Vergleich zu OBD genauer, welches lediglich die Diagonalelemente der Hesse-Matrix verwendet. Um den hohen Rechen- und Speicherbedarf zu verringern, zielt [Has+93] darauf ab, die Hesse-Matrix durch Zerlegung in ihre Eigenwerte anzunähern. Dong et al. wenden OBS schichtweise an und betrachten die Veränderung der Zielfunktion für jede Schicht eines neuronalen Netzes [DCP17]. Ihnen gelang es dabei, LeNet5 bei einem Genauigkeitsverlust von 0,77% auf 0,9% der ursprünglichen Größe zu reduzieren.

Herausforderungen: Das Entfernen von Verbindungen verringert die Parameteranzahl und somit den benötigten Speicherplatz des Netzes. Im praktischen Einsatz führt dies allerdings nicht unmittelbar zu einer Verringerung der Laufzeit des neuronalen Netzes. Deutlich wird dies bei der Betrachtung von Frameworks wie TensorFlow und Keras. Diese nutzen unter anderem die leichte Parallelisierbarkeit von Matrizenmultiplikationen aus. Erst damit ist das Training und die Ausführung von neuronalen Netzen auf massiv paralleler Hardware (bspw. GPUs) möglich. Sobald jedoch einzelne Verbindungen entfernt werden, entstehen dünnbesetzte (*engl. sparse*) Matrizen. Zusätzliche Indices sind schließlich notwendig, was sich negativ auf die Komplexität und Effektivität auf CPUs und GPUs ausübt [Han+16].

3.2.2 Pruning von Neuronen

Statt einzelne Verbindungen zu entfernen, kann ein neuronales Netz auch durch das Entfernen ganzer Neuronen komprimiert werden. Neben den Parametern des entfernten Neurons einer Schicht l fallen dabei gleichzeitig auch die zugehörigen Gewichtungen

der Schicht $l+1$ weg. Dabei entstehen keine dünnbesetzten Matrizen, sodass auch eine direkte Reduzierung der benötigten Rechenoperationen stattfindet.

Prunen anhand von entfernten Verbindungen: Eine naheliegender Ansatz ist es, die beschriebenen Techniken zum Prunen von Verbindungen einzusetzen und anschließend Neuronen ohne eingehende Verbindung zu entfernen. [Han+15] prunen Neuronen mit diesem Ansatz. Allerdings steht das Entfernen von Verbindungen im Vordergrund, sodass dies eher als Zusatz der eingesetzten Methoden aufgeführt ist. In Kapitel 2.2 wird anhand der Werteverteilung der Gewichte innerhalb eines Neurons deutlich, dass dies bei wertebasierten Verfahren in den durchgeführten Experimenten keine kompetitiven Resultate liefert.

Wertebasiertes Prunen: Die Werte der zugehörigen Verbindungen eines Neurons können als Kriterium für die Auswahl der zu prunenden Neuronen verwendet werden. Dabei kann beispielsweise die L_1-Norm $\|\mathbf{w}_i\|_1$ des jeweiligen Gewichtsvektors $\mathbf{w}_i$ zum Akkumulieren der Werte genutzt werden. Vergleichbar mit dem wertebasierten Prunen von Verbindungen wird hiermit die Idee verfolgt, dass diese Neuronen weniger stark zu dem Endergebnis des neuronalen Netzes beitragen. Der entscheidende Vorteil daran ist, dass es sich um eine leicht zu berechnende Auswahlheuristik handelt. In [Eng01] wird diese Technik eingesetzt.

Prunen von ähnlichen Neuronen: Srinivas et al. untersuchen in [SB15] Redundanzen zwischen Neuronen und nutzen diese, um ähnliche Neuronen zusammenzufassen. Der einfachste Fall ist dabei, wenn innerhalb einer Schicht l zwei Neuronen i und j identische Gewichtsvektoren $\mathbf{w}_i^l = \mathbf{w}_j^l$ annehmen. Dann sind auch die Aktivierungen a_i^l und a_j^l identisch. Wenn man nun ein einzelnes Neuron k der Folgeschicht $l+1$ betrachtet, zeigt Formel 3.5, dass die Gewichte w_i und w_j vereint werden können. Nachdem dieser Schritt für alle k durchgeführt wird, ist das Neuron j überflüssig. Abbildung 3.2 zeigt ein simples Beispiel dieses Verfahrens.

$$\begin{aligned} a_k^{l+1} &= \phi\Big(w_0 * a_0^l + \ldots + w_i * a_i^l + w_j * a_j^l + \ldots + w_n * a_n^l + b\Big) \quad \text{mit } a_j^l = a_i^l \\ &= \phi\Big(w_0 * a_0^l + \ldots + (w_i + w_j) * a_i^l + \ldots + w_n * a_n^l + b\Big) \end{aligned} \tag{3.5}$$

Da in der Praxis identische Neuronen selten auftreten, definieren Srinivas et al. eine Ähnlichkeitsmatrix $\mathbf{M}$ mit den Elementen $s_{i,j} = \langle w_j^2 \rangle \|\mathbf{w}_i - \mathbf{w}_j\|_2^2$. $\|\mathbf{w}_i - \mathbf{w}_j\|_2$ ist die L_2-Norm des Abstandsvektors und $\langle w_j \rangle$ die durchschnittliche Gewichtung des Neurons j in der folgenden Schicht. In dem vorgestellten Verfahren werden die niedrigsten Werte aus der Matrix entnommen, die zugehörigen Neuronen kombiniert und $\mathbf{M}$ aktualisiert.

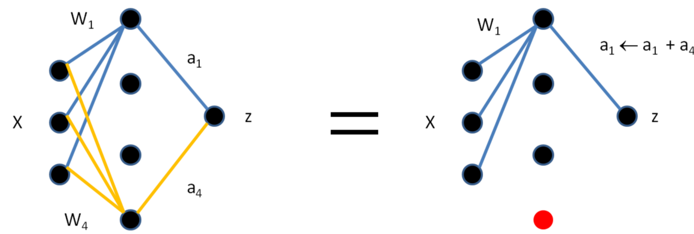


Abbildung 3.2: Zusammenführen von zwei gleichen Neuronen aus [SB15]

Eine Besonderheit gegenüber anderen Arbeiten ist, dass Srinivas et al. kein erneutes Training durchführen und es dennoch gelingt, etwa 35% der Neuronen von AlexNet mit minimalem Genauigkeitsverlust abzuschneiden.

Babaeizadeh et al. nutzen einen verwandten Ansatz. Statt der Ähnlichkeit zwischen den Parametern von Neuronen, verwenden sie die Korrelation zwischen Aktivierungen. Zusätzlich stärkten die Autoren die Bildung solcher Korrelationen, indem zufällige Ausgabeneuronen vor dem Training in das Netz eingefügt wurden. Insgesamt konnten sie die Parameter einer LeNet5-Architektur mit ihrem Verfahren um 97,75% reduzieren [BSC16].

3.2.3 Pruning von Filtern

Die Parameter von vollverbundenen Schichten machen häufig einen Großteil der Gesamtparameter aus. Aus diesem Grund beschränken sich Arbeiten wie [SB15] und [BSC16] ausschließlich auf das Prunen von Parametern in vollverbundenen Schichten. Dennoch ist die Parameteranzahl nur ein Faktor, der bei dem Einsatz eines neuronalen Netzes relevant ist. In VGGNet sind ca. 90% aller Parameter in vollverbundenen Schichten, diese tragen aber bei der Ausführung zu weniger als 1% der Fließkommaoperationen bei [Li+16]. Um also neben hohen Kompressionsraten auch eine Reduzierung der durchgeführten Rechenoperationen zu erzielen, veranschaulicht der kommende Unterabschnitt Methoden zum Prunen von Filtern.

Abbildung 3.3 verdeutlicht das Entfernen eines einzelnen Filters und die zugehörigen Parameter. Analog zu dem Entfernen eines Neurons in vollverbundenen Schichten entfallen neben den Parametern des Filters auch alle Gewichtungen der Filteraktivierungen in der darauffolgenden Schicht.

Grundsätzlich sind die bisher vorgestellten Methoden zur Entfernung von Neuronen auf das Prunen von Filtern übertragbar. Die Arbeit von Li et al. zeigt, dass durch eine wertebasierte Auswahl die Inferenzzeit einer VGG-16 Architektur, trainiert auf Cifar10, um 34% verringert werden kann [Li+16]. Das verwendete Bewertungsmaß s_i ist die Summe der L_1-Normen aller Kernel $\mathcal{K}$ (Formel 3.6). Schichtweise wird s_i für jeden Filter

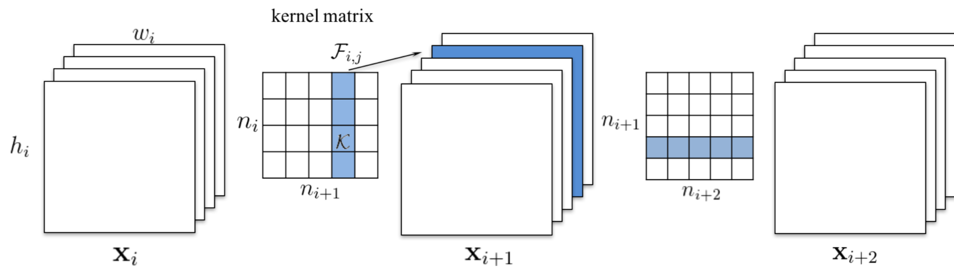


Abbildung 3.3: Prunen eines Filters $h_i = \mathcal{F}_{i,j}$ führt zu dem Entfernen der zugehörigen Parameter sowie aller verbundenen Kernel der folgenden Schicht. [Li+16].

h_i berechnet. Anschließend werden die Filter mit den m niedrigsten s_i entfernt und die entstandenen Fehler durch erneutes Training kompensiert.

$$s_i = \sum_{l=1}^{n_i} \|\mathcal{K}_l\| \tag{3.6}$$

Molchanov et al. demonstrieren in [Mol+16] eine Technik, welche äquivalent zu OBD eine Taylorapproximation der Kostenfunktion C verwendet. Dabei steht aber nicht das Prunen von Verbindungen im Vordergrund, sondern von gesamten Filtern h_i. Der bedeutende Unterschied besteht aber darin, dass allein der erste Term aus Formel 3.4 zur Bewertung verwendet wird. Bei OBD wurde dieser nicht berücksichtigt, da auch bei ganzen Filtern in einem (lokalen) Minima $\frac{\partial C}{\partial h_i} = 0$ und somit auch $y = \frac{\partial C}{\partial h_i} h_i = 0$ gilt. Jedoch trifft dies nicht auf die Varianz von y zu, die sich nach den Autoren empirisch als informatives Kriterium erwiesen hat. Statt die Varianz direkt zu berechnen, dient $|y|$ als Kriterium, welches proportional zur Varianz ist. Schließlich ergibt sich Formel 3.7 als Bewertung für einen Filter h_i.

$$s_i = \left| \frac{\partial C}{\partial h_i} h_i \right| \tag{3.7}$$

3.3 Quantisierung

Ein weiteres Verfahren zur Kompression von neuronale Netzen ist die Quantisierung der Parameter. Da diese meist als Fließkommazahlen in einer bestimmten Codierung vorliegen, ist es naheliegend, durch eine kleinere Codierung Speicherplatz zu sparen. Athar stellt in [Ath18] verschiedene Quantisierungsmethoden vor, die auf neuronale Netze angewendet werden können. Nach Athar sind dabei die **Auflösung / Schrittgröße** und **Bitbreite** wichtige Merkmale eines Verfahrens.

Vanhoucke et al. beschreiben in [VSM11] unter anderem eine Verringerung der Ausführungszeit eines NNs auf CPUs durch Quantisierung. Für die Repräsentation der Gewichte wurde eine 8-Bit Codierung ausgewählt. Den entstandenen Quantisierungsfehler kompensierten die Autoren durch erneutes Training. Courbariaux et al. trainieren ein neuronales Netz mit Gewichten, welche nur die Werte -1 und 1 annehmen können. Pro Gewicht wird also nur ein einzelnes Bit benötigt [Cou+16].

Lin et al. zeigen experimentell, dass im Kontext von neuronalen Netzen eine feste Bitbreite für alle Schichten schlechtere Resultate liefert als eine individuelle Bitbreite für jede Schicht [LTA15]. Sie erläutern außerdem für verschiedene Verteilungen, welche Quantisierungswerte und Auflösungen geeignet sind. Dies nutzen die Autoren, um in einem Optimierungsverfahren geeignete Bitbreiten für ein CNN zu ermitteln.

Han et al. kombinieren wertebasiertes Prunen mit Quantisierung [HMD15]. Nach einem iterativen Pruningprozess werden aus den Gewichten Cluster berechnet und jedes Gewicht wird dem Mittelpunkt eines Clusters zugeordnet. Abschließend wird die Kompressionsrate durch eine Huffman-Codierung der Indizes weiter verbessert. Die Größe von *VGG16* konnten die Autoren um den Faktor 59 komprimieren. Ein vergleichbares Verfahren, welches bereits im Pruningprozess quantisiert, ist in [TM18] zu finden.

Quantisierungstechniken sind allgemein nicht nur auf Parameter beschränkt. So wird in [LTA15] die Quantisierung auch auf Aktivierungen im gesamten Netz angewendet.

4 Reduktionsstrategien

In Kapitel 3 sind einige Strategien aus der Literatur zur Komprimierung eines trainierten neuronalen Netzes vorgestellt worden. In diesem Kapitel wird anhand von verschiedenen Experimenten die Qualität der einzelnen Verfahren untersucht. Dabei werden auch Modifikationen vorgestellt und evaluiert. Die Reduktionsverfahren zum Entfernen von Neuronen oder Filtern beschränken sich dabei zunächst auf einzelne Schichten. In Kapitel 5 werden anschließend Möglichkeiten erläutert, diese auf gesamte Netzwerke zu übertragen.

Inhaltlich gliedert sich das Kapitel wie folgt: Nach einer kurzen Motivation werden im zweiten Unterkapitel die übergeordneten Ziele der Experimente formal beschrieben und der grundlegende Versuchsaufbau erläutert. Alle weiteren Abschnitte enthalten die Ergebnisse der durchgeführten Experimente.

4.1 Motivation

Ein Problem der zahlreichen Arbeiten im Bereich der Reduktion von neuronalen Netzen ist, dass sich die darin durchgeführten Experimente stark voneinander unterscheiden und deshalb kaum ein Vergleich zwischen den Verfahren gezogen werden kann. Arbeiten wie [SB15], [Han+16], [GYC16] oder [Li+16] demonstrieren zwar ein neuartiges Verfahren, testen dieses allerdings auf unterschiedlichen Architekturen, Datensätzen oder Metriken. Dagegen vergleichen Molchanov et al. oder Dong et al. ihre Ansätze zwar mit anderen Techniken, der Vergleich ist jedoch auf einen Teil des Netzwerkes [Mol+16] oder aber ein einzelnes verwandtes Verfahren beschränkt [DCP17].

Ein weitere Motivation ist, dass bei verwandten Arbeiten und deren angegebenen Reduktionsraten der Versuchsaufbau ein wichtiger Faktor sein kann. Beispiele dafür sind die LeNet-5 Architekturen in [SB15] oder [GYC16]. In [SB15] unterscheidet sich die gesamte Architektur erheblich von dem ursprünglichen LeNet-5 aus [Lec+98]. Statt 120 Neuronen verfügt die größte vollverbundene Schicht 500 Neuronen. Das in [GYC16] untersuchte Netz besitzt nahezu das Zehnfache an trainierbaren Parametern. Es ist denkbar, dass sich derartige Modifikationen positiv auf die erzielten Resultate auswirken.

4.2 Grundlagen

Die Ausgangssituation aller in dieser Arbeit untersuchten Reduktionsverfahren ist ein neuronales Netz $\mathcal{N}$ mit einem trainiertem Parametersatz λ. Bei gewünschter Reduktionsrate r wird ein Parametersatz λ' gesucht, der die Kostenfunktion C minimiert.

$$\min_{\lambda'} C(\mathbf{X}, \mathbf{Y}|\lambda') \qquad \text{s.t.} \quad \|\lambda'\|_0 * r \leq \|\lambda\|_0 \tag{4.1}$$

Dabei ist $\mathbf{X}$ die Menge aller Eingabebeispiele mit der zugehörigen Menge an erwarteten Ausgaben $\mathbf{Y}$. Die l_0 Norm $\|\cdot\|_0$ und r werden verwendet, um die Anzahl der nicht-Null Parameter zu beschränken. Auffallend ist dabei, dass eine Verbesserung zu der ursprünglichen Kostenfunktion $C(\mathbf{X}, \mathbf{Y}|\lambda)$ nicht ausgeschlossen ist.

Schon bei geringer Parameteranzahl eines neuronalen Netzes ist es nicht möglich, dieses Optimierungsproblem erschöpfend zu lösen. Alle im vorherigen Kapitel beschriebenen Verfahren sind aus diesem Grund Heuristiken, die weitere Informationen über die einzelnen Parameter zur Bewertung heranziehen.

Das oben beschriebene Optimierungsproblem eignet sich für den Vergleich von Reduktionsstrategien gut, da bei gegebener Reduktionsrate r in einer direkten Gegenüberstellung die erzielten Kosten verglichen werden können. Dennoch ist für ein Tool zur Reduktion von neuronalen Netzen eine dynamische Reduktionsrate besser geeignet. Der Grund dafür ist, dass bei einer statischen Reduktionsrate die Höhe der Kostenfunktion nicht berücksichtigt wird. So kann bei hoher Reduktionsrate r der bestmögliche Parametersatz λ' durch entstehendes Underfitting zu derart hohen Kosten führen, dass der Einsatz des Netzes unrentabel wird. Stattdessen wird ein beliebig wählbarer Schwellwert ε festgelegt, der in Bezug auf die ursprüngliche Kostenfunktion $C(\mathbf{X}, \mathbf{Y}|\lambda)$ nicht überschritten werden darf. Es wird also eine optimale Reduktionsrate gesucht, die noch innerhalb der Grenzen des akzeptierten Kostenanstiegs liegt. Dies lässt sich wie folgt formulieren:

$$\max_{r} C(\mathbf{X}, \mathbf{Y}|\lambda') - C(\mathbf{X}, \mathbf{Y}|\lambda) < \varepsilon \tag{4.2}$$

4.2.1 Weitere Metriken

Beim Training eines neuronalen Netzes ist eine Kostenfunktion wie der mittlere quadratische Fehler oder die Kreuzentropie erforderlich. Um die Qualität eines neuronalen Netzes zu beurteilen, wird im Kontext der Klassifizierung allerdings in der Praxis eine andere Metrik ausgewählt. Die Genauigkeit (*engl. accuracy*), also der Prozentsatz korrekt klassifizierter Beispiele, ist besser interpretierbar und wird deshalb in allen

weiteren Experimenten der Kostenfunktion vorgezogen. Nachfolgend ist dabei immer die Genauigkeit auf den Validierungsdaten gemeint.

Ein weiterer wichtiger Faktor bei der Ausführung eines neuronalen Netzes ist die Anzahl der benötigten Gleitkommaoperationen (*FLOPs, engl. floating point operations*). Im Gegensatz zu der Ausführungszeit handelt es sich hierbei um eine hardwareunabhängige Aussage darüber, wie viele Berechnungsschritte durchgeführt werden müssen.

4.2.2 Architekturen

In diesem Abschnitt werden Informationen über die zur Evaluation verwendeten Datensätze und Architekturen zusammengefasst.

Die Netze sind individuell in Keras mit TensorFlow Backend konstruiert oder nachgebaut worden. Wenn nicht anders angegeben, ist das Training mit Hilfe von *Stochastic Gradient Descent*, einer statischen Lernrate von 0,01 und einer Batchgröße von 128 Trainingsbeispielen durchgeführt worden. Insgesamt wurden jeweils 200 Epochen trainiert. Da das Training bei den unterschiedlichen Architekturen unterschiedlich schnell konvergiert und die Kosten des Netzes wieder steigen können, wurden die trainierten Parameter der Epoche mit der höchsten Testgenauigkeit festgehalten. So ist beispielsweise die höchste Genauigkeit der LeNet-5 Architektur nach der 127. Epoche erreicht worden.

Das initiale Training aller Netze sowie das Retraining wurden auf einer NVIDIA GeForce GTX 1060 Grafikkarte mit 6 GB Speicher durchgeführt.

LeNet Architekturen

Lecun et al. stellen in [Lec+98] erste neuronale Netze mit Convolutional und Pooling Schichten vor. Im Zuge dessen ist der MNIST-Datensatz zur Erkennung von handgeschriebenen Ziffern entstanden. Dieser Datensatz gliedert sich in 60000 Trainings- und 10000 Testdaten. Jedes dieser Beispiele ist ein 28x28 Graustufenbild einer handgeschriebenen Ziffer von ungefähr 250 unterschiedlichen Personen.

Um die Wirksamkeit gegenüber vollverbundenen Schichten zu veranschaulichen, vergleichen Lecun et al. die vorgestellten Netze auch mit Architekturen mit ausschließlich vollverbundenen Schichten. Das LeNet 300-100 besteht neben Ein- und Ausgabeschicht aus einer Schicht mit 300 gefolgt von einer Schicht mit 100 Neuronen mit *tanh*-Aktivierung. Die erzielte Genauigkeit beträgt 88,4%.

LeNet-5 dagegen setzt sich als CNN-Architektur aus vier Convolutional Schichten gefolgt von einer vollverbundenen Schicht zusammen. Da die vierte Convolutional Schicht jedoch

einen 1x1 Kernel mit voller Verbindung aufweist, wird diese nachfolgend ebenfalls als vollverbundene Schicht betrachtet. Nach der ersten und dritten Schicht werden mit Hilfe von Pooling Schichten die Feature Maps halbiert. Abbildung 4.1 zeigt den Gesamtaufbau.

Die Keras-Adaption erreicht nach dem Training von LeNet-5 eine Genauigkeit von 99% auf den Testdaten. Zwei geringe Unterschiede zu der Orginalarchitektur aus Abbildung 4.1 sind allerdings vorhanden. Nach [Lec+98] sind die Filter der 3. und 4. Schicht nicht alle miteinander verbunden, sondern verfolgen ein spezifisches Schema. Da dies in moderneren Architekturen wie in [KSH12] oder [SZ14] unüblich ist und auch in Keras nicht nativ unterstützt wird, enthält der Nachbau in dieser Arbeit diese Restriktion nicht. Dies erhöht insgesamt die Anzahl an trainierbaren Parametern von 48540 auf 60360. Aus gleichen Gründen besitzt der Nachbau als Kostenfunktion die kategorische Kreuzentropie mit einer Softmax-Aktivierung in der Ausgabeschicht.

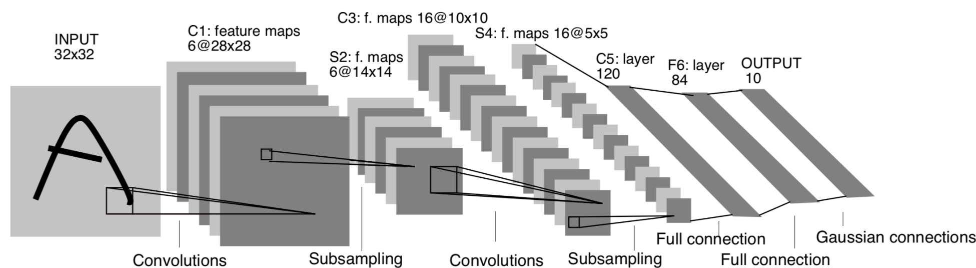


Abbildung 4.1: Architektur von LeNet5 aus [Lec+98]

UCI Smartphone Architektur

Wie bereits in Abschnitt 2.1 beschrieben, gibt es neben der Bildklassifizierung noch viele weitere Anwendungsgebiete von neuronalen Netzen. Um die Wirksamkeit der Techniken auch in einem anderen Kontext zu zeigen, ist aus diesem Grund ein neuronales Netz zur Erkennung von menschlichen Bewegungen untersucht worden.

Trainiert wurde dieses auf dem UCI Smartphone Datensatz [Ang+13]. Darin enthalten sind Bewegungen von 30 Personen, die mit Hilfe von Smartphone Sensordaten aufgezeichnet wurden. Die gemessenen Accelerometer und Gyroskopdaten sind von den Autoren bereits für eine Klassifizierung in Fenster der Größe 2,56s eingeteilt und annotiert worden. Tabelle 4.1 zeigt die verschiedenen Bewegungsarten und die Anzahl der Beispiele.

Motiviert durch verschiedene Anwendungen von neuronalen Netzen für Zeitserien [Faw+18], ist ein 1D-CNN entworfen worden. Dies trägt im Folgenden den Namen *UCI Custom*. Statt den in [Ang+13] ausführlich erarbeiteten Merkmalsvektoren, klassifi-

Bewegungsart	**#Training**	**#Test**
Laufen	1226	496
Treppensteigen abwärts	1073	471
Treppensteigen aufwärts	986	420
Sitzen	1286	491
Stehen	1374	532
Liegen	1407	537
Gesamt	7352	2947

Tabelle 4.1: Die Klassen des UCI Datensatzes sowie die Anzahl der jeweiligen Trainings- und Testbeispiele

ziert es die Sensordaten des Accelerometers direkt. Vergleichbar mit den Kanälen eines Eingabebildes, bilden die einzelnen Zeitreihen der x-, y-, und z-Werte des Accelerometers die Dimensionen der Eingabe. Abbildung 4.2 zeigt den Aufbau des Netzes. Mit Hilfe von Batch Normalization [IS15] vor jeder ReLU-Aktivierung konnte die Testgenauigkeit um fast 10% auf 86,43% erhöht werden.

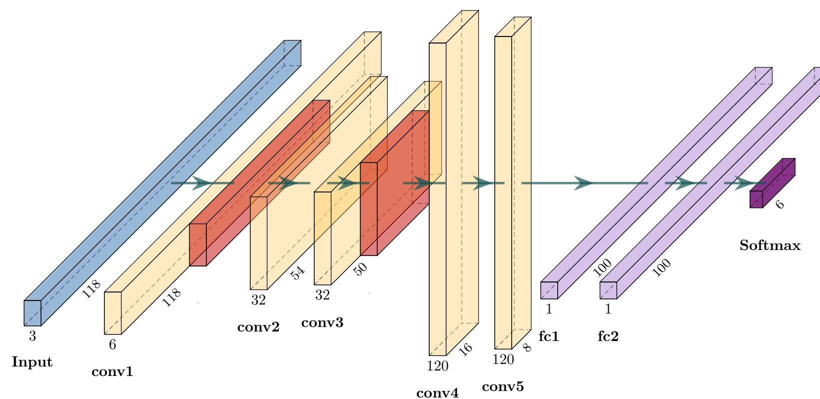


Abbildung 4.2: UCI Custom Architektur mit fünf 1D-Convolutional Schichten(gelb), zwei Max-Pooling Schichten(rot) und zwei vollverbundenen Schichten(violett)

Schließlich konnte die Genauigkeit durch simple Vorverarbeitungsschritte noch auf 90,48% gesteigert werden. Dabei wurde zunächst zur Entfernung von hohen Frequenzen ein gleitender Mittelwert-Filter der Breite 10 angewendet. Anschließend wurde der Mittelwert und die Standardabweichung dimensionsweise für den Trainingsdatensatz berechnet. Diese Werte wurden in einem letzten Schritt zur Normalisierung aller Trainings- und Testbeispielen verwendet. Abbildung 4.3 visualisiert einzelne Trainingsbeispiele und demonstriert die Vorverarbeitung exemplarisch.

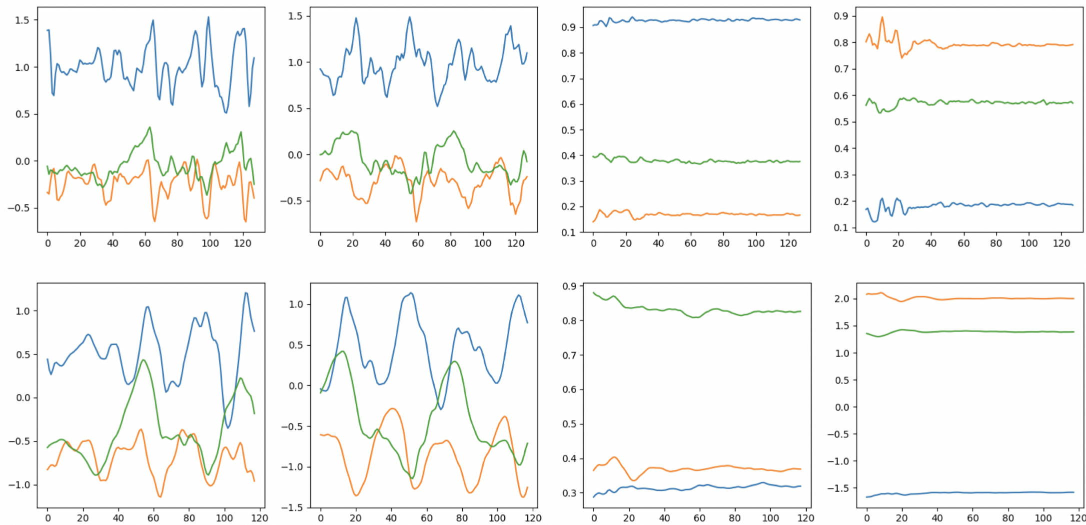


Abbildung 4.3: Accelerometer-Trainingsbeispiele aus dem UCI Smartphone Dataset (oben) mit durchgeführter Vorverarbeitung (unten)

Transfer Learning VGG16-Architektur

Aufgrund mangelnder Trainingsdaten ist Transfer Learning in der Praxis ein verbreitetes Konzept [Tan+18]. Dabei geht es darum, Merkmale aus einem großen Datensatz durch Training zu extrahieren und diese Information zur Klassifizierung eines kleineren Datensatzes zu nutzen. Tan et al. haben mehrere Varianten in [Tan+18] erläutert. Eine dieser Varianten sieht vor, ein Netzwerk zu trainieren und Teile des trainierten Netzes für einen anderen Datensatz zu übernehmen. Oftmals sind dies die trainierten Convolutional Schichten, die Strukturen und Merkmale in Eingabebildern identifizieren. Die eigentliche Klassifizierung der vollverbundenen Schichten wird nicht übernommen.

Um Pruning auch in diesem Kontext zu untersuchen, wird eine VGG16-Architektur zur Klassifizierung des *Stanford-Dogs-Dataset* betrachtet. Dies ist ein Datensatz mit 20000 Hundebildern aus 120 Hunderassen. Die Parameter der Convolutional Schichten stammen dabei aus der öffentlich zugänglichen Orginalnetzwerk aus [SZ14], welches auf dem *ImageNet-Dataset* [Rus+14] trainiert wurde. Lediglich die vollverbundenen Schichten sind zufällig initialisiert und auf dem Hundedatensatz trainiert worden. Deshalb werden auch nur diese in den nachfolgenden Evaluationen betrachtet.

In der Architektur aus [SZ14] haben die beiden vollverbundenen Schichten 4096 Neuronen. Angelehnt an [Yos+14] wurde eine geringere Anzahl an Neuronen für das Transfer Learning getestet. Tabelle zeigt die Eigenschaften der vollverbundenen Schichten und die Gesamtgenauigkeit für 4096, 2048 und 1024 Neuronen in den vollverbundenen Schichten.

#Neuronen	#Parameter	Genauigkeit
4096	123,43M	68,83%
2048	60,56M	69,43%
1024	26,73M	69,73%

Tabelle 4.2: Eigenschaften der untersuchten VGG16 Architekturen

4.3 Pruning von Verbindungen

Die ersten untersuchten Pruningmethoden umfassen das Entfernen von Verbindungen. In Abschnitt 3.2.1 sind dazu bereits mehrere Strategien vorgestellt worden. Die nachfolgenden Experimente beschränken sich dabei auf folgende Strategien:

- **Wertebasierte Auswahl (Wertebasiert):** Die Verbindungen mit den betragsmässig niedrigsten Werten werden ausgewählt.
- **Optimal Brain Damage (OBD)**: Alle Parameter werden nach der Veränderung der Kostenfunktion mit Hilfe der Hessian-Matrix beurteilt. Anstatt die gesamte Hessian-Matrix auszuwerten, werden wie in [LDS90] empfohlen nur die Diagonalelemente berechnet.
- **Taylorapproximation 1. Ordnung (Taylor)**: Molchanov et al. nutzten in [Mol+16] eine Taylorapproximation für die Auswahl von Filtern innerhalb von CNNs. Eine Übertragung auf einzelne Verbindungen w_i ist dabei jedoch genauso umsetzbar (Formel 4.3). Verbindungen mit niedriger Bewertung s_i werden verworfen.

$$s_i = \left| \frac{\partial C}{\partial w_i} w_i \right| \tag{4.3}$$

- **Zufällige Auswahl (Random):** Die abgeschnittenen Parameter werden zufällig ausgewählt.

Vollverbundene Schichten: Abbildung 4.4 zeigt die Anwendung der verschiedenen Verfahren auf die vollverbundenen Schichten der LeNet-5 Architektur. Schrittweise wurde mit den aufgeführten Methoden die jeweilige Schicht reduziert. Die dargestellten Werte des Graphen sind auf einen Prozent der Parameter genau.

Bei der Betrachtung der größten Schicht erzielen alle Verfahren problemlos eine Reduktionsrate über den Fator 10 hinaus. Selbst das zufällige Abschneiden von Verbindungen ist hierbei eine valide Option, um einen Großteil der Gewichte zu entfernen. Dies zeigt, dass nur ein sehr geringer Anteil der Verbindungen nötig ist, um die Ausgabe einer Schicht

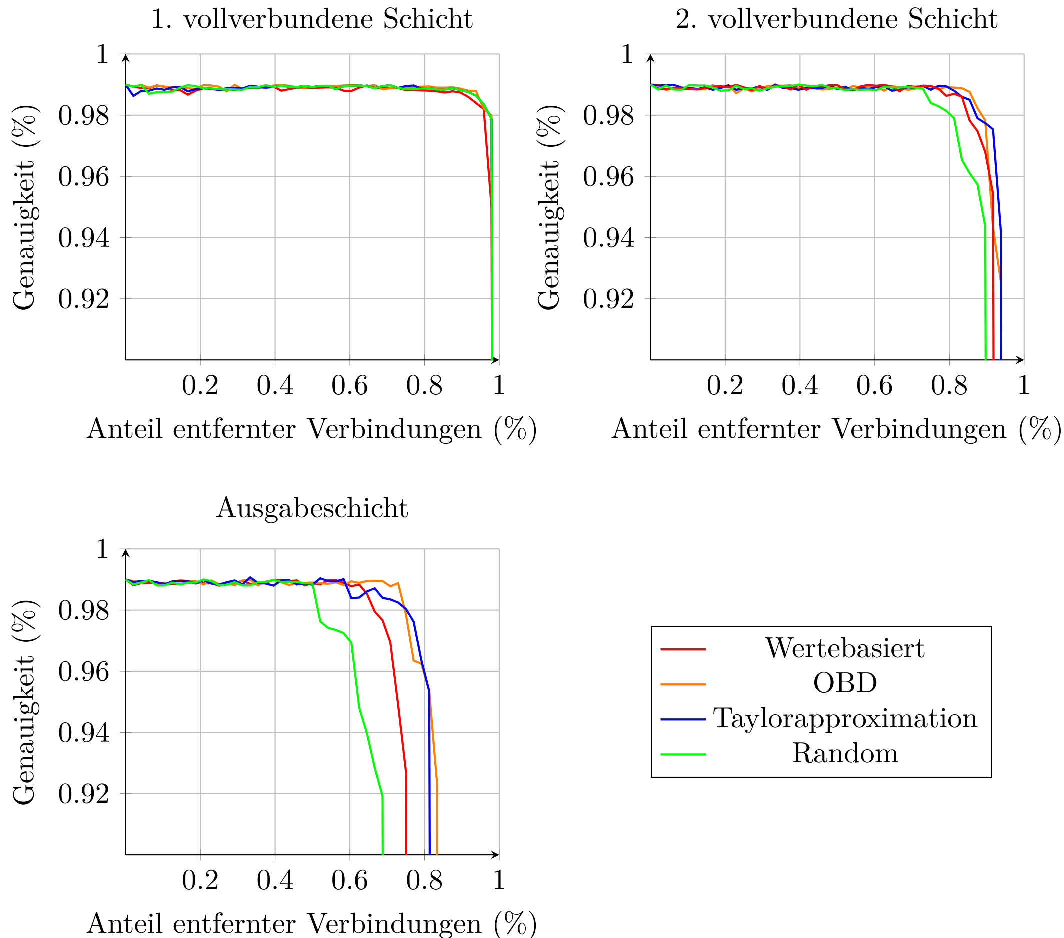


Abbildung 4.4: Pruning von Verbindungen der vollverbundenen Schichten der LeNet-5 Architektur

zu produzieren. Die übrigen trainierten Verbindungen propagieren also ausschließlich redundante Informationen durch das neuronales Netz. Weitere Evaluationen auf LeNet 300-100, dem eigenen UCI-Custom-Netz und der angepassten VGG16-Architektur bestärken diese Hypothese. Tabelle 4.3 zeigt Pruningergebnisse auf diesen Architekturen. Die dafür benötigten Experimenten beziehen sich auf das Optimierungsproblem aus Formel 4.2. Als Grenzwert des Genauigkeitsverlustes wurde $\varepsilon = 1\%$ gewählt. Für einen verbesserten Vergleich sind die darin befindliche Werte prozentuale Reduktionsraten. Alle Angaben sind auf 0,1% der Gesamtparameter einer Schicht genau. Die besten Werte sind jeweils fett markiert.

Es wird deutlich, dass Schichten mit wenigen Parametern im Vergleich sensitiver gegenüber Pruningoperationen sind. Daraus folgt der Vorteil, dass bei der Reduktion von allen vollverbundenen Schichten eines Netzes, Schichten mit vielen Parametern bevorzugt und zeitgleich auch hohe Reduktionsraten erzielt werden können.

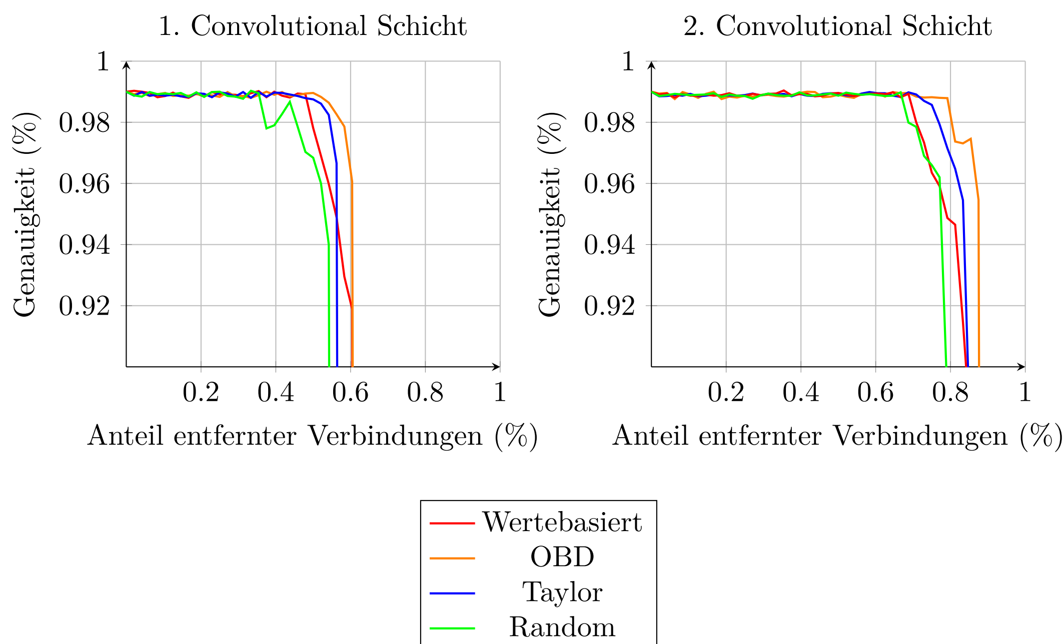


Abbildung 4.5: Pruning der Convolutional Schichten der LeNet-5 Architektur

Convolutional Schichten: Abbildung 4.5 zeigt das Abschneiden der Convolutional Schichten der LeNet-5 Architektur und Tabelle 4.3 die Ergebnisse aller untersuchten Netze.

Auch wenn die einzelnen Reduktionsraten niedriger ausfallen, sind die Unterschiede gegenüber den vollverbundenen Schichten gering. Auf Schichten mit vielen Parametern sind prozentual und absolut in der Anzahl der Parameter bessere Ergebnisse erzielt worden. So ist es zum Erreichen von hohen Reduktionsraten ratsam, sich auf Schichten mit vielen Parametern zu konzentrieren.

Vergleichbar mit den vollverbundenen Schicht zeigen die Experimente generell, dass mit geringem Genauigkeitsverlust jedes Verfahren ein großen Teil der Parameter entfernen kann. Dennoch werden die Unterschiede zwischen den einzelnen Strategien deutlicher. Das zufällige, aber auch das wertebasierte Pruning schnitt bei nahezu allen Schichten schlechter als die anderen beiden Methoden ab. Besonders hervorzuheben ist, dass die Variation der Taylorapproximation aus [Mol+16] in mehreren Architekturen und Schichten die besten Ergebnisse lieferte.

Retraining

Arbeiten wie [HMD15], [Han+15] oder [DCP17] zeigen, dass durch erneutes Training mit wenigen Iterationen ein durch Pruning entstandener Fehler kompensiert werden kann. Folglich ist es naheliegend, auch alle Strategien mit erneutem Training zu vergleichen.

		#Parameter	Taylor	Wertebasiert	OBD	Zufällig
LeNet-5	conv1	156	57,3	50,0	**58,8**	36,9
	conv2	2,416k	68,4	70,1	**80,0**	69,1
	dense1	48,120k	96,6	**96,7**	**96,7**	96,6
	dense2	10,184k	89,3	**89,9**	89,0	88,2
	out	850	**76,2**	66,7	76,1	48,6
LeNet 300-100	dense1	235,500k	98,0	97,9	**98,1**	97,1
	dense2	30,100k	93,6	**93,7**	93,6	93,5
	out	1,010k	73,4	73,2	**74,9**	67,6
UCI-Custom	conv1	96	37,1	33,1	**29,9**	29,1
	conv2	992	61,6	61,4	**62,1**	56,1
	conv3	5,152k	**73,1**	71,7	71,7	66,6
	conv4	34,680k	**77,7**	73,8	77,2	71,5
	conv5	129,720k	**79,2**	76,6	78,8	77,6
	dense1	96,100k	**94,7**	94,5	94,5	94,2
	dense2	10,100k	90,1	89,6	**90,3**	88,9
	out	606	**70,4**	79,8	78,7	79,2
VGG16	dense1	102,765M	**92,5**	**92,4**	92,1	91,6
	dense2	16,781M	**90,9**	90,7	90,6	90,6
	out	491,520k	**85,3**	84,5	84,4	80,8

Tabelle 4.3: Pruning von Verbindungen auf unterschiedlichen Architekturen

Die Tabelle 4.4 umfasst Evaluationen der LeNet-5 Architektur mit erneutem Training. Dazu wurde nach jedem entfernten Prozent an Parametern das gesamte CNN 5 Epochen trainiert.

Auffällig ist, dass sich die Reduktionsraten der verschiedenen Strategien angleichen. Nach umfangreichem Retraining ist es also weniger relevant, ein bestimmtes Verfahren auszuwählen. Eine mögliche Erklärung dafür ist, dass sich alle Verfahren dem optimalen Parametersatz λ' aus Formel 4.1 annähern.

		#Parameter	Taylor	Wertebasiert	OBD	Zufällig
LeNet-5	conv1	156	**77,8**	77,7	**77,8**	77,6
	conv2	2,416k	**98,0**	97,9	**98,0**	97,9
	dense1	48,120k	**99,6**	99,5	99,5	99,2
	dense2	10,184k	99,1	99,3	**99.4**	99.1
	out	850	**94,9**	94,7	94,5	94,4

Tabelle 4.4: Pruning von Verbindungen mit erneutem Training

4.4 Pruning von Neuronen / Filtern

Wie bereits in Abschnitt 4.4 beschrieben, hat das Entfernen von Neuronen oder Filtern in der Praxis entscheidende Vorteile gegenüber dem Pruning von Verbindungen. Ein wichtiger Faktor ist, dass neben der Parameterreduktion eine Reduktion der Gleitkommaoperationen einhergeht. Deshalb enthalten alle nachfolgenden Evaluationen ebenfalls die erreichte FLOP-Reduktionrate. Folgende Strategien wurden evaluiert:

- **Wertebasierte Auswahl:** Für jedes Neuron mit Gewichtsvektor w_i wird die L_1-Norm $\|\mathbf{w}_i\|_1$ als Bewertung berechnet. Neuronen mit niedriger Bewertung werden entfernt. Da die Anordnung der Gewichte dabei unerheblich ist, kann durch Vektorisierung der Gewichtstensoren auch von Convolutional Schichten die L_1-Norm bestimmt werden. Die Technik ist damit äquivalent zu dem Verfahren aus [Li+16] und wird nachfolgend als Variante 1 betitelt.

  Aus der zugrundeliegenden Idee lässt sich noch ein weiteres Verfahren ableiten, was in dieser Form neu ist. Statt Parameter eines Neurons i in die Analyse miteinzubeziehen, können auch die Gewichtungen des Neurons in der kommenden Schicht genutzt werden (Variante 2).

  $$s_i^l = \sum_{j=0}^{\left|h^{l+1}\right|} \left|\mathbf{W}_{ij}^{l+1}\right| \tag{4.4}$$

  $\left|h^{l+1}\right|$ beschreibt die Anzahl der Neuronen und $\mathbf{W}^{l+1}$ ist die Gewichtsmatrix der Schicht $l+1$. $\mathbf{W}_{ij}^{l+1}$ ist also die Gewichtung des Neurons i im j-ten Neuron der folgenden Schicht. Damit wählt die Bewertung Neuronen aus, die von allen Neuronen der kommenden Schicht gering gewichtet werden. Bei Filtern ist

die Gewichtung $\mathbf{W}_{ij}^{l+1}$ ein mehrdimensionaler Kernel, der auch hier durch eine vektorisierte L_1-Norm zu einem Wert reduziert werden kann.

Weiterhin können die beiden Varianten durch Aufsummierung kombiniert werden, sodass sowohl die eigenen Parameter als auch die Gewichtungen der folgenden Schicht in die Bewertung einfließen (Variante 3).

- **Zufällige Auswahl:** Die abgeschnittenen Neuronen / Filter werden zufällig ausgewählt.
- **Aktivierung**: Neuronen oder Filter mit niedriger (aufsummierter) Aktivierung werden reduziert.
- **Taylorapproximation**: Dieses Verfahren stammt vollständig aus [Mol+16] und wurde ebenfalls in Abschnitt 4.4 näher beschrieben. Es benötigt zur Berechnung der Gradienten wie im Backpropagation-Algorithmus auch annotierte Trainingsbeispiele $(\mathbf{X}, \mathbf{Y})$.
- **Ähnlichkeit**: In Abschnitt 4.4 wurde die Technik aus [SB15] bereits erläutert. Ein wichtiger Unterschied zu allen anderen Verfahren ist, dass hierbei Neuronen nicht einzeln bewertet und entfernt, sondern jeweils zwei Neuronen zusammengefasst werden. Srinivas et al. wenden es nur auf Neuronen in vollverbundenen Schichten an. Das verwendete Distanzmaß lässt sich in gleicher Weise jedoch auch auf Kernel von Convolutional Filtern übertragen.

Auswertung

Um die Unterschiede zwischen einzelnen Schichten in einem etwas größeren Netz zu visualisieren, werden die Ergebnisse auf der UCI-Custom Architektur grafisch illustriert. Abbildung 4.6 zeigt die Reduktion der größten Convolutional Schicht im Vergleich. Zur besseren Darstellung wurden die wertebasierten Verfahren in den rechten Teil der Abbildung ausgegliedert. Für das Experiment wurden die 120 Neuronen der Schicht einzeln mit dem jeweiligen Verfahren entfernt. Tabelle 4.5 enthält Evaluationen auf weiteren Architekturen. Kongruent zu dem Pruning von Verbindungen beziehen sich die Untersuchungen auf das Optimierungsproblem aus 4.2 mit $\varepsilon = 1\%$. Die Tabellenwerte beschreiben den prozentualen Anteil der geprunten Parameter oder Neuronen, die zum Einhalten von ε möglich waren.

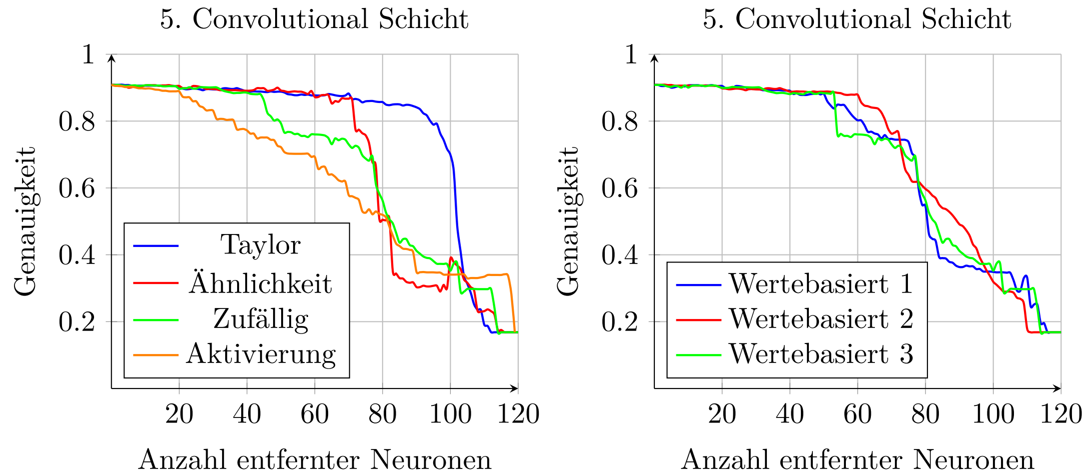


Abbildung 4.6: Pruning der größten Convolutional Schicht der LeNet-5 Architektur

Um die Sensitivität der unterschiedlichen Schichten eines CNNs zu zeigen, wurde in Abbildung 4.7 ein Pruningexperiment mit der Taylorapproximation auch auf den übrigen Schichten der UCI-Custom Architektur visualisiert. Die Abbildung bekräftigt die Hypothese, dass Schichten mit vielen Neuronen höhere Reduktionsraten ermöglichen. Dennoch sind große Teile der Graphen in der Praxis wenig relevant. Ein entscheidendes Ziel des Prunings ist es, die Genauigkeit zu erhalten. Ohne erneutes Training sind deshalb ausschließlich die Teile des Graphen von Bedeutung, die zu einem akzeptablen Genauigkeitsverlust führen.

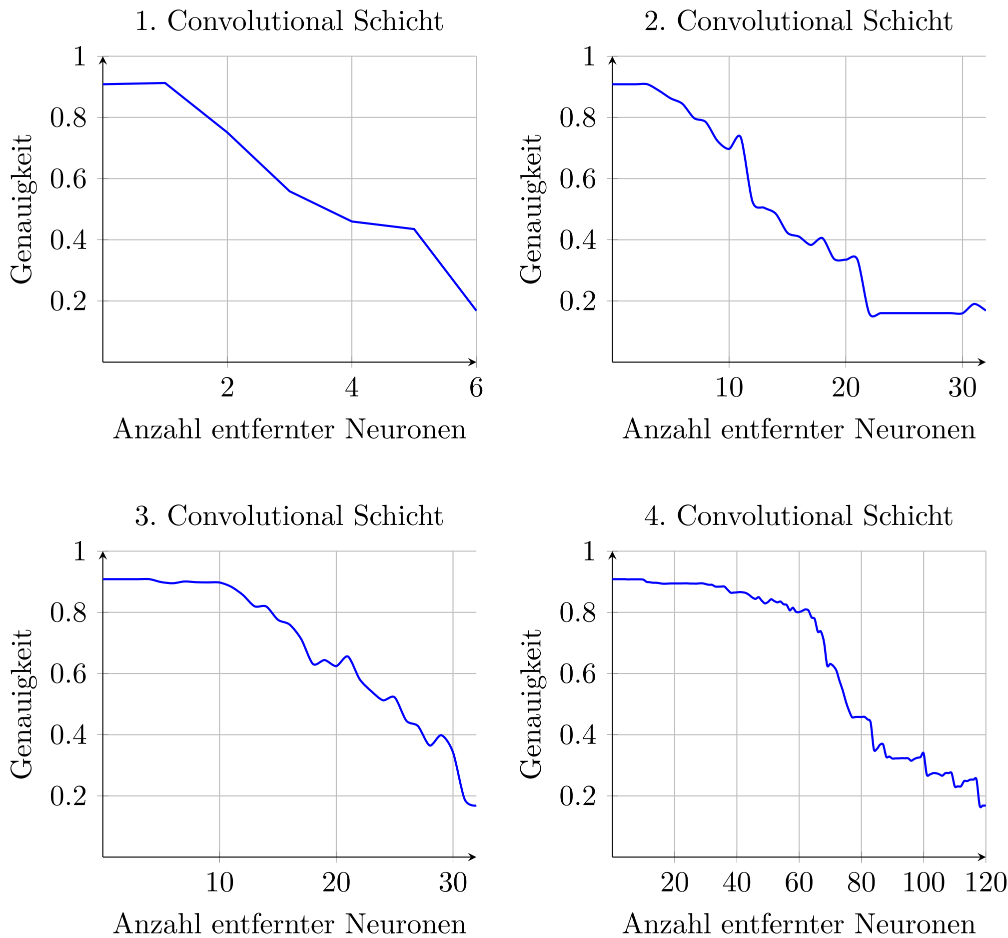


Abbildung 4.7: Taylorpruning der Convolutional Schichten der LeNet-5 Architektur

Retraining

Die Arbeit von [SB15] zeigt, dass mit geeigneten Strategien ein neuronales Netz auch ohne Training auf weniger als ein fünftel der Gesamtgröße komprimiert werden kann. Dennoch trainieren alle anderen in Kapitel 3 vorgestellten Arbeiten ein reduziertes Netz erneut, um den entstandenen Fehler zu kompensieren.

Deshalb wird auch hier erneutes Training zur Fehlerkompensation evaluiert. Dabei wird nach jedem abgeschnittenen Neuron jeweils 5 Epochen neu trainiert (Tabelle 4.4). Das derartig durchgeführte Retraining hat in der Praxis allerdings einen entscheidenden Nachteil. Durch die kleinen Pruningsschritte werden viele Trainingsiterationen oder Epochen benötigt, was bei großen Netzen zu erheblichem Ressourcenverbrauch führt. Zusätzlich müssen auch die benötigten Datenstrukturen des Pruningverfahrens wie die Hessian-Matrix bei OBD häufig neu berechnet werden. Dies ist der Grund gewesen, dass bei der untersuchten VGG16-Architektur nicht nach jedem einzelnen abgeschnittenen

		#Parameter	Taylorapproximation	Ähnlichkeit	Aktivierung	Zufällig	Wertebasiert 1	Wertebasiert 2	Wertebasiert 3
LeNet-5	conv1	156	**16,67**	**16,67**	0,00	0,00	**16,67**	**16,67**	**16,67**
	conv2	2,416k	**12,50**	**12,50**	**12,50**	**12,50**	**12,50**	**12,50**	**12,50**
	dense1	48,120k	**23,33**	21,67	17,50	15,83	18,33	16,67	20,83
	dense2	10,184k	**20,24**	19,05	19,05	16,67	19,05	17,86	19,05
	out	850	-	-	-	-	-	-	-
LeNet 300-100	dense1	235,500k	49,00	46,33	45,00	37,00	47,67	46,67	**49,33**
	dense2	30,100k	**35,00**	34,00	29,00	30,00	32,00	32,00	33,00
	out	1010	-	-	-	-	-	-	-
UCI-Custom	conv1	96	**16,67**	**16,67**	**16,67**	0,00	**16,67**	**16,67**	**16,67**
	conv2	992	**15,63**	**15,63**	12,50	12,50	**15,63**	15,63	15,63
	conv3	5,152k	**18,75**	**18,75**	12,50	12,50	15,63	12,50	15,63
	conv4	34,680k	14,17	13,33	13,33	12,50	**15,00**	14,17	14,17
	conv5	129,720k	**22,50**	**22,50**	21,67	19,17	20,83	21,67	**22,50**
	dense1	96,100k	**33,00**	30,00	28,00	28,00	30,00	29,00	31,00
	dense2	10,100k	**28,00**	27,00	27,00	22,00	24,00	25,00	25,00
	out	606	-	-	-	-	-	-	-
VGG16	dense1	102,765M	33,74	**34,18**	31,20	29,30	32,01	33,18	32,96
	dense2	16,781M	19,14	**19,56**	17,07	17,58	17,82	17,33	19,07
	out	491,520k	-	-	-	-	-	-	-

Tabelle 4.5: Resultate von SNARE auf unterschiedlichen Architekturen

Neuron trainiert wurde, sondern erst nach $N = 10$ abgeschnittenenen Neuronen. Ansonsten wären bei 4096 Neuronen in der ersten vollverbundenen Schicht $4095 * 5 = 20475$ Epochen zu trainieren, welches mehr als dem 100-fachen des ursprünglichen Trainings entspricht. Um schnellere Pruningergebnisse zu erhalten, kann N beliebig vergrößert werden. Durch einen zu hohen Wert kann die ursprüngliche Genauigkeit allerdings nicht wiederhergestellt werden. Die richtige Auswahl eines guten Wertes ist ein offenes Forschungsproblem. Für das wertebasierte Prunen gibt es mit der Schwellwertsuche aus [Li+18] bereits einen Ansatz. Wie SNARE mit diesem Problem umgeht, wird im kommenden Kapitel erläutert.

Die Resultate in Tabelle 4.4 legen nahe, dass es nach ausreichendem Training keinen bedeutenden Unterschied zwischen den einzelnen Verfahren gibt. Die übrigen Schichten sind selbst bei hohen Reduktionsraten in der Lage, den Fehler auszugleichen. Eine

gewisse Mindestanzahl an Neuronen ist jedoch stets erforderlich, um die Informationen durch das Netz zu propagieren. Die betrachtete Schicht kann dadurch nicht weiter reduziert werden. Für eine weitere Komprimierung des Netzes reicht es also nicht aus, nur eine einzelne Schicht zu reduzieren.

		#Parameter	Taylorapproximation	Ähnlichkeit	Aktivierung	Zufällig	Wertebasiert 1	Wertebasiert 2	Wertebasiert 3
LeNet-5	conv1	156	**66,67**	**66,67**	**66,67**	**66,67**	**66,67**	**66,67**	**66,67**
	conv2	2,416k	**75,00**	**75,00**	68,75	**75,00**	**75,00**	**75,00**	**75,00**
	dense1	48,120k	**97,50**	95,83	95,00	92,50	95,83	96,67	95,83
	dense2	10,184k	**95,24**	**95,24**	92,86	91,67	94,05	92,86	94,05
	out	850	-	-	-	-	-	-	-

Tabelle 4.6: Pruning von Neuronen mit erneutem Training

5 SNARE

Bisher ist ausschließlich das Entfernen von Neuronen oder Filter eines neuronalen Netzes innerhalb einzelner Schichten erläutert worden. Ziel dieses Kapitel ist es, beliebige Strategien zur Entfernung von Neuronen auf gesamte Netzwerke zu übertragen. Im Zuge dessen wird der Ansatz SNARE vorgestellt, welcher in einem iterativen Prozess Netzwerk analysiert und schichtweise reduziert.

Im Kontext dieser Arbeit ist eine Implementierung von SNARE für Keras-Netze mit TensorFlow Backend enstanden. Es ist öffentlich verfügbar und kann ohne Nutzerinteraktion ein sequenzielles CNN automatisch reduzieren. Die damit einhergehenden Probleme und Lösungswege werden ebenfalls in diesem Kapitel beschrieben.

5.1 Aufbau

Die grundlegenden Komponenten und der Ablauf von SNARE (Score-based Neural Architecture REduction) sind in Abbildung 5.1 illustriert. Als iteratives Verfahren reduziert es in jeder Phase ein neuronales Netz $\mathcal{N}$ und produziert ein verkleinertes valides neuronales Netz $\mathcal{N}'$. Es handelt sich dabei weiterhin um das Optimierungsproblem aus Formel 4.2, bei dem der Nutzer selbst den akzeptierten Genauigkeitsverlust ε angeben kann.

Startpunkt jeder Iteration ist die Schichtanalyse. Dabei wird für jede Schicht des Eingabenetzes ein Score berechnet und Pruningoperationen definiert. Mithilfe dieses Scores werden alle Operationen bewertet und ausgewählt. Eine solche Operation gibt dabei die Anzahl der zu entfernenden Neuronen an. Anschließend wird das Pruning von Neuronen mit einem aus Abschnitt 4.4 bekanntem Verfahren schichtweise angewendet. Der dadurch entstandene Fehler wird durch erneutes Trainieren kompensiert. Der letzte Schritt einer Iteration ist die Evaluation des durchgeführten Prunings. Falls das resultierende Netz die gewünschte Genauigkeitsgrenze unterschreitet, wird der Iterationsschritt verworfen und die Operationsgröße reduziert.

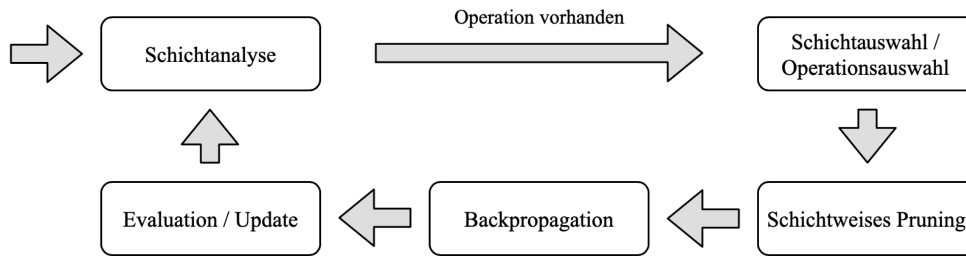


Abbildung 5.1: Vereinfachter Ablauf einer Iteration von SNARE

5.2 Identifikation von relevanten Schichten

Der erste Teil der Schichtanalyse von SNARE besteht in der Auswahl aller zu reduzierenden Schichten. Eine Möglichkeit ist, dies manuell durchzuführen. In Arbeiten wie [Mol+16], [Han+15], [SB15], [Li+16] wählen die Autoren für jede Architektur alle untersuchten Schichten individuell aus. Die Auswahl kann dabei auf eine Schicht wie in [SB15] oder aber Gruppen wie in [Mol+16] festgelegt werden. Eine dritte Möglichkeit ist es, zunächst alle unterstützten Arten von Schichten in einer Architektur zu betrachten. In objektorientierten Bibliotheken wie Keras ist dies einfach zu realisieren, da alle Schichten einer bestimmten Klasse zugehören. Bestimmte Arten von Schichten wie Dropout oder Pooling-Schichten, die nicht direkt reduziert werden können, sind dadurch leicht herauszufiltern. Alle drei Arten werden in der SNARE-Implementierung unterstützt.

5.2.1 Vorteile der Betrachtung ganzer Netzwerke

Srinivas et al. wenden ihr Verfahren zur Reduktion von Neuronen lediglich auf die erste vollverbundene Schicht der untersuchten Netze an. Die Autoren begründen ihre Entscheidung damit, dass einzelne Schichten in Architekturen wie VGG16 oder LeNet-5 einen Großteil der Parameter enthalten. Bei der VGG16-Architektur enthält diese Schicht beispielsweise ca 80% aller Parameter. Trotzdem gibt es unterschiedliche Argumentationen, die eine Reduktion ganzer Netwerke begründen.

Ein naheliegender Vorteil ist, dass alle Neuronen/Filter und auch Parameter untersucht und gegebenenfalls reduziert werden können. Zudem variiert die Schicht mit den meisten Parametern zwischen unterschiedlichen Architekturen. So beträgt der Anteil dieser Schicht in der LeNet-5-Architektur 65% und in noch tieferen Netzen wie GoogleNet nur noch 10% der Gesamtparameter. Durch diese Konstanten ist somit bereits im Vorfeld eine maximale mögliche Reduktionsrate des Gesamtnetzes definiert. Bei manchen Netzwerken liegen diese Grenzen weit unter den Werten, die ein Ansatz wie SNARE erreichen kann (Abschnitt 5.4.2).

```
function SNARE(𝒩, X, Y, ε)
    layers ← [l ∈ 𝒩 | prunable(l, 𝒩)]
    scores ← {}
    op_sizes ← {}

    for all layer ∈ layers do
        scores[layer] ← calculate_score(layer, 𝒩, β, γ)
        op_sizes[layer] ← 64%

    error ← eval(𝒩′, X, Y)
    while scores ≠ ∅ do
        best ← {layer | layer = arg min_{l′} scores[l′] ∗ op_sizes[l′]}
        p ← op_sizes[best]

        𝒩′ ← apply_layer_pruning(𝒩, best, p)
        𝒩′ ← retrain(𝒩′)
        e ← eval(𝒩′, X, Y)

        if e − error < ε then
            𝒩 ← 𝒩′
            scores ← update_scores(𝒩, β, γ)
            error ← min(e, error)
        else
            p ← p/2
            op_sizes[best] ← p
            if p = 4% then
                scores ← scores \ scores[best]
    return 𝒩
```

Abbildung 5.2: Pseudocode-Implementierung von SNARE

Weiterhin hängt die Anzahl der Parameter einer einzelnen Schicht auch von der Größe der Eingabe ab. Diese wiederum entspricht in einem sequenziellen Feedforward-Netz der Ausgabegröße der vorhergehenden Schicht. Eine Reduktion von Neuronen innerhalb dieser Schicht führt also in gleichem Umfang zu einer Reduktion der ursprünglich betrachteten Schicht. Als Alternative zur Reduzierung einer bestimmten Schicht kann deshalb auch immer die vorhergehende Schicht reduziert werden. Die gewünschte Reduktionsrate auf beide Schichten aufzuteilen, ist ebenfalls möglich.

Im Pruningvorgang zeigt sich noch ein weiterer Effekt. Die Genauigkeit kann sich durch gleichmässiges Pruning mit erneutem Training sogar verbessern. So zeigt Tabelle 4.2 Ergebnisse aus einem Experiment auf der VGG16-Architektur aus Abschnitt 4.2.2. Es handelt sich dabei um das Netz mit 2048 Neuronen in den vollverbundenen Schichten mit einer Genauigkeit von 69,43%. Durch wertebasiertes Pruning wurde die größte vollverbundene Schicht auf die in Spalte 1 angegebene Anzahl Neuronen in einem Schritt

reduziert. Nach diesem 1. Pruningvorgang wurden 10 Epochen neu trainiert. Spalte 2 ist die erzielte Genauigkeit des reduzierten Netzes nach dem Training. In einem 2. Pruningvorgang wurden 2048 Neuronen der zweiten vollverbundenen Schicht entfernt und weitere 10 Epochen trainiert (Spalte 4). Auffällig ist, dass sich die Genauigkeit nach der 2. Reduktion in allen Fällen erhöht hat. Auch ein längeres Trainieren mit 20 Epochen nach dem ersten Pruningvorgang (Spalte 3) führt nicht zu besseren Ergebnissen.

#Neuronen	1. Pruningvorgang		2. Pruningvorgang
	10 Epochen	20 Epochen	10 Epochen
2048	-	-	-
1536	69,73%	69,85%	69,87%
1024	69,86%	69,99%	70,10%
512	69,75%	69,79%	70,06%

Tabelle 5.1: Pruning von zwei Schichten gegenüber Pruning einer einzelnen Schicht

5.3 Bewertung von Schichten

Beim Entwurf eines iterativen Verfahrens wie SNARE ist besonders die Auswahl der zu prunenden Schichten entscheidend. Wiederholt muss eine Schicht l ausgewählt und reduziert werden, die zu einem möglichst geringen Genauigkeitsverlust führt. Bei fester Operationsgröße p formalisiert Formel 4.1 das Optimierungsproblem. $f(a,b)$ ist dabei eine schichtweise Pruningoperation auf a der Größe b und liefert ein reduzierten Parametersatz λ' des Gesamtnetzes zurück.

$$l = \arg\min_{l'} C(\mathbf{X}, \mathbf{Y}|f(l', p)) \tag{5.1}$$

Kongruent zu dem Optimierungsproblem aus Formel 4.1 ist eine erschöpfende Suche bei großen Netzen technisch aufwendig oder nicht umsetzbar. Es gilt also eine Heuristik zu finden, die Schichten bewertet und eine möglichst gute Annäherung liefert. Problem ist jedoch, dass sich Schichten innerhalb eines neuronalen Netzes stark unterscheiden können. Die Architekturen aus [SZ14] oder [KSH12] sind nur wenige Beispiele, in denen die Anzahl der Parameter zwischen den Schichten stark variiert. Basierend auf Designentscheidungen und des damit einhergehenden Datenfluss des Netzes ergeben sich auch Unterschiede in der Ein- und Ausgabegröße sowie der Anzahl der benötigten Gleitkommaoperationen in der Ausführung. In Abschnitt 4.4 wurde gezeigt, dass die Schichten auch unterschiedlich sensitiv gegenüber Pruningoperationen sind.

Die in SNARE eingesetzte Bewertung basiert auf einem neuartigen Bewertungskriterium, dem Pruningpotenzial. Dieses ist definiert als die Reduktionsrate, die durch ein Pruningverfahren f auf einer Schicht erzielt werden kann und gleichzeitig ein erlaubter Genauigkeitsverlustes ε berücksichtigt wird. (Formel 5.2). Die Idee dahinter ist, dass durch die Auswahl einer Schicht mit hohem Pruningpotenzial in jeder Iteration eine Schicht reduziert wird, die geringe Auswirkungen auf die Genauigkeit des Gesamtnetzwerkes hat. Aus den Experimenten des vergangenen Kapitels lässt sich das Pruningpotenzial für alle untersuchten Netze entnehmen.

$$\max_{r} C(\mathbf{X}, \mathbf{Y}|f(l', p)) - C(\mathbf{X}, \mathbf{Y}|\lambda) < \varepsilon \tag{5.2}$$

Abbildung 5.3 visualisiert das Pruningpotenzial der Convolutional Schichten der *UCI Custom* Architektur für unterschiedliche Pruningmethoden und ε. Daran ist zwar erkennbar, dass beide Faktoren auf das Pruningpotenzial Einfluss nehmen. Bei der Bewertung einer Schicht ist allerdings nur die Reihenfolge der Schichten relevant, die in den gezeigten Beispielen gleich bleibt.

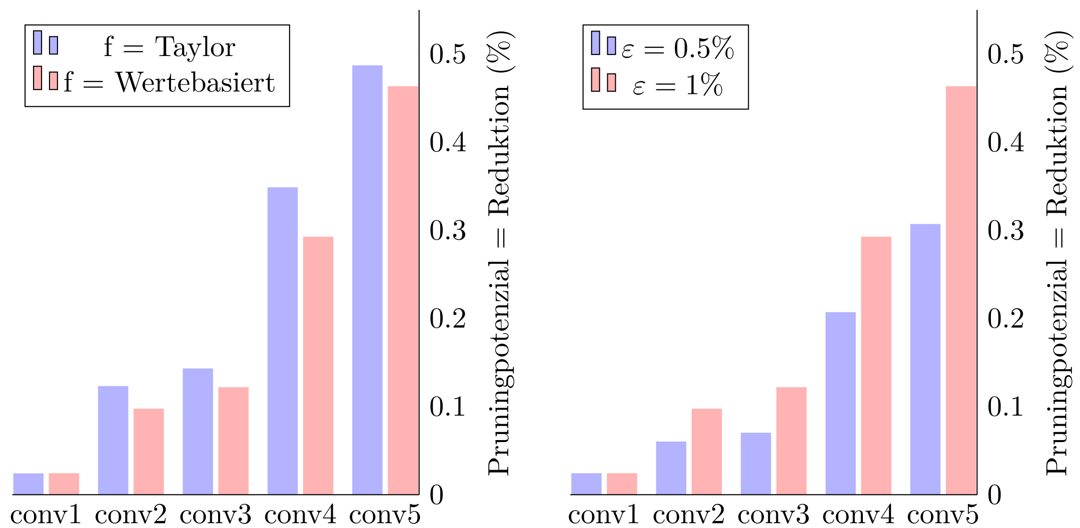


Abbildung 5.3: Pruningpotenzial anhand der Convolutional Schichten der UCI=Custom Architektur

SNARE nutzt die Anzahl der Neuronen / Filter einer Schicht als Approximation des Pruningpotenzials zur Berechnung eines Scores $\mathcal{S}$ (Formel 5.3). Abbildung 5.4 zeigt den Score im Vergleich zu dem errechneten Pruningpotenzial. Als heuristische Annahme dient es dazu, eine Bewertung der Schichten zu ermöglichen und damit die Ausführungszeit von SNARE zu verkürzen.

$$\mathcal{S}(z_l) = \frac{\|z_l\|_{neurons}}{\|\mathcal{N}\|_{neurons}} \tag{5.3}$$

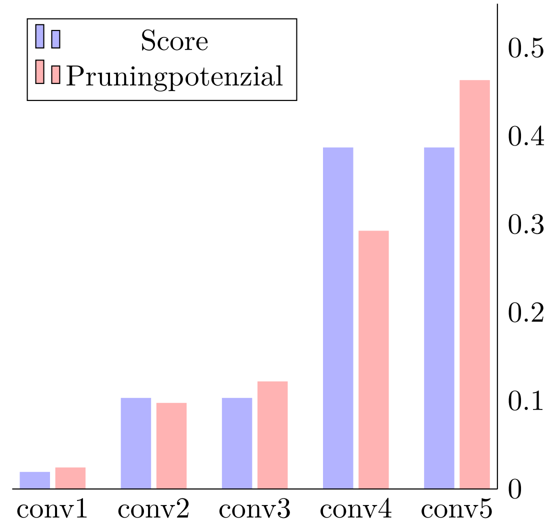


Abbildung 5.4: Pruningpotenzial im Vergleich zu dem Scoring von SNARE

5.3.1 Regularisierung

SNARE berücksichtigt neben des Pruningpotenzials noch weitere Faktoren bei der Bewertung der Schichten. Diese können vom Anwender beliebig abgeändert werden. Die Motivation dahinter ist, dass einerseits verschiedene Schichten mit hohem Pruningpotenzial existieren können, aber auf praxisrelevante Faktoren wie den Speicherplatzbedarf keinen Einfluss haben. Andererseits ist es denkbar, dass in bestimmten Anwendungsfällen besonders eine Verminderung der Ausführungszeit im Vordergrund steht. Dies ist jedoch mit der bisherigen Bewertung in dieser Art nicht möglich.

Parameter: Die Anzahl der Gesamtparameter und somit die Größe eines Netzes wird in nahezu allen Arbeiten des Prunings als Evaluation herangezogen, da es auch in der Praxis von großer Bedeutung ist. Durch die Parameter-Regularisierung aus Formel 5.4 ist es in SNARE möglich, Schichten mit vielen Parametern bevorzugt auszuwählen.

$$\mathcal{S}(z_l) = \mathcal{S}(z_l) + \beta \frac{\|z_l\|_{params}}{\|\mathcal{N}\|_{params}} \tag{5.4}$$

FLOPs: Die Anzahl der FLOPs hängt direkt mit der benötigten Rechenzeit eines neuronalen Netzes zusammen. Formel **??** ermöglicht in SNARE aus diesem Grund, Schichten mit vielen FLOPs in der Auswahl höher zu gewichten. Dies ist auch besonders

deshalb wichtig, da die Schichten mit vielen FLOPs in allen in dieser Arbeit untersuchten Architekturen ein geringes Pruningpotenzial aufweisen. Zudem haben diese als früher Schichten eines CNNs häufig wenige Parameter und werden somit auch nicht durch die Parameterregularisierung abgedeckt.

$$\mathcal{S}(z_l) = \mathcal{S}(z_l) + \gamma \frac{\|z_l\|_{flops}}{\|\mathcal{N}\|_{flops}} \tag{5.5}$$

5.3.2 Operationen

Bei allen bisher beschriebenen Bewertungen handelt es sich um Heuristiken, die eine Schicht auswählen. Unbestimmt ist bis dahin aber noch, wie viele Neuronen oder Filter abgeschnitten werden können. Ebenso ist unklar, inwiefern das Netz überhaupt zu reduzieren ist. Aus diesen Gründen definiert SNARE Operationen, die zunächst von großen Reduktionsraten ausgehen und sich im Laufe des Prozesses anpassen. Eine Operation ist definiert als Tripel (l, f, p). l ist die Schicht auf der das Pruningverfahren f ausgeführt wird. p bestimmt die Operationsgröße, welche der prozentualen Reduktionsrate von l nach dem Pruningschritt entspricht.

In jeder Iteration erstellt SNARE für jede zu prunende Schicht eine Operation. Dabei werden die Operationsgrößen initial auf 64% festgelegt und pro Schicht gespeichert. Diese Angabe ist bewusst hoch gewählt, da bereits aus der Literatur Reduktionsraten weit über 50% auf verschiedenen Architekturen bekannt sind [SB15]. Eine zu niedrige Angabe bei derartigen Netzen würde dazu führen, dass SNARE eine höhere Anzahl an Iterationen benötigt, um diese Reduktionsraten zu erreichen.

Gleichzeitig ist es Ziel von SNARE, auch kleinere Reduktionsraten automatisiert zu ermöglichen. So kann bei bereits geprunten Netzwerken eine Operationsgröße von 64% einer einzelnen Schicht signifikanten Genauigkeitsverlust bewirken. SNARE löst dies, indem jede durchgeführte Operation am Ende einer Iteration evaluiert und die Operationsgröße aktualisiert wird. Falls der akzeptierte Genauigkeitsverlust ε in dieser Evaluation überschritten wurde, wird die Iteration verworfen und die Operationsgröße der Schicht l für folgende Iterationen halbiert. Der Hintergrund der exponentiellen Abnahme ist, dass SNARE sowohl große Reduktionsraten erreichen, aber auch die Operationsgröße nach wenigen fehlgeschlagenen Pruningversuchen ausreichend korrigieren kann. Ohne Kenntnis der möglichen Reduktionsraten zielt dieser Ansatz darauf ab, die Gesamtanzahl und besonders die Anzahl der verworfenen Iterationen zu minimieren.

Durch die Scoreberechnung des vorherigen Abschnittes werden Schichten bewertet. Zusätzlich berücksichtigt SNARE auch die gespeicherten Operationsgrößen. Die individuellen Scores werden dafür mit den geschätzten Operationsgrößen gewichtet. Insgesamt

wählt SNARE also keine Schichten aus, sondern Operationen der Größe p auf den bewerteten Schichten. Dadurch bevorzugt SNARE Pruningoperationen auf Schichten mit geringem Score, aber hoher geschätzter Operationsgröße.

Sobald die Operationsgröße einer Schicht unter 8% sinkt, wird die Schicht in nachfolgenden Iterationen nicht mehr bewertet und keine Operation definiert. SNARE terminiert, wenn keine Schicht als reduzierbar angesehen wird. In der Implementierung von SNARE kann darüber hinaus eine maximale Iterationsanzahl angegeben werden. Berechnungen von späten Iterationen mit niedrigen Operationsgrößen sind somit regulierbar.

Für SNARE wurde der beschriebene Ansatz zur Definition von Operationen verwendet. Generell sind weitere Verfahren ebenfalls realisierbar. Denkbar ist beispielsweise eine statische, anstelle der dynamischen Operationsgröße von SNARE. Diese muss jedoch deutlich geringer sein, um beliebige Architekturen zu reduzieren, was sich wiederrum negativ auf die Anzahl der benötigten Iterationen auswirkt. Eine weiterer Ansatz für zukünftige Arbeiten ist es, die Operationsgrößen in jeder Iteration neu zu schätzen aus den Informationen einer Schicht.

5.4 Pruning und Retraining

Nach der Auswahl einer geeigneten Operation für das Pruning, ist der nächste Schritt von SNARE die schichtweise Reduktion von Neuronen. SNARE ist dabei nicht an ein bestimmtes Verfahren gebunden, sondern funktioniert mit beliebigen Techniken. Die SNARE Implementierung in Keras / TensorFlow unterstützt zum Zeitpunkt dieser Arbeit alle evaluierten Techniken aus Abschnitt 4.4 und ist zusätzlich leicht erweiterbar. Der Nutzer des Tools kann daraus frei auswählen. Standardmässig ist die Taylorapproximation aus [Mol+16] eingestellt.

Wie auch beim initialen Training eines Netzes gibt es zahlreiche Optionen, den Trainingschritt von SNARE zu konfigurieren. Dies beinhaltet die Festlegung eines Optimierers, der Lernrate, der Trainingsdaten, der Batchgröße und des Trainingsumfangs. In der SNARE Implementierung werden diese Einstellungen aus dem initialen Training übernommen. Einzig der Trainingsumfang wird auf 5 Epochen festgelegt, kann aber ebenfalls durch den Nutzer abgeändert werden. Dieser Wert hat sich in den Experimenten als guter Kompromiss zwischen akzeptabler Trainingszeit und erreichter Genauigkeit bewährt.

5.4.1 Implementierungsdetails

Schichtanordnung

Um den Pseudocode aus Abbildung 5.2 in einer Bibliothek wie Keras zu realisieren, müssen weitere Details beachtet werden. Formal lässt sich beispielsweise das Pruning eines Neurons umsetzen, indem jeder zugehöriger Parameter der Gewichtsmatrix $\mathbf{W}$ auf 0 gesetzt wird. Bei dem Neuron i sind alle Werte der Zeile i in $\mathbf{W}$ somit 0. Mit dem Einsatz von GPUs hat dies zur Folge, dass diese Nullen für die Ausführung des neuronalen Netzes weiterhin geladen und für die Matrixmultiplikation verwendet werden. Pruning hat in dieser Form von Keras CNNs also keinen positiven Effekt. Die Lösung ist, die Zeile der Gewichtsmatrix komplett zu entfernen. Bei einer Schicht l der Größe $\|\| = n$ entsteht durch das Reduzieren von p Neuronen also eine Zeilenanzahl der Größe $n - p$. Jedoch besitzt die $\mathbf{W}_{l+1}$ der nachfolgende Schicht $l+1$ n Spalten und die Neuronen sind somit nicht mit einer Eingabe der Größe $n - p$ kompatibel. Deshalb ist es erforderlich, jede Spalte der k entfernten Neuronen aus $\mathbf{W}_{l+1}$ zusätzlich zu entfernen.

Selbst auf Filter von Convolutional Schichten in Keras ist dies übertragbar. Für 2D-Convolutional Schichten speichert Keras die Parameter in einem vierdimensionalen Tensor der Größe (n, c, k_w, k_h). n ist dabei die Anzahl der Filter, c die Anzahl der Eingangskanäle und k_w / k_w definiert die Dimension der Kernel. Durch das Entfernen eines Neurons i entfällt ein Element der ersten Dimension, was zu einer Verringerung der Parameterzahl um $c * k_w * k_h$ führt. In der nachfolgenden Schicht muss stattdessen ein Element der 2. Dimension abgeschnitten werden. Damit fallen $n^{l+1} * k_w^{l+1} * k_h^{l+1}$ Parameter weg.

Diese Identifikation von Parametern ist bei konsekutiven vollverbundenen Schichten oder Convolutional Schichten unkompliziert. Ein Tool wie SNARE, welches beliebige Folgen von Schichten verarbeiten soll, muss auch Mischformen korrekt bearbeiten können. Ein konkretes Neuron aus einer Schicht zu entfernen ist nach oben genanntem Schema immer realisierbar. Das Problem entsteht erst bei den Gewichtungen der Folgeschicht. Ein klassisches Beispiel dafür ist, dass in vielen Architekturen wie LeNet-5 nach mehreren Convolutional Schichten vollverbundene Schichten zur Klassifizierung folgen. Nach der letzten 2D-Convolutional Schicht l wird eine Dimensionsreduktion der Ausgabe durchgeführt. Die Ausgabe der Größe $w * h$ von n Filtern wird in ein eindimensionalen Vektor der Größe $n * w * h$. überführt. Unter Beachtung des eingesetzten Verfahrens und der Größe dieser 3 Hyperparameter lassen sich im Pruningverfahren eines Filters i die zugehörigen Parameter ermitteln.

Pooling Schichten erschweren diese Identifikation zusätzlich. Zwischen einzelnen Convolutional Schichten hat eine eingesetzte Pooling Schicht jedoch keine Auswirkung,

weil die Größe der Ausgabe nicht mit den Kernelgewichten der nachfolgenden Schicht zusammenhängt. Vor vollverbundenen Schichten kann es je nach Poolingart die Ausgabe und somit die Position der Gewichtungen beeinflussen. Besonders das Verhalten an Rändern muss individuell berücksichtigt werden.

Modellabstraktion

In Keras heißen alle neuronalen Netzen Modelle, die nach der Modelldefinition im Speicher gehalten werden. Bei Modellen mit vielen Parametern kann der zugrundeliegende TensorFlow-Graph große Teile des verfügbaren Speichers einnehmen. Um dennoch für das Pruning unterschiedliche Architekturen aufbauen zu können, wurde für SNARE eine Modellabstraktion designt. Es enthält alle Informationen, die zum Aufbau eines Modells notwendig sind, ohne das Modell komplett im Hauptspeicher zu halten. Die Parameter werden auf der Festplatte gespeichert und lediglich Referenzen im Hauptspeicher hinterlegt. Diese sogenannten *ModelWrapper* können aus validen Modellen generiert und in valide Kerasmodelle überführt werden. Dies ermöglicht SNARE Kopien von Modellen speichersparsam zu erstellen und Zwischenergebnisse von früheren Iterationen zu sichern.

Auch die durchgeführten Pruningoperationen sind für eine geringe Speicherauslastung konzipiert. Statt Kopien von allen Parametern im Speicher zu halten, werden lediglich Referenzen zu den Parametern und die durchgeführten Operationen gesichert. Konkret ist dies der Index der zu entfernenden Neuronen. Erst bei der Umwandlung in ein gültiges Modell, werden die übrigen Neuronen aus dem gespeicherten Parametersatz in den Hauptspeicher kopiert.

Die Motivation dieser Abstraktion ist, dass Bibliotheken wie TensorFlow nicht dafür ausgelegt sind, verschiedene Schichten und deren Parameter zwischen Architekturen zu teilen. Das wirkt sich negativ auf Pruningmethoden aus. Bereits das Entfernen eines einzelnen Neurons erzwingt eine Kopie eines gesamten Netzes im Hauptspeicher. So zeigte sich bei Experimenten, dass SNARE durch das Erstellen von potenziellen neuen Kerasmodellen aus der VGG16 Architektur bereits in der dritten Iteration den verfügbaren Hauptspeicher von 16GB überschritten hat. Mit der Abstraktion ist es problemlos möglich gewesen, Ergebnisse von hunderten Iterationen zur Verfügung zu stellen.

Die Toolimplementierung erstellt in jedem Schritt potenzielle Architekturen, die trainiert und evaluiert werden müssen. Aktuell werden diese rechenintensiven Schritte sequenziell ausgeführt. Die Serialisierbarkeit der *ModelWrapper* kann es zukünftig ermöglichen, das Training und die Evaluation von verschiedenen Architekturen auf unterschiedliche Prozesse oder Rechner auszulagern.

5.4.2 Resultate

Die Ergebnisse von SNARE auf unterschiedlichen Architekturen befinden sich in Tabelle 5.2. PR ist die erzielte Parameterreduktion des Gesamtnetzes und FR die FLOP-Reduktion. Als den erlaubten Genauigkeitsverlust ε gegenüber der besten erreichten Genauigkeit wurde jeweils 0,5% festgelegt. Um zu zeigen, dass SNARE auch für kleinere ε noch gute Reduktionsraten erzielen kann, wurden die LeNet-5-Architekturen mit $\varepsilon = 0{,}1\%$ separat reduziert, was allerdings die Anzahl der benötigten Iterationen signifikant erhöht hat.

LeNet-300-100

	Top-1-Fehler (%)	ΔGenauigkeit	#param.	PR	FR	Iterationen
Basis	1,63	-	266,200k	-	-	-
mit FLOPs Reg.	1,82	-0,19	16,064k	16	15	15
mit Param. Reg.	1,79	-0,16	6,224k	44	41	15
mit beidem	1.80	-0,17	2,004k	133	140	15
	1.97	-0.34	1,662k	166	144	18
$\varepsilon = 0{,}1$	1,65	-0,02	10,201k	26	26	30
	1,67	-0,04	8,112k	33	30	60

LeNet-5

	Top-1-Fehler (%)	ΔGenauigkeit	#param.	PR	FR	Iterationen
Basis	1,00	-	61,706k	-	-	-
mit FLOPs Reg.	1,27	-0,27	9,067k	7	2,77	15
mit Param. Reg.	1,23	-0,23	3,237k	19	1,46	15
mit beidem	1.21	-0,21	3,914k	16	2,42	15
	1.36	-0.36	1,782k	35	2,44	29
$\varepsilon = 0{,}1$	1,01	-0,01	14,501k	4	1,2	30
	1,03	-0,03	7,701k	8	1,35	60

UCI Custom

	Top-1-Fehler (%)	ΔGenauigkeit	#param.	PR	FR	Iterationen
Basis	9,52	-	279,086k	-	-	-
mit FLOPs Reg.	9,68	-0,16	17,691k	16	4,66	15
mit Param. Reg.	9,57	-0,05	5,906k	47	2,67	15
mit beidem	9,35	+0,17	6,975k	40	4,38	15
	9,61	-0.09	1,847k	151	6,79	30
	9,33	+0,19	1,134k	245	8,01	48

VGG16 (4096 Neuronen, nur vollverbundene Schichten)

	Top-1-Fehler (%)	ΔGenauigkeit	#param.	PR	Iterationen
Basis	31,17	-	123,432M	-	-
mit FLOPs Reg.	30,90	+0,27	10,682M	12	8
mit Param. Reg.	30,85	+0,32	9,919M	12	8
mit beidem	30,86	+0,31	10,567M	12	8
	29,86	+1.31	8,426M	15	16
	29,97	+1,19	8,239M	15	18

Tabelle 5.2: Resultate von SNARE auf unterschiedlichen Architekturen

6 Abschluss

In diesem Kapitel werden die Erkenntnisse der vorliegenden Thesis abschließend zusammengefasst. Daraufhin werden weitere mögliche Erweiterungen von SNARE diskutiert. Im letzten Abschnitt wird ein Ausblick auf weiterführende Arbeiten gegeben.

6.1 Zusammenfassung

In der vorliegenden wurden verschiedene Verfahren vorgestellt, trainierte neuronale Netze zu komprimieren. Es wurde an gängigen Architekturen demonstriert, wie Pruningmethoden die Anzahl der Parameter und Rechenschritte verringern können. Die Experimente haben dabei gezeigt, dass selbst simple Verfahren in der Lage sind, einzelne Schichten der LeNet-5 Architektur ohne Genauigkeitsverlust über den Faktor 10 hinaus zu reduzieren. Selbst das zufällige Abschneiden von Neuronen ist hierbei eine valide Strategie.

Der wichtigste Beitrag dieser Arbeit ist die Vorstellung von SNARE. Dieser neue Ansatz ermöglicht es, CNNs zu automatisiert zu reduzieren. In einem iterativen Prozess analysiert und bewertet SNARE die Schichten eines neuronales Netzes. Als Grundlage für die Bewertung dient eine neue heuristische Metrik, das Pruningpotenzial. Ergänzend berücksichtigt SNARE weitere Faktoren wie die Anzahl der benötigten Inferenz-Rechenoperationen.

Im Kontext der Arbeit ist auch eine Implementierung von SNARE entstanden. Als Tool reduziert es automatisch Keras-CNNs mit TensorFlow Backend. Die damit verbundenen Herausforderungen wurden in dieser Arbeit ebenfalls erläutert. Der Nutzer des Tools kann insbesondere die Bewertung der Schichten individuell konfigurieren. Als Ausgabe produziert das Tool ein gültiges reduziertes CNN, welches den gewählten Genauigkeitsverlust nicht überschreitet. Weiterhin wurde in dieser Arbeit die Implementierung von SNARE an verschiedenen Architekturen evaluiert. Bei zwei der untersuchten Architekture erreichte SNARE Parameterreduktionen, die den Faktor 150 überschreiten.

Die Wirksamkeit von erneutem Training ist dabei besonders hervorzuheben. Dieses anschließende Training dient dazu, die induzierten Fehler des Prunings zu kompensie-

ren. Erst dieses Training ermöglicht die angegebenen Reduktionsraten von SNARE. Durch sogenanntes Blocktraining wird ein möglicher Ansatz illustriert, das Training auf bestimmte Teile des Netzes zu beschränken.

6.2 Erweiterungen SNARE

Die Implementierung von SNARE ist zum aktuellen Zeitpunkt nur auf sequenzielle CNNs ausgelegt. Der Datenfluss in Architekturen wie GoogLeNet [Sze+14] oder ResNet [He+15] dagegen kann sich aufspalten. Um diese Art von Netz zu unterstützen, müssen Anpassungen der Implementierung vorgenommen werden. Da dadurch eine einzelne Schicht mehrere Nachfolger hat, müssen die Architekturveränderungen durch einen Pruningvorgang für diesen Fall angepasst werden.

Der modulare Aufbau der SNARE-Implementierung vereinfacht es, weitere Pruningmethoden für einzelne Schichten zu integrieren. Für die Entwicklung von neuen Bewertungsheuristiken und Operationsdefinitionen sind auch diese Teile der Implementierung leicht erweiterbar.

Eine weitere potenzielle Erweiterung von SNARE ist, das Verfahren zu parallelisieren. In Abschnitt 5.4.1 sind dazu bereits Ansätze diskutiert worden. Die größte Herausforderung ist dabei das erneute Trainieren von neuronalen Netzen. Alternativen wie das vorgestellte Blocktraining können die Laufzeit der SNARE-Implementierung signifikant verbessern.

6.3 Ausblick

Literatur

[KSH12] Alex Krizhevsky, Ilya Sutskever und Geoffrey E. Hinton. „ImageNet Classification with Deep Convolutional Neural Networks“. In: NIPS’12 (2012), S. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[Mol+16] Pavlo Molchanov et al. „Pruning Convolutional Neural Networks for Resource Efficient Inference“. In: (19. Nov. 2016). arXiv: http://arxiv.org/abs/1611.06440v2 [cs.LG].

[DCP17] Xin Dong, Shangyu Chen und Sinno Jialin Pan. „Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon“. In: (22. Mai 2017). arXiv: http://arxiv.org/abs/1705.07565v2 [cs.NE].

[Han+15] Song Han et al. „Learning both Weights and Connections for Efficient Neural Networks“. In: (8. Juni 2015). arXiv: http://arxiv.org/abs/1506.02626v3 [cs.NE].

[Lin+16] Shaohui Lin et al. „Towards Convolutional Neural Networks Compression via Global Error Reconstruction“. In: *IJCAI*. 2016.

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[CLG01] Rich Caruana, Steve Lawrence und C Lee Giles. „Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping“. In: *Advances in neural information processing systems*. 2001, S. 402–408.

[Sri+14] Nitish Srivastava et al. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15 (2014), S. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[Fis+07] A Fiszelew et al. „Finding optimal neural network architecture using genetic algorithms“. In: *Advances in computer science and engineering research in computing science* 27 (2007), S. 15–24.

[Ian+16] Forrest N. Iandola et al. „SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size“. In: (24. Feb. 2016). arXiv: http://arxiv.org/abs/1602.07360v4 [cs.CV].

[FC18] Jonathan Frankle und Michael Carbin. „The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks“. In: *ICLR 2019* (9. März 2018). arXiv: http://arxiv.org/abs/1803.03635v5 [cs.LG].

[Lin+17] Ji Lin et al. „Runtime neural pruning“. In: *Advances in Neural Information Processing Systems*. 2017, S. 2181–2191.

[Ros58] F. Rosenblatt. „The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain“. In: *Psychological Review* (1958), S. 65–386.

[Kal17] Shivaram Kalyanakrishnan. „The Perceptron Learning Algorithm and its Convergence“. In: (2017).

[SZ14] Karen Simonyan und Andrew Zisserman. „Very Deep Convolutional Networks for Large-Scale Image Recognition“. In: (4. Sep. 2014). arXiv: http://arxiv.org/abs/1409.1556v6 [cs.CV].

[Zop+17] Barret Zoph et al. „Learning Transferable Architectures for Scalable Image Recognition“. In: (21. Juli 2017). arXiv: http://arxiv.org/abs/1707.07012v4 [cs.CV].

[FH19] Matthias Feurer und Frank Hutter. „Hyperparameter Optimization“. In: *Automated Machine Learning: Methods, Systems, Challenges*. Hrsg. von Frank Hutter, Lars Kotthoff und Joaquin Vanschoren. Cham: Springer International Publishing, 2019, S. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

[Ber+11] James S. Bergstra et al. „Algorithms for Hyper-Parameter Optimization“. In: *Advances in Neural Information Processing Systems 24*. Hrsg. von J. Shawe-Taylor et al. Curran Associates, Inc., 2011, S. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[HAH01] Xuedong Huang, Alex Acero und Hsiao-Wuen Hon. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 0130226165.

[Lec+98] Y. Lecun et al. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), S. 2278–2324. DOI: 10.1109/5.726791.

[Sze+14] Christian Szegedy et al. „Going Deeper with Convolutions“. In: (17. Sep. 2014). arXiv: http://arxiv.org/abs/1409.4842v1 [cs.CV].

[GR18] Tarun Kumar Gupta und Khalid Raza. „Optimizing Deep Neural Network Architecture: A Tabu Search Based Approach“. In: (17. Aug. 2018). arXiv: http://arxiv.org/abs/1808.05979v1 [cs.LG].

[KV95] Anders Krogh und Jesper Vedelsby. „Neural network ensembles, cross validation, and active learning". In: *Advances in neural information processing systems*. 1995, S. 231–238.

[Hea08] Jeff Heaton. *Introduction to Neural Networks with Java, 2nd Edition.* Heaton Research, Incorporated, 2008. ISBN: 1604390085. URL: https://www.amazon.com/Introduction-Neural-Networks-Java-2nd/dp/1604390085?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1604390085.

[R+88] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams et al. „Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (1988), S. 1.

[Nie15] Michael A Nielsen. *Neural networks and deep learning*. Bd. 25. Determination press San Francisco, CA, USA: 2015.

[PG17] Josh Patterson und Adam Gibson. *Deep Learning: A Practitioner's Approach.* 1st. O'Reilly Media, Inc., 2017. ISBN: 1491914254, 9781491914250.

[L+95] Yann LeCun, Yoshua Bengio et al. „Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10 (1995), S. 1995.

[S+03] Patrice Y Simard, David Steinkraus, John C Platt et al. „Best practices for convolutional neural networks applied to visual document analysis." In: *Icdar*. Bd. 3. 2003. 2003.

[YTT11] Haruo Yanai, Kei Takeuchi und Yoshio Takane. *Projection Matrices, Generalized Inverse Matrices, and Singular Value Decomposition.* Springer-Verlag GmbH, 12. Apr. 2011. Kap. 4. ISBN: 1441998861. URL: https://www.ebook.de/de/product/14685515/haruo_yanai_kei_takeuchi_yoshio_takane_projection_matrices_generalized_inverse_matrices_and_singular_value_decomposition.html.

[RV15] Tim Roughgarden und Gregory Valiant. *The Modern Algorithmic Toolbox Lecture# 9: The Singular Value Decomposition (SVD) and Low-Rank Matrix Approximations.* 2015.

[BHH19] Bernhard Bermeitinger, Tomas Hrycej und Siegfried Handschuh. „Singular Value Decomposition and Neural Networks". In: *ICANN 2019: Artificial Neural Networks and Machine Learning - Deep Learning* (27. Juni 2019). DOI: 10.1007/978-3-030-30484-3_13. arXiv: http://arxiv.org/abs/1906.11755v1 [cs.LG].

[Yag+19] Atsushi Yaguchi et al. „Scalable Deep Neural Networks via Low-Rank Matrix Factorization“. In: (29. Okt. 2019). arXiv: http://arxiv.org/abs/1910.13141v1 [cs.LG].

[Den+14] Emily Denton et al. „Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation“. In: (2. Apr. 2014). arXiv: http://arxiv.org/abs/1404.0736v2 [cs.CV].

[Lee+19] Dongsoo Lee et al. „Learning Low-Rank Approximation for CNNs“. In: (24. Mai 2019). arXiv: http://arxiv.org/abs/1905.10145v1 [cs.LG].

[Kim+15] Yong-Deok Kim et al. „Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications“. In: (20. Nov. 2015). arXiv: http://arxiv.org/abs/1511.06530v2 [cs.CV].

[Li+18] Guiying Li et al. „Optimization based Layer-wise Magnitude-based Pruning for DNN Compression“. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, Juli 2018, S. 2383–2389. DOI: 10.24963/ijcai.2018/330. URL: https://doi.org/10.24963/ijcai.2018/330.

[KML04] Greer B. Kingston, Holger R. Maier und Martin F. Lambert. „A Statistical Input Pruning Method for Artificial Neural Networks Used in Environmental Modelling“. In: 2004.

[AK11] M. Gethsiyal Augasta und T. Kathirvalavakumar. „Reverse Engineering the Neural Networks for Rule Extraction in Classification Problems“. In: *Neural Processing Letters* 35.2 (Dez. 2011), S. 131–150. DOI: 10.1007/s11063-011-9207-8.

[LDS90] Yann LeCun, John S. Denker und Sara A. Solla. „Optimal Brain Damage“. In: *Advances in Neural Information Processing Systems 2*. Hrsg. von D. S. Touretzky. Morgan-Kaufmann, 1990, S. 598–605. URL: http://papers.nips.cc/paper/250-optimal-brain-damage.pdf.

[GYC16] Yiwen Guo, Anbang Yao und Yurong Chen. „Dynamic Network Surgery for Efficient DNNs“. In: (16. Aug. 2016). arXiv: http://arxiv.org/abs/1608.04493v2 [cs.NE].

[HS93] Babak Hassibi und David G. Stork. „Second order derivatives for network pruning: Optimal Brain Surgeon“. In: *Advances in Neural Information Processing Systems 5*. Hrsg. von S. J. Hanson, J. D. Cowan und C. L. Giles. Morgan-Kaufmann, 1993, S. 164–171. URL: http://papers.nips.cc/paper/647-second-order-derivatives-for-network-pruning-optimal-brain-surgeon.pdf.

[Has+93] Babak Hassibi et al. „Optimal Brain Surgeon: Extensions and Performance Comparisons“. In: *Proceedings of the 6th International Conference on Neural Information Processing Systems*. NIPS'93. Denver, Colorado: Morgan Kaufmann Publishers Inc., 1993, S. 263–270. URL: http://dl.acm.org/citation.cfm?id=2987189.2987223.

[Han+16] Song Han et al. „EIE: Efficient Inference Engine on Compressed Deep Neural Network“. In: (4. Feb. 2016). arXiv: http://arxiv.org/abs/1602.01528v2 [cs.CV].

[Eng01] A. P. Engelbrecht. „A new pruning heuristic based on variance analysis of sensitivity information“. In: *IEEE Transactions on Neural Networks* 12.6 (Nov. 2001), S. 1386–1399. ISSN: 1941-0093. DOI: 10.1109/72.963775.

[SB15] Suraj Srinivas und R. Venkatesh Babu. „Data-free parameter pruning for Deep Neural Networks“. In: (22. Juli 2015). arXiv: http://arxiv.org/abs/1507.06149v1 [cs.CV].

[BSC16] Mohammad Babaeizadeh, Paris Smaragdis und Roy H. Campbell. „NoiseOut: A Simple Way to Prune Neural Networks“. In: (18. Nov. 2016). arXiv: http://arxiv.org/abs/1611.06211v1 [cs.NE].

[Li+16] Hao Li et al. „Pruning Filters for Efficient ConvNets“. In: (31. Aug. 2016). arXiv: http://arxiv.org/abs/1608.08710v3 [cs.CV].

[Ath18] Ali Athar. „An Overview of Datatype Quantization Techniques for Convolutional Neural Networks“. In: (22. Aug. 2018). arXiv: http://arxiv.org/abs/1808.07530v1 [cs.NE].

[VSM11] Vincent Vanhoucke, Andrew Senior und Mark Z Mao. „Improving the speed of neural networks on CPUs“. In: (2011).

[Cou+16] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG].

[LTA15] Darryl D. Lin, Sachin S. Talathi und V. Sreekanth Annapureddy. „Fixed Point Quantization of Deep Convolutional Networks“. In: (19. Nov. 2015). arXiv: http://arxiv.org/abs/1511.06393v3 [cs.LG].

[HMD15] Song Han, Huizi Mao und William J. Dally. „Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding“. In: (1. Okt. 2015). arXiv: http://arxiv.org/abs/1510.00149v5 [cs.CV].

[TM18] F. Tung und G. Mori. „Deep Neural Network Compression by In-Parallel Pruning-Quantization“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2018), S. 1–1. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2018.2886192.

[Ang+13] Davide Anguita et al. „A Public Domain Dataset for Human Activity Recognition using Smartphones“. In: Jan. 2013.

[Faw+18] Hassan Ismail Fawaz et al. „Deep learning for time series classification: a review“. In: (12. Sep. 2018). DOI: 10.1007/s10618-019-00619-1. arXiv: http://arxiv.org/abs/1809.04356v4 [cs.LG].

[IS15] Sergey Ioffe und Christian Szegedy. „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“. In: (11. Feb. 2015). arXiv: http://arxiv.org/abs/1502.03167v3 [cs.LG].

[Tan+18] Chuanqi Tan et al. „A Survey on Deep Transfer Learning“. In: (6. Aug. 2018). arXiv: http://arxiv.org/abs/1808.01974v1 [cs.LG].

[Rus+14] Olga Russakovsky et al. „ImageNet Large Scale Visual Recognition Challenge“. In: (1. Sep. 2014). arXiv: http://arxiv.org/abs/1409.0575v3 [cs.CV].

[Yos+14] Jason Yosinski et al. „How transferable are features in deep neural networks?“ In: *Advances in neural information processing systems*. 2014, S. 3320–3328.

[He+15] Kaiming He et al. „Deep Residual Learning for Image Recognition“. In: (10. Dez. 2015). arXiv: http://arxiv.org/abs/1512.03385v1 [cs.CV].

Abkürzungsverzeichnis

OBD *Optimal Brain Damage*

OBS *Optimal Brain Surgeon*

SNARE *Scorebased Neural Architecture REduction*

CNN *Convolutional Neural Network*

SVD *Singular Value Decomposition*

Abbildungsverzeichnis

1.1 Vorgehensweise einer Iteration von SNARE 2

2.1 Aufbau eines Perzeptrons 6
2.2 Separierbarkeit von booleschen Operatoren 7
2.3 Illustration eines neuronalen Netzes mit zwei verdeckten Schichten . . . 8
2.4 Aktivierungsfunktionen im Vergleich 11
2.5 2D-Faltung einer Eingabe I mit Kernel K 16
2.6 Max Pooling der Größe 2x2 mit Downsampling 17

3.1 Einsatz von SVD in neuronalen Netzen aus [Yag+19] 20
3.2 Zusammenführen von zwei gleichen Neuronen aus [SB15] 25
3.3 Prunen eines Filters $h_i = \mathcal{F}_{i,j}$ führt zu dem Entfernen der zugehörigen Parameter sowie aller verbundenen Kernel der folgenden Schicht. [Li+16]. 26

4.1 Architektur von LeNet5 aus [Lec+98] 32
4.2 UCI Custom Architektur mit fünf 1D-Convolutional Schichten(gelb), zwei Max-Pooling Schichten(rot) und zwei vollverbundenen Schichten(violett) 33
4.3 Accelerometer-Trainingsbeispiele aus dem UCI Smartphone Dataset (oben) mit durchgeführter Vorverarbeitung (unten) 34
4.4 Pruning von Verbindungen der vollverbundenen Schichten der LeNet-5 Architektur 36
4.5 Pruning der Convolutional Schichten der LeNet-5 Architektur 37
4.6 Pruning der größten Convolutional Schicht der LeNet-5 Architektur . . 45
4.7 Taylorpruning der Convolutional Schichten der LeNet-5 Architektur . . 46

5.1 Vereinfachter Ablauf einer Iteration von SNARE 50
5.2 Pseudocode-Implementierung von SNARE 51
5.3 Pruningpotenzial anhand der Convolutional Schichten der UCI=Custom Architektur 53
5.4 Pruningpotenzial im Vergleich zu dem Scoring von SNARE 54

Tabellenverzeichnis

4.1 Die Klassen des UCI Datensatzes sowie die Anzahl der jeweiligen Trainings- und Testbeispiele . . . 33
4.2 Eigenschaften der untersuchten VGG16 Architekturen . . . 35
4.3 Pruning von Verbindungen auf unterschiedlichen Architekturen . . . 38
4.4 Pruning von Verbindungen mit erneutem Training . . . 39
4.5 Resultate von SNARE auf unterschiedlichen Architekturen . . . 47
4.6 Pruning von Neuronen mit erneutem Training . . . 48

5.1 Pruning von zwei Schichten gegenüber Pruning einer einzelnen Schicht . 52
5.2 Resultate von SNARE auf unterschiedlichen Architekturen . . . 60