



HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Methoden des Deep Learning im Bereich Convolutional Neural Networks

Matthias Hermann

Konstanz, 30.09.2015

MASTERARBEIT

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science (M. Sc.)

an der

Hochschule Konstanz

Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Master of Science, Informatik

Thema: **Methoden des Deep Learning im Bereich Convolutional Neural Networks**

Masterkandidat: Matthias Hermann, Zimmererweg 3, 78467 Konstanz

1. Prüfer: Prof. Dr. Matthias O. Franz

2. Prüfer: Martin Schall, M. Sc.

Ausgabedatum: 01.04.2015

Abgabedatum: 30.09.2015

Abstract

Thema:	Methoden des Deep Learning im Bereich Convolutional Neural Networks
Masterkandidat:	Matthias Hermann
Firma:	HTWG Konstanz
Betreuer:	Prof. Dr. Matthias O. Franz Martin Schall, M. Sc.
Abgabedatum:	30.09.2015
Schlagworte:	Machine Learning, Deep Learning, Convolutional Neural Network, Convolutional Autoencoder, Backpropagation, Gradient Descent, Regularization, Visualization MNIST, CIFAR-10

This thesis deals with Convolutional Neural Networks (CNNs) and their application in Deep Learning. CNNs represent a special case of Neural Networks which assume that the inputs have local features. These assumptions allow to encode certain properties into the architecture and make CNNs the state-of-the-art technology for image recognition, speech recognition and natural language processing. CNNs are Neural Network with special Convolution- and Pooling-Layers in the first layers for implicit feature extraction. Based on these features the last layers of the neural network perform classification or regression tasks. The new field of Deep Learning provides some very useful methods and tricks to address the difficulties in training such deep Neural Networks with Backpropagation. These tackle mainly the so called vanishing gradient effect and issues in optimazation of non-convex cost functions. Within the frame of this thesis the development of a CNN library is being described. It reaches a test error of 0.61 % on MNIST respectively 25.02 % on CIFAR-10 and provides most of the current techniques as Max-Pooling, Dropout training, unsupervised pre-training and visualization of features. Further it contains recent gradient descent acceleration methods as Equilibrium SGD, Nesterov-Momentum, RMSprop and AdaDelta.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Matthias Hermann, geboren am 04.12.1988 in Wangen im Allgäu*, dass ich

- (1) meine Masterarbeit mit dem Titel

Methoden des Deep Learning im Bereich Convolutional Neural Networks

bei der HTWG Konstanz unter Anleitung von Prof. Dr. Matthias O. Franz selbstständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

- (2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 30.09.2015

(Unterschrift)

Danksagung

Für die wissenschaftliche Betreuung meiner Masterarbeit und die interessanten und motivierenden Besprechungen möchte ich mich herzlich bei Prof. Dr. Matthias Franz bedanken. Ein herzlicher Dank geht auch an Martin Schall für die Erstellung des Zweitgutachtens sowie die vielen aufschlussreichen Diskussionen.

Dank gilt außerdem Michael Grundwald für die Durchsicht meiner Arbeit und die zahlreichen konstruktive Vorschläge.

Ganz herzlich möchte ich mich schließlich bei ZF Friedrichshafen AG und insbesondere bei Timo Möllers, Andreas Bauer und Wolfgang Duttle bedanken. Diese ermöglichten mir das Studium und standen während meines gesamten Masterstudiums unterstützend zur Seite.

Inhaltsverzeichnis

Danksagung	IV
Inhaltsverzeichnis	VI
1 Einleitung	1
1.1 Deep Learning	2
1.2 Ziel der Arbeit	4
1.3 Kapitelübersicht	5
2 Grundlagen Neuronaler Feedforward Netze	6
2.1 Perzeptron und Multilayer Perceptron	6
2.2 Aktivierungsfunktionen	8
2.3 Fehlermaße	11
2.3.1 Mittlerer Quadratischer Fehler	11
2.3.2 Cross-Entropy	12
2.4 Backpropagation-Algorithmus	12
2.5 Basisarchitekturen	14
2.5.1 Softmax-Regression (Klassifikation)	14
2.5.2 Funktionsapproximation (Regression)	16
2.6 Convolutional Neural Networks	17
2.6.1 Modellbeschreibung	17
2.6.2 Training mit Backpropagation	21
3 Spezielle Methoden im Deep Learning	24
3.1 Vorverarbeitung	26
3.1.1 Kontrastnormalisierung	27
3.1.2 ZCA-Whitening	27
3.2 Initialisierung	28
3.2.1 Normalisierte Fehlerpropagierung	28
3.2.2 Unüberwachtes Vortraining	29
3.3 Gradientenabstieg	33
3.3.1 Momentum	35
3.3.2 Adaptive Lernrate	36
3.3.3 Hessian Free Optimazation (HF)	40

3.4	Regularisierung und Generalisierung	40
3.4.1	A-priori Annahmen	41
3.4.2	Modellkapazität	43
3.4.3	Erweitern der Trainingsdaten	47
3.5	Visualisierungsmethoden	49
3.5.1	Primitiv	49
3.5.2	Gradientenbasiert	50
3.5.3	Nachverarbeitung	52
4	Implementierung des Prototyps	55
4.1	Architektur	56
4.1.1	C++ ConvNet	56
4.1.2	Python Module Extension	58
4.2	Implementierungsdetails	61
4.2.1	Vereinfachungen	61
4.2.2	Vektorisierung	61
4.2.3	Parallelisierung	63
4.2.4	Debugging	64
5	Experimente	65
5.1	Modelselektion	66
5.1.1	Modelarchitektur	67
5.1.2	Vorverarbeitung	68
5.2	Trainingsmethode	69
5.2.1	Vortraining	69
5.2.2	Initialisierung	71
5.2.3	Gradientenabstieg	73
5.2.4	Regularisierung	76
5.3	Visualisierung	82
5.3.1	Neuronen-Visualisierung	82
5.3.2	t-SNE-Methode	83
5.3.3	Autoencoder-Visualisierung	86
6	Zusammenfassung	89
6.1	Ergebnisse	89
6.2	Ausblick	91
	Literaturverzeichnis	93

Kapitel 1

Einleitung

Maschinelles Lernen stellt den Oberbegriff für Computerprogramme dar, welche hinsichtlich eines Performanzmaßes P (*Performance Measure*) eine Aufgabe A mit zunehmender Erfahrung E besser lösen können. Erfüllt ein Programm diese Anforderung wird es als lernend bezeichnet (vgl. Mitchell, 1997, S. 2). Die verschiedenen Teilgebiete dieser Disziplin unterscheiden sich in der Struktur der Aufgabe A sowie des auf die Erfahrung beziehungsweise Trainingsdaten angewandten Lernverfahrens. Seit der Erfindung des Computers wurden so eine Vielzahl verschiedener Modelle entwickelt (vgl. z.B. Hastie et al., 2009). Ganz allgemein können diese Modelle in vier Gruppen aufgeteilt werden. Diese entsprechen den verschiedenen Anwendungsgebiete des maschinellen Lernens und sind in Tabelle 1.1 aufgeführt.

Klassifikation	Regression
Dichteschätzung	Rangfolgebestimmung

Tabelle 1.1: Die verschiedenen Methoden im maschinellen Lernen (vgl. Hastie et al., 2009, S. 1 f.)

Ein weiteres integrales Merkmal eines Models im maschinellen Lernen ist das angewandte Lernverfahren, auch Lernparadigma genannt (vgl. hierzu und im Folgenden Haykin (1998), S. 63 ff. und Becker (1991)).

- Überwachtes Lernen (*Supervised learning*): Die Trainingsdaten $D_1 = \{(x_1, y_1), \dots, (x_n, y_n)\}$ liegen in Eingabe-Ausgabe-Beispielen vor, mit dem Ziel die Abbildung von Ein- nach Ausgabe beziehungsweise Trainingsbeispiel x_i und Label y_i zu lernen.
- Unüberwachtes Lernen (*Unsupervised learning*): Die Trainingsdaten $D_2 = \{(x_1), \dots, (x_n)\}$ liegen ohne Labels vor. Das Modell lernt statistische Merkmale der Daten und somit eine interne Repräsentation.

- Bekräftigungslernen (*Reinforcement learning*): Hierbei wird ebenfalls eine Abbildung von Ein- nach Ausgabe gelernt. Allerdings erhält das System lediglich zeitlich verzögert Rückmeldung über die Performanz der Abfolge von Ausgaben. Ziel ist es, hinsichtlich eines skalaren Performanzmaßes optimale Ausgaben zu erzeugen.

1.1 Deep Learning

Moderne Lernalgorithmen wie Support Vector Machines (SVMs) bieten viele interessante Eigenschaften, wie Schlupfvariablen, maximale Trennbreite und eine konvexe Kostenfunktion. Analysen der letzten Jahre zeigen jedoch, dass solche modernen parameterlosen Lernalgorithmen fundamentalen Restriktionen unterliegen (vgl. hierzu und im Folgenden Bengio und LeCun, 2007).

Zum einen ruhen diese auf einem dem *Fluch der Dimensionalität* sehr ähnlichen Problem (siehe Duda und Hart, 1973). Außerdem auf der Berechnung der lokalen Ähnlichkeit (Euklidischen Distanz) einer neuen Eingabe zu bereits bekannten Beispielen. Kern-basierte Verfahren mit lokalem Kern, wie dem Gauß-Kern, arbeiten unter der Annahme, dass die Zielfunktion glatt ist und verwandte Daten im Eingaberaum einer gewissen Ähnlichkeit unterliegen. Dies ist beispielsweise nicht mehr gegeben, wenn verwandte Daten in veränderter Form, wie affin transformiert, in unterschiedlichen Ausprägungen oder verstärkt, vorliegen. Nach Bengio und LeCun (2007) erfordert das Lernen von Funktionen mit großen Variationen im Eingaberaum ebenso viele Trainingsdaten, was in Verbindung zum *Fluch der Dimensionalität* steht. Des weiteren können Verfahren, welche auf lokaler Generalisierung beruhen, nicht zwischen relevanten und irrelevanten Merkmalen (z. B. Vorder- und Hintergrund) unterscheiden, was gerade in der Bilderkennung zu erheblichen Beschränkungen führt. Wünschenswert ist somit ein Verfahren, das nicht lokal generalisiert und mit steigender Anzahl Trainingsdaten (mehrere 10.000) gut skaliert.

Ein Lösungsansatz, um sehr komplexe Funktionen kompakt mit wenigen Parametern darzustellen, ist die Kombination vieler nicht-linearer Funktionen in Form der Gleichung 1.1.

$$f(x) = f''(f'(x)) \quad (1.1)$$

Dies ermöglicht es den Eingaberaum in unterschiedlichen Abstraktionsebenen zu partitionieren und so eine verteilte Repräsentation der Eingabe zu formen (*Distributed Representation*) (vgl. Rumelhart et al., 1986a). So können nicht-lokale Merkmale und große Variationen im Eingaberaum abgebildet werden, was gerade im Bereich Bildverarbeitung, natürliche Sprache und Robotik von großer Relevanz ist. Durch die Wahl eines parametrisierten Modells kann das Modell darüber hinaus mit steigender Anzahl Trainingsdaten

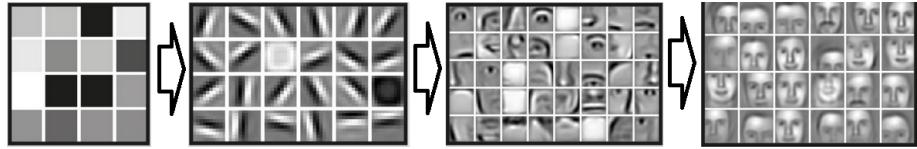


Abbildung 1.1: Merkmale unterschiedlichen Abstraktionsgrads von links (Pixel) bis rechts (Gesichter) (Einzelbilder: Andrew Ng, Stanford Artificial Intelligence Lab)

skalieren —Derartige Modelle heißen Künstliche Neuronale Netze (KNNs) (vgl. McCulloch und Pitts, 1943).

Angetrieben von der architektonischen Tiefe biologischer Gehirne, obgleich kein Beleg für eine derartige Funktionsweise dieser existiert (vgl. Becker, 1991), wurde so in der Vergangenheit oftmals versucht tiefe Netze (DNNs) überwacht zu trainieren (*Deep Learning*). Allerdings mit bedingtem Erfolg (vgl. Bengio und LeCun, 2007). Denn trotz der Ausdrucksstärke tiefer Modelle, stellt das Training solcher eine enorme Schwierigkeit dar (vgl. Becker, 1991). Oft genannte Gründe stehen in Zusammenhang mit zufälliger Initialisierung der Parameter und der Optimierung nicht-konvexer Zielfunktionen, welche durch viele Plateaus und lokalen Minima charakterisiert sind (vgl. Bengio, 2009). Bis auf bemerkenswerte Erfolge sogenannter *Convolutional Neural Networks* (vgl. LeCun et al., 1989) sind somit tiefe Architekturen in der Vergangenheit weitestgehend unbeachtet geblieben.

Renaissance durch Vortraining

Den Beginn des heutigen Deep Learning stellt die Arbeit von Hinton et al. (2006) dar: Ein unüberwachtes Lernverfahren, welches tiefe Architekturen schichtweise mittels eines *Greedy*-Algorithmus trainiert. Dadurch können, wie in Abbildung 1.1 dargestellt, unterschiedliche Schichten Merkmale mit unterschiedlichem Abstraktionsgrad extrahieren. Die Gesamtstrategie sieht dabei vor, zuerst Merkmale unüberwacht zu extrahieren und diese im Anschluss für ein etwaiges überwachtes Lernverfahren, wie beispielsweise Kernbasierte Verfahren (vgl. Hinton und Salakhutdinov, 2008), zu verwenden. Dies ähnelt stark den Ideen von Becker (1991), die Principle Component Analysis (PCA) zur Vorverarbeitung der Trainingsdaten zu verwenden. Durch die tiefe, schichtweise Extraktion der Merkmale sind diese jedoch nicht exklusiv, sondern verteilt, was auch als *Distributed Representation* bezeichnet wird (vgl. Hinton, 1986).

In den vergangenen Jahren konnten so bahnbrechende Fortschritte im Bereich der Spracherkennung (vgl Sainatha et al., 2015), der Verarbeitung natürlicher Sprache (NLP) (vgl. Socher et al., 2011) sowie der Bilderkennung erzielt werden. Den aktueller Benchmark in Letzterem stellt das *GoogLeNet* von Szegedy et al. (2014) mit 26% Top-5 Fehlern auf dem bekannten ImageNet-

Problem¹ (vgl. Russakovsky et al., 2015). Von den neuen Möglichkeiten im *Deep Learning* profitieren auch die im letzten Kapitel erwähnten *Convolutional Neural Networks* insofern, dass bestehende Ideen weiterentwickelt oder fehlende gelabelte Trainingsdaten kompensiert werden (vgl. Le et al., 2012).

Heute werden unter *Deep Learning* meist folgende fünf Modelle unterschieden:

- *Recurrent Neural Network* (RNN) mit Long Short Term Memory (LSTM) (vgl. Hochreiter und Schmidhuber (1997))
- *Convolutional Neural Network* (CNN) (vgl. LeCun et al. (1998a), Krizhevsky et al. (2012) und Simonyan und Zisserman (2014))
- *Deep Belief Network* (DBNs) mit beschränkten Boltzmann-Maschinen (RBMs) (vgl. Bengio et al. (2007) und Ranzato et al. (2007a))
- *Stacked Denoising Autoencoder* (SDA) (vgl. Vincent et al. (2008) und Vincent et al. (2010))
- *Deep Reinforcement Learning* (DRL) (vgl. Mnih et al. (2013))

1.2 Ziel der Arbeit

Convolutional Neural Networks (CNNs) sind eine besondere Klasse von Neuronalen Netzen. Diese von der Neurobiologie inspirierten Modelle in Verbindung mit neueren Methoden aus dem Bereich *Deep Learning*, stellen derzeit die besten Systeme für viele Probleme aus Bilderkennung, Spracherkennung und Verarbeitung natürlicher Sprache. Grundsätzlich werden drei Typen klassischer neuronaler Netze unterschieden: *Single-Layer Feedforward Networks*, *Multilayer Feedforward Networks* und *Recurrent Networks* (vgl. Haykin, 1998, S. 22f), wobei rekurrente Varianten in dieser Arbeit nicht weiter betrachtet werden.

Diese Arbeit beschränkt sich auf *Feedforward Networks* in den Bereichen Klassifikation und Regression und beschreibt somit das klassische, überwachte, neuronale Lernmodell. Für den Entwurf dieser neuronalen Netze existieren bereits eine Vielzahl an Softwaretools. Als Beispiel werden hier folgende bekannte Softwareprodukte angeführt:

- Theano (Python) (vgl. Bergstra et al., 2010)
- Torch (Lua) (vgl. Collobert et al., 2011)
- Cuda-Convnet (Cuda) (vgl. Nouri, 2013)

¹Der ImageNet-Datensatz besteht aus 200 Klassen, rund 450.000 Trainingsbeispielen, rund 20000 Beispielen zur Validierung und rund 40.000 Testbeispielen. Jedes Bild hat eine Größe von 482×415 (<http://www.image-net.org/> (26.08.2015)).

- PyLean2 (Python) (vgl. Goodfellow et al., 2013a)
- Caffe (C/C++) (vgl. Jia et al., 2014)

Ziel dieser Arbeit ist es zum einen die Methoden im Bereich *Deep Learning* aufzubereiten und zu analysieren. Darauf aufbauend wird ein eigener funktionierender Prototyp eines *Convolutional Neural Networks* entwickelt. Dieser soll möglichst effizient sein, wenige Abhängigkeiten haben sowie eine Python-Schnittstelle bereitstellen. Im Zentrum der Untersuchungen stehen dabei überwachte Lernverfahren mit dem Ziel, Bilddaten mit *State of the Art*-Performance zu klassifizieren, was allgemein als Bilderkennung bezeichnet wird. Darüber hinaus werden Methoden zum unüberwachten Vortraining und zur Visualisierung neuronaler Netze untersucht und vorgestellt.

Diese Arbeit entsteht am Institut für Optische Systeme (IOS) der HTWG Konstanz und dient zusammen mit zwei weiteren Arbeiten im Bereich *Stacked Denoising Autoencoder* und *LSTM-Recurrent Neural Network* als Grundlage für weitere Aktivitäten im Bereich *Deep Learning* im IOS.

1.3 Kapitelübersicht

Diese Thesis ist unterteilt in sechs Kapitel. Kapitel 1 beginnt mit der thematischen Einordnung dieser Arbeit sowie der Motivation. Der Hauptteil ist in zwei Blöcke unterteilt. In Kapitel 2 und 3 werden notwendige Grundlagen für die Implementierung des Prototyps vorgestellt. So führt Kapitel 2 die Methode Neuronale Netze ausgehend vom Perzeptron ein und beschreibt im Speziellen die Besonderheiten der *Convolutional Neural Networks* (CNNs). In Kapitel 3 werden spezielle Methoden des *Deep Learning* vorgestellt. Der zweite Block besteht aus Kapitel 4 und 5. Kapitel 4 beschreibt die Entwicklung des Prototyps und Kapitel 5 die verschiedenen durchgeführten Experimente mit dem Ziel den Prototyp auf Richtigkeit zu überprüfen sowie verschiedene Aspekte im Deep Learning zu untersuchen. Im abschließenden 6. Kapitel werden die relevanten Ergebnisse der Arbeit nochmals zusammengefasst und ein Ausblick auf über diese Arbeit hinausgehende Aspekte gegeben.

Kapitel 2

Grundlagen Neuronaler Feedforward Netze

Neuronale Netze sind äußerst vielseitig. Inspiriert von Gehirnen von Lebewesen, die einem komplexen, nichtlinearen, parallelen Computer ähneln, finden sie heute Anwendung in der Modellierung, Zeitreihenanalyse, Mustererkennung und Signalverarbeitung. Eine sehr wichtige Eigenschaft ist hierbei die Möglichkeit, von Eingabedaten beziehungsweise Trainingsdaten überwacht, unüberwacht und bekräftigend zu lernen (vgl. Haykin, 1998, S. 1). Dies erlaubt einen universellen Einsatz in allen Teilgebieten des maschinellen Lernens (vgl. Kapitel 1). Grundsätzlich werden drei Typen neuronaler Netze unterschieden: *Single-Layer Feedforward Networks*, *Multilayer Feedforward Networks* und *Recurrent Networks* (vgl. Haykin, 1998, S. 22f). Diese Arbeit beschränkt sich jedoch auf nicht rekurrente *Multilayer Feedforward Networks* in den Bereichen Klassifikation und Regression und beschreibt somit das klassische überwachte, neuronale Lernmodell (siehe Kapitel 1).

2.1 Perzeptron und Multilayer Perceptron

Heutige Neuronale Netze basieren auf dem in den späten 1950er von Rosenblatt (1962) entwickelten Perzeptron. Wie Abbildung 2.1 zeigt, ist die Architektur des Perzeptron an biologische Neuronen angelehnt. Die Ausgabe wird berechnet indem die Summe über die Produkte zwischen der Eingabe x_i und den Gewichten w_i gebildet wird. Erreicht die Summe einen definierten Schwellwert (*Bias*) $-b$, wird das Neuron aktiviert und gibt die binäre Eins aus, ansonsten die binäre Null. Formal berechnet das Perzeptron somit die in Gleichung 2.1 aufgeführte Funktion.

$$f(x) = \text{sign}\left(\sum_i^n w_i x_i + b\right) = \text{sign}(\langle w, x \rangle + b) \quad (2.1)$$

Obwohl das Perzeptron sich in dieser Form bereits als linearer Klassifikator

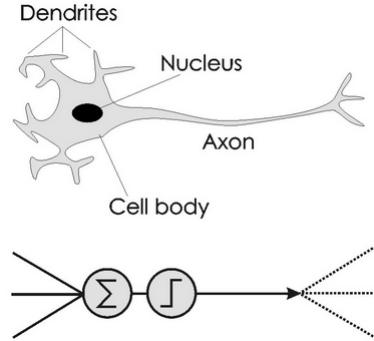


Abbildung 2.1: Schematischer Vergleich von biologischen und künstlichen Neuronen von McCulloch und Pitts (1943)(nach Winston (1992), S. 444 ff.)

eignet, ist es in seiner Ausdrucksstärke sehr eingeschränkt. So ist es beispielsweise nicht möglich Entscheidungshierarchien, wie in Netzwerken von McCulloch und Pitts (1943), zu bilden und somit das bekannte XOR-Problem zu lösen.

Die Erweiterung zum Perzeptron stellt das in den 1980er Jahren von Rumelhart et al. (1986a) entwickelte *Multilayer Perceptron* (MLP) dar. Die Architektur eines MLP wird bestimmt durch die Anzahl an Schichten (*Layers*) sowie der pro Schicht enthaltenen Neuronen. Abbildung 2.2 zeigt die generische Organisation mehrerer Neuronen in der Eingabeschicht (Input-Layer), der Verdeckten Schicht (Hidden-Layer) und der Ausgabeschicht (Output-Layer). Neuronen verallgemeinern hierbei das Perzeptron insofern, dass diese die Schwellwert-Aktivierungsfunktion durch eine beliebige Funktion $\phi(\cdot)$ ersetzen können. Somit ermöglichen diese auf die reellen Zahlen \mathbb{R} abzubilden. Darüber hinaus kann durch die Darstellung von Entscheidungshierarchien auch das XOR-Problem gelöst werden (vgl. Rojas, 1996, S. 125).

Zusammenfassend lässt sich sagen, dass durch die richtige Wahl der Architektur und Aktivierungsfunktion mit MLPs eine beliebige Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ beliebig genau approximiert werden kann, was durch das *Universal Approximation Theorem* beschrieben ist. Die Schwierigkeit besteht jedoch darin, dass die zu approximierende Funktion lediglich durch Trainingsbeispiele gegeben ist und die Gewichte und Schwellwerte durch Training so angepasst werden müssen, dass neue Werte möglichst gut extrapoliert werden (Generalisierung) (vgl. Rojas, 1996, S. 24 ff.).

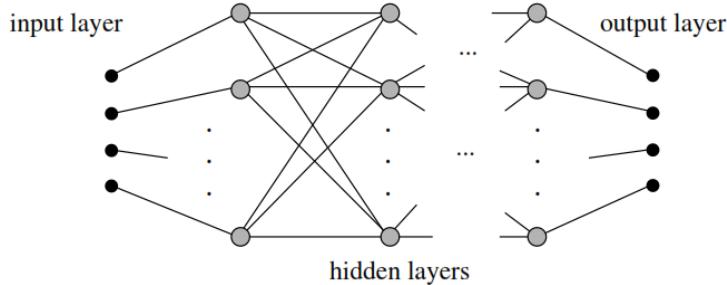


Abbildung 2.2: Generische mehrschichtige Architektur eines MLP (siehe Rojas (1996), S. 126)

Ein einfaches Beispiel soll an dieser Stelle die Verbindung zwischen MLP und *Deep Learning* herstellen. Die einfachste Architektur eines MLP umfasst die Eingabe x (Eingabeneuronen), ein *Hidden*-Neuron w_2 sowie ein Ausgabeneuron w_1 . Formal wird diese Architektur durch die Funktion in Gleichung 2.2 beschrieben, welche in unmittelbarer Verbindung zu den in Kapitel 1.1 beschriebenen tiefen Architekturen steht.

$$f(x) = \phi(\langle w_1, \phi(\langle w_2, x \rangle + b_2) \rangle + b_1) \quad (2.2)$$

Analog dazu fusionieren die Gewichtsvektoren w_i bei mehrere Neuronen in einer Schicht zu einer Matrix W in der die einzelnen w_i der Neuronen zeilenweise organisiert sind. Das innere Produkt in der Gleichung 2.2 wird so durch eine Matrix-Vektor-Multiplikation ersetzt und der skalare Schwellwert b_i mit einem Vektor b . Eine Schicht eines MLP kann folglich mit $\phi(Wx + b)$ formal dargestellt werden.

2.2 Aktivierungsfunktionen

Im letzten Kapitel wurde vorgestellt wie sich das Perzeptron zum Neuron und insgesamt zu einem *Multilayer Perceptron* (MLP) verallgemeinern lässt. In diesem Kapitel werden nun die wichtigsten Aktivierungsfunktionen für neuronale Netze vorgestellt (vgl. Hagan et al., 2014, S. 2-17).

Linear

Die einfachste Funktion ist die Identitätsfunktion. Diese ist in Abbildung 2.3 dargestellt.

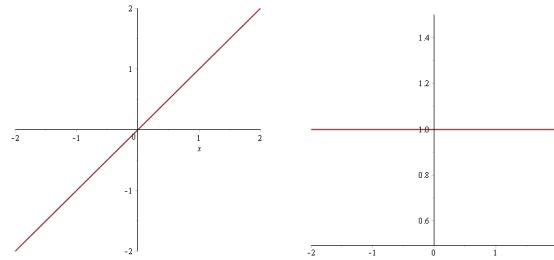


Abbildung 2.3: Lineare Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$f(x) = x \quad (2.3)$$

- Ableitung:

$$f'(x) = 1 \quad (2.4)$$

Logistische Sigmoidfunktion

Ein Spezialfall der Sigmoidfunktion ist die logistische Funktion. Diese nichtlineare Funktion bildet $\mathbb{R} \rightarrow (0, 1)$ ab (siehe Abbildung 2.4). Eine Besonderheit der Sigmoidfunktion ist, dass die Ableitung an einer bestimmten Stelle durch die Funktion selbst berechnet werden kann.

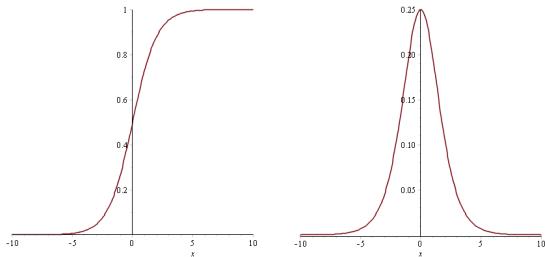


Abbildung 2.4: Sigmoid Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$\text{sig}(x) = \frac{1}{1 + \exp^{-t}} \quad (2.5)$$

- Ableitung:

$$\text{sig}'(x) = \text{sig}(x)(1 - \text{sig}(x)) \quad (2.6)$$

Tangens Hyperbolicus

Die Tangens Hyperbolicus-Funktion ist eine nichtlineare Funktion, welche $\mathbb{R} \rightarrow (-1, 1)$ abbildet (siehe Abbildung 2.5). Wie bei der Sigmoidfunktion, kann die Ableitung an einer bestimmten Stelle ebenfalls durch die Funktion selbst berechnet werden.

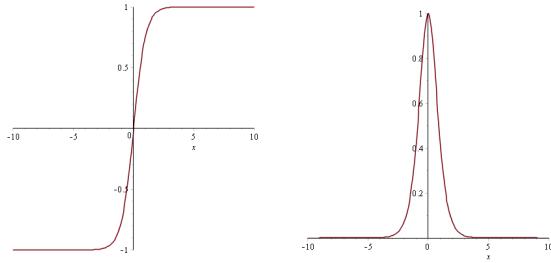


Abbildung 2.5: Tangens Hyperbolicus (tanh) Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$\tanh(x) = \frac{\exp^x - \exp^{-x}}{\exp^x + \exp^{-x}} \quad (2.7)$$

- Ableitung:

$$\tanh'(x) = 1 - \tanh(x)^2 \quad (2.8)$$

Rectified Linear

Die *Rectified Linear*-Funktion (ReLU) ist eine von Glorot et al. (2011) eingeführte Aktivierungsfunktion mit Eigenschaften, die im Bereich der Neuro-nalen Netze von Interesse sind. So ist bei dieser Aktivierungsfunktion für alle Werte, welche eine von 0 verschiedene Ausgabe erzeugen auch die Ableitung von 0 verschieden. Außerdem können keine negativen Werte auftreten. Die Funktion bildet $\mathbb{R} \rightarrow [0, \infty)$ ab und ist in Abbildung 2.6 dargestellt.

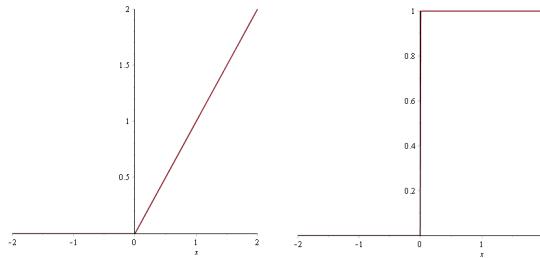


Abbildung 2.6: Rectified Linear (ReLU) Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$f(x) = \max(0, x) \quad (2.9)$$

- Ableitung:

$$f'(x) = \begin{cases} 0 & \text{für } 0 \geq x \\ 1 & \text{für } 0 < x \end{cases} \quad (2.10)$$

Softplus

Die *Softplus*-Funktion ist eine glatte Variante der ReLu-Funktion, welche $\mathbb{R} \rightarrow (0, \infty)$ abbildet. Die *Softplus*-Funktion ist in Abbildung 2.7 dargestellt.

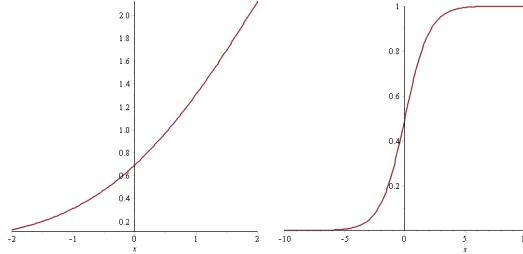


Abbildung 2.7: Softplus Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$f(x) = \ln(1 + e^x) \quad (2.11)$$

- Ableitung:

$$f'(x) = \frac{1}{1 + e^{-x}} \quad (2.12)$$

2.3 Fehlermaß

Feedforward-Netze basieren auf dem sogenannten *Error-Correction Learning*. Das bedeutet, dass das Ausgabesignal y_k eines Neuron k (*Output*) des Netzwerks zu einem Eingangssignal x mit einem Zielsignal d_k (*Target*) verglichen und so das Fehlersignal $e_k = d_k - y_k$ erzeugt wird (vgl. Haykin, 1998, S. 51f). Um das Fehlersignal in MLPs zu messen, wird ein entsprechendes Fehlermaß benötigt. Dabei werden meist der Mittlere Quadratische Fehler (*Mean Squared Error*) für Regression- und die negative Log-Likelihood (*Cross-Entropy*) für Klassifikationsaufgaben verwendet (vgl. Golik et al., 2013).

2.3.1 Mittlerer Quadratischer Fehler

Der *Mean Squared Error* (MSE) ist für ein gegebenes MLP $f(x)$ und gegebene Trainingsdaten $D = \{(x_1, y_1), \dots, (x_{n,n})\}$ wie in Gleichung 2.13 definiert. Eine Besonderheit stellt der zusätzliche Faktor $\frac{1}{2}$ dar, welcher letztlich aber nur die Ableitung vereinfacht.

$$MSE = \frac{1}{2N} \sum_i^N (f(x_i) - y_i)^2 \quad (2.13)$$

2.3.2 Cross-Entropy

Das *Cross-Entropy*-Fehlermaß (CE) ist in Gleichung 2.14 dargestellt. Es beschreibt die negative Log-Likelihood über den gesamten Trainingsdaten $D = \{(x_1, y_1), \dots (x_n, y_n)\}$ (vgl. Mitchell, 1997, S. 118). Als Vorgriff auf Kapitel 3.3 wird hier bereits CE über die Trainingsdaten gemittelt angegeben.

$$CE = -\frac{1}{N} \sum_i^N (y_i \ln(f(x_i)) + (1 - y_i) \ln(1 - f(x_i))) \quad (2.14)$$

Durch die $\ln(\cdot)$ -Funktion beachtet CE im Gegensatz zu MSE wie nah das Ausgabesignal am Zielsignal ist, vernachlässigt allerdings im Gegenzug fehlerbehaftete Ausgaben. Dieser Zusammenhang trägt dazu bei, dass CE für Klassifikationsaufgaben besser geeignet erscheint.

2.4 Backpropagation-Algorithmus

In den vergangenen Kapiteln wurden, ausgehend vom Perzeptron, die nichtlinearen MLPs eingeführt und verschiedene Aktivierungsfunktionen vorgestellt. Außerdem wurde gezeigt, dass der Fehler eines Neuronalen Netzes mittels zwei verschiedenen Fehlermaßen angegeben werden kann. Wie im vergangenen Kapitel festgestellt wurde, wird bei Neuronalen Netzen ein sogenanntes *Error-Correction Learning* angewandt. Dazu ist es nötig formal eine Regel zu definieren, mit welcher die Gewichte des Netzwerks verändert werden, um die Performance hinsichtlich des gewählten Fehlermaßes zu optimieren (Gradientenverfahren): Die sogenannte Delta-Regel (vgl. Widrow und Hoff (1960) und Widrow und Hoff (1988)).

Für das Anpassen der Gewichte und Schwellwerte in einem MLP mit nichtlinearen Aktivierungsfunktionen wird eine Verallgemeinerung, der von Rumelhart et al. (1986b) entwickelte Backpropagation-Algorithmus, verwendet. Dieser besteht abstrakt gesehen aus zwei Phasen. In der ersten Phase wird die Ausgabe des Netzes sowie der entsprechende Fehler berechnet (*Forward Pass*), in der zweiten der Fehler mittels Delta-Regel zurück propagiert und so der Gradient schichtweise durch Anwendung der Kettenregel berechnet (*Backward Pass*).

Im Folgenden wird der Backpropagation-Algorithmus vorgestellt (vgl. z.B. Rojas, 1996, S. 151 ff.). Beispielhaft dient hier der MSE als Fehlermaß.

- W^l : Gewichtsmatrix
- b^l : Vektor mit Schwellwerten
- $J(\cdot)$: Kostenfunktion
- x : Eingabevektor
- x^l : Layer-Eingabevektor
- $f(\cdot)$: Ausgabevektor
- $\phi(\cdot)$: Aktivierungsfunktion
- z^l : Linearer Ausgabevektor

- a^l : Aktivierter Ausgabevektor
- N : Anzahl Trainingsbeispiele
- L : Anzahl Layer
- \circ : Hadamard-Produkt
- η : Lernrate

Gleichung 2.15 definiert die Kostenfunktion:

$$J(W, b) = \frac{1}{2N} \sum_{i=1}^N (f(x_i) - y_i)^2 \quad (2.15)$$

Die Gleichungen 2.16 und 2.17 beschreiben den allgemeinen Gradientenabstieg:

$$W_{t+1}^l = W_t - \eta \nabla W_t \quad (2.16)$$

$$b_{t+1}^l = b_t - \eta \nabla b_t \quad (2.17)$$

Der Backpropagation-Algorithmus kombiniert die Delta-Regel mit der Kettenregel und beschreibt so ein Verfahren zur Berechnung der Ableitungen von zusammengesetzten Funktionen. Es wird somit der Gradient für jede Schicht des MLP iterativ berechnet. Um den Gradienten der Kostenfunktion $\nabla J(W, b)$ zu berechnen, wird zuerst für jedes Trainingsbeispiel $\nabla J(W, b, x_i, y_i)$ einzeln berechnet und aufsummiert. Folgende Formeln 2.18 und 2.19 beschreiben die Delta-Regeln des Backpropagation-Algorithmus:

Delta Output-Layer:

$$\delta^L = (a_i^L - y_i) \circ \phi'(z^L) \quad (2.18)$$

Delta für Hidden-Layer l :

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \circ \phi'(z^l) \quad (2.19)$$

Die partiellen Ableitungen pro Schicht (Gleichung 2.20 und 2.21) werden über alle Trainingsbeispiele $1..N$ aufsummiert und am Ende eines Durchlaufs mit $\frac{1}{N}$ gemittelt.

Partielle Ableitung für Layer l :

$$\frac{\partial J(W, b)}{\partial W^l} = \frac{1}{N} \sum_{i=1}^N \delta^l (x_i^l)^T \quad (2.20)$$

$$\frac{\partial J(W, b)}{\partial b^l} = \frac{1}{N} \sum_{i=1}^N \delta^l \quad (2.21)$$

Betrachtet man die Delta-Formeln 2.18 und 2.19, wird ersichtlich, dass sich die Verwendung einer Aktivierungsfunktion, deren Ableitung sich durch den Funktionswert selbst berechnen lässt, besonders eignet. Im Falle der Sigmoid-Funktion vereinfacht sich $\phi'(z^l)$ so zu $\text{sig}'(z^l) = \text{sig}(a^l)(1 - \text{sig}(a^l))$, was das Speichern von z^l erübriggt.

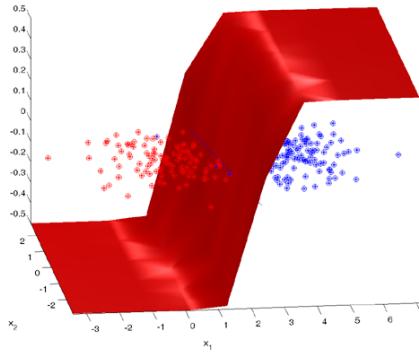


Abbildung 2.8: Logistische Regression mit zwei Klassen (Bild: Vadim Strijov, Computing Centre of the Russian Academy of Sciences)

2.5 Basisarchitekturen

Dieses Kapitel beschreibt die zwei Grundarchitekturen Neuronaler Netze (MLPs). Diese umfassen einerseits die Softmax-Regression zur Klassifikation und andererseits die Nichtlineare-Regression.

2.5.1 Softmax-Regression (Klassifikation)

Die Logistische Regression stellt ein bekanntes lineares statistisches Regressionsverfahren zur Modellierung diskreter abhängiger Variablen dar, welches ursprünglich nichts mit dem Neuronalen Lernmodell zu tun hat. Praktischerweise entspricht dieses jedoch der gleichen Architektur, was die Verwendung als Ausgabeschicht für die Klassifikation mit MLPs nahelegt. Abbildung 2.8 zeigt das Verfahren für die diskreten Werte 0 und 1, wobei die Hyperebene zur Visualisierung um 0.5 verschoben ist.

Allgemein wird mittels des Verfahrens, bei gegebenen Gewichten W , die Wahrscheinlichkeit einer binären Ausgabe $y_i \in \{0, 1\}$ für eine gegebene Eingabe x_i berechnet (vgl. z. B. Hastie et al., 2009, S. 119 ff.). Die Güte des Models über den gesamten Trainingsbeispielen N kann mit der Formel 2.22 bestimmt werden. Zur Vereinfachung wird von erweiterten Gewichtsvektoren \hat{w} ausgegangen, in denen der Schwellwert b integriert ist.

$$p(y|X, \hat{W}) = \prod_{i=1}^N \left[\frac{1}{1 + \exp^{-\hat{W}x_i}} \right]^{y_i} \left[1 - \frac{1}{1 + \exp^{-\hat{W}x_i}} \right]^{1-y_i} \quad (2.22)$$

Wird nun die Likelihood maximiert beziehungsweise die negative Log-Likelihood minimiert, erhält man die Formel 2.23 mit $f(x_i) = \frac{1}{1 + \exp^{-\hat{W}x_i}}$. Dies entspricht proportional einem einschichtigen MLP mit Sigmoid-Aktivierungsfunktion sowie *Cross-Entropy*-Fehlermaß und lässt sich somit mittels des Backpropagation-Algorithmus optimieren.

$$-\ln(p(y|X, \hat{W})) = -\sum_{i=1}^N y_i \ln(f(x_i)) + (1 - y_i) \ln(1 - f(x_i)) \quad (2.23)$$

Multinomiale Regression

Mehrdimensionale Ausgaben können in neuronalen Netzen durch einfaches Hinzufügen von Neuronen zum Output-Layer erzeugt werden. Ein gern verwendeter Code, um mehrere Klassen darzustellen, ist der sogenannte *Grandmother-Cell*-Code (vgl. LeCun et al., 1998a). Hier wird für d -Klassen ein d -dimensionaler Zielvektor oder erweiterter Labelvektor definiert mit $d_j = 1$ für die Klasse j . Das Modell der Logistischen Regression lässt sich ebenso leicht auf mehrere Klassen erweitern. Dazu muss die Ausgabe, um ein valides probabilistisches Modell zu erhalten, normalisiert werden. Gleichung 2.24 beschreibt die Kostenfunktion der multinomialen Regression. Diese ist im Bereich Neuronale Netze eher bekannt als Softmax-Regression (vgl. z.B. Krizhevsky et al. (2012) und Bengio et al. (2015)).

$$-\ln(p(y|X, \hat{W})) = -\sum_{i=1}^N \sum_{j=1}^d 1\{y^i = j\} \ln\left[\frac{\exp^{\hat{w}_j^T x^i}}{\sum_{l=1}^d \exp^{\hat{w}_l^T x^i}}\right] \quad (2.24)$$

Neben der Softmax-Regression wird teilweise auch ein Output-Layer mit radialen Basisfunktionen (RBFs) verwendet (vgl. LeCun et al., 1998a). Dies hat den Vorteil, dass für jede Klasse ein eigener spezifischer Labelvektor definiert werden kann. Ein Nachteil ist jedoch, dass diese Variante die Ausgabe nicht normalisiert und somit nicht probabilistisch interpretiert werden kann. Als Basisfunktionen im Hidden-Layer eignen sich RBFs nur sehr bedingt, da diese lokal im Eingaberaum sind und daher sehr viele Einheiten für hochdimensionale Räume benötigt werden (vgl. LeCun et al., 1998b).

Besonderheit in Verbindung mit Backpropagation

Wird Backpropagation angewandt, muss im Output-Layer die Ableitung der Aktivierungsfunktion berechnet werden. Gleichung 2.25 zeigt diese für die Softmax-Funktion. δ entspricht hierbei dem Kronecker-Delta.

$$\text{softmax}'(x)_{ij} = \text{softmax}(x)_i(\delta_{ij} - \text{softmax}(x)_j) \quad (2.25)$$

Dieser Ausdruck ist sehr unhandlich. Es lässt sich allerdings zeigen, dass die Softmax-Regression in Verbindung mit einem *Cross-Entropy*-Fehlermaß die Berechnung von δ^L nicht erschwert, sondern zu Gleichung 2.26 vereinfacht (vgl. z. B. Bengio et al., 2015, Kap. 6.3.2, S. 167).

Delta Output-Layer:

$$\delta^L = a_i^L - y_i \quad (2.26)$$

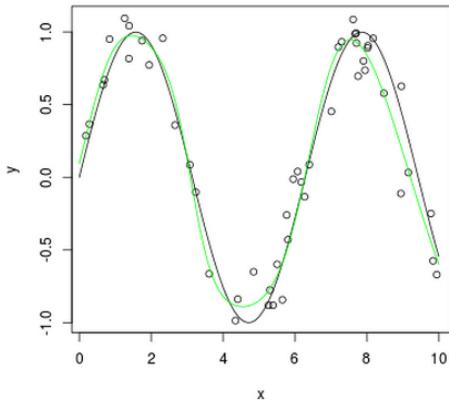


Abbildung 2.9: Approximierte Sinuskurve mit einem Hidden-Layer mit 6 Neuronen (grün)

Dies bietet damit den großen Vorteil, dass δ^L nur noch vom Fehler abhängt und nicht mehr von der Ableitung der Aktivierungsfunktion und damit die Nichtlinearität am Ausgang nicht gesättigt wird.

2.5.2 Funktionsapproximation (Regression)

Neuronale Netze können beliebige Funktionen $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ approximieren und so mittels Backpropagation auf beliebige Daten trainiert werden (vgl. Rojas, 1996, S. 269 ff.). Es ist lediglich darauf zu achten, dass der Wertebereich des Output-Layers zu dem der Labels passt. So werden im Output-Layer meist lineare Aktivierungsfunktionen verwendet. Alternativ können auch die Labels auf den Wertebereich der verwendeten Aktivierungsfunktion skaliert werden.

Abbildung 2.9 zeigt eine Sinuskurve, welche von einem MLP mit 6 Neuronen im Hidden-Layer approximiert wird. Die zugrundeliegende Trainingsdaten sind aus einer Sinuskurve mit additivem normalverteilten Rauschen generiert.

2.6 Convolutional Neural Networks

Bildklassifikation, Objektlokalisierung und Objekterkennung rücken derzeit immer mehr in den Fokus. Dies ist beispielsweise an einer immer größeren Verbreitung von Drohnen und autonomen Fahrzeugen zu erkennen ist. Diese Entwicklung geht einher mit den Fortschritten im Bereich Neuronale Netze und den Möglichkeiten des *Deep Learning*. Neuere Arbeiten aus den Neurowissenschaften stützen darüber hinaus getroffene Annahmen mit realen Beobachtungen an Lebewesen, was die Entwicklung weiter unterstützt. In diesem Zusammenhang wird gerne das berühmte *Halle Berry*-Neuron im menschlichen Temporallappen genannt, welches Halle Berry erkennt. Dies deutet darauf hin, dass das biologischen Modell des Sehens auf einem invarianten, dünnbesetzten (*spare*), expliziten Code basiert (vgl. Quiroga et al., 2005).

Dieses Kapitel behandelt eine spezielle Klasse Neuronaler Netze, die das biologische Modell des Sehens und Hörens versuchen zu imitieren. Die sogenannten *Convolutional Neural Networks*.

2.6.1 Modellbeschreibung

Convolutional Neural Networks (CNNs) sind eine Erweiterung des *Multi Layer Perceptrons* (MLPs), welche den biologischen visuellen Cortex zu imitieren versuchen. Die Idee selbst beruht auf den Arbeiten von Hubel und Wiesel (1962) bezüglich des Sehempfindens von Katzen, die zeigen, dass der visuelle Cortex aus komplex angeordneten Zellen besteht, wobei jede einzelne Zelle einen kleinen Bereich des Sichtfeldes abdeckt. Diese Architektur erlaubt es, räumlich voneinander getrennte Muster zu erfassen.

Im künstlichen Modell setzt sich ein CNN aus mehreren Convolution- und Pooling-Layern sowie einem klassischen MLP zusammen. Dieses spezielle Design erlaubt es, eine 2D-Struktur im Input-Layer zu erfassen. Dies wird durch sogenannte lokale Verbindungen (*Local Connections*) erzielt.¹. Das erste künstliche Modell dieser Art ist das *NeoCognitron* von Fukushima (1980), welches zwei verschiedene Arten lokaler rezeptiver Felder (siehe Abbildung 2.10) definiert: Eines zur Detektion von Kanten und eines mit lokaler Invarianz hinsichtlich Translation.

¹Auch wenn die Ursprünge des Modells im Bereich der visuellen Wahrnehmung liegen, wird es heute beispielsweise auch für Spracherkennung und NLP verwendet (vgl. dazu Socher et al. (2011) und Sainatha et al. (2015))

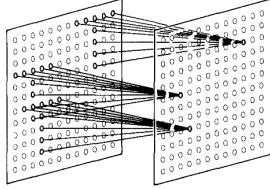


Abbildung 2.10: Lokales rezeptives Feld (siehe Fukushima (1980))

Das *LeNet* von LeCun et al. (1989) ist die Weiterentwicklung dieses Models. Hier teilen sich mehrere rezeptive Felder dieselben Gewichte beziehungsweise Parameter (*Parameter Sharing*), was die Verbindungen und dadurch auch die Anzahl der Gewichte reduziert. Die Anzahl ist damit von der Dimensionalität der Eingabe entkoppelt. Algebraisch entspricht dies einer Faltung (*Convolution*), wovon das Model seinen Namen ableitet. Diese Architektur ist die Grundlage heutiger CNNs und definiert die damit verbundenen Eigenschaften (vgl. LeCun et al. (1998a), Bengio und LeCun (2007) und Zeiler und Fergus (2014)):

- Lokale Extraktion von Merkmalen (*Local Feature Extraction*)
- Translationsinvarianz hinsichtlich Eingabedaten (geringe Skalierungs- und Rotationsinvarianz)
- Einfacheres Training durch weniger Parameter und Verbindungen als MLP mit gleich vielen Neuronen
- Nicht-lokale Generalisierung durch Verschachtlung nichtlinearer Funktionen

Abbildung 2.11 stellt ein CNN mit 8 Schichten zur Klassifikation von 32×32 Pixel großen Eingabebildern dar. Die ersten 6 Schichten zählen zum CNN, während die letzten beiden Schichten ein klassisches MLP repräsentieren. Der zugrundeliegende Ansatz ist hierbei, mittels CNN Merkmale (*Features*) derart zu extrahieren, sodass die Klassen am Ende möglichst gut trennbar sind.

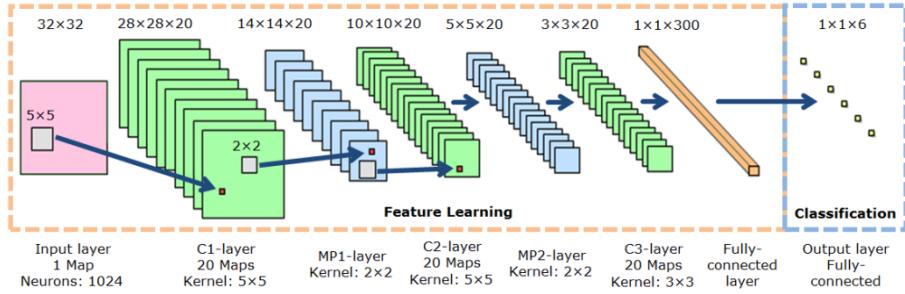


Abbildung 2.11: Convolutional Neural Network mit 8 Schichten (siehe Nagi et al., 2011)

Aus Gründen der Vereinfachung werden im weiteren Verlauf des Kapitels lediglich quadratische Eingaben sowie quadratische Filtermasken betrachtet.

Convolution-Layer

Der Convolution-Layer ist einer der beiden zentralen Bestandteile eines CNN (vgl. hierzu und im Folgenden LeCun et al., 1998a). Dieser transformiert eine 3D-Eingabe x mit Tiefe m , bezeichnet als *m-Input-Maps*, zu einer 3D-Ausgabe a . Die Tiefe n der Ausgabe wird bestimmt durch die Anzahl an Faltungskernen und selbst meist als *n-Feature-Maps* bezeichnet. Ein Faltungskern i ist in Abbildung 2.12 dargestellt. Er entspricht ebenfalls einer 3D-Struktur, welche durch Höhe k_h und Breite k_w der Faltungsmaske sowie der Tiefe k_d bestimmt ist. Die Tiefe k_d muss der Tiefe der 3D-Eingabe m entsprechen. Es gibt somit gleich viele Faltungsmasken pro *Feature-Map* wie es *Input-Maps* gibt: Es gilt $k_d = m$. Alle Faltungskerne zusammen werden kompakt mit W bezeichnet und einzelne Filtermasken mit W_{ij} eindeutig indiziert. Wird nun die Netz-Ausgabe (*Forward Pass*) für eine *Feature-Map* i berechnet, muss für jede *Input-Map* x_j die Filtermaske W_{ij} aus dem aktuellen Faltungskern gezogen und die Ausgabe z_{ij} berechnet werden. Dies geschieht mittels diskreter Faltung. Die einzelnen Ausgaben z_{ij} werden über alle *Input-Maps* $1..m$ zu z_i aufsummiert.

Die 3D-Ausgabe a kann auch als Neuronen-Ausgabe interpretiert werden, in welcher jedes Element der Aktivierung eines Neurons entspricht. Bei der Berechnung eines a_i wird folglich zu z_i zunächst ein *Bias* b_i addiert und anschließend die Aktivierungsfunktion $\phi(\cdot)$ berechnet. Dies geschieht elementweise und analog für jede der n *Feature-Maps*.

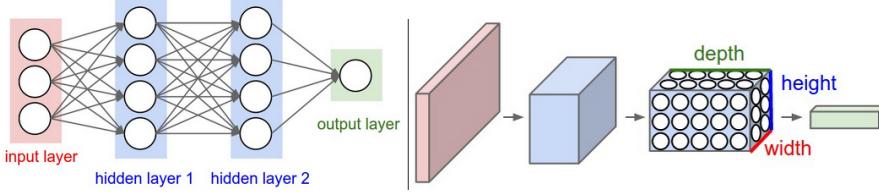


Abbildung 2.12: Jeder Convolution-Layer transformiert eine 3D-Eingabe (z.B. RGB-Bild) in eine 3D-Ausgabe (siehe Fei-Fei und Karpathy (2014))

Folgende Hyperparameter definieren einen Convolution-Layer:

- *Feature-Maps*: n
- *Input-Maps*: $m = k_d$
- Größe Faltungskern: $k_w = k_h$

Formal berechnet der Convolution-Layer für jede der n *Feature-Maps* zuerst die 2D-Faltung (Faltungsoperator: $*$) der $N \times N$ -großen m *Input-Maps* mit der Filtermaske W_{ij} (Gleichung 2.27) und im Anschluss elementweise die Aktivierungsfunktion $\phi(\cdot)$. Das Resultat ist die *Feature-Map* a_i (Gleichung 2.28).

$$z_i = \sum_{j=0}^m x_j * W_{ij} \quad (2.27)$$

$$a_i = \phi(z_i + b_i) \quad (2.28)$$

Durch die Randbehandlung *valid* reduziert sich die Größe der *Feature-Maps* auf $(N - k_w + 1) \times (N - k_w + 1)$.²

Die Anzahl der Gewichte berechnet sich aus $n \cdot k_d \cdot k_w \cdot k_w$. Die Anzahl Pixel in der 3D-Ausgabe entspricht der Menge an Neuronen. Im Rahmen des Trainings gilt es, diese Gewichte zu trainieren. Die exakte mathematische Berechnung von Delta δ und den partiellen Ableitungen $\frac{\partial J(W, b; x, y)}{\partial W_{ij}^l}$ und $\frac{\partial J(W, b; x, y)}{\partial b_i^l}$ für das Training mit Backpropagation folgt in Kapitel 2.6.2.

Pooling-Layer

Der zweite wichtige Bestandteil eines CNN ist das sogenannte *Pooling*. LeCun et al. (1998a) bezeichnen diesen Vorgang als *Subsampling*, wobei heute der Begriff *Pooling* eher geläufig ist (vgl. Zeiler und Fergus (2013) oder Glorot

²Es existieren bei der Faltung mehrere Modi zur Randbehandlung. Modus *valid* berechnet nur die tatsächlich vorhandenen Elemente, wodurch die Ausgabe in der Größe schrumpft. Im Unterschied dazu berechnet der Modus *full* die Faltung über eine mit Padding versehene Eingabe, wodurch sich die Ausgabe entsprechend vergrößert. Der Modus *same* erzeugt eine gleich große Ausgabe.

et al. (2011)). Allgemein berechnet der Pooling-Layer ein Downsampling, wobei dessen Art nicht fest definiert ist. Abbildung 2.13 stellt das sogenannte Max-Pooling dar (vgl. Glorot et al. (2011) und Zeiler et al. (2011)). Jede der m *Input-Map* x_j der 3D-Eingabe wird vom Pooling-Layer einzeln bearbeitet und ausgegeben. Die Anzahl m ändert sich nicht und es werden folglich m *Feature-Maps* erzeugt. Für jedes x_j wird nun ein Bereich der Größe $k_h \times k_w$, wobei k_h der Filterhöhe und k_w der Filterbreite entspricht, ausgeschnitten und verarbeitet. Im Falle des Max-Pooling wird das Maximum berechnet und in der Ausgabe gespeichert.

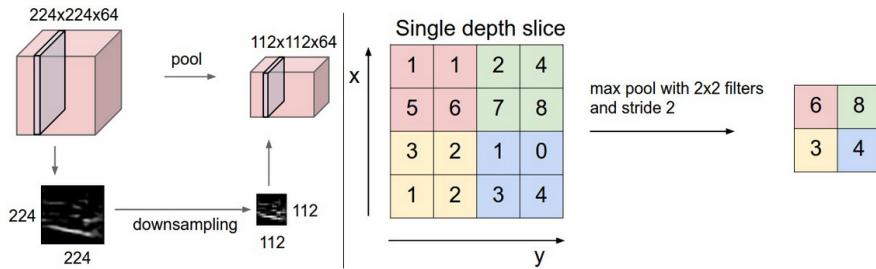


Abbildung 2.13: Ein Pooling-Layer transformiert eine 3D-Eingabe (z.B. RGB-Bild) in eine kleinere 3D-Ausgabe (Downsampling) (siehe Fei-Fei und Karpathy (2014))

Folgende Hyperparameter sind damit für einen Pooling-Layer notwendig:

- Pooling-Methode
- Filtergröße: $k_w = k_h$

Formal teilt der Max-Pooling-Layer jede *Input-Map* x_j in $k_w \times k_w$ disjunkte Bereiche und wählt in jedem Bereich das Maximum. Dadurch reduziert sich die Ausgabe auf $\frac{N}{k_w} \times \frac{N}{k_w}$.

Alternative, ebenfalls verbreitete Formen von Pooling stellen das *Average-Pooling* (vgl. LeCun et al., 1998a) oder das L_p -Pooling (vgl. Pierre Sermanet und LeCun, 2012) dar. Die Besonderheit beim Pooling-Layer sind die fehlende Aktivierungsfunktion sowie Gewichte und Schwellwerte, weshalb Convolution- und Pooling-Layer auch zusammen in einer Schicht berechnet werden können (vgl. Simard et al., 2003). Die exakte mathematische Beschreibung von Delta δ für das Training mit Backpropagation folgt in Kapitel 2.6.2.

2.6.2 Training mit Backpropagation

Im letzten Teil des Kapitels wurde bereits das Modell eines CNN und dessen elementaren Bestandteile beschrieben. Außerdem wurden die enthaltenen Parameter vorgestellt. Das Ziel dieses Abschnitts ist es zu beschreiben,

wie die beiden neuen Schichten mittels Backpropagation trainiert werden können (vgl. z.B. Bouvrie, 2006). Wie in Kapitel 2.4 beschrieben, ist für Backpropagation (Backward Pass) pro Trainingsbeispiel die Berechnung dreier Größen notwendig: δ^l sowie $\frac{\partial J(W, b; x_i, y_i)}{\partial W_{ij}^l}$ und $\frac{\partial J(W, b; x_i, y_i)}{\partial b_i^l}$.

Um die Formeln zu vereinfachen, werden an dieser Stelle sogenannte *Messages* eingeführt. Dies erleichtert die Notation insofern, dass die Berechnung von δ^l nicht mehr von den Gewichten der vorherigen Schicht W^{l+1} abhängt. Delta $\delta_{message}^l$ bezeichnet so den über die Schicht hinaus zurück propagierten Fehler. Im Falle von gewöhnlichen MLPs gilt $\delta_{message}^{l+1} = (W^{l+1})^T \delta^{l+1}$. Für Convolutional-Layer wird das Delta δ_i^l korrespondierend zu jeder einzelnen der *n-Feature-Maps* aus dem eingehenden $\delta_{message_j}^{l+1}$ berechnet. An dieser Stelle sei nochmals betont, dass die Anzahl *Feature-Maps* n der Anzahl *Input-Maps* m der nächsten Schicht entspricht. Es gilt folglich $n^l = m^{l+1}$. Im Folgenden werden zuerst die notwendigen Formeln für Pooling-Layer und im Anschluss die der Convolution-Layer vorgestellt.

Formel 2.29 berechnet das Delta $\delta_{message_j}^l$ im Pooling-Layer mittels Kronecker-Produkt $kron(\cdot)$. Da der Pooling-Layer keine Parameter besitzt ist die Berechnung von δ_i^l irrelevant.

Delta Average-Pooling-Layer:

$$\delta_{message_j}^l = kron(delta_{message_j}^{l+1}, ones(k_w, k_w)) \circ \frac{1}{k_w^2} \quad (2.29)$$

Je nach Pooling-Art muss hier unterschiedlich vorgegangen werden. Für einen Max-Pooling-Layer müssen im *Forward Pass* die Positionen der Maxima gespeichert und bei der Rückpropagierung (Backward Pass) der Fehler an die gespeicherten Positionen geschrieben werden. Im Lp-Pooling muss der Fehler entsprechend der Norm p und der verwendeten Filtermaske aufgeteilt werden.

Delta Convolution-Layer:

Für jede *Feature-Map* i im Convolution-Layer wird das zugehörige Delta δ_i^l mit Formel 2.30 berechnet.

$$\delta_i^l = \delta_{message_j}^{l+1} \circ \phi'(z_i) \quad (2.30)$$

Jedes $\delta_{message_j}^l$ für die Rückpropagierung des Fehlers wird mit Formel 2.31 berechnet. Die Funktion $rot180(\cdot)$ bezeichnet das Drehen einer Matrix um 180° beziehungsweise das Vertauschen beider Axen (*Flipping*).

$$\delta_{message_j}^l = \sum_{i=0}^n \delta_i^l * rot180(W_{ij}) \quad (2.31)$$

Durch die Randbehandlung *full* erhöht sich die Größe des Delta $\delta_{message_j}^l$ von $(N - k_w + 1) \times (N - k_w + 1)$ wieder auf $N \times N$.

Gradient Convolution-Layer:

Der Gradient für den Convolution-Layer kann mittels Formel 2.32 und 2.33 aus der Eingabe x und Delta δ^l berechnet werden. Die Randbehandlung *valid* ist anzuwenden.

$$\frac{\partial J(W, b)}{\partial W_{ij}^l} = \frac{1}{N} \sum_{e=1}^N \text{rot180}(x_{ej}^l * \text{rot180}(\delta_{ei}^l)) \quad (2.32)$$

$$\frac{\partial J(W, b)}{\partial b_i^l} = \frac{1}{N} \sum_{e=1}^N \sum_{u=0}^{k_w} \sum_{v=0}^{k_w} \delta_{eiuv}^l \quad (2.33)$$

Analog zum Vorgehen bei MLPs wird der Gradient (Gleichung 2.32 und 2.33) ebenfalls über alle Trainingsbeispiele $1..N$ aufsummiert und am Ende eines Durchlaufs mit $\frac{1}{N}$ gemittelt. Außerdem kann bei Aktivierungsfunktionen, deren Ableitung sich durch den Funktionswert selbst darstellen lässt, auf die Speicherung von z verzichtet werden.

Kapitel 3

Spezielle Methoden im Deep Learning

Bereits zu Beginn (siehe Kapitel 1.1), wie auch im vergangenen Kapitel 2.6.1 wurde darauf hingewiesen, dass Convolutional Neural Networks (CNNs) allgemeinen Multilayer Perceptrons (MLPs) in zwei zentralen Punkten überlegen sind. So reduzieren sie das Problem des *Overfittings*, wobei es sich um die Überanpassung des Modells an die Trainingsdaten mit dem Resultat schlechter Generalisierung auf neue unbekannte Daten handelt. Dies geschieht durch Reduktion der zu lernenden Gewichte und Verbindungen. Des Weiteren erlauben CNNs die lokale Extraktion von Merkmalen durch Einbezug räumlicher Korrelationen im Eingaberaum (vgl. LeCun et al. (1998a)).

Auch durch die Verwendung von CNNs bleiben dennoch zentrale Schwierigkeiten im Deep Learning bestehen:

- Verschwinden des Gradienten (*Vanishing Gradient*) in tiefen Netzen (vgl. Hochreiter (1991))
- Gradientenabstieg bei nicht-konvexer Zielfunktion (vgl. Martens (2010) und Dauphin et al. (2014))
- *Overfitting* bei großen Netzen, insbesondere in nachgeschalteten MLPs (vgl. Hinton et al. (2012))
- Schlechte Konditionierung der Fehlerlandschaft aufgrund von *Parameter Sharing* (vgl. LeCun et al. (1998a))

Historisch betrachtet liefert die Arbeit von Hochreiter (1991) die Grundlage zu dem Problem, das heute als *Vanishing Gradient*-Effekt bekannt ist. Dieser beschreibt den Zusammenhang zwischen der abnehmenden Größe des Fehlersignals und der Tiefe des Netzwerks. Das heißt, je mehr Hidden-Layer ein Netz besitzt, desto kleiner wird der Gradient in den vorderen Schichten und desto langsamer lernen diese Schichten in der Folge (siehe Abbildung 3.1).

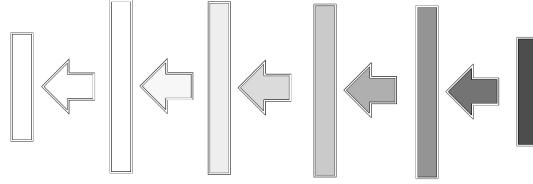


Abbildung 3.1: Der *Vanishing Gradient*-Effekt beschreibt bei der Rückpropagierung das abklingende Fehlersignal von Output- hin zu Input-Layer

Deutlich wird der Effekt bei der Betrachtung der Formel 3.1 für das zurück propagierte Fehlersignal $\delta_{message}^l$ im Layer l , in der aus Kapitel 2.6.2 bekannten $\delta_{message}$ -Notation.

$$\delta_{message}^l = (W^l)^T (\delta_{message}^{l+1} \circ \phi'(z^l)) \quad (3.1)$$

Setzt man klassische Aktivierungsfunktionen wie $sig(\cdot)$ oder $tanh(\cdot)$ ein (siehe Kapitel 2.2), gilt $\phi'(z^l) \leq 1$. Dies führt zwangsläufig zu einem abklingenden Fehlersignal (*Vanishing Gradient*). Man könnte argumentieren, dass dies durch ein großes W^l , sodass $(W^l)^T (\delta_{message}^{l+1} \circ \phi'(z^l)) \geq 1$, verhindert werden könnte. Dem ist nicht der Fall, da gleichzeitig $z^l = W^l x^l + b^l$ gilt und damit ein großes W^l unweigerlich zu einem großen z^l und damit zu einer kleineren Ableitung der Aktivierungsfunktion, welche ihr Maximum bei 0 besitzt, führt. Ist zusätzlich zu einem großen W^l der Schwellwert b^l negativ, sodass $z^l \approx 0$, führt dies ebenso nicht zum Verschwinden des Effekts sondern zu einem, nicht weniger ungünstigen, *Exploding Gradient*-Effekt (vgl. Nielsen, 2015, Kapitel 5)). Selbst im linearen Fall ohne Aktivierungsfunktion tritt dieser Effekt auf, wenn die Gewichte jedes Neurons nicht die Norm $\|w\| \approx 1$ besitzen.

Eine normalisierte Initialisierung der Gewichte wirkt diesem Effekt entgegen (vgl. Glorot und Bengio (2010)). Allerdings scheint erst die Verwendung der neuartigen ReLu-Aktivierungsfunktion (vgl. Kapitel 2.2) den Effekt gänzlich zu unterdrücken und führt darüber hinaus zu oftmals erwünschten, dünnbesetzten (*sparse*) Aktivierungen (vgl. Glorot et al. (2011)). Der Nachteil von ReLu-Neuronen ist allerdings, dass diese bei zu großen Lernraten und damit zu großen Änderungen der Gewichte unwiderruflich sterben (*Dying ReLu*) und somit Teile des Netzes auslöschen können (vgl. Maas et al. (2013)).

$$f(x) = \max(0, x) \quad (3.2)$$

Viele der in diesem Kapitel vorgestellten Methoden zielen auf den *Vanishing Gradient*-Effekt ab, welcher für viele Probleme im Deep-Learning verantwortlich ist. Darüber hinaus zeigt Martens (2010) empirisch, dass die Schwierigkeiten im Deep-Learning auch auf eine ungünstige Fehlerlandschaft (*Pathological Curvature*) zurückzuführen sind. Eine solche ist in Form der Rosenbrock-Funktion in Abbildung 3.2 dargestellt.

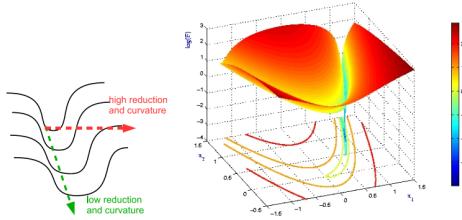


Abbildung 3.2: Pathological Curvature in der Rosenbrock-Funktion: $f(x, y = (1 - x)^2 + 100(y - x^2)^2$ (vgl. Martens (2010))

Neben den benannten Problemen hinsichtlich der Fehlerfunktion, führt *Overfitting* häufig ebenso zu schlechten Ergebnissen und entsprechende Regularisierungsmethoden werden benötigt.

3.1 Vorverarbeitung

Der Bereich Vorverarbeitung steht nicht unmittelbar in Zusammenhang mit Deep Learning, wird jedoch zwecks der Vollständigkeit, besonders in Verbindung zu CNNs, vorgestellt. Becker (1991) stellte bereits Anfang der 1990er Jahre fest, dass durch Dekorrelation der Trainingsdaten MLPs effizienter trainiert werden können. LeCun et al. (1998b) fassen die klassische Vorverarbeitung in linearen MLPs wie folgt zusammen:

- Mittelwertfreie Trainingsdaten verhindern eine steile Fehlerlandschaft.
- Die Normalisierung der Varianz unterschiedlicher Merkmale verhindert deren unterschiedliche Gewichtung.
- Die Dekorrelation der Trainingsdaten führt zwar zu einer diagonalen Hesse-Matrix, allerdings zeigt der Gradient nicht in Richtung Minimum. Dies muss durch dedizierte Lernraten, entsprechend den Kehrwerten der Eigenwerte pro Gewicht, korrigiert werden.
- Das Whitening der Daten führt zu einer kreisförmigen Fehlerlandschaft mit korrektem Gradienten.

Diese Hinweise gelten nur für lineare MLPs und somit quadratische Fehlerlandschaften. Die Fehlerlandschaft der hier verwendeten nicht-linearen MLPs kann jedoch lokal quadratisch approximiert werden (vgl. Hinton et al. (2015)).

Die genannten Techniken können allerdings nicht unmittelbar auf CNNs übertragen werden, da diese versuchen lokale Korrelationen an verschiedenen Orten im Eingaberaum zu extrahieren und die Dekorrelation hochdimensionaler Daten in großer Anzahl sehr rechenintensiv ist. So zeigt sich bei der Verwendung von CNNs, dass die erfolgreichsten Architekturen lediglich

den Mittelwert über die gesamten Trainingsdaten berechnen und diesen von jedem Pixel subtrahieren (vgl. Krizhevsky et al. (2012) und Simonyan und Zisserman (2014)). Fei-Fei und Karpathy (2014) kommen zu einem ähnlichen Schluss und beschreiben für CNNs lediglich die Zentrierung der Daten mittels globalen Mittelwert, alternativ pro Farbkanal oder Pixel, als nötige Vorverarbeitung.

Teilweise werden für CNNs dennoch zwei weitere Techniken als Praxis beschrieben (vgl. Andrade (2014) und Goodfellow et al. (2013b)) und deshalb im Folgenden kurz eingeführt.

3.1.1 Kontrastnormalisierung

Das Ziel der Kontrastnormalisierung ist es, unterschiedliche Trainingsbeispiele desselben Objekts in einen ähnlichen Kontrastbereich zu transformieren und so Helligkeitsunterschiede zu kompensieren (vgl. Zeiler und Fergus (2013)). Die globale Kontrastnormalisierung (GCN) bearbeitet jedes Trainingsbeispiel einzeln und berechnet den Mittelwert und die Varianz über alle Merkmale beziehungsweise Pixel. Im Anschluss wird der Mittelwert subtrahiert und durch die Varianz geteilt (vgl. Pierre Sermanet und LeCun, 2012). Neben der GCN wird häufig auch die lokale Kontrastnormalisierung (LCN) angewandt. Diese nimmt pro Trainingsbeispiel als Eingabe jeweils nur kleine Umgebungen $k_h \times k_w$ und berechnet die Kontrastnormalisierung für diese Bereiche einzeln (vgl. Jarrett et al., 2009).

Zeiler und Fergus (2013) verwenden für Farbbilder den RGB-Raum und normalisieren jeden Kanal einzeln. Dies kann jedoch zu einer Verschiebung des Farbtone führen. Soll dies vermieden werden, wird für Farbbilder entweder der HSV-Raum vorgeschlagen, in welchen nur die Intensität V normalisiert wird (vgl. Pink, 2011, S. 56). Oder es wird der YUV-Farbraum verwendet und die Normalisierung lediglich auf den Y-Kanal angewandt (vgl. Pierre Sermanet und LeCun, 2012).

3.1.2 ZCA-Whitening

Der Begriff *Zero Component Analysis* (ZCA) beschreibt ein der *Principle Component Analysis* (PCA) sehr ähnliches Verfahren (vgl. hierzu und im Folgenden Krizhevsky und Hinton (2009)). Es zielt darauf ab, die Eingangsdaten so zu dekorrelieren, sodass die Transformation so nahe wie möglich am Original ist.

Die Transformationsmatrix W_{ZCA} ist in Gleichung 3.3 angegeben, wobei die Kovarianzmatrix $C = X^T X$ ist. Der einzige Unterschied zum PCA-Whitening liegt darin, dass eine weitere Rotation zurück in den Bildraum durchgeführt wird. Dies ist möglich, da das Whitening der Daten auch nach einer Rotation mit einer orthogonalen Matrix erhalten bleibt.

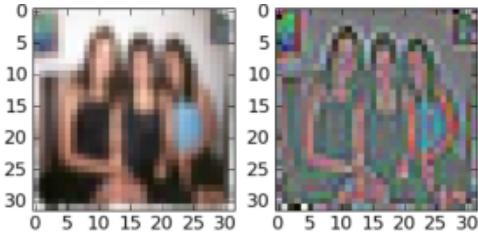


Abbildung 3.3: Originales Bild (links) und ZCA-transformiertes Bild (rechts) (siehe. Krizhevsky und Hinton (2009))

$$W_{ZCA} = C^{-\frac{1}{2}} = P(D + \epsilon)^{-\frac{1}{2}}P^T = PW_{PCA} \quad (3.3)$$

Abbildung 3.3 zeigt die Transformation für ein Beispiel.

3.2 Initialisierung

Die Initialisierung eines neuronalen Netzes ist äußerst wichtig, da die Gewichte ungleich Null sein müssen (*Breaking the Symmetry*). Ansonsten berechnen alle Neuronen die gleiche Ausgabe und es ergeben sich somit die gleichen partiellen Ableitungen (vgl. Rojas, 1996, S. 201). LeCun et al. (1989) initialisieren die Gewichte beispielsweise zufällig zwischen $-\frac{2.4}{fan_{in}}$ und $\frac{2.4}{fan_{in}}$, wobei fan_{in} der Anzahl Eingangsneuronen entspricht. Das hat zum Ziel die Nichtlinearität mit Werten um Null zu versorgen und damit nicht zu sättigen (*Vanishing Gradient*). Das erfolgreiche Netz von Krizhevsky et al. (2012) verwendet beispielsweise die Standard-Initialisierung, eine einfache Normalverteilung mit $W \sim \mathcal{N}(0, 0.01)$. Dies ist möglich, da als Aktivierungsfunktionen ReLu-Funktionen verwendet werden, welche nicht sättigen können. Daneben existieren im Deep Learning heute zwei Arten zur Initialisierung, welche im Folgenden aufgeführt sind.

3.2.1 Normalisierte Fehlerpropagierung

Stellvertretend für eine derartige Initialisierung der Gewichte, sodass die Varianz des Signals sowohl bei der Berechnung der Ausgabe (*Forward Pass*), als auch bei der Rückpropagierung des Fehlers (*Backward Pass*) gleich groß bleibt (vgl. z.B. Rojas, 1996, S. 199 ff.), steht heute die sogenannte *Xavier-Initialization*. Dies führt dazu, dass die Varianz im Gradienten in jeder Schicht in etwa gleich groß ist, was aus Abbildung 3.4 zu entnehmen ist (vgl. hierzu und im Folgenden Glorot und Bengio (2010)). Die *Xavier-Initialization* kombiniert beide Aspekte und initialisiert die Gewichte in der Art, dass die Varianz des Signals von Schicht zu Schicht sowohl im *Forward Pass* als auch im *Backward Pass* erhalten bleibt. Dies wird durch Formel 3.4 erreicht,

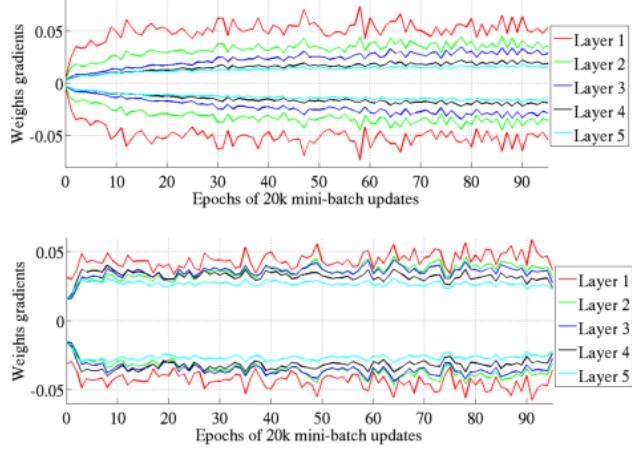


Abbildung 3.4: Vergleich der Varianz der Gradienten im Laufe des Trainings (Standard-Initialisierung (oben) und Xavier-Initialisierung (unten)): *Layer 1* entspricht dem Output-Layer mit der größten Varianz im Gradient (siehe Glorot und Bengio (2010))

was letztlich der Anwendung der Varianz-Formel für Gleichverteilung mit $Var(W) = \frac{2}{fan_{in} + fan_{out}}$ entspricht.

$$W \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}\right] \quad (3.4)$$

Für den Fall, dass ReLu-Funktionen eingesetzt werden, verallgemeinern He et al. (2015) diese Formel insofern, dass eine lineare Approximation um Null für diese und davon abgeleitete Aktivierungsfunktionen nicht mehr gültig ist. Formel 3.5 entspricht der Verallgemeinerung für Funktionen dieser Art.

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{fan_{out}}}\right) \quad (3.5)$$

Mit der Xavier-Initialisierung werden in Folge häufig bessere Ergebnisse erreicht, als mit der Standard-Initialisierung (*Random Initialization*) in Verbindung mit Informationen über die Krümmung (Approximation der Hesse-Matrix - siehe Kapitel 3.3.2) (vgl. Chapelle und Erhan (2011) und Glorot und Bengio (2010)). Für Convolutional Neural Networks berechnen sich fan_{in} und fan_{out} durch Multiplikation der Filtergröße mit der Anzahl *Input-Maps* respektive *Feature-Maps* c : $k_w \cdot k_h \cdot c$ (vgl. He et al., 2015).

3.2.2 Unüberwachtes Vortraining

Dieses Kapitel weicht die in Kapitel 1 gemachte Einschränkung auf das klassische überwachte, neuronale Lernmodell insofern auf, als dass ein un-

überwachtes Lernverfahren vorgestellt wird. Dieses Vorgehen dient jedoch zum einen dem höheren Ziel, das überwachte Verfahren durch eine bessere Initialisierung der Gewichte besser zu konditionieren und in der Performanz zu verbessern. Zum anderen wird sich zeigen, dass es sich bei genauerer Betrachtung nicht um ein klassisches unüberwachtes Lernverfahren handelt: Es wird lediglich ein unüberwachtes Problem als überwachtes Problem angesehen. Dies lässt sich am einfachen Beispiel eines nichtlinearen Autoencoders verdeutlichen (vgl. hierzu und im Folgenden Masci et al., 2011). Der Encoder in Formel 3.6 nimmt als Eingabe einen Vektor x und berechnet einen Code h .

$$h = \phi(Wx + b) \quad (3.6)$$

Dieser Code wird im zweiten Schritt durch den Decoder in Formel 3.7 dekodiert und so die Ausgabe berechnet. Das Ziel des Autoencoders ist es den Fehler zwischen Eingabe und Rekonstruktion, wie z.B. $MSE = \mathbb{E}[(x - y)^2]$, durch Anpassen der Gewichte $\theta = W, b$, wobei $W' = W^T$ (*Tied Weights*), zu minimieren.

$$y = \phi(W'h + b') \quad (3.7)$$

Der Autoencoder prädiziert folglich die Eingabe selbst und lernt die sogenannte Identität (*Identity Mapping*). Die Ideen von Hinton et al. (2006), die gleichzeitig auch den Beginn der Renaissance von Deep Learning darstellen, beschreiben ein Verfahren große Netzwerke mit vielen Schichten schichtweise *greedy* mit beschränkten Boltzmann Maschinen (RBMs) zu trainieren (vgl. Bengio et al. (2007)). Das bedeutet, dass jede Schicht als Eingabe die Ausgabe der vorherigen Schicht erhält und versucht, diese wiederum zu prädizieren (vgl. Ranzato et al. (2006)). Dies wirkt ähnlich wie ein Regularisierer und initialisiert die Gewichte hinsichtlich der Optimierung und Generalisierung in einer besseren Ausgangssituation (vgl. Erhan et al., 2010). Hiervon profitieren besonders große MLPs, welche bis dato aufgrund des *Vanishing-Gradient*-Effekts als schwer zu trainieren galten.

Eine wichtige Weiterentwicklung des Konzepts sind sogenannte *Denoising Autoencoder* (DAs) beziehungsweise *Stacked Denoising Autoencoder* (SDA), welche versuchen die Eingabe x ausgehend von einer korrumptierten Version \hat{x} zu prädizieren. Dies erlaubt das Erlernen robuster Merkmale und kann als diskriminative Alternative zum generativen stochastischen RBM-Modell gesehen werden (vgl. Vincent et al. (2008)).

Convolutional Neural Networks wie das *LeNet 5* von LeCun et al. (1998a) können aufgrund ihrer besonderen Struktur dennoch trainiert werden. Trotzdem ist es auch bei CNNs schwierig die ersten Schichten zu trainieren, da die hinteren Schichten generell schneller unscheinbares *Overfitting* mit entsprechend kleinem Gradienten aufweisen (vgl. Erhan et al. (2010)). Der erste Versuch unüberwachtes Vortraining auf CNNs anzuwenden stammt von

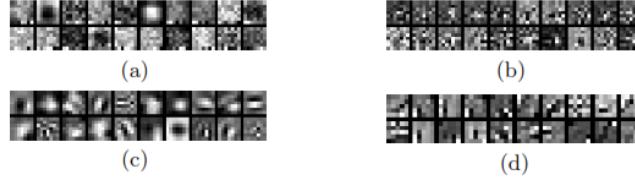


Abbildung 3.5: Filter der ersten Schicht eines Convolutional Autoencoders
(a) Kein Max-Pooling, 0 % Rauschen, (b) Kein Max-Pooling, 50 % Rauschen,
(c) Max-Pooling 2×2 , (d) Max-Pooling 2×2 , 30 % Rauschen (siehe Masci et al. (2011))

Ranzato et al. (2006). Hier wird die erste Schicht eines veränderten *LeNet-5* mittels Vortraining initialisiert, was den Fehler auf dem MNIST-Datensatz¹ von 0.7% auf 0.6% verringert. Das Vortraining der 5×5 -Filter geschieht mit zufällig gewählten 5×5 -Bildausschnitten auf der Basis von RBMs.

Convolutional Autoencoder

Es existieren verschiedenste Architekturvorschläge für Autoencoder und RBMs zum Vortraining von CNNs. Die Grundlage hierfür ist meist die Extraktion den Filter entsprechend großer Bildausschnitte und das Trainieren gewohnter Modelle (vgl. Desjardins und Bengio (2008), Ranzato et al. (2007b) oder Lee et al. (2009)). Im Folgenden wird der Convolutional Autoencoder von Masci et al. (2011) vorgestellt, da dieser auf Faltungen basiert und, wie Abbildung 3.5 zeigt, die gewünschten translationsinvarianten Filter erzeugt. Die vorgestellte Variante verbessert im Rahmen der Experimente von Masci et al. (2011) den Testfehler auf dem MNIST-Datensatz von 0.79% auf 0.71%. Die Architektur des vorgestellten Convolutional Autoencoders besteht aus einem Convolution-Layer sowie einem nachgeschalteten Pooling-Layer. Das bedeutet, dass sich der gesamte Autoencoder letztendlich aus zwei hintereinander ausgeführte Layern dieser Art zusammensetzt.

Gleichung 3.8 zeigt die Berechnung des Codes h_i der i -ten *Feature-Map*. Dieser wird wie in gewöhnlichen CNNs durch Faltung mit Randbehandlung *valid* berechnet.

$$h_i = \phi\left(\sum_{j=0}^m x_j * W_{ij} + b_i\right) \quad (3.8)$$

Der Decoder berechnet die Funktion 3.9, wobei ähnlich dem *Backward Pass* im Backpropagation die Filtermaske über beide Seiten geflippt be-

¹Der MNIST-Datensatz handgeschriebener Ziffern besteht aus 60,000 Trainingsbeispielen und 10,000 Testbeispielen. Es stellt eine Untermenge des größeren NIST-Datensatzes dar. Die Bilder sind 28×28 und die Ziffern zentriert (<http://yann.lecun.com/exdb/mnist/> (26.08.2015)).

ziehungsweise um 180° gedreht und die Randbehandlung *full* verwendet wird. Die Dekodierung ist beeinflusst von gewöhnlichen Autoencodern mit *Tied Weights*, bei denen als Dekodierungsfunktion ebenfalls die transponierte Gewichtsmatrix des *Forward Pass* verwendet wird.

$$y_j = \phi\left(\sum_{i=0}^n h_i * \text{rot180}(W_{ij}) + c_j\right) \quad (3.9)$$

Der Gradient für den Autoencoder berechnet sich mit Formel 3.10. Diese setzt sich, wie der Autoencoder, aus zwei Teilen zusammen. Der erste Teil wird für den Encoder, der zweite für den Decoder benötigt. Im Unterschied zur Berechnung des Gradienten beim CNN wird anstatt δy der Code h der Flip-Operation unterzogen. Dies tritt auf, da der *Backward Pass* dem eigentlichen *Forward Pass* entspricht.² Eine zusätzliche Besonderheit besteht in Zusammenhang mit der Randbehandlung *valid*. Um den Code $\text{rot180}(h)$ mit dem räumlich größeren δy der Ausgabe zu falten, muss dieser doppelt mit Nullen erweitert werden (Padding). Zum einen um die gleiche Größe wie die Ausgabe zu erhalten und zum anderen um die Faltung zu ermöglichen.

$$\frac{\partial J(W, b, c)}{\partial W_{ij}^l} = \frac{1}{M} \sum_{m=1}^M \text{rot180}(x_{mj}^l * \text{rot180}(\delta h_{mi}^l) + \text{rot180}(h_{mi}^l) * \delta y_{mj}^l) \quad (3.10)$$

Analog zum bekannten Convolution-Layer wird über alle Trainingsbeispiele M gemittelt und die beiden Schwellwerte b und c berechnen sich aus der Summe der zugehörigen Deltas δy und δh , wie in Gleichung 3.11 und 3.12 dargestellt ist.

$$\frac{\partial J(W, b, c)}{\partial b_i^l} = \frac{1}{M} \sum_{m=1}^M \sum_{u=0}^{k_w} \sum_{v=0}^{k_w} \delta h_{miuv}^l \quad (3.11)$$

$$\frac{\partial J(W, b, c)}{\partial c_j^l} = \frac{1}{M} \sum_{m=1}^M \sum_{u=0}^{k_w} \sum_{v=0}^{k_w} \delta y_{miuv}^l \quad (3.12)$$

Weiterentwicklungen wie die *Tiled CNNs* versuchen weitere Invarianzen, wie beispielsweise die Rotationsinvarianz, zu lernen (vgl. Le et al. (2010)). Einer der größten Autoencoder, der *Google Autoencoder*, bedient sich ebenfalls der lokalen rezeptiven Felder, verzichtet allerdings auf *Parameter Sharing*, um verschiedene Invarianzen zu lernen (vgl. Le et al. (2012)). Eine Schicht dieser Architektur ist in Abbildung 3.6 abgebildet.

²Im Unterschied zur von Masci et al. (2011) beschriebenen Berechnung wird in Konsequenz auch auf die Flip-Operation von δh nicht verzichtet und der Gradient am Ende gesamtheitlich geflippt. Der Gradient für den Encoder wird damit äquivalent zu dem in einem CNN berechnet. In Kapitel 4.2.4 wird die Richtigkeit dieser Änderung überprüft.

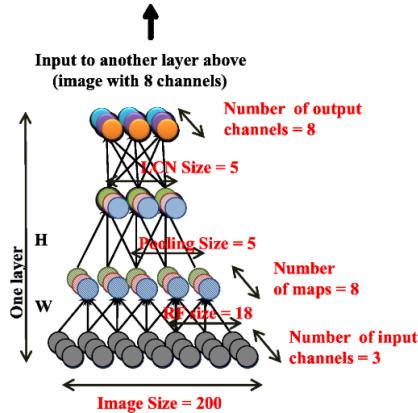


Abbildung 3.6: Autoencoder mit lokalen rezeptiven Feldern aber ohne *Parameter Sharing* (siehe Le et al. (2012))

Auch wenn das Verwenden von unüberwachtem Vortraining für die ersten Schichten eines CNNs passend erscheinen mag, führt es nicht zu einer solchen Verbesserung der Ergebnisse wie bei tiefen MLPs (DNNs) (vgl. Abdel-Hamid et al. (2013)). Durch die Entwicklung neuer Aktivierungsfunktionen wie ReLu und raffinierter Initialisierung rückt das Thema somit in solche Bereiche, in denen wenig gelabelte Trainingsdaten zur Verfügung stehen, um das CNN vollständig überwacht zu trainieren (vgl. Masci et al. (2011) und Le et al. (2012)). Darüber hinaus können auch verbesserte Optimierungsverfahren das unüberwachte Vortraining hinsichtlich einer besseren Initialisierung ersetzen (vgl. Martens (2010) und Sutskever et al. (2013)).

Eine neue Herangehensweise ist es, bereits trainierte CNNs für andere Probleme zu übernehmen, was als sogenanntes Transferlernen bezeichnet wird (vgl. Wagner et al. (2013)). So liefert zum Beispiel die 7. Schicht des *AlexNet* von Krizhevsky et al. (2012) bereits sehr gute Bildbeschreibungen (vgl. Bell und Bala (2015) und Fei-Fei und Karpathy (2014)).

3.3 Gradientenabstieg

Dieses Kapitel behandelt verschiedene Methoden zur Verbesserung des Gradientenabstiegs. Diese können das Training erheblich beschleunigen, da es sich im Allgemeinen im Deep Learning um nicht-konvexe Fehlerlandschaften wie in Abbildung 3.7 handelt.

Die Grundlage für das effiziente Lernen mit vielen Trainingsdaten ist der sogenannte stochastische Gradientenabstieg (SGD) (vgl. hierzu und im Folgenden Bottou (1998)). Hierbei geht es um die Frage, was die erwartete durchschnittliche Richtung des Gradienten ist. Durch die Einführung des MSE-Fehlermaßes ist der Ansatz des SGD bereits gegeben. So entspricht die

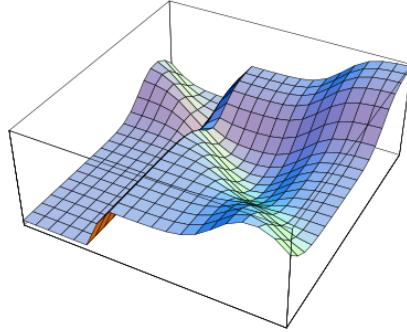


Abbildung 3.7: Fehlerfunktion beziehungsweise Zielfunktion im Gewichtsraum mit lokalem Minimum (siehe Rojas, 1996, S. 155)

MSE-Fehlerfunktion dem Erwartungswert der einzelnen Fehlerquadrate. Wie in Gleichung 3.13 zu sehen ist, entspricht analog dazu der Erwartungswert der einzelnen Gradienten dem Gradient der Fehlerfunktion.

$$\nabla J(W, b) = \mathbb{E}[\nabla(f(x_i) - y_i)^2] = \frac{1}{N} \sum_i^N \nabla(f(x_i) - y_i)^2 \quad (3.13)$$

Für eine allgemeine Fehlerfunktion $e(x)$ gilt: Nimmt man an Stelle der gesamten Trainingsmenge N (Batch) einzelne Trainingsbeispiele i (Online) oder eine Teilmenge M (Averaged SGD), so ergibt sich Formel 3.14 für den Gradientenabstieg mit Lernrate beziehungsweise Schrittweite η . Der Gradient zeigt somit in Richtung des Durchschnitts und damit in Richtung des Erwartungswerts.

Neben einer effizienteren Berechnung des Gradientenabstiegs, liefert SGD eine Zufallskomponente (*Randomization*) (vgl. hierzu und im Folgenden Bengio, 2012). Diese kann zu beschleunigter Konvergenz führen, da durch variierende (*noisy*) Gewichtsupdates ein größeres Gebiet untersucht werden kann. Außerdem erlaubt SGD Training ohne unmittelbarem Zugriff auf die gesamte Trainingsmenge. Üblicherweise werden die für die Berechnung entnommenen Teilmengen als *Mini-Batch* bezeichnet.

$$W_{t+1} = W_t - \eta \frac{1}{M} \sum_{i=1}^M \nabla e(x_i) \quad (3.14)$$

SGD bildet die Grundlage für viele Erweiterungen und stellt so die erste Optimierung des Gradientenabstiegs dar. Um die Formeln zu vereinfachen, wird im Folgenden auf die sonst obligatorische Mittelung des Gradienten verzichtet. An den entsprechenden Stellen wird darauf hingewiesen, ob es sich um die Erweiterung für den Online-/Mini-Batch-Modus oder um den

klassischen Batch-Modus, welcher für Deep Learning in Verbindung mit vielen Trainingsdaten unpraktikabel ist, handelt.

Lernrate und Mini-Batch-Größe

Wird der klassische Gradientenabstieg ohne Optimierungen verwendet, ist es besonders wichtig η richtig zu wählen. Aufgrund des *Vanishing Gradient-Effekts* sollte die Lernrate in den letzten Schichten kleiner gewählt werden. Außerdem ist darauf zu achten, dass die Lernrate durch *Parameter Sharing* proportional zur Wurzel der Verbindungen ist, die sich ein Gewicht teilen. Letzteres ist gerade bei CNNs zu beachten, um eine unnötig schlechte Konditionierung der Fehlerlandschaft zu vermeiden (vgl. LeCun et al. (1998b)).

Die Bestimmung der Mini-Batch-Größe M erlaubt es zwischen Online- und Batch-Training zu interpolieren. Dies ist wichtig, da die richtige Wahl der Größe von den Trainingsdaten abhängt und nicht allgemein vorhergesagt werden kann. So kann beispielsweise Online-Training in vielen Situationen schneller sein als Batch-Training (vgl. Wilson und Martinez, 2003). Als Stellgröße kann der Grad an Redundanz in den Trainingsdaten dienen. Mit steigender Redundanz sollten die Batches entsprechend kleiner gewählt werden, um nicht unnötig Rechenzeit zu verbrauchen. Bengio (2012) nennt $M = 32$ einen guten Standardwert.

3.3.1 Momentum

Eine der ersten erfolgreichen Erweiterungen für iterative Verfahren ist die so genannte Momentum Methode von Polyak (1964). Die Momentum-Methode ist durch die Physik motiviert, dadurch dass dem Gradienten eine potentielle Energie zugewiesen wird. Dies erlaubt es in gleichbleibender Richtung des Gradienten Geschwindigkeit aufzunehmen. Dies führt letztendlich dazu, dass in flachen Regionen der Fehlerlandschaft größere und in unwegsamen Bereichen kleinere Schrittweiten effektiv ausgeführt werden (vgl. LeCun et al. (1998b)). Dies wird durch das Hinzufügen eines weiteren Terms in das Gewichtsupdate erzielt. Wie Formeln 3.15 und 3.16 zeigen, bildet die Momentum-Methode einen gewichteten Durchschnitt der vergangenen Gradienten der Gewichte.

$$\Delta W_t = \mu \Delta W_{t-1} - \eta \nabla J(W_t, b_t) \quad (3.15)$$

$$W_{t+1} = W_t + \Delta W_t \quad (3.16)$$

Typische Werte für die Momentum-Rate μ sind Werte größer 0.9. Diese sind allerdings abhängig vom Lernproblem und können nicht allgemein formuliert werden (vgl. Fei-Fei und Karpathy (2014)). Die Momentum-Methode ist eine sehr einfache Optimierung für SGD und findet in erfolgreichen Netzen häufig Anwendung (vgl. Krizhevsky et al. (2012)).

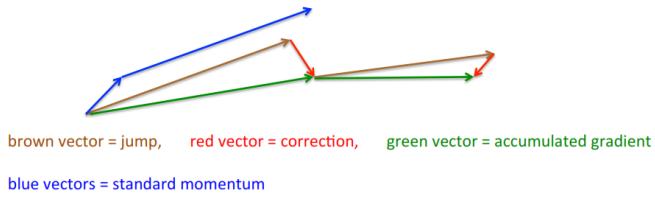


Abbildung 3.8: Das effektivere Nesterov-Momentum korrigiert Fehler im Nachhinein (siehe Hinton et al., 2015)

Nesterov-Methode

Die Nesterov-Methode (NAG) ist eine verbesserte Momentum-Methode, die von Nesterov (1983) beeinflusst ist (vgl. hierzu und im Folgenden Sutskever, 2013). Diese Methode führt zunächst den über die vergangenen Perioden gewichteten Momentum-Term aus und berechnet dann den Gradient an dieser Stelle. Die Methode fungiert somit als eine Art Ausblick auf die zukünftige Fehlerlandschaft. Der eigentliche Gradient ergibt sich schließlich aus Ausblick und Korrektur, wie Abbildung 3.8 zeigt.

Da diese Ausführung die Reihenfolge des üblichen Trainings beeinflussen würde, wird die Methode derart umgeschrieben, dass die aktuellen Gewichte immer dem Ausblick entsprechen. Daraus ergeben sich die Regeln 3.17 und 3.18 für die Aktualisierung der Gewichte (vgl. Fei-Fei und Karpathy (2014)).

$$\Delta W_t = \mu \Delta W_{t-1} - \eta \nabla J(W_t, b_t) \quad (3.17)$$

$$W_{t+1} = W_t + (-\mu \Delta W_{t-1}) + (1 + \mu) \Delta W_t \quad (3.18)$$

Die beschriebene Methode ist nach Sutskever et al. (2013) robuster hinsichtlich der Momentum-Rate und lässt höhere Raten zwischen 0.9 und 0.99 zu.

3.3.2 Adaptive Lernrate

Ein allgemeines Problem in Verbindung von Gradientenabstieg und dem Training im Deep Learning ist der *Vanishing Gradient*-Effekt nach Hochreiter (1991). Ein Algorithmus der direkt auf diesen Effekt abzielt, ist der sogenannte Resilient Propagation, auch Rprop genannt (vgl. Riedmiller und Braun (1992) und Igel und Hüskens (2000)). Rprop weist jedem Gewicht eine eigene Lernrate zu und verzichtet verwendet nur das Vorzeichen des Gradienten. Die grundlegende Arbeitsweise sieht vor, bei gleichbleibendem Vorzeichen der partiellen Ableitung eines Gewichts die zugehörige Lernrate zu erhöhen. Tritt ein Vorzeichenwechsel auf, so liegt es nahe, dass in der Richtung dieser Ableitung eine Senke übersprungen wurde und die Lernrate wird reduziert. Rprop ist nach Hinton et al. (2015) äquivalent zum Gradientenabstieg im

Batch-Modus, wenn bei letzterem durch die Länge des Gradient geteilt wird. Hier liegt ein fundamentales Problem hinsichtlich Deep Learning: Um die Länge des Gradienten korrekt zu schätzen, bedarf es der gesamten Trainingsmenge, was die Verwendung von SGD ausschließt. Somit ist Rprop nur für Batch-Training geeignet (vgl. Hinton et al. (2015)).

Der bekannte Newton-Algorithmus für Optimierung in Gleichung 3.19 multipliziert nicht mit einer globalen Lernrate, sondern mit der Inversen der Hesse-Matrix. Unter bestimmten Annahmen, wie etwa einer quadratischen Zielfunktion, erreicht dieser Algorithmus innerhalb eines Schrittes das Minimum, was als Newton-Schritt bezeichnet wird (vgl. Bottou (1998)).

$$W_{t+1} = W_t - H_t^{-1} \nabla J(W_t, b_t) \quad (3.19)$$

Die Hesse-Matrix ist im Allgemeinen sehr teuer zu berechnen, da jede partielle Ableitung der N Gewichte nochmals abgeleitet werden müsste. Dies führt auf eine $N \times N$ -Matrix, welche in neuronalen Netzen nicht effizient zu berechnen ist und deshalb approximiert werden muss (vgl. LeCun et al., 1998b).

Algorithmen mit sogenannter adaptiver Lernrate gehen so vor, dass sie lediglich die Diagonale der Hesse-Matrix $\text{diag}(H)$ approximieren. Dies führt zu eigenen Lernraten pro Gewicht, welche idealerweise den echten Eigenwerten der Hesse-Matrix entsprechen (vgl. LeCun et al., 1998b). Adaptive Lernraten erlauben es, Schrittweiten in Gegenden schwacher Krümmung zu erhöhen und entsprechend in anderen zu verkürzen. Dieser Mechanismus wirkt damit ebenso unmittelbar dem *Vanishing Gradient*-Effekt entgegen, da kleiner werdende Gradienten zu den ersten Schichten hin skaliert werden (vgl. Martens, 2010). Problematisch bei der Verwendung der Hesse-Matrix für nicht-konvexe Probleme sind negative Eigenwerte an lokalen Maxima oder gemischt positive und negative Eigenwerte an Sattelpunkten, die zu einer falschen Richtung des Gradienten führen (vgl. Dauphin et al., 2014). Aus diesem Grund ist es notwendig, eine Methode mit entsprechender Vorkonditionierung zu verwenden (vgl. Dauphin et al., 2015).

Stochastischer Levenberg-Marquardt

Der bekannte stochastische Levenberg-Marquardt-Algorithmus (LMA) von LeCun et al. (1998b) ist ein Algorithmus im Bereich Neuronale Netze und SGD, der versucht die $\text{diag}(H)$ zu approximieren. Hierzu sind allerdings zwei weitere Approximationen notwendig. So wird die Gauß-Newton Approximation zur Vermeidung negativer Eigenwerte, wie im klassischen LMA-Algorithmus, angewandt und die $\text{diag}(H)$ nur auf einer kleinen Teilmenge der Trainingsdaten (Mini-Batch) berechnet. Darüber hinaus wird die Berechnung nur einmal pro Epoche durchgeführt. Die Kosten entsprechen denen eines normalen *Forward Pass* sowie einem angepassten *Backward Pass*, der zur Rückpropagierung der zweiten Ableitungen dient, und sind somit zu vernachlässigen. Neben der Kompensation des *Vanishing-Gradient*-Effekts

durch adaptive Lernraten, kompensiert der stochastische LMA den Effekt der schlechten Konditionierung der Fehlerlandschaft durch *Parameter Sharing* (vgl. LeCun et al., 1998b). Die neue Lernrate pro Gewicht i ergibt sich mit Formel 3.20.

$$\eta_i = \frac{\eta}{\mu + H_{ii}} \quad (3.20)$$

Wie im klassischen LMA wird ein Dämpfungsfaktor μ verwendet, um zu verhindern, dass die Schrittweite zu groß wird. Dieser Faktor kann auch als Vorannahme gesehen werden, die angibt, dass sich das Gewicht nicht ändern soll, wenn die Krümmung entsprechend klein wird (vgl. Martens (2010)). Im *LeNet 5* wird $\mu = 0.02$ verwendet.

Equilibrium SGD

Eine Verbesserung zum stochastischen LMA ist die von Dauphin et al. (2015) beschriebene Equilibrium-Methode, welche die Matrix $D^{Eq} = \sqrt{diag(H^2)}$ berechnet. Diese Vorkonditionierung bietet einige Vorteile hinsichtlich der Behandlung der genannten indefiniten Hesse-Matrizen, die durch Sattelpunkten innerhalb der Fehlerlandschaft von MLPs auftreten. Darüber hinaus kann diese Methode effizient durch den Zusammenhang in Formel 3.21 mit $v \sim \mathcal{N}(0, 1)$, beispielsweise mit dem $R\{\cdot\}$ -Operator (vgl. Pearlmutter (1994)), berechnet werden.

$$D^{Eq} = diag(H^2) = E[(Hv)^2] \quad (3.21)$$

Im Unterschied zum stochastischen LMA bildet die Equilibrium-Methode, wie Formel 3.22 zeigt, einen *Root Mean Square*-Durchschnitt (RMS) über die vergangenen Perioden. Dauphin et al. (2015) schlagen eine Neuberechnung von D^{eq} nach jeweils 20 Iterationen und $\mu \in [10^{-4}, 10^{-6}]$ vor.

$$D_{t+1}^{eq} = \rho D_t^{eq} + (1 - \rho)(Hv)^2 \quad (3.22)$$

Damit ergibt sich die in Formel 3.23 gezeigte effektive Lernrate, welche ebenfalls einen Dämpfungsfaktor μ zur Vermeidung großer Schrittweiten beinhaltet.

$$\eta_i = \frac{\eta}{\mu + \sqrt{D_{ii}^{eq}}} \quad (3.23)$$

RMSprop

Der RMSprop-Algorithmus stammt von Hinton et al. (2015) und kombiniert zwei Ideen. Einerseits die Idee des Rprop nur auf das Vorzeichen des Gradienten zu achten, indem er den *Root Mean Square*-Durchschnitt (RMS) der partiellen Ableitungen über mehrere Zeitschritte berechnet. Andererseits

wird das Hauptproblem von AdaGrad der stetig kleiner werdenden effektiven Lernraten dadurch vermieden (vgl. Bengio et al., 2015, Kap. 8.4.1, S. 257). Die Formel 3.24 beschreibt die Berechnung des RMS. Die effektive Lernrate berechnet sich mit der Formel 3.25, wobei ebenfalls ein Dämpfungsfaktor μ verwendet wird. Der Parameter ρ gibt an mit welcher Rate der exponentiell geglättete Mittelwert abnimmt. Dauphin et al. (2015) schlagen $\mu = 10^{-2}$ und $\rho = 0.9$ als robuste Werte vor.

$$\hat{H}_{t+1} = \rho \hat{H}_t + (1 - \rho) \nabla J(W_t, b_t)^2 \quad (3.24)$$

$$\eta_i = \frac{\eta}{\mu + \sqrt{\hat{H}_{ii}}} \quad (3.25)$$

Eine sehr interessante Eigenschaft von RMSprop ist die Fähigkeit zur Approximation der Equilibrium-Matrix $D^{Eq} = \sqrt{diag(H^2)}$ (vgl. Dauphin et al. (2015)). RMSprop stellt eine sehr effektive und praktische Optimierung dar, welche dazu einfach zu implementieren ist. Dies macht diese derzeit sehr beliebt (vgl. Bengio et al. (2015)).

AdaDelta

AdaDelta ist ein von Zeiler (2012) entwickeltes Verfahren, welches ebenfalls das Problem der immer kleiner werdenden Lernrate von AdaGrad adressiert (vgl. Bengio et al. (2015)). Darüber hinaus versucht AdaDelta aus der ersten Ableitung die Krümmung der Funktion (*Second Order Information*) zu berechnen. Der Divisor in Formel 3.26 entspricht, bis auf dem unter der Wurzel stehenden μ , dem des RMSprop aus Formel 3.24. Eine Neuheit stellt der RMS über den Gewichtsupdates $RMS[\Delta x]$, der einen Zeitschritt versetzt berechnet wird und als eine Art Momentum wirkt. Insgesamt schätzt die AdaDelta-Methode die absolute Hesse-Matrix $|diag(H)|$ aus Gradient und Gewichtsupdate (vgl. Zeiler, 2012).

$$\Delta x_t = \frac{\sqrt{\mu + RMS[\Delta x]_{t-1}^2}}{\sqrt{\mu + \hat{H}_t}} \quad (3.26)$$

Die effektive Lernrate ergibt sich aus Formel 3.27.

$$\eta_i = \Delta x_i \quad (3.27)$$

Eine Besonderheit von AdaDelta ist, dass gänzlich auf eine globale Lernrate verzichtet und lediglich μ und ρ als Hyperparameter benötigt werden. Da sich μ unter der Wurzel befindet ist dieses entsprechend kleiner als bei den vorherigen Methoden. Zeiler (2012) schlägt hierbei Werte bis $\mu = 10^{-8}$ vor. Auch wenn ursprünglich in Zeiler (2012) keine Lernrate beschrieben ist, kann

es für manche Szenarios dennoch nützlich sein, zusätzlich eine Lernrate zu implementieren. Beispielweise um diese im Laufe des Trainings zu reduzieren.

3.3.3 Hessian Free Optimazation (HF)

Algorithmen mit adaptiver Lernrate vernachlässigen, im Sinne der Approximation, alle nicht-diagonalen Werte der Hesse-Matrix. Damit wird ein gewisser Fehler akzeptiert, da diese Werte gerade die Interaktion der Gewichte hinsichtlich der Zielfunktion beschreiben (vgl. hierzu und im Folgenden Martens (2010)).

Die *Hessian Free Optimazation* (HF) von Martens (2010) versucht, im Gegensatz zu den vorherigen Methoden, nicht den *Vanishing Gradient*-Effekt zu eliminieren oder Plateaus zu überwinden, sondern zielt direkt auf die Optimierung von ungünstigen Fehlerlandschaften (*Pathological Curvature*) ab. Für eine gegebene Stelle w_t im Gewichtsraum wird ein lokales quadratisches Modell angenommen und dieses mittels Conjugate Gradient (CG) optimiert. Deshalb kann HF nur im *Batch*-Modus betrieben oder, analog zum stochastischen LMA, mit einer hinreichend großen Teilmenge der Trainingsdaten. Eine Besonderheit dieser Methode ist, dass der gefundene Vektor von CG im letzten Zeitschritt als Initialisierung für den aktuellen Zeitschritt gewählt werden und dadurch Informationen über mehrere Iterationen weitergeben kann.

Sutskever et al. (2013) unterziehen diese Methode einer kritischen Betrachtung und stellen fest, dass sie, aufgrund der Verwendung des linearen CG, keine Methode 2. Ordnung und daher den anderen Methoden nicht allgemein überlegen ist. So zeigen Sutskever et al. (2013), dass die vorgestellte HF-Methode sehr viele Ähnlichkeiten zum Nesterov-Momentum aufweist und zu diesem unter der Annahme, dass nur ein Schritt im CG-Algorithmus ausgeführt wird, sogar äquivalent ist. Sie kommen zu dem Schluss, dass die interessante Eigenschaft, die Schrittweiten in Bereichen mit geringer Krümmung zu erhöhen und in Bereichen großer Krümmung entsprechend zu verringern, auf beide Methoden HF und NAG zutrifft. Damit sprechen sie der NAG-Methode besonders im Bereich tiefer Netze großes Potenzial zu.

3.4 Regularisierung und Generalisierung

Lernprobleme selbst lassen sich beispielsweise mit bayesianischer Inferenz anschaulich beschreiben (vgl. hierzu und im Folgenden Mitchell, 1997, S. 159f.).

$$P(h|\mathcal{D}) = \frac{P(\mathcal{D}|h)P(h)}{P(\mathcal{D})} \quad (3.28)$$

So beschreibt die Formel 3.28 den Zusammenhang zwischen gegebenen Trainingsdaten \mathcal{D} und der Hypothese h , welche den zu erlernenden Parametern

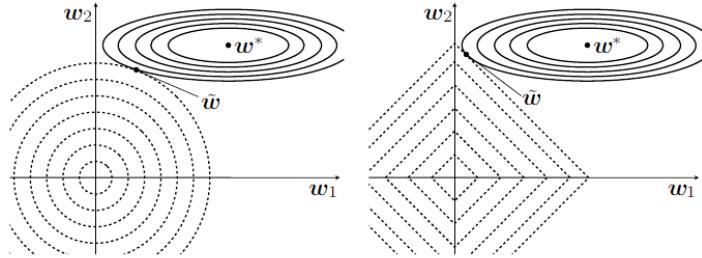


Abbildung 3.9: Effekt der L^2 - (links) und L^1 -Norm (rechts) auf den Wert der optimalen Gewichte W mit empirischem Optimum W^* (vgl. Bengio et al., 2015, Kap. 7.2, S. 199)

$\Theta = \{W, b\}$ entspricht. Sie soll die gegebenen Daten \mathcal{D} möglichst gut abbilden können sowie eine gute Generalisierung für unbekannte Daten leisten. In der bayesianischen Interpretation korrespondieren Regularisierer mit der A-priori Verteilung der Hypothese h .

Im Deep Learning existieren verschiedenste Methoden der Regularisierung, welche auf eine bessere Generalisierung abzielen (vgl. hierzu und im Folgenden Bengio et al., 2015, Kap. 7, S. 196). Manche betreffen die Parameter selbst, andere codieren Expertenwissen oder regulieren die Kapazität von Modellen mit sehr vielen freien Parametern. Im Umfeld neuronaler Netze betrifft die Regularisierung von Parametern meist lediglich die Gewichte W , der Schwellwert b bleibt unbeachtet (vgl. Hinton et al. (2012), Fei-Fei und Karpathy (2014) und Srivastava et al. (2014)).

3.4.1 A-priori Annahmen

Die A-priori Verteilung der Parameter Θ gibt an wie wahrscheinlich verschiedene Werte für die einzelnen Parameter sind. Die klassischen Methoden des maschinellen Lernens sowie der Statistik bestrafen die Norm der Parameter (*Parameter Norm Penalty*) und vermeiden so große Werte dieser. Die Norm wird als $\Omega(\Theta)$ definiert und zur Zielfunktion, wie in Gleichung 3.29, gewichtet mit dem Hyperparameter α addiert (vgl. Bengio et al., 2015, Kap. 7.2, S. 200).

$$\hat{J}(W, b) = J(W, b) + \alpha\Omega(\Theta) \quad (3.29)$$

Typische Normen sind die L^1 - und L^2 -Norm. Abbildung 3.9 stellt die beiden Varianten gegenüber.

L^1 -Norm

Bei der Verwendung der L^1 -Norm zur Berechnung von $\Omega(\Theta)$ bleibt nach dem Ableiten lediglich das Vorzeichen $sign(w_i)$ der einzelnen Gewichte übrig. Damit ergibt sich die in Gleichung 3.30 dargestellte Regel für die Berechnung des neuen Gradienten $\nabla_w \hat{J}(W, b)$ pro Gewicht w (vgl. Bengio et al., 2015, Kap. 7.2, S. 203 f.).

$$\nabla_w \hat{J}(W, b) = \nabla_w J(W, b) + \alpha sign(w) \quad (3.30)$$

Die L^1 -Regularisierung bestraft die Summe aller Absolutwerte der Gewichte und verkleinert diese somit während des Gradientenabstiegs stetig mit gleicher Rate. Die rechte Grafik in Abbildung 3.9 zeigt, dass L^1 -Regularisierung einerseits zu größeren Werten führt, im Gegenzug jedoch auch sehr kleine Werte bevorzugt. Diese Eigenschaft führt zu *Sparsity*, da kleine Gewichte w zwangsläufig im Laufe des Trainings den Wert Null annehmen (vgl. Bengio et al., 2015, Kap. 7.2, S. 203). Die bayesianische Interpretation der Methode ist die A-priori Annahme einer isotropischen Laplace-Verteilung der Gewichte (vgl. Bengio et al., 2015, Kap. 7.2, S. 206).

L^2 -Norm

Trotz der interessanten *Sparsity* Eigenschaften der L^1 -Regularisierung wird im Deep Learning meist die L^2 -Norm zur Regularisierung verwendet und die *Sparsity* mit ReLu-Funktionen erzwungen (vgl. z.B. Krizhevsky et al., 2012). Diese aus der *Ridge Regression* bekannte Form der Regularisierung, führt zu Parameterwerten näher dem Ursprung. In der bayesianischen Interpretation entspricht diese somit einer Gauß-Verteilung mit Mittelwert Null (vgl. Bengio et al., 2015, Kap. 7.2, S. 200). Der Gradient der L^2 regularisierten Fehlerfunktion ist in Gleichung 3.31 aufgeführt.

$$\nabla_w \hat{J}(W, b) = \nabla_w J(W, b) + \alpha w \quad (3.31)$$

Diese Form der Regularisierung führt dazu, dass das Modell eine größere Varianz in den Trainingsdaten X annimmt. Somit werden Gewichte zu Merkmalen mit geringer Kovarianz zur Ausgabe verkleinert (vgl. Bengio et al., 2015, Kap. 7.2, S. 200 f.). Die *Ridge Regression* in Gleichung 3.32 verdeutlicht diesen Aspekt, indem ein Vielfaches der Identitätsmatrix I auf die Kovarianzmatrix $X^T X$ addiert wird.

$$w = (X^T X + \alpha I)^{-1} X^T y \quad (3.32)$$

In Abbildung 3.9 erkennt man diesen Effekt daran, dass das Gewicht w_1 , in dessen Richtung die Zielfunktion im Gewichtsraum eher flach ist, geschrumpft wird.

Max- L^2 -Norm

Max- L^2 -Norm Regularisierung beschreibt eine Methode, welche die L^2 -Norm der Gewichte eines Neurons p beschränkt sodass $\|w_p\|_2 \leq c$ gilt. Werte für den Hyperparameter c liegen meist zwischen 3 und 4 (vgl. Srivastava et al., 2014). Diese Art der Regularisierung projiziert folglich die Gewichte eines Neurons w_p auf eine Kugel mit Radius c , sobald die Norm diese verlässt. Nach Srivastava et al. (2014) verhindert dies den *Exploding Gradient*-Effekt, da die Gewichte den Fehler beim Rückpropagieren nicht mehr überproportional verstärken können. Dies ermöglicht größere Lernraten im Vergleich zum Training ohne Max-Norm-Regularisierung, was gerade in Verbindung mit Dropout-Learning (siehe Kapitel 3.4.2) von Vorteil ist, da so größere Gebiete im Gewichtsraum untersucht werden können.



Abbildung 3.10: Ein dominierender Filter ohne Max-Norm Regularisierung (siehe Zeiler und Fergus, 2014)

Abbildung 3.10 zeigt einen weiteren Grund für die Notwendigkeit dieser Form der Regularisierung. Die Dominanz eines Filters über alle anderen kann effektiv verhindert werden (vgl. Zeiler und Fergus, 2014).

3.4.2 Modellkapazität

Beim Training großer Modelle mit hoher Kapazität ist *Overfitting* von zentraler Bedeutung. *Overfitting* beschreibt den Effekt eines kleiner werdenden Fehlers auf den Trainingsdaten bei gleichzeitiger Verschlechterung der Performance auf den Testdaten. Abbildung 3.11 stellt diesen Effekt grafisch dar.

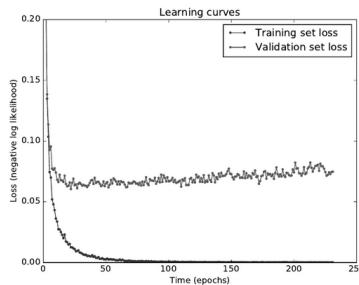


Abbildung 3.11: *Overfitting* am Beispiel des MNIST-Datensatzes (siehe Bengio et al., 2015, Kap. 7.3, S. 216)

Im Folgenden werden Methoden zusammengefasst, welche die Kapazität des Modells regulieren und damit aktiv versuchen *Overfitting* zu verhindern.

Early Stopping

Die einfachste, sehr häufig angewandte Methode *Overfitting* aktiv zu verhindern, ist das sogenannte *Early Stopping*. Diese Methode teilt die Trainingsmenge zuerst in zwei Teilmengen. Sie setzt somit einen Satz Validierungsdaten voraus (vgl. hierzu und im Folgenden Bengio et al., 2015, Kap. 7.3, S. 216 ff.).

Beim *Early Stopping* wird während des Trainings in regelmäßigen Abständen der Fehler auf den Validierungsdaten beobachtet. Verbessert sich dieser, werden die aktuellen Gewichte und Schwellwerte des Modells gesondert gespeichert. Am Ende des Trainings werden folglich nicht die aktuellsten Gewichte zurückgegeben sondern diejenigen, welche den kleinsten Validierungsfehler erzeugten. Anstelle eines lokalen Minimums über den Trainingsdaten wird somit ein lokales Minimum des Validierungsfehlers gesucht. Das Training stoppt, sobald sich der Validierungsfehler über mehrere Epochen nicht verbessert hat (*Early Stopping Patience*). Die Kapazität des Modells wird dahingehend beschränkt, dass die Trainingszeit limitiert ist. Bei der L^2 -Regularisierung wurde beobachtet, dass Gewichte in Richtungen hoher Krümmung hinsichtlich der Zielfunktion weniger stark reguliert werden als andere. Diese Beobachtung lässt laut Bengio et al. (2015) eine Ähnlichkeit zum *Early Stopping* erkennen, da die Gewichte in Verbindung mit hoher Krümmung relativ zu den anderen Gewichten früher im Trainingsprozess gelernt werden.

Parameteranzahl limitieren

Die Kapazität des Modells lässt sich ebenso durch Vereinfachung beziehungsweise Verkleinerung und somit einer Verringerung der zu lernenden Gewichte

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X		X	X	X		X	X	X	X		X	X			
1	X	X		X	X	X		X	X	X	X	X		X		
2	X	X	X		X	X	X		X		X	X	X			
3	X	X	X	X	X	X	X	X	X		X	X	X			
4		X	X	X	X	X	X	X	X	X	X	X	X			
5		X	X	X	X	X	X	X	X	X	X	X	X			

Abbildung 3.12: Verbindungsmaatrix der Merkmale beziehungsweise *Feature-Maps* (Zeilen) mit den dazugehörigen Gewichten beziehungsweise Faltungsmasken (Spalten) zweier aufeinanderfolgenden Schichten (siehe LeCun et al., 1998a)

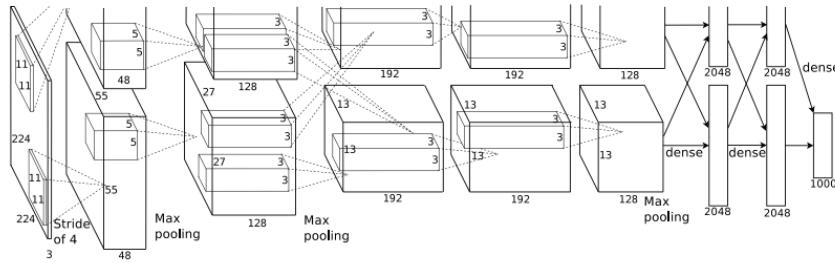


Abbildung 3.13: Architektur für *ImageNet 2012* mit limitierten Verbindungen zwischen *Feature-Maps* zwecks Parallelisierung (siehe Krizhevsky et al., 2012)

regulieren. Im Umfeld des Deep Learning werden grundsätzlich zwei verschiedene Methoden beschrieben: Teilen der Gewichte (*Parameter Sharing*) und Reduktion der Verbindungen zwischen Merkmalen.

Parameter Sharing

Diese Methode ist am meisten verbreitet in CNNs und die Grundlage für die großen Erfolge im maschinellen Lernen und Deep Learning. CNNs berechnen an verschiedenen Stellen des Eingabebeispiels mit den selben Gewichten eine gewichtete Summe, was der algebraischen Faltung entspricht (vgl. LeCun et al., 1998a). Durch *Parameter Sharing* ist es möglich große neuronale Netze zu trainieren, ohne die Trainingsmenge entsprechend zu vergrößern. Damit sind CNNs ein herausragendes Beispiel für die Integration von Expertenwissen in einem neuronalen Netz (vgl. Bengio et al., 2015, Kap 7.8, S. 224).

Limitierung von Verbindungen zwischen Merkmalen

Bei Nutzung dieser Methode wird ebenfalls das Ziel verfolgt Parameter zu reduzieren, wobei allerdings ein anderer Ansatz als bereits vorgestellt gewählt wird. Anstatt die Eingabe nur in überlappende, rezeptive Felder zu unterteilen und mit den selben Gewichten zu gewichten, werden zusätzlich die unterschiedlichen Dimensionen der Eingabe (z. B. RGB-Kanäle) überlappend aufgeteilt. Die bekannte Verbindungsmaatrix von *LeNet 5* ist zur Veranschaulichung

in Abbildung 3.12 dargestellt. Neben der Reduktion von zu trainierenden Gewichten erlaubt dies auch eine bessere Extraktion unabhängiger Merkmale (vgl. LeCun et al., 1998a). Darüber hinaus bedient sich die Architektur von Krizhevsky et al. (2012) zwecks Parallelisierung des Netzes auf mehrere Rechenkerne (hier GPUs) derselben Methode, wie Abbildung 3.13 zeigt.

Dropout

Die Dropout-Methode wurde von Hinton et al. (2012) eingeführt und bezeichnet eine Methode, mehrere kleinere Modelle zu lernen und die verschiedenen Ausgaben zu kombinieren (*Model Averaging*). Hierbei ist hervorzuheben, dass dies gleichzeitig und in einem neuronalen Netz geschieht. (vgl. Srivastava et al., 2014).

Die Dropout-Methode wurde mit dem Ziel entwickelt MLPs wenigen Trainingsdaten zu trainieren, was typischerweise in *Overfitting* resultiert. Hierbei soll die Architektur des Netzes im Training zufällig für jedes Trainingsbeispiel verändert werden, um Abhängigkeiten (*Co-Adaptions*) extrahierter Merkmale zu vermeiden (vgl. Hinton et al., 2012). Dazu werden zufällig einzelne Neuronen deaktiviert. Dieses Grundprinzip ist in Abbildung 3.14 dargestellt. Um den Gradienten richtig zu berechnen, ist es wichtig, dass die deaktivierten Neuronen auch im Backpropagation beim *Backward Pass* deaktiviert bleiben. Wird das zufällige ausschalten von Neuronen deaktiviert, existiert ein trainiertes Netz, welches implizit mehrere Modelle kombiniert und somit eine deutliche Verbesserung gegenüber bestehenden Regularisierungsmethoden hinsichtlich *Overfitting* darstellt (vgl. Srivastava et al., 2014). Damit die Gewichte durch die Kombination der Modelle im Testlauf nicht zu groß sind, müssen diese mit $1 - p$ skaliert werden. Als Hyperparameter muss für Dropout lediglich die Wahrscheinlichkeit p ein Neuron zu deaktivieren angegeben werden. Typische Werte reichen von 0.1 bis 0.5 (vgl. Krizhevsky et al. (2012), Srivastava et al. (2014) oder Simonyan und Zisserman (2014)). Wird die Standardinitialisierung in Verbindung mit ReLu-Funktionen verwendet, muss darauf geachtet werden, dass die Neuronen positive Ausgaben erzeugen, um ebenso positive Gradienten für das Lernen zu erhalten. Dies kann mittels des Schwellwertes $b = 1$ oder einer hinreichend großen Varianz erreicht werden (vgl. Hinton et al., 2012).

Die beschriebene Methode kann auch bestehende, gut funktionierende Netze verbessern. Da die effektive Kapazität durch Dropout verringert wird, ist es sinnvoll die Anzahl existierender Neuronen n auf $n/(1 - p)$ zu erhöhen (vgl. Srivastava et al., 2014). Aufgrund der Tatsache, dass CNNs die Modellkapazität durch *Parameter Sharing* bereits stark regulieren, ist Dropout in Convolution-Layer nicht gleichermaßen effektiv wie in Hidden-Layer (vgl. Hinton et al., 2012).

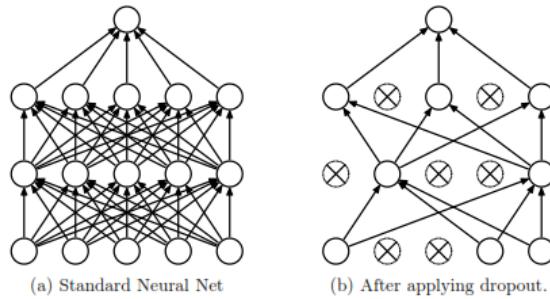


Abbildung 3.14: Schematische Darstellung eines gewöhnlichen MLP (links) und eines Dropout-MLP (rechts) (siehe Srivastava et al., 2014)

Padding im Convolution-Layer

Wird in den Convolution-Layern die Eingabe bzw. die Ausgabe an den Rändern mit $(K-1)/2$ Nullen erweitert (Padding), verändert der Convolution-Layer nicht die Größe der Eingaben. Gerade bei der Verwendung von großen Filtermasken, wie beispielsweise $K = 7$, verhindert diese Methode eine schnelle Verkleinerung der Größe der Ausgabe. Damit wird auch verhindert, dass die Informationen an den Rändern der Eingabe früh im Netz verbllassen (vgl. Fei-Fei und Karpathy, 2014).

3.4.3 Erweitern der Trainingsdaten

Die Kapazität eines Modells ist im Allgemeinen eine relative Größe basierend auf der zur Verfügung stehenden Trainingsmenge. Folglich lässt sich das Problem des *Overfittings* auch umgekehrt formulieren: Anstatt die Kapazität zu regulieren, kann die Menge der Trainingsdaten erhöht werden. Diese Methode wird häufig als *Data Augmentation* bezeichnet und umfasst drei Bereiche: Affine Transformationen, Elastische Transformationen und Additives Rauschen. Diese Formen der Regularisierung haben oftmals großen Einfluss auf die Performanz des Systems hinsichtlich Invarianzen und müssen deshalb beim Vergleich unterschiedlicher Algorithmen und Lernmaschinen berücksichtigt werden. (vgl. Bengio et al., 2015, Kap. 7.5, S. 210 f.)

Affine Transformationen

Affine Transformationen können die Generalisierung des Netzes immens verbessern. Eingaben sollen hierbei so verändert werden, dass gewünschte Invarianzen besser trainiert werden. Dies ist auch dann von Vorteil, wenn das Modell selbst bereits gewisse Invarianzen, wie beispielsweise die Translationsinvarianz von CNNs, enthält. So finden affine Transformationen Anwendung

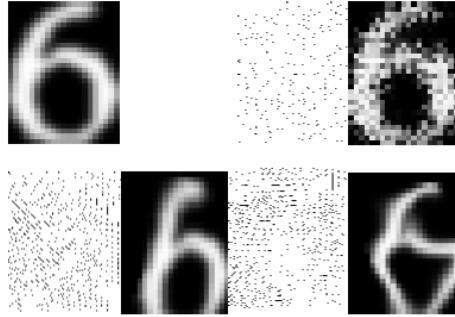


Abbildung 3.15: Originale Eingabe (oben links) und drei Beispiele elastisch veränderter Eingaben mit zugehörigem Vektorfeld (siehe Simard et al., 2003)

im Training von *LeNet 5* von LeCun et al. (1998a) und im *AlexNet* von Krizhevsky et al. (2012). Eine analytische Methode *Data Augmentation* in den Trainingsprozess einzubauen, beschreibt Simard et al. (1992) mittels des Tangent-Prop-Algorithmus.

Elastische Transformationen

Elastische Transformationen erweitern das Set an verfügbaren affinen Operationen und verbessern die Generalisierung dadurch weiter. Zunächst wird ein Vektorfeld pro Dimension generiert, welches zufällige Verschiebungen der einzelnen Eingabemerkmale (z. B. Pixel) beschreibt. Die generierten Felder werden anschließend mit einem Gauß-Kern gefaltet, beziehungsweise weichgezeichnet, und auf die originale Eingabe angewandt (vgl. Simard et al., 2003). Das Ergebnis elastischer Transformationen ist in Abbildung 3.15 dargestellt.

Additives Rauschen

Eine weitere Möglichkeit das System robuster zu machen, ist das Training mit additivem Rauschen. So korrumptiert beispielsweise Vincent et al. (2008) die Trainingsdaten zu einem gewissen Grad mit additiven Rauschen, Maskierung oder *Salt-and-Pepper* und konstruiert damit einen *Denoising Autoencoder* (DAE). Mehrere DAEs bilden zusammen *Stacked Denoising Autoencoder* und schließen damit die Lücke zu den beschränkten Boltzmann-Maschinen und (DBNs) im Bereich des unüberwachten Vortrainings (vgl. Vincent et al., 2010).

Gerade die Maskierung einzelner Merkmale im Eingaberaum liefert die Grundlage für das spätere Dropout, welches sowohl für die Hidden-Layer als auch im Input-Layer geeignet ist und die Generalisierung deutlich verbessern kann (vgl. Srivastava et al., 2014).

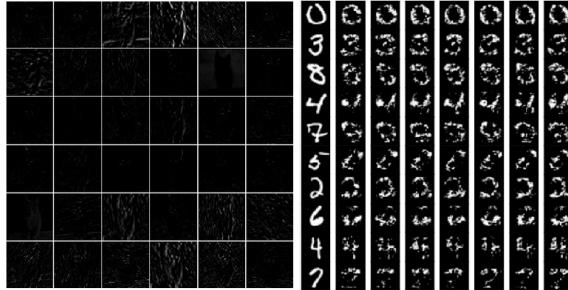


Abbildung 3.16: Aktivierungen des *AlexNet* nach erstem Layer (links) (siehe Fei-Fei und Karpathy, 2014) und Rekonstruktionen eines Autoencoders (rechts) (siehe Vincent et al., 2010)

3.5 Visualisierungsmethoden

Gerade die Verbindung von Bilderkennung und CNNs macht die Anwendung von Visualisierungsmethoden besonders attraktiv, da sich gewisse Verhaltensweisen des neuronalen Netzes so besser verstehen lassen. Die meisten der vorgestellten Methoden haben das Ziel der Kritik, *Features* aus neuronalen Netzen ließen sich nicht interpretieren, entgegenzuwirken (vgl. Zeiler und Fergus, 2014).

3.5.1 Primitiv

Die einfachste Möglichkeit des Sichtbarmachens des Neuronalen Netzes, ist die Visualisierung der Ausgabe beziehungsweise der Gewichte und wird deshalb als primitiv bezeichnet. Dennoch können diese Techniken wichtige Einblicke gewähren.

Neuronen-Aktivierung

Die Visualisierung der Ausgabe eines *AlexNet* von Krizhevsky et al. (2012) ist in Abbildung 3.16 (links) dargestellt. Die Eingabe ist ein Bild einer Katze. Es fällt sofort auf, dass die Aktivierungen sehr dünnbesetzt (*sparse*) sind, was letztendlich auf die ReLu-Funktionen zurückzuführen ist (vgl. Glorot et al., 2011).

Diese Visualisierungsmethode kann in mehrerer Hinsicht unterstützen. Einerseits können so tote Neuronen identifiziert werden (vgl. *Dying ReLu*-Effekt in Maas et al. (2013)). Andererseits kann überprüft werden, ob die Aktivierungen die gewünschte *Sparsity* und Lokalität aufweisen. Wird ein Autoencoder verwendet, kann darüber hinaus interessant sein, die Aktivierung beziehungsweise Ausgabe des Netzes zu visualisieren, da diese der Rekonstruktion der

Eingabe entsprechen. Abbildung 3.16 (rechts) zeigt derartige Rekonstruktionen auf dem MNIST-Datensatz.

Faltungskerne

Eine weitere einfache Möglichkeit, Informationen über das CNN zu erhalten ist die Visualisierung der gelernten Filtermasken. Abbildung 3.17 zeigt die Faltungskerne des ersten Layers eines trainierten *AlexNet*.



Abbildung 3.17: Die Faltungskerne des ersten Layers eines trainierten *AlexNet* (siehe Krizhevsky et al., 2012)

Trainierte Netze haben typischerweise glatte, Gabor-Filtern ähnliche Filtermasken. Verrauschte Filtermasken können hingegen entweder ein Indikator für eine fehlende Regularisierung und damit einhergehendes *Overfitting* sein oder darauf hinweisen, dass das Netz nicht ausreichend konvergiert ist (vgl. Fei-Fei und Karpathy, 2014).

3.5.2 Gradientenbasiert

Eine weitere Klasse von Visualisierungsmethoden arbeiten auf Basis des Backpropagation-Algorithmus. Dies bedeutet, dass sie sich der Fehlerrückführung (*Backward Pass*) bedienen und diese zur Visualisierung verwenden.

Neuronen-Visualisierung

Die erste Methode ist die sogenannte Deconvnet-Visualisierung von Zeiler und Fergus (2014). Diese kann als Neuronen-Visualisierung bezeichnet werden, da letztlich Aktivierungen einzelner Neuronen visualisiert werden. Die Grundidee des Verfahrens ist es, einzelne Aktivierungen im Netz im Eingaberaum sichtbar zu machen. Dazu wird ein bestimmtes Neuron beziehungsweise eine bestimmte *Feature-Map* ausgewählt und die Aktivierungen werden über die gesamte Testmenge gemessen. Die Beispiele mit den höchsten Aktivierungen werden gespeichert. Im Anschluss nutzt die Visualisierung diese Beispiele und rekonstruiert die erzeugten Aktivierungen, indem alle anderen *Feature-Maps* dieses Layers Null gesetzt werden. Für die Rekonstruktion R der Aktivierung wird im Grunde die Fehlerrückführung des Backpropagation-Algorithmus

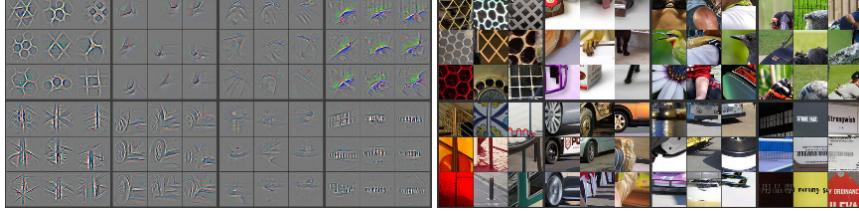


Abbildung 3.18: Mit Deconvnet rekonstruierte Aktivierungen im dritten Layer (links) und dazugehörige originale Eingabebilder (rechts) (siehe Zeiler und Fergus, 2014)

verwendet. Damit gilt in einem Convolution-Layer die Gleichung 3.33 zur Rekonstruktion einer Feature-Map.

$$R_j^l = \sum_{i=0}^n \hat{R}_i^{l+1} * rot180(W_{ij}) \quad (3.33)$$

Im Unterschied zur Berechnung des Gradienten mit Backpropagation, wird bei diesem Verfahren die Aktivierungsfunktion nicht abgeleitet. Stattdessen wird die Aktivierungsfunktion ϕ selbst mit der Rekonstruktion R^{l+1} im Sinne des *Forward Pass* berechnet. Formel 3.34 verdeutlicht diesen Zusammenhang für den Layer l .

$$\hat{R}_i^{l+1} = \phi_l(R_i^{l+1}) \quad (3.34)$$

Eine weitere Besonderheit stellen die Pooling-Layer dar, da diese im Prinzip nicht umkehrbar sind. Die von Zeiler und Fergus (2014) vorgeschlagene Methode sieht deshalb lediglich die Variante Max-Pooling vor und verbietet alle anderen Pooling-Verfahren. Max-Pooling bietet hierbei den Vorteil, dass im *Forward Pass* die Stellen der Maxima vermerkt werden.

Acht ausgewählte Neuronen im dritten Layer mit den jeweiligen zur höchsten Aktivierung korrespondierenden Eingabebildern, sind in Abbildung 3.18 dargestellt.

Saliency-Visualisierung

Die sogenannte Saliency-Visualisierung von Simonyan et al. (2013) ist dem Deconvnet-Verfahren im Grunde sehr ähnlich. Es unterscheidet sich allerdings in einem zentralen Punkt. Während das Verfahren von Zeiler und Fergus (2014) Veränderungen im Vergleich zum Backpropagation-Algorithmus vornimmt, hält sich dieses Verfahren strikt an die Fehlerrückführung und berechnet damit die Ableitungen der Aktivierungsfunktionen, wie Formel 3.35 zeigt.

$$\hat{R}_i^{l+1} = R_i^{l+1} \circ \phi'(z_i) \quad (3.35)$$



Abbildung 3.19: Originale Eingabebilder (oben) und dazu gehörige *Saliency-Maps* (unten) (siehe Simonyan et al., 2013)

Dies bietet den Vorteil, dass sich das Verfahren somit sowohl auf die Convolution-Layer als auch die Hidden-Layer anwenden lässt. Dies lässt sich damit begründen, dass das Verfahren im ursprünglichen Sinne einzelne Klassen im Eingaberaum visualisiert (vgl. Simonyan et al., 2013).

Abbildung 3.19 zeigt die beschriebene Technik, angewandt auf Beispiele aus dem *ImageNet*-Datensatz. Die *Saliency-Map* zeigt die Rekonstruktion der entsprechenden Aktivierung im Output-Layer.

3.5.3 Nachverarbeitung

Methoden im Bereich Nachverarbeitung sind dadurch charakterisiert, dass ein trainiertes MLP, beziehungsweise CNN, lediglich zur Erzeugung von Merkmalsvektoren oder zur Berechnung von Ausgaben herangezogen wird. Diese Art der Visualisierung beschränkt sich demnach auf den Kontext und betrachtet das Netz als *Blackbox*, beziehungsweise als eine Funktion $h(x)$.

t-SNE

Die Methode t-Distributed Neighbor Embedding (t-SNE) beschreibt ein von Maaten et al. (2008) entwickeltes Optimierungsverfahren zur Dimensionsreduktion. Es verbessert das klassische SNE-Verfahren insofern, dass die Gradienten für die Optimierung einfacher zu berechnen sind. Darüber hinaus wird anstelle einer Gauß- eine Student-t-Verteilung als Ähnlichkeitsmaß im niedrigdimensionalen 2D oder 3D Zielraum verwendet (vgl. Maaten et al., 2008). Der Grundmechanismus von t-SNE ist die Minimierung des Unterschieds zweier Wahrscheinlichkeitsverteilungen (Kullback-Leibler-Divergenz): Der Wahrscheinlichkeitsverteilung von Paaren im hochdimensionalen P Raum sowie der im niedrigdimensionalen Raum Q . Damit ergibt sich die zu minimierende Zielfunktion C in Formel 3.36.

$$C = KL(P||Q)) = \sum_i \sum_j p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right) \quad (3.36)$$



Abbildung 3.20: t-SNE Einbettung einer Auswahl an *ImageNet*-Bildern auf Basis der Aktivierung des letzten Layers im *AlexNet* (Bild: Laurens van der Maaten, Facebook AI Research)

Grundsätzlich können die von neuronalen Netzen erzeugten Merkmalsvektoren (*Features*) auch mittels der Hauptkomponentenanalyse (PCA) visualisiert werden. Die t-SNE unterscheidet sich von PCA allerdings in einigen zentralen Eigenschaften (vgl. hierzu und im Folgenden Maaten et al., 2008):

- Wenn ein Großteil der Varianz nicht in den ersten zwei, beziehungsweise drei, Hauptkomponenten beschrieben ist, kann t-SNE bessere Ergebnisse liefern, da es versucht die gesamte Information abzubilden.
- t-SNE ist im Vergleich zur PCA keine orthogonale, lineare Transformation sondern eine nichtlineare Reduktion mit nicht zwingend orthogonalen Komponenten.
- t-SNE findet, aufgrund der nicht-konvexen Zielfunktion, nicht immer zum globalen Minimum.
- t-SNE ist darauf spezialisiert hochdimensionale Daten auf maximal zwei beziehungsweise drei Dimensionen zu reduzieren.

Abbildung 3.20 zeigt die t-SNE 2D-Transformation des Merkmalsvektors vor dem Output-Layer eines *AlexNet*. Als Eingabedaten dient eine Teilmenge des *ImageNet*-Datensatzes. Diese Art der Visualisierung zeigt, dass das CNN Merkmale extrahiert und darauf aufbauend relevante von irrelevanten Merkmalen trennen kann. Dies lässt sich sehr deutlich daran erkennen, dass inhaltlich ähnliche Bilder näher zusammen sind als andere und somit Kategorien oder Gruppen und nicht Hintergrund oder Färbung die Daten charakterisieren (vgl. Bell und Bala, 2015).



Abbildung 3.21: Mögliche Rekonstruktionen des originalen Eingabebilds (oben links) (siehe Simard et al., 2003)

Inceptionism

Ein weiteres Verfahren, welches als *Inceptionism*³ bezeichnet wird, wählt eine andere Herangehensweise. Durch Nutzung dieses Verfahrens wird versucht, ausgehend von einer Aktivierung, beziehungsweise einem Merkmalsvektor eines Layers, eine dazu passende Eingabe zu rekonstruieren. Das Grundverfahren, um Merkmalsvektoren zu invertieren, stammt von Mahendran und Vedaldi (2014). Dieses Verfahren nimmt ein beliebiges Eingabebild x_0 und berechnet mittels eines trainierten CNN die Ausgabe $h_0 = h_l(x_0)$ in einem beliebigen Layer l . Das Ziel des Verfahrens ist es, die Zielfunktion C in Gleichung 3.37 mit Regularisierer $R(x)$ zu minimieren und ausgehend vom Rauschen ein optimales x zu finden.

$$C = \|h(x) - h_0\|^2 + \lambda R(x) \quad (3.37)$$

Der Regularisierer sorgt dafür, dass das optimale x innerhalb eines für Bilder üblichen Intervalls $B = [-128, 128]$ liegt (Formel 3.38) und das resultierende Bild stückweise konstant ist (Formel 3.39).

$$R_1 = \|x\|_6^6 \quad (3.38)$$

$$R_2 = \sum_{i,j} [(x_{i,j+1} - x_{i,j})^2 + (x_{i+1,j} - x_{i,j})^2]^{\frac{1}{2}} \quad (3.39)$$

Abbildung 3.21 zeigt fünf Optimierungen auf Basis des Eingabebilds oben links.

³Das Kunstwort *Inceptionism* geht auf einen Blog-Eintrag von *Research at Google* zurück (<http://googleresearch.blogspot.de/2015/06/inceptionism-going-deeper-into-neural.html> (26.08.2015)).

Kapitel 4

Implementierung des Prototyps

Nachdem in den vergangenen Kapiteln die theoretischen Grundlagen zu Convolutional Neural Networks (CNNs) gelegt wurden, widmet sich dieses Kapitel der Implementierung des Prototyps *ConvNetCPP*. Es bildet somit die Basis für das darauffolgende Kapitel, in welchem die verschiedenen Aspekte des Prototyps hinsichtlich Deep Learning experimentell untersucht werden. Der Prototyp setzt folgende Systemparameter voraus:

Systemvoraussetzungen und Abhängigkeiten:

- Python 3.4
- SciPy — Scientific Python (siehe Jones et al. (2001))
- C++11
- Eigen-Bibliothek für Lineare Algebra (siehe Guennebaud et al. (2010))
- OpenMP kompatibler Compiler (siehe OpenMP Architecture Review Board (2008))
- Optional: Boost Bibliotheken (siehe Schling (2011))

ConvNetCPP unterstützt insgesamt folgende Funktionen:

- | | |
|---------------------------|--------------------------------|
| • SGD | • L1-/L2-/Max-Norm-Penalty |
| • Nesterov Momentum (NAG) | • Dropout |
| • Rprop/RMSprop | • MAX-/AVG-/Stochastic-Pooling |
| • Equilibrium SGD | |
| • AdaDelta | • Ausgabe- <i>Padding</i> |

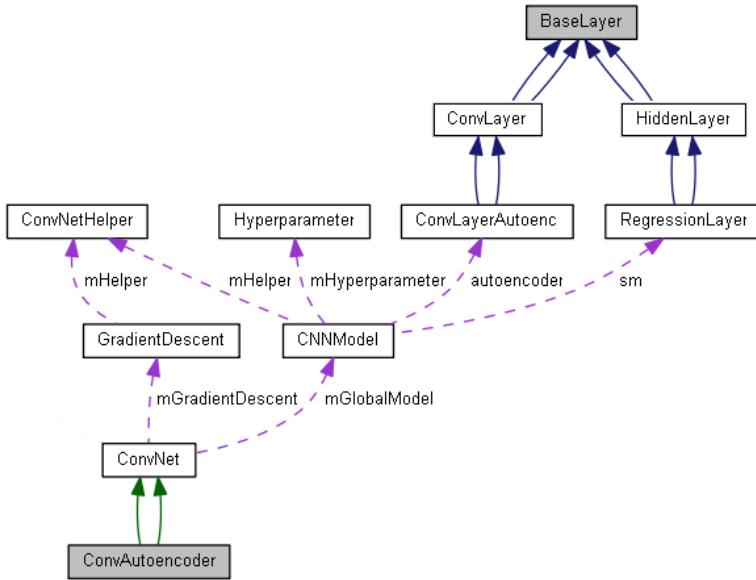


Abbildung 4.1: Klassenabhängigkeitsgraph des Prototyps (*ConvNetCPP*)

- Early Stopping
- Sig/Tanh/ReLU
- Softmax-Regression
- CE/MSE-Fehlermaß
- Conv. Denoising Autoencoder
- Neuronen-Visualisierung
- Ausgabe-Visualisierung
- Kernel-Visualisierung

4.1 Architektur

Die Architektur von *ConvNetCPP* besteht aus zwei Teilen. Den in C++ implementierten Klassen sowie einem in C geschriebenen *Python Module Extension*, welches die Schnittstelle zu Python darstellt.

4.1.1 C++ ConvNet

Der C++ Teil ist aufgeteilt in 9 Klassen, welche die unterschiedlichen Teile des CNN implementieren. Die Abhängigkeiten der Klassen sind in Abbildung 4.1 dargestellt.

Die Klasse ConvNetHelper beinhaltet die für ein CNN benötigten Basisfunktionalitäten wie das Drehen von Matrizen, Parallelisierung und die Konvertierung zwischen Vektoren und 3D-Daten für den Übergang zwischen Convolution- und Hidden-Layer.

Optional kann die Abhängigkeit zu den Boost-Bibliotheken durch Angabe des Compiler-Flag `no_serialization` aufgehoben werden. Damit wird die

Serialisierung deaktiviert und der Status des Trainings nicht mehr länger persistiert. Dies bietet sich insbesondere unter Verwendung der Python API an, da hierbei der aktuelle Status des Models in Form einer 2D Datenstruktur an Python zurückgegeben werden.

Die vier zentralen Klassen `CNNModel`, `BaseLayer`, `ConvNet`, `ConvAutoencoder` und `GradientDescent` werden im Folgenden genauer beschrieben.

CNNModel.cpp

Die Klasse `CNNModel` repräsentiert das eigentliche CNN. Es beschreibt die Architektur und hält die Instanzen der entsprechenden Schichten des Netzwerks. Die Kommunikation zwischen den einzelnen Schichten geschieht mittels *Messages* (vgl. Kapitel 2.6.2). Diese *Messages* werden durch die zwei Structs in Listing 4.1 beschrieben. Eines definiert die Kommunikation bei der Berechnung der Netzaktivierung (*Forward Pass*). Hierbei werden nur Pointer übergeben, da die entsprechenden Ausgaben pro Schicht für die Berechnungen des Gradienten weiter benötigt werden und dadurch keine Kopie der Eingabe pro Schicht erzeugt wird. Die *Message* für die Rückpropagierung des Fehlers (*Backward Pass*) wird nicht weiter benötigt, weshalb die Daten an die vorherige Schicht übergeben werden können.

```
struct ForwardMessage{
    vector<const vector<MatrixXd*>> vecData;
    MatrixXd* matData;
};

struct BackwardMessage{
    vector<vector<MatrixXd>> vecData;
    MatrixXd matData;
};
```

Listing 4.1: Definitionen der Structs für die Kommunikation zwischen den einzelnen Schichten des CNN

BaseLayer.cpp

Die abstrakte Klasse `BaseLayer` deklariert mehrere virtuelle Methoden, die von allen abgeleiteten Klassen implementiert werden müssen. Dies sind die für den Backpropagation-Algorithmus notwendigen Methoden für *Forward Pass* und *Backward Pass* sowie Methoden zur Visualisierung. Darüber hinaus implementiert die Klasse die verschiedenen Aktivierungsfunktionen mit zugehörigen Ableitungen und Methoden zur Erzeugung gleich- oder normalverteilten Gewichten.

ConvNet.cpp

Die Klasse **ConvNet** bildet die Schnittstelle zum CNN nach außen und beinhaltet die gesamten Methoden zum Training, Prädiktion, Validierung und Visualisierung. Die Methoden arbeiten auf Basis einer Instanz des **CNNModel**, welche außerhalb erzeugt werden muss und dem Konstruktor übergeben wird. Damit sind Steuerungslogik und Modelarchitektur voneinander getrennt.

ConvAutoencoder.cpp

Die Klasse **ConvAutoencoder** bildet die Schnittstelle zum Convolutional Autoencoder nach außen und beinhaltet die gesamten Methoden zum Vortraining, Prädiktion und Validierung. Die Klasse ist von **ConvNet** abgeleitet und arbeitet deshalb ebenso auf Basis einer Instanz des **CNNModel**, welche außerhalb erzeugt werden muss und dem Konstruktor übergeben wird.

GradientDescent.cpp

Die Klasse **GradientDescent** implementiert den Gradientenabstieg und die verschiedenen Adaptionen wie Momentum und adaptive Lernraten. Darüber hinaus ist in dieser Klasse auch die Max-Norm-Regularisierung implementiert, welche die Parameter unmittelbar nach einem Update auf die entsprechende Norm prüft und skaliert. Sind die partiellen Ableitungen berechnet, wird die Instanz des **CNNModel** an **GradientDescent** übergeben und die entsprechenden Gewichtsupdates werden ausgeführt. Etwaige weitere Algorithmen, wie das Schätzen der aktuellen Krümmung (siehe Equilibrium SGD), sind ebenfalls in dieser Klasse implementiert.

4.1.2 Python Module Extension

Python Module Extensions erlauben es in C/C++ implementierte Software in Python zur Verfügung zu stellen. Sie bilden die Standardvorgehensweise, um Python mit nativem Code zu erweitern.¹

Im Rahmen des Prototyps kann die CNN Funktionalität somit bequem durch das bereitgestellte Python Module verwendet werden, ohne auf die wichtige Performanz von nativem Code verzichten zu müssen. Der folgende Teil beschreibt die grundlegende Konzeption der API und gibt einen Überblick der zur Verfüigung stehenden Funktionen. Die Funktionen des CNN und des Convolutional Autoencoders sind aufgrund deren gemeinsamen Codebasis in einem Modul *ConvNet* enthalten.

¹Siehe dazu weitere Details in der offiziellen Python Dokumentation: <https://docs.python.org/3.4/extending/extending.html> (1.9.2015)

Modellbeschreibung

Im ersten Schritt wird das Modell spezifiziert. Für den Lebenszyklus eines Modells stehen die beiden Methoden `reset()` und `create()` zur Verfügung. Erstere Methode löscht bestehende Modelle, während letztere ein neues Modell erstellt. Mehrere Modelle sind bereits vordefiniert. Spezielle Architekturen können mittels einer Konfigurationsdatei (vgl. Datei `/py/Modeldefinition.py`) spezifiziert werden und als String der Methode `create()` übergeben werden. Dazu gehören Anzahl der Schichten, Neuronen und Zielklassen, Aktivierungsfunktionen, Dropout, Pooling, Padding sowie das Fehlermaß.

Eine weitere zur Beschreibung gehörende Methode ist `setHyperParameters()`. Diese erlaubt das Überschreiben der Standardparameter. Folgende Hyperparameter werden bereitgestellt:

- INITIALIZATION
- INIT_STD_DEVIATION
- ALGORITHM
- VISUALIZATION
- THREAD_COUNT
- TRAINING_COST_PERIOD
- FEATUREMAP_DROPOUT
- MATRIX_CONVOLUTION
- ENABLE_INFO_OUTPUT
- ENABLE_DEBUG_OUTPUT

Ist die Modellspezifikation abgeschlossen, kann ein zum Modell passender Parametersatz mit der Methode `loadState()` geladen werden.

Training

Bevor das Training gestartet werden kann, müssen die entsprechenden Trainingsparameter übergeben werden. Dies geschieht mittels der Methode `setTrainingParameters()`. Folgende Parameter müssen angegeben werden:

- MAXITERATIONS
- BATCHES
- LRATE
- LRATE_ALPHA
- L2_REG
- L1_REG
- MAXNORM_REG
- MOMENTUM
- LIMIT
- EARLY_STOPPING
- VALIDATION_PERIOD
- TRAIN_VALID_RATIO

Das Training wird mit der Methode `fit()` gestartet. An dessen Ende werden die besten Parameter des Modells zurückgegeben und können gespeichert

werden. Das Training kann so zu einem späteren Zeitpunkt fortgesetzt oder für Testzwecke verwendet werden.

Unüberwachtes Vortraining

Für unüberwachtes Vortraining sind speziell drei Hyperparameter wichtig, welche mittels der Methode `setAutoencoderParameters()` übergeben werden:

- TRAINING_LAYER
- FIX_FIRST_CONV_LAYER
- INPUT_DROPOUT_RATE

Das unüberwachte Vortraining wird durch den Aufruf der Methode `autoencoder_fit()` gestartet, wobei die entsprechend zu trainierende Schicht angegeben werden muss. Im Anschluss werden die besten Parameter des Modells ebenfalls zurückgegeben und die gelernten Gewichte können dadurch als Initialisierung verwendet werden. Durch den Hyperparameter `FIX_FIRST_CONV_LAYER` können die Gewichte der ersten Schicht des Netzwerks fixiert werden, um ein Überschreiben während des Trainings zu vermeiden.

Validierung

Zur Validierung des Modells werden die beiden Methoden `check()` und `autoencoder_check()` angeboten. Diese liefern Informationen über die aktuellen Gradienten je Schicht, etwaige Hesse-Approximationen sowie über die Fehlrate auf Test und Validierungsdaten.

Prädiktion

Die Methoden `predict()` und `autoencoder_predict()` berechnen für beliebige neue Daten die entsprechende Prädiktion.

Visualisierung

Zur Visualisierung des Modells stehen mehrere Methoden bereit. Zum einen primitive Methoden, die lediglich die Ausgabe und trainierte Parameter anzeigen: `getOutput()` und `getKernel()`. Darüber hinaus existieren auch komplexere gradientenbasierte Methoden. Diese umfassen einerseits die Visualisierung einzelner Neuronen (`visualizeNeuron()`) oder der gesamten Ausgabe einer Schicht (`visualizeOutput()`). Die entsprechende Visualisierungsmethode wird als Hyperparameter definiert. Daneben kann die Methode `visualizeSample()` dazu verwendet werden, korrumptierte Ausgaben zu visualisieren.

4.2 Implementierungsdetails

Dieser Teil beschreibt wichtige Details in der Implementierung. Die beschriebenen Techniken vereinfachen die Berechnungen teilweise, beschleunigen diese oder können für das Debugging verwendet werden.

4.2.1 Vereinfachungen

Die Implementierung eines CNN lässt sich durch zwei Anpassungen deutlich vereinfachen.

Korrelation statt Faltung

Betrachtet man die Formeln des Backpropagation in Kapitel 2.6.2, fallen einige Flip-Operationen ($rot180(\cdot)$) auf. Durch den Zusammenhang von Faltung und Kreuzkorrelation in Formel 4.1 kann so auf einige Flip-Operationen verzichtet werden. Das Symbol \star bezeichnet die algebraische Kreuzkorrelation von Eingangssignal x mit Filtermaske y .

$$x \star y = x * rot180(y) \quad (4.1)$$

Werden alle Faltungen durch Kreuzkorrelationen ersetzt entfallen alle Flip-Operationen bei der Berechnung der partiellen Ableitungen. Die Flip-Operation für $\delta_{message}^l$ bleibt jedoch erhalten.

Reverse Dropout

Eine weitere nützliche Vereinfachung betrifft die Dropout-Methode. Das Standard-Dropout sieht vor im Testbetrieb die Gewichte mit $1-p$ zu skalieren (vgl. Srivastava et al., 2014).

Einfacher ist es allerdings die Testfunktion unverändert zu lassen und die Dropout-Methode gänzlich innerhalb des Trainings zu implementieren. Anstatt die Gewichte im Testbetrieb, also während Mittlung der einzelnen Dropout-Netze, zu verkleinern, werden die Gewichte während des Trainings mit $\frac{1}{1-p}$ vergrößert.

Eine Besonderheit muss allerdings hinsichtlich der Hyperparameter beachtet werden. Durch die vergrößerten Gewichte enthält der Gradient einen gewissen zusätzlichen Faktor, der bei der Wahl der Lernrate berücksichtigt werden muss.²

4.2.2 Vektorisierung

Die Vektorisierung von Code ist hinsichtlich Performanz besonders wichtig. So kann die Verwendung von Vektor-Instruktionen wie des *Intel Streaming*

²Vgl. dazu *Reproduce Geoff Hinton's MNIST dropout results in PyLearn2*: <https://github.com/lisa-lab/pylearn2/issues/193> (1.9.2015)

SIMD Extension 2 die Berechnungen im Vergleich zu Schleifen deutlich beschleunigen. Da *ConvNetCPP* eine Bibliothek für Lineare Algebra verwendet, welche die grundlegenden Funktionen wie Matrix-Matrix-Multiplikation (GEMM) bereits performant implementiert, ist es naheliegend zu versuchen so viele Berechnungen wie möglich auf GEMM zu reduzieren.

Hidden-Layer

In Kapitel 2 wurde bereits vorgestellt wie eine Schicht eines MLP mit $\phi(Wx + b)$ berechnet werden kann. Anstelle die einzelnen Neuronen-Aktivierungen mittels For-Schleife zu berechnen, wird die gesamte Schicht mit einer GEMM-Operation berechnet.

Da im Rahmen von Deep Learning meist *Mini-Batch*-Training angewandt wird, lässt sich die Vektorisierung zusätzlich noch auf mehrere Beispiele erweitern. Im Falle eines Hidden-Layers werden die einzelnen Schichten mit Formel 4.2 berechnet, wobei die Matrix X einen *Mini-Batch* mit i Beispielen als Spaltenvektoren und die Matrix B den um i erweiterten Vektor mit Schwellwerten b repräsentiert:

$$\phi(WX + B) \quad (4.2)$$

Dies lässt sich analog auf den *Backward Pass* sowie die Berechnung des Gradienten übertragen. Damit lassen sich alle Berechnungen im Hidden-Layer jeweils mit GEMM-Operationen durchführen.

Convolution-Layer

In einem CNN fällt die meiste Rechenzeit in den Convolution-Layern an, weswegen es sich besonders lohnt auch hier die Berechnungen nach Möglichkeit auf GEMM-Operationen zu reduzieren (vgl. im Folgenden Abuzaid et al., 2015).

Der integrale Bestandteil eines Convolution-Layer ist die Berechnung der Faltung bzw. Kreuzkorrelation. Durch einen einfachen Trick lässt sich diese Berechnung für ein Trainingsbeispiel in eine GEMM-Operation überführen. Diese berechnet gleichzeitig alle *Feature-Maps* aus allen *Input-Maps*, wie Abbildung 4.2 anschaulich darstellt.

Die Reduktion funktioniert, indem einerseits die verschiedenen Filtermasken ausgerollt werden und andererseits Filtermasken, die zu einer *Feature-Map* gehören hintereinander in einen Zeilenvektor konkateniert werden. Mehrere solcher Zeilenvektoren werden zu einer Matrix F_m zusammengefasst. Außerdem müssen die einzelnen Bildausschnitte, welche mit den Masken gewichtet werden, ebenfalls ausgerollt und jede *Input-Map* untereinander in einen Spaltenvektor konkateniert werden. Jede gültige Stelle der Eingabe wird so zu einer Matrix I_m zusammengefasst. Als Ergebnis erhält man zwei Matrizen die mittels GEMM berechnet werden können. Die Berechnung eines

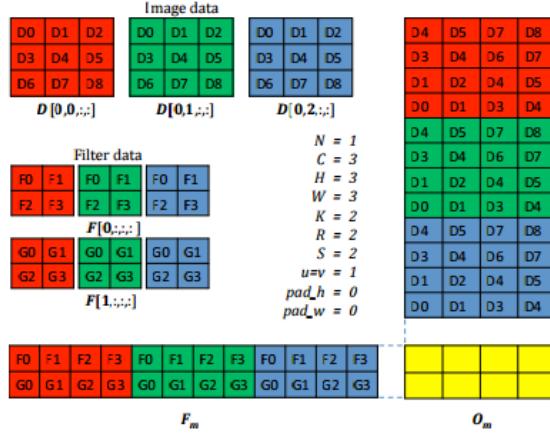


Abbildung 4.2: Schema der Faltung als Matrix-Matrix-Multiplikation (siehe Chetlur et al. (2014))

Convolution-Layer wird so durch Formel 4.3 beschrieben, wobei entsprechend erweiterte Schwellwerte B zur Anwendung kommen.

$$\phi(F_m I_m + B) \quad (4.3)$$

Um wieder die Struktur der ursprünglichen Eingabe zu erhalten, müssen die resultierenden Zeilenvektoren O_m nochmals entsprechend umorganisiert werden. Diese Methode lässt sich auch auf *Mini-Batch*-Training erweitert, indem mehrere Trainingsbeispiele analog zum Hidden-Layer spaltenweise konkateniert werden.

Grundsätzlich können Faltungsoperationen performant mittels der schnellen Fourier-Transformation im Frequenzbereich berechnet werden. Für die Anwendung im Bereich CNNs ist diese Methode allerdings nicht unbedingt von Vorteil, da beispielsweise die Filtergröße im Vergleich zur Eingabegröße gerade in den ersten Schichten sehr viel kleiner ist oder die Faltung mit größeren Schrittweiten ($Stride > 1$) angewandt wird (vgl. Chetlur et al., 2014).

4.2.3 Parallelisierung

Betrachtet man die Verteilung der Rechenressource innerhalb eines CNN, fällt auf, dass 90 % der Rechenzeit, aber nur 5 % der Parameter auf die Convolution-Layer entfällt (vgl. Krizhevsky, 2014). Daher bietet es sich an, die in Krizhevsky (2014) vorgeschlagenen Strategien zu verwenden und zwar Daten-Parallelisierung in den Convolution-Layern und Modell-Parallelisierung in den Hidden-Layern.

Aus Gründen der Einfachheit und aufgrund der Tatsache, dass der CPU-basierte Prototyp für große Netze ohnehin nicht geeignet ist, wird auf Model-

Parallelisierung verzichtet und das gesamte Netzwerk auf Basis der Daten parallelisiert. In *ConvNetCPP* basiert die Parallelisierung auf OpenMP und damit auf parallel ausgeführten For-Schleifen. Ein *Mini-Batch* wird somit auf die zur Verfügung stehende Threads aufgeteilt. Die Strategie ist unterteilt in drei Schritte.

1. Initialisierung — Jeder Thread bekommt eine Kopie der Modellarchitektur, inklusive eigenem Speicher für Gradient und Aktivierungen.
2. Parallelisierung — Der aktuelle *Mini-Batch* wird auf die einzelnen Threads aufgeteilt und die Netzaktivierung (*Forward Pass*) sowie die Gradientenberechnung (Backward Pass) werden lokal ausgeführt.
3. Zusammenführung — Die berechneten Gradienten werden über die Threads gemittelt und die Parameter aktualisiert.

4.2.4 Debugging

Die Implementierung eines CNN oder eines Convolutional Autoencoder ist sehr fehleranfällig und im Allgemeinen schwierig zu debuggen. Eine sehr nützliche Methode die Richtigkeit der berechneten Gradienten in neuronalen Netzen zu überprüfen ist die Finite-Differenzen-Methode (vgl. z. B. Bouvrie (2006)). Die Approximation zweiter Ordnung an die partiellen Ableitungen liefert eine bessere Genauigkeit und kann mit Formel 4.4 berechnet werden (vgl. Bengio (2012)).

$$\frac{\partial J}{\partial w_i} \approx \frac{J(w_i + \epsilon) - J(w_i - \epsilon)}{2\epsilon} \quad (4.4)$$

Mittels dieser Methode kann die Richtigkeit der im Convolutional Autoencoder (siehe Kapitel 3.2.2) gemachte Anpassung erfolgreich überprüft werden.

Kapitel 5

Experimente

Trotz der vielen unterschiedlichen Techniken und Methoden aus dem letzten Kapitel, sind die Modelle Selektion und die Wahl der richtigen Methoden weiterhin sehr problemspezifisch.

Um für ein Lernproblem ein passendes Modell zu finden, beschreiben Bengio und LeCun (2007) drei essentielle Komponenten, die mit entsprechendem Vorwissen spezifiziert werden müssen:

1. Die Repräsentation und Vorverarbeitung der Daten
2. Die Architektur der Maschine oder des Neuronalen Netzes
3. Die Optimierung der Fehlerfunktion sowie Regularisierung

Dieses Kapitel gliedert diese drei Komponenten in die Bereiche Modelle Selektion, Training und Visualisierung. Als Datenmaterial (vgl. Abbildung 5.1) dienen die beiden folgenden bekannten Datensätze:

- MNIST¹ — Der MNIST-Datensatz handgeschriebener Ziffern besteht aus 60000 Trainingsbeispielen und 10000 Testbildern. Es stellt eine Untermenge des größeren NIST-Datensatzes dar. Die binarisierten Bilder sind 28×28 Pixel groß und die Ziffern zentriert. Richtwerte für Fehlerraten liegen mit *Data Augmentation* bei 0.23 % (Ciresan et al. (2012)), mit Vortraining bei 0.53 % (Jarrett et al. (2009)) und ohne Vorverarbeitung bei 0.7 % (Ranzato et al. (2006)). Das beste Ergebnis erreicht das *DropConnect Network* von Wan et al. (2013) mit 0.21 % Fehler ohne *Data Augmentation*.
- CIFAR-10² — Der CIFAR-10-Datensatz besteht aus 50000 32×32 Pixel großen Trainingsbeispielen und 10000 Testbildern. Die Bilder sind aufgeteilt in 10 Klassen: Flugzeug, Auto, Vogel, Katze, Rotwild, Hund,

¹MNIST-Datensatz: <http://yann.lecun.com/exdb/mnist/> (10.09.2015)

²CIFAR-10-Datensatz: <http://www.cs.toronto.edu/~kriz/cifar.html> (10.09.2015)



Abbildung 5.1: CIFAR-10 Beispiele (links) und MNIST-Beispiele (rechts)

Frosch, Pferd, Schiff und Lastwagen. Richtwerte für Fehlerraten liegen mit *Data Augmentation* bei 9.3 % (Wan et al. (2013)), ohne bei 11.7 % (Goodfellow et al. (2013b)). Mit unüberwachten Vortraining erreichen Masci et al. (2011) 21.8 %, ohne 22.5 % Fehler. Mit dem Standard Cuda-Convnet werden ohne *Data Augmentation* und Vorverarbeitung ein Fehler von 18 % erreicht³. Das beste Ergebnis mit 8.2 % Fehler erzielt das *Deeply-Supervised Net* von Lee et al. (2014) mittels *Data Augmentation*.

5.1 Modelselektion

Typischerweise bestehen die zur Verfügung stehenden Daten aus Trainingsdaten und Testdaten. Um die verschiedenen Ergebnisse zu vergleichen, werden von den vorhandenen Trainingsdaten zufällige Beispiele entnommen. Wie Abbildung 5.2 zeigt, bilden diese Daten die Validierungsdaten, welche zur Überwachung des Trainings und zur Bestimmung der aktuellen Performanz verwendet werden. Die Testdaten werden verwendet, um das Modell mit anderen Modellen zu vergleichen und dürfen nicht in die Validierung einbezogen werden (vgl. z. B. Hastie et al., 2009, S. 222). Eine Erweiterung zu diesem Verfahren ist die Kreuzvalidierung, wobei die Trainingsdaten in k Blöcke unterteilt werden und das Training mehrfach mit unterschiedlichem Block zur Validierung durchgeführt wird. Die Performanz des Modells ergibt sich aus dem Durchschnitt der einzelnen Trainings (vgl. z. B. Bengio et al., 2015, Kap. 7.2, S. 219).



Abbildung 5.2: Aufteilung der Trainingsdaten in Trainings- und Validierungsdaten

³Cuda-Convnet Projekt: <https://code.google.com/p/cuda-convnet/> (10.09.2015)

5.1.1 Modelarchitektur

Anfang der 2000er war es gängige Praxis die Anzahl der Parameter entsprechend der Menge an Trainingsdaten zu wählen (vgl. Zander, 2001, S. 95 f.). Dies ist beispielsweise auch am bekannten *LeNet 5* von LeCun et al. (1998a) mit 60.000 freien Parametern für das Training des MNIST-Datensatzes zu erkennen. Neuere vorgestellte Architekturen beinhalten deutlich mehr freie Parameter (vgl. z. B. Andrade, 2014). Aufgrund besserer Techniken zur Regularisierung, wie beispielsweise Dropout, ist es möglich eine bedeutend größere Zahl an Gewichten als Trainingsdaten zu verwenden, ohne dabei mit *Overfitting* konfrontiert zu werden (vgl. Bengio, 2012). Aufgrund empirischer Untersuchungen hat sich herausgestellt, dass die frühere angewandte Pyramidenform für die Layer-Größen (z. B. *LeNet 5*: 6-16-120/84-10)⁴ suboptimal ist und gleichbleibende Größen der Schichten zu bevorzugen sind (vgl. Larochelle et al., 2009).

Dieser Teil stellt die für die Experimente herangezogenen Architekturen der Modelle vor. In allen Experimenten werden, aufgrund ihrer guten Ergebnisse, die ReLu-Funktionen als Aktivierung und Max-Pooling als Pooling-Methode verwendet (vgl. z. B. Krizhevsky et al. (2012) und Simonyan und Zisserman (2014)). Außerdem wird standardmäßig kein Padding angewandt. Dies bedeutet, dass die Größe nach einem Convolution-Layer mit Filtergröße $K \times K$ auf $N - K + 1 \times N - K + 1$ schrumpft.

LeNet 5+

Das erste vorgestellte Architektur ist ein erweitertes *LeNet 5* und orientiert sich an dem von Ranzato et al. (2006) vorgestellten Modell. Es wird mit 50-50/200-10 abgekürzt und umfasst im Detail die folgenden sechs Schichten:

1. Convolution-Layer: 50 *Feature-Maps* mit 5×5 Filtermasken
2. Pooling-Layer: 2×2 Filtermasken
3. Convolution-Layer: 50 *Feature-Maps* mit 5×5 Filtermasken
4. Pooling-Layer: 2×2 Filtermasken
5. Hidden-Layer: 200 Neuronen
6. Output-Layer: 10 Neuronen mit Cross-Entropy Fehlermaß

Dieses Netz wird für das Training der MNIST-Daten verwendet und umfasst für die Eingabe der Größe $28 \times 28 \times 1$ 63.750 Gewichte in den Convolution-Layern und 162.000 Gewichte im MLP. Insgesamt enthält es somit 223.950 Gewichte.

⁴Die Darstellung der Modellarchitektur in der Form 6-16-120/84-10 steht für drei Convolution-Layer mit 6, 16, 120 *Feature-Maps* und einem Hidden-Layer mit 84 sowie einem Output-Layer mit 10 Neuronen. Zur Klassifikation wird stets die Softmax-Regression angewandt.

Net-7

Das zweite vorgestellte Architektur besitzt eine zusätzliche Schicht und besteht folglich aus sieben Schichten. Es orientiert sich an den Modellen von Hinton et al. (2012) und Zeiler und Fergus (2013). Das Modell wird mit 64-64-64/64-10 abgekürzt und umfasst im Detail die folgenden sieben Schichten:

1. Convolution-Layer: 64 *Feature-Maps* mit 5×5 Filtermasken
2. Pooling-Layer: 2×2 Filtermasken
3. Convolution-Layer: 64 *Feature-Maps* mit 5×5 Filtermasken
4. Pooling-Layer: 2×2 Filtermasken
5. Convolution-Layer: 64 *Feature-Maps* mit 5×5 Filtermasken
6. Hidden-Layer: 64 Neuronen
7. Output-Layer: 10 Neuronen mit Cross-Entropy Fehlermaß

Dieses Netz wird für die CIFAR-10-Daten verwendet. Aufgrund der größeren Eingabe von 32×32 ist es möglich, drei Convolution-Layer zu verwenden, da nach zwei Convolution- und Pooling-Layern noch die Eingangsgröße von 5×5 für den dritten Convolution-Layer verbleibt. Dieses Netz umfasst für die Eingabe der Größe $32 \times 32 \times 3$ 209.600 Gewichte in den Convolution-Layern und 10.496 Gewichte im MLP. Insgesamt enthält es somit 220.096 Gewichte.

In diesem Zusammenhang ist die Analyse von Zeiler und Fergus (2014) von Bedeutung. Hier wird festgestellt, dass zum einen die Gesamttiefe des Netzes wichtig ist und zum anderen die Größe der Hidden-Layer keinen großen Einfluss auf die Performanz hat. Dariüber hinaus hat die Größe der mittleren Convolution-Layer enormen Einfluss auf das Ergebnis, während ein zu großer Hidden-Layer nach den Convolution-Layern ohne Regularisierung zu *Overfitting* führt.

5.1.2 Vorverarbeitung

Die beiden verwendeten Datensätze werden auf verschiedene Arten vorverarbeitet. Somit werden mehrere Versionen erzeugt. Zunächst wird der Wertebereich beider Datensätze auf Werte zwischen 0 und 1 skaliert. Der MNIST-Datensatz wird im Anschluss lediglich zentriert, indem der Mittelwert pro Pixel subtrahiert wird. Der ebenfalls zentrierte CIFAR-10-Datensatz wird als CIFAR-10A bezeichnet. Eine zweite Variante, CIFAR-10B genannt, wird in den HSV-Farbraum⁵ transformiert und ebenfalls zentriert. *Data-Augmentation* kann die Performanz immens verbessern. Hierauf wird allerdings im Folgenden verzichtet, da nicht die absolute Leistung des Modells

⁵Der HSV-Farbraum bietet den Vorteil, dass die Farbe lediglich im H-Kanal codiert ist und die Kanäle S und V Informationen über Sättigung und Helligkeit enthalten.

von Interesse ist, sondern Unterschiede der einzelnen Aspekte und Methoden vorgestellt werden.

Da im Rahmen der Experimente oftmals *Early Stopping* zum Einsatz kommt, wird die Zentrierung der Daten lediglich mit den echten Trainingsdaten und nicht mit den Validierungsdaten berechnet. Im Anschluss werden die Validierungs- und Testdaten entsprechend transformiert.

5.2 Trainingsmethode

Dieses Kapitel beschreibt die verschiedenen Experimente im Bereich des Trainings. Diese betreffen die Bereiche unüberwachtes Vortraining, Initialisierung, Gradientenabstieg und Regularisierung.

Als Größe für einen *Mini-Batch* wird in Anlehnung an Bengio (2012) immer eine Größe von 40 gewählt. Dies ermöglicht, dass bei 20 Threads jeweils zwei Trainingsbeispiele pro Thread parallel gerechnet werden können. Zur Begrenzung der Trainingszeit sowie zur Vermeidung von *Overfitting* kommt die Methode *Early-Stopping* zum Einsatz. Wie zu Beginn des Kapitels beschrieben, müssen die Trainingsdaten dazu jeweils in ein Trainings- und Validierungsset unterteilt werden. Das Validierungsset umfasst stets 10 % der Trainingsbeispiele und das Trainingsset somit entsprechend 55.000 bei MNIST beziehungsweise 45.500 bei CIFAR-10. *Early-Stopping* wird derart verwendet, sodass das Modell immer fünf Durchläufe über das Trainingsset (Epochen) auf eine Verbesserung des Validierungsfehler wartet und das Training ansonsten abbricht. Im Anschluss gibt es verschiedene Varianten mit den verbliebenen Validierungsdaten umzugehen. Goodfellow et al. (2013b) beschreiben hierfür zwei Strategien: Bei Nutzung der ersten Strategie werden die Daten zur Trainingsmenge hinzugefügt und solange weiter trainiert bis der Validierungsfehler dem des Trainings entspricht. Wird die zweite Strategie verwendet, so wird das gesamte Training neu gestartet und solange trainiert bis der Validierungsfehler dem des Fehlers aus dem *Early-Stopping*-Lauf entspricht.

Im Rahmen dieser Arbeit wird allerdings die etwas praktikablere Methode von Ranzato et al. (2006) angewandt. Die Validierungsdaten werden hierbei ebenso am Ende des Trainings mit den Trainingsdaten kombiniert, jedoch wird maximal weitere 5 Epochen mit der gesamten Trainingsmenge trainiert. Das Training wird früher abgebrochen, falls sich der Validierungsfehler nicht weiter verbessert. Anschließend wird der Fehler auf den Testdaten berechnet, um die Ergebnisse der einzelnen Experimente zu vergleichen.

5.2.1 Vortraining

In diesem Teil soll das unüberwachte Vortraining untersucht werden. Dessen Ziel ist es, die erste Schicht im Modell zu initialisieren (vgl. z. B. Ranzato et al. (2006) und Masci et al. (2011)). Dabei sollen die enthaltenen Gewichte mittels

des in Kapitel 3.2.2 beschriebenen Convolutional Autoencoder trainiert werden. Alle Gewichte werden mittels der Standard-Initialisierung initialisiert (vgl. Kapitel 5.2.2). Die Lernrate muss mit $\eta = 1^{-4}$ sehr klein gewählt werden, da das Modell im Rahmen dieser Experimente ansonsten divergiert.

MNIST

Im ersten Experiment wird kein Input-Dropout, beziehungsweise keine Maskierung vorgenommen. Als Pooling-Methode kommt das Max-Pooling zum Einsatz. Die Trainingsdaten liefert der MNIST-Datensatz mit 55.000 Trainings- und 5.000 Validierungsbeispielen. Bei Standard-Initialisierung und SGD mit einer Lernrate von $\eta = 1^{-4}$ erreicht das Modell nach 39 Epochen ein Minimum mit einem MSE von 0.31 auf den Testdaten.

Abbildung 5.3 zeigt gelernte 5×5 Filtermasken der ersten Schicht auf der linken Seite.

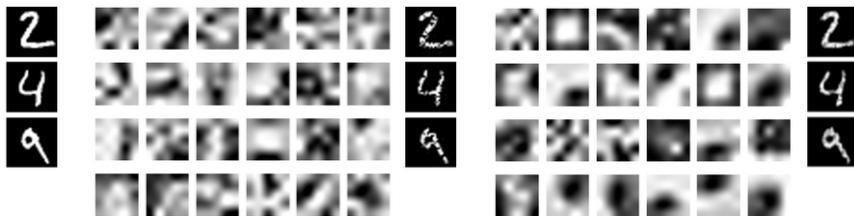


Abbildung 5.3: Mit dem Convolutional Autoencoder trainierte Filtermasken und Rekonstruktionen der ersten *LeNet 5+*-Schicht auf Basis des MNIST-Datensatz. Max-Pooling (links) und Max-Pooling und Maskierung (rechts)

Das zweite Experiment verwendet dieselbe Konfiguration, allerdings wird nun zusätzlich ein Input-Dropout von 30 % verwendet. Nach 14 Epochen erreicht das Modell einen MSE von 2.07 auf den Testdaten. Abbildung 5.3 zeigt die gelernten 5×5 Filtermasken der ersten Schicht auf der rechten Seite. Die Filter des *Denoising Autoencoder* aus dem zweiten Experiment wirken robuster und die Rekonstruktionen glatter. Deshalb werden diese für weitere Experimente beibehalten.

CIFAR-10

Im ersten Experiment wird kein Input-Dropout, beziehungsweise keine Maskierung vorgenommen. Als Pooling-Methode kommt ebenfalls das Max-Pooling zum Einsatz. Die Trainingsdaten stellt der CIFAR-10A/B-Datensatz mit 45.500 Trainings- und 4.500 Validierungsbeispielen. Bei Standard-Initialisierung und SGD mit einer Lernrate von $\eta = 1^{-4}$ erreicht das Modell auf den Testdaten (CIFAR-10A) nach 30 Epochen einen MSE von 74.06 ohne Maskierung und 80.1 mit Maskierung. Mit CIFAR-10B und Maskierung erreicht das Modell nach 20 Epochen einen deutlich besseren MSE von 26.2

auf den Testdaten. Abbildung 5.4 zeigt eine Auswahl der 5×5 Filtermasken der ersten Schicht in Graustufen. Es ist deutlich zu sehen, dass die Filter des CIFAR-10A optisch robuster wirken, während die Filter des CIFAR-10B sehr verrauscht sind.

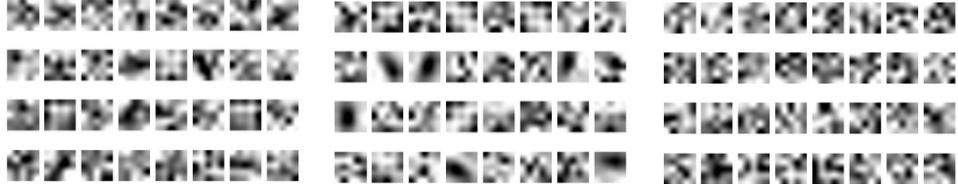


Abbildung 5.4: Mit dem Convolutional Autoencoder trainierte Filtermasken der ersten *Net-7*-Schicht auf Basis des CIFAR-10-Datensatz. CIFAR-10A: Max-Pooling (links), Max-Pooling und Maskierung (mitte) CIFAR-10B: Max-Pooling und Maskierung (rechts)

Für die weiteren Experimente werden jeweils die Filter aus dem Training mit Maskierung beibehalten.

5.2.2 Initialisierung

In diesem Abschnitt werden verschiedene Initialisierungsmethoden verglichen. Als Basis dient in allen Beispielen der standardmäßige SGD mit einer *Mini-Batch*-Größe von 40. Als Lernrate wird der universelle Wert $\eta = 0.01$ gewählt und es werden wieder 10 % der Trainingsdaten zur Validierung verwendet. Tabelle 5.1 zeigt die jeweils erreichten Fehlerraten nach einer Epoche Training.

	Standard (mit $\sigma = 0.1$)	Xavier	Vortraining (mit $\sigma = 0.1$)
MNIST	16.2/16.2 %	1.9/6.0 %	12.1/12.2 %
CIFAR-10A	88.0/89.8 % (68.4/72.2 %)	55.0/63.0 % (67.2/71.6 %)	87.2/89.8 % (64.8/68.0 %)
CIFAR-10B	88.0/89.8 % (66.8/71.4 %)	54.6/62.2 %	87.2/89.8 % (64.8/68.0 %)

Tabelle 5.1: Fehler auf den Trainings-/Validierungsdaten nach einer Epoche (Ep.) Training

Standard-Initialisierung

Bei der Standard-Initialisierung werden alle Gewichte mit $W \sim \mathcal{N}(0, 0.01)$ initialisiert. Abbildung 5.5 (mitte) zeigt pro Schicht die Verteilung der Werte im Gradienten über die erste Epoche beim Training des MNIST-Datensatz. Es fällt auf, dass die Varianzen über die verschiedenen Schichten gleich groß sind und somit kein *Vanishing-Gradient*-Effekt erkennbar ist. Auch die

angegebenen Fehlerraten in Tabelle 5.1 zeigen einen deutlichen Trainingsfortschritt.

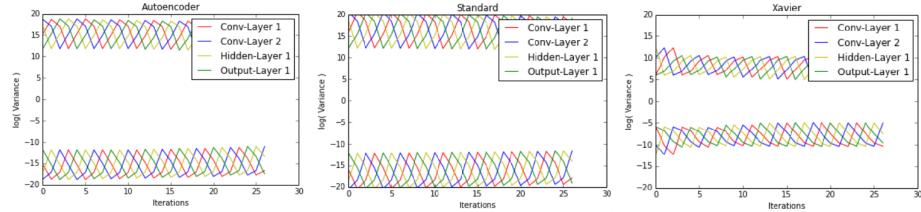


Abbildung 5.5: Log-Varianzen des Gradienten (gespiegelt) der einzelnen Schichten während der ersten Epoche des Trainings auf dem MNIST-Datensatz.

Betrachtet man das Training mit CIFAR-10A/B in Abbildung 5.6, zeigen die Varianzen der Werte im Gradienten ein ähnliches Bild wie beim MNIST. Allerdings sind die Varianzen hier kleiner (siehe Log-Varianz). Außerdem findet während der ersten Epoche kein erkennbarer Trainingsfortschritt statt. Dies deutet darauf hin, dass die Standard-Initialisierung nicht optimal für dieses Problem ist. Werden stattdessen die Gewichte mit $W \sim \mathcal{N}(0, 0.1)$ initialisiert, sind die Varianzen größer und das Netz trainiert, was sich in den Werten in Tabelle 5.1 zeigt.

Xavier-Initialisierung

Bei der Xavier-Initialisierung werden alle Gewichte in Schichten mit ReLU-Aktivierungsfunktionen mittels $W \sim \mathcal{N}(0, \sqrt{2/fan_{out}})$ initialisiert. Die Abbildungen 5.5 und 5.6 (rechts) zeigen die Verteilung der Werte im Gradienten während der ersten Epoche pro Schicht. Im Vergleich zu den anderen Initialisierungstechniken sind die Varianzen insgesamt deutlich größer und das Modell erreicht bereits nach der ersten Epoche mit 6.0 % (MNIST) respektive 63.0 und 64.2 % (CIFAR-10A/B) sehr gute Fehlerraten auf den Validierungsdaten.

Initialisierung durch Vortraining

Nun werden die Gewichte mittels den gelernten Filtern aus Kapitel 5.2.1 initialisiert. Die Abbildungen 5.5 und 5.6 (links) zeigen die Verteilung der Werte im Gradienten über die ersten Epoche pro Schicht. Beim MNIST-Datensatz ist zu erkennen, dass die Varianzen insgesamt zwar kleiner sind als bei der Xavier-Initialisierung, allerdings im Vergleich zur Standard-Initialisierung gegen Ende der Epoche größer werden. In der Folge führt diese Initialisierung zu einem besseren Ergebnis als bei Verwendung der Standard-Initialisierung, was Tabelle 5.1 zeigt. Anders verhält es sich beim CIFAR-Datensatz. Sowohl bei Verwendung der Datensätze CIFAR-10A/B als auch der Standard-

Initialisierung findet in der ersten Epoche kein erkennbares Training statt und die Varianzen sind sehr klein. Analog zur Standard-Initialisierung findet ein Trainingsfortschritt statt, sobald die Standardabweichung im restlichen Netz auf $\sigma = 0.1$ erhöht wird. In diesem Modus erreicht das Netz binnen einer Epoche einen etwas geringeren Fehler als die gesamtheitliche Standard-Initialisierung.

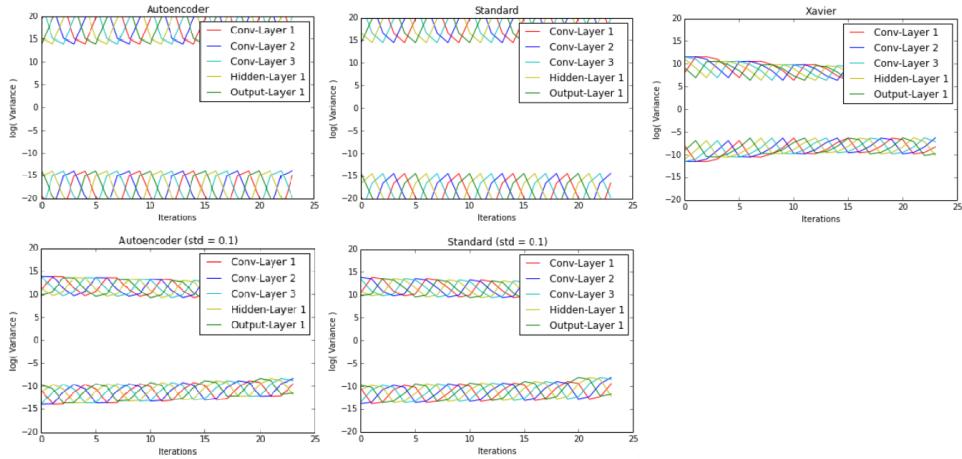


Abbildung 5.6: Log-Varianzen des Gradienten (gespiegelt) der einzelnen Schichten während der ersten Epoche des Trainings auf dem CIFAR-10A-Datensatz.

5.2.3 Gradientenabstieg

In diesem Abschnitt werden die Adaptionen des Gradientenabstiegs untersucht, die im Rahmen dieser Arbeit von Relevanz sind. Zu diesen zählen SGD, Nesterov Momentum, RMSprop, AdaDelta und Equilibrium SGD. Da RMSprop den Equilibrium SGD sehr gut approximiert, bleibt letzterer im Folgenden unberücksichtigt. Im letzten Abschnitt konnte festgestellt werden, dass die Xavier-Initialisierung für beide Probleme eine geeignete Initialisierungsmethode darstellt. Die Experimente in diesem Teil werden deshalb mit dieser Methode initialisiert. Außerdem führt die Vorverarbeitung des CIFAR-10B-Datensatz zu einer konstanten, wenn auch kleinen, Verbesserung der Fehlerraten. Im folgenden wird deshalb lediglich der MNIST und CIFAR-10B-Datensatz betrachtet. Folgende Hyperparameter sind für alle Methoden gleich:

- 90 % Trainings- und 10 % Validierungsdaten
- Lernrate $\eta = 0.01$
- *Early Stopping Patience* von fünf Epochen mit reduzierten Trainingsdaten und eine Epoche beim Training mit den kombinierten Trainings-

	MNIST	CIFAR-10B
SGD	0.40/0.82/0.93 % (21 Ep.)	18.87/30.29/33.31 % (27 Ep.)
Nesterov	0.06/0.12/0.69 % (11 Ep.)	17.77/28.94/34.01 % (34 Ep.)
RMSprop	0.22/0.48/0.89 % (10 Ep.)	29.03/37.74/41.92 % (29 Ep.)
AdaDelta	0.02/0.18/0.61 % (12 Ep.)	20.86/21.99/36.65 % (12 Ep.)

Tabelle 5.2: Fehler auf den Trainings-/Validierungs-/Testdaten nach dem gesamten Training mit verschiedenen Methoden zum Gradientenabstieg

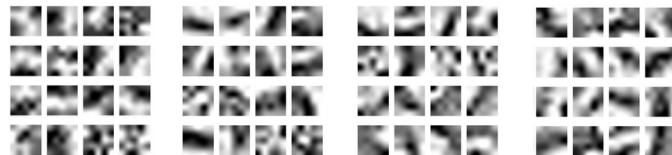


Abbildung 5.7: Beispielhafte Filter der ersten Schicht des *LeNet 5+*: SGD (links), Nesterov (mitte-links), RMSprop (mitte-rechts) und AdaDelta-Methode (rechts)

daten

- Validierung des Modells nach jeweils 1000 Gradientenupdates (*Mini-Batch*-Iterationen)

Im Rahmen der folgenden Experimente ist kein Absenken der Lernrate über das Training vorgesehen. Lediglich beim standardmäßigen SGD wird die Lernrate während des abschließenden Trainings mit kombinierten Daten auf $\eta = 0.001$ gesenkt, da diese Methode selbst keine Möglichkeit zur Veränderung der Lernrate besitzt. Neben dem *Early-Stopping* werden keine weiteren Methoden zu Regularisierung verwendet. Die Tabelle 5.2 zeigt eine Übersicht der jeweils erreichten Fehlerraten auf den Validierungs- und Testdaten.

In Abbildung 5.7 sind Beispiele der Filter der ersten Schicht des *LeNet 5+* nach dem Training mit dem MNIST-Datensatz dargestellt. Die Abbildung 5.8 zeigt beispielhafte Filter der ersten Schicht des *Net-7* nach dem Training mit den CIFAR-10B-Daten in Graustufen.

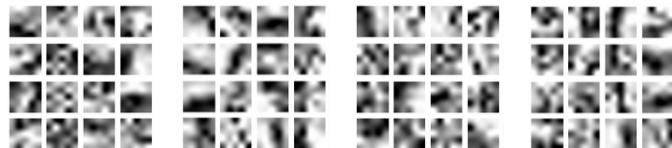


Abbildung 5.8: Beispielhafte Filter der ersten Schicht des *Net-7*: SGD (links), Nesterov (mitte-links), RMSprop (mitte-rechts) und AdaDelta-Methode (rechts)

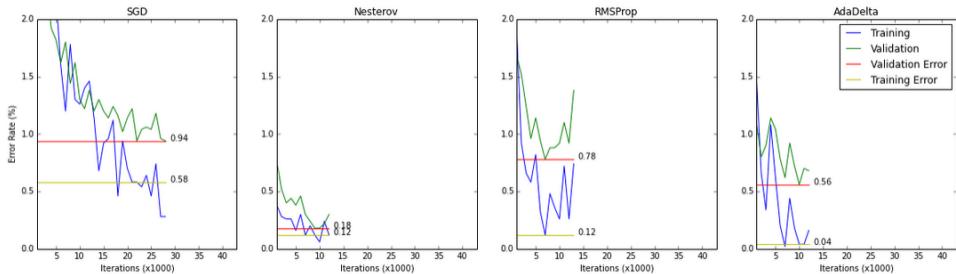


Abbildung 5.9: Trainings- und Validierungsfehler während des Trainings auf dem reduzierten MNIST-Datensatz mittels SGD-, Nesterov-, RMSprop und AdaDelta-Methode

SGD

Das Training auf den MNIST-Daten mit SGD dauert mit 21 Epochen im Vergleich zu den anderen Methoden sehr lange. Auch sind die erreichten Fehlerraten im Vergleich die schlechtesten. Die Abbildung 5.9 zeigt den Verlauf der Fehlerraten über das gesamte Training. Das Training mit den CIFAR-10B-Daten gestaltet sich schwieriger und dauert mit SGD 27 Epochen. Die SGD Methode erzielt im Rahmen dieser Experimente das zweitbeste Ergebnis. Allerdings ist dieses Ergebnis mit einer Fehlerrate von 33.31 % relativ schlecht und das Training mit SGD insgesamt sehr langsam. Die Abbildung 5.10 zeigt den Verlauf der Fehlerraten über das gesamte Training.

Nesterov-Momentum

Für das Training mittels Nesterov-Momentum wird als Hyperparameter für den Momentum-Term exemplarisch $\mu = 0.95$ gewählt (vgl. Sutskever et al., 2013). Das Training auf den MNIST-Daten ist deutlich beschleunigt und endet bereits nach 11 Epochen. Daneben weist es mit einem Testfehler von 0.69 % die zweitbeste Fehlerrate auf. Beim Training mit den CIFAR-10B-Daten fällt auf, dass der Validierungsfehler kleiner ist als beim Training mit SGD, der Testfehler jedoch etwas höher. Das Training dauert mit 34 Epochen in Relation zu den anderen Methoden am längsten. Die Lernkurve ähnelt beim Training mit CIFAR-10B sehr jener des SGD, weist allerdings gegen Ende einen flacheren Verlauf auf und ist insgesamt weniger volatil.

RMSprop

Für das Training mit RMSprop werden folgende Werte exemplarisch als Hyperparameter gewählt (vgl. Dauphin et al., 2015):

- $\rho = 0.90$
- $\mu = 10^{-2}$

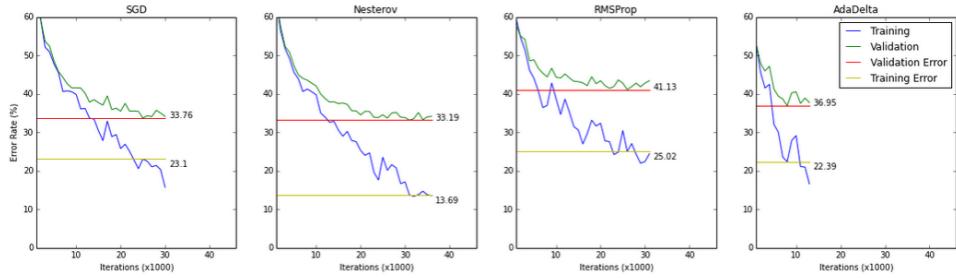


Abbildung 5.10: Trainings- und Validierungsfehler während des Trainings auf dem reduzierten CIFAR-10B-Datensatz mittels SGD-, Nesterov-, RMSprop und AdaDelta-Methode

Das Training mit RMSprop erreicht auf den MNIST-Daten nach 10 Epochen bessere Fehlerraten als der SGD, allerdings nicht so gute Ergebnisse wie die Methoden Nesterov-Momentum und AdaDelta. Beim Training mit CIFAR-10B-Daten erzielt die RMSprop-Methode nach 29 Epochen die schlechtesten Ergebnisse im Rahmen dieser Experimente. Insgesamt weisen die Lernkurven im Vergleich zum Training mit Nesterov-Momentum einen volatileren Verlauf auf. Beim Training mit den MNIST-Daten fällt auf, dass die Fehlerraten nach dem Minimum sich sehr schnell verschlechtern.

AdaDelta

Für das Training mit AdaDelta werden folgende Werte exemplarisch als Hyperparameter gewählt (vgl. Zeiler, 2012):

- $\rho = 0.95$
- $\mu = 10^{-6}$

Das Training mit AdaDelta erreicht auf den MNIST-Daten bereits nach 12 Epochen einen Testfehler von 0.61 %. Damit erzielt die AdaDelta-Methode im Rahmen dieser Experimente die besten Ergebnisse. Ähnlich verhält es sich bei Durchführung des Experiments auf Basis der CIFAR-10B-Daten. Hier erreicht das Training mit der AdaDelta-Methode bereits nach 12 Epochen den besten Validierungsfehler und einen Testfehler im oberen Bereich. Auffällig ist, dass der Validierungsfehler beim Training mit den kombinierten Trainingsdaten sehr schnell fällt und sich damit das Modell rasch an die neuen Daten anpasst. Letztendlich handelt es sich hierbei jedoch um *Overfitting*. Insgesamt weist das Training mit AdaDelta die höchste Volatilität in den Fehlerraten auf.

5.2.4 Regularisierung

Im letzten Abschnitt konnte festgestellt werden, dass der Validierungsfehler deutlich unter dem Testfehler liegt. Während das *LeNet 5+* dennoch sehr gute

	CIFAR-10B
MLP/1-Conv 0.3 % + Max/L2 + Padd.	27.61/27.06/28.41 % (21 Ep.)
MLP 0.3 % + Max + Padd.	16.68/21.51/29.69 % (36 Ep.)
MLP/1-Conv 0.3 % + Max + Padd.	22.48/25.55/29.97 % (32 Ep.)
Dropout MLP 0.3 %	20.51/27.15/33.13 % (20 Ep.)
Max-Norm-Regularisierung	21.64/22.82/35.07 % (18 Ep.)
L2-Regularisierung	22.43/27.97/35.21 % (21 Ep.)
Dropout MLP/1-Conv 0.3 % + Max	27.63/33.87/35.24 % (44 Ep.)
Dropout MLP 0.5 % + Max	25.40/29.96/35.89 % (20 Ep.)
Net-7 ohne Regularisierung	20.86/21.99/36.65 % (12 Ep.)
Dropout MLP/1-Conv 0.5 % + Max	50.97/51.02/51.81 % (27 Ep.)

Tabelle 5.3: Fehler auf den Trainings-/Validierungs-/Testdaten nach dem gesamten Training mit verschiedenen Methoden zur Regularisierung sortiert nach Testfehler

Ergebnisse erzielt, zeigt das *Net-7* bereits früh im Training eine Stagnation der Validierungsfehler. Dieses als *Overfitting* bekannte Phänomen gilt es zu vermeiden. Begründen lässt sich dies damit, da ein solches Modell mit großer Sicherheit zu einem schlechteren Ergebnis auf unbekannten Daten führt, als ein regularisiertes Vergleichsmodell. In diesem Abschnitt gilt es daher entsprechende Techniken zur Regularisierung zu untersuchen.

Als Basis dient das *Net-7* sowie der CIFAR-10B-Datensatz. Für den Gradientenabstieg wird die AdaDelta-Methode mit den Hyperparametern aus den vorhergegangenen Experimenten eingesetzt, da diese Methode gute Ergebnisse mit kurzer Trainingszeit erzielt. Als Referenzergebnis dienen 21.99/36.65 % Fehlerrate auf den Validierungs-/Testdaten. Auf die L1-Regularisierung wird im Folgenden verzichtet, da die erwünschte *Sparsity* der Aktivierungen bereits durch die Verwendung von ReLu-Aktivierungsfunktionen erzielt wird (siehe. Kapitel 3.4.1). In Tabelle 5.3 sind die Ergebnisse der Experimente sortiert nach Testfehler aufgeführt.

L2-Regularisierung

In diesem Experiment wird dem Modell die L2-Regularisierung hinzugefügt. Die Regularisierung wird dabei exemplarisch mit Parameter $\alpha = 0.0005$ in die Zielfunktion mit aufgenommen (vgl. Krizhevsky et al., 2012). Wird das Training wiederholt, so zeigt diese Art der Regularisierung den gewünschten Effekt. Die Tabelle 5.3 lässt einen erhöhten Validierungsfehler von 27.97 % und einen etwas verbesserten Testfehler von 35.21 % erkennen. Betrachtet man die Lernkurven in Abbildung 5.11 zeigt das Training mit L2-Regularisierung allerdings ein sehr ähnliches Verhalten wie ohne Regularisierung und weist gerade in der Mitte des Trainings starkes *Overfitting* auf. Insgesamt verlängert

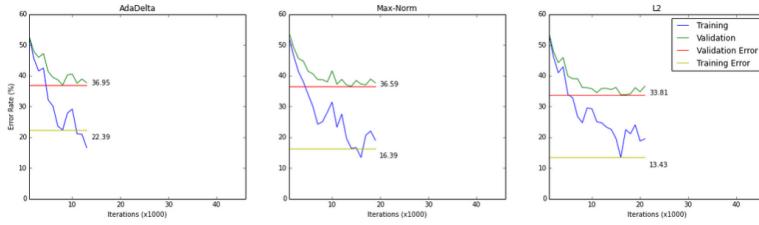


Abbildung 5.11: Trainings- und Validierungsfehler während des Trainings auf dem reduzierten CIFAR-10B-Datensatz mittels AdaDelta-Methode, Max-Norm- und L2-Regularisierung

sich die Trainingszeit um 9 Epochen.

Max-Norm-Regularisierung

Im zweiten Experiment wird eine maximale Norm für die Gewichte festgelegt. Diese beträgt exemplarisch $\|w\|_2 \leq 3$ (vgl. Srivastava et al., 2014). Diese Art der Regularisierung hat weniger Einfluss auf das Training als die L2-Regularisierung. Dennoch wird der Validierungsfehler bei gleichzeitiger Verbesserung des Testfehlers um 1.58 % etwas verschlechtert. Darüber hinaus ist der Testfehler in etwa gleich groß wie jener der L2-Regularisierung. Das Training mit Max-Norm-Regularisierung verhindert zu Beginn des Trainings eine Überanpassung, wie Abbildung 5.11 zeigt. Das Training wird durch dies Regularisierung um 6 Epochen verlängert und *Overfitting* tritt weiterhin sehr stark im Verlauf des Trainings auf.

Dropout

Die dritte betrachtete Methode zur Regularisierung ist das Dropout-Training. In diesen Experimenten werden zwei unterschiedliche Konfigurationen untersucht. Zuerst wird Dropout nur im Hidden-Layer des MLP angewandt. Anschließend wird die Dropout-Methode sowohl im Hidden-Layer des MLP als auch im letzten Convolution-Layer angewandt. Als Dropout-Raten werden exemplarisch 0.3 % und 0.5 % untersucht (vgl. Srivastava et al. (2014)). Im ersten Experiment wird Dropout nur im MLP verwendet. Bei beiden Dropout-Raten ist die Auswirkung auf das Training deutlich zu erkennen und der Unterschied zwischen Validierungs- und Testfehler fällt kleiner aus. Das bessere Ergebnis erreicht die Dropout-Rate von 0.3 %, wodurch der Validierungsfehler auf 27.15 % verschlechtert und der Testfehler gleichzeitig auf 33.13 % verbessert wird. Ein Dropout von 0.5 % führt hingegen zu einer Verschlechterung des Testfehlers. Betrachtet man die Lernkurven in Abbildung 5.12 fällt auf, dass die Netze nach wie vor großes *Overfitting* aufweisen. Anders verhält es sich bei der zusätzlichen Anwendung von Dropout im letzten Convolution-Layer. Das Modell wird hierbei stärker regularisiert,

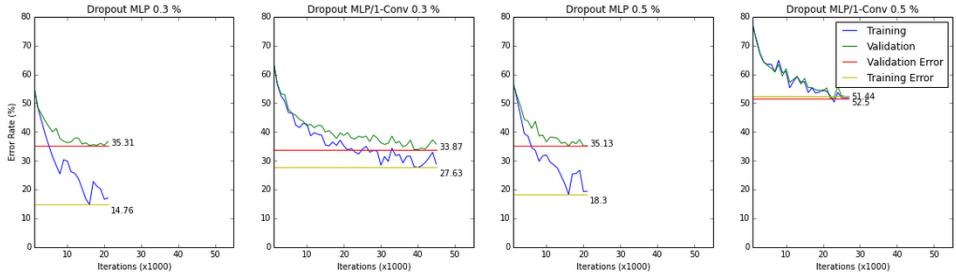


Abbildung 5.12: Trainings- und Validierungsfehler während des Trainings auf dem reduzierten CIFAR-10B-Datensatz mittels Dropout

was zu einer weiteren Verringerung des Abstandes zwischen den Fehlerraten führt. Allerdings verlängert sich die Trainingszeit deutlich. Hierbei führt eine Dropout-Rate von 0.3 % zu einem etwas besseren Ergebnis als das Training ohne Regularisierung. Eine Dropout-Rate von 0.5 % hingegen beschränkt das Modell sehr stark und das Ergebnis verschlechtert sich. Die Lernkurven weisen bei Dropout in mehreren Schichten jedoch deutlich weniger *Overfitting* auf.

Padding

Die letzten Experimente zeigen, dass sowohl die L2-Regularisierung als auch das Dropout-Training zu einem kleinerem Abstand zwischen dem Validierungs- und Testfehler führen können. Die folgenden Experimente sollen die Auswirkung von Padding in den Convolution-Layern zeigen. Zusätzlich zum Padding wird die Dropout-Regularisierung mit einer Rate von 0.3 % sowie die Max-Norm- und L2-Regularisierung aus den letzten Experimenten verwendet. Durch Padding wird die Größe der Eingabe nur in den Pooling-Layern verringert. Damit erhöht sich die Dimension der Ausgabe des letzten Convolution-Layers von 64 auf 1024, was zu deutlich mehr trainierbaren Gewichten im MLP führt.⁶

Wie das erste Experiment zeigt, steigt hierdurch die Kapazität des Modells deutlich und der Testfehler verbessert sich bei Dropout im MLP auf 29.69 %. Im Vergleich zum Modell ohne Padding tritt am Ende außerdem ein größeres *Overfitting* auf, durch Nutzung von Dropout allerdings weniger als im Modell ohne Regularisierung. Insgesamt zeigen die Lernkurven in Abbildung 5.13 einen ähnlichen Verlauf wie jene des Trainings ohne Padding. Der größere Hidden-Layer im MLP wird oftmals durch die Max-Norm-Regularisierung regularisiert, was zu einem volatileren Verlauf im Trainingsfehler führt. Wird

⁶Hier wird in den Convolution-Layern aus Gründen der Implementierung Ausgabe-Padding eingesetzt und die CIFAR-10B-Eingaben werden entsprechend vor dem Training auf die Größe 36×36 vergrößert. Außerdem entfällt somit das Padding der Ausgabe des letzten Convolution-Layers.

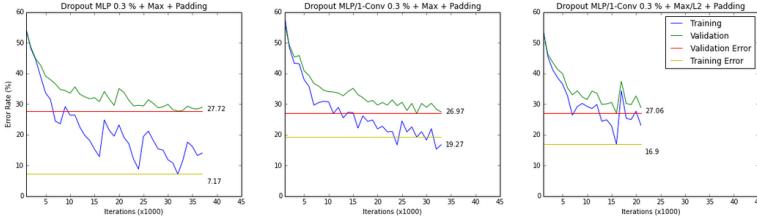


Abbildung 5.13: Trainings- und Validierungsfehler während des Trainings auf dem reduzierten CIFAR-10B-Datensatz mittels Dropout und Padding

	CIFAR-10B
<i>Net-8</i> MLP/1-Conv 0.3 % + L2/Max	21.92/23.32/27.50 % (25 Ep.)
<i>Net-8</i> MLP/1-Conv 0.3 % + Max	16.15/21.20/27.55 % (27 Ep.)
<i>Net-8</i> MLP 0.3 % + Max	13.30/13.26/27.84 % (25 Ep.)
<i>Net-7-Small</i> + L2	30.33/31.02/35.35 % (37 Ep.)
<i>Net-7</i> ohne Regularisierung	20.86/21.99/36.65 % (12 Ep.)
<i>Net-7-Small</i> + Max	29.85/31.86/36.69 % (20 Ep.)

Tabelle 5.4: Fehler auf den Trainings-/Validierungs-/Testdaten nach dem gesamten Training mit *Net-7-Small* und *Net-8* unter Verwendung verschiedener Methoden zur Regularisierung sortiert nach Testfehler

das Dropout-Training zusätzlich im letzten Convolution-Layer angewandt, wird das *Overfitting* deutlich verringert, allerdings verschlechtert sich der Testfehler etwas. Auffällig ist, dass das Modell hierbei weniger regularisiert wird als ohne Padding. Wird zusätzlich die L2-Regularisierung aktiviert, wird das *Overfitting*, bei gleichzeitiger Verbesserung des Testfehlers und verkürzter Trainingszeit, deutlich reduziert.

Modellkapazität

Durch die Kombination von Padding, Dropout und Max-Norm- sowie L2-Regularisierung kann ein gewünschtes Ergebnis mit guter Generalisierung erzielt werden. Die folgenden Experimente sollen die Auswirkung der Modellkapazität und damit die Größe des Modells aufzeigen. Dazu werden zwei neue Modelle eingeführt, wobei das ein im Vergleich zum *Net-7* eine Vergrößerung darstellt, das andere eine Verkleinerung. Beide neuen Netze verwenden ebenfalls Ausgabe-Padding.

Die Ergebnisse der einzelnen Experimente werden in Tabelle 5.4 sortiert nach Testfehler aufgeführt.

Net-7-Small

Das erste Netz, welches genutzt werden soll, besitzt keinen Hidden-Layer und die Anzahl der *Feature-Maps* ist auf 20 reduziert. Es orientiert sich am Modell

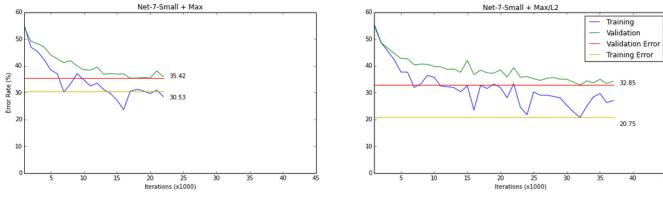


Abbildung 5.14: Trainings- und Validierungsfehler während des Trainings auf dem reduzierten CIFAR-10B-Datensatz mittels *Net-7-Small* sowie Max-Norm- und L2-Regularisierung

von Fei-Fei und Karpathy (2014). Insgesamt besitzt dieses Netz lediglich 24.700 Gewichte, wobei die Architektur durch Padding einen weiteren Pooling-Layer nach dem letzten Convolution-Layer erlaubt. Das Modell wird mit 20-20-20/10 abgekürzt und umfasst im Detail die folgenden sieben Schichten:

1. Convolution-Layer: 20 *Feature-Maps* mit 5×5 Filtermasken
2. Pooling-Layer: 2×2 Filtermasken
3. Convolution-Layer: 20 *Feature-Maps* mit 5×5 Filtermasken
4. Pooling-Layer: 2×2 Filtermasken
5. Convolution-Layer: 20 *Feature-Maps* mit 5×5 Filtermasken
6. Pooling-Layer: 2×2 Filtermasken
7. Output-Layer: 10 Neuronen mit Cross-Entropy Fehlermaß

Die Ergebnisse in Tabelle 5.4 zeigen, dass das verkleinerte *Net-7-Small* nahezu dasselbe Ergebnis wie das *Net-7* erreicht. Gleichzeitig erreicht es eine bessere Generalisierung und benötigt eine kürzere Trainingszeit. Wird jedoch zusätzlich die L2-Regularisierung angewandt, verlängert sich die Trainingszeit hinsichtlich der durchlaufenen Epochen immens. Die Abbildung 5.14 zeigt auf, dass die Lernkurve hierbei einen flacheren Verlauf aufweist. Insgesamt verringert die Verkleinerung des Netzes das *Overfitting* deutlich und führt bei zusätzlicher L2-Regularisierung zu einem besseren Testfehler.

Net-8

Beim zweiten Netz werden die Anzahl der *Feature-Maps* sowie die Anzahl der Neuronen im Hidden-Layer erhöht. Es stellt eine etwas kleinere Variante des Modells von Masci et al. (2011) dar. Insgesamt besitzt dieses Netz 510.100 Gewichte. Durch Padding erlaubt die Architektur, wie das *Net-7-Small*, einen weiteren Pooling-Layer nach dem letzten Convolution-Layer, wodurch sich die Anzahl der Schichten auf acht erhöht. Das Modell wird mit 100-100-100/100-10 abgekürzt und umfasst im Detail die folgenden acht Schichten:

1. Convolution-Layer: 100 *Feature-Maps* mit 5×5 Filtermasken

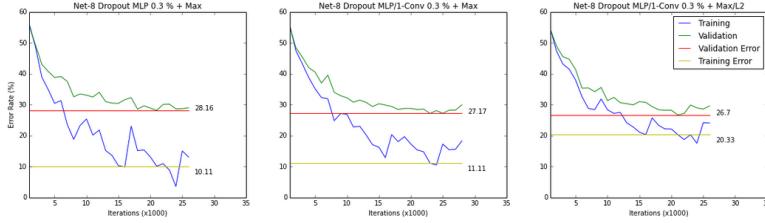


Abbildung 5.15: Trainings- und Validierungsfehler während des Trainings auf dem reduzierten CIFAR-10B-Datensatz mittels *Net-8* mit Max-Norm- und L2-Regularisierung sowie Dropout

2. Pooling-Layer: 2×2 Filtermasken
3. Convolution-Layer: 100 *Feature-Maps* mit 5×5 Filtermasken
4. Pooling-Layer: 2×2 Filtermasken
5. Convolution-Layer: 100 *Feature-Maps* mit 5×5 Filtermasken
6. Pooling-Layer: 2×2 Filtermasken
7. Hidden-Layer: 100 Neuronen
8. Output-Layer: 10 Neuronen mit Cross-Entropy Fehlermaß

Mit dem vergrößerten Netz steigt die Kapazität des Netzes im Vergleich zum *Net-7* mit Padding nochmals, was zu den besseren Fehlerraten führt, die in Tabelle 5.4 aufgeführt sind. Das Netz weist am Ende ebenfalls ein großes *Overfitting* auf und die Lernkurven in Abbildung 5.15 weisen einen ähnlichen Verlauf wie die Kurven im *Net-7* auf. Wird das Netz mit L2-Regularisierung trainiert, verbessert sich die Generalisierung und das Netz erreicht den besten Testfehler im Rahmen dieser Experimente.

5.3 Visualisierung

Neben der Darstellung der Filtermasken existieren zur Visualisierung weitere nützliche Methoden. Im folgenden Teil soll die Anwendung zweier zentraler Techniken zur Visualisierung von CNNs gezeigt werden. Einerseits die sogenannte Neuronen-Visualisierung zur Rekonstruktion der Aktivierung eines beliebigen Neurons, andererseits die t-SNE zur Dimensionsreduktion. Darüber hinaus werden mittels des Convolutional Autoencoders die Aktivierungen im CNN untersucht.

5.3.1 Neuronen-Visualisierung

Mit der Neuronen-Visualisierungen können einzelne aktivierte *Feature-Maps* in den Eingaberaum zurück propagiert werden. Bis auf wenige Anpassun-



Abbildung 5.16: Visualisierung der neun stärksten Aktivierungen im *LeNet 5+* mit der Methode von Zeiler und Fergus (2014) auf den gesamten Testdaten: Erste *Feature-Map* in der zweiten Schicht (links) und erste *Feature-Map* in der vierten Schicht (rechts)

gen, funktioniert diese analog zum *Backward Pass* des Backpropagation-Algorithmus.

Im folgenden soll die Methode von Zeiler und Fergus (2014) angewandt werden, um beispielhafte Neuronen zu visualisieren. Abbildung 5.16 zeigt je eine *Feature-Map* der zweiten sowie der vierten Schicht eines mit MNIST-Daten trainierten *LeNet 5+*. Die *Feature-Map* der ersten Schicht fungiert eindeutig als Kantendetektor, was auch bei der Betrachtung der Filtermasken in Abbildung 5.7 plausibel erscheint. Interessanter erscheint die *Feature-Map* der vierten Schicht. Diese scheint gerade Kanten besonders zu detektieren, was bei den Ziffern fünf und sieben besonders auffällt.

Wird dieselbe Methode auf ein mit CIFAR-10B-Daten trainiertes *Net-8* angewandt, ergeben sich die Rekonstruktionen in Abbildung 5.17, wobei jeweils der V-Kanal dargestellt ist. Hier werden jeweils zwei *Feature-Maps* in der zweiten und vierten Schicht gezeigt. Wie auch im *LeNet 5+* fungieren die Filtermasken der zweiten Schicht als Kantendetektoren. Darüber hinaus detektieren sie bei Farbbildern die Tönung des Bildes. Die obere *Feature-Map* der vierten Schicht hingegen scheint, bis auf zwei Ausnahmen, Schiffe zu detektieren. Betrachtet man die Rekonstruktionen, ist erkennbar, dass besonders die länglichen Kanten erkannt werden. Außerdem weist die Rekonstruktion der Aktivierung des oberen Autos starke Ähnlichkeit zu einem Boot auf. Die zweite *Feature-Map* detektiert augenscheinlich längliche Objekte. Diese lassen sich alleine mit dieser *Feature-Map* nicht einzelnen Klassen zuordnen. Dennoch weist diese eine gewisse Relevanz auf, da sowohl drei sehr ähnliche Tarnkappen-Flugzeuge und drei sitzende Katzen detektiert werden.

5.3.2 t-SNE-Methode

Die t-SNE-Methode von Maaten et al. (2008) kann wie die Hauptkomponentenanalyse (PCA) zur Dimensionsreduktion eingesetzt werden. Im Unterschied zur PCA berücksichtigt diese bei der Reduktion alle Hauptkomponenten. An dieser Stelle wird t-SNE verwendet, um die von den

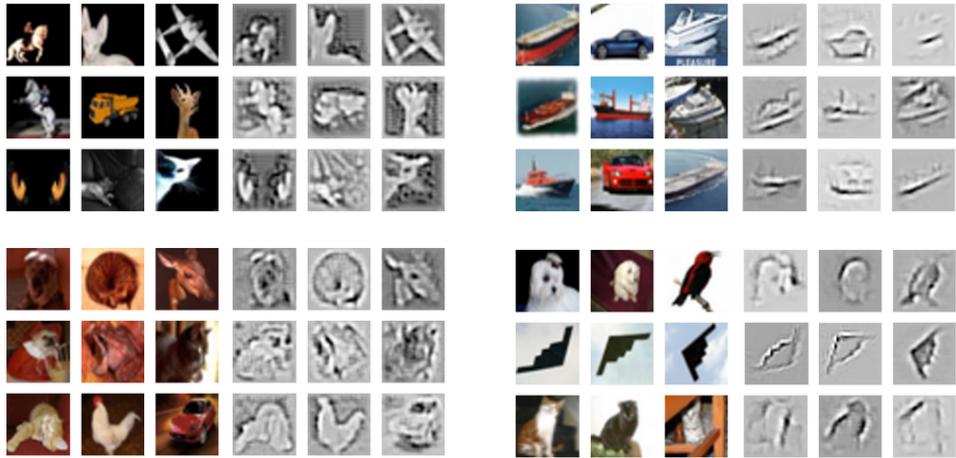


Abbildung 5.17: Visualisierung der neun stärksten Aktivierungen im *Net-8* mit der Methode von Zeiler und Fergus (2014) auf den gesamten Testdaten: Erstes und neuntes Neuron in der zweiten Schicht (links) und erstes und neuntes Neuron in der vierten Schicht (rechts)

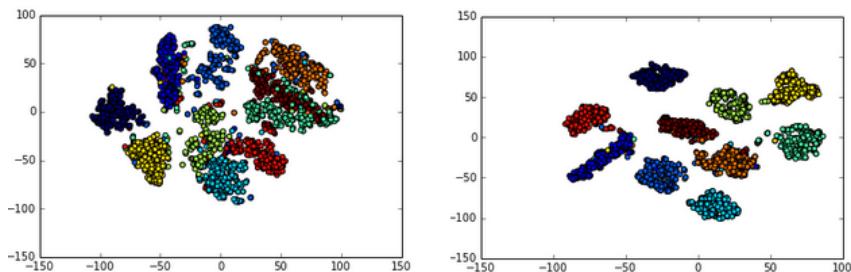


Abbildung 5.18: Visualisierung von 2500 MNIST-Beispielen mit t-SNE im Original (links) und nach der vierten Schicht des trainierten *LeNet 5+* (rechts)

Convolution-Layern extrahierten Merkmale im 2D-Bildraum zu visualisieren. Die t-SNE wird im Folgenden ausgehend von den ersten 50 Hauptkomponenten 1000 Iterationen lange gerechnet.

Abbildung 5.18 zeigt zufällige 2500 originale sowie durch das CNN transformierte MNIST-Beispiele in einem Streudiagramm. Es ist deutlich erkennbar, dass die zehn Klassen nach den beiden Convolution-Layern des *LeNet 5+* kompaktere Gruppen bilden. Dies ermöglicht eine bessere Trennung der Daten im nachfolgenden mit abschließender Softmax-Regression.

Für die Visualisierung der CIFAR-10B-Daten wird eine andere Methode gewählt. An Stelle des Streudiagramms werden die einzelnen Beispiele selbst dargestellt. Die Abbildungen 5.19 und 5.20 zeigen 1000 originale sowie durch das CNN transformierte CIFAR-10B-Beispiele. Hierbei fällt zum einen auf, dass die Originale eindeutig anhand des Hintergrunds getrennt werden,

während die Farbtöne in der Variante des CNNs etwas verwaschener sind. Darüber hinaus sind in der Punktwolke mit den extrahierten Merkmalen Lücken erkennbar. Bei genauer Betrachtung kann, beispielsweise anhand des grünen Autos im Original rechts in der Mitte, die Gruppenbildung nach Inhalt, statt nach Hintergrund oder Farbtönen festgestellt werden. Das genannte grüne Auto befindet sich nach Anwendung der Convolution-Layer bei den anderen Autos links in der Mitte. Außerdem ist die rötliche Gruppe unten-rechts bei Anwendung der t-SNE mit extrahierten Merkmalen aufgelöst.



Abbildung 5.19: Visualisierung von 1000 CIFAR-10-Beispielen mit t-SNE im Original



Abbildung 5.20: Visualisierung von 1000 CIFAR-10B-Beispielen mit t-SNE nach der sechsten Schicht des trainierten *Net-8*

5.3.3 Autoencoder-Visualisierung

Der eingangs zum unüberwachten Vortraining untersuchte Convolutional Autoencoder kann ebenso zur Visualisierung verwendet werden. Dazu wird zunächst die Aktivierung für Beispieldaten in einer bestimmten Schicht berechnet. Anschließend wird diese Aktivierung anstatt des Verfahrens von Zeiler und Fergus (2014) mit dem Decoder des Convolutional Autoencoders in den Bildraum zurück propagiert. Die Schichten mit Max-Pooling werden zwecks Umkehrbarkeit mit Average-Pooling ausgestattet.



Abbildung 5.21: Zufälligen MNIST-Beispiele im Original (links) und visualisiert mit der Autoencoder-Methode auf Basis der Aktivierungen in der vierten Schicht im *LeNet 5+* (rechts)

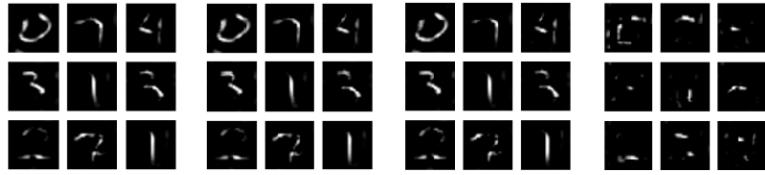


Abbildung 5.22: Mittels der Autoencoder-Methode visualisierte, zufällige MNIST-Beispiele mit verdeckter zehnter *Feature-Map* (links), zwanzigster (mitte-links), den ersten vier (mitte-rechts) und allen ab der vierten *Feature-Map* (rechts) auf Basis der Aktivierungen in der vierten Schicht im *LeNet 5+*

Die Abbildung 5.21 zeigt dieses Verfahren angewandt auf zufällige MNIST-Beispiele, wobei die Rekonstruktionen auffällig gut sind. Durch die ReLU-Aktivierung verschwinden außerdem negative Werte und der Hintergrund ist im Unterschied zur Methode von Zeiler und Fergus (2014) schwarz. Dies ist eine sehr interessante Feststellung, da die verschiedenen Filtermasken nicht mittels Autoencoder trainiert wurden. Darüber hinaus erklärt dies, warum unüberwachtes Vortraining im Deep Learning eine geeignete Methode für das Training der Filtermasken darstellt.

Maskierung von Merkmalen

Die beschriebene Methode erlaubt ein weiteres Experimente hinsichtlich der Aktivierungen einer Schicht. Die Abbildung 5.22 zeigt vier mittels Autoencoder rekonstruierte Aktivierungen. Bei den ersten Rekonstruktionen wurde die zehnte *Feature-Map* der vierten Schicht verdeckt, bei den zweiten Rekonstruktionen die zwanzigste, bei den dritten die ersten vier und bei den vierten alle, außer die ersten vier. Interessanterweise sehen die ersten drei Rekonstruktionen nahezu identisch aus, während jene der vierten Abbildung kaum mehr zu erkennen sind. Dies zeigt, dass sich einzelne *Feature-Maps* nicht an einzelnen Trainingsbeispielen orientieren, sondern, im Sinne der *Distributed Representation*, die Kombination aus Merkmalen wichtig ist (vgl. Hinton, 1986).

In Abbildung 5.23 sind mehrere Kovarianzen mit spaltenweise addiertem Mittelwert dargestellt. Im ersten Bild wird die Kovarianzmatrix aller Aktivierungen der vierten Schicht mit der Ziffer 1 berechnet, im zweiten jene der Ziffer 8 und im dritten die Kombination der Ziffern 1 und 8. Insgesamt besitzt die vierte Schicht 800 Merkmale, wobei jede *Feature-Map* davon 16 bündelt. Zunächst fällt auf, dass bei der Ziffer 1 der Mittelwert der 21. *Feature-Map* die anderen dominiert. Außerdem ist die Information über mehrere Merkmale verteilt, was die erhöhten Kovarianzen innerhalb der Zeile zeigen. Analog verhält es sich bei den Aktivierungen der Ziffer 8. Darüber hinaus sind in der Kombination der Ziffern entsprechend beide Muster in den Aktivierungen

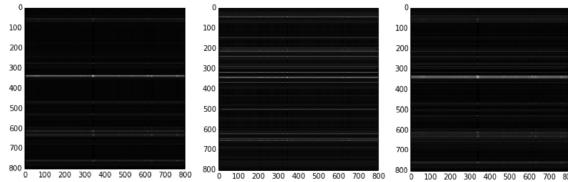


Abbildung 5.23: Kovarianzmatrix der Aktivierungen der vierten Schicht im *LeNet 5+* auf Basis aller MNIST-Beispiele mit der Ziffer 1 (links), der Ziffer 8 (mitte) und den Ziffern 1 und 8 (rechts)



Abbildung 5.24: Neun zufälligen MNIST-Beispielen mit der Ziffer 1 und 7 im Original (links) und visualisiert mit der Autoencoder-Methode auf Basis der gemittelten Merkmale (mitte-rechts) und addierten Merkmale (rechts) nach der vierten Schicht im *LeNet 5+*

erkennbar, wobei die Mittelwerte der 21. und 22. *Feature-Maps* dominieren.

Kombination von Merkmalen

Im letzten Experiment konnte festgestellt werden, dass mehrere *Feature-Maps* bei ähnlichen Eingaben korrelieren. In einem abschließenden Experiment soll die Kombination der Aktivierungen in der vierten Schicht und folglich eine Merkmalskombination visualisiert werden. Die Abbildung 5.24 zeigt einerseits die originalen Eingaben und andererseits, auf Basis der kombinierten Merkmale der Ziffern 1 und 8, die entsprechenden Rekonstruktionen. In den Rekonstruktionen ist eindeutig erkennbar, dass die Mischung von Merkmalen zu einer entsprechenden Vermischung im Eingaberaum führt und somit entartete Kombinationen aus den Ziffern 1 und 8 entstehen. Dabei hat interessanterweise die Stärke der Aktivierung wenig Einfluss auf das Ergebnis, was der Vergleich der Rekonstruktionen mit und ohne Mittelung zeigt.

Weiterführend können auf Basis der Aktivierungen, analog zum Verfahren von Vincent et al. (2010), auch mittels Convolutional Autoencoder synthetische *Samples* erzeugt werden. Diese Methode wird allerdings in dieser Arbeit nicht näher betrachtet.

Kapitel 6

Zusammenfassung

Das Feld von Deep Learning und Convolutional Neural Networks (CNNs) fasst sehr viele interessante Facetten. In dieser Arbeit wurden die Bereiche des überwachten neuronalen Lernmodells zur Klassifikation und Regression aufbereitet. Aufbauend auf das klassische Multilayer Perceptron (MLP), wurden die speziellen Convolutional Neural Networks eingeführt. Diese lassen sich wie gewohnte mehrschichtige Netze mittels Backpropagation-Algorithmus trainieren, bieten jedoch aufgrund ihrer Architektur große Vorteile gegenüber MLPs. Hierbei ist besonders deren Eigenschaft hervorzuheben, lokale Merkmale im Eingaberaum zu berücksichtigen. Darüber hinaus stellt die Entkopplung der Dimensionalität der Eingabe von der Anzahl der trainierbaren Gewichte einen weiteren Vorteil der Convolutional Neural Networks dar. In den darauffolgenden Kapiteln wurde aufgezeigt, warum tiefe Netze schwer zu trainieren sind und wie moderne Methoden im Bereich Deep Learning dabei helfen können, diese Schwierigkeiten zu überwinden. Als die beiden wichtigsten Beispiele wurden das *Overfitting* sowie der *Vanishing Gradient*-Effekt vorgestellt. Als Einstieg in das Thema seien an dieser Stelle die Arbeiten von Bengio (2009) und Bengio (2012) empfohlen. Im weiteren Verlauf der Arbeit wurde, basierend auf den vorgestellten Methoden, ein eigenes CNN implementiert und mit diversen Experimenten auf Richtigkeit überprüft. In diesem Kapitel werden die Ergebnisse nochmals zusammengefasst und ein Ausblick auf weitere mögliche Forschungstätigkeit im Umfeld von CNNs gegeben.

6.1 Ergebnisse

Der im Rahmen dieser Arbeit entwickelte CPU-basierte Prototyp eines CNN konnte durch die Experimente bereits seine Leistungsfähigkeit unter Beweis stellen. Neben den aktuellen Methoden im Deep Learning zur Initialisierung und Optimierung, sind auch nützliche Methoden zur Visualisierung sowie zum unüberwachten Vortraining implementiert worden. Neben einer Python-

Schnittstelle weist das *ConvNetCPP* wenige Abhangigkeiten auf und kann so leicht auf anderen Systemen eingesetzt werden.

Fur die Anwendung ist als Vorverarbeitungsstufe die Zentrierung der Daten besonders wichtig, da ansonsten die Logistische Sigmoidfunktion oder der Tangens Hyperbolicus schnell sattigen konnen. Auch der HSV-Farbraum liefert konsistent etwas bessere Ergebnisse als der RGB-Farbraum. Im Rahmen der Initialisierung kann gefolgert werden, dass die Initialisierung der ersten Schicht durch unuberwachtes Vortraining nicht ausreicht, um die schlechte Initialisierung des restlichen Netzes zu reparieren. Ebenso scheint die Standardabweichung von $\sigma = 0.01$ der Standard-Initialisierung fur das CIFAR-10-Problem zu klein zu sein. Dies zeigt auch das Experiment, bei dem ein sonst Xavier-initialisiertes Netz vortrainiert wird. Hierbei fallt der Validierungsfehler auf CIFAR-10A nach der ersten Epoche von 63 %, bei reiner Xavier-Initialisierung, auf den besseren Wert von 61.8 %. Insgesamt werden die Gewichte des *Net-7* bei Xavier-Initialisierung mit $\sigma \approx 0.035$ in den Convolution-, $\sigma \approx 0.18$ im Hidden- und $\sigma \approx 0.16$ im Output-Layer initialisiert.

Wahrend der Experimente wurde stets eine *Early Stopping Patience* von finf Epochen verwendet und das Training anschlieend mit den gesamten Daten fur maximal finf weitere Epochen fortgesetzt. Dies bedeutet eine starke Beschrankung der Trainingszeit, was sich gerade beim Training mit den CIFAR-10-Daten bemerkbar macht. Hier wirken durch die kurze Trainingszeit die Filtermasken noch recht verrauscht. Trotz der Beschrankung erreicht das *LeNet 5+* einen bemerkenswerten Testfehler von 0.61 %. Wird die Wartezeit beim *Early Stopping* auf 15 Epochen erhoht, erreicht auch das *Net-7* mit *Padding* und angewandtem Dropout von 0.3 % im MLP und dem letzten Convolution-Layer nach 72 Epochen einen Testfehler von 25.02 %. Das stellt im Vergleich zum deutlich groeren Netz von Masci et al. (2011) (100-150-200/300-10) einen plausiblen Wert dar.

Zusammenfassend lassen sich folgende Folgerungen aus den Experimenten ableiten:

- Die Zentrierung der Daten ist obligatorisch.
- Padding erhoht die Kapazitat und verbessert die Performanz ohne markant groeren Rechenaufwand.
- Max-Norm-Regularisierung verhindert *Overfitting* zu Beginn des Trainings und beschleunigt letzteres.
- Dropout und L2-Regularisierung sind gute Verfahren zur Verbesserung der Generalisierung.
- Die L2-Regularisierung dominiert die Max-Norm-Regularisierung.

- Die Anwendung einer starken Regularisierung verlängert die Trainingszeit immens.
- Ein kleiner dimensioniertes Netz kann gleichwertige Ergebnisse wie ein großes, nicht regularisiertes Netz erzielen.
- Methoden zur Visualisierung sind gute Werkzeuge, um erlernte Merkmale sichtbar zu machen und die Funktionsweise von CNNs zu erklären.

Im Rahmen der Experimente kamen bewährte Standardwerte für die Hyperparameter zum Einsatz. Für spezielle Probleme müssen diese durch Kreuzvalidierung oder ähnliche Verfahren bestimmt werden. Ein guter Ausgangspunkt für die Optimierung sind das AdaDelta-Verfahren oder das SGD-ähnliche Nesterov-Momentum ohne adaptive Lernrate, da diese das Training stark beschleunigen können. Für die Bestimmung der optimalen Hyperparameter ist die Rechenkapazität eine restriktive Größe. So ist das Training der MNIST-Daten mit *ConvNetCPP* auf dem *LeNet 5+* mit etwa einer Stunde CPU-Zeit pro Epoche noch recht schnell. Auf dem CIFAR-10-Datensatz in Kombination mit dem *Net-7* dauert das Training mit etwa drei Stunden CPU-Zeit pro Epoche jedoch sehr lange. Damit ist das Training größerer Netze, wie des CIFAR-10-Netz von Masci et al. (2011), sehr zeitaufwändig. Auch sind längere Trainingszeiten von 280 Epochen für das CIFAR-10-Netz von Zeiler und Fergus (2013) nur schwer möglich. In der Praxis sollte mit einem kleinen Netz gestartet und dieses sukzessive ausgebaut werden, bis der Validierungsfehler nicht weiter sinkt oder der Trainingsfehler den Wert Null erreicht. Anschließend sollten die Fehlerraten von Trainings- und Validierungsdaten mit den beschriebenen Techniken der Regularisierung angeglichen werden, um eine bestmögliche Generalisierung zu ermöglichen.

6.2 Ausblick

Betrachtet man die Laufzeiten der verschiedenen Experimente, wird klar, dass auch ein optimiertes CPU-basiertes CNN für große Datensätze und tiefe Architekturen nicht geeignet ist. Durch die Reduktion der 2D-Faltung auf eine Matrix-Matrix-Multiplikation (GEMM), wird die meiste Rechenzeit für solcherlei Multiplikationen aufgewendet. Diese kann durch eine entsprechende GPU-Implementierung deutlich reduziert werden. Darüber hinaus lässt sich das Training eines Modells durch spezielle Modell-Parallelisierung ebenfalls beschleunigen (vgl. Krizhevsky, 2014). Die Beschleunigungen sind auch hinsichtlich erweiterter Trainingsdaten (*Data Augmentation*) zur Steigerung der Performanz von Bedeutung.

Die Performanz eines CNN lässt sich auch durch weitere spezielle Schichten verbessern. An dieser Stelle sei auf die speziellen *Max-Out*-Netze von Goodfellow et al. (2013b) und die *DropConnect*-Netze von Wan et al. (2013) verwiesen,

welche besonders auf dem CIFAR-10-Datensatz hervorragende Ergebnisse liefern. Weiterhin existieren Schichten, welche nach jedem Convolution-Layer zur lokalen Kontrastnormalisierung eingesetzt werden. Dies verhindert, dass einzelne *Feature-Maps* die Aktivierungen der Schicht dominieren (vgl. Jarrett et al. (2009)). Darüber hinaus zeigen neuere Netze eine weitere Adaption der üblichen Convolutional-Layer. Meist werden diese als *locally connected* oder LC-Layer bezeichnet (vgl. Nouri (2013) und Le et al. (2012)). Solche Schichten zeichnen sich dadurch aus, dass lokale rezeptive Felder zwar definiert, die Parameter zwischen den einzelnen Feldern allerdings nicht geteilt werden. Der ursprüngliche Vorteil von CNNs durch *Parameter Sharing* das Modell zu beschränken wird hierdurch verworfen, allerdings kann die größere Kapazität durch anderweitige Regularisierung dennoch zu besseren Ergebnissen führen (vgl. Hinton et al. (2012) und Krizhevsky et al. (2012)). In diesem Zusammenhang sind auch weitere Experimente im Rahmen des unüberwachten Vortrainings von Interesse, in denen beispielsweise das gesamte CNN mittels eines Stacked Denoising Autoencoders (SDA) initialisiert wird.

Im Allgemeinen ist das Training eines zufällig initialisierten Netzes, beispielsweise auf Basis der ImageNet-Daten (vgl. Russakovsky et al., 2015), sehr zeitaufwändig. An dieser Stelle soll deshalb noch ein Hinweis auf eine Ablage bereits trainierter Netze für Caffe gegeben werden (vgl. Jia et al., 2014), welche für weitere Experimente von Interesse sein kann.¹ Diese Netze können beispielsweise zur Merkmalsextraktion oder für Feintuning im Rahmen des Transferlernens eingesetzt werden.

Insgesamt zeigt sich, dass Convolutional Neural Networks in Verbindung mit Methoden des Deep Learning ein ausgezeichnetes Modell für die Klassifikation und Regression von Daten mit lokalen Merkmalen im Eingaberaum darstellen. Insbesondere die Möglichkeit Vorwissen über die Struktur der Daten einzubetten bietet eine große Vielfalt an Optimierungsmöglichkeiten. Dennoch ist die Suche nach den optimalen Hyperparametern schwierig und die Suche nach Fehlern oftmals zeitintensiv. Besonders hier gilt es neue Verfahren zu entwickeln, um die Handhabung solcher Modelle zukünftig einfacher zu gestalten.

¹Projektseite von Caffe: <https://github.com/BVLC/caffe> (22.09.2015)

Literaturverzeichnis

- Abdel-Hamid, O., Deng, L., und Yu, D. (2013). Exploring convolutional neural network structures and optimization techniques for speech recognition. In Bimbot, F., Cerisara, C., Fougeron, C., Gravier, G., Lamel, L., Pellegrino, F., und Perrier, P., Hrsg., *Proceedings of the International Conference of the 14th International Speech Communication Association, Interspeech*. ISCA.
- Abuzaid, F., Hadjis, S., Zhang, C., und Ré, C. (2015). Caffe con troll: Shallow ideas to speed up deep learning. *CoRR*, abs/1504.04343.
- Andrade, A. d. (2014). Best practices for convolutional neural networks applied to object recognition in images. Thesis. URL: <http://www.cs.toronto.edu/~adeandrade/assets/bpfcnnatorii.pdf> (15.08.2015).
- Becker, S. (1991). Unsupervised learning procedures for neural networks. *International Journal of Neural Systems*, 2(Feb):17–33.
- Bell, S. und Bala, K. (2015). Learning visual similarity for product design with convolutional neural networks. *ACM Transactions on Graphics*, 34(4):98:1–98:10.
- Bengio, Y. (2009). Learning deep architectures for ai. *Foundation and Trends in Machine Learning*, 2(1):1–127.
- Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533.
- Bengio, Y., Goodfellow, I. J., und Courville, A. (2015). Deep learning. Buch in Vorbereitung für MIT Press. URL: <http://www.iro.umontreal.ca/~bengioy/dlbook/> (15.08.2015).
- Bengio, Y., Lamblin, P., Popovici, D., und Larochelle, H. (2007). Greedy layer-wise training of deep networks. In Schölkopf, B., Platt, J., und Hoffman, T., Hrsg., *Proceedings of the 20th International Conference on Advances in Neural Information Processing Systems, NIPS*, Seiten 153–160. MIT Press. ISBN: 9781622760381.

- Bengio, Y. und LeCun, Y. (2007). Scaling learning algorithms towards AI. In Bottou, L., Chapelle, O., DeCoste, D., und Weston, J., Hrsg., *Large Scale Kernel Machines*. MIT Press. ISBN: 9780262026253.
- Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., und Bengio, Y. (2010). Theano: a cpu and gpu math expression compiler. In Varoquaux, G., Walf, S. v. d., und Millman, K. J., Hrsg., *Proceedings of the 8th Python for Scientific Computing Conference, SciPy*, Band 4, Seite 3. Caltech. ISBN: 9780557232123.
- Bottou, L. (1998). Stochastic gradient tricks. In Montavon, G., Orr, G. B., und Müller, K.-R., Hrsg., *Neural networks: Tricks of the trade*, Seiten 430–445. Springer. ISBN: 9783642352898.
- Bouvie, J. (2006). Notes on convolutional neural networks. Technical Report. URL: http://cogprints.org/5869/1/cnn_tutorial.pdf (15.08.2015).
- Chapelle, O. und Erhan, D. (2011). Improved preconditioner for hessian free optimization. In Coates, A., Bengio, Y., LeCun, Y., Roux, N. L., und Ng, A. Y., Hrsg., *Proceedings of the NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. NIPS.
- Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., und Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759.
- Ciresan, D. C., Meier, U., und Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745.
- Collobert, R., Kavukcuoglu, K., und Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *Proceedings of the NIPS Workshop BigLearn*. NIPS.
- Dauphin, Y., Pascanu, R., Gülcühre, Ç., Cho, K., Ganguli, S., und Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572.
- Dauphin, Y. N., de Vries, H., Chung, J., und Bengio, Y. (2015). Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390.
- Desjardins, G. und Bengio, Y. (2008). Empirical evaluation of convolutional rbms for vision. Technical Report. URL: <http://www.iro.umontreal.ca/~lisa/publications2/index.php/attachments/single/194> (21.09.2015).
- Duda, R. O. und Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. John Wiley & Sons. ISBN: 9780471223610.

- Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., und Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11(Feb):625–660.
- Fei-Fei, L. und Karpathy, A. (2014). CS231n Convolutional Neural Networks for Visual Recognition, University of Stanford. Course Material. URL: <http://cs231n.github.io/> (14.08.2015).
- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202.
- Glorot, X. und Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. und Titterington, M., Hrsg., *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, AISTATS*, Seiten 249–256. JMLR W&CP.
- Glorot, X., Bordes, A., und Bengio, Y. (2011). Deep sparse rectifier neural networks. In Teh, Y. W. und Titterington, M., Hrsg., *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics, AISTATS*, Seiten 315–323. JMLR W&CP.
- Golik, P., Doetsch, P., und Ney, H. (2013). Cross-entropy vs. squared error training: a theoretical and experimental comparison. In Bimbot, F., Hrsg., *Proceedings of the International Conference of the International Speech Communication Association, Interspeech*, Seiten 1756 –1760. ISCA. ISBN: 9781629934433.
- Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., und Bengio, Y. (2013a). Pylearn2: a machine learning research library. *arXiv preprint*, 1308.421.
- Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., und Bengio, Y. (2013b). Maxout networks. In Lawrence, N. und Reid, M., Hrsg., *Proceedings of the 30th International Conference on Machine Learning, ICML*, Seiten 1319–1327. JMLR W&CP.
- Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. Linear Algebra Library. URL: <http://eigen.tuxfamily.org/index.php> (21.09.2015).
- Hagan, M. T., Demuth, H. B., Beale, M., und Jesús, O. D. (2014). *Neural Network Design*. Martin Hagan, 2. Auflage. ISBN: 9780971732117.
- Hastie, T., Tibshirani, R., und Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, 2. Auflage. ISBN: 9780387848570.

- Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. Pearson, 2. Auflage. ISBN: 9780132733502.
- He, K., Zhang, X., Ren, S., und Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852.
- Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the 8th Annual Conference of the Cognitive Science Society, CogSci*.
- Hinton, G. E., Osindero, S., und Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.
- Hinton, G. E. und Salakhutdinov, R. R. (2008). Using deep belief nets to learn covariance kernels for gaussian processes. In Platt, J., Koller, D., Singer, Y., und Roweis, S., Hrsg., *Proceedings of the 20th International Conference on Advances in Neural Information Processing Systems, NIPS*, Seiten 1249–1256. MIT Press.
- Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., und Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.
- Hinton, G. E., Srivastava, N., und Swersky, K. (2015). CSC321 Neural Networks for Machine Learning, University of Toronto. Course Material. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (14.08.2015).
- Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen. Thesis. URL: <http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf> (15.08.2015).
- Hochreiter, S. und Schmidhuber, J. (1997). Long short-term memory. *Neural Computing*, 9(8):1735–1780.
- Hubel, D. H. und Wiesel, T. N. (1962). Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160(1):106–154.
- Igel, C. und Hüskens, M. (2000). Improving the rprop learning algorithm. In *Proceedings of the 2nd international symposium on neural computation, ICSC*, Seiten 115–121. ICSC Academic Press.
- Jarrett, K., Kavukcuoglu, K., Ranzato, M., und LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Proceedings of the 12th IEEE International Conference on Computer Vision, ICCV*, Seiten 2146–2153. IEEE.

- Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., und Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint*, 1408.5093.
- Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. Python Library. URL: <http://www.scipy.org/>.
- Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997.
- Krizhevsky, A. und Hinton, G. (2009). Learning multiple layers of features from tiny images. Thesis. URL: <http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf> (15.08.2015).
- Krizhevsky, A., Sutskever, I., und Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., und Weinberger, K., Hrsg., *Proceedings of the 25th International Conference on Advances in neural information processing systems, NIPS*, Seiten 1097–1105. MIT Press. ISBN: 9781627480031.
- Larochelle, H., Bengio, Y., Louradour, J., und Lamblin, P. (2009). Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, 10(Jan):1–40.
- Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J., und Ng, A. (2012). Building high-level features using large scale unsupervised learning. In Langford, J. und Pineau, J., Hrsg., *Proceedings of the 29th International Conference on Machine Learning, ICML*. ACM. ISBN: 9781450312851.
- Le, Q. V., Ngiam, J., Chen, Z., Chia, D., Koh, P. W., und Ng, A. Y. (2010). Tiled convolutional neural networks. In Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., und Culotta, A., Hrsg., *Proceedings of the 20th International Conference on Advances in Neural Information Processing Systems, NIPS*, Seiten 1279–1287. Curran Associates.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., und Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.
- LeCun, Y., Bottou, L., Bengio, Y., und Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- LeCun, Y. A., Bottou, L., Orr, G. B., und Müller, K.-R. (1998b). Efficient backprop. In Montavon, G., Orr, G. B., und Müller, K.-R., Hrsg., *Neural networks: Tricks of the trade*, Seiten 9–48. Springer. ISBN: 9783642352898.

- Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., und Tu, Z. (2014). Deeply-supervised nets. *CoRR*, abs:1409.5185v2.
- Lee, H., Grosse, R., Ranganath, R., und Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In Danyluk, A. P., Bottou, L., und Littman, M. L., Hrsg., *Proceedings of the 26th International Conference on Machine Learning, ICML*, Seiten 609–616. ACM. ISBN: 9781605585161.
- Maas, A. L., Hannun, A. Y., und Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning, ICML*. JMLR W&CP.
- Maaten, L., Hinton, G. E., und Bengio, Y. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605.
- Mahendran, A. und Vedaldi, A. (2014). Understanding deep image representations by inverting them. *CoRR*, abs/1412.0035.
- Martens, J. (2010). Deep learning via hessian-free optimization. In Fürnkranz, J. und Joachims, T., Hrsg., *Proceedings of the 27th International Conference on Machine Learning, ICML*. Omnipress.
- Masci, J., Meier, U., Cireşan, D., und Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. In *Proceedings of the International Conference on Artificial Neural Networks and Machine Learning, ICANN*, Seiten 52–59. Springer. ISBN: 9783642217357.
- McCulloch, W. und Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill. ISBN: 978-0071154673.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., und Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.
- Nagi, J., Ducatelle, F., Caro, G. A. D., Cires, D., Meier, U., Giusti, A., Nagi, F., Schmidhuber, J., und Gambardella, L. M. (2011). Max-pooling convolutional neural networks for vision-based hand gesture recognition. In *Proceedings of the IEEE International Conference on Signal and Image Processing Applications, ICSIPA*. IEEE.
- Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate $O(1/k^2)$. *Soviet Mathematics Doklady*, 27(2):372–376.

- Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. URL: <http://neuralnetworksanddeeplearning.com/> (03.09.2015).
- Nouri, D. (2013). cuda-convnet. Cuda Framework. URL: <https://github.com/dnouri/cuda-convnet> (21.09.2015).
- OpenMP Architecture Review Board (2008). OpenMP application program interface version 3.0. Compiler Extension. URL: <http://openmp.org/wp/> (21.09.2015).
- Pearlmutter, B. A. (1994). Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160.
- Pierre Sermanet, S. C. und LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. *CoRR*, abs/1204.3968.
- Pink, O. (2011). *Bildbasierte Selbstlokalisierung von Straßenfahrzeugen*. KIT Scientific Publishing. ISBN: 9783866447080.
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics*, 4(5):1–17.
- Quiroga, R. Q., Reddy, L., Kreiman, G., Koch, C., und Fried, I. (2005). Invariant visual representation by single neurons in the human brain. *Nature*, 435:1102–1107.
- Ranzato, M., Poultney, C. S., Chopra, S., und Lecun, Y. (2006). Efficient Learning of Sparse Representations with an Energy-Based Model. In Bernhard Schölkopf, J. P. und Hofmann, T., Hrsg., *Proceedings of the 19th Conference on Neural Information Processing Systems, NIPS*, Seiten 1137–1144. MIT Press.
- Ranzato, M., Poultney, C. S., Chopra, S., und Lecun, Y. (2007a). Efficient learning of sparse representations with an energy-based model. In Schölkopf, B., Platt, J., und Hoffman, T., Hrsg., *Proceedings of the 19th International Conference on Advances in Neural Information Processing Systems, NIPS*, Seiten 1137–1144. MIT Press.
- Ranzato, M. A., Huang, F. J., Boureau, Y.-L., und LeCun, Y. (2007b). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, Seiten 1–8. IEEE. ISBN: 1424411793.
- Riedmiller, M. und Braun, H. (1992). Rprop-a fast adaptive learning algorithm. In *Proceedings of the 7th International Symposium on Computer and Information Science, ISCIS*.

- Rojas, R. (1996). *Neural Networks: A Systematic Introduction*. Springer. ISBN: 9783540605058.
- Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books. ISBN: 9783642709135.
- Rumelhart, D. E., Hinton, G. E., und Williams, R. J. (1986a). Learning internal representations by error propagation. In Rumelhart, D. E. und McClelland, J., Hrsg., *Parallel Distributed Processing: Implications for Psychology and Neurobiology*, Band 1, Seiten 318–362. MIT Press. ISBN: 0262680531.
- Rumelhart, D. E., Hinton, G. E., und Williams, R. J. (1986b). Learning representations by back-propagating errors. *Nature*, 323:533–536.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., und Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 1(Apr):1–42.
- Sainatha, T. N., Kingsburya, B., Saona, G., Soltau, H., rahman Mohamedb, A., Dahlb, G., und Ramabhadran, B. (2015). Deep convolutional neural networks for large-scale speech tasks. *Neural Networks: Deep Learning of Representations*, 64(Apr):32–48.
- Schling, B. (2011). *The Boost C++ Libraries*. XML Press. ISBN: 9780982219195.
- Simard, P., Steinkraus, D., und Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the 7th International Conference on Document Analysis and Recognition, ICDAR*, Seiten 958–963. IEEE. ISBN: 0769519601.
- Simard, P., Victorri, B., LeCun, Y., und Denker, J. (1992). Tangent prop: a formalism for specifying selected invariances in adaptive networks. In Moody, J. M., Hanson, S. J., und Lippman, R. P., Hrsg., *Proceedings of the 4th International Conference on Advances in Neural Information Processing Systems, NIPS*, Seiten 895–903. Morgan Kaufmann. ISBN: 1558602224.
- Simonyan, K., Vedaldi, A., und Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034.
- Simonyan, K. und Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

- Socher, R., Lin, C. C.-Y., Ng, A., und Manning, C. D. (2011). Parsing natural sciences and natural language with recursive neural networks. In Getoor, L. und Scheffer, T., Hrsg., *Proceedings of the 28th International Conference on Machine Learning, ICML*. ACM. ISBN: 9781450306195.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., und Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.
- Sutskever, I. (2013). Training recurrent neural networks. Thesis. URL: http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf (20.08.2015).
- Sutskever, I., Martens, J., Dahl, G., und Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning, ICML*, Seiten 1139–1147. JMLR W&CP.
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., und Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, abs/1409.4842.
- Vincent, P., Larochelle, H., Bengio, Y., und Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning, ICML*, Seiten 1096–1103. JMLR W&CP.
- Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., und Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11(Dec):3371–3408.
- Wagner, R., Thom, M., Schweiger, R., Palm, G., und Rothermel, A. (2013). Learning convolutional neural networks from few samples. In *Proceedings of the International Joint Conference on Neural Networks, IJCNN*, Seiten 1–7. IEEE. ISBN: 9781467361286.
- Wan, L., Zeiler, M. D., Zhang, S., LeCun, Y. A., und Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning, ICML*. JMLR W&CP.
- Widrow, B. und Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record, Part 4*, Seiten 96–104.
- Widrow, B. und Hoff, M. E. (1988). Adaptive switching circuits. In Anderson, J. A. und Rosenfeld, E., Hrsg., *Neurocomputing: Foundations of Research*, Seiten 123–134. MIT Press. ISBN: 0262010976,.

- Wilson, D. R. und Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451.
- Winston, P. H. (1992). *Artificial Intelligence*. Pearson, 3. Auflage. ISBN: 9780201533774.
- Zander, A. (2001). *Neuronale Netze zur Analyse von nichtlinearen Strukturmodellen mit latenten Variablen*. Deutscher Universitäts-Verlag. ISBN: 9783824472598.
- Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.
- Zeiler, M. D. und Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. *CoRR*, abs/1301.3557.
- Zeiler, M. D. und Fergus, R. (2014). Visualizing and understanding convolutional networks. In Fleet, D. J., Pajdla, T., Schiele, B., und Tuytelaars, T., Hrsg., *Proceedings of the 13th European Conference on Computer Vision, ECCV*, Seiten 818–833. Springer. ISBN: 9783319105901.
- Zeiler, M. D., Taylor, G. W., und Fergus, R. (2011). Adaptive deconvolutional networks for mid and high level feature learning. In Metaxas, D. N., Quan, L., Sanfeliu, A., und Gool, L. J. V., Hrsg., *Proceedings of the 14th IEEE International Conference on Computer Vision, ICCV*, Seiten 2018–2025. IEEE. ISBN: 9781457711015.