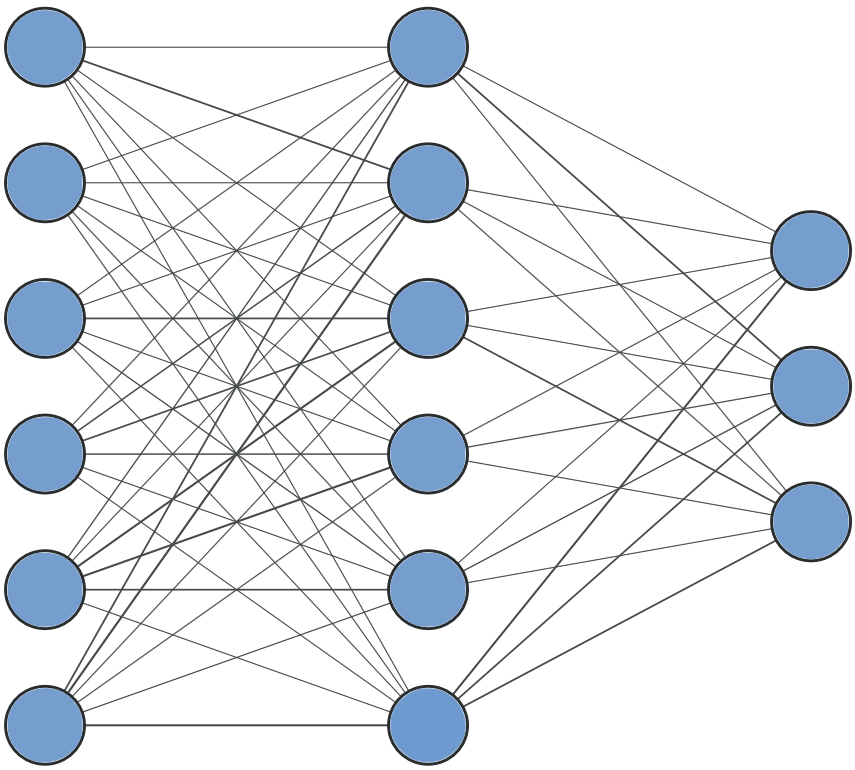
Benjamin Säfken/Alexander Silbersdorff/
Christoph Weisser (Eds.)

Learning Deep

Perspectives on Deep Learning Algorithms and Artificial Intelligence



Universitätsdrucke Göttingen

Benjamin Säfken/Alexander Silbersdorff/ Christoph Weisser (Eds.)

Learning Deep

erschienen in der Reihe der Universitätsdrucke
im Universitätsverlag Göttingen 2020

Benjamin Säfken
Alexander Silbersdorff
Christoph Weisser (Eds.)

Learning Deep

Perspectives on Deep Learning Algorithms and Artificial Intelligence

Universitätsverlag Göttingen
2020

Bibliographische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet über <http://dnb.dnb.de> abrufbar.

Adresse der Herausgeber
Georg-August-Universität Göttingen
Lehrstühle für Statistik und Ökonometrie
Humboldtallee 3
D-37073 Göttingen
https://www.uni-goettingen.de/de/411195.html
E-Mail: asilbersdorff@uni-goettingen.de
E-Mail: benjamin.saefken@uni-goettingen.de
E-Mail: c.weisser@stud.uni-goettingen.de

Satz und Layout: Dominik Becker
Titelabbildung: René Kruse: Neuronales Netz

https://univerlag.uni-goettingen.de
ISBN: 978-3-86395-462-8
DOI: https://doi.org/10.17875/gup2020-1338

Learning Deep - Perspectives on Deep Learning Algorithms and Artificial Intelligence

A. Silbersdorff[*], B. Säfken[*], L. M. Dammann[*], R. Kruse[*], and C. Weisser[*]

[*]Georg-August-Universität Göttingen, Germany

Preamble

The rise of artificial intelligence has come to the forefront of both academic and public discussion in recent years and accordingly the topic has gained considerable interest among students. Many of the recent advances in and the growing use of artificial intelligence are built around the applications of deep learning algorithms. In the winter semester 2018/19, we thus decided to offer a new seminar on deep learning algorithms to students of the masters programme in applied statistics at the University of Göttingen and did so again in the winter semester 2019/20. Following the Humboldtian model of higher education, we aimed to allow students to conduct their own research into the basic ideas, mechanics and practical applications of deep learning algorithms and thereby learn about the issue more deeply than by conventional lecture-based teaching. Their findings were presented in the seminar and subsequently portrayed in article-styled seminar papers. The results of this seminar, both in terms of the advancements made by many students in their understanding and the quality of many of the submitted seminar papers, were strikingly positive and deserving of publication in our eyes. Thus we decided to give the students the chance to publish their work in this edited volume. The seven best seminar papers were thus selected for publication and the selected students went through a full review process conducted by two researchers active in the field of deep learning. Upon successfully addressing the issues raised by the reviews, the articles were included in this volume. Given that the publication of the seminar papers was not our original intention, we left it up to the students to decide whether they would write the seminar paper in English or in German, with some groups choosing the former and some the latter option. Accordingly, the content of this book entails contributions in the two different languages.
The contributions are structured as follows:

The first paper by Clemens Haerder entitled "Deep Learning und Machine Learning: Ein Vergleich anhand des Boston Housing Value Datensatzes" provides an introductory contrast between simple neural nets and deep learning algorithms on the one

hand and classical machine learning techniques like random forest and boosting on the other hand. Using a dataset regarding the Boston housing market entailing only 506 observation the paper highlights the problems of the elaborate neural network structure when adapting it to a dataset that is limited in sample size.

The second paper by Martin Wutke provides a comprehensive introduction to deep feedforward neural networks. It starts with a general introduction to neural networks and the training of networks with the back-propagation algorithm. For illustrative purposes, a detailed example based on the MNIST dataset is provided.

The third paper by Nikos Bosse entitled "An Introduction to deep learning and the concept of regularization provides an intuitive introduction to the requirement for and possible approaches for regularisation techniques with regard to machine learning in general and deep learning algorithms in particular. Using three exemplary datasets it illustrates the fine line between overfitting and underfitting that needs to be achieved for adequately calibrating the many parameters of neural networks.

The fourth paper by Felix Süttmann provides a comprehensive introduction to recurrent neural networks (RNN). It starts with a general and technical introduction. Subsequently, it provides a detailed example from the field of Natural Language Processing (NLP). In the illustrative application a RNN is trained and applied for the classification of offensive or non-offensive Twitter posts.

The fifth paper by Anton Thielmann, Quentin Seifert and Jens Lichter provides an introduction to convolutional neural networks as well as to important regularization strategies like data augmentation, dropout, early stopping and weight penalization. The described methods are implemented and evaluated for the MNIST American Sign Language data set.

The sixth joint paper by Tim Ruhkopf and Tim Toebrock is concerned with the usage of different neural network architectures for denoising MR images. The general methods of neural nets and especially convolutional neural nets is explained. Specific architectures such as Denoising Convolutional Neural Network and specifications as the Autoencoders (U-Net) are discribed in detail. The methods are applied to a dataset of 3D images of knees, although these images are reduced to two dimensions by slicing. The results are discribed and the underlying model is evaluated.

The seventh and last paper by Andreas Buchmüller and Christoph Gerloff uses artifical neural nets for music genre classification. For the analysis they use sampled track data. They create genre classifiers based on different spectrogram-like features such as timbre or pitch with convolutional neural networks. The resulting models are evaluated based on the performance and accuracy with different measures.

From these contributions of the selected students participating in the seminar, we hope on the one hand that the reader gains insights into the topics addressed in the contributions. On the other hand, these contributions hopefully portray the scope

and depth of the understanding developed by students when left to explore deep learning algorithms in a deep manner following the Humboldtian model of higher education.

We want to thank Dominik Becker who made considerable effort in formatting this publication and joining the different papers together. Furthermore we thank the Campus-Institut Data Science (CIDAS) for funding this project.

Contents

T. Ruhkopf and T. Toebrock

DeepMRI: Using Deep Convolutional Networks to improve MR Images

Deep Learning und klassisches Machine Learning

Ein Vergleich anhand des Boston Housing Value Datensatzes

C. Haerder

Georg-August-Universität Göttingen, Germany

Zusammenfassung. The paper provides an introductory contrast between deep learning algorithms on the one hand and classical machine learning techniques like random forest and boosting on the other hand. Using a dataset regarding the Boston housing market entailing only 506 observations the paper highlights the problems of the elaborate neural network structure when adapting it to a dataset that is limited in sample size.

1 Einführung

Neuronale Netze haben sich bei komplexen dedizierten Problemen mindestens als Teil der Lösung durchgesetzt. Regelmässig werden mit dem Einsatz von Neuronalen Netzen Data Science Wettbewerbe auf der Data Science Online-Community Kaggle gewonnen.

Deep Learning ist grundsätzlich ein Teilgebiet des Machine Learnings. Die beiden Herangehensweisen unterscheiden sich jedoch in der Art der Aufgabenlösung. Während Machine Learning meist statistische Methoden nutzt, verwendet Deep Learning künstliche Neuronale Netze, welche biologischen Nervensystemen nachempfunden sind. Klassisches Machine Learning ist nicht im Stande zentrale Probleme der künstlichen Intelligenz, wie Sprach- und Objekterkennung, zu lösen. Die hohe Dimensionalität dieser Probleme erschwert klassischem Machine Learning das Lernen komplexer Funktionen. Zudem ist die praktische Anwendung durch den Rechenaufwand in hoch dimensionalen Problemen beeinträchtigt. Hierbei stellt der Einsatz Neuronaler Netze eine sinnvolle Lösung dar (Goodfellow et al. 2016).

Die dieser Arbeit gegenständliche Frage ist, ob der Einsatz von Neuronalen Netzen auch bei klassischen Machine Learning Problemstellungen sinnvoll sein kann. Hierfür werden werden Neuronale Netze mit zwei Methoden des klassischem Machine Learnings verglichen (Random Forests & Gradient Boosting). Es wird ein einfacher Datensatz ausgewählt, bei welchem der negative Effekt der Dimensionalität nicht besteht. Die Bewertung der Methoden erfolgt hinsichtlich der Einfachheit der Anwendung, des Zeitaufwands zum Modelltraining, sowie der Interpretierbarkeit des Modelles und der Präzision der Ergebnisse.

2 Methoden

Es wird der bekannte Boston Housing Datensatz verwendet. Er beinhaltet 14 Variablen mit 506 Beobachtungen. Ziel ist es, den Wert von Bostoner (USA) Vorstadthäusern mithilfe aller erklärenden Variablen des Datensatzes vorherzusagen (Harrison & Rubinfeld 1978). Der Vergleich und die Bewertung der Methoden finden anhand von vier Kriterien statt.
Der Mean Absolute Error (MAE) ist die mittlere Abweichungen der Vorhersage des Modelles zu den tatsächlichen Beobachtungen und wird als erstes Kriterium genutzt, um die Präzision der Modelle zu bestimmen. Zur Ermittlung des MAEs wird der Validierungsdatensatz verwendet, da dieser nicht zum Modelltraining genutzt wurde. Als zweites Kriterium wird die Komplexität der Anwendung herangezogen. Diese gibt an, wie einfach ein zufriedenstellendes Ergebnis erreicht werden kann. Das dritte Kriterium, die Interpretierbarkeit des Modelles, bezieht sich auf die Fähigkeit zu erkennen, welche Variablen einen bedeutsamen Einfluss auf die Zielvariable haben. Das letzte Kriterium ist die Performance des Modelltrainings. Hierzu wird die benötigte Zeit als Bewertungsmaßstab herangezogen.

2.1 Deep Learning

Deep Learning nutzt Neuronale Netze zur Aufgabenlösung. Diese sind biologischen Nervensystemen nachempfunden, bei welchen Neuronen miteinander vernetzt sind. Dies geschieht konzeptionell über verschiedene Layer. Abbildung 1 stellt ein simples Single-Layer Feedforward Netzwerk dar. Dabei soll das Netzwerk über einen Input Layer Werte x erhalten, über Hidden Layer Berechnungen durchführen und einen Wert y im Output Layer ausgeben. Dies ist natürlich nicht nur auf einen Hidden Layer begrenzt, sondern es können beliebig viele hinzugefügt werden. Das Modell versucht entsprechend mit einer Funktion f^* den Output y zu bestimmen ($y = f^*(x)$). Durch die optimalen Gewichte w wird $y = f(x, w)$ am besten geschätzt.

Die Hidden Layer bestehen aus einer zu definierenden Anzahl Neuronen (auch Units genannt). Die Anzahl der Neuronen in einem Layer i sind auch die akzeptierte Größe der Input-Tensoren. Somit ist auch die Ausgabe des Layers i direkt mit den Input Anforderungen des nächsten Layers j verknüpft. Die Dimension der Tensoren, die von einem Hidden Layer akzeptiert werden, definiert auch gleichzeitig das Netzwerk. Beispielsweise werden Netzwerke zur Bildverarbeitung üblicherweise mit 4-D Tensoren verarbeitet. Diese Netzwerke werden Convolutional Neural Network (CNN) genannt. Die in dieser Arbeit genutzten Tensoren sind lediglich 2-D und werden auch fully-connected Layer genannt (Chollet & Allaire 2018; Goodfellow et al. 2016).

Unterschiedliche Aktivierungsfunktionen ermöglichen das Verbinden nicht-linearer Zusammenhänge zwischen den Layern. Die in dieser Arbeit verwendete Aktivierungsfunktion ist die Rectified Linear Unit-Aktivierungsfunktion (ReLU). Sie stellt besonders bei CNN eine effiziente und optimierungssichere Variante dar (Glorot et al. 2011; Goodfellow et al. 2016). Die Aktivierungsfunktion des letzten Layers definiert somit auch den Wertebereich des Neuronalen Netzes. Eine Sigmoid Aktivierungsfunktion des letzten Layers kann für Binärklassifikation, Multi-Label-Mehrfachklassifikation und Regression (zwischen 0 und 1) eingesetzt werden. Die Softmax-Aktivierungsfunktion

kann für Single-Label-Mehrfachklassifikation verwendet werden. Bei Regressionen mit Output-Wertebereich $y \in \mathbb{R}$ ist keine Aktivierungsfunktion nötig (Chollet & Allaire 2018).

Die Optimierung der Gewichte w wird für gewöhnlich mittels Gradienten basierten Verfahren (z.B. dem stochastischem Gradientenverfahren) durchgeführt (vgl. Chollet & Allaire 2018). In dieser Arbeit findet die Rückpropagierung ihre Anwendung. Hierbei wird das Modell über n Epochen optimiert. Bei jeder Epoche wird der Output mit den realen Werten verglichen und die Abweichung mittels einer Verlustfunktion berechnet. Dies könnte typischerweise bei einer Regression der Mean Squared Error (MSE) sein. „Die Rückpropagierung, oder mitunter auch Fehlerrückführung, beginnt mit dem letzten Wert der Verlustfunktion, arbeitet sich rückwärts von oben nach unten durch die Layer vor und wendet die Kettenregel an, um die Beiträge der einzelnen Gewichte w zum Wert der Verlustfunktion zu berechnen“ (Chollet & Allaire 2018, S. 81). So auch schematisch dargestellt durch die bidirektionalen Pfeile der Gewichte w in Abbildung 2.

Die Performance des Netzwerkes kann während des Trainings pro Epoche dargestellt werden. Keras wird ein Trainings- und Validierungsdatensatz zur Verfügung gestellt. In jeder Epoche werden die Gewichte nun auf Basis des Trainingsdatensatzes angepasst. In jeder Epoche wird zwangsläufig die Genauigkeit größer (besser) und der Verlust bzw. der MSE kleiner (besser), da sich das Netzwerk immer mehr auf den Trainingsdatensatz spezialisiert. Der Validierungsdatensatz hingegen wird nicht zum Training der Gewichte genutzt. Aber auch in diesem werden bei jeder Epoche Genauigkeit und Verlust bestimmt. Anfangs werden beide Metriken besser und pendeln sich beim jeweiligen Optimum ein. Mit zunehmenden Epochen verschlechtern sich die Werte aber im Folgenden. Es kommt zu einer Überanpassung an die Trainingsdaten. Verschiedene Methoden der Regularisierung sind einsetzbar, um Überanpassung zu vermeiden. Zum Beispiel können Normen wie L1 oder L2 für die Gewichte w eingeführt werden. In dieser Arbeit werden Genauigkeit und Verlust grafisch dargestellt, um die beste Anzahl an Trainingsepochen anschließend manuell und visuell auszuwählen.

Eine etablierte Methode, um das Risiko von Überanpassung bei kleinen Datensätzen zu reduzieren, ist die k-fache Kreuzvalidierung. Bei einem Stichprobenumfang von nur 506 Beobachtungen, würde eine einfache Unterteilung in Trainings-, Test- und Validierungsdaten mit hoher Wahrscheinlichkeit zu suboptimalen Ergebnissen führen. Eine intuitive schematische Darstellung der Kreuzvalidierung ist in Abbildung 3 zu sehen.

Die Neuronalen Netze werden mithilfe des Deep Learning Frameworks Keras trainiert. Keras, eine Python Bibliothek, wird mittels einer R-Schnittstelle in R/RStudio genutzt. Keras bietet High-Level Bausteine für Deep Learning Modelle. Die Low-Level Operationen (z.B. Tensor-Operationen und Differenzierungen) können von unterschiedlichen Deep-Learning-Frameworks bearbeitet werden. Das Backend Tensorflow wird in diesem Paper verwendet und via diesem Framework wird Keras auf der Graphical Processing Unit (GPU) ausgeführt. NVIDIA CUDA Deep Learning Network Library bietet hierfür eine Bibliothek an. Eine Graphikkarte dieses Herstellers muss aber in dem System verbaut sein (Chollet & Allaire 2018, vgl.). Folgende Hardware steht für die Modellierung zur Verfügung: Intel Core i7-4710HQ @ 2.5 GHz, NVIDIA

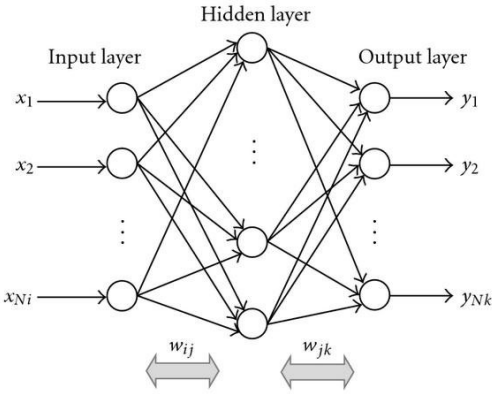


Abbildung 1: Schematisches Diagramm des Neuronalen Netzes einer Single-Layer Back Propagation (Chen et al. 2009, S. 4).

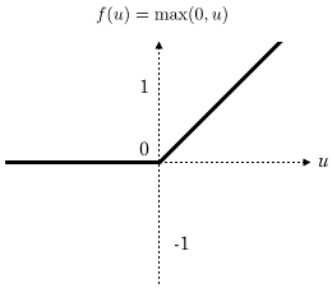


Abbildung 2: Rectified Linear Unit Funktion (Glorot et al. 2011, S. 318).

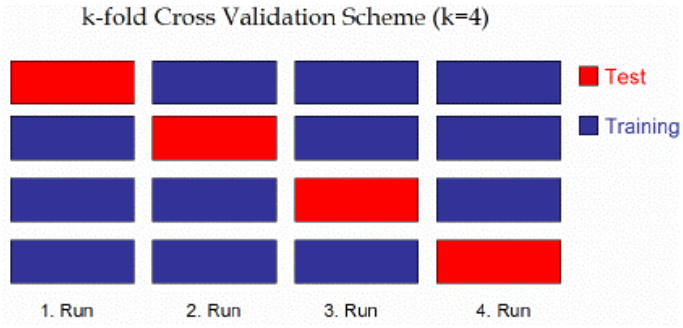


Abbildung 3: k-fache Kreuzvalidierung (k = 4) (Bisgin et al. 2011, S. 268).

GeForce GTX 860M, 8 GB DDR3 1600 MHZ.

Das Neuronale Netz wird mit unterschiedlichen Kombinationen erstellt. Ausgangslage ist die vorgeschlagene Architektur von Chollet & Allaire (2018) mit 2 Hidden Layer, die jeweils 64 Neuronen enthalten und der eine 4-fache Kreuzvalidierung zugrunde liegt. Alle Hidden Layer haben ReLU als Aktivierungsfunktion. Die Verlustfunktion ist der MSE. Als Metrik wird der MAE verwendet. Variiert wird bezüglich der Anzahl an Hidden Layer, der Anzahl an Neuronen innerhalb der Hidden Layer, wobei jeder Hidden Layer dieselbe Anzahl Neuronen hat, und der Anzahl der k-fachen Kreuzvalidierungen. In einem ersten Schritt wird mittels Kreuzvalidierung das Netzwerk über 200 Epochen trainiert. Der MAE aus dem Validierungsdatensatz wird für jede Epoche geplottet. Somit lässt sich die Epoche bestimmen, die den MAE minimiert und das Netzwerk kann auf die Testdaten angewandt werden. Anschließend wird das Netzwerk mit den Trainingsdaten, einem Batchsize von 16 und den definierten Epochen angepasst.

2.2 Klassisches Machine Learning

Die ausgewählten klassischen Machine Learning Methoden sind Gradient Boosting und Random Forests. Random Forests wurden 2001 von Breiman vorgesellt. Sie sind eine Kombination aus (Regression-) Trees, wobei jeder dieser Trees abhängig ist von einem zufälligen und unabhängigen Vektor von Werten innerhalb derselben Verteilung für jeden Tree des Forests (Breiman 2001). Der Vorteil von Random Forests ist, dass die üblicherweise hohe Varianz von einem Tree durch das Mitteln von vielen erwartungstreuen Trees zu guten Modellen führt (Hastie & Tibshirani 2009).

Gradient Boosting ist ein „greedy“ Algorithmus, der bei jedem Schritt den Tree auswählt, der eine Verlustfunktion minimiert. Diese Verlustfunktion muss ableitbar sein. Anschließend werden die einzelnen Trees gewichtet, wobei der optimale Tree die leichteste Gewichtung erhält. Das führt aber dazu, dass die einzelnen schwachen Trees innerhalb des Random Forests abhängig sind und sich bei jeder Iteration verbessern. Für weitere Informationen bezüglich Trees und deren Funktionsweise siehe Hastie & Tibshirani (2009). Die Trees werden mit dem R-Package *caret* erstellt (Kuhn 2018). Klassifizierungs- und Regressions-Trees bieten Datenaufteilung, Vorverarbeitung, Variablenselektion, Modell-Einstellung und eine Variablen-Wichtigkeits-Schätzung (Kuhn 2018). Wie bei den Neuronalen Netzen, wird auch hier der Datensatz in Trainings-, Validierungs- und Testdaten aufgeteilt. Dazu wird eine 10-fach Kreuzvalidierung zur Aufteilung in Trainings- und Validierungsdaten für das Modelltraining angewandt.

3 Resultate

Eine Übersicht der Resultate befindet sich in Tabelle 1. Die Plots der MAEs über die einzelnen Epochen sind in Abbildung 4 zu finden. Den minimalen MAE weist mit 2.179 das Neuronale Netzwerk mit 2 Layern, jeweils 32 Units und 8-fach Kreuzvalidierung auf. Zuletzt wird das Modell anhand der Testdaten evaluiert. Der jeweils erreichte Verlust und MAE befinden sich in Tabelle 2. Hier erreicht das Neuronale

Tabelle 1: Minimaler MAE der Validierungsdaten für jede Variante.

Layer	2		3	
k-folds	32	64	32	64
4	2.211	2.275	2.206	2.194
8	2.179	2.261	2.217	2.204

Netzwerk mit 2 Layern, jeweils 32 Units und 8-fach Kreuzvalidierung das zweitkleinste Resultat bezüglich des MAEs.

Die Median Ergebnisse und die Parametrisierung der am besten trainierten Random Forests und Gradient Boosting Machine befinden sich, gemeinsam mit den erreichten MAEs der Testdaten, in Tabelle 3.

In Abbildung 5 sind Konfidenzintervalle für den MAE beider Modelle visualisiert. In Abbildung 6 ist die Wichtigkeit der Variablen für den Random Forest abgebildet. Die Wichtigkeit misst die Bedeutung der Variablen hinsichtlich der Performance des Modelles. Die Variablen werden permutiert und die Abnahme der Genauigkeit, bzw. die Zunahme des MSEs festgehalten. Eine wichtige Variable hat somit eine hohe Wichtigkeit (Molnar 2019). Der Bevölkerungsanteil mit niedrigem Status (lstat) und die durchschnittliche Anzahl an Zimmern pro Wohnung sind bei der Prognose der Hauspreise demnach am bedeutendsten.

Ein Vergleich der Laufzeiten des kleinsten und größten Neuronalem Netz mit dem Random Forest und der Gradient Boosting Machine ist Tabelle 4 zu entnehmen. Ein 3 Layer, 64 Neuronen und 8-fach Kreuzvalidierung Neuronales Netz braucht über 20 Minuten, um trainiert zu werden. Dieses beinhaltet auch die Findung der optimalen Anzahl an Epochen. Das einfachste Neuronale Netz mit 2 Layern, 32 Neuronen und 4-fach Kreuzvalidierung lässt sich in etwa 11 Minuten trainieren. Der Random Forest benötigt etwas mehr als 3 Minuten und die Gradient Boosting Machine 9.77 Sekunden.

Tabelle 2: Verlust und MAE der Testdaten aller Varianten.

Layer		2		3	
k-fach	Neuronen	32	64	32	64
4	Verlust	21.628	18.519	19.861	22.414
	MAE	2.860	2.688	2.870	3.388
8	Verlust	16.640	20.871	16.772	19.206
	MAE	2.645	2.771	2.628	2.651

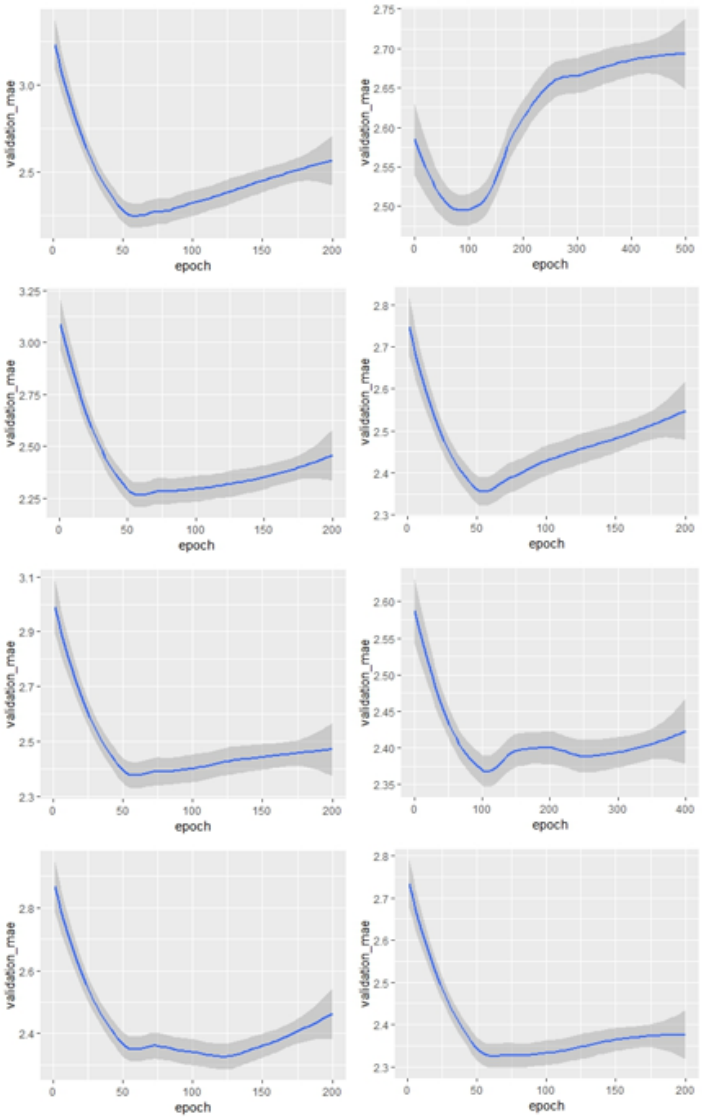


Abbildung 4: Validierung der Neuronalen Netze (von oben nach unten). 4-fach Kreuzvalidierung: 2 Layer 32 Neuronen (links) 2 Layer 64 Neuronen (rechts), 8-fach Kreuzvalidierung: 2 Layer 32 Neuronen (links) 2 Layer 64 Neuronen (rechts), 4-fach Kreuzvalidierung: 3 Layer 32 Neuronen (links) 3 Layer 64 Neuronen (rechts), 8-fach Kreuzvalidierung: 3 Layer 32 Neuronen (links) 3 Layer 64 Neuronen (rechts).

Tabelle 3: Finale Parameter der beiden Methoden und Median MAE, Residual MSE und Rsquared.

	Anzahl Variablen	Anzahl Trees	Tiefe	Median MAE	Median RMSE	Median Rsquared	Testdata MAE
Random Forest	7	-	-	2.155	3.095	0.900	2.348
Gradient Boosting Machine	-	150	3	2.274	3.104	0.888	2.198

Tabelle 4: Laufzeiten zum Trainieren des kleinsten und größten NN, sowie RF und GBM in Sekunden.

Methode	Zeit [Sekunden]
NN	696.90
NN (3L 64U 8k)	1551.48
Random Forest	191.07
Gradient Boosting Machine	9.77

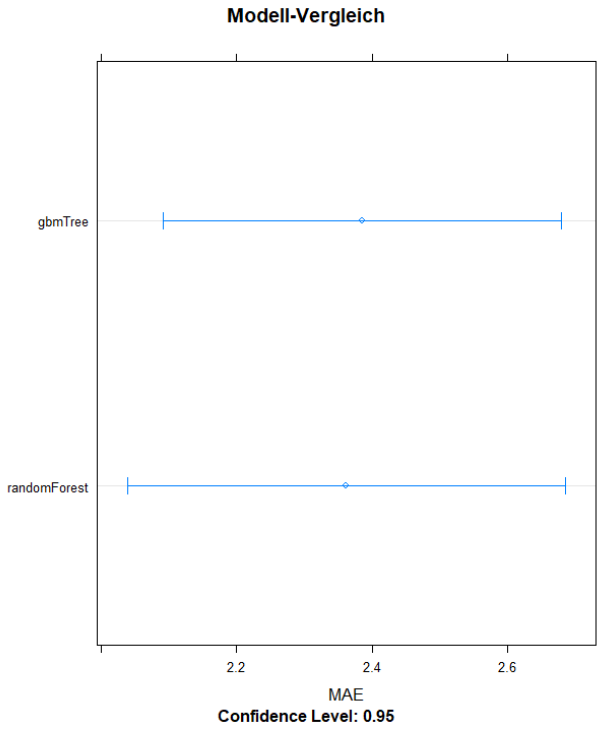


Abbildung 5: 95-Prozent Konfidenzintervalle des MAEs beider Tree-basierten Methoden.

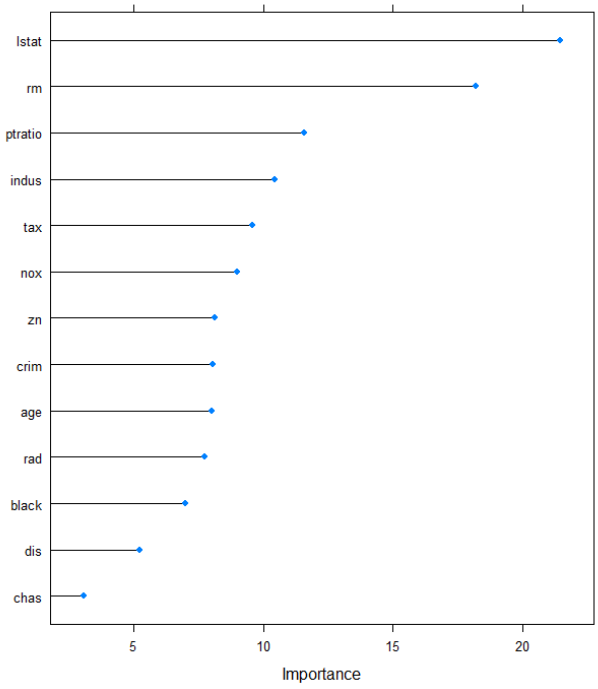


Abbildung 6: Wichtigkeit der Variable des Random Forests.

4 Diskussion

Im vorliegenden Beitrag wurden der Einsatz Neuronaler Netze anhand der Kriterien Einfachheit der Anwendung, Zeitaufwand des Modelltrainings, Interpretierbarkeit des Modells und der Präzision der Ergebnisse evaluiert und im Vergleich zu zwei Methoden des klassischen Machine Learnings bewertet. Die Tree-basierten Methoden sind den Neuronalen Netzen hinsichtlich der vier Evaluationskriterien an diesem einfachen Datensatz überlegen.

Die MAEs der Neuronalen Netze und der Tree-basierten Methoden sind im Test und der Validierung vergleichbar. Der minimale MAE ist 2.179 bei dem Neuronalen Netzwerk mit 2 Layern, jeweils 32 Units und 8-fach Kreuzvalidierung. Dennoch ist der Random Forest mit 2.155 etwas besser. Gradient Boosting ist mit 2.274 etwas schlechter. Trotzdem übertrifft Gradient Boosting mit einem Testdaten-MAE von 2.198 den des Random Forests (2.348) und des besten Neuronalen Netzwerkes (3 Layer, jeweils 32 Neuronen und 8-fach Kreuzvalidierung, MAE: 2.628). Somit ist die Performance bezüglich des MAEs der Tree-basierten Methoden höher als die des besten Neuronalen Netzwerkes.

Die Komplexität der Anwendung ist bei den Tree-basierten Methoden niedriger. Erwähnenswert ist zudem, dass Gradient Boosting und Random Forests mit den Default-Einstellungen des *caret* R-Paketes (Kuhn 2018) ausgeführt wurden. Das Finden einer passenden Neuronalen Netzwerkstruktur führt durch die Anwendung einer Versuch-und-Irrtums Methode zu einem insgesamt höheren Aufwand.

Die Performance der Trees, bezüglich der benötigten Zeit zur Modellanpassung, ist signifikant besser als die der Neuronalen Netze. Gradient Boosting ist ca. 70-mal schneller als das verwendete Neuronale Netz. Die Effizienz birgt einen erheblichen Vorteil. Eine Optimierung der Hyperparameter kann mit gridsearch durchgeführt werden. Ein gridsearch bei den beschriebenen Neuronalen Netzen bedeutet erhebliche Einschränkungen der Rechenleistung. Hingegen kann gerade Gradient Boosting, das aufgrund der Vielzahl an Hyperparametern besonders von gridsearch profitieren sollte, problemlos ausgeführt werden.

Neuronale Netze sind aufgrund der Konstruktion kaum oder äußerst schwierig zu interpretieren. Die Bedeutsamkeit einzelner Variablen bei der Marktwertprognose der Häuser in Boston lässt sich bei den Neuronalen Netzen nicht determinieren. Die Tree-basierten Methoden hingegen können die Wichtigkeit von Variablen darstellen. Somit sind Trees bezüglich der Interpretierbarkeit ebenfalls vorzuziehen. Es ist allerdings zu beachten, dass die ermittelte Wichtigkeit nicht notwendigerweise zuverlässig ist (Molnar 2019). Der Einsatz weiterer klassischer Machine Learning Methoden, wie bspw. die statistische Regression, sollte daher ebenfalls in Betracht gezogen werden.

Diese Arbeit zeigt, dass Neuronale Netze keine einfache Lösung für Probleme jeder Art darstellen. Vergleichbar einfach zu implementierende klassische Machine Learning Methoden können zu effizienteren und besseren Lösungen führen.

Literaturverzeichnis

Bisgin, H., Kilinc, O. U., Ugur, A., Xu, X., & Tuzcu, V. 2011, Journal of Biomedical Science and Engineering, 264

Breiman, L. 2001, 5

Chen, P. Y., Chen, C. H., & Wang, H. 2009, Applied Computational Intelligence and Soft Computing

Chollet, F. & Allaire, J. J. 2018, Deep Learning mit R und Keras: Das Praxishandbuch (mitp)

Glorot, X., Bordes, A., & Bengio, Y. 2011, Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS), 315

Goodfellow, I., Bengio, Y., & Courville, A. 2016

Harrison, D. & Rubinfeld, D. L. 1978, Journal of Environmental Economics and Management, 81

Hastie, T. & Tibshirani, R. 2009

Kuhn, M. 2018, Journal of Statistical Software, 1

Molnar, C. 2019, Interpretable machine learning: a guide for making Black Box Models interpretable

Deep Feedforward Neural Networks

M. Wutke

Georg-August-Universität Göttingen, Germany

Zusammenfassung. Die Begriffe Deep Learning und maschinelles Lernen sind mittlerweile fester Bestandteil des täglichen Lebens und die Popularität künstlicher neuronaler Netzwerke nahm in den vergangenen Jahren stetig zu. Als eine der bekanntesten und verbreitetsten Netzwerkarchitekturen stellt die Klasse der Feedforward Netzwerke einen Einstieg in die Deep Learning Thematik dar und ermöglicht zugleich die Adressierung unterschiedlichster Problemstellungen. Um dem Leser ein zielorientiertes Grundwissen über die Implementierung und Anwendung solcher Netzwerktypen zu vermitteln, fokussiert sich diese Arbeit einerseits auf die Darstellung des modelltheoretischen Fundamentes und andererseits auf einen möglichst hohen Praxisbezug durch Betrachtung eines Klassifikationsbeispiels anhand eines frei verfügbaren Datensatzes.

1 Problembeschreibung

Die voranschreitende Digitalisierung und stetig ansteigende Rechenkapazität der vergangenen Jahre führte zu einem verstärkten Interesse im Bereich künstlicher Intelligenz und rückte die Thematik neuronaler Netzwerke stärker in den Fokus der Öffentlichkeit (vgl. Sun et al. 2017). Einer der bekanntesten Netzwerktypen stellt hierbei das Feedforward Netzwerk dar, welches die Adressierung verschiedenster Problemstellungen ermöglicht.

Die vorliegende Arbeit hat zum Ziel, eine Einführung in die Thematik neuronaler Netzwerke anhand des Feedforward Netzwerks zu geben. Hierfür wird zunächst die Struktur und generelle Wirkungsweise eines neuronalen Netzes in Kapitel 2.1 dargestellt. Darauf aufbauend erfolgt in Kapitel 2.2 die Beschreibung des Modelltrainings anhand des Backpropagation-Algorithmus. Kapitel 3 veranschaulicht die theoretischen Darstellungen durch ein Fallbeispiel anhand des MNIST-Datensatzes. Kapitel 4 rundet diese Arbeit durch ein Fazit und einen Ausblick auf weitere Forschungsarbeiten ab.

2 Thematische Einordnung

2.1 Deep Feedforward Netzwerke

Das oftmals als Deep Learning (DL) bezeichnete Gebiet neuronaler Netzwerke wird thematisch dem Bereich des maschinellen Lernens zugeordnet, was seinerseits eine Untergruppe der künstlichen Intelligenz darstellt. Im Unterschied zur klassischen

Programmierung, bei der sowohl die vorhandenen Daten als auch die anzuwendenden Bearbeitungsregeln als Inputfaktoren dienen und das Ziel darin besteht ein spezifisches Ergebnis zu produzieren, zielt maschinelles Lernen darauf ab, die Bearbeitungsregeln zur Erzeugung dieser Ergebnisse durch einen Algorithmus zu generieren. Als Inputfaktoren werden hierfür einerseits die vorhanden Daten und andererseits die Ergebnisse selbst verwendet. Das Erlernen der Bearbeitungsregeln wird in diesem Kontext als Modelltraining bezeichnet und benötigt oftmals eine nicht unerhebliche Menge an Eingangsdaten. Die auf diese Weise durch den Algorithmus erlernten Regeln lassen sich anschließend auf neue Daten übertragen, um neue Ergebnisse zu produzieren (vgl. Chollet 2018, S.4f).

Obwohl erste Untersuchungen im Bereich neuronaler Netzwerke bereits in den 50er Jahren unter dem Begriff *Artificial Neural Network* (ANN) durchgeführt wurden, dauerte es noch viele Jahrzehnte bis dieser Thematik erhöhte Aufmerksamkeit geschenkt wurde. Primärer Treiber des wachsenden Erfolgs neuronaler Netzwerke ist die voranschreitende Digitalisierung und verbesserte Computerleistung der vergangenen Jahre (vgl. Goodfellow et al. 2016, S.19f.). Die steigende Popularität führte zur Entwicklung verschiedenster Netzwerkarchitekturen, wobei einer der bekanntesten Netzwerktypen das Feedforward Netzwerk, auch Multi-Layer Perceptron genannt, darstellt (vgl. Sundermeyer et al. 2013). Der grundlegende Aufbau eines Feedforward Netzwerks ist in Abbildung 1 schematisch dargestellt.

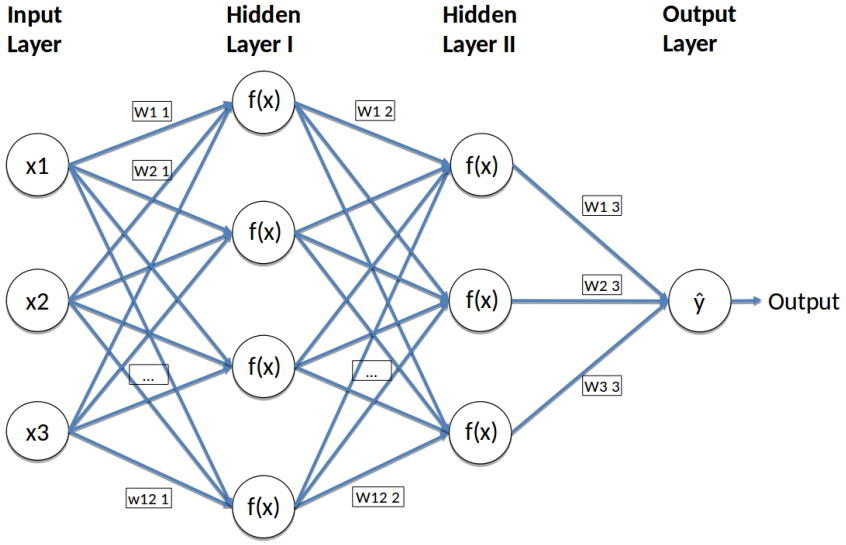


Abbildung 1: Die über die Input-Layer eingelesenen Eingangsdaten werden über die Hidden-Layer zur Output-Layer verarbeitet. Jedem Eingangspfad eines Neurons ist dabei ein spezifischer Gewichtungsparameter zugeteilt, welcher mit den Outputwerten der vorgelagerten Schicht multipliziert wird. Alle eingehenden Information eines Neurons werden anschließend summiert und durch eine Aktivierungsfunktion innerhalb des Neurons transformiert.

Wie in Abbildung 1 zu erkennen ist, besteht die Struktur eines Feedforward Netzes aus schichtweise angeordneten Lagen, die sich wiederum aus Untereinheiten, den Neuronen, zusammensetzen (vgl. Gurney 2014). Neuronen stellen im Kontext neuronaler Netze eine mathematische Funktion dar, welche ihren Input von Neuronen der vorgelagerten Schicht erhalten und innerhalb einer Schicht keine Verbindung untereinander aufweisen (vgl. Salzi 2006). Der spezifische Name des Feedforward Netzwerk ergibt sich aus der Richtung des Informationsflusses, bei dem die eingehenden Informationen durch eine Eingangsschicht (Input-Layer) eingelesen und über eine oder mehrere Zwischenschichten (Hidden-Layers) zur finalen Ausgangsschicht (Output-Layer) verarbeitet werden (vgl. Goodfellow et al. 2016, S.167).

Jedes Neuron der Hidden- und Output-Layer erhält als Eingangswert den Output vorgelagerter Neuronen, welcher zunächst mit einem separaten Gewichtungsparameter multipliziert und anschließend aufsummiert wird. Die gewichtete Summe dient als Eingangsinformation für eine nichtlineare Aktivierungsfunktion innerhalb des Neurons, welche darüber entscheidet, ob ein Neuron aktiviert oder als inaktiv eingestuft wird. Dieser Wirkungsmechanismus ist in Abbildung 2 für ein einzelnes Neuron dargestellt.

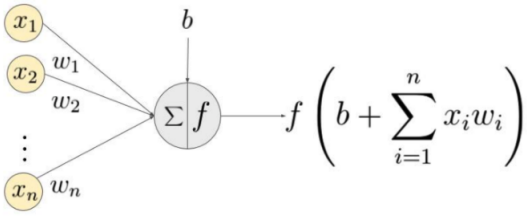


Abbildung 2: Die numerischen Eingangswerte werden jeweils mit einem individuellen Gewichtungsparameter multipliziert, zu einer Summe zusammengefasst und mit einem neuronenspezifischen Biasparameter addiert. Der daraus resultierende Wert wird als Eingangswert für die Aktivierungsfunktion verwendet (Verma & Singh 2015).

Beim betrachteten Feedforward Netzwerk findet für alle Neuronen innerhalb einer Schicht derselbe Funktionstyp Anwendung, wobei die Wahl einer geeigneten Aktivierungsfunktion durch die zu lösende Problemstellung determiniert wird (vgl. Ramachandran et al. 2017). Drei der am häufigsten verwendeten Aktivierungsfunktionen stellen die Sigmoid-Funktion, die hyperbolische Tangens-Funktion (tanh) und die Rectified Linear Unit-Funktion (ReLu) dar, welche in Abbildung 3 aufgeführt sind.

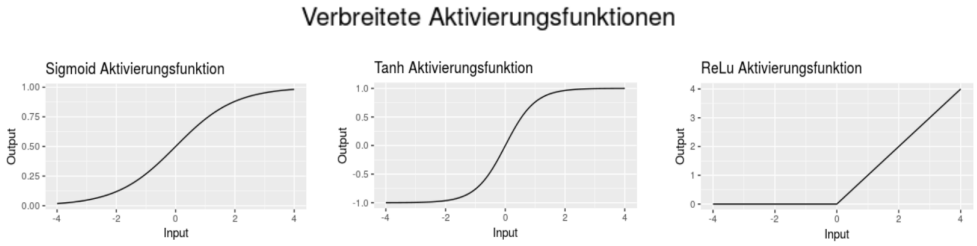


Abbildung 3: Übersicht über drei der gebräuchlichsten Aktivierungsfunktionen. Die Sigmoid-Funktion transformiert den Input auf einen Bereich $[0, 1]$. Die tanh-Funktion transformiert den reelwertigen Input auf einen Wertebereich $[-1, 1]$. Die ReLu-Funktion weist einem Input $x \leq 0$ den Wert Null zu. Für Inputwerte $x > 0$ wird der Wert des Inputs wiedergegeben.

Wie in Abbildung 3 zu erkennen ist, transformiert sowohl die Sigmoid-, als auch die tanh-Aktivierungsfunktion den numerischen Input auf einen eingeschränkten Wertebereich von $[0, 1]$ (Sigmoid) bzw. $[-1, 1]$ (tanh). Die Sigmoid-Funktion, definiert als

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

wird vor allem im Bereich binärer Klassifikationsaufgaben als Aktivierungsfunktion in der letzten Netzwerkschicht eingesetzt, da sich ihr Funktionswert als Wahrscheinlichkeit interpretieren lässt. Sowohl die hyperbolische Tangens-Funktion, definiert als

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2}$$

als auch die Rectified Linear Unit (ReLu)-Aktivierungsfunktion, definiert als

$$y(x) = max(0, x) \tag{3}$$

werden häufig im Bereich der Zwischenschichten eingesetzt, wobei sich besonders die ReLu-Funktion aufgrund ihrer einfachen Implementierung großer Beliebtheit erfreut (vgl. Goodfellow et al. 2016; Salehinejad et al. 2017).

Indem die vektorisierten Eingangsinformationen in das neuronale Netz eingelesen und anschließend schichtweise in Richtung der Outputschicht verarbeitet werden, wird die Approximation nichtlinearer, mehrdimensionaler Funktionen ermöglicht. Das grundlegende Wirkungsprinzip basiert hierbei auf einem Vergleich der Eingangsdaten mit vorab festgelegten Zielwerten im Rahmen einer Verlustfunktion $L(y, \hat{y})$. Hierbei lassen sich zudem alternative Begriffe wie Kostenfunktion, Fehlerfunktion oder Objektfunktion für die Verlustfunktion in der gängigen Fachliteratur finden. Im Rahmen des Modelltrainings gilt es, die Modellparameter (Gewichte) so anzupassen, dass die Verlustfunktion minimiert wird (vgl. Goodfellow et al. 2016). Der Prozess des Modelltrainings wird im folgenden Kapitel beschrieben.

2.2 Modelltraining und -optimierung

Neben der Auswahl geeigneter Aktivierungsfunktionen ist auch die Wahl des Verlustkriterium von der gegebenen Problemstellung abhängig, wobei sich in der bestehenden Forschungsliteratur die Verwendung einiger spezifischer Kriterien etabliert hat. Im Fall eines multinomialen Klassifikationsproblems findet beispielsweise das kategorische Kreuzentropie-Kriterium (Gleichung 4) Anwendung, wohingegen der Mean Squared Error-Loss (Gleichung 5) häufig bei Regressionsproblemem verwendet wird (vgl. Salehinejad et al. 2017).

$$L(y, \hat{y}) = -(y_i - log(\hat{y_i}) + (1 - y_i)\ log(1 - \hat{y_i}) \tag{4}$$

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{5}$$

Nachdem die Eingangswerte durch das Netzwerk verarbeitet und der numerische Wert der Verlustfunktion gebildet wurde, werden die Modellgewichte der Neuronen auf Grundlage des Backpropagation-Algorithmus (BP-Algorithmus) aktualisiert und der Wert der Verlustfunktion reduziert. Je nach Häufigkeit der Anwendung des Backpropagation-Algorithmus wird somit eine lokale Minimierung der Verlustfunktion angestrebt.

Der BP-Algorithmus selbst verwendet hierfür die Methode des steilsten Abstiegs, auch Gradientenverfahren genannt, bei dem die Gradienten der Aktivierungsfunktionen schichtweise durch Anwendung der Kettenregel berechnet werden (vgl. Rojas 2013; Hinton & Salakhutdinov 2006). Gleichung 6 zeigt diesbezüglich beispielhaft die Anwendung des Gradientenverfahrens für ein Netzwerk mit einer Tiefe von zwei Hidden-Layers sowie einer Input-Layer und einer Output-Layer. Die Funktion L bezeichnet hierbei die Verlustfunktion, W die Gewichte des Netzwerks, $\hat{y}$ steht für den berechneten Wert der Ouput-Layer und X und Z für den Output der Hidden-Layer.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial X} \frac{\partial X}{\partial Z} \frac{\partial Z}{\partial W_Z} \tag{6}$$

Wie zu erkennen ist, verwendet der BP-Algorithmus die Kettenregel, um ausgehend von der Output-Layer $\hat{y}$ den Gradienten der Netzwerkschicht Z nach den Gewichten in dieser Schicht W_Z zu bestimmen. Basierend auf den Informationen des Gradienten werden die korrespondierenden Gewichte in die entsprechende Richtung aktualisiert. (vgl. Goodfellow et al. 2016, S.204ff.)

3 MNIST-Fallbeispiel

Das nachfolgende Kapitel dient der Darstellung der Wirkungsmechanik eines Feedforward Netzwerks anhand des bekannten MNIST-Datensatzes (LeCun et al. 1998). Dieser Datensatz ist frei verfügbar und enthält 70.000 annotierte Abbildungen über handgeschriebene Ziffern im Zahlenbereich von Null bis Neun, wovon 60.000 Abbildungen in Trainingsdatensatz und 10.000 Abbildungen im Testdatensatz unterglie-

dert sind (vgl. Kussul & Baidyk 2004). Die Abbildungen sind bereits in numerischer Form als 28 x 28 Matrix mit Pixelwerten im Graustufenformat verfügbar.

Für dieses Fallbeispiel wird ein Feedforward Netzwerk mit einer Netzwerktiefe von zwei Hidden-Layers und einer Neuronenanzahl von 128 und 256 Neuronen implementiert. Für die Netzwerkimplementierung wird die Programmiersprache R (Version: 3.6.2) und die Deep-Learning-Programmbibliothek Keras (Version: 2.2.5) verwendet. Keras dient in diesem Zusammenhang als high-level-API für die Deep-Learning Plattform Tensorflow und wurde sowohl für die Programmiersprache Python als auch für R veröffentlicht und ermöglicht eine verhältnismäßig einfache Implementierung verschiedenster Netzwerkmodelle (Arnold 2017).

Da die Aufgabe des Netzwerks darin besteht eine eingelesene Ziffer einer von zehn Klassen zuzuordenen, wird eine Output-Layer mit zehn Neuronen und einer Softmax-Aktivierungsfunktion gewählt. Für die Hidden-Layers wird die ReLu-Funktion verwendet. Da die Input-Layer einen vektorisierten Input verlangt, werden die Zeilen der Pixelmatrizen des Trainingsdatensatzes aneinandergereiht und zu einem Vektor mit einer Länge von 784 (28×28) transformiert. Die Eingangsschicht verfügt somit über 784 Neuronen und einer ReLu-Aktivierungsfunktion. Über alle Netzwerschichten ergibt sich somit eine Gesamtanzahl an Netzwerkgewichten von 94.154 Modellparametern, deren Ausprägung im Rahmen des Modelltrainings iterativ ermittelt wird. Für den Backprogation-Algorithmus wird dem Vorgehen früherer Publikationen wie von Kurbiel & Khaleghian (2017) gefolgt und der RMSProp-Algorithmus verwendet, welcher eine Modifikation des Gradientenverfahrens darstellt, bei der eine schnellere Konvergenz des Algorithmus durch eine Adjustierung der Lernrate erreicht wird (vgl. Kurbiel & Khaleghian 2017). Darüber hinaus wird das Modell über 25 Iterationen mit einem kategorischen Kreuzentropie-Kriterium trainiert und über einen zehnprozentigen Validierungssplit evaluiert. Um das Modelltraining zu verkürzen, wird eine Batchgröße von 10 gewählt, wodurch dem Netzwerk in einem Verarbeitungsschritt 10 Abbildungen zur Verfügung gestellt werden. Bevor das Netzwerk trainiert werden kann, müssen die Daten in eine geeignete Struktur gebracht werden. Da der erforderliche MNIST-Datensatz bereits in der Keras-Bibliothek vorimplementiert ist, werden die Daten nach dem Einlesevorgang in einen Trainings- und Testdatensatz unterteilt. Diese Differenzierung ist notwendig, da das Modelltraining lediglich anhand der Trainingsdaten durchgeführt wird. Die Modellevaluation erfolgt anschließend anhand des Testdatensatzes. In einem nachfolgendem Prozessschritt werden die Inputdaten wie oben beschrieben in eine Vektordarstellung überführt und die annotierten Zieldaten in einen Vektor der Länge zehn mit einer one-hot-Kodierung transformiert. Diese Transformation ergibt sich aus der Struktur der Output-Layer, da für jede Zahlenkategorie ein separates Neuron vorhanden sein muss. Um einen Vergleich der Outputwerte mit den Zielwerten zu ermöglichen, ist somit eine one-hot-Kodierung der annotierten Daten notwendig. Nachdem die Datensätze in eine für das Netzwerk geeignete Struktur überführt wurden, wird das Feedforward Netzwerk über 25 Perioden trainiert. Wie in Abbildung 4 zu erkennen ist, erreicht das Modell nach 25 Trainingsperioden einen Wert der Verlustfunktion von 0.0189 für den Trainingsdatensatz und 0.0914 für die Validierungsdaten. Hinsichtlich der Genauigkeit erzielt das Feedforward Netzwerk einen Wert von 0.9943 für die Trainingsdaten und 0.9775 für die Validierungsdaten.

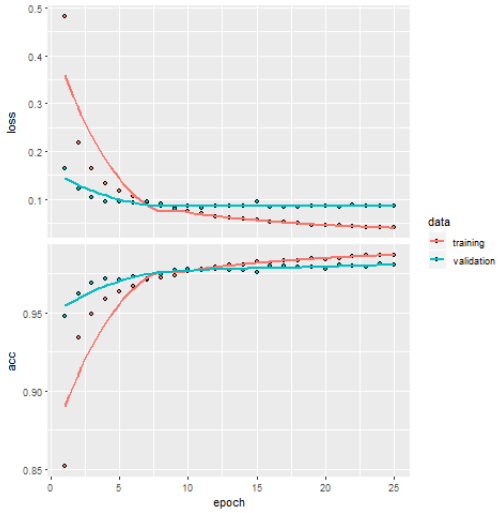


Abbildung 4: Die Abbildung zeigt den Wert der Verlustfunktion (kategorische Kreuzentropie) sowie die Genauigkeit für den Trainings- als auch den Validierungsdatensatz. Am Ende des Trainingsprozesses wird für den Validierungsdatensatz eine Genauigkeit von 0.9775 bei einem Verlustwert von 0.0914 erzielt.

Für die Modellevaluation werden die Testdaten in das nun trainierte Netzwerk eingelesen und mit den Zielwerten verglichen. Das Ergebnis ist in Tabelle 1 anhand einer Konfusionsmatrix abgebildet.

Tabelle 1: Die Elemente auf der Hauptdiagonalen der Konfusionsmatrix stellen die Anzahl korrekt klassifizierter Beobachtungen dar. Die Elemente auf den Nebendiagonalen beschreiben die Anzahl falsch klassifizierter Beobachtungen.

Realität / Vorhersage	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	972	0	2	0	2	2	3	2	2	2
1	0	1121	0	0	0	0	2	1	0	3
2	1	2	1003	3	3	1	0	7	1	0
3	2	2	3	987	81	11	1	2	5	8
4	0	0	3	0	951	1	2	0	1	4
5	1	1	1	6	1	864	6	0	6	3
6	2	1	3	0	7	5	942	0	3	1
7	1	1	10	6	0	2	0	1010	7	6
8	1	7	6	5	1	4	2	2	948	5
9	0	0	1	3	16	2	0	4	1	977

Die Konfusionsmatrix vermittelt einen Überblick über die Häufigkeiten korrekter und inkorrekter Klassifikationen. Die Summe der Elemente auf der Hauptdiagonalen beschreibt hierbei die Anzahl richtig klassifizierter Beobachtungen. Von den 10.000 Beobachtungen im Testdatensatz konnten 9.701 Beobachtungen korrekt eingestuft werden. Um die Fehlerrate von 0.0299 zu reduzieren, gilt es in einem nachfolgendem Prozessschritt die Hyperparameter des neuronalen Netzes, wie bspw. die Netzwerktiefe, die Anzahl der Trainingsperioden oder den verwendeten Optimierungsalgorithmus zu verbessern. Aus Gründen der Einfachheit wird dieser Prozessschritt im Rahmen dieser Arbeit nicht betrachtet.

4 Fazit

Die vorliegende Arbeit hatte das Ziel eine Einführung in die Thematik der Feedforward Netzwerke zu geben. Hierzu wurden die grundlegenden Mechanismen neuronaler Netzwerke und deren Struktur erörtert und im Rahmen eines Fallbeispiels dargestellt. Aufgrund der steigenden Popularität künstlicher neuronaler Netzwerke, halten netzwerkbasierte Problemlösungen vermehrt Einzug in unser alltägliches Leben, wobei Feedforward Netzwerke als ein erster Einstieg in diese Thematik betrachten werden können. Das Potenzial dieser Modellklasse ist noch nicht erschöpft und neue Erkenntnisse im Bereich der Netzwerkarchitekturen und Trainingsalgorithmen könnten ihren Erfolg in den kommenden Jahren noch verstärken (Goodfellow et al. 2016).

Aus Gründen der Einfachheit beschränkte sich diese Arbeit auf die wesentlichen Charakteristika neuronaler Feedforward Netzwerke. Für eine tiefergehende Beschreibung dieses Netzwerktypes empfiehlt sich eine intensivere Betrachtung der verschiedenen Optimierungsalgorithmen und Regularisierungstechniken. Darüber hinaus erweitern alternative Netzwerkstrukturen wie Konvolutionsnetzwerke oder rekursive Netzwerke die Einsatzmöglichkeiten neuronaler Netze auf hochdimensionale Datenstrukturen wie Audio- und Videodaten oder zeitlich korrelierte Daten wie Zeitreihen (vgl. LeCun et al. 1995).

Literaturverzeichnis

Arnold, T. 2017, Journal of Open Source Software, 2, 296
Chollet, F. 2018, Deep learning with Python (Shelter Island, New York: Manning Publications Co), oCLC: ocn982650571
Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (Adaptive Computation and Machine Learning series) (The MIT Press)
Gurney, K. 2014, An introduction to neural networks (CRC press)
Hinton, G. E. & Salakhutdinov, R. R. 2006, science, 313, 504
Kurbiel, T. & Khaleghian, S. 2017, arXiv preprint arXiv:1708.01911
Kussul, E. & Baidyk, T. 2004, Image and Vision Computing, 22, 971
LeCun, Y., Bengio, Y., et al. 1995, The handbook of brain theory and neural networks, 3361, 1995
LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. 1998, Proceedings of the IEEE, 86, 2278
Ramachandran, P., Zoph, B., & Le, Q. V. 2017, arXiv preprint arXiv:1710.05941
Rojas, R. 2013, Neural networks: a systematic introduction (Springer Science & Business Media)
Salehinejad, H., Sankar, S., Barfett, J., Colak, E., & Valaee, S. 2017, arXiv preprint arXiv:1801.01078
Salzi, M. 2006, Communications, Faculty Of Science, University of Ankara, 11
Sun, C., Shrivastava, A., Singh, S., & Gupta, A. 2017, Proceedings of the IEEE international conference on computer vision, 843
Sundermeyer, M., Oparin, I., Gauvain, J.-L., et al. 2013, 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 8430
Verma, K. & Singh, P. K. 2015, International Journal of Modern Education and Computer Science, 7, 52

An Introduction to Deep Learning and the Concept of Regularization

N. Bosse

Georg-August-Universität Göttingen, Germany

Abstract. With the advent of ever-increasing computing power, learning algorithms dating back multiple decades have recently started to see astounding accomplishments in solving a variety of real-world tasks. While neural networks are made of simple building blocks that perform basic arithmetic operations, the ensemble can handle complex tasks very well, given enough data and computing power. Neural networks are therefore used in a multitude of fields such as language analysis, self-driving cars, face recognition and even gaming. This essay will give an introduction to the basic concepts behind deep learning and neural networks and will outline the idea underlying regularization, an ensemble of techniques used to increase the predictive performance of neural networks. First, the basics of machine learning, in general, and deep learning, in particular, are shortly presented and general characteristics of deep neural networks are detailed. Secondly, the need for regularization is motivated and the following regularization techniques are introduced: Early Stopping, L2- and L1-regularization, and regularization through Data Set Augmentation. Three examples are used for illustrative purposes: The first example deals with the task of assessing whether reviews on the Internet Movie Database imdb are positive or negative. The dataset of 50,000 imdb reviews is included in the Keras package used to set up the neural network in R. In the second example, a set of short newswires, published by Reuters in 1986 is classified into 46 different topics. The third example deals with classifying handwritten digits from the famous MNIST data set. All data sets are annotated, thus the learning can be classified as supervised learnings. The examples are adapted from Chollet & Allaire (2018).

1 The Basics of Deep Learning

1.1 Machine Learning, Deep Learning and Neural Networks

Machine learning, in essence, is about transforming input data to obtain more useful and meaningful representations. In a classical linear regression or a principal component analysis, for example, this means obtaining insights by looking for linear combinations of variables that best explain variation in a target variable, or the entire dataset, respectively. Deep learning takes this approach further by applying multiple layers of increasingly useful transformations to the data. This is illustrated in Figure 1, which shows the sketch of a neural network tasked with the classification of handwritten digits from 0 to 9. Transformations are applied successively to obtain representations of the data that are increasingly useful for the classification task at

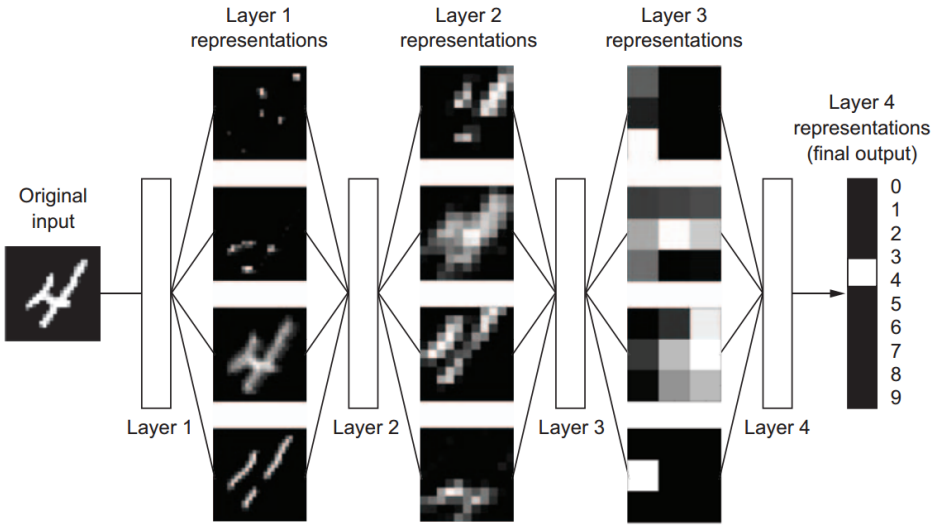


Figure 1: Schema of a neural network that classifies handwritten digits from 0 to 9. Reference: Chollet & Allaire (2018).

hand. In this sense, the layers can be thought of as filters that filter out more and more relevant information. In a more mathematical sense, the layers of a neural network can be interpreted as functions that are applied to tensors, i.e. to multi-dimensional arrays. Tensors hold all the information present in the neural network. The original data, fed in the network as a tensor, is modified through successive layers that apply transformations and each output another tensor. What transformations are applied by a specific layer depends on that layer's activation function and its weights, which are in turn stored in tensors. A typical layer of a neural network might look like this:

$$Z = relu(W'X + b) = \max(0, W'X + b) \tag{7}$$

where X is the tensor containing a representation of the data and W and b are tensors containing the weights, the trainable parameters of that specific layer. The rectified linear unit (ReLU) function replaces every single value smaller than zero with zero, making the transformation non-linear. This ability to model non-linear functions is one of the features that give great power to the neural network as a whole. The output of the function is the tensor Z that will be used as the input tensor for the next layer which applies a new transformation on it. The last layer of the network is called output layer and usually outputs some kind of prediction or classification.

1.2 Learning

In order to learn, the machine learning procedure needs input data, a criterion for success and a way to adapt according to its current performance. Usually, results of the output layer of the neural network are therefore compared against some target, e.g. the true known values in the training data. The difference between predictions and true values is measured by some predefined loss function (also called cost function) $J(\theta; X, y)$. θ holds the weights of the network (W and b), X is the input, y, is the target, and the cost function itself is the criterion for success. The values of the tensors W and b are initially set to random values. Initial predictions are made and the current loss is calculated. The weights W are then updated in a way that slightly reduces the loss. This update of the weights is done iteratively until the performance on the training data is optimized.

This weight updating is achieved by calculating the gradient of the loss function with respect to all individual weights of the network. In order to reduce the loss, weights have to be moved by a certain factor (the learning rate) in the opposite of the gradient at the current value of the loss. The individual weight components ω

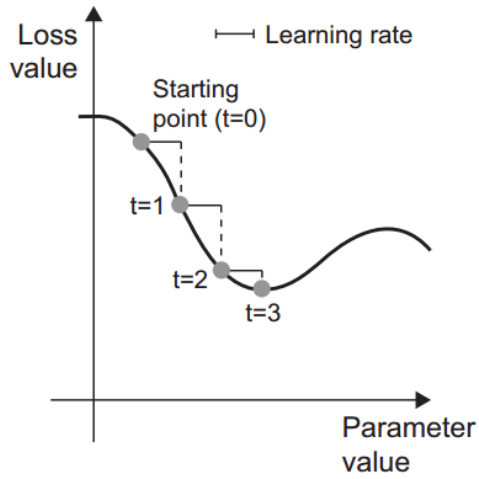


Figure 2: Univariate illustration of gradient descent. The weight is iteratively increased to minimize loss. Reference: Chollet & Allaire (2018)

are updated according to

$$\omega \leftarrow \omega - \nabla_\omega J(\omega, X, y), \tag{8}$$

where $\nabla_\omega J(\omega, X, y)$ is the gradient of the loss function at the current value of ω. In the illustration shown in Figure 2, the gradient (slope) of the loss at the current weight values is negative, and therefore the parameter value of the weight is successive increased until the loss is minimal. This iterative process of weight adjustment and

reevaluation is called gradient descent. There exist a number of variations; stochastic gradient descent, for example, only makes adjustments for a randomly drawn set of weights. The general idea, however, remains the same.

1.3 Over- and Underfitting

At the beginning of the learning process, predictions made by the network will be inaccurate because of underfitting. Underfitting characterizes the situation where a neural network has (yet) failed to learn the complexity of the features of the training data set to a sufficient extent. This is illustrated in the left panel in Figure 3. The simple model (a straight line) is not able to account for the complexity of the underlying relationship in the data. Predictions based on this fit will exhibit what is called bias - errors that result from a systematic failure to capture the true nature of the data generating process. Through iterative weight updating, the loss, i.e. the mismatch between predictions and true values in the training data set, is continuously reduced. Over time, the performance on the training data set therefore gradually improves as the network learns to absorb the specifics of the training data. While the fit to the training data can only improve with every weight update, the predictive performance vis-à-vis a second validation data set, a measure of generalizability, may eventually degrade. At some point, illustrated in the right panel in Figure 3, the network will overfit the data. It has then become extremely good at explaining the specific features of the training data, but increasingly worse at recognizing generalizable trends and features not specific to the training data. The model will not be able to produce stable and reliable predictions on new data. This pattern of steadily decreasing loss with respect to the training data set and increasing, then decreasing performance with regards to the validation data is illustrated for example in Figures 4 and 6. Both figures show the loss calculated on the training and the validation data set as the learning process progresses, as well as the accuracy of the predictions.

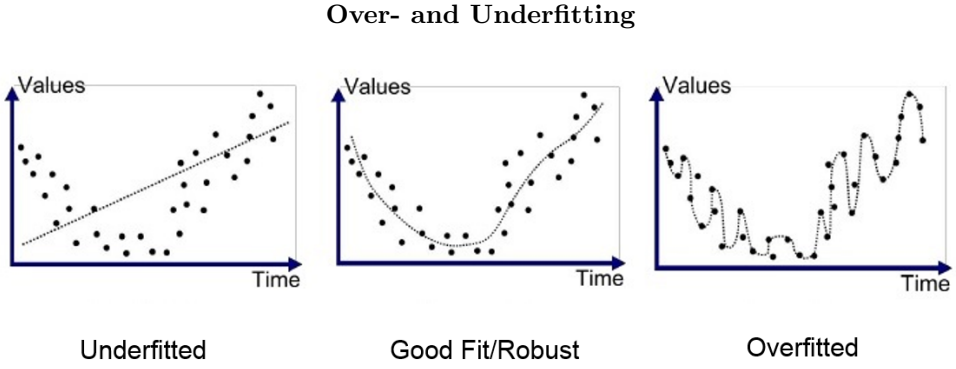


Figure 3: Illustration of different stages of the learning process (underfitting, good fit, overfitting). Reference: Bhande (2018)

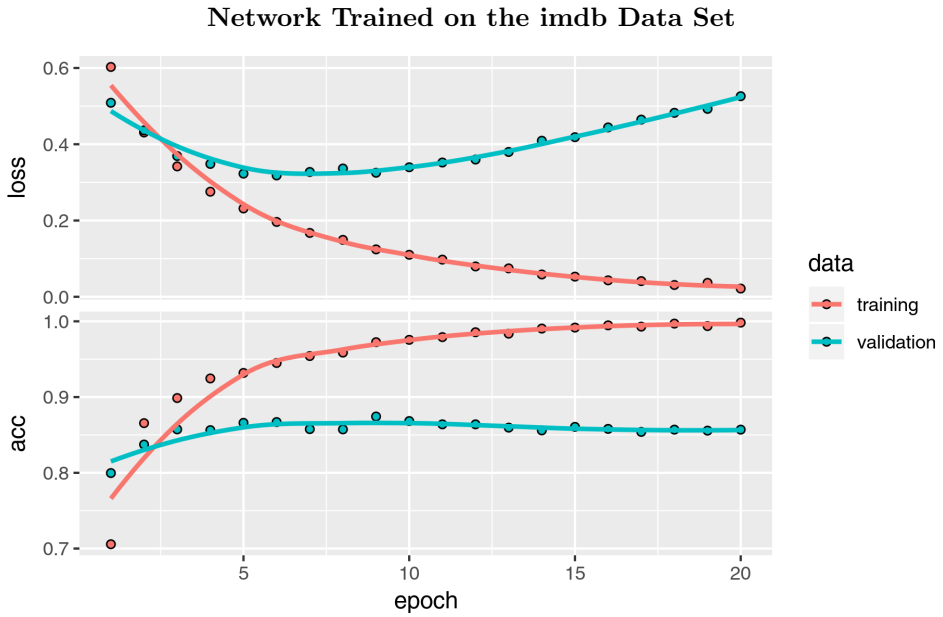


Figure 4: Training a neural network to classify whether movie reviews on the Internet Movie Data Base imdb are positive or negative. After epoch 6 we see overfitting, as the performance of the network on the validation data set decreases (validation loss increases again).

1.4 Regularization

Machine learning approaches require fine balancing. While their goal is to make predictions that are highly generalizable to new data, their only means of learning is optimization on a limited set of training data. The algorithm will always be torn between overfitting and underfitting, trying to balance biases (due to missing important features of the data), with excessive variance in predictions (due to taking too many random features of the given data into account). A good model fit is illustrated in the middle panel of Figure 3. A means of balancing this fine line is regularization. Regularization, in essence, comprises different strategies to make neural networks select models that generalize well. The goal is to reduce variance, i.e. make predictions more robust across different new input sets, without increasing bias, i.e. without increasing the error due to a systematic failure to account for the true complexity of the training data. Usually, this is done by nudging the network to focus on the most simple and prominent features of the data, Practice shows that, generally, complex models that have been regularized to select simple solutions perform very well.

1.5 Regularization and Network Capacity

The need for regularization increases with the capacity of the neural network. Capacity is a measure of the complexity of the network and comprises for example the number of layers a network has and the size of the layers, i.e. the number of individual weights that can be adjusted. The higher the capacity, the more a network is able to absorb complex relationships from the data and come up with more complex models to make predictions. While overfitting eventually happens for all kinds of models that are trained for a long enough period of time, it happens much faster for complex models than for simple ones. The reason is that smaller models are much less capable of (and slower in) absorbing information from the data than more complex ones. This is illustrated in Figure 5, where two different networks with different capacities are tasked with predicting whether movie reviews are positive or negative. Compared to the network used in Figure 4 the size of each layer has been increased or decreased, respectively. For the large network, we see almost immediate overfitting, while the network with smaller capacity reduces validation loss only slowly. Note that the loss of the smaller network is much higher than the loss of the bigger model in the beginning. We can therefore conclude that the smaller network is still underfitting and may never be able to adequately capture the complexity of the data. While usually not explicitly mentioned as a form of regularization, reducing the capacity of a model can therefore also be understood in terms of regularization.

2 Regularization Strategies

There are a large number of different regularization strategies. The following collection of techniques is by no means exhaustive but is rather designed to give the reader a good intuition about the ideas behind regularization.

2.1 Early Stopping

Perhaps the simplest form of regularization is early stopping. Early stopping treats the duration for which a model is trained as a hyperparameter to be optimized. The duration of training is usually expressed in epochs. One epoch represents one pass of the entire training data set through the network. Alternative measures are for example the total number of weight updates performed. A practical algorithm for early stopping can be implemented as follows: The data set is split into three parts: A training data set, a validation data set and a test data set. The network will be trained on the training data set. After every epoch, the current state of the neural network is stored and the performance of the network is tested on the validation data set. This performance is recorded over time. Learning stops when the performance on the validation set does not increase for a predefined number of times. The optimal number of epochs (or alternatively, the optimal number of weight updates done) is registered and kept. In order to not waste the validation data, the network is sequentially trained once more on the ensemble of training and validation data set with that optimal number of epochs. The final performance can then be evaluated on the test data set. The reason why the data set is split into three is to avoid spilling

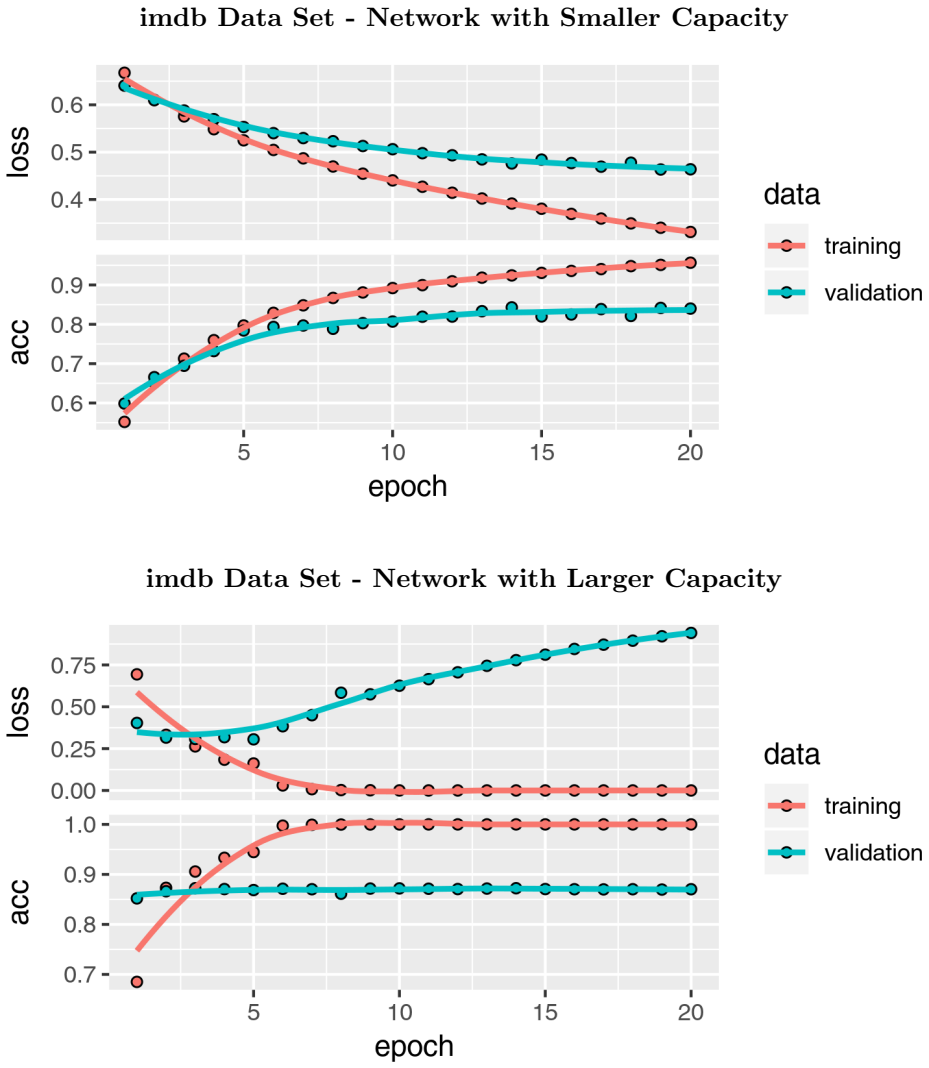


Figure 5: Training and validation for two networks with differently sized layers.

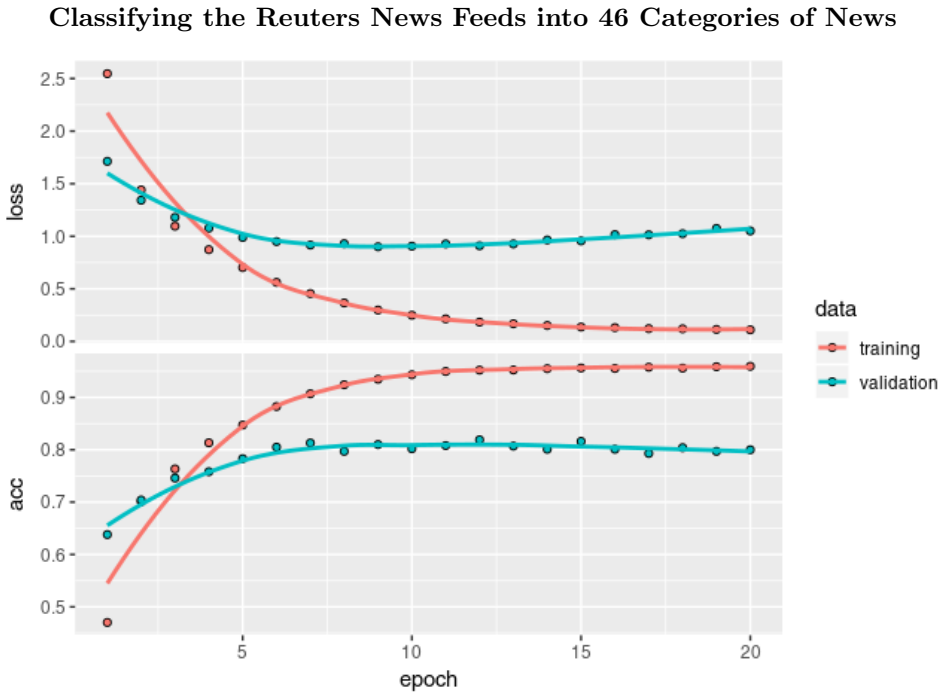


Figure 6: Loss and prediction accuracy for a neural network tasked to classify Reuters news feeds into 46 categories. The optimal number of training epochs is 9.

.

information from the test data set into the network by using it for validation after each epoch. The performance of a network can only be fairly evaluated by testing it against completely novel data.

As recording the states of the network at different time points is very simple and the number of epochs to be trained is always a parameter to be optimized, early stopping is employed in virtually any deep learning endeavor. It is important to note, however, that limiting the number of training epochs is also a form of regularization, as this in effect also limits the number of weight updates and thus the complexity of information that can be learned from the data. Limiting the time spent on training limits, therefore, the capacity of the network to overfit, i.e. to absorb specifics from the data that do not generalize well.

Figure 6 shows the training and validation loss over 20 epochs for a network trained on the Reuters data set to classify news articles into 46 categories. One can see that the loss computed on the validation data set first reduces with each iteration, but starts to rise again after epoch 9. To implement early stopping regularization, one can therefore take 9 to be the optimal number of epochs and retrain the model with 9 epochs, this time including the validation data set. Increasing the number of examples for training increases performance on the test set further.

2.2 Parameter Norm Penalties

Parameter norm penalties are among the most widely used regularization techniques not only in deep learning, but also in other domains of machine learning and statistics. They are implemented by amending the above introduced cost function with a complexity penalty. The new regularized cost function is

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta). \tag{9}$$

There are different possibilities to model $\Omega(\theta)$. The two most common, and thus the two presented in the following, are L1- and L2-regularization.

2.3 L2 Normalization

For L2-regularization, the parameter norm penalty has the form

$$\Omega(\theta) = \frac{1}{2}\|\omega\|_2^2 = \frac{1}{2}\omega'\omega \tag{10}$$

The penalty thus corresponds to the sum of all squared weights. As will become evident in a moment, L2-regularization is also called weight decay, as weights are constantly pulled towards zero in each update step. In classical regression settings, L2-regularization is also known as ridge-regression. Based on the cost function

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \frac{\alpha}{2}\omega'\omega \tag{11}$$

and its gradient

$$\nabla_\omega \tilde{J}(\theta; X, y) = \nabla_\omega J(\theta; X, y) + \alpha\omega \tag{12}$$

the weights are updated in each step per

$$\omega \leftarrow \omega - \epsilon(\nabla_\omega J(\omega; X, y) + \alpha\omega) \tag{13}$$

$$\omega \leftarrow (1 - \epsilon\alpha)\omega - \epsilon\nabla_\omega J(\omega; X, y) \tag{14}$$

Equation 14 corresponds to the weight updating shown in equation 8, apart from the fact that the weights are shrunk by $(1 - \epsilon\alpha)$ with every iteration. It is important to note that this shrinkage is geometric. This means that while weights are continuously pulled towards zero they will never reach zero. The left side of Figure 7 gives a graphical intuition for L2-regularization. The ellipsoids and the gray circle represent the two terms of the cost function in equation 9 that need to be minimized, the unregularized cost function and the complexity penalty. For cost functions like the mean squared error (MSE), points with equal losses form ellipsoids around the minimum possible loss. The shape of the isocurves of equally sized penalties around the minimum penalty depends on the penalty function. The resulting weight will always lay on the tangent of two isocurves. The magnitude of the regularization parameter α determines how much the weight is drawn to one or the other extreme. As is intuitive by looking at the shape and curvature of the loss isocurves, weights

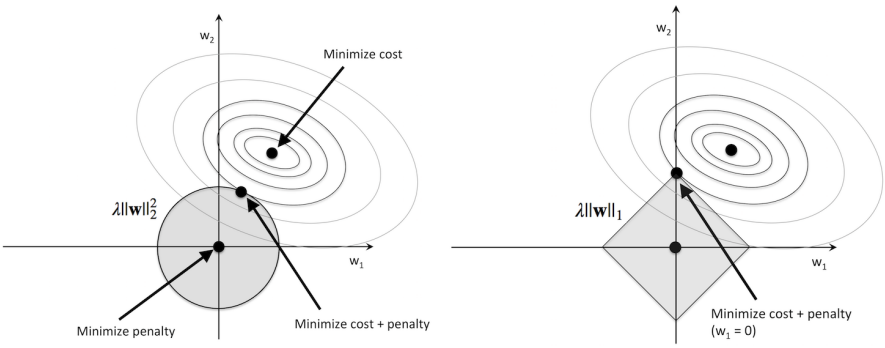


Figure 7: Graphical intuition for L2-regularization (left) and L1-regularization (right). Reference: Sebastian Raschka (2019)

that have less impact on the loss (less curvature of the isocurves) are more strongly drawn towards zero. Figure 8 illustrates the effect of L2-regularization applied to the imdb data set. The overall performance of the regularized network is worse than in the case of the unregularized network before, as the capacity and the penalty rate α were not optimized. Note however that the regularized version of the network is indeed much less prone to overfitting than the original model.

2.4 L1-Regularization

L2-regularization will shrink weights near zero, but will not make them exactly zero. In order for weights to be actually set to zero, L1-regularization can be used. This is graphically illustrated on the right side in Figure 7. The special shape of the L1-regularization shown isocurves induces weights to be more likely set to zero, resulting in a sparse weight matrix. In a classical regression setting, L1-regularization is also known as LASSO and is used as a form of model selection by setting unimportant parameters to zero. The penalty for the L1-regularization is

$$\Omega(\theta) = \|\omega\|_1 = \sum_i \omega_i \tag{15}$$

This results in the cost function

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\|\omega\|_1 \tag{16}$$

and the gradient

$$\nabla\tilde{J}(\theta; X, y) = \nabla J(\theta; X, y) + \alpha \cdot sign(\omega) \tag{17}$$

the corresponding weight updating is therefore

$$\omega \leftarrow \omega - \alpha \cdot sign(\omega) - \nabla J_\omega(\omega; X, y) \tag{18}$$

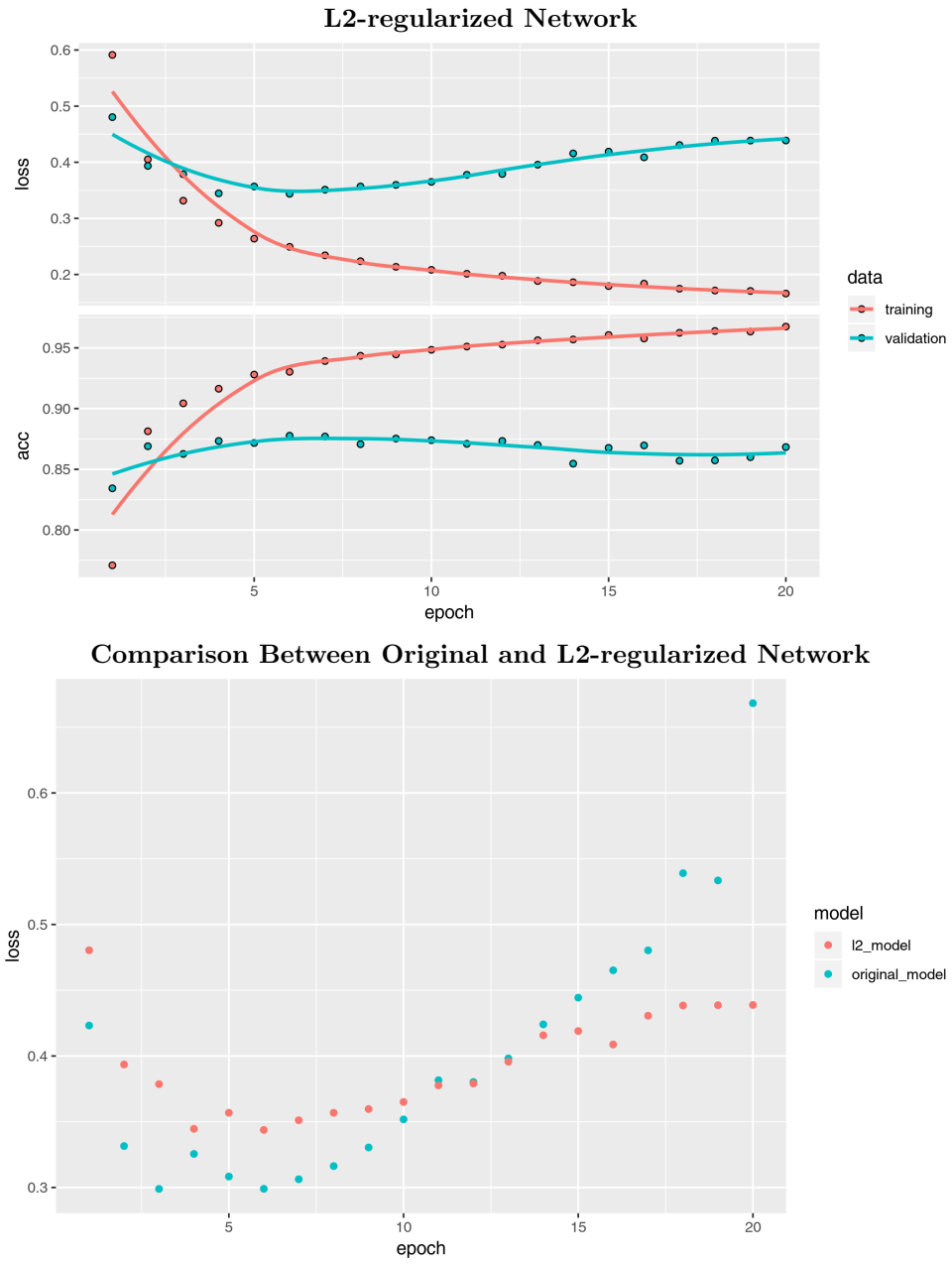


Figure 8: Simple example of an application of L2-regularization to reviews from the imdb data set.

In contrast to the proportional decrease that we saw with L2-regularization, $|\omega|$ is now reduced by a constant amount, α, with each weight update. This means that weights can actually get set to zero which results in a sparse representation of the weight matrix. As with L2-regularization, the weights that are most pulled towards zero are the ones that would least influence loss, if reduced. Therefore, L2- and L1-regularization implicitly make sure that the weights most important for prediction are allowed to be high while the rest is regularized towards zero. The concept of parameter norm penalties can be generalized to higher dimensions, but L1- and L2-regularization are the ones most frequently used.

2.5 Optimizing the Hyperparameters

The regularization parameter α is a hyperparameter that needs to be optimized. Oftentimes it is exactly this kind of optimization that makes deep learning difficult or at least computationally very expensive. Given unlimited computing power, one would test every combination of hyperparameters and validate its performance. If enough data is available, validation can be done on a validation data set. If the amount of data is very small, it is more efficient and robust to use approaches like k-fold cross-validation. For k-fold cross-validation, samples are iteratively drawn at random as validation sets. The remaining data is then validated against those different randomly drawn sets of the data. Optimizing hyperparameters in this regard means changing for example α for a given capacity of each layer and a given number of training epochs, then changing the capacity for given values of α and training duration and so on. As retraining the model every time and evaluating its performance is computationally expensive, this soon becomes unfeasible.

Several strategies try to improve on this. Instead of trying out all possible values one could define a possible range of values for the different hyperparameters and choose to try out a predefined number for each. This is called Grid Search, as the combinations to be tried of can be thought of as a multidimensional grid. More efficiently than that, combinations of different parameters can be checked at random within predefined boundaries. The advantage of exploring random combinations is that for every parameter, many more values are tried if parameters are not held constant. This is illustrated in Figure 9, where nine runs are performed to test different combinations of hyperparameters. With the use of grid search on the left, three parameter values are tried for each parameter. Random search, shown on the right, tests nine values for every parameter (as it is very unlikely to choose the exact same value twice) and therefore explores the marginal distributions much more thoroughly. One downside with random search is that there is less of a guarantee to find the optimal combination quickly. One can further improve on this by directing the search towards smaller ranges of values that look promising. This somewhat mitigates the inefficiency that stems from the different steps of the search being independent. Being able to learn from previous trials makes this search more effective (while on the other hand limiting the researcher to a sequential approach). Going further in this regard leads to a variety of search algorithms that use Bayes statistics and machine learning to solve the problem at hand. Sequential Model Based Optimization (SMBO) algorithms for example use concepts like ‘expected improvement’

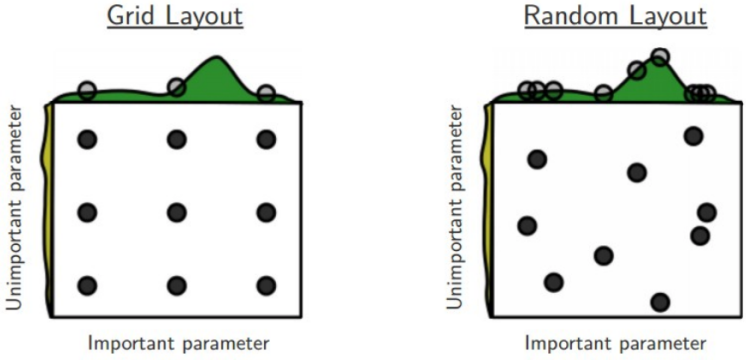


Figure 9: Illustration of Grid Search (left) and Random Search (right) to search optimal values for combinations of hyperparameters. Reference: Bergstra & Bengio (2012)

.

to combine uncertainty about hitherto unexplored parameter values with previously acquired knowledge to make educated guesses about the most promising values for the hyperparameters to be evaluated.

2.6 Data Set Augmentation

We have defined regularization as a means of reducing variance, i.e. stabilizing the performance of the neural network across different sets of new data, without increasing its bias, i.e. the systematic ways in which the network fails to grasp the true underlying data generating process. In this sense, we can interpret data set augmentation as a form of regularization. It makes intuitive sense that a more diverse dataset forces the network to only learn features that generalize well. A network that is tasked with speech recognition will perform much better on a given new input, if it has already analyzed a multitude of different voices, instead of having been trained on only one voice. Instead of learning particularities of one voice, this network will focus on features that are shared across different speakers. Usually, the data available for training is limited. One can, however, try to make learning more robust and to reduce sensitivity to specific features of the data by training the network on slight variations of the input data, instead of using the same data set for every epoch. For image recognition, this can for example mean to artificially distort pictures or to tilt them or change the hue and color. For speech input this could be done by changing the pitch or speed of the input, or to add random noise to make predictions more robust against random fluctuations of the input data.

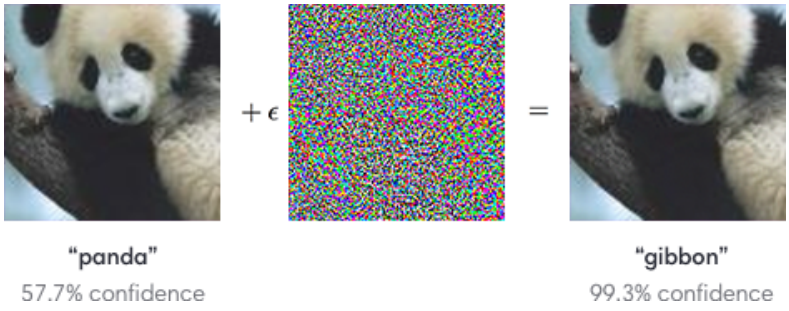


Figure 10: Adversarial example constructed on the image of a panda. Reference: OpenAI (2017)

2.7 Adversarial Training

One special form of dataset augmentation is adversarial training. In practice, it may often occur that classifications of input data can radically change if the right kind of noise is added. Those changes are often invisible to the human eye, but have a large effect on the predictions made by the neural network, as illustrated in Figure 10, where the right amount of seemingly random noise causes the computer to misclassify a panda as a gibbon, even though the image has not changed to the human eye. Adversarial examples are usually constructed by examining the gradients of a model. In the example of the panda image, one could test which small changes in the picture would result in the highest increase of probability assigned to the classification "panda" and then alter the image accordingly. Training the neural network explicitly on such examples makes it less prone to such mistakes or even hostile attacks. Training with adversarial examples is therefore most important where the risk of misclassification of objects can pose direct health or security concerns, for example in the domain of self-driving cars or in the domain of image classification for medical purposes.

3 Conclusion

Over the course of the last years, machine learning and especially deep learning has gained increasing attention and has played a dominant role in major technological and research advancements. While the building blocks of neural networks are rather simple, put together they can be very powerful. With all neural networks having to balance over- and underfitting, complexity and the power of simplicity, regularization is probably the most important tool to achieve that balance. Regularization allows handling the sheer infinite complexity that tasks from image classification to autonomous driving and speech recognition demand, while at the same time making sure that predictions are robust and focus on the most essential features. This essay has laid out the principle ideas behind basic neural networks. It has given an overview of some of the regularization techniques employed to optimize the generalizability of predictions made by the network. While there are many other important

regularization techniques like dropout or bagging, the overview covered in this essay was chosen to give the reader a good intuition about the way neural networks can be designed.

References

Bergstra, J. & Bengio, Y. 2012, 25

Bhande, A. 2018, What is underfitting and overfitting in machine learning and how to deal with it.

Chollet, F. & Allaire, J. J. 2018, Deep Learning with R, 1st edn. (Greenwich, CT, USA: Manning Publications Co.)

OpenAI, Ian Goodfellow, N. P. S. H. R. D. P. A. J. C. 2017, Attacking Machine Learning with Adversarial Examples

Sebastian Raschka. 2019, Regularization of Generalized Linear Models - mlxtend

Recurrent Neural Networks

An Introduction to Binary Classification using Natural Language Processing

F. Süttmann

Georg-August-Universität Göttingen, Germany

Abstract. This essay outlines the applicability of Recurrent Neural Networks for supervised binary classification tasks in Natural Language Processing. It gives a general overview of the relevant theory and different recurrent layers. The theory is then applied to classify twitter posts in German as offensive or non-offensive. Different model structures are compared to discover dependencies on batch size and the number of iterations for generalization.

1 Introduction

Recurrent Neural Networks (RNN) are a special Deep Learning architecture optimized for working with sequences. Their name has its origin in a paper by Rumelhart et al. (1986) and has developed much further since then. This essay aims at outlining the general theory behind RNN and their most prominent extensions. The focus will be on binary classification tasks in Natural Language Processing because sentences can be viewed as sequences. Information about the meaning of a sentence can be found in multiple words or even span over multiple sentences. The models need to be able to make a connection of information over a long sequence of words with variable length to evaluate the sentiment of the text. (Goodfellow et al. 2016). We will outline why RNN are specially adapt to that and where they perform less. For simplicity, matters of unsupervised learning are left unattended. The last chapter compares different recurrent layers on a corpus of German Twitter posts that are either offensive or not. Along with that, different batch sizes and numbers of iterations are compared on four different recurrent layers.

2 Theoretical Foundations

Recurrent neural networks are designed to handle long sequences of data of variable length, which would be difficult for classical feed-forward networks. To process such a sequence of values $x_0, x_1 \ldots, x_t$ with $t = 0, 1, \ldots, \tau$, parameters are shared over the whole sequence and updated all at once. This chapter first gives an introduction to simple Recurrent Neural Networks. In Section 2.2 problems during the optimization of the model, that led to the creation of different, more complex recurrent layers, are

described. The most prominent two, Long Short Term Memory networks and Gated Recurrent Units, are outlined in section 2.3 and 2.4. Section 2.5 gives an overview of different other architectures and how RNN can be deep.

2.1 Recurrent Neural Network

A simple Recurrent Neural Network has a specific, recurrent layer that evaluates a sequence. The sequence of values $x_0, x_1 \ldots, x_t$ with $t = 0, 1, \ldots, \tau$ is first broken down into single pieces. These parts of the sequence are then used as input for the recurrent layer, see Figure 1. The special characteristic of a recurrent layer is that it feeds its output back into itself and is able to generalize to sequence length. Figure 1 represents two common representations of RNN, a rolled computational graph on the left and an unrolled one on the right. In both illustrations output h_t of the hidden unit A is, together with the next input from the sequence x_{t+1}, fed back into the same cell.

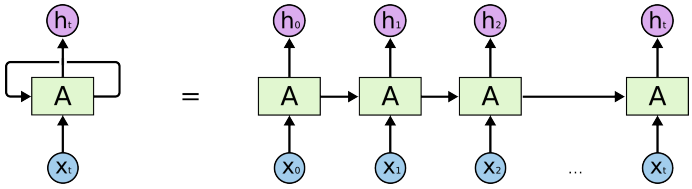


Figure 1: Rolled and unrolled Recurrent Neural Network (Olah 2015)

A is a simple RNN cell which commonly contains either a logistic sigmoid activation function (sigmoid)

$$sigmoid(z) = \sigma(z) = \frac{1}{1 + e^{-z}}. \tag{19}$$

or hyperbolic tangent activation function (tanh)

$$tanh(z) = \frac{sinh(z)}{cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{20}$$

Assuming that we use the tanh function as activation function, h_t the output of the hidden recurrent layer is generated by

$$h_t = tanh(W^{(hh)} h_{t-1} + W^{(hx)} x_t) \tag{21}$$

with $W^{(hh)}$ being the weight matrix between this and the previous hidden state h_{t-1} and $W^{(hx)}$ are the weights between the input and the hidden state. The tanh function is a non-linear function applied element wise to the product

$$\left[\begin{array}{c|c} W^{(hh)} & W^{(hx)} \end{array} \right] \begin{bmatrix} h \\ x \end{bmatrix}.$$

The block matrix has dimensions $D^{(h)} \times (d + D^{(h)})$ and the vector $(d + D^{(h)}) \times 1$, with $x \in \mathbb{R}^d$ and $h \in \mathbb{R}^{D^{(h)}}$ (Manning & Socher 2017). The initialization vector h_0

is commonly just a vector of all zeros and x_t is a row vector of a large matrix E , defining for example a word (see Section 3.2).

Assuming no further hidden layers, the output of our recurrent layer can be evaluated directly. This can either be done at every output of the hidden layer, for example if we want to predict the next word in a sequence, or only at the end, given we want a binary or categorical classification. For that we can choose different activation functions. We want do do a binary classification and therefore use the sigmoid activation function. The prediction is then done by

$$\hat{y}_t = sigmoid(W^{(s)} h_t), \tag{22}$$

with $\hat{y} \in \mathbb{R}^{|V|}$ and $W^{(s)} \in \mathbb{R}^{|V| \times D^{(h)}}$. $|V|$ represents the number of possible values y_t can take, for example the whole vocabulary or in the binary case, two (Manning & Socher 2017). Equations 21 and 22 are commonly referred to as forward propagation equations (Goodfellow et al. 2016).

2.2 Model Optimization

To optimize the RNN a loss function L_t has to be specified. The loss function determines the divergence of the prediction from the true parameter values. Depending on the task different loss functions have to be used. This is often some form of maximum likelihood criterion, where the cost function is described as the cross-entropy between the training data and the model distribution (Goodfellow et al. 2016). If we assume binary classification the cross entropy loss can be specified as

$$L = -(y \log(p) + (1 - y) \log(1 - p)) \tag{23}$$

with p being the softmax probability for either class and $y \in 0, 1$ our binary target variable. For evaluating the loss at each concatenation of the RNN a negative log-likelihood of the from

$$L(x_t, y_t) = -\sum_t \log p_{model} \left(y_t \mid \{x_0, \ldots, x_t\} \right), \tag{24}$$

where y_t is the entry from the model output vector $\hat{y}_t$, can be defined (Goodfellow et al. 2016). Goodfellow et al. (2016) note that computing the gradient from this loss function can be computationally expensive for RNN as it involves first a long forward pass through the sequence, followed by a backwards pass of the same length. They also mention that the run time cannot be reduced by parallelization as forward propagation is inherently sequential and all states have to be saved resulting in high memory costs.

A RNN, like most other Neural Networks (NN), is trained by computing the gradient and iteratively adjusting the weights. In case of a RNN this process is called back-propagation through time as we have go back along the concatenation to the first instance $W^{(hh)}$ or $W^{(hx)}$ was used.

As a simplified example we can look at determining the gradient of the $W^{(hh)}$ weight matrix at time t. The derivative of the loss function at time t, with respect to the weight matrix of the hidden layer, is determined by applying the chain rule

$$\frac{\partial L_t}{\partial W^{(hh)}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W^{(hh)}}, \tag{25}$$

where

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{26}$$

is the product of jacobian matrices and total derivative

$$\frac{\partial h_k}{\partial W^{(hh)}} \tag{27}$$

(Pascanu et al. 2013).

This product of partial derivatives (Equation 25) can lead to problems with the gradients resulting in the optimization either failing completely or losing the ability memorize past information. Both issues were first formally described by Bengio et al. (1994). We separate two sub-problems. Firstly, by multiplying many large values, gradients can get so large that the weights are updated to NaN such that they cannot be updated anymore. This issue is called exploding gradients and Pascanu et al. (2013) recommend solving it by clipping the gradients if they get too large.

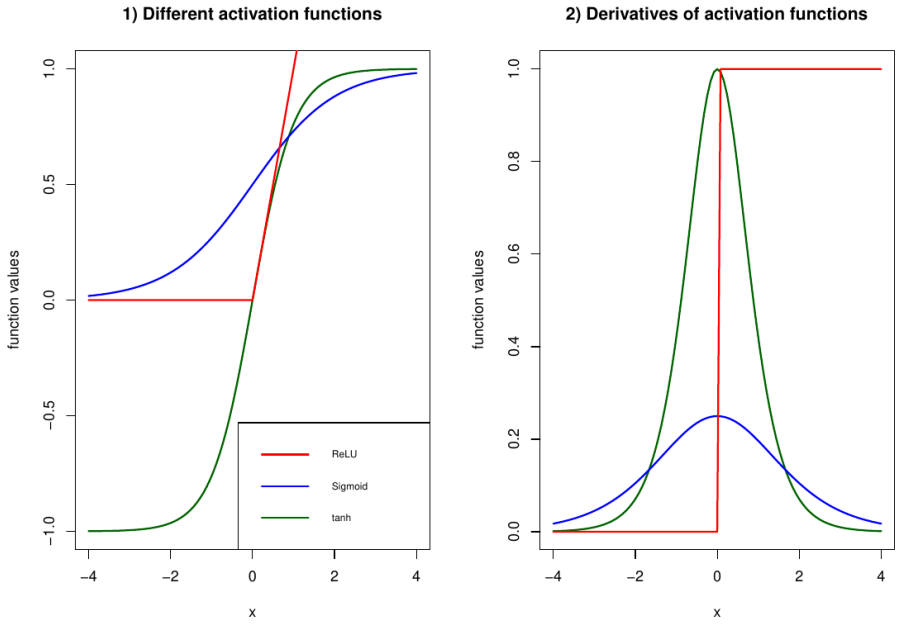


Figure 2: Different activation functions and their derivatives (self)

The larger problem is vanishing gradients. They are the result of taking the product of many small partial derivatives (Hochreiter 1991), resulting in very low values for the gradient. As a consequence networks lose their ability to memorize information from earlier parts of the sequence. The cause of this phenomenon can have multiple reasons. Most commonly the derivatives have values close to zero. If the chosen activation function in the recurrent layer is for example a sigmoid function, with a derivative that is bound upwards by 0.25 (Figure 2), products of derivatives can get small fast. As such it is better to use other activation functions like tanh instead. Figure 2 gives an illustration of different activation functions and their derivatives. The advantage of tanh is its derivative is bound upwards by one. Another cause of vanishing gradients can be the initialization of the weight matrices close to zero (Pascanu et al. 2013), this can be solved by specifying initialization values that are not close to zero. The most severe problem is that the sequence length evaluated by the network can make the problem more severe. The shorter the sequence, the less of a problem it poses. To solve this within a simple RNN is difficult, especially if the network is supposed to remember information far in the past. Other network structures building on top of the idea of RNN are therefore introduced in Section 2.3 and 2.4, that uplift this restriction and are better at retaining information far in the past. Vanishing and exploding gradients are not unique to RNN, but because of the special structure and their recurrence, they are most prominent in networks of this kind.

2.3 Long Short Term Memory

A common extension of RNN is Long Short Term Memory (LSTM) networks which are specialized on connecting information over long sequences. First described by Hochreiter & Schmidhuber (1997) to create a network that is designed to learn long-term dependencies, it introduces a much more complex structure withing the hidden recurrent cells than just the one activation function from before. An illustration of a LSTM in its unrolled state can be seen in Figure 3.

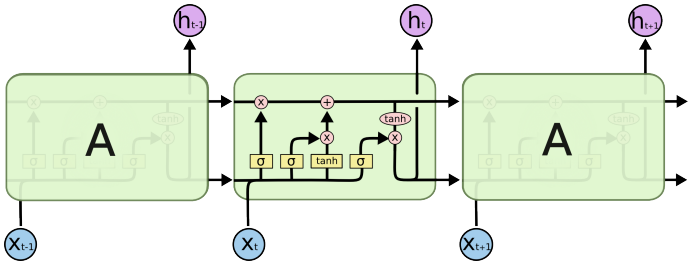


Figure 3: Unrolled Long Short Term Memory network (Olah 2015)

The general functionality of a recurrent neural network stays the same and the sequence is still evaluated step by step while taking the previous inputs into account. Newly added is the cell state C_t which is represented by the upper horizontal line, running from left to right. It acts as the memory of the network where information

can be stored and removed.

The first upwards arrow to the cell state on the left is called "forget get gate layer" and consists of a single sigmoid activation function

$$f_t = \sigma(W^{(f)} \cdot [h_{t-1}, x_t] + b_f) \tag{28}$$

which outputs a matrix of values between zero and one (Hochreiter & Schmidhuber 1997). By multiplying this matrix with the cell state values close to zero indicate that specific information in the cell state should be forgotten and vice versa.

The two arrows in the middle, that are multiplied and then added to the cell state are called "input gate layer". They form a unit to update the cell state with new information. The update to the cell state is determined by applying a *tanh* activation function, creating a proposed new cell state $\tilde{C}_t$ and multiplying it with the output of a sigmoid activation function. In this case the sigmoid layer acts as a type of update function, where values close to one indicate storing new information. The "input gate layer" cannot delete information from the cell state because it is added to the old cell state only after that. The input gate can be expressed as

$$\begin{aligned} i_t &= \sigma(W^{(i)} \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W^{(c)} \cdot [h_{t-1}, x_t] + b_c). \end{aligned} \tag{29}$$

To generate output h_t a *tanh* activation function is applied to the updated cell state at time t and multiplied with the output of a sigmoid function

$$\begin{aligned} C_t &= f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \\ o_t &= \sigma(W^{(o)} \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= o_t \cdot tanh(C_t). \end{aligned} \tag{30}$$

The matrix o_t can be considered as a filter and also means that the hidden state h_t does not go directly through the whole network. Only the cell state forms a direct path through the whole sequence of recurrent cells and can be considered as having an identity function as activation. Because of that the LSTM does not suffer from the vanishing gradient problem caused by small derivatives.

The weight matrices $W^{(f)}$, $W^{(i)}$, $W^{(o)}$ and $W^{(C)}$ are now being trained instead of $W^{(hh)}$ and $W^{(hx)}$. This greatly increases the number of parameters a LSTM has to learn. The parameters $b_{(f)}$, $b_{(i)}$, $b_{(o)}$ and $b_{(C)}$ are added bias terms. Jozefowicz et al. (2015) recommend choosing 1 as an initialization for bias $b_{(f)}$ to boost the models learning speed by preventing issues with the gradients. There are also other variations of the classical LSTM that either merge the forget and input gate layers or add peepholes for the hidden state to have a look at the cell state (Gers & Schmidhuber 2000).

2.4 Gated Recurrent Unit

The most common alternative to LSTMs is the Gated Recurrent Unit (GRU) introduced by Cho et al. (2014). It has the advantage of having slightly less parameters to train than a LSTM, while retaining the same memory capacity. Figure 4 represents the structure of a single unrolled GRU cell.

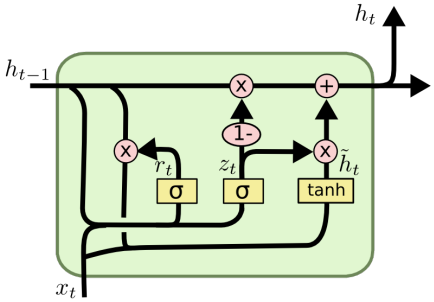


Figure 4: Unrolled Long Short Term Memory network (Olah 2015)

The most notable difference is that it lacks a dedicated cell state like the LSTM. Instead the hidden state is again looped back into the cell. In a GRU the hidden state and the cell state are one and the input and forget gates are also merged. This reduces the number of trainable parameters (Cho et al. 2014). A GRU can be represented by

$$\begin{aligned} z_t &= \sigma(W^{(z)} \cdot [h_{t-1}, x_t]) \\ r_t &= \sigma(W^{(r)} \cdot [h_{t-1}, x_t]) \\ \tilde{h}_t &= tanh(W \cdot [r_t \cdot h_{t-1}, x_t]) \\ h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \end{aligned} \tag{31}$$

A comparison between different LSTM variations can be found in a a paper by Greff et al. (2017). They find that most of these different variations perform about the same. This goes along with Jozefowicz et al. (2015) that resume that there are no different RNN architectures that can consistently outperform LSTM and GRU.

2.5 Recurrent Neural Network Variants

In the field of deep learning, depth is considered as the number of hidden layers in a NN and width is the number of hidden units (nodes) each hidden layer possesses. The RNN variants we have seen so far in Section 2.1 to 2.4 only had a depth of one and a width depending on the maximum sequence length. These are only some possible variants of recurrent neural network layers. They can also be extended to be deep by returning a sequence as input for the next recurrent layer and are often paired with some non recurrent dense layers, before generating output. Recent developments have also often paired recurrent layers with convolutional layers that are

then mostly used for preprocessing the data. For example in augmented attention see Xu et al. (2015).

RNN can not only be one-directional but also bidirectional (Schuster & Paliwal 1997). The advantage of Bidirectional Recurrent Neural Networks (BRNN) is that they can also take future context information into account and do not require a fixed input length. BRNN have shown to perform very well in language-based models (Salehinejad et al. 2017). There are many other variants of RNN, for an overview of advantages and disadvantages of different layers see Salehinejad et al. (2017).

3 Example

To illustrate the previous theory a binary classification task of German Twitter posts regarding offensive and non-offensive language was chosen. The data was collected by Wiegand et al. (2018) and originally designed as a shared task to explore the identification of offensive language under the name "GermEval-2018". It consists of binary classification and a finer differentiation between four categories. Here only the binary classification of offensive language will be attempted. Wiegand et al. (2018) define offensive language as "hurtful, derogatory or obscene comments made by one person to another person". We will first give an overview of the data and how it had to be processed to match the word embeddings in Section 3.2. At last different models are compared in Section 3.3 and problems with training NN are discovered.

3.1 Data

The raw Twitter data from "GermEval-2018" shared task requires a couple of preprocessing steps to be useful for the evaluation with Deep Learning models. As we attempt a binary classification task we first have to check if both classes appear equally often. From 5009 observations in total 3321 (66%) are declared as "other" and 1688 (34%) as "offensive". If a model is trained on imbalanced data it tends towards guessing the more frequent class without adjusting its weights properly. This phenomenon is called "accuracy paradox" and results in a poor model performance (Sun et al. 2009). To avoid this, different approaches are feasible. Ideally, more data is collected. As that is not possible here one can either under-sample cases from the dominant class or over-sample cases from the underrepresented class (Chawla 2009). Other approaches like model penalization or different metrics can also be feasible. For simplicity and due to the small size of the data set the oversampling approach was chosen. This led to a total of 6369 observations.

Due to the special nature of Twitter data, strings have to be preprocessed (Cieliebak et al. 2017). Hashtags and URLs, along with digits that are larger than nine, are replaced by tokens. Some special characters are removed but otherwise most punctuation will be kept as it often contains information in social media posts. Emojis were not considered in these processing steps and were omitted.

Next, all the words and symbols are assigned a word token in the form of a number. These are then used to generate unique number vectors for each post. The maximum length was 58 words or symbols long.

At last, we split the training data into three parts. For that, we first split of 20% of the data for testing purposes. The remaining 80% are then split into 75% for training and 25% for validation. This concludes the data preparation. The corpus of words and symbols is now in a tokenized form consisting of number vectors with a length of 58. Strings with less than 58 words or symbols are padded, longer ones would be truncated.

3.2 Word Embedding

To improve the performance of language-based Deep Learning models one often uses word embeddings. These embeddings map a vocabulary into a high dimensional vector space, usually 300 or larger, by analyzing a huge corpus of text. Word embeddings are language-specific and can be trained using different criteria (Lavelli et al. 2004). Finding good and in **R** usable German word embeddings is more difficult than for English ones. The word embeddings that were used in the end are from Müller (2019) and are trained using "word2vec" (Mikolov et al. 2013) on a corpus of Wikipedia and news articles from 2013 to 2015.

The advantage of using word embeddings is that they bring in a lot of information about the language they are trained on and how vocabulary interacts. As such, they can offset problems when working with small data sets, where some words might be very infrequent and seldom appear during training. The downside to the specific word embeddings by Müller (2019) is, that they do not contain emoji which can be important to understand irony or emotions in social media posts. The common procedure would be to update the pretrained word embeddings with the information on emoji as done by Cieliebak et al. (2017). This was not done here. After the preprocessing steps of the data, about 87% of the words or punctuation signs in the data corpus match with words or signs in the embeddings.

3.3 The Model

For the model design different types of Recurrent Neural Networks were tested. The general structure was inspired by the design chosen by Corazza et al. (2018), who won the "GermEval-2018" task with the best performing model. Their model takes emoji into account and also splits hashtags with the help of n-grams, both were not done here. Four models are compared. All four consist of a fixed embedding layer followed by one or two recurrent layers. The output of the recurrent layers then moves through two dense layers with rectified linear unit activation functions, one with 500 units and the next with 300 units. The last layer is a single unit with sigmoid activation function. Apart from the recurrent layers, all hidden layers are the same in all four models.

Model_1 has two bidirectional recurrent layers. The first is a LSTM layer returning a sequence which is used as input for a GRU layer, Model_2 is a bidirectional LSTM and Model_3 a bidirectional GRU. Model_4 only has a simple recurrent layer. The models were trained using binary cross entropy as a loss function. The disadvantage of doing so can be illustrated using a short example. If the output layer returns a vector for four different strings of the form $(0.3, 0.2, 0.44, 0.36)$ then the binary cross entropy with a threshold of 0.5 sets this to $(0, 0, 0, 0)$. Assuming that the true target is $(0, 0, 1, 0)$, binary accuracy is 75%. Although this indicates high accuracy it needs to be treated with care. If the target of the NN is to classify all tweets with offensive language correctly, then binary cross entropy is not a good measure for the loss. In that case, a custom metric that measures the accuracy in one respective class is recommended. The "GermEval-2018" shared task measured models' performance with the F1-score, which gives equal weight to both classes. It represents the harmonic mean between precision and recall and its disadvantages are outlined by Hand & Christen (2018). We failed to specify a working custom loss function in **R** and were forced to work with the binary cross entropy. The F1-score is still computed as a custom metric to measure model performance after training.

Each of the four models was trained nine times with different specifications, resulting in 36 model fits. The nine different specifications had batch sizes of 25, 75 and 125, with 6, 12 and 18 training epochs each. The result can be seen in Table 3.1. The table reports the name of the model along with the batch size and number of training epochs to look for trends during training. As performance measures loss, binary accuracy and F1-score on the test data that was split of the training data. The F1-score of the real test data with $n = 3532$ is displayed as "F1 (test)". Finally, the computation time using three CPU processors is reported in the last column. The average computation time was 7.6 min, although very long computation times for Model_4 seem to be the cause of the high mean.

Table 1: Four different models trained with nine different specifications of batch size and number of epochs.

	Model	Batch	Epoch	Loss	Binary Acc.	F1	F1 (test)	Time
1	Model_1	25	6	0.65	0.64	0.55	0.71	3.84
2	Model_2	25	6	0.63	0.63	0.54	0.69	2.20
3	Model_3	25	6	0.65	0.61	0.52	0.67	1.86
4	Model_4	25	6	0.70	0.47	0.00	0.00	32.82
5	Model_1	25	12	0.62	0.64	0.53	0.70	7.48
6	Model_2	25	12	0.64	0.64	0.59	0.73	4.69
7	Model_3	25	12	0.64	0.64	0.53	0.69	3.67
8	Model_4	25	12	0.69	0.53	0.63	0.79	1.11
9	Model_1	25	18	0.63	0.67	0.58	0.73	10.93
10	Model_2	25	18	0.70	0.65	0.53	0.69	6.32
11	Model_3	25	18	0.63	0.67	0.62	0.75	5.24
12	Model_4	25	18	0.69	0.52	0.56	0.72	1.67
13	Model_1	75	6	0.64	0.63	0.54	0.68	2.69
14	Model_2	75	6	0.65	0.62	0.59	0.74	1.64
15	Model_3	75	6	0.66	0.60	0.53	0.66	1.30
16	Model_4	75	6	0.69	0.54	0.54	0.69	32.79
17	Model_1	75	12	0.64	0.65	0.57	0.71	5.03
18	Model_2	75	12	0.65	0.66	0.60	0.75	3.18
19	Model_3	75	12	0.65	0.63	0.52	0.66	2.64
20	Model_4	75	12	0.69	0.52	0.52	0.64	1.03
21	Model_1	75	18	0.64	0.67	0.55	0.69	7.64
22	Model_2	75	18	0.66	0.66	0.60	0.73	4.96
23	Model_3	75	18	0.64	0.66	0.58	0.72	4.40
24	Model_4	75	18	0.69	0.53	0.55	0.68	1.66
25	Model_1	125	6	0.66	0.63	0.59	0.75	3.30
26	Model_2	125	6	0.64	0.63	0.58	0.73	2.21
27	Model_3	125	6	0.65	0.62	0.54	0.68	1.88
28	Model_4	125	6	0.69	0.53	0.63	0.79	52.92
29	Model_1	125	12	0.63	0.65	0.54	0.67	7.56
30	Model_2	125	12	0.63	0.64	0.55	0.70	5.84
31	Model_3	125	12	0.66	0.62	0.51	0.65	4.93
32	Model_4	125	12	0.69	0.51	0.54	0.68	1.85
33	Model_1	125	18	0.63	0.65	0.54	0.68	17.87
34	Model_2	125	18	0.64	0.65	0.56	0.73	11.56
35	Model_3	125	18	0.65	0.65	0.52	0.64	9.77
36	Model_4	125	18	0.68	0.56	0.63	0.79	3.31

If we take a look at performance, by comparing the F1-score, there appears to be no trend regarding the choice of hyperparameters. Values are consistently between 0.51 and 0.63 disregarding the one zero value for Model_4. The models even achieve F1 values as high as 0.79 on the test data.

Table 2: Confusion matrix for Model_4 in row 36.

		True		
		Other	Offensive	Total
Prediction	Other	26	13	39
	Offensive	1176	2317	3493
	Total	1202	2330	3532

The best performing model from Corazza et al. (2018), that won the competition, achieved a F1-score of 0.68. This indicates that we might have some issues that are not displayed by the F1-score. By looking at a confusion matrix for Model_4 in row 36 (Table 3.2) one can see that the simple RNN mostly predicted "offensive". This led to a sensitivity of 0.02 and a specificity of 0.99 resulting in such a high F1-score. The cause of this problem is probably the binary crossentropy loss used for training the model.

Table 3: Confusion matrix for Model_1 in row 12.

		True		
		Other	Offensive	Total
Prediction	Other	598	858	1456
	Offensive	604	1472	2076
	Total	1202	2330	3532

We can look at Model_1 in row 12 of Table 3.1 for another comparison. The confusion matrix (Table 3.3) for the test data that achieved an F1-score of 0.72 has a much better allocation of predictions. As a result, sensitivity is 0.50 and specificity 0.63. These results are to be favored compared to the simple RNN layer as we are trying to predict both classes as accurately as possible.

Similar observations can be made by examining Model_2 and Model_3. Both have much better values for sensitivity and specificity compared to the simple RNN in Model_4.

4 Conclusion

We have shown how Neural Networks can be adapted to evaluate sequences of flexible length and connect information at different time points. The particularity of Recurrent Neural Networks is that they loop the output of each time step back into itself. Because this can increase the chance of vanishing and exploding gradients, solutions were proposed. Clipping large gradients was recommended to prevent exploding gradients and different, advanced RNN architectures like Gated Recurrent Units and Long Short Term Memory networks were proposed to prevent vanishing gradients and boost the memory capacity of RNN.

Chapter 3 introduced the applicability of RNN to Natural Language Processing by applying four different RNN layers to a binary classification task of Twitter posts in German. Additionally, the use and benefit of word embeddings were described. We were able to show that the number of training epochs and the batch size had little effect on the model performance. Further LSTM and GRU layers seemed to be better adapted to the task of binary classification with social media data. Especially the simple RNN tended towards guessing only one class, resulting in a decent score, but low performance. This illustrated how important it is to choose the right loss for training and choosing certain meaningful evaluation metrics. Because of the special nature of NN issues with misclassification can be more difficult to notice than in classical statistics. For further reading on advanced RNN and state of the art developments Xu et al. (2015) can be recommended.

References

Bengio, Y., Simard, P., Frasconi, P., et al. 1994, IEEE transactions on neural networks, 5, 157

Chawla, N. V. 2009, in Data mining and knowledge discovery handbook (Springer), 875–886

Cho, K., Van Merriënboer, B., Gulcehre, C., et al. 2014, arXiv preprint arXiv:1406.1078

Cieliebak, M., Deriu, J. M., Egger, D., & Uzdilli, F. 2017, in 5th International Workshop on Natural Language Processing for Social Media, Boston, MA, USA, December 11, 2017, Association for Computational Linguistics, 45–51

Corazza, M., Menini, S., Arslan, P., et al. 2018, in GermEval 2018 Workshop

Gers, F. A. & Schmidhuber, J. 2000, in Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, Vol. 3, IEEE, 189–194

Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (MIT Press), `http://www.deeplearningbook.org`

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. 2017, IEEE transactions on neural networks and learning systems, 28, 2222

Hand, D. & Christen, P. 2018, Statistics and Computing, 28, 539

Hochreiter, S. 1991, TU Münich

Hochreiter, S. & Schmidhuber, J. 1997, Neural computation, 9, 1735

Jozefowicz, R., Zaremba, W., & Sutskever, I. 2015, in International Conference on Machine Learning, 2342–2350

Lavelli, A., Sebastiani, F., & Zanoli, R. 2004, in Proceedings of the thirteenth ACM international conference on Information and knowledge management, ACM, 615–624

Manning, C. & Socher, R. 2017, CS224n: Natural Language Processing with Deep Learning (Winter 2017), `https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/`, accessed: 2019-03-24

Mikolov, T., Chen, K., Corrado, G., & Dean, J. 2013, arXiv preprint arXiv:1301.3781

Müller, A. 2019, German Word Embeddings, `https://devmount.github.io/GermanWordEmbeddings/`, accessed: 2019-03-05

Olah, C. 2015, Understanding LSTM Networks, `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, accessed: 2019-03-24

Pascanu, R., Mikolov, T., & Bengio, Y. 2013, in International conference on machine learning, 1310–1318

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. 1986, Learning internal representations by error propagation, Tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science

Salehinejad, H., Sankar, S., Barfett, J., Colak, E., & Valaee, S. 2017, arXiv preprint arXiv:1801.01078

Schuster, M. & Paliwal, K. K. 1997, IEEE Transactions on Signal Processing, 45, 2673

Sun, Y., Wong, A. K., & Kamel, M. S. 2009, International Journal of Pattern Recognition and Artificial Intelligence, 23, 687

Wiegand, M., Siegel, M., & Ruppenhofer, J. 2018, in 14th Conference on Natural Language Processing KONVENS 2018

Xu, K., Ba, J., Kiros, R., et al. 2015, in International conference on machine learning, 2048–2057

Sign Language Recognition using Regularized Convolutional Neural Networks

A. Thielmann, Q. Seifert, J. Lichter

Georg-August-Universität Göttingen, Germany

Abstract. For the majority of the population, communication is limited when confronted with a deaf person. Newly developed deep learning algorithms try to solve this communicational problem, as convolutional neural networks allow for exceedingly high accuracies in image classification. We use deep convolutional neural networks with data augmentation and dropout to classify images of the American sign language. The accuracy results for the MNIST American Sign Language data set are promising and accuracies of roughly 97% are achieved.

1 Introduction

The human visual system efficiently and effectively recognizes, localizes and categorizes objects within clustered scenes. Subjectively, the recognition of objects, scenes and images happens instantaneous and nearly flawless (Thorpe et al. 1996). In the past and to a certain extent still today, the human eye outperforms algorithms in object recognition accuracy by far. With enough training images, however, machine learning algorithms are slowly approaching human accuracy (Ciregan et al. 2012). The practical application of such techniques is extremely diverse. One possible field of application for image classification algorithms and the subject of the present paper is the translation of sign language into text. Whereas sign language includes single signs for complete words and incorporates its own grammar, the focus of this paper is the mere recognition and translation of the sign language alphabet.

4 to 11 in 10,000 children suffer from early-onset deafness (Marazita et al. 1993). Confronted with a deaf person, communication is limited for the majority of the population. If one is neither proficient in sign language, nor pen and paper or related things are accessible, communication might be problematic up to the case of nearly being impossible. The two most obvious solutions for this problem would be first: To look up every sign language character on the internet. And second: To open the camera of a smartphone and it decodes the sign language into sentences instantly.

The images used for tackling this problem stem from the different letters of the American sign language alphabet (ASL) and are taken from the MNIST (Modified National Institute of Standards and Technology) Sign Language data set and downloaded from Kaggle. The Sign Language MNIST data set consists of about 35,000 images and comes from greatly extending the small number of 1,704 images (see for

example Figures 1 and 2) by various techniques of image processing. To create more data, an image pipeline was used based on ImageMagick. The modification and expansion strategy was to apply filters ('Mitchell', 'Robidoux', 'Catrom', 'Spline', 'Hermite'), along with 5% random pixelation, +/- 15% brightness/contrast, and finally 3 degrees rotation. Unlike the conventional alphabet, the data set does not contain 26 different letters but only 24 different letters, because the American "J" and the American "Z" require a movement of the hand which cannot be depicted by a non-moving picture. The data has the single image pixels as entries, in the form of pixel one, pixel two, up to pixel 784 which represents a single 28 times 28 pixel image with grayscale values between zero and 255. The data set is already stored separately in a training data set and a test data set. The training data set consists of 27,455 images and the test data set of 7,172 images, which gives us a total of 34,627 images. Importantly, the test and training records were only expanded after the separation. This leaves us with relatively independent test and training data sets, which in themselves contain replications of the previously separated images. Due to the high magnification of the original rather small data set, the proposed division of the data sets should be retained, otherwise the results may be distorted.

Figure 1: Example of originally taken images[1]

[1]Source: Sign Language MNIST, Kaggle, 2020,
https://www.kaggle.com/datamunge/sign-language-mnist/

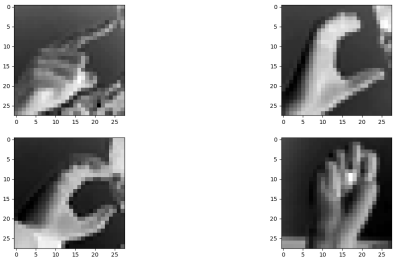


Figure 2: Example of downsampled handsigns shaped into arrays and plotted

In this paper, we present a very common image classification approach using Convolutional Neural Networks. We describe the general functionality of Convolutional Neural Networks, introduce several methods of regularization and derive the proposed model structure. The empirical results are presented to compare different hyperparameters. The promising results validate the proposed model and the chosen regularization methods.

2 Convolutional Neural Networks

Current approaches to object and image classification make extensive use of machine learning methods. The idea to roughly mimic the nature of the human visual cortex with the help of deep hierarchical neural methods are among the most promising architectures for such tasks. Neural network models solve simple recognition tasks with relatively few input data already with remarkable results. For example, the current best error rate on the MNIST digit-recognition task (<0.3%) approaches human performance (Ciregan et al. 2012). The different approaches are various, the best results particularly for image classification, however, are mostly achieved with Convolutional Neural Networks (CNN) (LeCun et al. 1995). The reason for the superiority of CNNs in terms of image classification is twofold.

Firstly, CNN's make use of sparse connectivity (Goodfellow et al. 2016, p.324-330), which enables an efficient processing of high-resolution images. In particular, the efficiency results from how the input data is transferred and analyzed. Depending on the size and the resolution of the images, the input data contains many pixel values that need to be evaluated. In a CNN, "subpictures" or "windows" of the input images are taken and connected via filters (kernels) to single elements of matrices in the following layer. Due to this analysis of "subpictures", only adjacent pixels are connected on the following layer. Therefore, some of the neurons, as opposed to a traditional fully-connected neural network, are not connected, leading to a more efficient, sparsely connected model.

Secondly, the strength of CNNs is their efficiency due to parameter sharing (Goodfellow et al. 2016, p.324-330). A single parameter is used multiple times for transforming the input data, whereas in fully-connected neural networks a single parameter is only used once.

Due to these two features of CNNs, the vulnerability of the model to aspects such as size and position of objects in the images (LeCun et al. 1998) is reduced. In our case, such variations could include the position of the hand, its distance to the camera or the angle in which the hand is held. CNNs are designed in a way that allows the networks to extract specific features from the input images and are therefore to some extend invariant to variations in size and position.

The two features also utilize the particular structures of images. Looking at single batches of the image has still valuable information, as adjacent pixels are more correlated then pixels further apart (Raschka & Mirjalili 2019, 518-520). For these reasons, patterns in the batches contain specific information about the scenes depicted in the images. Usually, CNNs are constructed from *convolutional layers* that are followed by *pooling layers*. Lastly, *fully-connected layers* lead to the final output layer. A detailed description of the functionality of the different components is explained in the following sections.

2.1 Convolutional Layer

The convolutional layers have the purpose of extracting features from the images using kernels. The convolutional layers are constructed from these *convolutional kernels*. The kernels are each applied to the image in small two-dimensional windows defined by the size and width of the kernel. The output of one kernel is a *feature map*, which is a two-dimensional array. As convolutional layers consist of multiple kernels, the output of one convolutional layer are multiple feature maps.

When constructing a convolutional layer, one defines the shape of the kernel, the stride size and the number of filters. The stride size determines how many pixels at a time a filter moves, hence it also determines the size of the output feature map. If one uses zero padding, a stride size of one results in a feature map that has the same size as the input. Larger stride sizes reduce the output size (O'Shea & Nash 2015). The kernels consist of weights that are adjusted during the training of the network so that the network trains itself in order to find the necessary features to extract from the input to successfully classify said input (LeCun et al. 1998).

2.2 Pooling Layer

Convolutional layers are usually followed by *pooling layers*. The main purpose of pooling layers is to shrink the size of the respective feature maps, thus reducing the complexity of the model (O'Shea & Nash 2015) and enabling positional invariance over larger local regions. This not only increases the overall training speed and makes the memory usage more efficient, but also reduces the risk of overfitting. In pooling layers, the feature maps are processed one small field at a time. The elements of one field are pooled using an aggregating function such as *min*, *mean* or *max*. The models we trained for this paper all use the *max* function, as Scherer et al. (2010) found that maxpooling can lead to faster convergence. Depending on the size of the pooling field and the stride size, the feature maps can be shrunk drastically.

In line with Ciresan et al. (2011), the usage of maxpooling is one of the deviations we make in our model from the famous CNN of LeCun et al. (1998).

2.3 Fully-Connected Layer

After extracting the features of interest using the combination of convolutional and pooling layers, the final pooling layer is usually followed by a fully-connected layer (Jarrett et al. 2009; Cireşan et al. 2011), which then leads into the final output layer. Therefore, the two-dimensional feature maps have to be flattened into a one-dimensional vector. The number of neurons required for this fully-connected layer can be easily calculated by multiplying the number of feature maps with the corresponding sizes. For example, a layer of 16 feature maps with a size of 4x4 would lead to a fully-connected layer consisting of 256 neurons.

The last fully-connected layer in the model can also be called the *classification layer*, as it gives the output which in a sense classifies the given input. According to the number of input classes, the classification layer has as many output units as class labels.

3 Overfitting and Regularization

In statistics, overfitting describes the specification of a model that contains too many parameters. Applied to CNNs, overfitting means that a model with many hidden layers learns the complicated relationships of the input data extraordinarily well. So well in fact, that the model corresponds too closely to a particular data set, and may therefore fail to fit additional data or predict future observations reliably. The model learns the random fluctuations and noise in the training data as concepts, which leads to this negative performance impact. In machine learning, overfitting is sometimes referred to as *overtraining*, as we do not *"fit"* a model, but rather *"train"* it. Additionally, an overfitted model suffers from a larger variance due to too many included parameters.

In contrast, an underfitted model has a simpler neural network structure that overgeneralizes the input data. The detailed relations given in the input data are only vaguely learned. An underfitted model has a lower variance compared to an overfitted model, but suffers from a high bias and therefore a large prediction error.

Figure 3 illustrates an example of overfitting and underfitting. In this example, the problem is simplified to 2 variables with 15 observations each. The three colored lines represent different fitted models that try to capture the relation of these variables. The red line represents an underfitted model, as particular variations of the data are simply ignored. The blue line considers the entire variation of the data and connects each individual data point. The aim of a good model, however, is to find a fit that corresponds closely to the green line of figure 3, which represents a smooth curve through the data points. The optimal fitted model does not suffer from a high bias or a high variance and has a high prediction accuracy.

This corresponds to the so-called *bias-variance trade off* (Raschka & Mirjalili 2019, p. 75-78). The variance measures the consistency of the prediction over multiple periods. When the model perfectly fits the training data, minor differences between the test- and the training data can lead to miss-classified predictions, in which case the predictions are not consistent and the model is sensitive to the randomness of the input data. If the test data looks very similar to the training data, the model could

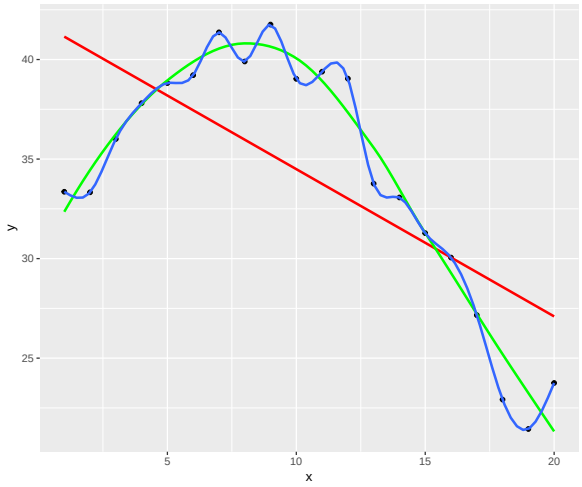


Figure 3: Overfitting vs. Underfitting (Source: Own graph)

still yield a high test accuracy, but fail to predict new and independent observations reliably. An underfitted model may have consistent predictions but most likely these predictions do not coincide with the true data. Generally, a high variance is proportional to overfitting and a high bias is proportional to underfitting (Raschka & Mirjalili 2019, p. 75-78).

One intuitive way to find a good balance between bias and variance is to use a more complex model in the beginning and reduce its complexity by applying regularization methods. Four common and in the present model used regularization methods are:

1. Data augmentation, 2. L1 and L2 regularization, 3. Dropout and 4. Early stopping. Each of these methods, their advantages, drawbacks and usage in our model are discussed in the following sections.

3.1 Data Augmentation

A methodically very simple and intuitive approach to counteract the problem of overfitting is to increase the size of the training data set. Although easy in theory, to simply generate a bigger data set is not only often time- and money consuming, but sometimes simply impossible. In our case, we could of course simply take new pictures of our own hands or search for pictures on the internet. But even if this is theoretically possible, it is too time-consuming and not a general solution for this common problem. An easy to implement and at least for image classification mostly available method is data augmentation. Data augmentation encompasses a suite of techniques that enhance the size of training data sets such that better Deep Learning models can be built using them (Shorten & Khoshgoftaar 2019). The intent is to expand the training data set with new, plausible examples, as it is common knowledge that the more data a machine learning algorithm has access to, the more effective it can be (Wang & Perez 2017). Those techniques include for example cropping

the image, a change in brightness, a rotation of the image, a zoom, a horizontal or vertical flip or a horizontal or vertical shift. The approach we applied in our model, is to generate augmented data before training the classifier. Out of the traditional transformations, only three are applied, as in the case of pictures of sign language, the method of flipping images for example is omitted because hands turned or flipped too much have a different meaning and therefore cause trouble in classifying. Shifts, small rotations and changes in brightness, however, can be safely applied.

Although easy to implement and effective, further enhancing the data set should be done with caution, since the MNIST Sign Language Data set was already enhanced to about 20 times its original size by various techniques of data augmentation. In order not to make the data set too similar and further reduce the variance, we used only the three previously mentioned data augmentation techniques. In total, we expanded our dataset by 6000 newly generated images. This is a much more conservative approach than in Wang & Perez (2017), where they extend a dataset of size N to the size of $2N$ due to the fact that we are already dealing with a heavily enhanced dataset.

3.2 L1 and L2 Norm

When a model perfectly fits the training data, the problem might be that the model has a complex structure with large weights in each layer. Changing the training or input data, in this case, can finally result in very different outcomes. In order to avoid that the weights adapt too strongly to the given input data, there are certain penalizing methods which are *punishing* large weights. Having small weights or zero weights in the network encourages the model to filter out irrelevant information (Géron 2017, p. 127-131).

The penalty for the weights is added to the *cost* function during the optimization process. The loss of the model will increase as additional weights will be considered. Optimizing over the loss will then decrease the weights of the network again, and thereby reduce the tendencies of the model to perfectly adapt to the training data.

The three most common penalizing methods are L1 and L2 regularization and a combination of both of them (Géron 2017, p. 127-131). L1 is also often referred to as *lasso* and L2 is often referred to as *shrinkage* or *weight decay*. Combining both *lasso* and *weight decay* is called *elastic net* regularization. The general formulas for the two methods and their combination is given in equations 1-3, where the first parts of the equations represent the *mean squared error* as an exemplary cost function.

$$C_{lasso} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} |W_j| \tag{32}$$

$$C_{ridge} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \frac{1}{2}\lambda \sum_{j=0}^{M} W_j^2 \tag{33}$$

$$C_{elastic} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + r\lambda \sum_{j=0}^{M} |W_j| + \frac{1-r}{2}\lambda \sum_{j=0}^{M} W_j^2 \tag{34}$$

For the lasso method, the penalty is the sum of the absolute values of all weights. The property of the lasso method is that during optimization the resulting weights are reduced and kept smaller and therefore the model learns better to filter out irrelevant information more efficiently (Raschka & Mirjalili 2019, p.127-134). The weights are kept very close to or exactly at zero (sparse) (Buduma & Locascio 2017, p. 35).

The weight decay method does not include the absolute values of the weights but the squared values. Therefore, larger weights are penalized more severely in comparison to the lasso method and the resulting weights are reduced to very small values (Buduma & Locascio 2017, p.35).

A combination of both methods is called elastic net (Géron 2017, p.132). As can be seen in formula 3, the lasso and weight decay regularizations are added to the cost function. The weighting of the respective methods can be determined by selecting the mixing ratio r. For $r = 0$, the equation reduces to the ridge equation and for $r = 1$ to the lasso equation. Any values in between include a mixture of both methods.

For all three methods, an additional smoothing parameter is given with λ. For higher values of λ, the penalty term gains increasing importance. This has the effect of decreasing the weights even further, resulting in even simpler model structures. If λ goes to infinity, the penalty term completely dominates the loss function. All weights would become zero and the model could no longer filter any patterns of the input data.

To examine the optimal value for λ using cross validation by changing the parameter λ is the most appealing approach.

3.3 Dropout

Dropout is another regularization method that particularly addresses the weights of the neural network. It is a regularization method that approximates training a large number of neural networks with different architectures in parallel. The resulting model thereby is a combination of many different trained models that are averaged in an efficient way (Srivastava et al. 2014). According to Srivastava et al. (2014), dropout significantly reduces overfitting.

More precisely, dropout has the effect of forcing units within a layer to probabilistically take on more or less responsibility for the given inputs. The proposed and in the final model used dropout is the so called “random” or “binary dropout”, which gives big improvements on many benchmark tasks and set new records for speech and object recognition (Hinton et al. 2012). According to Baldi & Sadowski (2013), feature detectors are deleted during the training with a previously determined probability q. By dropping a unit out, it is temporarily removed from the network, along with all its incoming and outgoing connections (Srivastava et al. 2014). To compensate for the loss of these weights, the remaining weights, which now process all the information, must be adjusted in order to obtain continuously accurate prediction results. The dependencies on particular weights, and with this the dependency on the training data, are thereby reduced and the model complexity is simplified.

There are several different ways to determine the optimal value of q. A very common choice is to simply set $q = 1 - p = 0.5$ (Liu & Deng 2015), where p denotes

the probability of the units to stay in the model and q denotes the counter probability of the units to be dropped out. p can also be chosen arbitrarily by using a validation set, or simply by rerunning the whole model with different values between 0 and 1 and evaluating the model performance.

3.4 Early Stopping

A major issue for our model and machine learning in general is the question of the duration of training. Too little training will result in a badly trained and therefore badly performing model. But training too long will again lead to the problem of overfitting. A simple, effective and widely used approach is the implementation of early stopping. The challenge with this approach is the question of when the perfect amount of training has been done.

The general idea of early stopping is to stop training once the model stops improving. As the model is trained using the training data and the model parameters are optimized regarding the training error, the validation error is supposed to function as an unbiased indicator of the model performance (Prechelt 1998). It is reasonable to assume that continuing training for a very long time would result in a model that fits the training data almost perfectly. The validation loss, however, would probably start rising after some time, as at some point, the model would not learn what features to extract from the image but simply specialize in recognizing the training data.

A very simple method of implementing early stopping would be to monitor the validation error after each training epoch, check whether it has improved or not and stop training once it has not improved as compared to the previous training epoch. This approach has the major disadvantage that it assumes the validation to be very smooth over time. In reality, however, the validation error may decreases again after it started increasing (Prechelt 1998). When using this form of early stopping, we observed models with the same hyperparameters to stop at different times and perform quite differently. In order to receive more reliable results, we implemented an early stopping criterion proposed by Prechelt (1998) based on the generalization loss. The generalization loss is defined as

$$GL(t) = 100 * \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right) \tag{35}$$

where E_{va} denotes the validation loss in epoch t and E_{opt} denotes the lowest loss so far. The generalisation loss measures the increase of the validation loss as compared to the best validation loss in percent. The generalisation loss is calculated after each epoch and training is stopped once $GL(t) > \alpha$, where α is a predefined threshold value. Experimenting with different values for α showed that for our case $\alpha = 5\%$ yields the most reliable results.

4 Network architecture

The architecture of our network closely resembles the *LeNet 5* architecture introduced in LeCun et al. (1998). Our 28x28 input images are fed into a convolutional layer using 16 3x3 filters with a stride size of one. The original 5x5 filters of *LeNet 5* did not perform as well as the 3x3 filters. As we use zero padding, the resulting feature maps are of the same size as our input. We then use a maxpooling layer with receptive fields of the size 2x2 and a stride size of one, followed by a dropout layer. The feature maps are again filtered in a convolutional layer of 32 3x3 filters and maxpooled using the same parameters as in the previous maxpooling layer. Again, the maxpooling layer is followed by a dropout layer. We then flatten our 32 feature maps of size 7x7 and lead them into a fully-connected layer consisting of 256 neurons, which is followed by the last dropout layer and finally leads into our output layer. The output layer consists of 24 neurons as we have 24 possible classes representing the used 24 letters of the sign language alphabet.

Throughout our network, we use the Rectified Linear Unit (ReLU) activation function except for our final output layer, in which we use the *softmax* activation. Thanks to its simplicity and effectiveness, ReLU has become the default activation function used across the deep learning community (Ramachandran et al. 2017). The simplicity of ReLU is easily depicted in its enticingly simple formula, given by $f(x) = max(x, 0)$. As complicated activation functions consistently underperform simpler activation functions (Ramachandran et al. 2017), we follow Wang & Perez (2017) in their usage of the ReLU activation function. Furthermore, ReLU consistently matched nearly all of the alternative activation functions introduced by Ramachandran et al. (2017) and has the highest validation on challenging data sets. The densely connected softmax layer follows the approach of Goodfellow et al. (2013), where they achieved their best results on the MNIST database of handwritten digits with a similar model structure as proposed by us. The softmax layer calculates the specific probabilities for each letter, expressing the certainty for a label prediction (Buduma & Locascio 2017, p.15). For an input vector $\mathbf{x}$, the softmax formula for each letter $i = 1, ..., 24$ is given by:

$$softmax(\mathbf{x})_i = \frac{exp(x_i)}{\sum_j exp(x_j))} \tag{36}$$

Since our last layer is activated with softmax activation and we have a probability distribution as the final output, we use the cross-entropy loss function. The strength of the cross-entropy loss function is that misclassified predictions are strongly penalized, resulting in a fairly fast convergence of the algorithm (Géron 2017, p.366-367).

To optimize the loss of the model, we use the Adam optimizer. The Adam optimizer is generally an efficient optimization method (Géron 2017, p. 293), as it tends to be robust to the choice of hyperparameters (Goodfellow et al. 2016, p.301-302).

Since the weights and biases adjust after the first iteration in the training process, the weights as well as the bias need to be initialized. For the bias, it is exceedingly important that during the first training some units are activated inside the ReLU function, in which case the initial values for the biases should be greater than zero (Goodfellow et al. 2016, p.292-298). For this reason, we set all biases to a slightly

higher value than zero. We used a constant value of 0.01 as initialization for all biases as typically the initial biases are chosen to be constant (Goodfellow et al. 2016, p.292-298). Setting the initialization values of the weights is to some extent more difficult. To avoid that the same initial parameters change identically during the training process, we assigned some random continuous values to the weights (Goodfellow et al. 2016, p.292-298). In Buduma and Locascio (2017, p.59-61) a normal distribution was used to set the initial values for the weights, but in order to have fewer outliers in our initialization, we used a truncated normal distribution. As a standard deviation, we found that a value of 0.01 leads to the best results. Values higher than 0.5 significantly hurts the performance of the model.

5 Results

To optimize the choice of hyperparameters, we followed the widely spread approach of grid searching. We did an exhaustive search through manually specified sets of the hyperparameter space and evaluated the simulated models via the test accuracy. The considered parameters for the grid search are the dropout rate, lambda, and the mixing ratio in varying combinations.

To better evaluate the performance of the model, we constructed a very simple baseline model for comparison. The structure of the baseline model is briefly defined in the following section, followed by potential distortions due to the split in the data sets. Sequent thereto, the final results are presented.

5.1 Simple Model

The simple model consists of only one convolutional layer with 3 filters. The resulting values are activated with the ReLU function and then run through a maxpooling process with a 2x2 window. The next layer is a fully-connected layer that shapes the output back into a vector with 24 elements, where each element represents a letter of the American sign language alphabet. For the baseline model, we did not implement any regularization method except for a simple implementation of early stopping. If the validation loss of the following epoch is smaller than the validation loss of the previous epoch, the model keeps on training. If on the other hand, the validation loss of the following epoch is bigger or equal to the validation loss of the previous epoch for two consecutive epochs, the training stops and the model is evaluated.

The baseline model performs remarkably well already. The test accuracy is within 78.5% and 83.5%, while the training of the model stops after 15 to 19 epochs. The batch size was fixed to 100 and the initializers for the weights and biases were set to a truncated normal distribution with a standard deviation of 0.01 and a constant bias with a value of 0.01. These parameters are chosen identically to the proposed more sophisticated model defined in section 4.

5.2 Data Split

The results of the baseline model confirm what the initial split of the test and training data set already implies. The variance within the data sets is very small, resulting

from the large image enhancement, whereas the variation between the data sets is larger, as the split was performed before the enhancement. As a result, the validation and training accuracies were significantly greater than the test accuracy and were both already at 100% after only a few epochs (12-15). Although the baseline model is held extremely sparse, the noise of the training data set seems to already affect the model and overfitting might be a problem.

As previously mentioned, the proposed division of the data sets should definitely be retained, since an incorrect allocation significantly affects the results. The same baseline model "achieved" an accuracy of 99.23% on the test data set, when said test data set was taken out of the training data set before running the model. This is a typical case of dramatically overfitting the model and adapting the weights too strongly to the training data set. Considering the relatively small number of original images of 1704, it is exceedingly important to maintain relatively "independent" training- and test data sets.

5.3 Model Evaluation

Although the benchmark of the simple model is already high, a more sophisticated model outperforms the simple model by more than 15 percentage points. To determine the optimal batch size, we followed the approach of Radiuk (2017) and simply tested our model with the suggested batch sizes of 50, 100, 150 and 64, 128, 256. In our case, we found the optimal batch size for the proposed model to be 100.

The model with which we achieved the best results consists of three convolutional layers. The first layer consists of 16 filters with a size of 3x3, followed by a 2x2 maxpooling layer. To battle overfitting, a dropout of 0.3 is already applied after the maxpooling layer. Another dropout with the same rate is applied after the second convolutional layer as well as after the first fully-connected layer. λ was set to be 0 and the results imply that in combination with the chosen dropout rate a regularization via a penalty term in the loss function is not effective (see Table 5). If the dropout is chosen differently, however, for instance 0.5, the best results were achieved with setting λ to 0.001 and a very small mixing ratio. The test accuracy of this hyperparameter specification was at about 96% ($\pm$1%). The final results of the proposed model are depicted in Table 1.

The final model outperforms the baseline model by 17 percentage points and the regularization methods improve the model by about 4 percentage points. Although only sparsely used, data augmentation also leads to a higher accuracy.

Table 1: Test Accuracy results for the MNIST ASL data set

Model	Test accuracy
Baseline (ours)	0.81 ± 0.025
No regularization	0.905 ± 0.01
Only Data Augmentation	0.9584 ± 0.008
Final model	0.9756 ± 0.01

The exactly defined hyperparameters of the dropout rate and λ have a certain margin, as setting the dropout rate to 0.4 and changing the values of λ or even using the *elastic net* method only decreases the accuracy marginally (see Tables 2-5). Due to our limited computational power, all of our defined networks were only run 3-5 times each. The results should therefore be interpreted with caution, as we experienced a slightly noticeable fluctuation in the model performance. While we have chosen a combination of hyperparameters that optimizes our test accuracy, there is some randomness in the results which makes drawing definite conclusions regarding optimal values of hyperparameters difficult. The achieved accuracy for the ASL dataset, although roughly three percentage points away from perfection, is also notable in comparison with classification approaches used in other papers. Bilgin & Mutludoğan (2019) applied the original LeNet and the CapsNet, as well as various techniques of data augmentation, to the dataset and achieved maximum accuracies of 95.08%. Chakraborty et al. (2018) achieved similar results with their trigger detection model. Interestingly, their model had the most difficulties accurately recognizing the letter "T". The same is true for our proposed model, although our model outperforms their proposed technique in nearly every other letter of the American sign language alphabet. The confusion matrix (Figure 6) shows that our proposed model structure achieves accuracies below 90% only for the letters "T" and "Y". Figures 4 and 5 illustrate examples of correctly and incorrectly classified hand signs. The final model architecture with the optimized hyperparameters was also tested on the classical MNIST handwritten digit database. An accuracy of 98% ($\pm$1%) was achieved. The performance is not as good as the performances of the very best algorithms but still can compete with most of the tested algorithms (LeCun et al. 1995). Interestingly, the performance of the proposed model is negatively affected by the in image classification widely used normalization method of dividing every pixel value by the maximum pixel value of 255. The results for the MNIST ASL dataset decrease dramatically by -17% ($\pm$2%) when normalization is applied. As this is not the case for the MNIST handwritten digits dataset, reasons for this drastic decrease in performance are hard to define. The characteristics of the data give no indication for possible explanations, the network architecture however seems to perform well when applied to different data sets.

6 Conclusion

In this paper, we applied various regularization methods to a relatively easy to implement but efficient model. The combination of the introduced regularization techniques improves the model's performance and seems adequate when dealing with small, overly simplistic or strongly enhanced data sets. The most significant improvements could be seen for the applied techniques of data augmentation. Surprisingly, our final model does not include any penalization method. The random dropout, however, seems to impact the model performance positively. To achieve even better results, a look at the Confusion Matrix (Figure 6) could provide interesting approaches. It seems that our model is confusing specific letter combinations, as a "Y" is often predicted as a "W" and a "T" is often mixed up with an "X". To improve

the model's accuracy, it could be helpful to look specifically at the characteristics of these letters or increase their occurrence in the training dataset.

While we fine-tuned the detailed model structure specifically for the present data set, the regularization methods and their combination can easily be translated to other data sets. The model cannot compete with the very best models when it comes to the MNIST digit recognition data set, but it impresses with its simplicity and still yields promising results.

7 Appendix

Table 2: Dropout = 0, average epochs = 20

	Lambda 0		**Lambda** $1e^{-4}$		**Lambda 0.001**		**Lambda 0.01**	
	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss
Ratio 0	0.958	0.002	0.943	0.006	0.951	0.009	0.955	0.006
Ratio $1e^{-10}$	0.951	0.003	0.952	0.003	0.956	0.000	0.940	0.338
Ratio $1e^{-8}$	0.968	0.003	0.945	0.005	0.956	0.002	0.948	0.006
Ratio $1e^{-6}$	0.959	0.000	0.959	0.001	0.949	0.001	0.935	0.019

Table 3: Dropout = 0.5, average epochs = 25

	Lambda 0		**Lambda** $1e^{-4}$		**Lambda 0.001**		**Lambda 0.01**	
	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss
Ratio 0	0.939	0.334	0.959	0.201	0.971	0.209	0.943	0.422
Ratio $1e^{-10}$	0.947	0.318	0.945	0.445	0.967	0.293	0.943	0.408
Ratio $1e^{-8}$	0.952	0.279	0.949	0.289	0.962	0.244	0.948	0.222
Ratio $1e^{-6}$	0.974	0.201	0.950	0.251	0.962	0.229	0.958	0.334

Table 4: Dropout = 0.4, average epochs = 22

	Lambda 0		**Lambda** $1e^{-4}$		**Lambda 0.001**		**Lambda 0.01**	
	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss
Ratio 0	0.976	0.103	0.962	0.117	0.968	0.169	0.959	0.181
Ratio $1e^{-10}$	0.969	0.159	0.960	0.202	0.967	0.129	0.954	0.262
Ratio $1e^{-8}$	0.971	0.083	0.959	0.082	0.969	0.177	0.953	0.181
Ratio $1e^{-6}$	0.963	0.124	0.964	0.153	0.959	0.166	0.968	0.191

Table 5: Dropout = 0.3, average epochs = 20

	Lambda 0		**Lambda** $1e^{-4}$		**Lambda 0.001**		**Lambda 0.01**	
	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss	Test Acc	Val Loss
Ratio 0	0.972	0.082	0.953	0.055	0.966	0.104	0.969	0.167
Ratio $1e^{-10}$	0.971	0.083	0.976	0.111	0.968	0.119	0.959	0.150
Ratio $1e^{-8}$	0.977	0.057	0.970	0.060	0.963	0.077	0.937	0.286
Ratio $1e^{-6}$	0.974	0.072	0.957	0.113	0.963	0.063	0.964	0.175

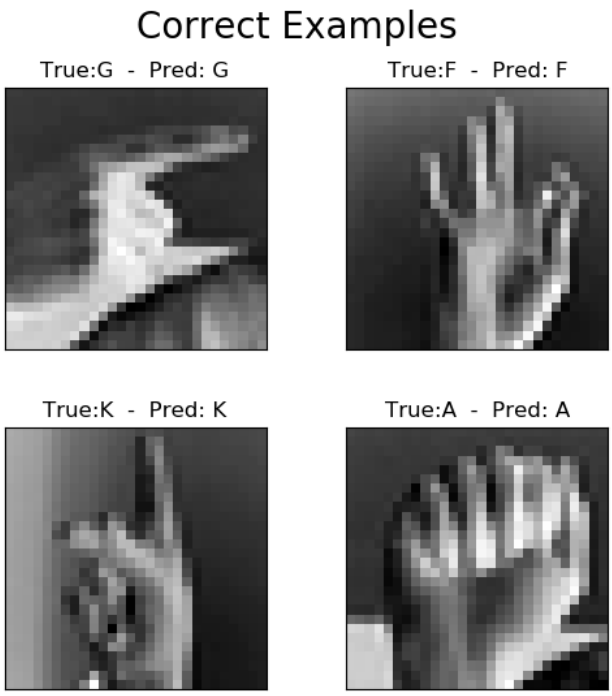


Figure 4: Example of correctly classified hand signs

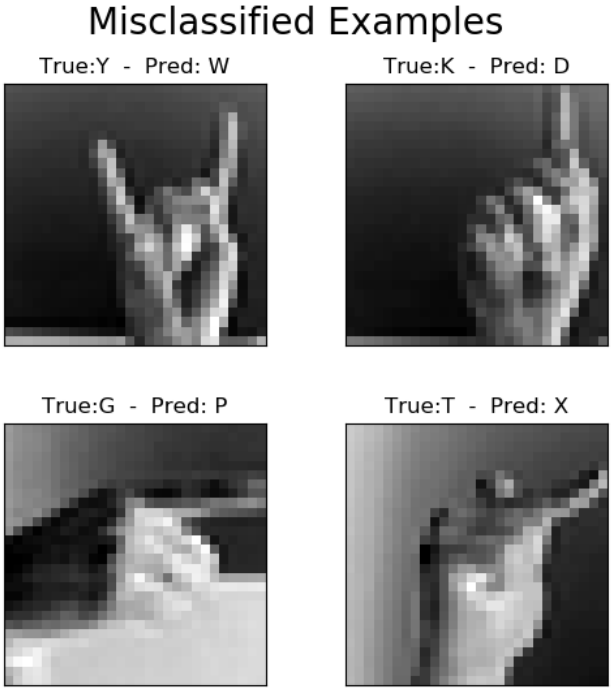


Figure 5: Example of incorrectly classified hand signs

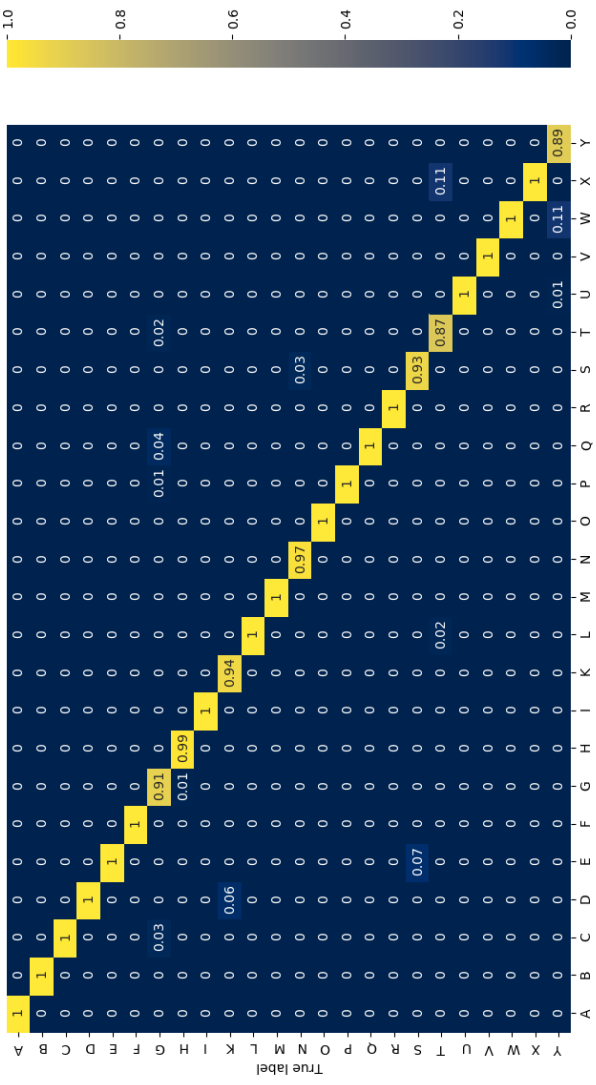


Figure 6: Confusion Matrix

References

Baldi, P. & Sadowski, P. J. 2013, 2814
Bilgin, M. & Mutludoğan, K. 2019, in 2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), IEEE, 1–6
Buduma, N. & Locascio, N. 2017, Fundamentals of deep learning: Designing next-generation machine intelligence algorithms (" O'Reilly Media, Inc.")
Chakraborty, D., Garg, D., Ghosh, A., & Chan, J. H. 2018, in Proceedings of the 10th International Conference on Advances in Information Technology, 1–6
Ciregan, D., Meier, U., & Schmidhuber, J. 2012, in 2012 IEEE conference on computer vision and pattern recognition, IEEE, 3642–3649
Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. 2011, in Twenty-Second International Joint Conference on Artificial Intelligence
Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. 2011, arXiv preprint arXiv:1102.0183
Géron, A. 2017, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems (O'Reilly Media)
Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (MIT Press), `http://www.deeplearningbook.org`
Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. 2013, arXiv preprint arXiv:1302.4389
Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. 2012, arXiv preprint arXiv:1207.0580
Jarrett, K., Kavukcuoglu, K., Ranzato, M., & LeCun, Y. 2009, in 2009 IEEE 12th international conference on computer vision, IEEE, 2146–2153
LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. 1998, Proceedings of the IEEE, 86, 2278
LeCun, Y., Jackel, L., Bottou, L., et al. 1995, in International conference on artificial neural networks, Vol. 60, Perth, Australia, 53–60
Liu, S. & Deng, W. 2015, in 2015 3rd IAPR Asian conference on pattern recognition (ACPR), IEEE, 730–734
Marazita, M. L., Ploughman, L. M., Rawlings, B., et al. 1993, American journal of medical genetics, 46, 486
O'Shea, K. & Nash, R. 2015, CoRR, abs/1511.08458
Prechelt, L. 1998, in Neural Networks: Tricks of the trade (Springer), 55–69
Radiuk, P. M. 2017, Information Technology and Management Science, 20, 20
Ramachandran, P., Zoph, B., & Le, Q. V. 2017, arXiv preprint arXiv:1710.05941
Raschka, S. & Mirjalili, V. 2019, Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2 (Packt Publishing Ltd)
Scherer, D., Müller, A., & Behnke, S. 2010, in International conference on artificial neural networks, Springer, 92–101
Shorten, C. & Khoshgoftaar, T. M. 2019, Journal of Big Data, 6, 60
Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. 2014, Journal of Machine Learning Research, 15, 1929
Thorpe, S., Fize, D., & Marlot, C. 1996, nature, 381, 520
Wang, J. & Perez, L. 2017, Convolutional Neural Networks Vis. Recognit

DeepMRI

Using Deep Convolutional Networks to improve MR Images

T. Ruhkopf and T. Toebrock

Georg-August-Universität Göttingen, Germany

Abstract. MRI data are subject to various sources of disruption. One kind of disruption of particular interest in real application are systematic artifacts that result from undersampling the K-space, with the latter aimed at alleviating the sampling duration. This paper seeks to remove these kind of artifacts based on both image and coil information input, exploiting residual learning strategies with convolutional neural networks and discusses the influence and implied penalties of specialised learning objectives ($\mathcal{L}^{l1}$, $\mathcal{L}^{l2}$, $\mathcal{L}^{SSIM}$, $\mathcal{L}^{MS\text{-}SSIM}$, $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$) for denoising. The proposed architectures are inspired by the Denoising Convolutional Neural Network (Zhang et al. 2016) and the U-Net (Ronneberger et al. 2015). The hyperparameters of the modified architectures are extensively grid searched.

Keywords— Magnetic Resonance Imaging, Denoising, Deep Learning, Convolutional Neural Networks, Residual learning

1 Introduction

Magnetic Resonance Imaging (MRI) is an important imaging procedure and a reliable tool for medical diagnosis in analyzing the structure and function of tissue and organs. The MRI procedure exploits the principles of nuclear spin resonance stating that atoms of a probe placed in a constantly alternating electromagnetic field absorb and emit radio-frequencies. The Fourier approximations of these radio-frequencies yield information about the underlying local composition of those atoms (Gallagher et al. 2008).

According to Flögel (2019), hydrogen is placed in a strong magnetic field precesses around the magnetic field lines. The precession's frequency is determined particularly by the power of the applied strong magnetic field and the gyromagnetic ratio - a material property, both of which form the Larmor frequency. A stimulation of hydrogen with radio-frequencies of the exact same Larmor frequency causes the precession to alter its relative angle into a less favourable energetic state. After stimulation, the precession relaxes to its original state, emitting the applied energy in terms of measurable radio-frequency that are measurable with a detection coil. The relaxation process of atoms in a strong magnetic field is affected by its environment significantly, such that inter alia, the duration time can provide details on the structure of the underlying tissue.

The localization of the signal also heavily exploits the Larmor frequency and the

implied stimulation by a specific radio-frequency, by alternating the magnetic field locally. To do so, additional magnetic coils introduce local gradients in the main magnetic field, altering the local Larmor frequency and thereby local stimulation capabilities. This allows for precise local sampling. As only directions in space can be sampled, the emitted radio-frequencies during the relaxation phase overlap resulting in a joint signal. Observing the relaxation of the signal's components over time yields information on the tissue's structure. To decompose this signal, the Fourier transformation is employed. The measured signal is then transformed into the so called K-space. The K-space image, containing all frequency information can be inverse Fourier transformed to yield an actual image representation. Note that the spatial sampling procedure to fill the entire K-space allows for various strategies.

The cumbersome MR image acquisition suffers various issues, such as its severe sensitivity to alterations of the magnetic field, which causes unintended and shifted stimulus. Furthermore, depending on the applied signal measurement strategy (e.g. the progression of relaxation through time), the acquisition time may be very exhaustive. The latter is very costly and may yield an additional error if the subject moves. To reduce cost and gain precise measurements, inter alia, different strategies of undersampling in the process of filling the K-space are introduced (compare e.g. Boyer et al. (2016)). An undersampled K-space transformed to the image-space yields global artifacts. Since due to convention, the image's detail information is contained in the high-frequency domain placed in the outer bounds and the contrast information contained in the low-frequency domain is placed in the center of a K-space image, the undersampling scheme has strong implications on the resulting artifacts (Gallagher et al. 2008).

To compensate for the implied information loss and to reduce the resulting artifacts, reconstruction methods are applied that inter alia use redundancy in the K-space image or context information. The main contribution of this paper is to employ Convolutional Neural Networks (CNNs), a special kind of Neural Networks (NN) that are capable of estimating highly complex nonlinear correlation structures, for the removal of artifacts in the image-space introduced by radial undersampling. More broadly, this thesis investigates different noise levels in radial undersampling, Poisson-disc undersampling and the use of more complex coil information directly for denoising.

2 Neural Networks

NNs are a supervised learning tool to find correlation structures on the pair (x, y) of hypothetically arbitrary complexity (Hornik 1991). There are various kinds of architectures and 'flavours' of NNs, but following Goodfellow et al. (2016, Chapter 6), NN's pass an input-vector x through stacked layers[2] of linear combinations Wx, which themselves are passed to activation functions $f(Wx)$ such as Rectified Linear Units (ReLUs) $max(0, Wx)$ or Sigmoids $S(Wx) = (1 + exp(-Wx))^{-1}$, to introduce

[2]For notational convenience indices are omitted: in fact the jth layer's activation is $h^{(j)} = f(W^{(j)}h^{(j-1)})$ with $h^{(0)} = x$ and $h^{(J)} = \hat{y}$. In this paragraph, x refers to the appropriate input of the jth layer.

nonlinearity. Depending on the problem at hand, a prediction layer maps the output of the last activation to the target space. Given the target value y, the networks prediction $\hat{y}$ is associated with a loss metric $L(y, \hat{y})$. To improve prediction, the resulting error is back-propagated with gradients $\frac{\partial L}{\partial W}$ obtained from the chain rule, updating the weights in the Ws according to a gradient descent based algorithm. Thereby NNs are capable of learning any nonlinear function in order to reduce the loss function. Note that the theoretical results in Hornik (1991) consider the width of a single layer only, that is how many neurons are contained in a single layer. The depth, that is how many layers are stacked, is another dimension of NNs that requires considerable attention in the form of regularization to avoid training issues. Since W is a dense matrix and the weights in a row i connect all inputs x to an output neuron $(f(Wx))_i$, the term fully connected NN is framed and indicates the richness of connections in a network. As there are potentially exceedingly many weights in the entire network and due to the nonlinear activation functions, the resulting non-convex loss surface can be of arbitrary complexity. This may lead to early convergence in local minima far from an applicable solution in use cases. In order to arrive at the optimal or a reasonable solution and to train the network with large amounts of data in a reasonable time, each of the building blocks requires special attention. Handcrafted architectures and best practices regularize the optimization problem for particular use-cases.

2.1 Convolutional Neural Networks

CNNs are a special kind of NNs, that are particularly well suited - but are not restricted to - deal with image data due to their eponymous convolution operation. CNNs are a favourable choice in image analysis, due to the dimensionality of image-inputs. A conventional NN would need to establish non-trivial connections between all pixels across the entire image, highly sensitive to little variations in their arrangements. Even though NNs are capable to learn these connections, the learning problem can be conceptionally simplified to yield more reliable solutions and faster optimization. Convolutions heavily exploit locality that is inherent to images and the relative composition of local correlation structures. To see this consider the convolution of two functions p and g as illustrated by Goodfellow et al. (2016, Chapter 9.1):

$$(p * g)(t) = \int p(\tau) g(t - \tau) dt \tag{37}$$

which can be conceived as a measure of similarity of those two functions. In the discrete two dimensional case of images,

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n) \tag{38}$$

a two dimensional learnable kernel K is strode over all index positions (i, j) of an image I yielding a 'feature map' S. The weights w contained in K are learnable via back-propagation and form a feature function g such as horizontal or vertical edges, that are searched for across the image. This is also a mathematical intuitive representation of the convolution operation: receive $S(i, j)$ by summing over the

Hadamard product of a weight matrix or feature matrix K with a crop of the image matrix I of equal dimension, at 'support' pixel (i, j). This depiction of the convolution operation places the similarity of the feature detector K with the local crop of I into the forefront. The relation between NNs and CNNs becomes more apparent, if the convolution operation is represented in matrix form Wx, where W is a highly sparse " band diagonal matrix" with shared parameters across rows. An important detail to this perception is the vector representation x of an image i.e. a row-wise stacked version of the underlying array. The result of the convolution y is the stacked version of S correspondingly. This representation makes the simplification during gradient calculation easily apparent. Consider a 4 by 4 pixel image with a 2 by 2 kernel and stride 2 in both horizontal and vertical direction, as Jordan (2018) proposes:

$$\begin{pmatrix} w_{11}\, w_{12} & 0 & 0 & w_{21}\, w_{22} & & & & & & & & \\ & w_{11}\, w_{12} & 0 & 0 & w_{21}\, w_{22} & & & & & & & \\ & & & & & w_{11}\, w_{12} & 0 & 0 & w_{21}\, w_{22} & & & \\ & & & & & & w_{11}\, w_{12} & 0 & 0 & w_{21}\, w_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{16} \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_4 \end{pmatrix}$$

However, the simplicity of this feature detector does not restrict the flexibility of CNNs compared to NNs: multiple learnable kernels can be applied to the same image yielding various feature maps that are aggregated in later convolutional layers. This gives rise to a different interpretation of depth and width in CNNs compared with NNs: depth is the number of stacked convolutional layers. The width is determined by the number of filters applied in each layer. However, there is another dimension: the receptive field, which is strongly associated with the size of the kernels of a layer. Even though early layers analyse local structures only, convolutions placed deeper in a CNN gain a greater receptive field, as each value in a feature map is a dense and compressed representation of parts of the original image. In fact, unless I is padded with a frame of zeros, a 'valid' convolution shrinks the dimension of the image. 'Valid' convolutions also exhibit fewer traversals of K over the boundary regions, which, depending on the application, may be undesirable. Appropriate padding preserves the dimensions and is called 'same' convolution. Note, that just as with NNs, Wx is supplied to an activation function f to introduce nonlinearity.

Noteworthy is also the reverse operation of learnable convolution; transpose convolution, also but less frequently called deconvolution. It follows a similar procedure as convolution, but the compressed feature map value $S(i, j)$ is multiplied with a learnable kernel K and placed into the image-space. During striding over the feature map, which is now input to the operation, overlapping parts of the kernels placed in the image-space I are summed element-wise. Deconvolution has the favourable property, that it up-samples the image from a compressed representation. The notion of transpose convolution becomes apparent from the matrix representation $Wx = y$ of convolutions outlined earlier. Convolution transpose can be formulated as $W'y = x$. Note that the structure of W is the of a convolution and x and y refer to the in- and output of convolutional layers. Without nonlinearity and under necessary constraints upon W, such as $W'W = I$, the original image x can be reconstructed from a feature map y.

2.2 Loss Functions

Loss functions $\mathcal{L}$ are an integral part of the training procedure, as they define the loss landscape and indicate which parts of the prediction require improvement. Therefore, depending on the type of variable, different loss functions may yield better performance and more applicable solutions. Typical candidates and starting point for the development of architectures predicting continuous variables are Mean Absolute Error (MAE):

$$\mathcal{L}^{l1}(p) = \frac{1}{N} \sum \|y(p) - \hat{y}(p)\| \tag{39}$$

and Mean Squared Error (MSE):

$$\mathcal{L}^{l2}(p) = \frac{1}{N} \sum (y(p) - \hat{y}(p))^2 \tag{40}$$

with N, the number of pixels in image patch p. Their widespread application is due to the convenient derivations and their good overall performance in numerous fields. Note, that a $\mathcal{L}^{l1} = 0$ needs no adjustment. In image restoration tasks, the functional form of the used loss function has direct and partly interpretable implications, as Zhao et al. (2017) points out. MSE for instance heavily punishes large deviations, that usually occur around edges, resulting in potentially sharp edges after training. However, it is less sensitive to small changes in 'flat' areas than MAE and may therefor sustain some artifacts in those regions. As Wang et al. (2004) illustrate, neither MAE nor MSE are necessarily sufficient measures to find visually pleasant solutions, which is the main goal of noise reduction in image restoration tasks. The reason is, that various error distributions may yield the same level of MSE or MAE whilst having significant visual effects. The authors propose two alternative measures: (1) Structural Similarity Index (SSIM) and a multiscale version of it, (2) MS-SSIM (Wang et al. 2003). SSIM is a weighted compound measure comprising of three parts, namely (i) luminance $l(x, y)$, (ii) contrast $c(x, y)$ and (iii) structure $s(x, y)$, comparing two images x and y.

$$SSIM(x, y) = [l(x, y)]^{\alpha} \cdot [c(x, y)]^{\beta} \cdot [s(x, y)]^{\gamma} \tag{41}$$

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{42}$$

$$c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{43}$$

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{44}$$

where μ, σ describe the appropriate empirical mean, standard deviation and covariance parameters. C are constants to ensure numerical stability depending on the dynamic range of the image e.g. grayscale images have a range of 255. $\alpha = \beta = \gamma = 1$ are scaling parameters, adjusting the relative importance.
Desirable properties of SSIM and its extensions are

- symmetry in arguments $SSIM(x, y) = SSIM(y, x)$,
- boundedness $SSIM(x, y) \leq 1$,
- a unique maximum at $SSIM(x, y) = 1$ iff $x = y$

An important extension to SSIM as a measure for image quality assessment is also proposed, accounting for spatial locality of the image's parameters. SSIM is applied 'convolutionaly' i.e. on image overlapping patches p of square size 11 with the peculiarity, that the windows associated parameters μ and σ are obtained with a weighing scheme on that patch by a normalized Gaussian kernel, such that the weights $\sum_i w_i = 1$.

$$\mu_x = \sum_{i=1}^{N} w_i x_i \tag{45}$$

$$\sigma_x = \left(\sum_{i=1}^{N} w_i \left(x_i - \mu_x \right)^2 \right)^{\frac{1}{2}} \tag{46}$$

$$\sigma_{xy} = \sum_{i=1}^{N} w_i \left(x_i - \mu_x \right) \left(y_i - \mu_y \right) \tag{47}$$

This introduces a spatially smooth and more or less isotropic loss function, accounting for the local structure at the cost of dependency to a smoothness hyperparameter σ_G and a predefined patch size. Besides, it allows for a local image degradation analysis. The overall scalar quality measure is the mean of $SSIM(x, y)(p)$ over all patches p, which in abuse of notation is termed SSIM. The associated loss of a patch is

$$\mathcal{L}^{SSIM}(p) = 1 - SSIM(p) \tag{48}$$

A detailed description of its proposed extension MS-SSIM in Wang et al. (2003) can be found among others in Zhao et al. (2017). MS-SSIM is taking into account the variable perception of the exact same image at different relative distances and display resolutions. Further, it avoids strong smoothness assumptions. It calculates $s(x, y)$ as well as $c(x, y)$ at M sampled image scales, but $l(x, y)$ only once on the most aggregated sample instead.

$$\text{MS-SSIM}(x, y)(p) = \left[l_M(x, y)(p) \right]^{\alpha_M} \cdot \prod_{j=1}^{M} \left[c_j(x, y)(p) \right]^{\beta_j} \left[s_j(x, y)(p) \right]^{\gamma_j} \tag{49}$$

with the default setting on importance parameters, $\alpha_M = \beta_j = \gamma_j = 1, \forall j \in \{1, \ldots, M\}$. There is still a dependency to σ_G, but its influence is mitigated by the sampling scheme. The authors annotate, that this sampling scheme may become extremely expensive if used to train NNs. Instead of physically sampling M levels, they suggest to use different levels of σ_G for each 'scale-level' M. The associated loss on a patch is

$$\mathcal{L}^{MS\text{-}SSIM}(p) = 1 - MS\text{-}SSIM(p) \tag{50}$$

The authors of Zhao et al. (2017) suggest another and more performant measure, combining $\mathcal{L}^{MS\text{-}SSIM}$ and $\mathcal{L}^{l1}$ to a joint measure. To level their receptive fields, $\mathcal{L}^{l1}$ is calculated with the same Gaussian weighing scheme on an image patch, yielding:

$$\mathcal{L}^{MS\text{-}SSIM-Gl1}(p) = \alpha\mathcal{L}^{MS\text{-}SSIM}(p) + (1-\alpha)G_{\sigma_G^M} * \mathcal{L}^{l1}(p) \tag{51}$$

$G_{\sigma_G^M}$ is a normalized Gaussian kernel with the highest 'perceptive field' σ_G^M and empirically $\alpha = 0.84$.[3] Again, to find an overall loss, the patch loss function needs to be applied convolutionally across the image and averaged over all patches. Further discussion on these measures concerning their derivatives can be found in Zhao et al. (2017, Section III. B-C). They also provide an extensive performance evaluation on all of the losses in super-resolution and JPEG-deblocking tasks, coming to the conclusion of MS-SSIM-GL1 outperforming all the previous losses in training their CNN. All the loss functions are implemented and can be found at Toebrock & Ruhkopf (2019).

2.3 Training NNs & CNNs

The training procedure is at the heart of statistical learning, as it attributes the loss associated with $\hat{y}^{(i)}$ to update all unique weights w in order to improve prediction. According to Goodfellow et al. (2016, Chapter 5.9), the main goal is to minimize the expected overall loss

$$J(W) = E_{x,y\sim\hat{p}_{data}}\mathcal{L}(x,y,W) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(x^{(i)},y^{(i)},W) \tag{52}$$

given the data to improve the network's performance and to generalize. The main issue associated with each updating step

$$w = w + \lambda\nabla_w J(W) \tag{53}$$

is the computationally infeasible size of m. Where w is a single weight vector of W and λ, the learning rate governing the update. Instead, one chooses a stochastic version of gradient descent (SGD) which update the weights iteratively, cycling through at each step randomly shuffled batches of the entire dataset. The resulting average mini-batch gradient is used for updating. The simplest form is SGD performed on a single instance i with the contribution $\nabla_w\mathcal{L}(x^{(i)},y^{(i)},W)$. Consider the one step chain rule along the backward pass through the network to determine the gradients:

$$\frac{\partial\mathcal{L}(x^{(i)},y^{(i)},W)}{\partial w} = \frac{\partial\mathcal{L}(x^{(i)},y^{(i)},W)}{\partial\hat{y}}\frac{\partial f(Wx^{(i)})}{\partial(Wx^{(i)})}\frac{\partial Wx^{(i)}}{\partial w} \tag{54}$$

where e.g. in case of $\mathcal{L}^{l2}$,

$$\frac{\partial\mathcal{L}(x^{(i)},y^{(i)},W)}{\partial w} = -2(y^{(i)}-\hat{y}^{(i)})f'(Wx^{(i)})\frac{\partial Wx^{(i)}}{\partial w} \tag{55}$$

[3] Note, that in contrast to (39) in this notation $\mathcal{L}^{L1}(p)$ actually refers to the loss image of patch p, and not the average loss on the entire patch such that the convolution with G makes sense.

with $\delta = -2(y^{(i)} - \hat{y}^{(i)})f'(Wx^{(i)})$; the back-propagated error of the last layer, that is passed further back through the network to find the gradient of the appropriate w in earlier layers.

The NN representation of CNNs makes it apparent, that CNNs can learn through back-propagation as well, but the gradients exhibit a particular mathematical structure. Compare Kafunah (2019) and Rai (2019). In fact, the gradients can be obtained convolutionally. Consider the last layer's gradients with respect to $w_{[a',b']}$, a specific weight in Kernel K at position $[a', b']$, assuming ReLU activation functions and appropriate padding as simplification. To see the convolutional nature, consider how the prediction of $\hat{y}_{[r,c]}$ results from one particular convolution with kernel K of dimension $k_1 * k_2$:

$$\hat{y}_{[r,c]} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x_{[r+a,c+b]} w_{[a,b]} \tag{56}$$

with r and c being the respective row and column index of the predicted image. The insight hinges upon looking at those particles of the input x, that affect $\hat{y}_{[r,c]}$ through $w_{[a',b']}$:

$$\frac{\partial \hat{y}_{[r,c]}}{\partial w_{[a',b']}} = x_{[r+a',c+b']} \tag{57}$$

And collecting all changes in the loss image attributed to changes in one particular weight, yielding the total change in $\mathcal{L}$:

$$\frac{\partial \mathcal{L}}{\partial w_{[a',b']}} = \sum_{r=0}^{N_1-1} \sum_{c=0}^{N_2-1} \frac{\partial \mathcal{L}}{\partial \hat{y}_{[r,c]}} \frac{\partial \hat{y}_{[r,c]}}{\partial w_{[a',b']}} \tag{58}$$

Summing over the loss image's dimensions $N_1 * N_2$ with respect to the implied indexing of (57) and (58) and comparing with (38) yields one offsetted convolution. The offset is determined by the position of the weight in K. To gain all $k = |K|$ gradients, the loss image is stridden over the input image k times, with appropriate offsets. This highlights the effectiveness of parameter sharing in CNN's gradient estimation.

Irrespective of how the gradients are computed, flavours of SGD are applied to them during the learning process. There are various extensions to SGD, aimed at faster and more robust convergence in the non-convex optimization problem emerging from nonlinearity. Most of which make use of the update or gradient-history to adjust the learning rate(s) and rescale the current gradient for updating. For a detailed discussion of adaptive learning rate algorithms see Goodfellow et al. (2016, Chapter 8.5). A particular instance used later in this paper is ADAM, which makes use of 1st and 2nd order moments of the gradients to gain momentum in the learning process.

2.4 Regularization

The non-convex optimization problem poses various issues regarding training efficiency and the prediction's quality. Aside from the sparsity of connections implied by **convolutions**, the choice of **training algorithm** briefly outlined above or the

use of **ReLU** instead of Sigmoid activation functions, there exist further considerations and strategies incorporated in the network's structure, intended to find simpler representations and ease training. One of them is **batch normalization** (BN). The main issue it tries to alleviate is the covariate shift between layers (Ioffe & Szegedy 2015) which is introduced by higher order interactions between layers that are ignored by gradient descent Goodfellow et al. (2016, Chapter 8.7.1). Any updating step during back-propagation updates the weights of a layer ceteris paribus. However, this may change the distribution of consecutive layers' inputs considerably, such that the learning process is occupied to some extent in finding suitable initial conditions. In the same way, saturating nonlinear activations such as sigmoids are likely to get stuck during training. Both conditions make training inefficient. To avoid this internal covariate shift and thereby reduce sensitivity to the initial distribution of weights, the authors of Ioffe & Szegedy (2015) propose to normalize the layers' inputs by their batch mean μ_B and standard deviation σ_B, before introducing nonlinearity:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sigma_B} \tag{59}$$

According to Goodfellow et al. (2016, Chapter 8.7.1) BN reparametrizes the model such that the output is normalized in first and second order statistics, which is all that a linear network could influence, without preventing the network to learn nonlinear relationships between the data and higher order statistics. Note, as the batch statistics are estimated at every iteration during training, the network never sees the exact same training instance $\hat{x}_i$. In this sense, it fosters robust learning much like the randomness introduced by dropout methods, which according to Ioffe & Szegedy (2015) are rendered obsolete. Maintaining the expressiveness of the network, learnable parameters γ & β are introduced, whose updating scheme can be found in Ioffe & Szegedy (2015, Section 3.1):

$$BN(x_i) = \gamma \hat{x}_i + \beta \tag{60}$$

Following Goodfellow et al. (2016, Chapter 8.7.1), BN and loss penalties are designed to address a similar issue, but BN is far more precise in achieving its purpose, yielding more efficient training.

A frequently applied regularization scheme in CNNs is **max pooling** (Goodfellow et al. 2016, Chapter 9.3), an operation intended to reduce the representation size and introduce translational invariance. Max pooling is in fact only one potential pooling operation, its wide application, however, follows from the desirable invariance property and simplicity. To see this, consider a kernel strided over a feature map whose result is the maximum value of its current receptive field. The input feature map, as described earlier, results from a measure upon the similarity of an image to a feature kernel. The pooled feature map aggregates local information upon the presence or absence of features in the poolings receptive field. The precise location of the detected feature is less important, hence the translational invariance. From a training perspective, the max pooling kernel requires no learning, but the gradients can pass through the maximum value only. As a consequence, max pooling has similarities to dropout in the sense of randomly blocked gradients during back-propagation.

Another regularization scheme is **skip connections**[4] (He et al. 2016) which forwards a layer's activation to another layers input, skipping intermediate layers. The conceptual idea is to ease degradation & exploding gradient issues related to learning identity mappings through multiple nonlinear layers. As a consequence it allows training considerably deeper networks, which may gain predictive performance from their depth. Note that even though the following architectures both employ residual learning strategies, they do so with slight alterations.

3 Network Architectures for Denoising

3.1 Denoising Convolutional Neural Network (DnCNN)

The DnCNN is a state of the art denoising network and has been used in various areas such as image restoration in Zhang et al. (2017), super-resolution problems in Timofte et al. (2018) or for general artifact removal in Galteri et al. (2017). According to Zhang et al. (2016), denoising is a discriminative learning problem, separating noise from the latent image. However, one of the main disadvantages of commonly used specialized methods is their sensitivity to particular noise assumptions such as additive noise $y = x + v$ with x, the original image distorted with some Gaussian noise v of some disruption level σ, yielding the noisy image y. DnCNN is intended to propose a robust framework, unifying efforts on denoising particularly Gaussian- & Gaussian blind denoising, single image superresolution (SiSR) as well as less structured and more general noise patterns such as JPEG image Deblocking. The main difference is the required training set for a particular application.

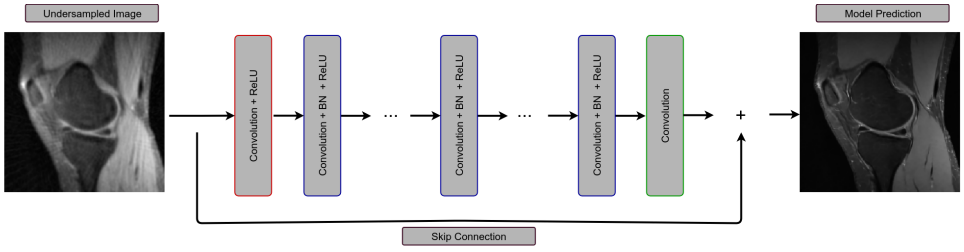


Figure 1: Modified DnCNN architecture, graphic similar to Zhang et al. (2016).

The main contribution of DnCNN in denoising is their use of a residual learning strategy in combination with BN to regularize the learning problem. The general structure of the DnCNN is displayed in figure 1. To do so, the network is intended to reduce the structural part of the image to the, in case of Gaussian noise, more or less uncorrelated error mapping $\hat{v}$. The conceptual idea behind is, that learning the structure of an image is close to an identity mapping and can be learned more easily than a complex prediction on noise and the latent image jointly. In addition, it improves on degradation issues related to deeper networks, enabling the training of even deeper networks due to faster and more stable convergence properties. In

[4]The name skip connection and residual learning can be used interchangeably.

their DnCNN, there is only one implicit skip connection from input to output, which is stated in the residual mapping $\mathcal{R}$ of the used $\mathcal{L}^{l2}$:

$$\mathcal{L}^{l2}(w) = \frac{1}{2N} \sum_{i=1}^{N} (\mathcal{R}(y_i, w) - (y_i - x_i))^2 \tag{61}$$

with $\mathcal{R}(y_i, w) = \hat{v}(w)$ and N the number of training instances. This skip connection is purely additive and hard-wired into the learning strategy.

Similar training instances allow the removal of the structural part more efficiently and robustly, motivating the use of BN as outlined in chapter 2.4. They annotate, that BN actually might profit from the residual learning strategy in denoising tasks, as the layer-wise removal of structural information yields less correlated error predictions. The batch's distribution on these errors might be more Gaussian-like, and less variable per sé. This can alleviate the covariate shift across the layers' distributions even further and add to the BN's resilience.

Their model consists of three parts: (i) the input is fed to a convolutional layer with ReLU activation consisting of 64 filters of size (3*3*c), with c the number of channels. Passed to (ii) stacked layers of convolution with BN and ReLU, each with 64 filters of size (3*3*64) and the last convolutional layer (iii) with c filters of size (3*3*64), combining the networks information on $\hat{v}$. To retain the originals image size for prediction, the same convolutions are applied exclusively. Considering (3*3) filters in all layers and the depth of the network, this has considerable implications on the possible degree of inclusion of context information in form of an increasingly receptive field in the later convolutional layers.

The authors choose (1) stochastic gradient descent with momentum (SGD) and the (2) Adam optimizer (Kingma & Ba 2014) with both yielding comparable performance indicating, that rather the incorporated regularizations are responsible for robust convergence instead of the choice of an optimizer. Regarding the performance of DnCNN, they conclude that these models recover sharp edges and fine details, but also yield visually pleasant results in the smooth region on Gaussian denoising with and without specific disruption level σ.

3.2 DnCNN Modifications

There is a subtle change, that we actually predict $\hat{y}$ not $\hat{v}$, since our defection BART script physically alters y to x immediately through information loss in the K-space instead of generating and adding v separately. The latter approach extensively used in the paper introducing DnCNN to simulate noise, makes an explicit skip connection superfluous as the loss can be formulated on the noise level: $\mathcal{L}(v, \hat{v})$. The former case requires an explicit skip connection of the form $\hat{y} = x + \hat{v}$. However, the training should be unaffected, as the pixel-wise loss remains the same.

3.3 Autoencoders (U-Net)

Since the U-Net was introduced in Ronneberger et al. (2015), it was used in many different image-to-image problems such as image segmentation, object detection as in

Yap et al. (2017) and is often used as the generator network in Generative Adversarial networks as in Isola et al. (2017). Therefore, it may also be a promising candidate in denoising MR images. Following Goodfellow et al. (2016, Chapter 14), Autoencoders are NNs, trained to copy its input through a hidden layer h to its output with the aim of learning distributional features of the training set i.e. a sparser representation containing all relevant features of the general input. To learn a sparser representation instead of an identity mapping, the layers contained within an Autoencoder are deliberately shrunk in dimension, such that they can only approximately copy the input. Such networks are called undercomplete. Generally, they can be divided into an encoder $e(x)$ and a decoder part $d(x)$. The resulting general minimization problem can be framed in terms of:

$$\mathcal{L}(x, d(e(x)) \tag{62}$$

Goodfellow et al. (2016, Chapter 14) even draw the direct analogy of e learning the PCA space of x, if they were restricted to linear combinations only. As Autoencoders are NNs, they generalize to nonlinear PCA. However, with to much capacity, Autoencoders focused on the copying task could easily default to an identity mapping hypothetically. Overcomplete Autoencoders are conceptionally also capable of yielding useful representations if $\mathcal{L}$ is regularized with an associated penalty on e.g. sparsity of parameters. This can enhance the capabilities of the architectures towards e.g. robustness to noise or missing data; broadening the task from mere representation learning. The implementation of the DnCNN architecture can be found at Toebrock & Ruhkopf (2019).

One performant representative of his kind can be found with Ronneberger et al. (2015), originally designed for image segmentation in biomedical image processing. The original architecture operating on image tiles is shown in figure 2. In detail, its task is to get precise locations of image context parts by assigning class labels to each of the pixels. To do so, the commonly used max-pooling operation in CNNs, which improves their resilience in classification tasks are partly replaced with deconvolutions in d in order to decode the gained representation. Further, to retain detailed information in image segmentation, skip connections in the form of concatenated earlier layers are introduced. Note that the U-Nets residual learning strategy is far more sophisticated compared to the of DnCNN with the aim of predicting y directly instead of $\hat{v}$. This is caused by the propagated information in the form of preserved feature maps, upon which the network can select and thereby produce higher resolution output. In order to avoid limited resolution of larger images due to memory constraints, the original paper applies an overlap-tile strategy for seamless segmentation on image patches. This concept summarizes as follows: A pixel's class prediction is based on the context of that pixel i.e. only valid convolutions are applied on image patches. The resulting convolved image of shrunken size is the prediction of the centered rectangle of the original image, as only those pixels have full context information. Patches on edges gain their context information by mirroring extrapolation.

The cell culture context, in which the proposed U-Net has its origin, allows for large scale image augmentation by deformation and therefor sparsity of the training data set as well as invariance to such deformations. Another segmentation related

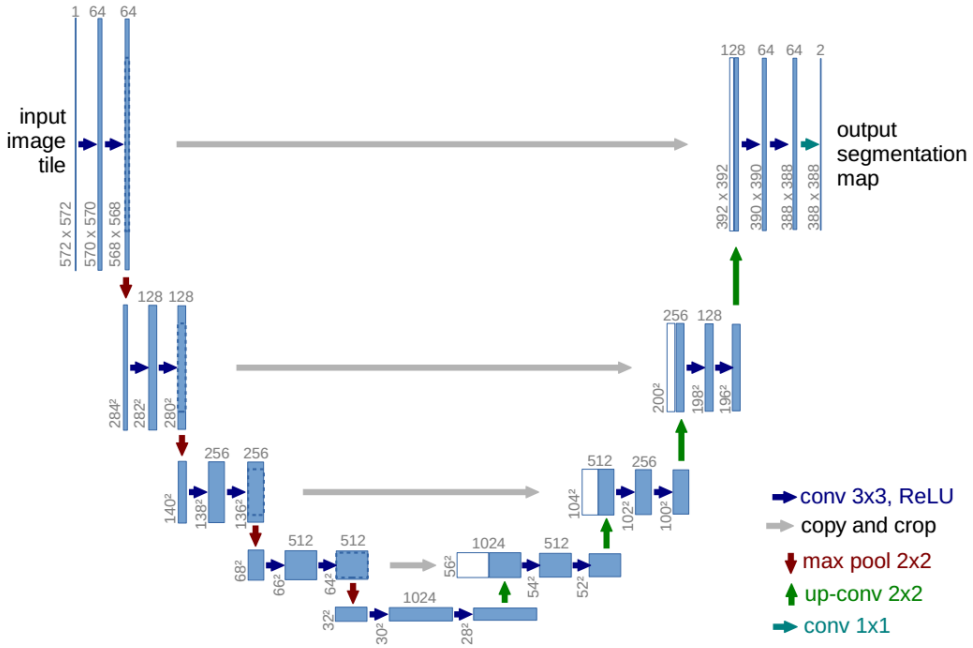


Figure 2: Original U-Net structure, taken from Ronneberger et al. (2015).

contribution is their associated weighted loss function, that is forcing the network to learn the precise borders between adjacent cell objects of the same class by severely punishing miss-classification of background pixels. Due to technical considerations, they choose to neglect BN in favour of high momentum in gradient descent, such that a large number of previously seen tiles promote regularization.

The architecture's encoder consists of stacked 3x3 valid convolutions and ReLUs as a building block, paired with 2x2 max-pooling with stride two to down-sample the image between the stacked convolutions. In each down-sampled layer, the number of filters doubles. The decoder part replaces max-pooling with 2x2 deconvolutions with stride one to up-sample the image. The stacked convolution blocks halve their filters at each up-sampling step and receive the center-tile of the encoder's output at the same level as concatenated feature maps to overcome the differently shaped inputs introduced by different strides. This is in accordance with using the tile's context information, without predicting the class of the pixels in that context-frame. To accommodate the additional information contained in the concatenated feature maps, the first convolution in each stack has twice as many filters as the remaining. At last, c 1x1x64 filters aggregate the information to the desired number of c classes. As its task is classification, the associated handcrafted loss function is a weighted version of a pixel-wise soft-max on the activations combined with a cross entropy penalty for whose peculiarities the inclined reader is referred to the original paper.

3.4 U-Net Modifications

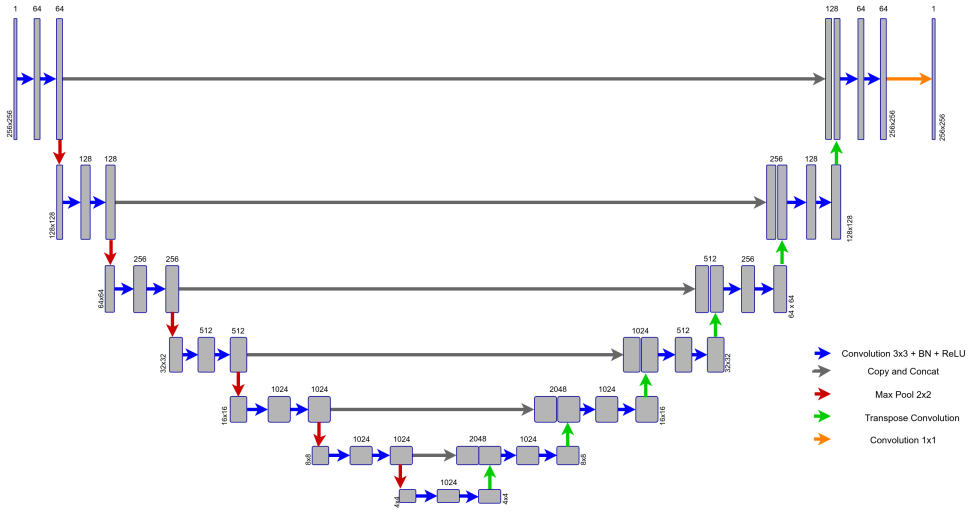


Figure 3: Structure of the Autoencoder in reference to Ronneberger et al. (2015).

The design proposed in this paper is loosely based on the structure of the previous U-net, but combines several ideas from the DnCNN and from the general Autoencoder to arrive at a network structure capable of denoising MR data instead of segmenting images and can be seen in figure 3.

The residual learning strategy for efficient and robust denoising, supposedly helps the network to identify and focus on the disrupted parts of the image in the encoding, whereas detail information is propagated through the network. It also allows for training a network of considerable depth. Particularly, BN in combination with the residual learning strategy is of considerable interest as the authors of Zhang et al. (2016) pointed out and may ease training. The dataset feature of high similarity of global structures between instances of the knees motivates the distributional learning in e on x, potentially enhancing the prediction in areas of greater information loss introduced by the Fourier transform. The encoding is achieved within the lower layers and pooling. Following this line of thought concerning the Fourier transform, the architecture neglects the idea of tiles, but rather trains on the entire image as a whole, since the disruption follows from information loss in the frequency domain affecting the entire image consistently. As a result, the image size is retained and same convolutions are applied. The loss function is altered to those of the chapter 2.2, owing to the different tasks and the continuous scale of y.

Further, as it is no longer a mere copy task, it is also less likely to gain an identity mapping and thus the models capabilities are enhanced such that it is overcomplete, with the ability to learn salient features of the data beyond the knee structures including correlations of the disruption pattern.

Lastly, the depth of the network is adjusted. A deeper NNs is able to approximate

more complicated functions and hence better suited to deal with the global correlation structure of the artifacts. Also, since a deeper Autoencoder does result in a smaller representation of the image at the lowest level, more global information is compressed there and the global correlation structure of the noise can be better accounted for. The U-Net as presented in Ronneberger et al. (2015) has four levels of downsampling and ends up with a 32x32 pixel representation at the lowest level. The network presented in this analysis has six levels of downsampling and ends up with 4x4 pixel representation of the image. The difference in the size of the images stems also from the fact the images used in the U-Net are bigger and have a size of 527x527 instead of 256x256 as in this analysis. Also, in the original structure, with each level of downsampling as the image size is cut in half, the number of filters is doubled. In the first part of our analysis, after the fourth level of downsampling the number of filters remains constant exactly at the maximal number of filters that are used in U-Net with 1048 filters. The implementation of the modified U-Net, the so called Autoencoder, can be found at Toebrock & Ruhkopf (2019).

4 Literature Review

Despite these general architectures that can be used for all-purpose denoising there exist some papers that are concerned with the specialized topic of denoising MR images. There are three potential levels in the creation of MR images at which NNs might be used to improve the image. First, multiple coils in the MR scanner save the detected frequencies in multiple K-space cuboids. Then these multiple K-spaces are combined into a single one and using inverse Fourier transformation the final image is created. The architectures proposed in Eo et al. (2018) and Lee et al. (2017) are examples of NN architectures where the K-space image or the coil information is used within the deep learning model. Other papers such as Wang et al. (2016) use the image-space for their model.

Using K-space images as input to a deep learning model also comes with inherent format issues. First of all, the K-space consists of complex numbers and therefore the network has to be altered to deal with this input format. Although there are different solutions to this problem such as using complex weights, most of the authors resort to separating the real and imaginary parts of the images, treating them as additional channels of the input image. Using this strategy to deal with complex input comes with some disadvantages. Most deep learning models for denoising such as the DnCNN which can be found in Zhang et al. (2016) were built for a gray-scale image with one, or in the case of coloured images, three channels. It is unclear, whether the originals network's capabilities, incorporated in its design concerning depth and width are sufficient to capture the inherent features of this increased input format to a satisfactory degree. When including the additional coil information into the model, these concerns are even more severe since the input format is increased even further. Having a scanner with eight coils this yields a total of eight complex-valued arrays and hence a total of 16 channels. Aggravating, it is unclear whether some information is lost by the separation on the complex numbers.

One particularly interesting method is described in Eo et al. (2018). There, the

authors use a two step approach where they fit alternately two CNN's. The first one working on K-space images and tries to improve this representation. Then, the improved K-space is converted to the image-space and fed to the next network. Both networks use an architecture that mainly consists of stacked layers of convolutions with ReLUs as activation functions. For the second network handling image-space input, a skip connection from the first to the final layer is used which makes the network even more similar to the DnCNN as describe in Zhang et al. (2016). Interestingly, for the network using the K-space domain, they neglect the idea of skip connections.

In contrast, the authors of Lee et al. (2017) use image-space inputs and particularly focus on the benefits of residual learning for denoising MR images. Therefore, they include multiple skip connections into their network. As in this analysis, they compare two types of network architectures. First of all, they use a U-Net architecture as described in Ronneberger et al. (2015) that they call multi-scale residual learning. As a second architecture, they propose a network consisting of 28 layers of stacked convolutions with ReLU activations and BN. In dissociation of DnCNN, as described in Zhang et al. (2016), they included a total of four skip connection so that it resembles the architecture of the U-Net without up- and downsampling in between. Comparing the networks, the authors concluded that the multi-scale residual learning performs best. Nevertheless, we cannot conclude that the U-Net is inherently better suited to denoise MR images than the DnCNN since there are some considerable differences between the DnCNN and their single-scale residual learning architecture.

This analysis compares the performance of the most promising architectures the U-Net and the DnCNN for denoising MR images. Our models predominantly operate on the image-space and chapter 6 will investigate how the models can generalize to other or more intense distortions applied to the MR data. Therefore, we train on data created by multiple distortion mechanisms and evaluate how the models perform on different noise settings and how fast they can adapt to them. This is of particular interest since for a real-world application of these algorithms they have to be able to perform on heterogeneous distortions and have to be able to adapt. At last, as a final step, we will test whether the developed architectures work also with the K-space as input especially when additional coil information is included. Beforehand, the following chapter will cover the data creation in more detail.

5 Data

A dataset that is close to the real application of removing artifacts in MR data of heterogeneous scanners and undersampling procedures could be created by taking both fully and undersampled MR images of different objects in the respective scanners. Unfortunately, such a dataset was not available and we had to resort to artificially undersampling fully sampled data. This process has some disadvantages. First of all, although the undersampling process resembles distortion that could be created in MR images, distortions unassociated with the underlying sampling scheme such as contaminations in the scanner room are not accounted for. The dataset we finally

resorted to the Stanford Fullysampled 3D FSE Knees dataset which is available at Michael Lustig (2018). This dataset contains fully sampled, three-dimensional MR images of 20 knees. Since we wanted to develop an algorithm for two-dimensional images and it increases the number of images drastically, two dimensional slices of the three-dimensional MR images are taken. The slices are created by slicing the hyper rectangles through the x-, y- and z-axis while holding the other axis constant. Thereby around 250 slices per axis are created. The imprecision stems from the fact that out of 320 possible slices for the x and y axis only 255 slices are used while for the z-axis out of 255 slices only 210 are used. This is done as some slices do not contain any part of the object. In figure 4 one example slice for each of the three dimensions of the first knee in the dataset is displayed. As can also be seen, due to the rectangular nature of the 3-dimensional images, two images are of dimensions 320 x 255 and one image is of dimension 255 x 255. Therefore, the images of size 320 x 255 are cut to the matching size 255 x 255, to have homogeneous input for the network. On the rightmost picture of figure 4 only white noise can be seen. This occurs on the outer slices since the knees do not perfectly fit in the rectangle. After some consideration and due to the manual nature of the removal, we decided to keep some of these images in the dataset. As they may appear in real-world MR images the NNs should be able to predict based on them as well.

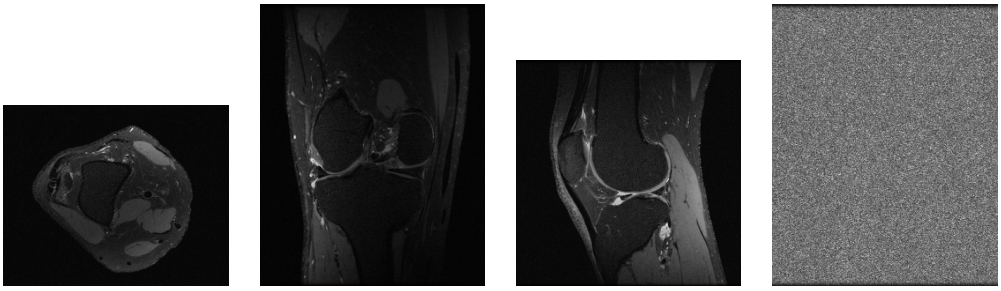
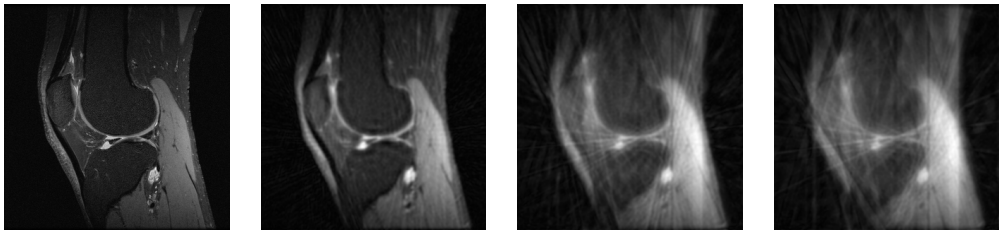


Figure 4: Slices through different dimensions and an empty image at the dimensions' boundaries.

For the creation of the fully sampled images in figure 4, the undersampling manipulations as well as for the creation of the entire datasets the BART Toolbox for Computational Magnetic Resonance Imaging software is employed. The newest software version is available at Uecker et al. (2019). The bash scripts are developed in cooperation with the chair of Diagnostic and Interventional Radiology at the University Medical Center and can be found at Toebrock & Ruhkopf (2019). From the raw, three-dimensional knee data two types of input data for the model are created. First, gray-scale images are produced from K-space information by the BART software. Using the image-space, general network architectures for denoising can be applied and it is certain that they are able to deal with the input data structure. Therefore, the models are mainly build based on the image-space, although it contains less information than the original K-space. Later on, the models are also trained with K-space input. Based on the image-space, three datasets with different

forms and intensities of noise are created since one of the main ideas of this project is to test the performance of the networks on previously unseen artifacts of different strength or structure. Additionally, it is tested how fast the networks can adapt to new noise structures based on additional training on the new data. In a real-world application of the algorithms, one has to deal with different kinds of noise patterns due to variations between scanners and therefore, the algorithm has to be able to generalize to different forms and intensities of artifacts.

For the first two datasets, radial undersampling is used. In order to create different strengths of noise, the parameters of the undersampling process, namely the spokes are varied. In figure 5 radial artifacts of different strengths can be seen. The left picture does correspond to the fully sampled image while the other images from left to right correspond to lower values of the spokes parameter, hence higher intensity of distortion. For the two datasets rad_41 and rad_15, the parameters leading to the two images in the middle are used.



fully sampled spikes = 41 spikes = 15 spike = 11

Figure 5: Left: Original picture, different value for spikes in radial undersampling.

The pois dataset is created based on the Poisson disk undersampling pattern. This dataset is created to simulate a change in the types of artifacts and only one picture per slice is created. To see the implications of this sampling scheme to the resulting noise pattern compare figure 6 from left to right, where the fully sampled image, the distorted image based on radial undersampling and the distorted images based on Poisson disk undersampling are displayed.

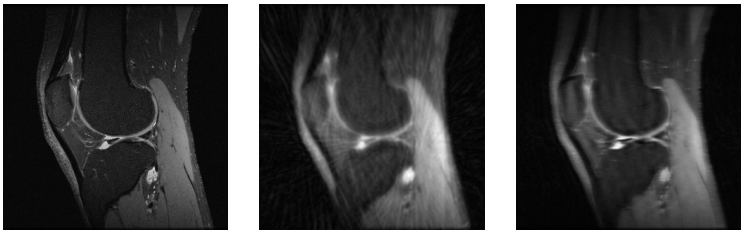


Figure 6: From Left to right: Original Picture, radial undersampling, Poisson disk undersampling.

To create the output of the BART scripts, it is iterated over the 20, three-dimensional input images and for all three dimensions slices are created. Based on these fully

sampled slices, undersampled ones are created. In the case of radial undersampling five manipulated versions of each slice are generated for the rad_41 dataset and three slices are generated for the rad_15 dataset using rotated spikes to create different artefacts on the same fully sampled image. For the pois dataset on the other hand, only one undersampled image is created. This is done to reduce the total number of repetitions of the fully sampled images when later on all datasets are combined to a joint dataset.

In the case of radial undersampling, multiple images are used since the algorithm is mainly trained on this dataset. The additional creation of training samples based on simulations of rotated artifacts is a form of data augmentation. Thereby the limited number of images is increased. In this order, the rad_41, rad_15 and pois dataset consist of roughly 70000, 40000 and 14000 images of a total size of 30, 20 and 7 gigabytes respectively.

For the coil dataset containing the K-spaces inputs, each input sample consists of 8 layers with complex numbers that are split into 16 layers, separating the real and imaginary parts of the image. Only every third slice could be used since otherwise the size of the dataset would have been around 130 gigabytes. Due to computational reasons, only the Poisson disk undersampling method is used to distort the inputs. The coil dataset contains 5000 samples and has a size of roughly 40 gigabytes.

Lastly, each dataset is separated into train and test sets, ensuring that the exact same instances that relate to one fully sampled image are contained either in the training or testing set for all datasets. This is more complicated since depending on the dataset from each slice, multiple, a single or no undersampled slices were generated. By doing so it is ensured that when evaluating on the test dataset, non of the slices was previously seen by any model.

Nevertheless, there are two flaws in the generated data. Firstly, the dataset is highly specific since it only contains images of knees. Here, a dataset containing more heterogeneous objects would be advantageous and ease the generalization of the network to denoise objects of different structures. Additionally, due to the slicing of the three-dimensional input data of a single patient's knee, adjacent images are highly correlated. Both of these problems could lead to overfitting of the NN and worse performance on different, more heterogeneous MR datasets.

6 Model evaluation

The proposed architectures are tested in a grid of various specifications. We focus our analysis mainly on models based on the image-space but provide first insights gained from training the DnCNN and Autoencoder on coil input with Poisson disc undersampling and $\mathcal{L}^{l2}$. Further, at test time, we restricted our analysis to the losses $\mathcal{L}^{l1}$, $\mathcal{L}^{l2}$ and $\mathcal{L}^{SSIM}$, as it was unclear, if the implemented $\mathcal{L}^{MS\text{-}SSIM}$ and $\mathcal{L}^{MS\text{-}SSIM-Gl1}$ would learn properly. As consequence $\mathcal{L}^{MS\text{-}SSIM}$ and $\mathcal{L}^{MS\text{-}SSIM-Gl1}$ are excluded from the first part of our analysis. Recent efforts and the reasonable results of the $\mathcal{L}^{SSIM}$ test cases suggest that the training based on the loss functions $\mathcal{L}^{MS\text{-}SSIM}$ and $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$ in particular may yield visually more pleasant results as suggested in Zhao et al. (2017). To evaluate the performance of the models we

choose to present the mean of MAE (M-MAE), MSE (M-MSE) and SSIM (M-SSIM) on the entire test dataset. Even though the models are trained with these metrics as losses, the joined analysis of these metrics may yield valuable insights. The main idea, also presented in chapter 2.2 and laid out in detail in Zhao et al. (2017), is that the different measures focus on specific features such as edges or flat surfaces. Particularly, even with worse predictions based on the M-MAE and M-MSE metrics, the visual performance evaluation implied by an increased M-SSIM may have stronger indications for the overall image quality in the predictions. Especially, since SSIM is designated to mimic human perception to some extent. Note that the favourable properties of SSIM as measure outlined in chapter 2.2, still only allow for ordinal rating of improvement and deterioration of the prediction's accuracy.

6.1 Model Performance on Image-Space Input

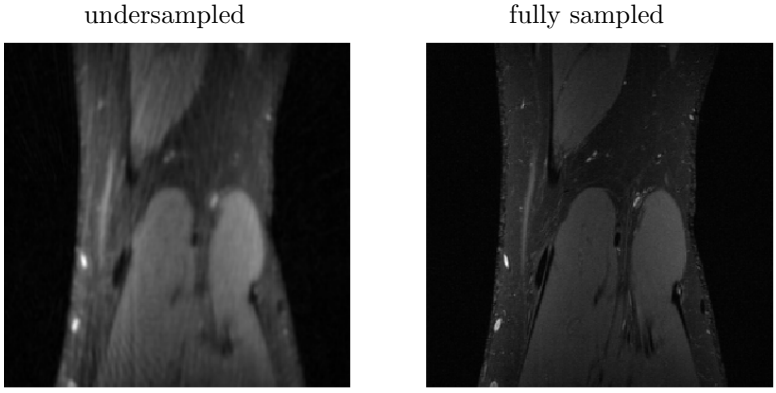


Figure 7: Under- & fullysampled image of the images in figure 8.

In this section we analyze the performance of the DnCNN and the Autoencoder with $\mathcal{L}^{l1}$, $\mathcal{L}^{l2}$ and $\mathcal{L}^{SSIM}$ specified as their loss functions under two distinct testing scenarios: In the first stage, the so-called 'V1' models have been trained on the rad_41 dataset. Due to the different training time of the Autoencoder and the DnCNN for a single training step, the number of epochs varies from ten to twenty respectively. Afterwards, the models are evaluated on the test datasets especially on the rad_15 and pois dataset that were created under a modified undersampling scheme. Then, in a second stage, the training of the models is continued based on a joined training dataset consisting of all three noise types to test whether the pre-trained models adapt to multiple kinds of noise. This test setup is referred to as 'V2'. A first visual impression of the model performances on the rad_41 dataset can be gained from figure 8 and figure 10.

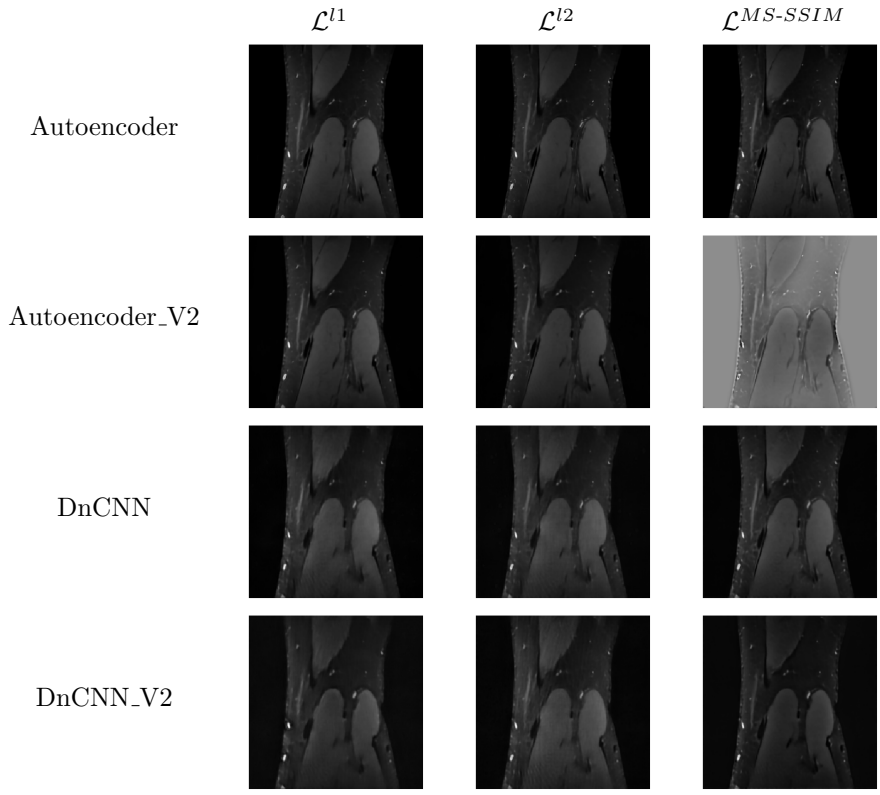


Figure 8: Predictions of all models on an image of the rad_41 dataset. Compare with figure 7.

The learning rate of the models is adapted in the following way. In the first stage, the learning rate is reduced to about a tenth of its original value. After the lower level is reached, throughout the second stage the learning rate remained constant. In both cases, the models are evaluated with the three test datasets separated by the noise structure.

All models are compared with the benchmark, i.e. the M-MAE, M-MSE and M-SSIM of the undersampled image to its fully sampled version on the respective dataset that can be seen in figure 11 as the black line. As a consequence, we can see whether the model on average improved the noisy images at all. Note in particular, that the severity of the information loss in each image due to the change in the sampling scheme from rad_41 to rad_15 is captured by the benchmark's peak in the M-MAE, M-MSE metric and M-SSIM metric. While the M-SSIM of the rad_41 and pois dataset are 0.72 and 0.7 respectively, the M-SSIM of rad_15 is on average only 0.36. From this perspective, even though they are of a different structure, the overall disruption level of rad_41 and pois seem to be comparable. Note that although the difference in the quality of the predictions is interpreted in terms of their model structure, one should

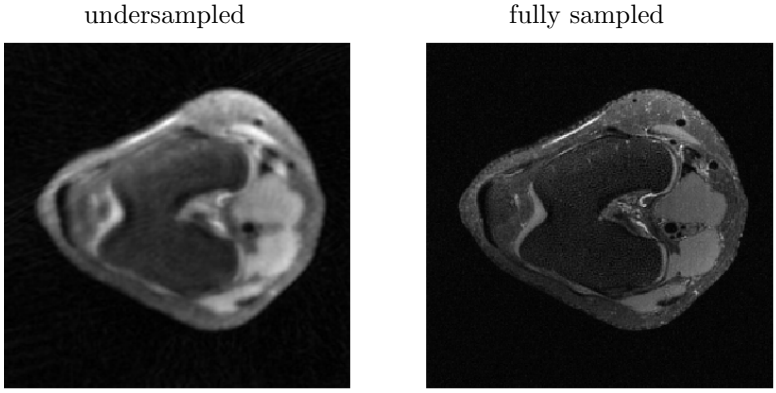


Figure 9: Under- & fully sampled image of the images in figure 10.

be aware that the models' realizations are determined by random initialization and random batch sampling during training. Another salient feature of the to be discussed graphics is the line type, separating the V1 and V2 setup of each model.
First, the performance of the DnCNN models is evaluated. As figure 11 indicates, all models broke the benchmarks. Unexpectedly, even though each metric was also used as loss, the respective model need not be the best performing one for that metric. This indicates that the models with different losses are robust against the evaluation metrics. Strikingly, the V1 models perform relatively similar on the rad_41 set-up that they also were trained on. Regardless of the test dataset, DnCNN_L1 and DnCNN_L2 are close to indistinguishable even on the unseen noise patterns in rad_15 and pois. The relative improvement of the DnCNN_L1, DnCNN_L2 and DnCNN_SSIM compared to the benchmark in the case of rad_15 is remarkable, especially since these models have never seen the far more disrupted data. This might indicate, that the models learned some general structure of the pattern introduced by radial undersampling, invariant of its level. In contrast, a major driver of M-MAE and M-MSE in the rad_15 benchmark is the overall bias in luminance due to radial undersampling in general, which the models correct for significantly.

Figure 12 concerns this suspicion: rad_15 and rad_41 introduce a considerable bias, that is corrected to some extent in the predictions. Of particular interest is the Poisson disc case, which produces a generally less biased image - at least in this parameter setting. The difference between DnCNN_L1 and DnCNN_L2_V2 is mainly due to bias-correction. The prediction of DnCNN_L2 on the pois dataset actually overcompensates the luminance bias. This is expected as the luminance bias in the rad_41 dataset is higher than in the pois dataset and the model is mainly trained on rad_41.

In general, the M-MAE and MSE metrics in figure 11 display a very similar pattern, but in comparison to the DnCNN_SSIM especially on the M-SSIM metric, there are considerable differences on the rad_15 and pois dataset. In M-MAE and M-MSE terms, the DnCNN_SSIM M-MAE and M-MSE error are double the errors of the

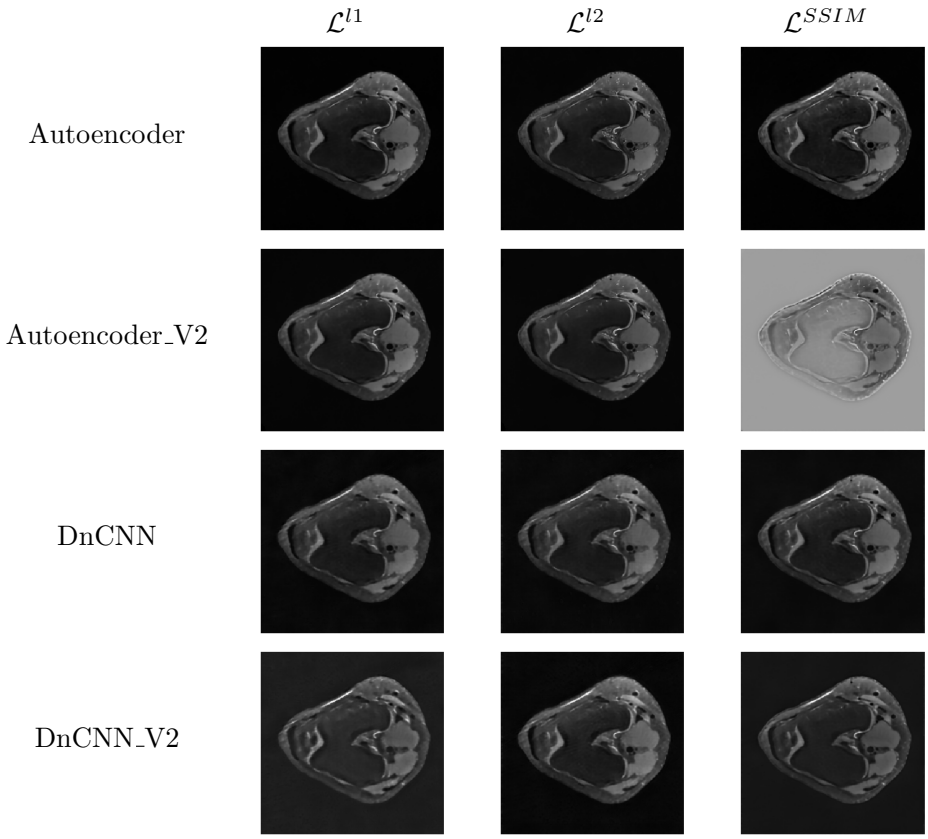


Figure 10: Predictions of all models on an image of the rad_41 dataset, second picture. Compare with figure 9.

DnCNN_L1 and DnCNN_L2 when applied to the pois test set. In terms of the M-SSIM metric however, all three models yield similar performance irrespective of the test dataset. Since the SSIM is designated to mimic the human perception of image quality, we consider the performance of the DnCNN_SSIM similar to the DnCNN_L1 and the DnCNN_L2. Nevertheless, it is interesting that the DnCNN_SSIM model found a way to perform similarly based on the M-SSIM metric while having a worse performance based on M-MAE and M-MSE. The motivation of Wang et al. (2004) regarding issues related to possible variations in visual quality under constant MSE and MAE, is a driving factor in this analysis. Considering the computational cost and potential instability associated with $\mathcal{L}^{SSIM}$'s complexity, the previous results commend the use of $\mathcal{L}^{l1}$ & $\mathcal{L}^{l2}$. However visual inspection of figure 8 suggests, that both DnCNN versions with $\mathcal{L}^{SSIM}$ actually remove those spikes, that are still present in those with $\mathcal{L}^{l1}$ and $\mathcal{L}^{l2}$.

Looking at the DnCNN_L2_V2's predictions in the respective datasets, it is apparent that in all three cases, the model actually broke the benchmark improving the

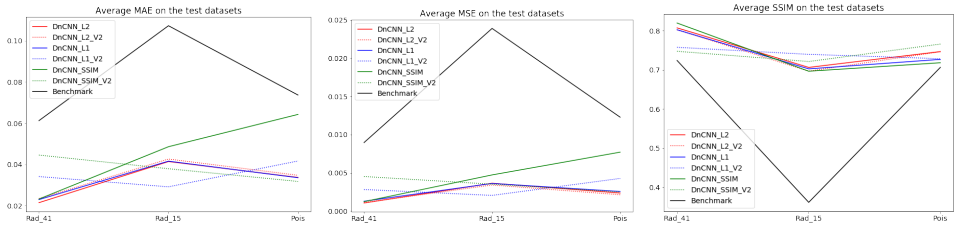


Figure 11: Performance of the DnCNN models on all three test datasets.

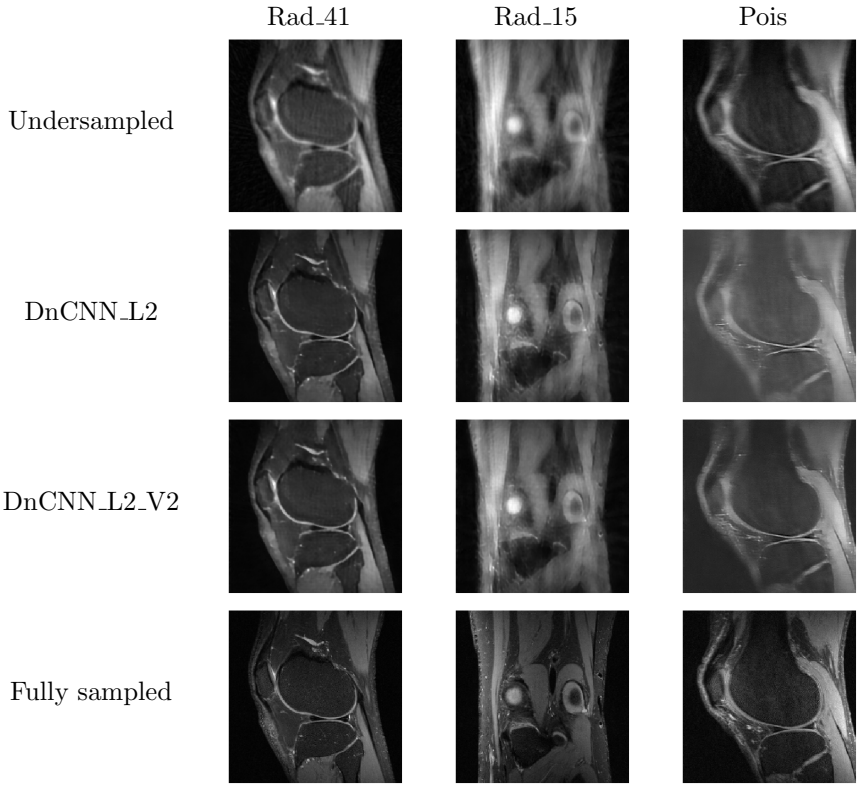


Figure 12: Performance of the DnCNN_L2 and the DnCNN_L2_V2 on images on all three test sets.

noisy image. Its overall results are not impressive, except for rad_41. In the latter case, we see in figure 12 it reduces the blur and removes some, but not all artifacts. This is more apparent in regions with tissue, that is lighter in colour. Although the bias on the overall luminance is reduced, it still exhibits a considerable bias between prediction and fully sampled image. In rad_15, the blur decreases, but the bias still remains unchanged. The overall image quality has hardly improved. In the

Poisson case, most of the undersampling is actually blur. In some sections, it gains considerable precision and unblurs, but the overall performance is still insufficient.

Considering the difference between the V1 and V2 models, one might imagine that after training on the joint dataset the models' performance, especially on the rad_15 and pois dataset, would increase. Interestingly, only minor improvements can be found for the DnCNN models. For the DnCNN_L2 model, almost no changes in performance on all test sets can be seen in comparison to the V2 version of the dataset. The DnCNN_L1 model V2 can improve its prediction, especially on the rad_15 while it deteriorates on both the rad_41 and the pois test dataset. The DnCNN_SSIM_V2 behaves similarly as it improves its performance on the rad_15 and the pois dataset while it loses accuracy on the rad_41 datasets. Overall the adaption to the new noise structure was not successful for the DnCNN model regardless of the loss function. Either no improvements are made or improvements on the new datasets come with major setbacks on the original dataset. This could indicate that the DnCNN architecture is not capable of denoising multiple different noise patterns at the same. Looking at figure 11 the performance of the DnCNN_L2 is compared to the DnCNN_L2_V2 leads to a similar conclusion. While for the sample from the pois dataset and the rad_15 the DnCNN_L2_V2 model improves the image slightly better, the prediction on the rad_41 dataset has deteriorated. The overall impression is that while the performance on the rad_41 dataset is reasonable for both models, on the other dataset the performance is not sufficient. When choosing the best model among the DnCNN models, both the DnCNN_L2 and DnCNN_L2_V2 seem viable choices as they perform robustly on all test sets and under all metrics. Although the DnCNN_L2_V2 has a slightly worse performance based on figure 11, the visual results in figure 12 indicate that the slight deterioration on the rad_41 dataset is justified by better performances on rad_15 and pois. Especially a slight bias correction can be seen for the image from the pois dataset.

In the case of the Autoencoder models, almost all models break the benchmark, except for the $\mathcal{L}^{SSIM}$ V2 configuration, as can be seen even visually in figure 8. As this specification breaks the scale, it is excluded from the figure. Apparently, during training, the network massively deteriorates for an unknown reason. Nevertheless, it is the only model in the vast amount of specifications, that significantly deteriorated in the very late second stage of training. While it is common for models to suffer from these problems especially after initialization, the fact, that we do not observe this for the other loss functions may hint towards a less robust learning procedure for the models using $\mathcal{L}^{SSIM}$ as a loss. Looking at figure 13 on the rad_41 dataset, all model performances are fairly similar, merely the $\mathcal{L}^{SSIM}$ seems to fall marginally behind in the M-MAE metric. The ordering of the models' denoising performance preserves across all metrics. Even with $\mathcal{L}^{SSIM}$ performing marginally worse on rad_41 in M-MAE terms, it has comparable performance in the other metrics. However, $\mathcal{L}^{SSIM}$ without further training seems to yield the best generalization capabilities on rad_15 and best performance on pois, so it is a promising candidate. One interesting observation is that for the DnCNN models as can be seen in figure 11 the DnCNN_SSIM does perform similar in terms of the M-SSIM metrics but has a worse performance based on M-MAE and M-MSE. The Autoncoder_SSIM on the other hand does perform similar based on the M-SSIM metric but actually beats the other models in

terms of M-MSE and M-MAE. This indicates that a model with the $\mathcal{L}^{SSIM}$ does not necessarily perform worse based on the M-MAE and M-MSE. Most interestingly and in contrast to the DnCNN configurations, having trained on the joined dataset, the Autoencoder_L1 and Autencoder_L2 have a better performance on all datasets. Note, however, that the V2 models always has seen more batches during training since they are the V1 version of the same model that was additionally trained on all datasets. Since Autoencoder_L2_V2 outperforms Autoencoder_L1_V2 consistently in all metrics, we choose it as the best performing model of all configurations.

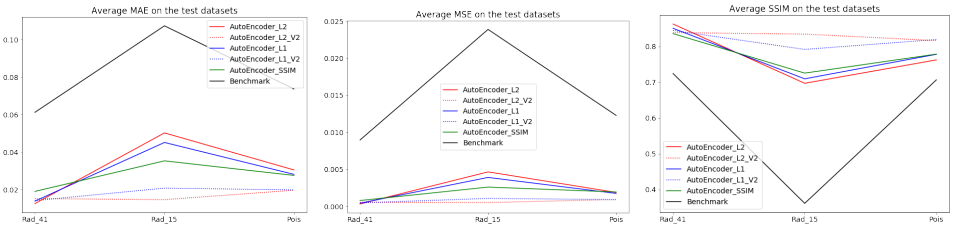


Figure 13: Performance of the Autoencoder models on the test datasets.

Figure 14 displays the visual performance of the Autoencoder and Autoencoder_V2 models. In both cases on rad_41, there are close to no artifacts remaining and the tissue surfaces are clean. Merely in the upper right light tissue, a mild wave pattern is maintained. The images' biases are corrected and considerable amount of detail is reconstructed. The generalization of $\mathcal{L}^{l2}$ in rad_15 already gains considerable precision in some parts of the tissue. In this example, the upper part of the image is already sufficiently denoised. The lower parts are still disrupted, but in comparison to its starting point, the image's artifacts are beginning to fade and even hardly existing detail is reconstructed. Note particularly the falsely translated part at the mid-left, where the artifacts in the undersampled image are interpreted as light tissue. Considering the performance on pois, most of the blur is removed introducing more contrast and better detail. Also, the mild bias is reduced most of which were prominent in the light tissue on the mid-right and the light grey of the bone in the center. The V2 version adapts to all three noises significantly with hardly any deterioration on rad_41. The overall image quality of predictions on the rad_15 and pois dataset is in no way inferior to the of rad_41. Considering their starting points, and the gained detail, this is a significant achievement.

In a direct visual comparison of all model configurations as in figure 8, the Autoencoders - irrespective of the applied loss function - provide very good predictions on rad_41, excluding $\mathcal{L}^{SSIM}$ V2, which failed during training. The DnCNNs performance on the same dataset is remarkable but comparatively inferior. Only the DnCNN $\mathcal{L}^{SSIM}$ models succeed in removing the artifacts entirely, but at cost of a lower luminance in comparison to the fully sampled in case of $\mathcal{L}^{l2}$ V2. In both cases, DnCNN and Autoencoder, $\mathcal{L}^{SSIM}$ seems to produce visually pleasant results, gaining a considerable amount of detail. The visual performance differences between the V2 versions DnCNN_L2_V2 in figure 12 compared with Autoencoder_L2_V2 in figure 14 are represented by the performance gap in figure 15.

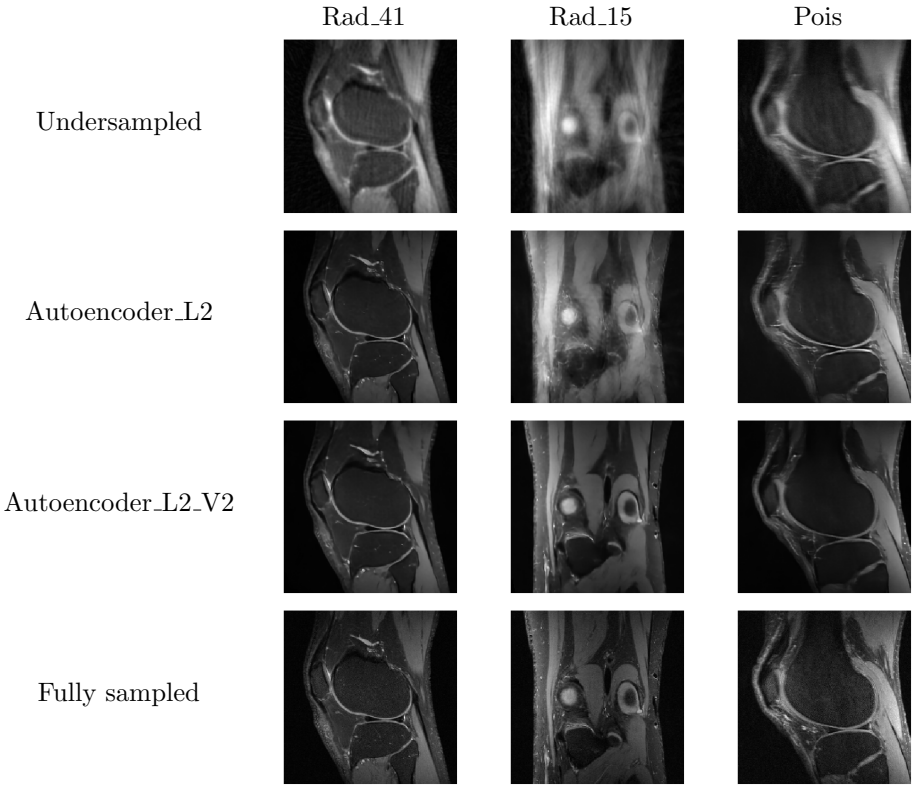


Figure 14: Performance of the Autoencoder_L2 and Autoencoder_L2_V2 on images on all three test sets.

6.2 Model Performance on Coil Input

As a last part of the analysis, it is tested whether the developed models, namely the DnCNN and the Autoencoder predict better when coil information is supplied as input. In figure 16 the predictions of the DnCNN, the Autoencoder and an adapted version of the Autoencoder are displayed. The coil information does consist of eight arrays of complex numbers, each representing the K-space by its respective coil. As explained in chapter 5 and in line with the typical methodology shown in chapter 4 the imaginary and real parts of the arrays are split and inserted as separate layers into the arrays. Then these arrays with 16 layers are inserted into the two architectures that were introduced in chapter 3.4 and chapter 3.2. These networks are trained for 48 hours. To produce the images shown in figure 16 from the predicted arrays the complex arrays are recreated with their eight layers of complex numbers. Using the BART software these are combined into one K-space and converted back to a single picture.

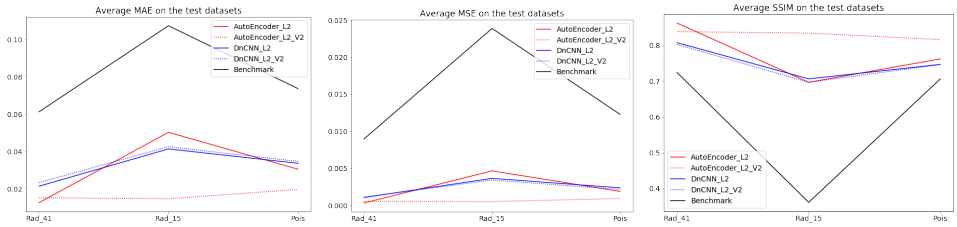


Figure 15: Performance of the best DnCNN & Autoencoder models on test datasets.

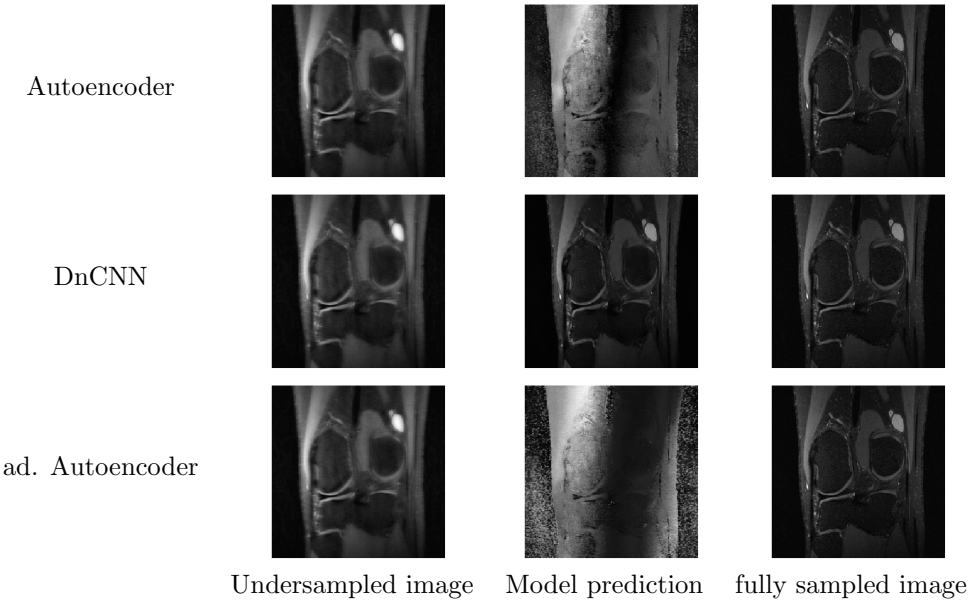


Figure 16: Top to bottom: Predictions on coil data with Poisson disc undersampling of the Autoencoder, DnCNN and adapted Autoencoder.

Comparing the predictions of the DnCNN and those of the Autoencoder in this setup, one clearly sees that while the DnCNN recreates the true image almost perfectly, the Autoencoder is not able to predict some meaningful results. Therefore, the DnCNN is the more promising architectures when it comes to using coil information. Since for the coil information only the Poisson disc undersampling was available, the direct comparison between the performance of the best models using the image-space and the DnCNN based on the coil information is complicated. The models working on the image-space are mainly trained on the radial undersampled data and only transfer learned to denoise the images created with Poisson disc undersampling. The coil DnCNN on the other hand is only trained on the Poisson disc undersampling dataset and may be more specialized to these artifacts therefore has an advantage obstructing a direct comparison to the previous models. Nevertheless, the DnCNN

using coil data has fewer observations available to learn from and had a shorter time to train. Additionally, the function the network has to approximate is considerably more complicated having an input and output of dimension 256x256x16 instead of 256x256. Comparing the images that result from the DnCNN based on the coil information to predictions of the other models based on the images-space, it performs similarly to the best of these models. As can be seen in figure 13 the Autoencoder_L1_V2 has an M-SSIM of 0.81 on the pois dataset and is the best performing model on that dataset. With an M-SSIM of 0.79 the DnCNN based on the converted coil information matches that performance and beats all other models.

The fact that the DnCNN is able to produce such strong results indicates that there is no mistake in the data pre- and postprocessing and that the lack of convergence of the Autoencoder is due to limitation of the architecture itself.

Still, since the Autoencoder is the most successful architecture in the models based on image-space, some effort is done to improve its performance on the dataset including the coil information. As introduced in chapter 3.4 and shown in chapter 3 the Autoencoder has 64 filter in the first layer of the model. Using a gray-scale image as input this results in 64 filters per input layer. But using the coil dataset as input one has 16 layers and hence 4 filters per layer. One reason the network does not perform could be the reduced number of effective filters per layer in reference to the input format and hence the lack of information that can enter the model from the different layers. Therefore, the number of filters of the first two levels of the Autoencoder is increased to 256. The results of the model with the adapted architecture can be seen in the figure's 16 last row. Unfortunately, no improvements of the undersampled images can be seen and the predictions look very similar to the output of the classical Autoencoder architecture. In figure 17 the loss functions of the tweaked Autoencoder and the DnCNN during the training with Adam over the first hundred-thousand iterations are visualized. For the tweaked Autoencoder the optimizer did not have success in minimizing the loss function and the latter seems to idle around the same level over the whole period of the training. Therefore, it seems that the network is not able to learn how to improve the input data. For the DnCNN we also see a lot of variation, but nevertheless, the loss function is on average significantly reduced during the training. This is most apparent from the predictions of the network that actually improve the images.

Unfortunately, even with some adjustments in the architecture, the Autoencoder does not seem to be able to handle the input of coil data. This is even more striking as the DnCNN in the same set-up is able to produce some of the best results among all models. Nevertheless, the proof of concept succeeded in showing that the DnCNN architecture can also perform based on coil information.

6.3 Discussion

First of all, some critical points have to be made. As mentioned throughout the analysis, the high correlation between the slices as well as the homogeneous structures within the datasets could induce overfitting. Additionally, the used undersampling schemes might not fully represent real-world MR image artifacts. Then the trained models would not perform as good on that data. One last point of concern is the

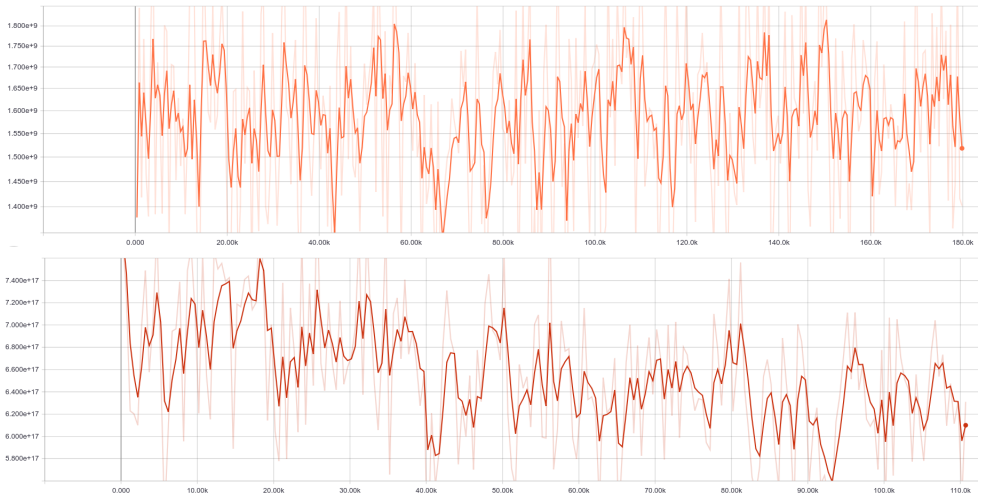


Figure 17: Top to bottom: Values of the loss function during training of the tweaked Autoencoder and the DnCNN.

different run-time of the models. Due to technical reasons, not all models had a comparable run-time, namely most of the DnCNN had fewer run-time and steps. Nevertheless, the models with higher run-time did not necessarily perform better as for instance, the DnCNN_L1, with longer run-time did perform significantly worse.

The result can be separated along with the input format that the models were trained on. First, for the models using the image-space, the two architectures were compared as well as different loss functions. When comparing the Autoencoder to the DnCNN architecture, the Autoencoder dominates both, in the evaluation metrics and the visual comparison of the predictions. Therefore, this analysis concludes that it is the better architecture to denoise MR images in the image-space. Comparing the different loss functions, namely the $\mathcal{L}^{l1}$, $\mathcal{L}^{l2}$ and $\mathcal{L}^{SSIM}$ the differences of the model performances are not as significant as anticipated. Nevertheless, the $\mathcal{L}^{SSIM}$ is a viable alternative. For the DnCNN the $\mathcal{L}^{SSIM}$ created the only comparable results to the AutoEncoder. In the V1 set-up of Autoencoder, the $\mathcal{L}^{SSIM}$ created the best performing model based on the evaluation metrics. Nevertheless, looking at the predictions no difference in quality could be identified. Unfortunately, the V2 Version did not learn properly although it is a promising candidate. Further research into the use of the loss function $\mathcal{L}^{MS\text{-}SSIM}$ and combinations of the $\mathcal{L}^{MS\text{-}SSIM}$ with $\mathcal{L}^{l1}$ and $\mathcal{L}^{l2}$ could result in even better performances.

Using the coil information we came to the opposite conclusion. Despite its great performance using the image-space, the Autoencoder did not produce meaningful results. The DnCNN produces some of the best results across all models, evaluated based on the image-space of the predictions. Restricting the comparison, all other models based on the image-space did only transfer learn on the pois dataset whereas the coil models exclusively trained on Poisson undersampled data. As only minor

changes to the architecture are introduced, further research into the use of the models trained on the coil information could result in even better performance.

7 Grid Search on U-net

Up until now, the discussed architectures' modifications are reasoned heuristics to make the originally proposed architectures suitable for the various denoising strategies employed earlier. Despite the astonishing visual performances of Autoencoder_V2 and DnCNN_L2, the next chapter is more rigorous about the choice of parameters in order to tailor the U-Net to the task of radial undersampling. This chapter introduces a flexible and scalable version of the U-Net, setting up the parameters under investigation. Particular interest lies in the various loss flavours of chapter 2.2 and the effect of skipping.

7.1 U-Net Modifications

In the previous chapters, we already modified the original U-Net as displayed in figure 2 in five important aspects to fit the denoising task. Firstly, the images are no longer cropped when skipped. As the artefacts propagation in the K-space transformation affects the entire image globally, this motivates to use of the entire image rather than tiles. Secondly, and in the same manner, we changed the upward path's valid convolutions to the same convolutions to retain the representation's size on the same depth level to facilitate skipping. Thirdly, we added BN, to reduce the burden of higher-order nonlinear interactions between layers during gradient descent. Fourthly, we tested three loss functions, namely mean absolute error ($\mathcal{L}^{l1}$), mean squared error ($\mathcal{L}^{l2}$) and the structural similarity index ($\mathcal{L}^{SSIM}$) for denoising, motivated by Zhao et al. (2017). Further, we considered the multiscale structural similarity index ($\mathcal{L}^{MS\text{-}SSIM}$) and a compound measure $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$, combining both MS-SSIM and a convolutionally Gaussian-weighted version of the L1 norm. However, at training time, the latter two were not available. Previously, both $\mathcal{L}^{l1}$ & $\mathcal{L}^{SSIM}$ proved to be the most promising candidates in image denoising, even though $\mathcal{L}^{l1}$ produced very reasonable results in benchmark comparison. Furthermore, the results suggested that $\mathcal{L}^{SSIM}$, due to its complexity, might be more unstable during training. The following analysis augments the previous loss functions by a running version of $\mathcal{L}^{MS\text{-}SSIM}$ and $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$. Fifth, the U-Net's autoencoding capabilities are enhanced with increased depth; such that two levels are added, each with two convolutional layers with 1024 feature maps.

Based on the previous modified U-Net architecture that produced visually astonishing denoising performance, even in transfer learning on new noises, this chapter seeks to optimise the parameters of the U-Net with regard to depth, width and width layout of the network's levels and layers, as well as with respect to the receptive field of the convolutions. To do so, this chapter provides a flexibly scalable version of the U-net, that is readily extended to facilitate further, more directed research. Consider the following code that determines the entire grid of architectures of the same depth that are to be trained, by specifying a parameter map only:

Source Code 1: Cast the grid of U-Nets

```
1  import itertools
2  import pickle
3
4  # parameter grid
5  configs = {
6      'lossflavour': ['MAE','MSE', 'SSIM', 'MS-SSIM', 'MS-SSIM-GL1'],
7      'batch': [2, 4],
8      'kernel': [5, 9],
9      'filters':[[125,100,50,100,125],
10     [100,100,100,100,100],[266,266,266,266,266]],
11     'reps': [[3,3,3,3,3]],
12     'dncnn_skip': [False],
13     'sampling' : [True],
14     'Uskip' : [True],
15     'trainsteps': [4000, 6000]
16 }
17
18 # carthesian product: parametermap
19 l = list(dict(zip(configs, x)) for x in itertools.product(*configs.values()))
20
21 # l[0] : {'lossflavour': 'MAE', 'batch': 2, 'kernel': 5,
22 # 'filters': [125, 100, 50, 100, 125], 'reps': [3, 3, 3, 3, 3],
23 # 'dncnn_skip': False, 'sampling': True, 'Uskip': True, 'trainsteps': 4000}
24
25 # save parametermap
26 with open(root + "grid.txt", "wb") as fp:
27     pickle.dump(l, fp)
```

The dictionary `configs` contains all the information regarding the blueprints of all architectures to be tested. Before going into detail, consider line 19, which expands the grid in the form of a Cartesian product on all list members of the dictionary's keys. The first instance of such an expansion is given as an example in the comment starting in line 21. Regarding the parameters, `lossflavour` specifies the loss function which is to be minimized. Most important is the `reps` parameter, as its nested list's length determines the depth of the entire network. This becomes apparent in the illustrative Tensorboard graphs in figure 18.[5] Each value of `reps` implies a REPEAT block, that has as many stacked convolutions as the actual value of `reps` at that

[5]Note, that these graphs are produced using `tf.contrib.layer` layers, that are replaced by `tf.layers` in the actual implementation due to technical issues. While the former's representation of the functionality produces visually pleasant and illustrative graphs, it fails consistently during training. The coding issue is not yet resolved. The intended operations are therefore replaced, but the general structure of the architecture is preserved.

index position. All of the stacked convolutions in one REPEAT block are supplied with the same number of filters at the respective index position in `filters`. E.g. in the printed example, the first REPEAT block consists of three convolutions, each generating 125 feature maps. Depending on being in the encoder or decoder part of the network, the respective down or upsampling operation in the form of max-pooling and deconvolution is applied to the result of the `Repeat` block. Note, how the skip connections are automatically adjusted and the Conv2d_transpose's result is added to the respective skipped image, before being passed as input to the next `Repeat` block.

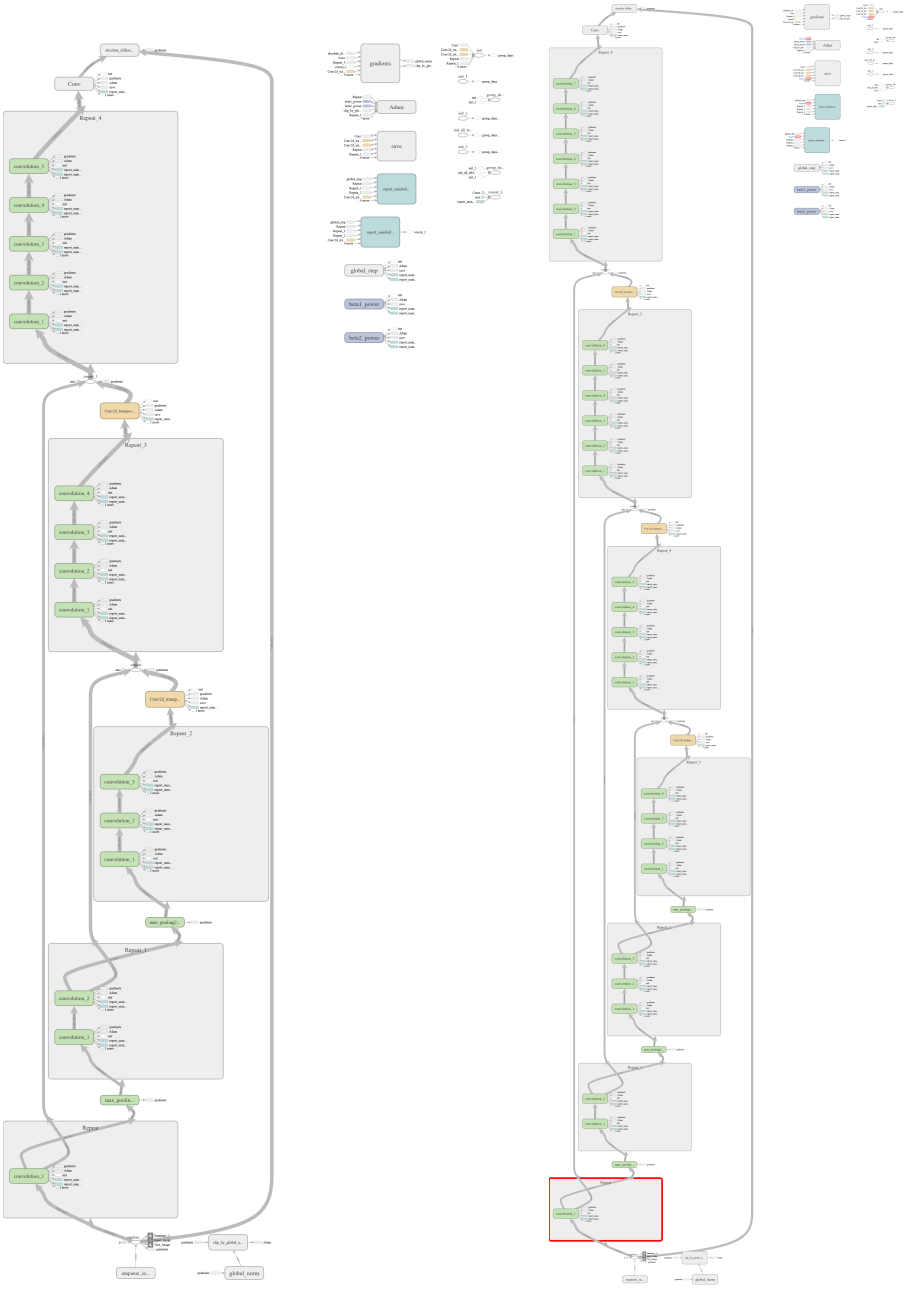


Figure 18: Examples of scalable graph architecture.
$'reps' = [[1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 6, 7]]$. These figures are created using Tensorboard. It is highly recommended to zoom in on them in the PDF Version of this document.

The drawback of the above interface is a fixed depth in one grid expansion, as otherwise, the Cartesian product would match up a `filters` and a `reps` parameter list, that is of uneven size. However, it enables the user to test various configurations of filters across the network. Consequently, to consider depth as parameter, another grid is expanded with `filters:[[314,314,314,314,314,314,314], [100,200,400,200,400,200,100]`[6]`]` and
`reps:[[2,2,2,2,2,2,2]]`, but else same parameters. Note that the modified U-Net design from which the above variations evolve can be obtained with:

Source Code 2: Modified U-Net Specifications

```
configs = {
    'lossflavour': ['MAE', 'MSE', 'SSIM'],
    'batch': [6],
    'kernel': [3],
    'filters':[[64,128,256,512,1024,1024,1024,512,256,128,64]],
    'reps': [[2,2,2,2,2,2,2,2,2,2,2]],
    'dncnn_skip': [False],
    'sampling' : [True],
    'Uskip' : [True],
    'trainingsteps':[100000]
}
```

To reduce computational complexity, the architectures are decreased in depth. This encourages us to examine the composition of the width and necessity of our previously modified U-Net's depth. To examine the width structure in the network, the shallower depth specifications in Source Code 1 are structured as follows: While maintaining the same number of `filters`, the configurations `[[125,100,50,100,125]` and `[100,100,100,100,100]` exhibit a U and uniform structure. The same was intended in the deeper grid specifications `[314,314,314,314,314,314,314]` and `[100,200,400,200,400,200,100]`[7]. In addition, the deeper network specifications in Source Code 1 are aimed at being comparable in total filters throughout the entire network, as the number of repetitions is decreased to 2, such that they contain only about 2.7 times as many filters in total. The flatter network with 266 filters and 3 repetitions is intended to be directly comparable in terms of total number of filters (i.e. `filters` $*$ *depth* $*$ `reps`: $266 * 5 * 3 = 3945$ opposed to $314 * 7 * 2 = 4396$ of the deeper uniform version). Ideally, this makes those models' training complexity comparable, if the increased number of skip connections and sampling layers are not considered. A thorough investigation of the performance difference of those two versions may yield insights upon the usage of sampling. Even further, due to computational constraints,

[6]This configuration has a typographical error, namely the deepest level is supposed to be 800, such that the number of filters match up with those of the filter configuration with same depth. However, the models were trained in this faulty specification, effectively reducing the autoencoders' capabilities.

[7]See footnote 6

the networks are trained with a rather small batch size and training step configurations. Consequently, the models are far from convergence. All of the computational constraints come at the cost of higher variability in the results and potentially at a preference for simpler models, in particular simpler loss functions, that yield reasonable performance early-on, but might be significantly outperformed later during training. Consider, that those model realizations are favoured, that learn the identity mapping fast, starting from the weight-initialisation. Nevertheless, the above test scenarios may yield valuable insight into how to improve the U-Net architecture and the necessities regarding structure for performance gains. To acknowledge the variability in some sense, the architectures with the exact same parameter maps are trained twice; once with 4000 and once with 6000 steps. This is done consciously at the cost of training one model with 10000 steps, as even then, the models are far from convergence. In this setup, at least two snapshots of training realizations are available. Consider that `configs` with shallow depth already implies 120 networks, half of them fitted with 4000 and the other half with 6000 steps. Including the deeper networks, the total number of fitted networks is 200. To reduce wall-clock-time, the configurations are split by `lossflavour` into multiple configuration files and scheduled as separate server jobs. The vast amount of models and the variability control has severe implications regarding the choice of parameters in the parameter grid due to computational effort: Apart from limitations concerning the depth, a total number of filters and the low number of training steps, the batch size is also decreased, as in early testing stage memory allocation issues prohibited[8] training. In line with the considerations in Zhang et al. (2016) concerning the receptive field implied by a global kernel size for all convolutions, two kernel configurations are tested. The employed optimizer for all configurations is ADAM (Kingma & Ba 2014).

[8]Considering the depth & width of the modified U-Net being trained with batch size 8, this is obscure. Later debugging at runtime attributes these memory issues to the checkpoint-step behaviour during training. Due to their computational cost, specifications with increased batch size are prone to memory-related issues

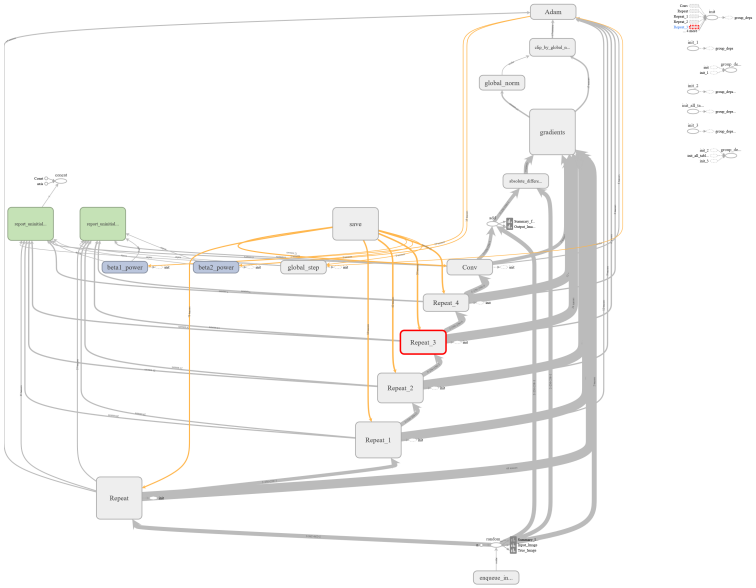


Figure 19: DnCNN structure with additive skip connection. This figure is created using Tensorboard. It is highly recommended to zoom in on it in the PDF version of this document.

Source Code 3: DnCNN Specification

```
configs = {
    'lossflavour': ['MSE'],
    'batch': [128], # (as mini-batch example taken from DnCNN)
    'kernel': [3],
    'filters':[[64]],
    'reps': [[18]],
    'dncnn_skip': [True],
    'sampling' : [False],
    'Uskip' : [False],
    'trainingsteps':[100000]
}
```

At last note that this architecture is written such, that with the parameter configuration {`dncnn_skip:[True], sampling:[False], Uskip:[False]`}, scalable versions of the DnCNN model as described in Zhang et al. (2016) are accessible as special case. Consider the original DnCNNs parameter map in Source Code 3. The obtained DnCNN graph is displayed in figure 19. Each of the REPEAT blocks follows the structure described above. The parameter `dncnn_skip:[True]` adds an additive skip-connection from input to the prediction layer. `sampling:[False]` removes the

pooling and deconvolution layers, that down and up sample the image, effectively removing the U-Net structure. `Uskip:[False]` removes the concatenating skip connections that are present in figure 2. These parameters can be switched on and off at the users disposal, allowing for various testing scenarios that are beyond the scope of this chapter.

7.2 Gridsearch Results U-Net

As Zhao et al. (2017) suggests, the measures MAE, MSE and SSIM weight disruptions in structural image aspects, such as surfaces or edges differently during optimization and therefore lend themselves as evaluation metrics. SSIM is even intended to mimic human perception to some extent, which may deviate from optimal MSE and MAE significantly. Also, SSIM has favourable properties regarding the unique maximum in 1 given the two images are identical. Additionally, SSIM is symmetric in its arguments. In the case of contrary model performances regarding these measures, SSIM is preferential since denoising aims at visual a performance. Our previous results (Toebrock & Ruhkopf 2019) also indicated that models trained with the respective metric need not be the best performing in that metric which underlines their usage as a performance measure.

Unfortunately, 62 of 125 configurations of the overly ambitious sized grid aborted their execution due to technical difficulties. The remainder's performance is depicted in 20 irrespective of losses and number of training steps and should be interpreted in the light of table 1. First and foremost, consider the benchmark value representing the mean difference between the underlying true image and its noisy version, representing the degree of mean distortion level in the test sample in the respective metric. M-MAE immediately reveals the brightness bias introduced by undersampling the underlying true image in the creation of the training set. Most architectures are centered around zero in M-MAE, indicating a brightness adjustment in almost all models. M-MSE reveals its high penalty on strong deviances, which are also apparent in M-MAE. Despite varying closely around zero in image mean in both M-MAE and M-MSE, most revealing of the little that all architectures have learned beyond the brightness adjustment is M-SSIM as it is robust towards this adjustment. Its benchmark reveals the high visual similarity apparent in figure 21. No model surpasses this benchmark by a significant amount towards the value of one or the visually astonishing performance of the architecture Autoencoder_L2 in the previous chapters with an M-SSIM value close to 0.85.

Table 1: Configuration IDs.

configID	filters	reps	batch	kernel
1	[100, 100, 100, 100, 100]	[3, 3, 3, 3, 3]	2	5
2	[100, 100, 100, 100, 100]	[3, 3, 3, 3, 3]	2	9
3	[100, 100, 100, 100, 100]	[3, 3, 3, 3, 3]	4	5
4	[100, 200, 400, 200, 400, 200, 100]	[2, 2, 2, 2, 2, 2, 2]	2	5
5	[100, 200, 400, 200, 400, 200, 100]	[2, 2, 2, 2, 2, 2, 2]	2	9
6	[100, 200, 400, 200, 400, 200, 100]	[2, 2, 2, 2, 2, 2, 2]	4	5
7	[100, 200, 400, 200, 400, 200, 100]	[2, 2, 2, 2, 2, 2, 2]	4	9
8	[125, 100, 50, 100, 125]	[3, 3, 3, 3, 3]	2	5
9	[125, 100, 50, 100, 125]	[3, 3, 3, 3, 3]	2	9
10	[125, 100, 50, 100, 125]	[3, 3, 3, 3, 3]	4	5
11	[125, 100, 50, 100, 125]	[3, 3, 3, 3, 3]	4	9
12	[266, 266, 266, 266, 266]	[3, 3, 3, 3, 3]	2	5
13	[266, 266, 266, 266, 266]	[3, 3, 3, 3, 3]	2	9
14	[314, 314, 314, 314, 314, 314, 314]	[2, 2, 2, 2, 2, 2, 2]	2	5
15	[314, 314, 314, 314, 314, 314, 314]	[2, 2, 2, 2, 2, 2, 2]	2	9

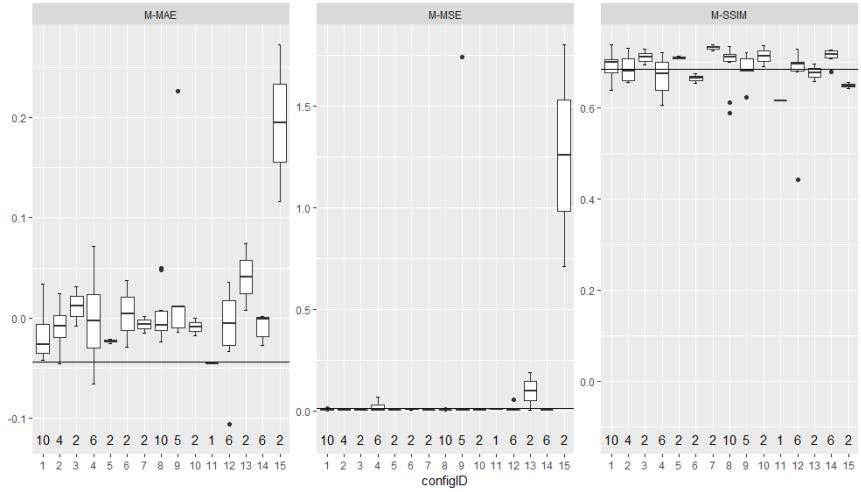


Figure 20: Average performance for configIDs 1-15. Benchmark mean value of the disrupted images in the test set compared with their underlying true image, evaluated in the respective metric is the horizontal black line. The number above configID is the count of models that did not abort execution. The possible number of successful models is 10: `lossflavour: ['MAE','MSE', 'SSIM', 'MS-SSIM', 'MS-SSIM-GL1'], trainingsteps:[4000, 6000]`. Consult table 1 for details on the parameter settings.

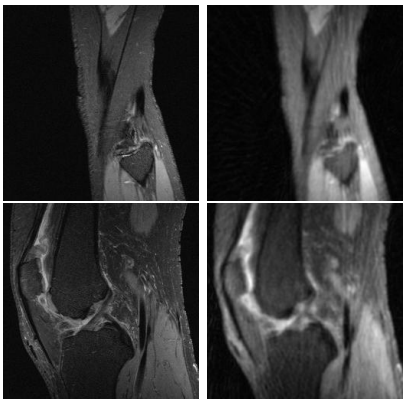


Figure 21: Underlying true and noisy image examples.

In terms of M-MAE and M-MSE, worst performing are both successful configurations 15 with different amounts of training steps, trained on the most complex loss $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$. Its results are depicted in the first two images of figure 22. A comparatively bad performance is achieved by one instance of configuration 9 trained on $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$. Noteworthy is also configuration 12 trained on $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$ in the lower-left image with its comparatively average performance in both M-MAE and M-MSE but underperformance in M-SSIM with a value of 0.443. The latter stresses the introductory observation of Zhao et al. (2017) and this thesis' approach to measure the performance of each architecture in all three metrics.

Considering the target and initial state in figure 21, the first four examples of $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$ display the least performing models mentioned above. The second row of figure 22 are rather illustrative for the predictions of all architecture configurations' performance at that very early stage. While most of the images produced are similar to the lower, rightmost image in 22; adjusting merely the brightness bias, many examples contain at least minor artifacts in varying degrees such as those of the remaining four images. This presumably can be attributed to both the random initialization and the beginning of learning beyond the identity mapping. Note that despite being less prominent due to the brightness adjustment, at this very early training stage, all architectures failed to remove the systematic ripples introduced by the undersampled k-space image and its Fourier transformation into image space. This is irrespective of both losses and the minor training step variations. Considering the complex composition of convolutions implied by these various architectures, and their kernel's random initialization, the already visible identity mapping with minor brightness adjustments are most probably a result of the skip connections inherent to the U-Net. Essentially, these images depict the skip connection's regularizing effect, allowing the Neural Nets to focus on residual learning early on. Interpretation beyond this level of aggregation is meaningless, as the resulting predictions this early reveal nothing but the identity mapping with minor artifacts in all architectures irrespective of the applied loss function.

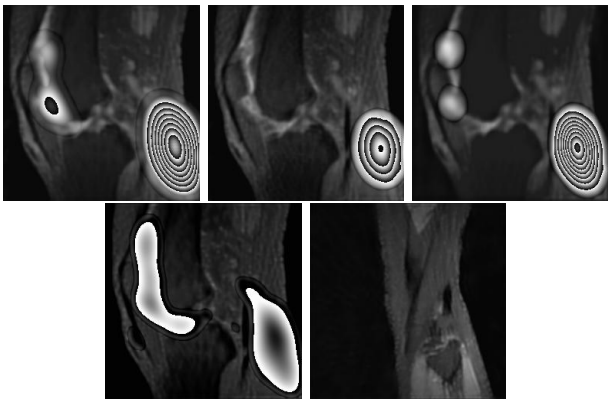


Figure 22: Prediction examples of multiple architectures trained with MS-SSIM-GL1 loss. From upper left to lower right, configIDs: 15, 15, 9, 12, 7 with trainingsteps: 4000, 6000, 4000, 4000, 6000 respectively.

7.3 Discussion

Augmenting the analysis' loss functions by $\mathcal{L}^{MS\text{-}SSIM}$ and $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$ has severe consequences on the total number of models to be fitted due to the Cartesian product. It comes with severe restrictions regarding the available network capabilities and the possible testing scenarios. Choosing one loss function reduces the grid to a large extent and allows examining architecture-related scenarios more detailed. Most importantly, the number of steps could be greatly increased, yielding less variable and hopefully more meaningful predictions. In this early stage, hardly any visual improvements are made. However, most of the models already predict images, that most likely result from identity mapping with minor variations. The learning process is only at its beginning, obstructing strong implications on the underlying architecture apart from how fast it can learn identity mapping after initialisation. To make use of the provided grid functions in larger step scenarios, from an implementational perspective, a warm start extension should be added, such that the progress in a stopped model is not lost. Apart from this, far more testing scenarios are implicitly available, namely all DnCNN parameters and U-Net & DnCNN architecture skip connection mixes. This may also enhance the DnCNN's performance. As the U-Nets stability in training presumably is due to the extensive use of skip connections, further conceivable extensions to the scalable U-Net architecture are residual blocks around larger Repeat blocks as in He et al. (2016) or even removing the U-Nets sampling in favour of Dense nets as described in Huang et al. (2017). From an implementation perspective, the `nodes` list in Source Code (Ruhkopf 2019), which effectively stores the entire `tf.graph`, requires only appropriate indexing to introduce this kind of skipping.

8 Grid Search on DnCNN

So far, the hyperparameter choices of the DnCNN, i.e. the depth, filter size and skipped connection layout were mostly based on the proposals of the original authors. The following chapter investigates the parameter space systematically. Therefore a grid over applicable values will be used to test models for improving radial undersampeled images.

8.1 Potential Parameters for the Grid Search

Based on the presented DnCNN architecture, potential candidates for hyperparameters are identified. The candidates can be separated into two groups. First, some of the parameters do not change the architectures from the proposed DncNN while others do. As previously explained, the DnCNN does use residual learning. But in comparison to the typical use of residual learning, as introduced in He et al. (2015), the skipped scheme was changed. Instead of multiple skip connections, where each connection skips two layers of the network, only a single skip connection over the whole network is used as can be seen in figure 19. Especially in the case of additive noise, this is very intuitive. There the noise is the difference between the original image and the noisy image and the network only learns to reconstruct the noise. Unfortunately, in the case of undersampled MR images, the noise is highly nonlinear and therefore the residual will also contain significant parts of the image. That's why one could also resolve to the original use of the skipped connection which might allow for the training of even deeper convolutional networks. From here on the way of designing the skipped layers is called " skipped scheme" and two set-ups are called the DnCNN or ResNet skipped scheme. The architecture of the network using the ResNet skipped scheme can be seen in figure 23.

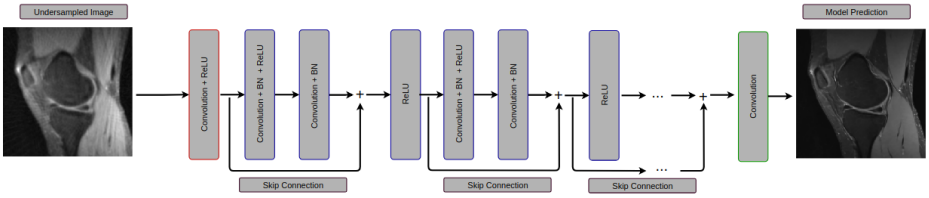


Figure 23: The ResNet skipped scheme for the DnCNN.

Beyond the skipped scheme one of the main parameters of the networks is the depth. It defines how often the middle layers using convolution, batch normalization and ReLU, are repeated. The Authors of Zhang et al. (2016) propose a depth between sixteen and eighteen layers depending on the complexity of the noise. Because of the complicated noise structure of undersampled MR images, eighteen repetitions of the main layer were chosen in the practical statistical training. As the authors already indicated, deeper networks might work better for complicated noise structures and the model could benefit from a depth beyond eighteen repetitions. On the other hand,

deeper networks are also harder to train and maybe a smaller network is already able to predict the noise correctly. Therefore, the number of repeated layers varies from ten to twenty-five.

Yet another important parameter of the network is the number of filters that are used in each of the convolutional layers. Although alternating the number of filters used between the layers is possible, the grid search will follow the authors of Zhang et al. (2016) and use a constant number of filters per layer. Using more filters gives the network more features to create which could improve the performance. Nevertheless, it also makes the network harder to train especially since more filters mean more trainable parameters and more partial derivatives to calculate. Since the DnCNN does use 64 filters, the parameter varies between 16 and 128 during the grid search.

For the convolution operation, filters of a certain size are strode over the image. Typical filter sizes are either 3x3, 5x5 or 7x7 where a higher number of filters allows for more complicated feature detectors. This could reduce the necessary size of the network. But it does also increase the number of trainable parameters significantly. For a single layer of convolution with 64 filters, using 3x3 filers results in 640 learnable parameters, including biases, while using 7x7 filters already results in 3200 parameters. Therefore, a higher filter size could make the learning process unstable and yield worse results. Commonly, the DnCNN uses 3x3 filters and during the grid search filter sizes vary between 3, 5 and 7.

Following the analysis of the practical statistical training, different loss functions will be tested. Once again the $\mathcal{L}^{l1}$, $\mathcal{L}^{l2}$ and $\mathcal{L}^{SSIM}$ operations will be used as candidates. networks using different loss functions might converge faster and the loss function could also interact with the optimal choice of the other parameters.

8.2 Challenges and Set-up of the Grid search

The grid search of the DnCNN comes with some particular challenges that will be addressed in this chapter.

Running a grid search for the hyperparameters of a neural network of unknown architecture is particularly challenging. Fortunately, in this paper the general architecture is already known. Therefore, for most of the parameters, the choice of the original authors is used and only some parameters of particular interest will be included in the grid search. Additionally, the use of convolutional neural networks eases the process even further as only the number of layers, the respective number of filters and their sizes have to be chosen.

The second problem is the massive computational complexity of training a convolutional network. To train the networks, graphical processing units were used that had either eight or eleven gigabytes of internal memory. For the training of a single gradient descent step, all activations and trainable parameters of the batch have to be held in memory. This limits the maximal size of the networks as well as the size of the batches. To make very large networks possible a batch size of three images was chosen.

Another concern is the time you have to give each network to learn from the data. First of all, this time is limited since many networks have to be trained and evaluated. Additionally, to compare the results of the grid search, each network should have a similar time to train on the data. There are two potential approaches to limiting training exposure. First, one could limit the wall-clock time each network has to train on the data. Deeper networks take longer for each step of gradient descent. Using the wall-clock time would result in the larger networks having fewer steps and hence seen fewer data samples. Another approach would be to allow each network the same amount of gradient descent steps. Then, the larger networks would have a longer wall-clock time to train. In this grid search, the number of steps of each network was limited to 4000. For comparison, the models presented in the practical statistical training were trained for hundreds of thousands of steps. This limits the validity of this grid search as a bias towards smaller networks which can potentially learn faster from the data is introduced. Since larger networks might take more time to converge, training only 4000 steps could lead to smaller models beating larger ones which would be significantly better with the appropriate training time. The only way to alleviate this concern is to use drastically more steps per model. This was based on the provided hardware infeasible. Therefore, the results should be seen with some precautions.

Finally, neural networks exhibit some inherent randomness due to the initialization of the trainable parameters and the sampling of the batches. To address the random initialization, a seed was set within the tensorflow software. Therefore, the initialization is the same for all models. But this does not alleviate all concerns since some model set-ups might benefit more from a specific initialization than others. Additionally, the random sampling of the batches was disabled and instead the order of the data samples was randomly chosen. Consequently, it is consistent for all models. Each model has seen 4000 training steps with batch size 3. As this is number is far exceeded by the total number of training samples, the impact of non-random sampling of the batches should not be significant.

8.3 Programming the Grid Search

When programming an exhaustive grid Search manually, there are two steps. First, based on the provided input parameters all possible combinations have to be created. Second, the respective models have to be fitted, evaluated and the results have to be stored for later analysis. As this grid search was executed on the GWDG high-performance cluster and run as multiple jobs at the same time, the second step has to be executed in multiple instances at the same time. The code can be found in the Github repository at Toebrock (2019). The two logical parts of the grid Search were separated into two programs. The program "make_grid" internally creates a data frame that contains all model specifications that should be run. Additionally, the fields that contain the results of the model evaluation as well as runtime, model name and directory are stored in distinct variables. As the latter variables are only known after the run of the respective model, these variables are initialized to starting values. The directories of the models that have not been run so far are initialized to "Not Run". Thereby, the program that runs the different set-ups can infer that

this model set-up was not run so far. Therefore, each model run is specified in a single row and by looping over the data frame and running each row as an individual model, the second program can run the grid search. The data frame is then saved as a .csv file. Additionally, when the number of model set-ups exceeds a threshold that can be specified with the parameter max_len, the "make_grid" program stores the data frame in multiple .csv files.

The second program consists of the function "run_grid". This function is generic as it takes the location of the directory of the grid specified in a .csv file as an argument, loads that data frame and iterates over its rows. If the model was not run so far and the model location is set to "Not run", it trains the model with respective parameters from the data frame, evaluates its run and stores the model results in the .csv file.

This logic is convenient for multiple reasons. First of all, it is easy to use. For each created .csv file that contains a part of the total grid a job has to be scheduled. The executed program is always the same and only one command line argument changes in order to run a different part of the grid. If the .csv file still contains some model specifications that were not run so far, one simply has to restart the job and it will conclude where it left of and not run any specifications again. Also, it is very easy to re-run, delete or alter specifications since the specific row in the file simply has to be changed, deleted or replaced. Thereby, when a specific grid does take more time than anticipated one can split the file and schedule more jobs with the newly created .csv files. Yet another important feature of the "run_grid" program is its exception handling. The training of a neural network is unstable due to the exhaustion of the computational resources such as the main memory or numerical under- or overflow. If an error would occur during the training of a single model this would end the execution of the script. Therefore, exception handling was introduced to the program so that in the case of failed training or evaluation runs of models, the specific error message was written into the .csv file and the script will continue with the execution of the next specification. The intuitive and reliable logic of this program made this extensive grid search possible.

8.4 Gridsearch Results DnCNN

For this analysis 504 models were fitted using the following specifications that took a total training and evaluation time of 530 hours.

Source Code 4: Cast the grid of DnCNNs.

```
depth = [10, 13, 16, 18, 20, 22, 25],
filter_num = [16, 32, 64, 128],
loss_scheme = ['l1', 'l2', 'SSIM'],
skipped_scheme = ['DnCNN', 'ResNet'],
filter_size = [3, 5, 7]
```

Each unique combination of the specified parameters was trained as a separate model with 4000 steps. As discussed in chapter 8.2, this results in vastly different run-times of the models. The smaller model using a depth of 10 and 16 filters will be magnitudes

faster than a model with 25 layers and a respective filter number of 128. A histogram of the run-times can be seen in figure 24.

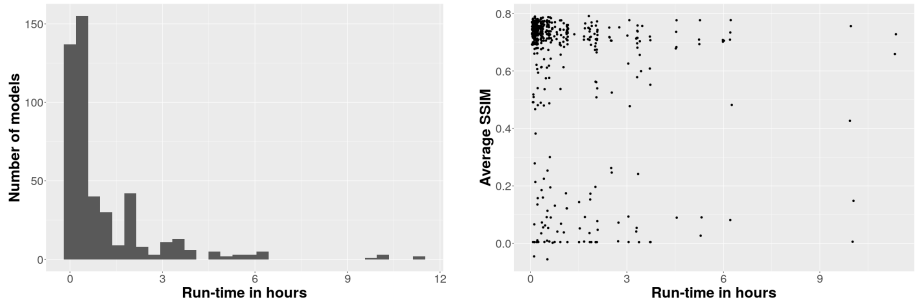


Figure 24: (left) Histogram of model Run-times, (right) Performance vs. Run-time.

While the majority of models took less than an hour to train there are some extreme outliers where models took up to twelve hours. This was anticipated as the number of layers and the number of filters does increase the number of computational operations drastically.

Nevertheless, one cannot conclude that the model with longer training time did necessarily perform better as larger models potentially need more data samples to converge. Looking at figure 24 it can be seen that the average performance of the models as measured by the M-SSIM does not increase with run-time. Especially models that took more than nine hours to complete seem to have a worse performance which is anticipated for the largest models.

Interestingly, some of the models did not converge at all. These models are referred to as failed models and are defined by having an M-SSIM below 0.5 or an M-MAE of above 1. In figure 25 a histogram of all model performances measured by the M-SSIM on the test dataset is shown. The vertical line indicates the benchmark, which is the M-SSIM between the fully- and undersampled images on the test dataset. A higher M-SSIM indicates that the model improved the quality of the undersampled images and made it more similar to the fully sampled images. A value below the threshold indicates a further deterioration of the images. The majority of the successful models are able to improve the quality of the images. This is a good result as the process of recreating the noisy image is already a considerable part of the learning process of convolutional neural networks. Additionally, one can clearly see the separation between the models that seem to perform with an M-SSIM above 0.5 and those with an average below 0.5. A M-SSIM below 0.5 indicates that the model was not able to reconstruct the noisy image and especially the stark contrast to the performing models indicates a failed training process. The artistic output of such a model in figure 25 shows that this model did not learn anything useful.

The first part of the analysis is a systematic comparison of failed and successful models. About 15 percent of all models failed. Firstly, the binary variable Model_failed was created that is 1 for failed and 0 for successful models. Then the different parameters were regressed on that variable. In figure 26b the results of

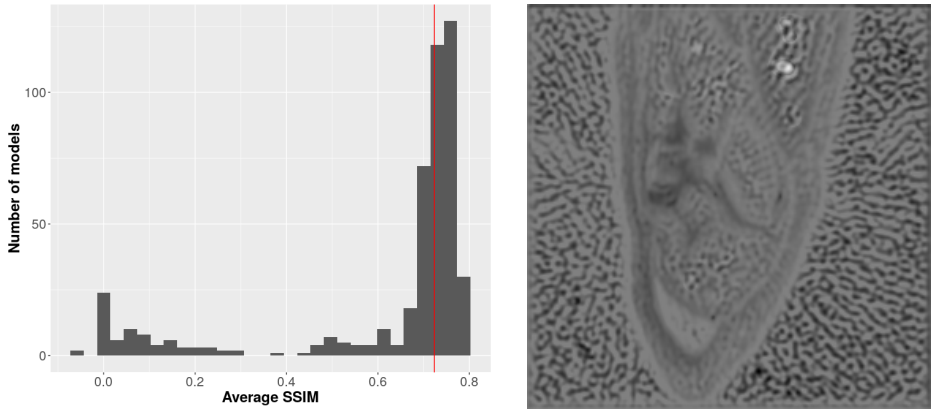


Figure 25: (left) Histogram of the model performances. (right) Prediction of a failed model.

the regression are shown. It can be seen that the number of filters, $\mathcal{L}^{SSIM}$ as loss function and Res_Net skipped_scheme are significant. The significances should be interpreted carefully as some underlying assumptions might be violated.

Nevertheless, the number of filters seems to increase the probability of failure as does the use of the ResNet skipped_scheme. But the most important covariate is the use of $\mathcal{L}^{SSIM}$ as it increases the probability of failure by roughly 50 percent. In the practical statistical training there was some indication that training based on the $\mathcal{L}^{SSIM}$ is more volatile than using the $\mathcal{L}^{l1}$ or $\mathcal{L}^{l2}$ loss function. This suspicion is validated by these results as all of the 79 models that failed did use $\mathcal{L}^{SSIM}$.

It is a major benefit of the analysis that due to the large number of fitted models this result from the practical statistical training is strengthened. The effect size of the significant variables from the regression on the probability of model failure can be seen in figure 27. It shows an interaction between the ResNet skipped_scheme and $\mathcal{L}^{SSIM}$ as 50 out of 79 models that failed also used ResNet skipped_scheme.

Although filter size is not significant in the regression, looking at figure 27 it seems that a higher number of filters increases the risk of model failure. As larger filters come with more trainable parameters it is intuitive that the learning process is more volatile. For the number of filters the same intuition holds and both the significance in the regression and a clear visual result indicate that more filters per layer come with a higher chance of model failure. These results also indicate that the failure of models is not entirely random but is related to the complexity of the model. Overall there is significant evidence that $\mathcal{L}^{SSIM}$ does make the learning phase of the models more unstable. Especially in combination with larger or more complicated models using ResNet skipped scheme this effect is stronger.

To analyze the reasons of model failure further, the loss curve of a failed model is compared to a successful one in figure 28. The successful model converges fast and maximizes the SSIM. The unsuccessful model is not able to make any significant advances towards maximizing the SSIM especially when the scale of the plot is taken

	Dependent variable:
	Average SSIM
depth	0.0001
	(0.001)
filter_size	-0.009^{***}
	(0.002)
filter_num	-0.0002^{***}
	(0.0001)
loss_scheme_l2	0.007
	(0.006)
loss_scheme_SSIM	0.015^{*}
	(0.008)
skipped_scheme_ResNet	0.023^{***}
	(0.006)
Constant	0.757^{***}
	(0.015)
Observations	399
R^2	0.120
Adjusted R^2	0.106
Residual Std. Error	0.056 (df = 392)
F Statistic	8.899^{***} (df = 6; 392)
Note:	*p<0.1; **p<0.05; ***p<0.01

	Dependent variable:
	Failed_model
depth	0.004
	(0.003)
filter_size	−0.008
	(0.007)
filter_num	0.003^{***}
	(0.0003)
loss_scheme_l2	0.012
	(0.030)
loss_scheme_SSIM	0.513^{***}
	(0.030)
skipped_scheme_ResNet	0.111^{***}
	(0.024)
Constant	-0.240^{***}
	(0.068)
Observations	465
R^2	0.515
Adjusted R^2	0.509
Residual Std. Error	0.263 (df = 458)
F Statistic	81.214^{***} (df = 6; 458)
Note:	*p<0.1; **p<0.05; ***p<0.01

Figure 26: (left) Regression of the hyperparameters on the model performance. (right) Regression of the hyperparameters on the probability of model failure.

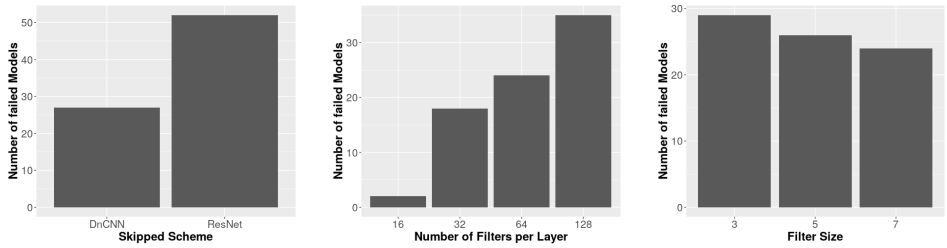


Figure 27: From left to right: Skipped scheme, number of filters per layer and filter size vs. model failure.

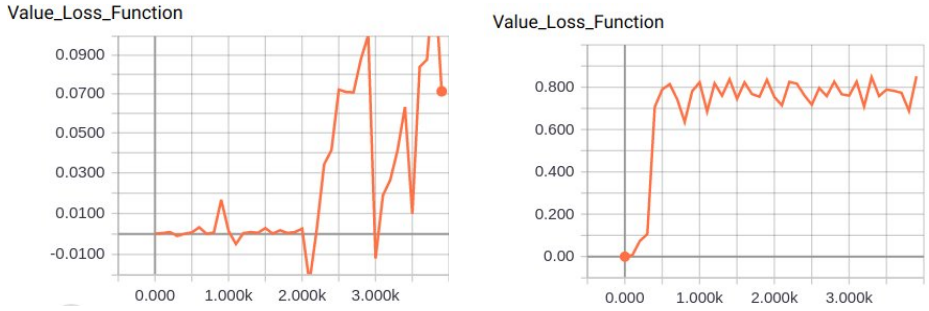


Figure 28: Comparing the loss function of a successful with an unsuccessful model during training.

into account. Unfortunately, it is unclear where the vast difference in performance stems from especially considering that the batches and the initialization of the models are the same.

Only the models that converged and have an M-SSIM above 0.5 are used in further analysis. In figure 26b the regression of the parameters on the M-SSIM can be seen. Additionally, in figure 29 the M-SSIM is shown for the different model specifications.

First, the regression results and visual analysis will be used to examine the different model structures regarding their performances. Using a regression assumes inherently that the different parameters do not interact and have a linear influence on the performance. As highly nonlinear effects of the parameters are likely and complicated interactions of the different parameter settings are possible this can only be seen as a first indication. Therefore, later on, the best fifteen model specifications are investigated to find more interesting results concerning the optimal parameter choice for the DnCNN.

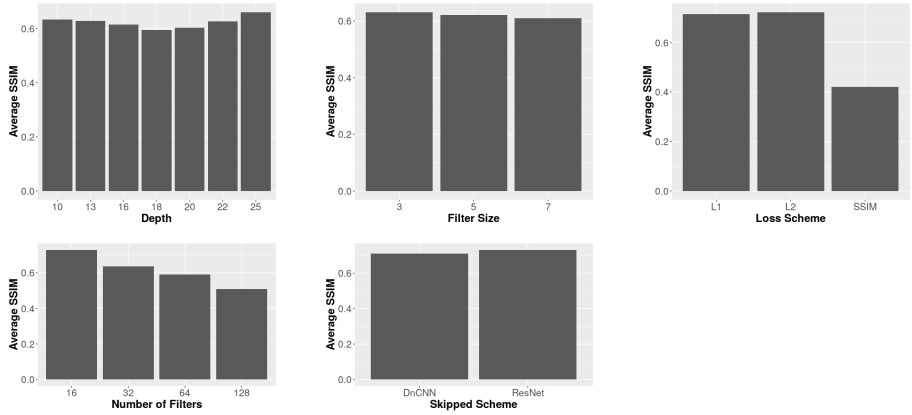


Figure 29: Comparing the model performances for all parameters.

Interestingly, the depth of the model does not have a significant influence on performance. The initial hypothesis was that larger models would either perform worse as they need more data to train effectively or better as they can adapt to the complicated noise structure. This result does not strengthen either initial hypothesis. No conclusion can be drawn from the results of the grid search concerning the model depth. The plot in figure 29 suggests that there is a U-shape effect of the depth on the model performance. But there is no intuitive reason why only particularly hollow or deep models would perform well. Therefore, it is most likely caused by interactions between the depth parameter with the filter sizes and filter numbers. Potentially, deep models using a small number of 3x3 filters and shallow models using a large number of bigger filters perform best.

The filter size 3x3, 5x5 and 7x7 were tested and the regression results as well as the visual analysis indicate that a higher filter size than 3x3 is not necessary. It increases the training time drastically but does not seem to improve the quality of the predictions.

The number of filters per layer could either have a positive effect as it increases the number of feature detectors or a negative effect as it introduces more trainable parameters to the model. The regression results in figure 26a indicate a negative effect on the average performance. This result is confirmed by the visual analysis in figure 29 as with each increase of the number of filters also the average performance declines. Nevertheless, one should not overestimate the positive effect of a small filter size as for the analysis the models had only 4000 steps to train. Models using more filters might need more data samples to converge which could drive the detected effect.

The practical statistical training indicated that while $\mathcal{L}^{SSIM}$ is a viable alternative to the other loss functions it is also more unstable during training. Having a small, positive coefficient in the regression results also indicates that a model, successfully trained on the $\mathcal{L}^{SSIM}$, might even surpass the other models. This introduces a trade-off between stability and model performance as only the rare successfully trained performed well.

Lastly, the ResNet skipped scheme performs slightly better than the DnCNN architecture using only a single skipped connection. This is very interesting as the DnCNN was specifically designed for the task of denoising. The authors did explicitly decide against the use of classical skipped connections as explained in chapter 3.1. This decision is justified in the case of additive noise as the residual image just contains the noise itself. Since this does not hold for MR images, the ResNet skipped scheme may be a viable alternative.

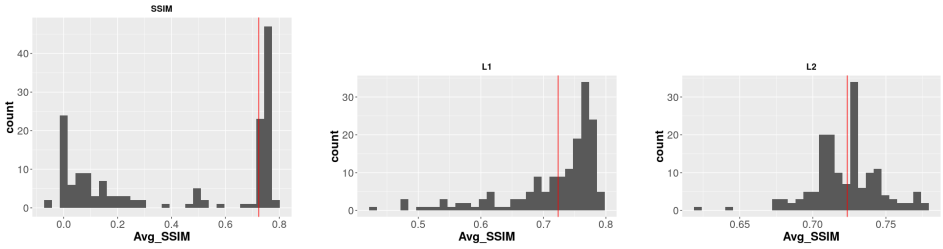


Figure 30: From left to right: M-SSIM of models using the $\mathcal{L}^{SSIM}$, $\mathcal{L}^{l1}$ and $\mathcal{L}^{l2}$ as loss function.

In figure 30 the histogram of the M-SSIM is split in the different loss functions. Here, all models, including the failed ones, were used. The best loss function seems to be the $\mathcal{L}^{l1}$ function, where 65 percent of models break the threshold. But it also has a significant left tail of unsuccessful models. This indicates a better performance that comes with a higher variance. The models trained on the $\mathcal{L}^{l2}$ loss function seem to be centered around the threshold and only very few models performed badly. Nevertheless, only 50 percent of the models beat the threshold. For the $\mathcal{L}^{SSIM}$ the performance of the models seems to be binary. Either the models train successfully and beat the threshold or were not able to train at all. There are almost no models close to the left of the threshold. This once more confirms that if the training based on the $\mathcal{L}^{SSIM}$ is successful it creates very good models. Overall the $\mathcal{L}^{l1}$ seems to be the

best choice as it creates the best models and also is reasonably stable. Nevertheless, if stability is the main concern $\mathcal{L}^{l2}$ should be used. The $\mathcal{L}^{SSIM}$ is very unstable and should only be used if many models can be fitted or if the architecture is specially created to ensure stability. Further research on the inherent stability of the $\mathcal{L}^{SSIM}$ and its causes would be interesting.

Surprisingly for the filter size, depth and number of filters no clear picture emerges. Both, very deep and very flat models, did perform well. For the depth, the filter size and the number of filters all tested parameter values are within the 15 best models. This is particularly interesting as the regression analysis indicated that a higher filter size results in worse models. There seems to be a trade-off in model complexity especially between the filer size and the number of filters. Models using 7x7 filters only use 64 or less filters when performing very well. All models using the highest number of filters use filters of size 3x3. All models using 5x5 filters use less than 64 filters. This indicates that a higher filter number can also be stituted by a smaller filter size vice versa.

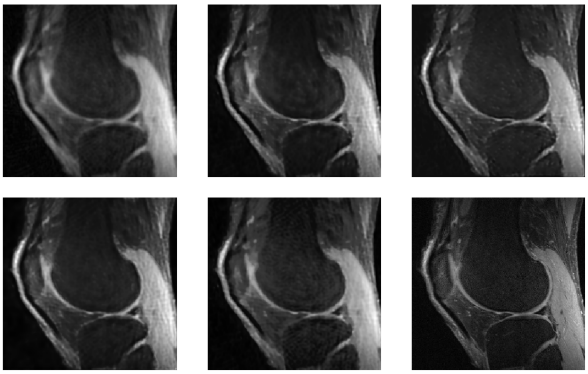


Figure 31: From upper left to lower right: undersampled picture, the best four models in correct order, the fully sampled picture.

Finally, the predictions of the top five models are compared visually. In figure 31 these can be seen. First of all, the models are able to reconstruct the noisy image. Additionally, all of them reduce some of the artifacts and the blurriness. Also, each model improves some aspects like the grey tone better then others. But it is hard to find some general patterns related to their architecture. Nevertheless, all models improve the MR images and are good candidates for further training.

8.5 Discussion

A potential concern of the grid search is that due to the small number of steps no significant results can be drawn at all. Fortunately, some interesting insights were found. Nevertheless, the small number of steps is the major limitation of this analysis and may bias the results towards small and less complicated models. Particularly striking was that the evidence towards the depth was not conclusive. Considering the

regression analysis, the visual inspection and the top 15 models do not give a clear recommendation on which depth to use. Additionally, the narrow range of optimal depths for the DnCNN that the authors of Zhang et al. (2016) recommend cannot be confirmed by this analysis.

Furthermore, the skipped connection scheme of the DnCNN can not be confirmed as the ResNet scheme does perform significantly better. This could be a result of the bias towards smaller and less complicated models. But if the ResNet skipped scheme does also perform better on longer trained models, this could benefit the models from the practical statistical training.

The evidence for the $\mathcal{L}^{SSIM}$ is conclusive and supports the suspicion that emerged during the practical statistical training. Training based on the $\mathcal{L}^{SSIM}$ leads to some of the best models, but the training is inherently unstable and does fail often.

Based on the results, an optimal model would use the $\mathcal{L}^{l1}$ loss function alongside the Resnet skipped scheme. As filter size, the classical 3x3 filers should be used as they perform best and are used in the majority of the top-performing models. The optimal selection for both the number of filters and depth of the model is more complicated. As a filter size of 3x3 usually works well with 64 or more filters one should use either 128 or 64 filters. For the depth, the narrow range of optimal parameters that the authors of Zhang et al. (2016) suggest cannot be validated. As the depth of the best models varies widely, no final recommendation can be made. But as deeper models could perform better with more training time one should choose a depth beyond 20.

9 Conclusion

As mentioned in the previous discussion, the key challenge of this analysis is the training data. As it is confined to high resolution knees images of a limited number of patients, the observations are highly spatially correlated. This could induce overfitting. In addition, the artificial, emulated undersampling may not resemble real-world artefacts sufficiently well, as the sampling scheme is not the single source of error. For these reasons the trained models could perform worse in practice. Beyond these limitations, the analysis shows that Convolutional Neural Networks can be used with great success to improve MR images. Surprisingly, the coil information-based models resulted in worse reconstruction performance compared to the image-based models. As the coil information is comprised of multiple detectors' data, it should contain more detail than the single image-space data. In the latter, the correlation across slices is not considered. The lack of performance improvement based on the coil information indicates that more specialised architectures are needed to exploit the complex data structure.

This analysis explores two convolutional denoising architectures. It re-purposes the U-Net (Ronneberger et al. 2015), an autoencoder originally developed for image segmentation, and employs the DnCNN (Zhang et al. 2016), a neural network developed for image denoising. The learning objective is to reconstruct precisely and produce visually appealing images. Therefore the $\mathcal{L}^{l1}$, $\mathcal{L}^{l2}$, $\mathcal{L}^{SSIM}$, $\mathcal{L}^{MS\text{-}SSIM}$ and $\mathcal{L}^{MS\text{-}SSIM\text{-}Gl1}$ functions were considered as learning objectives. As all of these functions penalize for different visual aspects, we propose, they should be employed

jointly for model evaluation. Based on these criteria as well as visual comparison, the autoencoder performs best. Therefore, it is suited best to denoise MRI images based on the image-space. Using $\mathcal{L}^{SSIM}$ and its descendants seem to be promising as they resemble human perception of differences in images. To the extent of our analysis, no visual differences in terms of reconstruction error could be found in comparison to models using the $\mathcal{L}^{l1}$ or $\mathcal{L}^{l2}$ objectives.

The hyperparameters spaces of the two architectures are explored more rigorously in two grid searches. Due to computational restrictions, not all possible parameter configurations could be tested or succeeded. In particular, the depth of the models and the number of training steps were severely limited. The successive models from the grid search of the U-Net reveal, that irrespective of size, depth and objective, the residual learning strategy regularises the learning task very early on. The adjustment of the luminescence bias introduced by the radial undersampling across nearly all successive models in the benchmark comparisons at the very early stage of training is strong evidence of this. Considering the benchmarks of all applied measures jointly, it is apparent that despite luminescence little is learned. The grid, laid out extensively, and the analysis with regard to the all discussed objectives would benefit greatly from a sufficient increase in training. In the grid search of the DnCNN, skip connections were found to be beneficial to the model. Interestingly, the depth of the model was not found to be significant for the model's performance. Although this could be caused by the low number of training steps, it contradicts the narrow ranges of depth proposed by the authors. According to general usage, a filter size of 3x3 was found to be the best performing. Optimally, one should use 64 or 128 filters. Finally, the best performing models were trained on $\mathcal{L}^{SSIM}$. Unfortunately, this also coincides with a higher variance in model performances so that using the $\mathcal{L}^{l1}$ might be beneficial if training stability is an important issue.

Based on these promising results, further research should investigate the following concerns and ideas: First, guarding against overfitting and to underline the capability of learning the sampling scheme specific artifact structure, our results should be replicated on more heterogeneous and real-world datasets. Second, the information contained in the coils should be exploited extensively, as theory suggests superior performance. Third, the usage of more complex loss functions such as $\mathcal{L}^{SSIM}$ should be explored more rigorously as they are intended to mimic human perception. The lack of empirical evidence in favour of these objectives might be caused by additional training complexity not yet accounted for. Finally, this analysis found transfer learning between different noise structures and levels to be successful. Assuming some underlying similarities between distortions, particularly on different levels of distortions within the same sampling scheme, transfer learning may prove this approach widely applicable and ease computational burdens.

References

Boyer, C., Chauffert, N., Ciuciu, P., Kahn, J., & Weiss, P. 2016, SIAM Journal on Imaging Sciences, 9, 2039

Eo, T., Jun, Y., Kim, T., et al. 2018, Magnetic Resonance in Medicine, 80

Flögel. 2019, *Kernspinresonanz am Institut für Molekulare Kardiologie*, `http://www.nmr.uni-duesseldorf.de/sets/theorie.html` [Accessed: 10.03.2019]

Gallagher, Thomas, Nemeth, Hacein-Bey, & Lotfi. 2008, American journal of roentgenology, 190, 1396

Galteri, L., Seidenari, L., Bertini, M., & Del Bimbo, A. 2017, in The IEEE International Conference on Computer Vision (ICCV)

Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (MIT Press), `http://www.deeplearningbook.org`

He, K., Zhang, X., Ren, S., & Sun, J. 2015, 7

He, K., Zhang, X., Ren, S., & Sun, J. 2016, in Proceedings of the IEEE conference on computer vision and pattern recognition, 770–778

Hornik, K. 1991, Neural networks, 4, 251

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. 2017, in Proceedings of the IEEE conference on computer vision and pattern recognition, 4700–4708

Ioffe, S. & Szegedy, C. 2015, arXiv preprint arXiv:1502.03167

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. 2017, in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Jordan, J. 2018, *Convolutional Neural Networks*, `https://www.jeremyjordan.me/convolutional-neural-networks/` [Accessed: 01.03.2018]

Kafunah, J. 2019, Backpropagation in Convolutional Neural Networks, `https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-\networks/` [Accessed: 10.03.2019]

Kingma, D. P. & Ba, J. 2014, Adam: A Method for Stochastic Optimization

Lee, D., Yoo, J., & Ye, J. C. 2017

Michael Lustig, S. V. 2018, *MRI Data*, `http://mridata.org` [Accessed: 31.10.2018]

Rai, S. 2019, Forward and Backpropagation in Convolutional Neural Network, `https://medium.com/@2017csm1006/forward-and-backpropagation-in-convolutional\-neural-network-4dfa96d7b37e` [Accessed: 10.03.2019]

Ronneberger, O., Fischer, P., & Brox, T. 2015, in , 234–241

Ruhkopf. 2019, GridMRI, `https://github.com/TiStat/GridMRI` [Commit: 0a5efe719d3fafcb8d74d0af26b47d5c9f84e95f]

Timofte, R., Gu, S., Wu, J., & Van Gool, L. 2018, in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops

Toebrock, T. 2019, DnCNN_Hyper_Params, `https://github.com/titoeb/DnCNN_Hyper_Params` [Commit: f84d4eccac9a008a002be80d262ad249998ed179]

Toebrock, T. & Ruhkopf, T. 2019, deepMRI, `https://github.com/titoeb/DeepMRI` [Commit: 972ecb76f9586b0b6c93e50695b10c2eb997e638]

Uecker, Ong, Tamir, et al. 2019, *Berkeley Advanced Reconstruction Toolbox.*, `https://github.com/mrirecon/bart` [Commit: 97d2ec7090c50926dfe8ef76b5caae71fa9fd232]

Wang, S., Su, Z., Ying, L., et al. 2016, 514–517

Wang, Z., Bovik, A. C., Sheikh, H. R., Simoncelli, E. P., et al. 2004, IEEE transactions on image processing, 13, 600

Wang, Z., Simoncelli, E. P., & Bovik, A. C. 2003, in The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003, Vol. 2, Ieee, 1398–1402

Yap, M. H., Pons, G., Marti, J., et al. 2017, IEEE Journal of Biomedical and Health

Informatics, PP, 1
Zhang, K., Zuo, W., Chen, Y., Meng, D., & Zhang, L. 2016, IEEE Transactions on Image Processing, PP
Zhang, K., Zuo, W., Gu, S., & Zhang, L. 2017, in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)
Zhao, Hang, Gallo, et al. 2017, IEEE Transactions on Computational Imaging, 3, 47

Music Genre Classification using Artificial Neural Networks

A. Buchmüller[1] **and C. Gerloff**[2]

[1]Georg-August-Universität Göttingen, Germany

[2]Chair of Statistics, Germany

Abstract. Music genre recognition is a promising field of research in the area of music information retrieval (MIR). Genre classifiers have many real-world applications, e.g. as a way to automatically tag large data sets suited as inputs to recommender systems.
In this paper, we propose a way to sample song data with the Spotify API and create a music genre classifier using artificial neural networks. We compare different feature sets to each other and evaluate their performance and accuracy using confusion matrices and more sophisticated metrics like F_1 scores. We show that convolutional neural networks using timbre values perform well on this task and also propose ways to handle class imbalance.

1 Introduction

The analysis and classification of sound signals have become of increasing importance, as not only Amazons Alexa or Windows Cortana show how it can be used to create a verbal interface but also music streaming services rely on sound analysis in order to recommend their users music based on their listening history. These technological improvements are especially advantageous for people with impaired senses such as blindness or deafness. In both cases, speech and sound recognition can assist those people in communicating with others. Apart from those applications, sound analysis can be used to enhance the users' experience. Genre classification is used for recommendation, as genres are a simple way to find similar music, which can then be suggested to individuals. With the onset of music streaming services, this task seems more relevant than ever. However, due to the characteristics of audio data, retrieving information from sound signals and classifying them is a non-trivial task. Recent efforts in the area of music genre classification have been fueled by the availability of large data sets for analysis. Due to copyright constraints however, source audio files cannot be freely distributed. Large pre-sampled data sets like the *Million Song Dataset* (Bertin-Mahieux et al. 2011) or the *Free Music Archive* (Defferrard et al. 2017) have existed for years now but have obvious drawbacks such as lack of sampling freedom, genre choice or the ability to generalize results onto commercial music.

In music genre classification two components play a major role. Feature extraction, which is the art of finding a suitable representation for music samples, as well as the

choice of a classifier. Efforts led by the Laboratory for Recognition and Organisation of Speech and Audio (LabROSA) [9] have made feature extraction of musical data accessible for researchers worldwide. Their python package `librosa` (McFee et al. 2015), is equipped with functions specifically tailored to extract features of musical data. For classification tasks, there have been a plethora of supervised and unsupervised machine learning algorithms suitable for large data sets like the k-nearest neighbors algorithm or support vector machines, both of which have been a popular choice for a while now. Recently, advances in neural networks and deep learning research demonstrated the power of convolutional neural networks at image recognition tasks, outperforming even the best machine learning algorithms. In the past, researchers like Li et al. (2010) have successfully demonstrated that music genre classification may be approached as an image recognition task by feeding spectrograms into convolutional neural networks.

In this paper, we are proposing a way to create a genre classifier on the basis of spectrogram-like features, namely pitch and timbre, of the Echo Nest application from the popular music streaming service Spotify. The Echo Nest is a music intelligence and data platform owned by Spotify, describing itself as the "Industry's leading music intelligence company, providing developers with the deepest understanding of music content" (The Echo Nest 2020). It is responsible for Spotify's curated personalized music recommendations. Both timbre and pitch are used individually as well as in conjunction within convolutional neural networks and then evaluated in terms of performance and different accuracy measures, such as confusion matrices and F_1 scores in balanced and unbalanced samples.

2 Methodology

2.1 Data Acquisition

There are numerous methods to obtain musical datasets, differing by the features that can be extracted. One simple method to create a dataset is extracting features from one's personal collection. The `librosa` package (McFee et al. 2015) provides a framework to extract meta and audio features of sound signals and in particular music. It works with most of the common sound data formats. This includes waveplots and spectrograms allowing to exactly analyse audio signals. This method however is limited by the size and diversity of your collection and the lack of a target label. A music recommender for example would need some kind of information, indicating what a user wants to hear (target). This could be achieved by giving songs your own review on a scale (very labor intensive) or extracting user reviews from external data bases. For our genre classifier, we need songs that are already labeled with their corresponding genres.

2.2 Spotify API

Spotify has one of the largest collections of music in history. This abundance of songs comes with the added cost of exact measures, as Spotify is careful concerning what

[9] https://labrosa.ee.columbia.edu/

parts of the songs are accessible. The *Spotify Web API* gives access to the Spotify library of song data. The `spotipy` package allows us to use the *Spotify Web API* with Python. Before this, a developer account and project have to be created on the *Spotify Dashboard*, where the `client_id` and the `client_secret` can be copied and then used in Python.

Spotify has grouped its songs within playlists, that are created by users or Spotify itself. This is especially useful when searching for songs of a specific genre, as for a multitude of them, numerous playlists exist and are relatively easy to extract. Unfortunately, the playlists and their contents are changed weekly, with varying degrees. This results in code that has to be adjusted from time to time in order to get working playlists. This potentially complicates replication of results.

`Spotipy` includes functions as `audio_features` and `audio_analysis`, that can pull meta and song data, provided by Echo Nest (Jehan T. 2014) from Spotify tracks. The former of these functions takes the `track_id` or `playlist_id` and returns features describing the music in high-level metadata, like danceability or energy in values, ranging from $[0, 1]$.

The `audio_analysis` function returns a number of features describing the song. Songs are divided into individual segments, which are varying in range and are generally shorter than one second. For each of those segments, a timbre or pitch vector is given, describing the fraction of the song. For a more detailed description of what a segment is refer to the Echo Nest documentation (The Echo Nest 2020).

2.3 Million Songs Dataset

In addition to the data Spotify provides, datasets incorporating Echo Nest metadata as well as audio features taken with `librosa` already exist. The *Million Songs Dataset* (Bertin-Mahieux et al. 2011) is a freely available collection of audio features and metadata for one million songs. It is also possible to extract snippets of audio samples as files, not just as descriptive data, which is needed to create spectrograms with `librosa`. The MSD has been widely used and analyzed, therefore it appears not that compelling to build models, as plenty of methods have already been used on the exact same data.

2.4 Free Music Archive

Another readily available collection of audio data is the *Free Music Archive* data set. The FMA (Defferrard et al. 2017) provides up to 917 GB of data containing 106,574 tracks from 16,341 artists and 14,854 albums, arranged in a hierarchical taxonomy of 161 genres. The data set provides full-length(copyright free) and high-quality audio, pre-computed features, together with track- and user-level metadata, tags and free-form texts such as biographies or lyrics. As with the MSD, the FMA is widely used.

2.5 Feature Engineering

There are a number of possible attributes that are extractable from the three aforementioned music collections. The pre-computed dataset of the FMA for example

contains 518 different features. This results in a multitude of possibilities for analysis. We are creating a genre classifier, which can classify music genres from input songs and as such focus on timbral coefficients.

2.6 Genres

The musical genre of a song often is interesting when analysing and working with music. Genres are groups of similar music divided into (for humans) distinguishable classes. Not every song is classifiable into a well-known genre, especially as the boundaries are fuzzy sometimes. For example, the genre term *Rock* invokes different songs for different people, as some use the term interchangeable with metal, punk or just as a general overall term, for music with an electric guitar.

For our project, we chose to use the following genres: Blues, Classical, Country, Hip-Hop, Jazz, Rock, Metal and Pop. For most common genres Spotify provides numerous playlists. One thing to be noted there is that the playlists in the genres are individualised, depending on the country, from which Spotify is accessed. We accessed the collection of songs from Germany, which results in the inclusion of songs of the sub-genre 'Schlager' in the pop-playlists, which are actually more related to german country music than pop music in general.

2.7 Spectrogram

The spectrogram can be seen as a basic tool for sound and speech recognition analysis and can be defined as an intensity plot of the "Short-Time Fourier Transformation" magnitude of an audio wave signal (Smith 2007). A spectrogram is constructed from the time, the frequency in hertz and the amplitude of the frequency at time t of an audio signal. In practice, the spectrogram is corresponding to the matching sound extremely well.

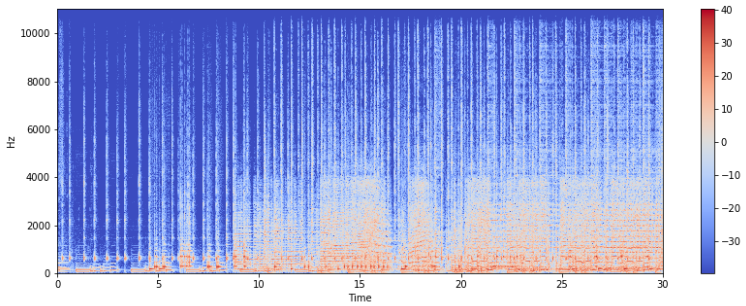


Figure 1: Spectrogram

The spectrogram is then often logarithmized, rendering the plot more interpretable. This log-spectrogram can then be converted onto the Mel-scale and transformed by the Discrete Cosine Transformation (DCT). The Mel-frequency Cepstral Coefficients

(MFCCs) (Logan et al. 2000) are the amplitudes of the resulting plot as seen in Figure 2. Spotify currently does not provide a way to extract or visualize spectrograms of songs in the Spotify library. It is possible to get audio samples, but even those are hard to extract and not that very accessible in practice. On the other hand, Spotify does offer a handful of different features that are closely related to MFCC features.

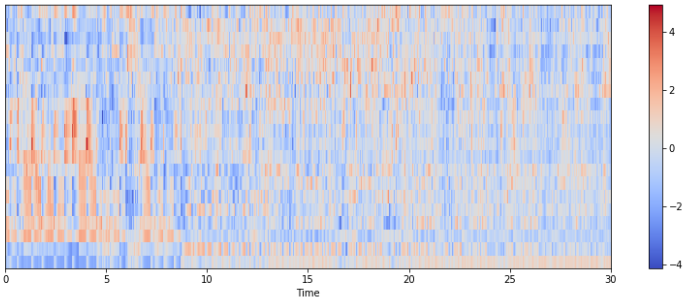


Figure 2: Mel-frequency Cepstral Coefficients (MFCCs)

2.8 Echonest Features

Spotify created high-level variables, which can be extracted with `spotipy`. These variables mainly try to quantify subjective attributes as "danceability", "acousticness" or "speechiness". Numerical values were assigned for these indirectly measurable attributes. Each song is given an average value for each attribute meaning these attributes cannot be analyzed by their variance within each song.

2.9 Timbre

As mentioned before, is not possible to get real spectrograms from the data extracted from Spotify. Instead, the timbre and the pitch can be used, making it possible to use spectrogram-like representations of songs for our model. The `audio_analysis` function gets analysis features from Echo Nest (Jehan T. 2014), of a specific track in the Spotify library. This returns a list of `segments` of differing lengths and quantities for each song. One segment is usually a fraction of a second and contains (besides the start point and duration) information about the loudness, pitch and timbre. For our model, the last two are of interest. The timbre is defined, as the quality of a sound, distinguishing different types of musical instruments or voices (Schindler & Rauber 2012). It is also referred to as tone quality or sound color. Each segment contains a timbre vector of 12 unbound values centered around 0. These values are corresponding to the twelve principal components derived from MFCCs.

$$\text{timbre} = \begin{Bmatrix} PC1 \\ PC2 \\ \vdots \\ PC12 \end{Bmatrix}$$

The timbre vectors in combination with the segment information can be used to get pseudo-spectrograms, having the segments on the x- and the PCs on the y-axis. Higher values of the timbres vectors are represented by a higher contrasting color.

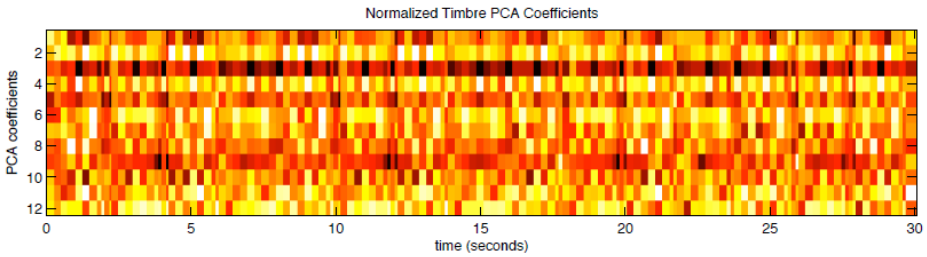


Figure 3: Spectrogram of Timbre averages (Jehan T. 2014)

2.10 Pitch

Another feature divided into segments by the Echo Nest is the pitch. The keys are track-leveled, ranging from 0-11 and are corresponding to the 12 musical keys, C, C#, D, up to B. The value is -1 if no key was detected, the mode equals 0 for a "minor" or 1 for a "major" note.

$$\text{pitch} = \{C \quad C\# \quad D \quad D\# \quad \dots \quad G \quad G\# \quad A \quad A\# \quad B\}$$

This allows for sounds to be scaled in relation to the frequency. It has to be noted, that the major key is more likely to be confused with the minor key three semitones lower, as both keys carry the same pitch. The structure of the pitch is described as a chroma-like vector, in which the values represent relative dominance of every pitch in the chromatic scale. Noise is represented in this by all values of every pitch being near 1, whereas the pure tones are described by one key at value 1 and every other key close to 0. Like the timbre, the pitch can be displayed in a Chroma-like representation, with the musical notes on the y-axis.

2.11 Our Choice

The MSD and FMA are both easier to implement and can contain the actual audio signal, making the deep learning implementation straight forward. The downside of these data sets is their pre-made structures. These two collections are widely used, therefore they have been subject to a lot of analysis. Therefore, we chose the more difficult route and use the Spotify Web API to sample our own data for analysis.

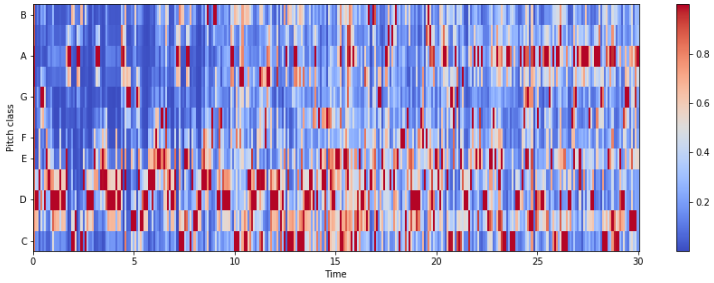


Figure 4: Chroma of the Pitch

This way we do not have access to audio signals itself as with the other two data sets and are limited to a spectrogram-like representation of our data but the data collection is highly flexible.

2.12 Data Pre-processing

Before we go into the architecture of our neural networks it is worth mentioning how we created our input data, i.e. how we sampled from the raw JSON files we created using `spotipy`.

The data pulled from Spotify comes in JSON format, which is very machine learning friendly as Python reads JSON files like hierarchically structured lists. We pulled audio features and analysis data from the 20 most popular playlists of 8 arbitrarily chosen genres resulting in 10115 tracks total. The JSON file comes with various track metadata but for our analysis only two features are of interest: timbre and pitch vectors. The raw Spotify data contains between 42 and 9000 segments per track. A segment is defined as "a set of sound entities (typically under a second) each relatively uniform in timbre and harmony" (Jehan T. 2014). Using the data we pulled from Spotify we created three samples out of which we created 3 input arrays each. One sample was unbalanced, as it contained between 1100 and 1400 songs depending on the genre. This is due to different playlist lengths and leads to class imbalance across train/- validation/- and test sets. Our second sample was balanced by downsampling. It consists of 1100 tracks per genre, that were randomly drawn without replacement if the genre originally had more than 1100 tracks. However, the train/- validation/- and test sets were also drawn randomly leading to a slight imbalance in the validation and test set. Lastly, we created a balanced sample, where we also made sure all sets contain exactly the same amount of genres. We choose an 8:1:1 train/- validation/- test split, which is typically used in machine learning problems (Goodfellow et al. 2016), resulting in an 880 / 110 / 110 split for each genre in our balanced set and slightly different sizes for each unbalanced set. It was interesting to compare the performance of our network across these samples for two reasons. First, in the case of the completely unbalanced sample, splitting the set into our three sets results in approximately proportional genre sizes across all sets i.e., if

Jazz is heavily over represented in the training set, it is also likely over represented in the validation and test set. In the semi-balanced sample, this does not hold true anymore. Because they are both relatively small (remember 880 tracks each means 110 tracks per genre at an 8:1:1 split) and we've drawn randomly, this results in a considerable variance in the validation and test set, as we may have 100 tracks of Jazz tracks versus 120 Rock tracks in the validation or test set. Now, this has an obvious drawback: Sampling this way, we cannot rely on simple accuracy measures like global accuracy on the test set anymore and need to use weighted averages or compare the accuracy for each genre individually. This stems from the fact that it may be that our network performs well on classical music and poorly on blues. If the test set now contains 120 tracks of classical music and only 100 blues songs, one may falsely conclude that the accuracy of the network is higher than it truly is, so well-performing genres might be under or over-represented in the test set used to measure accuracy. Also one must consider that if the train set is imbalanced, better performance of the network in a single genre might stem from the fact that there is more training data for this genre.

From the three samples, we created three different versions of an input array that goes into our models resulting in nine input arrays total. One 3D array of size (batch, segments, 12) for timbre and pitch each, and a 4D array of size (batch, segments, 12, 2) where we combined timbre and pitch vectors into a 12×2 matrix to see if we can increase the accuracy of our networks by adding additional data. Because of the added dimension 1D convolution is not suited for this input so we used 2D convolution layers and 2D max-pooling in models where the input is 4D.

2.13 Model Architecture

All models have been implemented using Keras (Chollet et al. 2015) with the TensorFlow backend (Abadi et al. 2015). Other important packages used in this paper are NumPy (Stéfan van der Walt & Varoquaux 2011), Pandas (Reback et al. 2019), scikit-learn (Grisel et al. 2019), matplotlib (Caswell et al. 2019) and seaborn (Waskom et al. 2018).

For our genre classifier, we experimented with convolutional neural networks. CNN's are widely used in image recognition tasks (Chollet 2018) and since our input data is derived from MFCC-like features it can be expressed as a kind of spectrogram, so using CNN's seemed the natural choice. CNN's have also been successfully demonstrated to be well suited for this task before (Choi et al. 2016).

We tried our architectures on varying levels of sample balance. We first tried our neural networks with imbalanced classes, then used balanced classes with slightly unbalanced train/- validation/- and test sets (i.e. randomly drawn from a balanced sample). Lastly, a sample where we additionally made sure the train/- validation/- and test sets are balanced. As we sampled songs from 8 genres our baseline accuracy was initially set to 12.5% meaning the first goal was to achieve higher accuracy than a random draw. Since this is a very low threshold and our dataset is very similar in size and complexity to the famous GTZAN (Tzanetakis & Cook 2002) data set our next goal was to achieve better accuracy than the authors of GTZAN paper. Its averages range between 44% and 62% overall and 40% to 75 % for each genre with

well established unsupervised machine learning methods like k-nearest neighbors.

Figure 5 shows the architecture of our CNN model without the input layer (starting at the first convolution layer). It consists of six layers in total, including input and output layers. The input layer is a 512 × 12 map, that hosts either 12 MFCC-like timbre values or 12 pitch classes (corresponding to one of the 12 keys C, C#, D, etc.) across 512 segments of one track. It is followed by a 1D convolution layer with 48 filters of size 5 (first layer in Figure 5).

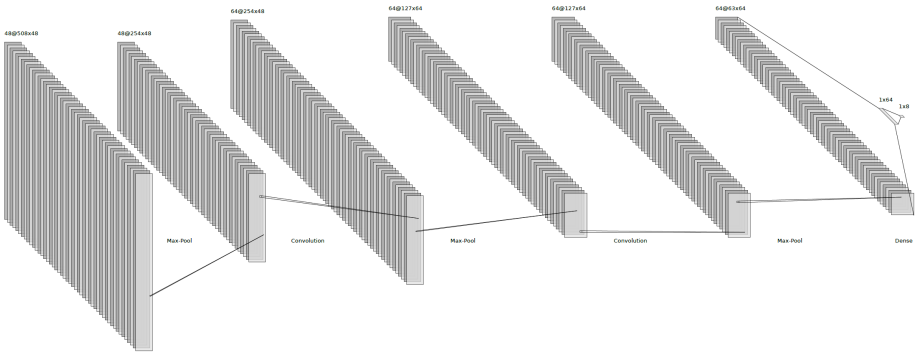


Figure 5: Schematic of our CNN

After each convolution layer, max-pooling and dropout is applied before the results are fed into the next layer. The first convolution layer is followed by two additional convolution layers, each with 64 filters and filter size 2. The output from the last convolution layer is then flattened and fed into a dense layer with 64 perceptrons before going into the output layer with 8 perceptrons - one for each genre. All layers used the ReLU activation function, except the output layer which is softmax (Goodfellow et al. 2016) activated and acts as a classifier. The ReLU (Rectified Linear Unit) activation function introduced by (Hahnloser et al. 2000) is a popular choice. It sets a threshold at 0 meaning it will set the output to 0 if $x < 0$ and produces a linear function with slope 1 when $x > 0$.

$$f(x) = \max(0, x) = \begin{cases} x_i, & \text{if } x_i \geq 0 \\ 0, & \text{if } x_i < 0 \end{cases}$$

The softmax function at the output layer, related to the sigmoid function, is often used in binary classification tasks. In contrast to the sigmoid function, the softmax function is capable of multi-class classification such as music genre classification see (Nwankpa et al. 2018). It computes a probability distribution over a vector of real numbers, i.e., the output and its weights from the last dense layer and outputs values in the range between 0 and 1 for each class, with the sum being equal to 1.

$$P(y = i|\mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_i}}{\sum_{c=1}^{M} e^{\mathbf{x}^\top \mathbf{w}_c}}$$

The predicted class $\hat{y}$ then equals the class with the highest probability.

Since we approached the task of genre classification as a multi-class single label problem (we assume one track can only belong to one genre) our loss is defined by categorical cross entropy. A mathematical definition, where y_0 is the ground truth, p_0 the corresponding prediction and we have M classes is given by:

$$CCE(p, y) = -\sum_{c=1}^{M} y_{o,c} \log\left(p_{o,c}\right) \text{ for } c = 1, \ldots, M$$

In our network however we used a slightly different version called sparse categorical cross entropy. Sparse categorical cross entropy is more memory efficient (especially when dealing with a lot of classes), since it takes a vector of integer target labels instead of a dense matrix of one-hot encoded target labels.

All networks in this paper used the same optimisation technique called Adam. Adam is a stochastic gradient descent type algorithm and an extension to classical stochastic gradient descent. Instead of maintaining a single learning rate for all weight updates, Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients (Kingma & Ba 2014). An explanation of Adam in pseudocode is given below:

Procedure 1 Adam: Adaptive moment estimation. Default parameters are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$

Initialize parameters for stepsize, exponential decay rates for the moment estimates and a small integer to avoid division by zero for use in stochastic objective function

Input: α, $\beta_1, \beta_2 \in [0, 1)$, ϵ

θ_0 *// Initial parameter vector*

$m_0 \leftarrow 0$ *// Initialize 1^{st} moment vector*

$v_0 \leftarrow 0$ *// Initialize 2^{nd} moment vector*

$t \leftarrow 0$ *// Initialize timestep*

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t\left(\theta_{t-1}\right)$ *// Get gradients w.r.t. stochastic objective at timestep t*

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ *// Update biased first moment estimate*

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ *// Update biased second raw moment estimate*

$\widehat{m}_t \leftarrow m_t / \left(1 - \beta_1^t\right)$ *// Compute bias-corrected first moment estimate*

$\hat{v}_t \leftarrow v_t / \left(1 - \beta_2^t\right)$ *// Compute bias-corrected second raw moment estimate*

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ *// Update parameters*

end while

return θ_t *// Resulting parameters*

Compared to stochastic gradient descent it works well without much tuning of hyperparameters and is computationally efficient, making it more suited towards local machines without a dedicated graphics card. We tried different learning rates and found that the default learning rate of $\alpha = 0.001$, as well as default values for $\beta_1 = 0.9$ and $\beta_2 = 0.999$ work well with batch sizes of 32 and 64 respectively.

Dropout rates vary depending on input data. In the case of the fully balanced input array, we found a weak dropout of 0.25 to be the best rate for timbre and pitch input respectively. Lower rates lead to early overfitting, higher rates to a flat learning curve. When using our 4D input array, where we combined timbre and pitch values into a 12 × 2 matrix for each segment, a slightly stronger dropout between 0.3 and 0.4 works well. If the data is unbalanced or semi-balanced, strong dropout rates between 0.3 and 0.5 work best. Table 1 summarizes the complete CNN models we used for timbre and pitch input, respectively.

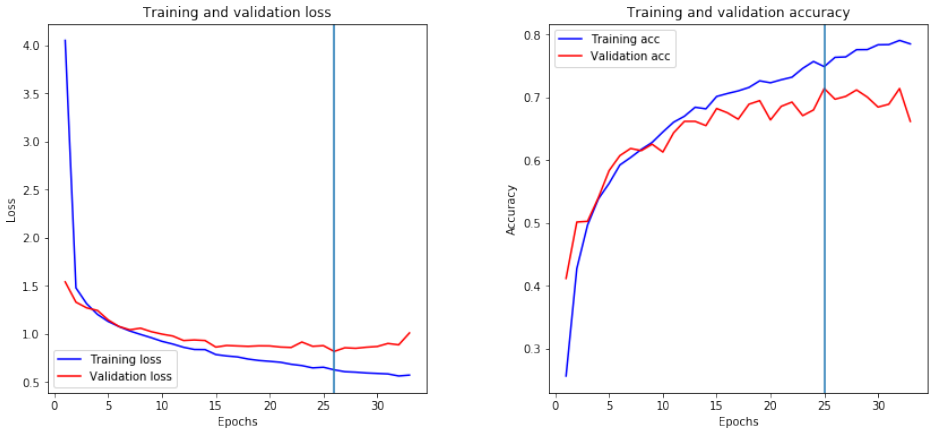
Table 1: Model summary of our CNN's (Timbre/Pitch input)

Layer (type)	Output shape	No. of Param
conv1d_7 (Conv1D)	(None, 508, 48)	2928
max_pooling1d_7 (MaxPooling1D)	(None, 254, 48)	
dropout_9 (Dropout)	(None, 254, 48)	
conv1d_8 (Conv1D)	(None, 254, 64)	15424
max_pooling1d_8 (MaxPooling1D)	(None, 127, 64)	
dropout_10 (Dropout)	(None, 127, 64)	
conv1d_9 (Conv1D)	(None, 127, 64)	20544
max_pooling1d_9 (MaxPooling1D)	(None, 63, 64)	
dropout_11 (Dropout)	(None, 63, 64)	
flatten_3 (Flatten)	(None, 4032)	
dropout_12 (Dropout)	(None, 4032)	
dense_5 (Dense)	(None, 64)	258112
dense_6 (Dense)	(None, 8)	520

The training time and required number of epochs vary considerably across our models. To reduce training time and overfitting we used callbacks in our training process so that the training stops early if the accuracy on the validation set does not increase by at least 1% for 8 successive epochs. Experimenting a bit we found that for some models (especially pitch input models) this threshold is a little bit too weak to prevent overfitting and better suited for timbre models when using the same hyperparameters but still decided to use this threshold for two reasons: First, it makes a comparison between the models easier and more meaningful and secondly overfitting does not worsen validation/test accuracy but underfitting does (a stronger threshold for early stopping leads to underfitting of timbre and mixed models).

Figure 6 shows the training process for our timbre model. We can see first signs of overfitting after epoch 15. From figure 6 (b) we can see our timbre model begins overfitting after epoch 20. The vertical bars show the epoch, where the validation loss reaches its minimum and the accuracy its maximum. The model reaches its highest validation accuracy at epoch 25 and its lowest validation loss at epoch 26. It stops training at epoch 34 due to early stopping. We conclude that, with the hyperparameters we set, our timbre model generalizes poorly after epoch 25 and so training should not continue much further. If we continue training, the validation and test accuracy stays at around 70% while the training accuracy climbs up to over

90% if we train our model long enough. This finding is also confirmed by repeated experiments, where our model always reaches its validation maximum between epoch 19 and 29. As mentioned earlier our pitch model reaches its maximum accuracy a bit earlier (around 5 epochs in repeated measurements) and overfits much quicker. Our mixed model trains a bit slower and reaches its validation maximum generally 5-10 epochs later than the pure timbre model. Training curves for pitch and mixed model can be found in the appendix.



(a) (b)

Figure 6: Training process of our timbre model

3 Results

To evaluate the results of our analysis, we used the following metrics.

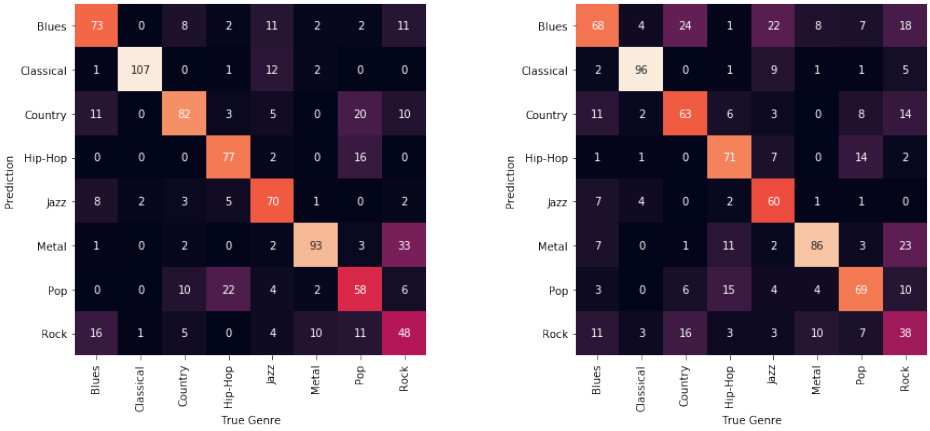
1. Accuracy on a test set.
2. Confusion Matrices. Tables describing classification performance in multi-class problems.
3. Precision, Recall, and F_1 score for each genre. Metrics defined in true and false positives and true and false negatives.

First, we will compare how timbre performed against pitch in the same model. Then we will look at how both compare against a model, where the input array consists of timbre and pitch combined. We did this for all our samples but will focus on our balanced sample since comparison is a easier here. To avoid confusion our CNN with timbre input will be referred to as c1m1, the model with pitch input as c1m2 and our mixed model with timbre and pitch input combined c2m1 (e.g. c1 for 1D convolution, c2 for 2D and m1 timbre input and m2 pitch input). Table 2 displays the accuracy of each model on our validation and test sets.

Model	Validation accuracy	Test accuracy
c1m1 (Timbre)	71.36%	68.18%
c1m2 (Pitch)	65.68%	62.61%
c2m1 (Mixed)	67.95%	67.84%

Table 2: Overall test accuracies for balanced sample models

From the table, we can see that all of these perform similarly well with the timbre model performing best overall. However, from this table, we cannot see which genres cause the models to perform better or worse. To see what makes one model perform better than the other it is worth taking a look at their respective confusion matrices.



(a) c1m1 (Timbre) (b) c1m2 (Pitch)

Figure 7: Confusion matrices for c1m1 and c1m2

Figure 7 shows confusion matrices for timbre and pitch models trained on the balanced sample. Comparing them side by side, we can see that four genres seem particularly difficult to classify: Blues, Country, Pop and Rock. Our timbre model does a better job at most of them (except Pop/Hip-Hop), while the pitch model has the most false positives. This makes sense intuitively since the boundaries between Rock and Country, Hip-Hop and Pop or Blues and Jazz are fuzzy and not easily distinguished even by human listeners. This is confirmed by looking where most false positives are: its exactly these genres where the models misclassify whereas, e.g. classical music, is easily distinguished with almost no false positives by all models.

From figure 8 we see that the addition of pitch does not necessarily increase the accuracy of the model. For most genres, the addition of pitch decreases accuracy except for Hip-Hop and Blues so we can conclude that increasing the dimensionality of a model does not necessarily yield increased accuracy. A reason for why this might be was given earlier. In section 2 we noted that a major key is likely to be confused

with a minor key three semitones lower, as both keys carry the same pitch. This could lead to reduced accuracy in models where pitch was used as input.

To dive even deeper into the results we can look at the classification reports for each model. A classification report mainly consists of three metrics for each genre: Precision, Recall and F_1 score. Before we discuss the results it is important to know what each of these means. Precision is the ability of a classifier not to label an instance positive that is actually negative and described by the ratio of true positives to the sum of true and false positives. Recall is the ability of a classifier to find all positive instances and is defined as the ratio of true positives to the sum of true positives and false negatives. Lastly, the F_1 score is the harmonic mean of precision and recall ranging from 0 (worst) to 1 (best). It is typically lower than global accuracy as it embeds both precision and recalls into calculation thus is more suited to compare classifiers.

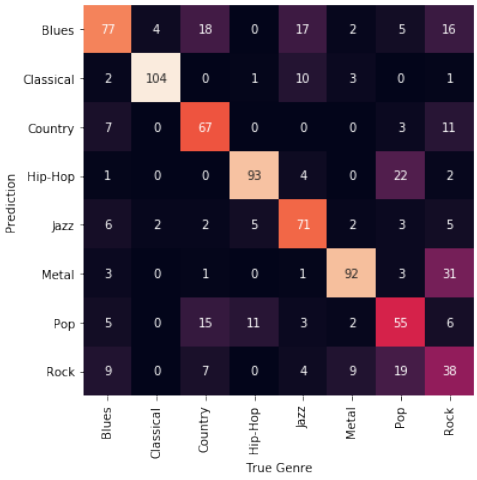


Figure 8: Confusion matrix for c2m1 (Mixed)

Table 3 again shows that the timbre model performs better on almost every genre except the for Pop genre, independent of which metric you choose to compare. The classification table for c1m2 can be found in the appendix (see table 4). It can be seen that our mixed model is somewhat in between our timbre and pitch models w.r.t. F_1 score with the exception of Hip-Hop where it beats both other models. We also see the recall for our timbre model in the Pop genre is particularly low meaning that for all instances that were classified as Pop only 35.5% were actually Pop songs. Considering the state of contemporary commercial music and that Pop was the dominating genre in charts for years now this finding makes sense intuitively since Pop has had a huge influence over all other genres over the last few years such that even though a song is labeled as Hip-Hop it has a spectrogram that looks very much like that of a Pop song. The addition of the pitch truly increases model accuracy here indicating that the PCA components derived from MFCCs are missing information related to the pitch of the song. However we also clearly see that the timbre values capture many features of

Table 3: Classification for reports c1m1 and c1m2

Genre	c1m1 (Timbre)			c1m2 (Pitch)		
	precision	recall	f1-score	precision	recall	f1-score
Blues	0.543	0.691	0.608	0.447	0.618	0.519
Classical	0.860	0.945	0.900	0.835	0.873	0.853
Country	0.573	0.782	0.662	0.589	0.573	0.581
Hip-Hop	0.833	0.727	0.777	0.740	0.645	0.689
Jazz	0.686	0.755	0.719	0.800	0.545	0.649
Metal	0.765	0.800	0.782	0.647	0.782	0.708
Pop	0.684	0.355	0.467	0.622	0.627	0.624
Rock	0.550	0.400	0.463	0.418	0.345	0.378

a song well, since they classify more accurate than the pitch, for most genres even without much hyperparameter tuning or complicated model architectures.

The high false positive rate between Country and Pop music might be due to our sampling process mentioned earlier. Since we sampled from Germany our Pop playlists contain a lot of "Schlager" music, which to non-native speakers can loosely be described as a Country/Pop hybrid genre. For other genres, the fuzzy boundaries have existed forever. Blues and Jazz for example are notoriously difficult to classify as even human listeners might struggle to identify the genre correctly. We were surprised however that our timbre model and even the pitch model did so well in the Metal genre, as the spectrograms of Metal and Rock are closely related to each other as well.

4 Conclusion

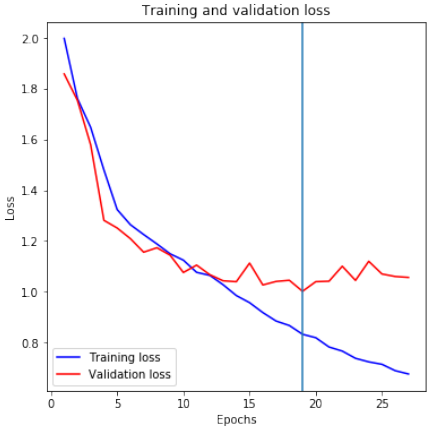
In this research paper we explained how we sampled track data using the Spotify Web API with `spotipy` and how we used this data to create a music genre classifier with artificial neural networks. From the data we pulled we created balanced and unbalanced samples that we used as an input and compared their respective performance and accuracy. We showed that, although not exactly equivalent to spectrograms and chroma plots, timbre and pitch values from the Echo Nest can be used as an input to CNN's in order to classify their genres with high accuracy. Especially timbre values are suited for this task since they take over a considerable part of the feature extraction, which otherwise the neural net would have to do. Although the pitch of a track generally does not perform as well, we found evidence that its addition can be beneficial for the accuracy of certain genres. We also found that when working with imbalanced samples, using higher dropout rates is necessary but also highly effective against overfitting, without impacting the performance significantly, and that these models provide similar accuracies to balanced sample models, although you need to be careful with interpretation if the validation and test sets are imbalanced. Weighted averages can be useful for a meaningful interpretation and comparison of imbalanced samples. With our simplistic CNN model, we achieved a global accuracy

of about 70% on our balanced sample, although the difference between some genres is substantial.

Due to the simplicity of our model we conclude that higher accuracies in the realm of 80% can be achieved with for example a larger sample size, additional convolution layers, sophisticated hyperparameter tuning (e.g. using Gridsearch), different activation functions (e.g. using SReLU instead of ReLU, which was tested on award winning CNN architectures for image recognition tasks like CIFAR-10, ImageNet etc. see Jin et al. (2015)), the addition of more genres to capture hybrid genres better or trying a different architecture like CRNN (Choi et al. 2016) altogether.

A Appendix

A.1 Additional Training Processes of our Models

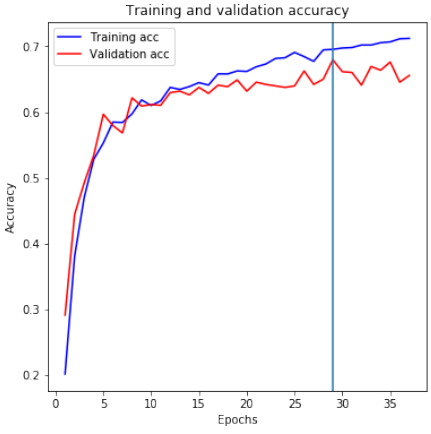


(a)

(b)

Figure 9: Training process of (Pitch model)

(a)



(b)

Figure 10: Training process of c2m1 (Mixed model)

A.2 Additional Classification Results

Table 4: Classification report of c2m1 (Mixed model)

	precision	recall	f1-score	support
Blues	0.554	0.700	0.618	110.000
Classical	0.860	0.945	0.900	110.000
Country	0.761	0.609	0.677	110.000
Hip-Hop	0.762	0.845	0.802	110.000
Jazz	0.740	0.645	0.689	110.000
Metal	0.702	0.836	0.763	110.000
Pop	0.567	0.500	0.531	110.000
Rock	0.442	0.345	0.388	110.000
accuracy	0.678	0.678	0.678	0.678
macro avg	0.673	0.678	0.671	880.000
weighted avg	0.673	0.678	0.671	880.000

References

Abadi, M., Agarwal, A., Barham, P., et al. 2015, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, software available from tensorflow.org

Bertin-Mahieux, T., Ellis, D. P., Whitman, B., & Lamere, P. 2011, in Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)

Caswell, T. A., Droettboom, M., Hunter, J., et al. 2019, matplotlib/matplotlib v3.0.3

Choi, K., Fazekas, G., Sandler, M. B., & Cho, K. 2016, CoRR, abs/1609.04243

Chollet, F. 2018, Deep learning with Python (Shelter Island, New York: Manning Publications Co), oCLC: ocn982650571

Chollet, F. et al. 2015, Keras, `https://keras.io`

Defferrard, M., Benzi, K., Vandergheynst, P., & Bresson, X. 2017, in 18th International Society for Music Information Retrieval Conference

Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (MIT Press), `http://www.deeplearningbook.org`

Grisel, O., Mueller, A., Lars, et al. 2019, scikit-learn/scikit-learn: Scikit-learn 0.21.3

Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., & Seung, H. 2000, Nature, 405, 947

Jehan T., DesRoches, T. J. 2014, Analyzer Documentation, the echonest

Jin, X., Xu, C., Feng, J., et al. 2015, CoRR, abs/1512.07030

Kingma, D. P. & Ba, J. 2014, Adam: A Method for Stochastic Optimization

Li, T., Chan, A., & Chun, A. 2010, Lecture Notes in Engineering and Computer Science, 2180

Logan, B. et al. 2000, in Ismir, Vol. 270, 1–11

McFee, B., Raffel, C., Liang, D., et al. 2015, in Proceedings of the 14th python in science conference, Vol. 8

Nwankpa, C., Ijomah, W., Gachagan, A., & Marshall, S. 2018, CoRR, abs/1811.03378

Reback, J., McKinney, W., den Bossche, J. V., et al. 2019, pandas-dev/pandas: v0.25.2

Schindler, A. & Rauber, A. 2012, in International Workshop on Adaptive Multimedia Retrieval, Springer, 214–227

Smith, J. O. 2007, Mathematics of the Discrete Fourier Transform (DFT) (http://www.w3k.org/books/: W3K Publishing)

Stéfan van der Walt, S. C. C. & Varoquaux, G. 2011, Computing in Science & Engineering, 13, 22

The Echo Nest. 2020, The Echo Nest Company Description

Tzanetakis, G. & Cook, P. 2002, IEEE Transactions on Speech and Audio Processing, 10, 293

Waskom, M., Botvinnik, O., O'Kane, D., et al. 2018, mwaskom/seaborn: v0.9.0 (July 2018)

Artificial intelligence is considered to be one of the most decisive topics in the 21st century. Deep learning algorithms, which are the basis of artificial intelligence applications, are of central interest for researchers but also for students that strive to build up academic knowledge and practical competences in this field.

The Deep Learning Seminar at the University of Göttingen follows the central notion of the Humboldtian model of higher education and offers graduate students of applied statistics the opportunity to conduct their own research. The quality of the results motivated us to publish the most promising seminar papers in this volume. For the selected papers a full peer review process was conducted.

The presented contributions cover a broad range of deep learning topics. The articles in the first part of this volume may serve the reader as introduction to deep learning algorithms. Subsequently, research applications allow the reader to gain deep insights into some of the latest developments in the field of artificial intelligence.

GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

ISBN: 978-3-86395-462-8

Universitätsdrucke Göttingen