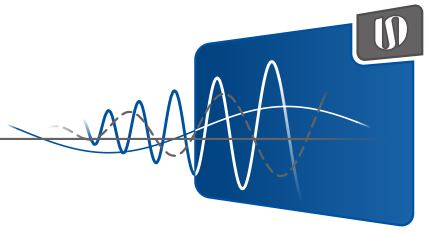
Institut für Statik und Dynamik
der Luft- und Raumfahrtkonstruktionen

Univ.-Prof. Dr.-Ing. Tim Ricken



Master-Thesis

Eigenfrequenzvorhersage für Verdichterschaufeln mit dreidimensionalen faltenden neuronalen Netzwerken

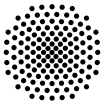
Marius Schwämmle

1. Juni 2022

Bericht Nr. 107

Editor: Univ.-Prof. Dr.-Ing. Tim Ricken
Betreuer: André Mielke (ISD)
Nicolai Forsthofer, Oliver Kunc (DLR e.V.)

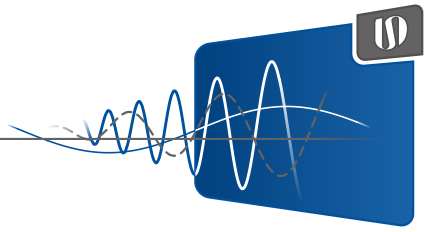
Bericht des Instituts für Statik und Dynamik der Luft- und Raumfahrtkonstruktionen



Universität Stuttgart

Institut für Statik und Dynamik
der Luft- und Raumfahrtkonstruktionen

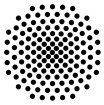
Univ.-Prof. Dr.-Ing. Tim Ricken



Master-Thesis

Eigenfrequenzvorhersage für Verdichterschaufeln mit dreidimensionalen faltenden neuronalen Netzwerken

Autor: Marius Schwämmle
Bericht: Nr. 107
Beginn: 1. Oktober 2021
Abgabedatum: 1. Juni 2022
Editor: Univ.-Prof. Dr.-Ing. Tim Ricken
Betreuer: André Mielke (ISD)
Nicolai Forsthofer, Oliver Kunc (DLR e.V.)
Institut: Institut für Statik und Dynamik
der Luft- und Raumfahrtkonstruktionen
Pfaffenwaldring 27
70569 Stuttgart
Universität Stuttgart
Firma: Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR e.V.)
Bauweisen und Strukturtechnologien (BT-BGF ST)
Pfaffenwaldring 38-40
70569 Stuttgart



Universität Stuttgart

Aufgabenstellung

für Marius Schwämmle

Titel: Eigenfrequenzvorhersage für Verdichterschaufeln mit dreidimensionalen faltenden neuronalen Netzwerken

Thema: Es wird eine Datenbasis parametrisierter Verdichterschaufelgeometrien (Inputs) und zugehöriger Eigenfrequenzen (Outputs) zur Verfügung gestellt. Diese wurde mittels FE-Simulationen erstellt, welche im Vergleich zu ML-Methoden aufwändig sind. Das Ziel ist, ein Ersatzmodell zu erstellen, zu trainieren und zu validieren, um die FE-Simulationen zu ersetzen und basierend auf der Schaufelparametrisierung die Lage der Eigenfrequenzen vorherzusagen. Dieses Ersatzmodell soll im Anschluss an diese Arbeit im Triebwerksvorentwurf am DLR zum Einsatz kommen. Die Arbeit besteht aus folgenden Teilaufgaben:

- Einarbeitung in die Blattparametrisierung und die bereitgestellte Datenbasis
- Literaturrecherche zu vergleichbaren Problemstellungen
- Definition des Softwarekonzeptes
- Verarbeitung der Trainingsdaten
- Training und Validerung des Ersatzmodells
- Dokumentation

Betreuer: André Mielke (ISD)
Nicolai Forsthofer, Oliver Kunc (DLR e.V.)

Editor: Univ.-Prof. Dr.-Ing. Tim Ricken

Beginn: 1. Oktober 2021

Abgabedatum: 1. Juni 2022

Institut: Institut für Statik und Dynamik
der Luft- und Raumfahrtkonstruktionen
Pfaffenwaldring 27
70569 Stuttgart

Firma: Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR e.V.)
Bauweisen und Strukturtechnologien (BT-BGF ST)
Pfaffenwaldring 38-40
70569 Stuttgart

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst habe, dass ich keine anderen als die angegebenen Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist, dass ich die Arbeit weder vollständig noch in Teilen bereits veröffentlicht habe und dass das elektronische Exemplar mit den anderen Exemplaren übereinstimmt.

Ort, Datum Marius Schwämmle

Kurzfassung

Eigenfrequenzvorhersage für Verdichterschaufeln mit dreidimensionalen faltenden neuronalen Netzwerken

Diese Arbeit stellt eine Machbarkeitsstudie des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR e.V.) dar, welche die Möglichkeit aufzeigen soll, klassische Lösungsansätze durch alternative Machine Learning (ML)-Algorithmen zu ersetzten und damit komplexe und zeitaufwendige Simulationen von Triebwerksprozessen zu beschleunigen. Aus diesem Grund wird die Verlässlichkeit von Vorhersage über die Eigenfrequenzen von Verdichterschaufeln unter Verwendung eines ML-Modells untersucht und dient als alternative Berechnungsmöglichkeit, der dynamischen Festigkeits- und Verformungsberechnung einer klassischen numerischen Simulation mittels der Finite Elemente Methode (FEM). Für die Vorhersagen der ersten zehn signifikanten Eigenfrequenzen der Verdichterschaufelblätter einer Variation des Trent 1000 Triebwerks wird ein Deep Neural Network (DNN) bestehend aus einem dreidimensionalen Convolutional Neural Network (CNN) und einem vollständig verknüpften Artificial Neural Network (ANN) verwendet. Die inhomogenen Verteilungen der Eigenfrequenzen und der geometrischen Merkmale der Verdichterschaufelblätter stellen eine große Herausforderung für die Genauigkeit der Vorhersagen des DNN dar. Innerhalb eines umfangreichen Preprocessings des Datensatzes werden die einzelnen Verteilungen durch Normalisierungen optimiert und die darin enthaltenen Ausreißer reduziert. Mittels einer Hyperparameter Optimierung mit dem Open-Source Framework Optuna und dem darin enthaltenen Bayes'schen Optimierungsalgorithmus, wird die beste Modell-Konfiguration des DNN ermittelt. Trotz Fine-Tuning und der optimalen Modell-Konfiguration sowie einer umfangreichen Aufbereitung des Datensatzes können jedoch nur für 82.14% der Testdaten verlässliche Vorhersagen erzeugt werden. 50% der vorhergesagten Eigenfrequenzen besitzen einen geringeren Fehler als 5.53% und zeigen das große Potenzial des DNN-Modells. Eine umfangreiche Analyse der Ausreißer mit großen relativen Fehlern in den Vorhersagen ergibt, dass Verdichterschaufelblätter mit einem Volumen größer als 4000mm^3 und Eigenfrequenzen größer 40000 Hz im zugrunde liegenden Datensatz extrem unterrepräsentiert sind. Über die Schnittmengen der Volumen- und Eigenfrequenzverteilungen der besten Vorhersagen und der Ausreißer werden die unterrepräsentierten geometrische Merkmale als weiter Ursache identifiziert. Abschließend wird die Fähigkeit des DNN dreidimensionale Merkmale in den Verdichterschaufelblättern zu erkennen durch die Analyse der Feature-Maps und Transfer-Learning in ein Logistic-Regression-Modell nachgewiesen.

Stichwörter: FEM, Modalanalyse, Mashinelles Lernen, Falltungsnetzwerke

Abstract

This work represents a feasibility study for the Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR e.V.), which shows the possibility to replace classical solution approaches by alternative ML-algorithms and thus accelerate complex and time-consuming simulations of engine processes. For this reason, the reliability of the predictions of natural frequencies of compressor blades using a ML model is investigated and serves as an alternative computational option for the dynamic strength and deformation calculation of a classical numerical simulation using the FEM. The constructed DNN consists of a three-dimensional CNN and a fully-connected ANN and is used to predict the first ten significant natural frequencies of the compressor blades of a variation of the Trent 1000 engine. The inhomogeneous distributions of the natural frequencies and geometric features of the compressor blades pose a major challenge to the accuracy of the predictions of the DNN. Within an extensive preprocessing of the data set, the individual distributions are optimized by normalizations and the outliers contained therein are reduced. By means of a hyperparameter optimization with the open-source framework Optuna and the Bayesian optimization algorithm contained therein, the best model configuration of the DNN is determined. However, despite fine-tuning and the optimal model configuration as well as extensive preparation of the data set, reliable predictions can only be generated for 82.14% of the test data. 50% of the predicted eigenfrequencies have an error lower than 5.53% and show the great potential of the DNN model. Extensive analysis of the outliers with large relative errors in the predictions reveals that compressor blades with volumes greater than 4000mm^3 and natural frequencies greater than 40000 Hz are extremely underrepresented in the underlying data set. Using the intersections of the volume and natural frequency distributions of the best predictions and the outliers, the underrepresented geometric features are identified as a further cause. In conclusion, the ability of the DNN to detect three-dimensional features in the compressor blades is demonstrated by analyzing the feature maps and transfer learning into a logistic regression model.

Keywords: FEM, Modalanalysis, Mashine Learning, Deeplearning, Convolutional Neural Networks

Inhaltsverzeichnis

Symbole und Abkürzungen

Abkürzungen

Abb. Abbildung xix, 6, 8, 9, 12, 16–22, 24, 28, 31, 32, 34–38, 43, 44, 46–49, 51–53, 56, 57, 62–79, 82–85

ACDC Advanced Compressor Design Code 43, 44

AdaGrad Adaptive Gradient Approach 26

Adam Adaptive Moment Estimation 26, 61, 64, 65, 70

AI Artificial Intelligence xvii, 5, 17, 18

ANN Artificial Neural Network v, vii, 1, 2, 5, 20, 30, 31, 34–36, 91

API Application Programming Interfacen 2, 43

BBX Bounding Box 46–48

BCE Binary Cross Entropy 29

CAD Computer-aided design 1, 2, 14

CNN Convolutional Neural Network v, vii, 1, 2, 5, 20, 34, 35, 37–39, 65, 76, 81–84, 91, 92

DL Deep Learning 5, 17, 18

DLR e.V. Deutsches Zentrum für Luft- und Raumfahrt e.V. 43

DNN Deep Neural Network v, vii, x, xvii, xviii, 1–3, 18, 34, 43, 45, 50–54, 56–59, 61, 62, 64–73, 75–79, 81–85, 87, 89, 91, 92, 97

FDM Finite Differenzen Methode 23, 31, 32

FEM Finite Elemente Methode v, vii, 1, 2, 5, 9, 10, 13, 44

Gl. Gleichung 6, 8, 11–13, 15, 27, 31–33, 37, 38, 41, 52, 53, 62, 68

GPU Graphics Processing Unit 57, 92

GTlab Gas Turbine Laboratory 1, 43–45, 47, 48

HDF Hierarchical Data Format 44

IQR Interquartile Range 67, 72

LE Leading Edge 84

LMS Least Mean Square 27, 28

MAC Multiply–Accumulate 30, 36, 37, 61

ML Machine Learning v, vii, xvii, 1–3, 5, 13, 18–22, 29, 32, 34, 39, 40, 43, 45, 50, 57, 69, 81, 84, 91, 92

MLPs Multi Layer Perceptrons 18

MSE Mean Square Error 29, 62, 85

NN Neural Network 2, 23

NumPy Numerical Python 46, 47

NURBS Non-uniform rational B-Spline 2, 3, 5, 14, 15, 44, 45, 91

OOM Out of Memory 57

RAM Random Access Memory 57

RelU Rectified Linear Unit 34, 58, 61, 64, 66, 68, 77

RGB Rot-Grün-Blau 35, 37

RNN Recurrent Neural Network 34

RPD Relative Percent Difference 48, 49

SelU Scaled Exponential Linear Unit 34, 61, 64, 66, 92

SGD Stochastic Gradient Descent 26

SMBO Sequential Model-Based Global Optimization 41, 63

SSE Sum Square Error 29

SSR Sum Square Regression 29

SST Sum Square Total 29

std::cout standard character output device 47

STL Standard Triangle Language 46, 47

TE Trailing Edge 14

TPE Tree-structured Parzen Estimater 41, 61, 62, 64, 65

VSB Verdichterschaufelblatt xviii, 2, 3, 5, 43–50, 52, 64, 75–77, 79–84, 91

VSBO Verdichterschaufelblattoberfläche x, xvii, 45–49

VSBV Verdichterschaufelblattvolumen 45, 48, 49

VTK Visualization Toolkit 47

Griechische Symbole

Symbol	Einheit	Beschreibung	Seite
α		Learnrate | Learning Rate	24, 39
$\boldsymbol{\delta}$		Fehlervektor in Aktivierungsfunktion	33
δ		Fehler in Aktivierungsfunktion	32, 33
ε		Fehler	40
ϵ		Fehler Limit	25, 30
η		Zeitfunktion | modale Koordinate	6, 8, 9
γ		Teilfaktor	26
δ_{ij}		Kronecker Delta	11
λ		Eigenwert	6
ξ		Dimensionslose Länge	10–12
ω		Eigenkreisfrequenz	6
ω_d		gedämpfte Eigenkreisfrequenz	6
$\boldsymbol{\varphi}$		Eigenvektor | Eigenform	6, 8, 9
ψ		Ansatzfunktion	11, 12
ρ	kg/m^3	Dichte	12
μ	–	Erwartungswert	48, 49
σ	–	Standardabweichung	40, 48, 49
σ		Sigmoid Aktivierungsfunktion	28
ξ		Finiter Nenner Wert	26

Lateinische Symbole

Symbol	Einheit	Beschreibung	Seite
a		Aktivierungsfunktion | Activation Function	19, 30
C		NURBS-Kurven-Funktion	14
d	kg/sec	Dämpfungskonstante	6–9, 12
$\mathbf{D}_e$	kg/sec	Elementdämpfungsmatrix	12, 13
$\mathcal{D}^{(e)}$		Element	10
$\mathbf{D}$	kg/sec	Dämpfungsmatrix	6, 13
E	$\mathrm{N\,m^{-2}}$	Elastizitätsmodul	12
E_k	J	Kinetische Energie	7
E_p	J	Potentielle Energie	7
F_n	N	Schubkraft	43
f	$-\|\mathrm{N}\|\mathrm{Hz}$	Funktion | äußere Kraft | Eigenfrequenz	xix, 8, 12, 51, 52, 72–74
$\mathbf{f}$	N	Äußerer Kraftvektor	8
$\mathbf{f}_e$	N	Elementlastvektor	12, 13
A	$\mathrm{m^2}$	Längenänderung	12
$\mathbf{f}$	N	Lastvektor	6, 13
G		Kumulierter Gradient	26
$\mathcal{G}^{(e)}$		Art der Stetigkeit	14
h		Hypothese | Hypothesis	23, 31
$\mathbf{I}$		Einheitsmatrix	41
IQR		Interquartile Ranges | Wertebereiche zwischen mehrerer erster und dritter Quartile	67, 68
IQR		Interquartile Range | Wertebereich zwischen ersten und dritten Quartil	67
J		Loss-Funktion	23, 24, 27–29, 31
k	$\mathrm{N\,m^{-1}}$	Federsteifigkeit	7, 8, 12
$\mathbf{K}_e$	$\mathrm{N\,m^{-1}}$	Elementsteifigkeitsmatrix	12, 13
$\mathbf{K}$	$\mathrm{N\,m^{-1}}$	Steifigkeitsmatrix	6, 8, 13
$\mathbf{L}_e$		Zuordnungsmatrix	13
l_e	m	Länge Elemente	10, 12
$\mathbf{L_w}$		Lower Whiskers | Untere Grenzen mehrerer Whisker-Fehlerbalkens	67–71, 73–75

Symbol	Einheit	Beschreibung	Seite
L_w		Lower Whisker | Untere Grenze eins Whisker-Fehlerbalkens	67
m	kg	Masse	7, 8, 12
$\mathbf{m}$		Momentum erster Ordnung	27
$\mathbf{M}_e$	kg	Elementmassenmatrix	12, 13
$\mathbf{M}$	kg	Massenmatrix	6, 8, 13
μ		Bypass- oder Nebenstromverhältnis	43
N		B-Spline Basisfunktion	14, 15
$N_{d,e}$		Anzahl Freiheitsgrade pro Element	11–13
N_e		Anzahl Elemente	10, 13
N_n		Anzahl Knoten	10
$N_{\text{vox,bbx}}$		Anzahl Voxel in Bounding Box	48
$N_{\text{vox,VSBO}}$		Anzahl als innerhalb der VSBO erkannter Voxel	48
P		Kontrollpunkt	14, 15
P		Präzision | Precision	30
Φ	N	Schubverhältnis	43
p	N	Äußere Last	12
$q^{(e)}$		Verallgemeinerte Koordinate Element	11, 12
Q_s^{nk}	J	Nicht-konservative Kraftgröße	7
$\mathbf{Q}$		Quartile	67, 68
Q		Quartil	67
R		rationale B-Spline Basisfunktion	14, 15
R		Rückruf | Recall	30
RPD		Relative Prozentuale Differenz	48, 49
S		NURBS-Flächen-Funktion	15
t	sec	Zeit	6, 7, 11, 12
$\mathbf{u}$		Knotenvektor	14, 15
u		Parametrische Richtung	14, 15
u	m	Verschiebung	7, 8, 11–13
$\mathbf{u}$	m	Verschiebungsvektor, Freiheitsgrade	6, 8, 9, 13
$\ddot{u}$	m/sec^2	Beschleunigung	7, 8
$\ddot{\mathbf{u}}$	m/sec^2	Beschleunigungsvektor	6, 8
$\dot{u}$	m/sec	Geschwindigkeit	7, 8
$\dot{\mathbf{u}}$	m/sec	Geschwindigkeitsvektor	6, 8, 9
$\mathbf{U_w}$		Upper Whiskers | Obere Grenzen mehrer Whisker-Fehlerbalkens	67–71, 73–75

Symbol	Einheit	Beschreibung	Seite
U_w		Upper Whisker | Obere Grenze eins Whisker-Fehlerbalkens	67
$\mathbf{v}$		zweiter Knotenvektor	15
$\mathbf{v}$		Momentum zweiter Ordnung	27
v		zweite parametrische Richtung	15
V_{bbx}		Volumen Bounding Box	48
V_{VSB}		Verdichterschaufelblattvolumen	48–50, 52
$\mathbf{W}$		Gewichtmatrix | Weight-Matrix	30, 31, 33, 36
W		Gewichtungsfuntion	12
$\mathbf{w}$		Gewichtvektor | Weight-Vector	23, 24, 26, 27, 31
w		Gewicht | Weight	12, 14, 15, 18, 19, 23, 24, 27, 30, 36
$\mathbf{x}$		Eingangsschicht | Input Layer (Vektor)	23, 27
x		Input | Eingangswert | Koordinate	19, 23, 30, 51, 52
$x^{(e)}$	m	Lokale Koordinate Element	10, 12
y		Tatsächlicher Ausgangswert | Groud Truth	19, 23, 27–30
$\hat{y}$		Vorhersage | Ausgangswert Neuron nach Aktivierung | Output	19, 23, 27–31
$\hat{\mathbf{y}}$		Ausgangsschicht | Output Layer (Vektor)	32
z		Ausgangswert Neuron ohne Aktivierung	19, 23, 28, 30

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Entwicklungsprozesse optimieren, kürzere Entwicklungszyklen und Kostenminimierung, neue innovative Lösungsansätze für multidimensionale, nichtlineare technische wie physikalische Problemstellungen in Forschung und Entwicklung. Die Anwendungsgebiete von ML-Algorithmen sind nicht nur auf Sprachassistenten, autonomes Fahren oder das verbessern von Wetterprognosen beschränkt. Speziell in der seit dem Frühjahr 2020 vorherrschenden Pandemie haben besonders ML-Methoden in der Forschung und Medizin einen großen Beitrag zur Beschleunigung von Entwicklungszyklen geleistet, die zu einer Eindämmung der Ausmaße und Auswirkungen dieser weltweiten Krise beigetragen haben. Beispielsweise ermöglicht ML durch die Fähigkeit von Semantic-Segmentaion eines CNN die Detektion eines Chemielumineszenz-Signals in amplifizierter DNA von PCR-Tests die Analyse-Zeit im Vergleich zu klassischen biologischen Test deutlich zu reduzieren.

Das Deutsche Zentrum für Luft- und Raumfahrt e.V. (DLR e.V.) sucht aus diesen Gründen ebenfalls nach Möglichkeiten klassische Lösungsansätze durch alternative ML-Algorithmen zu ersetzten und damit beispielsweise komplexe und zeitaufwendige Simulationen von Triebwerksprozessen zu beschleunigen. Die in dieser Arbeit realisierte Vorhersage der Eigenfrequenzen von Verdichterschaufeln einer Variation eines bereits ausgelegten Flugzeugtriebwerks unter Verwendung eines ML-Modells stellt somit eine Machbarkeitsstudie für alternative Berechnungsmöglichkeiten dar, dynamischen Festigkeits- und Verformungsberechnungen klassischer regelbasierter Systeme wie einer numerischen Simulation mittels der FEM zu ersetzten. Das explizite Ziel dieser Arbeit ist es, die ersten zehn Eigenfrequenzen der Rotoren und Statoren eines Hochdruckverdichters auf Basis der geometrischen Zusammenhänge aus den dazugehörigen Computer-aided design (CAD)-Modellen des Flugzeugtriebwerks vorherzusagen. Die bereitgestellte Datenbasis wurde dabei mit der DLR e.V. eigenen Kollaborationsplattform Gas Turbine Laboratory (GTlab) [27] erzeugt, welches eine plattformübergreifende Verknüpfung von Simulations- und Vorentwurfsumgebungen ermöglicht. Speziell der darin enthaltene Prozess mit welchem die Eigenfrequenzen mittels einer Modalanalyse des FEM Simulationsprogramms PERMAS erzeugt werden, soll durch ein geeignetes ML-Modell ersetzt werden.

Für die Lösung der Aufgabenstellung wird in dieser Arbeit ein DNN bestehend aus Teilen eines dreidimensionalen CNN und einem vollständig verknüpften ANN verwendet. Darin wird die Fähigkeit der dreidimensionalen Convolutional-Layer ausgenutzt, die versteckten Merkmale in den geometrischen Daten der CAD-Modelle zu erkennen und auf die Eigenfrequenzen abzubilden. Dem Training des DNN gehen weitere Methoden zur Aufbereitung

des Datensatzes wie beispielsweise die Normalisierung der Verteilungen jeweiligen Eigenfrequenzen und die Voxelization der in den CAD-Daten beinhalteten polygonalen Oberflächenmodellen voraus. Bei der Voxelization entsteht ein Diskretisierungsfehler zwischen Oberflächen- und Volumenmodell, welcher durch eine Normalisierung der Vierdichterschaufelblattgeometrien deutlich reduziert werden kann. Diese Normalisierung verändert jedoch das Volumen der einzelnen VSBs, welche dann als zusätzliche Information über einen zusätzlichen Modell Eingang in das DNN eingespeist wird. Für die bestmögliche Konfiguration des ML-Modells wird eine Bayes'sche Optimierung mit dem Open-Source Framework Optuna [1] zur Bestimmung der besten Hyperparamter durchgeführt. Generell wird für die Aufbereitung der Daten, die Erstellung der DNN-Struktur, das Training und die anschließende Validierung und Visualisierung der Ergebnisse mit der High-Level Programmiersprache Python [28] realisiert. Die DNN-Struktur wird mit Keras der High-Level Application Programming Interfacen (API) des Python Pakets Tensorflow [18] erstellt und mittels des darin enthaltenen Backends trainiert und evaluiert.

Zu Beginn dieser Arbeit werden im Kapitel 2 die benötigten Theoretischen Grundlagen behandelt. Im Abschnitt 2.1 wird das mathematische Verfahren der Modalanalyse zur Bestimmung der Modalgrößen eines betrachteten Objekts oder Systems vorgestellt. Ein Bestandteil dieser Modalgrößen sind die Eigenfrequenzen und Eigenformen, welche beispielsweise das Schwingverhalten der in dieser Arbeit verwendeten VSBs beschreiben. Auf diese Weise liefert die Modalanalyse das Hintergrundwissen über die mathematische Berechnung der behandelten Eigenfrequenzen, sowie ein Verständnis für deren Einfluss bei der Auslegung von technischen Anwendungen. Darauf aufbauend bietet der Abschnitt 2.2 mit dem numerischen Verfahren der FEM das mathematische Werkzeug mit welchem viele moderne Simulationsprogramme Festigkeits- und Verformungsberechnungen sowie die Modalanalyse für komplexe Systeme lösen. Von den vorangegangen Abschnitten losgelöst, führt Abschnitt 2.3 in die Berechnung von polygonalen Oberflächenmodellen mittels Non-uniform rational B-Spline (NURBS) ein, welche zur geometrischen Modellierung in CAD ihre Anwendung finden und in dieser Arbeit für die Rekonstruktion der geometrischen Gestalt der im ASCII-Format zur Verfügung gestellten VSB verwendet werden. Für die Verarbeitung der rekonstruierten polygonalen Oberflächenmodelle müssen diese mit der in Abschnitt 2.4 thematisierten Methode der Voxelization in ein binäres dreidimensionales Voxelgitter umgewandelt werden. In diesem Abschnitt werden die beiden Methoden Parity-Count und Ray-Stabbing vorgestellt, welche das in C++ programmierten hocheffiziente Executable (.exe) Binvox von Patrick Min [20] für die Voxelization nutzt und in dieser Arbeit seine Verwendung findet. Im Anschluss deckt der Abschnitt 2.5 alle wichtigen Terminologien und verwendeten Verfahren im Bereich Machine-Learning ab, welche in dieser Arbeit zur Lösung der Aufgabenstellung verwendet werden. Diese erstrecken sich über den in Abschnitt 2.5.1 grundsätzlichen Unterschied zwischen dem ML-Paradigma und klassischen Rule-Based-Systems bis hin zu den verscheiden Arten von Neural Network (NN), CNN in Abschnitt 2.5.7 und ANN in Abschnitt 2.5.6 die in dieser Arbeit zur Vorhersage der Eigenfrequenzen mit den dazugehörigen dreidimensionalen binären Voxelgitter genutzt werden. Das Kapitel der Theoretischen Grundlagen schließt

die Hyperparamter-Optimierung in Abschnitt 2.5.8 und das darin erläuterte ML-Verfahren der Bayes'schen Optimierung ab, welches für die Optimierung der in Abschnitt 3.2.6 vorgestellten variablen DNN-Struktur seine Anwendung findet. In Kapitel 3 wird neben der variablen DNN-Struktur die signifikanten Python Pakete und deren Anwendungsgebiete in dieser Arbeit erläutert. Der Abschnitt 3.2 zeigt explizit, wie die Rekonstruktion der polygonalen Oberflächen eines VSB mittels der Python-Library `geomdl` [4] und den in den Grundlagen erläuterten NURBS realisiert wird. Darauf folgt in Abschnitt 3.2.2 die Anwendung der bereits erwähnte Voxelization mit dem Open-Source Programm Binvox und die Normalisierung der ungleichmäßig verteilten Eigenfrequenzen und Volumen im Abschnitt 3.2.3. Zusätzlich werden die normalisierte Volumen in Abschnitt 3.2.4 durch eine Art Binary-Encoding ähnlich den Voxeln in eine binäre Vektordarstellung überführt, welche die Verarbeitung durch das DNN verbessern soll. Im Kapitel 4 werden die Ergebnisse der mit dem Open-Source Framework-Optuna [1] durchgeführten Hyperparameter-Optimierung analysiert und die unterschiedlichen Parameterkombinationen diskutiert. Auf Basis der daraus resultierenden besten DNN-Konfiguration, wird diese durch Fine-Tuning in Abschnitt 4.3 auf den gesamten Trainings-Daten trainiert und die daraus resultierenden Vorhersagen für die Test-Daten anschließend Ausführlich ausgewertet. Abschließend wird in den Abschnitten 4.3.4 und 4.3.5 ein tiefere Blick in die Funktionsweise des DNN genommen, indem die von den Convolutional-Layern erkannten Merkmale untersucht werden und die Klassifkationsfähigkeit des Modells mittels Transfer-Learning nachgewiesen wird.

2 Theoretische Grundlagen

Dieses Kapitel beinhaltet die benötigten theoretischen Grundlagen, die für das tiefere Verständnis der in dieser Arbeit verwendeten mathematischen Verfahren und Methoden benötigt wird. Im ersten Abschnitte wird die Modalanalyse vorgestellt, die als mathematisches Verfahren zur Bestimmung der Schwingungsgrößen eines System verwendet wird. Zu diesen Schwingungsgrößen zählen die Eigenfrequenzen eines Systems, welche in dieser Arbeit mit einem ML-Model für die geometrischen Gestalt von VSBs vorhergesagt werden. Dieser ML-Ansatz ersetzt eine komplexe Berechnung dieser Eigenfrequenzen durch eine FEM Simulation. Das numerische Verfahren FEM wird im Abschnitte 2.2 erläutert. Die Abschnitte 2.3 und 2.4 behandeln die Berechnung einer geometrischen Oberfläche mit NURBSs und die Umwandlung einer Oberfläche mittels Voxelization in ein Volumenmodel. Die darin beinhalteten mathematischen Methoden werden für die Aufbereitung der zur Verfügung gestellten geometrischen Informationen über die jedes VSB benötigt. Die Aufbereitung ermöglicht die anschließende Verarbeitung durch das in dieser Arbeit erstellte ML-Model. Der letzte und umfangreichste Abschnitt beinhaltet alle benötigten Informationen zum Thema Machine Learning. Darin geklärt werden die Unterschiede und Gemeinsamkeiten häufig verwendeter Terminologien wie AI, Deep Learning (DL) und ML. Des Weiteren werden die Anforderungen an einen Datensatz in den Abschnitten 2.5.2 und 2.5.3 behandelt. Ausgehend von dem biologisch inspirierten Neuron werden die verschiedene Optimierungsverfahren im Abschnitt 2.5.4 vorgestellt. Zuletzt werden die in dieser Arbeit verwendeten ML-Modelle, ANN und CNN vorgestellt.

2.1 Modalanalyse

Die Modalanalyse, auch Eigenschwingungsanalyse genannt, ist nach Natke [22] ein mathematisches Verfahren zur Bestimmung der einzelnen Modalgrößen eines betrachteten Objekts oder Systems. Die Modalgrößen bestehen dabei aus den Eigenfrequenzen, Eigenschwingungsformen, der modalen Masse und der modalen Dämpfung. Diese Parameter sind charakteristisch für das dynamische Verhalten eines schwingenden Systems. Diese Parameter können dabei experimentell bestimmt oder numerisch mittels eines FEM Simulationsprogramms, beispielsweise PERMAS berechnet werden. Im Folgenden wird die analytische Berechnung der Eigenwerte und Eigenfrequenzen mittels der Modalanalyse hergeleitet. Für die exakten Herleitung und Beweise der folgenden Gleichungen wird auf die Fachliteratur [22] verwiesen.

Die Modalanalyse wird anhand eines passiven Systems mit viskoser Dämpfung, ohne gyroskopische sowie zirkulatorische Kräfte für n Freiheitsgrade hergeleitet. Eine Bewegungsgleichung die ein solches System beschreibt lautet:

$$\mathbf{M}\ddot{\mathbf{u}}(t) + \mathbf{D}\dot{\mathbf{u}}(t) + \mathbf{K}\mathbf{u}(t) = \mathbf{f}(t) \tag{2.1}$$

Darin enthalten ist die Massenmatrix $\mathbf{M}$, Dämpfungsmatrix $\mathbf{D}$, Steifigkeitsmatrix $\mathbf{K}$ und die zeit-veränderlichen Vektoren der Verschiebung $\mathbf{u}$, Geschwindigkeit $\dot{\mathbf{u}}$ und Beschleunigung $\ddot{\mathbf{u}}$ der n Freiheitsgrade und der rechten Seite mit dem Lastenvektor $\mathbf{f}$.

Um die Eigenfrequenzen sowie die Eigenformen analytisch zu berechnen, muss das Eigenwertproblem mittels der Bewegungsgleichung sowie einer Ansatzfunktion für die zeitveränderlichen Funktionen gelöst werden. Zur Lösung des homogenen Teils der Bewegungsgleichung wird die rechte Seite $\mathbf{f}(t) = \mathbf{0}$ gesetzt und der Exponentialansatz

$$\mathbf{u}_k(t) = \boldsymbol{\varphi}_k e^{\lambda_k t} \tag{2.2}$$

in die homogene Differentialgleichung eingesetzt. Dies führt auf das quadratische Eigenwertproblem

$$[\mathbf{M}\lambda_k^2 + \mathbf{D}\lambda_k + \mathbf{K}]\boldsymbol{\varphi}_k = 0, \tag{2.3}$$

welches gelöst die nicht reale Eigenwerte der Form

$$\lambda_k = -\omega_k d_k \pm i \underbrace{\omega_k \sqrt{1 - d_k^2}}_{\omega_{dk}} \tag{2.4}$$

und die dazugehörigen Eigenvektoren $\boldsymbol{\varphi}_k$ ergibt. Darin enthalten ist die jeweilige Eigenkreisfrequenz $\omega_k = |\lambda|$ und der Dämpfungsterm d_k. Die allgemeine Lösung des homogenen Teils der Bewegungsgleichung $\mathbf{u}_h(t)$ ist die Linearkombination aller Teillösungen,

$$\mathbf{u}_h(t) = \sum_{k=1}^{n} \mathbb{R}\left\{\boldsymbol{\varphi}_k e^{\lambda_k t}\right\} = \sum_{k=1}^{n} \boldsymbol{\varphi}_k \eta_k(t) \tag{2.5}$$

bestehend aus den jeweiligen Eigenvektoren oder Eigenformen und Eigenwerten, die zusammen mit dem Expotentialansatz 2.2 die Zeitfunktion $\eta(t)$ in den *modalen Koordinaten* bilden. Eigenform und modale Koordinate bilden zusammen eine Eigenmode. Die einzelnen Eigenmoden zusammengenommen definieren schließlich das Schwingverhalten und die Schwingungseigenschaften eines Systems.

Ein vereinfachtes Beispiel zur Ausformulierung der in Gleichung (Gl.) 2.1 beschriebenen Bewegungsgleichung ist die in Abb. 2.1 dargestellte gedämpfte Schwingerkette. In einem solchen Starrkörpersystem sind die einzelnen Massen $m_1 = m_2 = m_3$ in unverformbaren Körpern konzentriert. Die Steifigkeiten beziehungsweise Nachgiebigkeiten sind in masselosen Federn $k_1 = k_2 = k_3$ und Dämpfern $d_1 = d_2 = d_3$ konzentriert. Die Auslenkung der

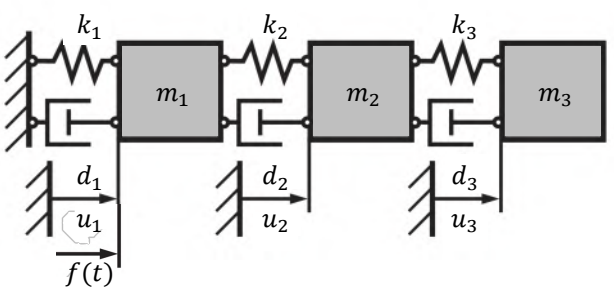


Abbildung 2.1: Eine einfache Schwingerkette.In einem solchen Starrkörpersystem sind die einzelnen Massen $m_1 = m_2 = m_3$ in unverformbaren Körpern konzentriert. Die Steifigkeiten beziehungsweise Nachgiebigkeiten sind in masselosen Federn $k_1 = k_2 = k_3$ und Dämpfern $d_1 = d_2 = d_3$ konzentriert. Die Auslenkung der Masse u_1, u_2 und u_3 entsprechen der Anzahl der n Freiheitsgrade des Systems. [1]

Masse u_1, u_2 und u_3 entsprechen der Anzahl der n Freiheitsgrade des Systems. Die Bewegungsgleichung kann mittels des *Newton-Euler* oder dem *Lagrange-Verfahren* berechnet werden. Nach Lagrange werden zuerst die kinetische

$$E_\mathrm{k} = \frac{1}{2}m_1\dot{u}_1^2 + \frac{1}{2}m_2\dot{u}_2^2 + \frac{1}{2}m_3\dot{u}_3^2 \tag{2.6}$$

und potentielle Energie

$$E_\mathrm{p} = \frac{1}{2}k_1u_1^2 + \frac{1}{2}k_2(u_2 - u_1)^2 + \frac{1}{2}k_3(u_3 - u_2)^2 \tag{2.7}$$

des Systems aufgestellt. Im zweiten Schritt werden die nicht-konservativen Kräfte Q_s^nk für jeden Freiheitsgrad

$$\begin{aligned} Q_1^\mathrm{nk} &= f(t) - (d_1 + d_2)\dot{u}_1 + d_2\dot{u}_2, \\ Q_2^\mathrm{nk} &= -(d_2 + d_3)\dot{u}_2 + d_2\dot{u}_1 + d_3\dot{u}_3, \\ Q_3^\mathrm{nk} &= -d_3\dot{u}_3 + d_3\dot{u}_2 \end{aligned} \tag{2.8}$$

bestimmt. Durch aufstellen der Lagrange‘schen Gleichung 2. Art

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial E_\mathrm{k}}{\partial \dot{u}_s}\right) - \frac{\partial E_\mathrm{k}}{\partial u_s} + \frac{\partial E_\mathrm{p}}{\partial u_s} = Q_s^\mathrm{nk} \tag{2.9}$$

und ableiten der einzelnen Ausdrücke ergibt sich für den ersten Freiheitsgrad $s = 1$:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial E_\mathrm{k}}{\partial \dot{u}_1}\right) = \frac{\mathrm{d}}{\mathrm{d}t}(m_1\dot{u}_1) = m_1\ddot{u}_1, \quad \frac{\partial E_\mathrm{k}}{\partial u_1} = 0, \quad \frac{\partial E_\mathrm{p}}{\partial u_1} = k_1u_1 - k_2(u_2 - u_1) \tag{2.10}$$

mit den dazugehörigen nicht-konservativen Kräften Q_1^nk die Bewegungsgleichung

$$m_1\ddot{u} + (d_1 + d_2)\dot{u}_1 - d_2\dot{u}_2 + (k_1 + k - 2)u_1 - k_2u_2 = f(t) \tag{2.11}$$

für den ersten Freiheitsgrad. Werden diese Schritte analog für die Freiheitsgrade $s = 2$ und $s = 3$ ausgeführt ergibt sich die systembeschreibende Bewegungsgleichung in Matrixschreibweise

$$\underbrace{\begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix}}_{\mathbf{M}} \underbrace{\begin{bmatrix} \ddot{u}_1 \\ \ddot{u}_2 \\ \ddot{u}_3 \end{bmatrix}}_{\ddot{\mathbf{u}}} + \underbrace{\begin{bmatrix} (d_1+d_2) & -d_2 & 0 \\ -d_2 & (d_2+d_3) & -d_3 \\ 0 & -d_3 & d_3 \end{bmatrix}}_{\mathbf{K}} \underbrace{\begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \end{bmatrix}}_{\dot{\mathbf{u}}} + \underbrace{\begin{bmatrix} (k_1+k_2) & -k_2 & 0 \\ -k_2 & (k_2+k_3) & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix}}_{\mathbf{K}} \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{\mathbf{u}} = \underbrace{\begin{bmatrix} f(t) \\ 0 \\ 0 \end{bmatrix}}_{\mathbf{f}} \tag{2.12}$$

Unter der Verwendung des in Gl. 2.3 eingeführten quadratischen Eigenwertproblems, ergeben sich für $n = 3$ Freiheitsgrade die in Abb. 2.2 schematisch dargestellten n Eigenformen $\boldsymbol{\varphi}$. Die Eigenformen sind entsprechend der Größe ihrer Eigenfrequenzen aufsteigend sortiert.

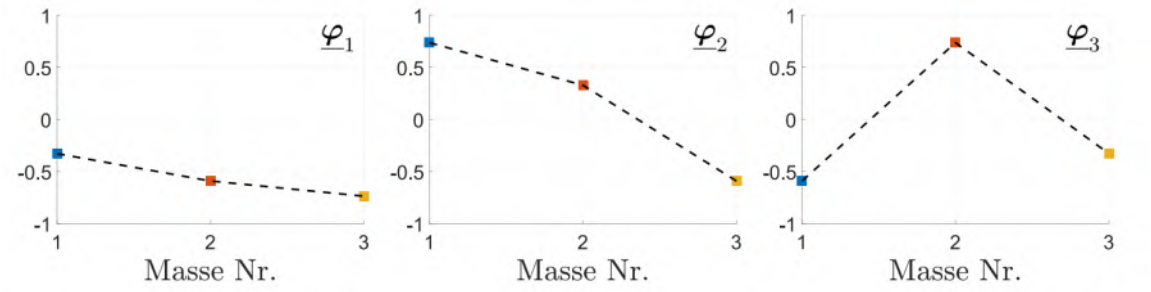


Abbildung 2.2: Eigenformen $\boldsymbol{\varphi}_i$ der Schwingerkette aus Abb. 2.1, mit $k = [1, 2, 3]$.[1]

Dies ist damit zu begründen, dass bei stetig steigender Energieeinbringung in ein System, die Mode mit der niedrigsten Eigenfrequenz zu erst angeregt wird. Die Eigenform $\boldsymbol{\varphi}_1$ der ersten Mode ist ein Beispiel für eine gleichphasige Schwingung, bei der sich alle Massen *in Phase* bewegen und gleichgerichtet ausgelenkt sind. Die zweite und dritte Mode sind dagegen Beispiele deren dritte, respektive zweite Masse sich in *Gegenphase* zu den anderen Massen bewegen. Wie vorangegangen durch Gl. 2.5 erläutert, besitzt jede Eigenmode auch einen zeitlichen Anteil, welcher durch die zu den Eigenformen korrespondierenden modalen Koordinaten $\eta_k(t)$ beschrieben wird. Aufgrund der Dämpfungskonstanten d_k und die in diesem Beispiel gewählte steifigkeitsproportionale modale Dämpfung, ergeben sich die in Abb. 2.3 dargestellten exponentiell abnehmenden Amplitudenverläufe der einzelnen unterkritisch gedämpften modalen Koordinaten, welche zu jedem Zeitpunkt den Anteil an der Eigenform an der Gesamtschwingung angeben. Das in Abb. dargestellte Gesamtschwingungsverhalten der Schwingerkette ergibt sich durch die Superposition Gl. 2.5 der einzelnen Moden.

Das Gesamtschwingverhalten eines Systems zu kennen ist von größter Bedeutung, denn eine Anregung dieses Systems in seinen Eigenfrequenzen kann zu einer Resonanzkata-

[1] Entnommen aus Krack[16]

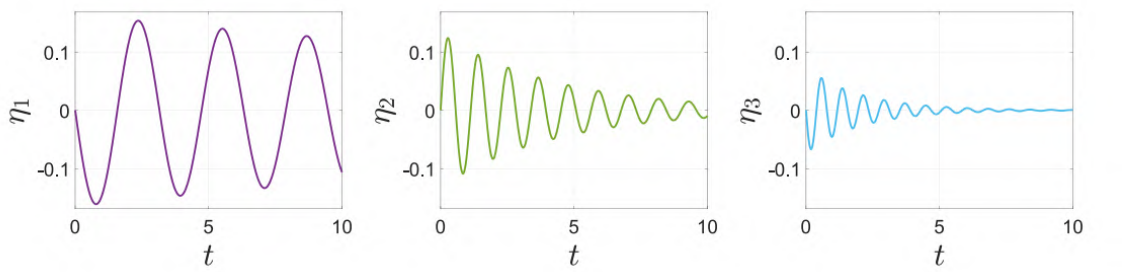


Abbildung 2.3: Zeitlicher Verlauf der modalen Koordinaten $\eta_k(t)$ der Schwingerkette aus Abb. 2.1 bei modaler Dämpfung $d_1 = d_2 = d_3$ mit Anfangsbedingungen $\mathbf{u}_0 = \mathbf{0}$ und $\dot{\mathbf{u}}_0 = [1\ 0\ 0]^T$.[1]

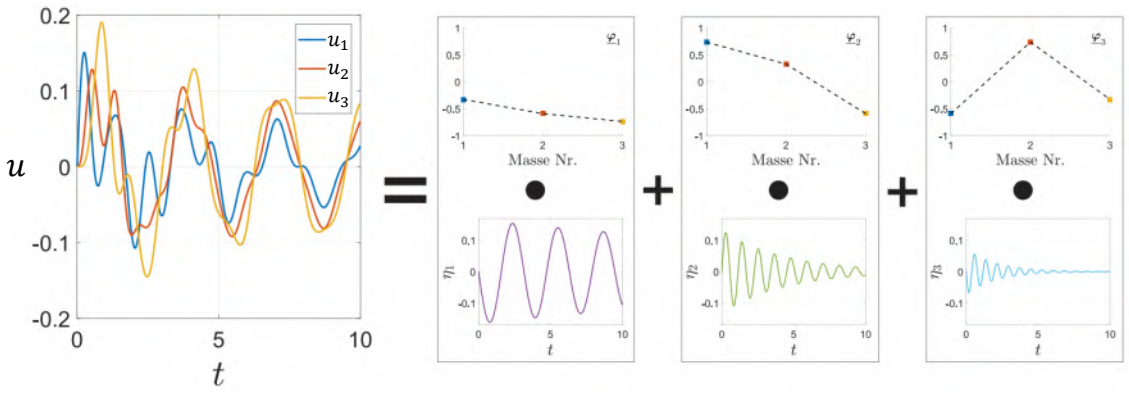


Abbildung 2.4: Gesamtschwingverhalten der gedämpften Schwingerkette aus Abb. 2.1 durch Superposition $\mathbf{u}(t) = \sum \boldsymbol{\varphi}_k \eta_k(t)$ der einzelnen Moden.[1]

strophe und dem Versagen des Systems führen. Dies kann beispielsweise wie in Fengjun [17] beschrieben durch Anregung in der ersten Eigenfrequenz eines Rotorblattes in einem Triebwerkskompressor zu Ermüdungsschädigungen bis hin zum Bruch infolge von Biegeschwingungen führen.

2.2 Finite-Elemente-Methode

Die FEM ist ein numerisches Verfahren zur Lösung partieller Differentialgleichungen, welche häufig zur statischen sowie dynamischen Festigkeits- und Verformungsberechnung verwendet wird. Numerische Verfahren finden überall dort ihre Anwendung, wo eine analytische Lösung des Problems nicht möglich oder zu komplex ist. Abbildung 2.5 zeigt hierbei die Vorgehensweise zur Modellierung eines realen Systems. Sobald eine analytische Lösung des mathematischen Modells beziehungsweise der System beschreibenden

[2]Entnommen aus Wagner[29]

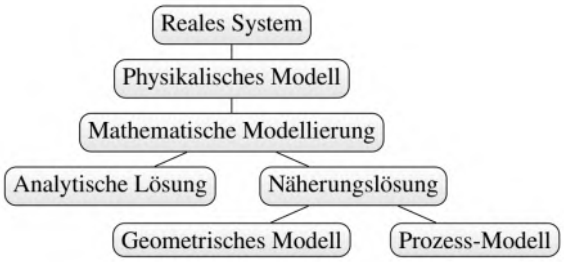


Abbildung 2.5: Vorgehensweise bei der Modellierung von Rule-Based-Systems.[2]

Gleichungen nicht möglich ist, muss eine sogenannte Näherungslösung dieser Gleichungen mittels eines numerischen Verfahrens approximiert werden. Hierbei wird unterteilt zwischen einem geometrischen stationären Modell, sowie einem instationären Prozess-Model. Wie der Name FEM suggeriert, wird das geometrische Modell mittels finiter, also endlich vieler Elemente und Knoten diskretisiert. Dies ermöglicht mittels des Prozess-Models und geeigneter Ansatzfunktionen, sowie Randbedingung die Elementmatrizen und -lastvektoren zu bestimmen und anschließend über den Zusammenbau des Gleichungssystems das physikalische Verhalten des gesamten Systems abzubilden. Im folgenden wird die Lösung eines realen Systems mittels FEM Anhand eines Dehnstabs erläutert. Die Abbildung 2.6

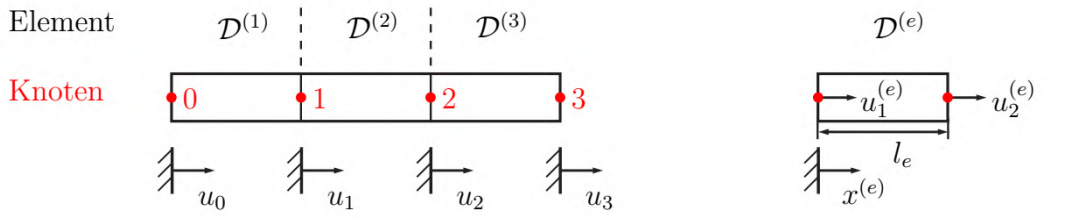


Abbildung 2.6: Unterteilung einer Struktur in finite Elemente, am Beispiel des Dehnstabs.[3]

zeigt die Unterteilung des Dehnstabs in $N_e = 3$ Elemente. In diesem eindimensionalen Beispiel sind die Ränder der Elemente $\mathcal{D}^{(e)}$, durch ihre Knoten begrenzt und überlappen sich nicht. Knoten dürfen sich ebenso innerhalb, beispielsweise im Zentrum der Elemente befinden, jedoch sind in diesem Beispiel nur $N_n = 4$ Knoten definiert. Für eine allgemeine Berechnung der kinematischen Zwänge in jedem Element, werden lokale Koordinaten wie $x^{(e)}$ verwendet und zusätzlich über die Länge l_e dimensionslos gemacht

$$\xi := \frac{x^{(e)}}{l_e}. \tag{2.13}$$

Daraus resultiert für jedes Element am linken Ende $\xi = 0$ und am rechten Ende $\xi = 1$. Auf dieser Grundlage wird für das Verschiebungsfeld ein Ansatz mit Linearkombination

gewählt.

$$u^{(e)}(\xi, t) = \sum_{i=1}^{N_{d,e}} \psi_i^{(e)}(\xi) q_i^{(e)}(t) \tag{2.14}$$

Dieser Ansatz beinhaltet die Methode der separierten Variablen zur Lösung partieller Differentialgleichungen, nachzulesen in Epstein [10]. Diese Methode ermöglicht die separate Berechnung der räumlichen $\psi_i^{(e)}$ und zeitabhängigen Komponente $q_i^{(e)}(t)$. Im Folgenden wird ausschließlich die Berechnung der räumlichen Komponenten vertieft. Hier sei vermerkt, dass die zeitabhängigen verallgemeinerten Koordinaten $q_i^{(e)}$, ebenfalls ausgehend von Anfangswerten durch Ansatzfunktionen oder mittels eines nummerischen Zeitschrittverfahrens gelöst werden. Das implizite und unbedingt stabile Newmark-Verfahren ist speziell geeignet für Vibrationsprobleme und in vielen Finite-Elemente-Programmen verfügbar. $N_{d,e}$ steht für die Anzahl der Freiheitsgrade oder verallgemeinerten Koordinaten $q_i^{(e)}$ und Ansatzfunktionen $\psi_i^{(e)}$. Die Wahl der Ansatzfunktionen und Positionierung der Knotenpunkte sind für die Integrationsgenauigkeit und die daraus resultierende Größe des numerischen Fehlers entscheidend. Dabei werden meistens Polynome als Ansatzfunktionen gewählt. Wird das Systems fein genug diskretisiert, sind lineare oder quadratische Polynome als Ansatzfunktionen in ihrer Güte ausreichend. Die in Abbildung 2.7 a) dargestellten linearen Ansatzfunktionen lauten:

$$\psi_1^{(e)}(\xi) = 1 - \xi, \quad \psi_2^{(e)}(\xi) = \xi \tag{2.15}$$

und besitzen die Lagrange-Eigenschaft

$$\psi_i^{(e)}(\xi_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} =: \delta_{ij}. \tag{2.16}$$

Alle Ansatzfunktionen, welche die Eigenschaft 0 oder 1 des Kronecker Delta δ_{ij} erfüllen, werden Lagrange-Basisfunktionen genannt und sind durch die folgenden Gl. 2.18 definiert.

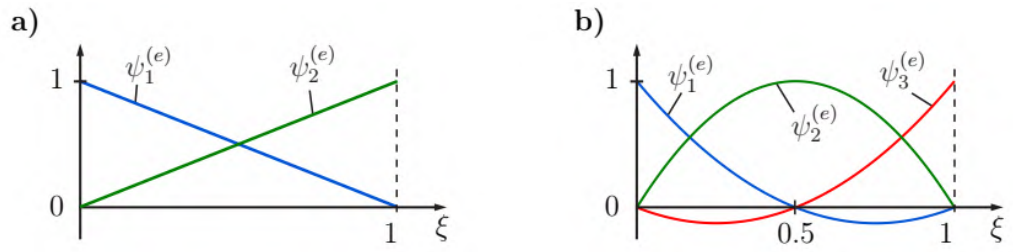


Abbildung 2.7: Ansatzfunktionen eines eindimensionalen Elements. **a**) Linear. **b**) Quadratisch.[3]

[3] Entnommen aus Krack[16]

Aus der Eigenschaft 2.16 und Gl. 2.14 resultiert

$$u_i^{(e)} = u^{(e)}(\xi_i, t) = q_i^{(e)}. \tag{2.17}$$

Dadurch wird gezeigt, dass die Knotenverschiebungen mit den verallgemeinerten Koordinaten übereinstimmen und ihre Positionen ξ_i die Stützstellen repräsentieren. Zwischen den Stützstellen müssen die Werte über die Ansatzfunktionen interpoliert werden. Gl. 2.14 die diese Interpolation ermöglicht, wird deshalb auch Interpolationseigenschaft genannt. Die in Abb.2.7 b) dargestellten quadratischen Ansatzfunktionen mit drei Knoten an den Stellen $\xi_1 = 0$, $\xi_2 = 0.5$ und $\xi_3 = 1$ lassen sich mittels der Lagrange Basis Polynome als Ansatzfunktionen sowie Gl. 2.14 aufstellen. In 1D sind die Lagrange Basis Polynome in geschlossener Form wie folgt definiert:

$$\psi_i^{(e)}(\xi) = \prod_{i=1, i\neq j}^{N_{d,e}} \frac{\xi - \xi_i}{\xi_j - \xi_i} = \left(\frac{\xi - \xi_1}{\xi_j - \xi_1}\right)\left(\frac{\xi - \xi_2}{\xi_j - \xi_2}\right)... \tag{2.18}$$

Mittels der Stützstellen erhält man aus Gl. 2.18

$$\psi_1^{(e)}(\xi) = 2\xi^2\ ,\ \psi_2^{(e)}(\xi) = -(1-2\xi)^2\ ,\ \psi_3^{(e)}(\xi) = 2(\xi-1)^2, \tag{2.19}$$

Gl. 2.14 und unter Verwendung von Gl. 2.16 für die in Abb. 2.7 b) dargestellten Knotenverschiebungen, die gewünschten Ansatzfunktionen

$$\psi_1^{(e)}(\xi) = 1 - 3\xi + 2\xi, \quad \psi_2^{(e)}(\xi) = 4\xi(1-\xi)^2, \quad \psi_3^{(e)}(\xi) = \xi(2\xi - 1). \tag{2.20}$$

Unter Verwendung einer Methode der Gewichteten Residuen, dem Galerkin Verfahren, lassen sich elementweise die Massenmatrix $\mathbf{M}_e$, Steifigkeitsmatrix $\mathbf{K}_e$ und Dämpfungsmatrix $\mathbf{D}_e$, sowie der Lastenvektor $\mathbf{f}_e$ aufstellen.

$$\mathbf{M}_e = [m_{ij}] \quad m_{ij} = \int_0^1 \rho A \psi_i^{(e)}(\xi)\psi_j^{(e)}(\xi) l_e \mathrm{d}\xi, \tag{2.21}$$

$$\mathbf{K}_e = [k_{ij}] \quad k_{ij} = \int_0^1 EA \frac{\partial \psi_i^{(e)}(\xi)}{\partial \xi} \frac{\partial \psi_j^{(e)}(\xi)}{\partial \xi} \underbrace{\left(\frac{\partial \xi}{\partial x^{(e)}}\right)}_{1/l_e^2} l_e \mathrm{d}\xi, \tag{2.22}$$

$$\mathbf{D}_e = [d_{ij}] \quad d_{ij} = \int_0^1 d\psi_i^{(e)}(\xi) \frac{\partial \psi_j^{(e)}(\xi)}{\partial \xi} l_e \mathrm{d}\xi, \tag{2.23}$$

$$\mathbf{f}_e = [f_i] \quad f_i = \int_0^1 p\psi_i^{(e)}(\xi) l_e \mathrm{d}\xi, \tag{2.24}$$

Die numerische Lösung der integralen analytischen Formulierung wird numerische Integration genannt. Für die Integration einer Funktion f mit Gewichtungsfunktion (Ansatzfunktion) $W(x)$ und $N_{d,e}$ Stützstellen

$$\int_{-1}^{1} f(x)W(x)\mathrm{d}x \simeq \sum_{l=1}^{N_{d,e}} f(x_l)w_l, \tag{2.25}$$

lässt sich daraus eine Aussage über die numerische Integrationsgenauigkeit treffen. Beispielsweise lassen sich Funktionen vom Grad p, welche die Lagrange-Eigenschaft erfüllen, mit $N_{d,e} = p + 1$ Stützstellen, die entsprechend der Legendre-Gauß-Punkte verteilt sind sogar bis zum Grad $2p + 1$ exakt integrieren. Für die mathematische Herleitung und beweise der Gausschen Integrationsmethode, wird auf Freund [11] verwiesen. In den Fällen wo diese Bedienungen jedoch nicht gleichzeitig erfüllt werden entsteht ein numerischer Fehler bei der Integration.

Zuletzt müssen die einzeln betrachteten Elemente zu den Elementmatrizen zusammengesetzt werden, aus denen die systembeschreibende Bewegungsgleichung 2.1 besteht. Die Interpolationseigenschaft der Ansatzfunktionen ermöglicht es die einzelnen Elemente so zu verknüpfen, dass diese sich kinematisch verträglich verhalten. Hierfür müssen an den Rändern benachbarter Elemente die selben Knoten-Freiheitsgrade verwendet werden. Mittels einer Zuordnungsmatrix $\mathbf{L}_e$ werden globale den lokalen Knoten-Freiheitsgraden der einzelnen Elemente zugewiesen. Diese hat die Größe $N_{d,e} \times n$. n steht für die Anzahl aller System-Freiheitsgrade, beziehungsweise alle globaler Knoten-Freiheitsgrade. Dies führt auf die Zuordnung

$$\begin{bmatrix} u_1^{(e)} \\ \vdots \\ u_{N_{d,e}}^{(e)} \end{bmatrix} = \mathbf{u}^{(e)} = \mathbf{L}_e \mathbf{u} \tag{2.26}$$

und die Gesamtmassen-, Gesamtdämpfungs- und Gesamtsteifigkeitsmatrix, sowie dem Gesamtlastenvektor

$$\mathbf{M} = \sum_{e=1}^{N_e} \mathbf{L}_e{}^{\mathrm{T}} \mathbf{M}_e \mathbf{L}_e \quad \mathbf{D} = \sum_{e=1}^{N_e} \mathbf{L}_e{}^{\mathrm{T}} \mathbf{D}_e \mathbf{L}_e \quad \mathbf{K} = \sum_{e=1}^{N_e} \mathbf{L}_e{}^{\mathrm{T}} \mathbf{K}_e \mathbf{L}_e \quad \mathbf{f} = \sum_{e=1}^{N_e} \mathbf{L}_e{}^{\mathrm{T}} \mathbf{f}_e. \tag{2.27}$$

Dies ermöglicht auch das numerische lösen des quadratischen Eigenwertproblems Gl. 2.3 und die daraus resultierenden Eigenwerte, Eigenkreisfrequenzen und Eigenformen. Wie in Abschnitt 2.1 erläutert, sind diese Größen entscheidend für die Schwingeigenschaften eines Systems. Der Fokus dieser Arbeit liegt dabei auf den Eigenkreisfrequenzen. Die numerische Modalanalyse ist eine standardisierte Berechungsmethode in allen FEM-Programmen, jedoch sind die folgenden Schritte der FEM

- Vernetzen: Unterteilen der Geometrie in Elemente und Knoten
- Berechnung der Elementmatrizen und -lastvektoren
- Zusammenbau und Lösen der systembeschreibenden Bewegungsgleichung

insbesondere für komplexe Objekte immer noch rechen- und zeitintensiv. Die Motivation hinter dieser Arbeit ist deshalb die Ausarbeitung eines ML-Modells, dass die genannten Schritte der FEM-Simulation ersetzt.

2.3 Non-uniform rational B-Splines

NURBS basieren auf den von Pierre Etienne Bezier entwickelten und nach ihm benannten *Bézier-Splines*, deren Kontrollpunkte nicht auf der durch diese beschriebenen Kure liegen. Non-uniform rational B-Splines repräsentieren eine Generalisierung der *Bézier-Splines* und werden als mathematisches Model zur Berechnung von Kurven und Freiformflächen verwendet, welche beispielsweise zur geometrischen Modellierung in CAD ihre Anwendung finden. Die folgenden Ausführungen beinhalten alle für diese Arbeit relevanten Formeln und Informationen zum Thema NURBS. Für alle weiteren Details sei hier auf die Lektüren von Bingol [4] und De Boor [8] verwiesen. Ähnlich den *Bézier-Spline*-Kurven werden NURBS-Kurven $C(u)$ vom Grad p stückweise aus Basispolynomen $R_{i,k}$ vom Grad $k = p-1$, gewichteter (w_i) Kontrollpunkte P_i und einem Knotenvektor $\mathbf{u}$ aufgespannt. Der Unterschied liegt in den Gewichten w_i, welche die Splines rational werden lassen. Die NURBS-Kurve

$$C(u) = \sum_{i=0}^{n} R_{i,k}(u) P_i \tag{2.28}$$

besteht aus n Kontrollpunkten P_i und den dazugehörigen rationalen B-Spline-Basisfunktionen

$$R_{i,k}(u) = \frac{N_{i,k}(u) w_i}{\sum_{j=0}^{n} N_{j,k}(u) w_j}, \tag{2.29}$$

die wiederum aus einzelnen B-Spline-Basisfunktionen $N_{i,k}$ bestehen. Der Knotenvektor

$$\mathbf{u} = \{\underbrace{a, \ldots, a}_{p+1}, u_{p+1}, \ldots, u_{r-p-1}, \underbrace{b, \ldots, b}_{p+1}\}, \tag{2.30}$$

besteht aus den Parameterwerten u_i, die eingesetzt in die B-Spline-Basisfunktion deren Anteil an den einzelnen Segmenten $R_{i,k}(u)$ und Kontrollpunkten P_i beeinflussen und somit auch die gesamten NURBS-Kurve. Die Werte der einzelnen Elemente des Knotenvektors müssen dabei monoton steigen. Für die Endknoten muss $a_{(i)} = a_{(j)}$ und $b_{(i)} = b_{(j)}$ gelten. Häufig werden den Endknoten die Werte $a = 0$ und $b = 1$ zugewiesen. Die Anzahl der Knoten ($|\mathbf{u}| = r = n + p + 1$) ergibt sich aus der Summe aus Kontrollpunkte und dem Grad der gesamten Kurve plus eins. Mit NURBS können verschiedene geometrische Stetigkeiten $\mathcal{G}^{(e)}$ erzeugt werden. $\mathcal{G}^{(0)}$ ist die *Positions-Stetigkeit* in welcher zwei Kurven in einem Punkt zusammentreffen und auch Kanten bilden können. Dies ermöglicht beispielsweise die Modellierung einer spitzen Schaufelblatthinterkante Trailing Edge (TE). $\mathcal{G}^{(1)}$ ist die *Tangential-Stetigkeit* in welcher neben dem zusammentreffen der Endpunkte zweier Kurven auch die Parallelität der Endvektoren zutreffen muss. $\mathcal{G}^{(3)}$ ist die *Krümmungsstetigkeit* in welcher die tangentiale Stetigkeit der Endpunkte zweier Kurven erfüllt ist, sowie die zweite Ableitung der Kurven in den Endpunkten übereinstimmt. Dies ermöglicht beispielsweise die Konstruktion eines Einheitskreises, die mit nicht-rationalen Splines wie den Bézier-Splines nicht möglich ist.

Die Konstruktion von NURBS-Flächen

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} R_{i,j}(u,v) P_{i,j}, \tag{2.31}$$

ist eine Erweiterung von Gl. 2.28, mit einer weiteren parametrisierten Richtung v. Mit den stückweise rationalen Basis-Funktionen

$$R_{i,j}(u,v) = \frac{N_{i,k}(u) N_{j,q}(v) w_{i,j}}{\sum_{k=0}^{n} \sum_{l=0}^{m} N_{k,p}(u) N_{l,q}(v) w_{k,l}}, \tag{2.32}$$

und den Knotenvektoren

$$\begin{aligned} \mathbf{u} &= \{\underbrace{0,\ldots,0}_{p+1}, u_{p+1}, \ldots, u_{r-p-1}, \underbrace{1,\ldots,1}_{p+1}\} \\ \mathbf{v} &= \{\underbrace{0,\ldots,0}_{q+1}, v_{q+1}, \ldots, v_{s-q-1}, \underbrace{1,\ldots,1}_{q+1}\}, \end{aligned} \tag{2.33}$$

für die beiden parametrisierten Richtungen u und v. Die Anzahl der in den Knotenvektoren enthaltenen monoton steigenden Elemente $u_i \in (0,1)$ und $v_i \in (0,1)$ beträgt jeweils $r = n + p + 1$ und $s = m + q + 1$.

2.4 Voxelization

Die Umwandlung eines polygonalen Oberflächen-Modells in ein aus einzelnen Voxeln (*Volumenelementen*) bestehendes Volumen-Modell wird als *Voxelization* bezeichnet. Die im vorangegangenen Abschnitt eingeführten NURBS-Flächen stellen dabei eine Möglichkeit zur Erzeugung der Oberfläche eines solchen polygonal Modells dar. In dem von Fakir Nooruddin und Greg Turk veröffentlichten Artikel [23] werden die zur Umwandlung benötigten Methoden *Parity-Count* und *Ray-Stabbing* vorgestellt. Beide Methoden ermöglichen die Unterteilung eines polygonal Modells in ein reguläres Gitter, bestehend aus einzelnen würfelförmigen Zellen (Voxel) deren Zentrum auf den Gitterpunkten liegen und einen Dichte-Faktor zwischen Null und Eins besitzen. Voxel die ganzheitlich außerhalb des polygonal Models liegen und keine Überschneidungen mit dessen Oberfläche besitzen, haben einen Dichte-Faktor von Null. Voxel welche im Ganzen von dem polygonal Model eingeschlossen werden, haben einen Dichte-Faktor von Eins. Mit den beiden Mothoden kann nicht nur eine dünnschalige Darstellung eines Polygonen Modells erzeugt werden, in der nur die Voxel nahe der polygonalen Oberfläche erkannt werden. Sondern *Parity Count* und *Ray Stabbing* erkennen auch die von den Oberflächen eingeschlossenen Gebiete und weisen diesen internen Gitterpunkten Voxel mit einem Dichte-Faktor von Eins zu. Dies wird in den folgenden Abschnitten verdeutlicht.

2.4.1 Parity Count

In dieser Methode, werden für ein das polygonal Model einschließendes Volumen der Größe $N \times N \times N$, $N \times N$ geradlinige Strahlen betrachtet, die das Volumengleich einem regulären Gitter durchdringen. Dabei passiert jeder Strahl N Voxel Zentren. Mittels Orthogonalprojektion und polygonaler Scan-Konvertierung werden Stichproben den einzelnen Polygonen zugewiesen. Diese Stichproben beschreiben einen Funktionswert und die dazugehörigen Koordinaten auf der Oberfläche des polygonal Models. Fallen diese Koordinaten der Stichproben mit den Strahlen zusammen, welche die Oberfläche des Models durchdringen, so werden diese als Tiefen-Stichproben (*depth samples*) gespeichert. Ist die Anzahl der Tiefen-Stichproben die sich vor und hinter einem Voxel-Zentrum befinden ungerade, so befindet sich das Voxel für diesen Strahl innerhalb des polygonal Models. Diese Untersuchung anhand der Tiefen-Stichproben wird für k verschiedene Richtungen der Orthogonalprojektion durchgeführt und eine Mehrheitsabstimmung (*majority vote*) über die Stimmen der einzelnen Richtungen entscheidet über die endgültige Klassifizierung diese Voxels. Die aus Fakir Nooruddin und Greg Turk [23] entnommene Abb. 2.8 veranschaulicht die Stärken und Schwächen des Verfahrens für unterschiedlich Entartungen einer Modell Oberfläche.

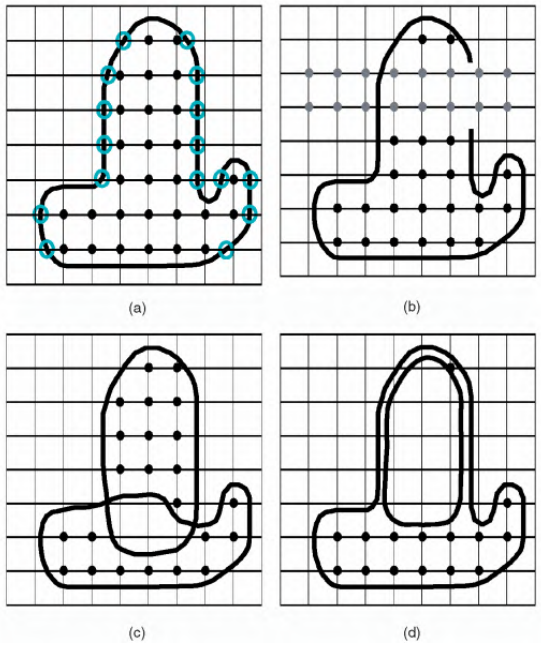


Abbildung 2.8: Scan-Konvertierung unterschiedlich entarteter Oberflächen eines polygonal Modells mit der Parity-Count-Methode.(a) Idealbeispiel geschlossenes Modell,(b) Fallbeispiel für die Scan-Konvertierung mit nicht geschlossener Oberfläche,(c) Fallbeispiel mit sich überschneidenden Teilflächen und (d) Fallbeispiel mit dünnwandigen Modell Abschnitten.[4]

[4] Entnommen aus Fakir Nooruddin und Greg Turk [23]

(a) Das geschlossene Modell stellt das Idealbeispiel für die Scan-Konvertierung dar. Die Tiefen-Stichproben, an welchen die Strahlen das Modell schneiden sind mit blauen Kreise markiert. Die Zentren der Voxel, welche zwischen den Tiefen-Stichproben liegen und als innerhalb des Modells erkannt werden, sind mit schwarzen Punkten markiert. (b) Fallbeispiel für die Scan-Konvertierung eines Modells, dessen Oberfläche nicht geschlossen ist. Die grauen markierten Gitterpunkte stellen Voxel dar, die in horizontaler Richtung als außerhalb und in vertikaler Richtung als innerhalb des Modells klassifiziert werden. Die Scan-Konvertierung aus mehreren Richtungen mit Mehrheitsentscheid identifiziert diese anschließend als innerhalb des Modells. (c) Scan-Konvertierung eines Modells, welches aus sich überschneidenden Teilflächen besteht. Dies ist ein Fallbeispiel, in welchem die *Ray-Stabbing*-Methode zur Identifikation der fehl klassifizierter Voxel verwendet werden muss. Voxel die sich in der Schnittmenge zweier Oberflächen Modelle befinden, werden durch die *Parity-Count*-Methode als außerhalb klassifiziert. (d) Ein weiteres Fallbeispiel in welchem die Scan-Konvertierung der *Parity-Count*-Methode fehlschlägt sind dünnwandigen Abschnitte eines polygonal Modells. Hier wird wieder deutlich, warum bei manchen Modellen die *Ray-Stabbing*-Methode erforderlich ist.

2.4.2 Ray-Stabbing

Wie speziell Fallbeispiel (c) der Abb. 2.8 gezeigt hat, schlägt die Scan-Konvertierung der *Parity-Count*-Methode bei sich überlappenden Oberflächen einzelner Komponenten eines aus diesen zusammengesetzten Gesamtmodells fehl. Aus diesem Grund haben Fakir Nooruddin und Greg Turk [23] die *Ray-Stabbing*-Methode entwickelt. Diese Methode nutzt gleich der *Parity-Count*-Methode die Orthogonalprojektion zur Identifikation der Tiefen-Stichproben, welche die Schnittpunkte der Strahlen mit den Oberflächen beschreiben. Dabei speichert die *Ray-Stabbing*-Methode im Gegensatz nur die erste und letzte Tiefen-Stichprobe eines jeden Strahls. Alle Voxel die zwischen diesen beiden Tiefen-Stichproben liegen werden als innerhalb des Models klassifiziert. Um Fehlklassifikationen zu verhindern, werden die Voxel wieder führ mehrere Richtungen Ausgewertet. Klassifiziert nur eine Richtung das Voxel Zentrum als außerhalb des polygonal Models, so überstimmt diese alle anderen Richtungen. In Akir Nooruddin und Greg Turk [23] wird darauf verwiesen, dass drei Strahlenrichtungen für diese Methode und die Meisten Anwendungen ausreichend sind, der Standard aber auf 13 Richtungen festgelegt ist.

2.5 Machine-Learning

Machine-Learning, AI, DL, sind Fachgebiete die teils mit revolutionären Ansätzen die Probleme der Neuzeit zu lösen versuchen und in der Vergangenheit oft unter vielen verscheiden Namen bekannt waren. Dabei befassen sie sich alle mit der Verarbeitung und Analyse von Datensätzen und wurden mit dem voranschreiten des digitalen Zeitalters und

dem damit einhergehenden Wachstum von nutzbaren Trainingsdaten, von immer größerer Bedeutung. Das parallele quantitative und qualitative Wachstum in der Infrastruktur von Computer Hard- und Software ermöglicht es, zusätzlich immer größere und komplexere Anwendungsgebiete abzudecken. Zur Einordnung der verschiedenen Terminologien dient

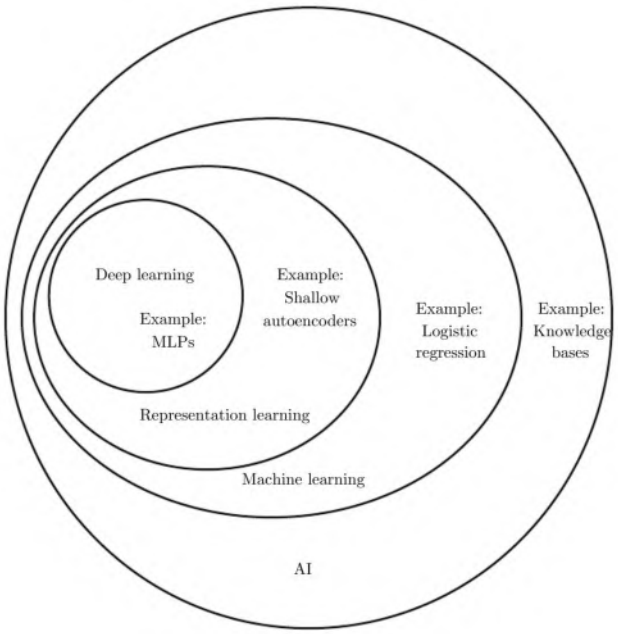


Abbildung 2.9: Einordnung ML, AI, DL.[5]

das in Abb. 2.9 dargestellte Mengendiagramm. Dieses zeigt, dass DL eine Unterdisziplin von Representation Learning ist, welches wiederum eine Unterdisziplin von ML darstellt und all diese Themengebiete unter dem Begriff AI zusammengefasst werden. Diese Arbeit befasst sich dabei explizit mit den Methoden des Deep Learnings, wofür ein Beispiel Multi Layer Perceptrons (MLPs) sind, die wiederum eine Art DNN bilden. Der Grundgedanke von AI beinhaltet, menschliche Intelligenz künstlich in Anwendungen, Systemen oder Prozessen zu replizieren. Zur Abbildung des Prozesses der visuelle Wahrnehmung lernen AI System, Muster (*Patterns*) und Eigenschaften (*Features*) in Datensätzen -in diesem Fall Bildern- zu erkennen und diese in einer bedeutungsvollen weise zu kombinieren. Diese Fähigkeit wird auch als maschinelles Lernen bezeichnet. Speziell die in Datensätzen hinterlegten Muster und Eigenschaften zu detektieren, ist wie der Name impliziert, dass Ziel von Representation Learning. Diese Arbeit verwendet speziell Deep Learning Verfahren, welche als eine Weiterentwicklung von Representation Learning betrachtet werden können. Deep Learning Architekturen verwenden als Universalbausteine, einzelne *künstliche Neuronen*. Sobald diese Neuronen in mehr als einer Schicht, sogenannten *Layern* verwendet werden, wird von einem Deep Neural Network DNN gesprochen.

Ein solches Neuron, beispielhaft dargestellt in Abb. 2.10, besteht aus anpassbaren Gewichten w_0 bis w_2, auch *Weights* genannt. Diese Gewichte werden mittels einer Linearkombination

mit den Eingangswerten x_i multipliziert und das Neuron führt auf das skalare Ergebnis z eine Aktivierungsfunktion $a(z)$ aus, deren Ergebnis der Ausgabewert des Neurons $\hat{y}$ darstellt.

Unter Verwendung von *Gradient Descent* basierten Verfahren wie beispielsweise *Linear Regression* oder *Logistic Regression* werden die Gewichte so optimiert, dass die bestmögliche Abbildung zwischen den Eingangsdaten x_i und der Ausgangsdaten y_i den *Ground Truths* gefunden wird. Detailliert werden die Konzepte der *Regression* sowie die unterschiedlichen Modell Architekturen in den folgenden Abschnitten erläutert.

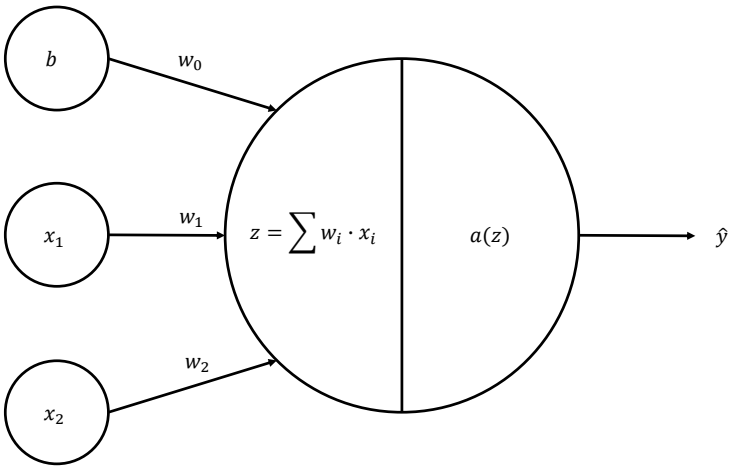


Abbildung 2.10: Schematische Darstellung eines künstlichen Neurons mit den inbegriffenen Gewichte w_i, Bias b und Aktivierungsfunktion $a(z)$

2.5.1 Das ML-Paradigma

Der Unterschied zwischen dem ML-Paradigma und klassischen *Rule-Based-Systems* ist, wie der in Abschnitt 2.2 erläuterten Finite-Elemente-Methode, dass Rule-Based-Systems ein umfangreiches problemorientiertes Modell verlangen, welches mittels Eingangsdaten (Input) eine Lösung (Output) des Problems liefert. Wie anhand Abb. 2.5 gezeigt wird, bedarf es zur Erstellung eines solchen Modells, fundiertes Wissen über das physikalische Modell, welches das reale System abbildet, sowie dessen mathematische Modellierung und der anschließenden numerischen Verfahren zur Erzeugung einer Näherungslösung. Der ML Ansatz hingegen verlangt Eingangs- sowie Ausgangsdaten und versucht über Algorithmen ein Modell zu erstellen welches diese bestmöglich miteinander Verknüpft. Abb. 2.11 hebt in grau hervor, welche Teile der einzelnen Ansätze zuerst unbekannt sind. Im Detail, sind ML

Verfahren in der Lage, sogenannte *Mappings* und *Features* zu lernen. Klassische ML Verfahren wie ANNs, lernen die bestmögliche Abbildung (*Mapping*) zwischen *Input* und *Output*. *Representation Learning* Verfahren wie beispielsweise CNNs, lernen zusätzlich in den Daten versteckte Merkmale (*Features*) zu erkennen. Als Deep Learning wird bezeichnet, wenn in einem CNN weiterer *Layer* existieren, die in mehreren einzelnen Merkmalen (*simple features*) weitere hochwertigere Merkmale (*highlevel features*) erkennen. Zusammengefasst in Abb. 2.11 als *Additional abstact features*.

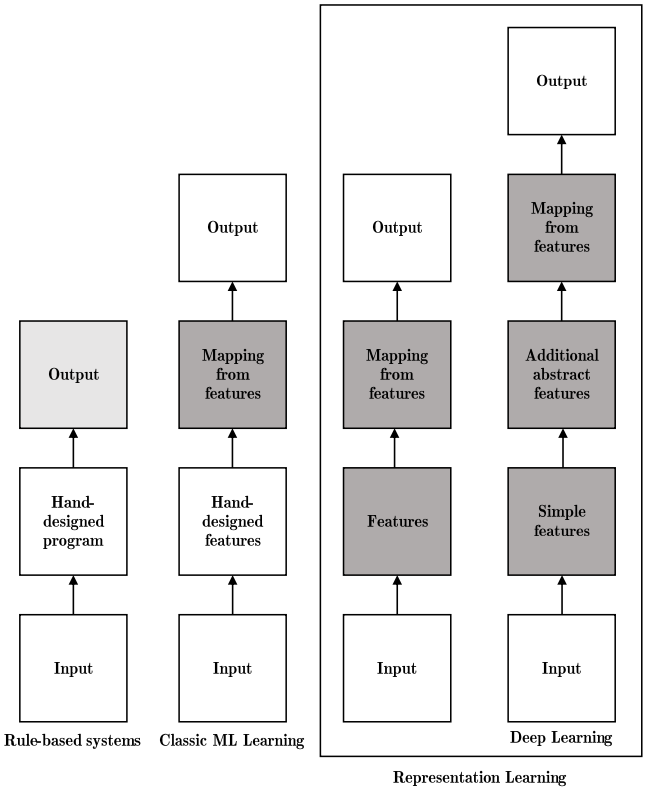


Abbildung 2.11: Ablaufpläne von Rule-Based-Systems (Regelbasierten Systemen) im Vergleich zu ML-Paradigmen. Dunkel Grau Markierte Felder zeigen Komponenten welche die Fähigkeit besitzen von den präsentierten Daten zu lernen.[5]

2.5.2 Fluch der Dimension

Mit der steigender Anzahl von Dimensionen die ein Problem aufspannt, steigt üblicherweise dessen Komplexität. Dieses Phänomen gilt insbesondere für Probleme im Bereich ML und ist bekannt als **Fluch der Dimension** (*Curse of Dimensionality*). Grundsätzlich

bedeutet dies, dass die Anzahl an benötigten Daten pro zusätzlicher Dimension exponentiell anwächst. Die Dimensionen sind in diesem Fall, unterschiedliche Merkmale (*features*). Im Idealfall deckt der Datensatz den Wertebereich jedes Merkmals vollständig ab, damit das ML Model generalisiert und verlässliche Vorhersagen über die ganze Bandbreite an Daten treffen kann. Wie Abb. 2.12 veranschaulicht, decken angenommen zehn verschiedene Werte die erste Dimension (*Links*) ausreichend ab. Mit zwei Dimensionen (*Mitte*) , werden schon $10 \times 10 = 100$ verschieden Wertepaare für eine akzeptable Abdeckung des Gebiets benötigt. Schlussendlich werden bei drei Dimensionen (*Rechts*), mindestens $10^3 = 1000$ an repräsentativen Trainingsdaten benötigt. Allgemeingültig formuliert bedeutet dies, damit v

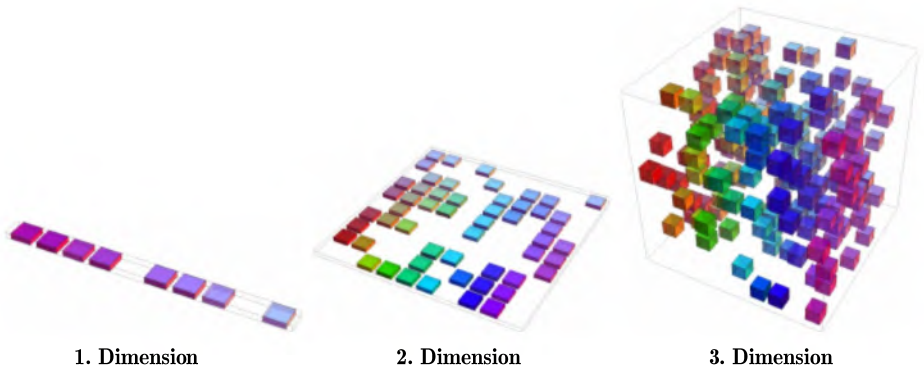


Abbildung 2.12: Fluch der Dimension: Die Abbildung zeigt schematisch die für eine ausreichende Abdeckung pro Dimension exponentiell zunehmende Anzahl an benötigter Sample.[5]

Werte in d Dimensionen unterschieden werden können, benötigt man $O(v^d)$ Trainingsdaten, damit das Model repräsentative Vorhersagen treffen kann.

2.5.3 Validierung-, Test- und Training-Datensatz

Wie das Phänomen *Curse of Dimensionality* zeigt, gibt es eine gewisse Grenze für die Menge an benötigten Trainingsdaten. Zusätzlich sollten die Stichproben im Idealfall gleichmäßig über den gesamten Wertebereich eines Merkmals verteilt sein, damit ein ML-Model das dem Datensatz zugrunde liegende Problem bestmöglich abbildet. Anhand eines solchen

[5] Entnommen aus Goodfellow [13]

Datensatzes trainiertes Model wird als generalisiert bezeichnet, wenn dieses ebenfalls für eine im Training zurückgehaltene Teilmenge des gesamten Datensatzes verlässliche Vorhersagen liefert. Diese zurückgehaltene Teilmenge des gesamten Datensatzes wird weiter in einen Validierung- und Test-Datensatz unterteilt. Diese werden zur Evaluation der Güte der Vorhersagen eines auf dem Trainings-Datensatz optimierten Modells verwendet. Die Stichproben im Validierung-Datensatz, werden innerhalb des Trainings-Prozesses nach jeder Epoche (Iteration) für die Evaluation des ML-Modells verwendet. Nach Abschluss des Trainings-Prozesses wird die Güte des trainierten Models anhand der im Test-Datensatz enthaltenen Stichproben evaluiert. Im Detail ist der Trainings-Prozess und die Aktualisierung der Gewichte pro Epoche im folgenden Abschnitt 2.5.4 zur Regression erläutert. Für die exakte prozentuale Aufteilung des gesamten Datensatzes gibt es keine einheitliche Regelung. Die optimale Aufteilung ist dabei dem Anwender überlassen. Als Anhaltspunkt sind in Abb. 2.13 übliche Unterteilungen dargestellt. Zu beachten ist bei der Aufteilung des

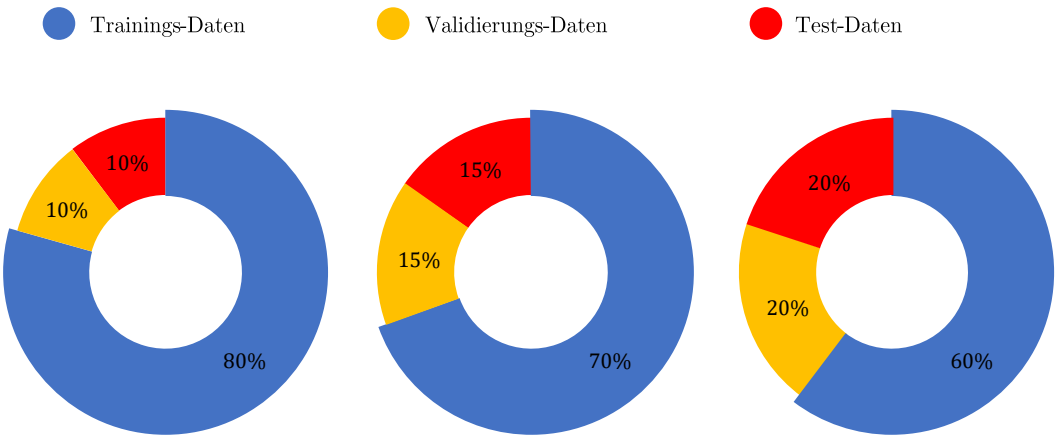


Abbildung 2.13: Häufig verwendete Aufteilungen von Validierung-, Test- und Training-Datensatz

gesamten Datensatzes, dass die darin enthaltenen Stichproben möglichst einer Gleichverteilung unterliegen. Dies bedeutet für Training-, Validierung- und Test-Datensatz, dass diese ebenso den gesamten Wertebereich der Merkmale abdecken müssen. Durch der Unterteilung vorangehendem zufälligen mischen (*shuffling*) der Stichproben des gesamten Datensatzes, kann dies garantiert werden. Wird ein ML-Modell auf einem zu großen Anteil des gesamten Datensatzes trainiert und besitzt deshalb schlechte Evaluationsergebnisse für Validierung- und Test-Datensatz, wird dies als *Overfitting* bezeichnet. Im Gegensatz dazu steht *Underfitting*, in welchem Fall das Modell mit einem zu geringen Anteil an Daten trainiert wird und schlechte Evaluationsergebnisse für den Trainingsdatensatz erzeugt. Die bereits

erwähnte Generalisierung eines Models bezeichnet den Schnittpunkt zwischen *Overfitting* und *Underfitting*, an welchem das Modell die bestmöglichen Ergebnisse für Validierung- und Test-Datensatz erzeugt.

2.5.4 Regression

Zu Beginn wurde das Neuron als Grundbaustein eines jeden NN vorgestellt. Die diesem Neuron zugrundeliegende Funktion, welche Eingangs- und Ausgangsdaten miteinander verknüpft, nennt man *Hypothesis*. Diese besitzt beispielsweise die Form

$$z = h(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2, \tag{2.34}$$

in der Gewichte w_i per Linearkombination mit den Inputs x_i multipliziert werden. Die Gewichte, welche nicht mit einem Input x_i multipliziert werden wie beispielsweise w_0, geben den Ursprung der Hypothese an und werden als *Bias* bezeichnet. Dabei kann die Form beliebig gewählt und auf das gegebene Problem angepasst werden. Mit vorhandenen Inputs x_i und zuerst beliebig gewählter Gewichte w_i, resultiert aus der *Hypothesis* $h(\mathbf{x}, \mathbf{w}) = z$ und ohne Aktivierungsfunktion $\hat{y} = z$ eine Vorhersage über das tatsächliche Ergebnis y, dem *Ground Truth*. Diese nennt man einen *Forward Pass* oder auch *Feed-Forward Process*. Nun sollen jedoch die Vorhersagen der Hypothesis $\hat{y} = h(\mathbf{x}, \mathbf{w})$ den tatsächlichen Zusammenhang zwischen Input $\mathbf{x}$ und dem Ground Truth y möglichst genau abbilden. Um den Fehler der Vorhersage zu quantifizieren, wird eine sogenannte *Loss-Function* $J(y, \hat{y})$ eingeführt, deren Ergebnis bei richtiger Vorhersage Null ergibt und fehlerhafte Vorhersagen durch sehr hohe Werte kennzeichnet. Eine solche Funktion wird auch *Cost* oder *Objective-Function* genannt. Denn für einen gegeben Input $\mathbf{x}$, liefert eine solche Funktion einen objektiven Wert oder anders ausgedrückt die Kosten (*Cost*) einer falschen Vorhersage, anhand dessen die Gewichte $\mathbf{w}$ optimiert und die Kosten anschließend minimiert werden. Die Optimierung der Gewichte nennt man *Feedback-Process* und wird beispielsweise mit einem der folgenden *Gradient Descent* Verfahren realisiert.

Gradientenabstiegsverfahren

Die Objective-Function zu optimieren, bedeutet das globale Minimum der durch die Gewichte aufgespannten *Loss-Landscape* zu finden. In den meisten Fällen ist jedoch die analytische Lösung der Ableitung nach den Gewichte einer Loss-Funktion aufgrund von zu vielen betrachteten Merkmalen oder Variablen nicht möglich. Dies führt wieder auf eine iterative, also numerische Lösung der Ableitung. Der Gradient kann mittels Finite Differenzen Methode (FDM)

$$\frac{\partial J}{\partial w_i} = \frac{J(w_1, w_2, \ldots, w_i + \Delta w_i \ldots, w_n) - J(w_1, w_2, \ldots, w_i, \ldots, w_n)}{\Delta w_i} \tag{2.35}$$

berechnet werden. Dies ermöglicht die Berechnung des Gradienten, jedoch erschweren auch hierbei große Input Vektoren $\mathbf{x}$ die Berechnung sehr. Aus diesem Grund werden bei dem Gradientenabstiegsverfahren (*Gradient-Descent-Verfahren* zuerst zufällige Werte für die Gewichte $\mathbf{w}$ gewählt und der Gradient an dieser Stelle bestimmt. Anschließend wird ein Schritt entgegen, also in Richtung des steilsten Abfalls (*steepest descent*) getätigt. Als skalare Gleichung formuliert

$$w_i^{(k+1)} = w_i^{(k)} - \alpha \frac{\partial}{\partial w_i} J(w_i^{(k)}), \tag{2.36}$$

oder in Vektorschreibweise

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(k)}), \tag{2.37}$$

beinhaltet der *Gradient Descent* Algorithmus den Parameter α, welcher dessen Lernrate (*Learning Rate*) vorgibt. Abb. 2.14 zeigt hierbei, welchen Einfluss die Lernrate auf das finden des Minimums der Parabel hat. Eine niedrige Lernrate von $\alpha = 0.1$ hat eine

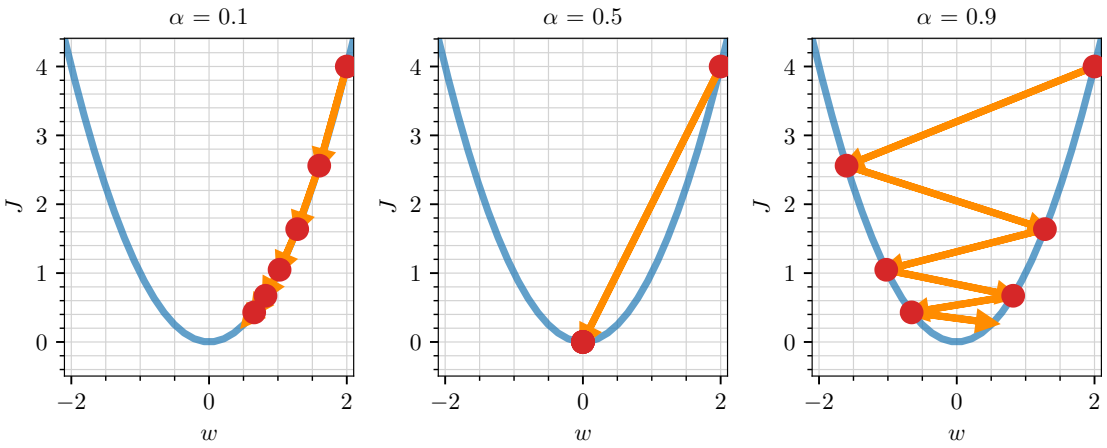


Abbildung 2.14: Dargestellt sind die Unterschiede in der Learning Rate α. Eine niedrige Learning Rate von $\alpha = 0.1$ bewirkt eine stetige Konvergenz entgegen des Minimums, die Learning Rate von $\alpha = 0.5$ stellt einen Sonderfall dar, welcher zum direkten erreichen des Minimums der Parabel führt. Eine zu große Learning Rate führt zum überschießen des Minimums und Oszillationen um das Minimum.

langsamen Annäherung an das Minimum zur Folge. Die mittlere Grafik repräsentiert hier

einen Sonderfall, da für eine Parabel eine Lernrate von $\alpha = 0.5$, von jedem Startpunkt aus innerhalb eines Schrittes auf das Minimum führt. Eine zu große Lernrate von beispielsweise $\alpha = 0.9$ führt zu überschießen (*overshooting*) des Minimums. Allgemein gilt, je weiter der Ausgangspunkt von einem Minimum entfernt ist, desto größer ist der Gradient und umgekehrt. Große Lernraten ermöglichen es lokalen Minima zu entkommen, oder gar zu überspringen. Niedrige Lernraten resultieren in stetiger Konvergenz und können zum feststecken in suboptimalen lokalen Minima führen. In den meisten fällen wird der Gradient einer Loss Funktion nie exakt Null ergeben. Dies ist dem relativen Approximationsfehler durch runden auf Gleitkommazahlen (*Floating Point Arithmetic*), sowie komplexer (*ragged*) Loss-Landscapes geschuldet. Die Definition einer Fehler Grenze wie beispielsweise $\epsilon = 10^{-5}$ beendet den Algorithmus, sobald ein gewähltes Abbruchkriterium

- $\|\mathbf{w}^{(k+1)} - \mathbf{w}^{(k)}\| \leq \epsilon$
- $\|J(\mathbf{w}^{(k+1)}) - J(\mathbf{w}^{(k)})\| \leq \epsilon$
- $\|\nabla_{\mathbf{w}} J(\mathbf{w}^{(k)})\| \leq \epsilon$

erfüllt ist.

Gradient-Descent-Varianten

Es wird zwischen drei Vorgehensweisen unterschieden:

- Batch-Gradient-Descent: Hier wird der gesamte Trainingsdatensatz für die Berechnung der Loss Funktion verwendet, der Mittelwert über die Gradienten errechnet und anschließend die Aktualisierung der Gewichte durchgeführt. Durch die Bildung des Mittelwerts über die Gradienten resultiert ein stabiler Verlauf in Richtung der optimalen Lösung.
- Stochastic-Gradient-Descent: Hier wird für jede einzelne Stichprobe (*Sample*) im Trainingsdatensatz die Loss Function berechnet und eine Aktualisierung der Gewichte durchgeführt. Die Berechnung der Gradienten für jedes Sample kann zu Oszillationen im Verlauf des Loss führen.
- Mini-Batch-Gradient-Descent: Ist eine Kombination der ersten beiden Vorgehensweisen. Hier wird für eine Teilmenge (*Mini-Batch* oder einfach *Batch*) des gesamten Trainingsdatensatzes die Loss-Funktion berechnet und eine Aktualisierung der Gewichte durchgeführt. Die Größe der Mini-Batches wird meist als eine Potenz von zwei gewählt, um diese auf die Catch-Größe des Computers abzustimmen.

Damit der Gradient-Descent-Algorithmus das tatsächliche globale Minimum einer komplexen Loss-Landscape findet und die Lernrate individuell anpassen kann, existieren weiterentwickelte Gradient-Descent-Algorithmen. Im Folgenden werden die drei wichtigsten Weiterentwicklungen vorgestellt:

Momentum: Speziell Stochastic Gradient Descent (SGD) und Mini-Batch Gradient-Descent sind anfällig für Oszillationen, verursacht durch Ausreißer in den Stichproben, die nicht entlang eines stetigen Pfades in Richtung des globalen Optimums führen. Dem Algorithmus fehlt somit die Fähigkeit geradlinig dem Impuls (*Momentum*) der Gradienten zu folgen. Die Hinzunahme eines Teils γ einer vorangegangen Aktualisierung der Gewichte löst dieses Problem und führt auf

$$\begin{aligned} \Delta \mathbf{w}^{(k+1)} &= \gamma \Delta \mathbf{w}^{(k)} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(k)}) \\ \mathbf{w}^{(k+1)} &= \mathbf{w}^{(k)} - \Delta \mathbf{w}^{(k)} \end{aligned} \tag{2.38}$$

Der Term $\gamma \Delta \mathbf{w}^{(k)}$ wird auch als das Momentum bezeichnet und der Parameter γ gibt gleich der Lernrate, die Rate des Momentums an der Aktualisierung an. Der Wert des Momentums steigt für die Dimensionen (*Features*) deren Gradienten in eine gemeinsame Richtung zeigen und wird für abweichende Gradienten gemindert. Dies führt auf geringere Oszillationen und schnellere Konvergenz.

AdaGrad: Die vorangegangen Algorithmen besitzen für alle Gewichte die gleiche konstante Lernrate α. Wie der Name Adaptive Gradient Approach (AdaGrad) impliziert, sorgt eine individuelle Anpassung der allgemeine Lernrate α an die jeweiligen Gewichte für eine erhöhte Effizienz. Diese wird durch den Betrag der kumulierten Gradienten $G_i^{(k)} = \sqrt{(g_i^{(1)})^2 + (g_i^{(2)})^2 + \cdots + (g_i^{(k)})^2}$ vorangegangener Iterationsschritte individuell skaliert. Mit den partiellen Ableitung $g_i^{(k)} = \nabla_{w_i} J$ und dem Aktualisierungsalgorithmus

$$w_i^{k+1} = w_i^k + \frac{\alpha}{\sqrt{G_i^{(k)} + \xi}}. \tag{2.39}$$

ξ soll das Teilen durch Null verhindert werden. Die für jedes Weight individuelle Skalierung von α, verhindert große Unterschiede in den Aktualisierungen, die einerseits zu Oszillationen und anderseits zu sehr geringfügigen Verbesserungen führen können. Jedoch Besitz auch dieses Verfahren einen Nachteil, denn der Skalierungsfaktor $G_i^{(k)}$ wird mit zunehmender Anzahl an Iterationen sehr groß und führt zu sehr kleinen Schrittweiten in der Aktualisierung, was eine schnelle Konvergenz erfordert.

Adam: Der Adaptive Moment Estimation (Adam)-Algorithmus verbindet die Vorteile der beiden vorangegangen Algorithmen.

$$\begin{aligned} \mathbf{m}^{(k+1)} &= \gamma_1 \mathbf{m}^{(t)} + (1 - \gamma_1) \nabla_{\mathbf{w}} J^{(t)} \\ \mathbf{v}^{(k+1)} &= \gamma_2 \mathbf{v}^{(t)} + (1 - \gamma_2) (\nabla_{\mathbf{w}} J^{(t)})^2 \\ \hat{\mathbf{m}} &= \frac{\mathbf{m}^{(k+1)}}{1 - \gamma_1} \;\; 0 \leq \gamma_1 \leq 1 \\ \hat{\mathbf{v}} &= \frac{\mathbf{v}^{(k+1)}}{1 - \gamma_2}, \;\; 0 \leq \gamma_2 \leq 1 \\ \mathbf{w}^{(k+1)} &= \mathbf{w}^{(k)} - \alpha \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}} + \xi} \end{aligned} \tag{2.40}$$

$\mathbf{m}^{(k+1)}$ ist das Momentum erster Ordnung und repräsentiert den exponentiell abfallenden Mittelwert vom Gradienten $\mathbf{g}^{(k)} = \nabla_{\mathbf{w}} J$ (*Mean*). $\mathbf{v}^{(k+1)}$ ist das Momentum zweiter Ordnung und repräsentiert den exponentiell abfallenden Mittelwert von $(\mathbf{g}^{(k)})^2$ (*Variance*).

Die beiden Hauptanwendungsgebiete von Gradient Descent Verfahren sind *Logistic* und *Linear Regression*. Beide Verfahren unterscheiden sich alleine in deren Aktivierungsfunktion und der daran geknüpften Objective Function.

Lineare Regression

Die Lineare Regression (*Linear Regression*) verwendet grundsätzlich Inputs und Outputs die jede beliebige reale Zahl besitzen können. Als Objective-Function hat sich aus diesem Grund die Least Mean Square (LMS) Cost-Function

$$J(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2N} \sum_{i}^{N} (y^{(i)} - \hat{y}^{(i)})^2 \tag{2.41}$$

etabliert. N is die Anzahl der Samples die dem Algorithmus als Trainingsdaten dienen. Für die lineare Hypothesis mit einer Eingangsgröße von N_w

$$\hat{y} = h(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j}^{N_w} x_j w_j \tag{2.42}$$

ergibt sich dann der Gradient zu

$$\frac{\partial J}{\partial w_j} = \frac{1}{N} \sum_{i}^{N} (y^{(i)} - \hat{y}^{(i)}) x_j. \tag{2.43}$$

Das Update der Gewichte wird hier mit dem klassischen Gradient Descent Verfahren Gl. 2.37

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(k)}),$$

berechnet und anschließend der *Loss* mit den neuen Gewichte

$$J(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{2N} \sum_{i}^{N} (y^{(i)} - w_0^{(i)} - \sum_{j}^{N_w} x_j^{(i)} w_j^{(i)})^2 \tag{2.44}$$

ausgewertet.

Logistische Regression

Ähnlich wie Lineare Regression verwendet die Logistische Regression (*Logistic Regression*), Inputs die jede beliebige reale Zahl besitzen können. Die Outputs werden hingegen durch eine nichtlineare Aktivierungsfunktion auf den Wertebereich $[0, 1]$ überführt. Die daraus resultierenden Outputs, werden als Wahrscheinlichkeiten interpretiert, inwiefern die Eingangsdaten zur Klasse 0 oder 1 gehören. Logistic-Regression löst somit ein Binäres Klassifikationsproblem (*Binary Classification Problem*). Ein Wert von 0.5 wird auch als Punkt maximaler Unsicherheit (*Point Of Greatest Uncertainity*) bezeichnet. Die Sigmoid Aktivierungsfunktion

$$\hat{y}^{(i)} = \sigma(z) = \frac{1}{1 + e^{-\beta z}} \tag{2.45}$$

ermöglicht dies. Abb. 2.15 zeigt den Verlauf der Sigmoid Funktion für unterschiedliche β Parameter. Fast ausschließlich wird dieser Parameter gleich eins gesetzt. Unter Verwendung

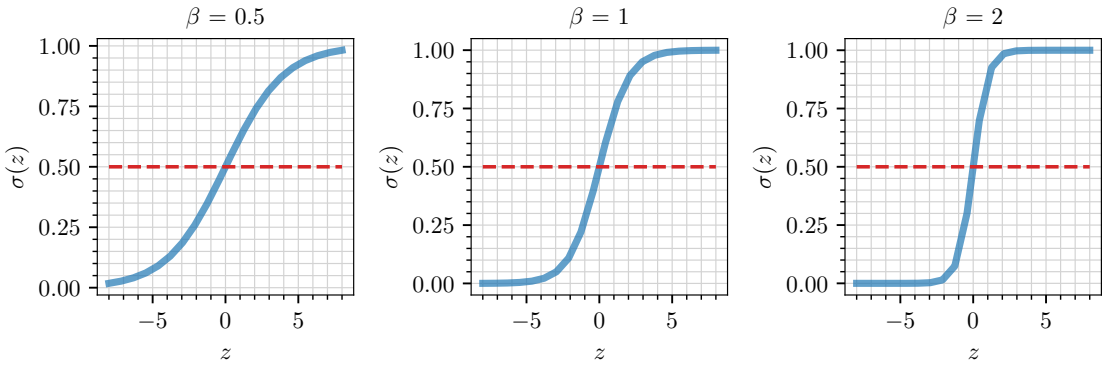


Abbildung 2.15: Sigmoid Aktivierungsfunktion für unterschiedliche Parameterwerte β.

der Sigmoid Funktion, ist die LMS Loss-Funktion nicht mehr ausreichend. Dies wird ersichtlich, wenn das Auftreten einer absolut falsche Klassifikation, $(y = 0, \hat{y} = 1)$ oder $(y = 1, \hat{y} = 0)$ betrachtet wird. Beide führen auf einen sehr geringen Wert von $J = 1/2N$. Eine dem Problem angemessene Loss-Funktion erfüllt jedoch die Bedingungen

- $J = 0$ für $y = \hat{y}$
- $J \gg 0$ für $y \neq \hat{y}$

- $J \geq 0$ muss immer gelten.

und kennzeichnet falsche Klassifikationen mit sehr hohem Loss-Wert. Aus diesem Grund wird für Binäre Klassifikationsprobleme, die *Binary Cross Entropy Cost Function*

$$J(\hat{y}, y) = -[y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y})] \tag{2.46}$$

verwendet. Für die Aktualisierung der Gewichte, wird wiederum einer der oben genannten Gradient Descent Varianten genutzt.

2.5.5 Metriken

Der Loss zwischen der Vorhersage $\hat{y}$ eines ML-Models und dem Ground Truth $\hat{y}$ welcher bei der Berechnung des Mean Square Error (MSE) für die Linear Regression oder der Binary Cross Entropy (BCE) im Fall von Klassifikationsproblemen entsteht, ist ein guter Indikator für die Leistung eines Models. Der Wertebereich der Loss-Funktionen ist jedoch nur ins negativen $J \geq 0$ begrenzt. Dies kann beispielsweise für nicht normalisierte Eingangswerte zu sehr großen Fehlern führen, welche nicht einfach zu interpretieren sind. Aus diesem Grund werden zusätzliche Metriken für die Überwachung des Trainingsprozesses und zur Evaluation verwendet, deren Ausgabewerte im Bereich $(0, 1)$ liegen.

R2-Score: Für Modelle die Optimierungsprobleme mit Regression lösen und reale Ausgangswerte Vorhersagen ist der *Coefficient of Determination* R^2 eine sehr wichtige Metrik. Der R2-Score lautet

$$R^2 = \frac{\text{SSR}}{\text{SST}} = 1 - \frac{\text{SSE}}{\text{SST}} \tag{2.47}$$

mit der gesamten Varianz Sum Square Total (SST)

$$\text{SST} = \sum_{i=1}^{N} (y_i - \bar{y})^2 \tag{2.48}$$

in den Trainingsdaten und der Varianz in der Vorhersage Sum Square Regression (SSR)

$$\text{SSR} = \sum_{i=1}^{N} (\hat{y}_i - \bar{y})^2. \tag{2.49}$$

Anstatt der Varianz in der Vorhersage SSR, kann auch der summierte und quadrierte Fehler der Vorhersage Sum Square Error (SSE)

$$\text{SSE} = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2. \tag{2.50}$$

verwendet werden. $\bar{y}$ ist der Mittelwert über die Vorhersagen. Vereinfacht ausgedrückt entspricht ein R2-Score von Null nach Drew Conway [6], einer Vorhersage für alle Outputs, die dem Mittelwert über alle Ground Truths entspricht. Dagegen entspricht eine perfekte Vorhersage für jeden Ausgangswert einem R2-Score von Eins.

F1-Score: Ähnlich dem R2-Score für Linear Regression ist der F1-Score für binäre Klassifikationsprobleme wie der Logistic Regression eine geeignetere Metrik zur Charakterisierung der Modell Vorhersage. Der F1-Score lautet

$$F1 = 2\frac{P \cdot R}{P + R} \tag{2.51}$$

und repräsentiert nach Murphy [21] und Goodfellow [13] den harmonischen Mittelwert der Genauigkeit über die Vorhersagen P (*Precision*) und dem Rückruf (*Recall*) der falschen positiven Werte in den tatsächlichen Werten (*ground truth*) R. Für den Fall der Logistic Regression kann P wie folgt formuliert werden

$$P = \frac{\sum_{i=1}^{N}(\hat{y}_i)}{\sum_{i=1}^{N}(y_i \cdot \hat{y}_i) + \epsilon}, \tag{2.52}$$

mit den wahren positiven Werten der Trainingsdaten im Zähler und allen positiven Werten der Trainingsdaten im Nenner. Dies gilt ebenfalls für

$$R = \frac{\sum_{i=1}^{N}(y_i)}{\sum_{i=1}^{N}(y_i \cdot \hat{y}_i) + \epsilon} \tag{2.53}$$

mit den wahren positiven Werten der Vorhersage im Zähler und allen positiven Werten der Vorhersage im Nenner.

2.5.6 Künstliche Neuronale Netzwerke

Das erste Rechenmodell für Artificial Neural Networks ANN (Künstliches Neuronale Netzwerke) entwickelten Warren McCulloch und Walter Pitts [19] schon Ende 1943 auf Basis Schwellenwert-Logik-Baustein (*Threshold-Logic*) basierter Algorithmen. Diese sollten das Verhalten neuronaler Aktivitäten sowie die Verknüpfungen untereinander adaptieren. Auch die in modernen Neuronalen Netzen verwendeten Universalbausteine sind nach dieser Herangehensweise von biologischen Neuronen inspiriert und besitzen wie zu Beginn im Abschnitt 2.5 vorgestellt, mehrere Inputs x_i die mit Gewichte w_i multipliziert und aufsummiert (*akkumuliert*) werden. Das Ergebnis der Multiply–Accumulate (MAC) Operation wird in eine Aktivierungsfunktion $a(z)$ eingesetzt, welche im Sinne einer *Threshold-Logic* den Ausgang des Neurons bestimmt. Die Optimierung der Gewichte erfolgt mittels eines der in Abschnitt 2.5.4 beschrieben Verfahren und Varianten von Gradient Descent. Hierbei folgen all diese Verfahren dem selben Ablauf:

1) Wähle initiale Gewichte $\mathbf{W}$

2) Mittels *Forward-Pass* und der gewählter Hypothese $\hat{y} = h(\mathbf{x}, \mathbf{w})$ wird die in $\mathbf{x}$ enthalten Information vorwärts durch das Netzwerk propagiert und ein Ausgabewert (*Vorhersage*) berechnet

3) Berechnung des Fehlers (*Loss*) zwischen der Vorhersage $\hat{y}$ und dem tatsächlichen Wert (*ground truth*) y durch Auswertung der *Loss Function* $J(y, \hat{y})$

4) Berechnung der partiellen Ableitung der *Loss Function* nach den Gewichte $\nabla_{\mathbf{w}} J(\mathbf{w}^{(k)})$

5) Aktualisierung der Gewichte $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(k)})$

Ist die analytische Formulierung des Gradienten der *Loss Function* bekannt, so ist dessen Lösung einfach. In vielen Fällen ist diese jedoch nicht bekannt und eine numerische Näherungslösung muss beispielsweise mit der FDM Gl. 2.35 berechnet werden. Eine nummerische Lösung ist speziell bei einer großen Anzahl von Gewichte und deren Aktualisierung sehr rechenintensiv. Dieses Problem tritt speziell bei Neuronalen Netzwerken NN im allgemeinen und ANNs auf, da diese aus mehreren Schichten, sogenannten *Layern* und einer darin enthaltenen großen Anzahl an Neuronen bestehen. Beispielhaft ist in Abb. 2.16 ein vollständig verknüpftes (*fully connected*) ANN abgebildet. Dieses Netz-

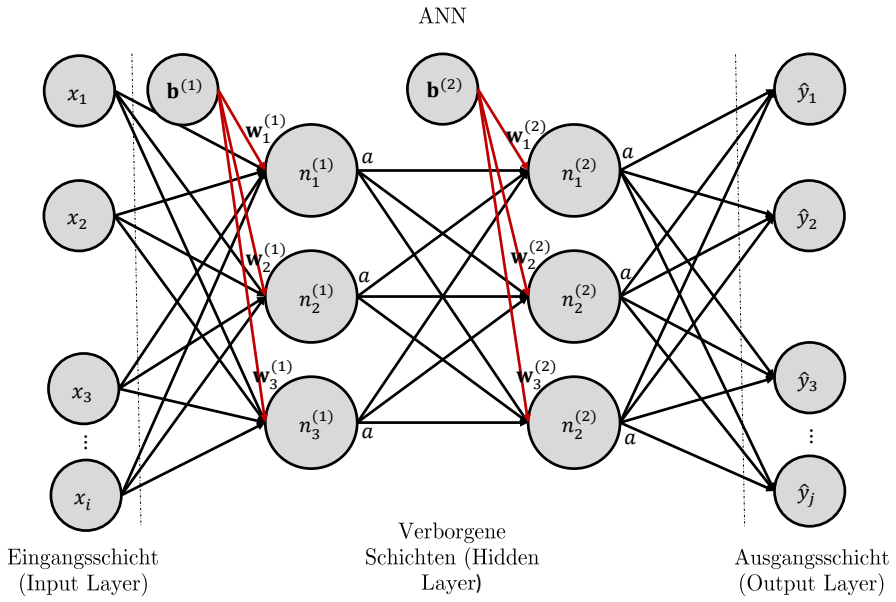


Abbildung 2.16: Beispiel für eine vollständig verknüpftes Artificial-Neural- Network mit zwei Hidden-Layer mit jeweils drei Neuronen. Die Anzahl der Inputs und Outputs in den Input und Outpu Layern kann theoretische beliebig groß sein und voneinander abweichen. Der Biasvektor ist darin explizit $\mathbf{b}_k^{(l)}$ dargestellt und nicht Teil der Weightmatrix $\mathbf{W}$

werk besteht dabei aus einer Eingangsschicht (*Input Layer*) mit $i \in (1, n)$ Eingängen $\mathbf{x} = [x_1, x_2, x_2, x_3, \ldots, x_n]$. Darauf folgen zwei verborgene Schichten (*Hidden Layer*), die

jeweils drei Neuronen besitzen. Die Neuronen bestehend aus drei, als Vektor, dargestellten Gewichte $w_k^{(l)}$ und einem (*Bias*) $\mathbf{b}_k^{(l)}$ Wert. Am Ende schließt eine Ausgangsschicht (*Output Layer*) $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \hat{y}_2, \hat{y}_3, \ldots, \hat{y}_j]$ mit $j \in (1, m)$ das Netzwerk ab. Zur Veranschaulichung is der Biasvektor Abb. 2.16 separiert dargestellt. Im weiteren Verlauf sind die darin enthaltenen Bias jedoch in den Gewichtungsvektoren $w_k^{(l)}$, an erster Stelle geführt $w_{1,k}^{(l)} = b_{1,k}^{(l)}$. Der Index k steht für das k-te Neuron in der Schicht l, das wiederum i Gewichte besitzt. Es wird von einem *Fully Connected* oder auch *Dense Neural Network* gesprochen, wenn alle Neuronen aus einem Layer mit jedem Neuron aus dem darauf folgenden Layer verknüpft sind. Zur Berechnung der einzelnen Gradienten und Aktualisierung der Gewichte wird die mathematische Methode der automatischen Differentiation verwendet, die im ML Kontext als *Back-Propagation* bekannt ist. Wie der Name suggeriert, wird nach Berechnung der Ausgangswerte $\hat{\mathbf{y}}$ diese Information vom Output Layer ausgehend durch das Netzwerk propagiert und somit die Gradienten der Gewichte der einzelnen Neuronen in den Hidden Layern berechnet. Die Rechenintensität der Back-Propagation ist im Vergleich zur FDM viel geringer, denn es müssen nicht für jede einzelne Aktualisierung eines Gewichte, die Loss-Funktion für zwei unterschiedliche initial Werte ausgewertet werden. Zur Berechnung der Gradienten wird die Information ausgehend der Eingangsschicht einmal durch einen Feed-Forward vorwärts und einen Feed-Back rückwärts durch das Netzwerk propagiert. Mittels der Kettenregel können die Gradienten im Gegensatz zur FDM ohne eine Approximation berechnet werden. Folgend wird Anhand Abb. 2.16 die Berechnung der gesuchten Gradienten $\partial J/\partial w_k^{(1)}$, $\partial J/\partial w_k^{(2)}$ zum besseren Verständnis im skalaren Fall ausgehend einer Vorhersage $\hat{y}_k$ schrittweise erläutert. Für eine einheitliche Notation gilt für die erste Schicht, $n_k^{(0)} = x$, $z_k^{(1)} = w^{(1)} n_k^{(0)}$, $n_k^{(1)} = a(z_k^{(1)})$, dies ist auf die zweite Schicht übertragbar und führt zuletzt auf $n_k^{(3)} = \hat{y}_k$. Zu erst wird der Gradient $\partial J/\partial w_k^{(2)}$ für ein Weight eines Neutrons der zweiten Schicht mittels der Kettenregel berechnet. Dieser ergibt sich

$$\frac{\partial J}{\partial w_k^{(2)}} = \frac{\partial J}{\partial n_k^{(3)}} \frac{\partial n_k^{(3)}}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial w_k^{(2)}} = -[y_k - \hat{y}_k] n_k^{(2)} \tag{2.54}$$

für a mit der Sigmoid Aktivierungsfunktion Gl. 2.45 und der für J dazugehörigen *Binary-Cross-Function* Gl. 2.46. Dieser setzt sich aus dem Fehler der Aktivierungsfunktion $\partial J/\partial z_k^{(3)} = -[y_k - \hat{y}_k] = -[y_k - n_k^{(3)}]$ und dem Ausgangswert eines Neurons der zweiten Schicht $n_k^{(2)} = z_k^{(3)}/\partial w_k^{(2)}$. Der Fehler in der Aktivierungsfunktion wird abgekürzt mit

$$\frac{\partial J}{\partial z_k^{(l)}} = \delta^{(l)_k}. \tag{2.55}$$

Analog ist die Formulierung mit der Kettenregel eines Gradienten für ein Weight eines Neurons in der zweiten Schicht

$$\frac{\partial J}{\partial w_k^{(1)}} = \underbrace{\frac{\partial J}{\partial n_k^{(3)}} \frac{\partial n_k^{(3)}}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial n_k^{(2)}} \frac{\partial n_k^{(2)}}{\partial z_k^{(2)}}}_{\frac{\partial J}{\partial z_k^{(2)}} = \delta_k^{(2)}} \underbrace{\frac{\partial z_k^{(2)}}{\partial w_k^{(1)}}}_{n_k^{(1)}} = \delta_k^{(2)} n_k^{(1)}. \tag{2.56}$$

Aus Gl. 2.54 und Gl. 2.56, für die Gradienten der ersten und zweiten Schicht kann geschlossen werden, dass für die allgemeine Formulierung gilt

$$\frac{\partial J}{\partial w_k^{(l)}} = \delta^{(l+1)} n_k^{(1)}. \tag{2.57}$$

Durch den vor der Back-Propagation durchgeführten Forward-Pass sind die Ausgangswerte der Neuronen $n_k^{(1)}$ bekannt. Hingegen ist anfangs nur der Aktivierungsfehler der einzelnen Neuronen in der letzten Schicht $\delta_k^3 = -[y_k - \hat{y}_k]$ bekannt. Unter einer weiteren geschickten Verwendung der Kettenregel ergibt sich für den Fehler in der Ausgangsschicht

$$\delta_k^{(3)} = \frac{\partial J}{\partial z_k^{(3)}} = \frac{\partial J}{\partial n_k^{(3)}} \frac{\partial n_k^{(3)}}{\partial z_k^{(3)}}, \tag{2.58}$$

sowie dem Aktivierungsfehler der Neuronen in der davor liegenden ersten Schicht

$$\begin{aligned} \delta_k^{(2)} &= \frac{\partial J}{\partial z_k^{(2)}} = \frac{\partial J}{\partial n_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial n_k^{(2)}} \frac{\partial n_k^{(2)}}{\partial z_k^{(2)}} \\ &= \delta_k^{(3)} \frac{\partial z_k^{(3)}}{\partial n_k^{(2)}} \frac{\partial n_k^{(2)}}{\partial z_k^{(2)}} \\ &= \delta_k^{(3)} w_k^{(3)} a'(z_k^{(3)}). \end{aligned} \tag{2.59}$$

Die rückwärts gewannte Berechnung der Fehler in der Aktivierungsfunktion lässt sich ebenfalls zu

$$\delta_k^{(l)} = \delta_k^{(l+1)} w_k^{(l)} a'(z_k^{(l)}) \tag{2.60}$$

verallgemeinern und die Gradienten jedes einzelnen Gewichten eines Neurons in einem mehrschichtigen neuralen Netzes ausgehend von der Ausgangsschicht mit den Gleichungen 2.57 und 2.60 berechnen.

Die vektorielle Schreibweise der Gleichungen 2.57 und 2.60 ist

$$\frac{\partial J}{\partial \mathbf{W}_{ik}^{(l)}} = \delta_i^{(l+1)} n_k^{(l)} \tag{2.61}$$

und

$$\boldsymbol{\delta}^{(l+1)} = [\mathbf{W}^{(l)} \boldsymbol{\delta}^{(l)}] \circ [a'(\mathbf{z}^{(l)})]. \tag{2.62}$$

Der Index i steht für das i-te Gewichte des k-ten Neuron der l-ten Schicht und $\boldsymbol{\delta}^{(l)}$ beinhaltet die Aktivierungsfehler aller Neuronen in der l-ten Schicht.

Die Back-Propagation wird rein zur Berechnung der Gradienten der einzelnen Gewichte genutzt. Daran anknüpfend müssen die Aktualisierungen wiederum mit einem der im Abschnitt 2.5.4 vorgestellten Gradient-Descent Verfahren durchgeführt werden.

Im Fall der Logistic-Regression wurde auch die Sigmoid-Aktivierungsfunktion für Klassifikationsprobleme eingeführt. Neben dieser existieren weitere gängige Aktivierungsfunktionen die in dieser Arbeit ihre Anwendung finden. Im Folgenden sind diese Aktivierungsfunktionen mit ihren Vor- und Nachteilen Aufgelistet.

Sigmoid: Bereits im Abschnitt 2.5.4 eingeführt, wird die Sigmoid Aktivierungsfunktion $\sigma(z)$ für binäre Klassifikationsprobleme verwendet, beispielsweise bei der Logistic Regression, indem der vorhergesagte reale Ausgabewert als Wahrscheinlichkeit interpretiert werden kann. Dies bedingt den Wertebereich $\sigma(z) \in [0, 1]$ des Ausgangs der Aktivierungsfunktion. Der Gradient $\sigma(z)' = \sigma(z)(1 - \sigma(z))$ kann bei tiefen neuronalen Netzen verschwinden. Dies ist auf die Sättigung (*Saturation*) der Grenzwerte 0 und 1 im Ausgang der Aktivierungsfunktion zurückzuführen.

tanh: Besitzen die Eigenschaften der Sigmoid Aktivierungsfunktion. Bildet den Ausgabewert der Vorhersage jedoch auf den Wertebereich $\tanh(z) \in [-1, 1]$ ab. Deshalb treten auch hier Sättigungsprobleme aufgrund der Grenzwerte -1 und 1 auf. Häufiges Anwendungsgebiet für die versteckten Schichten von CNN's und den finiten Ausgabewerten von Recurrent Neural Network (RNN)'s. Bei der Rückverfolgung mittels Back-Propagation tritt das Problem der verschwindenden Gradienten nicht so deutlich wie bei der Sigmoid Aktivierungsfunktion auf.

ReLU: Rectified Linear Unit (RelU) hat die Gestalt $f(x) = \max(0, x)$ und besitzt nur Sättigungsprobleme für negative Eingangswerte. Es müssen positive Initial-Gewichten (*Bias*) für die Existenz eines Gradienten und die daraus resultierende Funktion der Optimierung des ML-Models gewählt werden.

SeLU: Scaled Exponential Linear Unit (SelU) hat die Form

$$f(x) = \beta \begin{cases} x, & x > 0 \\ \gamma e^x - \gamma & x \leq 0 \end{cases} . \tag{2.63}$$

Mit den Parametern $\beta \approx 1.0507$, $\gamma \approx 1.6733$ und normalisierten Eingangsdaten (Mittelwert von Null und Varianz von Eins) werden die Gewichte mit der Aktualisierung über die Gradienten automatisch normalisiert. Normalisiert n Gewichte pro Schicht eines Neuronalen Netzwerkes mit einer Varianz von $1/n$. Besonders geeignet für CNNs, RNNs und DNNs.

2.5.7 Faltenden neuronale Netzwerke

Wie zu Beginn des Kapitels mit der Abb. 2.9 veranschaulicht wird, sind Representation-Learning und in Folge Deep-Learning Unterdisziplinen von ML, zu dessen vielfältigen Optimierungsverfahren und Modellen gehören die bereits eingeführten ANN's. Die Klasse der CNNs (*Faltenden neuronale Netzwerke*) ist auf Grund ihres Anwendungsgebiets, bestehend aus Bildverarbeitungsaufgaben, im Bereich des Deep-Learnings einzuordnen und ist

eine Unterklasse der ANN's. Vollständig verknüpfte ANN's wie in Abb. 2.16 sind ebenfalls in der Lage Bild basierte Aufgaben zu Lösen. Die Eingangsgröße eines beispielsweise sehr gering aufgelösten Graustufenbildes mit 128×128 Pixel fordert alleine in der ersten Schicht eines ANN-Models $128 \times 128 + 128 = 16512$ Gewichte, Bias miteinbezogen. Verwendet man ein Rot-Grün-Blau (RGB)-Bild welches aus drei Farbkanälen (*Channel*) besteht, so repräsentiert jede Farbe eine eigene Bildmatrix mit 128×128 Einträgen, respektive Pixeln und lässt die Anzahl der alleine für den Input benötigten Gewichte auf $16512 \times 3 = 50 \cdot 10^3$ anwachsen. Möchte man in einem solchen Bild beispielsweise ein Klassifikationsproblem lösen und die Zahlen null bis neun aus dem MNIST-Datensatz mittels Logistic-Regression vorhersagen, wird mindestens eine weitere Schicht mit Zehn Neuronen benötigt,die das entsprechende Bild mit den zehn Klassen abbildet. Eine solche zweite Schicht lässt die Anzahl an zu optimierenden Gewichte auf $50\dot{1}0^4$ Anwachsen. Dieses sehr einfache Beispiel zeigt, dass Bild basierte Aufgaben mittels eines ANN möglich, aber sehr rechenintensiv und daher ineffizient sind. Das erste CNN-Model, von Kunihiko Fukushima [12] im Jahre 1980 unter dem Namen *Neocognitron* entwickelt, basiert auf der 1986 veröffentlichten Forschungsarbeit von Hubel und Wiesel [7], die in den 1950er Jahren entdeckten, dass spezielle visuelle Gehirnzellen auf Teilgebiete des Sichtfeldes von Katzen reagieren ohne das der visuelle Fokus auf diese Teilgebiet gelegt wurde. Ein solches Teilgebiet, dass zu der Aktivierung eines visuellen Neurons führt, heißt rezeptives Feld. Eine Kombination dieser Rezeptiven Felder, ergibt das kontralaterale visuelle Gesamtbild des Sichtfelds. Hubel und Wiesel konnten dabei zwei Typen von visuellen Zellen identifizierten, deren Funktionsweise seit *Neocognitron* von modernen CNN's adaptiert wird.

- Simple-Cells: reagieren linear auf elementare geometrische Merkmale, wie Kanten
- Complex-Cells: führen eine Linearkombination der Ausgänge von *Simple Cells* durch und liefern Merkmale mit erhöhtem Informationsgehalt.

Kunihiko Fukushima [12] überträgt diese Funktionsweise der visuellen Zellen auf *Convolutional-* und *Pooling-Layer*, die zwei Basis-Schichten eines jeden modernen CNN's.

Convolutional Layer

Eine Convolution oder Faltung, ist ein mathematischer Operator, welcher die Überlappung einer stationären und einer instationären Funktion wiedergibt. Im Fall von CNN's werden anstatt von Funktionen, die Überlappung zweier diskreter gewichteter Matrizen gemessen. Die in-stationäre Matrix wird dabei als Filter oder Kernel bezeichnet und dessen Dimensionen entsprechen dem rezeptiven Feld einer visuellen Zelle. Die Fähigkeit von Simple Cells, Kanten zu detektieren, ist beispielsweise die erste Convolution von vielen in einem CNN in Schichten aneinander gereihter Convolutions deren detektierte Feature pro Schicht an Informationsgehalt zunehmen. Welches Feature die einzelnen Kernel detektieren, wird rein durch die Optimierung der Gewichte der einzelnen Kernel realisiert und müssen nicht vom Anwender definiert werden. Dies ist der kennzeichnende Vorteil von CNN's. In Abb.

2.17 ist beispielhaft der filter Prozess eines 2×2 Kernel über eine 4×4 Input Matrix dargestellt. Der Kernel durchfährt dabei von rechts oben ausgehend die Input Matrix mit einer Schrittweite (*Stride* von $s = 1$ und führt wie die Hypothese innerhalb der Neuronen eines ANN, eine MAC-Operation zwischen den Gewichte $\mathbf{W} = [[w_1, w_2], [w_3, w_4]]$ und dem aktuellen Teilgebiet der Input Matrix aus. Die daraus resultierenden *Feature-Maps* werden anschließend zu einem Output zusammengesetzt. Die Richtung in welcher der Kernel in Abb. 2.17 über die Eingangsmatrix gezogen wird

Richtung 1. Rot→Rot-Blau→Blau
→Rot-Gelb→Mitte→Blau-Grün
→Gelb→Gelb-Grün→Grün

Richtung 2. Grün→Gelb-Grün→Gelb
→Blau-Grün→Mitte→Rot-Gelb
→Blau→Rot-Blau→Rot

verändert das Ergebnis nicht. Dies ist mit der translatorischen Invarianz der Faltung zu begründen und ist eine besondere Eigenschaft der Convolution. Des Weiteren müssen zur Analyse der Gesamten Eingangsmatrix nur die Gewichte eines Kernels pro Kanal (*Channel*) trainiert werden. Im Beispiel der Kantendetektion, reicht daher ein einziger Kernel aus um alle in einem Bild enthalten Kanten zu erkennen. Diese Eigenschaft nennt man auch *Parameter-Sharing*. Bei einer Schrittweite $s = 1$ in horizontaler und $t = 1$ in vertikaler

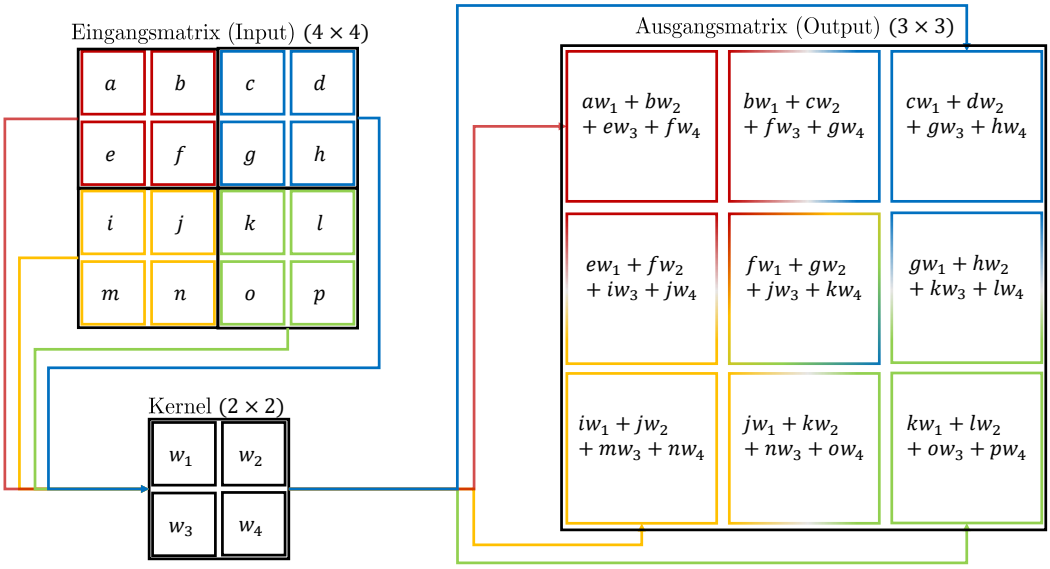


Abbildung 2.17: Beispiel für eine 2D Convolution.

Richtung, einer Eingangsmatrix mit den Dimensionen $n \times m$ und einem Kernel der Größe

$f \times g$, resultiert daraus eine Ausgangsmatrix der Größe

$$\frac{n-f}{s} + 1 \times \frac{m-g}{t} + 1. \tag{2.64}$$

Ein Grund die Schrittweite (*Stride*) zu erhöhen, ist die Rechenzeit bei sehr großen Matrizen zu reduzieren. Dies wird als *Strided-Convolution* bezeichnet. Weitere Arten von Convolutions sind

Volume Convolutions: Wie die Betrachtung eines RGB Bildes aufzeigt, sind 2D Convolutions wie in Abb. 2.17 für die meisten Anwendungen nicht ausreichend. Üblicherweise werden *Volume-Convolutions* auf dreidimensionale Matrizen angewendet. Im Falle der drei Farbkanäle eines RGB Bildes, besitzt ein Kernel der Größe $f \times g$, die tatsächlichen Dimensionen $f \times g \times 3$. Für jeden Channel wird ein unabhängiger Kernel mit unabhängigen Gewichte trainiert. Aus K Kernels entstehen K *Feature Maps*, die in der darauffolgenden Schicht Kernel der Dimension $f \times g \times K$ verlangen.

Padded Convolutions: Wie Gl. 2.64 zeigt, führen aufeinander folgende Convolutions zu immer kleiner werdenden Matrizen (*Feature Maps*), bis eine Convolution nicht mehr möglich ist. Durch umschließen (*Padding*) mit Schichten von Nullen der kleiner gewordenen Feature Maps kann die ursprüngliche Größe erhalten werden. Die Dimensionen des Ausgangs eines Kernels mit *Padding* sind

$$\frac{n-f+2p}{s} + 1 \times \frac{m-g+2p}{t} + 1, \tag{2.65}$$

wobei $p = 1$ eine Schicht Nullen um die Ausgangsmatrix bedeutet. Mit der Wahl des *padding* Faktors $\frac{p-f}{2}$ bleiben die Eingangsdimensionen erhalten.

Dilated Convolution Diese Art der Convolution fungiert ähnlich wie *Padding*, mit dem Ziel die Berechnungsgeschwindigkeit gleich der *Striding-Convolution* zu erhöhen. Hierfür wird das rezeptive Feld des Kernels erweitert, indem Schichten an Nullen im Zentrum hinzugefügt werden, ohne die Anzahl der Gewichte zu erhöhen. Beispielsweise entspricht das rezeptive Feld eines 5×5 Kernels mit *Dilation-Factor* $d = 2$, einem Kernel der Größe 7×7.

Pooling

In einer klassischen CNN Layer folgen auf eine oder mehrere der vorgestellten Convolutions, zwei weitere Schritte die deren Output weiter verändern. Nach der MAC-Operation des Kernel auf ein Teilgebiet wird dessen Output gleich einem *Artificial-Neuron* mittels einer nicht-linearen Aktivierungsfunktion weiter verändert. Im dritten Schritt führt Pooling eine statistische Summierung (*subsampling*) ähnlich dem Kernel auf Teilgebiete der aktivierten Feature Map aus. Die meist verwendeten Typen sind *Max-Pooling* und *Average-Pooling*. Beispielsweise reduziert eine Pooling Operation mit der Größe 8×8 bei einer Schrittweite

von $s = 1$ eine Feature Map der Größe 128 um exakt den Faktor Acht. Zur Berechnung der Ausgangsgröße nach einer Pooling Operation ist die Gl. 2.64 äquivalent anwendbar. Die Kerneigenschaft von Pooling ist die aus ihr resultierende translatorische Invarianz, die in einem CNN pro Schicht zunimmt.

Max-Pooling: Im Fall von Max-Pooling bewegt sich eine Pooling-Matrix entsprechend eines Kernels über die Input-Matrix hinweg und fügt die lokalen Maxima der Reihe nach, zu einer neuen Ausgangsmatrix zusammen. Beispielhaft dargestellt in Abb. 2.18

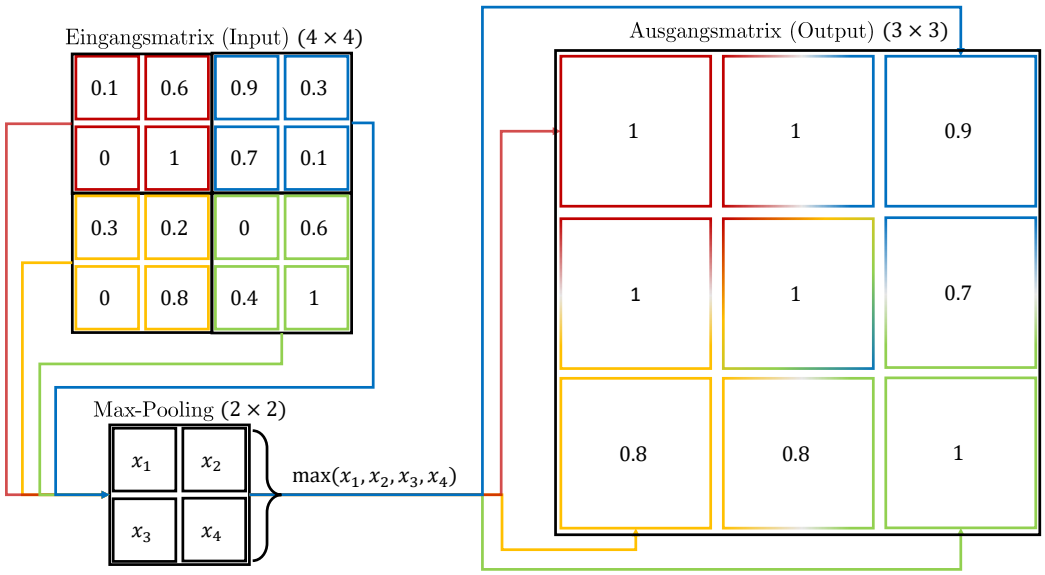


Abbildung 2.18: Beispiel für 2D Max Pooling mit Schrittweite $s = 1$.

Average-Pooling: Analog bildet in diesem Fall die Pooling Matrix den Mittelwert ihres rezeptiven Feldes, welche sich ebenfalls Schrittweise über die Input Matrix bewegt. Beispielhaft dargestellt in Abb. 2.19

Spätestens seit 2010 und dem Beginn des ImageNet Software Contest [9], haben ich CNN's bei Bild basierten Aufgaben durchgesetzt. Dieser Erfolg ist den besonderen Eigenschaften von CNN's wie beispielsweise der Fähigkeit des *Parameter-Sharings*, sowie der durch Pooling eingebrachten translatorischen Invarianz zuzuschreiben. Für die Aktualisierung der Gewichte des Kernels reicht ebenfalls pro Epoche ein einziger Feed-Forward und Feed-Back durch das Netzwerk aus.

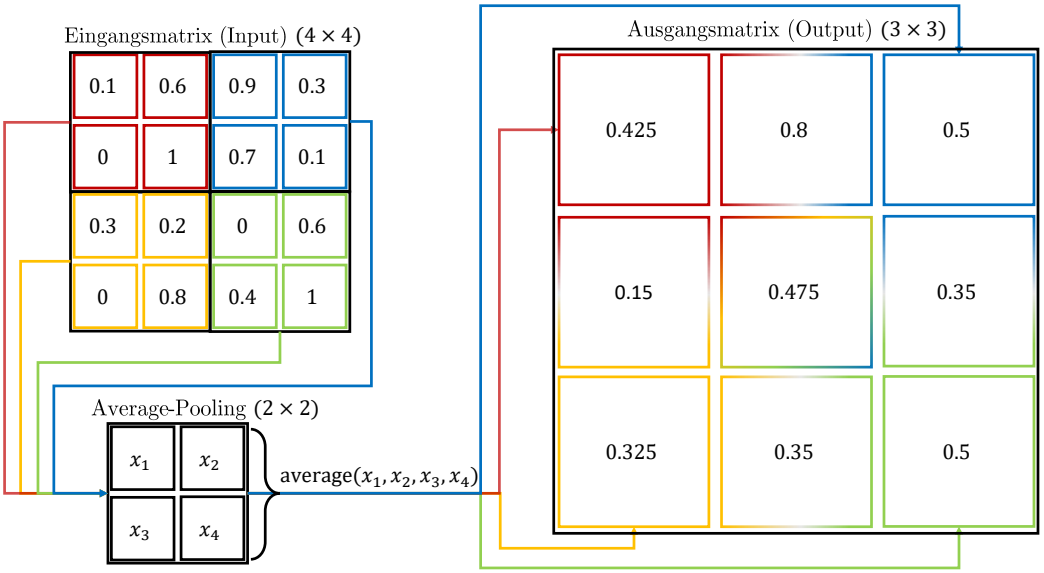


Abbildung 2.19: Beispiel für 2D Average Pooling mit Schrittweite $s = 1$.

2.5.8 Hyperparameter Optimierung

Die Initialisierung von Hyperparametern wie beispielsweise der in Abschnitt 2.5.4 eingeführten Lernrate α ist für eine optimale Vorhersage eines ML-Modells entscheidend. Dabei ist die Lernrate nur ein Parameter von Vielen. Beispielsweise kann die Anzahl von Neuronen und Schichten in einem Modell, die Wahl der Aktivierungsfunktion oder die Kernel Größe eines CNN in einer Studie als Hyperparameter gewählt und optimiert werden. Bei einer Großzahl an Parametern mit unterschiedlichen Wertebereichen die optimale Konfiguration zu finden, kann eine große Herausforderung darstellen, welche von den folgenden Herangehensweisen auf unterschiedliche Weise gelöst wird.

Hand Tuning: Die erste Herangehensweise ist die Wahl der Hyperparameter anhand der eigenen Erfahrung oder bereits erschienen Publikationen mit vielversprechenden Modellkonfigurationen. Ausgehend einer daraus hervorgehenden Hyperparameterkonfiguration, können schrittweise einzelne Parameter leicht verändert und das Modell neu evaluiert werden. Die manuelle Wahl erfordert jedoch ein tiefgehendes Verständnis und Erfahrung über die Eigenschaften der einzelnen Hyperparameter und deren Auswirkungen auf den Trainings-Fehler, die Generalisierung des Modells und benötigte Rechenleistung. In vielen Fällen ist eine der folgenden automatisierten Herangehensweisen empfehlenswerter.

Grid Search: Ohne jegliche Optimierungsstrategie wird bei dieser Vorgehensweise jede mögliche Parameterkombination der zuvor festgelegten diskreten Wertebereiche einzelner Hyperparameter ausgewertet. Kontinuierliche Wertebereiche mit einem breiten

Spektrum an möglichen optimalen Werten, die sich beispielsweise im Falle der Lernrate im Bereich $\alpha \in [10^{-5}, 10^{-1}]$ befinden können, stellen die größte Herausforderung dar. Eine logarithmische Skalierung

$$\alpha = 10^{\alpha_{\log}} \text{ mit } \alpha_{\log} \in [-1, -5] \tag{2.66}$$

löst diese Problem. Sehr breiten Wertebereiche, die eine Parametersuche auf mehreren Skalen gleichzeitig erfordern, werden somit komprimiert. Wenn eine gute Parameterkonfiguration am Rand des Rasters gefunden wird, kann eine Erweiterung des Suchspektrum empfehlenswert sein. Des Weiteren kann eine weitere Suche um eine gefunden Konfiguration mit verfeinertem Raster (*Grid*) sinnvoll sein. Großer Nachteil dieser Vorgehensweise ist die pro Hyperparameter exponentiell zunehmende Anzahl an zu erprobenden Konfigurationen.

Random Search: Diese Herangehensweise kann als eine verbesserte Variante von Grid Search betrachtet werden. In diesem Fall, werden die einzelnen Hyperparameter zufällig aus einer Wahrscheinlichkeitsverteilung gezogen. Für binäre oder diskrete Hyperparameter ist dies meist eine Bernoulli oder multinominale Verteilung. Für positive reale Werte wird in den meisten Fällen eine Gleichverteilung verwendet. Nach einer Studie von Bergstra und Bengio [2] kann Random-Search im Vergleich zu Grid Search ohne zusätzliche Rechenleistung annähernd exponentiell effizienter im finden optimaler Hyperparameter sein und zusätzlich größere Rasterspektren abdecken. Zusätzlich reduziert Random-Search im Vergleich den Validierungsfehler deutlich schneller, im Bezug auf trainierte Parameterkonfigurationen.

Bayes'sche Optimierung Eine der elegantesten und effizientesten Herangehensweisen ist es, ML-Verfahren zur Vorhersage der besten Hyperparameterkonfiguration zu nutzen. Wie bei Optimierungsproblemen üblich, wird der Gradient zur Berechnung einer Parameteraktualisierung benötigt. Die Formulierung einer Funktion welche den Gradient bezogen auf den Fehler im Validierungsdatensatzes beschriebt, ist in viele Fällen aufgrund der diskreten Wertebereiche der Hyperparameter nicht möglich. Für die Optimierung von Datensätzen ohne bekannte Objective-Function $J(\mathbf{y} - \mathbf{W}^T\mathbf{x})$ sind speziell Bayes'schen Optimierungsalgorithmen geeignet. Die Herangehensweise der Bayes'sche-Regression ist es, anzunehmen das der Validierungsfehler beispielsweise einer Gauß-Verteilung unterliegen. Angenommen der Fehler des verwendet Modells hat die Form

$$\varepsilon = (y_i - \mathbf{w}^T\mathbf{x}_i)^2 = (y_i - \hat{y}_i)^2, \tag{2.67}$$

dann lässt sich dessen Gauß-Verteilung

$$p(y_i \mid \mathbf{w}, \mathbf{x}_i) = p(\varepsilon) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{2\sigma^2}\right) \tag{2.68}$$

mit einem Mittelwert von Null und der Varianz σ^2 formulieren. $p(y_i \mid \mathbf{w}, \mathbf{x})$ wird nach der Wahrscheinlichkeitstheorie als Plausibilitätsfunktion (*Likelihood-Function*)

bezeichnet. Für einen Trainingsdatensatz mit M Stichproben ist die Plausibilität

$$p(y \mid \mathbf{w}, \mathbf{x}) = \prod_{i=1}^{M} p(y_i \mid \mathbf{w}, \mathbf{x}_i). \tag{2.69}$$

Über den Satz von Bayes (*Bayes-Rule*) wird die bedingte Wahrscheinlichkeit (Posterior)

$$p(\mathbf{w} \mid y, \mathbf{x}) = \frac{\prod_{i=1}^{M} p(y_i \mid \mathbf{w}, \mathbf{x}_i) p(\mathbf{w})}{\sum_{i=1}^{M} p(y_i, \mathbf{x}_i)} \tag{2.70}$$

berechnet. $p(y_i, \mathbf{x}_i)$ repräsentiert die Wahrscheinlichkeit den zugrundeliegenden Datensatz abzubilden. Die Gewichte $\mathbf{w}$ werden ebenfalls Stichprobenartig einer zugrunde liegenden Gauß-Verteilung mit Normalisierungsfaktor n

$$p(\mathbf{w}) = n \exp\left(\frac{-\|\mathbf{w}\|^2}{2\sigma^2 \mathbf{I}}\right) \tag{2.71}$$

entnommen. Diese Art der Gewichtinitialisierung hat den großen Vorteil, dass die Wahrscheinlichkeit für sehr große und kleine Werte der Gewichte aufgrund der Verteilung unwahrscheinlich ist und somit Overfitting beziehungsweise Underfitting der Feature $\mathbf{x}$ verhindert wird. Wie alle Regression-Verfahren ist auch Bayes'sche-Regression ein iteratives Verfahren, welches ausgehend von 2.71 stichprobenartig gezogenen Gewichte, mit den Gl. 2.69 und 2.70 die Gewichte immer weiter aktualisiert, bis die optimale Vorhersage gefunden wird. Die optimale Vorhersage wird dabei durch die Wahrscheinlichkeitsverteilung des Fehlers im Validierungsdatensatz,

$$p(y_{\text{val}} \mid x_{\text{val}}, x, y) = \int p(y_{\text{val}} \mid x_{\text{val}} \mathbf{w}) p(\mathbf{w}, x, y) \mathrm{d}\mathbf{w} \tag{2.72}$$

für $(x_{\text{val}}, y_{\text{val}})$ und den Trainingsdaten (x, y) bestimmt.

Moderne Bayes'schen Optimierungsalgorithmen sind Spearmint, SMAC und der in dieser Arbeit verwendete Tree-structured Parzen Estimater (TPE). Nach Bergstra [3] schneiden Sequential Model-Based Global Optimization (SMBO)-Algorithmen wie TPE bei der Hyperparameter Optimierung deutlich Besser ab als *Grid-Search* und *Random-Search*.

3 Software, Hardware, Preprocessing und Modellstruktur

In diesem Kapitel wird die Analyse und schrittweise Aufbereitung der bereitgestellten Daten sowie die ML-Model Struktur beschrieben. Die Aufbereitung der Daten, die Erstellung der ML-Model-Struktur, das Training und die anschließende Validierung und Visualisierung der Ergebnisse ist mit der High-Level Programmiersprache Python [28] realisiert. Für die Rekonstruktion eines VSBs wird die Python-Library `geomdl` [4] verwendet. Für einen Teil der Normalisierung der Daten und zur Unterteilung des Datensatzes in Trainings-, Validierungs- und Testdatensatz werden die Methoden der Python Library `scikit-learn` [24] verwendet. Die Verknüpfung der dreidimensionalen geometrischen Eingangsdaten der Triebwerkschaufelblätter mit den dazugehörigen zehn Eigenfrequenzen, erfordern die Erstellung eines DNN mit 3D Convolution- und Pooling-Schichten, die aus den Geometrien markante Features extrahieren und mittels vollständig verknüpfter Neuronen auf eine Ausgangsschicht mit zehn Ausgängen abgebildet werden. Die DNN-Struktur wird mit Keras der High-Level API des Python-Pakets Tensorflow [18] umgesetzt. TensorFlow ursprünglich als Backend entwickelt beinhaltet seit dem 2.0 Release die ebenfalls von Google entwickelte ML API Keras. Das Backend erzeugt einen Graphen des mit Keras generierten Models, anhand dessen ein Forward-Pass mit anschließender Back-Propagation durchgeführt werden kann. Die Hyperparamter-Optimierung wird mit dem Open-Source-Framework Optuna [1] umgesetzt.

3.1 Datengenerierung

Die für das Training und die Evaluierung des DNN-Models zur Verfügung gestellte Datenbasis, wurden mit dem Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR e.V.) eigenen Programmsystem GTlab [27] anhand Variation des in Abb. 3.1 dargestellten vorausgelegten Triebwerk-Typs Trent 1000 erzeugt. GTlab ist ein in C++ programmiertes objektorientiertes Software-Konzept, das eine plattformübergreifende Verknüpfung von Simulations- und Vorentwurfsumgebungen ermöglicht. Mit vorgegebener Triebwerkskenngrößen wie Schub F_n, Schubverhältnis Φ und Nebenstromverhältnis μ führt GTlab die integrierte Software Rubber-Engine [15] aus, welche aerodynamisch sinnvolle Parameter der einzelnen Triebwerksstufen berechnet. Advanced Compressor Design Code (ACDC) [25] erzeugt anschließend anhand der aerodynamischen Parameter die Profile der einzelnen

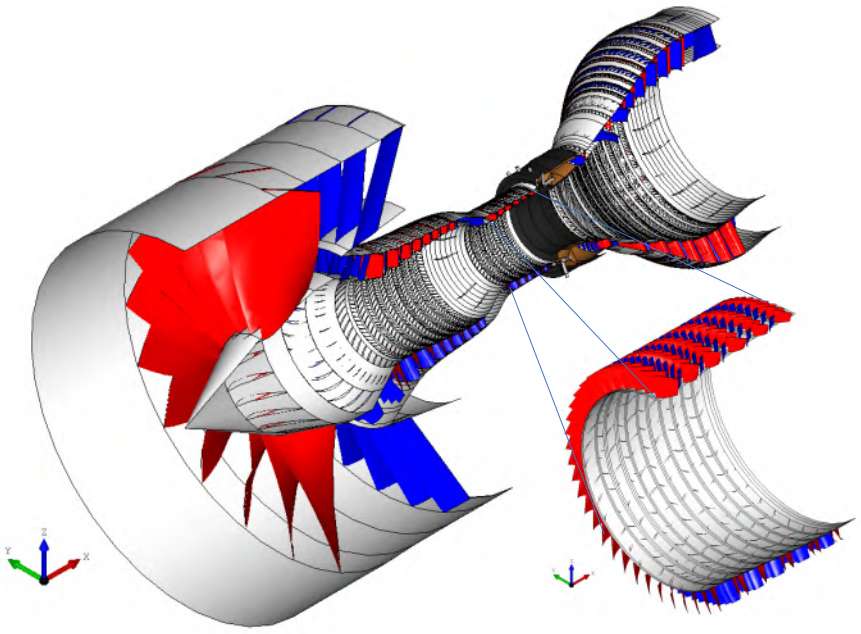


Abbildung 3.1: Variation des Triebwerk-Typs Trent 1000 mit Ausschnitt Hochdruckverdichters. Die Rotoren sind in rot und Statoren in blau dargestellt.

Triebwerksschaufelblätter wie beispielsweise die im Ausschnitt des Abb. 3.1 darstellten Verdichterschaufelblätter. Die darauf aufbauende Strukturmechanikberechnung und Modalanalyse der einzelnen in dieser Arbeit verwendeten VSBs erfolgt mit dem ebenfalls in GTlab integrierten FEM-Simulationsprogramm PERMAS. Die bereitgestellten Eingangsdaten (Inputs) entsprechen den Kontrollpunkten der durch ACDC erzeugten Oberflächen der sich im Verdichter befindlichen Rotor- und Statorschaufeln. Die dreidimensionalen kartesischen Koordinaten der Oberflächen beschreibenden Kontrollpunkte jedes VSBs wurden mit BladeGen als ASCII-Format (.dat) exportiert. Die mittels der FEM-Simulation berechneten, zu den VSBs korrespondierenden zehn Eigenfrequenzen (Outputs) sind als Hierarchical Data Format (HDF) bereitgestellt. Zusätzlich beinhaltet jede HDF-Datei die mit BladeGen errechnete Masse der einzelnen VSBs. Die auf diese Weise generierte Datenbasis umfasst jeweils für Statoren und Rotoren 39280 dat- und HDF-Dateien.

Beispielhaft ist für ein VSB im Anhang 1 ein Ausschnitt der darin enthaltenen ersten dreizehn Zeilen einer im ascii Format bereit gestellten Dateien dargestellt. Die erste Zeile gibt an, dass es sich um kartesische Koordinaten handelt. Die zweite Zeile verweist mit dem Parameter T = „NURB“ auf das Verfahren, mit welchem BladeGen die Oberfläche anhand der in den folgenden Zeilen dargestellten Koordinaten berechnet. Im nachfolgenden Unterkapitel 3.2.1 wird die Berechnung der Oberfläche mit den in Kapitel 2.3 eingeführten NURBS erläutert. Die Einheit der Koordinaten ist in Meter angegeben. Die Parameter I, J und K geben die Anzahl an Kontrollpunkten in den drei parametrisierten Richtungen an.

3.2 Preprocessing

In diesem Abschnitt wird die Aufbereitung (*Preprocessing*) der mit GTlab erzeugten Datenbasis beschrieben. Zuerst wird die Rekonstruktion der VSBO mit den vorgestellten NURBS und der Python Library `geomdl` [4] behandelt. Anschließend wird die Voxelization der VSBOs in Volumenmodelle mit dem Open-Source-Program Binvox von Patrick Min [20] durchgeführt und der entstehende Diskretisierungsfehler analysiert. Die daraus generierten 3D Voxelgitter werden als erstes Feature der einzelnen Sample in das DNN als `numpy.ndarray` eingelesen. Die vorangegangene Fehleranalyse hat zur Folge, dass die Information über die Verdichterschaufelblattvolumen (VSBV) als zweites Feature jedes Samples dem DNN zugeführt werden muss. Anschließend werden die Verteilung und die Wertebereiche der Volumen und Eigenfrequenzen analysiert. Die ungleiche Verteilung der Volumen und Eigenfrequenzen wird durch eine logarithmische Normalisierung, sowie eine MinMax-Skalierung optimiert. Das anschließende Binary-Encoding der Volumen in ein binäres 1D `numpy.ndarray`, hat eine gleichwertige Behandlung der beiden dem DNN zugeführten Feature zum Ziel und schließt das Preprocessing ab. Ein Trainingssample besteht anschließend aus einem binären 3D Voxelgitter und einem 1D Volumenvektor, die zusammen die Input-Daten bilden, sowie einem Ground-Truth mit den dazugehörigen zehn normalisierten Eigenfrequenzen. Am Ende des Abschnitts wird die Variable DNN-Architektur vorgestellt.

3.2.1 Rekonstruktion der VSBO

Die Eigenfrequenzen und das Schwingverhalten eines Körpers sind untrennbar mit dessen geometrischer Form verknüpft. In Kapitel 2.2 wurde dies mit der elementweisen integralen Berechnung der System beschreibenden Massen-, Steifigkeits- und Dämpfungsmatrix gezeigt. Damit die Information des dreidimensionalen geometrischen Zusammenhangs, der VSBs für das ML-Model nicht verloren geht, müssen die Oberflächen aus den bereitgestellten Koordinaten der Kontrollpunkte rekonstruiert werden. Äquivalent dem von BladeGen verwendeten Berechnungsverfahrens, werden die Oberflächen mittels der in Kapitel 2.3 vorgestellten NURBS rekonstruiert. Die Rekonstruktion der rationalen B-Splines, sowie der Knotenvektoren U, V für die beiden parametrisierten Richtungen u, v wird mit dem Open-Source Object-Oriented (NURBS) Modeling-Framework von Onur Rauf Bingol umgesetzt, welches unter der Bezeichnung geomdl als Python Library veröffentlicht [4] ist. Die Rekonstruktion jeder einzelnen VSBO durchläuft die folgenden Schritte:

1) Öffnen der dat-Datei im Python-Script

2) Auslesen der hinter den Parameterschlüssel I und K angegeben Anzahl an Kontrollpunkten für die verschiedenen parametrisierten Richtungen.

3) Auslesen der Kontrollpunkte und speichern als `numpy.ndarray`

4) Transformation des `numpy.ndarray` in eine Matrix `ctrlpts2d` der Dimension I×K.

5) Normalisierung der in `ctrlpts2d` gespeicherten Kontrollpunkte mit

```
scaler = preprocessing.StandardScaler(with_std=True,with_mean=True).fit(
    ↪ ctrlpts2d)
ctrlpts2d = scaler.transform(ctrlpts2d)
```

dem von `scikit-learn` bereitgestellten `StandardScaler` auf einen Mittelwert von Null und eine Standardabweichung von Eins.

6) Instanziieren eines Oberflächen Objekts `surf = BSpline.Surface()` des `geomdl`-Pakets

7) Festlegen der Polynom Grade für die beiden parametrisierten Richtungen `surf.degree_u` und `surf.degree_v`

8) Berechnung der Knotenvektoren

```
surf.knotvector_u = utilities.generate_knot_vector(surf.degree_u, surf.
    ↪ ctrlpts_size_u)
surf.knotvector_v = utilities.generate_knot_vector(surf.degree_v, surf.
    ↪ ctrlpts_size_v)
```

9) Anschließende Evaluation der berechneten Oberfläche `surf.evaluate()`

10) Speichern der Abmessungen und des Volumens der Bounding Box (BBX). Die BBX beschreibt die maximale Ausdehnung der Oberfläche in den drei Raumrichtungen.

11) Im letzten Schritt wird die berechnete Oberfläche als Standard Triangle Language (STL) Datei Format exportiert.

Die Notwendigkeit von Schritt 5) wird im folgenden Kapitel 3.2.2 und mit der Reduzierung des darin auftretenden Fehlers erläutert. Das `geomdl`-Paket bietet zusätzliche Funktionen zur Visualisierung der Oberflächen. In Abb. 3.2 a) ist die aus den unveränderten Kontrollpunkten rekonstruierte Oberfläche eines VSBs dargestellt. Der Übersichtlichkeit geschuldet sind die sich an den Ecken der in Blau dargestellten dreieckigen Teilflächen befindlichen Kontrollpunkte ausgeblendet. In Abb. 3.2 b) ist die rekonstruierte Oberfläche des gleichen VSBs mit normalisierten Kontrollpunkten dargestellt.

3.2.2 Voxelization

Für das Training eines mit Tensorflow Keras erstellten Netzwerkes, werden Input und Output Daten in der Gestalt eines Tensors verlangt. Input und Output Daten in Form von Numerical Python (NumPy)-Arrays erfüllen diese Anforderung. Die Überführung der VSBOs in ein Tensor ähnliches Objekt wird durch die Voxilization ermöglicht. Binvox von Patrick Min

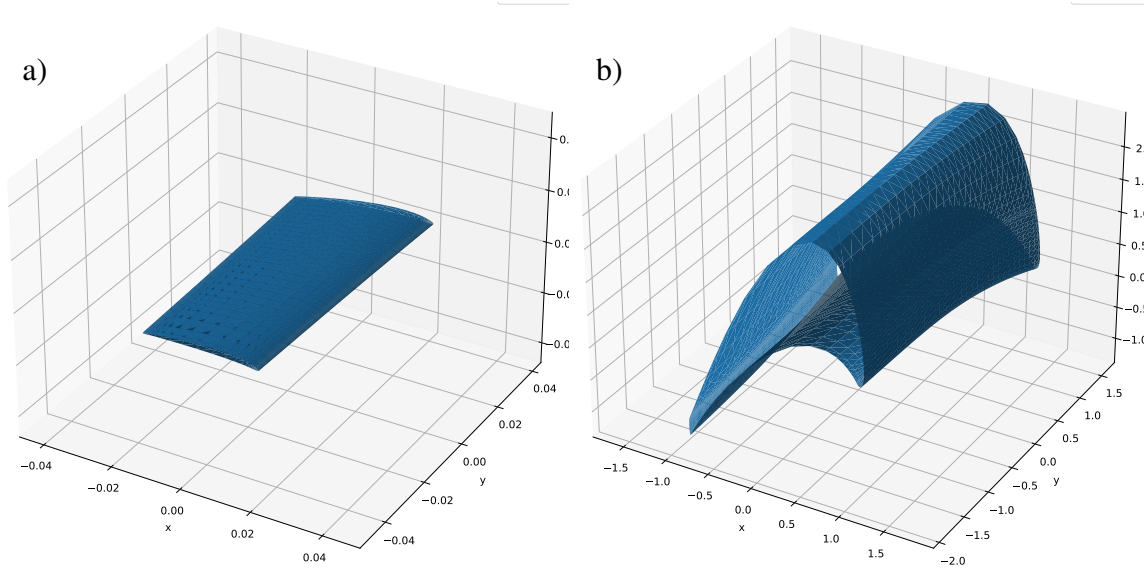


Abbildung 3.2: Rekonstruierte VSBO. Abb. a) Original. Abb. b) Normalisiert

[20] ist ein mit C++ programmiertes hocheffizientes Executable (.exe), welches die mit `geomdl` erstellten und als STL Dateien exportierten 3D VSB-Modelle einliest, diese mit den in Abschnitt 2.3 vorgestellten Parity Count und Ray Stabbing Algorithmen in ein binäres 3D Voxel Gitter umwandelt und in einer Voxel Datei abspeichert.

Die Auflösung des Voxelgitters kann zwischen 128 und 512 Voxel pro Dimension beliebig gewählt werden. Nach der Voxelization eines polygonalen Oberflächenmodells schreibt Binvox Prozessinformationen wie die erkannte BBX und die Anzahl der als Modell intern erkannten Voxel in den standard character output device (std::cout). Binvox setzt vor der Voxelization die maximale Länge der erkannten BBX mit der gewählten Auflösung in eine Raumrichtung gleich. Diese maximale Länge jedes VSB ist in diesem Datensatz, dessen Spannweite. Als Output-Format für die Voxelgitter wird Visualization Toolkit (VTK) als Datei-Format gewählt. VTK ist eine Open-Source 3D Computer Grafik, Modellierung, Bildverarbeitung und Volumen-Rendering Software [26], deren Funktionen in dieser Arbeit durch die dazugehörige Python Library verwendet werden. Die darin enthalten Funktionen ermöglichen das Einlesen und Umwandeln der binären 3D Voxelgitter in die benötigten NumPy-Arrays. In Abb. ist das dreidimensionale binäre Voxelgitter der in Abb. 3.2 dargestellten aus GTlab exportierten unveränderten VSBO dargestellt. Dieses ist mit $128 \times 128 \times 128 = 2097152$ Voxeln aufgelöst. Die von Binvox als innerhalb der VSBO erkannten Voxel sind in Blau dargestellt. Das Profil ist im Zentrum des Voxelgitters positioniert und definiert mit seiner Spannweite die Seitenlänge des Voxelgitters. Die Ausdehnung der Spannweite entspricht in Abb. 3.3 der Ausdehnung in y-Richtung. Somit ist der Diskretisierungsfehler in y-Richtung am geringsten. Wie in Abschnitt 2.4 sind parallele dünnwandige Oberflächenabschnitte am schwierigsten als Teil eines Models zu erkennen. Durch die großen Längenunterschiede zwischen Profildicke und Spannweite wird ersteres

immer schlechter aufgelöst. Wenn also die Spannweite einer Auflösung von 128 Voxeln entspricht, werden Bereiche des VSB die eine geringere Profildicke als eine Voxellänge besitzen, mit großer Wahrscheinlichkeit nicht erkannt. Diese Diskretisierungsfehler sind in Abb. 3.3 a) an der Vorderkante und in Abb. 3.3 b) an der Hinterkante zu sehen. Insbesondere bei sehr spitz zulaufenden Hinterkanten werden auf diese Weise ganze Bereiche nicht erkannt, durch welche keine Gitterpunkte verlaufen.

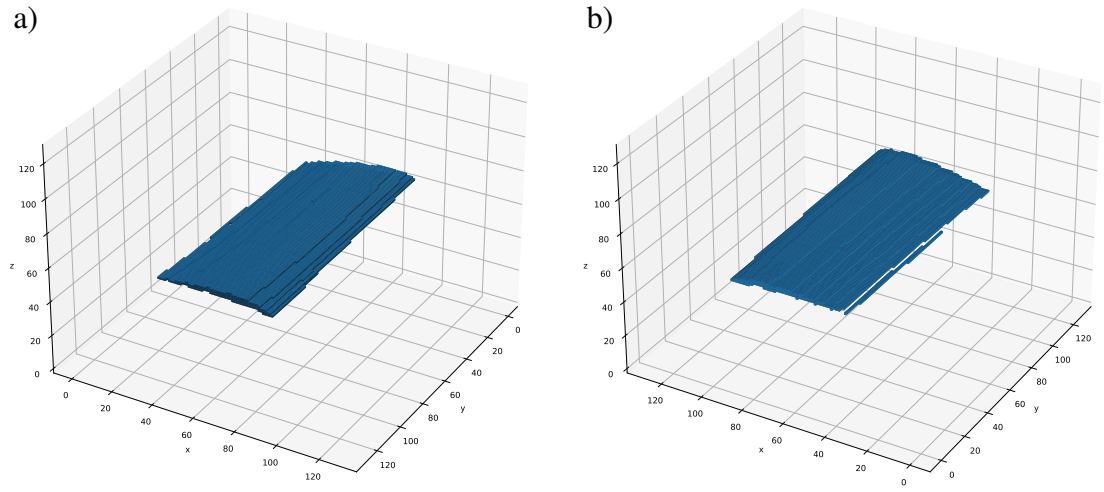


Abbildung 3.3: 3D Voxelgitter der original VSBO. a) zeigt die Auflösung der Vorderkante (*Leading Edge*) des Profils. b) zeigt die Auflösung der Hinterkante (*Trailing Edge*) des Profils

Der durch die Voxelization auftretende Fehler in der Diskretisierung mit Volumenelemente wird für das VSBV mit der Relative Percent Difference (RPD)-Methode

$$\mathrm{RPD}_{\mathrm{vox}} = \frac{\frac{N_{\mathrm{vox,VSBO}}}{N_{\mathrm{vox,bbx}}} - \frac{V_{\mathrm{VSB}}}{V_{\mathrm{bbx}}}}{\frac{V_{\mathrm{VSB}}}{V_{\mathrm{bbx}}}} \tag{3.1}$$

der relative Fehler bezogen auf die BBX berechnet. $N_{\mathrm{vox,VSBO}}$ ist die Anzahl der als innerhalb der VSBO erkannten Voxel und $N_{\mathrm{vox,bbx}}$ ist die Anzahl der Voxel innerhalb der BBX. V_{VSB} ist das von GTlab exportierte Volumen eins VSB und V_{bbx} ist das Volumen der BBX, welche dieses begrenzt. Über die Berechnung des Mittelwerts

$$\mu_{\mathrm{rpd,vox}} = \frac{1}{N}\sum_{i=1}^{N}\mathrm{RPD}_{\mathrm{vox},i} \tag{3.2}$$

und der Standardabweichung

$$\sigma_{\mathrm{rpd,vox}} = \frac{1}{N}\sum_{i=1}^{N}(\mathrm{RPD}_{\mathrm{vox},i} - \mu_{\mathrm{rpd,vox}}) \tag{3.3}$$

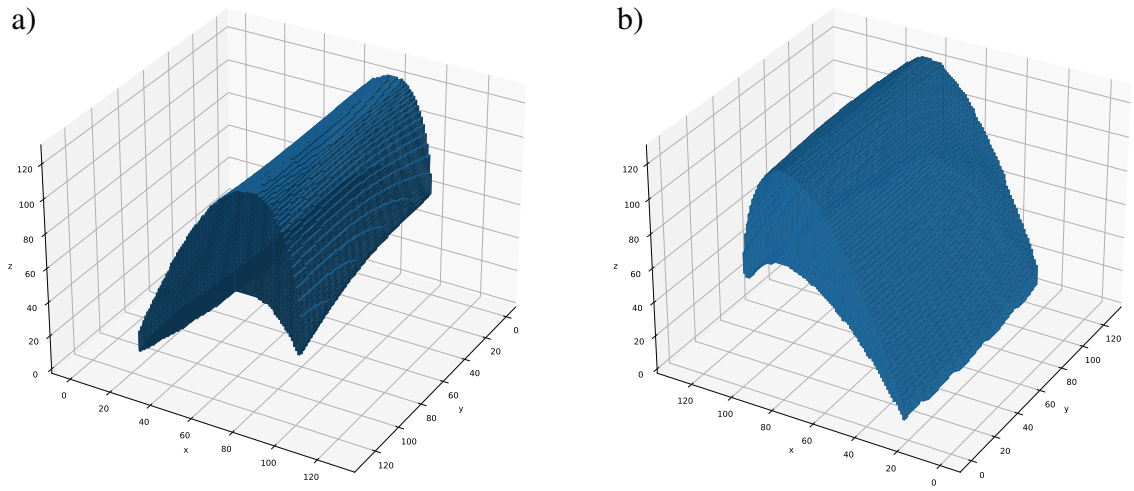


Abbildung 3.4: 3D Voxelgitter der normalisierten VSBO. Abb. a) zeigt die Auflösung der Vorderkante (*Leading Edge*) des Profils. Abb. b) zeigt die Auflösung der Hinterkante (*Trailing Edge*) des Profils

von RPD über jedes VSB ermöglicht den Fehler in der Diskretisierung zwischen verschieden Auflösungen zu vergleichen. In Abb. 3.5 sind die Standardabweichungen und Mittelwerte als Fehlerbalken über die Auflösungen im Bereich 128 bis 288 Voxel aufgetragen. Die relativen Fehler der unveränderten VSBV sind in Blau dargestellt. Die relativen Fehler mit vorangegangener Normalisierung der Geometrie in Orange. Die Einhüllende der Standardabweichung ist eine lineare Interpolation zwischen den Auflösungen mit einem Abstand von 32 Voxeln. Abb. 3.5 zeigt, dass die Voxelization in beiden Fällen über alle Auflösungen im Mittelwert eine Überschätzung des tatsächlichen Volumens V_{VSB} zur Folge hat. Für beide Fälle gilt ebenfalls, dass alle Voxelizierten VSBs deren Fehler innerhalb der Standardabweichung liegen ebenfalls das tatsächliche Volumen überschätzen. Die blau gekennzeichneten Standardabweichung von RPD der unveränderten VSBV nimmt wie erwartet mit zunehmender Auflösung kontinuierlich ab. Im Vergleich bleibt die orange gekennzeichneten Standardabweichung von RPD der normalisierten VSBV über die zunehmende Auflösung fast konstant. Liegt der relative Fehlerbereich für die unveränderten VSBV im Fall von 128 Voxel mit $\Delta\text{RPD}_{\text{vox},0,128}[\sigma_{\text{rpd,vox}} - \mu_{\text{rpd,vox}}, \sigma_{\text{rpd,vox}} + \mu_{\text{rpd,vox}}] \approx [0.003, 0.0135]$ noch deutlich über dem relativen Fehlerbereich der normalisierten VSBV $\Delta\text{RPD}_{\text{vox},1,128} \approx [0.0048, 0.0105]$, konvergiert dieser über die steigende Auflösung gegen $\text{RPD}_{\text{vox},1,288}$ und ist für 288 Voxel mit einem Fehlerbereich von $\text{RPD}_{\text{vox},1,288} \approx [0.00305, 0.0101]$ nur geringfügig kleiner als $\Delta\text{RPD}_{\text{vox},1,128}$. Wird die benötigte Speicherkapazität für eine `numpy.ndarray` mit dem kleinsten Datentype `uint8` betrachtet, resultiert daraus für ein Voxelgitter der Dimension $N = 128$ mit N^3 Elementen ein Speicherbedarf von $N^3 \times 8\text{Byte} = 1024\text{Byte}$. Für eine Voxel Gitter der Dimension $N = 288$ wird dann schon ein Speicherbedarf von 2304Byte benötigt. Daraus geht hervor, dass der Speicherbedarf pro Voxel exponentiell anwächst und

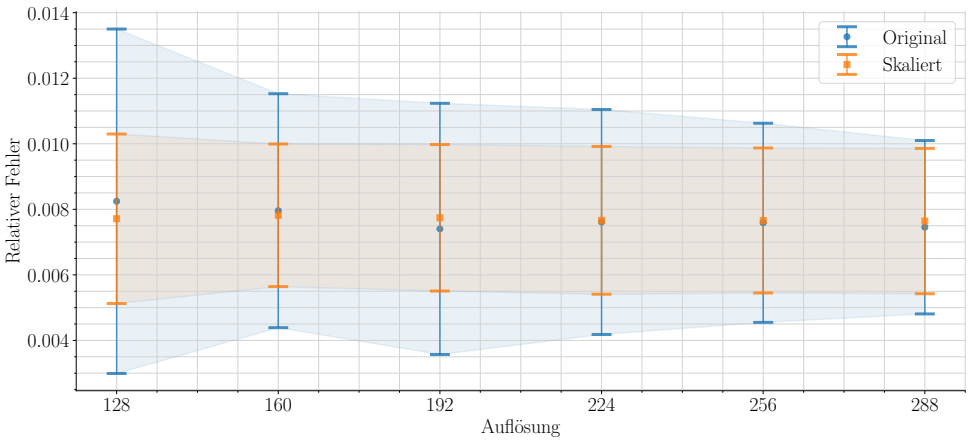


Abbildung 3.5: Relativer Fehler der Voxelization bezogen auf die Auflösung

daher eine geringe Auflösung speziell in Bezug auf den Arbeitsspeicher von Bedeutung ist. Die Größe des Arbeitsspeichers definiert die maximale Größe der Batch-Größe. Wie in Abschnitt 2.5.4 erklärt, ist die Batchsize die Anzahl an Sample für die gleichzeitig eine Aktualisierung der Gewichte mit einem Feed-Forward und anschließendem Feed-Back durchgeführt wird. Die Größe an zur Verfügung stehendem Arbeitspeicher Limmitiert deshalb die maximal mögliche Batch-Größe. Eine geringere Auflösung führt daher zu einer geringeren Input-Größe und mehr Sample in einem Batch mit dem das DNN gleichzeitig trainiert werden kann. Für eine geringere Input-Größe wird auch eine geringere Anzahl an Convolutional- und Pooling-Operationen benötigt, was die Rechengeschwindigkeit erhöht. Eine so gering wie mögliche Auflösung mit einem gleichzeitig geringen relativen Fehler bietet daher eine Auflösung mit 128 Voxeln und normalisierter VSB-Geometrie. Der durch die Normalisierung verlorene geometrische Zusammenhang zu den Eigenfrequenzen wird über das tatsächliche Volumen V_{VSB} als zusätzliches Feature in das DNN eingespeist.

3.2.3 Normalisierung

Die Normalisierung der Input-Daten und der Ground-Truths ist eine übliche Methode im Bereich ML zur Transformation in einen gewünschten Wertebereich. Nach Abschnitt 2.5.2 wird im Idealfall eine Gleichverteilung der Sample über die einzelnen Wertebereiche der von den Input-Daten und Ground-Truths aufgespannten Dimensionen angestrebt. Ungleichmäßige Verteilungen können mittels einer Normalisierung optimiert werden. Die Transformation oder Skalierung auf einen Wertebereich von null bis eins führt auch zu kleineren Gewichten und geringeren Loss-Werten, die eine Optimierung des DNN verbessern. Die vorangegangene Normalisierung der geometrischen Zusammenhänge jedes VSBs, motiviert durch die Reduzierung des Diskretisierungsfehlers, resultiert in einer Normalverteilung des ersten

Features, welche die Generalisierung des DNNs ebenfalls begünstigt. Die in diesem Abschnitt folgende Normalisierung der Verteilungen von Volumen und Eigenfrequenzen ist davon motiviert, die bestmöglichen Vorhersage des DNN zu garantieren.

Normalisierung der Eigenfrequenzen

In Abb. 3.6 ist die Verteilung der zehn Eigenfrequenzen aufgeteilt in Rotoren und Statoren des Triebwerksverdichters aufgetragen. Der ursprüngliche Wertebereich von $f \in (700, 71200)$ Hz kann beispielsweise mittels der MinMax-Skalierung

$$\text{MinMax}(x) = \frac{(x - x_{\text{min}})}{(x_{\text{max}} - x_{\text{min}})} \tag{3.4}$$

auf den Wertebereich $f \in (0, 1)$ skaliert werden. Darin ist zu erkennen, dass die Eigenfrequenzen der Statoren annähernd den gesamten Wertebereich vollständig abdecken. Die Statoren besitzen entgegen der Rotoren Eigenfrequenzen die größer $f = 30000$Hz sind und durch wenige Ausreißer den Wertebereich bis $f = 71200$Hz ausdehnen. Der Wertebereich von $f \in (0, 1200]$ Hz wird von den Rotoren dominiert und erstrecken sich bis circa $f = 30000$Hz. Der doppelt so große Wertebereich der Statoren ist auf deren beidseitige Verankerung zurückzuführen, die im Vergleich zu den dazugehörigen einseitig im Verdichter eingespannten Rotoren für die annähernd doppelt so großen Eigenfrequenzen verantwortlich ist. Die große Mehrheit der Sample decken dabei alleine die untere Hälfte des gesamten Wertebereichs ab. Eine solche Verteilung ist nicht wünschenswert, denn eine ungleichmäßige exponentiell abfallende Verteilung führt im Training des DNN auf ebenfalls ungleich verteilte Gewichte, die Samples mit niedrigen Eigenfrequenzen eine höhere Bedeutung zuschreiben. Dies führt auf eine schlechtere Generalisierung des Modells und in Konsequenz auf schlechtere Vorhersagen, speziell für Sample, die Eigenfrequenzen größer als 40000 Hz besitzen. Werden die auf den Wertebereich $f \in (0, 1)$ skalierten Verteilungen jeder einzelnen Eigenfrequenz in Abb. 3.7 betrachtet, so fällt speziell für die Rotoren eine noch ungleichmäßigere Verteilung auf. Die einem Sample zugehörigen Eigenfrequenzen f_1 bis f_{10} sind nach der in Abschnitt 2.1 formulierten Konvention entsprechend aufsteigend sortiert. Im Fall der Verteilung der ersten Eigenfrequenz deckt diese in Abb. 3.7 f_1) nicht einmal ein Zehntel des gesamten Wertebereichs ab. Dieser Anteil wächst über die zehn Eigenfrequenzen bis knapp über 40 Prozent an. Die Abdeckung der Wertebereiche der Statoren steigt beginnend bei der ersten Eigenfrequenz f_1 bei knapp unter 25 Prozent bis 100 Prozent in der zehnten Eigenfrequenz f_{10}. Jedoch sind die Sample der Statoren ab der skalierten Eigenfrequenz von 0.8 nahezu unmerklich vertreten. Im Gegensatz zu den Verteilungen der Eigenfrequenzen der Rotoren verschieben sich die Verteilungen der Statoren ausgehend von f_1 in Richtung des Maximalwerts und eine Lücke in der Abdeckung der niedrigen Eigenfrequenzen entsteht, welche bis zu einem Anteil von 20 Prozent in der zehnten Eigenfrequenz anwächst.

Für die Korrektur dieser ungleichmäßigen Verteilungen mit hohen Dichte-Anteilen in den niedrigen Wertbereichen wird die Normalisierung mit dem natürlichen Logarithmus

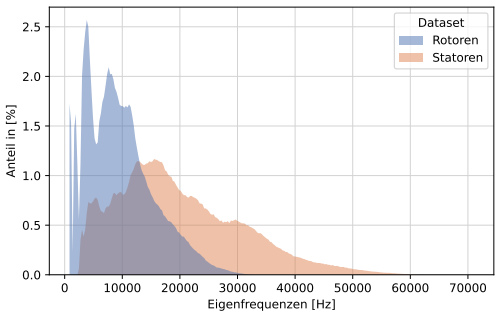


Abbildung 3.6: Verteilung der Eigenfrequenzen, dargestellt für Rotoren und Statoren des gesamten Datensatzes

Log(x) verwendet. Bei dieser Normalisierung ist zu beachten, dass der Wertebereich der Eingangsgröße x die Bedingung $1 \leq x \leq \infty$ erfüllen muss, damit der positive Wertebereich erhalten bleibt. Nach der Log-Normalisierung wird mit Gl. 3.4 der Wertebereich wieder auf $f \in (0, 1)$ skaliert. Das Ergebnis der Normalisierung ist für die zehn Eigenfrequenzen in Abb. 3.8 zu sehen. Die dargestellten Verteilungen für Statoren und Rotoren ähneln jetzt einer Normalverteilung. In beiden Fällen wird die Abdeckung über den Wertebereich von $(0, 1)$ in den einzelnen Eigenfrequenzen auf mindestens 30 Prozent erhöht. Speziell die Wertebereiche der niedrigen Eigenfrequenzen sind stark verbessert. Zusammenfassend lässt sich über die Log-Normalization der Eigenfrequenzen festhalten:

- Reduziert die hohe Varianz in den Eigenfrequenzen
- Reduziert Ausreißer im oberen Frequenzbereich
- Verändert nicht den relativen Zusammenhang zwischen den zehn Eigenfrequenzen jedes VSBs.
- Konvention der Reihenfolge der Eigenfrequenzen bleibt erhalten.
- Deutliche Verbesserung in den Verteilungen der ersten Eigenfrequenzen. Diese sind von besondere Bedeutung, denn diese werden in der Anwendung als erstes angeregt und verlangen daher eine höhere Genauigkeit der Vorhersage des DNNs.

Normalisierung der Volumen

Mittels der Normalisierung der Volumen wird entsprechend der Eigenfrequenzen eine verbesserte Generalisierung und höhere Genauigkeit des DNN angestrebt. Die ursprüngliche Verteilung und der Wertebereich der Volumen in [mm^3] ist in Abb. 3.9 dargestellt. Darin sind ebenfalls die Verteilungen der Rotoren (blau) und Statoren (rot) separiert aufgetragen. Rotoren und Statoren besitzen auch in diesem Fall den größten Anteil in der unteren Hälfte des Wertebereichs $V_{\mathrm{VSB}} \in (0, 10000)\mathrm{mm}^3$, dessen obere Grenze durch wenige Ausreißer

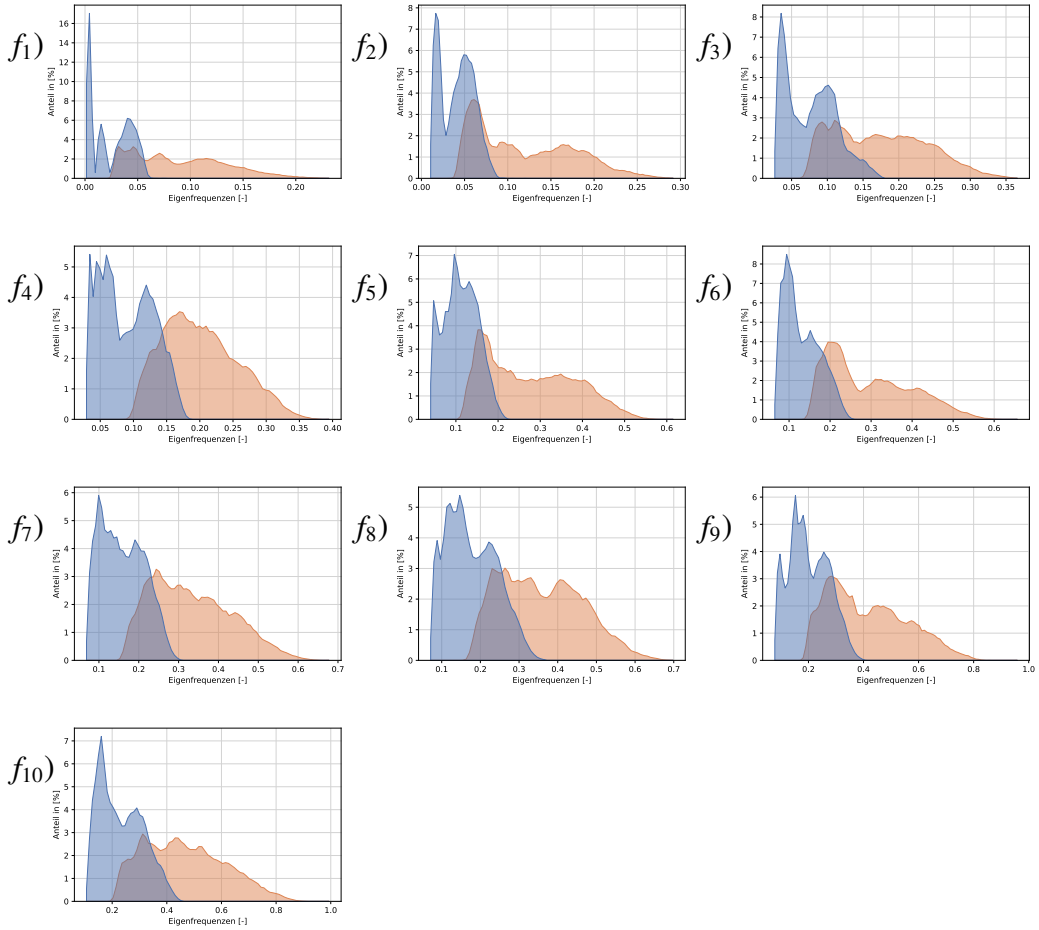


Abbildung 3.7: Verteilungen der zehn Eigenfrequenzen f_1 bis f_{10}, dargestellt für die Rotoren (orange) und Statoren (blau) des gesamten Datensatzes

definiert wird. Die Ausreißer und die damit verbundene hohe Varianz in der Verteilung wird ebenfalls durch die Normalisierung mit dem natürlichen Logarithmus $\mathrm{Log}(x)$ reduziert. Die anschließende MinMax-Skalierung mit Gl. 3.4 bringt den Wertebereich $V \in (0, 1)$ in Einklang mit den Wertbereichen von der Voxelgitter und Ground-Truth. Die daraus resultierend normalisierte Verteilung der Volumen, welche in dieser Form das zweite Feature des DNN bilden, ist in Abb. 3.10 abgebildet. Die endgültigen Verteilungen von Statoren und Rotoren, ähneln nun einer Normalverteilung und decken zusammen den gesamten Wertbereich von $V \in (0, 1)$ ab.

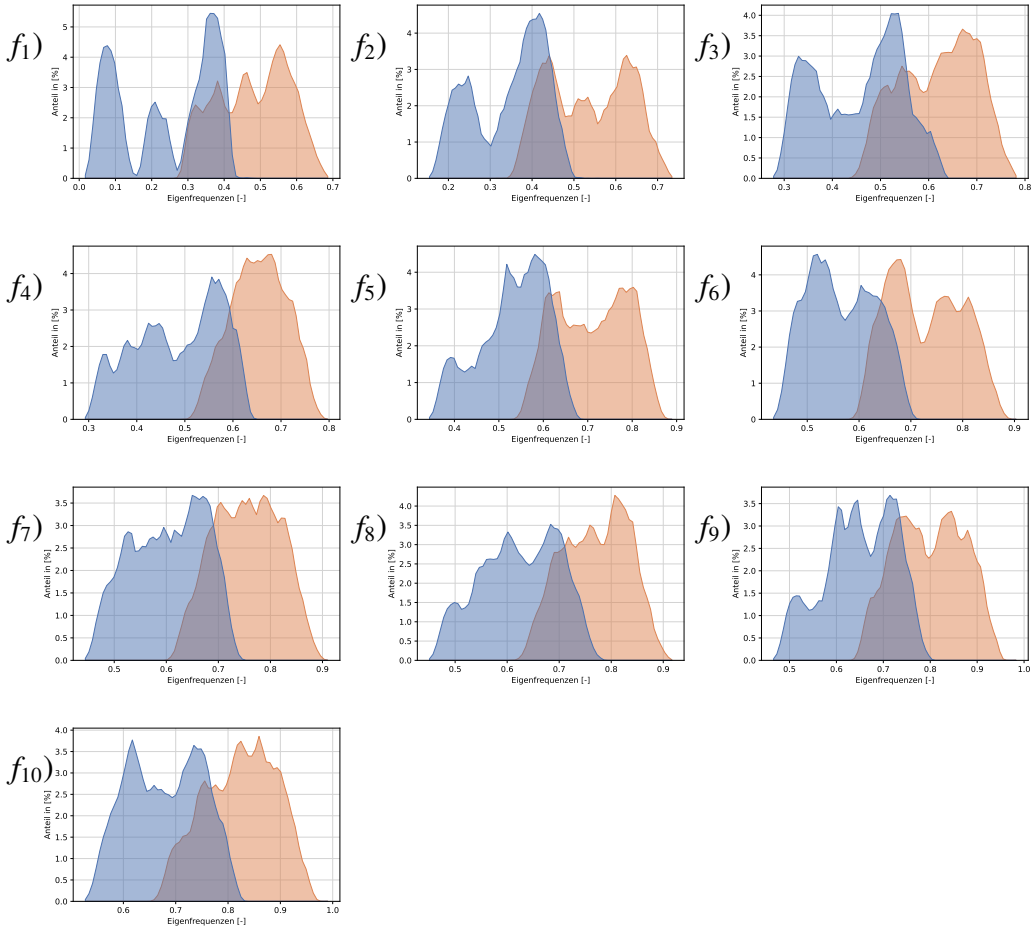


Abbildung 3.8: Verteilungen der zehn Eigenfrequenzen f_1 bis f_{10} nach Log-Normalization, dargestellt für die Rotoren (orange) und Statoren (blau) des gesamten Datensatzes

3.2.4 Binary Encoding

Um eine ungleiche Priorisierung des DNN der binären dreidimensionalen Voxelgitter und den skalaren Volumen präventiv zu vermeiden, werden die Volumen in eine binäre Vektordarstellung überführt. Diese Umwandlung entspricht einer Art Binary-Encoding von Fließkommazahlen. Mit der im Anhang 3 beigefügten Python Funktion `binary_encoding` wird eine Umwandlung vom Dezimal- ins Binärsystem ermöglicht. Für eine einheitliche Codierung sind die an das Argument `dec_num` übergebenen Werte auf positive natürliche Zahlen beschränkt. Die maximale Anzahl an Binärstellen und damit die Größe des Volumenvektors wird über das Argument `bin_length` übergeben. Im Fall des normalisierten Wertebereichs $V \in (0, 1)$ der Volumen, muss die benötigte Anzahl an Binärstellen mit den folgenden Schritten berechnet werden.

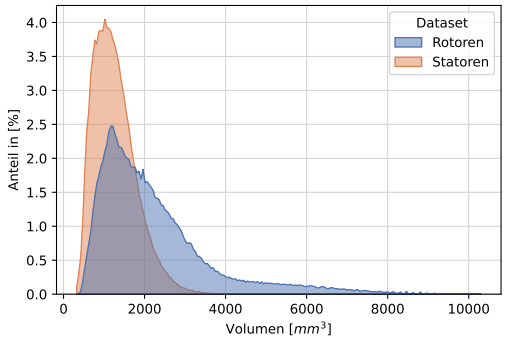


Abbildung 3.9: Verteilung Volumen im gesamten Datensatz, dargestellt für die Rotoren (orange) und Statoren (blau).

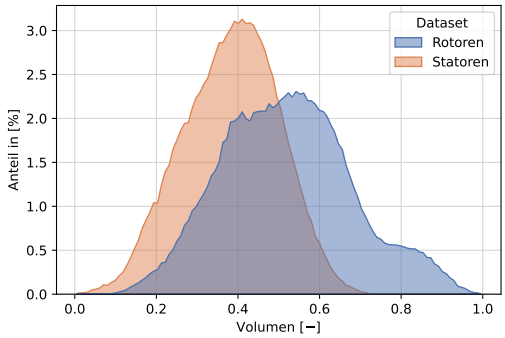


Abbildung 3.10: Logarithmische Verteilung Volumen des gesamten Datensatzes, dargestellt für die Rotoren (orange) und Statoren (blau).

1) Auslesen der geringsten Fließkommazahl V_{min} der Volumen.

2) Bestimmen der Anzahl an Nachkommastellen von V_{min} =`v_min` und hinterlegen der für das Binary-Encoding benötigten Präzision p =`p`. Die Präzison wird in Python mit

```
    decimal_places = int(f'{v_min:e}'.split('e')[-1])
    p = 10**(decimal_places*(-1))
```

berechnet.

3) Die maximal benötigte Anzahl an Binärstellen `max_bin_length` entspricht der Binärdarstellung der größten Fließkommazahl V_{max} =`v_max` in den Volumen. Dies wird in Python mit

```
    num = round(v_max)
    res = bin(num).lstrip("0b").replace('',',').split(',')
    binary_num = res[1:-1]
    max_bin_length = len(binary_num)
```

berechnet.

Mit `max_bin_length` wird dann über die Funktion `binary_encoding` jedes der im Datensatz enthaltenen Volumen in seine entsprechende binäre Vektordarstellung überführt. `max_bin_length` legt die minimale Größe des zweiten DNN-Inputs `in2_size` fest. Der in Abb. 3.9 dargestellte Wertebereich der Volumen führt nach der Normalisierung und der Berechnung der benötigten Präzision, auf `max_bin_length`= 24. Folgend sind die Vor- und Nachteile des Binary-Encodings aufgelistet.

Vorteile:

- Binäre Voxelgitter und Volumen werden beide als mehrdimensionale Numpy Arrays an das DNN übergeben.
- Die Binärdarstellung führt auf eine gleiche Behandlung binäre Voxelgitter und der Volumen, durch das DNN.
- Das DNN lernt eine mehrdimensionale Darstellung der Volumen.

Nachteile:

- Feste Input Größe `in2_size`= 24.
- Andere Triebwerkskonfigurationen erfordern eventuell eine Erweiterung der Input Größe.

3.2.5 Unterteilung des Datensatzes

Wie Abschnitt 3.1 erläutert, besteht der gesamte Datensatz aus 78560 Samples. Nach dem Preprocessing besitzt jedes Sample ein Voxelgitter mit 128^3 Elementen, ein Volumenvektor mit 24 Elementen und ein Ground-Truth mit 10 Elementen. Alle sind als `numpy.ndarray` gespeichert. Die binären Input-Daten werden mit dem kleinsten Datentype `uint8` gespeichert. Die Output-Daten erfordern `float32`. Ein Sample benötigt daher einen Speicherbedarf von $(128^3 + 24 + 10 \cdot 4\text{Byte}) = 2.0972\text{MByte}$. Dies würde für ein Training des DNN mit dem gesamten Datensatz ein Speicherbedarf von $78560 \times 2.0972\text{MByte} = 164.75$ GByte an Arbeitsspeicher voraussetzten. Aufgrund des großen Speicherbedarfs wird bereits während des Preprocessing eine effizientere stückweise Verarbeitung von Pakete bestehend aus 1280 Sample verwendet. Nach jeder Oberflächen-Rekonstruktion, Voxelization, Normalisierung und Binary-Encoding der in einem Paket enthalten Sample, werden diese mittels der Funktion `train_test_split` der `sklearn` Python Library durch setzten des Arguments `shuffle=True` gemischt und in 42 Prozent Trainings und 58 Prozent Test-Daten aufgeteilt. Der prozentuale Anteil an Validierung-Daten entspricht 15% der Trainings-Daten, welcher während des Trainings intern durch das DNN, von dem Trainingsanteil getrennt wird. Die Unterteilung des Datensatzes ist in Abb. 3.11 dargestellt. Die ungewöhnliche Aufteilung von 42% Trainings-Daten zu 58% Test-Daten, ist mit dem großen Speicherbedarf der Voxelgitter und den hardwareseitigen Limitierungen des in dieser Arbeit zur Verfügung

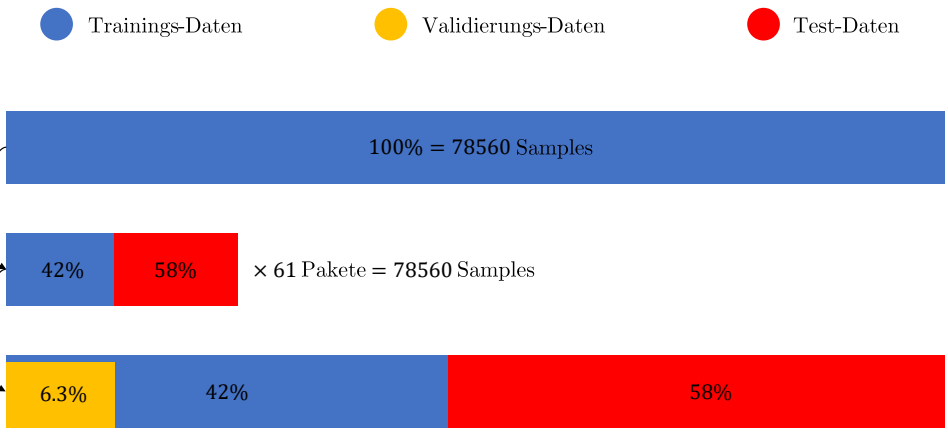


Abbildung 3.11: Angewandte Unterteilung des gesamten Datensatzes mit 100%. Während des Preprocessings werden insgesamt 61 Pakete sequentiell verarbeitet, gemischt, in 42% Trainings-Daten und 58% Test-Daten unterteilt und vor dem Training wieder zusammengeführt. Während des Trainings des ML-Modells werden 15% der Trainings-Daten für die Validierung verwendet. Was auf einem Anteil von 6.3% an Validierungs-Daten am gesamten Datensatz führt.

stehenden Arbeitsspeicherkapazität von 128GByte Random Access Memory (RAM). Trotz augenscheinlich ausreichendem RAM, werden bereits beim Laden der von 33280 Sample und somit 42% des gesamten Datensatzes die bereitgestellten 128GByte von Python allokiert. Diese Beschränkungen führen auf die endgültige in Abb. 3.11 dargestellte Aufteilung des Datensatzes in 35, 7% Trainings-, 6.3% Validierungs- und 58% Test-Daten. Für die Berechnung des Forward-Pass und der Back-Propagation wird eine Graphics Processing Unit (GPU) aufgrund ihrer deutlich besseren Rechenleistung verwendet. Der grafische Speicher der GPU von 8GByte erfordert das in Abschnitt 2.5.4 beschriebene Mini-Batch Gradient-Descent Verfahren. Die für diese Anwendung und Hardware von Tensorflow ohne Out of Memory (OOM)-Fehler zugelassene maximale Batch-Größe beträgt dabei $2^5 = 32$ Sample.

3.2.6 Modellstruktur

Die klassische Modellstruktur mit zusätzlicher Batch-Normalisierung des in dieser Arbeit verwendeten DNN ist in Abb. 3.12 dargestellt. Diese Modellstruktur entspricht ebenfalls

jener des DNN aus Trials 236 der folgenden Hypeparameter-Optimierung. Die mit Tensorflow Keras erstellte variable Modellstruktur ist als Python-Funktion `dnn_model` im Anhang 2 hinterlegt. Das DNN besteht aus zwei Eingängen `InputLayer1` und `InputLayer1`, sowie einem `OutputLayer`. Die Größen der Input Layer können über die Funktionsargumente `in1_size` und `in2_size` auf die gewählte Auflösung der Voxel und die Anzahl an Binärstellen des Volumenvektors angepasst werden. Die Größe des Output-Layers ist auf zehn Ausgangswerte festgelegt, welche mittels der RelU-Aktivierungsfunktion auf beliebige reelle Fließkommazahlen größer null abgebildet werden. Die Aktivierungsfunktion, welche für die Hidden-Layer verwendet werden soll, kann über das Argument `af` dem Netzwerk zugewiesen werden. Für die dreidimensionalen Convolutional-Layer `tf.keras.layers.Conv3D` kann über das Argument `ks` die Größe des dreidimensionalen Kernels festgelegt werden. Mit dem festgelegten Argument `padding='SAME'` entsprechen die dreidimensionalen Convolutional-Layer, den in Abschnitt 2.5.7 vorgestellten Volume-Convolutions mit Schrittweite $s = 1$ und Padding $p = \frac{f-1}{2}$. Auf diese Weise entsprechen die Dimensionen der Layer Outputs, der Dimension des Layer-Inputs. Mit dem Argument `n_filters` wird eine Liste mit der gewünschten Anzahl an Feature-Maps pro Convolutional-Layer übergeben. Auf jedes Convolutonal-Layer folgt ein dreidimensionales Pooling-Layer. Die Faktoren, mit welchen die Pooling-Layer die Dimensionen der Layer-Inputs reduzieren, wird analog über eine Liste an das Argument `n_pool_size` übergeben. Mit setzten des dazugehörigen Arguments `po='max'` wird Max-Pooling verwendet, ansonsten gilt Average-Pooling. Zuletzt kann durch setzten des Arguments `bn=True` die Layer-Outputs jedes Dense- und Max-Pooling-Layers gleich 5) auf einen Mittelwert von null und Standardabweichung von eins mit Batch-Normalisierung normalisiert werden. Mit `merge = tf.keras.layers.Concatenate()([x,y])` werden die ausgehend von `in1_size` in den 3D Voxelgittern identifizierten Feature mit dem Mapping ausgehend von `in2_size` des binären Volumenvektors miteinander verknüpft. Die darauf folgenden Dense-Layer, erlernen dann das bestmögliche Mapping auf die zehn Outputs im Output-Layer. Die Batch-Normalisierungen `batch_normalization_5` und `batch_normalization_5` vor der Verknüpfung `merge = tf.keras.layers.Concatenate()([x,y])` der jeweiligen 64 Outputs der einzelnen Zweige des DNN sorgt für die gleichwertige Gewichtung der 3D Voxelgitter und der binären Volumenvektoren.

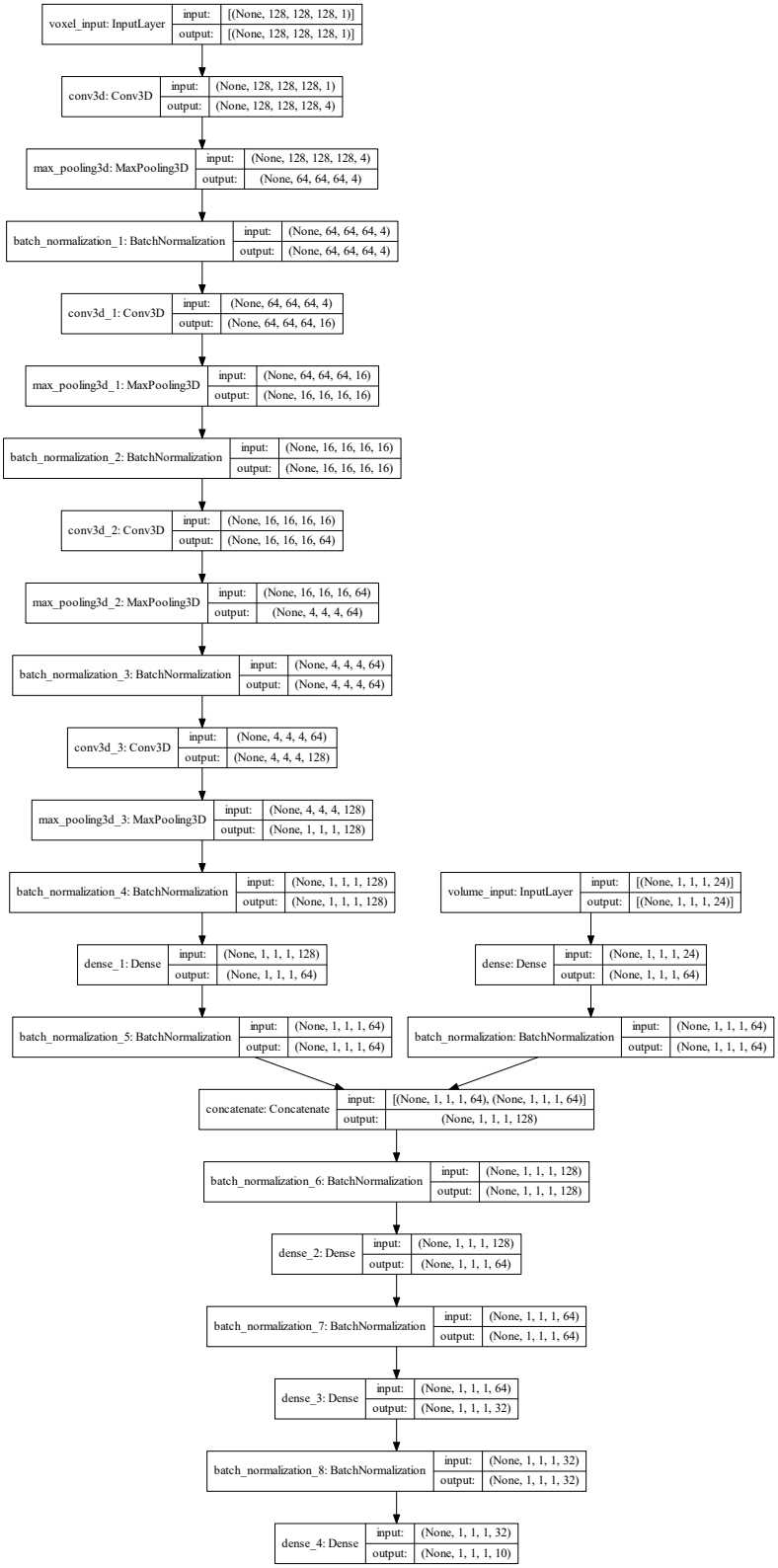


Abbildung 3.12: Übersicht der Modellarchitektur von DNN Trial 236

4 Ergebnisse

In diesem Kapitel wird die in Abschnitt 3.2.6 beschriebene variable DNN-Struktur in unterschiedlichen Konfigurationen, auf Basis des mittels Preprocessing 3.2 optimierten Datensatz trainiert, validiert und getestet. Zuerst werden die idealen Hyperparameter mit dem Open-Source Framework Optuna [1] identifiziert. Dieses nutzt den Bayes'schen Optimierungsalgorithmus TPE für die Hyperparameter Optimierung. Darauf aufbauend werden die einzelnen Parameter der besten Konfigurationen analysiert und die optimale Modell-Konfiguration in Abschnitt 4.3.1 mit der maximal möglichen Anzahl an Samplen mittels Fine-Tuning weiter optimiert und anhand der absoluten und relativen Fehler der Vorhersagen ausgewertet. Anschließend folgt in den Abschnitten 4.3.2 und 4.3.3 eine Analyse der Input und Output Daten, für die das DNN ausgesprochen gute, beziehungsweise schlecht Vorhersagen liefert. In den abschließenden Abschnitten 4.3.4 und 4.3.5 werden die Feature Maps der einzelnen Convolutional Layer ausgewertet und die darin hinterlegte Fähigkeit des DNN zwischen Statoren und Rotoren zu unterscheiden mittels Transfer-Learning nachgewiesen.

4.1 Hyperparameter-Optimierung

Mit Optuna und dem Bayes'schen Optimierungsalgorithmus TPE wird in einer Studie die beste Parameterkombination für das in Abschnitt 3.2.6 eingeführte DNN-Modell ermittelt. Dafür wird zuerst ein Optuna-Study-Objekt mit `study = optuna.create_study()` initialisiert. Über eine eigens geschrieben `Objective()` Klasse werden dann die zu optimierenden Parameter und deren Wertebereichen entsprechend Anhang 4 an das `trial` Objekt der Studie übergeben. Zur Auswahl für das Argument `po` stehen Max Pooling und Average Pooling. Des Weiteren werden die Aktivierungsfunktionen tanh, RelU, SelU und linear zur Auswahl für den Parameter `af` festgelegt. Eine lineare Aktivierungsfunktion bedeutet, dass das Ergebnis der MAC $y = a(z) = z$ nicht verändert wird. Die Kernel Größen $3, 5$ und 7 stehen für den Parameter `ks` zur Auswahl. Wie in Abschnitt 3.2.6, erläutert kann über den Parameter `bn` eine Batch Normalisierung nach jedem Hidden-Layer durchgeführt werden. In der `Objective()` Klasse befindet sich ebenfalls das zu optimierende DNN Modell, welches in Form der im Anhang 2 befindlichen Funktion als Objekt `model` übergeben wird. Der Hyperparameter der Lernrate `lr` wird über den Gradient Descent Algorithmus Adam `optimizer = tf.keras.optimizers.Adam(learning_rate=lr)` an das `model` Objekt übergeben. Der Wertebereich $[10^{-5}, 10^{-1}]$, aus welchem `lr` gewählt werden kann, wird

nach Abschnitt 2.5.8 und Gl. 2.66 durch eine logarithmische Skalierung angepasst. Das tatsächliche Training pro Trial erfolgt bei Ausführung der `model.fit()` Funktion, über deren Argument `batch_size=bs` der TPE-Algorithmus über das setzten der entsprechenden Zweierpotenz zwischen einer Batch Größe `bs` von 2 bis 32 Sample variieren kann. Die Gradienten pro Batch werden über den MSE berechnet und anschließend entsprechend des in Abschnitt 2.5.4 beschrieben Mini-Batch-Gradient-Descent-Verfahrens über deren Mittelwert die Gewichte aktualisiert.

Die gesamte Anzahl an Versuchen (*Trials*), in welcher die Optuna-Study nach Ausführen von `study.optimize(Objective(), n_trials=n_trials)` über verschiedene Kombinationen die beste Konfiguration berechnet, muss ausreichend hoch gewählt werden, um ein aussagekräftiges Ergebnis zu erzeugen. Aus diesem Grund wird eine gesamte Anzahl von `n_trials=` 250 Trials festgelegt, was gleichbedeutend mit dem Training und der Validierung von 250 verschieden DNN-Konfigurationen ist. Ein Training mit der maximalen Anzahl an Samplen pro Trial ist aus diesem Grund nicht effizient und sehr zeitaufwändig. Pro Trial und Epoche wird der R2-Score als Metrik von der `Objective()` Klasse an die Studie übergeben. Der R2-Score dient dann dem TPE Algorithmus als Vergleichswert (*Objective Value*) zwischen den Trials und den einzelnen Epochen. Dieser direkte Vergleich ermöglicht mit einem repräsentativen Bruchteil von 2560 Sample und maximal 100 Epochen pro Trial eine effiziente und aussagekräftige Identifikation der besten Hyperparameter.

4.2 Evaluation der Hyperparameter-Optimierung

In Abb. 4.1 ist der R2-Score der Validierung als Objective-Value über die nacheinander durchgeführten Trials abgebildet. Die jeweilige Hyperparameter Initialisierung der einzelnen Trials wird zufällig von den dazugehörigen Verteilungen gezogen. Beispielsweise wird die Wahl der Aktivierungsfunktion `af = trial.suggest_categorical("af", ["linear", "tanh", "selu", "relu"])` von der dazugehörigen `optuna.distributions.CategoricalDistribution()` bestimmt. Pro Trial werden diese Verteilungen im Sinne einer Bayes'schen Optimierung durch den TPE-Algorithmus angepasst und somit die Ergebnisse, Schritt für Schritt optimiert. Dies führt in den ersten fünfzig Trials noch zu drei Trials mit negativen R2-Scores. Mit voranschreitender Anzahl der durchgeführten Trials, richtet der Optimierungsalgorithmus den Fokus auf die Bereiche der einzelnen Verteilungen aus, an welchen dieser die beste Parameterkombination erwartet. Dies führt über die Anzahl der Trials, zu einer Reduktion der suboptimalen Vorhersagen, welche ab spätestens ab 200 Trials nicht mehr signifikant vom perfekten R2-Score von 1 abweichen. In Abb. 4.3 ist die empirische Verteilung der Trials über den R2-Score dargestellt. Diese Art der Darstellung zeigt den prozentualen Anteil der Trials mit dem gleichen R2-Score innerhalb der Studie und vermittelt einen Eindruck über die Fähigkeit des Bayes'schen Optimierungsalgorithmus TPE, die bestmögliche Kombination der Hyperparameter zu finden. So beträgt der Anteil von Trials mit einem schlechteren R2-Score als 0.5 weniger als 10% und über 80% der

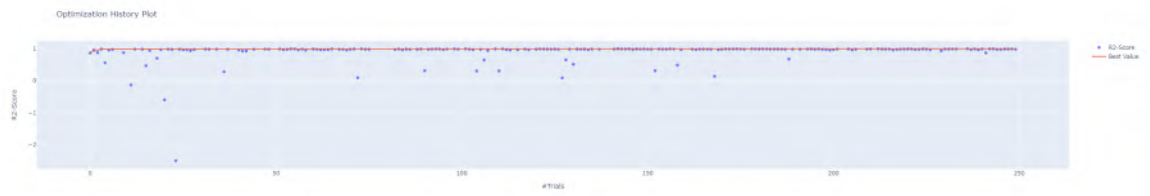


Abbildung 4.1: Alle Resultate der Versuche (Trials) der Hyperparameter Optimierung mit dazugehörigem R2-Score. Die Rote Linie markiert den Besten R2-Score.

Abbildung 4.2: Einfluss der Hyperparameter.

Trials besitzen einen besseren R2-Score als 0.9. Dieser hohe Anteil an Trials mit einem Validerungs-R2-Score über 0.9 bestätigt die von Bergstra [3] beschrieben Überlegenheit von SMBO-Algorithmen gegenüber klassischen Methoden wie Grid Search und Random Search. Optuna ermöglicht zusätzlich in Abb. 4.2 den berechneten Einfluss der Hyperparameter auf

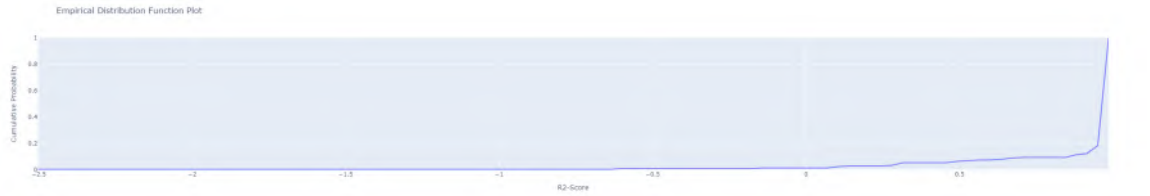


Abbildung 4.3: Empirische Verteilung der Hyperparameter Optimierung

den R2-Score der einzelnen Trials darzustellen. Diese Abbildung zeigt einen sehr großen Einfluss von 79% der Lernrate `lr` auf den R2-Score. Den zweitgrößten Einfluss besitzt hiernach die Kernel Größe mit 16%, gefolgt von der Wahl der Aktivierungsfunktion mit 4% und der Batch-Größe mit 1%. Keinen Einfluss auf den R2-Score nimmt nach Abb. 4.2 die Wahl der Pooling-Methode `po` und der Batch-Normalisierung `bn`. Dies ist jedoch auf die von Optuna zu geringe prozentuale Darstellung zurückzuführen, welche sich auf die gesamte Anzahl von 250 Trials bezieht. In Abb. 4.4 ist klar zu erkennen, dass trotz sehr guter R2-Scores mit Max- oder Avergae-Pooling, beziehungsweise mit oder ohne Batch-Normalisierung, die Bayes'schen Optimierung mit steigender Anzahl von Trials, Max-Pooling und eine Batch-Normalisierung als überlegen betrachtet. Die Priorisierung von Max-Pooling über Average-Pooling ergeben mit Bezug auf die binären Voxelgitter Sinn, denn Average-Pooling führt zu einer Glättung von scharfen Kanten, welche die Detektion der VSB Gestalt erschweren kann. Ebenso besitzt eine Batch-Normalisierung nach Ioffe

[14] große Vorteile, welche zu einer besseren Generalisierung des DNN führen. Eine Initialisierung der Gewichte in einem suboptimalen lokalen Minimum mit einer sehr geringen Lernrate wird trotz der Fähigkeit des Adam-Algorithmus, die Lernrate anzupassen, im Vergleich zu einer Initialisierung der Gewichte nahe des optimalen globalen Minimums, mit derselben niedrigen Lernrate einen deutlich schlechteren R2-Score erzielen. Die unterschiedlichen Auswirkungen mit derselben Lernrate und die adaptive Fähigkeit des Adam Algorithmus auch ausgehend von einer suboptimalen initialen Lernrate gute Ergebnisse zu erzielen beeinflusst somit den prozentualen Einfluss in Abb. 4.2. Trotz dessen zeigt die in Abb. 4.4 dargestellte Verteilung und Häufung der initialen Lernrate um den Idealwert von circa 0.02 welchen Einfluss die Lernrate besitzt. Der Einfluss der Kernel-Größe ist ebenfalls nachvollziehbar, denn eine kleinere Kernel Größe kann feinere Strukturen und Features in den Inputs erkennen, was zu einem geringeren Informationsverlust und somit zu einer höheren Genauigkeit führt. So zeigt Abb. 4.4, dass für jede der drei Kernel Größen `ks` sehr gute R2-Scores nahe dem Wert Eins erzielt werden können, sich mit steigender Anzahl von Trials der TPE Algorithmus die geringste Kernel Größe von $3 \times 3 \times 3$ als ideal betrachtet. Analog verhält es sich bei der Batch-Größe. Abgesehen von der niedrigsten zur Auswahl stehenden Batch-Größe von $2^1 = 2$ Samplen, können für die restlichen Batch-Größen sehr gute Ergebnisse erzielt werden. Mit zunehmender Anzahl an Trials erkennt der Bayes'schen Optimierungsalgorithmus, dass desto mehr Sample sich in einem Batch befinden, eine bessere Generalisierung des DNN-Modells erreicht wird und der Loss sowie der R2-Score weniger stark oszilliert. Diese Oszillationen sind besonders in Abb. 4.6 und dem Trial 27 mit einer Batch-Größe von 2^3 zu sehen. Abgesehen von tanh können mit einer linearen, RelU und SelU Aktivierungsfunktion sehr gute Ergebnisse erzielt werden. Auch hier zeigt die Häufigkeit der Trials bei der SelU Aktivierungsfunktion an, dass diese als Ideal betrachtet wird. Diese Wahl steht im Einklang mit der besonderen Eigenschaft von SelU, welche aufgrund der normalisierten Inputs zu einer automatischen Normalisierung der Gewichte und somit zu einer verbesserten Generalisierung des DNN führt. Der Zusammenhang zwischen

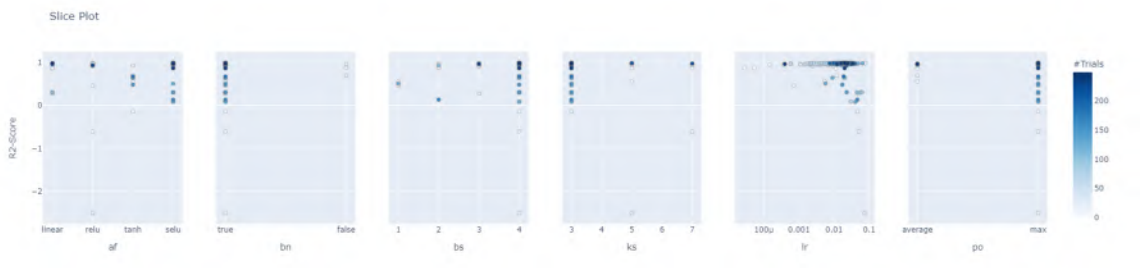


Abbildung 4.4: Beste Versuche der Hyperparameter Optimierung

den einzelnen Trials und den Hyperparametern ist ebenfalls in Abb. 4.5 und dem darin von Optuna dargestellten *Parallel Coordinate Plot* zu erkennen. Zusätzlich ist im Anhang die multidimensionale Darstellung der durch die Hyperparameter Optimierung gefunden lokalen Minima in Abb. 5 zu sehen. In Tabelle 4.1 sind die nach dem R2-Score besten fünf Trials der Hyperparameter Optimierung aufgelistet. Diese bestätigen die vorangegangene Evalua-

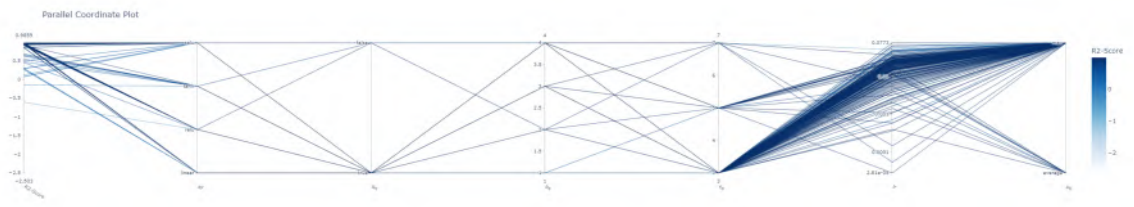


Abbildung 4.5: Koordinaten der einzelnen Versuche der Hyperparameter-Optimierung

tion der in Abb. 4.4 dargestellten Ergebnisse und den pro Trial erkennbaren zunehmenden Fokus des TPE-Algorithmus auf die besten Hyperparameter. Zum Vergleich des Einflusses der unterschiedlichen Aktivierungsfunktionen ist der jeweilige beste Trial in Tabelle 4.2 dargestellt. Die unterschiedliche Trainingsdauer der einzelnen Trials ist auf die Verwendung eines eigens programmierten Abbruchkriteriums zurückzuführen, welches im Sinne der in Abschnitt 2.5.4 aufgeführten Abbruchkriterien ein suboptimales Training frühzeitig beendet und Rechenleistung einspart. Nach einer gewählten Anzahl von 7 Epochen, in welchen sich der R2-Score im Vergleich zur vorangegangenen Epoche nicht verbessern kann oder den Zähler durch einen neuen Bestwert zurücksetzt, beendet das Abbruchkriterium das Training frühzeitig. Die Notwendigkeit eines solchen Kriteriums ist am Trainingsverlauf von Trial 27 zu erkennen.

In Abb. 4.6 sind die Trainingsverläufe der in den Tabellen 4.1 4.2 am Validierungs-R2-Score gelisteten besten Trials dargestellt. Trotz der für CNNs und DNNs empfohlenen tanh Aktivierungsfunktion, verhindern die anderen suboptimal gewählten Hyperparameter eine zielgerichtete Verbesserung der Gewichte pro Epoche. Zu Beginn des Trainings führt die geringe Lernrate `lr` = 0.003952 und eine offensichtlich nicht ideale Initialisierung der Gewichte zu einer allenfalls langsamen Verbesserung innerhalb der ersten sechs Epochen, bis die Lernrate durch den Adam-Algorithmus angepasst wird und das lokale Minimum verlassen wird. Die niedrigere Batch-Größe `bs` = 3 sorgt im Vergleich zu den anderen Trials für stärkere Oszillationen bei der Aktualisierung der Gewichte und somit im Verlauf des Validierungs-R2-Scores. Aus diesem Grund beendet das Abbruchkriterium nach Erreichen des maximalen Validierungs-R2-Scores von 0.924225 in Epoche 14 nach sieben weiteren abfallenden Validierungs-R2-Scores das Training bei Epoche 36. Die Trainingsverläufe der besten fünf Trials unterscheiden sich dabei alleine aufgrund der sich gering unterscheidenden Lernrate und der zufällig Initialisierung der Gewichte. Für weitere Untersuchungen der besten Trials hinterlegt das Abbruchkriterium den Zustand der Gewichte im Punkt des besten Validierungs-R2-Scores und speichert mit dem Befehl `model.save()` die besten DNN-Modelle der Hyperparameter Optimierung.

Die Frage nach der besten Kombination an Hyperparametern hat die mit dem Open-Source Framework Optuna [1] durchgeführte Studie eindeutig beantwortet. Die aus Trial 236 resultierende beste Konfiguration des DNN wird im folgenden Abschnitt mit der maximalen Anzahl an Samplen trainiert und evaluiert.

Tabelle 4.1: Die besten fünf Trials der Hyperparameter Optimierung, geordnet nach dem Valiederungs-R2-Score

Trial Nr.	af	ks	bs	bn	lr	po	R2-Score
236	SelU	3	4	True	0.020837	max	0.985926
142	SelU	3	4	True	0.018730	max	0.984318
101	SelU	3	4	True	0.017017	max	0.983085
98	SelU	3	4	True	0.022326	max	0.982523
212	SelU	3	4	True	0.023148	max	0.982100

Tabelle 4.2: Die besten Trials mit unterschiedlichen Aktivierungsfunktionen der Hyperparameter Optimierung, geordnet nach dem Valiederungs-R2-Score

Trial Nr.	af	ks	bs	bn	lr	po	R2-Score
236	SelU	3	4	True	0.020837	max	0.985926
246	linear	3	4	True	0.01603	max	0.975621
31	RelU	3	5	True	0.030634	max	0.974615
27	tanh	3	3	True	0.003952	average	0.924225

4.3 Bestes DNN Modell Trial 236

Aufbauend auf den Ergebnissen der Hyperparameter Optimierung, wird das Training jenes besten Modells aus Trial 236 fortgeführt. Wie die Abb. 4.6 im vorangegangen Abschnitt zeigt, hat das Modell aus Trial 236 bei einer gesamten Anzahl von 49 Epochen, den maximalen R2-Score von 0.985926 bezogen auf den Anteil an Validierungs-Daten erreicht. Aus Gründen der Effizienz haben die einzelnen trainierten Modelle während der Hyperparameter Optimierung nur einen Bruchteil von 2560 Samplen und somit 3.26% des gesamten Datensatzes gesehen. Dieser Bruchteil reicht jedoch nicht für eine verlässliche Vorhersage und eine Generalisierung des besten Modells von Trial 236 über die restlichen 96.74% des Datensatzes aus. Mittels der Funktion `model_236=tf.keras.models.load_model()` wird der optimale Zustand aus Epoche 39, mit den erlernten Features und Mappings des mit Tensorflow Keras trainierten DNN-Modells aus Trial 263 wieder hergestellt und der Variable `model_236` zugewiesen.

Damit ein Eindruck über die Generalisierung des bis dahin trainierten Modells entsteht, werden die Vorhersagen der zehn Eigenfrequenzen für die dem DNN vorenthaltenen 96.74% der Daten und der Funktion `prediction_236=model_236.predict()` berechnet. Über die Differenz zu den dazugehörigen Ground Truths wird der absolute und relative Fehler berechnet. In Abb. 4.7 ist der absolute Fehler und in Abb. 4.8 der relative Fehler zum Ground Truth der vorenthaltenen Daten dargestellt. In der jeweiligen Teilabbildung *b*) ist ein klassischer Box-Whisker-Plot abgebildet. Darin sind die Fehler in den Vorhersagen für die ersten zehn Eigenfrequenzen f_1 bis f_{10} abgebildet. Der Median in den Fehlern der

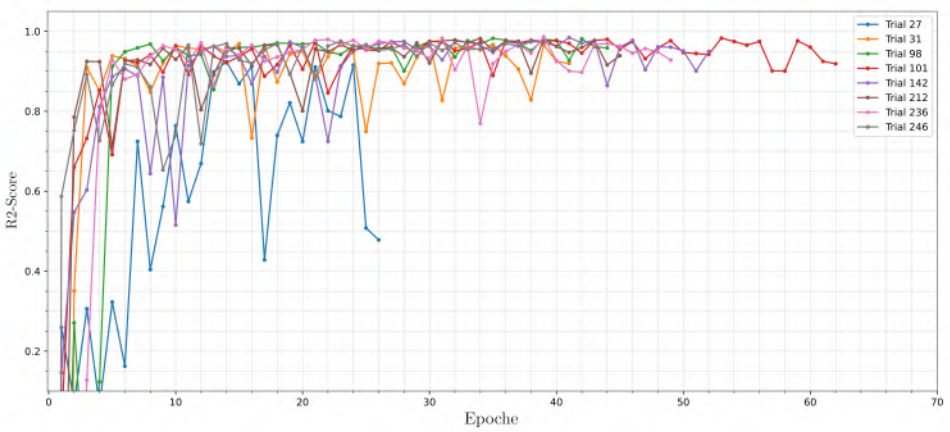


Abbildung 4.6: Zeigt den Trainingsverlauf er besten beste Versuche der Hyperparameter-Optimierung, welche in den Tabellen 4.1 und 4.2 aufgelistet sind. Darin ist der erreichte Validierungs-R2-Score über die Epochen aufgetragen.

einzelnen Eigenfrequenzen ist als orangener Balken dargestellt und liegt innerhalb des ersten Q_1 und dritten Q_3 Quartils, welche die unter beziehungsweise obere Grenze der Box definieren. Wie beispielsweise in Cleff [5] nachzulesen ist, wird der Median auch als zweites Q_2 Quartil bezeichnet. Zwischen den einzelnen Quartilen befinden sich jeweils 25% der Fehler, wodurch die jeweilige Box 50% der Fehler beinhaltet und das Interquartile Range (IQR) aufspannt. In den jeweiligen Eigenfrequenzen befinden sich somit unterhalb von Q_1 25% und unterhalb von Q_3 75% der Fehler. Die Grenzen U_w und L_w eines Fehlerbalken (Whiskers) werden mit

$$U_w = Q_3 + 1.5 \cdot IQR \tag{4.1}$$

beziehungsweise

$$L_w = Q_1 - 1.5 \cdot IQR \tag{4.2}$$

und $IQR = Q_3 - Q_1$ berechnet. Vektoriell können auch die Grenzen der Fehlerbalken $\mathbf{U_w}$, $\mathbf{L_w}$, die Quartile $\mathbf{Q}$ und IQRs $\mathbf{IQR}$ für alle Eigenfrequenzen gleichzeitig formuliert werden. Dies führt auf die zu Gl. 4.1 und 4.2 äquivalenten vektoriellen Schreibweisen

$$\mathbf{U_w} = \mathbf{Q}_3 + 1.5 \cdot \mathbf{IQR}, \tag{4.3}$$

$$\mathbf{L_w} = \mathbf{Q}_1 - 1.5 \cdot \mathbf{IQR} \tag{4.4}$$

mit

$$\mathbf{IQR} = \mathbf{Q}_3 - \mathbf{Q}_1. \tag{4.5}$$

Fehler, welche sich außerhalb der Fehlerbalken befinden, sind als Ausreißer durch eine kreisförmige Markierung in den Box-Whisker-Plots der Abb. 4.7 *b*) und 4.8 *b*) dargestellt. Ein negativer Fehler bedeutet, dass die Vorhersage den Ground-Truth überschätzt. Umgekehrt bedeutet ein positiver Fehler, dass der Ground-Truth unterschätzt wird. Speziell

Tabelle 4.3: Zusammenfassung der in Abb. 4.7 dargestellten absoluter Fehler der einzelnen Eigenfrequenzen f_1 bis f_{10} in [Hz].

Fehler	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
Mittelwert	177.9	335.4	479.0	-235.4	653.2	245.5	522.2	501.6	920.1	1072.0
STD	920.6	1214.2	1265.3	1721.4	1958.3	2928.6	2646.0	2818.9	2736.0	2890.4
Min. Fehler	-12481.4	-20668.3	-13103.1	-14840.3	-20836.4	-569670.6	-474040.1	-506393.5	-31271.4	-31084.3
Q_1 = 25%	-203.5	-200.2	-189.9	-670.5	-306.7	-621.9	-347.4	-496.2	-535.6	-452.7
Q_2 = 50%	-30.2	16.9	168.2	-168.7	185.2	-82.4	222.6	194.9	399.4	544.3
Q_3 = 75%	290.7	414.1	859.9	552.2	1069.6	573.0	1046.5	1124.9	1579.7	2103.2
Max. Fehler	13048.5	16919.7	20986.7	22558.7	32316.9	23600.9	28707.2	26287.0	44309.8	52530.8
IQR	494.2	614.3	1049.7	1222.7	1376.3	1194.9	1393.9	1621.1	2115.3	2555.9
L_w	-944.8	-1121.6	-1764.4	-2504.5	-2371.1	-2414.2	-2438.2	-2927.8	-3708.6	-4286.6
U_w	1032.0	1335.6	2434.5	2386.2	3134.0	2365.4	3137.3	3556.4	4752.7	5937.0

Tabelle 4.4: Zusammenfassung der in Abb. 4.8 dargestellten relativen Fehler der einzelnen Eigenfrequenzen f_1 bis f_{10} in [%].

Fehler	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
Mittelwert	-0.30	1.56	2.17	-3.21	2.07	-0.56	1.40	1.12	1.95	1.94
STD	16.91	13.64	11.31	13.23	11.21	9.70	9.33	9.46	10.46	10.02
Min. Fehler	-474.20	-358.15	-157.53	-147.90	-349.19	-411.89	-478.28	-330.88	-222.36	-117.56
Q_1 = 25%	-7.79	-4.70	-3.29	-9.48	-3.08	-5.50	-2.80	-3.63	-3.25	-2.66
Q_2 = 50%	-1.36	0.40	2.17	-2.18	1.61	-0.68	1.45	1.19	2.05	2.51
Q_3 = 75%	5.91	6.60	7.55	4.88	7.41	3.81	5.77	5.84	7.31	7.06
Max. Fehler	93.15	94.90	90.83	96.10	92.78	73.82	79.23	77.55	93.61	95.99
IQR	13.70	11.30	10.84	14.36	10.49	9.31	8.57	9.47	10.56	9.71
L_w	-28.34	-21.65	-19.55	-31.02	-18.83	-19.46	-15.66	-17.84	-19.08	-17.23
U_w	26.46	23.55	23.80	26.42	23.15	17.77	18.63	20.04	23.14	21.62

das Überschätzen des Ground-Truth fällt in der Größe der absoluten wie relativen Fehler deutlich stärker aus, weshalb in den Box-Whisker-Plots die schlechtesten drei Vorhersagen für eine bessere Darstellung der restlichen Fehler nicht mehr enthalten sind. Diese sind jedoch in den Tabellen 4.3 und 4.4 aufgeführt. Die quantitativ größeren negativen Fehler stehen in Zusammenhang mit der RelU Aktivierungsfunktion im Output-Layer des DNN, welche die Outputs auf einen Wertebereich größer gleich null begrenzt. Die absoluten und relativen Fehler der einzelnen Eigenfrequenzen, die innerhalb der durch die Gl. 4.1 4.1 berechneten Fehlerbalken liegen, sind in den dazugehörigen Histogrammen, in den Abb. 4.7 *a*) und Abb. 4.8 *a*) dargestellt. Wie zu erwarten ist, nehmen die relativen Fehler, über die größer werden Eigenfrequenzen von f_1 bis f_{10} ab. Umgekehrt nehmen die absoluten Fehler von f_1 bis f_{10} zu. Der Anteil an Samples mit einem absoluten Fehler innerhalb der Fehlerbalken beträgt 73.35%. Dies führt nach dem absoluten Fehler auf einen Anteil an Ausreißern von 26.25%. Der Anteil an Sample mit einem relativen Fehler innerhalb der Fehlerbalken beträgt dagegen 85.86%. Dies führt nach dem relativen Fehler auf einen Anteil an Ausreißern von 14.14%. Eine zusammenfassende Darstellung der in Abb. 4.7 abgebildeten absoluten Fehler liefert die Tabelle 4.3. Analog ist die Zusammenfassung der in Abb. 4.8 abgebildeten relativen Fehler in Tabelle 4.4 dargestellt.

Die aufgelisteten und visualisierten absoluten und relativen Fehler von `model_236` zeigen die Folgen von Underfitting, welche aus dem Training mit einem zu kleinen Anteil von 3.26% des gesamten Datensatzes resultiert. Wie im Abschnitt der Hyperparameter Optimierung und zu Beginn des Abschnitts erläutert, ist das Underfitting und die nicht ausreichende Generalisierung der in den Trials trainierten Modelle in Bezug auf die ansonsten enorme Rechenleistung vernachlässigbar, da dieses Modell für die Bestimmung der besten Hyperpa-

rameter nicht auf einem repräsentativen Anteil des Datensatzes trainiert werden muss. Aus diesem Grund sorgt im folgenden Abschnitt das sogenannte *Fine-Tuning* von `model_236` für eine Reduktion der absoluten und relativen Fehler.

a) b)

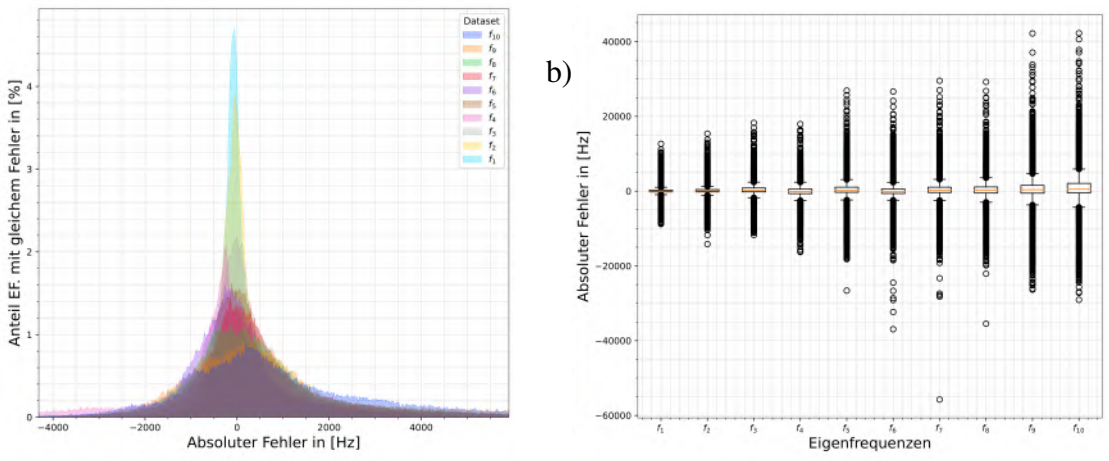


Abbildung 4.7: Absoluter Fehler über Test-Datensatz von Trial Nr. 236 nach der Hyperparameter-Optimierung. Die Modell-Vorhersagen von 76000 Samplen 96.74% der gesamten Daten) basieren auf einem Training mit 2560 Samples (3.26% der gesamten Daten). Abb. a) zeigt den absoluten Fehler zwischen der Vorhersage und dem Ground Truth als Histogramm innerhalb der Grenzen der Fehlerbalken ($\mathbf{L_w}$,$\mathbf{U_w}$). Abb. b) zeigt den absoluten Fehler zwischen der Vorhersage und dem Ground Truth als Box-Whisker-Plot.

4.3.1 Fine-Tuning DNN Modell Trial 236

Fine-Tuning beschreibt im Bereich ML das erneute Trainieren eines bereits trainierten Modells mit einer niedrigen Lernrate. Auf diese Weise werden die bereits aktualisierten Gewichte, die im idealen Fall einen Punkt nahe dem globalen Minimum der Loss-Landscape beschreiben, als Ausgangspunkt für den Adam-Optimierungsalgorithmus verwendet. Die geringe Lernrate soll ein Verlassen der Umgebung um das globale Minimum verhindern und eine weiterführende Konvergenz in Richtung des Optimums bewirken. Für eine bestmögliche Generalisierung des DNN aus Trial 236 wird dieses mit dem maximalen Anteil von 42% des gesamten Datensatzes trainiert und validiert. Wie in Abschnitt 3.2.5 beschrieben, wird das Modell während des Trainings, nach jeder Epoche, mit einem Anteil von 15% der Trainings-Daten validiert. So entsteht die Aufteilung in 35.7% Trainings-Daten, 6.3% Validierungs-Daten und 58% Test-Daten. Der Trainingsverlauf des Fine-Tunings von Trials 236 ist in Abb. 4.9 dargestellt. Abb. 4.9 *a*) zeigt den berechnet Loss und Abb. 4.9 *b*) den berechneten R2-Score für die Trainings-Daten in Blau beziehungsweise den

a) b)

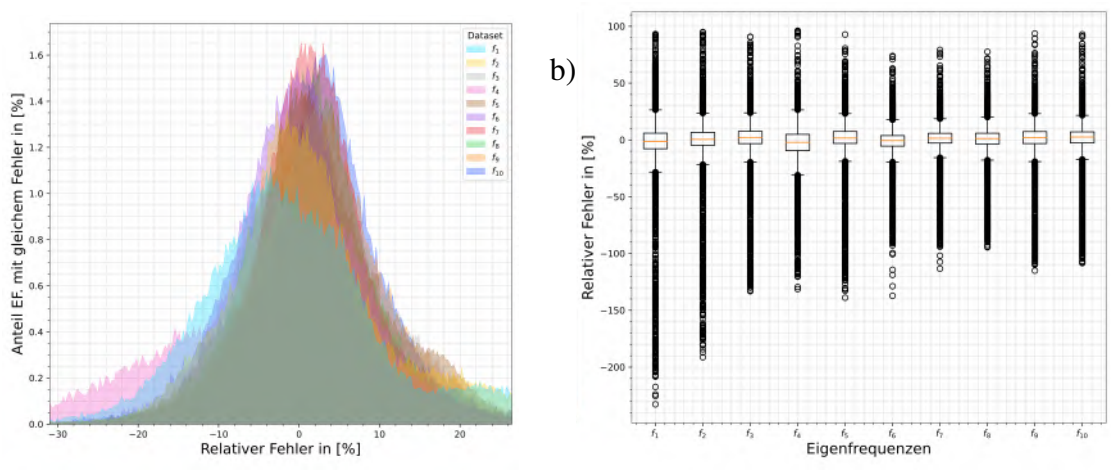
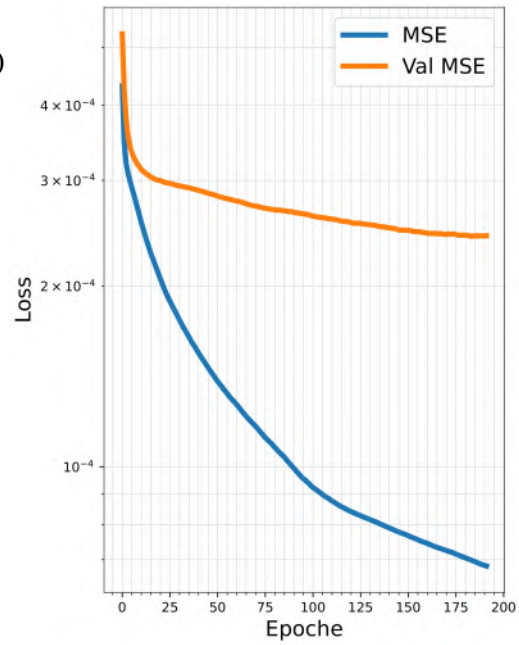


Abbildung 4.8: Relativer Fehler über Test-Datensatz von Trial Nr. 236 nach der Hyperparameter-Optimierung. Die Modell-Vorhersagen von 76000 Samplen 96.74% der gesamten Daten) basieren auf einem Training mit 2560 Samples (3.26% der gesamten Daten). Abb. a) zeigt den relativen Fehler zwischen der Vorhersage und dem Ground Truth als Histogramm innerhalb der Grenzen der Fehlerbalken ($\mathbf{L_w}$,$\mathbf{U_w}$). Abb. b) zeigt den relativen Fehler zwischen der Vorhersage und dem Ground Truth als Box-Whisker-Plot.

Validierungs-Daten in Orange. Wird der Validerungs-R2-Score betrachtet, nimmt das Fine-Tuning das Training wieder bei dem R2-Score von 0.9859 auf, welchen die Hyperparameter-Optimierung am Ende von Trial 236 erreicht hat. Mit einer niedrigen Lernrate von `lr` = 10^{-5} und der maximal möglichen Batch-Größe von 2^5 = 32 Samples werden die Gewichte schrittweise verbessert und der Validierungs- und Trainings-Loss konvergiert ohne Oszillationen mit jeder weiteren Epoche gegen das Minimum. Auf diese Weise kann der Validierungs-R2-Score über annähernd 200 Epochen weiter verbessert werden und erreicht das Maximum von 0.9914.

Die an dem höheren Validierungs-R2-Score zu erkennende verbesserte Generalisierung kann erneut durch die Berechnung der absoluten und relativen Fehler zwischen den Vorhersagen des DNN und den Ground-Truths des Test-Datensatzes gezeigt werden. In Abb. 4.10 b) und Abb. 4.11 b) sind ohne Ausnahme alle berechneten absoluten und relativen Fehler in einem Box-Whisker-Plot dargestellt. Darin sind die Ausreißer erneut mit kreisförmigen Markierungen hervorgehoben. Die absoluten und relativen Fehler weisen im Vergleich zu Abb. 4.7 b) eine deutliche geringere Dichte auf. Speziell bei den Ausreißern bezogen auf den relativen Fehler, befinden diese sich deutlich näher an den Grenzen der Fehlerbalken. Die dazugehörigen Histogramme in Abb. 4.10 a) und Abb. 4.11 a) zeigen die Verteilung der Sample, deren Fehler sich innerhalb der Fehlerbalken ($\mathbf{L_w}$, $\mathbf{U_w}$) befinden. Die zu den Abbildungen gehörenden statistischen Auswertungen der Fehler sind in den Tabellen 4.5 und 4.6 zusammengefasst.

a)



b)

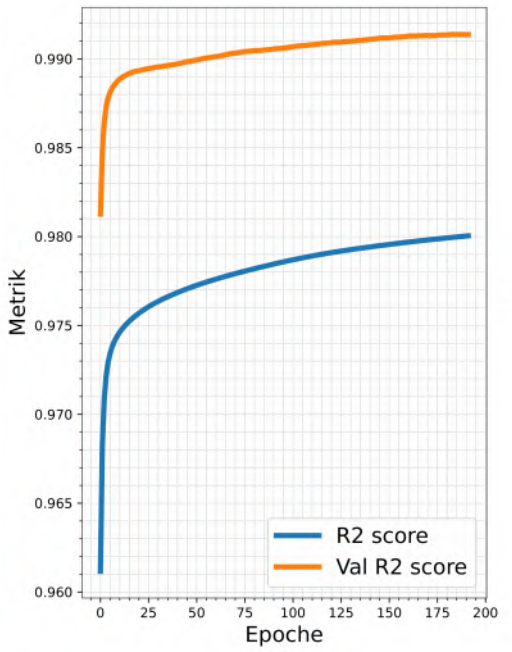


Abbildung 4.9: Abb. a) zeigten den Trainings-Loss in Blau und den Validierungs-Loss in Orange über 200 Epochen während des Fine-Tunings mit 33280 Sample (42% der gesamten Daten). Abb. b) zeigt analog den Trainings-R2-Score in Blau, sowie den Validierungs-R2-Score in Orange von Trial Nr. 236 während des Fine-Tunings mit 33280 Sample (42% der gesamten Daten) ausgehend des Trainingsstands nach der Hyperparameter-Optimierung und dem R2-Score von 0.985926

Werden die Fehler des Modells aus Trial 236 nach der Hyperparameter Optimierung mit den Ergebnissen nach dem Fine-Tuning des gleichen Modells verglichen, können die folgenden Verbesserungen formuliert werden:

- Der betragsmäßig maximale relativen Fehler wird durch das Fine-Tuning des DNN um 303, 1%.
- Der maximale relative Fehler von 50% der Vorhersagen über die Eigenfrequenzen f_1 bis f_{10} verbessert sich von 9.48% auf 5.53%.
- Die IQR verbessert sich von 14.36% auf 9.59%
- Nach dem Fine-Tuning besitzen 82.14% der Vorhersagen einen relativen Fehler zwischen 19.92% und −18.45%

Trotz des Fine-Tunings mit dem größtmöglichen Anteil Trainings-Daten des gesamten Datensatzes, zeigen speziell die noch immer großen absoluten und relativen Fehler der Ausreißer in den Vorhersagen, dass die limitierte Anzahl an Trainings-Daten und die unterrepräsentierten Merkmalen der Ausreißer zu einer nicht vollständigen Generalisierung

Tabelle 4.5: Zusammenfassung der in Abb. 4.10 dargestellten absoluter Fehler der einzelnen Eigenfrequenzen f_1 bis f_{10} in [Hz].

Fehler	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
Mittelwert	195.93	206.42	185.25	223.19	355.75	330.58	314.31	286.63	425.83	354.26
STD	612.85	779.53	867.18	743.10	1395.78	1289.25	1295.71	1421.45	1754.66	1902.73
Min. Fehler	-7352.17	-9300.36	-10103.64	-8013.39	-15522.34	-13586.33	-14697.18	-15249.61	-20014.12	-22625.57
Q_1 = 25%	-92.83	-111.17	-234.25	-161.98	-233.28	-275.89	-292.84	-364.01	-305.40	-479.95
Q_2 = 50%	24.17	6.52	5.09	82.66	-14.69	62.37	14.37	-38.81	20.13	-15.34
Q_3 = 75%	364.95	293.80	516.68	461.88	666.04	598.15	774.85	637.00	728.98	881.32
Max. Fehler	3751.70	4790.42	5047.60	5657.62	10952.39	11952.02	10436.83	10181.51	20971.40	17585.00
IQR	457.78	404.97	750.93	623.86	899.32	874.04	1067.69	1001.00	1034.38	1361.26
L_w	-779.51	-718.62	-1360.65	-1097.76	-1582.27	-1586.95	-1894.39	-1865.51	-1856.96	-2521.84
U_w	1051.63	901.25	1643.07	1397.66	2015.02	1909.22	2376.39	2138.51	2280.54	2923.21

Tabelle 4.6: Zusammenfassung der in Abb. 4.11 dargestellten relativen Fehler der einzelnen Eigenfrequenzen f_1 bis f_{10}. in [%]

Fehler	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
Mittelwert	-0.79	-0.25	-0.72	0.36	0.22	0.13	0.07	-0.11	0.34	-0.29
STD	13.14	11.18	9.16	6.23	8.29	6.68	6.42	6.42	6.79	7.08
Min Fehler	-177.18	-166.25	-117.47	-67.73	-114.82	-77.31	-79.94	-77.18	-95.09	-95.50
Q_1 = 25%	-4.06	-2.82	-3.98	-2.44	-2.48	-2.74	-2.36	-2.80	-2.10	-2.86
Q_2 = 50%	0.68	0.15	0.07	0.80	-0.14	0.44	0.10	-0.25	0.11	-0.07
Q_3 = 75%	5.53	3.88	4.10	3.41	3.68	3.21	3.25	2.61	2.80	2.78
Max. Fehler	56.81	41.49	39.99	44.63	40.56	34.90	36.67	36.38	40.32	35.51
IQR	9.59	6.70	8.08	5.85	6.16	5.95	5.62	5.41	4.90	5.64
L_w	-18.45	-12.88	-16.09	-11.21	-11.72	-11.66	-10.79	-10.92	-9.45	-11.32
U_w	19.92	13.94	16.21	12.19	12.91	12.13	11.68	10.73	10.14	11.24

und Underfitting des DNN-Modells führen. Die folgenden Unterkapitel zeigen die besten und schlechtesten Vorhersagen des mittels Fine-Tunings verbesserten DNN-Modells und die den Vorhersagen zugrunde liegenden Verteilungen der Inputs und Outputs. Mittels dieser Verteilungen kann gezeigt werden, welche Sample in dem bereitgestellten Datensatz unterrepräsentiert sind und somit eine Generalisierung des DNN verhindern.

4.3.2 Analyse der besten Vorhersagen

Wie die Auswertung der relativen und absoluten Fehler gezeigt hat, ist das DNN in der Lage für einen großen Anteil von 82.14% akzeptable bis sehr gute Vorhersagen zu treffen. Um einen Eindruck über das Potenzial des DNN zu erhalten, sind in Abb. 4.12 die besten fünf Vorhersagen mit den niedrigsten relativen Fehlern abgebildet. Darin sind die Ground-Truths der einzelnen Eigenfrequenzen durch farbige Punkte markiert. Die gestrichelten Linien verbinden die einzelnen Ground-Truths miteinander. Die zu den Ground Truths farblich abgestimmten Linien stellen an der entsprechenden Positionen der Eigenfrequenzen f_1 bis f_{10} die dazugehörigen Vorhersagen des DNN dar. Die Verbindungslinien zwischen den einzelnen Eingenfrequenzen f_1 bis f_{10} der Vorhersagen und Ground-Thruths, sind lineare Interpolationen, die für eine ansprechendere Visualisierung gewählt werden, aber keine tatsächlichen Werte repräsentieren. Die durchgängigen Linien der Vorhersagen bilden den stückweise zunehmenden Verlauf der zugrundeliegenden Eigenfrequenzen (Ground-Truths) annähernd identisch ab.

Einen Einblick über die Wertebereiche der einzelnen Eigenfrequenzen f_1 bis f_{10}, welche

a) b)

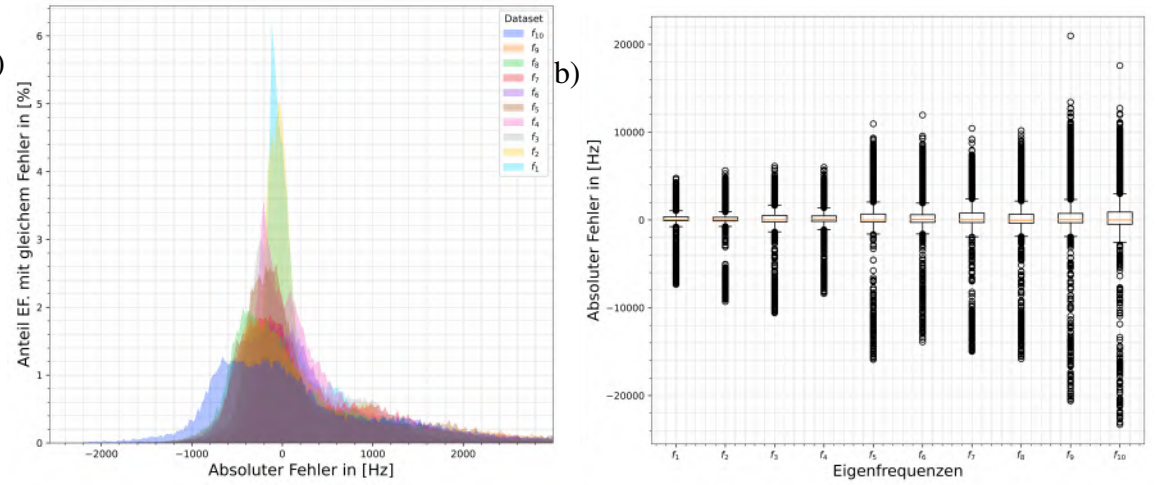


Abbildung 4.10: Absoluter Fehler über Test-Datensatz von Trial Nr. 236 nach Fine-Tuning. Die Modell-Vorhersagen von 45280 Samplen (58% der gesamten Daten) basieren auf einem Training mit 33280 Samples (42% der gesamten Daten). Abb. a) zeigt den absoluten Fehler zwischen der Vorhersage und dem Ground Truth als Histogramm innerhalb der Grenzen der Fehlerbalken ($\mathbf{L_w}$,$\mathbf{U_w}$). Abb. b) zeigt den relativen Fehler zwischen der Vorhersage und dem Ground Truth als Box-Whisker-Plot.

Tabelle 4.7: Aus Abb. 4.13 ermittelte Eigenfrequenzen f_1 bis f_{10} in Hz mit hohem Potential minimale relative Fehler zwischen Vorhersage und Ground Truth zu erzielen.

f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
5000	6000	8000	9500	11000	14000	15000	16000	18000	22000

innerhalb der dazugehörigen Fehlerbalken ($\mathbf{L_w}$, $\mathbf{U_w}$) aus Abb. 4.11 liegen und akzeptable bis sehr gute Vorhersagen liefern, sind in Abb. 4.13 abgebildet. Werden die Stellen mit den größten prozentualen Anteilen in den Verteilungen der einzelnen Eigenfrequenzen in Abb. 4.13 mit denen des gesamten Datensatzes Abb. 3.8 im Abschnitt 3.2.3 verglichen, dann fallen diese mit den Werten der skalierten Eigenfrequenzen zusammen, an denen sich die größten Anteile von Rotoren und Statoren überschneiden. Dies zeigt, dass das DNN besonders für die im Datensatz sehr gut vertreten Eigenfrequenzen verlässliche Vorhersagen liefert. Somit versprechen Sample, welche Eigenfrequenzen um die in der Tabelle 4.7 gelisteten Werte besitzen, eine hohe Genauigkeit in der Vorhersage des DNN. Werden die in Abb. 4.13 einzeln dargestellten Verteilungen der Eigenfrequenzen zusammen genommen, repräsentieren diese eine Teildarstellung der in Abb. 3.6 zusehenden Schnittmenge der gesamten im Datensatz enthaltenen Rotoren und Statoren. Diese Teilmenge der Sample mit guten Vorhersagen beinhaltet jedoch nicht die in den Statoren enthaltenen Ausreißer mit Eigenfrequenzen größer 45000 Hz, welche somit trotz Preprocessing und Log-Normalisierung zu schlechten Vorhersagen führen. Wird die Verteilung der Volumen, deren relativer Fehler innerhalb der Fehlerbalken ($\mathbf{L_w}$, $\mathbf{U_w}$) aus Abb. 4.11 liegen, in Abb. 4.14 betrachtet, ist die zugrunde

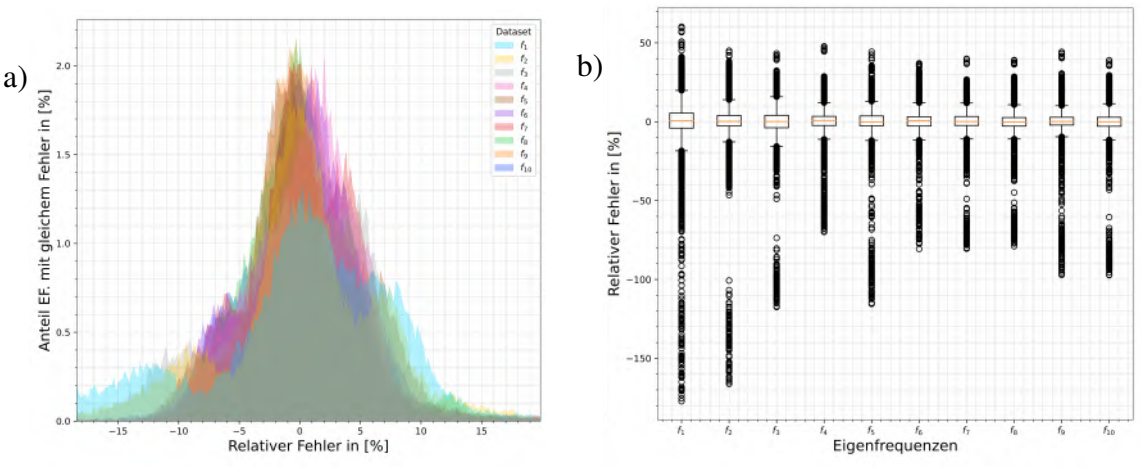


Abbildung 4.11: Relativer Fehler über Test-Datensatz von Trial Nr. 236 nach Fine-Tuning. Die Modell-Vorhersagen von 45280 Samplen (58% der gesamten Daten) basieren auf einem Training mit 33280 Samples (42% der gesamten Daten). Abb. a) zeigt den relativen Fehler zwischen der Vorhersage und dem Ground Truth als Histogramm innerhalb der Grenzen der Fehlerbalken ($\mathbf{L_w}$,$\mathbf{U_w}$). Abb. b) zeigt den relativen Fehler zwischen der Vorhersage und dem Ground Truth als Box-Whisker-Plot.

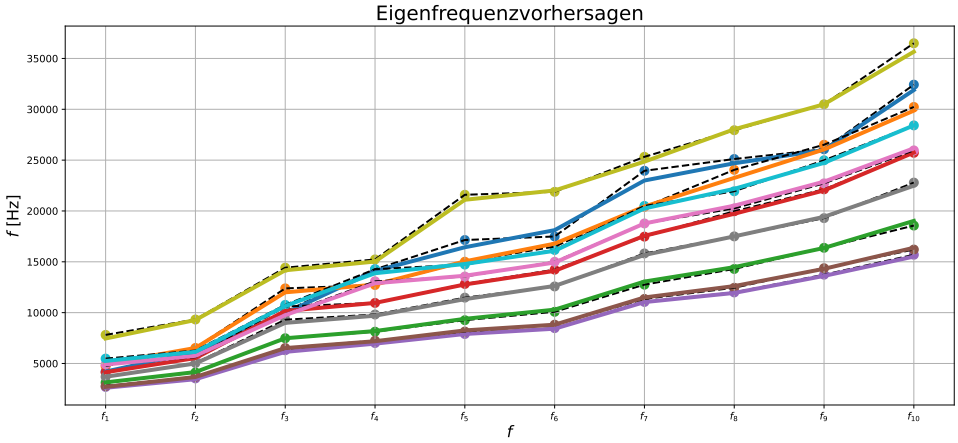


Abbildung 4.12: Die zehn Vorhersagen mit geringsten relativen Fehler: Die Ground-Truths sind in unterschiedlichen Farben für die zehn Vorhersagen als Punkte abgebildet. Der schematische Verlauf der Eigenfrequenzen ist mittels gestrichelter schwarzer Linien dargestellt. Die Schnittpunkte der farbig an die dazugehörigen Ground-Truths abgestimmten Linien, repräsentieren die Vorhersagen an den Positionen der einzelnen Eigenfrequenzen f_1 bis f_{10}.

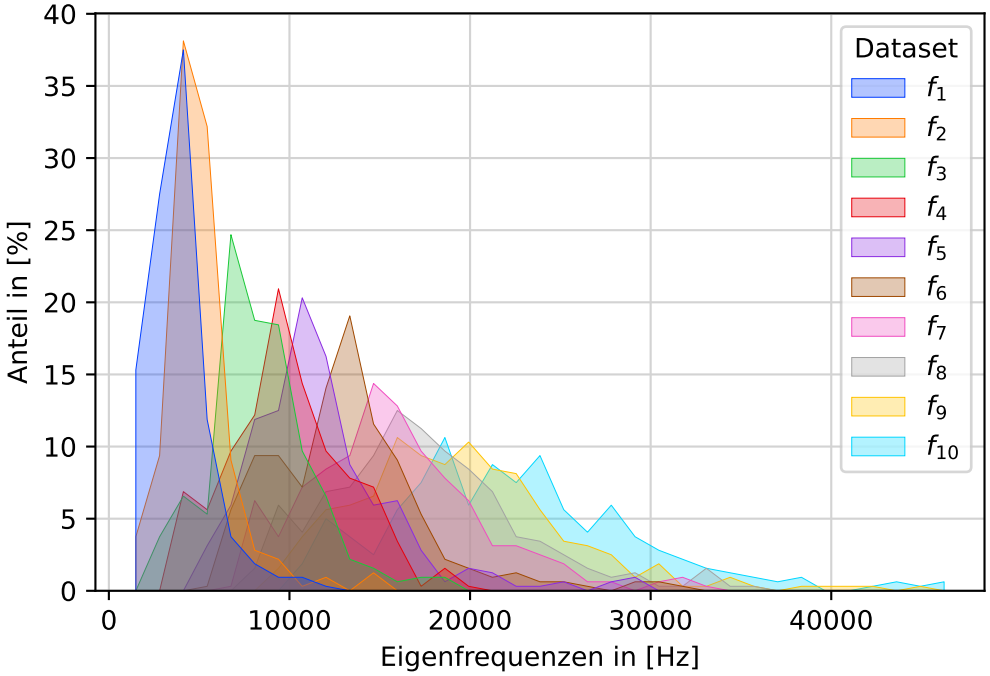


Abbildung 4.13: Verteilung der Eigenfrequenzen (Ground Truths) mit den niedrigsten relativen Fehlern

liegende Verteilung des gesamten Datensatzes in Abb. 3.9 erkennbar. Die am häufigsten auftretende Volumen liegen in beiden Verteilungen im Bereich von 1000 bis 1500mm^3 für Statoren und Rotoren. Unter den besten Vorhersagen sind Volumen bis etwa 4000mm^3 enthalten. Alle Sample, die ein größeres Volumen als 4000mm^3 besitzen, sind somit in den Ausreißern enthalten und führen zu schlechten Vorhersagen des DNN. In Abb. 4.15 sind Schnittebenen in den Voxelgittern der besten 100 VSBs abgebildet. Die abgebildeten Schnittebenen stellen das jeweilige vierundsechzigste 2D Voxelgitter in Richtung der ersten Dimension des entsprechenden 3D Voxelgitters eines VSBs dar. Die einzelnen Schnittebenen zeigen die charakteristische umgekehrte Orientierung der Rotoren und Statoren in einem Hochdruckverdichter. Diese um 90 Grad versetzte Orientierung ermöglicht dem DNN das den 3D Voxelgittern zu zugrundeliegende Merkmal der Statoren und Rotoren zu erkennen und diese Eigenschaft für die Vorhersage der Eigenfrequenzen zu nutzen. Diese Fähigkeit zwischen Statoren und Rotoren zu unterscheiden wird im folgenden Abschnitt 4.3.5 gezeigt. Abgesehen von dem offensichtlich auftretenden Merkmal, mit dem Rotoren und Statoren voneinander unterschieden werden können, ist eine verallgemeinerte Aussage über die präferierten geometrischen Eigenschaften, die ein VSB erfüllen muss, damit für diese das DNN zuverlässige Vorhersagen liefert, anhand Abb. 4.15 reine Spekulation. Eine umfangreiche Untersuchung der geometrischen Eigenschaften und Merkmale der verwendeten VSBs und die Entwicklung eines Clustering-Verfahrens würde einen interessanten Ansatzpunkt für

zukünftige Arbeiten zur Optimierung der Vorhersagen von dreidimensionalen CNNs bieten.

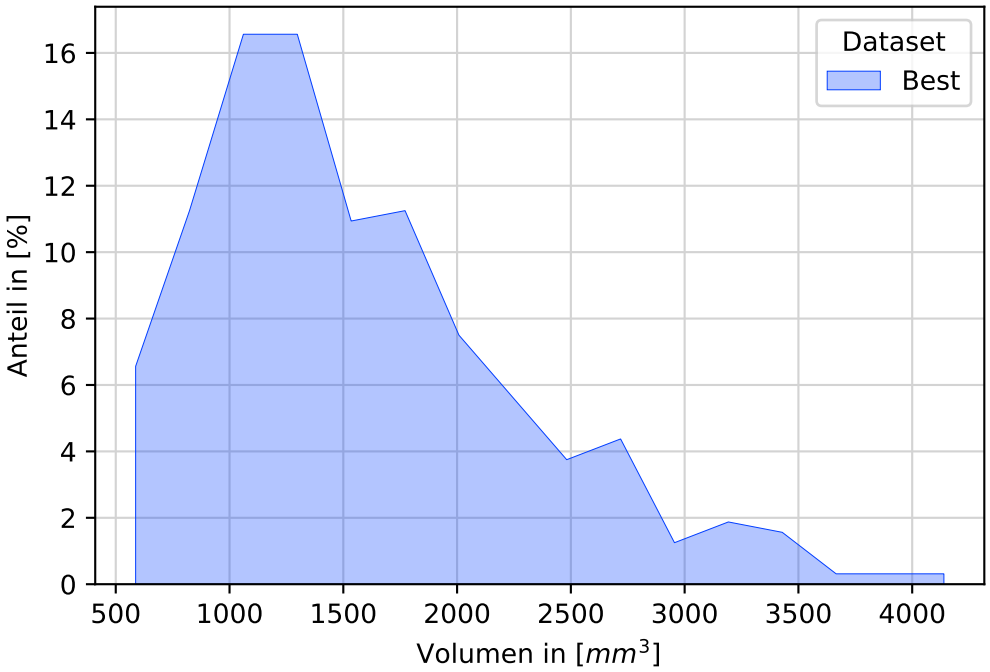


Abbildung 4.14: Verteilung der Volumen (Input 2) mit den niedrigsten relativen Fehlern

4.3.3 Analyse der schlechtesten Vorhersagen

Nach der Analyse der besten Vorhersagen und den in Input- und Output-Daten hinterlegten Merkmale und Verteilungen, soll eine analoge Untersuchung der schlechtesten Vorhersagen, weitere Erkenntnisse über die von dem DNN präferierten Eigenschaften in dem zugrunde liegenden Datensatz liefern. Entsprechend Abb. 4.12 sind in Abb. 4.16 die zehn schlechtesten Vorhersagen abgebildet. Wie bereits zuvor erläutert und in Abb. 4.11 *b*) beziehungsweise Abb. 4.10 *b*) erkennbar ist, besitzen die größten relativen und absoluten Fehler ein negatives Vorzeichen und zeigen an, dass die Vorhersage den Ground-Truth im Wert überschätzt. Die im Betrag größeren Überschätzungen wie Unterschätzungen des Ground-Truths sind auf die RelU-Aktivierungsfunktion zurückzuführen, welche die Vorhersagen auf Werte größer Null begrenzt. Die in Abb. 4.16 enthaltene signifikante Erkenntnis kann anhand der farbigen Verläufe der Vorhersagen abgeleitet werden. Denn trotz des Versatzes, welcher in Form der absoluten Fehler zwischen der Vorhersagen und den Ground-Truths vorliegt, hat das DNN ein Mapping erlernt, welches die Konvention der im Wert aufsteigend sortierten Eigenfrequenzen abbildet. Die Betrachtung der einzelnen Verteilungen der Eigenfrequenzen,

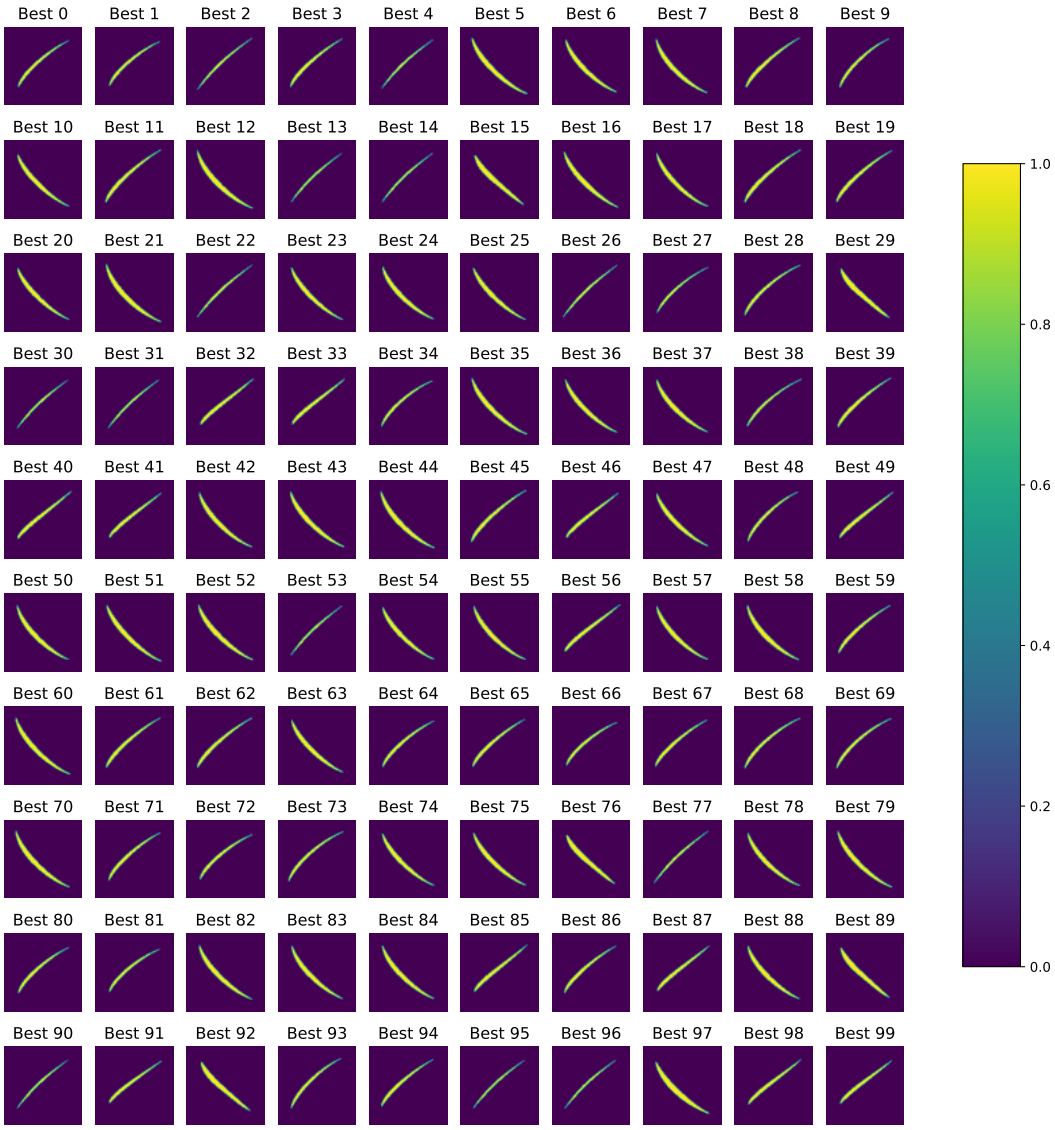


Abbildung 4.15: Abbildung von Schnittebenen der besten 100 VSBs Voxelgitter (Input 2) mit dem niedrigsten relativen Fehlern

dargestellt in Abb. 4.17, ermöglicht eine Analyse der Ursachen für die großen relativen Fehler der in Abb. 4.11 *b*) darstellten Ausreißer. Ähnlich den Stellen mit den größten Anteilen in den einzelnen logarithmisch skalierten Eigenfrequenzen in Abb. 4.13, liegen die Werte der einzelnen Eigenfrequenzen, welche in den Ausreißern aus Abb. 4.17 am häufigsten auftreten, in den jeweiligen Bereich der größten Dichte der Eigenfrequenz-Verteilungen des gesamten Datensatzes Abb. 3.6 im Abschnitt 3.2.3, an denen sich die größten Anteile von Rotoren und Statoren überschneiden. Dennoch weichen die in Tabelle 4.8 aufgeführten Werte der Eigenfrequenzen mit den größten Anteilen in den Ausreißern etwas von denen der besten Vorhersagen in Tabelle 4.7 ab. Die Schnittmenge der Verteilungen der besten und

Tabelle 4.8: Aus Abb. 4.17 ermittelte Eigenfrequenzen f_1 bis f_{10} in Hz mit hohem Potential maximale relative Fehler zwischen Vorhersage und Ground Truth zu erzielen.

f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
5000	6000	8000	11000	13000	16000	18000	20000	21000	23000

schlechtesten Eigenfrequenzen der Abb. 4.13 und 4.17 lässt den Schluss zu, dass Sample deren Vorhersagen zu den Ausreißern gezählt werden, aber Eigenfrequenzen in den idealen Frequenzbereichen besitzen entweder unterrepräsentierte Volumen, beispielsweise größer $4000mm^3$ oder geometrische Merkmale in den Voxelgittern besitzen.

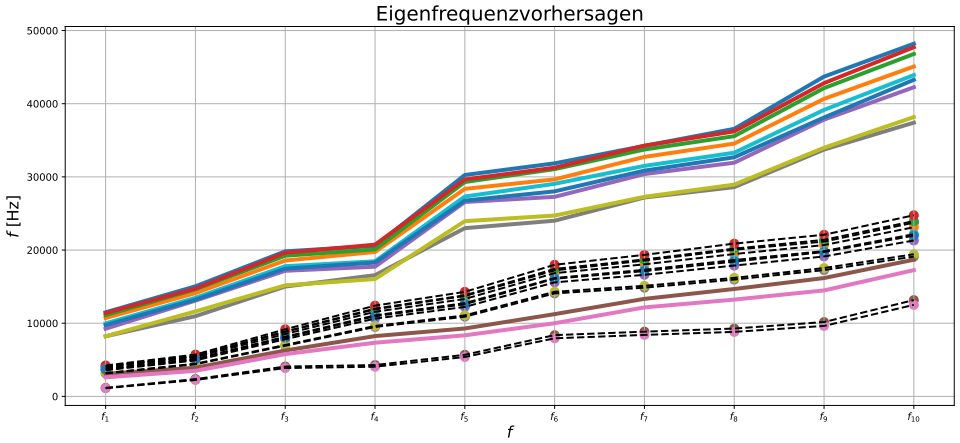


Abbildung 4.16: Vorhersagen mi den größten relativen Fehlern in den zehn Eigenfrequenzen f_1 bis f_{10}

Die Verteilung der Volumen in den Ausreißern der relativen Fehler aus Abb. 4.11 bilden ebenfalls die Verteilung des gesamten Datensatzes aus Abb. 3.9 ab. Das Volumen, welches circa 25% der Ausreißer besitzen, beträgt $800mm^3$. Da Sample mit einem Volumen um den Wert $800mm^3$ auch in der Verteilung der besten Vorhersagen in Abb. 4.14 am häufigsten vertreten sind, müssen diese und alle weiteren Schnittmengen entweder dazugehörige Eigenfrequenzen oder geometrische Merkmale der Voxelgitter aufweisen, welche in den Datensätzen unterrepräsentiert sind. Im Fall der Sample mit einem Volumen größer $4000mm^3$ sind nicht in der Verteilung der akzeptablen Vorhersagen Abb. 4.14 enthalten, aus diesem Grund im gesamten Datensatz extrem unterrepräsentiert und resultieren in Vorhersagen mit sehr hohen relativen wie absoluten Fehlern. Zuletzt sind in Abb. 4.19 die Schnittebenen der Voxelgittern der schlechtesten 100 VSBs abgebildet. Die abgebildeten Schnittebenen stellen das jeweilige vierundsechzigste 2D Voxelgitter in Richtung der ersten Dimension des entsprechenden 3D Voxelgitters eines VSBs dar. Wie bereits bei den besten Vorhersagen verdeutlicht, verlangt eine qualitative Aussage eine umfangreiche Untersuchung der geometrischen Eigenschaften und Merkmale der verwendeten VSBs und die Entwicklung eines Clustering-Verfahrens, welche diese den Ausreißern eindeutig zuweist.

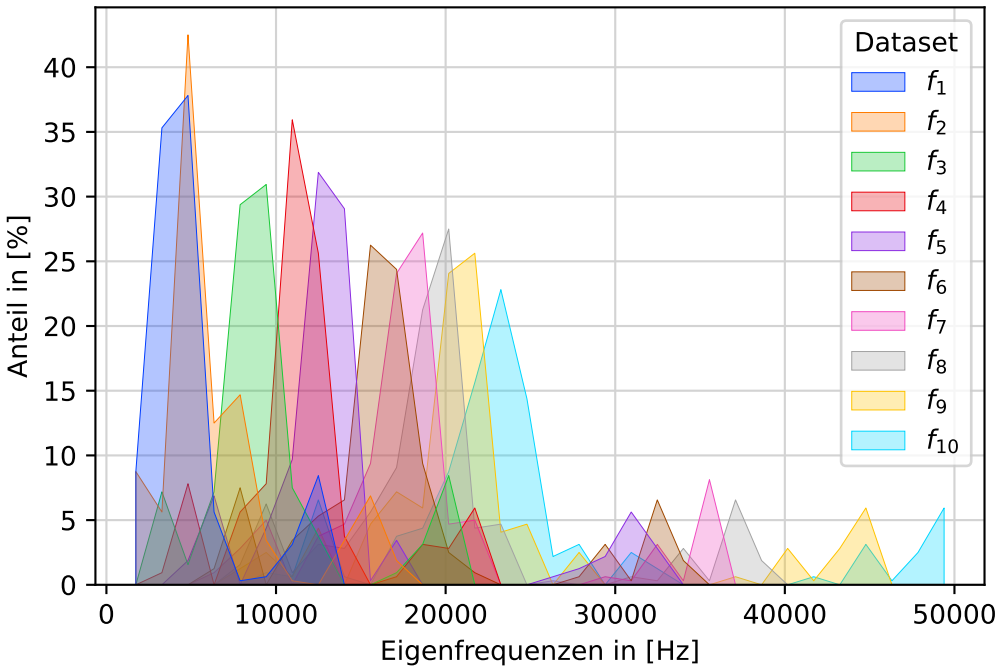


Abbildung 4.17: Verteilung der Eigenfrequenzen (Ground Truths) mit dem höchsten relativen Fehler

Zusammenfassung Fine-Tuning und Fehleranalyse von DNN Trial 236:

- Das Fine-Tuning mit 36% Trainingsdaten erreicht einen Trainings-R2-Score von 0.9800 und einen Validierungs-R2-Score von 0.9914
- Die Evaluation und Vorhersage der zehn Eigenfrequenzen für die restlichen 58% Testdaten
- Sehr gute Vorhersagen für 50% der Testdaten mit einem relativen Fehler kleiner als 5.53% in den einzelnen Eigenfrequenzen.
- 82.14% der Eigenfrequenzen besitzen einen geringeren relativen Fehler als 19.92%.
- Der Anteil an Ausreißern in den Testdaten mit einem größeren Fehler als 19.92% beträgt 17.86%.
- Der Anteil an Ausreißern mit hohen Fehlern entsteht durch die inhomogen verteilten Input- und Output-Daten im bereitgestellten Datensatz.
- Trotz einer Optimierung der Verteilungen mittels einer logarithmischen Skalierung bleiben gewisse VSBs und deren Merkmale unterrepräsentiert.

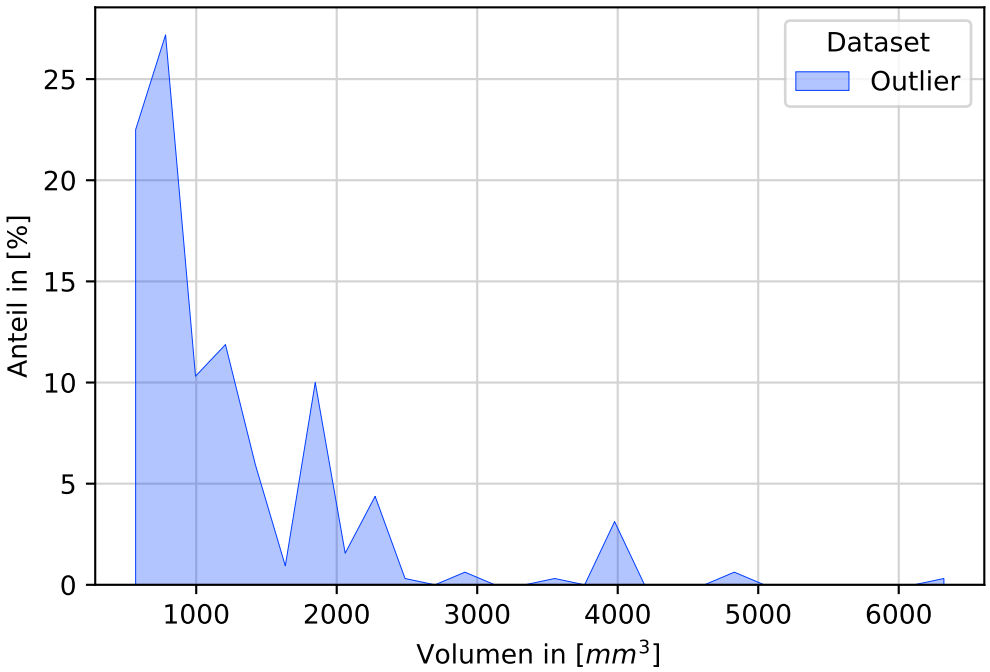


Abbildung 4.18: Verteilung der Volumen (Input 2) mit dem höchsten relativen Fehler

- VSBs mit einem Volumen größer 4000mm^3 sind extrem unterrepräsentiert und führen über eine schlechte Vorhersage zu großen relativen wie absoluten Fehlern.
- Dies gilt ebenfalls für VSBs mit Eigenfrequqnzen größer 40000 Hz
- 20 bis 35% der Ausreißer besitzen Schnittmengen in Volumen und Eigenfrequenzen der besten Vorhersagen und besitzen daher geometrische Merkmale, welche im Datensatz unterrepräsentiert sind und eine Generalisierung des DNN-Modells verhindern, sowie zu größeren Fehlern in den Vorhersagen führen.
- Die Identifikation dieser geometrischen Eigenschaften und Merkmale, welche die Ausreißer kennzeichnen, verlangt eine umfangreiche Untersuchung und die Entwicklung eines ML basierten Clustering-Verfahrens. Dies bietet einen interessanten Ansatzpunkt für zukünftige Arbeiten zur Optimierung der Vorhersagen von dreidimensionalen CNNs.

4.3.4 Visualisierung der Feature-Maps

Für einen Einblick in die Funktionsweise der Layer des CNN werden die trainierten Gewichte aus dem im letzte Abschnitt durch Fine-Tuning erzeugten besten DNN extrahiert.

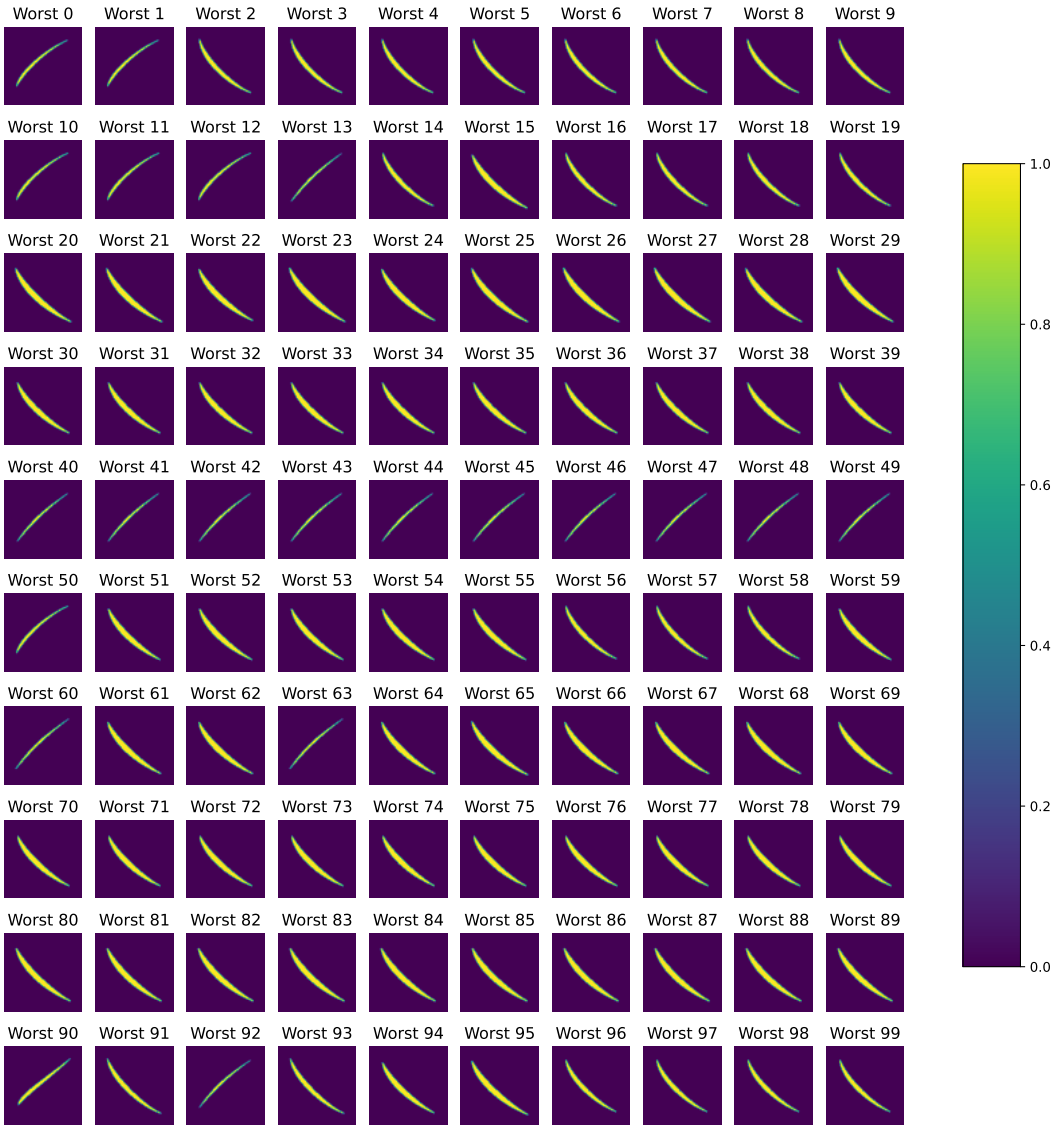


Abbildung 4.19: Abbildung von Schnittebenen der schlechtesten 100 VSBs Voxelgitter (Input 2) mit dem höchsten relativen Fehler

Die Gewichte der dreidimensionalen Convolution-Layer, repräsentieren wie in Abschnitt 2.5.7 erklärt, die Kernel der einzelnen Channels. Nach laden des zuvor gespeicherten besten DNN `best_model = tf.keras.models.load_model()` werden mit dem im Anhang 6 beigefügten Code die einzelnen Layer und die darin enthaltenen Gewichte, Filter und Aktivierungsfunktionen aus dem bereits trainierten DNN extrahiert. Anschließend werden die Kernel der Convolutional Layer und damit die gelernten Merkmale und Outputs der Pooling-Layer in den dazugehörigen Listen `conv_output` und `pooling_output` hinterlegt. Für die Erzeugung der Feature-Maps, werden die in der Liste `conv_output` enthaltene Kernel der einzelnen Layer und Channel zusammen mit dem DNN-Input der Voxelgit-

ter `best_model.inputs[0]` einem neuen Modell `feature_map_model` zugewiesen. Mit der Funktion `feature_map_model.predict(input)` wird für mit dem Abb. 4.20 zugrundeliegende dreidimensionale Voxelgitter `input` die Feature-Maps der einzelnen Convolutional-Layer berechnet. In Abb. 4.20 und den daraus erzeugten Feature-Maps ist die 64. Voxel-Ebene der ersten Dimension der $128 \times 128 \times 128$ großen Voxelgitter dargestellt. Entsprechen den Darstellungen der besten Abb. 4.15 und schlechtesten VSB-Geometrien Abb. 4.19 zeigt diese Voxelebene das mittlere VSB-Profil. Daraus ergeben sich für die in Abb. 3.12

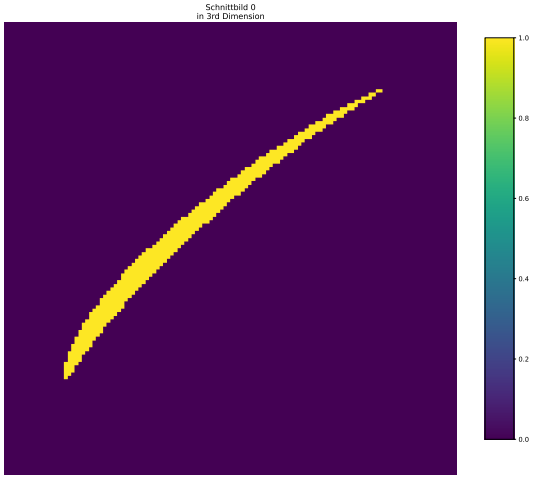


Abbildung 4.20: Input Feature-Maps: 2D Ausschnitt des 3D Voxelgitters

dargestellte Modellstruktur von dem DNN aus Trial 236, ausgehend von dem ersten Convolutional Layer `conv3d`, entsprechend der Anzahl an Channel pro Layer vier Feature-Maps in Abb. 4.21, 16 in Abb. 4.22, 64 in Abb. 4.23 und am Ende 128 skalaren High-Level Feature, die wiederum aus den Outputs der vorangegangenen Layer extrahiert werden und repräsentieren die in den Trainings-Daten verborgenen signifikanten Merkmale, welche die unterschiedlichen VSB Geometrien verbinden. Diese komprimierte Repräsentation der in den Daten versteckten Informationen wird *Latent-Space* genannt. Die in 4.21 dargestellten Feature-Maps, zeigen die ersten vier aus Abb. 4.20 extrahierten Merkmale. Die Merkmale Nr.2 und Nr.3 zeigen eine typische Kantendetektion der Umrisse des VSB-Profils. Ein Merkmal wie eine Kante oder der Umriss eines Objekts wird sehr häufig in den ersten Layern eines CNN erkannt und stellt ein sogenanntes Low-Level-Feature dar. Aus diesen Low-Level-Features werden in den darauf folgenden Layern sogenannte High-Level-Feature extrahiert. Den ersten beiden Merkmalen beziehungsweise Feature in Abb. 4.21 kann die Detektion der Kante der Druckseite, beziehungsweise der Kante der Saugseite zugeschrieben werden. Aus diesen Low-Level-Features der ersten Convolutional-Layer werden dann in Anlehnung an Abb. 2.11 weitere abstrakte Merkmale (*Additional Abstract Feature*) extrahiert. Pro Convolutional-Layer und zunehmender Anzahl an Channel werden die erkannten Merkmale weiter abstrahiert und auf 128 signifikante skalare Merkmale komprimiert. Aus

diesem Grund wird die menschliche Interpretation der dargestellten Merkmale mit der pro Layer zunehmenden Abstraktion immer schwieriger. Jedoch können beispielsweise in den sechzehn weiteren Feature-Maps, des zweiten Convolution-Layer der Fokus des CNN auf unterschiedliche Bereiche des VSB-Profils in Abb. 4.22 erkannt werden. Darin zeigen die Merkmale Nr.0 und Nr.15 den besonderen Fokus des CNN-Teils des DNN auf die Leading Edge (LE) jenes VSB. In den Merkmalen Nr.2, 4, 13 und 14 wird die Form oder Krümmung der Druckseite und Saugseite detektiert. Diese Feature-Maps werden in den darauf folgenden Convolutional-Layer wie in den Abb. 4.23 und 4.24 durch das Max-Pooling in der Größe immer weiter reduziert und nehmen in ihrer Abstraktion mit Fokus auf explizite Merkmale weiter zu, bis die Kerninformationen im Latent-Space konzentriert sind und durch die darauf folgenden Dense-Layer auf die zehn gesuchten Eigenfrequenzen abgebildet werden können.

4.3.5 Klassifikationsfähigkeit

In diesem Abschnitt wird die zuvor gezeigte Erkenntnis über die in den Gewichten und dem Latent-Space gespeicherten Merkmale des Datensatzes ausgenutzt, um mittels *Transfer-Learnings* die Fähigkeit des DNN nachzuweisen, zwischen Rotoren und Statoren zu unterscheiden. Transfer-Learning bezeichnet dabei eine ML-Methode ähnlich dem Fine-Tuning ein bereits trainiertes Modell zu verwenden, jedoch mit dem Unterschied den Zustand der bereits trainierten Gewichte einzufrieren und die Output-Layer durch ein dem neuen Problem angepasstes Mapping zu ersetzten. Wie bereits in Abb. 4.15 gezeigt, besitzen die Rotoren und Statoren eine um 90 Grad verdreht Orientierung. Damit nachgewiesen werden kann, dass diese Orientierung von dem DNN aus Trial 236 als Merkmal erkannt wird, müssen die Ground-Truths der zehn Eigenfrequenzen durch binäre Klassen oder Label ersetzt werden. Einfach ausgedrückt werden alle Sample, welche ein Rotorblatt repräsentieren durch die Zahl null und diejenigen, welche einen Stator repräsentieren durch die Zahl eins eindeutig gekennzeichnet.

Dies führt auf ein Klassifikationproblem welches mit der in Abschnitt 2.5.4 vorgestellten Logistic-Regression gelöst wird. Da diese sich alleine durch die Sigmoid-Aktivierungsfunktion, sowie die Binary-Cross-Entropy Loss-Funktion von der Linear-Regression unterscheidet, kann das bereits trainierte Modell sehr einfach mittels der im Anhang 7 beigefügten Funktion `logistic_model()` und der beschriebenen Transfer-Learning-Methode in ein Logistic-Regression-Modell überführt werden. Entscheidend für das Transfer-Learning ist nach dem Laden des bereits trainierten Modells mit `reconstructed_model=tf.keras.models.load_model()`, das Einfrieren des Zustands der bereits trainierten Gewichte mit `reconstructed_model.trainable = False`. Das letzte Dense-Layer `'dense_4'` in Abb. 3.12, welches das Mapping auf die zehn Eigenfrequenzen ermöglicht wird durch die Befehle

```
1  x = reconstructed_model.layers[-2].output
```

```
2   x = tf.keras.layers.Dense(1,activation='sigmoid',name='dense_sigmoid')(x)
3   model = tf.keras.Model(inputs = reconstructed_model.inputs,outputs=x)
```

ersetzt, indem der Layer-Output des vorangegangenen Layers `x` in das neue Layer `'dense_sigmoid'` eingefügt wird und schließlich mit dem Input-Layer `'voxel_input'` auf das neue Logistic-Regression Model `model=logistic_model()` führt. Die neue Modellstruktur nach dem Transfer mit den unveränderten Convolutional-Layer ist in Abb. 4.25 dargestellt.

Für die Evaluation des Modells wird der R2-Score `r2_score` durch die in Abschnitt 2.5.5 beschrieben Metrik des F1-Scores ersetzt. Mit den maximalen Anteilen der binär klassifizierten 35.7% Trainings-Daten und 6.3% Validierungs-Daten des gesamten Datensatzes werden die Weights in der letzten Schicht des aus dem Transfer-Learning entstanden Logistic-Regression-Modells in 200 Epochen aktualisiert. Der Trainingsverlauf, ist den jeweiligen Teilabbildungen von Abb. 4.26 durch die Verläufe des Trainings-MSE und F1-Score in Blau sowie der Validierungs-MSE und F1-Score in Orange dargestellt.

In Abb. 4.26 sind die nahezu perfekten Ergebnisse in Epoche 200

- (Training) F1-Score: 0.9986
- Validation F1-Score: 1.000

enthalten. Die Evaluation über den Anteil der binär klassifizierten 58% Test-Daten ergibt einen F1-Score von 0.9994 und eine binäre Genauigkeit von 99.94%.

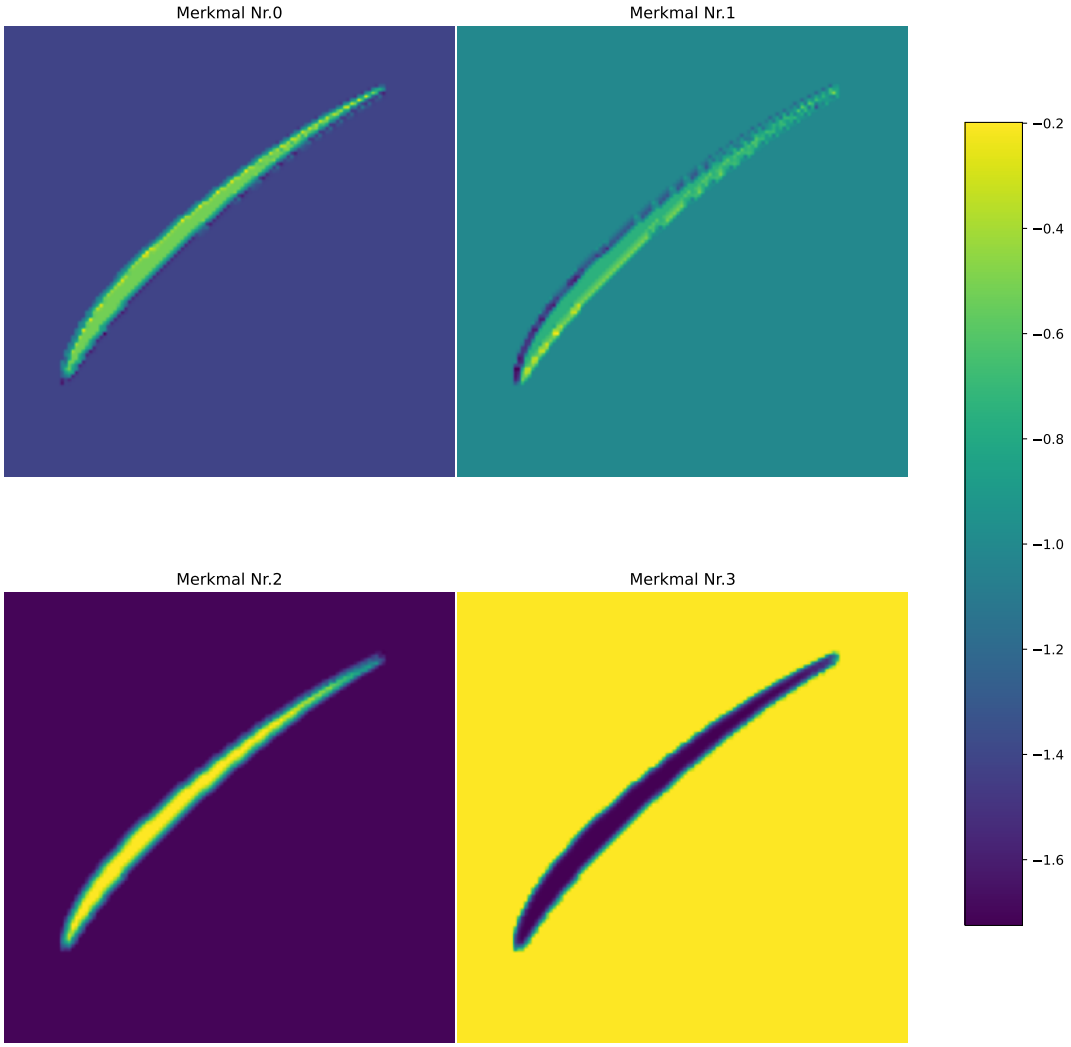


Abbildung 4.21: Feature-Maps der ersten Convolutional-Layer

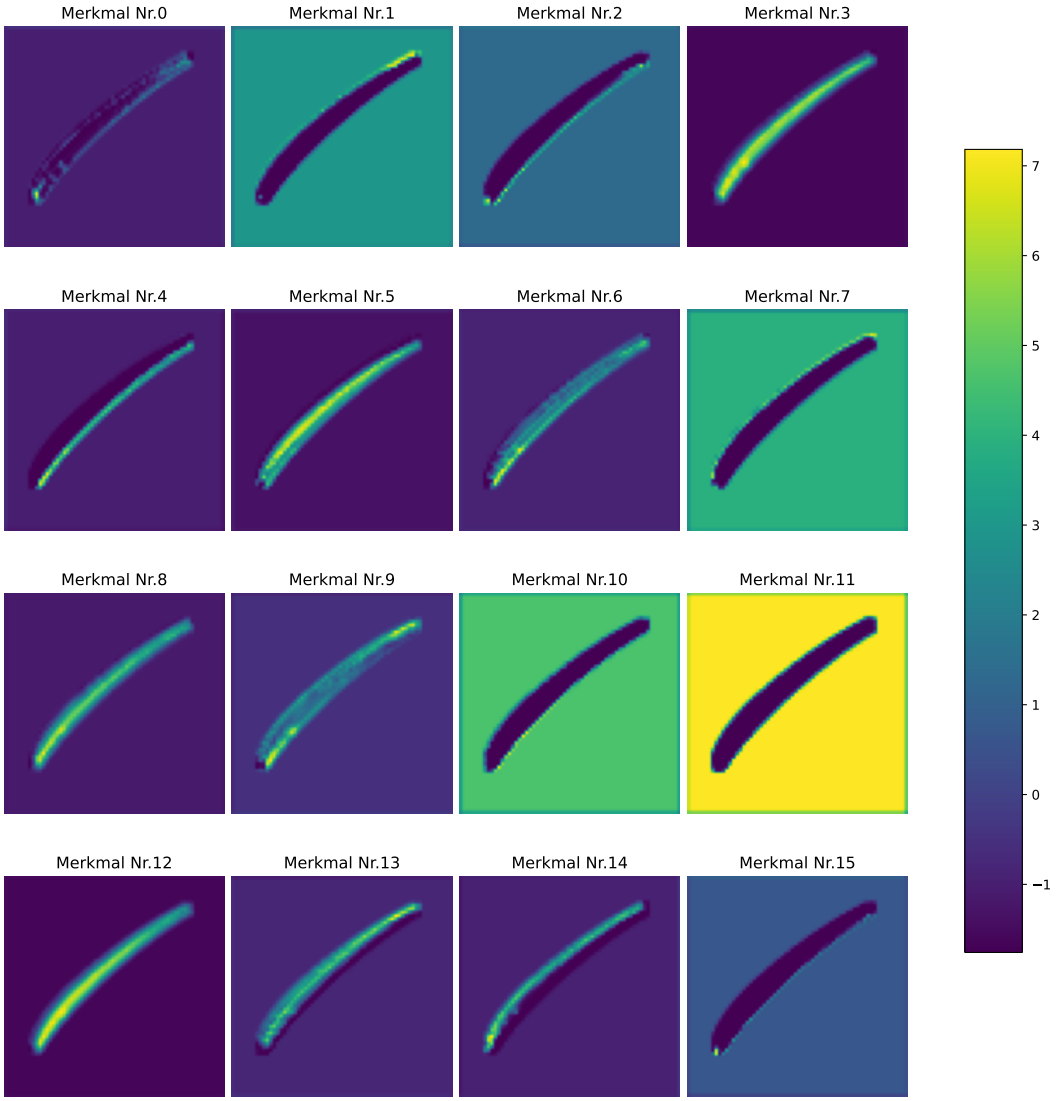


Abbildung 4.22: Feature-Maps der zweiten Convolutional-Layer

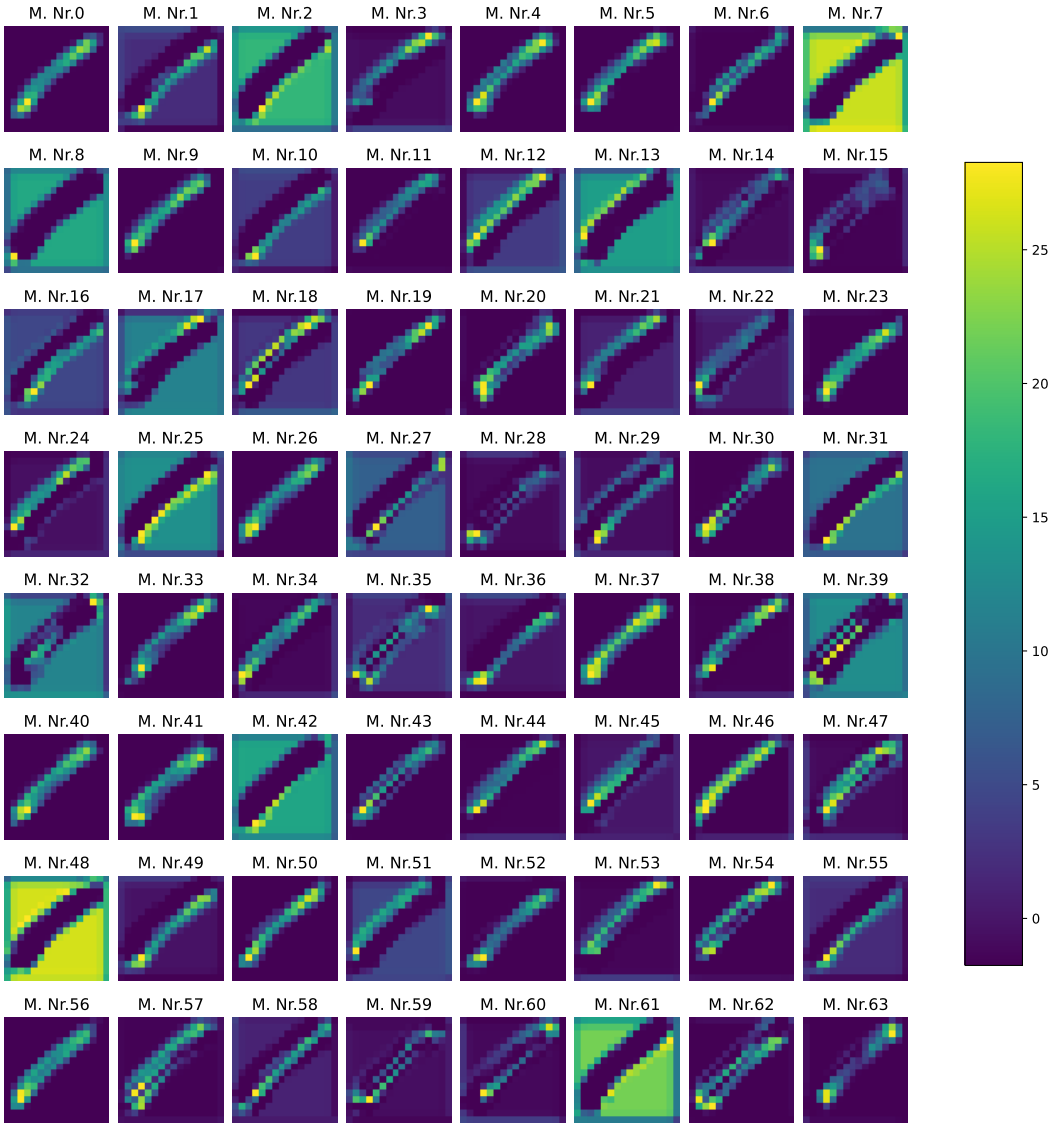


Abbildung 4.23: Feature-Maps der dritten Convolution-Layer

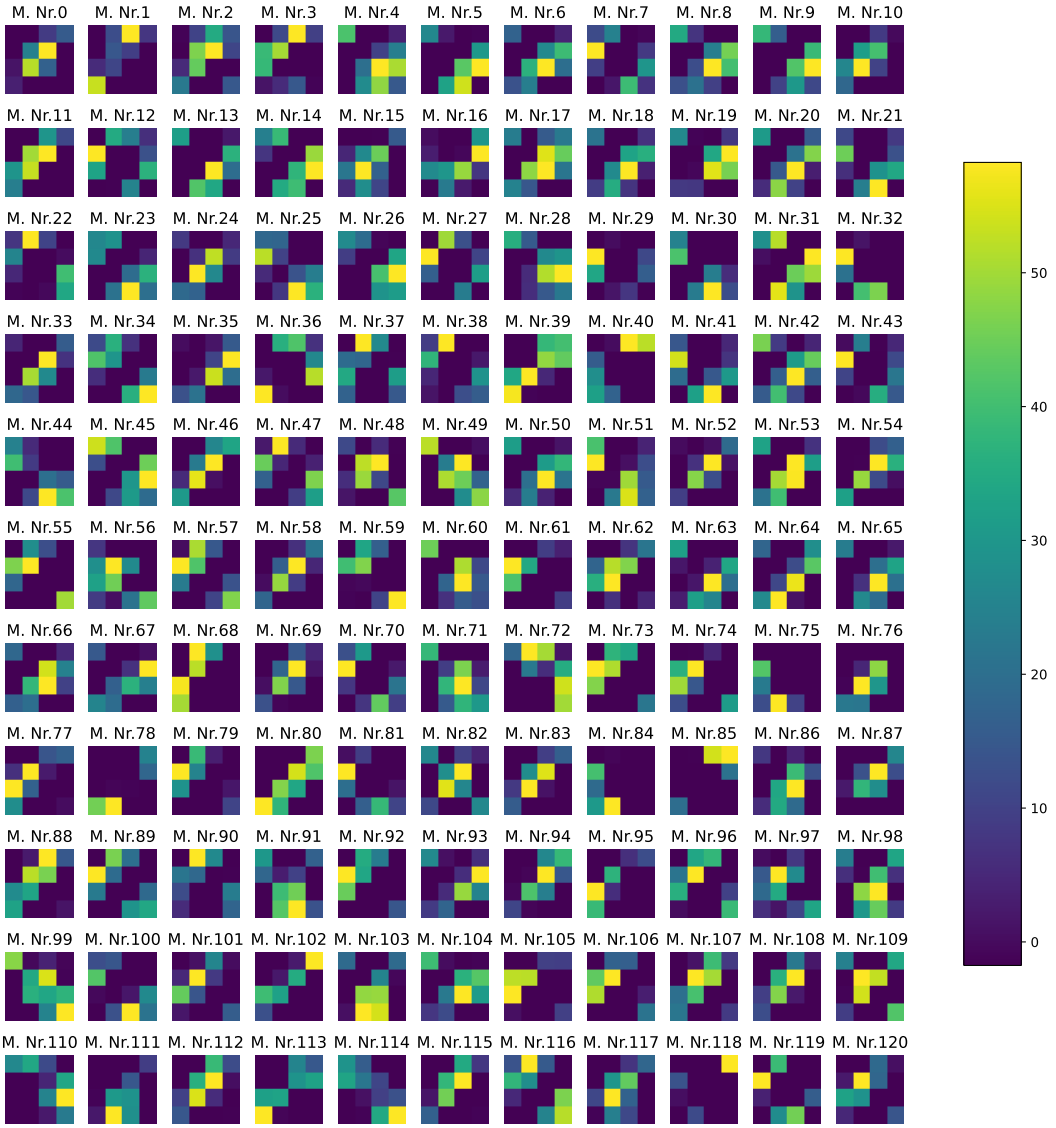


Abbildung 4.24: Feature-Maps der vierten Convolutioal-Layer

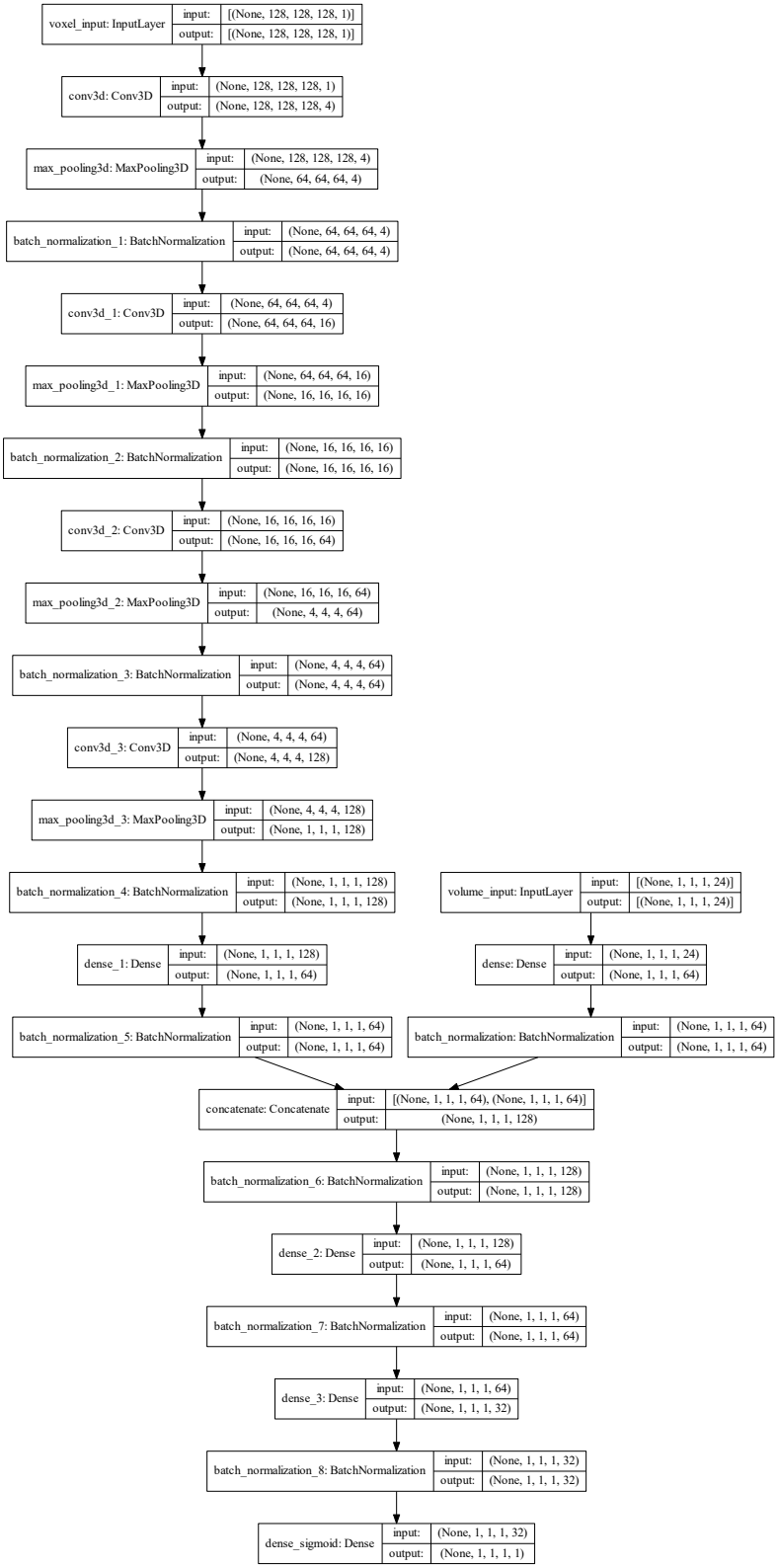


Abbildung 4.25: Die Modellarchitektur des aus dem DNN-Modell Trials 236 mit Transfer-Learning erzeugten Logistic-Regression-Modell

a) b)

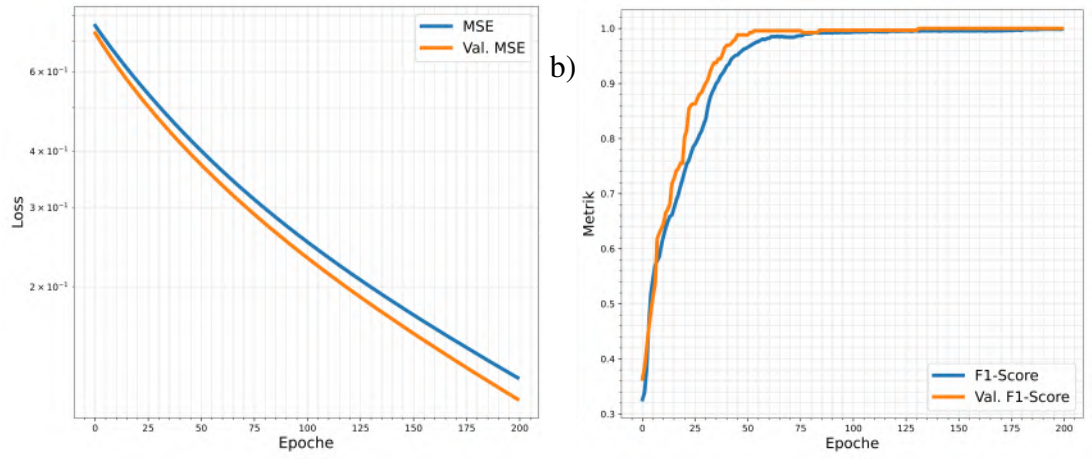


Abbildung 4.26: *a*) zeigt den Loss und *b*) den F1-Score über die Epochen der Logistic-Regression

5 Zusammenfassung

Die in dieser Arbeit durchgeführte Machbarkeitsstudie für die Vorhersage der ersten zehn signifikanten Eigenfrequenzen von Vierdichterschaufelblättern (VSBs) einer im Vorentwurf ausgelegten Variation des Triebwerk-Typs Trent 1000, wurde durch die Verbindung eines dreidimensionalen CNN mit einem vollständig verknüpften ANN in einem übergeordneten DNN realisiert. Alle in dieser Arbeit durchgeführten Schritte wie die Aufbereitung der Daten (Preprocessing), die Erstellung der ML-Model-Struktur, das Training und die anschließende Validierung und Visualisierung der Ergebnisse ist mit der High-Level Programmiersprache Python [28] realisiert.

Das in dieser Arbeit in Abschnitt 3.2 durchgeführte Preprocessing der 78560 VSBs umfassenden Datenbasis ermöglicht die Verarbeitung der darin enthaltenen parametrisierter Vierdichterschaufelgeometrien (Inputs) und eine verlässliche Vorhersage der Eigenfrequenzen (Outputs) durch das erstellte DNN. Diesbezüglich werden die polygonalen Oberflächenmodelle aus den Kontrollpunkten der parametrisierter Vierdichterschaufelgeometrien mit den in den theoretischen Grundlagen behandelten NURBS und dem Python-Paket `geomdl` [4] rekonstruiert. Die anschließende Voxelization mit dem hocheffiziente Executable (.exe) Binvox [20] der polygonalen Oberflächenmodelle in dreidimensionale binäre Voxelgitter ermöglicht das einlesen und verarbeiten der geometrischen Zusammenhänge durch das DNN. Bei der Voxelization der Oberflächenmodelle entsteht ein Diskretisierungsfehler. Mit einer Normalisierung der geometrischen Gestalt der VSBs kann der Diskretisierungsfehler reduziert und gleichzeitig die Auflösung der Voxelization auf 128 Voxeln pro Dimension beschränkt werden. Der Nachteil der geometrischen Normalisierung ist, dass der Zusammenhang zwischen den Eigenfrequenzen und dem Volumen jedes VSB verloren geht. Aus diesem Verlust resultiert, dass die Information über das ursprüngliche Volumen vor der Normalisierung als zusätzlicher Input in das DNN eingespeist werden muss. Die inhomogen Verteilungen der Eigenfrequenzen und Volumen werden mittels einer logarithmischen Normalisierung optimiert und darin enthaltene Ausreißer minimiert. Die MinMax-Skalierung der normalisierten Eigenfrequenzen und Volumen auf den Bereich (0,1) wird als präventiv Maßnahme angewandt, um eine ungleiche Gewichtung und große Loss-Werte zu verhindern.

Über ein eigens geschriebenes Binary-Encoding Verfahren werden die skalaren Volumen der einzelnen VSBs in eine binäre Vektordarstellung überführt. Die Batch-Normalisierungen vor der Verknüpfung der dreidimensionalen CNN und dem vollständig verknüpften ANN sorgt für die gleichwertige Gewichtung der aus den 3D Voxelgittern und den binären Volumenvektoren extrahierten Informationen innerhalb des DNN.

Für das Training des DNN wurde Tensorflow als Backend verwendet. Mittels des Mini-Bath-Gradient-Descent-Verfahrens wird eine effiziente Optimierung der Gewichte über eine GPU ermöglicht. Mit dem Open-Source Framework Optuna [1] wird in einer Studie die Optimierung der Hyperparmeter des DNN durchgeführt. Aus den 250 durchgeführten Trials der Hyperparamter-Optimierung resultiert die bestmögliche DNN Konfiguration. Diese Konfiguration besitzt vor jedem Layer-Input eine Batch-Normalisierung, welche in Kombination mit der SelU-Aktivierungsfunktion eine automatische Normalisierung der Gewichte in den Layern bewirkt. Die geringste mögliche ungerade Kernel-Größe von $3 \times 3 \times 3$ bewirken eine feinere Detektion der in den Inputs hinterlegten Merkmalen. Die größte mögliche Anzahl an Sample pro Bath verbessert die Generalisierung des Modells und stabilisiert der Trainingsverlauf. Für die nach dem Preprocessing vorliegenden binären Input-Daten wird das Max-Pooling-Verfahren, dem Average-Pooling vorgezogen. Als initiale Lernrate für den Adam-Optimierungsalgorithmus empfiehlt die Studie den Wert 0.02087. Über das Training innerhalb der Hyperprameter-Optimierung erzielt das DNN aus Trials 236 einen maximalen Validierungs-R2-Score von 0.9859.

Das anschließend in Abschnitt 4.3.1 durchgeführte Fine-Tuning mit den beschrieben besten Parametern und dem maximalen durch die Hardware zugelassenen Anteil von 35.7% an Trainings-Daten und 6.3% Vaildierungs-Daten in Bezug auf den gesamten Datensatz wird ein Validierungs-R2-Score von 0.9914 erreicht. Über eine ausführliche Auswertung der relativen und absoluten Fehler der Vorhersagen des DNN in Bezug zu den Ground-Truths können für 82.14% der Test-Daten verlässliche Vorhersagen generiert werden. 50% der Vorhersagen besitzen einen geringeren relativen Fehler als 5.53% in den einzelnen Eigenfrequenzen. Eine umfangreiche Analyse der Ausreißer mit großen relativen Fehlern in den Vorhersagen ergibt, dass Verdichterschaufelblätter mit einem Volumen größer als $4000 mm^3$ und Eigenfrequenzen größer 40000 Hz im zugrunde liegenden Datensatz extrem unterrepräsentiert sind. Über die Schnittmengen der Volumen- und Eigenfrequenzverteilungen der besten Vorhersagen und der Ausreißer werden die unterrepräsentierten geometrische Merkmale als weiter Ursache identifiziert. Die Identifikation dieser geometrischen Eigenschaften und Merkmale, welche die Ausreißer kennzeichnen und eine Generalisierung des DNN verhindern, verlangt eine umfangreiche Untersuchung und die Entwicklung eines ML basierten Clustering-Verfahrens. Dies bietet einen interessanten Ansatzpunkt für zukünftige Arbeiten zur Optimierung der Vorhersagen von dreidimensionalen CNNs. Die guten bis sehr guten Vorhersagen des DNN trotz Underfitting, zeigt das Potenzial der Vorhersagen des in dieser Arbeit erstellten DNN bei einem erneuten Training mit mehr Rechenleistung und einem neuen Datensatz mit annähernd gleichwertig repräsentierten Merkmalen.

Zusätzlich wurde in dieser Arbeit in Abschnitt 4.3.5 mittels Transfer-Learning des DNN in ein Logistic-Regression-Modell die Fähigkeit des Modells nachgewiesen, dreidimensionale Merkmale in den Verdichterschaufelblättern zu erkennen.

Literatur

[1] Takuya Akiba u. a. „Optuna: A Next-generation Hyperparameter Optimization Framework“. In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[2] James Bergstra und Yoshua Bengio. „Random search for hyper-parameter optimization.“ In: *Journal of machine learning research* 13.2 (2012).

[3] James Bergstra u. a. „Algorithms for Hyper-Parameter Optimization“. In: *Advances in Neural Information Processing Systems*. Hrsg. von J. Shawe-Taylor u. a. Bd. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[4] Onur Rauf Bingol und Adarsh Krishnamurthy. „NURBS-Python: An open-source object-oriented NURBS modeling framework in Python“. In: *SoftwareX* 9 (2019), S. 85–94.

[5] Thomas Cleff, Hrsg. *Deskriptive Statistik und Explorative Datenanalyse: eine computergestützte Einführung mit Excel, SPSS und STATA*. Deutsch. 3., überarb. und erw. Aufl. Lehrbuch. Wiesbaden: Springer Gabler, 2015, XVI, 264 Seiten. ISBN: 9783834947482 (Online-Ausgabe).

[6] Drew Conway und John Myles White, Hrsg. *Machine learning for hackers: [case studies and algorithms to get you started]*. Englisch. 1. ed. Beijing; Köln [u.a.]: O'Reilly, 2012, XIII, 303 Seiten. ISBN: 1-449-30371-4.

[7] T.N. Wiesel D.H. Hubel. „Receptive fields and functional architecture of monkey striate cortex“. Englisch. In: *Journal of Physiology* 195 (1 1968), S. 251–243. DOI: https://doi.org/10.1113/jphysiol.1968.sp19681951toc.

[8] Carl [Verfasser/in] De Boor, Hrsg. *A practical guide to splines*. Englisch. Applied mathematical sciences ; 27. New York ; Berlin ; Heidelberg [u.a.]: Springer, 1978, XXIV, 392 S. ISBN: 0387903569.

[9] Jia Deng u. a. „Imagenet: A large-scale hierarchical image database“. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, S. 248–255.

[10] Marcelo Epstein. *Partial differential equations: mathematical techniques for engineers*. Englisch. Mathematical engineering. Institut für Thermodynamik der Luft- und Raumfahrt. Cham, Switzerland: Springer, 2017, xiii, 255 Seiten. ISBN: 9783319552125 (Online-Ausgabe).

[11] Roland W. Freund und Ronald H. W. Hoppe, Hrsg. *Numerische Mathematik : eine Einführung - unter Berücksichtigung von Vorlesungen von F. L. Bauer. - 1*. Deutsch. 10., neu bearb. Aufl. Bd. 1. Numerische Mathematik ; 1. Institut für Angewandte Analysis und Numerische Simulation, Lehrstuhl Numerische Mathematik für Höchstleistungsrechner. Berlin ; Heidelberg [u.a.]: Springer, 2007, XI, 410 Seiten. ISBN: 9783540453901 (Online-Ausgabe).

[12] Kunihiko Fukushima. „Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position“. Englisch. In: *Biological Cybernetics* 36 (1980), S. 193–202. DOI: 10.1007/BF00344251.

[13] Ian Goodfellow u. a., Hrsg. *Deep learning*. Englisch. Adaptive computation and machine learning. Cambridge, Massachusetts ; London, England: The MIT Press, 2016, xxii, 775 Seiten. ISBN: 9780262035613.

[14] Sergey Ioffe und Christian Szegedy. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *International conference on machine learning*. PMLR. 2015, S. 448–456.

[15] Häßy Jannik u. a. „Hybrid Surrogate-Based Rubber Engine Model for Aircraft Multidisciplinary Design Optimization“. In: *Physics Informed Machine Learning, MetaModeling, and Reduced Order Modeling II* (2020). DOI: 10.2514/6.2020-3186. URL: https://doi.org/10.2514/6.2020-3186.

[16] Malte Krack, Hrsg. *Strukturdynamik*. Deutsch. 2018.

[17] Fengjun Lv u. a. „Failure analysis of components in compressor vane“. In: *Engineering Failure Analysis* 16.5 (2009), S. 1703–1710. ISSN: 1350-6307. DOI: https://doi.org/10.1016/j.engfailanal.2008.12.007. URL: https://www.sciencedirect.com/science/article/pii/S1350630708002860.

[18] Martin Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[19] Warren S. McCulloch. „A logical calculus of the ideas immanent in nervous activity“. Englisch. In: *The bulletin of mathematical biophysics* 5 (1943), S. 115–133. DOI: 10.1007/BF02478259.

[20] Patrick Min. *binvox*. 2004 - 2019. URL: http://www.patrickmin.com/binvox.

[21] Kevin P. Murphy, Hrsg. *Machine learning: a probabilistic perspective*. Englisch. Adaptive computation and machine learning series. Cambridge, Mass. [u.a.]: MIT Press, 2012, XXIX, 1071 Seiten. ISBN: 0262018020.

[22] Hans Günther [Verfasser/in] Natke, Hrsg. *Einführung in Theorie und Praxis der Zeitreihen- und Modalanalyse: Identifikation schwingungsfähiger elastomechanischer Systeme*. Deutsch. 3., überarb. Aufl. Grundlagen und Fortschritte der Ingenieurwissenschaften. Institut für Technische und Numerische Mechanik. Braunschweig ; Wiesbaden: Vieweg, 1992, XXXVIII, 546 S. ISBN: 9783322942678.

[23] Fakir S. Nooruddin und Greg Turk. „Simplification and Repair of Polygonal Models Using Volumetric Techniques“. In: *IEEE Transactions on Visualization and Computer Graphics* 9.2 (2003), S. 191–205.

[24] F. Pedregosa u. a. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830.

[25] Markus Schnoes, Christian Voß und Eberhard Nicke. „Design optimization of a multi-stage axial compressor using throughflow and a database of optimal airfoils“. In: *Journal of the Global Power and Propulsion Society* 2 (2018), S. 516–528. DOI: 10.22261/JGPPS.W5N91I. URL: https://doi.org/10.22261/JGPPS.W5N91I.

[26] William J. Schroeder und Kenneth M. Martin. „The Visualization Toolkit“. In: *The Visualization Handbook*. 2005.

[27] Reitenbach Stanislaus u. a. „Collaborative Aircraft Engine Preliminary Design using a Virtual Engine Platform, Part A: Architecture and Methodology“. In: *The AIAA 2020-0867*. 2020. DOI: 10.2514/6.2020-0867. URL: https://doi.org/10.2514/6.2020-0867.

[28] Guido Van Rossum und Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[29] Marcus [Verfasser/in] Wagner, Hrsg. *Lineare und nichtlineare FEM: Eine Einführung mit Anwendungen in der Umformsimulation mit LS-DYNA®*. Deutsch. 1 Online-Ressource(XI, 374 Seiten 1156 Abb.) Wiesbaden, 2022.

Anhang

1 Beispiel Inhalt der bereitgestellten .dat-Dateien im ascii-Format

```
VARIABLES="CoordinateX" "CoordinateY" "CoordinateZ"
ZONE T = "NURB",              I= 100, J= 1,  K= 30,  F = POINT
3.299663966956e-02  -8.874085990977e-03  1.815284214089e-01
3.297138666357e-02  -8.863191092104e-03  1.815269792895e-01
3.292596021371e-02  -8.843735971139e-03  1.815243678075e-01
3.285675304594e-02  -8.813835742869e-03  1.815204007880e-01
3.277294932413e-02  -8.777446087513e-03  1.815155996214e-01
3.265030820054e-02  -8.723770889386e-03  1.815085829802e-01
3.246980036431e-02  -8.643902966497e-03  1.814982724212e-01
3.220518782348e-02  -8.524976785939e-03  1.814831948633e-01
3.181706894794e-02  -8.346737133455e-03  1.814611528079e-01
3.124916850225e-02  -8.078148649747e-03  1.814290414121e-01
3.041958317952e-02  -7.670288438335e-03  1.813822809305e-01
```

2 Variables DNN Modell

```
def dnn_model(in1_size=128,in2_size=24,
  n_filters = [4,16,64,128],
  n_pool_size = [2,4,4,4],
  ks = 3,
  af='relu',
  bn=False,
  po='max'):

  assert len(n_filters) == len(n_pool_size) , print('Amount of filters and
      ↪ pooling kernels must be equal!')
  assert np.product(n_pool_size) == in1_size , print('Sizes of pooling kernels
      ↪ must add up to in1_size!')
```

```
11  assert n_filters[-1] ==  in1_size , print('The last filter sizes must be
        ↪ in1_size!')
12
13  def pooling(pool_size,x,po):
14    if po == 'average':
15      return tf.keras.layers.AveragePooling3D((pool_size))(x)
16    else:
17      return tf.keras.layers.MaxPool3D((pool_size))(x)
18
19  def batch_norm(bn,x):
20    if bn:
21      return tf.keras.layers.BatchNormalization()(x)
22    else:
23      return x
24
25  # ANN for volumetric data
26  input_layer2 = tf.keras.Input((1,1,1,in2_size),name='volume_input')
27  y = input_layer2
28  y = tf.keras.layers.Dense(64, activation = af)(y)
29  y = batch_norm(bn, y)
30
31  # CNN for geometrical data
32  input_layer = tf.keras.Input((in1_size,in1_size,in1_size,1),name = 'voxel_input
        ↪ ')
33  x = input_layer
34
35  for filter , pool_size in zip(n_filters,n_pool_size):
36  x = tf.keras.layers.Conv3D(filter, kernel_size=(ks, ks, ks), padding='SAME',
        ↪ activation = af)(x)
37  x = pooling(pool_size, x,ptype=po)
38  x = batch_norm(bn, x)
39
40  # Mapping of CNN features
41  x = tf.keras.layers.Dense(64, activation = af)(x)
42  x = tf.keras.layers.BatchNormalization()(x)
43
44  # Cocatenation of CNN and ANN outputs
45  merge = tf.keras.layers.Concatenate()([x,y])
46  x = batch_norm(bn, merge)
47  x = tf.keras.layers.Dense(64, activation = af)(x)
48  x = batch_norm(bn, x)
49  x = tf.keras.layers.Dense(32, activation = af)(x)
50  x = batch_norm(bn, x)
51
52  outputs = tf.keras.layers.Dense(10, activation = 'relu')(x)
```

```
53
54  model = tf.keras.Model(inputs = [input_layer,input_layer2],outputs=outputs)
55
56  return model
```

3 Binary Encoding Funktion

```
1   def binary_encoding(self,dec_num,bin_length):
2     num = round(dec_num)
3     res = bin(num).lstrip("0b").replace('',',').split(',')
4     binary_num = res[1:-1]
5     binary_num = [int(b) for b in binary_num]
6     add_zeros = bin_length - len(binary_num)
7     if add_zeros < 0:
8       return print('Binary representation is larger then the maximum
            ↪ representable binary number: ',2**bin_length)
9
10    zero_list = [0]*add_zeros
11    return zero_list + binary_num
```

4 Ausschnitt Optuna Objective Klasse mit Hyperparameter Vorschlägen

```
1
2   # pooling type
3   po = trial.suggest_categorical("po", ["average","max"])
4
5   # learning rate for the  adam optimizer, set to logarithmic float
6   lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)
7
8   # activations to be tried, only a single one for all hidden layers for now
9   af = trial.suggest_categorical("af", ["linear", "tanh", "selu", "relu"])
10
11  # batch_normalization
12  bn = trial.suggest_categorical("bn", [True, False])
13
14  ba = 4 # max value, batch size (power of two) for training
15  batch_size = 2**ba
16  while batch_size >  int(len(self.inputs_train[0])*(1-self.val_split)):
17  batch_size *= 2**(-1)
```

```
    ba -= 1

    bs_max = ba
    bs = trial.suggest_int("bs", 0, bs_max)

    # convolutional kernel size
    ks = trial.suggest_int("ks", 3, 7,step=2)


    model = hyper_cnn(in1_size=128,in2_size=24,
    n_filters = [4,16,64,128],
    n_pool_size = [2,4,4,4],
    ks = ks,
    af=af,
    bn=bn,
    po=po)

    optimizer = tf.keras.optimizers.Adam(learning_rate=lr)

    # actually compile the model
    model.compile(optimizer=optimizer,
      loss='MSE',
      metrics=[r2_score])
```

5 Multidimensionale Darstellung der Hyperparameter Optimierung

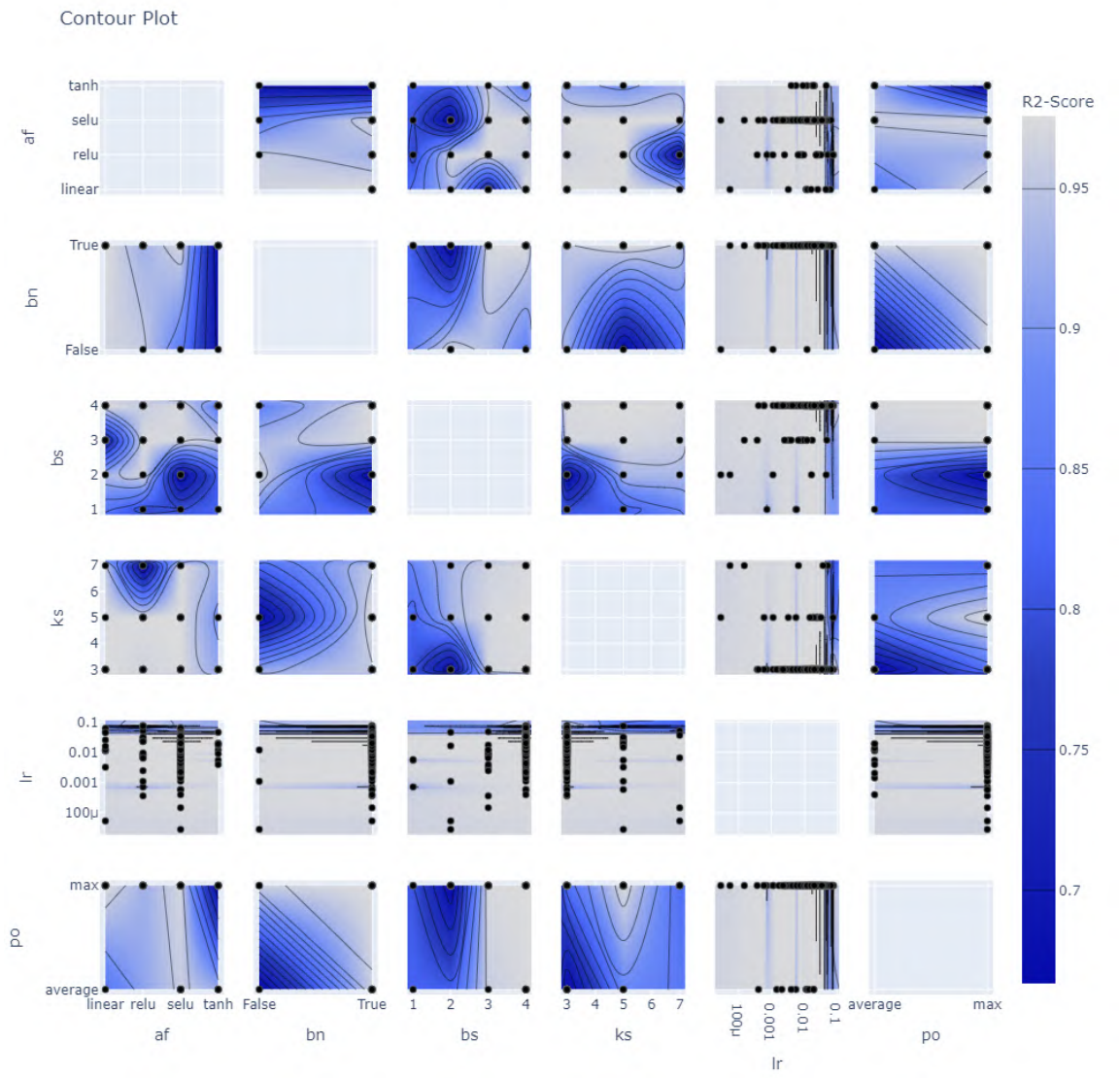


Abbildung 1: Darstellung von Schnittflächen der der multidimensionalen lokalen Minima, welche von den Wertebereichen der einzelnen Hyperparameter aufgespannt wird.

6 Auschnitt Python Skript für die Feature-Map-Extraktion

```
conv_output = []
pooling_output = []
dnn_inputs = []
conv_kernels = []
for layer in best_model.layers:
if relayer.name.find('conv')!=-1:
print('Layer: ', layer.name)
conv_output.append(layer.output)
conv_kernels.append(layer.kernel.numpy())
if layer.name.find('pooling')!=-1:
print('Layer: ', layer.name)
pooling_output.append(layer.output)
if layer.name.find('input')!=-1:
print('Layer: ', relayer.name)
dnn_inputs.append(layer)

feature_map_model = tf.keras.models.Model(inputs=best_model.inputs[0], outputs=
    ↪ conv_output)
pooling_model = tf.keras.models.Model(inputs=best_model.inputs[0], outputs=
    ↪ pooling_output)

feature_maps = feature_map_model.predict(input)
pooling_maps = pooling_model.predict(input)
```

7 Logistic-Regression Modell

```
def logistic_model(model_path='./best_model',custom_objects={"r2_score":
    ↪ r2_score},lr=0.001):

reconstructed_model = tf.keras.models.load_model(filepath=model_path,
    ↪ custom_objects=custom_objects,compile=False)
reconstructed_model.trainable = False
x = reconstructed_model.layers[-2].output
x = tf.keras.layers.Dense(1,activation='sigmoid',name='dense_sigmoid')(x)
model = tf.keras.Model(inputs = reconstructed_model.inputs,outputs=x)

# setup the optimizer with learning rate suggestion
optimizer = tf.keras.optimizers.Adam(learning_rate=float(lr))
model.compile(optimizer=optimizer,
loss='binary_crossentropy',
metrics=['binary_accuracy', f1_m])

```

```
15    return model
```