Institut für Flugzeugbau
der Universität Stuttgart

Masterarbeit

Symbolische und numerische Untersuchung dimensionshomogener Neuronaler Netze unter den Gesichtspunkten Lernfähigkeit und Genauigkeit

Bearbeiter

Christian Dreßler

Symbolische und numerische Untersuchung dimensionshomogener Neuronaler Netze unter den Gesichtspunkten Lernfähigkeit und Genauigkeit

Masterarbeit
von
Herr Christian Dreßler

durchgeführt am
Institut für Flugzeugbau
Universität Stuttgart

Stuttgart, im Dezember 2020

Masterarbeit

Symbolische und numerische Untersuchung dimensionshomogener Neuronaler Netze unter den Gesichtpunkten Lernfähigkeit und Genauigkeit

für BSc Christian Dressler

Aktuell stehen zahlreiche lernfähige Verfahren der Künstlichen Intelligenz wie Neuronale Netze (NN) unter den Stichworten *Deep Learning* und *Big Data* im Zentrum der Aufmerksamkeit. Um solche lernfähigen Verfahren wie Neuronale Netze für Ingenieuranwendungen sinnvoll einsetzbar zu machen, müssen allerdings einige wichtige Eigenschaften Neuronaler Netze noch genauer untersucht und insbesondere für eine spätere Zulassung in sicherheitsrelevanten oder gar sicherheitskritischen Einsatzszenarien besser verstanden sein. Hierfür müssen speziell das Verhalten Neuronaler Netze bzgl. Konvergenz, Lernfähigkeit, Robustheit und Genauigkeit noch genauer untersucht und verstanden werden.

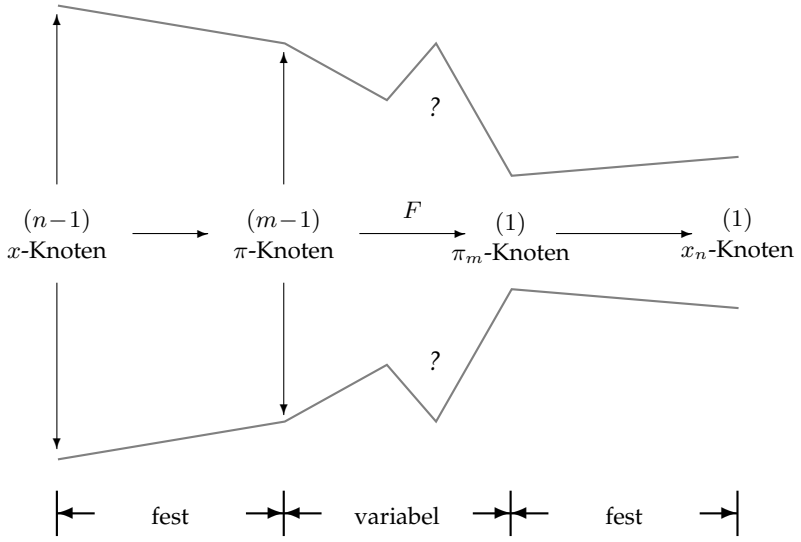

Abbildung 1: Abbildungssequenz in dimensionshomogenen Neuronalen Netzen

Dazu sind in der MSc-Thesis dimensionshomogene Neuronale Netze (DNN) nach Abb. 1 in Bezug auf die Lernfähigkeit aus einer vorgegebenen Menge aus Trainingsdaten, der Robustheit im Trainingsergebnis bezüglich der vorgegebenen Streuung und der Fehlerverteilung in den Daten sowie bezüglich einer möglichen Abschätzung der Ergebnisgenauigkeit genauer zu untersuchen.
Im Einzelnen sind die folgenden Aufgabenstellungen zu bearbeiten:

- Einarbeitung in die Theorie der DNN
  (Topologie, Aktivierungsfunktionen, Generalisierung, Fehlerdefinition)
- Generische Implementierung der theoretisch betrachteten Netzstrukturen in TensorFlow
  unter Berücksichtigung einer späteren Nutzung im Design Cockpit 43
- Erstellung, Auswertung und Darstellung der Untersuchungen von Lernfähigkeit, Ergebnisgüte und Robustheit für unterschiedliche Anwendungsfälle
- Dokumentation und Präsentation der Ergebnisse

Die MSc-Thesis wird im Kontext des Projekts „Künstliche Intelligenz Europäisch Zertifizieren 4.0“ (KIEZ40) im Luftfahrtforschungsprogramm VI.1 am Institut für Flugzeugbau in enger Kooperation mit dem Projektpartner Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH, Businesspark Leinfelden-Echterdingen, durchgeführt.

Ausgabedatum: 30.06.2020
Abgabedatum: 30.12.2020

Betreuer: MSc Moritz Neumaier, Institut für Flugzeugbau (IFB), Universität Stuttgart
Prüfer: Priv.-Doz. Dr.-Ing. Stephan Rudolph
Institut für Flugzeugbau (IFB), Universität Stuttgart

Kurzfassung

In dieser Masterarbeit wird untersucht, ob durch die Nutzung des Wissens um die Dimensionen von physikalischen Gleichungen die Leistungsfähigkeit eines neuronalen Netzes zur Approximation der Gleichung aus einzelnen Mess- bzw. Datenpunkten verbessert werden kann. Dazu wird die folgende Forschungsfrage gestellt: „Welchen Einfluss hat die Nutzung der Dimensionsinformationen auf die Lernfähigkeit und Genauigkeit eines neuronalen Netzes bei der Approximation physikalischer Gleichungen?“
Um die Forschungsfrage zu beantworten, wurden auf Basis eines eigens entwickelten Frameworks drei verschiedene quantitative Experimente durchgeführt. In diesen wurden die verschiedenen Nutzungsmöglichkeiten der Dimensionsinformationen miteinander, und mit herkömmlichen künstlichen neuronalen Netzen verglichen. Letztere wurden ohne die Nutzung der Dimensionsinformationen erstellt.
Die Auswertung der quantitativen Studie zeigt, dass die Nutzung der Dimensionsinformationen im Großteil der Fälle die Anzahl der benötigten Trainingsepochen bis zum Erreichen eines minimalen Trainingsfehlers reduziert. Gleichzeitig zeigten die Versuche auch, dass die Nutzung der zusätzlichen Informationen einen negativen Einfluss auf die mathematische Stabilität haben kann.

Abstract

The goal of this master's thesis is to determine whether using knowledge of the dimensions of physical equations can improve the performance of a neural network for approximating an equation from individual data points. Therefore, the following research question is posed, “What impact does the use of dimensional information have on the learning ability and accuracy of a neural network in approximating physical equations?“
In order to answer the research question, three different quantitative experiments have been conducted based on a specially developed design language, that compares different uses of the dimensional information with each other and with artificial neural networks that have been created without the use of the dimensional information.
The evaluation of the quantitative study shows that the use of the dimensional information in the majority of cases reduces the number of training epochs needed until a minimal training error is reached. The experiments also showed that the use of the additional information can have a negative impact on mathematical stability.

Inhaltsverzeichnis

Nomenklatur

Abkürzungen

Adam	Adaptive Moment Optimization
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Networks
GPU	Grafikprozessor
MAPE	Mean absolute Percentage Error
MSE	Mean Squared Error
PCA	Principal Component Analysis
Relu	rectified linear unit
RMSprop	Root Mean Square Propagation
Rprop	Resilient Propagation
TPU	Tensorprozessor

Lateinische Buchstaben

$[x_i]$	Einheit physikalische Größe
$\hat{y}_i$	vorhergesagter Wert
$\{x_i\}$	Wert physikalische Größe
b	Bias-Unit
C_j	dimensionslose Konstanten
c_{Rap}	Lichtgeschwindigkeit
D_{Feder}	Federkonstante
D_i	physikalische Basisdimension
D_i	physikalische Dimension
E_{Balken}	Elastizitätsmodul Balkenbiegeformel
E_{Fermi}	Energie Fermiverteilung
g_{Wurf}	Höhe senkrechter Wurf
h_r	Ausgabe Outputlayer
h_r	spezifisches Neuron
h_{Wurf}	Fallbeschleunigung senkrechter Wurf
I_{Balken}	Flächenträgheitsmoment Balkenbiegeformel
k	Anzahl hidden Layer
k_{Fermi}	Boltzmann-Konstante

L	Loss-Wert / Fehler-Wert
L_2	Penalty
L_{Balken}	Länge Balkenbiegeformel
L_{Dim}	Dimension Länge
M	Anzahl betrachtete Datenpunkte
m	Anzahl dimensionslose Potenzprodukte
M_{Dim}	Dimension Masse
m_{Feder}	Masse Federpendel
n	Anzahl der Variablen in der Relevanzliste
$N_{0,Zerfall}$	Anzahl Atomkerne Beginn
n_{Π}	Anzahl Fundamentalsysteme
$N_{Zerfall}$	Anzahl Atomkerne
P_{Balken}	Krafteinwirkung Balkenbiegeformel
$q_{Wärme}$	Wärme
r	Rang der Dimensionsmatrix
$r_{a,Wärme}$	Außenradius
$r_{i,Wärme}$	Innenradius
t	Zeitpunkt
T_a	Temperatur Außenwand
T_{Dim}	Dimension Zeit
t_{Feder}	Zeit Federpendel
T_{Fermi}	Temperatur Fermi
T_i	Temperatur Innenwand
t_{Wurf}	Zeit senkrechter Wurf
$t_{Zerfall}$	Zeit
u_{Balken}	Auslenkung Balkenbiegeformel
$v_{0,Wurf}$	Anfangsgeschwindigkeit senkrechter Wurf
v_{Rap}	Geschwindigkeit
W_{Fermi}	Besetzungswahrscheinlichkeit Fermiverteilung
w_i	Gewichte
x'	gestörter Datenpunkt
x_i	Eingänge eines neuronalen Netzes
x_j	Basiseinheiten abgeleitete Größe
$y_{0,Feder}$	Auslenkung Federpendel Beginn
y_{Feder}	Auslenkung Federpendel
y_i	Label

Griechische Buchstaben

β	Konstante Adam-Algorithmus
η	Lernrate
κ	Konstante Rprop-Algorithmus
$\lambda_{Wärme}$	Wärmeleitzahl
$\lambda_{Zerfall}$	Zerfallskonstante
μ	Erwartungswert
μ_{Fermi}	chemisches Potential Fermiverteilung
Φ	Aktivierungsfunktion
Π	Fundamentalsystem
Π'	abgeleitetes Fundamentalsystem
π_k	dimensionslose Potenzprodukte
Σ	Übertragungsfunktion
σ	Standartabweichung
σ_2	Varianz
σ_{Dim}	Dimension Temperatur
θ_{Rap}	Rapididät
$\vartheta_{Wärme}$	Temperaturdifferenz

1. Einleitung

Im ersten Kapitel werden die Ausgangssituation und Zielsetzung der Untersuchung beschrieben. Des Weiteren wird der generelle Aufbau der Arbeit dargelegt.

1.1. Ausgangssituation

Die ersten Forschungsarbeiten zu künstlichen neuronalen Netzen stammen bereits aus dem Jahr 1943, in dem McCulloch und Pitts [26] erstmals die Funktionsweise des Nervensystems zur Lösung logischer Probleme nachbildeten. Aufgrund der höheren verfügbaren Rechenleistung werden künstliche neuronale Netze in den letzten Jahren in verschiedenen Bereichen der Forschung und des alltäglichen Lebens verwendet [3]. Dies begründet auch den hohen Umfang an aktuellen Forschungsarbeiten auf dem Gebiet der künstlichen neuronalen Netze.
Ein Großteil dieser Arbeiten beinhaltet die Themenkomplexe Bilderkennung oder Verarbeitung und Sprachverarbeitung. Bei beiden Themenfeldern handelt es sich um sogenannte Klassifizierungsprobleme. Hierbei werden beispielsweise einige Pixel aus einem Bild als ein Objekt aus einer vorgegebenen Menge erkannt. Oder es wird ein gesprochener Satz erkannt, indem jedes Wort in Sprachform ebenfalls als ein Objekt aus einer vorgegebenen Menge an Objekten (dem Wortschatz) erkannt wird.[9]
Neben den Klassifizierungsproblemen bilden die Regressionsprobleme die zweite große Klasse von Problemstellungen, die durch neuronale Netze gelöst werden können. Häufig handelt es sich hierbei um Schätzungen oder Voraussagen wie zum Beispiel von zukünftigen Temperaturen oder Aktienwerten.[3]
Abstrakt formuliert handelt es sich bei Regressionsverfahren jedoch um das Feststellen von Beziehungen zwischen verschiedener Variablen. Aus diesem Grund eignet sich das Verfahren neben den bereits erwähnten beispielhaften Einsatzszenarien auch zum Erkennen von physikalischen Zusammenhängen und deren symbolischen Beschreibung. [3] In diesem Kontext werden entweder durch Beobachtungen oder durch Experimente Messwerte für die einzelnen Variablen gewonnen. Der Informationsgehalt dieser Messwerte ist zweigeteilt und enthält neben dem eigentlichen Wert noch die zusätzliche Information über die Dimension des Messwertes. Verständlich wird dies am Beispiel einer Tüte Äpfel mit der Masse von 5 kg. Während 5 den eigentlichen Wert des Messwertes der Waage angibt, beschreibt die Einheit „kg“ die Dimension des Messwertes (Masse) [5]. Wird dieser Messwert in einem Regressionsverfahren genutzt, so wird unter Verwendung konventioneller neuronaler Netze häufig lediglich der Wert des Messwertes betrachtet und die Dimensionsinfomation ignoriert. Dies führt zu nicht dimensionshomogenen Berechnungen.

1.2. Zielsetzung

Das Ziel dieser Arbeit ist es, Möglichkeiten zur Nutzung der Dimensionsinformation im Rahmen von Regressionsverfahren auf Basis künstlicher neuronaler Netze zu beschreiben und zu evaluieren.
Untersucht werden sollen hierbei die Auswirkungen der zusätzlichen Information und deren Nutzung für das Training und die erreichbare Genauigkeit von künstlichen neuronalen Netzen. Grundlage für die Arbeit sind somit die Forschungsergebnisse von Rudolph [34] [32] an.
Konkret soll im Rahmen der Arbeit untersucht werden, ob das Nutzen der Dimensionsinformation zur Erzeugung zusätzlicher vollähnlicher Datenpunkte und das Erzwingen von dimensionshomogenen künstlichen neuronalen Netzen einen Einfluss auf das Training und die Vorhersagegenauigkeit des künstlichen neuronalen Netzes haben. Die Untersuchung erfolgt dabei anhand einer Auswahl bereits bekannter physikalischer Formeln auf Basis eines eigens entwickelten und implementierten Frameworks zur Erzeugung künstlicher neuronaler Netze.

1.3. Aufbau der Masterarbeit

Die Masterarbeit ist in insgesamt fünf Kapitel gegliedert. Neben der Einleitung umfassen diese theoretischen Grundlagen, Methodik, Ergebnisse inklusive deren Diskussion und das Fazit der Arbeit.
Das Kapitel „Theoretische Grundlagen“ enthält neben einer Einführung in die Theorie künstlicher neuronaler Netze auch eine Beschreibung der aktuell populären Frameworks „Tensorflow“[1] und „Pytorch“[2]. Außerdem erfolgen eine Einführung in die Dimensionsanalyse und die Präsentation des Forschungsstandes zum Thema künstliche dimensionshomogene neuronale Netze.
Das Kapitel „Methodik“ beinhaltet neben dem Konzept des entwickelten Frameworks auch die Beschreibung der Implementierung. Zusätzlich werden die durchzuführenden Versuche definiert und beschrieben.
Im darauffolgenden Kapitel „Ergebnisse“ werden in insgesamt drei Abschnitten die Ergebnisse der durchgeführten Versuche dargelegt. Des Weiteren werden diese Ergebnisse diskutiert und interpretiert. Den Abschluss der Arbeit bildet das Kapitel „Fazit“. Dieses enthält im Wesentlichen eine Zusammenfassung der Arbeit und gibt einen Ausblick auf weiterführende mögliche Forschungsfragen.

[1] https://www.tensorflow.org
[2] https://pytorch.org

2. Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen der Arbeit vorgestellt. Diese umfassen sowohl den Forschungsstand als auch Erklärungen zu den Themenkomplexen künstliche neuronale Netze und Dimensionsanalyse bzw. Ähnlichkeitsmechanik. Abgeschlossen wird das Kapitel mit der Vorstellung derzeit verfügbarer Frameworks zur Erstellung und Verwendung von neuronalen Netzen.
Der Abschnitt zu künstlichen neuronalen Netzen umfasst die Themen grundsätzlicher Aufbau von neuronalen Netzen, die Verwendung der Netze und dabei auftretende Probleme sowie gängige Verfahren zur Validierung. Im Abschnitt Dimensionsanalyse und Ähnlichkeitsmechanik werden das Buckinghamsche-Pi-Theorem und das Prinzip der Vollähnlichkeit vorgestellt. Die Kombination dieser beiden Wissensgebiete und deren Anwendung innerhalb eines der präsentierten Frameworks bildet den theoretischen Rahmen für die folgenden Experimente und deren Interpretation.

2.1. Neuronale Netze

Wie bereits in der Einleitung beschrieben, basieren künstliche neuronale Netze weiterhin auf dem von McCullon und Pitts [26] formulierten Ansatz der Nachbildung von organischen Nervensystemen. Diese Nervensysteme bestehen aus einzelnen Nervenzellen, die wiederum aus einem Zellkern mit Dendriten und mittels Axonen verbundenen Synapsen zusammengesetzt sind. Dieser Aufbau ist in Abbildung 2.1 dargestellt.

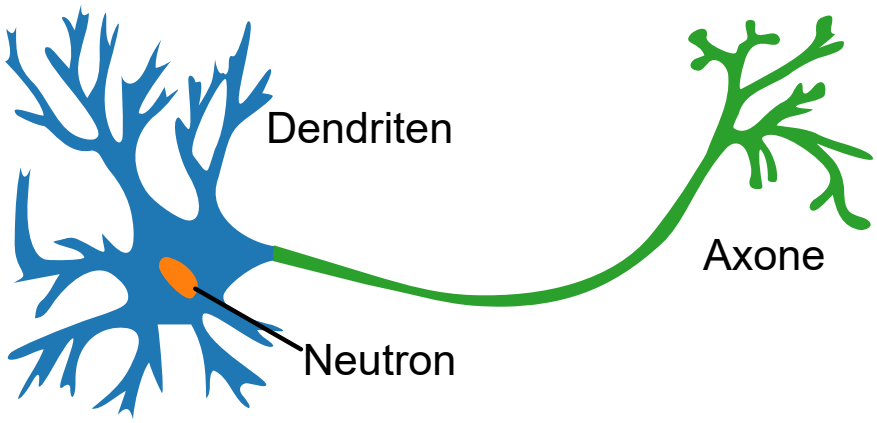

Abbildung 2.1.: vereinfachte Darstellung einer Nervenzelle [18]

Die Verbindung zweier Nervenzellen erfolgt über die Verbindung einer Synapse der einen Nervenzelle mit einem Dendriten der anderen Nervenzelle. Zum Austausch von Informationen werden zwischen den Synapsen und Dendriten der nachfolgenden Zelle sogenannte Botenstoffe

übertragen, sofern an den Dendriten der vorhergehenden Zelle ein bestimmtes elektrisches Potential überschritten wird. In diesem Fall ist vom Feuern einer Nervenzelle die Rede. Das Verhalten der Nervenzelle ist somit immer abhängig vom Verhalten der mit den Dendriten verbundenen Vorgängerzellen. Eine vom gesamten Nervensystem zu lösende Aufgabe wird hierdurch in viele kleine Entscheidungsvorgänge zwischen „Feuern“ und „nicht Feuern“ zerlegt. Entscheidend hierfür sind zum einen das Vorhandensein der Verbindungen zwischen den einzelnen Nervenzellen, und zum Anderen auch die zum „Feuern“ benötigten Aktivierungspotentiale. Die Geschwindigkeit der Signalübertragung hängt unmittelbar von der Dicke der Axonen ab. Beim Lernen werden die Anzahl und Stärke der Verbindungen zwischen den einzelnen Nervenzellen aufgrund von äußeren Einflüssen verändert (vgl. Pitts, McCulloch [29]).
Künstliche neuronale Netze bilden die beschriebenen Strukturen nach. Als Nervenzelle wird ein sogenanntes Perzeptron verwendet. Dieses in Abbildung 2.2 dargestellt [3].

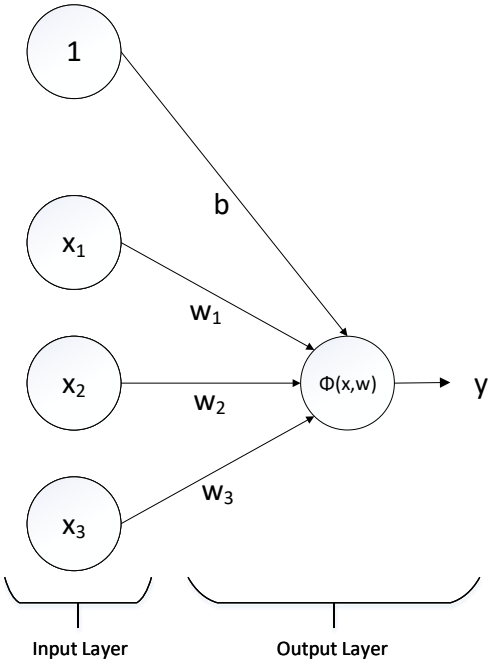

Abbildung 2.2.: Perzeptron

Über die Eingänge x_i ist das Perzeptron mit anderen Zellen verbunden. Eine sogenannte Aktivierungsfunktion Φ emuliert die Schwellwertfunktion und entscheidet somit, ob das Perzeptron analog der Nervenzelle, „feuert“ oder „nicht feuert“. Die Verbindungsstärke zwischen den Zellen wird durch sogenannte Gewichte nachgebildet. In Abbildung 2.2 sind diese Gewichte als w_i dargestellt. Ohne organisches Vorbild ist die sogenannte Bias b . Dieses verhält sich analog der Eingänge x_i, ist jedoch standartmäßig mit 1 belegt. Das elektrische Potential der Vorgängerzellen liegt an den Eingängen an. Die Gesamtheit des Aktivierungspotentials wird gemäß Formel 2.1 als Skalarprodukt der Aktvierungswerte x_i und der Gewichte w_i gebildet und als Übertragungsfunktion Σ bezeichnet. Die Variable n beschreibt die Anzahl der vorhandenen Verbindungen.

[3]

$$\Sigma(b, x, w) = b + \sum_{i=1}^{n} x_i \cdot w_i \tag{2.1}$$

Analog zum Nervensystem bestehen künstliche neuronale Netze aus einer Vielzahl an verbundenen Neuronen. Künstliche neuronale Netze lernen durch die Anpassung der Gewichte der einzelnen Neuronen mittels bereitgestellter Daten. Aufgrund dieses Aufbaus werden auch in künstlichen neuronalen Netzen die zu lösenden Probleme in viele kleine Einheiten aufgeteilt. Im Folgenden werden künstliche neuronale Netze lediglich als neuronale Netze bezeichnet.[3]
Wie in der Einleitung beschrieben, werden neuronale Netze heutzutage häufig für Aufgaben in den Bereichen Bilderkennung, Text- und Sprachverarbeitung, Anomalieerkennung und zur Vorhersage von Werten wie Preisen, Temperaturen oder anderer Indexe verwendet. Generell lassen sich neuronale Netze in Regressions- und Klassifikationsnetze einteilen. Regressionsnetze geben einen Wert aus einer stetigen Menge wieder, während Klassifikationsnetze einen Wert aus einer diskreten Menge ausgeben.[9]
Während für die meisten der genannten Anwendungsfälle Klassifikationsnetze verwendet werden, sind für die Approximation physikalischer Zusammenhänge lediglich Regressionsverfahren von Bedeutung. Die Klassifikationsverfahren werden aus diesem Grund nicht weiter beschrieben.
Im folgenden Absatz „Aufbau“ werden die Arten der Vernetzung der einzelnen Neuronen zu einem Netz und verschiedene gängige Aktivierungsfunktionen vorgestellt. Im Absatz „Verwendung neuronaler Netze“ werden die für Training und Validierung eines vorliegenden neuronalen Netzes benötigten Schritte und Verfahren beschrieben. Diese umfassen unter anderem Initialisierungsverfahren, Trainingsverfahren sowie Optimierungs- und Fehlerfunktionen. Im Abschnitt „Realisierungsprobleme“ werden bekannte Risiken und Probleme beim Training und Einsatz von neuronalen Netzen erläutert. Daran anküpfend werden im letzten Absatz des Abschnitts „neuronale Netze“ gängige Validierungsverfahren zur Problemerkennung und Risikominimierung beschrieben. Aufgrund der weiteren Verbreitung werden im weiteren Verlauf die englischen statt der deutschen Fachbegriffe verwendet.

2.1.1. Aufbau

Der Aufbau eines neuronalen Netzes wird als dessen Topologie bezeichnet. Diese Topologie besteht aus mehrern miteinander verbundenen Layern (deutsch Ebenen), die widerrum aus den einzelnen Perzeptronen bestehen. Wie in Abbildung 2.3 dargestellt, verfügt das Netz über jeweils einen Input- und Outputlayer sowie eine beliebige Anzahl an sogennanten hidden Layern. [9]

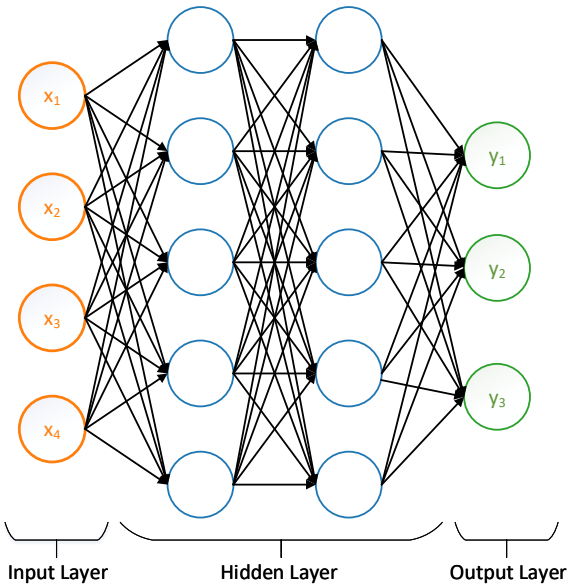

Abbildung 2.3.: Topologie eines neuronalen Netzes mit zwei hidden Layern [3]

Die einzelnen Neuronen werden als Kreise innerhalb der Layer dargestellt. Die Anzahl der Neuronen im Inputlayer entspricht der Anzahl der sogenannten Features (deutsch Attribute) des Datensatzes. Die Ergebnisse der Aktivierungsfunktionen der Neuronen im Outputlayer bilden die Ausgabe des neuronalen Netzes. Alle Layer des Netzes aus Abbildung 2.3 sind vom Typ dense. Dies bedeutet, dass jedes Neuron des Layers i mit jedem Perzptron des Layers $i-1$ verbunden ist.
Weitere bekannte Layertypen sind sogenannte convolutional und recurrent Layer. Innerhalb eines convolutional Layers wird zunächst eine Faltungsfunktion auf die Aktivierungswerte des Layers angewandt, bevor die Ausgänge in einem pooling Layer zusammengefasst und in einem darauffolgenden „dense“ Layer mittels der Übertragungs- und Aktivierungsfunktion weiterverarbeitet werden (siehe Abbildung 2.4). [3]

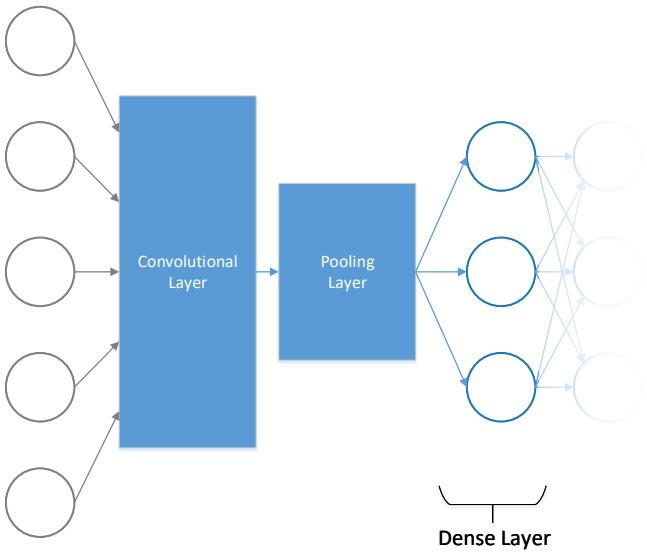

Abbildung 2.4.: Verwendung eines convolutional Layer

Convolutional Layer werden häufig in neuronalen Netzen zur Bild- und Sprachverarbeitung verwendet. Eines der bekanntesten neuronalen Netze mit convolutional Layern ist das Resnet Modell von He, Zhang, Ren und Sun [16].
Wie in Abbildung 2.5 dargestellt, werden bei recurrent Layern die Ausgänge der Perzeptronen auf die jeweiligen Eingänge rückgekoppelt. Diese Layer eignen sich für die Verarbeitung von Zeitreihen wie kontinuierlich aufgezeichnete Messwerte oder anderer sequentiell geordneter Daten. [3]

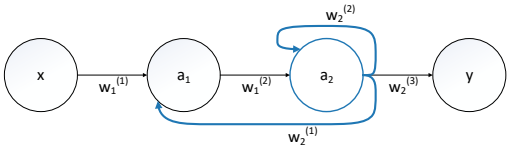

Abbildung 2.5.: Darstellung eines neuronalen Netzes mit einem recurrent Layer

Recurrent Layer finden aus diesem Grund hauptsächlich in der Sprachverarbeitung Anwendung [9].
Neuronale Netze mit wenigen hidden Layern werden als shallow (deutsch flach) bezeichnet. Im Gegensatz dazu heißen neuronale Netze mit vielen hidden Layern „Deep Neural Networks“ (abgekürzt DNN). Deep Neural Networks sind die derzeit am häufigsten verwendete Klasse neuronaler Netze. Sie eignen sich insbesonders zur Lösung von Problemen mit vielen Features. Im Vergleich zu shallow Networks werden in diesem Fall weniger einzelne Neuronen benötigt. DNNs werden deshalb unter anderem für komplexe bzw. rechenintensive Problemstellungen in der Bilderkennung oder Sprachverarbeitung eingesetzt. Aufgrund der geringeren Anzahl an Neuronen benötigen DNNs weniger Rechenzeit für das Training des Modells und verfügen über ein

verbesserte Interpretierbarkeit. Diese resultiert aus dem Fakt, dass über die einzelnen Layer hinweg beständig neue Zusammenhänge innerhalb des Netzes erkannt werden. Durch die Extraktion dieser Zwischenstände aus dem Netz, ist die Entscheidungsfindung des Netzes nachvollziehbar. Beispielsweise werden bei einem Gesichterkennungsalgorithmus, der durch ein DNN realisiert wird, zunächst einzelne Teile des Gesichtes wie Zähne oder Lippen von den ersten Layern erkannt. In darauffolgenden Layern wird aus diesen Bausteinen dann ein Mund ermittelt. Durch das Zusammensetzen der einzelnen Komponenten kann das Netz schlussendlich ein Bild bzw. einen Teil eines Bildes als Gesicht klassifizieren.[3]
Sowohl in shallow Networks als auch in DNN können verschiedene Layertypen miteinander verknüpft werden. Neben Typ und Anzahl der Layer wird die Topologie eines neuronalen Netzes maßgeblich von den Verbindungen der einzelnen Layer bestimmt. Die meisten Layer eines neuronalen Netzes sind sequentiell verbunden. Dies bedeutet, dass ein Layer immer mit exakt einem Vorgänger verbunden ist. Zunächst bedeutet dies, dass alle Neuronen des Layers mit allen Neuronen des Vorgängerlayers verbunden sind. Die Verbindungen zwischen zwei Neuronen können jedoch mit einem konstanten Gewicht w_i mit dem Wert 0 (vgl. Gleichung 2.1) praktisch gekappt werden. Entsprechend kann der Aktivierungswert des Vorgängerneurons in der Übertragungsfunktion des betrachteten Neurons ignoriert werden.[3]
Netze mit vielen konstanten Gewichten des Wertes 0 heißen sparse neural Networks (zu deutsch: spärlich besetzte Netze). Das Auflösen der Verbindungen auf diesem Weg kann entweder bei der Erstellung des Netzes oder dynamisch während des Trainings erfolgen. Generell wird versucht, Netze mit einer großen Anzahl an Features sparse zu gestalten. Es ergeben sich durch die daraus resultierende verringerte Anzahl an trainierbaren Gewichten sowohl eine kürzere Trainingslaufzeit als auch eine erhöhte Stabilität während des Trainings (vgl. Abschnitt 2.1.3). Außer durch das Auflösen von Verbindungen kann die Netztopologie auch durch das Hinzufügen von weiteren Verbindungen neben den bereits bestehenden sequentiellen Verbindungen verändert werden.[9]. Diese Verbindungen werden als „skip Connections“ oder „residual Connections“ bezeichnet und sind in Abbildung 2.6 dargestellt. [9] [16] Diese Verbindungen fungieren als Abkürzungen zwi-

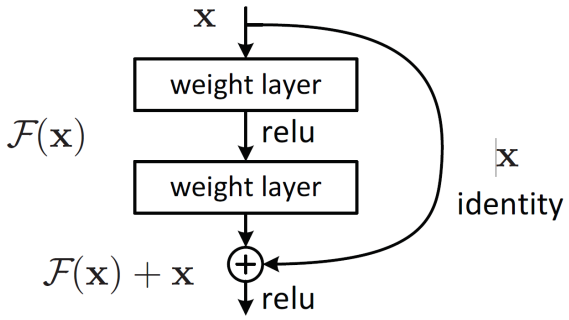

Abbildung 2.6.: Darstellung residual Connections [16]

schen bestimmten Layern. Bekannt wurden die residual Connections 2015 durch das „Resnet“ Netz (abgeleitet von residual neural network [16]) im Rahmen des Imagenet-Wettbewerbs 2015 aufgrund der wesentlich verbesserten Leistungsfähigkeit im Vergleich zu den vorherigen Wettbewerbsgewinnern.

Neben der Topologie werden beim Aufbau des Netzes die Aktivierungsfunktionen der Layer, bzw. der in den Layern enthaltenen Neuronen definiert. Die am häufigsten benutzten Funktionen sind die linear-, die Relu- und die Sigmoid- bzw. tanh- oder Hardtanh-Funktion. Die tanh-Funktion aus Formel 2.2 bildet die Funktion der organischen Nervenzelle mit den beiden Stati „feuern“und „nicht feuern“ nach.

$$f(x) = tanh(x) \tag{2.2}$$

Im Schaubild 2.7 ist deren Verlauf dargestellt.

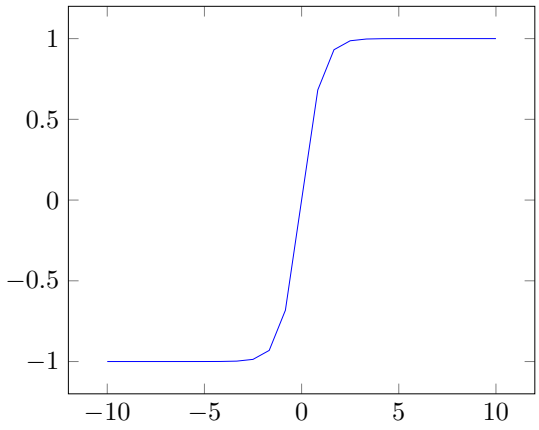

Abbildung 2.7.: tanh-Funktion

Analog zur tanh-Funktion bildet auch die Sigmoid-Funktion aus Formel 2.3 die Funktionsweise der organischen Nervenzelle nach.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

Der beinahe binäre Verlauf der Sigmoid-Funktion über den gesamten Definitionsbereich ist in Abbildung 2.8 dargestellt. [3]

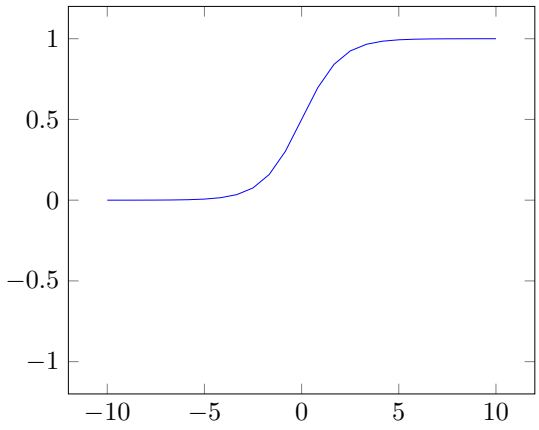

Abbildung 2.8.: Sigmoid-Funktion

Eine tatsächlich binäre Funktion wie die Vorzeichenfunktion aus Abbildung 2.9 ist aufgrund der fehlenden Differenzierbarkeit nicht für die Verwendung in neuronalen Netzen geeignet.

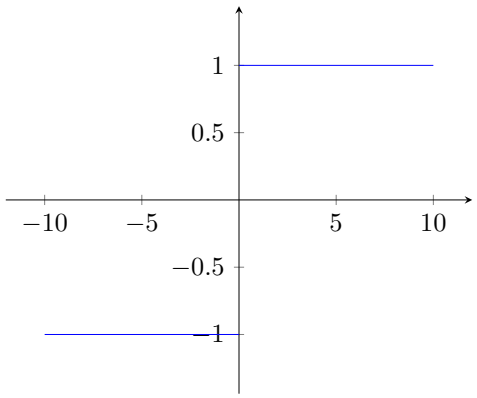

Abbildung 2.9.: Vorzeichen-Funktion

Die sigmoid-Funktion ist aufgrund der Beschränkung auf den Wertebereich von 0 bis 1 gut für die Repräsentation von Wahrscheinlichkeiten geeignet, während die tanh-Funktion aufgrund des größeren Gradienten und der möglichen Repräsentation von negativen Werten besser trainierbar ist und sich für Regressionsnetze eignet(vgl. Abschnitt 2.1.2).
Derzeit sind die Relu- (Rectified Linear Unit) und Hardtanh-Funktionen die am häufigsten verwendeten Aktivierungsfunktionen. Der Verlauf der Relu Funktion aus Formel 2.4 ist in Abbildung 2.10 dargestellt. Aufgrund des erweiterten Bildbereiches der Relu- und Linearfunktion (Formel 2.6) verfügen die zugehörigen Schaubilder über veränderte Achsenabschnitte.

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{2.4}$$

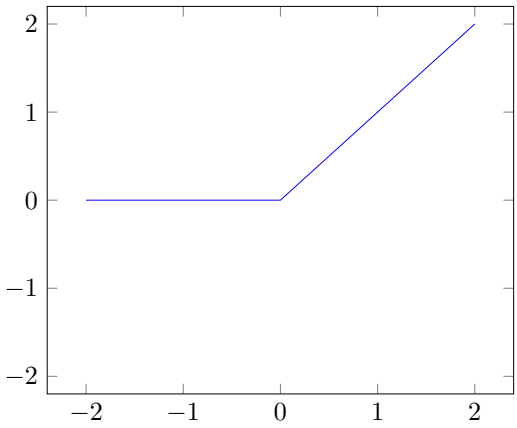

Abbildung 2.10.: ReLu-Funktion

Der Verlauf der Hardtanh-Funktion aus Formel 2.5 ist in Abbildung 2.11 dargestellt. [3]

$$f(x) = \begin{cases} -1 & x < -1 \\ x & -1 \leq x \leq 1 \\ 1 & x > 1 \end{cases} \tag{2.5}$$

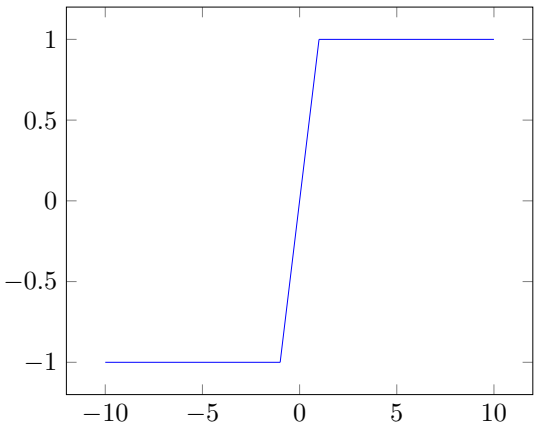

Abbildung 2.11.: Hardtanh-Funktion

Begründet wird die aktuelle Popularität durch deren konstanten Gradienten und die damit einhergehende vereinfachte Gradientenbestimmung im Trainingszyklus.
Die lineare Aktivierungsfunktion aus Formel 2.6 gibt den Wert der Übertragungsfunktion unverändert aus. Dies ist in Abbildung 2.12 dargestellt.

$$f(x) = x \tag{2.6}$$

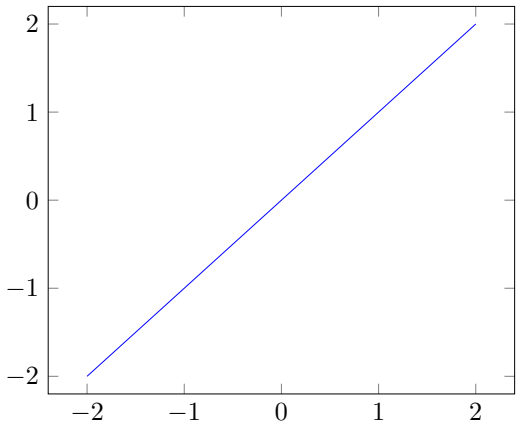

Abbildung 2.12.: linear-Funktion

Innerhalb eines Netzes können verschiedene Aktivierungsfunktionen verwendet und kombiniert werden. Dies ist insbesondere im Falle der Verwendung von linearen Aktivierungsfunktionen erforderlich, da sich neuronale Netze mit lediglich linearen Aktivierungsfunktionen analog zu einem Perzeptron verhalten.

2.1.2. Verwendung neuronaler Netze

Die Verwendung eines neuronalen Netzes wird in die vier Phasen Training, Validierung, Evaluierung und produktiver Einsatz gegliedert.
Ziel des Trainings ist, die für die verfügbaren Trainingsdaten idealen Gewichte des neuronalen Netzes zu ermitteln. Die daraus entstehende Abbildungsvorschrift wird als Modell bezeichnet. Durch die Validierung wird die generelle Anwendbarkeit des Modells auch auf unbekannte Validierungsdaten überprüft und gegebenenfalls die Topologie inkl. der Aktivierungsfunktionen des neuronalen Netzes angepasst.[3]
In der Evaluierungsphase wird anhand eines weiteren Datensatzes die schlussendliche Genauigkeit des Modells bestimmt. Zur Durchführung dieser drei Phasen wird der vorhandene Datensatz in die genannten Trainings-, Validierungs- und Evaluierungsdaten aufgeteilt. Ein typisches Verhältnis hierfür ist in Abbildung 2.13 abgebildet. [9]
Aufgrund der vorgegebenen Netztopologie und Aktivierungsfunktion, die sich aus dem Wissen um die tatsächlichen Formeln ergeben, wird im Rahmen der Arbeit auf die Evaluierungsphase verzichtet und lediglich mittels der Validierungsdaten die Generalisierbarkeit der ermittelten Abbildungsvorschrift überprüft.

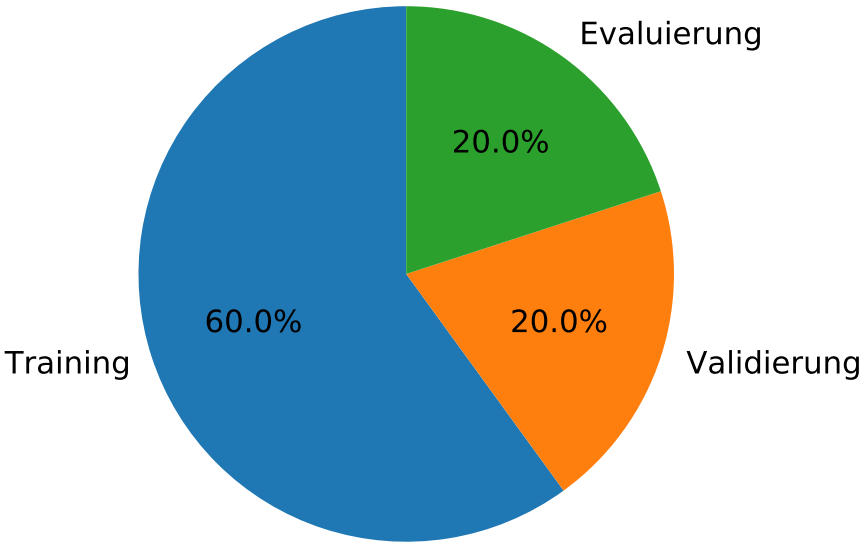

Abbildung 2.13.: Anteile der einzelnen Datensätze am gesamten Datensatz

Entscheidend für eine korrekte Validierung und Evaluierung des Modells ist, dass die entsprechenden Datensätze nicht Teil des Trainingsdatensatzes sind. Außerdem sollten die vorhandenen Daten zufällig auf die drei Datensätze aufgeteilt werden.
Das Training des neuronalen Netzes kann entweder in einem überwachten oder einem nicht

überwachten Verfahren erfolgen. In überwachten (englisch: supervised) Trainingsverfahren enthalten die Datensätze neben den Features auch Labels. Die Kombination aus Features und dem zugehörigen Label wird in überwachten Trainingsverfahren als Datenpunkt bezeichnet. Das Label eines Datensatzes entspricht dem durch das neuronale Netz gesuchten Wert und beschreibt somit die Sollausgabe des Modells. Überwachte Trainingsverfahren werden für Klassifikations- und Regressionsverfahren verwendet. Unüberwachte (englisch: unsupervised) Trainingsverfahren verfügen über keine Labels. Ihr Anwendungsschwerpunkt liegt auf Gruppierungsverfahren (z.B. Kaufempfehlungen ähnlicher Produkte).[9]
Unvollständige oder fehlerhafte Datenpunkte müssen aus dem Datensatz entfernt oder durch Approximationsverfahren korrigiert oder vervollständigt werden. Zur Verkürzung der Trainingslaufzeit und zur Erhöhung der Stabilität des Trainingsverfahrens kann die Anzahl der Features der Datenpunkte durch geeignete Dimensionsreduktionsverfahren wie durch z.B. die „Principal Component Analysis“ („PCA“) reduziert werden. Zusätzlich werden die Werte der Features meist in einem Vorverarbeitungsschritt vor dem Training normalisiert [17]. Der darauffolgende Prozess der Modellerstellung, Validierung und Evaluierung ist in Abbildung 2.14 dargestellt.

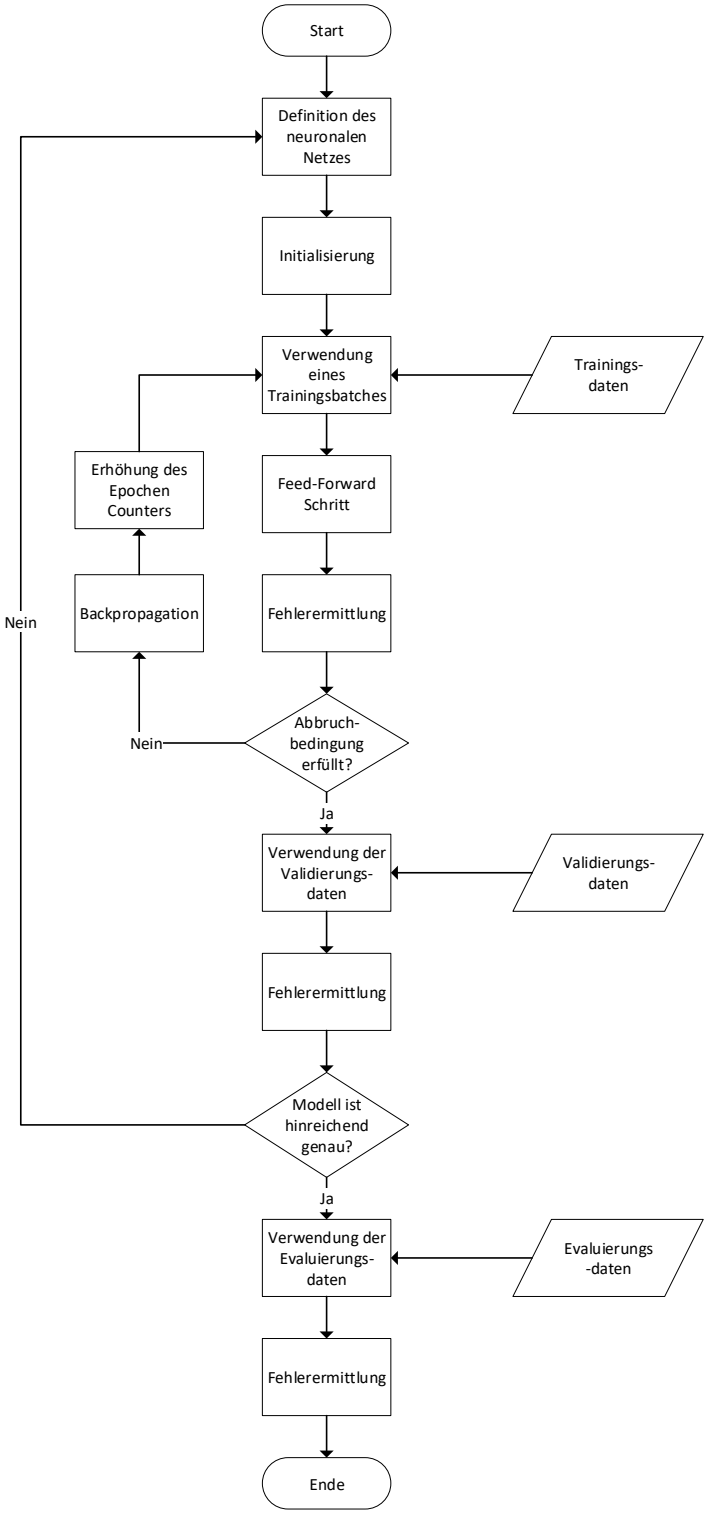

Abbildung 2.14.: Ablaufdiagramm des Trainings

In allen beschriebenen Netztopologien werden die Gewichte der einzelnen Layer zu Beginn initialisiert. Die Wahl dieser Initialisierung hat dabei einen entscheidenden Einfluss auf die für das Training benötigten Resourcen und die erreichbare Genauigkeit [23]. Zusätzlich hat die Initialisierung insbesondere bei DNN einen Einfluss auf die Stabilität des Netzes. In diesem Zusammenhang sei auf das Problem der verschwindenden oder explodierenden Gradienten aus Absatz 2.1.3 verwiesen [3].
Typischerweise werden für die Initialisierung Zahlen mit einem kleinen Betrag verwendet. Die für eine Problemstellung idealen Initialisierungswerte können jedoch nicht a priori bestimmt werden. Die verfügbaren Initialisungsverfahren lassen sich in konstante, stetig gleichverteilte, normalverteilte und komplexe Initialisierungsverfahren auf Basis der Netztopologie einordnen.
Konstanten als Initialisierungen sollten nach Möglichkeit vermieden werden, da ansonsten die einzelnen Neuronen innerhalb eines Layers die gleichen Gewichte annehmen und die Lernfähigkeit des Layers reduziert wird . Aus diesem Grund werden die Gewichte eines Layers normalerweise durch Zufallszahlen initialisiert [3].
Das einfachste Verfahren hierfür ist das Erzeugen von gleichverteilten Zufallszahlen mit kleinem Betrag (z.B. Wertebereich [-1,1]). Häufiger werden jedoch normalverteilte Zufallszahlen zur Initialisierung verwendet. Als Erwartungswert wird hierfür typischerweise der Wert 0 verwendet, während der Wert der Standartabweichung mit einem Wert $\sigma << 1$ besetzt wird. Es ergibt sich beispielhaft die in Abbildung 2.15 dargestellte Normalverteilung mit Erwartungswert $\mu = 0$ und der Varianz $\sigma^2 = 1$.

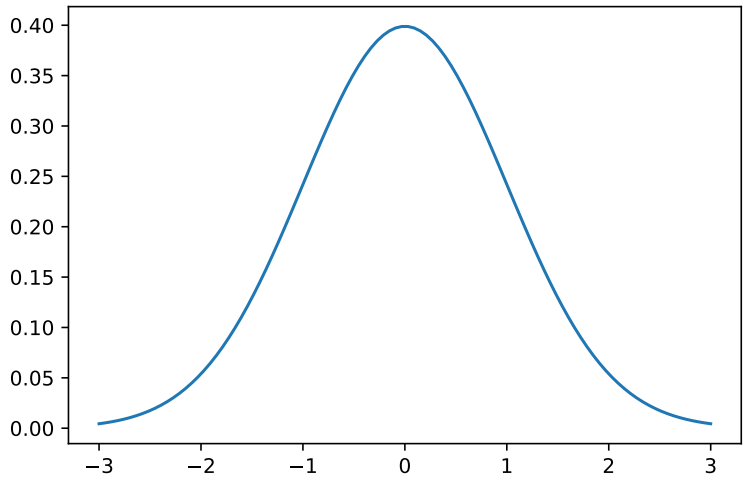

Abbildung 2.15.: Darstellung einer Normalverteilung

Im Gegensatz zu den vorhergehenden Initialisierungsverfahren nutzt der Xavier-Algorithmus die Anzahl der Neuronen im vorherigen Layer als Variable zur Erzeugung der Zufallszahlen [12]. Das Verfahren kann sowohl mit gleichverteilten als auch mit normalverteilten Zufallszahlen kombiniert werden. Im Falle einer gleichverteilten Initialisierung ergeben sich die Grenzen des Wertebereichs als $[-\sqrt{\frac{6}{n_{i-1}+n_i}}, \sqrt{\frac{6}{n_{i-1}+n_i}}]$ mit n_{i-1} als der Anzahl der Eingangsknoten und n_i als

die Anzahl der Ausgangsknoten des jeweiligen Layers. Für normalverteilte Initialisierungen wird die Standartabweichung σ zu $\sqrt{\frac{2}{n_{i-1}+n_i}}$. Ziel des Verfahrens ist es, die resultierende Verteilung der Zufallszahlen aus den einzelnen zufällig initialisierten Gewichten für das betrachetete Neuron zu normieren. Der Erwartungswert des normalverteilten Xavier-Verfahrens bleibt 0. Insbesondere bei einer großen Anzahl an Gewichten in einem Layer verhindert der Xavier-Algorithmus die in Absatz 2.1.3 beschriebenen Gradientenprobleme.
Im in Abbildung 2.14 dargestellten feedforward Prozess werden dem initialisierten Netz ein Batch an Trainingsdaten zugeführt und in Netzrichtung vom Input- zum Outputlayer verarbeitet. Bei einem Batch handelt es sich um eine festgelegte Menge an Datenpunkten. Diese Anzahl kann frei gewählt werden. Ist die Anzahl der Datenpunkte in einem Batch jedoch kleiner als die Anzahl der Datenpunkte im gesamten Trainingsdatensatz wird der Batch auch als Minibatch bezeichnet. Auf Basis der aktuellen Gewichte des Netzes, die im ersten Schritt noch den initialisierten Werten entsprechen, wird für jeden Datenpunkt ein Ausgabewert des Modells ermittelt [3].
Dieses Ergebnis wird im Anschluss mittels einer sogenannten Loss-Funktion im Prozessschritt Fehlerermittlung (vgl. Abbildung 2.14) mit dem tatsächlichen Label des Datenpunktes verglichen. Die vollständige Bearbeitung des Trainingsdatensatzes wird als Abschluss einer Epoche bezeichnet. Die Wahl der Fehlerfunktion erfolgt gemäß der zu lösenden Problemstellung bzw. aufgrund der Struktur des vorliegenden Datensatzes. Für die vorliegende Arbeit sind die Loss-Funktionen „Mean Squared Error“ (abgekürzt MSE[3], deutsch: Methode der Fehlerquadrate) und „Mean absolute Percentage Error“(abgekürzt MAPE) relevant [8].
Die MSE Funktion ist in Formel 2.7 dargestellt.

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.7}$$

L beschreibt den Error bzw. Loss, n die Größe des Batches , y_i den Wert des Labels eines Datenpunktes und $\hat{y}_i$ den durch das Modell berechneten Wert für den Datenpunkt [3].
Die MAPE-Funktion ist analog in Formel 2.8 dargestellt.

$$L = \frac{1}{n}\sum_{i=1}^{n}|\frac{y_i - \hat{y}_i}{y_i}| \tag{2.8}$$

Wie in Abbildung 2.14 dargestellt, wird nach der Ermittlung des Fehlers für die gesamte Epoche ein Abbruch des Trainings anhand zuvor festgelegter Abbruchparameter überprüft. Typische Parameter sind eine festgelegte Anzahl der Epochen oder das Unterschreiten eines bestimmten Loss-Wertes. Ist keine der Abbruchbedingungen erfüllt, versucht der Trainingsalgorithmus mittels der Backpropagation durch das Gradientenverfahren die Gewichte zu optimieren und den ermittelten Loss zu minimieren. Die Optimierung erfolgt entgegen der Netzrichtung vom Ouputlayer aus in Richtung des Inputlayers. Vorraussetzung ist, dass die Loss-Funktion differenzierbar ist. Da diese bei ausschließlicher Verwendung des Skalarproduktes als Übertragungsfunktion lediglich von den Gewichten und Aktivierungsfunktionen abhängen, müssen diese Funktionen differenzierbar sein (vgl. Abschnitt 2.1.1).
Das Gradientenverfahren kann durch Zerlegung in einzelne Terme effizent mittels dynamischer Programmierung berechnet werden. Die Anpassung der Gewichte erfolgt neuronenweiße. Jedes Neuron eines hidden Layers ist entweder direkt oder über andere Neuronen mit dem Outputlayer

verbunden. Diese Verbindungen werden im Folgenden als Pfade bezeichnet. Aus den Aktivierungsfunktionen Φ und Gewichten w_i des Neurons und der, in Netzrichtung gesehen, folgenden Neuronen wird die Formel für den Loss über den Gewichten des betrachteten Neurons gebildet [3].
Die Gradienten ergeben die Formel 2.9. L beschreibt den Loss, h_r das betrachtete Neuron, $w_{(h_{r-1},h_r)}$ das Gewicht des Neurons für die Verbindung mit einem Neuron aus dem vorhergehenden Layer, o die Ausgabe des Outputlayers und k die Anzahl der hidden Layer im Netz [3].

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial L}{\partial o}\left[\sum \frac{\partial o}{\partial h_k}\prod_{i=r}^{k-1}\frac{\partial h_{i+1}}{\partial h_i}\right]\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \tag{2.9}$$

Die Formel 2.9 kann in zwei Teile zerlegt werden. Der erste Teil entspricht Formel 2.10 mit dem Ergebnis a_{h_r} der Übertragungsfunktion des Neurons h_r und der Aktivierungsfunktion Φ. Diese Gleichung kann direkt gelöst werden.

$$\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} = h_{r-1} * \Phi'(a_{h_r}) \tag{2.10}$$

Der zweite Teil bildet die Formel 2.11. Diese muss für jeden vorhandenen Pfad ausgewertet werden.

$$\frac{\partial L}{\partial o}\left[\sum \frac{\partial o}{\partial h_k}\prod_{i=r}^{k-1}\frac{\partial h_{i+1}}{\partial h_i}\right] = \Delta(h_r, o) = \frac{\partial L}{\partial h_r} \tag{2.11}$$

Da es sich bei einem neuronalen Netz um einen gerichteten, kreisfreien Graphen handelt, kann diese Gleichung ausgehend vom Outputlayer entgegen der Netzrichtung für alle Neuronen gelöst werden. der Gradient für jedes Neuron im Outputlayer wird in Formel 2.12 ermittelt.

$$\Delta(L, o) = \frac{\partial L}{\partial o} \tag{2.12}$$

Ausgehend von diesen diesen Gradienten werden die Gradienten der Neuronen aus den Hidden Layern gemäß Formel 2.13 und Formel 2.14 ermittelt. Für h_k gilt $h_{k+1} = o$.

$$\Delta(h_r, o) = \frac{\partial L}{\partial h_r} = \sum \frac{\partial L}{\partial h}\frac{\partial h}{\partial h_r} = \sum \frac{\partial h}{\partial h_r}\Delta(h, o) \tag{2.13}$$

$$\frac{\partial h}{\partial h_r} = \Phi'(a_h) * w_{h_r,h} \tag{2.14}$$

Die hierdurch berechneten Gradienten aus Formel 2.9 passen gemäß Formel 2.15 die Gewichte des jeweiligen Neurons an.

$$w_{(h_{r-1},h_r)_{neu}} = w_{(h_{r-1},h_r)} - \eta \cdot \frac{1}{M}\sum_{k=1}^{M}\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} \tag{2.15}$$

Bei η handelt es sich um die Lernrate. Diese bestimmt die Geschwindigkeit der Anpassung der Gewichte und hat hierdurch einen großen Einfluss auf die durch das Modell erreichbare Genauigkeit und die Trainierbarkeit des neuronalen Netzes. M bezeichnet die Anzahl der untersuchten

Datenpunkte. Beim klassischen Gradientenabstiegsverfahren erfolgt die Anpassung der Gewichte nach jeder Epoche [9].
Durch große Lernraten η wird die Loss-Funktion schnell minimiert. Dies geschieht jedoch auf Kosten der erzielbaren Genauigkeit und Stabilität. Aufgrund von zu kleinen Lernraten η wird die Anzahl der benötigten Anpassungen zum Erreichen des Minimums der Loss-Funktion groß. Dies resultiert in einer langen Laufzeit des Trainingsalgorithmus. Die aus einer falsch gewählten Lernrate resultierenden Probleme werden im folgenden Absatz 2.1.3 näher beschrieben [3].
Zur Optimierung des Backpropagation-Algorithmus werden spezielle Optimierungsverfahren (englisch: optimizer) eingesetzt. Diese passen die Lernrate η dynamisch während des Trainings an oder aktualisieren die Gewichte nicht nur nach Abschluss einer Epoche. Ziel ist es, eine optimale Lernrate zu ermitteln, um den Bedarf an Trainingsresourcen zu beschränken. Die bekanntesten Verfahren sind „Stochastic Gradient Descent“ (Abgekürzt SGD), „RMSprop“ und „Adam“.
Im Gegensatz zum klassischen Gradientenabstiegsverfahren werden die Gewichte nach jedem Verarbeitszyklus eines Datenpunktes aktualisiert. Die Variable M in Gleichung 2.15 hat somit den Wert $M = 1$. Die Reihenfolge der Datenpunkte wird dabei zufällig gewählt. Das Verfahren führt zu wesentlich kürzeren Trainingszeiten und einem verminderten Speicherbedarf. Gleichzeitig führt das Verfahren jedoch zu einem zusätzlichen Fehler im Vergleich zum klassischen Gradientenabstiegsverfahren. Durch das Bilden des Durchschnitts der Gradienten werden starke Schwankungen vermieden und die Stabilität des Algorithmus verbessert.
Das Prinzip und die Anwendung werden in Bouttou [4] eingängig beschrieben. In Abbildung 2.16 ist der Verlauf der Loss-Funktion über den Gewichten für das „Stochastic Gradient Descent“-Verfahren links und das konventielle Verfahren rechts dargestellt.

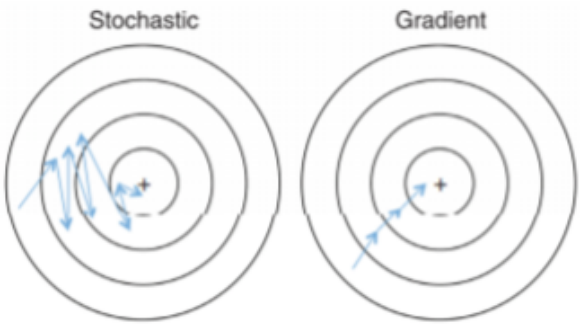

Abbildung 2.16.: Vergleich Gradientenverfahren [6]

Eine Mischform des SGD und des klassischen Gradientenabstiegsverfahrens ist das Mini-Batch-SGD-Verfahren. Hierbei werden die Gewichte mittels der Gradienten nach Bearbeitung eines Mini-Batches angepasst. M in Formel 2.15 entspricht der Anzahl der Datenpunkte im Mini-Batch. Analog dem klassischen Gradientenabstiegsverfahrens werden extrem große und kleine Gradienten durch die Bildung des Durchschnitts über alle Gradienten im Mini-Batch vermieden. Gleichzeitig werden die Anforderungen an den Speicherbedarf im Vergleich zum klassischen Verfahren reduziert, und die Anzahl der Aktualisierungen der Gewichte erhöht [3].
Dieses Verfahren wird vom „RMSprop“-Algorithmus [37] neben einem Verfahren zur dynamischen Anpassung der Lernrate verwendet. Hierfür basiert es auf dem Resilient-Backpropagation-(Abgekürzt „Rprop“) Verfahren von Riedmiller und Braun [31]. Die Methode basiert auf dem

Vergleich der Vorzeichen des aktuell ermittelten Gradienten eines Gewichts mit dem zuletzt ermittelten Gradienten (siehe Formel 2.16). Im Falle einer Übereinstimmung wird die Lernrate η erhöht, während im Falle von verschiedenen Vorzeichen die Lernrate η verringert wird. Begründet wird dies mit dem zwangsweißen Vorliegen eines zumindest lokalen Minimums innerhalb des zuletzt durchgeführten Optimierungsschrittes. Statt einer globalen und konstanten Lernrate η ergibt sich somit für jedes Gewicht eine angepasste Lernrate η_r. Die beiden Konstanten κ^+ und κ^- können frei gewählt werden. t gibt den Iterationsschritt an.

$$\eta_r = \begin{cases} \kappa^+ \cdot \eta_r^{t-1}, wenn \ \frac{\partial L}{\partial w_{(h_{r-1},h_r)}}^{(t-1)} \cdot \frac{\partial L}{\partial w_{(h_{r-1},h_r)}}^{(t)} > 0 \\ \kappa^- \cdot \eta_r^{t-1}, wenn \ \frac{\partial L}{\partial w_{(h_{r-1},h_r)}}^{(t-1)} \cdot \frac{\partial L}{\partial w_{(h_{r-1},h_r)}}^{(t)} < 0 \\ \eta_r^{t-1}, sonst \end{cases} \quad mit\ 0 < \kappa^- < 1 < \kappa^+ \tag{2.16}$$

Das RMSprop-Verfahren (Resilient Mean Square Backpropagation) kombiniert das Rprop Verfahren mit dem Mini-Batch-SGD-Verfahren indem der Gradient eines Gewichts durch den gleitenden Mittelwert nach Formel 2.17 geteilt wird.

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)\,neu}} = \frac{\partial L}{\partial w_{(h_{r-1},h_r)}} \cdot \frac{1}{\sqrt{MeanSquare(w,t)}} \tag{2.17}$$

mit $MeanSquare(w,t)$ gemäß Formel 2.18.

$$MeanSquare(w,t) = 0.9 \cdot MeanSquare(w,t-1) + 0.1 \cdot (\frac{\partial L}{\partial w_{(h_{r-1},h_r)}})^2 \tag{2.18}$$

Die spezifische Lernrate eines Gewichts bleibt hierdurch innerhalb einer Epoche nahezu konstant.
Das Adam (Adaptive-Moment-Optimization)-Verfahren [20] erweitert das RMSprop-Verfahren um den Momentum-Algorithmus [30]. Dieser bildet eine gewichtete Summe $v_{t,r}$ über die Gradienten der vorherigen Dimensionen gemäß Formel 2.19. λ ist eine festgelegte Konstante, für die gilt $0 < \lambda \leq 1$.

$$v_{t,r-1,r} = \lambda * v_{t-1,r-1,r} - \eta \cdot \frac{\partial L}{\partial w_{(h_{r-1},h_r)}} \tag{2.19}$$

Die neuen Gewichte ergeben sich aus der folgenden Formel 2.20.

$$w_{(h_{r-1},h_r)_{neu}} = w_{(h_{r-1},h_r)} + v_{t,r-1,r} \tag{2.20}$$

Das Momenten-Verfahren resultiert für aufeinanderfolgende Gradienten mit gleichem Vorzeichen für ein Gewicht in einer Vergrößerung des Betrags des zuletzt ermittelten Gradienten und somit einer größeren Änderung des Gewichts.
Das Adam-Verfahren erzeugt ähnlich dem Momentum-Verfahren (vgl. 2.19) eine gewichtete Summe der vorhergehenden Gradienten gemäß Formel 2.21

$$v_{t,r-1,r} = \beta_1 * v_{t-1,r-1,r} - (1-\beta_1) \cdot \frac{\partial L}{\partial w_{(h_{r-1},h_r)}} \tag{2.21}$$

und einen gewichteten Mittelwert ähnlich dem RMSprop-Algorithmus (vgl. 2.18) nach Formel 2.22.

$$s_{t,r-1,r} = \beta_2 \cdot s_{t-1,r-1,r} - (1-\beta_2) \cdot (\frac{\partial L}{\partial w_{(h_{r-1},h_r)}})^2 \tag{2.22}$$

Die neuen Gewichte ergeben sich dann aus Formel 2.23

$$w_{(h_{r-1},h_r)_{neu}} = w_{(h_{r-1},h_r)} - \eta \cdot \frac{v_{t,r-1,r}}{\sqrt{s_{t,r-1,r} + \epsilon}} \cdot \frac{\partial L}{\partial w_{(h_{r-1},h_r)}} \tag{2.23}$$

η beschreibt eine festgesetzte initiale Lernrate. Bei β_1 und β_2 handelt es sich um wählbare Konstanten. Die Konstante ϵ dient der Erhöhung der Stabilität und wird mit einem kleinen Wert belegt (Die Autoren in Kingma und Ba [20] empfehlen 10^{-8}).

2.1.3. Realisierungsprobleme

In diesem Abschnitt werden typische auftretende Probleme beim Trainieren und Verwenden von neuronalen Netzen beschrieben. Diese lassen sich in die Kategorien Probleme mit dem Datensatz, Stabilität des Verfahrens, Laufzeit des Trainings, mangelhafte Genauigkeit und ungenügende Generalisierbarkeit des Modells einteilen. Die Nutzbarkeit und Qualität des zum Training und der Validierung von neuronalen Netzen verfügbaren Datensatzes kann gemäß Klein und Rossin [21] in die Dimensionen Genauigkeit, Gültigkeit, Vollständigkeit und Konsistenz eingeteilt werden. In ihrer Veröffentlichung zeigen die Autoren zwar, dass neuronale Netze für Regressionsverfahren von geringen Abweichungen im Trainingsdatensatz profitieren können, die Leistungsfähigkeit eines Modells nimmt jedoch mit sinkender Qualität der Trainings- und Validierungsdaten ebenfalls ab.
Neben der Qualität der Datenpunkte ist nach Foody und McCulloch [11] die Anzahl der vollständigen Datenpunkte ausschlaggebend für die Genauigkeit und Zuverlässigkeit eines neuronalen Netzes. Neuronale Netze benötigen Datenpunkte mit vollständigen Features für die Trainings- und Validierungsphase. Um die Anzahl der nutzbaren Datenpunkte eines Datensatzes zu erhöhen existieren verschiedene Algorithmen [35], die unvollständige Datenpunkte vervollständigen. Hierzu zählt neben einfachen Interpolationsverfahren auch die Verwendung eigens erstellter separater, neuronaler Netze. Ein weiteres Verfahren zur künstlichen Vervielfachung der vorhandenen Trainingsdaten heißt Data Augmentation. Hierbei werden aus den bestehenden Datenpunkten durch Umformumgen neue Invarianten erzeugt [36]. Es stehen hierfür eine Vielzahl an verschiedenen Verfahren bereit(vgl. Golkov und Cremers [22]).
Eine große Anzahl an Datenpunkten hilft insbesondere gegen das sogenannte Overfitting. Dieses Phänomen beschreibt ein neuronales Netz, das über einen kleinen Loss bei der Verwendung der Trainingsdaten im feedforward-Schritt verfügt, wobei der Loss der Validierungsdaten jedoch wesentlich größer ist. In einem solchen Fall ist das Netz überangepasst auf die verwendeten Trainingsdaten, es generalisiert jedoch nur ungenügend. Die Modellierungsleistung für unbekannte Datenpunkte ist somit gering. In Abbildung 2.17 ist der Verlauf der Lossfunktion für zwei neuronale Netze über den Trainingsiterationen für das jeweilige Trainings- und Validierungsset dargestellt.

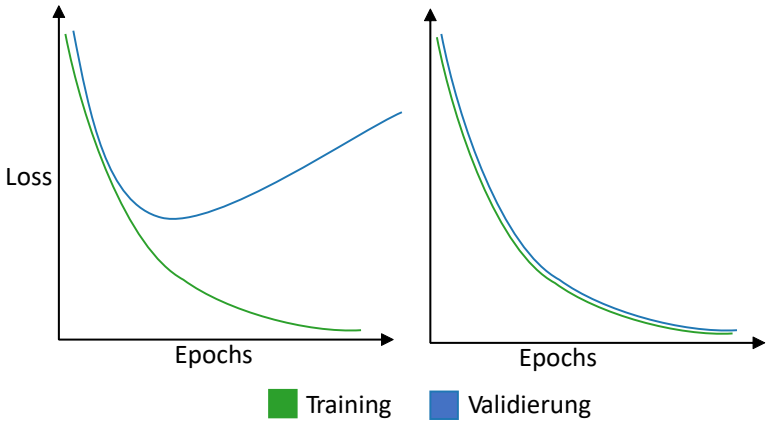

Abbildung 2.17.: Overfitting

Das neuronale Netz des linken Schaubilds ist von Overfitting betroffen, wie an der vergleichsweise großen Differenz der Werte der Loss-Funktion zu erkennen ist. Das Netz des rechten Schaubildes ist nicht von Overfitting betroffen. Overfitting lässt sich demzufolge durch eine hohe Varianz der Loss-Funktion für verschiedene Datensätze erkennen. Aus diesem Grund wird Overfitting auch als Variance-Problem bezeichnet.
Neben einer höheren Anzahl an Trainingsdatenpunkten beugen auch an die Struktur des Datensatzes, angepasste Netztopologien oder das sogenannte „Early Stopping“ Overfitting vor.
Beim „Early Stopping“ wird der Gradient der Loss-Funktion der Validierungsdaten als zusätzliche Abbruchbedingung für das Training genutzt. Das Training wird nach einer bestimmten Anzahl an Iterationen mit einem positiven Gradienten, bzw. einem steigenden Loss des Validierungsdatensatzes abgebrochen (vgl. Abbildung 2.17).
Durch Regularisierung kann Overfitting ebenfalls verhindert werden. Hierbei wird der Parameterraum der Gewichte reduziert bzw. minimiert. Dies kann entweder durch eine Minimierung der Anzahl der zu optimiereden Gewichte geschehen, indem die Anzahl der Neuronen verringert wird und die Komplexität des Netzes hierdurch sinkt, oder durch das Entfernen von Verbindungen zwischen Neuronen realisiert werden. Eine weitere Regularisierungsmethode ist die Limitierung bzw. Minimierung der absoluten Werte der Gewichte. Die Umsetzung erfolgt durch die Einführung einer sogenannten Penalty auf Basis des absoluten Gewichts.
Die am häufigsten verwendete Methode ist die sogenannte L_2-regularization. Diese Penalty L_2 wird gemäß Formel 2.24 auf Basis der Gewichte w_i und einer Konstansten λ berechnet und gemäß Formel 2.25 zum bestehenden Loss L des Modells addiert [3].

$$L_2 = \lambda \cdot \sum_{i=1}^{d} w_i^2 \tag{2.24}$$

$$L_{reg} = L(y, \hat{y}) + L_2(\lambda, w) \tag{2.25}$$

Beim Gegenteiligen Phänomen des „Underfitting“ verfügt das erstellte und trainierte neuronale Netz über einen zu kleinen Parameterraum bzw. eine unzureichende Komplexität. Ersichtlich

wird dies an einem hohen Loss-Wert der Trainingsdaten nach Abschluss des Trainings. Eine solcher Verlauf ist beispielhaft in Abbildung 2.18 dargestellt.

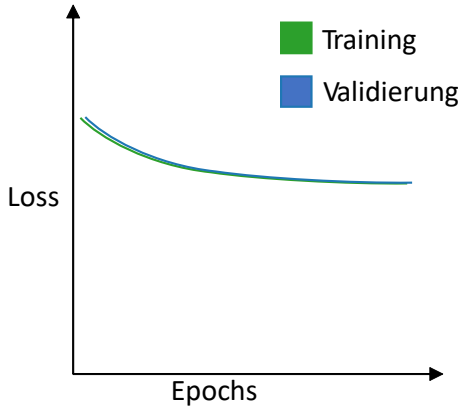

Abbildung 2.18.: Underfitting

Die Erhöhung der Komplexität des Netzes durch das Hinzufügen zusätzlicher Layer oder Verbindungen reduziert das Problem [9]. Bei der Konzeption eines neuronalen Netzes muss somit eine Balance (Bias-Variance-Tradeoff) zwischen Underfitting und Overfitting für die Ermittlung optimaler Netztopologien gefunden werden.
Hierfür werden die Loss-Werte der Trainings- und Validierungsdaten über verschiedene Netze für den selben Datensatz miteinander verglichen. Ein Beispiel hierfür ist in Abbildung 2.19 zu sehen.

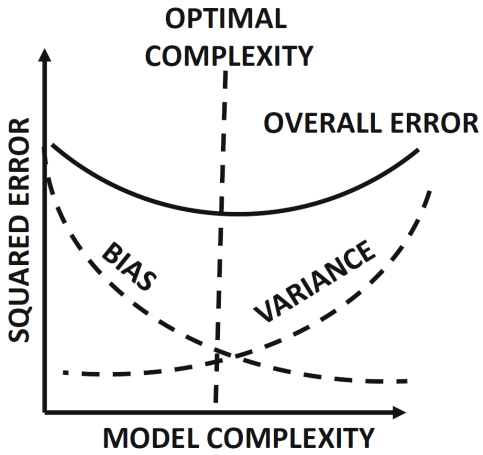

Abbildung 2.19.: Bias-Variance-Tradeoff [3]

Die benötigte Laufzeit für das Training eines neuronalen Netzes ist ebenfalls eine der zentralen Herausforderungen bei der Verwendung von neuronalen Netzen bzw. der Entscheidung für

komplexe Netztopologien. Das Training komplexer Netze kann Tage oder Wochen benötigen. Eine wesentliche Verkürzung der Laufzeit ist durch den Einsatz von spezieller Hardware wie Grafikkarten (Abgekürzt „GPU“ für Graphic Processing Units) oder Tensor Processing Units (abgekürzt „TPU“ [19]) möglich. Diese Architekturen ermöglichen die hochparallelisierte Verarbeitung der Trainingsdaten und verhelfen neuronalen Netzen zu der in der Einleitung erwähnten aktuellen Popularität.
Neben der hohen benötigten Rechenleistung können auch „vanishing Gradients“ die Dauer des Trainings verlängern. Dieses Phänomen wurde von Sepp Hochreiter [17] 1998 entdeckt und beschreibt die Tatsache, dass die Gradienten im Backpropagationverfahren entgegen der Netzrichtung gegen Null tendieren sofern in den Layern Aktivierungsfunktionen mit Gradienten mit einem Betrag kleiner 1 vorkommen. Somit sind insbesonders Netze mit den Aktivierungsfunktionen tanh und sigmoid von diesem Problem betroffen (vgl. Abbildung 2.7 und Abbildung 2.8). Da nach Formel 2.9 der Gradient eines Gewichtes aus dem Produkt der Gewichte der Nachfolgergradienten in Netzrichtung betrachtet gebildet kommt es nach Formel 2.26 für die Aktivierungsfunktion Φ zu einem exponentiellen Verfall der Gradienten entgegen der Netzrichtung.

$$\frac{\partial L}{\partial h_t} = \Phi'(h_{t+1}) \cdot w_{t+1} \cdot \frac{\partial L}{\partial h_{t+1}} \tag{2.26}$$

Aus den verschwindenen Gradienten ergeben sich analog zu den geringen Lernraten nur kleine Anpassungen der Gewichte innerhalb eines Trainingsschritts. Dieser Umstand erfordert eine größere Anzahl an Trainingsiterationen, um das Minimum der Loss-Funktion zu erreichen.
Das gegenteilige Phänomen der vanishing Gradients sind exploding Gradients, die sich für die ersten Layer in Netzrichtung für Beträge der Gradienten größer als 1 nach Formel 2.26 ergeben. Die Folge sind ein mathematisch instabiles Modell und eine verringerte Lernfähigkeit des neuronalen Netzes.

2.1.4. Validierung und Evaluierung

Wie bereits in Abbildung 2.13 dargestellt, wird der verfügbare Datensatz zum Training, zur Validierung und Evaluierung der Leistungsfähigkeit des neuronalen Netzes aufgeteilt. Das Ziel der Evaluierung von neuronalen Netzen ist es, die Genauigkeit der Ausgabe der trainierten Modelle für bisher unbekannte Eingangsdaten einzuschätzen. Entscheidend hierfür sind die im vorherigen Absatz 2.1.3 beschriebenen Kennzahlen Bias und Variance. Zur Ermittlung dieser Kennzahlen existiert das Hold-Out- und das Cross-Validation-Verfahren. Im Hold-Outverfahren werden, wie in Absatz 2.1.2 beschrieben, mit den Trainingsdaten zunächst die optimalen Gewichte der Neuronen bestimmt. Im zweiten Schritt werden die Validierungsdaten zur Feststellung der optimalen Netztopologie, Initialisierungsparameter und der Optimierungsfunktion genutzt. Abschließend wird durch die Test- oder Evaluierungsdaten die Leistungsfähigkeit des Modells bestimmt. Die Abweichung der vom Modell ermittelten Werte für die Features der Testdaten im Vergleich zu den Labels der Daten beschreiben die Bias und die zu erwartende Genauigkeit des Modells. Diese entspricht dem Ergebnis der Loss-Funktion. Die zweite Kennzahl der Modellgüte ergibt sich aus der Varianz zwischen den Ergebnissen der Loss-Funktionen für die betrachteten Datensätze. Diese Kennzahl gibt die Zuverlässigkeit der Genauigkeitsschätzung für unbekannte Werte an. Sowohl für die Bias als auch die Variance wird das Erreichen des Minimums angestrebt. Der Name „Hold-Out“-Verfahren ergibt sich aus der strikten Seperation der verschiedenen Daten-

sätze. Durch das Verfahren ist Overfitting (vgl. Abschnitt 2.1.3) eines Modells erkennbar. Das Hold-Out-Verfahren ist einfach und effizent anzuwenden, allerdings ist die durch das Verfahren ermittelte Genauigkeit prinzipbedingt schlechter als die tatsächlich zu erwartende Genauigkeit. Das Cross-Validation-Verfahren erreicht eine genauere Vorhersage der tatsächlichen Genauigkeit, erfordert jedoch mehr Rechenleistung. Bei diesem Verfahren wird der gesamte verfügbare Datensatz in k gleich große Teile aufgeteilt. Es wird k mal ein Modell mit den Daten erstellt, wobei jeweils einer der k Teildatensätze als Testdaten und die restlichen $k-1$ Teildatensätze als Trainingsdaten verwendet werden. Die Bias des Modells ergibt sich aus dem Durchschnitt der einzelnen Bias-Werte der jeweiligen Modelle. Die Variance entspricht der Verteilung der Bias-Werte. Aufgrund der großen Anzahl an zu trainierenden Modellen und der daraus resultierenden benötigten Rechenleistung bzw. Laufzeit eignet sich dieses Verfahren hauptsächlich für kleine Datensätze und ist in modernen Anwendungen nur selten anwendbar [3].
Ein weiteres Gütekriterium ist die Robustheit eines neuronalen Netzes. Diese beschreibt den Umgang mit kleinen Störungen in den Eingangsdaten des neuronalen Netzes bzw. den Einfluss dieser Störungen auf die Ausgabe des Modells [25]. Die Ausgabe eines robusten neuronalen Netz entspricht bei einer begrenzten Störung der Eingabe weitestgehend oder vollständig (vorallem bei Klassifikationsmodellen) der Ausgabe des jeweiligen ungestörten Datenpunktes. Eine formale Definition erfolgt in Formel 2.27 (Lipschitzbedingung [27]).

$$|\hat{y}(x) - \hat{y}(x')| \leq L|x - x'| \tag{2.27}$$

$\hat{y}(x)$ stellt die Ausgabe des Modells dar, während x die Eingabedaten repräsentiert. x' ist der gestörte abgeleitete Datenpunkt x. Für robuste neuronale Netze muss das Libischtzkriterium global mit einem kleinen L erfüllt sein. Letzteres gibt den Faktor zwischen der Distanz des ungestörten und des gestörten Eingangsdatenpunktes und der Distanz zwischen der Ausgabe des Modells für die jeweiligen Datenpunkte an.
Eine letzte Möglichkeit zur Validierung von neuronalen Netzen ist die Darstellung und Interpretation der algebraischen Funktion eines Modells. Hierdurch kann die „Black-Box“ des neuronalen Netzes aufgebrochen werden und die Entscheidungen des Modells in einer für den Menschen verständlichen Form dargestellt werden [24]. Das Ergebnis ist ein nachvollziehbares und damit validierbares Verhalten des neuronalen Netzes. Wie in Absatz 2.1.1 beschrieben, setzt sich die Formel eines Modells aus einer Vielzahl an untergeordneten Formeln zusammen, die alle über die Netztopologie, die Gewichte der einzelnen Neuronen und die einzelnen Übertragungs- bzw. Aktivierungsfunktionen definiert werden.

2.2. Verfügbare Frameworks

Die in der Einleitung der Arbeit beschriebene weite Verbreitung und aktuelle Beliebtheit von neuronalen Netzen geht mitunter auf die Existenz von leistungsfähigen Frameworks zurück. Diese stellen einfache Schnittstellen in verbreiteten Programmiersprachen zur Verfügung, um neuronale Netze zu erstellen, zu trainieren, zu evaluieren und schlussendlich produktiv einzusetzen. Zusätzlich erlauben die Frameworks eine weitgehend hardwareunabhängige Programmierung und ermöglichen den Einsatz und die Portierung auf besonders optimierte Hardwareplattformen wie GPUs oder TPUs. Die Frameworks stellen neben Werkzeugen zum Monitoring, Debugging und der Datenvorverarbeitung einen erweiterbaren Baukasten an Modulen zur Definition von

neuronalen Netzen zur Verfügung. Diese umfassen die in Absatz 2.1.1 beschriebenen Layerarten, populäre Aktivierungsfunktionen, Algorithmen zur Umsetzung des in Absatz 2.1.2 beschriebenen Backpropagationverfahrens inklusive verschiedener Optimierungsverfahren und Implementierungen unterschiedlicher Initialisierungsalgorithmen und Loss-Funktionen. Durch die große Anzahl an Modulen und der Erweiterbarkeit können diese Frameworks als Basis für neuronale Netze bei vielen Problemstellungen eingesetzt werden.
Die meisten Frameworks sind als Open-Source-Programme verfügbar und wurden entweder von den großen IT-Konzernen (Google, Microsoft, Facebook) oder akademischen Institutionen entwickelt und bereitgestellt. Die derzeit populärsten Frameworks sind „Pytorch“ und „Tensorflow“ (vgl. He [15]). Diese werden in den folgenden Abschnitten vorgestellt.

2.2.1. Pytorch

Das Framework verfügt über ein vollständiges Interface in Python und wurde ursprünglich von Facebook entwickelt. Es wird als Open-Source-Projekt unter einer BSD-Lizenz weiterentwickelt. Das Ziel der Entwickler ist eine hohe Anwendungsfreundlichkeit für Wissenschaftler und Progammierer. Um dieses Ziel zu erreichen, wird auf geringe Leistungsgewinne zulasten einer erhöhten Komplexität bewusst verzichtet. Zusätzlich lehnt sich das Framework semantisch an thematisch benachbarte Frameworks wie Numpy[1], Pandas[2] und Scipy[3] an und entspricht den generellen Designrichtlinien für Pythonprogramme (es ist also „Pythonic“). Zusätzlich bietet PyTorch vielerlei Schnittstellen zu den genannten Frameworks an. Die durch Pytorch erzeugten neuronalen Netze können zur Runtime dynamisch verändert werden. Hierdurch unterscheidet sich das Framework von anderen Frameworks (vgl. z.B. Absatz 2.2.2), welche vor der Trainingsphase einen statischen Graphen definieren und keine oder nur wenige Anpassungen während dem Training oder der Verwendung des Modells ermöglichen. Dies ermöglicht die Nutzung herkömmliche Werkzeuge zum Debuggen von Pythoncode. PyTorch verfügt über effiziente Algorithmen zur Verarbeitung von Tensoren und zur automatischen Berechnung von Gradienten. Die einzelnen Module des Baukastens liegen als Pythonklassen vor. Der Kern der einzelnen Funktionen ist in der Programmiersprache C++ geschrieben und separiert den sogenannten „Controlflow“ vom Datenfluss (englisch „Dataflow“). Der Controlflow steuert das Verhalten des Modells, während der Dataflow die Operationen des neuronalen Netzes auf die Daten des Datensatzes anwendet.
Das Framework ist weitestgehend plattformunabhägig und unterstützt GPUs (vgl. Abschnitt2.1.3) zur hochparallelisierten Verarbeitung. Hierbei stützt sich Pytorch auf das „CUDA“ Framework von Nvidia zur assynchronen Verarbeitung der Tensoren im Dataflow [28].

2.2.2. Tensorflow

Analog zu Pytorch handelt es sich bei Tensorflow um ein Open-Source-Framework zum Erzeugen, Trainieren und Nutzen von neuronalen Netzen mit einem besonderen Fokus auf DNNs (vgl. Absatz 2.1.1). Das Framework wurde ursprünglich von Google entwickelt und im Jahr 2015 vorgestellt [2]. Ziel des Frameworks ist es, zum einen die Programmierung von neuronalen Netzen zu vereinfachen und zum anderen das Training und die produktive Verwendung unabhängig von der verwendeten Hardware zu machen. Der einfache Zugang zur Nutzung von neuronalen

[1] https://github.com/numpy/numpy
[2] https://github.com/pandas-dev/pandas/
[3] https://github.com/scipy/scipy

Netzen wird sowohl durch die Verwendung der populären Programmiersprachen Python und C++ auf Nutzerebene als auch durch Bereitstellung von Tensorboard erreicht. Letzteres ist ein Webinterface zur Visualisierung der Trainingsmetriken und Netzparameter. Tensorflow ist auf einer Vielzahl von Hardwareplattformen lauffähig. Diese umfassen neben Smartphones (iOS, Android), Desktop-Workstations mit keiner, einer oder mehreren GPUs auch Konfigurationen aus mehreren Servern in einem Rechenzentrum.
Tensorflow erzeugt einen gerichteten Graphen aus Knoten und Kanten, durch den die Daten verarbeitet werden. Jeder Knoten kann genau eine Berechnung (engl. Operation) ausführen und kann mittels Ein- und Ausgängen über die Kanten mit anderen Knoten verbunden werden. Neben der Verarbeitung der Daten verfügen manche Knoten über einen eigenen definierten Zustand (engl. state). Neben den Knoten zur Verarbeitung der Daten existieren noch sogenannte control nodes. Diese steuern die Art der Datenverarbeitung ohne direkten Zugriff auf die zu verarbeitenden Daten und können mittels spezieller Kanten verbunden sein. Die verarbeitenden Daten werden als Tensoren bezeichnet und bestehen aus n-dimensionalen Matrizen mit einem einheitlichen Datentyp wie integer, float, double, o.ä. . Die Operations der Nodes werden bei der Erstellung des Graphen definiert. Tensorflow verfügt für die unterstützten Hardwareplattformen über spezialisierte Implementierungen dieser Knoten. Eine solche spezifische Implementierung wird als Kernel bezeichnet. Ein Tensorflowsystem besteht aus einem Client, einem Master und einem oder mehreren Workern. Der Client initialisiert und überwacht die Datenverarbeitung durch den erzeugten Graphen, während der Master die anfallenden Operations auf die Worker verteilt. Die Worker verfügen jeweils über einen oder mehrere (Grafik-, Tensor-) Prozessoren zur Ausführung der Operations. Ein solcher Prozessor wird als Device bezeichnet. Die Worker sind hierbei selbst für die Nutzung der zur Verfügung stehenden Hardwareresourcen und die Einhaltung der sich daraus ergebenden Beschränkungen (z.B. verfügbarer Speicher) verantwortlich. Die Verteilung der auszuführenden Operations auf die einzelnen Devices geschieht durch den Master. Nach der Verteilung werden innerhalb eines Devices Subgraphen gebildet. Die Kommunikation zwischen den Devices, bzw. die Verknüpfung dieser Subgraphen geschieht durch spezielle Send- und Receivenodes die, wie in Abbildung 2.20 dargestellt, die als Kreise visualisierten Operations verbinden.

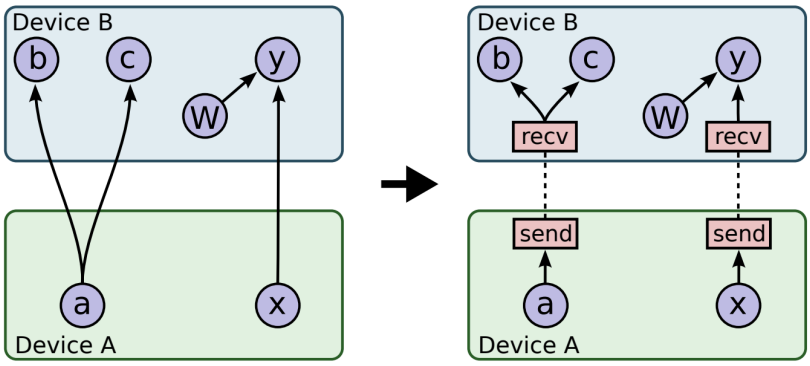

Abbildung 2.20.: Verteilung der Operations auf Devices [2]

Die Synchronisierung erfolgt ebenfalls durch die Send- und Receivenodes.

Neben der optimierten Verteilung der Operations auf die einzelnen Devices minimiert Tensorflow die Laufzeit durch parallele Verarbeitung der Datenpunkte innerhalb eines (Mini-) Batches (vgl. Absatz 2.1.2).
Die Implementierung der linearen Algebra Operations basiert auf der Open-Source-Bibliothek Eigen [14]. Seit Version 2.0 [1] unterstützt Tensorflow Keras [7] nativ als Benutzerschnittstelle und erlaubt analog zu PyTorch (vgl. Absatz 2.2.1) optional die Ausführung und das Debugging als normales Pythonprogramm statt der Definition und Ausführung eines Graphen. Des Weiteren wurden mit Tensorflow light und Tensorflow.js spezielle Versionen für embedded Geräte und Webservices ergänzt.

2.3. Dimensionsanalyse

Physikalische Größen werden gemäß Formel 2.28 als Produkt eines Zahlenwertes $\{x_i\}$ und einer Einheit $[x_i]$ beschrieben [10].

$$x_i = \{x_i\} \cdot [x_i] \tag{2.28}$$

Diese Einheiten setzen sich aus den sieben SI-Einheiten (französisch. Système International d'Unités, abgekürzt SI) zusammen. Jeder SI-Einheit ist eine eigene Dimension zugordnet. Die sieben SI-Einheiten und deren Dimensionen sind in Tabelle 2.1 dargestellt. Aus der Menge der

Einheit	Einheitenzeichen	Größe/Dimension
Meter	m	Länge
Kilogramm	kg	Masse
Sekunde	s	Zeit
Ampere	A	elektrische Stromstärke
Kelvin	K	thermodynamische Temperatur
Mol	mol	Stoffmenge
Candela	cd	Lichtstärke

Tabelle 2.1.: SI-Basiseinheiten

Basisdimensionen lässt sich ein Basissystem definieren. Für eine aus den Basiseinheiten abgeleitete Größe X ergibt sich die Einheit x_i nach Formel 2.29 als Potenzprodukt der Basiseinheiten x_j mit der Potenz γ_j und einer reelen dimensionslosen Konstanten C_j [5].

$$x_i = C_j \cdot \prod x_j^{\gamma_j} \tag{2.29}$$

Die physikalische Dimension D_i der Größe x_i ergibt sich analog aus dem Potenzprodukt der Basisdimensionen D_j ohne den dimensionslosen Vorfaktor (vgl Formel 2.30) [13].

$$D_i = \prod D_j^{\gamma_j} \tag{2.30}$$

Für eine dimensionslose Größe gilt $D_i = 1$. Der vektorielle Dimensionsraum einer physikalischen Funktion wird aus den voneinander unabhängigen Variablen gebildet. Größen mit gleichen Dimensionen werden als dimensionshomogen bezeichnet. Physikalische Funktionen, deren additive Terme nicht dimensionshomogen sind, sind a priori fehlerhaft [13].

2.3.1. Buckinghamsches Pi-Theorem

Wie in Gleichung 2.31 dargestellt, können physikalische Gleichungen mit den dimensionsbehafteten Größen x_i in Funktionen der dimensionslosen Potenzprodukte π_k transformiert werden.

$$f(x_1, ..., x_i) = f(\pi_1, ..., \pi_k) \tag{2.31}$$

Die dimensionslosen Potenzprodukte ergeben sich nach Formel 2.32, wobei n die Anzahl der dimenesionsbehafteten Größen ist.

$$\pi_j = \prod_{i=1}^{n} x_i^{\gamma_{ij}} \tag{2.32}$$

Berechnen lassen sich diese dimensionslosen Größen mittels der Dimensionsmatrix. In dieser werden die Dimensionen D_k über den vorkommenden dimensionsbehafteten Größen x_i aufgetragen. Nach dem Herstellen einer Diagonalform und dem Streichen eventuell auftretender Nullspalten ergeben sich die dimensionslosen Potenzprodukte π_j durch Formel 2.33.

$$\pi_j = x_{j+r} \cdot \prod_{i=1}^{r} x_i^{-a_{ij}} \tag{2.33}$$

r gibt hierbei den Rang der Dimensionsmatrix an und für j gilt $1 \leq j \leq (n-r)$. Diese Umformung ist in Abbildung 2.21 ersichtlich [32]. Die Variable n beschreibt die Anzahl der Variablen in der Relevanzliste.

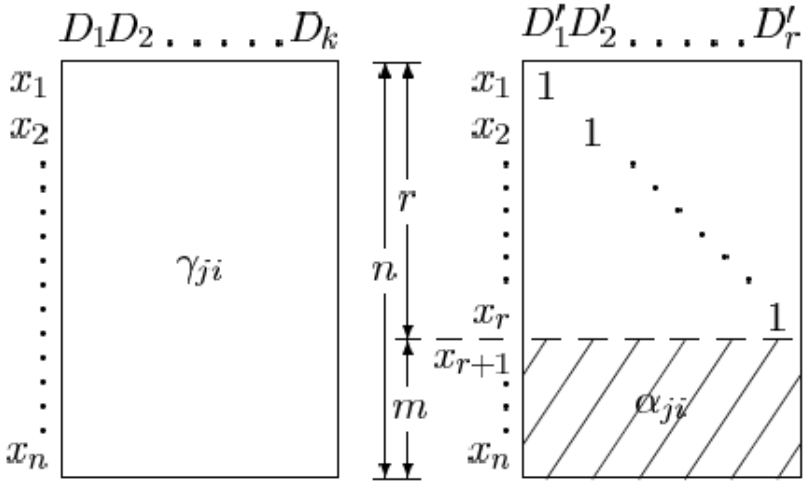

Abbildung 2.21.: Dimensionsmatrix mit Berechnungsschema [32]

Hieraus lässt sich das Buckinghamsche Pi-Theorem [5] ableiten. Dieses besagt, dass zu jeder vollständig dimensionshomogenen Beziehung f von n dimensionsbehafteten Größen $x_i \in \mathbb{R}^+$ auch eine dimensionslose Beziehung G von m dimensionslosen Potenzprodukten $\pi_j \in \mathbb{R}^+$ gemäß Formel 2.34 existiert.

$$f(x_1, ..., x_n) = G(\pi_1, ..., \pi_m) \tag{2.34}$$

Wobei die $m = n - r$ dimensionslosen Potenzprodukte π_j ein Fundamentalsystem $\Pi_m = \{\pi_1, ..., \pi_m\}$ bilden, das aus der Relevanzliste $(x_1, ..., x_n)$ der n physikalischen Größen gebildeten Dimensionsmatrix mit Rang r in Abbildung 2.21 bestimmt wird. Dabei gilt $i \in \{\mathbb{N}^+ | i = 1, ..., r\}, j \in \{\mathbb{N}^+ | i = 1, ..., m\}$. Durch die Anwendung des Buckinghamschen Pi-Theorems auf eine physikalische Funktion kann es also zu einer Reduzierung der Relevanzliste um den Rang der Matrix r kommen.
Das erzeugte Fundamentalsystem kann mittels Formel 2.35 in beliebig viele mathematisch gleichwertige Fundamentalsysteme Π' transformiert werden [32].

$$\pi'_j = \prod_{i=1}^{m} \pi_i^{\beta_{ji}} \tag{2.35}$$

2.3.2. Vollähnlichkeit

Fallen verschiedene Datenpunkte $X(x_1, ..., x_n)$ in einem Fundamentalsystem Π aufeinander, so wird von vollähnlichen Datenpunkten gesprochen. Die dimensionslosen Kennzahlen π_m sind über diese Datenpunkte hinweg konstant. Durch die Variation eines gemessenen originalen Datenpunktes können bei konstanten dimensionslosen Kennzahlen beliebig viele zusätzliche vollähnliche Datenpunkte gemäß Abbildung 2.22 gebildet werden.

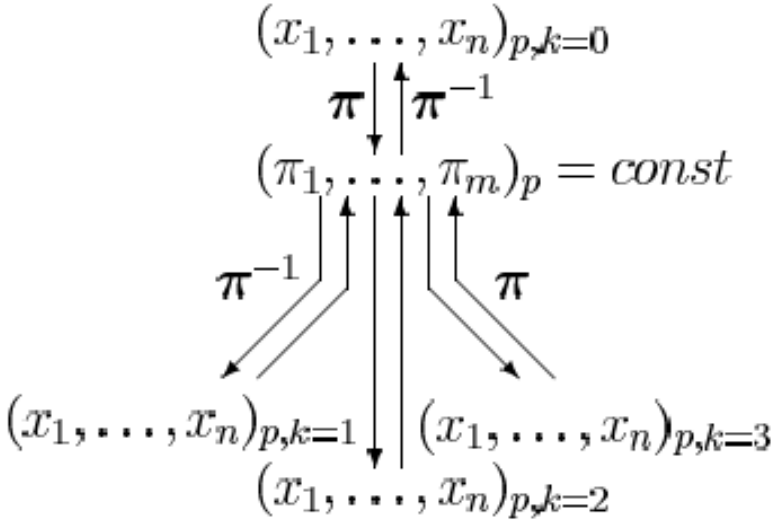

Abbildung 2.22.: Datenvervielfachung mit k vollähnlichen Datenpunkten [32]

2.4. Dimensionshomogene neuronale Netze

Die in Abschnitt 2.2 beschriebenen Frameworks verfügen über keine Algorithmen zur Prüfung und Sicherstellung der dimensionshomogenen Verarbeitung der Datenpunkte. Die Sicherstellung der Dimensionshomogenität innerhalb des neuronalen Netzes muss aus diesem Grund durch den Nutzer gewährleistet werden. Hierfür können unter anderem die dimensionsbehafteten Größen in dimensionslose Größen gemäß dem Pi-Theorem 2.3.1 transfomiert werden. Für ein neuronales Netz zur Approximation von physikalischen Funktionen mit dimensionsbehafteten Eingangsgrößen ergibt sich nach Rudolph [34] eine teilweise a priori definierte Netztopologie. Diese ist in Abbildung 2.23 dargestellt.

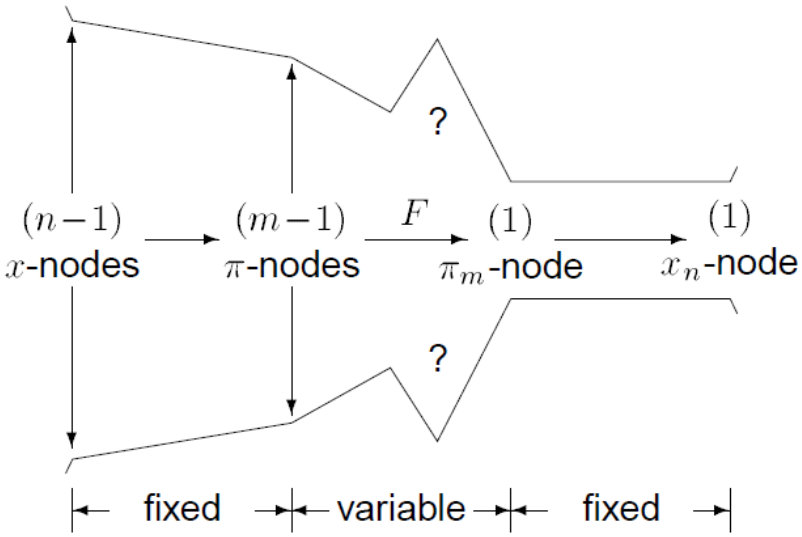

Abbildung 2.23.: Darstellung dimensionshomogene Netztopologie [34]

Der Inputlayer enthält alle dimensionsbehafteten Features der Relevanzliste (dargestellt als x-nodes bzw. Neuronen). Der Outputlayer enthält die ebenfalls dimensionsbehaftete gesuchte Größe bzw. das Label des Datenpunktes. Der erste Layer in Netzrichtung mit den $(m-1)\pi$-Nodes verfügt über konstante Gewichte a_{ij} (siehe Abbildung 2.21) der Gestalt, dass die Dimensionslosigkeit der einzelnen π-Nodes aus dem Inputlayer folgt. Der vorletzte Layer, in der Abbildung 2.23, als π_m-node dargestellt, wird im Gegensatz dazu erst durch die Kombination der Dimensionen der Werte des Inputlayers und des Outputlayers dimensionslos. Es handelt sich also bei dem präsentierten neuronalen Netz um ein Netz mit einer Residual-Connection und kein rein sequentielles Netz. In Abbildung 2.24 ist ein solches dimensionshomogenes Netz beispielhaft zur Abbildung eines Monoms dargestellt.

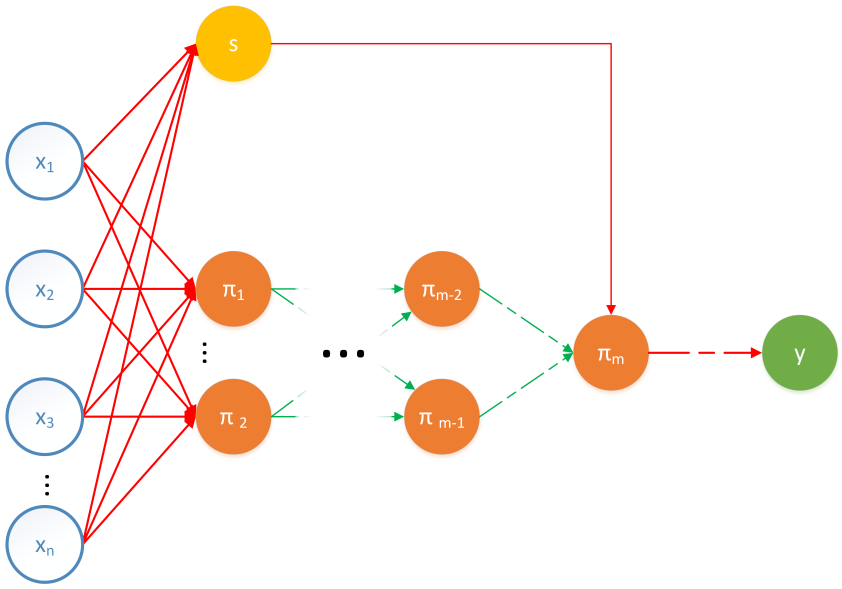

Abbildung 2.24.: Dimensionshomogenes Neuronales Netz

Die rot markierten Verbindungen zeigen die dimensionsbehafteten Verbindungen zwischen den einzelnen Neuronen an, während die grün markierten Verbindungen die dimensionslosen Verbindungen verdeutlichen. Die durchgezogenen Linien stehen für Verbindungen mit konstanten Gewichten, die Unterbrochenen hingegen für Verbindungen mit variablen bzw. erlernbaren Gewichten (vgl. Abschnitt 2.1.1). Die Neuronen des Beispiels verfügen über keine Bias. Das gelb markierte „Residual Connection“-Neuron wird lediglich zur besseren Verständlichkeit der folgenden symbolischen Herleitung benötigt.
Für ein beliebiges Monom $f(x_1, ..., x_n)$ mit dimensionsbehafteten Größen x_i existiert eine Form analog zu Formel 2.36.

$$f(x_1, ..., x_n) = k_0 \cdot \prod_{i=1}^{n} x_i^{k_i} = 1 \tag{2.36}$$

Gemäß dem Buckinghamschen-Pi-Theorem [5] existiert eine weitere Funktion $F(\pi_1, ..., \pi_m) = 1$ mit dimensionslosen Größen π_i und der Konstanten C_0. Hierraus ergibt sich Formel 2.37.

$$F(\pi_1, ..., \pi_m) = C_0 \cdot \prod_{i=1}^{m} \pi_i^{k_i} = 1 \tag{2.37}$$

Durch die Substitution 2.38 analog zu Formel 2.33 ergibt sich Formel 2.39.

$$\pi_m = x_n \prod_{i=1}^{r} x_i^{-a_{ni}} \tag{2.38}$$

$$F(\pi_1, ..., \pi_m) = c_0 \cdot x_n \prod_{i=1}^{r} x_i^{-a_{ni}} \cdot \prod_{j=1}^{m-1} \pi_j^{k_j} = 1 \tag{2.39}$$

Umgestellt gemäß der gesuchten dimensionsbehafteten Größe ergibt sich Formel 2.40.

$$c_0^{-1} \cdot \prod_{i=1}^{r} x_i^{a_{ni}} \cdot (G(\pi_1, ..., \pi_{m-1}))^{-1} = x_n \tag{2.40}$$

Der konstante Koeffizient dieser Formel c_0^{-1} kann durch eine Konstante k_0 ersetzt werden. Der Term $\prod_{i=1}^{r} x_i^{a_{ni}}$ beschreibt die Zusammensetzung des gelben Knotens (der residual Connection) in Abbildung 2.24. Die im Term vorkommenden Potenzen a_{ni} sind bereits durch die Dimensionsmatrix bestimmt (vgl. Abbildung 2.21). Variabel und damit vom neuronalen Netz im Training zu bestimmen sind lediglich noch die Konstante k_0 und die Potenzen k_j aus Gleichung 2.39.
Eine beispielhafte Realisierung eines solchen neuronalen Netzes ist in Abbildung 2.25 für die Balkenbiegeformel 2.41 dargestellt, wobei der Koeffizient k_0 als dimmensionsloses Biasneuron v_0 realisiert wurde.

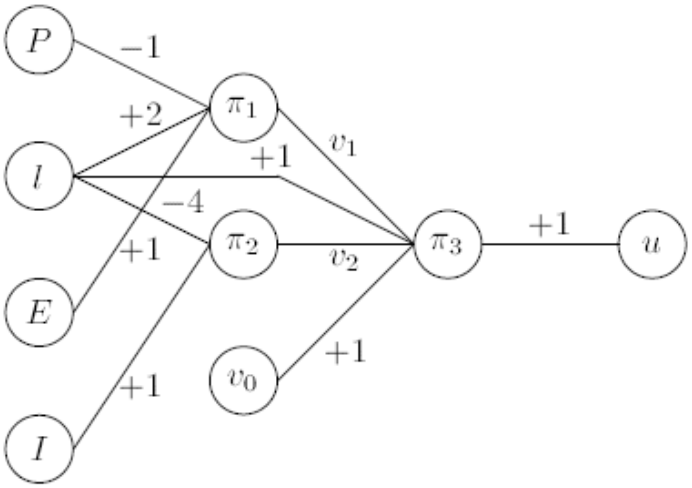

Abbildung 2.25.: Dimensionshomogenes Neuronales Netz zur Abbildung der Balkenbiegefunktion [34]

$$u = \frac{1}{3} \frac{Pl^3}{EI} \tag{2.41}$$

Die Eigenschaften von Polynomen und deren Auswirkungen auf die Netztopologien lassen sich aus den Monom-Eigenschaften ableiten. Bei den Termen des Polynoms handelt es sich um einzelne Monome. Der Graph eines Polynoms enthält demzufolge parallel angeordnete Subgraphen, die den jeweiligen Monomen entsprechen. Kombiniert werden diese Subgraphen im Outputlayer (siehe Abbildung 2.26).

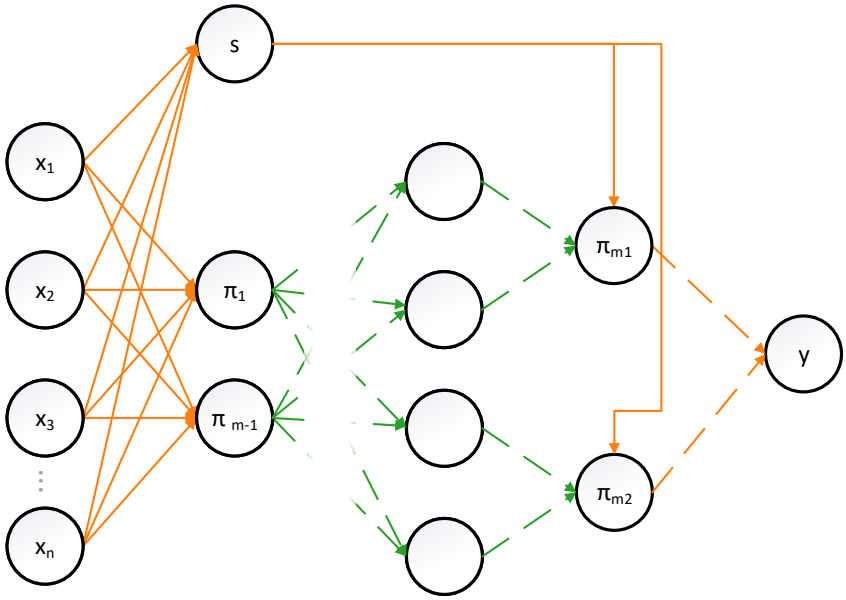

Abbildung 2.26.: Dimensionshomogenes Neuronales Netz zur Abbildung von Polynomen

Da nach Absatz 2.3 nicht dimensionshomogene additive Terme immer fehlerhaft sind, sind auch die Neuronen im letzten hidden Layer (vgl. Abschnitt 2.1.1) dimensionshomogen.

3. Methodik

Zur Untersuchung der möglichen Nutzung der Dimensionsinformation in neuronalen Netzen werden drei Experimente durchgeführt. Die Ergebnisse der Experimente werden quantitativ ausgewertet und induktiv interpretiert. Die Daten werden durch die Untersuchung der Genauigkeit und Trainierbarkeit von neuronalen Netzen erhoben. Hierfür wird auf Basis von Tensorflow (vgl. Abschnitt 2.2.2) ein Framework zum Erzeugen von dimensionshomogenen neuronalen Netzen und vollähnlichen Datenpunkten konzeptioniert und implementiert.
Im ersten Abschnitt des Kapitels werden die drei Experimente vorgestellt. Der zweite Abschnitt beinhaltet die Anforderungen an das Framework und das daraus abgeleitete Konzept zur Umsetzung als Programm. Im dritten Abschnitt ist die tatsächliche Implementierung des Konzepts dokumentiert. Im vierten Abschnitt werden auf Basis der Implementierung die Umgebungsparameter zur Durchführung der Experimente beschrieben. Die Validierung der Experimente erfolgt im letzten Abschnitt.
In allen Experimenten sollen die Lernfähigkeit und Genauigkeit der erzeugten Neuronalen Netze untersucht werden. Hierfür wird nach Abschluss jeder Epoche die Loss-Funktion für die Trainings- und Validierungsdaten ausgewertet und zur Auswertung gespeichert. Diese Kennzahlen werden hinsichtlich Genauigkeit und Trainierbarkeit des neuronalen Netzes ausgewertet.

3.1. Entwicklung der Experimente

In diesem Abschnitt wird der Aufbau der Experimente vorgestellt. Hierzu gehört neben der Zielsetzung des jeweiligen Experiments auch die Definition der abhängigen bzw. unabhängigen Variablen. Die verwendeten dimensionsbehafteten physikalischen Formeln werden ebenfalls eingeführt und beschrieben. Die Beschreibung der tatsächlichen Umsetzung und Durchführung der Experimente erfolgt in Abschnitt 3.4.
Im ersten Experiment soll die Bedeutung des gewählten Fundamentalsystems für ein dimensionshomogenes neuronales Netz untersucht werden. Im zweiten Experiment wird die Auswirkung von künstlich erzeugten vollähnlichen Datenpunkten auf das Trainingsverhalten eines neuronalen Netzes mit der Wirkung zusätzlicher realer bzw. pseudorealer Datenpunkten und dem Verhalten von neuronaler Netze ohne zusätzliche Datenpunkte verglichen. Im dritten Experiment wird die Auswirkung von zusätzlichen vollähnlichen Datenpunkten auf ein nicht zwangsläufig dimensionshomogenes neuronales Netz mit einem dimensionshomogenen neuronalen Netz verglichen.

3.1.1. Vergleich verschiedener Fundamentalsysteme

In diesem Versuch sollen die Auswirkungen der Wahl eines Fundamentalsystems (vgl. Absatz 2.3.1 und Gleichungen 2.35) auf die Trainierbarkeit und Genauigkeit untersucht werden. Hierzu soll die vereinfachte Balkenbiegeformel 3.1 durch ein dimensionshomogenes neuronales Netz

gemäß Abschnitt 2.4 approximiert werden.

$$u_{Balken} = \frac{1}{3} \frac{P_{Balken} L_{Balken}^3}{E_{Balken} I_{Balken}} \tag{3.1}$$

In dieser Formel wird die Auslenkung u_{Balken} eines einseitig eingespannten Balkens mit der Länge L_{Balken} unter der Krafteinwirkung P_{Balken} berechnet. E_{Balken} gibt das Elastizitätsmodul und I_{Balken} das Flächenträgheitsmoment an. Es werden keine zusätzlichen vollähnlichen Datenpunkte zur Vergrößerung des Trainings- und Testdatensatzes verwendet. Die Topologie des neuronalen Netzes soll unter Beachtung des im folgenden Abschnitt definierten Frameworks die zur exakten Nachbildung der Formel 3.1 erforderliche minimale Form besitzen.
Die unabhängige Variable des Versuchs stellt das gewählte Fundamentalsystem Π nach Absatz 2.3.1 dar. Die abhängigen Größen sind die Werte der Loss-Funktionen der Trainings- und Validierungsdaten über die Trainingsepochen.
Die verschiedenen Fundamentalsysteme werden durch die Variation der Reihenfolge der x_i aus Abbildung 2.21 erzeugt. Lediglich die Position der gesuchten Größe u_{Balken} bleibt konstant. Es ergeben sich somit nach Formel 3.2 bis zu 24 verschiedene Fundamentalsysteme.

$$n_\Pi =!(n-1) \tag{3.2}$$

Variable n_Π beschreibt die Anzahl der maximal möglichen Fundamentalsysteme, während n der Anzahl der Einträge in der Relevanzliste entspricht. Für die dimensionsbehafteten Größen aus Gleichung 3.1 ergibt sich die Dimensionsmatrix 3.1.

	E_{Balken}	I_{Balken}	L_{Balken}	P_{Balken}	u_{Balken}
L_{Dim}	-1	4	1	1	1
M_{Dim}	1	0	0	1	0
T_{Dim}	-2	0	0	-2	0

Tabelle 3.1.: Dimensionsmatrix Balkenbiegung

Für jedes Fundamentalsystem wird nach Abschnitt 2.4 ein dimensionshomogenes neuronales Netz erzeugt und zufällig initialisiert. Das Netz wird auf Basis des Frameworks aus Abschnitt 3.3 erzeugt. Das Training wird nach 50 Epochen abgebrochen und als Loss-Funktion wird die „Mean-Absolute-Percentage-Error“(MAPE)-Funktion verwendet. Es sollen jeweils insgesamt 3000 Datenpunkte zufällig erzeugt werden. Gemäß Anhang A.4 werden die zufällig erzeugten Datensätze als einheitlicher Datensatz angenommen.

3.1.2. Vervielfachte vollähnliche Datenpunkte

Der zweite Versuch soll die Auswirkungen von künstlichen zusätzlichen Datenpunkten im Datensatz sichtbar machen. Hierfür soll analog dem ersten Versuch die in Gleichung 3.1 vorgestellte vereinfachte Balkenbiegeformel wiederum approximiert werden. Die Topologie des erzeugten Netzes entspricht ebenfalls der des ersten Versuchs. Es handelt sich jedoch um kein dimensionshomogenes Netz, da die Gewichte des ersten Layers entgegen dem vorherigen Versuch zufällig initialisiert und nicht aus der Dimensionsmatrix ermittelt werden.
Die unabhängigen Variablen sind die Anzahl und Typen der Datenpunkte im Datensatz. Die

abhängigen Größen sind wiederum die Werte der Loss-Funktionen der Trainings- und Validierungsdaten über die Trainingsepochen.
In einem ersten Schritt wird die Größe des Vergleichsdatensatzes ermittelt. Hierfür wird zunächst eine Basisgröße festgelegt. Mit dem erzeugten Datensatz werden fünf neuronale Netze trainiert und evaluiert. Unterschreiten von diesen fünf Netzen weniger als zwei die Loss-Schwelle von 20 % innerhalb von 3000 Epochen, so wird der Umfang des Vergleichsdatensatzes verdoppelt; andernfalls wird die Suche nach der Größe des Vergleichsdatensatzes abgebrochen und die aktuelle Größe als Konstante gespeichert. Als Basisgröße wird für das Experiment $n = 25$ gewählt.
Dieser erste Schritt dient zur experimentellen Ermittlung der durch den Adam-Algorithmus voneinander abhängigen Größen der maximalen Epochenanzahl und der Größe des Datensatzes. Für dieses Experiment wurde die maximale Anzahl der Epochen als unabhänngige Variable und die Größe des Datensatzes als abhängige Variable festgelegt. Die Festlegung der maximalen Anzahl erfolgte auf Basis der Erfahrungen, die bei der Programmierung des Netzgenerators hinsichtlich Laufzeit und Konvergenz der Loss-Funktion gemacht wurden. Die Definition der Loss-Schwelle erfolgte mit dem Ziel, die generelle Trainierbarkeit des Modells unter den genannten Vorgaben sicherzustellen und gleichzeitig eine Varianz innerhalb der Ergebnisse zu erlauben. Durch die Anforderung der wiederholten Unterschreitung der Loss-Schwelle wird das Risiko eines zu hohen Einflusses eines einzelnen für das Netz besonders günstigen Datensatzes auf das Experiment minimiert. Die Anzahl der maximalen Epochen ergab sich aus den während der Entwicklung des Netzgenerators gewonnenen Erfahrungen. Innerhalb des Limits konnte für jede Problemstellung ein Modell erfolgreich trainiert werden.
Nach Bestimmung der Größe des Vergleichsdatensatzes wird das eigentliche Experiment in fünf Schritten durchgeführt.
Die Anzahl der maximal zu durchlaufenden Epochen wird aus Laufzeitgründen auf 5000 festgelegt. Das Training wird außerdem bei dauerhaftem Unterschreiten der Loss-Schwelle von 0,1% frühzeitig aus Laufzeitgründen abgebrochen. Mittels des in Abschnitt 3.3 beschriebenen Programms wurden für den Vergleichsdatensatz insgesamt sieben neuronale Netze trainiert. Im zweiten Schritt werden die Datenpunkte des Vergleichsdatensatzes durch vollähnliche Datenpunkte künstlich verzehnfacht und im dritten Schritt verhundertfacht (vgl. Abschnitt 2.3.2, insbesondere Abbildung 2.22).
Zu Vergleichszwecken wird im vierten und fünften Schritt die zehn- bzw. hundertfache Anzahl an realen Datenpunkten des Vergleichsdatensatzes erzeugt und zum Training des neuronalen Netzes verwendet.
Für Datensätze mit der zehnfachen Größe des Vergleichsdatensatzes werden ebenfalls jeweils sieben neuronale Netze trainiert und evaluiert. Für Datensätze mit der hundertfachen Größe des Vergleichsdatensatzes werden aus Laufzeitgründen lediglich jeweils drei neuronale Netze erzeugt. Für jedes neuronale Netz werden die Gewichte zufällig initialisiert und es wird ein neuer unbekannter Datensatz erstellt.

3.1.3. Vergleich vervielfachter Datenpunkte mit dimensionshomogenen Netzen

Im dritten Experiment sollen die Auswirkungen auf die Güte eines neuronalen Netzes untersucht werden, die durch die Nutzung der Dimensionsinformationen der dimensionsbehafteten Größen verursacht werden. Hierfür wird für ein neuronales Netz mit identischer Topologie die Dimensionsinformation zum einen zur künstlichen Vervielfachung der vorhandenen Datensätze

mittels des Prinzips der Vollähnlichkeit und zum anderen zur Erstellung dimensionshomogener neuronaler Netzen genutzt. Als Referenz wird zudem ein neuronales Netz ohne Beachtung der zusätzlichen Informationen aus den Dimensionen trainiert und validiert.
Darüber hinaus wird das konzeptionierte und implementierte Framework anhand der verschiedenartigen Versuchsformeln validiert.
Die unabhängige Variable des Experiments ist die Art der Nutzung der Dimensionsinformationen. Die abhängigen Variablen sind wiederum die Werte der Loss-Funktionen der Trainings- und Validierungsdaten über die Trainingsepochen. Auch in diesem Experiment soll die ausgesuchte physikalische Formel mit dimensionsbehafteten Größen durch das neuronale Netz approximiert werden.
Analog zu dem im vorherigen Versuch definierten Verfahren wird zunächst die benötigte Größe des Referenzdatensatzes ermittelt und es werden jeweils sieben neuronale Netze für jede Art der Dimensionsinformationsnutzung erzeugt. Die Limitierung auf jeweils sieben Durchläufe ist durch die für das Training der neuronalen Netze benötigte Zeit zu begründen. Die Gewichte werden ebenfalls für jedes neuronale Netz zufällig initialisiert und ein neuer Trainings-, Validierungs- und Evaluierungsdatensatz erzeugt. Die realen Datenpunkte werden in den entsprechenden Versuchen um den Faktor zehn durch vollähnliche Datenpunkte vervielfacht.
Neben der vereinfachten Biegebalkenformel als Monominalfunktion, 3.1 werden die Formel des senkrechten Wurfs, die Fermi-Verteilung, das Zerfallsgesetz, die Rapiditätsgleichung, die Formel des Federpendels und die Wärmeübergangsgleichung als Ziel- und Basisfunktionen genutzt.

Senkrechter Wurf

Formel 3.3 zur Berechnung der Höhe h_{Wurf} des senkrechten Wurfs nach oben in Abhängigkeit von der Anfangsgeschwindigkeit $v_{0,Wurf}$, der Fallbeschleunigung g_{Wurf} und der Zeit t_{Wurf} bildet ein Polynom ab.

$$h_{Wurf} = v_{0,Wurf} \cdot t_{Wurf} - \frac{1}{2} \cdot g_{Wurf} \cdot t_{Wurf}^2 \tag{3.3}$$

Die Dimensionsmatrix der dimensionsbehafteten Größen ist in Tabelle 3.2 dargestellt.

	$v_{0,Wurf}$	t_{Wurf}	g_{Wurf}	h_{Wurf}
L_{Dim}	1	0	1	1
M_{Dim}	0	0	0	0
T_{Dim}	-1	1	-2	0

Tabelle 3.2.: Dimensionsmatrix senkrechter Wurf

Da die Höhe des senkrechten Wurfs neben positiven auch negative Werte annehmen kann, wird als Loss-Funktion die „Mean-Squared-Error“(MSE)-Funktion verwendet.

Fermi-Verteilung

Die Fermi-Verteilung aus Gleichung 3.4 entstammt der Klasse der sigmoidalen Funktionen und bildet die dimensionslose Besetzungswahrscheinlichkeit W_{Fermi} in Abhängigkeit von der Energie E_{Fermi}, dem chemischen Potential μ_{Fermi}, der Boltzmann-Konstante k_{Fermi} und der Temperatur

T_{Fermi} ab.

$$W_{Fermi} = \frac{1}{e^{(\frac{E_{Fermi}-\mu_{Fermi}}{k_{Fermi}\cdot T_{Fermi}})}+1} \tag{3.4}$$

Es ergibt sich die Dimensionsmatrix 3.3.

	W_{Fermi}	μ_{Fermi}	E_{Fermi}	k_{Fermi}	T_{Fermi}
L_{Dim}	0	0	2	2	0
M_{Dim}	0	0	1	1	0
T_{Dim}	0	0	-2	-2	0
σ_{Dim}	0	0	0	-1	1

Tabelle 3.3.: Dimensionsmatrix Fermi-Verteilung

Die Verteilung W_{Fermi} nimmt für die Größe des Definitionsbereiches Werte nahe 0 an. Aus diesem Grund wird die MSE-Funktion als Loss-Funktion verwendet.

Zerfallsgesetz

Aus der Klasse der Expotentialfunktionen wird das Zerfallsgesetz aus der Kernphysik in Formel 3.5 als Versuchsfunktion herangezogen. Es gibt die dimensionslose Anzahl $N_{Zerfall}$ der Atomkerne über den Verlauf der Zeit $t_{Zerfall}$ bei einer Zerfallskonstanten $\lambda_{Zerfall}$ und einer Anzahl $N_{0,Zerfall}$ zu Beginn der Betrachtung an.

$$N_{Zerfall} = N_{0,Zerfall} \cdot e^{-\lambda_{Zerfall} t_{Zerfall}} \tag{3.5}$$

Die Dimensionsmatrix aller Größen aus der Relevanzliste ist in Tabelle 3.4 dargestellt.

	$N_{Zerfall}$	$N_{0,Zerfall}$	$\lambda_{Zerfall}$	$t_{Zerfall}$
L_{Dim}	0	0	0	0
M_{Dim}	0	0	0	0
T_{Dim}	0	0	-1	1

Tabelle 3.4.: Dimensionsmatrix Zerfallsgesetz

Aufgrund des Grenzwertes 0 der Formel 3.5 wird wiederum die MSE-Funktion als Loss-Funktion verwendet.

Rapidität

Die Formel der Rapidität θ_{Rap} 3.6 verknüpft als Hyperbelfunktion die Geschwindigkeit v_{Rap} mit der Lichtgeschwindigkeit c_{Rap}. Die MSE-Funktion wird als Loss-Funktion verwendet.

$$v_{Rap} = c_{Rap} \cdot \tanh \theta_{Rap} \tag{3.6}$$

In Tabelle 3.5 werden die physikalischen Größen der Variablen dargestellt.

	θ_{Rap}	v_{Rap}	c_{Rap}
L_{Dim}	0	1	1
M_{Dim}	0	0	0
T_{Dim}	0	−1	−1

Tabelle 3.5.: Dimensionsmatrix Rapidität

Federpendel

Aus der Klasse der trigonometrischen Funktionen wurde das Federpendel 3.7 als Versuchsfunktion ausgewählt.

$$y_{Feder} = y_{0,Feder} \cdot \cos(\sqrt{\frac{D_{Feder}}{m_{Feder}}} t_{Feder}) \tag{3.7}$$

Diese Funktion gibt die Auslenkung y_{Feder} in Abhängigkeit von der Auslenkung $y_{0,Feder}$ zum Zeitpunkt $t = 0$ an. Die Größe t_{Feder} gibt die vergangene Zeit, D_{Feder} die sogenannte Federkonstante und m_{Feder} die Masse des Versuchskörpers an. Da die Auslenkung y_{Feder} neben positiven auch negative Werte annehmen kann wird als Loss-funktion die MSE-Funktion verwendet.
Es resultiert die Dimensionsmatrix in Tabelle 3.6.

	y_{Feder}	t_{Feder}	$y_{0,Feder}$	D_{Feder}	m_{Feder}
L_{Dim}	1	0	1	0	0
M_{Dim}	0	0	0	1	1
T_{Dim}	0	1	0	−2	0

Tabelle 3.6.: Dimensionsmatrix Federpendel

Wärmeübergangsgleichung

Als letzte Versuchsgleichung wird die Wärmeübergangsgleichung eines Rohres 3.8 genutzt.

$$\dot{q} = 2\pi \frac{\lambda_{Wärme} \vartheta_{Wärme}}{\ln(\frac{r_{a,Wärme}}{r_{i,Wärme}})} \tag{3.8}$$

Diese logarithmische Funktion beschreibt den Zusammenhang zwischen dem Wärmefluss $\dot{q}_{Wärme}$, der Wärmeleitzahl $\lambda_{Wärme}$, dem Rohrinnenradius $r_{i,Wrme}$, dem Außenradius $r_{a,Wärme}$ und der Temperaturdifferenz $\vartheta_{Wärme}$, die sich gemäß Formel 3.9 bildet.

$$\vartheta = T_a - T_i \tag{3.9}$$

T_a und T_i beschreiben die Temperatur an der äußeren respektive inneren Wand des Rohres. Es gilt $r_{a,Wärme} > r_{i,Wärme}$ und somit Gleichung 3.10.

$$\frac{r_a}{r_i} > 1 \Rightarrow ln(\frac{r_a}{r_i}) > 0 \tag{3.10}$$

Die Dimensionsmatrix entspricht Tabelle 3.7 und als Loss-Funktion wird aufgrund von Formel 3.10 die MAPE-Funktion genutzt.

	$q_{Wärme}$	$\lambda_{Wärme}$	$\vartheta_{Wärme}$	$r_{a,Wärme}$	$r_{i,Wärme}$
L_{Dim}	1	1	0	1	1
M_{Dim}	1	1	0	0	0
T_{Dim}	-3	-3	0	0	0
σ_{Dim}	0	-1	1	0	0

Tabelle 3.7.: Dimensionsmatrix Wärmeübergangsgleichung

3.2. Konzeption eines graphenbasierten Frameworks

Um die Vielzahl der beschriebenen Experimente durchzuführen, wird ein Framework für neuronale Netze entwickelt. Diese soll sowohl die Dimensionsinformationen der dimensionsbehafteten Größen im Kontext neuronaler Netze verarbeiten können, als auch eine einfache Konfigurationsoberfläche zur Realisierung der Experimente bieten. Die detaillierten Anforderungen werden in diesem Abschnitt beschrieben.

3.2.1. Anforderungen

Um Underfitting (vgl. Abschnitt 2.1.4) a priori auszuschließen müssen die in Abschnitt 3.1 beschriebenen Formeln durch die erzeugten neuronalen Netze exakt abbildbar sein. Das zu erstellende Framework muss aus diesem Grund über alle in den Versuchsformeln verwendeten Operationen verfügen.
Beschreibung und Konfiguration sollen durch Konfigurationsdateien in einem offenen Format geschehen. Dies ermöglicht die Anbindung des Neuronale-Netze-Generators an weitere Programme und erlaubt die Erstellung verschiedener Versuchsfällen ohne Eingriffe in den Programmcode. Innerhalb der Konfigurationsdateien sollen die Netztopologie, die Loss-Funktion, der verwendete Optimierungsalgorithmus, die Anzahl der Trainingsepochen, etwaige zusätzliche aufzuzeichnende Metriken neben der Loss-Funktion, der Schwellwert der Loss-Funktion zum vorzeitigen Abbruch des Trainings, die Anzahl der Datenpunkte pro Batch, die prozentuale Verteilung des gesamten Datensatzes auf die Trainings-, Validierungs- und Testdaten, der Faktor zur Vervielfachung des Datensatzes durch vollähnliche Datenpunkte und die Einstellungen für das Logging zum Debugging definiert sein. Außerdem sollen durch die zentrale Konfigurationsdatei die realen vorhandenen Datenpunkte eingebunden werden, bzw. durch einen Datengenerator soll eine festgelegte Anzahl an Datenpunkten zufälig erzeugt werden. Die Dimensionsinformationen sollen in einer separaten Datei enthalten sein und ebenfalls durch die Konfigurationsdatei integriert werden.
Wie bei jeder Software soll die Laufzeit des Programmes minimal sein, bei maximaler Stabilität der Ausführung. Der Programmcode muss sowohl wart- als auch erweiterbar sein. Die Umsetzung soll daher in einer verbreiteten Programmiersprache und auf Basis von Frameworks mit hohem Dokumentationsstandard erfolgen.
Für die Auswertung der Experimente müssen die Daten der Loss-Funktion über den Trainingsverlauf gespeichert werden. Zur Validierung des neuronalen Netzes soll das Programm nach der Verarbeitung eines Batches aus dem Trainingsdatensatz mittels des Validierungsdatensatzes die Generalisierbarkeit des Modells bestimmen und diese Kennzahl analog dem Ergebnis der Loss-

Funktion für die Trainingsdaten speichern. Weiterhin soll die Netztopologie eines neuronalen Netzes grafisch ausgegeben werden können (vgl. Anhang A.2). Außerdem muss die aus der Topologie und den aktuellen Gewichten im Graphen abgebildete algebraische Formel im Modell in symbolischer Form exportiert werden können.
Auf Basis der in Abschnitt 3.1 beschriebenen Vergleichsformeln soll ein Datengenerator zufällige pseudoreale Datenpunkte generieren können. Aufwendige Versuchs- und Messreihen zum Erzeugen von Datensätzen werden somit nicht benötigt.
Für die eigentlichen Versuche muss das Programm sowohl aus der gegebenen Dimensionsmatrix bzw. dem abgeleiteten Fundamentalsystem dimensionshomogene neuronale Netze als auch vollähnliche Datenpunkte erzeugen können.

3.2.2. Konzept

Aus den Anforderungen leitet sich das in Abbildung 3.1 dargestellte Konzept ab.

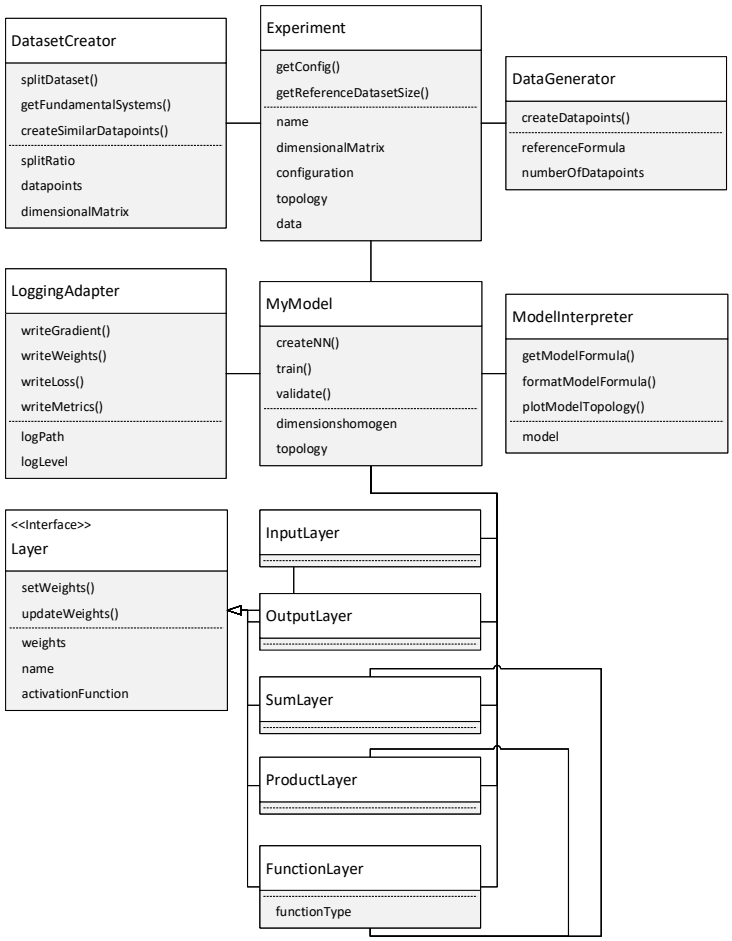

Abbildung 3.1.: Klassendiagramm des Konzepts

Die einzelnen Experimente werden als Instanz der Klasse „Experiment“ realisiert. Diese Klasse liest die bereitgestellte Konfigurationsdatei ein und übernimmt auf deren Basis die Steuerung des Ablaufs des Experiments. Details zu experimentspezifischen Durchführung werden in Abschnitt 3.4 beschrieben. Der Klasse sind weiterhin ein frei wählbarer eindeutiger Name des Experiments und die aus den jeweiligen Dateien eingelesene Dimensionsmatrix, die Topologiedefinition des neuronalen Netzes und eventuell ein bereits bestehender Datensatz zugeordnet.
Die Klasse „DataGenerator“ erstellt auf Basis der bereitgestellten Referenzformel die für das Experiment benötigten pseudorealen Datenpunkte. Hierfür sollen die Parameter der jeweiligen Funktion durch Zufallszahlen belegt werden. Die hieraus berechnete gesuchte Größe der Funktion wird zusammen mit den Parametern als Datenpunkt abgespeichert. Auf diese Weise generierte Datenpunkte können die Datenpunkte aus dem importierten Datensatz entweder erweitern oder ersetzen. Die Anzahl der durch die Instanz der Klasse „DataGenerator“ zu erzeugenden Datenpunkte wird durch die Experimentinstanz festgelegt.
Durch eine Instanz der Klasse „DatasetCreator“ wird ein Datensatz erzeugt, der alle im weiteren Verlauf des Experiments benötigten Daten enthält. Die Instanz bildet aus der Dimensionsmatrix durch Variation der Reihenfolge der Einträge der Relevanzliste x_i verschiedene Fundamentalsysteme. Auf Basis der erzeugten Fundamentalsysteme und der vorhandenen (pseudo-)realen Datenpunkte können durch die Klasse weitere vollähnliche Datenpunkte erzeugt werden. Ebenfalls nimmt die Klasse gemäß der Vorgabe die Verteilung der realen und erzeugten Datenpunkte auf die Trainings-, Validierungs- und Evaluierungsdatensätze vor.
Die „MyModel“-Klasse instanziiert das eigentliche neuronale Netz laut der vorgegebenen Topologiebeschreibung des Experiments durch die verfügbaren Layertypen. Die „MyModel“-Instanz steuert, ob das erzeugte Netz dimensionshomogen ist, und verfügt über alle Algorithmen für das Training und die Validierung des erzeugten neuronalen Netzes. Die Instanz ist das zentrale Element des Versuchs.
Die während des Trainings- und Validierungsablaufs erzeugten Kennzahlen werden durch eine Instanz der Klasse „LoggingAdapter“ verarbeitet und dauerhaft unter dem vom Benutzer definierten Pfad gespeichert. Der Detailgrad der zu speichernden Informationen wird durch ein sogenanntes Logging-Level festgelegt. Dies geschieht aufgrund der vergleichsweise langen Laufzeit der Schreiboperationen und ihrer hohen Anzahl im Falle der Verarbeitung zahlreicher Batches. Neben dem Ergebnis der Loss-Funktionen können auch die Gewichte aller Layer, die Gradienten zur Anpassung der Gewichte, und weitere Metriken durch die „LoggingAdapter“-Klasse nach jedem Batch oder jeder Epoche verarbeitet werden.
Die Klasse „ModelInterpreter“ stellt die Methoden zur Interpretation des Modells als algebraische Formel bereit. Der erzeugte symbolische Ausdruck kann durch die Klasse vereinfacht und in einem festen Format exportiert werden. Zusätzlich kann eine Instanz der Klasse eine Grafik der Modelltopologie erzeugen und exportieren.
Instanzen aus den Klassen „InputLayer“, „OutputLayer“, „SumLayer“, „ProductLayer“ und „FunctionLayer“ stehen dem Modell zur Realisierung der Topologievorgabe als neuronales Netz zur Verfügung. Alle diese Klassen erben ihre Attribute und Methoden von der abstrakten Klasse „Layer“. Diese enthält neben dem eindeutigen Namen der Instanz die Gewichte der einzelnen Neuronen zum betrachteten Zeitpunkt. Durch die Methoden der abstrakten Klasse können die Gewichte sowohl initialisiert als auch im Verlauf des Trainings verändert werden. Im Falle konstanter Gewichte können diese ebenfalls durch die jeweilige „Layer“-Instanz als konstante und demzufolge nicht trainierbare Größe definiert werden.

Die Klassen „SumLayer“, „ProductLayer“ und „FuntionLayer“ ermöglichen die exakte Modellierung der Versuchsformen als dimensionshomogenes oder nicht dimensionshomogenes neuronales Netz in Form eines gerichteten Graphen. Eine Instanz der Klasse „SumLayer“ bildet die gewichtete Summe aus den Eingängen des Layers, wobei die Gewichte der Summe den Gewichten des Layers entsprechen. Ein „ProductLayer“ bildet hingegen das mit den Gewichten des Layers potenzierte Produkt aus den Eingängen. Beide Layertypen können jeweils um einen Bias erweitert werden und somit einen zusätzlichen dimensionslosen Summanden oder Vorfaktor aus der Gleichung im Graphen abbilden. In einem „FunctionLayer“ wird auf die Eingangsdaten zunächst eine bestimmte Funktion $f(x)$ aus Formeln 3.11 bis 3.17 angewandt, bevor diese parallel in einem SumLayer und einem ProductLayer gewichtet aufsummiert bzw. mit den Gewichten des Layers potenziert und multipliziert werden.
Die Auswahl der Formeln erfolgte in Anlehnung an Rudolph [33] mit dem Ziel der Abbildung beliebiger Formeln in $\mathbb{R}_+^n$ und der Nutzung klassischer Aktivierungsfunktionen neuronaler Netze.

$$f(x) = sin(x) \tag{3.11}$$

$$f(x) = cos(x) \tag{3.12}$$

$$f(x) = x \tag{3.13}$$

$$f(x) = tanh(x) \tag{3.14}$$

$$f(x) = log(x) \tag{3.15}$$

$$f(x) = e^x \tag{3.16}$$

$$f(x) = \frac{1}{e^{-x} + 1} \tag{3.17}$$

Eine schematische Darstellung eines „FunctionLayers“ befindet sich in 3.2.

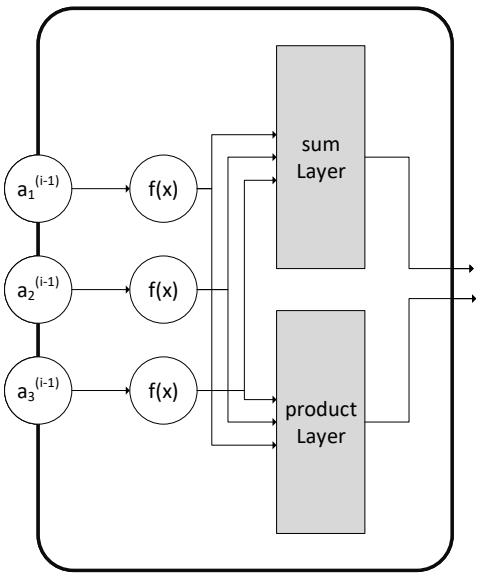

Abbildung 3.2.: schematische Darstellung Function-Layer

Das Entity-Relation-(ER)Diagramm in Abbildung 3.3 zeigt die Datenstruktur des Konzeptes.

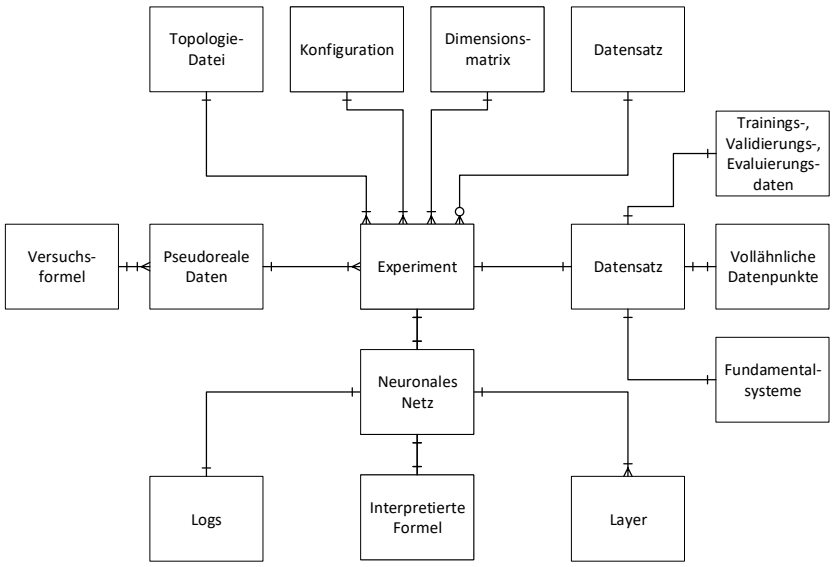

Abbildung 3.3.: Entity-Relationship-Diagramm des Konzepts

3.3. Implementierung des graphenbasierten Frameworks

Aufgrund der weiten Verbreitung, der umfangreichen Dokumentation und der Ressourceneffizienz ist das Programm auf Basis von Tensorflow in der Version 2.3.1 implementiert.
Zur Validierung und Visualisierung der Experimente wird Tensorboard aufgrund der engen Verzahnung mit Tensorflow verwendet. Als Programmiersprache wurde aufgrund der Kompatibilität zu Tensorflow und weiteren populären Frameworks auf dem Gebiet der neuronalen Netze Python ausgewählt.
Da in Python keine privaten Methoden oder Attribute definiert werden können, wird in den folgenden UML-Klassendiagrammen auf die Angabe der Sichtbarkeit verzichtet. Die Implementierung bildet zu großen Teilen das im vorherigen Absatz beschriebene Konzept ab. Aufgrund der Entscheidung für die Verwendung von Tensorflow waren jedoch einige Abweichungen nötig. In diesem Abschnitt wird deshalb die Implementierung detailliert beschrieben.
Die Klasse Experiment stellt alle benötigten Benutzerschnittstellen zur Durchführung des Versuchs bereit.
Sie verfügt zusätzlich über Methoden zum Einlesen der Konfigurationsdatei („get_config()“), der Topologiedatei, der Dimensionsmatrix und eines eventuell verfügbaren Datensatzes mittels der Methode „get_Data()“. Die Konfigurationsdatei ist im JSON-Format umgesetzt, die anderen Datei im CSV-Format. Beispiele für alle vier Dateien befinden sich im Anhang A.1. In Abbildung 3.4 ist die Struktur der Klasse in UML-Form dargestellt.

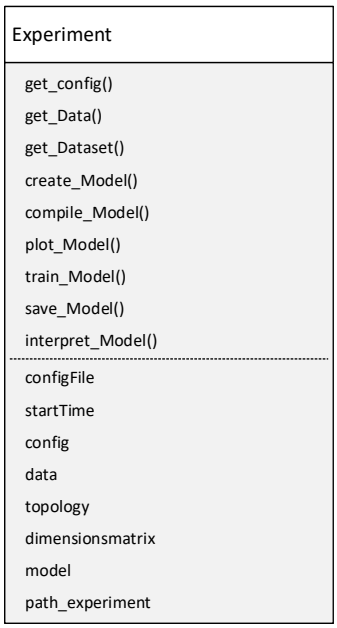

Abbildung 3.4.: Klasse „Experiment“ in UML-Darstellung

Neben dem Einlesen der Eingabe- und Konfigurationsdaten erzeugt die Klasse „Experiment“ auch die Instanzen des Datensatzes mittels der Methode „get_Dataset()“ und des eigentlichen neuronalen Netzes durch „get_Model()“. Die sonstigen in Abbildung 3.4 dargestellten Methoden dienen lediglich zur sequentiellen Durchführung des Versuchs.
Die Methode „compile_Model()“ konfiguriert das zuvor erstellte Modell durch die Definition der Loss-Funktion, des Optimizers und der weiteren Metriken, die durch das Modell ausgewertet und gespeichert werden sollen. Die Loss-Funktion wurde entsprechend der Experimentbeschreibung in Abschnitt 3.1 ausgewählt. Als Optimizer wurde in allen Versuchen der „Adam“-Optimizer (vgl. Abschnitt 2.1.2) genutzt.
Die Methode „plot_Model()“ erzeugt einen Plot der Topologie als PNG-Datei des durch „create_Model()“ erzeugten Modell-Objekts. Es erfolgt hierbei lediglich ein Aufruf einer Funktion aus dem Tensorflow Modul unter Vorgabe des Dateipfades und einiger gestalterischer Einstellungen.
Das Training des neuronalen Netzes wird durch die Methode „train_Model()“ angestoßen. Auf Basis des vorgegebenen Log-Levels wird zunächst eine Instanz der Klasse „MyCallback“ erzeugt. Diese fungiert als LoggingAdapter (vgl. Abbildung 3.1) und zeichnet je nach Log-Level verschiedene Kennzahlen während des Trainings auf. Des Weiteren werden innerhalb der Methode die Anzahl der maximalen Epochen und der Batchumfang festgelegt sowie die Trainings- und Validierungsdaten an die Modell-Instanz übergeben. Der eigentliche Trainingsablauf, wie in Abbildung 2.14 dargestellt, ist durch Tensorflow vorgegeben und wird in der Methode durch den Aufruf der „fit“-Methode des Modellobjekts initialisiert.
Durch die Methode „save_Model“ wird nach Abschluss des Trainings das erzeugte Modell im PB-Format im gewählten Pfad abgespeichert. Die Methode greift hierfür auf die entsprechende Funktion aus der Tensorflow-Bibliothek zurück.
Abschließend erzeugt die Methode „interpret_Model“ eine Instanz der Klasse „MyModelInter-

pretation“ nach den Vorgaben aus der Konfigurationsdatei und ruft deren Methode „getFormula“ auf, um einen symbolischen Ausdruck des Modells zu erhalten.
Die Klasse „FakeDataGenerator“ erzeugt auf Wunsch zufällige „pseudoreale“-Datenpunkte auf Grundlage einer definierten Funktion. In Abbildung 3.1 entspricht diese Klasse der Klasse „DataGenerator“. Die Klasse greift auf eine Funktionsbibliothek zu, die alle in Abschnitt 3.1 genutzten Formeln enthält. Für das Erzeugen einer Instanz der Klasse werden lediglich der Name der Funktion analog der Bibliothek sowie die Bezeichnungen der vorgegebenen Größen („x_Names“) und der gesuchten Größe („y_Names“) benötigt. In Abbildung 3.5 ist die Struktur der Klasse im UML-Format dargestellt.

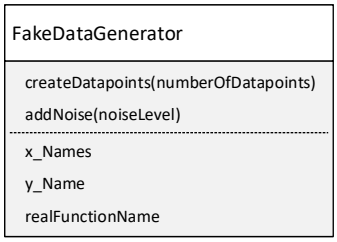

Abbildung 3.5.: Klasse „Fakedaten-Generator“ in UML-Darstellung

Die Methode „createDatapoints()“ erzeugt die angeforderte Menge an Datenpunkten, indem zunächst alle Eingangsgrößen mit einer zufällig gewählten Ganzzahl x_i aus dem Wertebereich von 1 bis 10 belegt werden. Die gesuchte Größe (das Label) ergibt sich durch die Auswertung der vorgegebenen Funktion und vervollständigt den Datenpunkt.
Durch die Methode „addNoise()“ kann nachträglich ein Rauschen nach Benutzervorgabe sowohl zu den Eingangsgrößen als auch zur gesuchten Größe hinzugefügt werden. Diese Funktion wurde in keinem der durchgeführten Experimente angewandt.
Die Klasse „Dataset“ bereitet die vorliegenden Daten zum Erstellen und Trainieren des Modells auf. Hierbei handelt es sich sowohl um die Dimensionsinformationen als auch um die eigentlichen realen oder pseudorealen Trainingsdaten. Zusätzlich verfügt die Klasse über Methoden um mittels vollähnlicher Datenpunkte die Anzahl an verfügbaren Datenpunkten zu vervielfachen. In Abbildung 3.6 wird die Klasse im UML Format dargestellt.

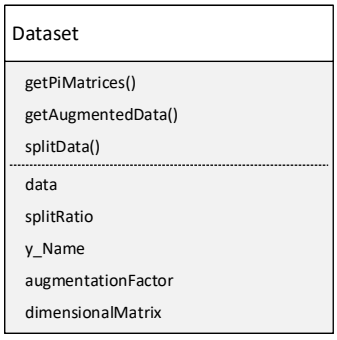

Abbildung 3.6.: Klasse „Dataset“ in UML-Darstellung

Die Klasse verfügt über die Methoden „getPiMatrices()“, „getAugmentedData()“ und „split-

Data()“.
Durch die Methode „getPiMatrices()“ werden die möglichen dimensionslosen Fundamentalsysteme aus der Dimensionsmatrix abgeleitet. Wie in Abschnitt 3.2 beschrieben entspricht die Menge der erzeugbaren Fundamentalsysteme der Menge, die durch das Varieren der Reihenfolge der dimensionsbehafteten Features entsteht. Im ersten Schritt wird eine Einheitsmatrix generiert, deren Rang der Anzahl der dimensionslosen Größen in der Dimensionsmatrix entspricht (z.B. Anzahl N).
Im zweiten Schritt werden aus den dimensionsbehafteten Größen die dimensionslosen Gruppen abgeleitet und damit die Fundamentalsysteme erzeugt. Die gesuchte Größe kann hierbei lediglich in einer dimensionslosen Gruppe mit einfacher Potenz vorkommen. Es entsteht eine Matrix, wie sie beispielhaft in Tabelle 3.8 dargestellt für die Dimensionsmatrix 3.1 ist.

	u_{Balken}	I_{Balken}	E_{Balken}	P_{Balken}	L_{Balken}
Π_1	1	0	0	0	-1
Π_2	0	1	0	0	-4
Π_3	0	0	1	-1	2

Tabelle 3.8.: π-Matrix Balkenbiegung

Im dritten Schritt wird gemäß Abbildung 3.7 die Identitätsmatrix aus allen dimensionslosen Größen mit der Stufenmatrix der Dimensionsbhafteten Größen kombiniert. Die daraus entstehende Matrix wird im Folgenden als π-Matrix bezeichnet.

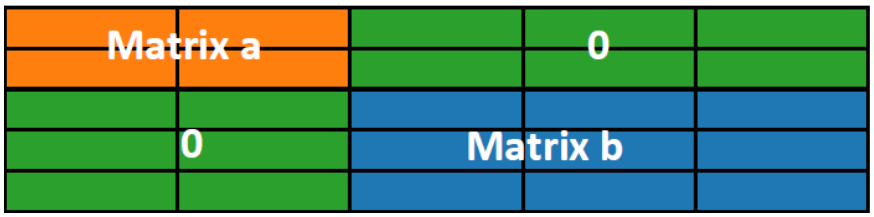

Abbildung 3.7.: Zusammengesetze π-Matrix

Zusätzlich werden die Spalten so angeordnet, dass die gesuchte Größe in der ersten Zeile und Spalte linear, d.h. mit Potenz 1 vorkommt.
Die Methode „getAugmentedData()“ vervielfacht jeden realen oder mittels eines instanzierten „FakeDataGenerator“ erzeugten pseudorealen Datenpunkte durch das Generieren vollähnlicher Datenpunkte. Die Anzahl der zu erzeugenden vollähnlichen Datenpunkte entspricht dem Attribut „augmentationFactor“ in der Konfigurationsdatei. Die vollähnlichen Datenpunkte werden in einem zufällig ausgewählten Fundamentalsystem erzeugt, nachdem sie durch die zuvor beschriebene Methode „getPiMatrices()“ aus der Dimensionsmatrix abgeleitet wurden. Zunächst wird zufällig eine dimensionsbehaftete Größe zum variieren aus den n rechten Spalten der π-Matrix ausgewählt, wobei n dem Rang der Dimensionsmatrix entspricht. Der Wert des realen Datenpunktes $x_{i,j}$ dieser Größe wird nun durch eine Normalverteilung variiert und als $x'_{j,i}$ mit $0 < i \leq n$ bezeichnet. Gemäß Abschnitt 2.3.2 sind alle dimensionslosen Kennzahl π_j für vollähnliche Datenpunkte konstant. Unter der Vorgabe, dass auf der Diagonalen der π-Matrix lediglich

die Größen $x_{j,j}$ linear vorkommen, ergibt sich der Wert der Größe $x_{j,j}$ nach Formel 3.18.

$$\left(\frac{x_{i,j}}{x'_{i,j}}\right)^{a_{i,j}} \cdot x_{j,j} = x'_{j,j} \tag{3.18}$$

Durch das Anwenden dieser Formel auf alle Zeilen der π-Matrix, in der $x_i \neq 0$ ist, wird der vollähnliche Datenpunkt erzeugt.
Die Klasse „MyModel“ wird von der Klasse „Model“ aus Tensorflow abgeleitet. Sie repräsentiert das neuronale Netz und verfügt über die Methoden zur Erstellung des Netzes aus der Topologiedatei, dem Training der Gewichte und der Ausgabe der benötigten Daten für das Debugging des Modells.
Die abgeleitete Klasse ergänzt die Elternklasse aus der Tensorflow Bibliothek hauptsächlich um die Fähigkeit neuronale Netze dynamisch auf Basis der Topologiebeschreibung zu erstellen. Diese Beschreibung ist in dem Attribut „layerTopology“ der Klasse gespeichert. Zusätzlich kann die „MyModel“-Klasse auf Wunsch das erzeugte neuronale Netz zu einem Dimensionshomogenen Netz konvertieren und zusätzliche Logging-Daten während des Trainings protokollieren.
Die Klasse ist in der folgenden Abbildung 3.8 als UML-Klassendiagramm dargestellt.

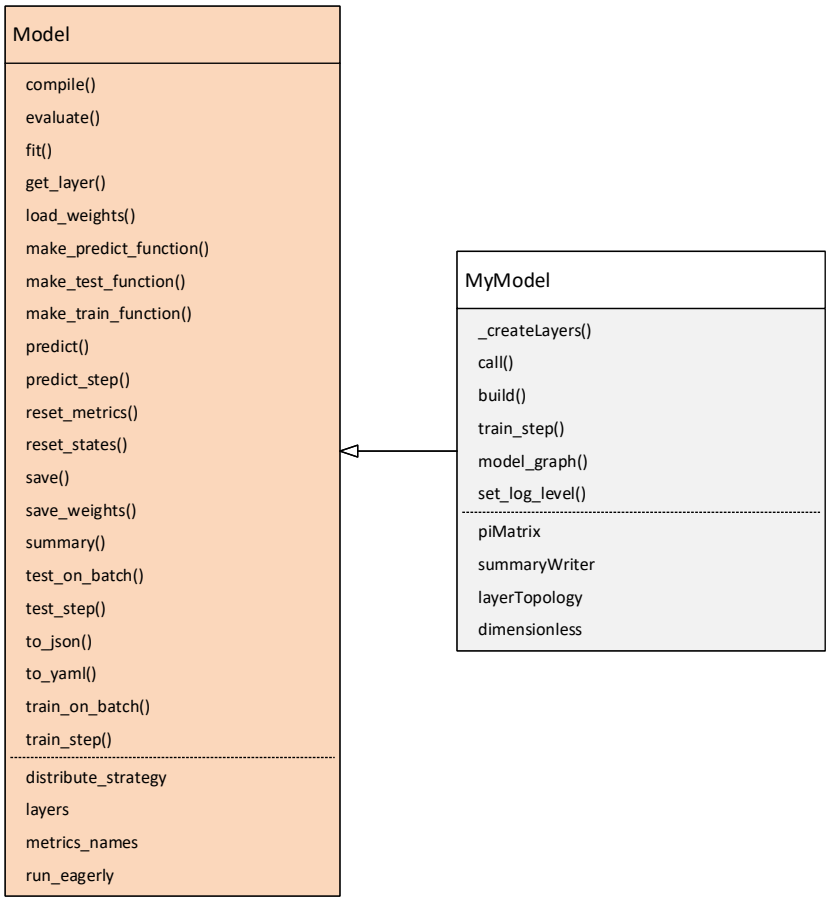

Abbildung 3.8.: Klasse „Model“ in UML-Darstellung

Mit der Initialisierung der Klasse werden die Layer des neuronalen Netzes auf Grundlage der Topologiedatei erstellt. Und die übergebenen Parameter als Attribute der Instanz gespeichert. Diese Attribute umfassen neben der bereits vorgestellten „layerTopology“ die aus der Dimensionsmatrix gebildete π-Matrix als Attribut „piMatrix“, einen Adapter zum Speichern der Logs im definierten Pfad, der als „summaryWriter“ übergeben wird, und die Angabe über die Dimensionshomognität des Modells aus der Experimentkonfiguration im Attribut „dimensionless“.
Das erzeugen der definierten Layer erfolgt bei der Initialisierung der Instanz durch die Methode „_createLayers()“. Diese Methode legt eine Liste an, in der alle Layerinstanzen abgelegt werden. Hierfür stehen die im folgenden beschriebenen „Layer“-Klassen zur Verfügung. Für jedes neuronale Netz werden ein sogenannter „Π-Layer“, ein „shortcut“-Layer und ein „Final“-Layer erzeugt. Die Anzahl der Neuronen n_π im „Π-Layer“ wird durch Formel 3.19 definiert, wobei M_Π der π-Matrix entspricht.

$$n_\pi = Rang(M_\Pi) - 1 \tag{3.19}$$

Durch die Methode werden weder die einzelnen Layer zu einem gerichteten Graphen verbunden, noch werden die Gewichte des Modells initalisiert oder mit den Werten aus der π-Matrix belegt. Die Verknüpfung der Layer erfolgt mit dem Aufruf der „MyModel“ Instanz über die angepasste „Call()“-Methode entsprechend der Topologiedefinition. Jeder Layer verfügt über exakt einen Eingang. Soll der Output verschiedener Layer zusammengefasst werden, wird dies über einen „concat“-, oder „multiply“-Layer realisiert. Diese Layertypen verfügen über keine Gewichte (sind also stateless) und aggregieren die eingehenden Tensoren entweder duch Aneinanderfügen oder durch elementweise Multiplikation zu einem einzelnen Output-Tensor, der für die nachfolgenden Layer wiederum als Inputtensor dient. Die Outputtensoren werden entsprechend der Position in der Topologiedefinition in einer Liste gespeichert. Der letzte Eintrag der Liste entspricht dem Output des Modells.
Die Methode „build()“ wird mit der Initialisierung des Netzes aufgerufen. Die Implementierung ruft zunächst die „build()“-Methode der Elternklasse auf, bevor sie die Gewichte des „Π-“ und des „shortcut“-Layers entsprechend Formel 2.40 aus der π-Matrix festlegen sofern es sich um ein dimensionshomogenes Modell handelt. Die Gewichte der beiden genannten Layer werden in diesem Zuge vom Training ausgeschlossen.
Die trainierbaren Gewichte werden mittels des Xavier-Algorithmus (vgl. Abschnitt 2.1.2) zufällig initialisiert. Die weiteren in Abbildung 3.8 dargestellten Methoden der „MyModel“-Klasse werden lediglich für das Erstellen von Grafiken der Netztopologie („model_graph()“) und zum Aufzeichnen der durch Tensorflow ermittelten Gradienten vor der Anpassung der Gewichte benötigt.
Die „Layer“-Klasse der Tensorflowbibliothek wird innerhalb des Programms lediglich als abstrakte Klasse verwendet. Von dieser Elternklasse werden die drei Klassen „MyBlock“, „MyProductBlock“ und „MyFunctionBlock“ abgeleitet. Alle drei Klassen verfügen über ein spezifisches Attribut „blockname“, das dem jeweiligen Layer einen eindeutigen Namen innerhalb des Netzes zuweist und eine Angabe darüber enthält, ob innerhalb des Layers ein Bias-Neuron existiert.

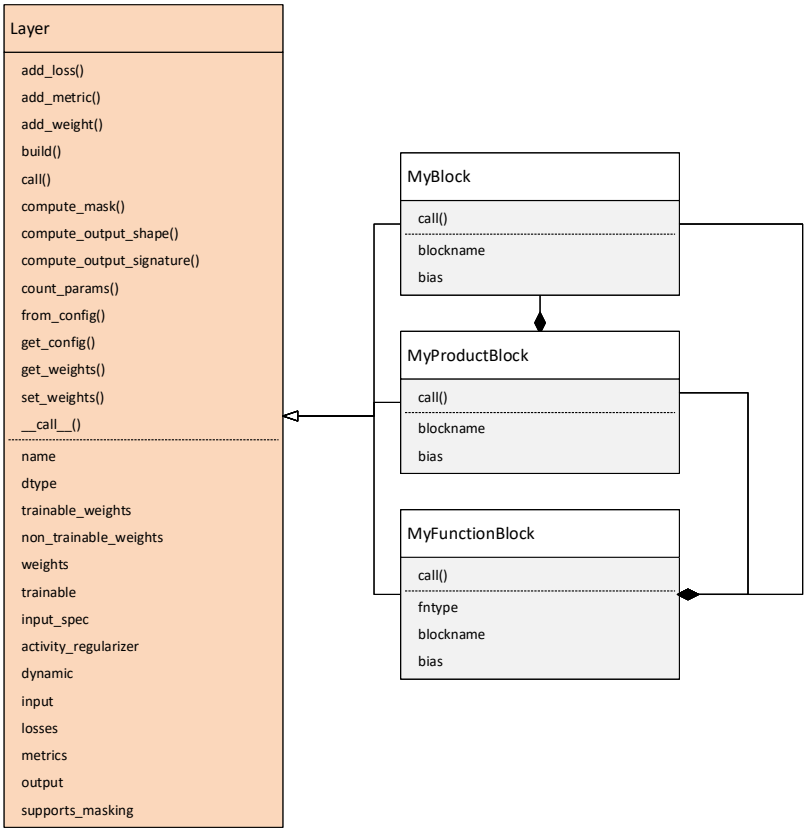

Abbildung 3.9.: Klasse „Layer“ in UML-Darstellung

Der Layer der Klasse „myBlock“ entspricht einem Perzeptron wie es in Abbildung 2.2 dargestellt und in Abschnitt 2.1 beschrieben ist. Als Aktivierungsfunktion Φ wird eine lineare Funktion verwendet. Standardmäßig verfügt der Layer über keinen Bias b und lediglich über ein Neuron. Es können jedoch durch den User weitere Neuronen zu dem Layer hinzugefügt werden. Wie bereits beschrieben bildet der Layer die gewichtete Summe der Eingänge ab.
Zur Abbildung des Produkts der mit den Gewichten potenzierten Eingänge kann ein Layer der Klasse „MyProductBlock“ verwendet werden. Dieser besteht, wie aus dem Klassendiagramm aus Abbildung 3.9 und Abbildung 3.10 entnommen werden kann, im Kern ebenfalls aus einem Layer der Klasse „MyBlock“.

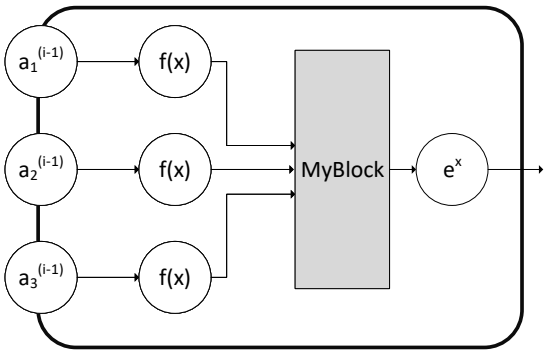

Abbildung 3.10.: Darstellung Produkt-Block

Ergänzt wird der Layer „MyBlock“ durch einen Funktionsknoten $f(x)$ für jeden Eingang und einen Funktionsknoten e^x für den Ausgang des inneren Blocks. Es ergibt sich entsprechend Formel 3.20 als Repräsentation der Klasse „ProductBlock“ für $x_i > 0$ und $b > 0$ mit n Eingängen.

$$\prod_{i=1}^{n} x_i^{a_i} = e^{\sum_{i=1}^{n} a_i ln(x_i)} \tag{3.20}$$

Die Instanz des „ProduktBlock“ übernimmt von der inneren „MyBlock“-Instanz die Standardbelegungen hinsichtlich der Aktivierungsfunktion und der Bias-Unit b.
Bei der Klasse „FunctionBlock“ handelt es sich um eine Umsetzung des Konzepts aus Abbildung 3.2. Der Layer besteht aus jeweils einer Instanz der Klassen „MyBlock“ (in der Abbildung als „sumLayer“ dargestellt) und „MyProductBlock“ bzw. „productLayer“. Zusätzlich kann dem "FunctionBlockLayer eine Funktion „fntype“ aus den Formeln 3.11 bis 3.17 zugeordnet werden. Testweise wurden alle Funktionen der Formeln 3.11 bis 3.17 parallel auf die Eingänge der „MyFunctionBlock“-Instanz angewandt, aufgrund mathematischer Instabilität konnten jedoch die definierten Versuche nicht erfolgreich durchgeführt werden.
Die Klasse „MyCallback“ ist eine Kindklasse der Tensorflow Klasse „Callback“. Durch die Klasse werden erweiterte Logging-Funktionen für das Modell implementiert. Sie erlaubt das dauerhafte Speichern sowohl aller Gewichte als auch der automatisch berechneten Gradienten für das Backpropagationverfahren am Ende jeder Epoche bzw. am Ende jedes Batches. Aufgrund der großen Anzahl an Schreiboperationen und der anfallenden Datenmenge muss dieses extensive Logging explizit über die Konfigurationsdatei mittels dem Attribut „log_level“ aktiviert werden.
Zusätzlich überprüft die aus der Klasse gewonnene Instanz am Ende jeder Epoche ob das festgesetzte „early_stopping_limit“ am Ende der letzten fünf Epochen unterschritten wurde, und bricht gegebenenfalls das Training bereits vor dem Erreichen der maximalen Epochenanzahl ab.
Die beschriebenen Attribute und Methoden der Klasse „MyCallback“ sowie deren Beziehung zur Elternklasse sind in Abbildung 3.11 dargestellt.

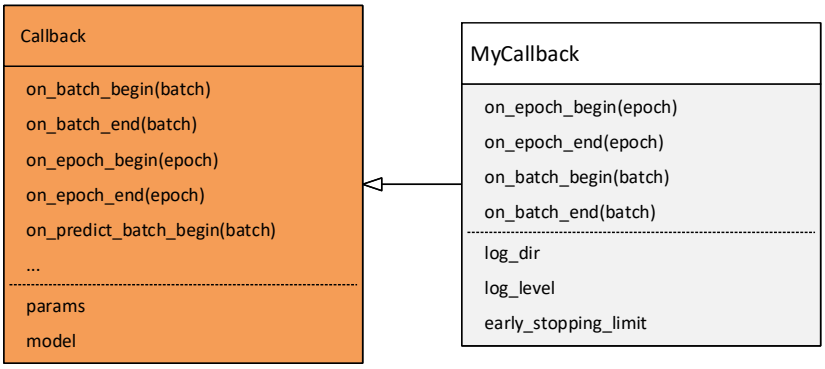

Abbildung 3.11.: Klasse „MyCallback“ in UML-Darstellung

Die Klasse „MyModelInterpretation“ konvertiert den Graphen des neuronalen Netzes in eine symbolische Darstellung. Die entstehenden Ausdrücke können mittels eines zugeordneten Adapters der Klasse „MatlabAdapter“ vereinfacht werden. Als Format der symbolischen Darstellung kann sowohl das Latex- als auch ein zum Design Cockpit 43®[1] kompatibles Format ausgewählt werden. Die Klasse „MatlabAdapter“ benötigt eine Instanz der Matlab-Software. Die beiden Klassen und deren Beziehung sind in Abbildung 3.12 dargestellt. Die Methode „getFormula“

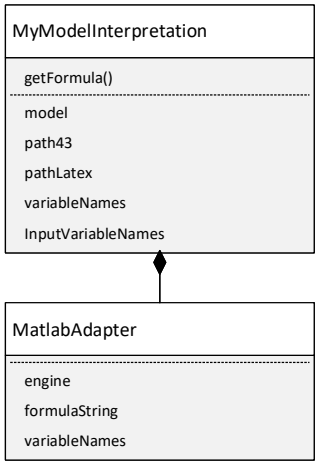

Abbildung 3.12.: Klasse „MyModelInterpretation“ in UML-Darstellung

interpretiert das als Attribut „model“ an die Instanz angehängte neuronale Netz und dessen Gewichte auf Basis des Attributes „InputVariableNames“ (Die symbolischen Bezeichnungen der Eingangsdaten).
Im ersten Schritt werden alle Layer des Modells ermittelt, die direkt mit dem Input-Layer verbunden sind. Für jeden dieser Layer wird auf Basis der Eingangsvariablennamen und einer in

[1] https://www.iils.de/

der Klasse enthaltenen Funktionsbibliothek der symbolische Ausdruck für jeden Ausgang des Layers aus den Gewichten des Layers entwickelt. Im zweiten Schritt wird der nächste Layer ermittelt. Im dritten Schritt werden an den ermittelten Layer die symbolischen Ausdrücke als neue Eingangsvariablennamen übergeben und der Prozess wird erneut durchgeführt. Es handelt sich hierbei um einen rekursiven Algorithmus, bei dem sich die Funktion selbst aufruft, bis kein nachfolgender Layer mehr existiert. Der symbolische Ausdruck am Ausgang des letzten Layers in Netzrichtung entspricht der Interpretation des gesamten Modells als symbolischer Ausdruck. Aggregationslayer wie der „Concate“ oder der „multiply“-Layer verzögern die Ausführung der rekursiven Funktion bis zum Vorliegen des symbolischen Ausdrucks für alle vorherigen Layer.

3.4. Durchführung des Experiments

Durchgeführt wurden die Versuche mit der Python-Version 3.7.7. Die Implementierung ist auszuschließlich CPU-basiert und die Versuche wurden auf einem Laptop unter Microsoft Windows 10 Pro in der Version 2004 durchgeführt. Bei der verwendeten Hardware handelt es sich um ein Lenovo Thinkpad T490 mit einem Intel-Core-i7-8565U Prozessor und 16 GB Arbeitsspeicher. Es wurde keine externe oder dedizierte Grafikkarte zur Programmausführung verwendet.
Die in Abschnitt 3.1 beschriebenen Experimente wurden durch das in Abschnitt 3.3 vorgestellte implementierte Framework realisiert. Mittels des Programms Tensorboard wurden die Trainingsfortschritte bereits zur Laufzeit überwacht. Die Auswertung der gewonnenen Daten erfolgte mittels des Python-Programms Matplotlib. Aufgrund von häufigen Programmabbrüchen wurde auf die Herstellung der vereinfachten symbolischen Form durch Matlab verzichtet.

3.5. Validierung des Experiments

Die Validierung der Experimente und der einzelnen Versuche erfolgte durch die Analyse der Losskurven für die Trainings- und Validierungsdaten mittels Tensorflow. Durch dieses Verfahren kann Overfitting (vgl. Abschnitt 2.1.3) des Modells leicht erkannt werden. Die korrekte Topologie der einzelnen Modelle wurde durch die grafischen Ausgaben der Modelltopologien validiert. Durch das Verfahren zum Finden einer Referenzgröße für den Datensatz kann systematisches Overfitting vermieden werden. Durch das Wissen um die korrekte Formel für jeden Versuch kann Underfitting ebnso ausgeschlossen werden. Aufgrund des Versuchsdesigns ist jedoch eine Varianz bei der erzielbaren Genauigkeit als Messgröße der Versuche aus den Absätzen 3.1.2 und 3.1.3 gewünscht.

4. Ergebnisse

4.1. Wahl der dimensionslosen Gruppen

Gemäß dem in Abschnitt 4.1 beschriebenen Experiment wurde die Loss-Funktion über die Epochen für verschiedene Fundamentalsysteme bestimmt. Die Balkenbiegegleichung aus Formel 3.1 die enthält fünf dimensionsbehafteten Größen $u_{Balken}, P_{Balken}, L_{Balken}, E_{Balken}, I_{Balken}$. Durch Variation der Reihenfolge der Spalten der Dimensionsmatrix ergeben sich 19 Fundamentalsysteme. Für jedes Fundamentalsystem wurde ein dimensionshomogenes neuronales Netz trainiert. Die Kurven der Loss-Funktion über den Epochen lassen sich in drei verschiedene Gruppen einteilen. Dargestellt sind die Kurven in Abbildung 4.1.

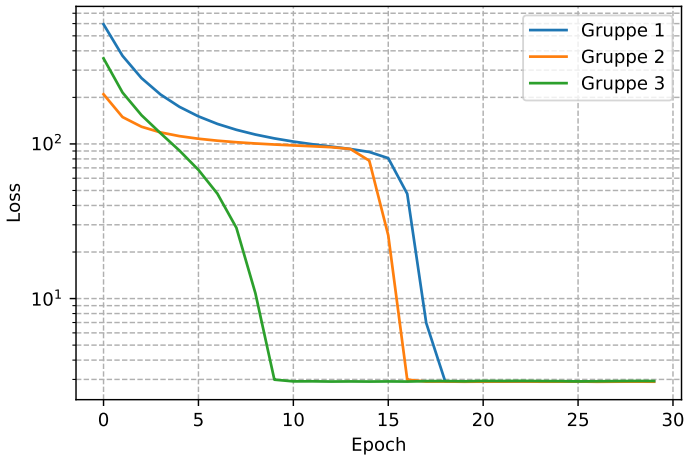

Abbildung 4.1.: Entwicklung der Loss-Funktion für verschiedene Fundamentalsysteme

Die benötigte Anzahl von Epochen variiert mit der Wahl des Fundamentalsystems und der darausfolgenden festgelegten Gewichten sichtbar. Eine Analyse der jeweils verwendeten Fundamentalmatrizen ergibt dass alle der Gruppe 3 zugeordneten Fundamentalsysteme über nicht ganzzahlige Einträge in der Fundamentalmatrix verfügen. Fundamentalmatrizen der Gruppe 2 verfügen über nichtganzzahlige Einträge in einer Spalte, Fundamentalmatrizen aus Gruppe 1 in zwei Spalten. Für alle nichtganzzahligen Einträge gilt jeweils $|a_{i,j}| < 1$.
In Tabelle 4.1 ist beispielhaft eine der insgesamt acht Fundamentalmatrizen der ersten Gruppe dargestellt.

π	u_{Balken}	P_{Balken}	L_{Balken}	I_{Balken}	E_{Balken}
0	1,0	0,0	0,0	-0,25	-0,0
1	0,0	1,0	0,0	-0,5	-1,0
2	0,0	0,0	1,0	-0,25	-0,0

Tabelle 4.1.: Beispielhafte Fundamentalmatrix Gruppe 1

Auffällig ist außerdem, dass die Fundamentalmatrizen der Gruppen 1 und 2 die Dimension L lediglich linear verwenden, diese Dimension somit lediglich auf der Diagonalen der Matrix vorkommt. In Tabelle 4.2 ist eine Fundamentalmatrix aus Gruppe 2 dargestellt. Die Fundamentalmatrizen der zweiten Gruppe unterscheiden sich von denjenigen der ersten Gruppe durch die Tatsache, dass sich die Dimension I stets auf der Diagonalen der Matrix befindet.

π	u_{Balken}	L_{Balken}	I_{Balken}	P_{Balken}	E_{Balken}
0	1,0	0,0	0,0	-0,5	0,5
1	0,0	1,0	0,0	-0,5	0,5
2	0,0	0,0	1,0	-2,0	2,0

Tabelle 4.2.: Beispielhafte Fundamentalmatrix Gruppe 2

In Tabelle 4.3 ist eine der sieben Fundamentalmatrizen von Gruppe 3 dargestellt. In allen Fundamentalmatrizen der Gruppe kommt die Dimension I linear vor, während die Dimension L nicht linear vorkommt. Hieraus folgt, dass in jeder Fundamentalmatrix eine dimensionslose Kennzahl π nach Formel 4.1 existiert.

	u_{Balken}	I_{Balken}	P_{Balken}	L_{Balken}	E_{Balken}
0	1,0	0,0	0,0	-1,0	-0,0
1	0,0	1,0	0,0	-4,0	-0,0
2	0,0	0,0	1,0	-2,0	-1,0

Tabelle 4.3.: Beispielhafte Fundamentalmatrix Gruppe 3

$$\pi = \frac{I}{l^4} \tag{4.1}$$

4.1.1. Diskussion

Da das neuronale Netz über keine Layer zur Normalisierung verfügt, besteht eine Tendenz zu Exploding bzw. Vanishing-Gradients gemäß Abschnitt 2.1.3. Da alle Variablen durch die Instanz der Klasse „FakeDateGenerator“ mit den gleichen Zufallswerten belegt werden, wirken sich die Einträge der Fundamentalmatrix direkt auf dieses Risiko und damit auf den Verlauf der Loss-Funktion aus. Beim Vergleich der verschiedenen Fundamentalmatrizen wird deutlich, dass die Matrizen von Gruppe 3 über betragsmäßig große Einträge verfügen. Dies resultiert in einem erhöhten Risiko für Exploding-Gradients, die eine mögliche Erklärung für die unmittelbar steil abfallende Loss-Kurve in Abbildung 4.1 ist.
Neben der gewünschten schnellen Minimierung der Fehlerfunktion wirken sich Exploding Gra-

dients negativ auf die Stabilität des Modells aus. Dieser Effekt konnte für diesen vergeichsweise einfachen Versuch weder beobachtet noch nachgewiesen werden.

4.2. Vervielfachung des Datensatzes

Das zweite Experiment wurde erneut entsprechend der in Kapitel 3 beschriebenen Definition mit dem Ziel der Approximation der Biegebalkenformel 3.1 durchgeführt.
In Abbildung 4.2 sind die Verläufe der verschiedenen Loss-Funktionen über den Epochen dargestellt. Bei der Kurve „Aug10“ handelt es sich um den mittels vollähnlichen Datenpunkten verzehnfachten Datensatz. Entsprechend beschreibt Kurve „Aug100“ den analog dazu verhundertfachten Datensatz. Die Kurven „Nor10“ und „Nor100“ beschreiben die Entwicklung der Loss-Funktion über den Epochen für Datensätze, die mittels des Datenpunktgenerators die zehn- bzw. hundertfache Anzahl von Datenpunkten des Referenzdatensatzes erzeugt wurde. Das Verhalten des Referenzdatensatzes wurde durch die Kurve „NoDH“ abgebildet.
Die Kurven in Abbildung 4.2 entsprechen dem Median der einzelnen Losskurven der Versuchsdurchläufe.

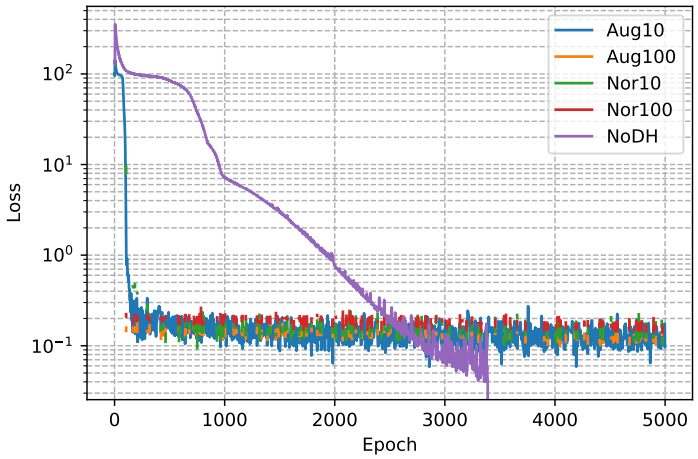

Abbildung 4.2.: Entwicklung der Loss-Funktion für vervielfachte Daten

Die Fehlerkurven der um den Faktor 10 oder 100 vervielfachten Datensätze fallen sichtbar schneller ab als die Kurve des Vergleichsdatensatzes.
Es ist kein Unterschied zwischen den Datensätzen bestehend aus nativen Datenpunkten und den Datensätzen, deren Umfang durch vollähnliche Datenpunkte vervielfacht wurde, erkennbar. Außerdem kann keine Differenz zwischen den verzehnfachten und verhundertfachten Datensätzen festgestellt werden.
In Abbildung 4.3 wird die Verteilung der erreichten Loss-Werte bei Abbruch des Trainings für jeden Durchlauf des Experiments beschrieben.

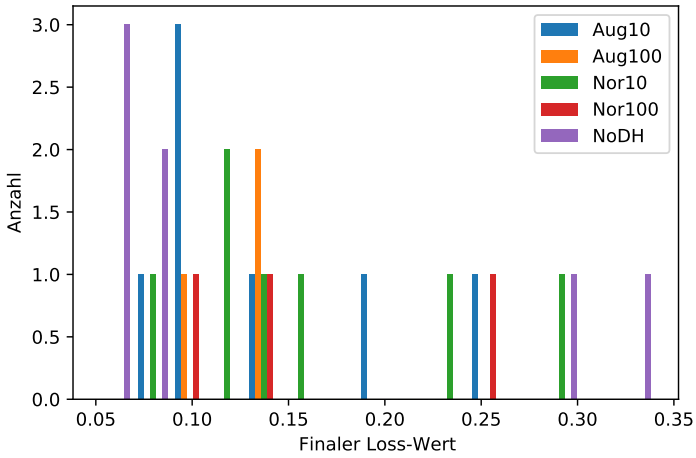

Abbildung 4.3.: Entwicklung der Loss-Funktion für vervielfachte Daten

4.2.1. Diskussion

Das neuronale Netz profitiert von einer, im Vergleich zum Referenzdatensatz, vergrößerten Anzahl an verfügbaren Datenpunkten durch eine veringerte Anzahl benötigter Trainingsepochen. Zurückzuführen ist dies, teilweise auf die höhere Anzahl von Batches und damit eine häufigere Anpassung der Gewichte innerhalb einer Epoche bei vervielfachten Datensätzen durch die Anwendung des Adam-Optimizers (vgl. Abschnitt 2.1.2).
Dieser Effekt scheint jedoch einer Sättigung zu unterliegen, da die weitere Vervielfachung des Datensatzes um den Faktor 100 der ursprünglchen Größe keinen Einfluss auf die Loss-Funktion des Modells hat.
Gemäß dem Versuch haben vollähnliche Datenpunkte im Vergleich zu zusätzlichen nativen Datenpunkten keine Nachteile bei der Betrachtung der Loss-Funktion. Durch die Vervielfältigung der Datenpunkte kann somit das kosten- und zeitintensive Sammeln von Trainings- und Validierungsdaten in realen Anwendungen minimiert werden.
Die erreichbare Genauigkeit war für alle Datensätze vergleichbar. Die Vervielfältigung des bereits ausreichend großen Referenzdatensatzes hat für dieses Experiment keine Auswirkung auf die erzielbare Genauigkeit des neuronalen Netzes.

4.3. Nutzung der Dimensionsinformationen

Das Experiment wurde gemäß der Spezifikation in Absatz 3.1.3 durchgeführt. Die in den Diagrammen 4.4, 4.6, 4.8, 4.10, 4.13, 4.16 und 4.19 dargestellten Kurven entsprechen dem Median der einzelnen Versuchsdurchgänge.
In den folgenden Schaubildern werden jeweils der Verlauf der Loss-Funktion über den Epochen und die Verteilung des Loss bei Abbruch bzw. Beendigung des Trainings dargestellt. Sofern es bei einem der Versuche zu mindestens einem Trainingsabbruch aufgrund von mathematischer Instabilität kam, wird die Anzahl dieser Abbrüche ebenfalls in einem Diagramm dargestellt.

In allen Diagrammen bezeichnen die „Aug10“-Daten analog der Darstellung in Absatz 4.2 die Versuche, die mit dem vervielfachten Datensatz durchgeführt wurden. Die Vervielfachung des Datensatzes geschah ausschließlich durch das Erzeugen von vollähnlichen Datenpunkten. Die mit „topo“ bezeichneten Daten in den Schaubildern zigen das Verhalten des dimensionshomogenen neuronalen Netzes auf. Mit „NoDH“ werden die Referenzdaten bezeichnet, die mit einem nicht dimensionshomogenen neuronalen Netz und einem Datensatz mit dem Referenzumfang durchgeführt wurden.

4.3.1. Balkenbiegung

In Abbildung 4.4 wird der Verlauf des Medians der Loss-Funktion für die einzelnen Durchläufe dargestellt. Analog Absatz 4.2 konvergiert der Loss-Wert der Versuche mit dem vervielfachten Datensatz schneller gegen die Nulllinie als die Kurve des Referenzversuchs. Die Kurve des Medians der Versuche mit dem dimensionshomogenen neuronalen Netz verbleibt bis zum Abbruch des Trainings aufgrund des Erreichens der maximalen Epochenanzahl bei ca. 70 %. Die

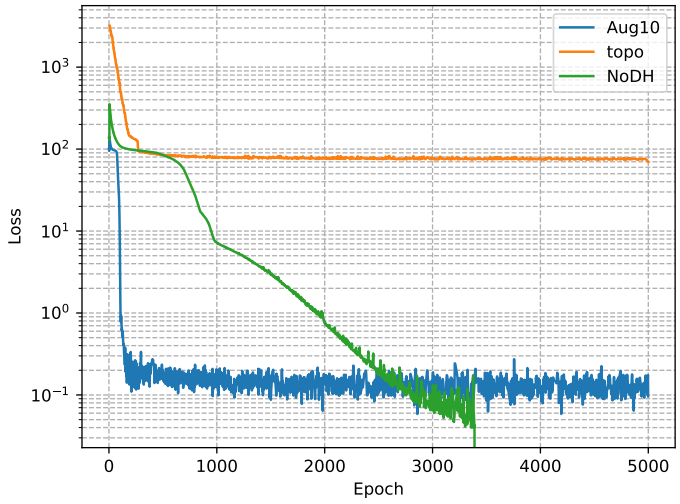

Abbildung 4.4.: Entwicklung der Loss-Funktion über die Epochen

Verteilung der zuletzt erreichten Loss-Werte der einzelnen Versuche zeigt, dass sowohl die Referenzversuche als auch die Versuche mit dem verzehnfachten Datensatz alle über einen geringen Loss-Wert am Ende verfügen. Die Verteilung der Werte für die Versuche mit dem dimensionshomogenen neuronalen Netz ist hingegen zweigeteilt. Drei der sieben Versuche erreichten eine ähnliche Genauigkeit wie die zuvor genannten Methoden, die Mehrheit (4 von 7) erreichte jedoch lediglich Loss-Werte von 70 bis 80 %.
Dies erklärt den Verlauf der Median-Loss-Kurve für die Versuche mit dimensionshomogenen neuronalen Netzen (topo) in Abbildung 4.4.
Keiner der durchgeführten Versuche brach wegen mathematischer Instabilität ab.

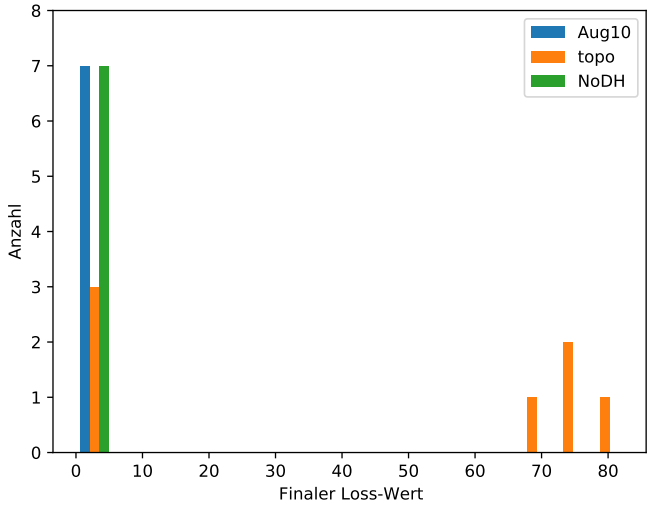

Abbildung 4.5.: Verteilung der finalen Ergebnisse der Loss-Funktionen

4.3.2. Senkrechter Wurf

In Abbildung 4.6 wird der Verlauf des Medians der Loss-Funktion für die einzelnen Durchläufe dargestellt. Die Kurven der Versuche mit den dimensionshomogenen neuronalen Netzen und der vervielfachten Datensätze konvergieren schneller gegen den Wert 0 als die Kurve der Referenzversuche.

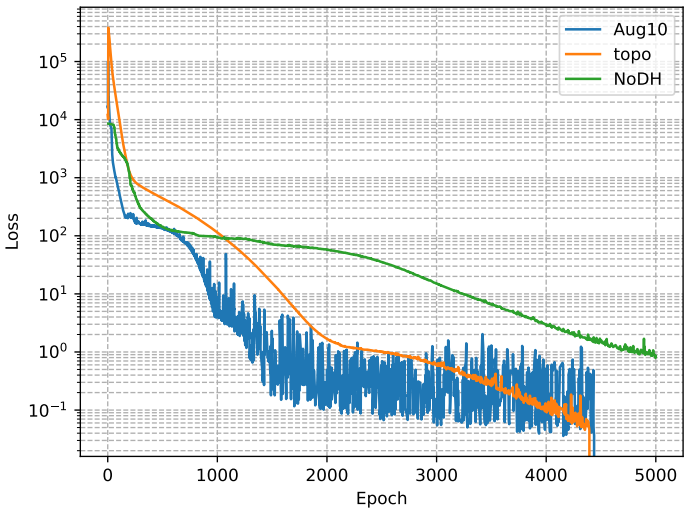

Abbildung 4.6.: Entwicklung der Loss-Funktion über die Epochen

Zusätzlich erreichen die Versuche „Aug10“ und „topo“ unter Einbeziehung der Dimensionsinformationen eine höhere finale Genauigkeit als die Referenzversuche. Dieser Sachverhalt wird auch im Verteilungsdiagramm der finalen Loss-Werte der einzelnen Versuche in Abbildung 4.7 teilweise dargestellt. Bis auf einen einzelnen Ausreißer liegen die zuletzt erreichten Loss-Werte der

Versuche mit Nutzung der dimensionsinformationen nahe dem Wert 0. Aufgrund der Zuordnung der einzelnen finalen Loss-Werte in diskrete Bereiche wird entgegen dem Epoch-Loss-Diagramm in Abbildung 4.6 die Mehrzahl der Werte der Referenzversuche ebenfalls dem Wert 0 zugeordnet.

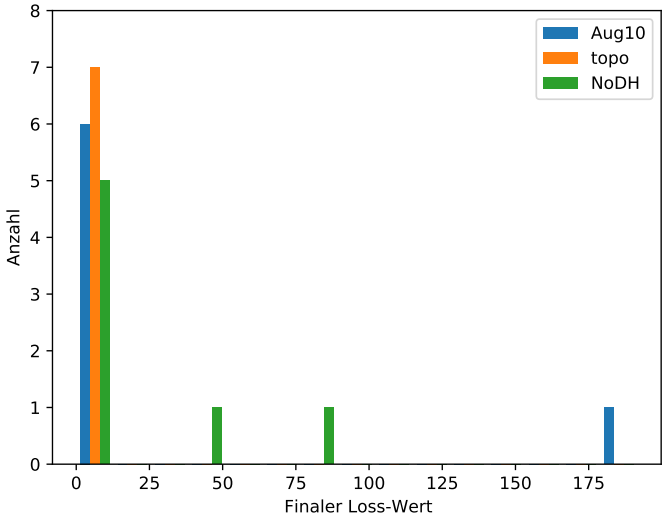

Abbildung 4.7.: Verteilung der finalen Ergebnisse der Loss-Funktionen

Es brach wiederum keiner der durchgeführten Versuche wegen mathematischer Instabilität ab.

4.3.3. Fermi-Verteilung

Das neuronale Netz zur Approximation der Formel 3.4 benötigt in der dimensionshomogenen Konfiguration weniger Epochen zum Erreichen eines genauen Ergebnisses als die Netze in der Vergleichskonfiguration oder der Konfiguration mit dem vervielfachten Datensatz (vgl. Diagramm in Abbildung 4.8).

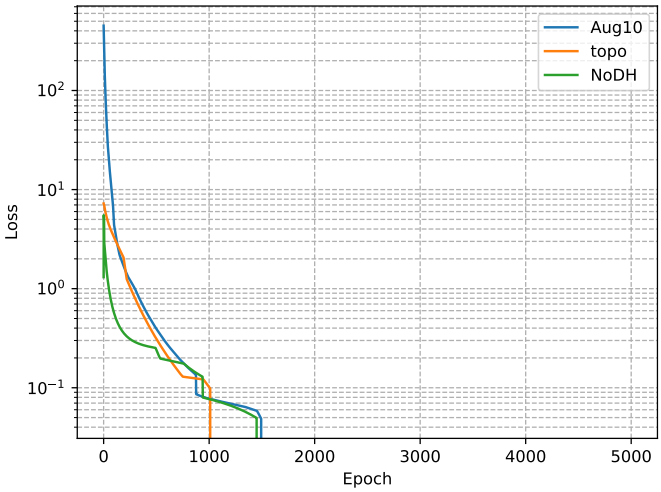

Abbildung 4.8.: Entwicklung der Loss-Funktion über die Epochen

Auch auf die Verteilung der finalen Ergebnisse der Loss-Funktion hat die Konfiguration keinen großen Einfluss wie im Verteilungsdiagramm in Abbildung 4.9 dargestellt ist.

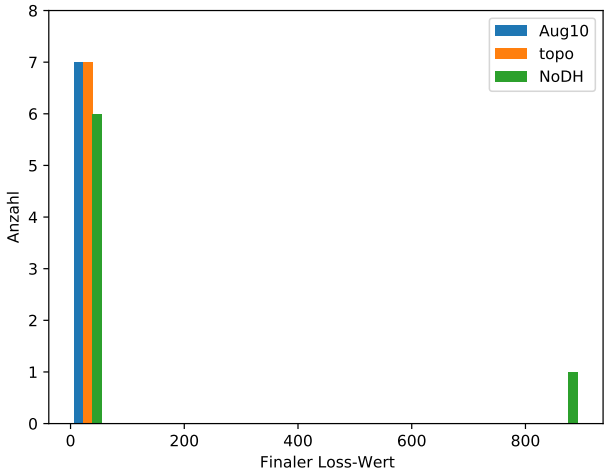

Abbildung 4.9.: Verteilung der finalen Ergebnisse der Loss-Funktionen

Es kam auch bei der Approximation dieser Formel zu keinen Abbrüchen aufgrund von mathematischer Instabilität.

4.3.4. Zerfallsgesetz

Das Training für die Approximation von Formel 3.5 durch das neuronale Netz ist in Abbildung 4.12 dargestellt. Für die Versuche mit dimensionshomogenen Netzen werden die wenigsten Epochen benötigt, um gegen die Nulllinie zu konvergieren. Beim Median der Versuche mit dem vervielfachten Datensatz bedarf es, verglichen mit der dimensionshomogenen Konfiguration, zusätzlicher Epochen zum Erreichen der Abbruchbedingung. Die Referenzkonfiguration benötigt ebenfalls ein Vielfaches der Epochen der dimensionshomogenen Konfiguration.

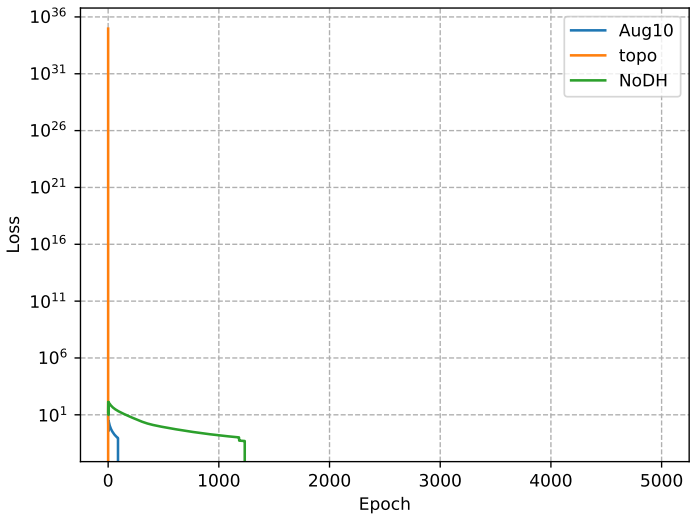

Abbildung 4.10.: Entwicklung der Loss-Funktion über die Epochen

Im Verteilungsdiagramm in Abbildung 4.11 ist erkennbar, dass die Mehrheit der Versuche innerhalb einer Konfiguration über einen kleinen Loss-Wert verfügt. Auffällig ist jedoch die Varianz der Werte mit der Bezeichnung „topo“ in Abbildung 4.11 des finalen Loss-Wertes für die dimensionshomogene Gleichung.

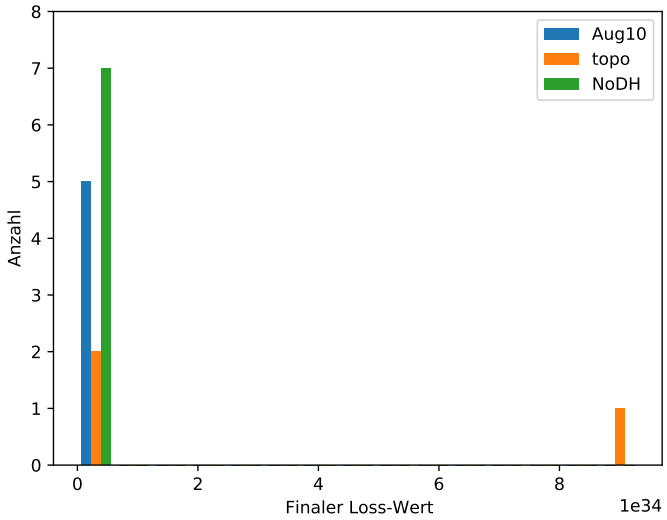

Abbildung 4.11.: Verteilung der finalen Ergebnisse der Loss-Funktionen

Die verglichen mit der Referenzfunktion spärliche Besetzung des Verteilungsdiagramms der beiden Konfigurationen, die die Dimensionsinformation verarbeiten, erklärt sich in Schaubild 4.12.

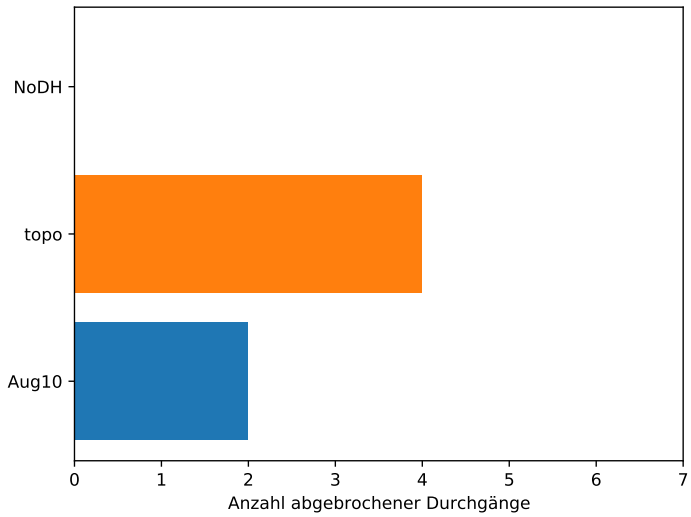

Abbildung 4.12.: Anzahl abgebrochener Trainingsvorgänge

Aufgrund von mathematischer Instabilität brachen vier der sieben Versuchsläufe der dimensionshomogenen Konfiguration und zwei der sieben Läufe mit vervielfachter Anzahl an Datenpunkten ab. Bei der Referenzkonfiguration kam es zu keinen Abbrüchen.

4.3.5. Rapidität

Die Approximation von Formel 3.6 kann durch die Versuchskonfiguration mit dem vervielfachten Datensatzumfang innerhalb der wenigsten Epochen (Im Vergleich zu den sonstigen Konfigurationen) mit dem Erreichen der Abbruchbedingung aufgrund eines bereits kleinen Loss-Wertes erziehlt werden (Abbildung 4.13). Für die Referenzkonfiguration ohne Beachtung der Dimensionsinformation wird bereits ein Vielfaches von Epochen zum Training der Gewichte benötigt. Der Median der Loss-Funktion für die Versuchsläufe mit dimensionshomogenem Netz erreicht die Genauigkeit der beiden zuvor genannten Konfigurationen innerhalb der maximalen Anzahl von Epochen nicht.

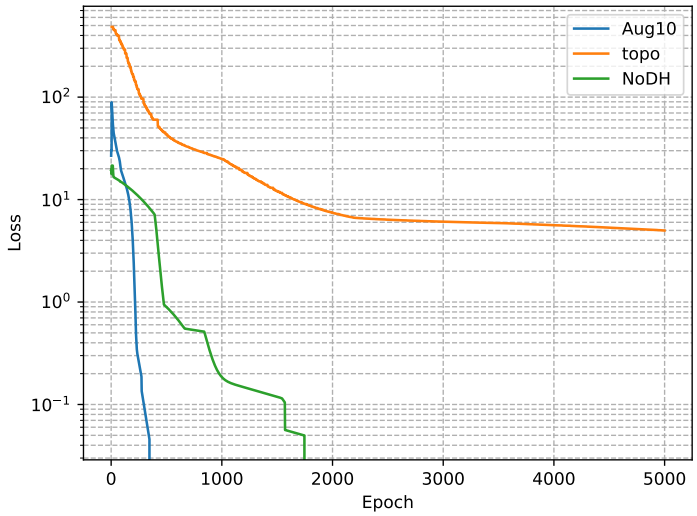

Abbildung 4.13.: Entwicklung der Loss-Funktion über die Epochen

Die Varianz der Verteilungsfunktion der finalen Loss-Werte ist für die dimensionshomogene Versuchskonfiguration „topo“ wesentlich größer als die jeweilige Varianz der sonstigen Konfigurationen (vgl. Abbildung 4.14). Es kam bei der Approximation dieser Formel zu keinen Abbrüchen aufgrund mathematischer Instabilität.

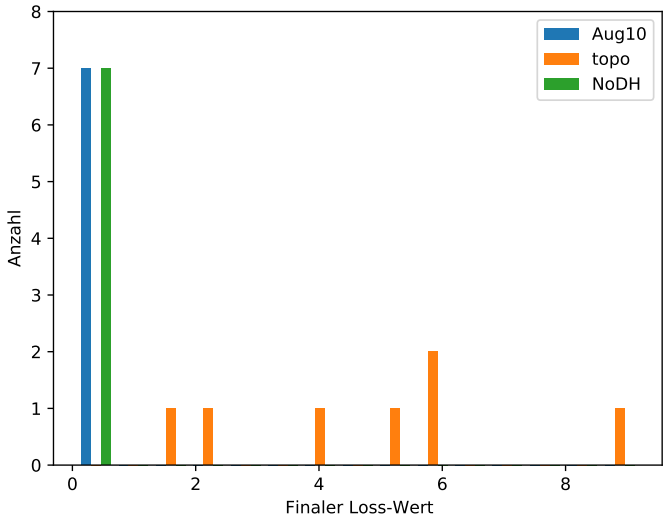

Abbildung 4.14.: Verteilung der finalen Ergebnisse der Loss-Funktionen

4.3.6. Federpendel

Für den Versuch der Approximation des Federpendels liegen keine Loss-Funktionskurven für die Versuchskonfiguration mit dem vervielfachten Datensatz vor. Grund hierfür ist, dass alle Versuchsläufe aufgrund mathematischer Instabilität abgebrochen wurden. Visualisiert ist dies in Abbildung 4.15.

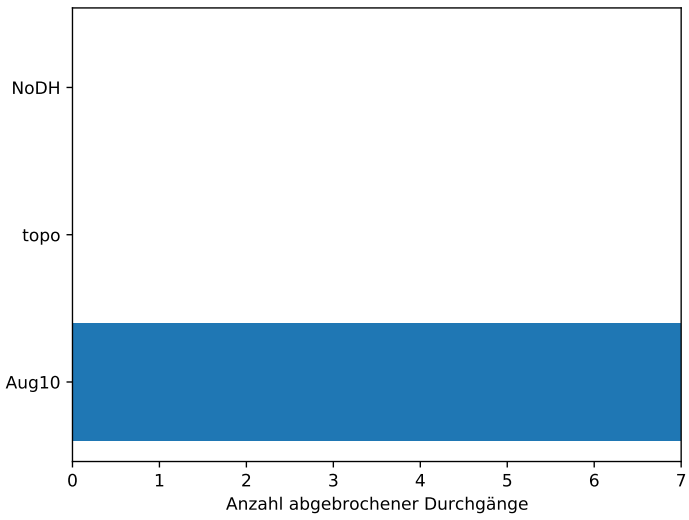

Abbildung 4.15.: Anzahl abgebrochener Trainingsvorgänge

Die verbleibenden untersuchten Konfigurationen erreichen den festgelegten Grenzwert zum Abbruch des Trainings innerhalb der maximalen Anzahl an Epochen nicht. Die Konfiguration mit dimensionshomogenen Netzen verfügt zwar über eine stetig fallende Loss-Funktion, diese erreicht jedoch nicht die Genauigkeit bzw. den geringen Loss-Wert der Referenzkonfiguration

(vgl. Abbildung 4.16).

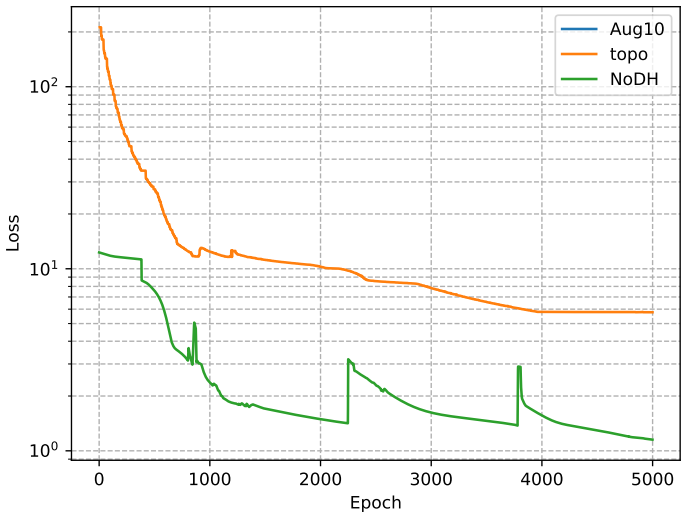

Abbildung 4.16.: Entwicklung der Loss-Funktion über die Epochen

Die Ergebnisse aus Abbildung 4.16 spiegeln sich auch in der Verteilungsfunktion der finalen Loss-Werte in Abbildung 4.17 wider. Die Varianz der dimensionshomogenen Konfiguration ist im Vergleich mit der Referenzkonfiguration kleiner, der Erwartungswert und somit auch der Median sind jedoch wesentlich größer.

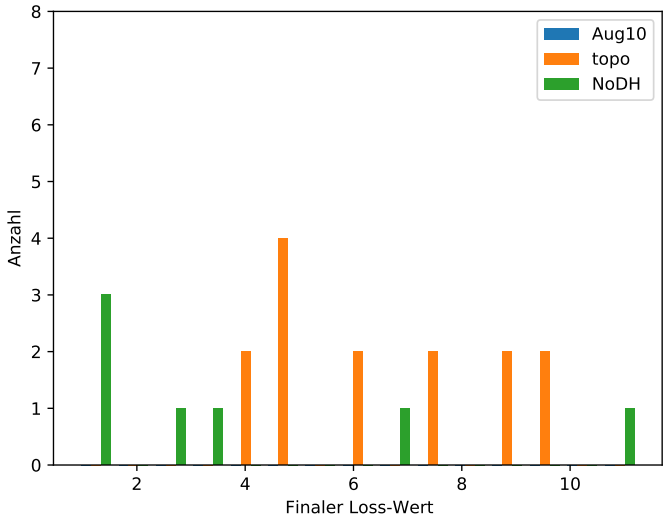

Abbildung 4.17.: Verteilung der finalen Ergebnisse der Loss-Funktionen

4.3.7. Wärmeübergangsgleichung

Auch bei der Approximation der Wärmeübergangsgleichung (Formel 3.8) kam es zu Abbrüchen bei der Durchführung der Experimente aufgrund mathematischer Instabilitäten. Wie im

Diagramm der Abbildung 4.18 deutlich wird, ist insbesondere die Versuchskonfiguration mit dem vervielfachten Datensatzumfang analog dem Versuch mit der Formel des Federpendels von diesem Problem betroffen. Zusätzlich kam es jedoch auch bei der Versuchskonfiguration mit dimensionshomogenen neuronalen Netzen bei fünf der sieben Versuchsdurchläufe zu einem Abbruch des Versuchs aufgrund von Instabilitäten. Die Referenzkonfiguration war lediglich in einem Fall von diesem Problem betroffen.

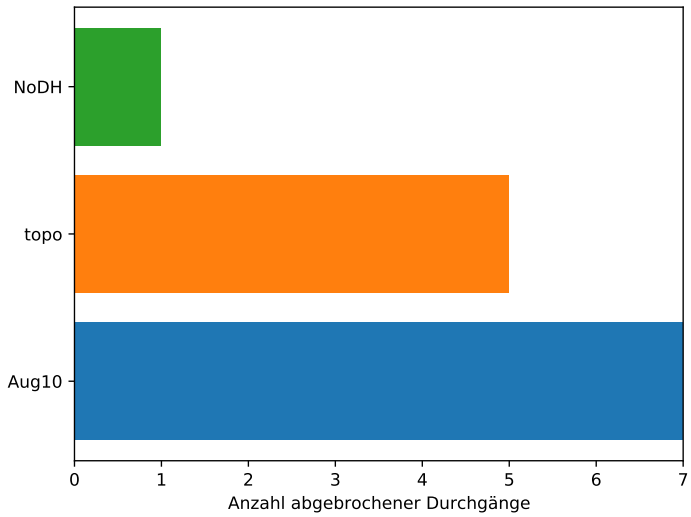

Abbildung 4.18.: Anzahl abgebrochener Trainingsvorgänge

Aufgrund der Tatsache, dass lediglich für die Versuche ohne die Nutzung der Dimensionsinformationen genügend gültige Daten zur Bildung eines Medians der Loss-Funktion über die Epochen vorliegen, wird in Abbildung 4.19 ausschließlich eine Kurve angezeigt.

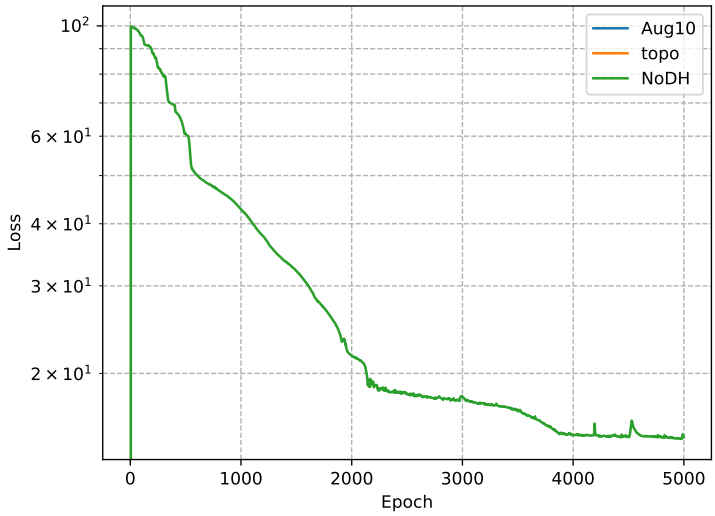

Abbildung 4.19.: Entwicklung der Loss-Funktion über die Epochen

Im in Abbildung 4.20 dargestellten Verteilungsdiagramm werden die neben den finalen Werten der Loss-Funktion für die Referenzkonfiguration des Versuchs auch die verbleibenden gültigen Werte für die dimensionshomogene Konfiguration angezeigt. Die Verteilungen der verschiedenen Versuchskonfigurationen unterscheiden sich sowohl in Bezug auf Genauigkeit als auch auf Streuung. Die Einträge der Referenzkonfiguration verfügen sowohl über die höhere Genauigkeit als auch über eine geringere Streuung im Vergleich zu den Einträgen der Konfiguration zur Erzeugung dimensionshomogener Netze.

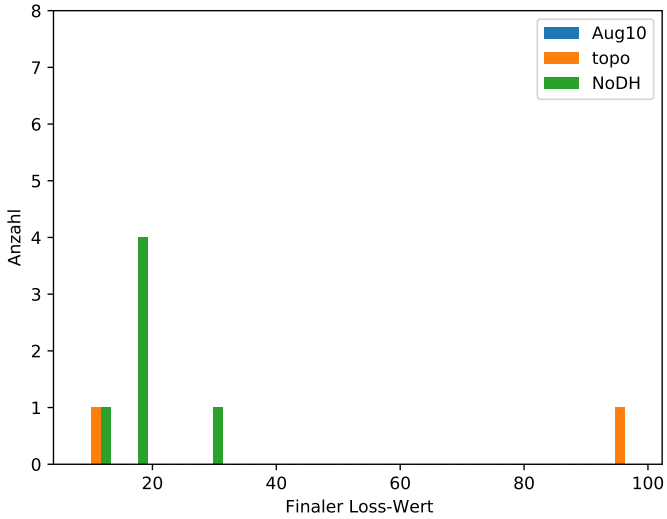

Abbildung 4.20.: Verteilung der finalen Ergebnisse der Loss-Funktionen

4.3.8. Diskussion

Über alle untersuchten Funktionen hinweg betrachtet kann durch die Nutzung der Dimensionsinformationen aus der Dimensionsmatrix die Anzahl der benötigten Epochen zum Erreichen einer festgelegten Genauigkeit reduziert werden. Das Erzwingen der Dimensionshomogenität innerhalb eines neuronalen Netzes führt dabei nicht zwangsläufig zu einem verbesserten Konvergenzverhalten der Loss-Funktion, wie aus den Epoch-Loss-Diagramme für die Gleichungen der Balkenbiegung und Rapididät sichtbar wird.
Das Erzeugen von zu den realen Datenpunkten vollähnlichen Datenpunkten und die damit einhergehende Vervielfachung der Anzahl verfügbarer Datenpunkte resultieren bei gegebener mathematischer Stabilität zuverlässig in einer verminderten Anzahl von benötigten Epochen zur Erreichung einer vorgegebenen Genauigkeit im Vergleich zur genutzten Referenzkonfiguration ohne Nutzung der Dimensionsinformation.
Im direkten Vergleich der beiden Verfahren kann keine generelle Empfehlung für die Nutzung eines der Verfahren abgegeben werden, da die Wirksamkeit der Verfahren entscheidend von der Komplexität der Netztopologie und der Anzahl der realen verfügbaren Datenpunkte abhängt.
Durch das Design der Experimente waren für alle Versuche bereits a priori eine ausreichende Anzahl von Datenpunkten und die optimale Netztopologie garantiert. Aus diesem Grund konnten die beiden Verfahren ihre inhärenten Vorteile im Falle von zu komplexen Netztopologien oder

einer nicht ausreichenden Anzahl von realen Datenpunkten nicht verdeutlichen.
Weiterhin hat sich in den Versuchen gezeigt, dass die Nutzung der Dimensionsinformation sowohl a priori zur Erzeugung dimensionshomogener neuronaler Netze als auch zur Vervielfachung bestehender Datensätze durch vollähnliche Datenpunkte zumindest in den mittels des eigens implementierten Frameworks erzeugten Netzen eine negative Auswirkung auf die mathematische Stabilität hat. Auffällig ist, dass dieser Nachteil insbesondere bei der Verwendung von Funktionen mit großen Gradienten wie der e^x- oder der $ln(x)$-Funktion auftritt. Dieser Umstand lässt darauf schließen, dass die Instabilität durch Exploding Gradients verursacht wird.

5. Fazit

Das vorliegende Kapitel der Arbeit ist zweigeteilt in die beiden Abschnitte „Zusammenfassung“ und „weiterer Ausblick“.
Im Abschnitt 5.1 „Zusammenfassung“ wird die geleistete Arbeit in Form der Entwicklung und Implementierung eines Frameworks zur Erstellung von neuronalen Netzen und die mithilfe des darauf basierenden Programms durchgeführten Experimente einschließlich ihrer Ergebnisse zusammengefasst. Im Abschnitt „Weiterer Ausblick“ werden Anknüpfungspunkte für weitere Forschungsarbeiten und konkrete Verbesserungsvorschläge für das erstellte Programm vorgestellt.

5.1. Zusammenfassung

Die vorliegende Arbeit ging der Frage nach, inwiefern das Wissen um die Dimension einer oder mehrerer phsyikalischen Größen in neuronalen Netzen genutzt werden kann und welchen Einfluss das Nutzen dieses Wissens auf die Leistungsfähigkeit der neuronalen Netze hat.
Hierfür wurde zunächst ein Framework zum Erstellen dimensionshomogener neuronaler Netze konzeptioniert und anschließend basierend auf dem Framework „Tensorflow“ in der Programmiersprache „Python“ realisiert. Ergänzt wurde das Programm zum Erstellen von neuronalen Netzen um Module zum Erzeugen von vollähnlichen Datenpunkten aus realen Datenpunkten und um ein Modul zur Konvertierung eines Modells in einen symbolischen Ausdruck.
Mittels des Programms wurde eine Versuchsreihe durchgeführt. Im Rahmen dieser Versuche wurden die Auswirkungen der Wahl der Fundamentalmatrix zur Erzeugung von dimensionshomogner Neuronaler Netze, die Wirkung von durch vollähnlichen Datenpunkten vergrößerten Datensätzen im Vergleich mit gleichgroßen Datensätzen bestehend aus realen Datenpunkten und der ursprünglichen kleineren Refernzdatensätze untersucht. Im dritten Experiment wurde anhand von insgesamt sieben physikalischen Formeln die mittels des Frameworks erzeugten neuronalen Netze für einen breiten Einsatzbereich validiert und die Verfahren zur Nutzung der Dimensionsinformation miteinander verglichen.
Das Ergebnis dieser Versuche zeigt, dass neuronale Netze vom Nutzen des Wissens um die Dimension von dimensionsbehafteten Eingangsgrößen in der Mehrheit der Fälle durch eine veringerte Anzahl an benötigten Trainigsepochen oder eine höhrere Genauigkeit profitieren. Dieses Wissen kann sowohl zur Erstellung von vollständig dimensionshomogenen neuronalen Netzen als auch zum Erzeugen von zusätzlich zu den bestehenden Datenpunkten vollähnlichen Datenpunkten genutzt werden. Beide Verfahren beeinflussen vor allem das Training des neuronalen Netzes.
In den durchgeführten Experimenten wurde gezeigt, dass die Wahl der Fundamentalmatrix und damit der Netztopologie bzw. der Gewichte im ersten Hidden-Layer und in der Residual Connection einen großen Einfluss auf die Anzahl der benötigten Epochen zum Erreichen einer festgelegten Genauigkeit hat. Die geschickte Wahl einer Fundamentalmatrix kann zu einer gewissen Normalisierung der Eingangsgrößen der Layer mit variablen Gewichten führen. Dies kann das Risiko von Exploding- oder Vanishing-Gradients minimieren. Im Falle der für das Experiment

zu approximierenden Balkenbiegeformel konnte das neuronale Netz durch Exploding-Gradients in Form einer geringeren Anzahl von benötigten Trainingsepochen profitieren.
Des Weiteren konnte anhand der Balkenbiegeformel nachgewiesen werden, dass die Auswirkungen von zusätzlichen vollähnlichen Datenpunkten mit den Auswirkungen von zusätzlichen realen Datenpunkten vergleichbar sind. Für die verwendete Balkenbiegeformel reduzierte sich die Anzahl der benötigten Trainingsepochen bis zum Unterschreiten eines festgelegten Loss-Wertes durch das Verwenden von Trainingsdatensätzen mit dem zehnfachen Umfang des Referenzdatensatzes im Vergleich zu diesem Referenzdatensatz erheblich. Gleichzeitig wird jedoch auch in Abbildung 4.2 deutlich, dass sich dieser Effekt nicht beliebig steigern lässt und das Training von einer erneuten Vervielfachung des Datensatzes um den Faktor zehn nicht erkennbar verbessert wird. Die Auswirkungen von zusätzlichen vollähnlichen Datenpunkten entsprechen denen von zusätzlichen realen oder nativen Datenpunkten.
Im dritten Versuch zeigte sich, dass die durch das implementierte Framework erzeugten neuronalen Netze zur Approximation von verschiedenartigen physikalischen Gleichungen geeignet sind. Neben der bereits in den beiden vorherigen Versuchen genutzten Balkenbiegeformel wurden in diesem Versuch Gleichungen mit $sin(x)-, tanh(x)-, ln(x)-, e^x-$Termen und Polynome erfolgreich approximiert. Im Ergebnis kann keine Empfehlung für eine bestimmte Art der Nutzung der Dimensionsinformation ausgesprochen werden. Sowohl das Erzeugen von vollständig dimensionshomogenen neuronalen Netzen als auch das Generieren von zusätzlichen vollähnlichen Datenpunkten reduziert die Anzahl der zum Unterschreiten eines bestimmten Losswertes benötigten Epochen oftmals. Die Leistungsfähigkeit der Verfahren ist jedoch von der Problemstellung bzw. der zu approximierenden Formel abhängig. Es konnten hierdurch die Erkenntisse aus den beiden vorherigen Versuchen bestätigt werden. Gleichzeitig mussten die Versuche, die die Dimensionsinformationen nutzten, häufiger aufgrund von mathematischen Instabilitäten abgebrochen werden als die Referenzversuche ohne Nutzung der Dimensionsinformationen.

5.2. Weiterer Ausblick

Die vorliegende Arbeit eignet sich als Ausgangspunkt für weitere Forschungsarbeiten.
Nachdem in der Arbeit die Möglichkeiten zur Nutzung der Dimensionsinformationen vorgestellt, untersucht und verglichen wurden, können in einer weiteren Arbeit nach Kombinationsmöglichkeiten und deren Auswirkungen auf das Training und die Genauigkeit der neuronalen Netze untersucht werden. Eine mögliche Anwendung wäre das sogenannte „Transfer-Learning“, bei dem ein neuronales Netz mehrfach trainiert wird. Ein neuronales Netz kann in diesem Fall beispielsweise zunächst durch einen mit vollähnlichen Datenpunkten vervielfachten Datensatz trainiert werden. Im zweiten Schritt kann es in ein dimensionshomogenes Netz verwandelt und im dritten Schritt erneut mittels der nativen Datenpunkte trainiert werden.
Des Weiteren kann der Grund für die Instabilität der neuronalen Netze bei der Anwendung der Dimensionsinformationen erforscht werden, um die Stabilität der erzeugten Netze zu verbessern.
Neben den Arbeiten an den Grundlagen der Nutzung von Dimensionsinformationen in neuronalen Netzen kann zukünftig auch das erzeugte Programm um weitere Funktionen ergänzt werden. Möglichkeiten hierfür sind die Portierung des Programms auf Grafik- oder Tensorprozessoren und das Programmieren einer grafischen Benutzerschnittstelle für den Entwurf von neuronalen Netzen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] *TensorFlow 2.0 is now available!* https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html, . – Accessed: 2020-12-29

[2] Abadi, M. ; Agarwal, A. ; Barham, P. ; Brevdo, E. ; Chen, Z. ; Citro, C. ; Corrado, G. S. ; Davis, A. ; Dean, J. ; Devin, M. ; Ghemawat, S. ; Goodfellow, I. ; Harp, A. ; Irving, G. ; Isard, M. ; Jia, Y. ; Jozefowicz, R. ; Kaiser, L. ; Kudlur, M. ; Levenberg, J. ; Mané, D. ; Monga, R. ; Moore, S. ; Murray, D. ; Olah, C. ; Schuster, M. ; Shlens, J. ; Steiner, B. ; Sutskever, I. ; Talwar, K. ; Tucker, P. ; Vanhoucke, V. ; Vasudevan, V. ; Viégas, F. ; Vinyals, O. ; Warden, P. ; Wattenberg, M. ; Wicke, M. ; Yu, Y. ; Zheng, X. : *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. https://www.tensorflow.org/. Version: 2015. – Software available from tensorflow.org

[3] Aggarwal, C. C. u. a.: *Neural networks and deep learning*. Springer, 2018

[4] Bottou, L. : Large-scale machine learning with stochastic gradient descent. In: *Proceedings of COMPSTAT'2010*. Springer, 2010, S. 177–186

[5] BRIDGMAN, P. : Dimensional Analysis. In: *New Haven; Connecticut* (1922)

[6] Carpenter, K. A. ; Cohen, D. S. ; Jarrell, J. T. ; Huang, X. : Deep learning and virtual drug screening. In: *Future medicinal chemistry* 10 (2018), Nr. 21, S. 2557–2567

[7] Chollet, F. u. a.: *Keras*. https://keras.io, 2015

[8] de Myttenaere, A. ; Golden, B. ; Le Grand, B. ; Rossi, F. : Mean Absolute Percentage Error for regression models. In: *Neurocomputing* 192 (2016), 38-48. http://dx.doi.org/https://doi.org/10.1016/j.neucom.2015.12.114. – DOI https://doi.org/10.1016/j.neucom.2015.12.114. – ISSN 0925–2312. – Advances in artificial neural networks, machine learning and computational intelligence

[9] Deru, M. ; Ndiaye, A. : *Deep Learning mit TensorFlow, Keras und TensorFlow.js: Über 450 Seiten Einstieg, Konzepte, KI-Projekte. Aktuell zu TensorFlow 2*. Rheinwerk Verlag GmbH (Rheinwerk Computing). https://books.google.de/books?id=T3KizAEACAAJ. – ISBN 9783836274258

[10] DIN: *Einheiten – Teil 1: Einheitennamen, Einheitenzeichen*. Beuth Verlag, Berlin, 2010

[11] Foody, G. ; McCulloch, M. ; Yates, W. : The effect of training set size and composition on artificial neural network classification. In: *International Journal of Remote Sensing* 16 (1995), Nr. 9, S. 1707–1723

[12] Glorot, X. ; Bengio, Y. : Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, S. 249–256

[13] Görtler, H. : *Dimensionsanalyse: Theorie der physikalischen Dimensionen mit Anwendungen.* Springer Berlin Heidelberg (Ingenieurwissenschaftliche Bibliothek Engineering Science Library). `https://books.google.de/books?id=iCrlAgAACAAJ`. – ISBN 9783540069379

[14] Guennebaud, G. ; Jacob, B. u. a.: *Eigen v3.* http://eigen.tuxfamily.org, 2010

[15] He, H. : The State of Machine Learning Frameworks in 2019. In: *The Gradient* (2019)

[16] He, K. ; Zhang, X. ; Ren, S. ; Sun, J. : *Deep Residual Learning for Image Recognition.* 2015

[17] Hochreiter, S. : The vanishing gradient problem during learning recurrent neural nets and problem solutions. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (1998), Nr. 02, S. 107–116

[18] Jarosz, Q. : *Neuron.* `https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg`, 2020. – [Online; accessed 02-December-2020]

[19] Jouppi, N. P. ; Young, C. ; Patil, N. ; Patterson, D. : A domain-specific architecture for deep neural networks. In: *Communications of the ACM* 61 (2018), Nr. 9, S. 50–59

[20] Kingma, D. P. ; Ba, J. : *Adam: A Method for Stochastic Optimization.* 2017

[21] Klein, B. ; Rossin, D. : Data quality in neural network models: effect of error rate and magnitude of error on predictive accuracy. In: *Omega* 27 (1999), Nr. 5, 569 - 582. `http://dx.doi.org/https://doi.org/10.1016/S0305-0483(99)00019-5`. – DOI https://doi.org/10.1016/S0305–0483(99)00019–5. – ISSN 0305–0483

[22] Kukacka, J. ; Golkov, V. ; Cremers, D. : Regularization for Deep Learning: A Taxonomy. In: *CoRR* abs/1710.10686 (2017). `http://arxiv.org/abs/1710.10686`

[23] Kumar, S. K.: *On weight initialization in deep neural networks.* 2017

[24] Lipton, Z. C.: *The Mythos of Model Interpretability.* 2017

[25] Mangal, R. ; Nori, A. V. ; Orso, A. : *Robustness of Neural Networks: A Probabilistic and Practical Approach.* 2019

[26] McCulloch, W. S. ; Pitts, W. : A logical calculus of the ideas immanent in nervous activity. In: *The bulletin of mathematical biophysics* 5 (1943), Nr. 4, S. 115–133

[27] Merzinger, W. : *Repetitorium Höhere Mathematik.* Binomi Verlag, 2010

[28] Paszke, A. ; Gross, S. ; Massa, F. ; Lerer, A. ; Bradbury, J. ; Chanan, G. ; Killeen, T. ; Lin, Z. ; Gimelshein, N. ; Antiga, L. ; Desmaison, A. ; Kopf, A. ; Yang, E. ; DeVito, Z. ; Raison, M. ; Tejani, A. ; Chilamkurthy, S. ; Steiner, B. ; Fang, L. ;

Bai, J. ; Chintala, S. : PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach, H. (Hrsg.) ; Larochelle, H. (Hrsg.) ; Beygelzimer, A. (Hrsg.) ; Alché-Buc, F. d'(Hrsg.) ; Fox, E. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 32, Curran Associates, Inc., 8026–8037

[29] Pitts, W. ; McCulloch, W. S.: How we know universals the perception of auditory and visual forms. In: *The Bulletin of mathematical biophysics* 9 (1947), Nr. 3, S. 127–147

[30] Polyak, B. T.: Some methods of speeding up the convergence of iteration methods. In: *USSR Computational Mathematics and Mathematical Physics* 4 (1964), Nr. 5, S. 1–17

[31] Riedmiller, M. ; Braun, H. : A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: *IEEE international conference on neural networks* IEEE, 1993, S. 586–591

[32] Rudolph, S. : Übertragung von Ähnlichkeitsbegriffen.

[33] Rudolph, S. : On a genetic algorithm for the selection of optimally generalizing neural network topologies. In: *Proceedings of the 2nd International Conference on Adaptive Computing in Engineering Design and Control* Bd. 96 Citeseer, 1996, S. 79–86

[34] Rudolph, S. : On topology, size and generalization of non-linear feed-forward neural networks. In: *Neurocomputing* 16 (1997), Nr. 1, S. 1–22

[35] Saar-Tsechansky, M. ; Provost, F. : Handling missing values when applying classification models. In: *Journal of machine learning research* 8 (2007), Nr. Jul, S. 1623–1657

[36] Taylor, L. ; Nitschke, G. : *Improving Deep Learning using Generic Data Augmentation.* 2017

[37] Tieleman, T. ; Hinton, G. : *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude.* COURSERA: Neural Networks for Machine Learning, 2012

A. Anhang

A.1. Schnittstellen

A.1.1. Konfigurationsdatei

```
    {
    "name_experiment": "WaermeUebergang",
    "path_experiment": "./Versuche/WaermeUebergang",
    "path_function_file": "./Versuche/WaermeUebergang/real_function.py",
    "y_name": "q",
    "x_names": [
        "lambda",
        "upsilon",
        "ra",
        "ri"
    ],
    "path_dimensionsmatrix_file": "./Versuche/WaermeUebergang/Dimensionsmatrix.csv",
    "path_topology_file": "./Versuche/WaermeUebergang/Topologie.csv",
    "path_data_file": "./Versuche/WaermeUebergang/Data.csv",
    "number_datapoints": 25,
    "noise_level": 0,
    "data_augmentation_factor": 1,
    "path_latex_file": "./Versuche/WaermeUebergang/latexFormula.txt",
    "path_dc43_file": "./Versuche/WaermeUebergang/dc43Formula.txt",
    "log_level": 1,
    "loss_function": "MAPE",
    "metrics": [
        "MAPE",
        "MSE"
    ],
    "max_epochs": 3000,
    "optimizer": "ADAM",
    "batch_size": 32,
    "split_ratio": {"train": 0.6, "val": 0.2, "test": 0.2},
    "start_tensorboard": "True",
    "dimensionhomogenous": false,
    "early_stopping_limit": 20
}
```

A.1.2. Topologiedatei

```
    Type;fn;Name;Input;
piLayer;;Pilayer;None;0
shortcut;;Shortcut;None;1
functionalBlock;log;BlockLog;0;2
functionalBlock;lin;Block1;0;3
concat;;Concat;2,3;4
functionalBlock;lin;Block2;4;5
multiply;;Multi1;1,5;6
finalLayer;;Final;6;7
```

A.2. Erzeugte Modelle

Beispielhaft einige neuronale Netze die mittels Tensorflow erstellt wurden

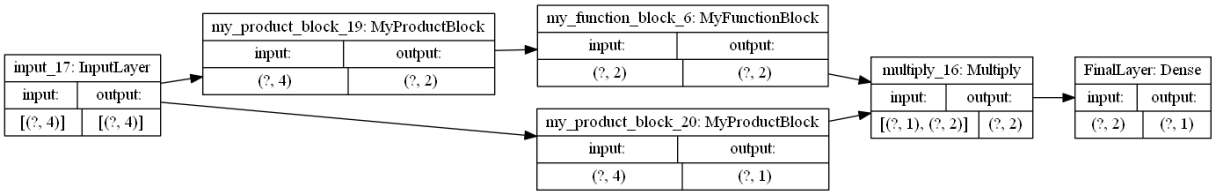

Abbildung A.1.: Netztopologie Versuch Balkenbiegeformel

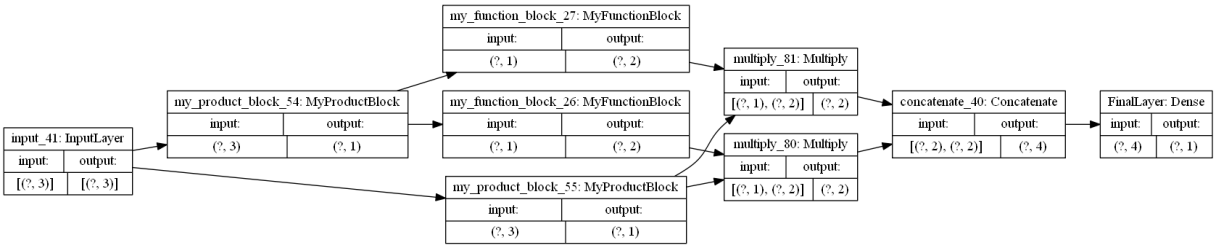

Abbildung A.2.: Netztopologie Versuch Senkrechter Wurf

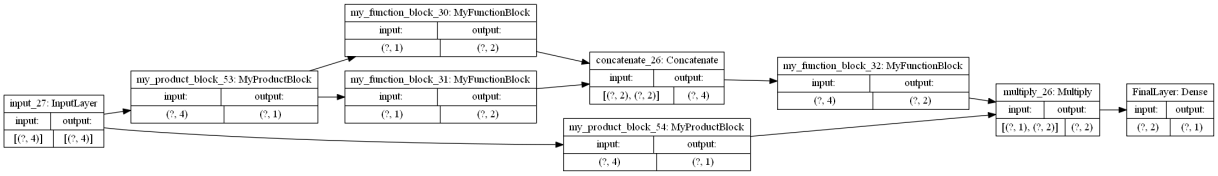

Abbildung A.3.: Netztopologie Versuch Wärmeübertragung

A.3. Inhalt der CD

Die CD enthält neben den Rohdaten der durchgeführten Versuche in einem Zip-Archiv im Ordner „Versuche“ auch den Programmcode der Implementierung der Entwurfssprache zum Erzeugen dimensionshomogener Netze im Ordner „Code“ und die Daten der schriftlichen Ausarbeitung im Ordner „Ausarbeitung“.
Die Daten im Ordner „Code“ enthalten sowohl die Python-Klassen des Programms als auch die zur Ausführung benötigte „NewMain.py“-Datei. Die „.py“-Dateien werden noch um ein Powershellskript mit dem Namen „startTB“ zum Start von Tensorboard und ein Matlabskript zur Verarbeitung der Modellinterpretation in Matlab ergänzt.
Der Ordner „Ausarbeitung“ enthält neben der Arbeit im pdf-Format die Rohdaten der Ausarbeitung im Latex-Format.

A.4. Validierung Datengenerator

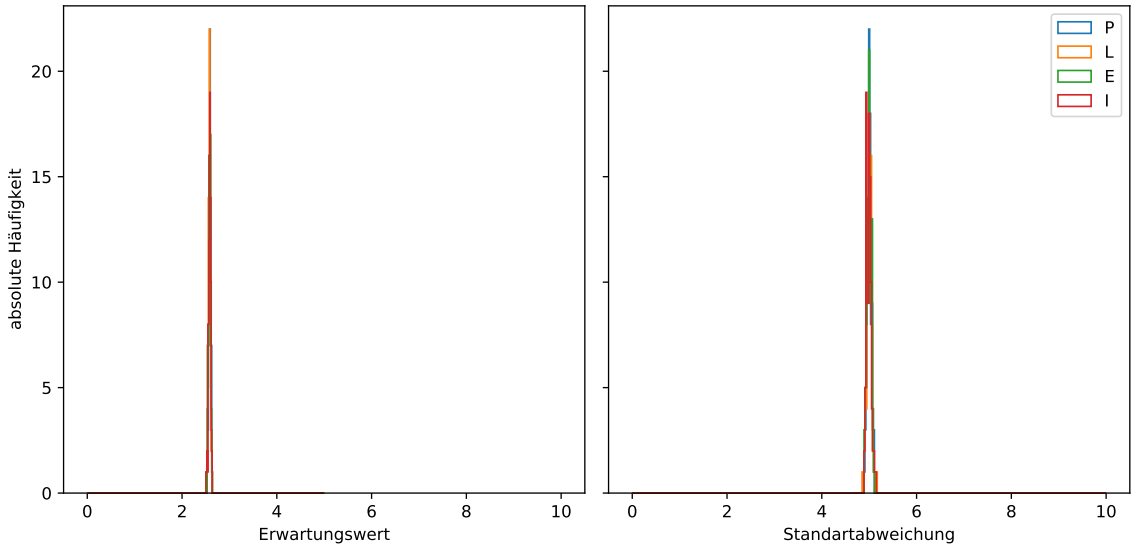

Abbildung A.4.: Verteilung der Erwartungswerte und Standartabweichungen erzeugter Datensätzen

Eidesstattliche Erklärung und Urheberrecht

Hiermit räume ich, Dreßler, Christian, Matrikelnummer 2785103 der Universität Stuttgart, Institut für Flugzeugbau, ein kostenloses, zeitlich und räumlich unbeschränktes, einfaches Nutzungsrecht an der von mir erstellten Masterarbeit mit dem Titel

Symbolische und numerische Untersuchung dimensionshomogener Neuronaler Netze unter den Gesichtspunkten Lernfähigkeit und Genauigkeit

und den im Rahmen dieser Arbeit entstandenen Arbeitsergebnissen ein. Ich erkläre, die Arbeit selbständig verfasst und bei der Erstellung dieser Arbeit die einschlägigen Bestimmungen, insbesondere zum Urheberrechtsschutz fremder Beiträge, eingehalten zu haben. Soweit meine Arbeit fremde Beiträge (z.B. Bilder, Zeichnungen, Textpassagen) enthält, erkläre ich, dass diese Beiträge als solche gekennzeichnet sind (z.B. Zitat, Quellenangabe) und ich eventuell erforderlich gewordene Zustimmungen der Urheber zur Nutzung dieser Beiträge in meiner Arbeit eingeholt habe. Für den Fall der Verletzung Rechter Dritter durch meine Arbeit, erkläre ich mich bereit, der Universität Stuttgart einen daraus entstehenden Schaden zu ersetzen bzw. die Universität Stuttgart auf deren Aufforderung von eventuellen Ansprüchen Dritter freizustellen.
Das der Universität Stuttgart hiermit eingeräumte Nutzungsrecht erstreckt sich auf sämtliche bekannte Nutzungsarten und umfasst neben dem Recht auf Nutzung der Arbeitsergebnisse in Forschung, Lehre und Studium, insbesondere das Recht der Vervielfältigung und Verbreitung, das Recht zur Bearbeitung und Änderung inklusive Nutzung, Vervielfältigung und Verbreitung der dabei entstehenden Ergebnisse, sowie insbesondere das Recht der öffentlichen Zugänglichmachung im Internet sowie das Recht der Weiterübertragung auf einen Dritten ohne meine erneute Zustimmung.
Mir ist bekannt, dass die Einräumung des Nutzungsrechts der öffentlichen Zugänglichmachung auch beinhaltet, dass mein Name im Zusammenhang mit dem Titel der oben genannten Arbeit auf den Webseiten der Universität Stuttgart genannt werden kann. Mir ist auch bekannt, dass – sofern meine Arbeit selbst nicht im Internet zugänglich gemacht wird – die Einräumung des Nutzungsrechts der öffentlichen Zugänglichmachung auch umfasst, dass die Universität Stuttgart auf ihren Webseiten meinen Namen im Zusammenhang mit dem Titel der oben genannten Arbeit (z.B. in Listen über am Institut abgeschlossene studentische Arbeiten) nennen kann.
Außerdem übertrage ich der Universität Stuttgart das Eigentum an einem von mir der Bibliothek des Instituts für Flugzeugbau kostenlos zur Verfügung gestellten Exemplars meiner oben genannten Arbeit und räume der Universität Stuttgart auch für dieses Exemplar die oben genannten Nutzungsrechte ein.

Ort, Datum	Christian Dreßler