TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Datenbanken

Masterarbeit

ERKENNUNG VON FEM-GEOMETRIEN MIT METHODEN DES MASCHINELLEN LERNENS

Nick Scheider

Matr.-Nr.: 4051545

Betreut durch:

Prof. Dr.-Ing. Wolfgang Lehner

Dr.-Ing. Maik Thiele

Eingereicht am 20.03.2020

ERKLÄRUNG

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.
Zur Verfügung gestellte Daten der externen Projektpartner habe ich gemäß der abgeschlossenen Sicherheitsvereinbarung entsprechend behandelt und Bezeichner, Abbildungen usw. werden in der Arbeit nur in sofern verwendet, dass sich keine Rückschlüsse auf die direkten Daten ziehen lassen.

Dresden, 20.03.2020

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik
Institut für Systemarchitektur
Professur für Datenbanken
Prof. Dr.-Ing. Wolfgang Lehner

Aufgabenstellung für die Masterarbeit

Name, Vorname des Studenten: Scheider, Nick

Immatrikulationsnummer: 4051545

Erkennung von FEM-Geometrien mit Methoden des maschinellen Lernens

Kurzfassung: Bei der Entwicklung von Fahrzeugkarosserien werden Blechteile häufig mittels Schweißpunkten verbunden. Aktuell werden diese noch manuell von Ingenieuren erstellt. Bei üblicherweise zwischen 7-12.000 Schweißpunkten für eine Karosserie, ist dies ein aufwendiger und fehleranfälliger Prozess. Das Ermitteln eines zutreffenden Schweißpunktdesigns (Anordnung der Schweißpunkte auf zusammengehörigen Bauteilen) erfordert zudem Ingenieure mit ausreichend Erfahrung. Zusätzlich existiert keine aktuelle Norm bei der Benennung von Bauteilen und das Finden ähnlicher Bauteil-Konstruktionen, ist nur erschwert möglich.

Ziel dieser Arbeit ist, einen konzeptionellen Ansatz zur Verarbeitung von FEM-Daten zu erarbeiten und Methoden aus dem Bereich des maschinellen Lernens zur Geometrie-erkennung bestehender Konstruktionen miteinander zu vergleichen. Schwerpunkt liegt dabei vor allem auf der ausführlichen Bewertung existierender Ansätze.

Nachfolgende Aufgaben sollen in der mit der Arbeit gelöst werden:

- Recherche zu existierenden Ansätzen zur Geometrieerkennung im Bereich des maschinellen Lernens
- Entwickeln eines Gesamtkonzepts zur Erkennung von Bauteilen
- Bewertung und Gegenüberstellung existierender Ansätze zur Geometrieerkennung, anhand von durchzuführenden Benchmarks
- prototypische Implementierung der Geometrieerkennung

Betreuender Hochschullehrer:	Prof. Dr.-Ing. Wolfgang Lehner
Betreuender Mitarbeiter:	Dr.-Ing. Maik Thiele Dipl.-Ing. Marko Thiele (SCALE GmbH)
Institut:	Systemarchitektur
Professur:	Datenbanken
Beginn am:	im Zeitraum 21.10.2019 – 23.03.2020

Verteiler: Student
HSL
Betreuer
Prüfungsamt

Unterschrift des betreuenden Hochschullehrers

1

KURZBESCHREIBUNG

Im Bereich der rechnergestützten Entwicklung im Fahrzeug- und Karosseriebau werden große Datenmengen mittels kostenintensiven Simulationsverfahren verarbeitet. Mit Hilfe dieser Daten und aktuellen Algorithmen im Bereich des maschinellen Lernens kann eine Unterstützung und Verbesserung der Fertigungsprozesse erzielt und damit eine allgemeine Steigerung der Effizienz erreicht werden.

Die vorliegenden Arbeit beschäftigt sich mit der Erkennung und Klassifizierung von Fahrzeugbauteilen in Form von FEM-Daten. Es wird eine Prozesskette beschrieben, die es ermöglicht aus vorliegenden FEM-Daten die 3D-Modelle der Fahrzeugteile zu extrahieren. Diese 3D-Modelle dienen als Datengrundlage für eine Machbarkeitsstudie zur Klassifizierung von aktuellen Deep-Learning-Architekturen, wovon zwei Vertreter gegenübergestellt werden. Mittels mehrere Benchmarks werden beide Ansätze evaluiert und Rückschlüsse auf die verwendeten Daten gezogen. Anhand der Ergebnisse und der zuvor analysierten Anwendungsfälle wird die Umsetzbarkeit einer solchen Anwendung untersucht.

Zusätzlich wird eine prototypische Umsetzung einer möglichen Prozesskette zur Klassifikation vorgestellt. Dafür wurde ein bereits trainiertes neuronales Netz in eine CAD-Software integriert, ein Anwendungsfall erstellt und prototypisch implementiert. Die entstandene Anwendung dient zur Visualisierung der erzielten Ergebnisse. Zudem wird über die Realisierbarkeit weiterer Anwendungsfälle diskutiert.

INHALTSVERZEICHNIS

1 EINLEITUNG

Mit dem Beginn der Industrialisierung im 18. Jahrhundert veränderte sich die Arbeit von vielen Menschen schlagartig. Maschinen wurden konstruiert, Fabriken gebaut und Autos fuhren auf den Straßen. Schwierige Aufgaben wurden nun nicht mehr von Menschen übernommen, sondern immer weiter durch Maschinen automatisiert. Dieser Wandel hatte zur Folge, dass die Produktivität stieg und somit die Wirtschaft boomte.
Das exakt gleiche Prinzip erleben wir heute in Zeiten der Digitalisierung erneut. Kommunikationswege verkürzen sich, aufwendige Berechnungen werden abgenommen und riesige Mengen an Informationen können kompakt gespeichert werden. All diese Eigenschaften bieten dem Menschen enorme Vorteile in der Arbeitswelt. Seit einigen Jahren wächst zudem ein neuer Bereich immer rasanter: Das maschinelle Lernen. Der Mensch nutzt jetzt nicht nur die positiven Eigenschaften, die die Digitalisierung mit sich bringt, sondern versucht die eigenen Denk- und Lernprozesse zu abstrahieren und Computern beizubringen, um weiteres Automatisieren zu ermöglichen. Mittlerweile ist es möglich, verschiedenste Aufgaben wie Objekterkennung, Klassifikation und Zeitreihenvorhersage durch Computer zu automatisieren. Die daraus resultierenden Möglichkeiten sind endlos, weshalb die Forschung in den letzten Jahren einen enormen Zuwachs bekam. Immer mehr Anwendungen machen sich das maschinelle Lernen zu Nutze. Große Konzerne wie Google, Amazon und Facebook analysieren und verarbeiten damit Unmengen an Daten und verbessern somit ihre Produkte. Auch in der Automobilindustrie ist die künstliche Intelligenz bereits angekommen. Fahrassistenzsysteme, wie beispielsweise die Einparkhilfe, erleichtern immer weiter die Benutzung von Fahrzeugen. Auch das autonome Fahren scheint, durch die stabilen Erkennungsalgorithmen im 2D-Bereich, in greifbarer Zukunft zu liegen.
Dennoch existieren einige Wirtschaftszweige, die nach wie vor nicht von diesem aktuellen Trend profitieren können. Besonders in der Produktion im Bereich CAE (*Computer Aided Engeneering*, zu deutsch: rechnergestützte Entwicklung) der Automobilindustrie werden viele Aufgaben von Menschenhand gelöst. Da in der rechnergestützten Entwicklung Computer und Simulationsverfahren bereits große Anwendung finden, kann mit Hilfe der entstehenden Daten und verschiedenen Verfahren der künstlichen Intelligenz eine weitere Unterstützung für die Anwender realisiert werden.
Im Erstellungsprozess für Fahrzeugkarosserien gibt es einige Anwendungsfelder, die durch die Verwendung von maschinellen Lernalgorithmen eine Steigerung der Effizienz bestehender Prozessketten erfahren könnten. Besonders das Erkennen und Klassifizieren von Fahrzeugbauteilen

anhand von 3D-Daten bietet viele Möglichkeiten, solche Prozesse zu verbessern. Im Folgenden werden einige Anwendungsfälle geschildert, die mittels einer automatischen Klassifikation realisiert werden können.

Automatisiertes Bennennungssystem In der Industrie ist es typisch, dass Fahrzeugteile eine spezielle Bezeichnung, je nach Typ, Material und Version besitzen. Diese Namen sind jedoch nicht uniform und werden nur nach einem groben Konzept und einer Teilenummer ausgewählt. Durch eine Bauteilklassifikation kann dem Nutzer, während der Erstellung des neuen Bauteils, bereits eine Bezeichnung vorgeschlagen werden. Dieses Vorschlagssystem hilft dabei, eine einheitliche Struktur der Bezeichnungen zu erreichen und dem Nutzer den Namensfindungsprozess zu erleichtern.

Automatisiertes Einsortieren in Bauteillisten Im Fertigungsprozess von Fahrzeugen entstehen sogenannte Bauteillisten. Diese sind hierarchischer Struktur und werden aktuell noch von Hand befüllt. Ein Klassifikationssystem kann diese Aufgabe komplett automatisieren und dabei den Prozess um ein Vielfaches beschleunigen.

Finden ähnlicher Konstruktionen Mit Hilfe einer Bauteilerkennung ist es möglich, dem Anwender bereits bekannte und ähnliche Bauteile vorzuschlagen. Dieses Empfehlungssystem kann dazu dienen, dass bereits bekannte Konstruktionsprobleme frühzeitig erkannt werden und gegebenenfalls mögliche Lösungsansätze vorgeschlagen werden können. Zusätzlich kann durch das Finden von Gleichteilen der Produktionsprozess beschleunigt und Geld gespart werden, da für die neuen Teile keine neuen Produktionsmaschinen gebaut werden müssen. Außerdem kann durch die vorgeschlagene Empfehlung ein Experte beim Designprozess unterstützt werden.

Segmentierung von Bauteilgruppen Im Erstellungsprozess von Fahrzeugen ist es oft üblich, dass zuerst großflächig ganze Bauteilgruppen erstellt werden. Diese müssen dann in einem weiteren Prozess in kleinere Einzelteile zerlegt werden. Durch eine Segmentierung dieser Fläche, mittels neuronaler Netze in bekannte Bauteilklassen, kann dieser Prozess beschleunigt und automatisiert werden.

Expertensystem zur Schweißpunktvorhersage Das Setzen von Schweißpunkten ist eine Aufgabe, die nur von einem Experten durchgeführt werden kann. Dabei werden pro Schweißpunktanordnung teure Simulationen gestartet, um die jeweilige Anordnung zu testen und zu evaluieren. Eine Bauteilerkennung in Verbindung mit einer Wissensdatenbank, in der alle Schweißpunktverteilungen gespeichert sind, kann den Experten dabei unterstützen, schnell eine valide Verteilung zu wählen und so einige aufwendige Simulationsdurchläufe einzusparen.

Diese Anwendungsfälle dienen als Motivation für die nachfolgenden Untersuchungen aktueller Ansätze im Bereich der 3D-Geometrie-Erkennung. Es soll untersucht werden, ob eine Klassifikation von Fahrzeugbauteilen mittels Methoden des maschinellen Lernens möglich ist und somit ein Grundstein, für die Realisierung der hier angerissenen Anwendungsfälle, gesetzt werden kann.

Ziele an die Arbeit

Ziel der Arbeit ist es, zu Untersuchen, inwiefern Methoden des maschinellen Lernens die Prozesse des Fahrzeugdesigns verbessern können. Dabei sollen vorhandene Daten so aufbereitet werden, dass sie flexibel verwendbar für unterschiedliche ML-Algorithmen sind. Der Schwerpunkt dieser Arbeit liegt auf dem ausführlichen Vergleich und der Bewertung existierender Ansätze. In Kapitel 2 werden dafür die Grundlagen der vorliegenden Arbeit erläutert. Danach wird in Kapitel 3 ein Überblick über die verwandten Arbeiten gegeben. In Kapitel 4 erfolgt die Konzeption eines Ablaufs zur Datenvorverarbeitung. Weiterhin findet in Kapitel 5 ein Vergleich verschiedener Ansätze statt. Die daraus resultierenden Benchmarks in Kapitel 6 stellen dann die Ergebnisse der Machbarkeitsstudie dar. Die gilt es anschließend prototypisch umzusetzen, um einen Anwendungsfall anschaulich darzustellen. Dazu wird im Kapitel 7 die Implementierung eines Prototyps beschrieben und Ausblick auf weitere Konzepte anderer Anwendungsfälle gegeben.

2 GRUNDLAGEN

Dieser Abschnitt beschreibt die fundamentalen Methoden des Simulationsprozesses, Grundlagen des maschinellen Lernens mit besonderen Fokus auf neuronale Netze sowie Konzepte der Datenvorverarbeitung.

2.1 SIMULATIONSPROZESS MIT FINITE ELEMENT METHODE

Wie in Kapitel 1 angedeutet sind Simulationsverfahren ein wichtiger Bestandteil des Entstehungsprozesses in der Fahrzeugindustrie. Ein wichtiger Vertreter ist die Finite-Element-Methode. Dabei handelt es sich um ein nummerisches Verfahren, dass in vielen verschiedenen Anwendungsbereichen zu finden ist und für zahlreiche Berechnungsaufgaben im Maschinen- und Fahrzeugbau eingesetzt wird. Mittels realitätsnaher Ergebnisse durch Rechnersimulationen trägt die Finite-Elemente-Methode (abk. FEM) [Kle13] zu einer Verkürzung der Produktentwicklungszeit bei, da vor allem aufwendige Crashtests entfallen. Dennoch bedeuten derartige Simulationen viel Rechenaufwand und sind sehr zeitintensiv. Aus diesem Grund kann eine zusätzliche Unterstützung des Simulationsprozesses durch Methoden des maschinellen Lernens eine weitere Verbesserung dieser Entwicklungszeit bedeuten.

Dazu muss analysiert werden, wie genau der Aufbau eines solchen Simulationsprozesses aussieht. Im Grundlegenden besteht der FEM-Prozess aus drei Teilen dem Präprozessor, dem Gleichungslöser oder auch Solver und dem Postprozessor. Dabei wird nach dem bekannten EVA-Prinzip gearbeitet, einem Grundprinzip der Datenverarbeitung, dass einen solchen Vorgang in Eingabe, Verarbeitung und Ausgabe unterteilt und beschreibt. Da die FE-Methode in einem engen Zusammenwirken mit CAD (Computer-Aided Design) steht, kann man diese Prozesskette um einen weiteren Vorverarbeitungsschritt erweitern. Die insgesamt vier Abschnitte werden im Folgenden einzeln vorgestellt. In Abbildung 2.1 ist diese Prozesskette mit jeweiligen Schwerpunkten der Bereiche grafisch dargestellt.

Konstruktion mit CAD

Dieser Bereich stellt den ersten Vorverarbeitungsschritt dar. Dabei erstellt der Ingenieur aus CAD-Bauteilen vereinfachte Analysemodelle, zum Teil durch das Herauslösen der Hauptgeometrie, um unwichtige Bestandteile aus den aufwendigen Berechnungen zu entfernen. Diesen Schritt nennt man auch Problemaufbereitung. Die dadurch entstehenden CAD-Daten werden mittel einer Schnittstelle an die FEM-Prozesskette weitergereicht. Mittlerweile besitzen auch einige CAD-Programme teile eines Präprozessors und Postprozessors.

Präprozessor

Im Präprozessor werden aus den übergebenen CAD-Daten, unter Angabe von Netzparametern (Elementtyp und Elementanzahl), mittels verschiedener Vernetzungsalgorithmen die Finiten Elemente erzeugt. Durch Hinzugabe von weiteren Metadaten wie Materialart, Werkstoffkennwerte und Kräfte kann mit Hilfe der mechanischen Werkstoffanalyse das Materialverhalten simuliert werden. Wichtige Designentscheidungen werden anhand der Ergebnisse der Simulationen getroffen, deshalb ist genau an diesem Punkt eine große Expertise seitens des Anwenders notwendig. Dieses Expertenwissen kann durch Methoden der künstlichen Intelligenz gesammelt, aufbereitet und den Anwendern als zusätzliche Unterstützung für Designentscheidungen präsentiert werden.

Solver

Der Solver oder auch Gleichungslöser berechnet die verschiedenen Simulationen, unter Berücksichtigung aller Metadaten aus dem vorhergehenden Schritt. So werden unter anderem die Verschiebung der Bauteile, die wirkenden Spannungen und die Reaktionskräfte berechnet. Da die FE-Methode nummerisch ist, handelt es sich um ein iteratives Verfahren, welches nach einer bestimmten Anzahl an Schritten konvergiert. Diese Näherung stellt dann das Ergebnis der Simulation dar.

Postprozessor

Durch den Postprozessor werden die erhaltenen Ergebnisse des Solvers ausgeben und verständlich für den Anwender visualisiert. Dabei werden insbesondere Spannungs- und Dehnungswerte des Bauteils als Falschfarbenbild dargestellt, um interessante Bereiche hervorzuheben.

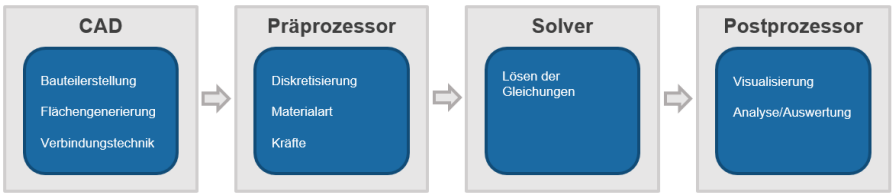


Abbildung 2.1: Pipeline des Simulationsprozesses im CAE-Bereich

2.2 STRUKTUR VON FEM-DATEN

Mittlerweile existieren in der Industrie viele Programme und Anwendungen, welche die Finite-Elemente-Methode realisieren [Mat10]. Unter anderem bieten auch einige CAD-Programme erste Vernetzungsalgorithmen an und nehmen so Teilbereiche des Präprozessors ab. Die Softwareindustrie findet weltweit Abnehmer und kommt auf ein Gesamtjahresumsatz von mehreren Milliarden US-Dollar [Roy01]. Ein wichtiger Vertreter ist unter anderem das Stand-Alone-Programm LS-Dyna [LD], welches von der Livermore Software Technology Corporation (LSTC) entwickelt wurde. Durch die Vielzahl an Programmen gibt es auch viele verschiedene Dateitypen, die alle ihr eigenes Format besitzen. Im Folgenden wird die Struktur zweier FEM-Dateitypen genauer vorgestellt. Diese werden später in der Arbeit zur Weiterverarbeitung benutzt und stellen die Grundlage der Untersuchungen dar.

LS-Dyna Dateien

LS-Dyna ist eine Simulationssoftware auf Basis der Finite-Elemente-Methode, welches von der LSTC entwickelt wurde. In Deutschland wird die Software nicht direkt über die LSTC vertrieben, sondern über einen gesonderten Ableger wie beispielsweise der DYNAmore [TEC07]. Um mit LS-Dyna komplexe Berechnungen durchführen zu können, muss der Benutzer eine spezielle Eingabedatei einlesen, die sogenannten Inputdecks. Diese Datei verfolgt eine bestimmte Struktur und wird mit der .key-Endung abgespeichert.
Das Grundkonzept der Inputdecks stellen sogenannte *Keywords* dar. Diese Schlüsselwörter werden durch ein führendes *-Symbol gekennzeichnet und signalisieren den Beginn eines dazugehörigen, tabellarisch aufgebauten Datenblocks. Das darauf folgende Keyword stellt zugleich das Ende des vorhergehenden Datenblocks dar. Durch diesen Aufbau entsteht eine flexible und logisch organisierte Datenbankstruktur, die für den Anwender leicht zu verstehen ist. Insgesamt gibt es knapp 40 verschiedene Keywords, womit eine Vielzahl an möglichen Parametereinstellungen definiert werden kann. Für die Arbeit sind besonders Datenblöcke mit Bezug auf die 3D-Repräsentation der Bauteile interessant, dazu zählen unter anderem:

- ***PARTS** - Definition eines Bauteils mit Verweis auf Materialien
- ***ELEMENTS** - Vereinigung aus mehreren Knoten; unterscheidbar in Stab-, Schalen- und Volumenelemente
- ***NODE** - Datenstruktur für einfache Knoten mit 3D-Koordinaten
- ***MAT** - Bereich zum Definieren bestimmter Materialien und deren Parametern

Abbildung 2.2 zeigt eine vereinfachte Struktur eines LS-Dyna Dateien mit dazugehörigen Spaltennamen und Verweisen der jeweiligen ID's. Ein **NODE* besteht aus einer *NID* und seinen Koordinaten *X, Y, Z*. Mehrere Knoten bilden ein **ELEMENT*, welches auch eine *EID* besitzt und zusätzlich auf einen **PART* verweist. Dabei richtet sich die Anzahl der Knoten eines **ELEMENTs* nach dem jeweiligen verwendeten Typ, beispielsweise Schalen-, Stab- oder Volumenelemente. Fahrzeugbauteile werden vor allem mit Schalenelementen beschrieben, dabei besteht ein Element entweder aus drei Knoten (Dreiecksnetz) oder aus vier Knoten (Viereckснetz). Der **PART*

repräsentiert ein Bauteil und wird durch eine eindeutige *PID* beschreiben, zusätzlich besitzt er Verweise auf unterschiedliche Keywords, unter anderem auf das verwendete Material **MAT*. In **MAT* werden die Materialien durch deren exakte Parameter beschrieben.

```
*PART
aluminum block
$--------+---------+---------+---------+---------+---------+---------+---------+
$       PID     SECID       MID     EOSID      HGID      GRAV    ADPOPT      TMID
         1         1         1
*SECTION_SOLID
$--------+---------+---------+---------+---------+---------+---------+---------+
$    SECID    ELFORM       AET
         1
*MAT_ELASTIC
$--------+---------+---------+---------+---------+---------+---------+---------+
$       MID        RO         E        PR        DA        DB         K
         1    2700.   70.e+09        .3
*ELEMENT_SOLID
$-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
$    EID     PID      N1      N2      N3      N4      N5      N6      N7      N8
       1       1       1       2       3       4       5       6       7       8
*NODE
$-------+---------------+---------------+---------------+-------+-------+-------+
$    NID               X               Y               Z      TC      RC
       1              0.              0.              0.       7       7
       2              1.              0.              0.       5       0
       3              1.              1.              0.       3       0
       4              0.              1.              0.       6       0
       5              0.              0.              1.       4       0
       6              1.              0.              1.       2       0
       7              1.              1.              1.       0       0
       8              0.              1.              1.       1       0
```

Abbildung 2.2: Struktur einer LS-Dyna .key-Datei [LD]

PAM-Crash Dateien

PAM-Crash ist ein Simulationsprogramm das von der französischen ESI-Group in den 80er Jahren entwickelt wurde [Groa]. Es wird hauptsächlich in der Automobilindustrie für Crash-Simulationen und Insassenschutzsysteme verwendet. Dabei haben die Ingenieure die Möglichkeit das Verhalten eines vorgestellten Fahrzeugdesigns in mehreren Crashtests zu simulieren und das resultierende Verletzungspotential der Insassen zu evaluieren.
Als Eingabedatei in PAM-Crash dienen sogenannte Inputdecks. Diese werden als .pc-Datei oder Include-Datei (.inc) abgespeichert. Das Grundkonzept unterscheidet sich dabei kaum von den von LS-Dyna verwendeten Eingabedateien. In PAM-Crash befinden sich die Keywords jedoch am Anfang jeder Zeile und verfolgen danach eine tabellarische Struktur. Einzelne Bauteile werden durch *PARTS* / , Schalenvolumen durch *SHELL* / und einfache Knoten durch *NODE* / definiert. Zusätzlich können noch weitere Anfangs- und Randbedingungen festgelegt werden, auf welche hier aber nicht weiter eingegangen wird [Gro06, Stu07].

```
#
#---5---10----5---20----5---30----5---40----5---50----5---60----5---70----5---80
NODE  /    100001497.13221537198-127.03320325163-109.90321768794
NODE  /    10001       2149.602       717.98833       89.008759
NODE  /    100021501.31042584565-125.42948868015-109.90183572906
NODE  /    100031502.27006012963-128.66816404668-112.30630533558
NODE  /    100041498.34208275679-130.16702927184-112.29983120515
NODE  /    100051495.93647057447-123.51983323836-107.26957220439
NODE  /    100061500.34168050003-121.86245542538-107.27515136938
NODE  /    100071504.06341464279-116.87901109496-104.38201346925
NODE  /    100081504.81297564961-120.79573638207-107.25955848424
NODE  /    100091499.35340512831-117.83729510658-104.33486946453
NODE  /    10010       1580.3826      -98.211922      -57.712928
NODE  /    10011       1575.3045      -97.765075      -57.713946
NODE  /    10012       1575.0449      -95.689566      -52.629052
NODE  /    10013       1580.6215      -95.843615      -52.187169
NODE  /    10014       1585.6657      -98.676951      -57.712928
NODE  /    10015       1586.0688      -96.001548      -51.803533
NODE  /    10016       1591.0862       -99.15408      -57.712928
NODE  /    10017        1591.649      -96.169006      -51.449381
NODE  /    100181448.12593771212  -171.518184738-111.44823647161
NODE  /    10019   1454.91873439-167.44628874206-111.54330706357
NODE  /    10020       2149.602       713.11425       88.214408
NODE  /    100211457.78024467816-170.95346813034-114.48038174161
NODE  /    100221451.57717639162-175.13066586506-114.46783745822
NODE  /    100231441.74573108712-171.13167226077-109.47394629528
```

Abbildung 2.3: Struktur einer PAM Crash Include-Datei

In Abbildung 2.3 ist ein Beispiel einer solchen PAM-Crash Include-Datei mit den Definitionen einzelner Knotenpunkte zu sehen. Aus diesen FEM-Dateien können die 3D-Daten der einzelnen Bauteile eines Fahrzeugs extrahiert und analysiert werden (siehe Kapitel 4).

2.3 DATENVORVERARBEITUNG

Um aussagekräftige Ergebnisse aus einer datenbasierten Analyse zu erlangen, ist die Qualität der entsprechenden Daten entscheidend. Gerade im Bereich des maschinellen Lernens und Data-Minings ist der Prozess der Datenbeschaffung und -aufbereitung ein nicht zu vernachlässigender Schritt [ES00, Run10, D.17]. Damit derartige Algorithmen die erwarteten Resultate liefern, müssen die Daten eine exakte Form und Struktur nachweisen.

Dieses Erzeugen von Daten in hinreichender Qualität ist die Aufgabe der Datenvorverarbeitung. Welche Schritte im Detail notwendig sind hängt stark vom gegebenen Kontext ab. Grundlegend kann aber der Prozess in drei einzelne Abschnitte unterteilt werden, welche sich an dem Kontext der Arbeit orientieren:

- **Datenbeschaffung** - Auswahl und Zusammentragen der Daten
- **Datenaufbereitung** - Löschen von inkorrekten, inkonsistenten und redundanten Daten
- **Datenreduktion** - Transformation der Daten in eine verwendbare Struktur

Dieser Abschnitt beschreibt die grundlegenden Methoden zur Datenvorverarbeitung, die in der Arbeit Anwendung finden. Zuerst folgt eine Analyse der 3D-Daten und deren Repräsentationsformen, danach werden die mathematischen Grundlagen zum Sampling und zur Normalisierung beschrieben. In Kapitel 4 erfolgt dann eine genaue Beschreibung des Anwendungsfalles der Arbeit anhand der oben vorgestellten Prozesskette. Dabei werden die einzelnen Abschnitte nochmal mit Hilfe der hier verwendeten Daten genauer erläutert. Dabei ist es möglich, dass es zu Überschneidungen der einzelnen Abschnitte kommen kann.

2.3.1 Repräsentation von 3D-Daten

3D-Modelle können durch unterschiedlichste Geräte aufgenommen beziehungsweise gescannt werden. Durch diese Variation an Aufnahmegeräten entsteht auch eine Vielzahl an verschiedenen Repräsentationen von 3D-Daten. Jede dieser Repräsentationen besitzt andere Eigenschaften und eine eigene Struktur, weshalb sowohl die Vorverarbeitung, als auch die entsprechenden Analysemethoden exakt darauf abgestimmt sein müssen.
Aus diesem Grund werden im Folgenden einige Repräsentationen von 3D-Daten genauer vorgestellt. Dieser Abschnitt orientiert sich dabei an der von Ahmed et al. vorgestellten Struktur [ASS+18]. Er unterteilt 3D-Daten in zwei große Hauptgruppen euklidisch-strukturierte Daten und nichteuklidisch-strukturierte Daten. Euklidisch-strukturierte 3D-Daten sind Daten, die eine globale Parametrisierung und ein allgemeines Koordinatensystem besitzen. Hingegen verfügen nichteuklidisch-strukturierte Daten über keine globale Struktur.

RGB-D Daten

Bei RGB-D Daten handelt es sich um normale 2D-Bilder mit einem zusätzlichen Kanal für die Tiefe. Diese Tiefenkarte bildet den Abstand der Objekte in der Szene zur Kamera ab und liefert dadurch prinzipiell die 3D-Informationen der Modelle. Durch die Popularität der RGB-D Sensoren, wie beispielsweise die Microsoft Kinect, stellt die Beschaffung solcher Daten keine große Herausforderung mehr dar. Da Methoden des maschinellen Lernens besonders im 2D-Bereich stark Fortgeschritten sind, bilden RGB-D Daten eine effektive Darstellungsform für Aufgaben im Bereich der Objekterkennung. In Abbildung 2.4 ist ein Beispiel eines solchen Tiefenbildes dargestellt [TL19].

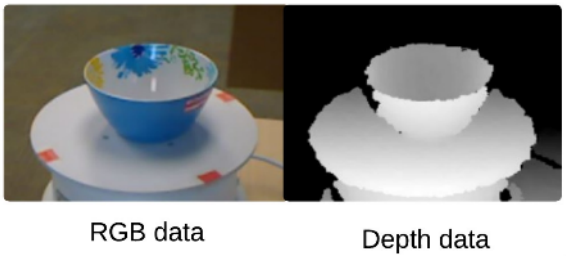


Abbildung 2.4: Beispiel eines Tiefenbildes einer Tasse [ASS+18]

Volumengrafiken

3D-Daten können auch als Volumengrafiken dargestellt werden. Dazu wird ein gleichmäßiges Voxelgitter im dreidimensionalen Raum zur Modellierung benutzt. Voxel bilden also das äquivalent zu Pixeln im 3D. Diese werden benutzt, um die Verteilung des 3D-Objektes im Raum zu beschreiben. Informationen der unterschiedlichen Betrachtungsweisen oder Blickwinkeln können dargestellt werden, indem die jeweiligen Voxel als sichtbar oder verdeckt markiert werden. Durch diese Repräsentation entsteht jedoch ein Overhead an Information, der besonders speicherplatzineffizient ist. Zusätzlich führt es zu einer Verlängerung der Laufzeit, da mehr Daten verarbeitet werden müssen. In Abbildung 2.5 ist der berühmte Hase der Stanford University als Voxelgrafik abgebildet.

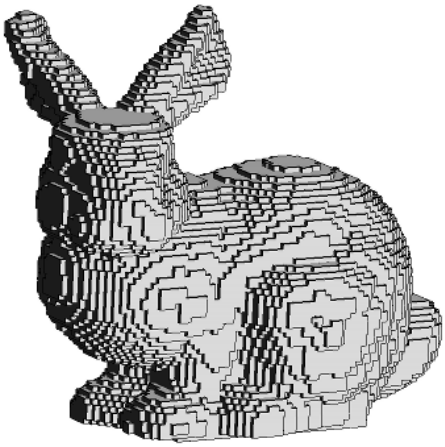


Abbildung 2.5: Beispiel einer Voxelgrafik eines Hasen [NKB]

Eine bessere Darstellungsform für Volumengrafiken bieten Octrees. Octrees haben variierende Voxelgrößen und eine hierarchische Struktur. Hierfür wird das 3D-Modell vom Wurzelvoxel aus rekursiv zerlegt und in kleinere Voxel unterteilt. Die Voxel befinden sich entweder im Inneren oder Äußeren des Objektes. Im Allgemeinen ist diese Darstellungsform für Aufgaben im Bereich Deep Learning sehr ineffizient.

Multi-View-Daten

Abbildung 2.6 symbolisiert den Aufnahmeprozess von Multi-View-Daten. Multi-View-Daten sind eine Sammlung aus mehreren 2D-Bildern, die von dem selben 3D-Objekt aus verschiedenen Blickwinkeln aufgenommen wurden. Diese Darstellung ist durch die Redundanz der Daten besonderes robust gegenüber Verdeckung, Rauschen und Unvollständigkeit. Ein besonderes Problem stellt die Frage nach der Anzahl der Blickwinkel dar. Wählt man eine zu geringe Anzahl kann es passieren, dass nicht alle Eigenschaften des 3D-Objektes aufgenommen werden. Ist die Anzahl der Bilder zu groß entsteht ein Overhead, der eine Ineffizienz der Laufzeit und des Speicherbedarfs mit sich zieht. Dennoch ist die Performanz der Multi-View-Daten besser als die von Volumengrafiken [ASS+18]

Abbildung 2.6: Visualisierung der Aufnahme von Multi-View-Daten [ASS+18]

3D-Punktwolken

Eine 3D-Punktwolken ist eine Ansammlung von Punkten in einem dreidimensionalen Vektorraum, die eine unstrukturierte räumliche Form besitzt. Die in der Punktwolke enthaltenen Punkte werden durch ihre jeweiligen Raumkoordinaten beschrieben und approximieren so das 3D-Modell. Diese einfache, diskrete Form vermeidet das Auftreten kombinatorischer Unregelmäßigkeiten und eignet sich dadurch gut zum Anlernen von KI-Algorithmen. Dennoch sollten diese Algorithmen invariant gegenüber Verschiebung und Rotation sein, da das Auftreten aus verschiedenen Perspektiven nicht in der Darstellungsform berücksichtigt wird.

Die Erzeugung von Punktwolken ist grundsätzlich simpel. In den meisten Fällen werden Scanning-Verfahren eingesetzt, dabei werden Laserscanner verwendet, um das 3D-Objekt oder die Umgebung abzutasten. Auch leichter verfügbare Technik, wie Kinect besitzen die Möglichkeit Punktwolken aufzunehmen. Zusätzlich können zeitinvariante Modelle erzeugt werden, indem der Raum mehrmals an unterschiedlichen Zeitpunkten erfasst wird. Unabhängig von der einfachen Beschaffung der Daten existiert das Problem der fehlenden Information zur Verbindung der Punkte, wodurch eine mehrdeutige Interpretation der Oberfläche entstehen kann.

In Abbildung 2.7 ist ein Beispiel einer Punktwolke eines Pferds zu sehen. Die in dieser Arbeit verwendeten Punktwolken sind Repräsentationen von Autobauteilen und werden genauer in Kapitel 4 behandelt.

Abbildung 2.7: Beispiel einer Punktwolke [ASS+18]

Mesh und Graphen

3D-Meshes stellen die häufigste Form von 3D-Daten dar und unterscheiden sich nur leicht von den Punktwolken. Ein Mesh besteht aus einer Menge an Polygonen, sogenannte *Facetten* und einer Menge an Punkten im dreidimensionalen Raum (*Knoten*). Facetten beschreiben, welche Knoten miteinander verbunden werden. Die Knoten selbst beinhalten eine Liste mit ihren verbundenen Nachbarn. Diese Struktur bietet eine exakte geometrische Darstellung der 3D-Objekte, ist jedoch durch ihre Unregelmäßigkeit eine Herausforderung für die Methoden des maschinellen Lernens. Abbildung 2.8 zeigt einen solchen Mesh.
Meshes können auch als Graph dargestellt werden, indem die Verbindungen zwischen den Knoten als Kanten dargestellt werden. Dadurch entsteht die Möglichkeit verschiedene Graphalgorithmen darauf anzuwenden, wie beispielsweise ein GCNN (Graph Convolutional Neural Network).

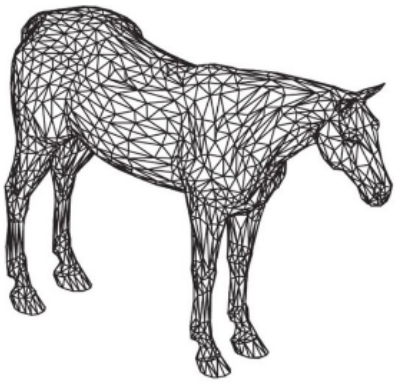


Abbildung 2.8: Beispiel eines Meshes [ASS+18]

2.3.2 Sampling

Extrahierte 3D-Modell Daten liefern nicht immer eine homogene Struktur, wodurch sie nicht effektiv von Methoden des maschinellen Lernens verarbeitet werden können. Beispielsweise ist es sehr wahrscheinlich, dass Modelle mit einer unterschiedlichen Punkt- oder Kantenanzahl vorkommen. Aus diesem Grund ist es notwendig sie in eine einheitliche Form zu bringen. Dieser Abschnitt befasst sich mit den mathematischen Grundlagen des in der Arbeit verwendeten Samplings.

Baryzenztrische Koordinaten

Die baryzentrischen Koordinaten sind in die affine analytische Geometrie einzuordnen. Sie dienen dazu die Lage eines bestimmten Punktes in Bezug auf ein Simplex zu beschreiben. Ein Simplex ist ein n-dimensionales Polytop beispielsweise eine Linie, ein Dreieck oder ein Tetraeder [Sch, Mat]. Die Kernidee der baryzentischen Koordinaten stellt das Teilverhältnis λ eines Punktes P auf einer Geraden $\overrightarrow{AB}$ dar. Somit kann der Punkt P durch eine Linearkombination dieser Verhältniskoeffizienten beschrieben werden.

Gegeben seien zwei Punkte A und B, der Differenzvektor berechnet sich wie folgt:

$$\overrightarrow{AB} = B - A$$

Aus dieser Gleichung kann nun das Grundkonzept der baryzentrischen Koordinaten hergeleitet werden. Es sei ein Punkt P auf einer Geraden $\overrightarrow{AB}$ gegeben mit $\overrightarrow{AP} : \overrightarrow{AB} = \lambda$

Dann gilt auch $P - A = \lambda(B - A) = \lambda B - \lambda A$ und somit durch Umstellung auch

$$P = (1 - \lambda)A + \lambda B \tag{2.3.1}$$

In diesem Fall bezeichnet man die Punkte A und B als Ursprungspunkte des eindimensionalen reellen baryzentrischen Koordinatensystems. Die allgemeine Definition sieht wie folgt aus:

Seien $x_1, ..., x_n$ Eckpunkte eines Simplex im Vektorraum A. Gilt für einen Punkt P aus A folgende Gleichung:

$$(\alpha_1 + ... + \alpha_n) * P = \alpha_1 x_1 + ... + \alpha_n x_n$$

So nennt man die Koeffizienten $(\alpha_1, ...\alpha_n)$ baryzentrische Koordinaten von p zu $x_1, ..., x_n$. Erfüllen die Koordinaten zusätzlich noch die Bedingung $\alpha_1 + ... + \alpha_n = 1$, dann spricht man von normierten baryzentrischen Koordinaten.
Im Rahmen dieser Arbeit werden baryzentrische Koordinaten zum Generieren von Punkten in Dreiecken benötigt. Aus diesem Grund wird nun die allgemeine Form an diesem Kontext angewandt. Gegeben sind drei Punkte A, B, C in einer Ebene. Angenommen für die Skalare $(\alpha_1, \alpha_2, \alpha_2)$ gilt, dass $\alpha_1 + \alpha_2 + \alpha_3 = 1$, dann ist ein Punkt P in der Ebene wie folgt definiert:

$$P = \alpha_1 A + \alpha_2 B + \alpha_3 C \tag{2.3.2}$$

Der Punkt liegt im Dreieck $\triangle ABC$, wenn folgende Gleichung erfüllt ist:

$$0 <= \alpha_1, \alpha_2, \alpha_3 <= 1 \tag{2.3.3}$$

Ist einer der Koeffizienten kleiner als Null oder größer als Eins, dann befindet sich der Punkt außerhalb des Dreiecks $\triangle ABC$. Ist einer der Koeffizienten gleich Null, dann liegt der Punkt auf einer der Geraden des Dreiecks $\triangle ABC$. Abbildung 2.9 zeigt eine Darstellung eines solchen Punktes P in dem Dreieck $\triangle ABC$ mit seinen baryzentrischen Koordinaten $P : \alpha_1 = \alpha_3 = 0,3; \alpha_3 = 0,7$

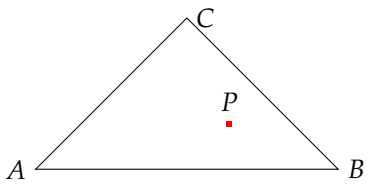


Abbildung 2.9: Darstellung der baryzentrischen Koordinaten anhand eines Dreiecks

Durch diese Repräsentation können nun beliebige Punkte im Dreieck erzeugt und beschrieben werden, solange keiner der Koeffizienten kleiner als Null oder größer als Eins ist. Um ein gleichmäßiges Sampling zu erreichen, müssen die Punkte einheitlich in dem Dreieck verteilt sein. Dafür wird das Latin Hypercube Sampling verwendet.

Latin Hypercube Sampling

Das Latin Hypercube Sampling (LHS) ist eine Methode in der Statistik, um zufällige Stichproben aus einer multivariaten Verteilung zu samplen [MBC79, IDZ80]. Diese Sampling Methode dient oft zum Erzeugen von Daten in verschiedenen Simulationsexperimenten. Im Zusammenhang mit statistischen Samplen ist ein Latin Square (dt. lateinisches Quadrat) ein quadratisches Raster, welches pro Zeile und Spalte exakt nur ein Sample besitzt. Ein Latin Hypercube ist dann die Übertragung dieses Konzeptes in einem n-dimensionalen Raum, wo jedes Sample das einzige in jeder Achsen ausgerichteten Hyperebene ist.

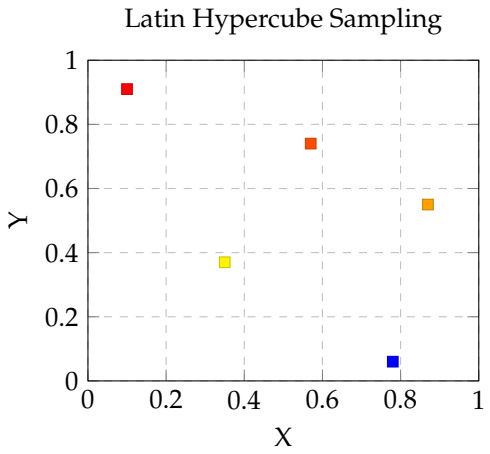


Abbildung 2.10: Darstellung fünf gesampleter Parameter nach Latin Hypercube Sampling

Angenommen es werden N Dimensionen gesamplet. Dann wird der Bereich der Variable in M gleichgroße Abschnitte unterteilt. Danach werden M Samplepunkte so verteilt, dass sie die Voraussetzungen des Latin Hypercubes erfüllen. Das bedeutet, dass die Anzahl an Abschnitten M für jede Dimension identisch sein muss. In Abbildung 2.10 ist ein Beispiel einer solchen Verteilung dargestellt.
Wie in Abbildung 2.10 zu sehen ist, besitzt jede Spalte und jeder Zeile unseres Raumes exakt ein Sample, dadurch kann eine gleichmäßige Struktur gewährleistet werden. Diese Technik wird nun benutzt, um die vorher vorgestellten Koeffizienten (baryzentrische Koordinaten) zu ermitteln und dadurch eine beliebige Anzahl an Punkten in einem Dreieck zu bestimmen.

2.3.3 Normalisierung

Normalisierung oder auch Merkmalsskalierung genannt ist eine Methode der Datenaufbereitung, um den Wertebereich von unabhängigen Variable zu vereinheitlichen [AH01]. Das bedeutet, dass alle Verteilungen einer Variable auf den gleichen Mittelpunkt verschoben werden und deren Standardabweichungen identisch sind.
Da die Wertebereiche von Rohdaten oftmals stark variieren, können einige Methoden des maschinellen Lernens nicht ordnungsgemäß funktionieren. Zum Beispiel hat die euklidische Distanz zweier Vektoren, die oftmals als Maß für Ähnlichkeit genommen wird, einen größeren Einfluss bei größeren Wertebereichen. Deshalb ist es notwendig diese Daten vorher aufzubereiten und zu normalisieren. Zusätzlich kann durch die viel kleineren Werte erreicht werden, dass das Gradientenverfahren schneller konvergiert, wodurch die Performanz der Algorithmen steigt.
Im Rahmen dieser Arbeit werden die Daten anhand der Mittelwertnormalisierung angeglichen, diese ist wie folgt definiert [Shi]:

$$x\prime = \frac{x - \bar{x}}{max(x) - min(x)} \tag{2.3.4}$$

Dieser Ansatz wird nun auf die zugrundeliegenden Daten angewandt. Bei der Normalisierung einer Punktwolke P wird zuerst der Mittelwert μ_P aller Punkte p in der Punktwolke berechnet.

$$\mu_P = \frac{1}{N} \sum_{p \in P} p,$$

wobei N die Anzahl der Punkte ist. Der Punkt μ_P wird oft auch als Zentroid der Punktwolke bezeichnet. Danach erfolgt die Berechnung der Distanzen aller Punkte zum Ursprung, um die maximale Distanz $dist_{max}$ zu ermitteln. Dazu muss der Mittelwert von allen Punkten subtrahiert werden, wodurch der Koordinatenursprung das neue Zentrum definiert. Dieser Sachverhalt wird durch die Gleichungen (2.3.5) beschrieben.

$$p_c = p - \mu_p, \qquad dist_{max} = \max_{p_c \in P_c} (\sqrt{\sum_{i=1} p_{c_i}^2}) \tag{2.3.5}$$

Die daraus resultierende Gesamtform für die Berechnung einer normalisierten Punktwolke sieht dann wie folgt aus:

$$p\prime = \frac{p - \mu_P}{dist_{max}} \tag{2.3.6}$$

2.4 GRUNDLAGEN DES MASCHINELLEN LERNENS

Dieser Abschnitt behandelt die Grundlagen verschiedener Methoden des maschinellen Lernens. Zuerst wird ein kurzer Überblick über den gesamten Bereich vorgestellt, um den Schwerpunkt der Arbeit abzugrenzen. Danach folgt eine genauere Beschreibung der Funktionsweise von neuronalen Netzen, da sie die Grundlage der Untersuchung darstellen. Im Anschluss wird die Architektur von Convolutional Neural Networks (CNN) beschrieben, eine Unterkategorie der Neuronalen Netze, die besonders in der Bilderkennung ihren Einsatz findet. Da 3D-Daten lediglich eine weitere Dimension beinhalten, lässt sich die Funktionsweise der 2D-CNN's auch einfach auf 3D-Daten übertragen.

2.4.1 Überblick

Die praktische Umsetzung des maschinellen Lernens geschieht durch Algorithmen. Diese können in zwei große Gruppen eingeteilt werden, die sich anhand der jeweiligen Eingabedaten und dadurch auch Funktionsweisen unterscheiden [GBC16].

Überwachtes Lernen

Das Überwachte Lernen fasst all die Methoden und Algorithmen zusammen, die mit Hilfe von gelabelten Daten arbeiten. Grob gesagt muss der Algorithmus eine Funktion lernen, welche die jeweiligen Eingabedaten auf bestimmte Ausgabedaten abbildet. Dies geschieht indem der Algorithmus in mehreren Rechenschritten, der so genannten „Trainingsphase", gewisse Eingabedaten erhält und dazu den zugehörigen Funktionswert kennt. Dadurch soll eine Assoziation zwischen Eingabe- und Ausgabedaten hergestellt werden. Mit Hilfe dieser Assoziationen können später, vorher noch unbekannte Eingabedaten einem möglichen Funktionswert zugeordnet werden. Ein Teilgebiet des überwachten Lernens ist die Klassifikation, welche in dieser Arbeit näher betrachtet wird.

Beim Klassifizieren geht es darum, die Eingabedaten in eine der vorher festgelegten Klassen zuzuordnen. Der Algorithmus findet demnach eine Assoziation zwischen der Eingabe und einem Label (Klasse). Im Kontext dieser Arbeit stellen die 3D-Modelle unsere Eingabedaten dar und die jeweiligen Bauteile definieren unsere Klassen. Eine detailliertere Beschreibung erfolgt später im Kapitel 3.

Unüberwachtes Lernen

Beim unüberwachten Lernen verfügt der Nutzer über keine Ausgabedaten. Der Algorithmus erhält lediglich die Eingabedaten und soll daraus ein statistisches Modell erzeugen. Dabei gilt es das Auftreten und die Korrelation der Daten zu untersuchen und zusammenzufassen, wodurch mit Hilfe des erlernten Modells Vorhersagen möglich sind. Ein wichtiger Vertreter des unüberwachten Lernens ist die Clusteranalyse. Bei dieser wird versucht möglichst ähnliche Daten in eine bestimmte Anzahl an Gruppen (Cluster) zu unterteilen. Diese Methode bietet auch eine

Möglichkeit zur Klassifikation und eignet sich gut, wenn die Anzahl und Art der Klassen nicht genau vorliegt. Da dies im Rahmen dieser Arbeit nicht der Fall ist, werden die Methoden des unüberwachten Lernens hier nicht weiter betrachtet.

2.4.2 Künstliche Neuronale Netze

Ein bekannter Vertreter des überwachten Lernens und einen wichtigen Bereich der künstlichen Intelligenz stellen künstliche neuronale Netze (KNN) dar. Neuronale Netze sind angelehnt an das menschliche Gehirn und stellen somit das Pendant zu den vernetzen Neuronen im Nervensystem dar. Im Folgenden wird genauer auf die grundlegende Funktionalität und den Aufbau von neuronalen Netzen eingegangen. Dieser Abschnitt orientiert sich am Kurs *Convolutional Neural Networks for Visual Recognition* der Standford University von Andrej Karpathy [Kar19c].

Perzeptron

Neuronale Netze modellieren grob das biologische Nervensystem. Das Fundament stellen demnach die sogenannten Neuronen dar. Ein Neuron ist die kleinste Recheneinheit und besteht aus drei Hauptbestandteilen. Der Eingabesignale mit jeweiligen Gewichten, einer Aktivierungsfunktion mit zusätzlichen Bias, der einen Schwellwert darstellt und dem Ausgangssignal. Mit Hilfe eines Neurons, können beispielsweise einfache logische Operationen dargestellt werden. In Ab-

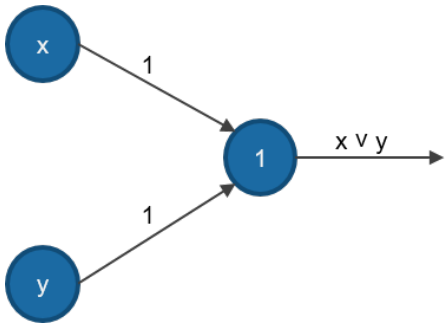


Abbildung 2.11: Perzeptron eines logischen ODERs

bildung 2.11 ist ein simpler Aufbau eines solchen Neurons als logisches ODER abgebildet. Die Eingaben x und y werden mit ihren jeweiligen Gewichten multipliziert und anschließend aufsummiert. Der daraus resultierende Wert muss größer als ein bestimmter Schwellwert (Bias) sein, um den Ausgang des Neurons zu „aktivieren“. Die daraus resultierende Funktion wird Aktivierungsfunktion genannt und ist in diesem Beispiel eine einfache Schwellenwertfunktion, die eine boolsche Variable widerspiegelt. Im folgenden Abschnitt wird ein genauerer Blick auf die geläufigsten Aktivierungsfunktionen geworfen.

Aktivierungsfunktion

Verschiedenste Funtkionstypen können als Aktivierungsfunktion genutzt werden. Jede Aktivierungsfunktion nimmt einen einzelnen Eingabewert entgegen, in diesem Falle die Summe der ge-

wichteten Eingänge und führt eine festgelegte mathematische Operation darauf aus. Die Funktionen sind dabei nicht abhängig von der Topologie des Netzes und im Allgemeinen monoton steigend.

Sigmoid Die Sigmoid-Funktion ist eine nicht-lineare Funktion und eine der bekanntesten Aktivierungsfunktionen. Der Verlauf der Funktion ist s-förmig und bildet die Eingabe auf einen Wertebereich zwischen 0 und 1 ab (siehe Abbildung 2.12). Genauer gesagt, große negative Werte werden 0 und große positive Werte werden 1. Durch ihre einfache Differenzierbarkeit und der Interpretation als „Aktivierungsrate“: von das Neuron feuert nicht (0) zu das Neuron feuert komplett (1), wurde Sigmoid eine oft benutze Aktivierungsfunktion in der Vergangenheit. Jedoch verliert sie aktuell an Beliebtheit, da sie einige Nachteile mit sich zieht.

$$\text{sig(x)} = \frac{1}{1 + e^{-x}} \tag{2.4.1}$$

Eine negative Eigenschaft der Sigmoid-Funktion ist die schnelle Sättigung der Aktivierung. In den Bereichen von 0 und 1 ist der lokale Gradient der Funktion nahezu 0. Dieser wird während der Backpropagation mit den vorherigen Gradienten multipliziert, wodurch der Gradient Null wird und somit kein Signal durch das Neuron fließt. Deshalb ist es wichtig die Initial-Gewichte der Sigmoid-Funktion nicht zu groß zu wählen, damit nicht von Beginn an jegliche Neuronen gesättigt sind, denn dadurch würde das Netz kaum lernen. Zusätzlich ist die Ausgabe der Funktion nicht null-zentriert, wodurch die Gradienten entweder nur positive oder negativ werden, was ein unerwünschtes Verhalten beim Gradientenabstieg nach sich zieht.

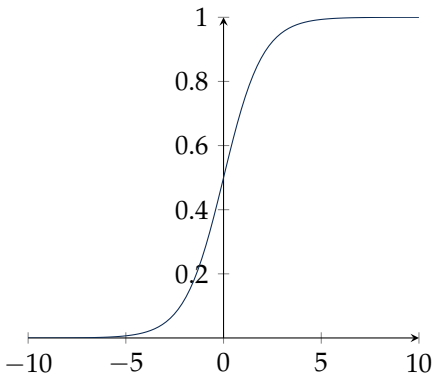


Abbildung 2.12: Graphische Darstellung der Sigmoid-Funktion

ReLU ReLU steht für Rectified Linear Unit. Die Aktivierungsfunktion hat in den letzten Jahren an Popularität gewonnen und wird vor allem in Bereichen des Deep Learnings verwendet. Die Funktion wird wie folgt berechnet:

$$f(x) = \max(0, x) \text{ oder Approximation: } f(x) = \log(1 + e^x) \tag{2.4.2}$$

Sie ist Null wenn die Eingabe $x < 0$ ist und verläuft danach linear mit einem Anstieg von 1. In anderen Worten, die Funktion besitzt den Schwellenwert Null. Es wurde gezeigt, dass durch ihre Linearität der stochastische Gradientenabstieg schneller konvergiert als im Vergleich zur Sigmoid-Funktion [Kar19c]. Zusätzlich ist die Implementierung der ReLU-Funktion effizienter, da weniger rechenintensive Operationen verwendet werden müssen. Jedoch kann es passieren, dass bei großen Gradienten sich die Gewichte so anpassen, dass die Funktion für keine Eingabe mehr aktiviert, dann spricht man von einem „toten“ Neuron. Aus diesem Grund gibt es eine leicht veränderte Version: Die Leaky ReLU. Anstatt dass der Funktionswert konstant auf Null gesetzt wird, bekommt die Funktion bei einer negativen Eingabe einen kleinen Anstieg. Dieser wird durch eine Konstante a ermöglicht (bsp. $a = 0.01$) Dadurch wird verhindert, dass negative Werte das Neuron nicht aktivieren und somit einzelne Neuronen „sterben“.

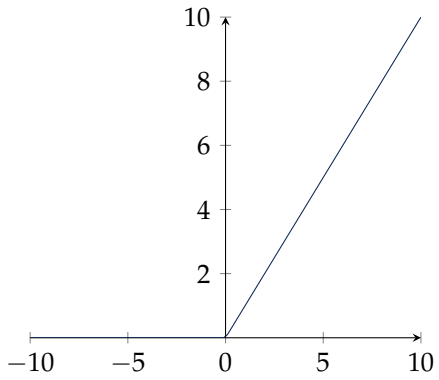


Abbildung 2.13: Graphische Darstellung der Rectified Linear Unit

Backpropagation

Der Grundaufbau eines neuronalen Netzes kann als azyklischen Graphen dargestellt werden. Ein Netz besteht aus mehreren Schichten (englisch Layer), welche jeweils eine beliebige Anzahl an Neuronen enthalten. Dabei besitzt das Netz immer einen Input-Layer und einen Output-Layer und eine bestimmte Anzahl an sogenannten Hidden-Layern. Sind die einzelnen Neuronen einer Schicht komplett mit allen Neuronen der nächsten Schicht verbunden, so spricht man von einem Fully-Connected-Netz. Einfach gesagt, die Ausgabe eines Neurons wird zur Eingabe jedes einzelnen Neurons in der nächsten Schicht. In Abbildung 2.14 sieht man den Aufbau eines solchen einfachen neuronalen Netzes. Der Output-Layer besitzt meist keine Aktivierungsfunktion, da er beispielsweise oft als Ausgabe für eine Klassenbewertung dient.

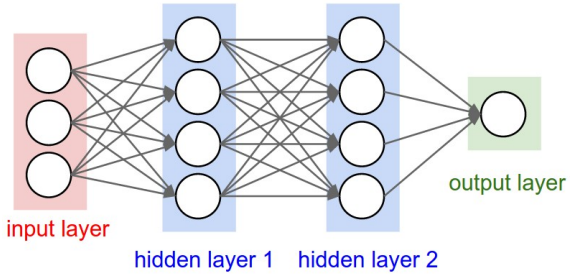


Abbildung 2.14: Beispiel eines einfachen neuronalen Netzes [Kar19c]

Einer der Hauptgründe für diesen schichtweisen Aufbau ist die simple und effiziente Berechnung eines Eingabevektors, wenn er vorwärts durch das Netz propagiert wird. Die Gewichte einer Schicht können einfach in einer Matrix dargestellt werden. Mittels Matrixmultiplikation kann dann die Eingabe der Aktivierungsfunktion berechnet werden und somit die Aktivierung aller Neuronen in einer Schicht. Die Aufgabe des Netzes ist es nun die Gewichtsmatrizen und Bias-Vektoren so anzulernen, dass ein gewünschter Ausgangswert erreicht wird. Dies geschieht mit Hilfe der Backpropagation [Kar19a]. Dabei wird ein Eingabevektor vorwärts durch das Netz propagiert und die resultierende Ausgabe mit der gewünschten Ausgabe verglichen. Danach wird das Fehlermaß berechnet, ein Maß dafür, wie weit die aktuelle Ausgabe von der erwarteten entfernt liegt. Ein Beispiel für eine Fehlerfunktion ist der quadratische Fehler:

$$E = \frac{1}{n}\sum_{i=1}^{n}(t_i - o_i)^2, \tag{2.4.3}$$

Dabei ist t_i die erwartete Ausgabe und o_i der tatsächlich errechnete Wert. Ziel ist es, den Fehler zu minimieren, um die Ausgabe näher an die erwartete Ausgabe heranzuführen. Mittels des mathematischen Optimierungsverfahrens Gradientenabstieg ist dies möglich. Dabei wird nun der Fehler durch das Netz zurück propagiert, indem die partielle Ableitung in Abhängigkeit der Gewichte berechnet wird. Das geschieht durch eine simple Anwendung der Kettenregel.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}, \tag{2.4.4}$$

Dadurch werden die Gewichte der einzelnen Neuronen abhängig von ihrem Einfluss auf den Fehler geändert und erlernt. Die Frequenz der Aktualisierung der Gewichte hängt vom jeweiligen gewählten Verfahren des Gradientenabstiegs ab. Es wird grob zwischen drei verschiedenen Methodiken unterschieden, dem Batch Gradient Descent, dem Stochastic Gradient Descent und dem Mini-Batch Gradient Descent [Rud16].
Beim Batch Gradient Descent erfolgt das Update der Gewichte nach einem kompletten Durchlauf der Trainingssamples. Diese Methode ist sehr speicheraufwendig, konvergiert aber sicher in ein globales Minimum bei einer konvexen Fehlerfunktion. Beim Stochastic Gradient Descent werden die Gewichte nach jedem Sample aktualisiert. Dadurch wird die Berechnung in der Regel schneller, aber durch die Fluktuation kann es passieren, dass nur lokale Minima gefunden werden. Als Letztes gibt es noch den Mini-Batch Gradient Descent, dieser ist eine Mischung aus beiden Varianten. Die Gewichte werden nach einem Batch aus n Samples aktualisiert. Dadurch wird die Varianz der Updates verringert und eine stabilere Konvergenz erreicht. Die Performanz ist je nach Wahl der Batchgröße immer noch effizient. Deep-Learning-Methoden implementieren in den meisten Fällen den Mini-Batch Ansatz, da diese Methodik die effizienteste darstellt.

2.4.3 Convolutional Neural Networks

Convolutional Neural Networks sind eine spezielle Form von neuronalen Netzen. Sie finden vor allem Anwendung im Bereich der Bildverarbeitung und visuellen Erkennung, da die Eingabe meist Bilddaten sind. Dennoch eignen sie sich auch zum Erkennen von 3D-Daten. In diesem Abschnitt werden die Grundkomponenten einer CNN-Architektur anhand eines Beispiels mit 2D Bildern beschrieben. Die Prinzipien eines 2D-CNN lassen sich leicht in den dreidimensionalen Raum übertragen, bieten jedoch eine simplere Veranschaulichung. Im Folgenden wird auf die drei Hauptschichten eines Convolutional Neutal Network eingegangen: Convolutional Layer, Pooling Layer, Fully-Connected Layer.

Convolutional Layer

Der Convolutional Layer ist der Kernbestandteil eines CNN und vollzieht die meiste Rechenarbeit. Er besteht aus einer Menge an erlernbaren Filtern. Ein Filter ist eine kleine Matrix mit der ein Bild gefaltet werden kann, um besondere Merkmale, wie beispielsweise Kanten oder Muster zu erkennen. Der Filter hat eine festgelegte Größe, ist also im Vergleich zum Eingabebild deutlich kleiner, reicht jedoch immer über die volle Tiefe des Bildes (RGB-Farbbild). Wenn nun der Filter über das Bild gleitet, wird eine 2D-Aktivierungsmap berechnet, die beschreibt wie der Filter auf den jeweiligen Stellen im Bild anspricht. Ein Convolutional Layer besteht aus einer beliebigen Anzahl an Filtern, die jeweils bestimmte Merkmale des Bildes erkennen. Die Größe des Outputs eines Convolutional Layers hängt von drei Hyperparametern ab - der **Tiefe**, **Schrittgröße** und **Zero-Padding**. Mit folgender Formel lässt sich der die Größe der Ausgabe bestimmen:

$$\text{Output} = \frac{(W - F + 2P)}{2} + 1, \tag{2.4.5}$$

Dabei ist W die Größe des Eingabebildes, F die Größe des Filters, P die Anzahl der Pixelränder des verwendeten Paddings und S der benutzte Schrittgröße. Im Gegensatz zu einem normalen Neuronalen Netz sind die einzelnen Neuronen in einem Convolutional Layer nur lokal mit der Eingabe verbunden.
Für beispielsweise eine Eingabebild der Größe [32x32x3] und einer Filtergröße von [5x5x3] ist die Anzahl der Gewichte eines Neurons 5*5*3 = 75 Gewichte (+1 Bias Parameter). Dabei sind die Pixel, die als Eingabe für das Neuron dienen, exakt an der gleichen Stelle und umfassen lediglich die Größe der Filtermatrix. Zusätzlich teilen sich die Neuronen eines Convolutional Layer die Gewichte, da sonst die Parameteranzahl bis in den Millionenbereich steigen kann. Somit spiegelt eine Schicht eine bestimmte Erkennung eines Features, wie beispielsweise die Erkennung von Kanten im Eingabebild wieder. Außerdem wird durch das Teilen der Gewichte pro Schicht eine Translationsinvarianz erreicht. Ein Feature ist somit nicht abhängig von seiner jeweiligen Position.

In Abbildung 2.15 ist eine vereinfachte Architektur eines Covolutional Layers zu sehen. Je tiefer der Convolutional Layer liegt, umso größer ist das Feld der betrachteten Pixel. Also die Pixel, die in die Berechnung der Ausgabe mit einfließen. Um nun überflüssige Informationen zu entfernen werden die sogenannten Pooling Layer benötigt.

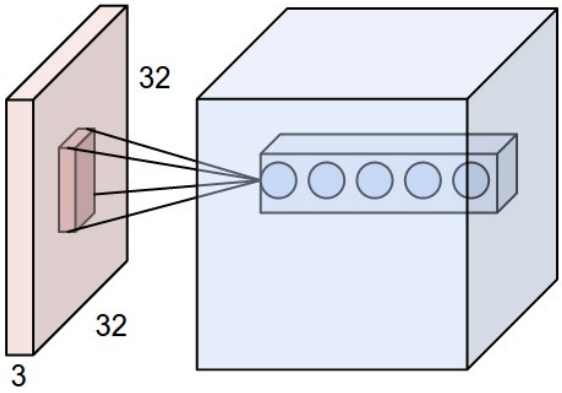


Abbildung 2.15: Lokale Verbindungen eines Neurons im Convolutional Layer [Kar19b]

Pooling Layer

Bei der Objekterkennung ist die exakte Position eines Features von nicht so großer Bedeutung, wie die ungefähre Position. Aus diesem Grund wird das sogenannte Pooling angewandt, wobei hier nur auf einem stark verbreiteten Vertreter eingegangen wird: Dem Max-Pooling. In der Praxis ist es gängig nach mehreren Convolutional Layern regelmäßig Pooling Layer einzufügen, um die Größe der Ausgabe zu verringern und unwichtige Information herauszufiltern. Zusätzlich wird dadurch die Performanz der Netze verbessert, da weniger Berechnungen durchgeführt werden müssen. Pooling wird mittels einer Max-Operation realisiert und unabhängig auf jeder Tiefenebene der Eingabe angewandt. Meistens wird ein Layer mit einem Filter der Größe 2x2 und einer Schrittweite von 2 verwendet. Somit wird das Maximum aus einem kleinen Bereich von vier Pixeln bestimmt und 75 % der Aktivierungen verworfen. In Abbildung 2.16 ist ein Beispiel eines solchen Filters dargestellt.

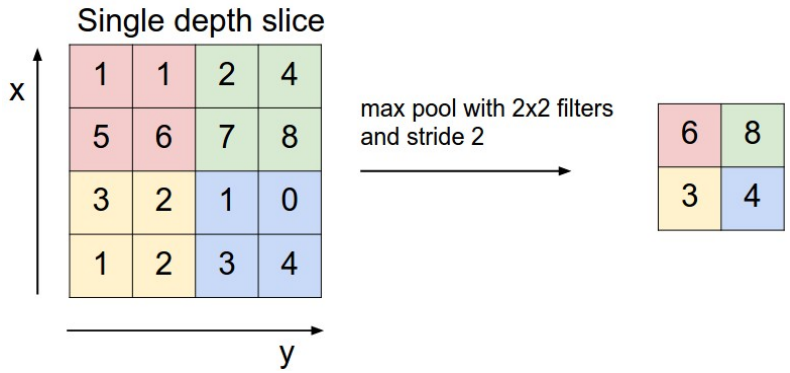


Abbildung 2.16: Beispiel der Max-Pool Operation [Kar19b]

Die Tiefendimension bleibt hiervon unberührt. Durch die Reduzierung der räumlichen Größe reduziert sich ebenfalls die Parameteranzahl und der dafür benötigte Rechenaufwand. Daraus resultiert die Möglichkeit komplexere Netze zu gestalten. Zusätzlich gewährt Pooling mehr Kontrolle gegenüber einer Überanpassung des Netzes, da durch die alleinige Nutzung prägnanter Features das neuronale Netz besser generalisieren kann.

Fully-Connected Layer

In der Praxis werden als letzte Schicht häufig Fully-Connected Layer benutzt. Die Neuronen eines Layers sind, wie im vorherigen Abschnitt beschrieben, mit jedem Neuronen des vorherigen Layers verbunden. Vor allem findet diese Architektur bei der Klassifikation verwendung. Dabei ist die Anzahl der benutzen Hidden-Layern beliebig. Jedoch wird die Anzahl der Neuronen im letzten Layer meist durch die Anzahl der zu unterscheidbaren Klassen bestimmt. Die Ausgabe der Neuronen der letzten Schicht geben die sogenannte Klassenbewertung an (engl. class score). Diese kann bei Bedarf mit Hilfe der Softmax-Funktion in eine Wahrscheinlichkeitsverteilung überführt werden (sie Gleichung 2.4.6).

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{2.4.6}$$

Die Softmax-Funktion normalisiert die Ausgabewerte auf einen Bereich von [-1, 1]. Diese Werte geben somit an, wie Wahrscheinlich die Eingabe zu einer bestimmten Klasse zugeordnet werden kann.

3 VERWANDTE ARBEITEN

In diesem Abschnitt werden die verwandten Arbeiten zum Thema Erkennung von FEM-Geometrien kurz beschrieben und von der vorliegenden Arbeit abgegrenzt. Dabei wird zuerst der Fokus auf die Arbeit von Akhil Pillai gelegt [Pil19], da diese die Grundlage einiger verwendeter Ansätze darstellt. Danach folgt die Masterarbeit von Max Knorr [Kno19], deren Schwerpunkt auf der Vorhersage von Schweißpunkten beruht. Im Anschluss erfolgt eine kurze Übersicht über verschiedene Methoden im Stand der Technik zur 3D-Daten Erkennung.

3.1 PARAMETERVORHERSAGE MIT POINTNET

Pillai beschäftigte sich in seiner Arbeit [Pil19] mit dem Vorhersagen von Schweißpunktparametern in Autokarosserien. Seine Arbeit unterteilt sich in zwei große Themenbereiche. Die Klassifikation von Geometriedaten im Allgemeinen, in diesem Falle FEM-Geometrien und der Schätzung von Schweißpunktparametern. Im ersten Abschnitt stellt er die Deep-Learning-Architektur PointNet [QSMG16] vor, auf die in Kapitel 5 genauer eingegangen wird. Pillai beschreibt, wie aus den vorliegenden Daten 3D-Meshes einzelner Autobauteile extrahiert und diese dann anschließend mittels einer Sampling Methode in 3D-Punktwolken konvertiert werden. Die Punktwolken wurden nicht normalisiert. Dieses Konzept stellt eine Grundlage für diese Arbeit dar.
Die somit entstandenen Punktwolken wurden als Trainingsdatensatz für PointNet verwendet. Anschließend zeigte Pillai mit einigen Testversuchen, wie die Klassifikation mit PointNet abschneidet. Hierfür wurden zehn Bauteile des Toyota Yaris extrahiert und antrainiert. Danach wurde unter anderem gezeigt, wie viel Einfluss die Größe der Punktwolke und die Anzahl der Trainingssamples auf die Genauigkeit von PointNet haben. Im Anschluss wurde gezeigt, dass PointNet den Toyota-Yaris-Datensatz, bestehend aus 250 Bauteilen, zu 95 % richtig klassifizieren kann. Dazu muss erwähnt werden, dass der Testdatensatz auf den selben Daten gesamplet wurde wie die Trainingsdaten. An dieser Stelle wäre eine Untersuchung mit mehreren Modellen wichtig um Nachzuweisen, dass PointNet auch komplett unbekannte Bauteile klassifizieren kann. Diese Untersuchung stellt einen Schwerpunkt der vorliegenden Arbeit dar.
Im zweiten Abschnitt beschreibt Pillai die Extraktion von Schweißpunkt-Informationen aus den PAM-Crash Daten. Da die Schweißpunkte zwischen zwei Bauteilen liegen, wurde versucht die

Bauteile zu erkennen und anhand der Klassifikation, sollte durch ein weiteres neuronales Netz auf die jeweiligen Schweißpunktparameter geschlossen werden. Im Weiteren wurden verschiedene Ansätze zur Vorhersage von Schweißpunktparametern inklusive Testergebnisse vorgestellt. Da der Schwerpunkt dieser Arbeit allein auf der Erkennung und Klassifikation von FEM-Geometrien liegt, wird hier nicht tiefer drauf eingegangen.

3.2 PARAMETERVORHERSAGE MIT STEIFIGKEITSMATRIZEN

Knorr beschreibt in seiner Arbeit [Kno19] eine automatisierte Methode für das Setzen von Schweißpunkten mittels Methoden des maschinellen Lernens. Dabei wird auf einigen grundlegenden Konzepten von Pillai [Pil19] aufgebaut. Knorr extrahiert jedoch nicht gesamte Bauteile, sondern lediglich die sogenannten Flanschbereiche, die Überlappungsfläche zweier verschweißter Bauteile und entwickelt damit einen neuen Ansatz zur Extraktion. Den Schwerpunkt der Arbeit bilden zwei Strategien die sich mit der Vorhersage der Schweißpunktparameter beschäftigen. Nach Strategie 1 werden die extrahierten Bauteilbereiche erkannt und klassifiziert. Anhand dieser klassifizierten Bauteile und mit Hilfe einer Wissensdatenbank werden die entsprechenden Schweißpunkte auf das jeweilige Bauteil gemappt.

Die zweite Strategie konvertiert die Flanschbereiche in eine Steifigkeitsmatrix. Das sind Matrizen, die zur Berechnung von FEM-Solvern benötigt werden. Sie stellen demnach eine weitere Darstellungsform von 3D-Daten dar. Diese neue Darstellung dient nach Knorr nun als Grundlage für ein neuronales Netz, welches die Eingabematrizen auf eine Klasse abbildet. Anschließend wurde die neue entwickelte Methode auf reale Flanschbereiche getestet. Aufgrund der resultierenden Genauigkeitswerte wird dieser Ansatz in der vorliegenden Arbeit nicht genauer untersucht.

3.3 STAND DER TECHNIK ZUR 3D-GEOMETRIE ERKENNUNG

Das Erkennen und Klassifizieren von 3D-Geometrien ist ein aktuelles, junges Forschungsgebiet. Aus diesem Grund gibt es einige Ansätze, die sich besonders in der Art der Repräsentation der 3D-Daten unterscheiden [SGWM18]. Seit ungefähr fünf Jahren versuchen Forscher mit ihren neuen Architekturen und Ansätzen immer besser Ergebnisse in Sachen Genauigkeit zu erzielen.

Ein bekannter Datensatz für Benchmarks stellt dabei der *ModelNet* Datensatz dar [Pri]. ModelNet ist eine Sammlung aus 3D-Objekten zusammengefasst in bekannte Klassen, wie beispielsweise Tasse, Stuhl und Flugzeug. Der Datensatz existiert in zwei Versionen, einmal mit 10 unterschiedlichen Klassen und einmal mit 40. Viele der vorgestellten Stand-der-Technik-Methoden beschäftigen sich mit Multi-View Daten [SMKL15, HLL$^+$19, Kan16]. Kanezake et al. stellen in ihrer Arbeit [Kan16] RotationNet vor, eine CNN-basierte Architektur, die Multi-View Daten kategorisiert. Diese erreicht eine der besten Genauigkeitswerte von 97,37 % auf ModelNet40. Ähnlich wie die Multi-View Ansätze ist auch die Ensemble Methode in [SPT18] von Sfikas et al.. Diese nimmt von jeder Achse ein Panoramabild des Objektes auf und reicht es an ein zuständiges CNN weiter. Anhand der so extrahierten Features wird dann klassifiziert. Dieser Ansatz erreicht mit ModelNet auch gute Genauigkeitswerte. Auch die Darstellungsform mit Voxels ist vertreten mit dem bekannten VoxNet von Maturana und Scherer [MS15]. Neuere Ansätze werden vorgestellt

von Zhou und Tuzel in [ZT17]. VoxelNet besteht aus einer CNN Architektur und unterteilt eine Sensor-Punktwolke in ein Voxelgitter. Das daraus resultierende Netz kann Objekte in einer Szene erkennen.

Einer der bekanntesten Vertreter der sich mit der Klassifikation von Punktwolken auseinandersetzt ist PoinNet von Qi et al. [QSMG16]. Dieser Ansatz wurde von Pillai [Pil19] verfolgt und findet in dieser Arbeit wieder Anwendung. Zusätzlich stellten Ben-Shabat et al. in ihrer Arbeit [BLF17] eine neue Darstellungsform von Punktwolken vor. Laut Ben-Shabat konnten mittels dieser die Ergebnisse von PointNet geschlagen werden. Demzufolge ist diese Methode von besonderem Interesse für die vorliegende Arbeit (siehe Kapitel 5). Im Rahmen dieser Arbeit werden Punktwolken als Repräsentation der 3D-Objekte verwendet, da die zugrundeliegende 3D-Bauteile aus Meshes bestehen und sich dadurch einfach in Punktwolken übertragen lassen. Zusätzlich bietet diese Darstellung im Vergleich zu den anderen Vertretern eine bessere Performanz. Aus diesem Grund werden die Ansätze mit Multi-View Daten und Voxels nicht genauer betrachtet.

4 AUFBEREITUNG VON 3D FEM-DATEN

Dieses Kapitel beschäftigt sich damit, wie FEM-Daten aufbereitet werden müssen, um sie für maschinelle Lernalgorithmen verwendbar zu machen. Dazu muss erwähnt werden, dass die Struktur der Daten sehr vielseitig ausfallen kann, wodurch es keine universelle und automatische Methode zur Vorverarbeitung gibt.

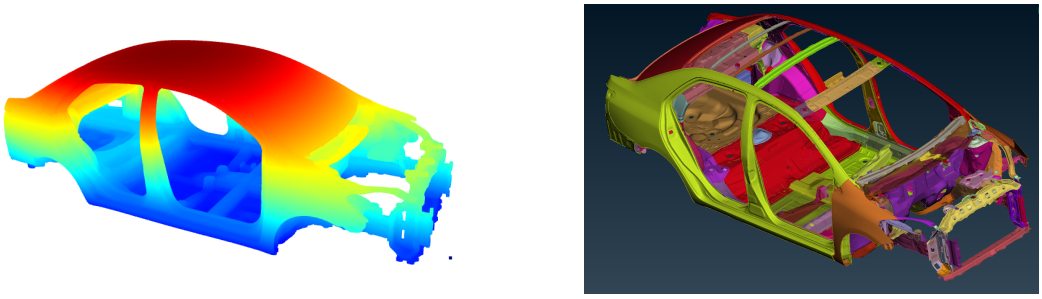
Wie in Kapitel 1 beschrieben ist das Ziel dieser Arbeit, die Untersuchung verschiedener maschineller Lernmethoden zur Klassifikation von FEM-Daten. Dafür müssen die vorhandenen FEM-Daten vorerst analysiert und die für das Training eines neuronalen Netzes notwendigen Informationen extrahiert werden. Dabei stellt im Rahmen dieser Arbeit eine FEM-Datei ein Modell eines bestimmten Fahrzeugs dar. Beispielsweise besteht das FEM-Modell des Toyota Yaris aus über 300 verschiedenen Bauteilen, die als geometrische 3D-Koordinaten hinterlegt sind. Die jeweiligen Bauteile stellen die zu erkennenden Einheiten dar.
Wie Pillai in seiner Arbeit [Pil19] erwähnte, kann die Erkennung für ein trainiertes Automodell gute Ergebnisse liefern. Das bedeutet, um ein neues Bauteil klassifizieren zu können, muss das neuronale Netz mit den entsprechenden Modellen neu trainiert werden. Aus Kapitel 1 geht jedoch hervor, dass die Klassifikation der Bauteile eines Fahrzeugs ohne Vorkenntnis des jeweiligen Modells erfolgreich sein soll. Aus diesem Grund ist der Schwerpunkt dieser Arbeit, die Untersuchung der Generalisierung der maschinellen Lernalgorithmen. Mit anderen Worten: Wie gut können die gewählten Methoden die vorhandenen Datensätze abstrahieren? Das heißt, auf einem bestimmten Datensatz trainiert und mit einem komplett neuen Datensatz getestet werden.
Die verwendeten maschinellen Lernalgorithmen werden später im Kapitel 5 genauer beschrieben und analysiert. Vorerst geht es um die Aufbereitung der zugrundeliegenden FEM-Daten.

4.1 DATENBESCHAFFUNG

Um die Generalisierung nachzuweisen werden mehr FEM-Modelle von verschiedenen Fahrzeugen benötigt. Im Rahmen dieser Arbeit werden, wie in Kapitel 2 beschrieben, zwei Datentypen von FEM-Daten verwendet: LS-Dyna und PAM-Crash Dateien. Die verwendeten PAM-Crash Daten wurden von Audi der SCALE GmbH zur Verfügung gestellt. Das Toyota Yaris Modell wird von LS-Dyna zur freien Verwendung angeboten. Insgesamt wurden sechs verschiedene Automodelle aus dem LoCo (*Load Case Composer*), eine Software zur Verwaltung der FEM-Daten von der SCALE GmbH, extrahiert und bereitgestellt. Darunter befanden sich fünf Modelle von Audi als PAM-Crash Datei sowie der Toyota Yaris als LS-Dyna Datei. Die Bezeichnungen der Audi-Modelle werden aus Diskretionsgründen umbenannt. Die in der vorliegenden Arbeit verwendeten Modellnamen lauten nun FM1, FM2, FM3, FM4 und FM5.
Durch den iterativen Fertigungsprozess eines solchen Automodells entsteht eine enorme Datenmenge, da die Modelle und ihre Parameter durch eine Vielzahl an Simulationen ständig angepasst werden. Somit waren mehrere Gigabyte an Daten des selben Modells vorhanden, die zuerst einmal auf Validität überprüft werden mussten, da durch verschiedene Einflüsse, Fehler in den Daten nicht vermieden werden können.
Mit Hilfe von halbautomatisierten Python-Skripten wurden die Daten verarbeitet und fehlerhafte FEM-Dateien entfernt. Die geometrische Struktur eines Fahrzeugmodells über mehrere Dateien unterschied sich nur geringfügig, weshalb pro Modell nur eine FEM-Datei ausgewählt wurde. Das Ergebnis stellen sechs valide FEM-Dateien dar, die weiterverarbeitet werden können. Im nächsten Schritt müssen die nötigen 3D-Informationen der jeweiligen Bauteile eines Modells aus diesen Daten extrahiert werden.

4.2 DATENAUFBEREITUNG

Wie in Kapitel 2 beschrieben, bestehen die FEM-Daten meist aus Keywords, die einen bestimmten Datenblock einleiten. Unter anderem sind die Punkte des 3D-Modells eines Fahrzeugs unter dem Keyword **NODE** gespeichert. In diesem tabellenartigen Datenblock befinden sich alle Punkte, die im Modell enthalten sind, jedoch ohne Verweis auf das zugehörige Bauteil. Um an die Punkte eines Bauteils zu gelangen, muss nach den entsprechenden Keywords gesucht werden. Bei den LS-Dyna Daten ist es das Keyword **PARTS** und bei den PAM-Crash Daten das Keyword **SHELLS**. In diesem Falle werden nur die Schalenvolumen berücksichtigt, da so die meisten Bauteile eines Fahrzeugs dargestellt werden. Bauteile eines Modells werden demnach separat in einzelnen Datenblöcken gespeichert. Zusätzlich besitzt jeder dieser Parts eine Bezeichnung, welche in der vorliegenden Arbeit das Label des jeweiligen Bauteils symbolisiert. Das Label stellt modellübergreifend die einheitliche und eindeutige Bezeichnung eines Bauteils dar und dient später zur Klassifikation.
In Abbildung 4.1 ist das Modell des Toyota Yaris als Punktwolke und in ANSA [Sol] visualisiert. In ANSA werden die Punkte jedes Bauteils mit einer anderen Färbe repräsentiert. Alle FEM-Modelle bestehen aus rund 350 verschiedenen Bauteilen, wovon nur eine kleine Menge in allen FEM-Daten wiederzufinden ist.



(a) Visualisierung des Toyota Yaris als Punktwolke **(b)** Visualisierung des Toyota Yaris in ANSA

Abbildung 4.1: Vergleich des FEM-Modells als Punktwolke und als Mesh

4.2.1 Parsen

Um nun die 3D-Daten extrahieren zu können, müssen die FEM-Dateien geparst werden. Dies geschieht mit Hilfe des *femparser*. Der *femparser* ist eine Python-Bibliothek entwickelt von der SCALE GmbH zum Verarbeiten und Parsen von FEM-Daten. Dabei werden die Dateien anhand der entsprechenden Keywords gescannt und in eine Mesh-Datenstruktur transferiert. Dadurch ist es möglich, auf die einzelnen Punkte der Bauteile zuzugreifen und iterativ die dazugehörigen Koordinaten zu extrahieren. Der *femparser* ist in der Lage verschiedenste Datentypen, wie die LS-Dyna Dateien und die PAM-Crash Dateien in die gleiche Form zu parsen, weshalb er für diese Arbeit verwendet wird.
Die daraus entstehenden Daten werden in einer JSON-Datei zwischengespeichert und mit dem Namen des entsprechenden Bauteils benannt, um die nachfolgenden Verarbeitungsschritte zu vereinfachen. Dieser Verarbeitungsschritt ist in Abbildung 4.2 dargestellt. Somit entsteht pro Bauteil eine JSON-Datei mit den jeweiligen 3D-Koordinaten des Bauteils.

Abbildung 4.2: Pipeline des Parsens von FEM-Inputdecks

Das Problem ist, dass jedes dieser Bauteile aus einer unterschiedlichen Anzahl an Punkten besteht. Die Anzahl ist jedoch von großer Bedeutung, da die aktuellen Architekturen der Convolutional Neural Networks eine gleichbleibende Größe der Punktwolken voraussetzen. Die vorliegenden Bauteile müssen demnach auf die gleiche Anzahl an Punkten gebracht werden. Dabei gilt es zu beachten, dass sich die grundlegende Form und Struktur des 3D-Modells nicht verändert wird. Eine Möglichkeit dieses Problem anzugehen ist es, die Oberflächen der Bauteile gleichmäßig abzutasten und die daraus generierten Punkte als Punktwolke zu speichern.

4.2.2 Sampling

Im vorherigen Abschnitt wurden die 3D-Geometrien aus den FEM-Daten extrahiert und in einer JSON-Datei gespeichert. Die daraus resultierende Datenstruktur besteht nun aus einer beliebigen Anzahl von n Facetten, die jeweils vier Punkte mit jeweiligen Koordinaten beinhalten. In diesem Abschnitt wird eine Methode beschrieben, die es ermöglicht die Anzahl der Punkte in der Punktwolke mittels Sampling auf eine konstante Größe zu bringen. Der mathematische Ansatz stammt aus der Arbeit von Pillai [Pil19]. Ziel ist es, eine möglichst gleichmäßig abgetastete Oberfläche zu erhalten, damit wenig bis keine Informationen der zugrundeliegenden 3D-Geometrie verloren gehen. Außerdem wird sich hier auf ein uniformes Sampling konzentriert, da die hier verwendeten Deep-Learning-Architekturen mit solchen Sampling-Methoden getestet wurden und die besten Ergebnisse liefern. Zusätzlich werden durch Pooling-Operationen markante Regionen der 3D-Modelle zusammengefasst und gruppiert, weshalb ein relevanz-basiertes Sampling zu Verlust von Informationen der Gesamtform führen kann [RACT15]. Ein genauerer Einblick auf die Funktionsweise der Deep-Learning-Algorithmen erfolgt in Kapitel 5.
Um eine uniforme Punktwolke aus einem 3D-Mesh zu generieren muss dessen Oberfläche gleichmäßig abgetastet werden und die komplette Grundform erfassen. Der trivialste Ansatz wäre zufällig Punkte in den Facetten der 3D-Geometrie zu generieren. Der extrahierte Mesh besteht jedoch aus vierseitigen Facetten und zusätzlich ist die Größe der eingeschlossenen Fläche einer Facette nicht gleichmäßig, wodurch ein uniformes Sampling erschwert wird. Die vorgestellte Sampling-Methode besteht aus zwei großen Schritten. Die Berechnung der Anzahl zu generierender Punkte pro Facette und die Triangulierung des Meshes mit zufälliger Erzeugung der ermittelten Punktanzahl pro Dreieck.

Zuerst wird eine feste Anzahl n an zu generierenden Punkten festgelegt, welche auf die einzelnen Facetten aufgeteilt werden muss. Um zu gewährleisten, dass das Objekt gleichmäßig abgetastet wird, müssen größere Facetten auch mehr Punkte beinhalten. Aus diesem Grund wird der relative Flächeninhalt einer Facette zur Gesamtfläche als Gewicht für die Anzahl der Punkte benutzt (siehe Gleichung 4.2.1). Die dabei entstehenden Rundungsfehler werden durch entsprechende Fallunterscheidungen ausgeschlossen.

$$n_{Dreieck} = \frac{A_i}{\sum_{i=1}^{n} A_i} * n_{Gesamt} \tag{4.2.1}$$

Im zweiten Schritt wird die vorher berechnete Punktanzahl in der gewählten Facette erzeugt. Dafür wird das in Kapitel 2 vorgestellte Konzept der baryzentrischen Koordinaten und des Latin Hyper Cube Samplings (LHS) verwendet. Die baryzentrischen Koordinaten lassen sich auf Dreiecke anwenden, weshalb der 3D-Mesh vorerst trianguliert werden muss. Am einfachsten geschieht das durch die Aufteilung der rechteckigen Facetten entlang einer der Diagonalen.

Angenommen es existiert ein Rechteck $\square ABCD$. Das Rechteck $\square ABCD$ wird in folgende zwei Dreiecke zerlegt: $\triangle ABC$ und $\triangle ACD$. Die Wahl der Diagonalen spielt dabei keine entscheidende Rolle, da die verwendete Fläche konstant bleibt und sich somit die Anzahl der Punkte in der Facette nicht ändert.
Nun können mit Hilfe der baryzentrischen Koordinaten Punkte im Dreieck gebildet werden. Baryzentrische Koordinaten sind Parameter, die einen Punkt in Abhängigkeit zu einem Simplex beschreiben. In diesem Fall können Punkte ermittelt werden, die in Abhängigkeit zu den Eckpunkten der jeweiligen Facette stehen. In Gleichung 4.2.2 ist die Berechnen eines Punktes p_{new} in

der Fläche des Dreiecks $\square ABCD$ zu sehen. Durch eine zufällige Wahl der Parameter u, v, w, kann ein Punkt generiert werden. Dabei muss beachtet werden, dass die Summe der baryzentrischen Koordinaten nicht größer als 1 wird, damit sich der Punkt im Dreieck befindet.

$$p_{new} = u * A + v * B + w * C \qquad \text{dabei gilt: } u + v + w \leq 1 \tag{4.2.2}$$

Um sicherzustellen, dass die zufällig erzeugten Punkte möglichst gleichmäßig im Dreieck verteilt sind, muss die Auswahl der Parameter u, v, w begrenzt werden. Dies geschieht durch das Latin Hypercube Sampling. Der Parameterraum wird in n Abschnitten in Richtung jeder Dimension zerteilt. n ist dabei die Anzahl der Punkte im Dreieck. Nun werden die Parameter so ausgewählt, dass genau nur ein Sample in Richtung der Achsen existiert. Somit wird eine gleichmäßige Abtastung der Dreiecksoberfläche erreicht. Dieser Vorgang wird auf alle Facetten im 3D-Mesh angewandt, mit dem Resultat einer möglich gleichförmig gesampleten Punktwolke (siehe Abbildung 4.3) .

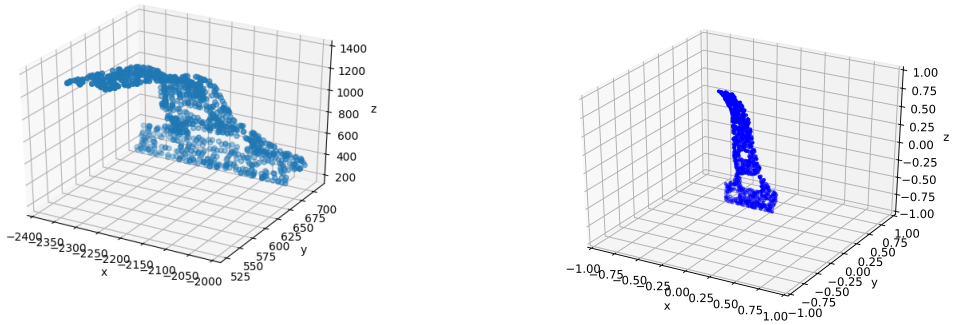
(a) Punktwolke einer A-Säule

(b) Punktwolke einer B-Säule

Abbildung 4.3: Uniform gesamplete Punktwolken von Bauteilen des Toyota Yaris Modells

4.2.3 Normalisierung

Da in den FEM-Daten ein gesamtes Fahrzeugmodell hinterlegt ist, sind die Koordinaten der einzelnen Bauteile des Fahrzeugs noch nicht zentriert oder normalisiert im dreidimensionalen Raum vorhanden. Damit die Bauteile modellübergreifend verglichen und angelernt werden können, müssen sie auf dieselbe Größe und Lage im Raum projiziert werden. Dazu werden die extrahierten und abgetasteten Punktwolken, wie in Kapitel 2 beschrieben, normalisiert. Das heißt der Zentroid der Punktwolke wird von den einzelnen Punkten abgezogen. Somit wird die Punktwolke hin zum Koordinatenursprung geschoben. Danach wird die maximale Distanz zweier Punkte im Raum ermittelt und die Punkte durch diese entsprechend geteilt.
Die daraus entstehenden Punkte sind invariant gegenüber der Position im Raum. Zusätzlich kann dadurch gewährleistet werden, dass die Punktwolken modellübergreifend verglichen werden können. Da unterschiedlich große Repräsentationen der gleichen Bauteile verschiedener Modelle zu Fehlern führen können. Außerdem wird durch die Normalisierung der Wertebereich der Koordinaten auf ein Intervall von $[-1, 1]$ gestaucht, wodurch sich die Werte verkleinern. Mit kleineren Werten verbessert sich die Effizienz bei komplexen Berechnungen enorm, weshalb die Normalisierung einen gängigen Vorverarbeitungsschritt bei Anwendungen für Deep-Learning-Architekturen darstellt. Abbildung 4.4b zeigt die normalisierte und zentrierte Punktwolke der B-Säule des Toyota Yaris.



(a) Extrahierte Punktwolke einer B-Säule **(b)** Normalisierte Punktwolke einer B-Säule

Abbildung 4.4: Vergleich gesamplete und normalisierte Punktwolke

Das Resultat der Vorverabeitungsschritte ist ein Array aus Punkten mit normalisierten Koordinaten extrahiert aus einem 3D-FEM Modell. Auf diesen Punktwolken können verschiedene Deep-Learning-Methoden angewandt werden. Es gilt nun einen Datensatz zu erstellen, der ausgewählte Bauteile verschiedener Automodelle vereint und für das Training einer solchen Netzarchitektur verwendet werden kann.
Im nachfolgenden Abschnitt wird genauer auf die Erstellung der Datensätze mit jeweiligen Vor- und Nachteilen eingegangen. Dabei besteht weiterhin das Ziel, die Generalisierung von aktuellen maschinellen Lernalgorithmen im Bereich der 3D-Objekterkennung nachzuweisen.

4.3 GENERIERUNG DER DATENSÄTZE

In den vorherigen Abschnitten wurde gezeigt, wie FEM-Daten vorverarbeitet werden müssen, um die entsprechenden Punktwolken für das Training zu extrahieren. Dieser Aufbereitungsprozess wurde mittels Python-Skripten realisiert, welche nun auf allen sechs vorhandenen FEM-Modelle, sprich den FM1, FM2, FM3, FM4, FM5 und Toyota Yaris angewandt werden können. Der Fokus liegt weiterhin darauf Datensätze zu erzeugen, die mehrere Versionen des gleichen Bauteils von unterschiedlichen Automodellen beinhalten. Ziel ist es, zu zeigen, dass Deep Learning Methoden in der Lage sind angelernte Bauteile zu abstrahieren und ein Bauteil eines anderen noch nicht gesehenen Automodells richtig zu klassifizieren.
Das Erzeugen eines qualitativen Trainingsdatensatzes wirft jedoch einige Probleme auf, die im Folgenden kurz genannt und anschließend im Detail erläutert werden.

- Geringe Anzahl an unterschiedlichen Automodellen
- Automodelle besitzen nur eine kleine Teilmenge gleicher Bauteile
- Keine einheitliche, modellübergreifende Bezeichnung der Bauteile

Geringe Anzahl an Modellen

Es besteht der Zugriff auf insgesamt sechs Automodellen. Wie im vorherigen Abschnitt beschrieben, wurde aus einer großen Menge an Daten, jeweils ein Modell also eine FEM-Datei ausgewählt. Die anderen Daten waren entweder unvollständige oder beinhalteten identische Koordinaten des 3D-Modells. Dies hat zur Folge, dass pro Modell nur eine Bauteilversion existiert. Um dieses Problem anzugehen, wurde entschieden mehrere Samples des gleichen Bauteils zu erzeugen. Demnach wird ein 3D-Mesh eines Bauteils mehrmals abgetastet und die entstandenen Punktwolken werden gespeichert. Dieses Verfahren stellt eine Art Data Augmentation (zu deutsch Datenerhöhung) dar, welches oft in Bereichen mit wenig Trainingsdaten Einsatz findet.
Durch die zufällige Abtastung entstehen leicht unterschiedliche Punktwolken, deren grundlegende Form unverändert bleibt. Dieser Sachverhalt wird in Abbildung 4.5 dargestellt, dabei sind leichte Unterschiede in der Punktverteilung zu erkennen. Es kann jedoch vorkommen, dass durch die zufällige Generierung einige Samples identisch sein können. Dieser Nachteil ist in Anbetracht der großen Dimensionen der Datensätze und Punktwolken vernachlässigbar gering. Eine mögliche Folge daraus wäre eine Überanpassung des Netzes, welches sich gut anhand der Accuracy-Werte im Trainingsprozess erkennen lässt.

(a) Sample der A-Säule des Toyota Yaris **(b)** Sample der A-Säule des Toyota Yaris

Abbildung 4.5: Vergleich zweier Samples des selben Bauteils

Teilmengen gleicher Bauteile

Es besteht nun eine Möglichkeit genügend Punktwolken für das Training von Deep-Learning-Architekturen zu erzeugen. Um die Generalisierung dieser nachzuweisen, werden die Varianten von Bauteilen unterschiedlicher Modelle benötigt. Da im Normalfall nicht alle Bauteile eines Fahrzeugmodells auch in einem anderen Modell enthalten sind, muss für die Datensätze eine Teilmenge an Bauteilen ausgewählt werden.

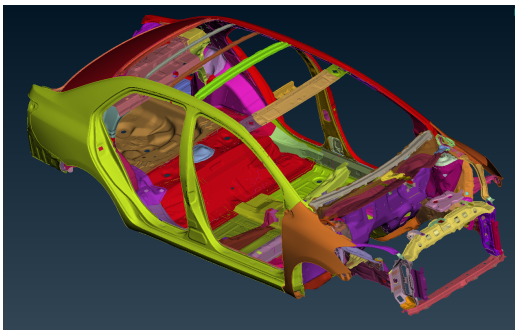


Abbildung 4.6: Bauteile des Toyota Yaris in ANSA visualisiert

In Abbildung 4.6 ist das komplette FEM-Modell des Toyota Yaris zu sehen, dass im Vergleich zu den Audi-Modellen aus einigen anderen Bauteilen besteht.
Aus diesem Grund wurde eine bestimmte Menge an Bauteilen ausgewählt, die in fast allen Automodellen gleichmäßig auftritt. Vereinzelt fehlende Teile wurden dabei weggelassen. Die Wahl fiel besonders auf markante Bauteile, die große Unterschiede aufweisen und zusätzlich die grobe Silhouette des Fahrzeugs abdecken. Die Menge der gewählten Bauteile ist in Abbildung 4.7 visualisiert. Außerdem wurden sowohl Außen- als auch Innenbleche der einzelnen Bauteile extrahiert sowie die äquivalenten Bauteile auf der gegenüberliegenden Seite des Fahrzeugs. Insgesamt wurden 35 verschiedene Fahrzeugteile extrahiert. Die abgebildete Menge an Bauteilen dient im Rahmen dieser Arbeit für nachfolgende Testversuche, die in Kapitel 6 im Detail beschrieben werden. Dabei stellen die Fahrzeugbauteile die Menge an Klassen dar, die für das Training verwendet werden. Sie bilden somit die Einheit, die durch den maschinellen Lernalgorithmus klassifiziert werden soll.

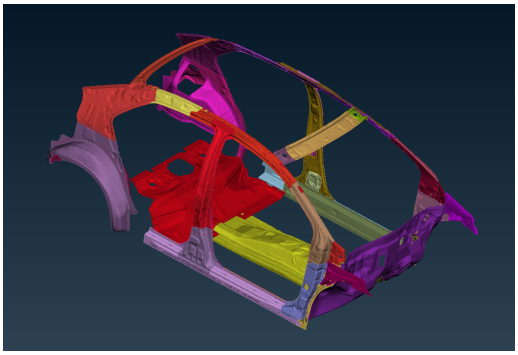


Abbildung 4.7: Menge an extrahierten Bauteilen für das Training

Nicht-einheitliche Bezeichnungen

Die ausgewählten Bauteilen musste nun aus den einzelnen FEM-Daten extrahiert werden. Dabei wurde jedoch deutlich, dass die Bezeichnungen der Bauteile sich von Modell zu Modell stark unterscheiden. Die Benennung der Teile gibt oftmals keinen genauen Aufschluss darüber, welches 3D-Modell sich dahinter befindet. Zusätzlich existieren einige Inkonsistenzen auch innerhalb eines Modells. Im Folgenden ist beispielhaft die Bezeichnung der B-Säule des Toyota Yaris aufgelistet. Im Gegensatz dazu besitzen die Modelle von Audi eine andere Struktur in der Bezeichnung der Bauteile, welche aus einer Teilenummer und internen Abkürzungen besteht.

494_bpillarupperinternalR — Bezeichnung der B-Säule des Toyota Yaris

Aus diesem Grund ist es nicht möglich, die Extraktion vollautomatisch mittels Python-Skripten durchzuführen. Um dieses Problem zu umgehen, wurden die Bezeichnungen der entsprechenden Bauteilklassen mit Hilfe der CAE Software ANSA händisch ermittelt. ANSA stellt verschiedene Mittel zur Visualisierung von FEM-Daten zur Verfügung und ist ein wichtiger Vertreter der CAE-Präprozessor Anwendungen. Die extrahierten Bauteilnamen wurden anschließend auf ein uniformes Labeling gemappt. Die neuen Labels der Bauteile besitzen somit eine einheitliche Form nach folgendem Muster:

[Bauteilname]_[oben/unten]_[innen/außen]_[rechts/links] Bespiel: bsaeule_innen_links

Dieses Mapping wurde in einer einfachen Textdatei gespeichert und ermöglicht nun den Extraktionsprozess vollautomatisch durchzuführen. Jedoch muss für jedes neue Automodell, welches antrainiert werden soll, ein solches händische Labeling durchgeführt werden. Außerdem kann vorkommen, dass gewisse Bauteile, wie zum Beispiel eine A-Säule, in mehrere noch kleinerer Bauteile unterteilt sind. In diesem Falle ist es möglich, durch händische Ermittlung der Namen aller kleinen Bestandteile, das große Bauteil als Gesamtes zu extrahieren. Hierfür wird ein Dictonary erstellt, bei dem das uniforme Label den Schlüssel und die Liste der Bauteilnamen den Wert darstellt. In Tabelle 4.1 ist das verwendete Mapping der internen Bezeichnungen des Toyota Yaris auf die uniformen Bauteilnamen dargestellt.

uniformer Bauteilname	Bauteilbezeichnung
bsaeule_innen_rechts	494_bpillarupperinternalR
bsaeule_innen_rechts	484_bpillarlowerR
bsaeule_schließteil_rechts	462_bpillarinner
asaeule_außen_rechts	465_apillarinnerupperR
asaeule_außen_rechts	463_roofrailrearR
asaeule_innen_rechts	464_apillarupperinnerR
asaeule_innen_rechts	471_roofrailrearR
asaeule_unten_außen_rechts	469_apillarlowerR
asaeule_unten_innen_rechts	315_apillarinnerlower
csaeule_oben_rechts	317_cpillarR
radhaus_außen_rechts	485_wheelwellrearouterR
radhaus_innen_rechts	208_wheelwellinnnerearR
schweller_außen_rechts	460_rockerpanelR
schweller_innen_rechts	461_rockerpanelinnerR

Tabelle 4.1: Mapping der Bauteilbezeichnungen des Toyota Yaris

Die Bauteilmenge besitzt somit einheitliche und modellübergreifende Bezeichnungen. Um nun die Datensätze zu generieren, muss zu jedem Bauteil die zugehörige Klasse angegeben werden, da es sich bei neuronalen Netzen, um eine Methode des überwachten Lernens handelt (siehe Kapitel 2). Das Zuordnen einer Klasse zu einem Bauteil erfolgt, wie auch das einheitliche Benennen, händisch über ein Mapping. Dabei werden die Bauteilnamen auf eine Zahl von 0 - $(n-1)$ gemappt, wobei n die Anzahl der Klassen ist.

Diese Methodik besitzt den Vorteil, dass die Zuordnung der Klassen schnell und einfach geändert werden kann. Dadurch können verschiedene Testversuche mit unterschiedlicher Klassengröße durchgeführt werden. Besonderer Fokus liegt dabei auf der Granularität der Klassen. Mit Granularität wird in diesem Kontext die Struktur der Untergliederung der Klassen gemeint, das heißt, wie genau kann der Algorithmus auch ähnliche Bauteile voneinander unterscheiden. Dabei geht es besonders um formgleiche Bauteile, wie zum Beispiel das Außen- und Innenblech einer A-Säule. Diese Testversuche werden im Kapitel 6 genauer beleuchtet und analysiert. Ein Beispiel eines solchen Klassenmappings wird in Tabelle 4.2 dargestellt. Darauf sind links die uniformen Bezeichnungen der Bauteile und rechts die zugeordneten Klassen als Zahl zu sehen.

Bauteilname	Klasse
bsaeule_innen_links	0
bsaeule_schließteil_links	0
asaeule_außen_links	1
asaeule_innen_links	1
asaeule_unten_außen_links	2
asaeule_unten_innen_links	3
csaeule_oben_außen_links	4
csaeule_oben_innen_links	4
csaeule_unten_links	5
radhaus_außen_links	6
radhaus_innen_links	7
schweller_verst_links	8
schweller_außen_links	8
schweller_innen_links	8
boden_hinten	9
mitteltunnel	10
stirnwand	11
dachrahmen_vorn	12
dachrahmen_hinten	13
lenktraeger_links	14

Tabelle 4.2: Mapping der Bauteilnamen auf entsprechende Klassen

Struktur der Datensätze

Das entstandene Konzept der Vorverarbeitung lässt sich anhand einer Prozesskette, wie in Abbildung 4.8 zu sehen, gut beschreiben. Prinzipiell unterteilt sich der Prozess in zwei große Teile: Der Extraktion der Daten und der Generierung der Datensätze. Mit einer händischen Vorauswahl der Fahrzeugbauteile und der Zuordnung zu bestimmten Klassen, lassen sich die Vorgänge halbautomatisiert für jedes FEM-Modell anwenden. Die verwendeten Python-Skripte werden im Folgenden nur semantisch beschrieben. Auf den Code wird kein genauerer Blick geworfen.

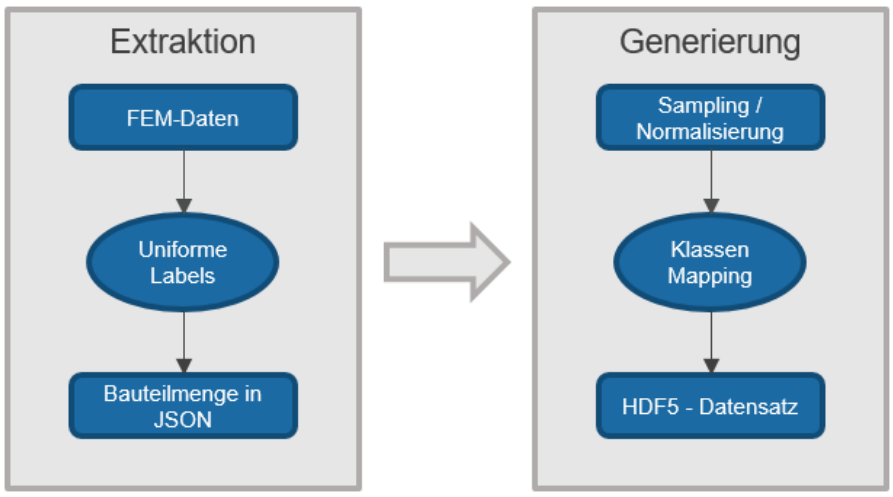


Abbildung 4.8: Grafische Darstellung des Vorverarbeitungsprozesses

Den Extraktionsschritt übernimmt ein Skript, dass als Eingabe eine FEM-Datei und eine Textdatei mit uniformen Namen entgegen nimmt. Anhand dieser händisch erstellten Textdatei werden die entsprechenden Bauteile innerhalb der FEM-Datei gesucht und mittels des *femparsers* in eine Mesh-Struktur überführt. Die daraus extrahierten Knoten werden samt ihrer Koordinaten facettenweise in einer JSON-Datei gespeichert, welche die uniforme Bezeichnung des Bauteils aus der Textdatei erhält.
Die Generierung der Datensätze wird von einem zweiten Skript ausgeführt, dass einen Ordner mit Bauteilen im JSON-Format und eine Textdatei mit der Zuordnung der Klassen empfängt. Zusätzlich können noch über die Befehlszeilenargumente der Speicherort, die Anzahl der Samples, die Größe der Punktwolke und die Normalisierung verändert werden. Mittels dieser Eingabeparameter werden die Bauteile im Ordner abhängig von der Datensatzgröße mehrfach gesamplet und mit der dazugehörigen Klasse als Punktwolke in einer HDF5-Datei gespeichert. Das Resultat stellt ein Datensatz dar, der mehrere Samples pro extrahiertes Bauteil beinhaltet und aus exakt n Samples besteht.

HDF steht für Hierachical Data Format und ist ein Dateiformat, dass vor allem in wissenschaftlichen Bereichen zur Speicherung von großen Datenmengen verwendet wird [Grob]. Es bietet die Möglichkeit, besonders speichereffizient, mehrdimensionale Tabellen zu speichern, wodurch es sich im Bereich des maschinellen Lernens bewährt hat. Jedes abgetastete Bauteil wird mitsamt seiner Klasse gespeichert. Somit entsteht genau ein HDF5-Datensatz pro FEM-Datei. Im Rahmen dieser Arbeit wurden mehrere Datensätze generiert. Im Allgemeinen wurde sich für eine Struktur entschieden, die bei den Tests der hier verwendeten Deep-Learning-Architekturen Anwendung fand. Insgesamt sind demnach sechs Datensätze generiert wurden, jeweils einer pro FEM-Modell. Dabei dient der Datensatz des Audi FM3 als Testdatensatz mit einer geringeren Anzahl an Samples. Im Folgenden wird die Struktur der Datensätze tabellarisch zusammengefasst (siehe 4.3).

Größe der Punktwolken:	1024
Anzahl der Samples (Training):	2048
Anzahl der Samples (Test):	1024
Dauer der Generierung:	60-90 Minuten

Tabelle 4.3: Struktur der Datensätze

Die Datensätze wurden abhängig vom jeweiligen Testszenario angepasst und neu generiert. Im nachfolgenden Kapitel wird ein genauerer Einblick auf die verwendeten Deep-Learning-Architekturen gegeben. Diese werden auf Grundlage der hier generierten Datensätze miteinander verglichen. Anschließend erfolgt eine Analyse der Resultate der Benchmarks mit dem Fokus auf die Fehleranfälligkeit der einzelnen Bauteile.

5 METHODEN ZUR ERKENNUNG VON 3D PUNKTWOLKEN

Kapitel 5 stellt zwei aktuelle Methoden des maschinellen Lernens im Bereich der Erkennung von 3D-Objekten vor. Bei diesen Ansätzen handelt es sich um vollständige Deep-Learning-Architekturen, welche anhand von großen Datensätzen evaluiert wurden. Beide Verfahren konzentrieren sich ausschließlich auf die Klassifikation von Punktwolken und erzielen auf dem bekannten ModelNet40 Datensatz sehr gute Ergebnisse.
Ziel dieses Kapitel ist es, die grundlegende Architektur und Funktionsweise beider Ansätze genauer zu beleuchten und dabei auf entstehende Vor- und Nachteile der Konzepte einzugehen. Zusätzlich sollen beide Methoden im Rahmen dieser Arbeit mit den im vorherigen Kapitel erzeugten Datensätzen trainiert und miteinander verglichen werden. Diese Benchmarks werden in Kapitel 6 behandelt und sollen Aufschluss darüber geben, welcher Ansatz besser auf der aktuellen Domäne abschneidet und somit für spätere Anwendungen am geeignetsten erscheint.

Die erste Architektur ist *PointNet*, welche auch in der Arbeit von Pillai [Pil19] Verwendung fand. Pillai hat gezeigt, dass sich *PointNet* zum Klassifizieren eines Fahrzeugmodells eignet. Nun muss überprüft werden, wie gut *PointNet* die gelernten Punktwolken abstrahieren kann. Die dabei erlangten Ergebnisse dienen als Vergleichswert für den zweiten Ansatz. Als zweite Methode wird die Architektur von Ben-Shabat [BLF17] verwendet. Die sogenannten *3D modified Fisher Vectors* (Abk. 3DmFV) wurden auf Basis der PointNet Architektur entwickelt und sollen laut Ben-Shabat die Ergebnisse von PointNet schlagen. Das grundlegende Konzept dieses Ansatzes ist jedoch ein anderes und wird in diesem Kapitel genauer betrachtet.

5.1 POINTNET

Dieser Abschnitt beschreibt kurz die Architektur und Funktionsweise der Deep-Learning-Architektur *PointNet*, welche von Charles R. Qi 2016 vorgestellt wurde und ist daher an sein veröffentlichtes Paper [QSMG16] angelehnt.

Laut Qi et al. benötigen die meisten Convolutional Architekturen eine gleichmäßige Struktur an Eingabedaten. Aus diesem Grund werden oft 3D-Grids oder Voxels verwendet. Da Punktwolken und Meshes nicht über so eine strikte Form verfügen, werden sie meist in ein passendes Format übertragen. Diese Transformation ist jedoch ineffizient und kann zusätzlich zu Informationsverlust während des Prozesses führen. Deshalb stellen Qi et al. in ihrer Arbeit *PointNet* vor. Eine Deep-Learning-Architektur die in einigen Aufgabenbereichen, wie beispielsweise Objekterkennung, Segmentierung und semantische Szenenerkennung Anwendung finden kann. Der Ansatz wurde speziell auf den besonderen Eigenschaften der Punktwolken entwickelt und stellt somit eine effiziente und effektive Methode dar.

Architektur

PointNet ist eine umfangreiche, abgeschlossene Deep-Learning-Architektur, welche Punktwolken als direkte Eingabe empfängt und ein entsprechendes Klassenlabel ausgibt. Eine Punktwolke ist eine Menge aus 3D-Punkten $\{P_i | i = 1, ..., n\}$, bei der jeder Punkt P_i einen Vektor mit den jeweiligen (x, y, z) Koordinaten darstellt. Als Ausgabe dienen k Klassenbewertungen die angeben, wie gut das Objekt zur jeweiligen Klasse zugeordnet werden kann.
Die Menge an Punkten im euklidischen Raum besitzt laut Qi et al. drei Haupteigenschaften, welche bei der Entwicklung von PointNet eine entscheidende Rolle gespielt haben.

Ungeordnete Menge Im Gegensatz zu Pixeln oder Voxel ist eine Punktwolke unstrukturiert und besitzt keine bestimmte Anordnung. Punkte können in jeder beliebigen Reihenfolge in der Menge vorkommen, weshalb die Architektur invariant zu allen möglichen Permutationen der Punktwolke sein muss.

Interaktion zwischen Punkten Die Punkte kommen aus einem euklidischen Raum und besitzen somit eine Abhängigkeit zueinander. Benachbarte Punkte stellen demnach wichtige Merkmale dar, die es nicht zu vernachlässigen gilt. Deshalb muss PointNet in der Lage sein lokale Merkmale von Punktwolken zu extrahieren.

Transformationsinvarianz Punktwolken sind geometrische Objekte im Raum. Daher sollten die angelernten Merkmale einer Punktwolke invariant zu bestimmten Transformationen sein, wie beispielsweise das Rotieren oder Verschieben der Punktwolke im Raum.

Die komplette Architektur von PointNet ist in Abbildung 5.1 zu sehen. Im Folgenden werden die Hauptkomponenten des Netzes beschrieben, dabei liegt der Fokus besonders auf die von Qi et al. getroffenen Designentscheidungen.
Um in PointNet Invarianz gegenüber allen Permutation an Eingabe-Punktwolken zu erreichen, sind laut Qi et al. drei verschiedene Strategien möglich: Das Sortieren in kanonischer Reihenfolge, das Anlernen eines RNN und die Aggregation von Information der Punkte über eine symmetrische Funktion. Qi et al. entschieden sich in ihrer Arbeit für die Verwendung einer symmetrischen Funktion. Diese Designentscheidung stellt eine Kernidee von PointNet dar.

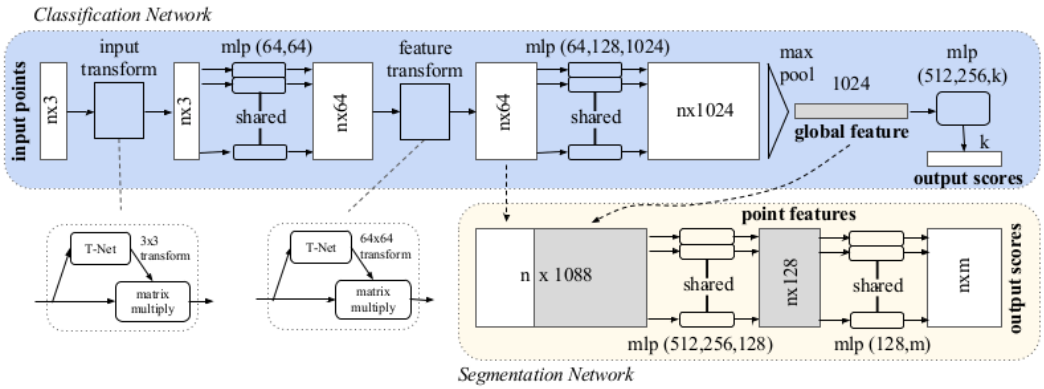


Abbildung 5.1: Darstellung der gesamten Architektur von PointNet [QSMG16]

Eine symmetrische Funktion ist eine Funktion, die unabhängig von der Permutation ihrer Eingabeparameter immer dieselbe Ausgabe liefert. Beispielsweise ist die Summen-Funktion eine bekannte symmetrische Funktion, denn egal in welcher Reihenfolge die Summanden addiert werden, das Ergebnis bleibt unverändert. Laut Qi et al. stellte sich heraus, dass die Max-Pooling Operation mitunter die besten Testergebnisse erzielt. Demnach dienen n Vektoren als Eingabe für eine Max-Operation und als Ausgabe entsteht ein neuer Vektor der invariant zur Eingabepermutation ist. Die mathematische Beschreibung dieses Sachverhalts ist in Gleichung 5.1.1 zu finden. Dabei ist die Funktion h eine Approximation mittels eines Multi-Layer Perzeptrons und g die Max-Pool Funktion, also demnach eine symmetrische Funktion.

$$f(x_1, ...x_n) \approx g(h(x_1), ..., h(x_n)) \tag{5.1.1}$$

Die zweite Komponente stellt das Segmentierungs-Netz dar, welches sich um das Problem der lokalen Merkmale der einzelnen Punkte kümmert und diese extrahiert. Darauf wird an dieser Stelle nicht genauer eingegangen, da das zugrundeliegende Problem sich auf ein Klassifikationsproblem beschränkt.
Die letzte Designentscheidung von PointNet ist die Invarianz gegenüber geometrischer Transformation. Diese Eigenschaft ist für den vorliegenden Anwendungsfall von großer Bedeutung, da nicht davon ausgegangen werden kann, dass alle 3D-Objekte immer die gleiche Lage im Raum besitzen. Die von Qi et al. vorgestellte Lösung besteht aus einem Mini-Netz, genannt *T-Net*, welche eine affine Transformation vorhersagen kann und diese sofort auf die ursprünglichen Koordinaten der Punktwolke anwendet. Zusätzlich ist es möglich diesen Ansatz auf den extrahierten Merkmalen anzuwenden, um eine Transformationsmatrix der Merkmale verschiedener Punktwolken vorherzusagen und richtig auszurichten.

PointNet wurde mit dem ModelNet40 Datensatz evaluiert und erreichte eines der besten Resultate im Stand der Technik. Insgesamt erzielt PointNet eine Accuracy von 89,2 % und gute Ergebnisse in den Robustheitstests. Qi et al. zeigten in ihrer Arbeit das PointNet besonders gut grobe Formen und Strukturen erkennen kann, da die Eingabepunktwolken auf merkmalsreiche Regionen reduziert werden. Der größte Vorteil den PointNet bietet, ist die geringe Komplexität gegenüber anderen Methoden im Stand der Technik. In Tabelle 5.1 ist ein Vergleich der Parameteranzahl und FLOPs zwischen PointNet und zwei anderen Vertretern zu sehen.

	Anzahl Parameter	FLOPs/Sample
PointNet (vanilla)	0.8 Mio.	148 Mio.
PointNet	3.5 Mio.	440 Mio.
Subvolume [QSN+16]	16.6 Mio.	3633 Mio.
MVCNN [SMKL15]	60 Mio.	62057 Mio.

Tabelle 5.1: Vergleich der Effizienz von PointNet und anderen Stand der Technik Methoden [QSMG16]

Da es sich um Deep-Learning-Architekturen handelt, ist die Parameterzahl generell hoch. PointNet besitzt jedoch rund viermal weniger Parameter als seine Mitstreiter und benötigt weniger Berechnungen pro Sample. Zusätzlich ist die Komplexität nur linear zur Anzahl an Eingabepunkten. PointNet ist implementiert in TensorFlow und frei zugänglich auf Github zu finden. In dieser Arbeit wird die PointNet Architektur benutzt, um die Machbarkeit der Klassifikation von 3D-FEM-Daten zu untersuchen. Weiterhin liegt dabei der Fokus auf der Generalisierung dieser Methode. Der Aufbau der durchgeführten Versuche wird später im Kapitel 6 genauer beleuchtet. Vorerst wird der zweite Ansatz der in dieser Arbeit verfolgt wird vorgestellt.

5.2 3D MODIFIED FISHER VECTORS

Dieser Abschnitt widmet sich der zweiten Deep-Learning-Architektur, die sich ebenfalls auf das Klassifizieren von Punktwolken fixiert hat. Der Ansatz wurde 2017 in der Arbeit von Ben-Shabat et al. veröffentlicht [BLF17]. Die Motivation des Ansatzes ist ähnlich wie die von *PointNet*. Es soll eine effiziente Methode zur Klassifikation und Segmentierung entwickelt werden, die die Punktwolken in nicht speicheraufwendigen Datenstrukturen transferiert. Ben-Shabat et al. stellen in ihrer Arbeit einen neue Darstellungsform für Punktwolken vor und nennen diese *3D Modified Fisher Vectors (3DmFV)*. Die Repräsentation ist ein Hybrid aus einer diskreten Gitterstruktur und der Generalisierung des Fisher-Vektors. Im Folgenden wird genauer auf die neue Darstellungsform eingegangen und die Gesamtarchitektur des 3DmFV-Ansatzes erläutert.

Fisher-Vektor

Perronnin et al. stellten, basierend auf den nach Ronald Fisher benannten Fisher-Kernel [JH99], eine alternative Deskriptor-Methode vor, den Fisher-Vektor [PSM10]. Der Fisher-Vektor charakterisiert unterschiedlich große Samples in Abhängigkeit ihrer Standardabweichung in einer multivariaten Verteilung, in diesem Fall ein Gaußsches Mischmodell (GMM). Dabei werden die Gradienten der Wahrscheinlichkeit der Samples in Bezug auf die Modelparameter (beispielsweise Gewicht und Kovarianz) berechnet.

Da diese Repräsentation unabhängig von der Samplegröße ist, stellt sie eine gute Wahl für Punktwolken dar. Ähnlich wie viele Voxel-Ansätze (siehe Kapitel 3) bedient sich der 3DmFV-Ansatz einer Gitterstruktur, die eine einfachere Verarbeitung mittels CNNs ermöglicht, jedoch zusätzlich weniger unter den Nachteilen der Quantisierung leidet. Genau wie bei *PointNet* bestehen die Features aus symmetrischen Funktionen, um Invarianz gegenüber Reihenfolge und Struktur zu gewährleisten.

Grundlegend besteht der 3DmFV-Ansatz aus zwei Komponenten: Die erste Komponente nimmt eine Punktwolke als Eingabe entgegen und konvertiert diese zur vorgestellten *3D Modified Fisher Vector* Repräsentation. Die zweite Komponente besteht aus einem CNN, welches die entsprechende Darstellung verarbeitet und einen Vektor von Klassenbewertungen ausgibt. Im Folgenden wird basierend auf der Arbeit von Ben-Shabat et al. die mathematischen Grundlagen des Fisher-Vektors beschrieben.
Der Fisher-Vektor basiert auf der Wahrscheinlichkeit eines Punktes p in einem Gaußschen Mischmodell. Gegeben sei $X = \{p_t \in \mathbb{R}^3, t = 1, ..., T\}$ die Menge an 3D Punkten in einer Punktwolke. Die Menge der Parameter eines GMM mit K Komponenten ist gegeben durch $\lambda = \{(w_k, \mu_k, \Sigma_k), k = 1, ..., K\}$ mit w_k, μ_k, Σ_k als Gewicht, Erwartungswert und Kovarianzmatrix des k-ten Gaußians. Die Wahrscheinlichkeit eines Punktes p in Abhängigkeit zum k-ten Gaußian berechnet sich wie folgt:

$$u_k(p) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\{-\frac{1}{2}(p-\mu_k)'\Sigma_k^{-}1(p-\mu_k)\} \tag{5.2.1}$$

Die Wahrscheinlichkeit eines einzelnen Punktes in Bezug auf die gesamte Dichtefunktion des GMM kann mit folgender Formel berechnet werden:

$$u_\lambda(p) = \sum_{k=1}^{K} w_k u_k(p) \tag{5.2.2}$$

Dabei wird die Wahrscheinlichkeit eines Punktes zum Gaußian über alle Gaußiane K berechnet und mit dem Gewicht w_k multipliziert. Die gewichtete Wahrscheinlichkeit eines Punktes p_t zu einem Gaußian k ist definiert durch:

$$\gamma_t(k) = \frac{w_k u_k(p_t)}{\sum_{j=1}^{K} w_j u_j(p_t)} \tag{5.2.3}$$

Der Fisher-Vektor besteht nun aus den normalisierten Gradienten der Dichtefunktion in Abhängigkeit zu ihren Parametern. Es existieren somit drei Gradienten, zum Gewicht (Gleichung 5.2.4), zum Mittelwert (Gleichung 5.2.5) und zur Kovarianz (Gleichung 5.2.6). Diese Gradienten bilden die einzelnen Komponenten des Fisher-Vektors.

$$\mathcal{G}_{\alpha_k}^X = \frac{1}{\sqrt{w_k}} \sum_{t=1}^{T} (\gamma_t(k) - w_k) \tag{5.2.4}$$

$$\mathcal{G}_{\mu_k}^X = \frac{1}{\sqrt{w_k}} \sum_{t=1}^{T} \gamma_t(k) (\frac{p_t - \mu_k}{\sigma_k}) \tag{5.2.5}$$

$$\mathcal{G}_{\sigma_k}^X = \frac{1}{\sqrt{2w_k}} \sum_{t=1}^{T} \gamma_t(k) [\frac{(p_t - \mu_k)^2}{\sigma_k^2} - 1] \tag{5.2.6}$$

Die Gradienten werden über die Summe aller Punkte gebildet. Zusätzlich wird der daraus resultierende Fisher-Vektor durch die Größe der Punktwolke normalisiert und somit invariant zur Punktwolkengröße. Der Fisher-Vektor stellt somit eine genaue Beschreibung der Dichteverteilung im Gaußian-Gitter dar. Diese neue Darstellungsform ist laut Ben-Shabat et al. vielversprechend, da sie eine feste Größe besitzt (Anzahl der Gradienten im Fisher-Vektor) und unabhängig von der Anzahl der Punkte in der Punktwolke ist. Außerdem ist die Repräsentation als Fisher-Vektor invariant gegenüber der Reihenfolge und Struktur. Es ist selten, dass Deep-Learning-Architekturen bereits extrahierte Features als Eingabe bekommen. Der Grund dafür ist, dass durch diese Extraktion wichtige Charakteristiken der Eingabe verloren gehen können, wie beispielsweise bei diskreten Verfahren. Nach Ben-Shabat et al. leidet der neue Ansatz weniger unter diesem Problem. Einerseits hat die Darstellung über das GMM viel mehr Variablen und gilt somit als überbestimmtes Gleichungssystem. Anderseits werden in der Arbeit verschiedene Rekonstruktionen aus einem Fisher-Vektor zur Punktwolke beschrieben. Diese Argumente zählen nicht als mathematische Beweise, werden aber in der vorliegenden Arbeit in dieser Form akzeptiert.

Der Raum von $[-1, 1]$ in jeder Dimension wird durch ein gleichmäßiges Gitter aus $m \times m \times m$ Gaußianen abgebildet. Diese Gaußiane sind Sphären im dreidimensionalen Raum und besitzen alle einen Mittelpunkt μ_k. Die anderen Parameter des GMM sind für alle Gaußiane identisch. Die Gewichte sind gleichmäßig über die Anzahl der Gaußiane $w_k = \frac{1}{K}$ und die Kovarianzmatrix mit $\Sigma_k = \sigma_k I$, wobei $\sigma_k = \frac{1}{m}$ ist. Die Anzahl der im Gitter verwendeten Gaußiane m kann angepasst werden, wodurch der Raum detaillierter dargestellt werden kann.
Ähnlich wie bei PointNet machen sich Ben-Shabat et al. die Funktionsweise der symmetrischen Funktionen zu Nutze. Der Fisher-Vektor ist bereits die Summe der Gradienten über alle Punkte der Punktwolke. Zusätzlich wird der Vektor durch weitere symmetrische Funktionen erweitert. Dabei handelt es sich um die Max- und Min-Funktion. Ein Fisher-Vektor für einen bestimmten Gaußian k besteht somit aus 20 Komponenten. Die beste Darstellungsform der *3D Modified Fisher Vectors* ist eine $20 \times K$ Matrix, von nun an Fisher-Matrix genannt.

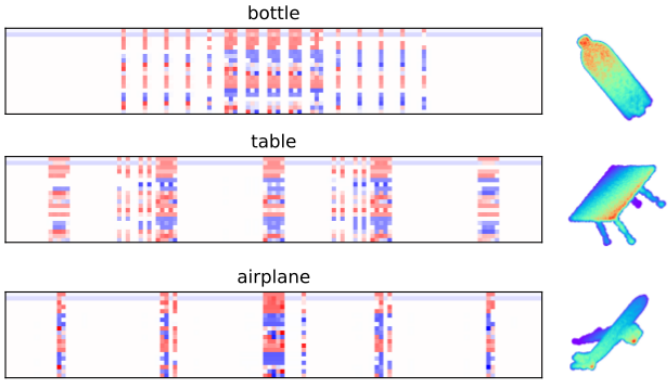


Abbildung 5.2: 3DmFV Darstellung als Matrix von drei Objekten [BLF17]

In Abbildung 5.2 sind drei 3DmFV-Matrizen dargestellt und daneben die dazugehörigen Punktwolken abgebildet. Jede Spalte entspricht dabei einem Gaußian im Raum. Dabei stellen die Ränder der Matrix auch die Gaußiane am Rand im Raum dar. Gleiches gilt für den Mittelpunkt. Um diese Darstellungsform richtig zu interpretieren, muss die Matrix als dreidimensionaler Würfel betrachtet werden, der schichtweise ausgelesen wurde. Zellen die weiß sind symbolisieren Null-

werte, hingegen sind die positiven Gradienten rot und die negativen blau gefärbt. Die weißen Spalten gehören demnach zu den Gaußianen, bei denen sich keine Punkte in der Nähe befinden. Es ist deutlich zu sehen, dass die gezeigten Objekte sehr unterschiedliche Repräsentationen erzeugen. Das Zentrum der Matrix spricht in den Beispielen immer stark an, da die Punktwolken sich zentriert und normalisiert im Raum befinden. Die Fisher-Matrix in dreidimensionaler Form dient nachfolgend als Eingabe für ein 3D-CNN.

Architektur

Das Netz bekommt als Eingabe eine normalisierte und zentrierte Punktwolke und konvergiert diese im ersten Schritt in die 3DmFV-Darstellung. In Abbildung 5.3 ist die komplette Architektur des 3DmFV-Ansatzes mit beiden Hauptmodulen abgebildet.

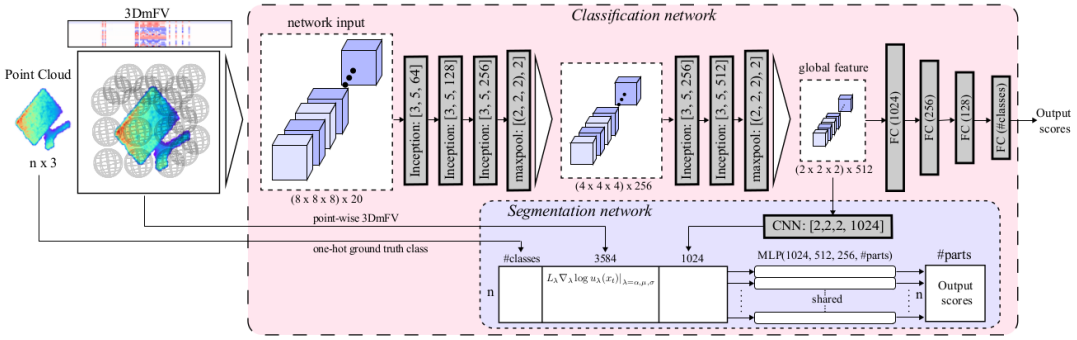


Abbildung 5.3: Darstellung der gesamten Architektur von 3DmFV [BLF17]

Die Funktionsweise eines 3D-CNN unterscheidet sich nur gering von der eines 2D-CNN (siehe Kapitel 2). Grundlegend wird die dreidimensionale Eingabematrix mit einem entsprechenden 3D-Filter gefiltert, wodurch bestimmte Features in der Matrix erlernt werden können. Das besondere an der 3DmFV-Architektur ist die Verwendung der sogenannten *Inception Modules*. Ein *Inception Module* ist eine besondere Anordnung von mehreren Convolutional Layern. Dabei wird die Eingabe durch mehrere Schichten mit variabler Filtergröße gefiltert und am Ende konkateniert, sodass eine $m \times m \times m \times 3N$ große Ausgabematrix entsteht (siehe Abbildung 5.4).

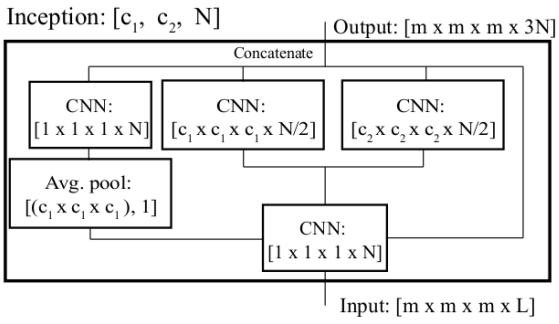
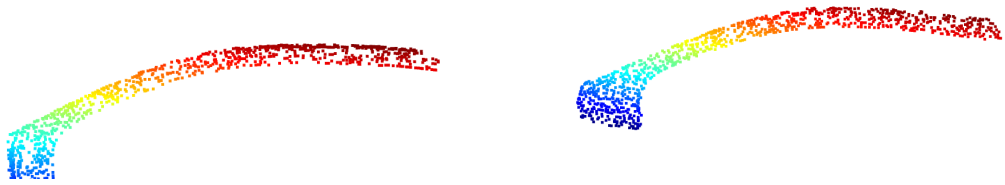


Abbildung 5.4: Darstellung einer Inception-Schicht [BLF17]

Nach den Inception Layern folgen typische Max-Pooling Layer, um die Menge an Daten zu verkleinern und globale Features zu extrahieren. Am Ende der Architektur befindet sich ein Fully-Connected Layer, ähnlich wie bei dem traditionellen 2D-CNN, um die 3D-Features zu klassifizieren. Das Netz benutzt den Softmax Cross-Entropy Loss und mit Batch Normalisierung und Dropout nach jedem FC-Layer trainiert.
Insgesamt besitzt die Architektur über 4,6 Mio. Parametern und somit leicht mehr, als die von PointNet. Implementiert ist die 3DmFV-Methode in Tensorflow, wobei der Aufbau des Codes stark an der Struktur der PointNet Architektur angelehnt ist. Die Deep-Learning-Architektur wurde auch mit dem ModelNet40 Datensatz evaluiert. Laut Ben-Shabat et al. erreicht der Ansatz dabei sehr gute 91.4% und ist damit unter den anderen Stand-der-Technik-Methoden. Die neue Repräsentationform von Punktwolken liefert die Motivation dieser Forschung, weshalb dieser Ansatz weiter verfolgt wird. Nachfolgend werden sowohl die PointNet-Architektur sowie die 3DmFV-Architektur auf den in Kapitel 4 generierten Daten trainiert und mittels verschiedener Testversuche evaluiert. Ziel ist es, zu analysieren, welche Methode sich in der Domäne der FEM-Daten besser schlägt, um diese für eine genauere Analyse auszuwählen.

6 EVALUATION

Im Folgenden werden die vorgestellten Ansätze verschiedenen Benchmarks unterzogen. Den Kern der Untersuchung bildet hierbei die Generalisierung der maschinellen Lernalgorithmen. Das bedeutet, wie gut können die Ansätze Merkmale aus einem bestimmten Trainingsdatensatz erkennen und diese in einem noch nicht gesehenen Testdatensatz wiederfinden. Zusätzlich soll festgestellt werden, wo die Limitierungen der Klassifikation der einzelnen Methoden liegen. Genauer gesagt wird untersucht, wie sich die Granularität der Klassen zu der resultierenden Genauigkeit verhält. Grobe Granularität bedeutet in diesem Kontext eine Zuordnung zu gesamten Baugruppen (beispielsweise B-Säule, A-Säule) und feine Granularität eine Unterscheidung zwischen Innen- und Außenteilen.
Es muss demnach klargestellt werden, dass ein Fahrzeugteil aus mehreren Elementen bestehen kann. So gehört zu einer Baugruppe A-Säule meist das A-Säulen-Innenblech und das A-Säulen-Außenblech. In Abbildung 6.1 sind diese beiden Teile abgebildet. Es ist deutlich zu erkennen, dass die grobe Form nahezu deckungsgleich ist und sie eine sehr ähnliche Struktur in der Darstellung als Punktwolke besitzen. Aus diesem Grund wird bei den Tests mit einem groben Labeling sowohl A-Säulen-Außenblech als auch A-Säulen-Innenblech der Klasse A-Säule zugeordnet. Im anderen Fall wird zwischen diesen beiden Klassen unterschieden.



(a) Außenblech einer A-Säule **(b)** Innenblech einer A-Säule

Abbildung 6.1: Vergleich der Bauteile einer A-Säule des Toyota Yaris

Zusätzlich wird getestet, wie gut die Klassifikation bei der Unterscheidung zwischen linken und rechten Bauteilen abschneidet. Ziel der Untersuchung ist es, mögliche Faktoren einer Fehlklassifikation, in diesem Fall die Ähnlichkeit der Bauteile, herauszufinden und zu analysieren.

6.1 BENCHMARKS

Für das Training von PointNet und 3DmFV wurden die in Kapitel 4 erzeugten Datensätze verwendet. In Tabelle 6.1 sind alle Datensätze aufgelistet. Fünf der insgesamt sechs Datensätze sind Fahrzeugmodelle der Marke Audi. Der sechste Datensatz beinhaltet die Bauteile des Toyota Yaris. Die Anzahl der extrahierten Bauteile unterscheidet sich leicht zwischen den einzelnen Datensätzen, da nicht alle Bauteile in jedem Modell vorhanden sind. So fehlen im FM5 beispielsweise der *Dachrahmen_hinten* und das *Radhaus_hinten*. Außerdem variiert die Anzahl der Bauteile je nach Art des Benchmarks. So wurden beispielsweise bei den Tests zur Unterscheidbarkeit linker und rechter Bauteile lediglich Bauteile verwendet, die ein Pendant auf der gegenüberliegenden Seite besitzen. Die Anzahl der Klassen bzw. Labels unterscheidet sich je nach Testfall und wurde zuvor händisch angepasst. In den folgenden Benchmarks wird gemäß deren Reihenfolge zwischen 15, 16 und 13 Klassen unterschieden.

Datensatz	Anzahl der Bauteile	Anzahl der Klassen	Art	Anzahl der Samples
Audi FM1	31	(15, 16, 13)	Training	2.048
Audi FM2	31	(15, 16, 13)	Training	2.048
Audi FM3	35	(15, 16, 13)	Test	1.024
Audi FM4	33	(15, 16, 13)	Training	2.048
Audi FM5	23	(15, 16, 13)	Training	2.048
Toyota Yaris	29	(15, 16, 13)	Training	2.048

Tabelle 6.1: Verwendete Datensätze im Trainingsprozess

Jeder Datensatz, der für das Training verwendet wurde, besteht aus 2048 Samples, wodurch eine Gesamtanzahl an 10240 Trainingssamples zur Verfügung stand. Der Datensatz des FM3 stellt für alle folgenden Benchmarks den Validierungs- und Testdatensatz dar, weil er über die meisten Bauteile verfügt. Der Datensatz besteht aus 1024 Samples, was ungefähr 10 % des Gesamtdatensatzes ausmacht. Dabei orientiert sich die Aufteilung stark an den Versuchen, die in den Arbeiten zu PointNet und 3DmFV vorgestellt wurden. Jeder Benchmark wurde auf dem selben externen Rechner durchgeführt, damit die Testergebnisse hinsichtlich der Performanz verglichen werden können. Die Spezifikationen des Rechners werden im Folgenden kurz aufgezählt: Intel Core i7-3930K 3.2 GHz CPU, 64 GB Arbeitsspeicher, Nvidia GeForce 750 Ti Grafikkarte und eine 512 GB große SSD.

Beide Ansätze wurden in Tensorflow programmiert und mit CUDA ausgeführt. Dadurch werden einzelne Berechnungsschritte auf die Grafikkarte ausgelagert, wodurch sich die allgemeine Performanz des Trainingsprozesses verbessert. Die verwendeten Hyperparameter des Trainings werden in Tabelle 6.2 abgebildet. Da die grundlegende Struktur der beiden Architekturen sehr ähnlich ist, unterscheiden sich die Hyperparameter nur in bestimmten Punkten und sind angelehnt an die empfohlenen Parameter der jeweiligen Architekturen.

Sowohl PointNet als auch 3DmFV benutzen den Mini-Batch Gradient Descent. Dabei werden die Trainingsdaten in kleinen Batches durch das Netz propagiert. Danach erfolgt die Backpropagation, in der die Gewichte des Netzes in Abhängigkeit zum resultierenden Fehler angepasst werden (siehe Kapitel 2). PointNet verwendet 32 Samples pro Batch und der 3DmFV-Ansatz doppelt so viele. Beide Methoden verwenden eine Lernrate von 0.001. Dieser Faktor sagt aus, wie stark der Einfluss der Gradienten auf die neuen Gewichte sein soll.

	PointNet	3D modified Fisher Vectors
Batch-Größe	32	64
Punktwolkengröße	1024	1024
Lernrate	0.001	0.001
Optimierungsverfahren	ADAM	ADAM
Epochen-Anzahl	100	100
Gaußian-Anzahl	-	125

Tabelle 6.2: Vergleich der Hyperparameter im Trainingsprozess zwischen PointNet und 3DmFV

Eine zu hohe Lernrate kann dabei verursachen, dass der Gradient Descent über das Minimum hinausschießt. Die meisten Parameter werden von dem verwendeten Optimierungsverfahren ADAM vorgegeben und nicht weiter angepasst, da es sich um empfohlene Parameter handelt, die bereits in den Arbeiten der Architekturen getestet wurden und sich nicht unterscheiden.
Das Training wird über 100 Epochen durchgeführt, wobei eine Epoche genau einen kompletten Durchlauf durch den Trainingsdatensatz darstellt. Nach jeder Epoche erfolgt eine Validierung mittels des Testdatensatzes des FM3. Dafür werden alle 1024 Samples getestet und die resultierenden Genauigkeitswerte gemessen. Diese Werte dienen als erster Vergleich für die Generalisierung der einzelnen Methoden. Die Gewichte des Netzes werden in dieser Phase nicht aktualisiert, wodurch das Netz die verwendeten Bauteile nicht lernt. Im Falle von 3DmFV wurde für die nachfolgenden Tests eine Gesamtanzahl von 125 Gaußianen festgelegt.

Bewertungsmetriken

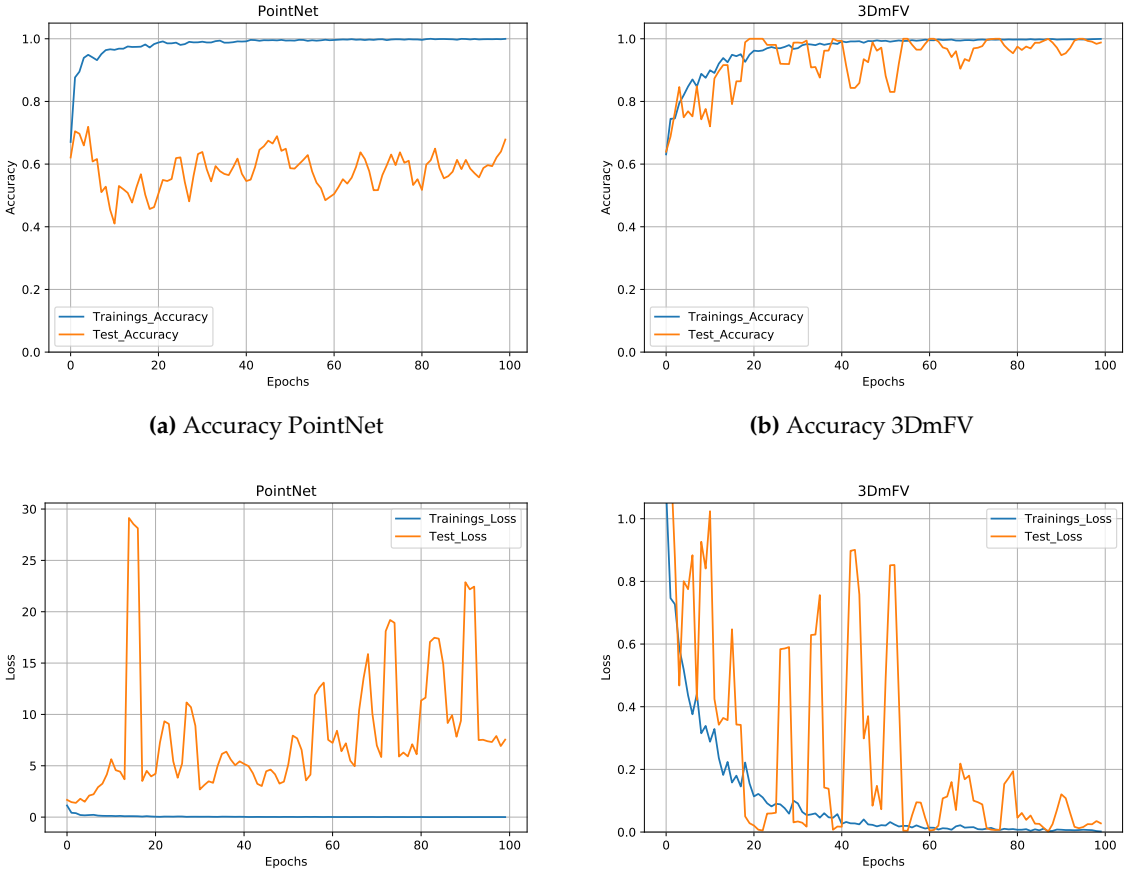
Um beide Ansätze miteinander Vergleichen zu können, werden in den darauffolgenden Versuchen verschiedene Metriken gemessen. Die Accuracy sagt aus, wie viele korrekte Klassifikationen es im Verhältnis zu allen Samples im Datensatz gibt. Mit anderen Worten: Wie viel Prozent der Samples werden richtig klassifiziert? Es werden jeweils die Trainings-Accuracy, die Genauigkeit des Netzes, die während des Trainings auf den Trainingsdaten gemessen wird und die Test-Accuracy, die Genauigkeit des Netzes, die nach jeder Epoche auf den Testdatensatz gemessen wird, verglichen. Außerdem wird der Loss des Trainingsprozesses dargestellt, welcher möglichst gering sein sollte.

$$F_1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{6.1.1}$$

Die Accuracy reicht als einziges Maß nicht aus, da sie keine Aussage über die fehlerhafte Zuordnung von Klassen liefert. Hierfür wird pro Klasse das F1-Maß berechnet. Das F1-Maß ist das harmonische Mittel aus Genauigkeit (engl.: precision) und Trefferquote (engl.: recall) (siehe 6.1.1). Diese Metriken nehmen besonders Fehlklassifikationen mit in ihr Gewicht, wodurch fehleranfällige Klassen besser erkannt werden können. Die Genauigkeit sagt aus, wie viel Prozent der zugeordneten Samples wirklich zu dieser Klasse gehören. Hingegen ist die Aussage hinter der Trefferquote, wie viel Prozent der Samples einer Klasse richtig erkannt wurden. Um ein Maß für das gesamte Modell zu erhalten, werden die F1-Maße aller Klassen addiert und mit der Anzahl der Samples pro Klasse im Datensatz gewichtet. Als letzte Metrik dient die Laufzeit des Trainingsprozesses. Sie ist für die grobe Einordnung der Performanz zuständig und hat auf den Entscheidungsprozess weniger Einfluss.

Unterscheidbarkeit grobgranularer Baugruppen

Der erste Testfall beschäftigt sich mit der Unterscheidbarkeit von grobgranularen Baugruppen. Wie bereits in diesem Kapitel erwähnt, bestehen einige der extrahierten Bauteile aus Außen- und Innenkomponenten, die sich in ihrer groben Form und Struktur nur wenig unterscheiden. Ziel dieses Versuchs war es, herauszufinden, wie gut die Klassifikation abschneidet, wenn die einzelnen Bauteile groben Baugruppen zugeordnet werden. Es wurden nur Bauteile zusammengefasst, die tatsächlich eine ähnliche Form aufwiesen. Die Bauteile *A_Säule_unten_außen* und *A_Säule_unten_innen* mussten aus diesem Grund getrennt betrachtet werden, ebenso die beiden Bauteile des *Radhauses*. Daher wurde das Netz bei diesem Test mit insgesamt 15 verschiedenen Klassen trainiert. Die Ergebnisse und das trainierte Modell wurden jeweils in einer Log-Datei gespeichert und sind für spätere Analysen verwendbar. Aus dem Training resultierten folgende Accuracy-Werte für PointNet und 3DmFV, welche in Abbildung 6.2 zu sehen sind.



(a) Accuracy PointNet

(b) Accuracy 3DmFV

(c) Loss PointNet

(d) Loss 3DmFV

Abbildung 6.2: Vergleich der Metriken zwischen PoinNet und 3DmFV

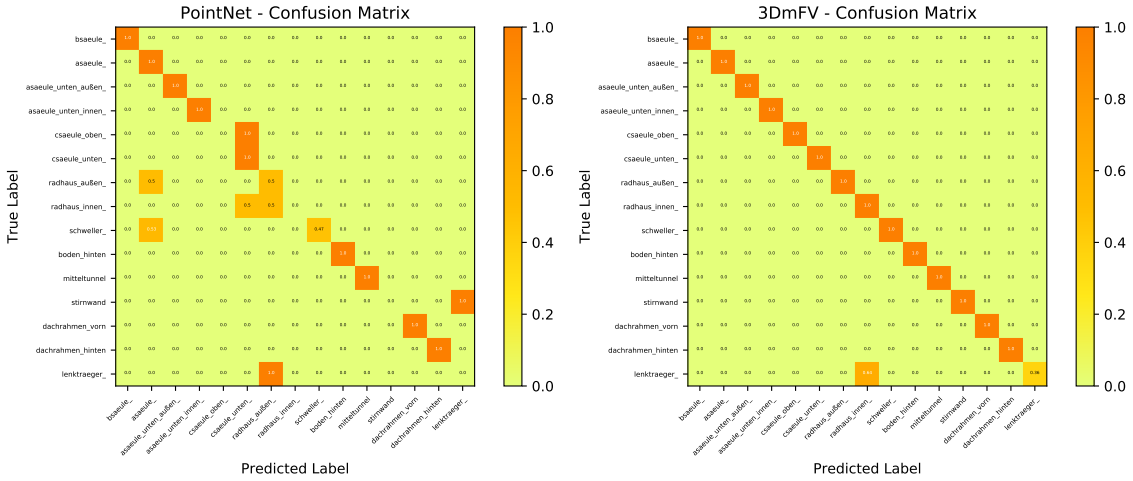
Die linken Diagramme zeigen die Ergebnisse des Trainings von PointNet und auf der rechten Seite sind die Ergebnisse von 3DmFV zu sehen. Die blauen Graphen symbolisieren immer die Metriken, die während des Trainings gemessen wurden. Der orangefarbene Graph zeigt die gemessenen Testergebnisse mit dem Testdatensatz des FM3. Auf den x-Achsen wird jeweils die Trainingsdauer in Epochen dargestellt, welche bei beiden Methoden genau 100 Epochen betrug.

Es ist deutlich zu erkennen, dass beide Ansätze während des Trainings sehr gute Accuracy-Werte von rund 99% erreichen. Daraus lässt sich schließen, dass die Methoden auf bereits gesehenen Daten gut abschneiden. Auch der jeweilige Loss nimmt mit der Dauer des Trainings ab, was bedeutet, dass die aktualisierten Gewichte ständig den Loss verringern. Beim Vergleich der Test-Accuracy lassen sich jedoch deutliche Unterschiede zwischen den beiden Ansätzen erkennen. Wie im Diagramm 6.2a zu sehen, ist die Test-Accuracy von PointNet weit unter der des Trainings. Die gemessenen Werte fluktuieren um die 60% Accuracy und zeigen keinen wirklichen Trend. Am Ende des Trainings erreicht PointNet eine Accuracy von 71.6%. Im Gegensatz dazu lässt sich bei den Ergebnissen von 3DmFV über die Dauer des Trainings ein deutlicher Trend nach oben erkennen. Die kleinen Ausschläge nach unten können durch ein schlechtes Batch an Samples entstanden sein, wodurch mehr Fehlklassifikationen zustande kamen. Am Ende erreichte der Ansatz sehr gute 98.8%.
Ein ähnliches Verhalten lässt sich auch in den Loss-Diagrammen erkennen. Während sich bei 3DmFV der Test-Loss über die Zeit verringert, erhöht sich der Loss bei PointNet sogar. Diese Beobachtung kann verschiedene Ursachen haben. Einerseits kann es bedeuten, dass es sich um eine Überanpassung (engl. overfitting) handelt. Das bedeutet, dass sich das Modell zu stark an die Trainingsdaten angepasst hat und somit neue Daten schlechter klassifiziert. Andererseits ist es möglich, dass bestimmte Klassen mit der Dauer des Trainings schlechter erkannt, aber dennoch richtig klassifiziert werden und sich dadurch der Loss erhöht. Mit anderen Worten: Die Sicherheit des Netzes, bestimmte Klassen richtig zu klassifizieren, sank mit der Trainingsdauer oder durch gewisse Samples. Mit dieser Vermutung lassen sich auch die Ausschläge nach oben des Losses von 3DmFV begründen (siehe Diagramm 6.2d). Die gemessenen Metriken am Ende des

Metriken	PointNet	3DmFV
Accuracy (Training)	99.5 %	**99.7 %**
Accuracy (Test)	71.6 %	**98.8 %**
Loss (Training)	0.0008	0.0018
Loss (Test)	8.4540	0.0206
Laufzeit (h)	~16	~38.5
F_1-Maß (gewichtet)	0.576	**0.959**

Tabelle 6.3: Übersicht der Endresultate des Trainings beider Ansätze

Trainingsprozesses können alle aus der Tabelle 6.3 entnommen werden. Die gemessene Laufzeit von PointNet ist mit rund 16 Stunden im Vergleich zu 3DmFV doppelt so schnell. Zu Begründen ist dieser Unterschied mit der vorherigen Transformation der Punktwolken in die Fisher-Vektor-Darstellung und der Verwendung eines 3D Convolutional Neural Networks. Da für die jeweiligen Anwendungsfälle eine hohe Genauigkeit von größerer Bedeutung ist, kann die längere Dauer eines einmaligen Trainings vernachlässigt werden.
Aus dieser Beobachtung lässt sich schlussfolgern, dass der 3DmFV-Ansatz auf den zugrundeliegenden Daten besser klassifizieren und generalisieren kann als PointNet. Dies kann zum einen an der anderen Darstellungsform als Fisher-Vektor liegen, zum anderen ist es möglich, dass die Unterschiede der spezifischen Merkmale der Bauteile zu gering für PointNet sind und es deshalb zur mehrmaligen Konfusion zwischen den Bauteilklassen kommen kann. Um dieses Verhalten genauer zu Untersuchen, muss ein Blick auf die Klassifikation der einzelnen Klassen geworfen werden. Dafür werden in Abbildung 6.3 die Konfusionsmatrizen beider Ansätze miteinander verglichen, welche mit dem Testdatensatz des FM3 auf dem bereits vollständig trainierten Netz erstellt wurden.



(a) Konfusionsmatrix PointNet

(b) Konfusionsmatrix 3DmFV

Abbildung 6.3: Vergleich der Konfusionsmatrizen bei der Klassifikation von Baugruppen

Die Konfusionsmatrix stellt die wahren Klassen, zu sehen auf der y-Achse, den vorhergesagten Klassen auf der x-Achse gegenüber. An jeder Achse sind alle 15 Klassen abgebildet, auf die das Netz trainiert wurde. Die Werte in den Zellen geben an, wie viel Prozent der Samples mit der entsprechenden Klasse vorhergesagt wurden. In Abbildung 6.3 ist dies zusätzlich durch eine Orangefärbung der Zellen dargestellt. Eine gute Klassifikation des Modells erkennt man, indem nur die Hauptdiagonale der Konfusionsmatrix mit Werten gefüllt ist. Das bedeutet, dass jedes Bauteil zu 100% ihrer richtigen Klasse zugeordnet werden konnte. Wie sich anhand der Konfusionsmatrizen erkennen lässt, ist die Klassifikation des 3DmFV-Ansatzes für fast alle Bauteile korrekt. Lediglich das Bauteil *Lenkträger* wird zu rund 60% als *Radhaus_innen* erkannt.

Klassen	F1-Score PointNet	F1-Score 3DmFV
A_Saeule	0.66	1.0
A_Saeule_unten_außen	1.0	1.0
A_Saeule_unten_innen	1.0	1.0
B_Saeule	1.0	1.0
C_Saeule_oben	0	1.0
C_Saeule_unten	0.44	1.0
Radhaus_außen	0.33	1.0
Radhaus_innen	0	0.76
Schweller	0.64	1.0
Boden_hinten	1.0	1.0
Mitteltunnel	1.0	1.0
Stirnwand	0	1.0
Dachrahmen_vorn	1.0	1.0
Dachrahmen_hinten	1.0	1.0
Lenktraeger	0	0.53

Tabelle 6.4: Übersicht der F1-Maße bei der Klassifikation von Baugruppen

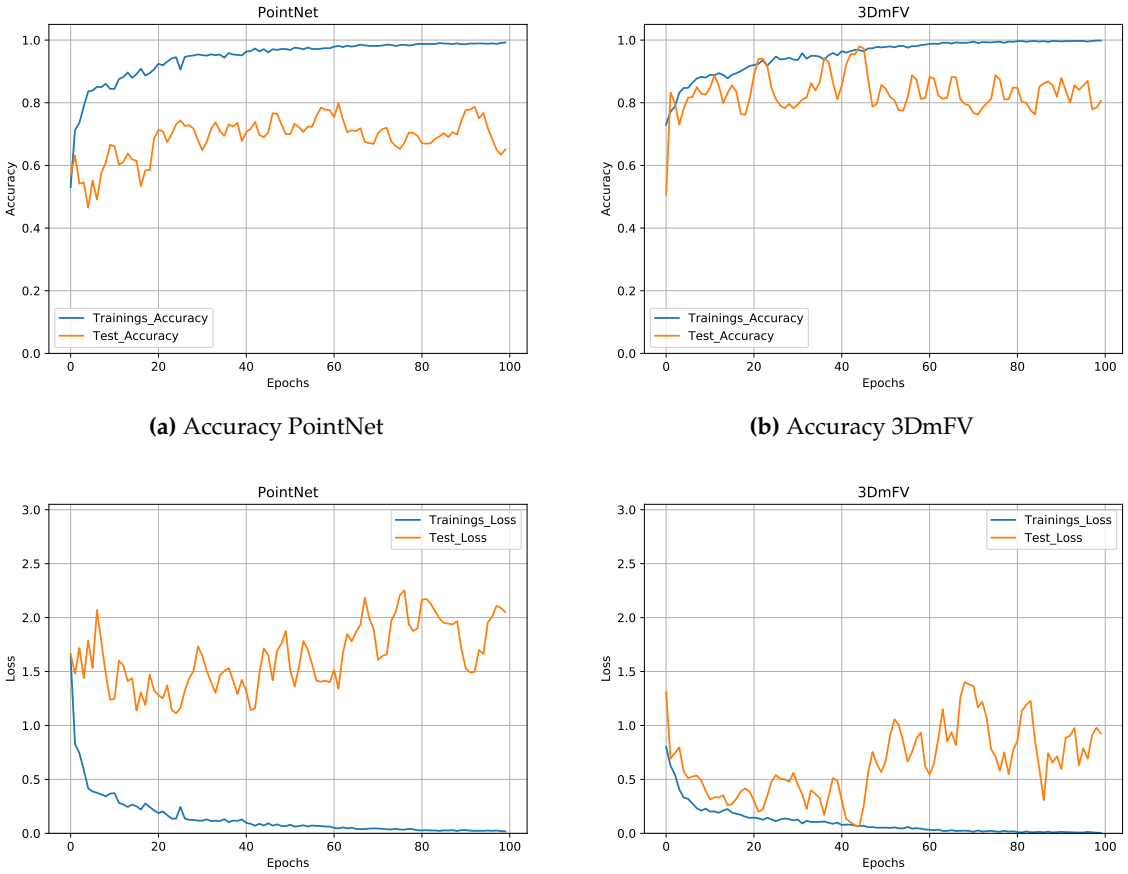
Um die Aussagen der Konfusionsmatrix in einer Metrik zusammenzufassen, wird das F1-Maß der einzelnen Bauteilklassen berechnet. In Tabelle 6.4 ist ein Vergleich der F1-Maße aller Klassen zwischen PointNet und 3DmFV dargestellt. Daraus lässt sich erkennen, dass nicht nur das F1-Maß des *Lenkträgers* niedriger ist, sondern auch das vom *Radhaus_innen*. Die Bauteile, die dieser

Klasse zugehören, wurden zwar alle mit einer hohen Trefferquote klassifiziert, aber durch die Fehlklassifikation des anderen Bauteils sank die Genauigkeit dieser Klasse.
Die Ergebnisse von PointNet sind deutlich schlechter im Vergleich zu 3DmFV, wie man der Konfusionsmatrix entnehmen kann. Es werden lediglich 7 von 15 Klassen sicher erkannt, wobei vier Klassen komplett fehlklassifiziert wurden (vgl. Tabelle 6.4). Aus der Tatsache, dass einige Bauteile, wie die *C_Säule_oben*, vollständig einer falschen Klasse zugeordnet wurden, lässt sich schließen, dass die grundlegenden Bauteile für das Netz zu identisch sind und es deshalb zur Verwechslung kam. Die Entscheidung, dass *Radhaus* nicht als Bauteilgruppe zusammenzufassen, lässt sich damit begründen, dass, wie oben beschrieben, die einzelnen Bauteile keine deckungsgleiche Form aufweisen. Anhand der Klassifikation von PointNet kann jedoch vermutet werden, dass sich die Bauteile ähneln. Auffällig ist, dass die Bauteile, bei denen der 3DmFV-Ansatz Probleme aufzeigte, auch von PointNet nicht korrekt zugeordnet werden konnten.

Umfassend betrachtet kann gesagt werden, dass die Ergebnisse des ersten Versuches für 3DmFV sehr gut ausfielen, da nur wenige falsche Zuordnungen auftraten. PointNet hingegen zeigt deutliche Schwächen bei der Klassifikation bestimmter Bauteilgruppen. Das lässt vermuten, dass die 3D-Repräsentation der Bauteile zu wenig unterschiedliche Merkmale aufweisen oder das Netz zu stark an die Trainingsdaten angepasst ist. Es kann jedoch sicher gesagt werden, dass sich der 3DmFV-Ansatz auf der zugrundeliegenden Domäne besser schlägt. Im anschließenden Versuch wurde geschaut, wie gut die Netze bei der Unterscheidung zwischen linken und rechten Bauteilen abschneiden. Es wird davon ausgegangen, dass dieser Test schlechtere Ergebnisse erzeugt, da sich die geometrische Repräsentation der Bauteile stark ähnelt.

Unterscheidbarkeit linker und rechter Bauteile

Für die meisten Anwendungsfälle (siehe Kapitel 1) in der Fahrzeugindustrie im Bereich Bauteilerkennung ist es wichtig, dass Systeme auch zwischen Bauteilen mit einem Pendant auf der gegenüberliegenden Seite differenzieren können. Genauer gesagt: Die Unterscheidung zwischen der linken *B-Säule* und der rechten *B-Säule* ist von Bedeutung. Um diesen Sachverhalt zu untersuchen, wurden beide Ansätze mit neu erstellten Datensätzen trainiert. Die Generierung der Datensätze erfolgte dabei ohne jene Bauteile, die kein Gegenstück auf der anderen Seite besitzen. So fielen unter anderem die Bauteile *Mitteltunnel* und *Stirnwand* weg, damit diese gut klassifizierbaren Bauteile nicht die Ergebnisse verfälschen. Zusätzlich wurde komplett auf die Differenzierung zwischen Innen- und Außenteilen verzichtet.
Da aus den vorherigen Ergebnissen hervorging, dass es beim Bauteil *Radhaus* durchaus zu einer Konfusion zwischen den Innen- und Außenkomponenten kommen kann, wurde die Klasse jetzt zusammengefasst, sodass lediglich zwischen *Radhaus_links* und *Radhaus_rechts* unterschieden wurde. Somit entstand eine neue Menge an insgesamt 16 verschiedenen Bauteilklassen. Die Ansätze wurden mit den gleichen Hyperparametern wie im vorherigen Versuch trainiert, um die Ergebnisse vergleichbar zu machen. Der Datensatz des FM3 stellt weiterhin den Testdatensatz mit 1024 Samples dar. Die Endresultate des Trainings beider Methoden sind in Abbildung 6.4 zu sehen.
Sowohl PoinNet als auch 3DmFV erzielen im Training wieder rund 99%, was bedeutet, dass beide Ansätze sehr gut auf den neuen Trainingsdaten abschneiden. Die Accuracy des Testdatensatzes liegt jedoch bei beiden Ansätzen deutlich darunter. 3DmFV erreicht am Ende des Trainings knapp die 80%.



(a) Accuracy PointNet

(b) Accuracy 3DmFV

(c) Loss PointNet

(d) Loss 3DmFV

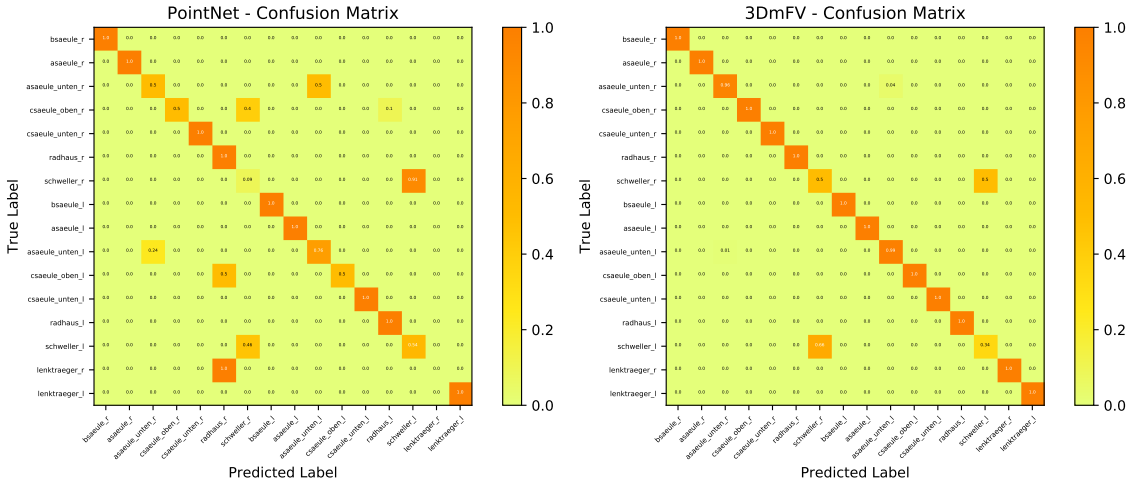
Abbildung 6.4: Vergleich der Metriken zwischen PoinNet und 3DmFV

Anhand der Diagramme (siehe Abbildung 6.4b) lässt sich kein Trend abzeichnen. Besonders auffällig ist, dass der Loss kurz nach der 40. Epoche anfängt, wieder zu steigen. Das kann zum einen bedeuten, dass sich das Netz überangepasst hat oder das gewisse Klassen, während des Trainings kontinuierlich schlechter klassifiziert worden sind. PointNet erzielt bei diesem Versuch ähnlich schlechte Ergebnisse wie im Versuch zuvor. Die Test-Accuracy erreicht am Ende des Trainings nur 68.8% und liegt somit rund 10% unter den Ergebnissen von 3DmFV. Überraschend ist jedoch, dass das F1-Maß von PointNet sich im Vergleich zur vorherigen Untersuchung verbessert hat (siehe Tabelle 6.5).

Metriken	PointNet	3DmFV
Accuracy (Training)	99.6 %	**99.8 %**
Accuracy (Test)	68.8 %	**81.0 %**
Loss (Training)	0.0068	0.0041
Loss (Test)	2.06	0.84
Laufzeit (h)	~18.5	~39
F_1-Maß (gewichtet)	0.70	**0.88**

Tabelle 6.5: Übersicht der Endresultate des Trainings beider Ansätze

Die Laufzeit blieb bei beiden Ansätzen so gut wie unverändert. Kleine Abweichungen können durch die unterschiedliche Klassenanzahl oder Schwankungen der Computerleistung entstanden sein.



(a) Konfusionsmatrix PointNet

(b) Konfusionsmatrix 3DmFV

Abbildung 6.5: Vergleich der Konfusionsmatrizen bei der Unterscheidung von linken und rechten Bauteilen

Wie im vorherigen Versuch werden auch hier die Konfusionsmatrizen erstellt, um einen Blick auf die Testergebnisse der einzelnen Klassen zu werfen. In der Konfusionsmatrix von 3DmFV (siehe Abbildung 6.5b) ist deutlich das erwartete Verhalten zu sehen.
Es wurden insgesamt nur zwei Bauteile nicht komplett ihren richtigen Klassen zugeordnet, da sie, wie vermutet, mit ihrem jeweiligen Pendant verwechselt wurden. Besonders das Bauteil *Schweller_links* konnte keine guten Werte erzielen und Schnitt mit einen F1-Maß von 0.37 eher schlecht ab (siehe Tabelle 6.6). Grund dafür ist sehr wahrscheinlich die fast deckungsgleiche 3D-Darstellung des *Schwellers*. Dabei handelt es sich um das längliche Bauteil unterhalb der Türen. Die spezifischen Merkmale beider Geometrien sollten nahezu identisch sein. Demnach liegt die Behauptung nahe, dass 3DmFV bei Bauteilen, deren geometrische Repräsentation fast identisch ist, keine zuverlässige Zuordnung gewährleisten kann.

Klassen	F1-Score PointNet	F1-Score 3DmFV
A_Saeule_rechts	1.0	1.0
A_Saeule_unten_rechts	0.58	0.97
B_Saeule_rechts	1.0	1.0
C_Saeule_oben_rechts	0.66	1.0
C_Saeule_unten_rechts	1.0	1.0
Radhaus_rechts	0.66	1.0
Schweller_rechts	0.1	0.46
Lenktraeger_rechts	0	1.0
A_Saeule_links	1.0	1.0
A_Saeule_unten_links	0.68	0.97
B_Saeule_links	1.0	1.0
C_Saeule_oben_links	0.66	1.0
C_Saeule_unten_links	1.0	1.0
Radhaus_links	0.95	1.0
Schweller_links	0.44	0.37
Lenktraeger_links	1.0	1.0

Tabelle 6.6: Übersicht der F1-Maße bei der Unterscheidung von linken und rechten Bauteilen

Die Verwechslung des Bauteils *Schweller* ist in der Konfusionsmatrix von PointNet auch zu erkennen. Jedoch neigt PointNet dazu, auch weitere Klassen miteinander zu vertauschen. Es konnten nur 7 von 16 Klassen exakt zugeordnet werden. Die Verwechslung der einzelnen Klassen fällt dabei sehr unterschiedlichen aus. Das könnte daran liegen, dass sich gewisse 3D-Modelle in ihren, von PointNet extrahierten, spezifischen Merkmalen zu sehr ähneln. Dies lässt vermuten, dass die Repräsentationsform für diese Art von Daten und Unterscheidung schlichtweg unzureichend ist.

Im Gegensatz zu den Vermutungen konnte ein Großteil der Bauteile von 3DmFV richtig klassifiziert werden. Lediglich Bauteile, die sich sehr stark in ihrer Form und Struktur ähneln, können nicht sicher zugeordnet werden. Die Ergebnisse von 3DmFV sind weiterhin vielversprechend und zeigen, dass der Ansatz auf gewissen Daten Potential besitzt. PointNets Ergebnisse liegen weiterhin unter denen von 3DmFV, woraus sich ableiten lässt, dass dieser Ansatz weniger geeignet für die Klassifizierung von Fahrzeugbauteilen ist. Der nächste Abschnitt widmet sich der Unterscheidbarkeit von Innen- und Außenbauteilen.

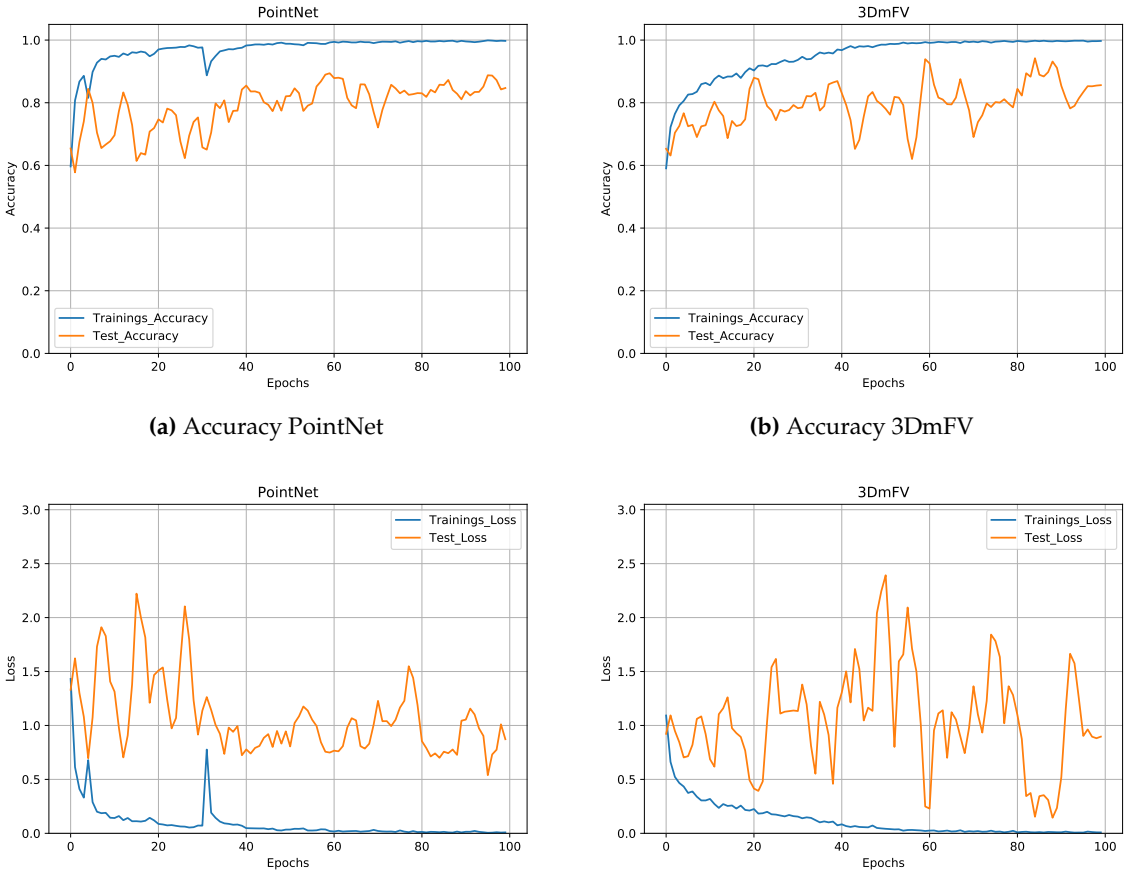
Unterscheidbarkeit von Innen- und Außenbauteilen

Wie zuvor erwähnt, existieren einige Bauteile im Fahrzeug, die jeweils aus einem Innen- und einem Außenteil bestehen (siehe Abbildung 6.1). In der Automobilindustrie werden diese beiden Bleche miteinander durch Schweißpunkte verbunden, wodurch das neue Bauteil entsteht. Aus diesem Grund ist es wichtig, dass beispielsweise Anwendungen für die Vorhersage von Schweißpunkten die gegebenen Innen- und Außenbleche voneinander unterscheiden können. Die Schwierigkeit hierbei ist, dass die Bauteile als Punktwolke fast identisch und deckungsgleich erscheinen, wodurch die Klassifikation der neuronalen Netze beeinflusst werden kann.
Diese Untersuchung soll zeigen, wie gut die vorgestellten Methoden bei der Unterscheidung solcher Bauteile abschneiden. Es wurden dabei nur Bauteile verwendet, die sowohl ein Innenblech als auch ein Außenblech besitzen. Die erstellten Datensätze bestehen aus 13 verschiedenen Klassen und wurden, wie in den vorherigen Versuchen, händisch gelabelt. Auch die Trainingsparameter blieben unverändert. In Abbildung 6.6 ist die Gegenüberstellung der Trainingsergebnisse beider Ansätze zu sehen.

Metriken	PointNet	3DmFV
Accuracy (Training)	99.8%	99.7%
Accuracy (Test)	83.2%	**85.6%**
Loss (Training)	0.0031	0.0059
Loss (Test)	0.73	0.83
Laufzeit (h)	~18	~38
F_1-Maß (gewichtet)	0.88	0.89

Tabelle 6.7: Übersicht der Endresultate des Trainings beider Ansätze

Im Gegensatz zu den Erwartungen fielen die Ergebnisse von PointNet besser aus und sind mit denen des 3DmFV-Ansatzes vergleichbar. Im Graph der Test-Accuracy in Abbildung 6.6a ist ein leichter Trend erkennbar. Am Ende erreichte PointNet eine Test-Accuracy von 83.2% und somit das beste Ergebnis der drei Untersuchungen. Dennoch ist der Loss weiterhin nicht wirklich gering, wodurch die Sicherheit des Netzes angezweifelt werden kann. Das bedeutet, dass zwar viele Samples richtig klassifiziert worden sind, die Zuordnung zu der richtigen Klasse jedoch noch unsicher ist. In diesem Versuch unterscheiden sich die beiden Ansätze nur geringfügig voneinander.



(a) Accuracy PointNet

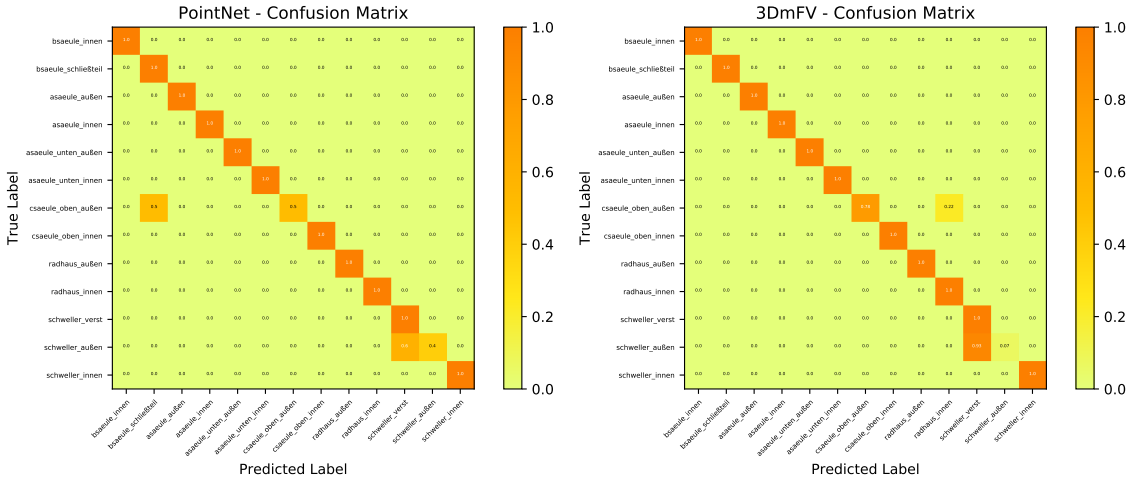
(b) Accuracy 3DmFV

(c) Loss PointNet

(d) Loss 3DmFV

Abbildung 6.6: Vergleich der Metriken zwischen PoinNet und 3DmFV

Sowohl die Verläufe der Accuracy-Graphen als auch die des Losses weisen einige Gemeinsamkeiten auf. 3DmFV erreichte mit 85,6% am Ende des Trainings eine knapp bessere Accuracy. Anhand der Tatsache, dass beide neuronale Netze ähnlich abschneiden, lässt sich schließen, dass entstandenes Fehlverhalten auf die verwendeten Trainingsdaten zurückzuführen ist. Zusätzlich kann die Annahme, dass deckungsgleiche Bauteile zwangsläufig zu Fehlklassifikationen führen, angezweifelt werden.
Beim Blick auf die Konfusionsmatrizen in Abbildung 6.7 ist deutlich zu sehen, dass bei beiden Ansätzen fast alle Bauteile richtig klassifiziert wurden. Auffällig ist dabei, dass von beiden Methoden die exakt gleichen Bauteile zu Fehlklassifikationen führten. Jeweils der *Schweller_außen* und die *C_Säule_außen* konnten nicht immer richtig zugeordnet werden. Besonders der *Schweller_außen* wurde oft dem sehr ähnlichen *Verstärkungsblech* zugeordnet, wodurch das F1-Maß dieser Klasse sank (siehe Tabelle 6.8). Beide Ansätze hatten Schwierigkeiten, die gleichen Bauteile exakt zuzuordnen. Daraus kann geschlussfolgert werden, dass die Ergebnisse abhängig von der 3D-Repräsentation der Bauteile sind. Der *Schweller* ist demnach ein schwer zu klassifizierendes Bauteil, wenn es um die Unterscheidung zwischen Innen- und Außenblech geht.
Es kann demnach gesagt werden, dass die Methoden doch für eine Unterscheidung von Innen- und Außenbauteilen geeignet sind, vorausgesetzt die jeweiligen Bauteile haben genug unterscheidbare spezifische Merkmale. Jedoch muss klargestellt werden, dass die Mehrheit der Bauteile im letzten Versuch immer gute Ergebnisse erzielte. Diese Aussage lässt sich demnach nicht auf alle Bauteile eines Fahrzeugs übertragen.



(a) Konfusionsmatrix PointNet

(b) Konfusionsmatrix 3DmFV

Abbildung 6.7: Vergleich der Konfusionsmatrizen bei der Unterscheidung von Innen- und Außenbauteilen

Klassen	F1-Score PointNet	F1-Score 3DmFV
A_Saeule_außen	1.0	1.0
A_Saeule_innen	1.0	1.0
A_Saeule_unten_außen	1.0	1.0
A_Saeule_unten_innen	1.0	1.0
B_Saeule_innen	1.0	1.0
B_Saeule_schließteil	0.66	1.0
C_Saeule_oben_außen	0.66	0.88
C_Saeule_oben_innen	1.0	1.0
Radhaus_außen	1.0	1.0
Radhaus_innen	1.0	0.82
Schweller_Verstaerk	0.77	0.68
Schweller_außen	0.57	0.13
Schweller_innen	1.0	1.0

Tabelle 6.8: Übersicht der F1-Maße bei der Unterscheidung von Innen- und Außenbauteilen

Zusammenfassend kann festgehalten werden, dass die drei Untersuchungen gute Ergebnisse lieferten und die vorgestellten Methoden geeignet für die Klassifikation von FEM-Daten erscheinen. Besonders 3DmFV konnte anhand der Trainingsdaten gut generalisieren und erreichte immer stabile Ergebnisse. Wie gezeigt wurde, können auch schwierigere Klassifikationsprobleme mit einer ausreichend guten Genauigkeit gelöst werden.
Da die Ergebnisse von 3DmFV in allen Versuchen besser ausfielen, als die von PointNet und bei der Klassifikation von Baugruppen die höchste Genauigkeit erreicht wurde, wird sich im weiteren Verlauf der Arbeit nur noch auf diesen Ansatz konzentriert. Dabei liegt der Fokus besonders auf der Untersuchung der neuen Darstellungsform von 3D-Geometrien. Genauer gesagt wird im nächsten Versuch der Einfluss der Gaußian-Anzahl im Raum untersucht.

Hyperparameteranpassung

Das besondere des 3DmFV-Ansatzes ist die neue Repräsentationsform der Punktwolken als Matrix von Fisher-Vektoren. Der Raum wird in Richtung jeder Achse im Intervall [-1, 1] mit einem $m \times m \times m$ Gitter aus Gaußianen besetzt. Ein Gaußian symbolisiert einen Fisher-Vektor. Die Anzahl an Gaußianen und damit die Größe der Matrix ist variabel anpassbar. Der Parameter m kann dabei Werte von 3 bis 9 annehmen. Die Anzahl der Gaußiane bestimmt in diesem Fall, wie fein der Raum abgetastet werden soll. In Abbildung 6.8 ist eine exemplarische Darstellung des abgetasteten Raums zu sehen. Es handelt sich dabei um ein $5 \times 5 \times 5$-Gitter, wobei jede blaue Kugel genau einen Gaußian darstellt.

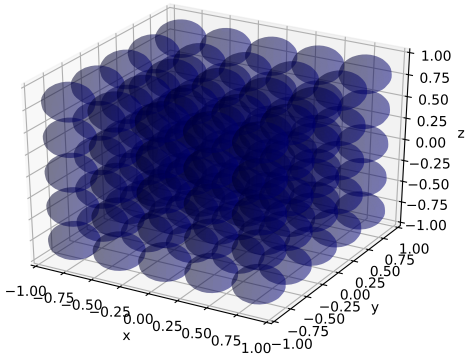
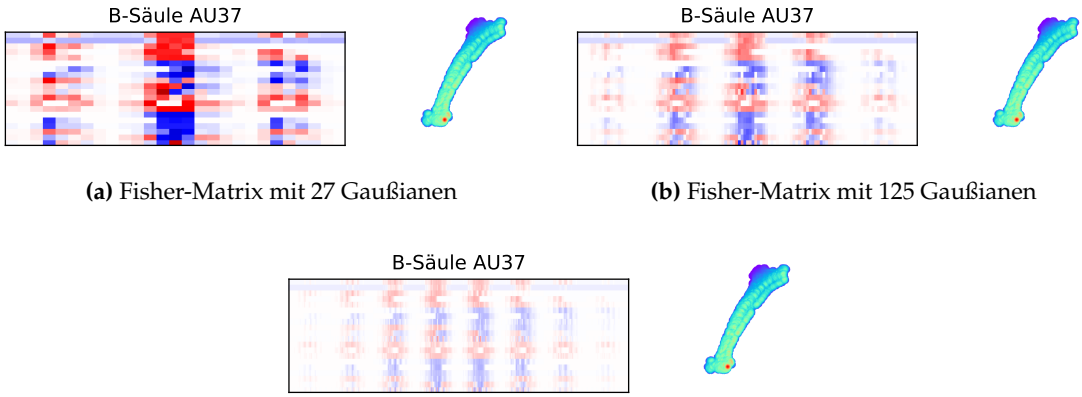


Abbildung 6.8: 5x5x5-Gitter

Mit der steigenden Anzahl an Fisher-Vektoren (Gaußianen) werden die 3D-Modelle im Raum genauer beschrieben. Das heißt, die Granularität verringert sich. Somit erhöht sich schon vorweg die Anzahl an charakteristischen Merkmalen des 3D-Modells. In diesem Test werden drei verschiedene Größen des Gaußian-Gitters evaluiert und miteinander verglichen. Ziel dabei ist es, herauszufinden, wie viel Einfluss die Anzahl an Fisher-Vektoren auf die Accuracy des Netzes hat. Genauer gesagt, kann durch eine Erhöhung dieses Parameters auch eine Erhöhung der Test-Genauigkeit erreicht werden. Zusätzlich wird die Laufzeit der verschiedenen Modelle gegenübergestellt, um auch die Auswirkungen auf die Performanz der Netze zu untersuchen.



(a) Fisher-Matrix mit 27 Gaußianen

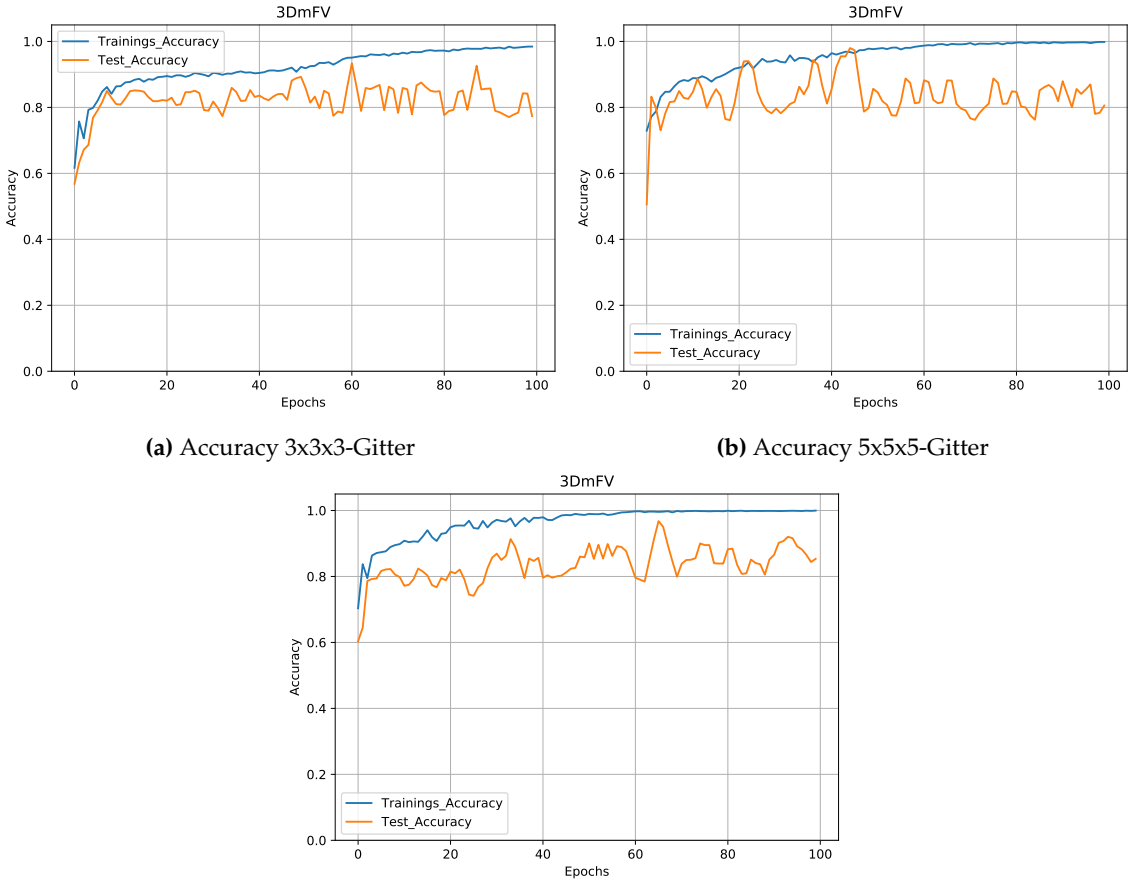
(b) Fisher-Matrix mit 125 Gaußianen

(c) Fisher-Matrix mit 512 Gaußianen

Abbildung 6.9: Vergleich der Fisher-Matrizen der B-Säule des FM2 bei unterschiedlicher Gaußian-Anzahl

In Abbildung 6.9 ist der Vergleich der unterschiedlichen Repräsentationen der Fisher-Matrizen zu sehen, die durch die variierende Gaußian-Anzahl entstehen. Dabei wurde beispielhaft die Fisher-Vektor Darstellung der B-Säule des FM2 als 3D-Modell verwendet, welches jeweils neben den Matrizen abgebildet ist.

Eine Spalte in der Matrix stellt einen Fisher-Vektor dar. Die Zellen bilden die Gradienten in Abhängigkeit zu dem entsprechenden Gaußian-Parameter. Rot gefärbte Zellen symbolisieren einen positiven Wert und die blauen Zellen einen negativen Wert. Alle Nullwerte werden mit weiß kodiert. Es ist zu erkennen, dass die Struktur der Matrix bei Erhöhung der Gaußian-Anzahl gleich bleibt, sich jedoch die Anzahl an möglichen Merkmalen stark erhöht. Es gilt nun zu Untersuchen, welchen Einfluss diese genauere Darstellung auf die Ergebnisse hat. Es wurden drei verschiedene Modelle trainiert, jeweils mit $m = [3, 5, 8]$. In diesem Fall fungierte die Unterscheidung von linken und rechten Bauteilen als Klassifikationsproblem, da die Ergebnisse aus diesem Versuch mehr Raum für eventuelle Verbesserungen bieten.



(a) Accuracy 3x3x3-Gitter

(b) Accuracy 5x5x5-Gitter

(c) Accuracy 8x8x8-Gitter

Abbildung 6.10: Vergleich der Accuracy-Werte des Trainings mit unterschiedlicher Anzahl an Gaußianen

In Tabelle 6.9 ist ein zusammenfassender Überblick über die entstandenen Ergebnisse zu sehen. Die Testlaufzeit beschreibt die Zeit, die das bereits trainierte Netz für die Vorhersage von 1024 Samples benötigte. Dabei wurde wieder der Testdatensatz des FM3 verwendet. Aus den Ergebnissen lässt sich ableiten, dass eine Erhöhung der Gaußian-Anzahl eine leichte Verbesserung der Genauigkeit am Ende des Trainings mit sich bringt.

Metriken	3DmFV - Gitter 3x3x3	5x5x5	8x8x8
Accuracy (Training)	98.5%	99.8%	99.9%
Accuracy (Test)	78.0%	81.0%	**85.0%**
Laufzeit (Training)	17 h	38 h	116 h
Laufzeit (Test)	43s	116s	373s

Tabelle 6.9: Vergleich der Endresultate bei unterschiedlicher Gaußian-Anzahl

An den Verläufen der Graphen aller Modelle in Abbildung 6.10 lässt sich jedoch keine signifikante Verbesserung erkennen. Lediglich die Stabilität der Trainings-Accuracy nimmt zu.
Im Kontrast dazu ist die Steigerung der Laufzeit deutlich zu erkennen. Das Netz benötigt beim Training rund 60% mehr Zeit. Die Ergebnisse der Testlaufzeit liegen im niedrigen Minuten-Bereich und auch hier lässt sich eine Steigerung bei Erhöhung der Gaußian-Anzahl feststellen. Die Klassifizierung eines einzigen Bauteils liegt bei allen Ansätzen jedoch im Millisekunden-Bereich. Im Kontext dieser Arbeit und bezogen auf die Anwendungsfälle der Fahrzeugindustrie kann eine längere Trainingszeit mit besseren Ergebnissen gerechtfertigt werden. Die Laufzeit im Test sollte jedoch möglichst gering sein, da es sich bei den meisten Anwendungsfällen um Echtzeitanwendungen handelt. Aus diesem Grund ist eine mittlere Anzahl an Gaußianen im Raum zu empfehlen, um einen Kompromiss zwischen Genauigkeit und Laufzeit zu erreichen.

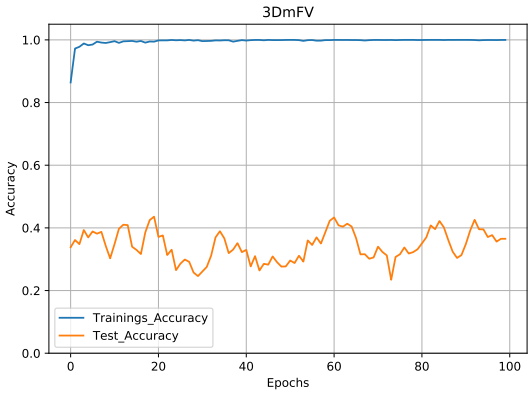
Verwendung unterschiedlicher Testdatensätze

Aus den gewonnenen Ergebnissen lässt sich schließen, dass der 3DmFV-Ansatz für die Erkennung bestimmter Fahrzeugbauteile grundlegend geeignet ist. Die Modelle wurden, wie oben erwähnt, auf genau fünf von sechs Datensätzen trainiert. Als Testdatensatz diente bei jedem Versuch der des FM3, da dieser Samples aller extrahierten Bauteile beinhaltet. Um sicherzustellen, dass die Resultate nicht nur abhängig von der Wahl dieses Datensatzes sind, wurden weitere Modelle mit einer anderen Aufteilung der Gesamtdaten trainiert.
Für diesen Test wurde die Klassifikation von Bauteilgruppen verwendet, da die Ergebnisse von 3DmFV bei dieser Klassenzuordnung am besten ausfielen. Die Modelle sind demnach mit 15 verschiedenen Klassen trainiert worden. Für das erste Modell diente der Toyota Yaris als Testdatensatz, die fünf Audi-Modelle (FM1, FM2, FM3, FM4, FM5) fungierten als Trainingsdatensätze.
In Abbildung 6.11c sind die Ergebnisse des Trainings dargestellt. Es ist klar zu erkennen, dass dieser Testfall keine guten Resultate lieferte. Die Test-Accuracy betrug lediglich knappe 38%, wohingegen die Accuracy des Trainings bei typischen 99% lag. Durch einen Blick auf die Konfusionsmatrix in Abbildung 6.11c wird deutlich, dass die Mehrheit der Bauteile einer kompletten Fehlklassifikation unterlag. Auch Bauteile, die vorher stabile Ergebnisse erzeugten, wie beispielsweise die *A_Säule* und *B_Säule*, konnten vom Modell nicht exakt zugeordnet werden.

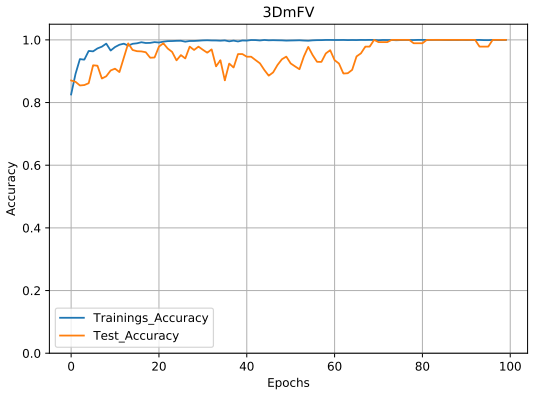
Dieses Verhalten kann mehrere Gründe besitzen. Auf der einen Seite kann ein Training, das ausschließlich aus Daten von Audi-Modellen besteht, bedeuten, dass das Modell sich zu sehr an diese Daten angepasst hat und neue Modelle, die größere Unterschiede aufweisen, dementsprechend fehlklassifiziert. Zusätzlich würde es bedeuten, dass die 3D-Geometrien der Bauteile der Audi-Daten eine stärkere Ähnlichkeit zueinander besitzen, was grundlegend logisch erscheint. Auf der anderen Seite kann auch die Repräsentation der Daten als Fisher-Matrix Ursache für die fehlerhafte Klassifikation sein. Diese Darstellung extrahiert bereits vor dem Training spezifische

Merkmale der Bauteile, welche innerhalb der Audi-Modelle mehr Gemeinsamkeiten aufweisen können.

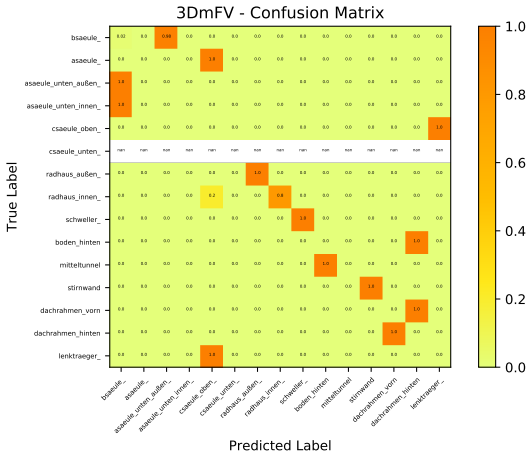
Um auszuschließen, dass die erreichten Testergebnisse nur mit dem Testdatensatz FM3 erreicht werden können, wurde ein weiteres Modell trainiert. Diesmal diente der Audi FM2 zum validieren des trainierten Modells. Die restlichen Datensätze wurden für das Training verwendet. Die Ergebnisse sind in Abbildung 6.11b dargestellt und fallen ähnlich wie die des FM3-Datensatzes aus. Die Zuordnung der Bauteile konnte sogar 100% erreichen, wie in der Konfusionsmatrix 6.11d zu sehen ist. Daraus lässt sich schließen, dass die Klassifikation mit den Modellen von Audi gute Ergebnisse liefert. Der 3DmFV-Ansatz kann demnach gut die 3D-Modelle der Bauteile von Audi generalisieren. Um Aussagen über die Generalisierung mehrerer Fahrzeugmodelle treffen zu können, müsste mehr Varianz in dem Datensatz vorhanden sein.



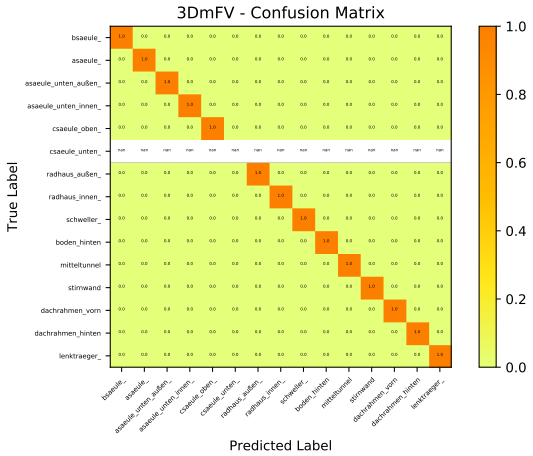
(a) Accuracy mit Yaris-Datensatz



(b) Accuracy mit Audi FM2-Datensatz



(c) Konfusionsmatrix mit Yaris-Datensatz



(d) Konfusionsmatrix mit Audi FM2-Datensatz

Abbildung 6.11: Vergleich der Ergebnisse von 3DmFV mit anderen Testdatensätzen

Den aus diesem Abschnitt resultierenden Sachverhalt gilt es im Folgenden zu untersuchen, um mögliche Gründe für die Fehlklassifikation des Toyota Yaris zu liefern. Dafür wird im nächsten Abschnitt ein genauerer Blick auf die Repräsentation der Bauteile als Fisher-Matrix geworfen, um eventuelle Parallelen zu den erhaltenen Ergebnissen zu analysieren. Zusätzlich werden die gewonnenen Ergebnisse der vorherigen Versuche noch einmal grob zusammengefasst, diskutiert und Rückschlüsse auf die Realisierbarkeit der Anwendungsfälle gezogen. Die daraus gewonnenen Aussagen stellen einen Kernpunkt dieser Arbeit dar.

6.2 DISKUSSION

Die gewonnenen Ergebnisse der Versuche im vorherigen Abschnitt zeigen, dass eine Klassifikation von Bauteilen mit aktuellen Deep-Learning-Methoden möglich ist. Die besten Ergebnisse erzielte 3DmFV bei der Klassifikation von Bauteilgruppen. Mit sehr guten 98% konnten fast alle Bauteile des Testdatensatzes richtig erkannt werden. Für Anwendungen im CAE-Bereich ist die Stabilität der Klassifikation essentiell. Durch die resultierenden Genauigkeitswerte ist es demnach möglich, einige Anwendungsfälle prototypisch zu realisieren. Beispielsweise ist eine Software erdenklich, welche die Bauteile in eine hierarchisch strukturierte Stückliste einordnet. So werden alle Bauteile automatisch der erkannten Baugruppe zugeordnet. Fehlklassifikationen können danach vom Ingenieur überprüft und korrigiert werden. Die Stücklisten werden bisher in mehreren Prozessschritten verwendet und müssten ansonsten vorerst mühselig von Hand erstellt werden.

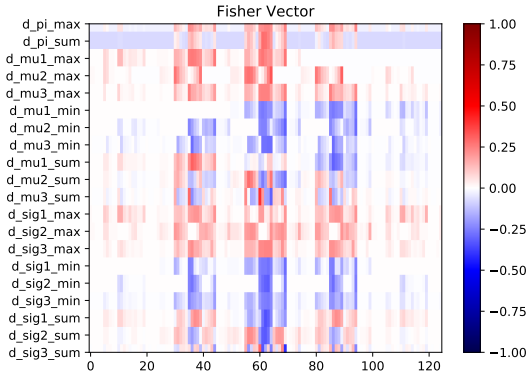
Wie die Untersuchungen ergaben, erzielten die anderen Klassifikationsprobleme leicht schwächere Ergebnisse. Mit rund 80% ist die Klassifikation zwar nicht bei allen Bauteilsamples exakt und sicher, ein Großteil der Bauteile kann aber dennoch erkannt werden. Für die Anwendungen in der Automobilindustrie ist es das Ziel, möglichst gute Ergebnisse nahe der 100% Genauigkeit zu erreichen. Solche Ergebnisse können von typischen Anwendungen des maschinellen Lernens jedoch kaum gewährleistet werden. Dennoch ist es möglich, Empfehlungssysteme zu entwickeln, die dem Anwender, basierend auf der Sicherheit der Klassifikation, Vorschläge unterbreiten. Damit können beispielsweise ein einheitliches Labelingsystem oder ein Empfehlungssystem für Bauteilparameter entwickelt werden. Bei unzureichender Sicherheit des Netzes kann der Ingenieur selbst entscheiden, wie hilfreich dieser Vorschlag war.
Es muss klargestellt werden, dass die hier getroffenen Aussagen nur für die extrahierte Teilmenge an Fahrzeugbauteilen gelten. Zusätzlich wurde gezeigt, dass es sowohl einfach als auch schwer klassifizierbare Bauteile gibt. Die Ansätze hatten beispielsweise das Problem, den länglichen *Schweller* von seinem Pendant zu unterscheiden. Dahingegen erzielten *A-Säule* und *B-Säule* oft eine exakte Zuordnung. Es liegt deshalb die Vermutung nahe, dass eine Vielzahl gleichartiger Bauteile starken Einfluss auf die Qualität der Klassifikation haben kann. Mit einer steigenden Anzahl an Bauteilen können die Ergebnisse deutlich schlechter ausfallen.

Es kann jedoch behauptet werden, dass 3DmFV als Ansatz für die Realisierung der Anwendungsfälle weiter untersucht werden kann, da die Ergebnisse in dieser Domäne sehr vielversprechend aussehen. Dazu muss man klarstellen, dass für die Repräsentation der Fisher-Vektoren die passende Größe analysiert werden muss. Verglichen mit den Resultaten der Untersuchung scheint eine mittlere Anzahl an Gaußianen am geeignetsten, da so ein Kompromiss zwischen Laufzeit und Genauigkeit geschlossen wird.
Die Versuche mit anderen Testdatensätzen werfen einige Fragen auf: Einerseits lässt sich aus den schlechten Ergebnissen mit den Toyota-Yaris-Datensatz schließen, dass die rohen 3D-Modelle der Audis zu wenig Varianz aufweisen, wodurch das Netz nicht gut generalisieren kann. Somit ist es möglich, dass der getestete Ansatz nur gut auf einer bestimmten Domäne von Fahrzeugmodellen abschneidet. Bezogen auf die Anwendungsfälle wäre es denkbar, dass die Modelle auch nur auf herstellerspezifischen Daten trainiert werden, wodurch die Anwendung nicht eingeschränkt sein sollte. Andererseits ist es möglich, dass die Repräsentation von 3DmFV sich bei einigen Bauteilen stark ähnelt. Um dies herauszufinden, wird im Folgenden ein Vergleich der Matrizen von Fisher-Vektoren zwischen den Modellen durchgeführt.

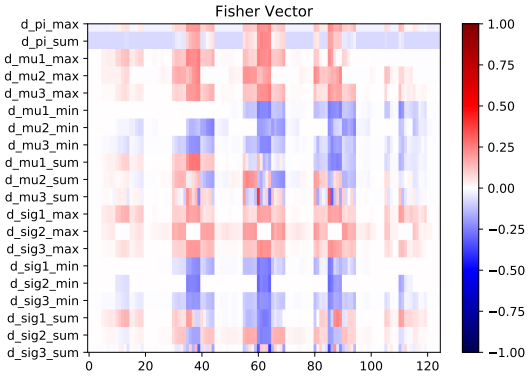
In Abbildung 6.12 werden die Fisher-Vektoren zweier verschiedener Bauteile von drei unterschiedlichen Fahrzeugmodellen miteinander verglichen. Dafür wurden die Bauteile *B-Säule* und *A-Säule-unten* ausgewählt, da diese bei der Klassifikation mit dem Toyota-Yaris-Datensatz vertauscht worden sind (siehe Abbildung 6.11c). In allen anderen Testfällen konnten diese Klassen jedoch sehr gute Ergebnisse liefern und wurden fast immer exakt zugeordnet. Der Vergleich erfolgte zwischen den Modellen FM2, FM3 und dem Toyota Yaris. Die Abbildungen auf der linken Seite zeigen die Fisher-Matrizen der *B-Säule*. Auf der rechten Seite ist die *A-Säule-unten* abgebildet.
Die Matrizen der Fisher-Vektoren werden in dieser Abbildung etwas detaillierter dargestellt. An der Y-Achse lassen sich die einzelnen Gradienten der Gaußiane ablesen. Die X-Achse der Matrix zeigt die Anzahl der Fisher-Vektoren. Die Matrizen wurden mit einem $5 \times 5 \times 5$-Gitter erstellt, so wie das Netz mit dem Testdatensatz des Toyota Yaris trainiert wurde. Anhand der Farbskala neben den Matrizen ist zu sehen, dass die rot gefärbten Zellen wieder positive Werte und die blauen negative symbolisieren (siehe Abschnitt Hyperparameteranpassung).

Wie sich auf den ersten Blick erkennen lässt, besitzen die Repräsentationen der Bauteile viele Gemeinsamkeiten. Die Fisher-Vektoren in der Mitte der Matrix stellen auch die Gaußiane im Zentrum des Raumes dar. Da die Bauteile zentriert im Raum stehen, sind genau an diesen Stellen die meisten Werte vertreten. Die grobe Struktur der Fisher-Matrizen der Bauteile ähnelt sich daher stark. Daraus lässt sich schließen, dass das Netz bei der Klassifikation der beiden Bauteile wahrscheinlich zu einer Fehlklassifikation neigen könnte. Es muss jedoch klargestellt werden, dass der reine Vergleich durch das Auge keine qualitativen Aussagen liefern kann, weshalb an dieser Stelle ausschließlich Vermutungen geäußert werden.
Besonders auffällig ist, dass die markantesten Unterschiede der Bauteile an den Rändern der Fisher-Matrizen zu finden sind. Bei der *B-Säule* der Audi-Modelle sind die Fisher-Vektoren am Rand nur teilweise gefüllt. Dadurch entsteht auf jeder Seite ein dünnes Streifenmuster (vgl. Abbildung 6.12a und 6.12c). Bei der *A-Säule-unten* sind die Bereiche eher zusammenhängend mit Werten nahe der Null. Wirft man nun einen genaueren Blick auf die Fisher-Matrizen der Bauteile des Toyota Yaris, kann man diese Besonderheiten genau andersherum beobachten. Wie in Abbildung 6.12f zu sehen ist, besitzt die *A-Säule-unten* das gleiche Streifenmuster wie die *B-Säule* der Audi-Modelle. Diese Beobachtung ist ein starkes Indiz dafür, dass das Netz die beiden Bauteile miteinander vertauschen wird.

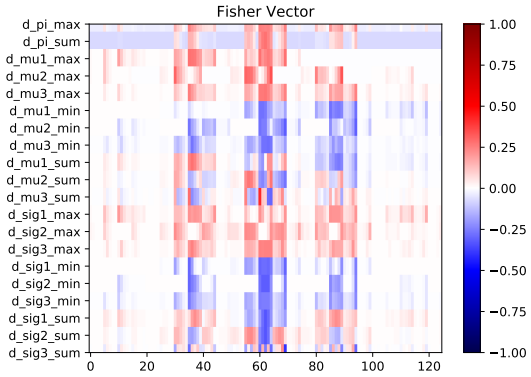
Ursache für die Fehlklassifikation ist demnach die Repräsentation der Bauteile als Fisher-Matrix. Die 3D-Modelle der Audi-Modelle besitzen eine stärkere Ähnlichkeit zueinander, als zum Toyota Yaris. Außerdem kann gesagt werden, dass der 3DmFV-Ansatz bei deckungsgleichen Bauteilen auch eine ähnliche Fisher-Matrix erzeugt. Das begründet die guten Ergebnisse im Test mit den Bauteilgruppen. Es ist demnach sehr wahrscheinlich, dass es zu einer Art Überanpassung des Netzes an die Audi-Modelle kommt. Damit das Netz robuster gegenüber solchen Unterschieden in der Repräsentation wird, muss ein Datensatz mit einer größeren Varianz an Bauteilen verwendet werden.
In der Untersuchung mit unterschiedlichen Datensätzen kommt 3DmFV an seine Grenzen. Aus dem Test geht hervor, dass der 3DmFV-Ansatz besonders gut auf Daten einer spezifischen Domäne abschneidet, jedoch auf Bauteile anderer Hersteller nicht schließen kann. Um über verschiedene Fahrzeuge zu generalisieren, benötigt es mehr Modelle unterschiedlicher Hersteller im Trainingsdatensatz. Außerdem wurde gezeigt, dass durchaus Bauteile mit einer grundlegenden Ähnlichkeit existieren. Um eine möglichst robuste Klassifikation zu erhalten, sollte die Anzahl dieser Bauteile gering gehalten werden.



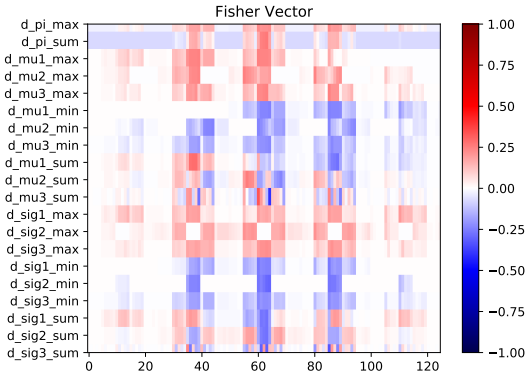
(a) FM2 Fisher-Matrix B-Säule



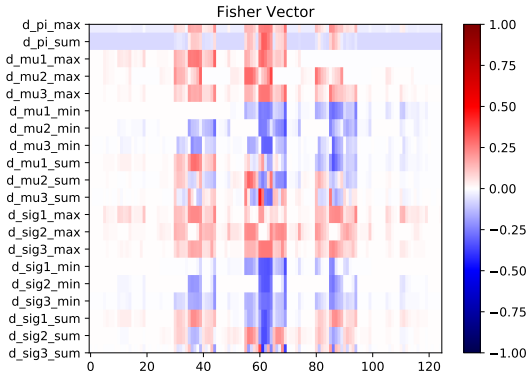
(b) FM2 Fisher-Matrix A-Säule-unten



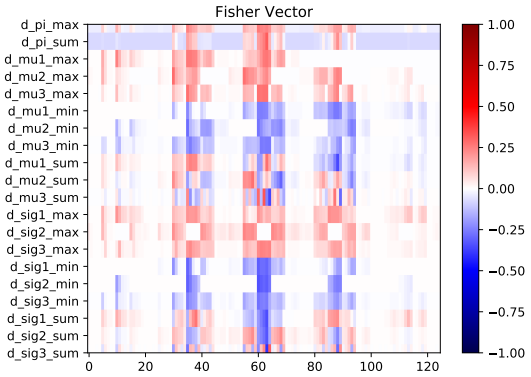
(c) FM3 Fisher-Matrix B-Säule



(d) FM3 Fisher-Matrix A-Säule-unten



(e) Toyota Yaris Fisher-Matrix B-Säule



(f) Toyota Yaris Fisher-Matrix A-Säule-unten

Abbildung 6.12: Vergleich der Fisher-Matrizen zweier Bauteile von verschiedenen Fahrzeugmodellen

7 IMPLEMENTIERUNG

Dieses Kapitel beschäftigt sich mit der Konzeption und Implementierung eines Prototyps. Der Prototyp dient dazu, die vorgestellten Ansätze aus Kapitel 1 zu visualisieren und dabei die Realisierbarkeit der Problematik anzudeuten. Dazu wird der 3DmFV-Ansatz als bereits trainiertes Modell zur Klassifikation von Bauteilen in einer typischen CAD-Anwendung benutzt. Im Folgenden wird die verwendete Software kurz beschrieben und die konzeptionelle Implementierung des Prototyps vorgestellt.

7.1 PROTOTYPISCHE IMPLEMENTIERUNG IN FREECAD

In Kapitel 2 wurde der Simulationsprozess im CAE-Bereich dargestellt und erläutert. Die meisten Anwendungsfälle befinden sich zwischen den ersten beiden Phasen der Prozesskette. Nach oder während des Erstellens des 3D-Modells in einer CAD-Anwendung und vor beziehungsweise in der Präprozessor-Phase kann der Anwender durch maschinelle Lernalgorithmen unterstützt werden. Der Ingenieur kann mit Hilfe einer funktionierenden Bauteilerkennung beim Erstellungsprozess beispielsweise Vorschläge zur Benennung des Bauteils bekommen. Dieses Empfehlungssystem liefert dem Ingenieur sofortiges Feedback und kann bei schwierigen Designentscheidungen von großem Nutzen sein.
Um zu zeigen, dass mit den gewonnenen Erkenntnissen der Arbeit die Realisierung eines solchen Anwendungsfalls grundlegend denkbar ist, wurde ein Prototyp in einer 3D-CAD-Software namens FreeCAD integriert.

7.1.1 FreeCAD

Auf dem Markt existiert eine Vielzahl an CAD-Anwendungen für die verschiedensten Bereiche wie Architektur, Automobilbau, Maschinenbau und Tiefbau. Für die Implementierung des Prototyps fiel die Wahl auf die 3D-CAD-Modellierungsanwendung FreeCAD [Fre19]. FreeCAD wurde 2002 von Jürgen Riegel veröffentlicht und seitdem weiterentwickelt. Dabei handelt es sich um

eine freie Software, deren Quellcode offen zur Verfügung steht. In erster Linie wurde FreeCAD zum Konzipieren mechanischer Konstruktionen erstellt und soll eine Alternative zu kommerziellen Software-Lösungen darstellen. Diese bieten zwar meist eine größere Auswahl an Funktionalitäten, jedoch wird FreeCAD ständig durch seine große Gemeinschaft weiterentwickelt.
Ein großer Vorteil ist die komplette Steuerung der Software mittels Python-Skripten und der Integration eines internen Python-Interpreters. Dadurch ist es möglich, die Software schnell und einfach zu erweitern. Grundlegend ist die Software in zwei Komponenten unterteilt. Im App-Modul sind alle Kernfunktionalitäten hinterlegt, die sich mit der Anwendung befassen. Das GUI-Modul widmet sich den Werkzeugen und der grafischen Darstellung der Inhalte.
Im Rahmen dieser Arbeit wurde FreeCAD so erweitert, dass es möglich ist, ein erstelltes Bauteil durch ein bereits trainiertes neuronales Netz klassifizieren zu lassen. Im folgenden Abschnitt wird die Konzeption dieser Anwendung beschrieben.

7.1.2 Konzeption des Prototyps

Der Prototyp soll ein Bauteil, welches in FreeCAD dargestellt wird, klassifizieren und mit der jeweiligen Klasse benennen. Dafür wird ein Skript benötigt, welches eine Punktwolke empfängt, ein vortrainiertes Modell lädt und die vorhergesagte Klasse als Ausgabe zurückliefert.
Für die Umsetzung des Prototyps dient das trainierte neuronale Netz zur Klassifizierung grober Bauteilgruppen mit 3DmFV, da es bei den Versuchen aus Kapitel 6 die besten Ergebnisse erzielte. Dieses Modell wurde in einer sogenannten Checkpoint-Datei gespeichert. Die Datei beinhaltet alle Gewichte des neuronalen Netzes zum Zeitpunkt der letzten Speicherung. Um die Gewichte zu laden, müssen der Graph und das Modell exakt wiederhergestellt werden. Darum kümmert sich das Python-Skript *predict_3D.py*. Die darin enthaltene Funktion *predict* erwartet eine Punktwolke als Eingabe, definiert den Graphen des neuronalen Netzes und lädt die gespeicherten Gewichte des Trainings. Als Ausgabe dient eine Liste von Tupel aller Klassen und ihrem Klassifikationswert. Dabei ist die Ausgabe so sortiert, dass der höchste Wert und somit die vorhergesagte Klasse ganz oben in der Liste steht.

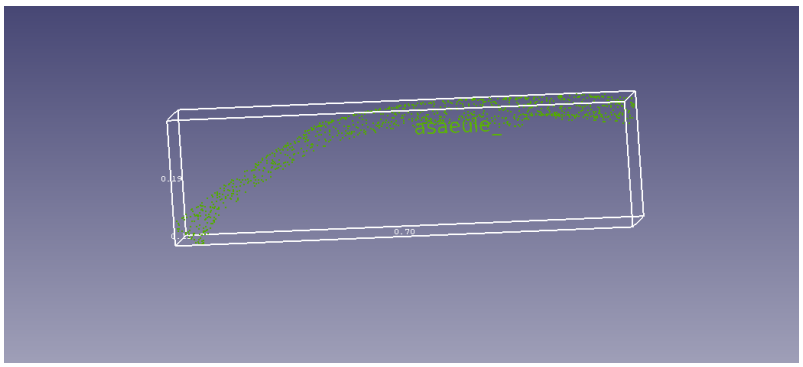


Abbildung 7.1: Klassifizierung einer Punktwolke in FreeCAD

Als Erstes wurde versucht, eine einfache Punktwolke in FreeCAD zu klassifizieren. Die Integration der Skripte ist recht simpel und wird durch Modules realisiert. In einem Unterverzeichnis *Mod* können verschiedene Werkzeugbanken zur Bearbeitung der 3D-Modelle angepasst werden. Alle in diesem Verzeichnis abgelegten Skripte werden von FreeCAD initialisiert und können mittels Import-Befehl über den integrierten Interpreter geladen werden. Da die Erstellung des

Graphen einige Hilfsfunktionen benötigt, wurden neben des *predict_3D.py* Skripts noch weitere Helfer-Skripte hinterlegt. Als Eingabe diente eine PLY-Datei einer extrahierten Punktwolke, welche aus den verwendeten Datensätzen erzeugt wurde. Diese Punktwolke konnte nun mit Hilfe des Skriptes klassifiziert werden. Das Resultat ist in Abbildung 7.1 dargestellt. Dabei handelt es sich um eine Punktwolke der A-Säule des FM3. Die Punktwolke wurde durch das Skript erkannt und grün eingefärbt. Zusätzlich wird bei einem erkannten Bauteil eine Box um das 3D-Modell gezeichnet. Das Label in der Mitte zeigt die vom Netz ermittelte Klasse an. Wie in Abbildung 7.1 zu sehen ist, wurde das Bauteil der richtigen Klasse zugeordnet. Es ist demnach möglich, ein Bauteil als Punktwolke in FreeCAD zu klassifizieren.

Da die Anwender nicht mit reinen Punktwolken arbeiten, wurde ein weiteres Dateiformat verwendet. Das STEP-Format (engl. Standard for the Exchange of Product model data) ist ein DIN-Standard zu Beschreibung von Produktdaten, der in der ISO-Norm 10303 definiert wurde. Es wird oft zum Datenaustausch zwischen Systemen im CAE-Bereich verwendet. In Abbildung 7.2 ist die B-Säule des Toyota Yaris im STEP-Format in FreeCAD dargestellt. In dem Dateibaum auf der linken Seite ist zu erkennen, dass das 3D-Modell der B-Säule nur mit dem Namen *462_SHELL* benannt wurde. Diesen Namen gilt es nun mit der richtigen Klasse des Bauteils zu ersetzen, um dem Anwender einen Vorschlag zur Benennung zu übermitteln. Wie jedoch in Abbildung 7.2b

(a) Ansicht als Bauteil **(b)** Punktansicht

Abbildung 7.2: B-Säule des Toyota Yaris im STEP-Format

zu sehen ist, ist die Anzahl der Punkte im STEP-Format, ähnlich wie bei den FEM-Daten aus Kapitel 4, ungleichmäßig groß und verteilt. Zusätzlich unterscheidet sich die Größe der Facetten, wodurch das Bauteil nicht aus einem einheitlichen Dreiecksnetz besteht. Aus diesem Grund benötigen STEP-Dateien weitere Vorverarbeitungsschritte, bevor sie durch das neuronale Netz klassifiziert werden können.

Der Ablauf der Schritte, die für die Klassifikation einer solchen STEP-Datei nötigt sind, ist in Abbildung 7.3 zu finden. Zuerst muss das Bauteil in ein gleichmäßiges Dreiecksnetz übertragen werden, damit die gleiche Ausgangslage wie bei den FEM-Daten geschaffen wird. Dafür wird im ersten Schritt das Bauteil mittels einer internen FreeCAD-Funktion in einen Mesh konvertiert und in einer STL-Datei exportiert. Die Konvertierung kann bei großen Bauteilen einige Zeit in Anspruch nehmen und die Dauer des gesamten Klassifikationsprozesses verlängern. Danach wird der Mesh geladen und der bereits vorgestellte Sampling-Algorithmus ausgeführt. Somit werden auf der gesamten Fläche des Bauteils insgesamt 1024 Punkte erzeugt.

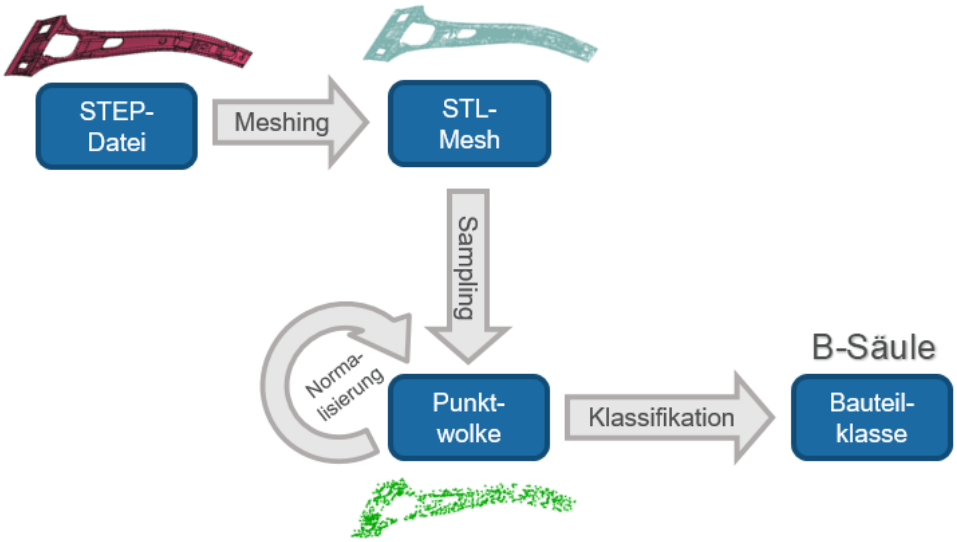


Abbildung 7.3: Ablauf der Klassifikation mit Bauteilen im STEP-Format

Die so erstellte Punktwolke wird anschließend normalisiert und dient danach als Eingabe für das neuronale Netz. Für die Konvertierung des Bauteils im STEP-Format zur Punktwolke ist das Skript *extract.py* zuständig. Darin werden alle geladenen Bauteile in FreeCAD in STL-Dateien konvertiert und anschließend gesamplet. Die normalisierten Punktwolken werden dann an das Skript *detect_ex.py* weitergeleitet. Dieses Skript dient als Schnittstelle zum Skript *predict_3D*, welches das neuronale Netz darstellt. Hauptsächlich ist es für die Visualisierung der empfangenen Klassifikation zuständig. Durch diese Kapselung ist es möglich, andere neuronale Netze für die Klassifikation der Punktwolke zu verwenden. Alle verwendeten Skripte sind im Anhang A zu finden.
In diesem Anwendungsfall wurden verschiedene Bauteile im STEP-Format mit dem 3DmFV-Netz klassifiziert und visualisiert. Das Ergebnis ist in Abbildung 7.4 dargestellt.

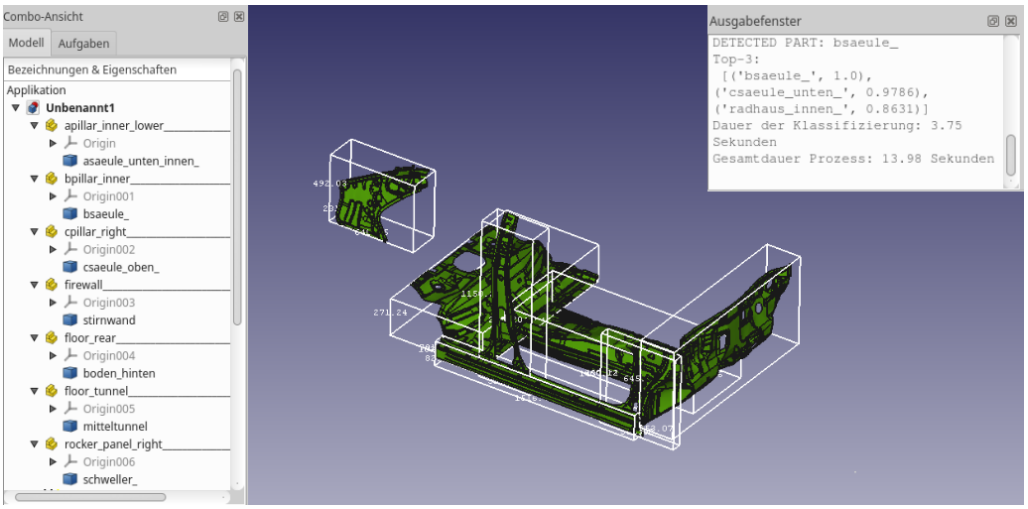


Abbildung 7.4: Klassifikation mehrerer Bauteile im STEP-Dateiformat

Wie zu erkennen ist, wurden alle klassifizierten Bauteile grün gefärbt und umrahmt, damit das Ergebnis für den Anwender sichtbar wird. Zusätzlich wurden die Namen der 3D-Modelle mit der entsprechenden Klasse versehen. In Abbildung 7.4 konnten alle Bauteile richtig zugeordnet

werden. Das liegt auch daran, dass ausschließlich Bauteile im STEP-Format vom Toyota Yaris zur freien Verfügung stehen. Das neuronale Netz wurde mit Teilen der FEM-Daten dieses Automodells trainiert, weshalb die Erkennung sehr stabil funktioniert. Dieser Prototyp soll lediglich die Realisierbarkeit einer solchen Anwendung nachweisen, was durch die erfolgreiche Integration in FreeCAD gezeigt wird. Neben der Umbenennung der 3D-Modelle werden dem Ingenieur zusätzlich die drei Klassen mit der höchsten Wahrscheinlichkeit im Ausgabefenster angezeigt. Somit kann der Anwender die Sicherheit der Klassifikation selbst bewerten und anhand dieser seine Entscheidung treffen.

Zusammenfassend kann gesagt werden, dass die Umsetzung eines automatisierten Benennungssystems, wie in Kapitel 1 beschrieben, durch die erfolgreiche Integration des neuronalen Netzes in FreeCAD denkbar ist. Der Prototyp stellt bereits im vorgestellten Zustand ein solches Vorschlagssystem dar, das jedoch nur auf einen Teilbereich von Bauteilen anwendbar ist. Wenn dem Netz eine einheitliche Bezeichnung der Bauteile zugrunde liegt, kann diese dem Ingenieur empfohlen werden. Somit kann eine Vereinheitlichung der Namensgebung erreicht werden, wodurch klarere Kommunikationswege und einheitlich strukturierte Daten entstehen. Dennoch muss gesagt werden, dass die Klassifikation aller Bauteile eines Fahrzeugs anhand der hier gewonnenen Ergebnisse und nach aktuellem Stand der Technik nur schwer umsetzbar ist.
Die anderen genannten Anwendungsfälle können, basierend auf der Klassifikation einzelner Bauteile, auch realisiert werden. Dazu werden im folgenden Abschnitt die Konzepte solcher Anwendungen kurz vorgestellt.

7.2 KONZEPTION ANDERER ANWENDUNGSFÄLLE

In Kapitel 1 wurden insgesamt fünf verschiedene Anwendungsfälle beschrieben, die eine automatische Erkennung eines Fahrzeugbauteils als Grundlage voraussetzen. Wie der Prototyp im vorherigen Abschnitt gezeigt hat, ist eine Umsetzung des automatisierten Bennennungssystems für bestimmte Bauteile durchaus möglich. Die Integration in eine existierende CAD-Anwendung kann mittels Skripten realisiert werden. Lediglich die Robustheit des Netzes sollte durch Datensätze mit größerer Varianz an Fahrzeugen verbessert werden.
Zusätzlich kann mittels der Unterscheidung in grobe Bauteilgruppen das Einsortieren von Bauteillisten realisiert werden. Dazu werden lediglich alle klassifizierten Bauteile ihrer Klasse untergeordnet und dem Anwender als hierarchische Liste zurückgegeben.

Das Finden von ähnlichen Konstruktionen oder die Vorhersage von Schweißpunktparametern können nach aktuellem Stand nur teilweise umgesetzt werden. Die Vorhersage von Schweißpunkten anhand reiner Geometriedaten erscheint schwierig, da die Stabilität der Erkennung nicht sicher gewährleistet werden kann. Ansätze im Bereich des maschinellen Lernens können in den wenigsten Fällen eine hundertprozentige Genauigkeit garantieren, welche in diesem Anwendungsfall jedoch nahezu gefordert ist. Aus diesem Grund wird in Abbildung 7.5 ein Konzept für eine alternative Realisierung der Anwendung dargestellt.
Durch die bereits umsetzbare Klassifikation eines Bauteils können gezielte Abfragen an eine Datenbank gesendet werden. Die Bauteilklasse dient in diesem Fall als Primärschlüssel, da sie eindeutig zu einer 3D-Geometrie zuordenbar ist. Die Datenbank beinhaltet die vorher aufbereiteten FEM-Daten und stellt eine Ansammlung aller Bauteile mit ihren spezifischen Parametern über alle Fahrzeugmodelle dar. Somit ist es durch einfache Datenbankabfragen möglich, spe-

zielle Informationen für das erkannte Bauteil anhand von alten Daten der selben Klasse anzuzeigen. Diese Informationen können beispielsweise verwendete Materialstärken, Verteilung der Schweißpunkte und geometrische Repräsentation sein. Zusätzlich ist es denkbar, diese Daten dem Anwender in einer aggregierten Form wiederzugeben, wodurch die Empfehlung kompakter und informativer ausfällt. So können unter anderem ähnliche Konstruktionen vorgeschlagen werden oder eine Anzeige der Verteilung von Schweißpunkten anderer Bauteile der selben Klasse als 3D-Koordinaten erfolgen.

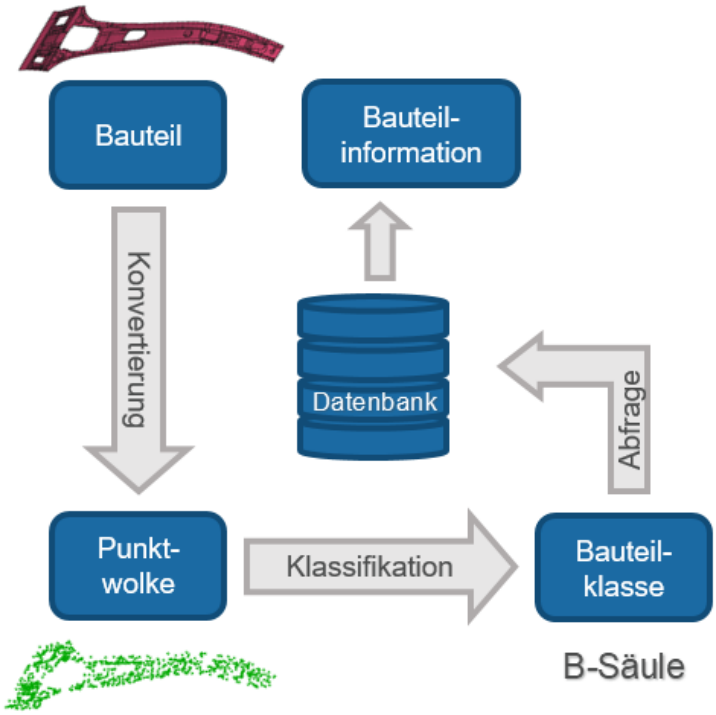


Abbildung 7.5: Schematischer Ablauf eines Empfehlungssystems

Das Segmentieren von Bauteilgruppen kann ebenfalls von Ansätzen wie PointNet oder 3DmFV gelöst werden. Dazu wird jedoch eine andere Datengrundlage benötigt. In diesem Fall müssen beispielsweise die *Seitenwandrahmen* verschiedener Fahrzeugmodelle extrahiert und gesamplet werden. Anschließend werden alle Punkte des Bauteils mit der zugehörigen Klasse annotiert. Für das Training mit dem neuen Datensatz besitzen beide Deep-Learning-Ansätze gesonderte Netzarchitekturen, welche direkt für die Segmentierung von Punktwolken entwickelt wurden. Die Evaluation dieser Architekturen wird nicht weiter in dieser Arbeit behandelt.
Die vorgestellten Konzepte und Lösungsansätze der Anwendungsfälle, können als Grundlage für weitere Forschungsfragen dienen.

8 ZUSAMMENFASSUNG

Ziel der Arbeit war es, mit Hilfe von Methoden des maschinellen Lernens Fahrzeugbauteile aus FEM-Daten zu erkennen und zu klassifizieren. Dafür wurde in Kapitel 4 ein konzeptioneller Ansatz zur Vorverarbeitung von FEM-Daten beschrieben, damit die für das Training eines neuronalen Netzes notwendigen Punktwolken extrahiert werden können. Bei der Entstehung des Konzepts ging hervor, dass die Struktur der FEM-Daten einige Nachteile mit sich bringt. Dabei wurde gezeigt, dass die Benennung der Bauteile von verschiedenen Modellen nicht uniform ist. Außerdem bestehen die Fahrzeuge verschiedener Hersteller aus einer unterschiedlichen Anzahl an Bauteilen, wodurch keine vollautomatische Generierung von Trainingsdaten möglich ist. Aufgrund dieser Tatsachen wurde ein Konzept zur Extraktion von Punktwolken speziell ausgewählter Bauteile aus FEM-Daten von insgesamt sechs Fahrzeugmodellen entworfen. Die so generierten Datensätze stellten die Grundlage des Vergleichs der in Kapitel 5 vorgestellten Deep-Learning-Methoden dar.

In Kapitel 6 wurden zwei Ansätze evaluiert und die entstandenen Trainingsergebnisse miteinander verglichen. Dabei lag der Fokus auf der Generalisierung der Ansätze, um die Frage zu klären, wie gut die Klassifikation der Deep-Learning-Architekturen auf einen noch nicht gesehenen Datensatz anwendbar ist. Dafür wurden verschiedene Klassifikationsprobleme entworfen, die jeweils einen unterschiedlichen Schwierigkeitsgrad aufweisen sollten. Die Untersuchungen ergaben, dass der 3DmFV-Ansatz in allen getesteten Fällen besser als PointNet abschnitt. Außerdem konnte aufgrund der sehr guten Ergebnisse gezeigt werden, dass eine Klassifikation von Bauteilen aus FEM-Daten durchaus möglich ist. Zusätzlich wurde gezeigt, dass Bauteile existieren, die anfälliger für Fehlklassifikationen sind. So ergab die Untersuchung, dass Bauteile, die eine starke geometrische Ähnlichkeit besitzen, wahrscheinlicher vertauscht werden.
In Anbetracht der Erkenntnisse wurden weitere Versuche allein mit dem 3DmFV-Ansatz durchgeführt. Diese ergaben, dass der Ansatz bessere Ergebnisse auf den Fahrzeugdaten einer spezifischen Domäne, sprich eines Herstellers, liefert. Hinsichtlich der in Kapitel 1 vorgestellten Anwendungsfälle sollte eine Begrenzung der Datendomäne keinen größeren Nachteil darstellen.

Die Resultate dieser Arbeit beziehen sich jedoch nur auf eine Menge bestimmter Bauteile, wodurch keine pauschale Aussage auf die gesamte Menge an Bauteilen eines Fahrzeugs getroffen werden kann. Außerdem werden nur insgesamt sechs verschiedene Fahrzeuge für das Training

verwendet. An dieser Stelle benötigt es zusätzliche Modelle, damit eine größere Varianz in den Daten vorhanden ist und somit bessere Aussagen über die Generalisierung der Methoden getroffen werden kann.
In Kapitel 7 wurde die Implementierung eines Prototyps vorgestellt. Dabei wurde gezeigt, dass die Realisierung einer Anwendung zur Klassifikation von Fahrzeugbauteilen möglich ist. Zudem konnten weitere Konzepte zur Umsetzbarkeit anderer Anwendungsfälle entworfen werden. Die daraus gewonnenen Erkenntnisse zeigen, dass mit Hilfe von Methoden des maschinellen Lernens eine Unterstützung des Ingenieurs in der CAE-Prozesskette erreicht werden können.

AUSBLICK

Aus den gewonnenen Erkenntnissen der vorliegenden Arbeit können einige Aussagen getroffen werden, die für eine weitere Bearbeitung des Forschungsthemas relevant sind. Es wurde gezeigt, dass eine Klassifikation von bestimmten Bauteilen aus FEM-Daten möglich ist. Für die nachfolgende Forschung sollte diese Menge an Bauteilen erhöht werden, um eine Aussage über die Klassifikation eines gesamten Fahrzeugs treffen zu können. Außerdem wäre es sinnvoll, die untersuchten Ansätze mit größeren Datensätzen zu trainieren. Hierbei sollte besonders auf die Varianz der Modelle geachtet werden, damit valide Erkenntnisse zur Generalisierung der Methoden gewonnenen werden können.
Weiterhin können die vorgestellten Anwendungsfälle aus Kapitel 1 verfolgt werden. So könnte die Vorhersage von Schweißpunkten oder das Finden von ähnlichen Bauteilkonstruktionen mit einem der eingeführten Konzepte realisiert werden. Die Vorhersage von Schweißpunken anhand von reinen Geometrie-Daten, erweist sich, nach Stand der Technik, als schwer. Durch Hinzunahme weiterer Bauteileigenschaften könnten komplett neue Deep-Learning-Ansätze trainiert werden.
Weiterer Forschungsbedarf ergibt sich bei der Segmentierung von Bauteilgruppen. Beide Ansätze verfügen über gesonderte Segmentierungs-Architekturen, womit Bauteilgruppen in ihre einzelnen Elemente zerlegt werden können. Dafür werden andere Trainingsdaten verwendet, wodurch ein komplett neues Konzept zur Generierung nötig wäre.
Zusammenfassend kann gesagt werden, dass maschinelles Lernen ein großes Potential im CAE-Bereich besitzt, wie die positiven Ergebnisse der Machbarkeitsstudie zeigen konnten. Um jedoch für die Industrie eine lukrative Unterstützung zu verwirklichen, müssen noch weitere spannende Forschungsfragen in Zukunft geklärt werden.

A ANHANG SKRIPTE

A.1 SKRIPT - EXTRACT.PY

```
import Points
import Mesh
import numpy as np
import os
import json
import time
from pointcloud_generator import point_cloud
from dmfv.utils import data_prep_util as ut
from detect_ex import detect
os.chdir('/home/nick.scheider.extern/Downloads/squashfs-root/usr/Mod/Points/')

def extract():
    """ extract the parts from mesh"""
    start = time.time()
    #get parts in document
    parts_in_doc = App.ActiveDocument.findObjects('Part::Feature')

    for part in parts_in_doc:

        mesh = getMeshfromStep(part)
        #iterate over mesh and save faces and coords of points
        pc = []
        for f in mesh.Facets:
            points = []
            for v in f.Points:
                points.append(v)
            pc.append(points)
        pc = np.array(pc).tolist()
        with open("coords.json", "w") as f:
            json.dump(pc,f, indent=4)

        pc_sample = point_cloud("./", "coords.json", 1024) #sample extracted pc
        pc_sample = normalize_pc(pc_sample) #normalize points

        #save pc in ply and show it in freecad
        ut.export_ply(pc_sample, "./pc.ply")
```

```
        Points.insert('pc.ply', App.ActiveDocument.Name)
        pc = App.ActiveDocument.getObjectsByLabel("pc")

        #detect label of pc
        label, prob = detect(pc_sample)

        if prob <= 1:
            part.ViewObject.BoundingBox = True
            part.ViewObject.ShapeColor = (0.33,0.67,0.00)
        elif prob < 0.95:
            part.ViewObject.ShapeColor = (1.00,0.56,0.02)
        elif prob < 0.9:
            part.ViewObject.ShapeColor = (0.95,0.19,0.00)
        else:
            part.ViewObject.BoundingBox = False
            part.ViewObject.ShapeColor = (0.76,0.00,0.00)
        #set name to label
        if label != "" :
            part.Label = label
        end = time.time() - start
        App.Console.PrintMessage(f'Gesamtdauer Prozess: {round(end,2)} Sekunden \n'
    )

def getMeshfromStep(part):
    """ konvert to stl mesh"""
    step = App.ActiveDocument.getObjectsByLabel(part.Label)
    Mesh.export(step,u"mesh.stl")
    mesh = Mesh.Mesh('mesh.stl')
    return mesh

def normalize_pc(points):
    """ normalize pc"""
    pc = np.asarray(points)
    print("normalizing point cloud...")
    pc_mean = np.mean(pc, axis=0)   #calcualte centroid of cloud
    cen_points = pc - pc_mean       #move centroid to zero
    max_dis = np.max(np.sqrt(np.sum(abs(cen_points)**2, axis=-1))) #calculate
    max_dist in pc
    norm_points = (cen_points/max_dis) #calculate normalized points
    return norm_points
```

Listing A.1: extract.py

A.2 SKRIPT - DETECT_EX.PY

```
import FreeCADGui as Gui
import FreeCAD as App
import Draft
import Part
import numpy as np
import os
import math
import time
from dmfv import predict_3D
import tensorflow as tf

def calc_probs(pred_values):
    """ calculate probability"""
```

```
    probs = []
    for val in pred_values:
        x = (1/(1+math.exp(-val)))
        probs.append(round(x,4))
    return probs

def detect(pc):
    """ classify pc with top-k values"""
    pc = np.array(pc) # convert to np array

    #connection to net
    os.chdir('/home/nick.scheider.extern/Downloads/squashfs-root/usr/Mod/Points/
    dmfv')
    start = time.time()
    cls_names, pred_vals = predict_3D.predict(pc)
    end = time.time()-start
    probs = calc_probs(pred_vals)
    label = cls_names[0]

    if pred_vals != []:
        k=3 # top k classes
        topk = [(cls_names[i], probs[i]) for i in range(k)]

        App.Console.PrintMessage(f'DETECTED PART: {label}\n Top-3: \n')
        for k in topk:
            App.Console.PrintMessage(f'{k}\n')
        App.Console.PrintMessage(f'Dauer der Klassifizierung: {round(end,2)}
    Sekunden \n')
        p1 = App.Vector(0, 0, 0)
        t1 = label
        XAxis = App.Vector(1, 0, 0)

        Label = Draft.makeText(t1, point=p1)
        place = App.Placement(p1, App.Rotation(XAxis, 90))
        Label.Placement = place
        Label.ViewObject.TextColor = (0.33,0.67,0.00)
        Label.ViewObject.FontSize = 0.03
    return label,probs[0]
```

Listing A.2: detect_ex.py

A.3 SKRIPT - PREDICT_3D.PY

```
import os
import sys
import numpy as np
import matplotlib
matplotlib.use('pdf')
import importlib
import argparse
import tensorflow as tf
import pickle
import time

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
os.chdir(BASE_DIR)
sys.path.append(BASE_DIR)
sys.path.append(os.path.join(BASE_DIR, 'models'))
```

```
sys.path.append(os.path.join(BASE_DIR, 'utils'))
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
import tf_util
import visualization
import provider
import utils
import pc_util

# ModelNet40 official train/test split. MOdelNet10 requires separate downloading and sampling.
MAX_N_POINTS = 1024
NUM_CLASSES = 15

augment_rotation, augment_scale, augment_translation, augment_jitter, augment_outlier = (False, True, True, True, False)

parser = argparse.ArgumentParser()
#Parameters for learning
parser.add_argument('--gpu', type=int, default=2, help='GPU to use [default: GPU 0]')
parser.add_argument('--model', default='3dmfv_net_cls', help='Model name [default: 3dmfv_net_cls]')
parser.add_argument('--log_dir', default='log_E100', help='Log dir [default: log]')
parser.add_argument('--num_point', type=int, default=1024, help='Point Number [256/512/1024/2048] [default: 1024]')
parser.add_argument('--max_epoch', type=int, default=200, help='Epoch to run [default: 200]')
parser.add_argument('--batch_size', type=int, default=1, help='Batch Size during training [default: 64]')
parser.add_argument('--learning_rate', type=float, default=0.001, help='Initial learning rate [default: 0.001]')
parser.add_argument('--momentum', type=float, default=0.9, help='Initial learning rate [default: 0.9]')
parser.add_argument('--optimizer', default='adam', help='adam or momentum [default: adam]')
parser.add_argument('--decay_step', type=int, default=200000, help='Decay step for lr decay [default: 200000]')
parser.add_argument('--decay_rate', type=float, default=0.7, help='Decay rate for lr decay [default: 0.7]')
parser.add_argument('--weight_decay', type=float, default=0.0, help='weight decay coef [default: 0.0]')

# Parameters for GMM
parser.add_argument('--gmm_type',  default='grid', help='type of gmm [grid/learn], learn uses expectation maximization algorithm (EM) [default: grid]')
parser.add_argument('--num_gaussians', type=int , default=5, help='number of gaussians for gmm, if grid specify subdivisions, if learned specify actual number[default: 5, for grid it means 125 gaussians]')
parser.add_argument('--gmm_variance', type=float,  default=0.04, help='variance for grid gmm, relevant only for grid type')
FLAGS = parser.parse_args()



N_GAUSSIANS = FLAGS.num_gaussians
GMM_TYPE = FLAGS.gmm_type
GMM_VARIANCE = FLAGS.gmm_variance

BATCH_SIZE = FLAGS.batch_size
```

```
58 NUM_POINT = FLAGS.num_point
59 MAX_EPOCH = FLAGS.max_epoch
60 BASE_LEARNING_RATE = FLAGS.learning_rate
61 GPU_INDEX = FLAGS.gpu
62 MOMENTUM = FLAGS.momentum
63 OPTIMIZER = FLAGS.optimizer
64 DECAY_STEP = FLAGS.decay_step
65 DECAY_RATE = FLAGS.decay_rate
66 WEIGHT_DECAY = FLAGS.weight_decay
67
68 MODEL = importlib.import_module(FLAGS.model) # import network module
69 MODEL_FILE = os.path.join(BASE_DIR, 'models', FLAGS.model+'.py')
70
71
72 LOG_DIR = 'log/modelnet' + str(NUM_CLASSES) + '/' + FLAGS.model + '/'+ GMM_TYPE +
       str(N_GAUSSIANS) + '_' + FLAGS.log_dir
73 if not os.path.exists(LOG_DIR):
74     print("ERROR: Logdir not exists!")
75
76 SHAPE_NAMES = [line.rstrip() for line in
77                open(os.path.join(BASE_DIR, f'{LOG_DIR}/shape_names.txt'))]
78 pickle.dump(FLAGS, open( os.path.join(LOG_DIR, 'parameters.p'), "wb" ) )
79
80
81 LOG_FOUT = open(os.path.join(LOG_DIR, 'predict.txt'), 'w')
82 LOG_FOUT.write(str(FLAGS)+'\n')
83 LOG_FOUT.write("augmentation RSTJ = " + str((augment_rotation, augment_scale,
       augment_translation, augment_jitter, augment_outlier))) #log augmentaitons
84
85 BN_INIT_DECAY = 0.5
86 BN_DECAY_DECAY_RATE = 0.5
87 BN_DECAY_DECAY_STEP = float(DECAY_STEP)
88 BN_DECAY_CLIP = 0.99
89
90 LIMIT_GPU = True
91
92 def log_string(out_str):
93     LOG_FOUT.write(out_str+'\n')
94     LOG_FOUT.flush()
95     print(out_str)
96
97 def predict(points):
98     """
99     classify on point cloud with trained model
100    """
101    log_dir = LOG_DIR
102    gmm = utils.get_3d_grid_gmm(subdivisions=[N_GAUSSIANS, N_GAUSSIANS, N_GAUSSIANS
       ], variance=GMM_VARIANCE)
103    pickle.dump(gmm, open(os.path.join(LOG_DIR, 'gmm.p'), "wb"))
104
105    # load the model
106    model_checkpoint = os.path.join(log_dir, "model.ckpt")
107    if not(os.path.isfile(model_checkpoint+".meta")):
108        raise ValueError("No log folder availabe with name " + str(log_dir))
109    # reBuild Graph
110    with tf.Graph().as_default():
111        with tf.device('/gpu:'+str(GPU_INDEX)):
112
113            points_pl, _,  w_pl, mu_pl, sigma_pl,  = MODEL.placeholder_inputs(
```

```
    BATCH_SIZE, NUM_POINT, gmm,)
114             is_training_pl = tf.placeholder(tf.bool, shape=())
115 
116             # Get model and loss
117             pred, fv = MODEL.get_model(points_pl, w_pl, mu_pl, sigma_pl,
    is_training_pl, num_classes=NUM_CLASSES)
118 
119             ops = {'points_pl': points_pl,
120                    'w_pl': w_pl,
121                    'mu_pl': mu_pl,
122                    'sigma_pl': sigma_pl,
123                    'is_training_pl': is_training_pl,
124                    'pred': pred,
125                    'fv': fv}
126             # Add ops to save and restore all the variables.
127             saver = tf.train.Saver()
128 
129             # Create a session
130             sess =  tf_util.get_session(GPU_INDEX, limit_gpu=LIMIT_GPU)
131 
132             # Restore variables from disk.
133             saver.restore(sess, model_checkpoint)
134             print("Model restored.")
135 
136             log_string('----PREDICTION----')
137             pc = points
138             current_data = np.expand_dims(pc, axis=0)
139             current_data = current_data[:, 0:NUM_POINT, :]
140 
141             feed_dict = {ops['points_pl']: current_data,
142                          ops['w_pl']: gmm.weights_,
143                          ops['mu_pl']: gmm.means_,
144                          ops['sigma_pl']: np.sqrt(gmm.covariances_),
145                          ops['is_training_pl']: False}
146 
147             pred_label, fv_data = sess.run([ops['pred'], ops['fv']], feed_dict=
    feed_dict)
148 
149             #reduce dimension and sort label map
150             pred_val = np.squeeze(pred_label, axis=0)
151             ind = np.argsort(pred_val)
152             pred_val = -np.sort(-pred_val)
153             class_names = [SHAPE_NAMES[i] for i in reversed(ind)]
154             class_probs = [(class_names[i], pred_val[i]) for i in range(len(
    class_names))]
155 
156             return class_names, pred_val
157 
158 if __name__ == '__main__':
159     p=[]
160     predict(p)
161     LOG_FOUT.close()
```

Listing A.3: predict_3D.py

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LITERATURVERZEICHNIS

[AH01] Selim Aksoy and Robert M. Haralick. Feature normalization and likelihood-based similarity measures for image retrieval. *Pattern Recognition Letters*, 22(5):563 – 582, 2001. Image/Video Indexing and Retrieval. URL: http://www.sciencedirect.com/science/article/pii/S0167865500001124, doi:https://doi.org/10.1016/S0167-8655(00)00112-4.

[ASS+18] Eman Ahmed, Alexandre Saint, Abd El Rahman Shabayek, Kseniya Cherenkova, Rig Das, Gleb Gusev, Djamila Aouada, and Björn E. Ottersten. Deep learning advances on different 3d data representations: A survey. *CoRR*, abs/1808.01462, 2018. URL: http://arxiv.org/abs/1808.01462, arXiv:1808.01462.

[BLF17] Yizhak Ben-Shabat, Michael Lindenbaum, and Anath Fischer. 3d point cloud classification and segmentation using 3d modified fisher vector representation for convolutional neural networks. *CoRR*, abs/1711.08241, 2017. URL: http://arxiv.org/abs/1711.08241, arXiv:1711.08241.

[D.17] Chicco D. Ten quick tips for machine learning in computational biology. *BioData mining*, abs/1904.10300, 2017. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5721660/.

[ES00] Martin Ester and Jörg Sander. *Knowledge Discovery in Databases*, volume 1. Springer-Verlag Berlin Heidelberg, 2000.

[Fre19] FreeCad. Freecad, 2019. URL: https://www.freecadweb.org/wiki/Main_Page.

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

[Groa] ESI Group. Pam-crash. URL: https://www.esi-group.com/de/pam-crash.

[Grob] HDF Group. Hdf5 data structure. URL: https://www.hdfgroup.org/solutions/hdf5/.

[Gro06] ESI Group. Solver reference manual, 2006. URL: http://www2.ifb.uni-stuttgart.de/fem/Ex_Materials/Composites_Mat131_BiPhase(PlyTypes0,8)/PAM-CRASH-2006_SolverReferenceManual_PARTS.pdf.

[HLL+19] Z. Han, H. Lu, Z. Liu, C. Vong, Y. Liu, M. Zwicker, J. Han, and C. L. P. Chen. 3d2seqviews: Aggregating sequential views for 3d global feature learning by cnn with hierarchical attention aggregation. *IEEE Transactions on Image Processing*, 28(8):3986–3999, Aug 2019. doi:10.1109/TIP.2019.2904460.

[IDZ80] Ronald Iman, James Davenport, and D. Zeigler. *Latin hypercube sampling (program user's guide). [LHC, in FORTRAN]*. Department of Energy, 01 1980.

[JH99] Tommi S. Jaakkola and David Haussler. Exploiting generative models in discriminative classifiers. In *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, page 487–493, Cambridge, MA, USA, 1999. MIT Press.

[Kan16] Asako Kanezaki. Rotationnet: Learning object classification using unsupervised viewpoint estimation. *CoRR*, abs/1603.06208, 2016. URL: http://arxiv.org/abs/1603.06208, arXiv:1603.06208.

[Kar19a] Andrej Karpathy. Backpropagation, intuitions, 2019. URL: http://cs231n.github.io/optimization-2/.

[Kar19b] Andrej Karpathy. Convolutional neural networks: Architectures, convolution / pooling layers, 2019. URL: http://cs231n.github.io/convolutional-networks/.

[Kar19c] Andrej Karpathy. Neural networks part 1: Setting up the architecture, 2019. URL: http://cs231n.github.io/neural-networks-1/.

[Kle13] B. Klein. *FEM: Grundlagen und Anwendungen der Finite-Elemente-Methode*. Studium Technik. Vieweg+Teubner Verlag, 2013. URL: https://books.google.de/books?id=h-v3BQAAQBAJ.

[Kno19] Max Knorr. Parametervorhersage und automatisiertes erzeugen von verbindungstechnologien in cax-anwendungen von automobil-karosserien. Master's thesis, Technische Universität Stuttgart, 9 2019.

[LD] LS-Dyna. Ls-dyna keyfiles. URL: https://www.dynasupport.com/tutorial/getting-started-with-ls-dyna/getting-started.

[Mat] Wolfram MathWorld. Barycentric coordinates. URL: http://mathworld.wolfram.com/BarycentricCoordinates.html.

[Mat10] Friedrich U. Mathiak. Die methode der finiten elemente (fem), 2010. URL: http://www.mechanik-info.de/dokumente/Skript_FEM.pdf.

[MBC79] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979. URL: http://www.jstor.org/stable/1268522.

[MS15] D. Maturana and S. Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928, Sep. 2015. doi:10.1109/IROS.2015.7353481.

[NKB] P. Bhowmick N. Karmakar, A. Biswas and B.B. Bhattacharya. Construction of 3d orthogonal cover. URL: https://cse.iitkgp.ac.in/~pb/research/3dpoly/3dpoly.html.

[Pil19] Akhil Pillai. Parameter estimation for spot weld design. Master's thesis, Technische Universität Dresden, 3 2019.

[Pri] Princeton. Princeton modelnet. URL: https://modelnet.cs.princeton.edu/.

[PSM10] Florent Perronnin, Jorge Sánchez, and Thomas Mensink. Improving the fisher kernel for large-scale image classification. In *Computer Vision*, volume 6314, pages 143–156, 09 2010. doi:10.1007/978-3-642-15561-1_11.

[QSMG16] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016. URL: http://arxiv.org/abs/1612.00593, arXiv:1612.00593.

[QSN+16] Charles Ruizhongtai Qi, Hao Su, Matthias Nießner, Angela Dai, Mengyuan Yan, and Leonidas J. Guibas. Volumetric and multi-view cnns for object classification on 3d data. *CoRR*, abs/1604.03265, 2016. URL: http://arxiv.org/abs/1604.03265, arXiv:1604.03265.

[RACT15] Emanuele Rodolà, Andrea Albarelli, Daniel Cremers, and Andrea Torsello. A simple and effective relevance-based point sampling for 3d shapes. *Pattern Recogn. Lett.*, 59(C):41–47, July 2015. URL: https://doi.org/10.1016/j.patrec.2015.03.009, doi:10.1016/j.patrec.2015.03.009.

[Roy01] David Roylance. Finite element analysis, 2001. URL: https://ocw.mit.edu/courses/materials-science-and-engineering/3-11-mechanics-of-materials-fall-1999/modules/MIT3_11F99_fea.pdf.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL: http://arxiv.org/abs/1609.04747, arXiv:1609.04747.

[Run10] Thomas A. Runkler. *Data Mining*, volume 1. Vieweg+Teubner Verlag, The address, 2010.

[Sch] Axel Schueler. Baryzentrischekoordinaten. URL: https://lsgm.uni-leipzig.de/KoSemNet/pdf/schueler-01-4.pdf.

[SGWM18] Jong-Chyi Su, Matheus Gadelha, Rui Wang, and Subhransu Maji. A deeper look at 3d shape classifiers. *CoRR*, abs/1809.02560, 2018. URL: http://arxiv.org/abs/1809.02560, arXiv:1809.02560.

[Shi] ShiQiu0419. Normalize the point sets. URL: https://github.com/charlesq34/pointnet2/issues/95.

[SMKL15] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik G. Learned-Miller. Multi-view convolutional neural networks for 3d shape recognition. In *Proc. ICCV*, 2015.

[Sol] Beta Simulation Solutions. Ansa preprocessor. URL: https://www.beta-cae.com/ansa.htm.

[SPT18] Konstantinos Sfikas, Ioannis Pratikakis, and Theoharis Theoharis. Ensemble of panorama-based convolutional neural networks for 3d model classification and retrieval. *Computers and Graphics*, 71:208 – 218, 2018. URL: http://

www.sciencedirect.com/science/article/pii/S0097849317301978, doi:https://doi.org/10.1016/j.cag.2017.12.001.

[Stu07] Universität Stuttgart. Tutorial shell elements, 2007. URL: http://www2.ifb.uni-stuttgart.de/fem/PDF_FILES/Tutorial1-2_ElasticCantilever_V4.pdf.

[TEC07] LIVERMORE SOFTWARE TECHNOLOGY. Ls-dyna keyword user's manual vol 1, 2007. URL: http://lstc.com/pdf/ls-dyna_971_manual_k.pdf.

[TL19] Yew Siang Tang and Gim Hee Lee. Transferable semi-supervised 3d object detection from RGB-D data. *CoRR*, abs/1904.10300, 2019. URL: http://arxiv.org/abs/1904.10300, arXiv:1904.10300.

[ZT17] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. *CoRR*, abs/1711.06396, 2017. URL: http://arxiv.org/abs/1711.06396, arXiv:1711.06396.