



PROTOTYP EINES INTELLIGENTEN
INFORMATIONSFILTER MITTELS VISUELLER
BAUTEILERKENNUNG AUF MOBILEN
ENDGERÄTEN

Master

im Studiengang Medieninformatik

vorgelegt von	Natalie Greß
Matrikelnummer	2895049
Abgabedatum	31. Juli 2021
Erstbetreuer	Prof. Dr. Bartosz von Rymon Lipinski
Zweitbetreuer	Prof. Dr. Timo Götzemann

Danksagung

An dieser Stelle möchte ich mich bei denjenigen Bedanken, die mich während der Anfertigung der Masterarbeit, und auch schon zuvor, unterstützt und motiviert haben.

Zuerst gebührt mein Dank Herrn Prof. Dr. von Rymon Lipinski für die angenehme Betreuung dieser Arbeit. Für die hilfreichen Anregungen und die konstruktive Kritik möchte ich mich herzlich bedanken.

Weiterhin gebührt mein Dank Quanos Content Solutions, genauer meinem Betreuer Jochen Marczinzik, für die stetige Hilfsbereitschaft und Unterstützung während der Entstehung dieser Arbeit.

Bei meinem Lebensgefährten Ilja Baumann möchte ich mich ganz herzlich für die erbrachte Unterstützung und Geduld während der Phase der Masterarbeit bedanken.

Abschließend möchte ich mich bei meinen Eltern bedanken, welche mir durch ihre Unterstützung mein Wunschstudium ermöglicht haben. Mein ganz besonderer Dank gilt meiner Mutter, welche mich in jeder Lebenssituation unterstützt sowie motiviert hat und ich dies nun zurück geben kann.

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit einer intelligenten und visuellen Suche nach 3D-Modellen aus 2D-Bildern. Bisherige Lösungen bieten Rekonstruktionen von 3D-Modellen aus Bildern an, aber keine Suche in weiteren 3D-Modellen. Außerdem sind diese nicht für mobile Endgeräte entwickelt. Das Ziel dieser Arbeit ist ein neuronales Netz speziell für mobile Endgeräte zu erstellen, welches 3D-Modelle aus 2D-Bildern generiert und diese anschließend mit einem anderen Set aus 3D-Modellen abgleicht, um das ähnlichste zu finden.

Im ersten Meilenstein wird entschieden, welcher Datensatz für das Trainieren des neuronalen Netzes eingesetzt werden soll. Hierbei wurde der ShapeNet Core Datensatz verwendet. Im nächsten Schritt wird ein Encoder-Decoder Netzwerk erstellt. Der Encoder soll zur Verarbeitung der 2D-Bilder einen Merkmalsvektor erstellen. Hierbei werden zwei Netze genutzt: Eine selbst erstellte Netzarchitektur und das MobileNetV2 von Google. Der Decoder soll diesen Vektor annehmen und daraus ein 3D-Modell generieren. Dabei werden auch hier zwei Netze verwendet: Eine selbst erstellte Architektur sowie das IM-NET. Es werden zwei Typen von 3D-Repräsentationen getestet, die Voxel und implizite Darstellung. Als Metriken werden Accuracy sowie Intersection over Union genutzt. Anschließend wird ein Ähnlichkeitsmaß zum Vergleich bzw. zur Suche in weiteren 3D-Modellen eingeführt. Zuletzt werden verschiedene Experimente zur Evaluation durchgeführt.

Nach Abschluss der Arbeit ist eine Suche entstanden, bei welcher nach ungewöhnlichen 3D-Modellen gesucht werden kann, ohne dass das neuronale Netz neu trainiert werden muss. Außerdem ist die Suche einfach für den Endbenutzer anwendbar und auf mobilen Endgeräten ausführbar, um diesem Zeit bei der Suche von tausenden Dokumenten zu ersparen.

Inhaltsverzeichnis

Abkürzungsverzeichnis	iv
Abbildungsverzeichnis	vi
Tabellenverzeichnis	viii
Formelverzeichnis	ix
1 Einleitung	1
1.1 Problemstellung	1
1.2 Zielsetzung	2
1.3 Methodisches Vorgehen	2
1.4 Aufbau der Arbeit	3
2 Repräsentationen dreidimensionaler Daten	5
2.1 Voxel	5
2.2 Mesh	6
2.3 Point Cloud	7
2.4 Implizit	7
3 Dataset	9
3.1 ShapeNet	11
4 Methoden	15
4.1 Neuronale Netze	15
4.2 Kostenfunktionen	18
4.3 Aktivierungsfunktionen	20
4.4 Modell Optimierungen	23
4.4.1 Gewichtsinitialisierung	23
4.4.2 Batch Normalisierung	25
4.4.3 Regularisierung	26
4.4.4 Optimierung	27
4.5 Convolutional Neural Networks	28
4.6 Residual Neural Networks	30
4.7 Transfer Learning	31

4.8 Metriken	32
4.8.1 Accuracy	33
4.8.2 Intersection over Union	33
4.9 Ähnlichkeitsmaß	33
5 Verwandte Arbeiten	35
6 Modellierungsherausforderungen	41
6.1 Objekt Rekonstruktion aus einem Bild	41
6.2 Mobile Endgeräte	43
7 Realisierung	46
7.1 Trainingssetup	47
7.2 Datenvorverarbeitung	48
7.3 Baseline	49
7.4 MobileNetV2	61
7.5 IM-NET	66
7.6 Ähnlichkeitsmaß	69
7.7 Realisierung für mobile Geräte	70
8 Experimente	72
8.1 Vorhersagen differierender Eingaben	72
8.2 Genauigkeit der Deep Neural Networks	77
8.3 Zeitlicher Aspekt: Inferenz	79
8.4 Zeitlicher Aspekt: Ähnlichkeitsmaß	80
8.5 Schwellwertbestimmung des Ähnlichkeitsmaßes	81
9 Schlussfolgerung und Ausblick	83
Literaturverzeichnis	86
A Anhang	i
A.1 Weitere Beispiele für Vorhersagen differierender Eingabe aller neuronaler Netze	i

Abkürzungsverzeichnis

Adam	Adaptive Moment Estimation
API	Application Programming Interface
CAD	Computer-Aided Design
CDP	Content Delivery Plattform
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DNN	Deep Neural Network
FFNN	Feedforward Neural Network
GAN	Generative Adversarial Network
GNN	Graph Convolutional Network
GPU	Graphics Processing Unit
IDCT	Inverse Discrete Cosine Transform
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IoU	Intersection over Union
LSTM	Long Short-Term Memory
MISE	Multiresolution IsoSurface Extraction
MLP	Multi Layer Perceptron
MSE	Mean Squared Error
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
RMSProp	Root Mean Square Propagation
RNN	Recurrent Neural Network
RPN	Region Proposal Network

SDF Signed Distance Function

TFLite TensorFlow Lite

VAE Variational Autoencoder

Abbildungsverzeichnis

2.1	Darstellung einer Voxel Repräsentation einer Bank mit einer Diskretisierung des Ausgaberaums.	6
2.2	Darstellung einer Mesh Repräsentation einer Bank mit einer Diskretisierung des Ausgaberaums.	7
2.3	Darstellung einer Point Cloud Repräsentation einer Bank mit einer Diskretisierung des Ausgaberaums.	8
2.4	Darstellung einer impliziten Repräsentation einer Bank mit einer Diskretisierung des Ausgaberaums.	8
3.1	Darstellung eines 3D-Voxel-Modells eines Autos mit in Relation stehenden 2D-Bildern aus dem ShapeNet Core Datasets [Cha+15], welche von Choy et al. [Cho+16] aufbereitet wurden.	14
4.1	Schematische Darstellung eines Prozesses zur Anpassung von Gewichtungen in einem Perzepron.	17
4.2	Graphen der Aktivierungsfunktionen (a) Sigmoid, (b) ReLU, (c) Leaky-ReLU und (d) Softmax.	23
4.3	Abbildung der Verarbeitung von 2D-Faltungen in CNN.	30
4.4	Abbildung der Verarbeitung von 3D-Faltungen in CNN.	30
4.5	Darstellung des Residual Learnings.	31
7.1	Darstellung der Encoder-Decoder Architektur für alle implementierten neuronalen Netze.	46
7.2	Darstellung eines High-Level-Graphen der Architektur des selbst erstellten Encoders.	56
7.3	Darstellung eines High-Level-Graphen der Architektur des selbst erstellten Decoders.	60
8.1	Darstellung eines Schranks aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	73
8.2	Darstellung eines Tisches aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	74

8.3	Darstellung eines komplexen Stuhls aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	75
8.4	Darstellung eines Autos als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	75
8.5	Darstellung einer Schraube als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	76
8.6	Darstellung eines nicht-synthetischen Flugzeug als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	77
A.1	Darstellung eines Autos aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	i
A.2	Darstellung eines Stuhls aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	ii
A.3	Darstellung einer synthetischen Eingabe einer Gießkanne als Bild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	ii
A.4	Darstellung einer synthetischen Eingabe einer Tasse als Bild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	iii
A.5	Darstellung eines nicht-synthetischen Autos als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	iii
A.6	Darstellung eines weiteren nicht-synthetischen Autos als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	iv
A.7	Darstellung eines nicht-synthetischen, abgeschnittenen Stuhl als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.	iv

Tabellenverzeichnis

3.1	Vergleich populärer Datasets für die Verarbeitung von 3D-Objekten.	11
3.2	Aufstellung der einzelnen Kategorien, mit der jeweiligen Anzahl an Objekten vom ShapeNet Dataset, welches von Choy et al. [Cho+16] aufbereitet wurde.	13
7.1	Zusammenstellung aller Schichten, deren Ausgabeform sowie Parameteranzahlen des neuronalen Netzes des Encoders.	55
7.2	Zusammenstellung aller Schichten, deren Ausgabeform sowie Parameteranzahlen des neuronalen Netzes des Decoders.	58
7.3	Detaillierte Struktur der Bottleneck Blöcke mit Transformation von k zu k' Channel, Stride s und dem Erweiterungsfaktor t	66
7.4	Überblick der gesamten Architektur von MobileNetV2. Dabei beschreibt jede Zeile eine oder identische Schichten, welche n -mal wiederholt werden. Alle Schichten einer Zeile haben die gleiche Größe c der Ausgabe. Zusätzlich besitzt die erste Schicht einen Stride s , alle andere Schichten besitzen einen Strike von 1. Der Erweiterungsfaktor t wird auf die Eingabegröße, beschrieben in Tabelle 7.3, angewendet.	66
7.5	Zusammenstellung aller Schichten, deren Ausgabeform sowie Parameteranzahlen des neuronalen Netzes des IM-NETs.	68
8.1	Genauigkeit verschiedener neuronalen Netze angegeben in IoU und Accuracy.	78
8.2	Zeitliche Angaben der Inferenz aller neuronaler Netze auf einem PC sowie mobilen Gerät mit (m.) und ohne (o.) laden des Modells.	80
8.3	Parameteranzahlen der Baseline, MobileNetV2 sowie des IM-NETs, aufgeteilt in En- und Decoder.	80
8.4	Zeitliche Angaben für die Ausführung des Ähnlichkeitsmaßes auf einem PC sowie mobilen Gerät mit 100 Modellen der Auflösung 32^3 (32) sowie 256^3 (256).	81
8.5	Schwellwertbestimmung der Ähnlichkeitsmaße IoU in Prozent und Hausdorffer Distanz für Modelle mit einer Auflösung von 32^3 und 256^3	82

Formelverzeichnis

4.1	Delta-Regel zur Anpassung der Gewichte eines Neurons.	16
4.2	Grundlegende Formel der Binary Cross Entropy.	19
4.3	Formel der Cross Entropy.	19
4.4	Detaillierte Formel der Binary Cross Entropy.	20
4.5	Detaillierte Formel des Mean Squared Error (MSE).	20
4.6	Formel der Kostenfunktion Sigmoid.	21
4.7	Formel der Kostenfunktion ReLU.	21
4.8	Formel der Kostenfunktion LeakyReLU.	22
4.9	Formel der Kostenfunktion Softmax.	22
4.10	Formel des Algorithmus der Batch Normalisierung.	26
4.11	Gleichungen des Adam-Optimierungs Algorithmus.	28
4.12	Formel der Metrik Accuracy.	33
4.13	Formel der Metrik Intersection over Union.	33
7.1	Formel der voxelweisen Cross Entropy Kostenfunktion.	49
7.2	Ausdruck des impliziten Feldes $F(p)$	67
7.3	Formel der gewichteten MSE Kostenfunktion.	68
7.4	Formel des Ähnlichkeitsmaß Hausdorffer Distanz.	70
7.5	Ergänzung für die Formel des Ähnlichkeitsmaß Hausdorffer Distanz.	70

1 Einleitung

Durch die stetige Digitalisierung der letzten Jahrzehnte, wurden immer mehr Smart Devices bzw. Anwendungen entwickelt. Diese sollen die Arbeitswelt wie auch den Alltag erleichtern. Repräsentanten für Smart Devices sind Smartphones, Tablets und Smartwatches. Mittlerweile ist die Digitalisierung vor allem für Unternehmen ein entscheidender Faktor, um zukunfts- und wettbewerbsfähig zu bleiben. Dies bringt den Vorteil, dass Gegenstände effizient genutzt und Arbeitsabläufe automatisiert werden können. Beispiele dafür sind digitale Dokumentationen von Maschinenstillständen bzw. Fehlererfassung sowie Echtzeitinformationen über Maschinen zur Produktionssteuerung zu erhalten [Roe+19].

In dieser Masterarbeit wird eine komplexe Funktion einer Anwendung, genannt Content Delivery Plattform (CDP), umgesetzt. CDP ist eine intelligente Software-Lösung für professionelles Informationsmanagement. Mit dieser Applikation haben mehrere Arten von Nutzern die Möglichkeit, alle relevanten und kontextspezifischen Informationen mobil abzurufen [SCH20]. Der größte Vorteil bei dieser Art der Informationsverteilung ist, dass nur Informationen angezeigt werden, welche wirklich relevant für den Nutzer sind. Vor allem wenn eine Datenbank mit vielen Einträgen existiert, wird der Vorteil deutlich sichtbar. Möchte der Nutzer einen bestimmten Datensatz aus der Datenbank finden, muss dieser sich durch alle Datensätze arbeiten. Eine Eingrenzung durch eine textuelle Suche ist hierbei wertvoll. Doch oftmals sind anschließend immer noch zu viele Datensätze für eine manuelle Suche vorhanden. Eine Möglichkeit diese nochmals zu mindern sind Filterungen. Diese sind hilfreich, können jedoch auch komplex, aufwändig und zeitintensiv werden, da sie je nach Gebrauch mehr oder weniger konfiguriert werden können. Für dieses Problem soll eine intelligente Informationsfilterung, auf Basis einer visuellen Bauteilerkennung, entwickelt werden. Dadurch soll die Suche für den Nutzer deutlich vereinfacht und effizienter werden.

1.1 Problemstellung

In dieser Masterarbeit wird eine intelligente Informationsfilterung für mobile Endgeräte realisiert. Dabei erhält die Anwendung ein visuelles Signal, in diesem Fall ein 2D-Bild, welches in der Praxis durch die Kamera des mobilen Endgerätes aufgenommen wird.

Durch ein einzelnes Bild wird ein bestimmtes, in Relation zum Bild stehendes, 3D-Objekt generiert und mit einer Sammlung von vielen 3D-Objekten verglichen. Das am besten übereinstimmende 3D-Objekt ist das Ziel des Nutzers.

Die Schwierigkeit liegt dabei in der begrenzten Ansicht des Objektes auf dem Bild. Da nur ein einziges Bild genutzt werden kann, sind viele Ansichten des Objektes verdeckt. Somit kann nicht gesagt werden, welche Form das Objekt aus anderen Perspektiven besitzt. Daraus ein 3D-Objekt zu generieren ist nicht trivial. Das 3D-Objekt soll alle Seiten des Objekts aus jeder Sichtweise darstellen. Dies abzubilden, ohne zu wissen wie das Objekt aussieht, ist eine enorme Herausforderung. Dieses Problem soll durch eine künstlich Intelligenz bewältigt werden. Genauer wird ein Deep Neural Network (DNN), welches zugehörig zur Familie des maschinellen Lernens ist, genutzt. Das neuronale Netz soll durch die Erfahrung bzw. das Training die Generierung von ähnlichen Objekten lernen. Dadurch kann das Gelernte auf die gesuchten Objekte transferiert werden.

1.2 Zielsetzung

Das Ziel dieser Veröffentlichung ist die Erleichterung der Suche nach individuellen Informationen in einer Vielzahl von Informationen. In zahlreichen Anwendungsfällen sind Servicetechniker die Benutzer der CDP. Diese sind vor Ort, um verschiedenste Geräte zu warten oder im Falle eines Defekts zu reparieren. Um die Handhabung zu erleichtern, soll eine Software für mobile Endgeräte entwickelt werden, welche eine visuelle Eingabe annimmt. Genauer soll ein neuronales Netz entstehen, welches als Eingabe ein einziges Bild entgegen nimmt. Dieses neuronale Netz besitzt die Aufgabe aus einem Bild das passende 3D-Objekt zu generieren, welches das Objekt selbst am besten digital widerspiegelt. Anschließend wird eine Suche über ein Ähnlichkeitsmaß mit allen 3D-Objekten der Datenbank durchgeführt. Zuletzt soll das 3D-Objekt bestimmt werden, welches dem generierten Objekt am ähnlichsten und somit auch das gesuchte Objekt vom Nutzer ist.

1.3 Methodisches Vorgehen

Um das angestrebte Ziel zu erreichen, werden folgende Schritte durchgeführt: Zuerst wird ein Dataset bestimmt, mit welchem das neuronale Netz trainiert. Dies muss bekannt

sein, bevor das neuronale Netz aufgebaut werden kann, da Informationen des Datasets wichtig für die Struktur sowie die Konfigurationen der einzelnen Schichten sind. Für die Wahl des Datasets müssen wichtige Kriterien erarbeitet und beachtet werden. Darunter fällt beispielsweise die Repräsentation der 3D-Modelle. Des Weiteren müssen die Daten aufbereitet werden, um diese nutzen zu können. Anschließend werden verwandte neuronale Netzwerkarchitekturen für die Rekonstruktion von 3D-Objekten aus einem Bild recherchiert. Aus der Recherche werden anschließend die Alleinstellungsmerkmale extrahiert, um diese zum Vorteil der eigenen Architektur nutzen zu können. Im Anschluss werden die Metriken des Systems bestimmt. Nachdem alle Vorbereitungen getroffen sind, wird das neuronale Netz selbst erstellt. Das neuronale Netz wird durch eine drei-stufige Entwicklung festgelegt. Es werden drei neuronale Netze betrachtet. Dabei ist das erste Netzwerk als Baseline zu sehen. Auf diesem wird die grobe Architektur erprobt und bestimmt. Das zweite Netzwerk baut auf der Baseline auf. Das dritte wiederum baut auf dem des zweiten auf. Dies soll eine Entwicklung eines grundlegenden neuronalen Netzes bis hin zu einem ausgereiften DNN zeigen. Außerdem werden Abstandsmaße zwischen dem resultierenden 3D-Modells und einer Sammlung von 3D-Modellen genutzt, um das am besten übereinstimmende Objekte zu finden. Zuletzt wird das Verfahren auf mobilen Endgeräten getestet.

1.4 Aufbau der Arbeit

Nach der Einleitung teilt sich diese Masterarbeit in acht weitere Kapitel auf. Die Struktur verfolgt das Ziel, den Inhalt sowie das zugrunde liegende Projekt dieser Arbeit dem Leser schrittweise näher zu bringen. Dabei werden zuerst die verschiedenen 3D-Repräsentationen vorgestellt, welche als Grundlage für die Auswahl des Datasets fungieren. Die 3D-Repräsentationen teilen sich in Voxel, Mesh, Point Cloud und implizite Repräsentationen auf. Anschließend wird der Abschnitt des Datasets eingeführt. Dabei werden verschiedene Datasets veranschaulicht und ausgewertet. Das Dataset, welches sich am besten für den vorliegenden Anwendungsfall eignet, wird darauf im Detail näher erläutert. Des Weiteren werden die zugrundeliegenden Methoden dargelegt, welche benötigt werden, um die Funktionen und Konfigurationen neuronaler Netze zu verstehen. Darunter wird eine Herleitung für einfache neuronale Netze gegeben. Außerdem wird auf Kostenfunktionen, Aktivierungsfunktionen und Modell Optimierungen eingegangen. Auch werden zwei Netzwerktypen vorgestellt: Convolutional Neural Network (CNN) und Residual Neural Network (ResNet). Da, wie bereits erwähnt, Teile der Netzwerkarchitektur verändert

werden, wird auch Transfer Learning erläutert. Zuletzt werden in diesem Abschnitt die Metriken sowie das Ähnlichkeitsmaß dargestellt. Weiterhin werden die Ergebnisse der Recherchen der verwandten Arbeiten veranschaulicht. Danach werden die Modellierungs-herausforderungen, welche in der 3D-Rekonstruktion aus einem Bild als auch auf mobilen Endgeräten besteht, präsentiert. Daraufhin wird die Realisierung der neuronalen Netze gegeben. Darunter sind auch das Ähnlichkeitsmaß und die Realisierung für mobile Geräte zu finden. Im Anschluss werden verschiedene Experimente und deren Interpretationen beschrieben. Zuletzt wird ein Fazit sowie ein Ausblick gegeben.

2 Repräsentationen dreidimensionaler Daten

Im folgendem Kapitel werden die gängigsten vier Repräsentationen dreidimensionaler (3D)-Daten für den Fachbereich der 3D-Objekt Generation bzw. Darstellung aufgezeigt: Die Voxel, Mesh, Point Cloud und implizite Repräsentation. Zu jeder Art der Darstellung wird eine Grafik mit dem selben 3D-Objekt, einer Parkbank, angezeigt. Dies soll die unterschiedliche Auflösung, Detailgrad wie auch das Aussehen selbst aufzeigen.

2.1 Voxel

Eine Voxel Darstellung ist eine Analogie von 2D-Pixel auf den 3D-Fall [Mes+19]. Jedes Voxel besitzt eine Höhe, Breite und Tiefe sowie eine Farbe und stellt einen einzelnen Datenpunkt dar. Außerdem können diese durch Koordinaten in einem dreidimensionalen Gitter lokalisiert werden. Sowohl CT- als auch MRT-Scanner nutzen diese Repräsentation für 3D-Daten. [SZ18]

Es existieren verschiedene Arten von Voxel, eine Auswahl davon sind beispielsweise [HLB19]:

- Binäre Belegungsraster: Dabei werden Voxel auf eins gesetzt, wenn diese zum relevanten 3D-Modell zugehörig sind und auf null wenn diese irrelevant sind.
- Probabilistisches Belegungsraster: Jedes Voxel in einem probabilistischen Belegungsraster kodiert seine Zugehörigkeitswahrscheinlichkeit zum entscheidenden Objekt.
- Vorzeichenbehaftete Abstandsfunktion, engl. *Signed Distance Function (SDF)*: Hierbei kodiert jedes Voxel seinen vorzeichenbehafteten Abstand zum nächstgelegenen Oberflächenpunkt. Der Wert ist negativ, wenn sich das Voxel innerhalb des Objekts befindet, andernfalls positiv.

Im weiteren Verlauf der Arbeit, wird ausschließlich das binäre Belegungsraster genutzt.

Voxel Modelle haben viele positive wie auch negative Eigenschaften. Einerseits können die Modelle einfach verarbeitet werden, dies wird später genauer behandelt. Andererseits steigt der Speicherbedarf der Modelle entsprechend der Auflösung. Die Seitenverhältnisse

der Modelle sind immer gleich. Hat das Modell eine Länge von 32, besitzt dieses auch die Höhe bzw. Tiefe von 32. So ergibt sich ein Größe von $32 \times 32 \times 32$. Dies bedeutet, dass die Größe eines Modells exponentiell nach (O^3) steigt. Durch diesen Wachstum sind komplexe Implementierungen bzw. deren genutzte Modelle limitiert auf relativ niedrige 256^3 Voxelraster [Mes+19]. Außerdem sind Modelle mit niedrigen Größen nicht detailliert und wirken sehr grob, wie in Grafik 2.1 ersichtlich wird. Modelle der CDP besitzen eine hohen Detailgrad. Dadurch wäre eine hohe Auflösung der 3D-Modelle als Voxel Darstellung nötig.

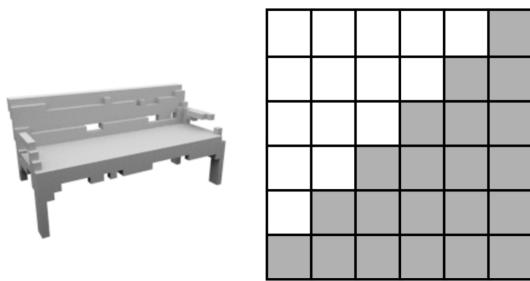


Abbildung 2.1: Darstellung einer Voxel Repräsentation einer Bank mit der Diskretisierung des Ausgaberaums. Darstellung entnommen von [Mes+19].

2.2 Mesh

Die Mesh-Darstellung eines 3D-Modells besteht aus vielen, einzelnen Dreiecken. Diese sind eine Sammlung von Eckpunkten, Kanten und Flächen. Wobei die Eckpunkte mit Kanten in Verbindung stehen und diese kombiniert eine Fläche bilden. Somit besitzen Meshes eine explizite Verbindungsbeziehung und können dadurch Strukturen deutlich zeigen. Ein Vorteil dieser Repräsentation ist die glatte Oberfläche. So ist das 3D-Modell klar zu erkennen und verliert weniger natürliche Informationen. Nachteile sind die unregelmäßigen und komplexen Elemente, welche aus mehreren Komponenten und einer unterschiedlichen Anzahl dieser Komponenten bestehen. [Fen+18]

In Abbildung 2.2 wird eine beispielhafte Darstellung eines generierten 3D-Modells von [Mes+19] als Mesh Darstellung aufgezeigt. Im rechten Teil des Bildes ist der Ausgaberaum

abgebildet. Dabei soll vor allem die Beziehung zwischen den einzelnen Komponenten hervorgehoben werden.

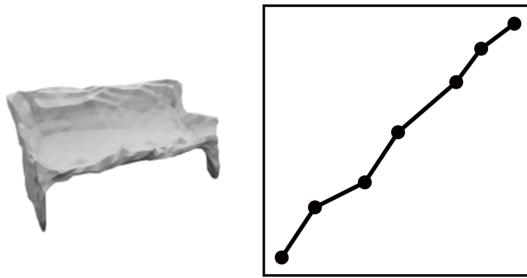


Abbildung 2.2: Darstellung einer Mesh Repräsentation einer Bank mit der Diskretisierung des Ausgaberaums. Darstellung entnommen von [Mes+19].

2.3 Point Cloud

Ein 3D-Modell kann durch ein unsortiertes Set $S = (x_i, y_i, z_i)_{i=1}^N$ von N Punkten dargestellt werden, so genannte Point Clouds [HLB19]. Sie sind vor allem für komplexe und detaillierte Objekte geeignet. Die Point Cloud Repräsentation wurde entwickelt, um zeitintensive Generationen von Meshes zu vermeiden [Lin01]. Außerdem sind diese sehr simpel und effizient im Bezug auf den Speicherbedarf. Die größte Herausforderung ist hierbei, dass die Punkte nicht in einer gitterförmigen Struktur vorliegen. Zwischen den einzelnen Punkten fehlt eine Konnektivitätsstruktur des zu Grunde liegenden Objekts. Resultierend werden zusätzliche Nachbearbeitungsschritte benötigt, um die 3D-Geometrie aus dem Modell zu extrahieren. [Mes+19]

In Grafik 2.3 (links) wird eine Bank in Form von einer Point Cloud dargestellt. Rechts wird eine Diskretisierung des Ausgaberaums aufgezeigt, dabei wird ersichtlich dass die einzelnen Punkte nicht zusammenhängend und frei von allen anderen Punkten existieren.

2.4 Implizit

Ein implizites Feld ist durch eine kontinuierliche Funktion über den 3D-Raum definiert. Die Funktion ordnet jedem Punkt (x, y, z) einen Wert zu [CZ19a]. Diese Art der Re-

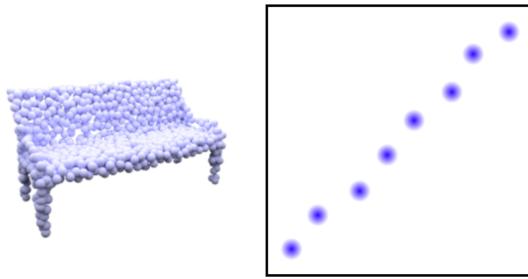


Abbildung 2.3: Darstellung einer Point Cloud Repräsentation einer Bank mit der Diskretisierung des Ausgaberaums. Darstellung entnommen von [Mes+19].

präsentation klassifiziert die bereits erwähnten Punkte als innerhalb oder außerhalb der 3D-Form. Dies wird auch als ein binäres Klassifikationsproblem bezeichnet [Fu+21]. Die 3D-Darstellung wird durch alle Punkte selbst repräsentiert.

Der größte Vorteil impliziter Repräsentationen, gegenüber den bereits erwähnten Arten, ist, dass die komplette Belegungsfunktion f_θ mit einem neuronalen Netz vorausgesagt wird. Im Gegensatz dazu, wird beispielsweise bei einer Voxel Darstellung eine feste Auflösung vorgegeben. Durch die Belegungsfunktion f_θ kann eine beliebige Auflösung ausgewertet werden. Dadurch wird der Speicherbedarf beim Training massiv reduziert. [Mes+19]

Abbildung 2.4 zeigt ein durch ein tiefes neuronales Netz erstelltes 3D-Objekt von [Mes+19]. Das Mesh des 3D-Objekts, wurde, durch einen Multiresolution IsoSurface Extraction (MISE) Algorithmus angewandt auf die Belegungsfunktion f_θ , generiert.

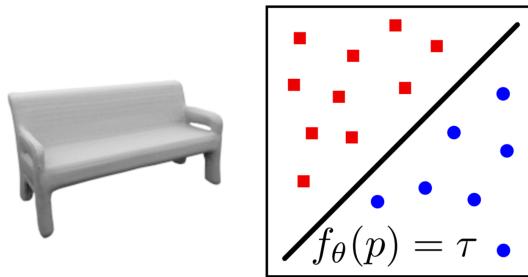


Abbildung 2.4: Darstellung einer Impliziten Repräsentation einer Bank mit der Diskretisierung des Ausgaberaums. Darstellung entnommen von [Mes+19].

3 Dataset

Ein Dataset ist eine Sammlung von Daten, welche in einem bestimmten Zusammenhang oder für einen bestimmten Zweck existieren. Im Bereich des Deep Learnings, welcher Schwerpunkt der vorliegenden Arbeit ist, bilden Daten die Grundlage jedes Deep Learning Systems. Ohne Daten können Modelle nicht lernen. Das heißt, dass diese nicht trainiert und nicht evaluiert werden können. [DN19, S. 40]

Datasets können in vielen unterschiedlichen Formen und Arten vorliegen. Beispiele dafür sind Datenbanken bzw. Tabellen oder als Dateien selbst. Als Form des Datasets, welches in dieser Arbeit benötigt wird, soll ein 2D-Bild als Eingabe gegeben sein. Als in Relation stehende Ausgabe bzw. Label soll ein 3D-Modell vorhanden sein. Welche 3D-Repräsentation dieses 3D-Modell haben soll, wurde offen gelassen, um eine größere Auswahl schaffen zu können. Zuletzt sollten die Eingabebilder verschiedene Ansichten aus unterschiedlichen Perspektiven des 3D-Modells darstellen. Dadurch soll ermöglicht werden, dass ein neuronale Netz ein Objekt aus verschiedenen Betrachtungsmöglichkeiten erkennt und somit auch rekonstruieren kann.

Für den vorliegenden Anwendungsfall wurde entschieden ein vorgefertigtes Dataset zu nutzen. Diese werden frei verfügbar von Firmen, Staaten, Kommunen, Institutionen oder Communitys im Internet zum Herunterladen bereitgestellt [DN19, S. 41]. Diese Entscheidung bringt mehrere Vorteile mit sich. Einerseits muss kein eigenes Dataset von Grund auf erstellt werden. Dies hätte eine lange Zeit in Anspruch genommen sowie viel Arbeit, die passenden Daten zu finden. Andererseits sind diese Datasets bereits teilweise vorverarbeitet. Damit ist gemeint, dass die Daten schon strukturiert, auf fehlende Einträge überprüft und bereinigt sind [DN19, S. 41]. So können die Daten einfach übernommen werden.

An vorgefertigten Datasets stehen folgende zur Auswahl, diese werden noch genauer erläutert:

- ABC [Koc+19]
- ModelNet [Zhi+15]
- Pascal 3D+ [XMS14]

- IKEA [LPT13]
- Pix3D [Sun+18]
- ShapeNet [Cha+15]

Das ABC Dataset ist eine Sammlung von einer Millionen Computer-Aided Design (CAD)-Modellen, welche speziell für Forschungszwecke an geometrischen Deep Learning Methoden bzw. Anwendungen erstellt wurde. Die Objekte selbst bestehen aus parametrisierten Kurven und Flächen. Das Sampling der parametrischen Beschreibungen ermöglicht die Generierung von Daten in verschiedenen Formaten und Auflösungen. Alle Objekte des Datasets bestehen fast nur aus mechanischen Teilen mit scharfen Ecken und Kanten. [Koc+19]

Da das Dataset nur 3D-Modelle enthält und so nur die Ausgabe des gewollten Datasets, müssten die passenden 2D-Bilder zusätzlich generiert werden.

Ein weiteres ist das ModelNet Dataset. Es ist ein vergleichbar großes Dataset, welches aus 660 individuellen Kategorien bzw. aus 151.128 3D-CAD-Modellen besteht. Außerdem existieren drei Untergruppen: ModelNet10, ModelNet40 sowie das Aligned 40-Class ModelNet. Der Training Split bzw. Test Split sind bei der jeweiligen Untergruppe bereits festgelegt. [Zhi+15]

Da auch hier nur die 3D-Modelle gegeben sind, müssten die in Relation stehenden 2D-Bilder zur Eingabe erstellt werden.

Pascal3D+ ist ein erweitertes Dataset auf Grundlage von PASCAL VOX 2012 [Eve+10], welches 3D-Anmerkungen nutzt. Das Dataset weist 12 Kategorien mit durchschnittlich mehr als 3000 Objekte per Kategorie auf. Es enthält 2D-Bilder als auch 3D-Objekte. Die Zuordnung zwischen Bild und 3D-Objekt ist hierbei herausfordernd. Die Eingabebilder enthalten teils mehrere Objekte, Okklusionen oder Objekte, welche nicht vollständig sichtbar sind. [XMS14]

Das nächste Dataset welches beschrieben wird, ist das IKEA Dataset. Es enthält ausschließlich Objekte, welche für die Innenausstattung eines Raums geeignet sind. Das Dataset besteht aus sechs Kategorien. Diese enthalten 800 Bilder sowie 225 3D-Modelle. Alle 800 Bilder sind mit 90 verschiedenen Modellen vollständig annotiert. Aus zwei Teilen besteht das Dataset: IKEAobject und IKEARoom. Das erstere enthält einfache Szenen mit größeren Objekten. Das Zweite besteht aus dem Gegenteil. [LPT13]

Das IKEA Dataset bietet vor allem eine akkurate Ausrichtung zwischen den 2D-Bildern und der 3D-Modelle. Der Nachteil hierbei ist, dass die Anzahl der Daten sehr gering ist. Um die Vorteile des IKEA Datasets nutzen zu können, wurde Pix3D erstellt. Es basiert auf dem IKEA Dataset und erweitert diesen. Es enthält neun Objektklassen mit 418 3D-Formen und 16.913 2D-Bildern. Jedes 3D-Objekt hat ein in einer realen Umgebung korrespondierendes 2D-Bild. Das Pix3D Dataset enthält auch nur Objekte, welche für den Innenbereich geeignet sind. [Sun+18]

Aufgrund dessen, dass das ShapeNet Dataset in dieser Arbeit verwendet wird, wird dieses in Kapitel 3.1 im Detail vorgestellt.

In Tabelle 3.1 werden alle vorgestellten Datasets aufgezeigt, mit der zugehörigen Anzahl der Kategorien, Bilder und 3D-Modellen.

	Kategorien	Bilder	3D-Modelle
ABC	1	-	1.000.000
ModelNet	660	-	151.128
Pascal3D+	12	≈ 36.000	≈ 36.000
IKEA	6	800	225
Pix3D	9	16.913	418
ShapeNet	3.135	-	3.000.000+
ShapeNet Core	55	-	51.300
ShapeNet von [Cho+16]	13	1.200.000	50.000

Tabelle 3.1: Vergleich populärer Datasets für die Verarbeitung von 3D-Objekten.

3.1 ShapeNet

Das ShapeNet Dataset besteht aus 3.000.000 3D-CAD-Modellen und besitzt keine Bilder. Von den 3D-Objekten wurden 220.000 in 3.135 Kategorien (nach WordNet Synsets [Mil95]) klassifiziert. Erstellt wurde das Dataset vor allem für datengetriebene Systeme mit 3D-Inhalten. Der Grund hierfür ist das Mängeln an großen, gepflegten Datensätzen,

welche für Forschungszwecke frei zur Verfügung stehen. Außerdem enthält das Dataset umfangreiche Mengen an Anmerkungen für jedes Modell und Beziehungen zwischen Modellen im Set sowie anderen multimedialen Daten außerhalb des Sets. Die Annotations umfassen geometrische Attribute, Teile und Keypoints, Formsymmetrien und den Maßstab des Objekts in realen Einheiten. Diese Attribute sind wichtige Ressourcen für die Verarbeitung, das Verständnis und die Visualisierung von 3D-Formen. [Cha+15]

Die 3D-Modelldaten selbst stammen aus öffentlichen Online-Repositories oder bestehenden Forschungsdatensätzen. Dafür wurden aus Trimble 3D Warehouse [Inc21] 2.3 Millionen 3D-Modelle sowie Szenen und aus Yobi3D 350.000 Objekte entnommen. Yobi3D wurde bereits gestoppt und existiert nicht mehr. Die Objekte selbst wurden so gewählt, dass sie die menschliche, alltägliche Welt widerspiegeln. So sind keine mechanischen Teile, molekulare Strukturen oder andere domänen spezifischen Modelle enthalten. Dennoch werden Szenen wie z. B. ein Büro oder einzelne Teile von Objekten, beispielsweise eine Tastatur, einbezogen. Die Kategorien von ShapeNet enthalten sehr starre und von Menschen erschaffene Objekte. Modelle wie Pflanzen und Tiere werden nicht aufgeführt. Dies liegt einerseits daran, dass natürliche Objekte mit gängiger 3D-Software schwieriger zu erstellen sind. Andererseits sind die Konsumenten des Modells in der Regel an künstlichen Objekten interessiert. [Cha+15]

Die Ersteller des ShapeNet Datasets haben mit diesem bestimmte Ziele verfolgt: Es soll eine Sammlung bzw. eine Zentralisierung von 3D-Modellen existieren, um den Aufwand für die Forschungsgemeinde zu reduzieren. Des Weiteren sollen datengesteuerte Methoden, welche 3D-Modellen benötigen, unterstützt werden. Außerdem ermöglicht dieses eine Evaluierung und einen Vergleich von Algorithmen für die verschiedensten Anwendungsgebiete. Beispiele dafür sind Segmentierung, Ausrichtung oder die Übereinstimmung von Objekten. Zuletzt soll das Dataset als Wissensbasis für die Darstellung von Objekten in der realen Welt und ihrer Semantik dienen. [Cha+15]

Eine Teilmenge des ShapeNet Datasets ist ShapeNet Core. Dieses enthält einzelne, bereinigte Inhalte und manuell verifizierte Kategorie- und Ausrichtungsannotationen. Es deckt 55 Kategorien mit 51.300 3D-Modellen ab. [Cha+15]

Es ist momentan eins der meist genutzten Datasets für die Rekonstruktion von 3D-Objekten [Fu+21].

Es existiert bereits eine Aufbereitung von Choy et al. [Cho+16] der Daten für die

Rekonstruktion von 3D-Objekten aus einem bzw. mehreren Bildern. Hierbei wurde eine Teilmenge des ShapeNet Core Datasets genutzt, welches aus etwa 50.000 Modellen sowie je Modell 24 gerenderte Bilder und 13 Hauptkategorien besteht. Die 3D-Modelle sind als Voxel Repräsentation gegeben. Eine weitere Veröffentlichung von Melesse [Mel18] hat diese Daten in ein passendes Format gebracht, welche für diese Implementierung sehr gut geeignet sind. Durch die hohe Beliebtheit des ShapeNet Core Datasets und auch dieser Aufbereitung, wurde sich für dieses in der vorliegenden Publikation entschieden. So wird der Vergleich zwischen dieser Arbeit und anderen Veröffentlichungen eindeutiger. Ein weiterer Grund dafür ist, dass die Vorverarbeitung der Daten deutlich vereinfacht wurde und sich so auf die Erstellung der Such-Pipeline selbst fokussiert werden konnte. In Tabelle 3.2 werden alle Kategorien sowie die Anzahl der Objekte in der jeweiligen Kategorie aufgezeigt. Außerdem wird in Abbildung 3.1 ein zufälliges 3D-Modell aus der Vorverarbeitung von Choy et al. als Voxel Repräsentation mit in Relation stehenden, gerenderten Bildern aus verschiedenen Winkeln dargestellt.

Kategorie	Anzahl der Objekte
Flugzeug	4.045
Bank	1.816
Schrank	1.572
Auto	7.496
Stuhl	6.778
Bildschirm	1.095
Lampe	2.318
Lautsprecher	1.618
Schusswaffe	2.372
Sofa	3.173
Tisch	8.509
Mobiltelefon	1.052
Wasserfahrzeug	1.939

Tabelle 3.2: Aufstellung der einzelnen Kategorien, mit der jeweiligen Anzahl an Objekten vom ShapeNet Dataset, welches von Choy et al. [Cho+16] aufbereitet wurde.

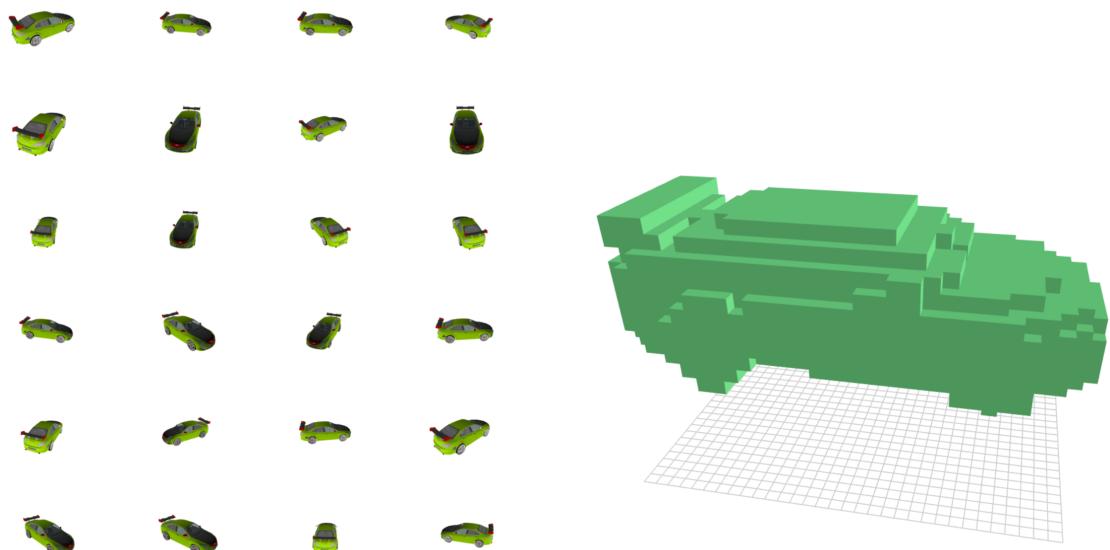


Abbildung 3.1: Darstellung eines 3D-Voxel-Modells eines Autos mit in Relation stehenden 2D-Bildern aus dem ShapeNet Core Datasets [Cha+15], welche von Choy et al. [Cho+16] aufbereitet wurden.

4 Methoden

In diesem Kapitel werden alle Methoden beschrieben, welche in der vorliegenden Arbeit eingesetzt werden. Fokus wird dabei auf die Grundlagen von Deep Learning gesetzt. Deep Learning ist ein spezielles Teilgebiet des Machine Learnings, wobei dies als Gegenstand der künstlichen Intelligenz eingeteilt ist. Deep Learning orientiert sich an der Funktionsweise des menschlichen Gehirns bzw. mit seinen miteinander verknüpften Neuronen. Diese Sinnbild wird virtuell durch künstliche Neuronen realisiert, welche in mehreren Schichten miteinander verbunden werden, um zunehmend komplexe Aufgaben zu lösen. Im Folgenden wird dies und weiteres genauer behandelt. [DN19, S. 57]

4.1 Neuronale Netze

Der grundlegendste Bestandteil eines neuronalen Netzes sind die einzelnen künstlichen Neuronen. Wie bereits in Abschnitt 4 erwähnt, wurde die Funktionsweise dieser Neuronen durch biologische Nervenzellen inspiriert. Die ersten Forscher, welche dieses Prinzip präsentierten, waren Warren McCulloch und Walter Pitts im Jahre 1943 [War43]. Daran ist zu erkennen, dass die Logik bereits vor Jahren existierte. Doch erst durch die Steigerung der Hardwareperformanz von Rechnern, ist es möglich diese Verfahren produktiv einzusetzen.

Ein künstliches Neuron, das sogenannte Perzeptron, wurde von Franck Rosenblatt 1957 [Ros57] eingeführt und baut auf dem Modell von Warren McCulloch und Walter Pitts auf. Das Perzeptron besteht aus mehreren Komponenten. Zuerst die Eingabe. Der Eingabewert ist ein n-dimensionaler Vektor X bestehend aus Werten x_i . Jeder Wert x_i ist mit einstellbaren Gewichtungen w_i versehen. Zusätzlich wird häufig ein sogenannter Bias b darauf addiert. Der Bias wird im mehrschichtigen Fall benötigt, wenn ein Neuron keine Eingänge hat, aber die Gewichte mit den darauf folgenden Neuronen verbunden sind. Die Summe α des Eingabevektors mit den jeweiligen Gewichtungen bilden den Wert, welcher für die Aktivierungsfunktion $\varphi(x)$ benötigt wird. Der daraus resultierende Wert bildet die Ausgabe y des Neurons. [DN19, S. 57–61]

Aktivierungsfunktionen können je nach Eingabe verschiedene Ergebnisse ausgeben. Dabei wird die Analogie zum biologischen Neuron sichtbar: Wenn die Eingabe einen bestimmten Schwellwert der Funktion übersteigt, wird diese aktiviert. Aktivierungsfunktionen werden

in Abschnitt 4.3 genauer erläutert.

Ein einzelnes Neuron berechnet eine lineare Funktion. Durch die Anpassung der Gewichte wird diese lineare Funktion erlernt. Die Eingabewerte werden iterativ eingespeist und die Berechnung der Ausgabewerte durchgeführt. Die resultierenden Werte werden mit den Zielwerten verglichen. Diese Prozedur wird so lange wiederholt, bis das Neuron die gewünschte Ausgabe errechnen kann, sofern dies überhaupt möglich ist. Diese komplexe Vorgehensweise wird in Abbildung 4.1 dargestellt. Zur bereits erwähnte Anpassung der Gewichte, wird die Delta-Regel verwendet: [DN19, S. 60]

$$w_{i_{new}} = w_{i_{old}} + \eta \cdot (y_i - \hat{y}_i) \cdot x_i \quad (4.1)$$

Dabei ist y_i die Soll-Ausgabe und \hat{y}_i die errechnete Ausgabe. η ist die Lernrate. Diese gibt an, wie schnell das Neuron in jedem Schritt lernen soll. Das Hauptziel dieses Prozesses ist es, die Differenz zwischen der erwartenden und der errechneten Ausgabe zu minimieren. [DN19, S. 60]

Ein neuronales Netz enthält eine mehrschichtige Zusammenstellung aus Neuronen mit gewichteten Verbindungen. Das Netz enthält dabei verschiedene Schichten [DN19, S. 61–62]:

- Die Eingabeschicht (engl. *Input Layer*): Diese besteht aus Neuronen, welche numerische Merkmalsvektoren abbilden. Jedes Neuron verkörpert ein Merkmal. Diese Merkmale können verschiedene Formen annehmen, zum Beispiel als ein Pixel eines Bildes.
- Eine oder mehrere verdeckte Schichten (engl. *Hidden Layer*): Diese Schicht bzw. Schichten sind für die Weiterverarbeitung der Eingabeschicht verantwortlich. Existieren mehr als eine verdeckte Schicht, wird von einem tiefen Netz gesprochen. Ein sogenanntes Deep Neural Network (DNN). Hierbei werden beispielsweise Filteroperationen auf Pixelebene ausgeführt.
- Die Ausgabeschicht (engl. *Output Layer*): Dabei werden die Ausgabewerte repräsentiert. Zum Beispiel gefundene Klassen bei einer Klassifikationsaufgabe, Zielwerte bei einer Regressionsaufgabe oder Clusteraufgaben.

Die simpelste Aufstellung eines neuronalen Netzes ist das Multi Layer Perceptron (MLP). Diese besitzen eine Eingabe- sowie eine Ausgabeschicht und müssen mindestens eine verdeckte Schicht besitzen. Oftmals sind diese einzelnen Neuronen einer Schicht mit allen

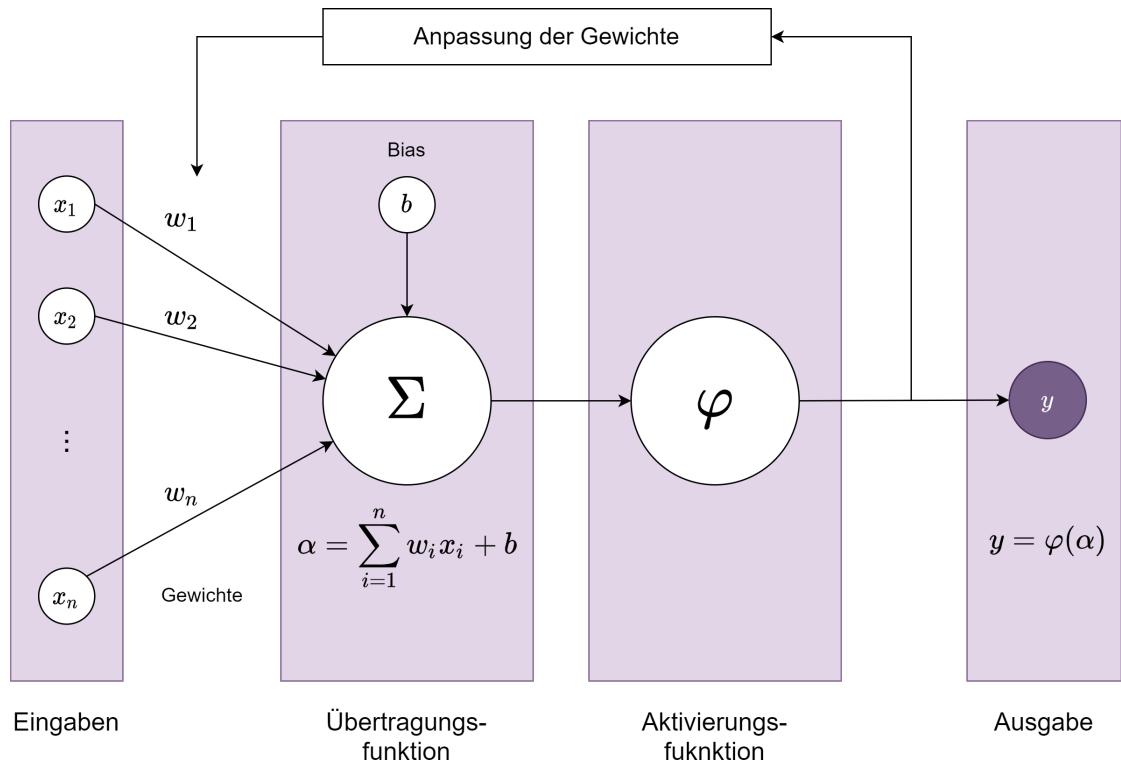


Abbildung 4.1: Schematische Darstellung eines Prozesses zur Anpassung von Gewichtungen in einem Perzeptron. Darstellung entnommen von [DN19, S. 61].

Neuronen der nächsten Schicht verbunden. Dies wird auch als Fully Connected bezeichnet. [DN19, S. 62]

Allgemein wird zwischen zwei Arten von Netzen unterschieden: vorwärtsgekoppelte Netze (engl. *Feedforward Neural Networks*) und rückgekoppelte Netze (engl. *Recurrent Neural Networks*). Ein Feedforward Neural Network (FFNN) besitzt nur eine Richtung, in der das Netz durch propagiert werden kann. Sie können ausschließlich vom Input zum Output durchlaufen werden. Das heißt, es gibt keine Verbindungen zurück zu einem Neuron oder zu einer vorherigen Schicht. Im weiteren Verlauf dieser Arbeit, in Abschnitt 4.5, wird eine typische Kategorie dieser Art von Netzen vorgestellt: das Convolutional Neural Network (CNN). Bei dem Recurrent Neural Network (RNN) hingegen sind Rückkopplungen zu ihren unmittelbaren Vorgängern oder zu anderen vorhergehenden Schichten möglich. Diese Art der Netze sind vor allem für Sequenzen geeignet, bei welchen temporäre Aspekte eine Rolle spielen. RNNs werden in dieser Arbeit nicht behandelt und entsprechend nicht

näher erläutert. [DN19, S. 62–63]

Der Lernvorgang eines neuronalen Netzes ist komplex. Bevor das Lernen möglich ist, muss erst eine Netzstruktur erstellt werden. Danach müssen Hyperparameter, wie beispielsweise die Anzahl der Epochen, festgelegt werden. Anschließend werden die Gewichtungen und der Bias zufällig initialisiert. Damit das neuronale Netz lernen kann, müssen Trainingsdaten hinzugefügt werden. Diese stellen die Eingabe des Netzes dar. Während des Trainings wird die Differenz zwischen errechneter und gewünschter Ausgabe berechnet. Diese Differenz wird für die Fehlerevaluation benötigt. Ist das Minimum dieser Fehlerevaluation und die Epochanzahl nicht erreicht, erfolgt eine Anpassung auf Basis der berechneten Differenz. Die Anpassung der Gewichte wird über den Backpropagation-Schritt vollzogen. Dabei wird von der Ausgabeschicht, über alle verdeckten Schichten bis zur Eingabeschicht zurückpropagiert. Der Backpropagation-Algorithmus verallgemeinert die Delta-Regel für Aktivierungsfunktionen und alle verdeckten Schichten. Nachdem die Gewichtungen angepasst wurden, beginnt der Trainingszyklus von Neuem. Die Anzahl der Epochen wird dabei um eins erhöht. Oft werden die Trainingsdaten in kleinen Paketen, beispielsweise die Datensätzen von 0 bis 64, 65 bis 128, usw., durch das Netz geschickt. Diese kleinen Teilmengen des Datensatzes werden auch Batches genannt. Das erstellte Netz wird nach jedem Durchlauf eines Batches aktualisiert. Ist der Algorithmus einmal durch alle Trainingsdaten bzw. Batches durchlaufen, wurde eine Epoche bewältigt. [DN19, S. 63–67]

4.2 Kostenfunktionen

Wie bereits in Kapitel 4.1 erwähnt, werden die Gewichte eines neuronalen Netzes infolge des Fehlers während eines Trainingszyklus kontinuierlich angepasst. Der Fehler selbst wird durch eine Kostenfunktion, engl. *Loss Function*, berechnet. Allgemein drückt das Ergebnis der Kostenfunktion die Qualität der Abbildung zwischen Eingabe und Ausgabe des Netzes aus. Außerdem muss diese Differenzierbar sein, da diese für das Gradientenabstiegsverfahren der Backpropagation genutzt wird. Der Wert der Kostenfunktion ist besser, umso niedriger dieser ist. Ein niedriger Loss zeigt eine gute Vorhersage an. Ziel ist es alle Parameter so zu wählen, dass der Loss minimiert wird. [DN19, S. 66–67]

Im Folgenden werden alle grundlegenden Kostenfunktionen erläutert, welche für diese

Arbeit relevant sind. Folgende Kostenfunktionen werden anschließend detailliert beschrieben:

- Binary Cross Entropy
- Mean Squared Error

Binary Cross Entropy

Wie bereits in Abschnitt 2.1 erwähnt, werden in dieser Arbeit binäre Belegungsraster für die Voxel Repräsentation genutzt. Die Belegung eines jeweiligen Voxel beschränkt sich dabei auf Eins oder Null. Dies kann als $y \in \{1, 0\}$ beschrieben werden, wobei y einzelne Voxel darstellt. Diese Zuordnung wird als Klassifikationsproblem gesehen. Entsprechend wird eine Kostenfunktion benötigt, welche für eine binäre Klassifikation geeignet ist. Für diesen Fall wird die Binary Cross Entropy genutzt.

Die grundlegende Formel der Binary Cross Entropy ist definiert als

$$C = \frac{1}{M} \sum_{i=1}^M Cost(h(x_i), y_i) \quad (4.2)$$

Dabei steht M für die Summe aller Beispiele, x_i für einzelne Eingaben, y_i für einzelne Ausgaben und h für die Hypothese des übergebenen Eingabewertes, wobei dies der errechneten Ausgabe gleichkommt. [Mur12, S. 56–57]

Da bei der Binary Cross Entropy zwei Fälle von Bedeutung sind, bestehen diese auch aus zwei Termen:

- $-\log(h(x))$, wenn $y=1$
- $-\log(1 - h(x))$, wenn $y=0$

Mit diesen sollen Werte, welche missklassifiziert werden, höher durch den Logarithmus bestraft werden. Da die gezeigten Terme immer nur Teil einer Klasse sein können, müssen diese zusammengefügt werden. Dies wird durch weitere Ausdrücke möglich, so werden aus den bereits gezeigten Formeln:

$$Cost(h(x_i), y_i) = -y\log(h(x)) - (1 - y)\log(1 - h(x)) \quad (4.3)$$

Mit der Basis der Binary Cross Entropy kombiniert, ergibt sich eine endgültige Kostenfunktion von: [Mur12, S. 56–57]

$$C = -\frac{1}{M} \sum_{i=1}^M y_i \log(h(x_i)) + (1 - y_i) \log(1 - h(x_i)) \quad (4.4)$$

Mean Squared Error

Der Mean Squared Error (MSE) ist auch als L_2 -Loss bekannt. Er stellt die quadrierte Differenz zwischen vorhergesagten Werten und den Zielwerten dar. Dabei werden durch die Quadrierung hohe Distanzen exorbitant bestraft. Die Kostenfunktion wird in Formel 7.3 dargestellt. Dabei ist m die Anzahl aller Merkmale, \hat{y} der vorhergesagte Wert und y der Zielwert. [Gér19, S. 113–114]

$$C = \frac{1}{m} \sum_{i=1}^m (\hat{y} - y)^2 \quad (4.5)$$

4.3 Aktivierungsfunktionen

Die Aktivierungsfunktion wird auf die gewichtete Summe eines einzelnen künstlichen Neurons angewendet. Die Aktivierungsfunktion der ersten neuronalen Netzen war eine Stufenfunktion bzw. Schwellwertfunktion. Ist der Eingabewert niedriger als der Schwellwert, wird eine Null zurück gegeben, bei einem Wert größer als der Schwellwert eine Eins. Diese Funktion ist sehr einfach aufgebaut, jedoch ist sie nicht stetig differenzierbar. So kann keine Ableitungsfunktion bestimmt werden, welche die Steigung angibt. Dies ist jedoch zum Lernen des neuronalen Netzes eine notwendige Voraussetzung, um die Backpropagation durchführen zu können. [NZ18, S. 149–150]

Im weiteren Verlauf dieses Abschnitts werden alle Aktivierungsfunktionen, welche in dieser Arbeit gebraucht werden, genauer erläutert. Diese sind:

- Sigmoid
- ReLU
- LeakyReLU
- Softmax

Sigmoid

Durch den wesentlichen Nachteil der Stufenfunktion, wurde eine weitere Aktivierungsfunktion benutzt, welche dieser nahe kommt, aber differenzierbar ist. Diese Funktion wird Sigmoid genannt und ist in Abbildung 4.2 (a) dargestellt. Im Grunde ist dies eine verschobene und skalierte Tangens-Hyperbolicus-Funktion, für gewöhnlich wird aber die Form über die Exponentialfunktion genutzt. Außerdem besitzt diese einen Wertebereich von $[0; 1]$ und eine probabilistische Ausgabe bei einem binären Klassensystem. Ein Nachteil dabei ist, dass eine Saturierung an den Enden vorhanden ist. Somit werden verschwindende Gradienten (engl. *Vanishing Gradient*) möglich. Dies hat zur Folge, dass das neuronale Netz bzw. Teile davon nicht mehr lernfähig sind. Der Term der Funktion ist wie folgt definiert: [NZ18, S. 150]

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad (4.6)$$

ReLU

Im praktischen Einsatz von neuronalen Netzen, hat sich die Rectified Linear Unit (ReLU) Aktivierungsfunktion durchgesetzt. Diese wird in Abbildung 4.2 (b) gezeigt. Durch Experimente hat sich herausgestellt, dass diese Aktivierungsfunktion besonders schnell konvergiert und weniger Berechnungen als bei der Sigmoid Funktion benötigt. Bei dieser Funktion werden Werte unter einem bestimmten Schwellwert ignoriert und als Null zurück gegeben. Werte über diesem Schwellwert, werden als die Werte selbst zurück gegeben. Die ReLU Funktion vermeidet außerdem das Problem des Vanishing Gradients, da hier keine Saturierung stattfindet. Der Wertebereich erstreckt sich von $[0, \infty]$. Ein Nachteil dieser Funktion besteht aus den toten Neuronen, welche durch den Null-Wert entstehen. Zusätzlich ist die Aktivierungsfunktion nicht um Null zentriert. Außerdem existieren hierbei keine Wahrscheinlichkeiten, somit wird diese Funktion nur in den verdeckten Schichten verwendet. [NZ18, S. 162]

Die Funktion selbst wird beschrieben als [MHN13]:

$$\text{ReLU}(x) = \begin{cases} 0, & \text{wenn } x \leq 0 \\ 1, & \text{wenn } x > 0 \end{cases} \quad (4.7)$$

LeakyReLU

Die LeakyReLU Aktivierungsfunktion ist eine, geglättete Version der ReLU Aktivierungsfunktion. Die Funktion wird in Abbildung 4.2 (c) gezeigt. Die LeakyReLU Funktion besitzt einen Wertebereich von $[-\infty, \infty]$. Dabei werden alle Eingabewerte die unter einem Schwellwert liegen nicht auf Null gesetzt, sondern mit der Konstante α multipliziert. Diese Konstante wird oft auf 0.1 gesetzt, dadurch wird der negativen Wert abgeflacht. Durch diesen Wertebereich entsteht das Problem des explodierenden Gradienten (engl. *Exploding Gradients*), dadurch könnten lokale bzw. globale Minimums übersprungen werden. Trotz dessen ist das Ziel des negativen Wertebereiches, dass tote Neuronen vermieden werden. Die Aktivierungsfunktion wird als folgende Formel definiert: [MHN13]

$$\text{LeakyReLU}(x) = \begin{cases} 0, & \text{wenn } x \leq 0 \\ \alpha x, & \text{wenn } x > 0 \end{cases} \quad (4.8)$$

Softmax

Die Softmax Aktivierungsfunktion, auch normalisierte Exponentialfunktion genannt, wird folgendermaßen dargestellt:

$$\text{Softmax}(x_k) = \frac{e^{x_k}}{\sum_{j=1}^K e^{x_j}} \quad (4.9)$$

Diese Funktion wandelt die Eingaben in Wahrscheinlichkeiten um. Dabei werden die Exponenten jedes Wertes ermittelt und durch die Summe aller Exponenten normalisiert. Die Summe der Exponenten ergibt aufaddiert Eins. Außerdem ist das Ergebnis einer Softmax Aktivierungsfunktion immer größer oder gleich Null. Der Graph der Funktion ist in Abbildung 4.2 (d) zu sehen.

Diese Aktivierungsfunktion wird häufig als letzte Schicht eines neuronalen Netzes für Klassifikationsaufgaben genutzt. Dadurch können die Wahrscheinlichkeiten für die jeweilige Klasse angegeben werden. [NZ18, S. 154–155]

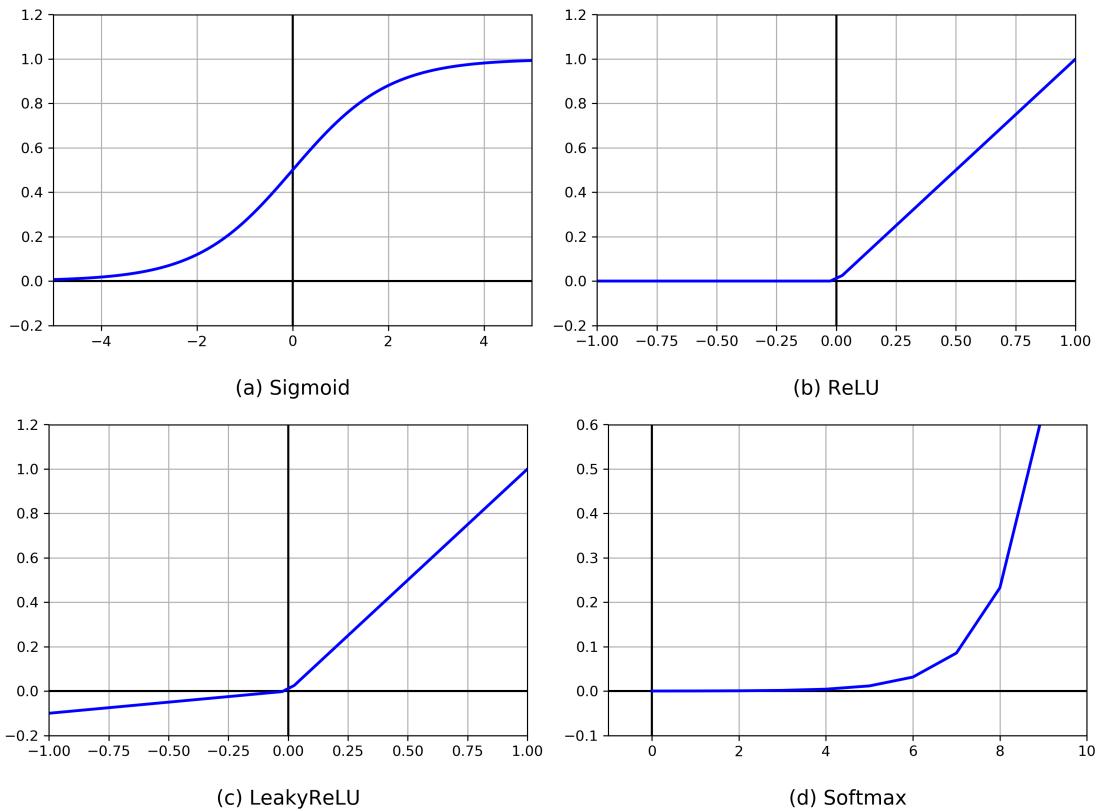


Abbildung 4.2: Graphen der Aktivierungsfunktionen (a) Sigmoid, (b) ReLU, (c) LeakyReLU und (d) Softmax.

4.4 Modell Optimierungen

Im Folgenden werden alle Möglichkeiten, welche in dieser Arbeit zur Optimierung des neuronalen Netzes bzw. des Modell dieses Netzes beitragen, aufgezeigt. Dazu werden diese Arten der Optimierung genutzt: Gewichtsinitialisierungen, Regularisierung, Optimierung sowie die Normalisierung.

4.4.1 Gewichtsinitialisierung

Das Ziel eines neuronalen Netzes ist es, die Gewichte so zu erlernen bzw. zu trainieren, das durch dieses ein optimales Ergebnis geliefert werden kann.

In der ersten Epoche, welche das neuronale Netz durchläuft, müssen diese Gewichte erst initialisiert werden. Dazu existieren mehrere Möglichkeiten. Eine Initialisierung

mit 0 ist nicht möglich, denn so besteht kein Gradient und resultierend kann nichts erlernt werden. Initialisiert man die Gewichte mit der selben Konstante, fallen alle Neuronen der verdeckten Schichten zu einem einzelnen Neuron zusammen. Somit wird das Lernen des Netzes sehr stark begrenzt und führt zu keinem großen Effekt. Eine Initialisierung mit zufälligen Werten führt dabei zu einem besseren Ergebnis. Dabei werden die Werte in einem bestimmten Intervall, beispielsweise im Intervall von $[-0.01; +0.01]$, zufällig generiert. Dabei können jedoch auch Nachteile entstehen. Das Signal kann so verschwinden oder auch explodieren bzw. saturieren. Um solche Probleme zu vermeiden, muss sichergestellt werden, dass die Varianz einer Ausgabe einer Schicht gleich der Varianz der Eingabe ist. Zusätzlich muss auch die Varianz der Gradienten vor und nach dem Durchlaufen einer Schicht in beide Richtungen gleich sein. Beide Fähigkeiten zu vereinen kann nur garantiert werden, wenn die Schichten die gleiche Anzahl von Eingaben und Neuronen haben. Dennoch wurden Möglichkeiten geschaffen, einen Kompromiss dafür zu erstellen. Darunter fallen die Glorot-Initialisierung sowie die He-Initialisierung. Diese werden im Folgenden genauer erläutert. [Gér19, S. 332–335]

Glorot-Initialisierung

Xavier Glorot und Yoshua Bengio [GB10] haben eine Arbeit veröffentlicht, bei welcher eine zufällige, heuristische Initialisierung der Gewichte aus einer Normalverteilung von $[-1; 1]$ mit anschließender Skalierung durch $\frac{1}{\sqrt{n}}$ genutzt wird. Wobei n die Anzahl der Neuronen der vorherigen Schicht darstellt. Dabei stellte sich heraus, dass diese Verteilung, wie bereits in Abschnitt 4.4.1 erwähnt, ungeeignet ist. Glorot und Bengio hat dies dazu veranlasst, eine eigene Strategie der Gewichtsinitialisierung vorzuschlagen. Diese wurde „normalisierte Gewichtsinitialisierung“ genannt, heute auch bekannt als die „Glorot-Initialisierung“.

Die Glorot Initialisierung setzt Gewichte einer Ebene auf Werte, die aus einer zufälligen Normalverteilung mit dem Mittelwert 0 und einer Standardabweichung von $\sigma^2 = \frac{1}{m+n}$ im uniformen Intervall $[-\frac{\sqrt{6}}{\sqrt{m+n}}; +\frac{\sqrt{6}}{\sqrt{m+n}}]$ entnommen werden. Dabei steht m für die Neuronen der vorherigen Schicht und n für die Neuronen der nachfolgenden Schicht. Diese Gewichtsinitialisierung ist für saturierende Aktivierungsfunktionen besonders geeignet, also speziell für Sigmoid im vorliegenden Anwendungsfall. [GB10]

He-Initialisierung

Eine andere Veröffentlichung von Kaiming He et al. [He+15] stellt Strategien für weitere Arten von Aktivierungsfunktionen bereit. Diese unterscheiden sich lediglich bei der Skalierung der Varianz.

Die Initialisierungsstrategie für nicht-symmetrische Aktivierungsfunktionen, hier speziell für die ReLU Aktivierungsfunktion, wird als He-Initialisierung bezeichnet. Diese besitzt eine Normalverteilung mit dem Mittelwert 0 und einer Varianz von $\sigma^2 = \frac{2}{n}$ und ist uniform im Intervall $[-\frac{\sqrt{6}}{\sqrt{n}}; +\frac{\sqrt{6}}{\sqrt{n}}]$. Außerdem kann diese Strategie auch für verwandte Aktivierungsfunktionen der ReLU genutzt werden. [He+15]

4.4.2 Batch Normalisierung

Durch die bereits erwähnten Gewichtsinitialisierungen in Abschnitt 4.4.1 können die Probleme des verschwindenden bzw. explodierenden Gradienten am Beginn eines neuronalen Netzes weitestgehend reduziert werden. Jedoch existiert keine Garantie, dass diese Probleme während des Trainings selbst nicht wieder auftreten. Um diesem Problem entgegen zu wirken, wurde im Jahre 2015 von Sergey Ioffe und Christian Szegedy die sogenannte Batch Normalisierung eingeführt [IS15].

Die Technik der Batch Normalisierung kann direkt vor oder nach der Aktivierungsfunktion stattfinden. Im Grunde ist die Aufgabe dieser Normalisierung die Eingabe um 0 zu zentrieren und normalisieren. Im Anschluss werden die Ergebnisse skaliert und verschoben. Dabei werden zwei Parametervektoren genutzt. Einer für das Skalieren und der andere für das Verschieben. Diese zwei Parametervektoren werden im Laufe des Trainings erlernt und somit die Skalierung bzw. Verschiebung auf Basis der Eingabe optimiert. [Gér19, S. 338–345]

Die genauere Vorgehensweise wird in Formel 4.10 aufgezeigt. Der Algorithmus dahinter schätzt im ersten Schritt den Mittelwertvektor μ_B der Eingaben über eine Minibatch. Im darauf folgenden Schritt wird die Varianz σ_B^2 der Eingaben einer Minibatch geschätzt. Anschließend wird der Mittelwertvektor von der Eingabe abgezogen und durch die Varianz geteilt. Das Ergebnis wird als \hat{x}_i definiert. Zuletzt wird \hat{x}_i skaliert und verschoben, mithilfe der zu schätzenden Parametervektoren γ und β . [IS15]:

$$\begin{aligned}
 \mu_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} x_i \\
 \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (x_i - \mu_B)^2 \\
 \hat{x}_i &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
 y_i &= \gamma \hat{x}_i + \beta = BN_{\gamma, \beta}(x_i)
 \end{aligned} \tag{4.10}$$

Durch die Anwendung der Batch Normalisierung ist eine höhere Lernrate durch die skalierten Gradienten möglich. Außerdem, findet eine implizite Regularisierung statt, da die Normalisierung Informationen über eine ganze Batch enthält. Des Weiteren wird die Nutzung saturierender Aktivierungsfunktionen ermöglicht.

Die Anwendung bringt dennoch auch Nachteile mit sich. Diese zeichnen sich als hohe Komplexität in Test und Training aus. Außerdem wird die Trainingszeit einer Epoche länger. [Gér19, S. 338–345]

4.4.3 Regularisierung

Tiefe neuronale Netze besitzen Zehntausende von Parametern, teilweise sogar Millionen. Dadurch besitzen diese eine große Menge an Freiheit und Vielfalt sich an komplexe Datensätze anpassen zu können. Ein Nachteil welcher dabei mitgebracht wird ist, dass diese Flexibilität sehr anfällig für eine Überanpassung des Trainingsdatasets ist. Dies bedeutet, dass durch die vielen Gewichte die Hypothese hervorragend an das Trainingsset angepasst wurde, aber auf neuen, ungesiehenen Daten nicht mehr generalisiert. Dies wird auch als *Overfitting* bezeichnet. Um dieses Phänomen zu erkennen, sind Validation Daten nötig. Durch diese wird überprüft, ob das neuronale Netz auf ungesiehenen Daten generalisiert oder nicht. Um Overfitting zu vermeiden werden Regularisierungen benötigt. Eine der populärsten Arten der Regularisierung ist Dropout. Da nur diese Regularisierung in der vorliegenden Arbeit genutzt wird, wird auch nur diese näher erläutert. [Gér19, S. 364–365]

Dropout

Dropout wurde im Jahre 2014 detailliert von Nitish Srivastava et al. [Sri+14] eingeführt. Der Algorithmus selbst ist dabei simpel gehalten. Bei jedem Trainingsschritt besitzt jedes Neuron eine Wahrscheinlichkeit p , dass dieses temporär herausfällt, also komplett ignoriert wird. Im nächsten Trainingsschritt kann dieses aber wieder aktiv werden. Diese Wahrscheinlichkeit p wird als die Dropout-Rate bezeichnet und befindet sich typischerweise zwischen 10% und 50%. Nach dem Training werden die Neuronen nicht mehr fallen gelassen. Diese Vorgehensweise kann auch anders verstanden werden, nach jedem Trainingsschritt entsteht ein neues individuelles neuronales Netz. Es besteht die Möglichkeit von insgesamt 2^N möglichen Netzen, da ein Neuron entweder aktiv oder inaktiv sein kann. N ist dabei die Anzahl der möglichen ignorierten Neuronen. Dies ist eine so große Anzahl, dass es sehr unwahrscheinlich ist, das selbe Netzwerk zweimal abzutasten. [Gér19, S. 365–367]

Dropout verbessert die Generalisierung eines Netzes deutlich, da Neuronen, durch die fehlenden Verbindungen im Training, sich nicht mehr auf ihre Nachbarn verlassen können. Ein resultierender Nachteil ist, dass Dropout dazu neigt, die Konvergenz deutlich zu verlangsamen. [Gér19, S. 368]

4.4.4 Optimierung

Das Trainieren eines neuronalen Netzes kann sehr lange dauern. Um diesen Prozess zu beschleunigen, wurden bisher Aktivierungsfunktionen, Initialisierungsstrategien für Gewichte und das Nutzen der Batch Normalisierung eingeführt. Bei diesen Funktionalitäten werden weniger Epochen benötigt, um das gleiche Ergebnis ohne dieser zu erreichen. Eine andere Art der Beschleunigung ist ein Optimierer, welcher statt des regulären stochastischen Gradientenabstiegs genutzt wird. Der Gradientenabstieg wird, wie in Abschnitt 4.1 erwähnt, für die Gewichtsanpassung bzw. für die Backpropagation gebraucht. Hierfür existieren verschiedene Möglichkeiten. Im vorliegenden Anwendungsfall wird der zur Zeit beliebteste Optimierer genutzt: Adam. Dieser wird nun genauer erläutert. [Gér19, S. 351]

Adam

Adam steht für *Adaptive Moment Estimations* und vereint bereits existierende Ideen der Optimierung aus Root Mean Square Propagation (RMSProp) und der Momentum Optimierung [KB14]. Außerdem enthält Adam eine adaptive Lernrate. Dies bedeutet, dass die Lernrate sich während des Trainings verändert. Genau wie die Momentum Optimierung verfolgt Adam die exponentielle Abnahme des Durchschnitts der vergangenen Gradienten. Aus RMSProp wird der Ansatz der exponentielle Abnahme des Durchschnitts vergangener quadrierter Gradienten beibehalten. Zusätzlich wird eine Bias-Korrektur aus numerischen Gründen hinzugefügt. Im Formel 4.11 werden die Gleichungen, welche für Adam relevant sind aufgezeigt. Dabei wird g_k als Repräsentation für die Gradienten genutzt. Des Weiteren wird v_k für die Momentum und r_k für die quadrierten Gradienten der RMSProp Darstellung verwendet. \widehat{v}_k und \widehat{r}_k stellen die Bias-Korrektur dar und w_{k+1} die Aktualisierung der Gewichte. [Gér19, S. 356–359]

$$\begin{aligned}
 g_k &= \nabla L(w_k) \\
 v_k &= \beta v_{(k-1)} - (1 - \beta)g_k \\
 r_k &= \rho r_{(k-1)} + (1 - \rho)g_k \odot g_k \\
 \widehat{v}_k &= \frac{v_k}{1 - \beta_k} \\
 \widehat{r}_k &= \frac{r_k}{1 - \rho_k} \\
 w_{k+1} &= w_k - \frac{\widehat{v}_k}{\sqrt{\widehat{r}_k} + \epsilon} g_k
 \end{aligned} \tag{4.11}$$

4.5 Convolutional Neural Networks

Convolutional Neural Network (CNN) sind vorwärtsgerichtete neuronale Netze, welche sich speziell für die Verarbeitung von Bild- und Videodaten eignen. Inspiriert sind diese durch die Beobachtungen der Informationsverarbeitung im visuellen Kortex des Menschen. [DN19, S. 80]

Werden Fully-Connected Netze für Bilddaten genutzt, steigt die Anzahl der Eingabeparameter mit der Größe der Bilder, da jeder Pixel einen einzelnen Eingabewert darstellt. Zusätzlich sind einzelne Pixel nicht geeignet um als Merkmale dargestellt zu werden. Des Weiteren fehlt dabei eine Lokalitätsbeziehung zu den nebenan liegenden Pixeln.

CNN bieten eine Lösung zu diesen Problemen. Diese besitzen eine Lokalitätsbeziehung in Bildern, durch welche Pixel zu ihren Nachbarn in Relation gesetzt werden können. Resultierend können bessere Merkmale identifiziert werden. Außerdem herrscht eine Hierarchie von Merkmale. Darüber hinaus führt die Komposition der Merkmale zu einem Gesamtobjekt. Zuletzt werden die Multiplikationen zur Berechnung der Gewichte und die Gewichte selbst deutlich durch die genutzten Matrizen in CNN Schichten reduziert. [Gér19, S. 445–462]

Im vorliegenden Anwendungsfall werden keine kompletten CNNs erarbeitet. Es werden hierbei ausschließlich die Faltungsschichten (engl. *Convolutional Layer*) sowie die Pooling Schichten genutzt. Faltungsschichten beinhalten einen Filter, welcher auf ein Eingabebild angewendet wird und somit sogenannte Feature Maps generiert. Beispieldhafte Filter sind die Kantendetektionen oder der Gaußsche Weichzeichner. Die Filter selbst werden im neuronalen Netz datengetrieben erlernt und nicht von dem Entwickler bzw. der Entwicklerin bestimmt. Anschließend wird eine Aktivierungsfunktion auf diese Feature Map angewendet. Zuletzt wird eine Pooling Schicht zur Dimensionsreduzierung der Feature Map genutzt und so die markantesten Stellen im Bild verstärkt. Durch eine mehrfache Sequenz von Faltungs-, Aktivierungs- und Pooling Schichten wird das Prinzip der CNN ausgeschöpft. Die richtige Reihenfolge der Filter, Parameter und Aktivierungsfunktionen ist schwierig zu finden. Diese wird meist durch Experimente bzw. Versuche erkannt. [DN19, S. 80–84]

3D Convolutional Neural Networks

In den traditionellen 2D Convolutional Schichten existiert eine Eintrag per Eintrag Multiplikation zwischen der Eingabe und den verschiedenen Filtern. Zusätzlich sind die Filter selbst zweidimensional und werden in zwei Richtungen bewegt: X- und Y- Richtung. In einer 3D Convolutional Schicht hingegen werden dreidimensionale Filter auf die Eingabedaten angewandt. Diese werden in Richtung von drei Achsen bewegt: X-, Y- und Z-Achsen. Die Ausgabe ist ein dreidimensionaler Würfel oder Quader. [Cha20; Ban19] Die Verarbeitung der Convolutionals bzw. Faltungen werden im 2D-Fall in Abbildung 4.3 und im 3D-Fall in Abbildung 4.4 illustriert.

3D-Faltungen werden oft bei der Erkennung in Videos oder medizinischen 3D-Bildern genutzt. Diese sind also nicht auf den 3D-Raum beschränkt, sondern können auch auf

2D-Daten angewendet werden. Im vorliegenden Anwendungsfall werden 3D-Faltungen dennoch genutzt, um 3D-Modelle zu verarbeiten.

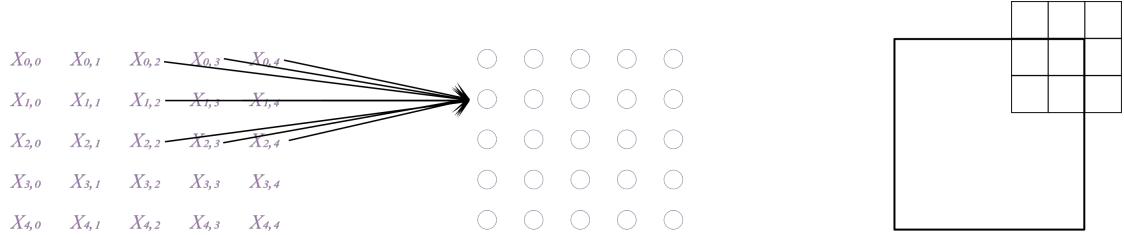


Abbildung 4.3: Abbildung der Verarbeitung von 2D-Faltungen in CNN.
Originale Darstellung entnommen aus [Mel18].

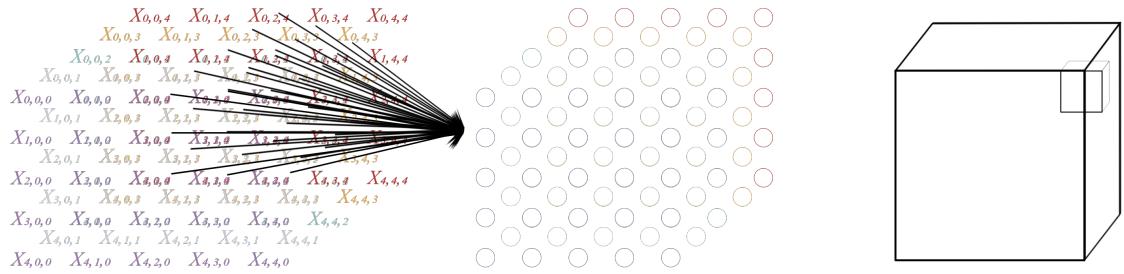


Abbildung 4.4: Abbildung der Verarbeitung von 3D-Faltungen in CNN.
Originale Darstellung entnommen aus [Mel18].

4.6 Residual Neural Networks

2015 wurde die ILSVRC Herausforderung von Kaiming He et al. [He+16] durch ein Residual Neural Network (ResNet) mit einer unter den besten fünf stehenden Error Rates mit 3,57% für sich entschieden. Diese Variante nutzte ein extrem tiefes CNN. Die Varianten, welche bis dort Standard waren, nutzten eine Anzahl von zwischen 34 und 101 Schichten. Das ResNet hingegen besaß 152 Schichten. Somit wurde ein allgemeiner Trend bestätigt: Modelle werden immer tiefer. Denn je tiefer das neuronale Netz, desto komplexere Elemente können modelliert werden. Dies birgt das Problem des *Vanishing Gradients*, die Ergebnisse können saturieren. Außerdem können sich die Ergebnisse mit einer großen Anzahl der Schichten sogar verschlechtern. Der entschiedene Schlüssel, um ein so tiefes Netz trainieren zu können sind Skip-Verbindungen, auch Shortcut-Verbindungen genannt. Der in eine Schicht eingespeiste Eingang wird auch dem Ausgang einer Schicht hinzugefügt. [Gér19, S. 471]

Beim Training eines neuronalen Netzes besteht das Ziel, eine Zielfunktion $h(x)$ zu modellieren. Wird der Eingang x zum Ausgang des Netzes hinzugefügt, also eine Skip-Verbindung eingeführt, ist das Netz gezwungen $f(x) = h(x) - x$ statt $h(x)$ zu modellieren. Dies wird als Residuallernen, engl. *Residual Learning*, bezeichnet. Dieser Vorgang wird in Abbildung 4.5 beschrieben. [Gér19, S. 471]

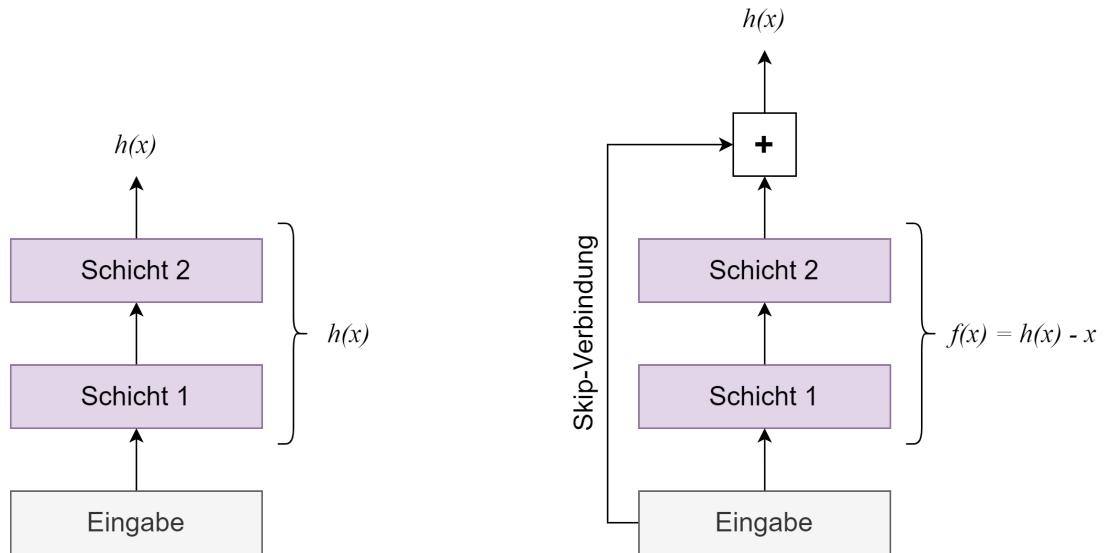


Abbildung 4.5: Darstellung des Residual Learnings. Grafik entnommen aus [Gér19].

Wird ein reguläres neuronales Netzwerk initialisiert, sind die Gewichte nahe bei Null, sodass das Netzwerk nur Werte nahe Null ausgibt. Werden Skip-Verbindungen hinzugefügt, gibt das resultierende Netz eine Kopie seiner Eingaben aus. Diese werden als Identitätsfunktion gesehen. Ist die Zielfunktion nahe der Identitätsfunktion, wird das Training erheblich beschleunigt. Werden viele Skip-Verbindungen zu einem Netz hinzugefügt, kann das Netz auch dann Fortschritte machen, wenn mehrere Schichten noch nicht mit dem Lernen begonnen haben. Ein tiefes ResNet kann als Stapel von Residual-Einheiten betrachtetet werden, wobei jede dieser Einheiten ein kleines neuronales Netz mit einer Skip-Verbindung darstellt. [Gér19, S. 471–474]

4.7 Transfer Learning

Nicht in jedem Anwendungsfall sollte ein tiefes neuronales Netz komplett von Anfang an selbst entwickelt und trainiert werden. Durch die verschiedensten Forscher und Firmen

existieren bereits viele vorgefertigte, tiefe neuronale Netze, welche eine breite Anzahl an Anwendungsfällen decken. Statt ein neues DNN selbst zu entwickeln, existiert die Möglichkeit ein bereits vorhandenes Netz zu suchen, welches die gleiche oder eine ähnliche Aufgabe besitzt, um die Schichten dieses Netzwerks für den eigenen Zweck wieder zu verwenden. Diese Technik wird Transfer Learning genannt. Dadurch wird die Geschwindigkeit des Trainings schneller und es werden deutlich weniger Trainingsdaten benötigt, da diese schon vortrainiert sind. [Gér19, S. 345]

Eine beispielhafte Anwendung wäre: Es existiert ein DNN, welches auf eine Klassifikationsaufgabe von verschiedenen Bildkategorien, darunter Pflanzen, Autos, Tiere und Objekte des täglichen Lebens, bereits trainiert ist. Nun soll ein DNN erstellt werden, welches spezifische Typen von Autos Klassifizieren kann. Da diese Aufgaben sehr ähnlich sind und sich teilweise überlappen, ist dies sehr gut für Transfer Learning geeignet. Es können Teile des einen Netzes für das neue Netz wieder verwendet werden. Die Ausgabe eines neuen Netzes sollte dennoch an die individuelle Aufgabe angepasst werden, da die Anzahl der Klassen nicht übereinstimmen muss oder gar eine komplett andere Ausgabe besitzen soll. Außerdem sollten nicht die letzten Schichten eines Netzes wieder verwendet werden, da die Merkmale, welche hierbei heraus genommen werden können, nicht mit der momentanen Aufgabe übereinstimmen können. Es sollten nur die ersten Schichten genutzt werden, da diese noch allgemeiner angewendet werden können. Eine gute Anzahl für Schichten zu finden, welche wieder verwendet werden, ist hierbei eine Schwierigkeit, welche nur durch Experimente herausgefunden werden kann. Im Allgemeinen hängt die Anzahl der wieder zu verwendeten Schichten mit der Größe der Trainingsdaten zusammen. Umso mehr Trainingsdaten existieren, umso weniger Schichten sollten wiederverwendet werden. [Gér19, S. 345–347]

4.8 Metriken

Für die Auswertung von neuronalen Netzen werden Metriken benötigt. Diese müssen für den Menschen einfach verständlich sein, um diese Interpretieren zu können. Anders als bei einer Kostenfunktion muss die Metrik nicht differenzierbar sein oder als Gradienten Null besitzen. Dennoch wird die Metrik, wie die Kostenfunktion, für jede Batch während des Trainings berechnet und der Mittelwert seit Beginn einer Epoche angegeben. [Gér19, S. 388–389]

Für den vorliegenden Anwendungsfall werden zwei Metriken genutzt. Diese sind Accuracy und Intersection over Union.

4.8.1 Accuracy

Die Accuracy ist als Verhältnis der Anzahl der Datenbeispiele mit richtig vorhergesagten Labels zu der Anzahl aller untersuchten Datenbeispiele definiert. In Formel 4.12 wird diese Berechnung aufgezeigt. Diese Metrik wird sehr häufig genutzt und ist auch der Standard bei Tensorflow Keras. [NZ18, S. 128–129]

$$\text{accuracy} = \frac{n(\text{korrekt vorhergesagte Werte})}{n(\text{alle Werte})} \quad (4.12)$$

4.8.2 Intersection over Union

Die Metrik Intersection over Union (IoU) gibt das Verhältnis der Übereinstimmung zwischen dem Volumen des vorhergesagten Modells und dem des Ground Truth an. Dies wird durch die Formel 4.13 ausgedrückt. Wobei $I(\cdot)$ die Indikatorfunktion, \hat{V}_i der vorhergesagte Wert an der Stelle i , V_i das Ground Truth und ϵ ein Schwellwert ist. Umso höher der Wert der IoU, umso besser ist die Rekonstruktion. Diese Metrik eignet sich vor allem für Voxel Repräsentationen. Handelt es sich bei den zu messenden Modellen um oberflächenbasierte Repräsentationen, sollten diese in Voxel umgewandelt werden. [HLB19]

$$IoU = \frac{\hat{V} \cap V}{\hat{V} \cup V} = \frac{\sum_i I(\hat{V}_i > \epsilon) * I(V_i)}{\sum_i I(I(\hat{V}_i > \epsilon) + I(V_i))} \quad (4.13)$$

4.9 Ähnlichkeitsmaß

Um schlussendlich ein 3D-Objekt mit anderen 3D-Objekten vergleichen zu können, muss ein Maß gefunden werden, welche die Distanz zwischen den Objekten darstellt. Dieses Maß wird im vorliegenden Kontext als Ähnlichkeitsmaß bezeichnet. Dieses weiß darauf hin, wie ähnlich sich zwei Objekte im multidimensionalen Raum sind [Sar19]. Die Idee für ein solches Maß wurde aus [Tai+14] entnommen. Bei dem sogenannten Siamese Network wird nur mit einem einzigen Bild einer Person als Trainingsbeispiel gelernt. Dieses Verfahren

wird auch One-Shot Learning genannt. Dabei werden zwischen den Merkmalsvektoren der Gesichter Ähnlichkeitsmaße berechnet. Somit definiert $d(\text{face}_1, \text{face}_2)$ die Ähnlichkeit zwischen zwei Bildern. Ist das Ergebnis gleich bzw. fällt es unter einen Schwellwert τ , wird die selbe Person auf beiden Bildern angezeigt. Ist dieser Schwellwert τ kleiner, sind unterschiedliche Personen zu erkennen. Liegt der Wert bei Null, zeigt dies die größtmögliche Übereinstimmung an. [Tai+14]

Das gewählte Maß muss bestimmte, allgemein definierte, Kriterien erfüllen. Diese sind Messbarkeit, Universalität, Eindeutigkeit sowie Stabilität und werden nun genauer erläutert [Sar19]:

- Messbarkeit: Es soll eine geeignete Methode zur Bestimmung der Messwerte existieren.
- Universalität: Die Art der Bildung des charakteristischen Messwerts soll für alle unterscheidenden Objekte möglich sein.
- Eindeutigkeit: Der charakteristische Messwert muss für alle zu messenden Objekte unterschiedlich sein.
- Stabilität: Die Messungen mit deren Messwerten müssen zu beliebigen Zeitpunkten wiederholt werden können.

Soll eine Suche nach 3D-Objekten ohne ein Ähnlichkeitsmaß durchgeführt werden, müsste das neuronale Netz von neu trainiert werden. Dies wäre eine äußerst ineffiziente Methode, da dies sehr viel Zeit und Energieaufwand bedeutet. Das Ähnlichkeitsmaß schafft dabei Abhilfe, da das Training des DNN nicht mehr angepasst bzw. verändert werden muss. Es müssen resultierend keine neuen Beispiele regelmäßig hinzugefügt, sondern nur mit den Vorhanden abgeglichen werden.

5 Verwandte Arbeiten

Im folgendem Kapitel werden wissenschaftliche Veröffentlichungen aufgezeigt, welche in Relation mit dieser Arbeit stehen. Es werden die wesentlichen Inhalte der Publikationen und anschließend der Bezug zur vorliegenden Arbeit dargestellt.

Da der Schwerpunkt dieser Arbeit auf der Generierung von 3D-Objekten aus 2D-Bildern besteht, werden ausschließlich solche Veröffentlichungen vorgestellt. Dieses Thema ist im Bereich der Computer Vision ein beliebtes Forschungsgebiet. Entsprechend existieren viele Publikationen mit den verschiedensten Lösungsansätzen dafür. Im Weiteren werden nur wenige dieser vielen aufgezeigt. Dabei wurden Ansätze, welche unüberwachtes Lernen behandeln, ausgeschlossen. Außerdem wurden nur Methoden betrachtet, welche 2D-Bilder als Eingabe nutzen. Somit fallen Vorgehensweisen, wie beispielsweise 2.5D (Depth) Maps, heraus.

3D-R2N2

3D-R2N2 [Cho+16] ist eine der fortschrittlichsten Architektur im Bereich der 3D-Rekonstruktion aus einem Bild. Dabei lernt das Netzwerk das Zuordnen von Merkmalen der 2D-Bilder zur grundlegenden 3D-Form des gesuchten Objektes einer großen, synthetischen Datenbank. Die größte Erneuerung zu der Zeit der Veröffentlichung ist, dass keine Anmerkungen der Bilder bzw. Klassenzuordnungen für die Rekonstruktion nötig sind. Als Dataset wird, unter anderen, ShapeNet Core [Cha+15] genutzt. Das Netzwerk selbst besteht aus drei Teilen. Die erste Einheit ist der Encoder und für die Verarbeitung bzw. Merkmalsextraktion der 2D-Eingaben zuständig. Die mittlere Komponente ist ein 3D Convolutional Long Short-Term Memory (LSTM). Diese ist ein rekurrentes Modul, welches es ermöglicht mehr als ein Bild in das Netzwerk zu geben. Dadurch kann entschieden werden, ob das gesehene Bild bereits bekannt ist oder nicht. Ist es nicht bekannt, wird der Speicher aktualisiert. Zuletzt wird ein Decoder gegeben. Dieser erhält den Hidden State des LSTM-Abschnitts als Eingabe und wendet darauf 3D Convolutions an, um somit das resultierende 3D-Modell als Voxel Repräsentation zu erhalten. Zusätzlich enthalten alle Bestandteile Skip-Connections.

Die grundlegende Architektur des neuronalen Netzes dieser Arbeit basiert auf der Struktur des 3D-R2N2 Netzwerkes. Da in dieser Arbeit nur ein einzelnes 2D-Bild als Eingabe

genutzt wird, wurde entschieden das mittlere Modul, also das rekurrente Modul, nicht zu übernehmen. Somit besteht eine klassische Encoder-Decoder Architektur, mit der Zuordnung der gleichen Verantwortlichkeiten wie in [Cho+16].

IDCT Decoder

Die Veröffentlichung von [Joh+17] stellt fest, dass Methoden welche eine klassische Encoder-Decoder Struktur nutzen zwar erfolgreich sind, aber teilweise Probleme im Bereich Speicherbedarf und Laufzeit bei der Voxel Repräsentation aufweisen. Die Idee der Autoren beläuft sich auf eine alternative Art des Decoders. So wird ein standardisierter 3D Deconvolutional Decoder durch einen Inverse Discrete Cosine Transform (IDCT) Decoder ersetzt. Dabei sagt der Decoder aus niedrig Discrete Cosine Transform (DCT) Koeffizienten ein solides Volumen vorher. Allgemein basiert dieses Netzwerk auf dem von [Cho+16] und nutzt außerdem die selbe Aufbereitung des ShapeNet Core Datasets.

Die Erkenntnis, dass im Bereich der Voxel Repräsentation Probleme vor allem bezüglich des Speicherbedarfs auftreten, wurden mit in die vorliegende Arbeit eingebracht. Somit wurde die mögliche niedrigdimensionale Auflösung der resultierenden 3D-Objekte durch eine implizite Repräsentation ersetzt. Dadurch müssen komplexe Verfahren, welche trotz dessen eine limitierte Auflösung besitzen, wie im erwähnten Paper, nicht implementiert werden.

Point Set Generation Network

Fan et al. [FSG17] nutzt ein Point Set Generation Network, um die Koordinaten von 3D-Punktwolken aus einem einzelnen Bild zu generieren. Außerdem adressieren die Autoren das Problem, dass die 3D-Ausgaben des Netzwerks mehrdeutig sein können. Damit wird eine mögliche, unterschiedliche Ausgabe gemeint, welche durch die verborgenen Ansichten aufgrund des einzelnen Bildes entstehen. Als Lösung wird eine Ausgabe vorgeschlagen, welches mehrere 3D-Objekte in Form von Punktwolken erzeugt. Das Netzwerk selbst besteht aus einem Encoder und einem Decoder. Der Encoder ist auch hier für die Merkmalsextraktion der 2D-Eingaben zuständig und übergibt diese in den Decoder. Die Decoder-Ausgabe ist eine $N \times 3$ Matrix, wobei in jeder Reihe die Koordinaten eines Punktes liegen. Außerdem besitzt das Netzwerk zwei Zweige für die Vorhersage von Werten. Einer dient für eine hohe Flexibilität bei der Erfassung komplexer Strukturen.

Der zweite ist für die geometrische Kontinuität angesetzt. Als Dataset wird das ShapeNet Dataset, genauer die Aufbereitung von Choy et al. [Cho+16], genutzt.

Da auch in dieser Veröffentlichung eine Encoder-Decoder Struktur genutzt wird, bestärkt dies die Entscheidung für die vorliegende Arbeit in ihrer Architektur. Als Ausgabe Repräsentation wurden Punktwolken ausgeschlossen, da diese mehrere Nachbearbeitungsschritte mit sich ziehen. Dennoch ist der Aspekt, dass eine Eingabe mehrere Ausgaben zur Folge haben kann interessant und könnte im zukünftigen Verlauf des Projektes betrachtet werden.

GAN Network

Das Ziel von [Jia+18] ist die Generation aus 2D-Eingaben zu Punktwolken. Dabei wird ein Generative Adversarial Network (GAN) genutzt. Dieses besitzt zwei Bestandteile: Einen Generator und einen Diskriminatoren. Der Generator ist für die Generierung der Punktwolken zuständig. Das Ziel des Diskriminators ist es zwischen der generierten und der realen Punktwolke zu unterscheiden. Dabei besteht der Generator aus einer Encoder-Decoder Architektur, welche auf der Architektur von Fan et al. [FSG17] basiert und von Jiang et al. [Jia+18] erweitert wurde. Der Generator ist für die Produktion von Punktstandorten zuständig. Diese ordnen ein Eingabebild seiner in Relation stehenden Punktwolke zu. Der Diskriminatoren hingegen enthält ein weiteres Netzwerk: PointNet [Qi+17]. Dieses extrahiert die Merkmale der generierten als auch die der Ground Truth Punktwolken. Außerdem werden durch ein CNN die semantischen Merkmale des Eingabebildes erstellt. Die erstellten Merkmale werden miteinander kombiniert. Das Ergebnis ist die finale Repräsentation. Als Dataset fungiert ShapeNet [Cha+15].

Hierbei wird ein anderer Ansatz für die Architektur genutzt. Allgemein bieten GANs den Vorteil, dass diese auch Ergebnisse liefern können ohne eine Eingabe zu verwenden. Im standardisierten Fall werden Gaußsche (verrauschte) Verteilungen für die Generierung benötigt. Somit können ungesehene Daten besser generiert und zusätzlich generalisiert werden. Dennoch wird in der Praxis des vorliegenden Anwendungsfall nur Objekte weniger Objektklassen genutzt. Dadurch wurde entschieden, für den Prototyp, eine simplere Architektur zu wählen. [Gér19, S. 592–595]

Image2Mesh

Die Publikation von [Pon+18] nutzt eine Mesh Repräsentation. Diese ist in der Lage feine Geometrien zu erfassen und für die Aufgabe der 3D-Rekonstruktion zu nutzen. Für die Rekonstruktion selbst werden keine Silhouetten oder Orientierungspunkte benötigt, was bei anderen Projekten oftmals der Fall ist. Das Netzwerk besteht aus mehreren Schritten. Zuerst wird das Eingabebild in einen Convolutional Autoencoder gegeben, um den latenten Raum z zu extrahieren. z wird verwendet, um die Eingabe mit Hilfe eines Multi-Label-Klassifikators zu einem Index zuzuordnen. Anschließend wird ein FFNN zur Regression einer kompakten Formparametrisierung verwendet. Es wird eine Grapheneinbettung für eine kompakte Darstellung von 3D-Netzobjekten verwendet. Dabei wählt der Index zuerst das am nächsten liegende 3D-Modell aus dem Grapen aus. Des Weiteren wird das ausgewählte Modell durch den geschätzten Parameter, sogenannte Free-Form Deformation, und spärliche Linearkombinationsparameter verformt. Durch diese Verformung entsteht das finale 3D-Objekt. Auch in dieser Veröffentlichung wurde ShapeNet [Cha+15] als Dataset eingesetzt.

Ein solcher Aufbau benötigt für jede Objektklasse ein Basismodell für 3D-Daten. Je nach dem wie viele Objektklassen gebraucht werden, steigt auch die Anzahl der Basismodelle und somit auch der nötige Speicherbedarf. Da in der vorliegenden Arbeit nur sehr begrenzt Speicherplatz verfügbar ist, wurde von solch einem Ansatz abgesehen.

Mesh R-CNN

Das Mesh R-CNN erfüllt mehrere Aufgaben gleichzeitig. Es nimmt 2D-Bilder der realen Welt an, also keine synthetischen Daten, und produziert daraus ein Mesh, welches die volle 3D-Struktur des detektierten Objektes wiedergibt. Dabei sollen alle Objekte, welche auf dem Bild zu erkennen sind, detektiert werden. Von diesen Objekten sollen die Namen der Kategorien, je eine Bounding Box, die Segmentierungen sowie die 3D-Meshes selbst generiert werden. Dafür passiert die Eingabe zunächst ein Backbone Network, zum Beispiel ResNet-50-FPN [Lin+17]. Anschließend wird ein Region Proposal Network (RPN), welches im Grunde ein performantes R-CNN ist [Ren+16], durch propagiert. Für das Ergebnis der Rekonstruktion werden im ersten Schritt grobe Voxel vorhergesagt. Diese werden anschließend in Meshes konvertiert. Zuletzt werden die Meshes durch ein Graph Convolutional Network (GNN) verfeinert, indem die Eckpunkte und Kanten der Dreiecke verändert werden. Als Dataset fungierte hierbei das ShapeNet Dataset [Cha+15].

Diese Publikation greift viele Ideen auf, welche für die vorliegende Arbeit interessant sind. Für den realen Gebrauch liegen nicht-synthetische Eingaben näher an der Praxis und sollten bevorzugt verwendet werden. Dennoch werden synthetische Daten genutzt, da das ShapeNet Dataset sich zum Vergleich mit anderen Methoden besser eignet. Außerdem können in dieser Veröffentlichung mehrere Objekte erkannt werden. Das soll vermieden werden, da in der Praxis nur nach einem bestimmten Objekt gesucht werden soll. Der Aufbau des Netzwerkes selbst, ist zu dem vorliegenden ähnlich. Zur Verarbeitung der Bilder wird ein Bildklassifikator genutzt. Für das Erstellen des 3D-Objektes im vorliegenden Projekt, werden auch Voxel erzeugt. Diese sind detaillierter angesetzt. Dafür wird der letzte Schritt, das Erzeugen der Meshes bzw. deren Anpassung, nicht umgesetzt. Dies soll Zeit bei der Berechnung des 3D-Modells einsparen.

Occupancy Networks

Die Autoren von [Mes+19] stellen in dieser Veröffentlichung eine neuartige Repräsentation für datengetriebene, lernbasierte 3D-Rekonstruktionsmethoden. Dabei stellt das Occupancy Network implizit die 3D-Oberfläche als kontinuierliche Entscheidungsgrenze (engl. *Continuous Decision Boundary*) eines Klassifikators eines DNN dar und ist somit eine implizite Repräsentationsart. Das Occupancy Network selbst wird als $f_\theta : \mathbb{R}^3 \times X \longrightarrow [0, 1]$ definiert. Dabei steht f_θ für das neuronale Netz selbst und X für die Eingabe. Somit werden die Wahrscheinlichkeiten der Belegung, also ob ein Punkt innerhalb oder außerhalb des Objektes liegt, beschrieben. In dem vorgestellten Netzwerk zur Generation von 3D-Daten, können Punktwolken, einzelne Bilder und Voxel Repräsentation, in einer niedrigen Auflösung, als Eingabe genutzt werden. Außerdem besitzt dieses auch eine Encoder-Decoder Architektur. Für die Generation von 3D-Objekten aus einzelnen Bildern als Eingabe, werden dabei zwei Netzwerkteile genutzt. Als Encoder wird eine ResNet18 Architektur [He+16] und als Decoder das Occupancy Network verwendet. Das Occupancy Network nutzt Fully-Connected neuronale Netze mit fünf ResNet Blöcken [He+16]. Für das Dataset wurde abermals die Aufbereitung von [Cho+16] des ShapeNet Dataset [Cha+15] genutzt.

Der Vorteil einer impliziten Darstellung wird durch die Experimente der Publikation sehr deutlich. Deswegen wurde für das dritte neuronale Netz, der in dieser Arbeit

implementierten neuronalen Netze, eine implizite Darstellung gewählt. Diese soll vor allem zu einen effizienten Speicherbedarf beitragen.

IM-NET

IM-NET [CZ19a] ist ein Decoder, welcher implizite Felder für das Lernen von generativen 3D-Formen nutzt. Vor allem adressiert die Veröffentlichung die visuelle Qualität der generierten Modelle. Durch den impliziten Decoder, kann die Ausgabe in jeder Auflösung erstellt werden. Dadurch besteht keine Limitierung der Auflösung für die generierten Daten durch die Trainingsdaten. Für das Trainieren des Decoders sind zweierlei Eingaben notwendig. Zum einen die Merkmalsextraktion des Encoders. Als Encoder würde ein Netzwerk dienen, welches die Merkmalsvektoren der Eingabebilder wiedergeben. Zum anderen werden 2D- oder 3D-Punktkoordinaten benötigt. So kann der Innerhalb/ Außerhalb Status jedes Punktes relativ zur Form gegeben werden. Ein klassisches CNN, welches auf Voxeldaten basiert ist, lernt die Verteilung der Voxel. Das IM-NET hingegen, lernt die Formgrenzen des 3D-Objekts. Um die 3D-Objekte als Mesh darstellen zu können, wird eine Methode genutzt, welche sich Marching Cubes [Cub87] nennt. Andernfalls können die 3D-Objekte als Voxel Daten weiterverwendet werden.

Da das dritte neuronale Netz der vorliegenden Arbeit eine implizite Repräsentation erzeugen soll, wurde sich dafür entschieden, diesen Decoder zu nutzen. Die Implementierung des Decoders ist öffentlich zugänglich. Auch die Ergebnisse, welche aus dem Paper ersichtlich sind, überzeugen. Eine genauere Erläuterung des IM-NET wird in Abschnitt 7.5 gegeben.

6 Modellierungsherausforderungen

Das Erstellen und Modellieren von neuronalen Netzen birgt viele Herausforderungen. Es muss eine große Anzahl von Eigenschaften beachtet werden, um ein passendes Netz zu finden. Besonders werden im Folgenden die Herausforderungen im Bereich der Rekonstruktion von 3D-Modellen aus einem einzigen Bild, sowie Herausforderungen für mobile Endgeräte hervorgehoben.

6.1 Objekt Rekonstruktion aus einem Bild

Bei der Rekonstruktion eines 3D-Objekts aus einem einzigen Bild mittels Deep Learning ergeben sich viele Hindernisse, da der Stand der Entwicklung in dieser Richtung noch am Anfang steht [Fu+21]. Generell besitzt die Rekonstruktion hauptsächlich die folgenden Herausforderungen: die Formkomplexität von Objekten, die Unsicherheiten der Rekonstruktion, die Rekonstruktion von sehr detaillierten Objekten, der Speicherbedarf und die Rechenzeit, Datasets und die Repräsentationen der 3D-Modelle. Alle Punkte werden in den nächsten Abschnitten im Detail erläutert.

Formkomplexität von Objekten

Die Formkomplexität von Objekten spiegelt sich in den Unterschieden zwischen den Formen verschiedener Objektklassen wieder. Ein gutes Rekonstruktionsmodell sollte die Fähigkeiten besitzen, Objekte mit unterschiedlicher Komplexität zu erfassen. Zusätzlich muss das Modell Verbindungen zwischen verschiedenen Klassen von Objekten erlernen und gleichzeitig die eigene Einzigartigkeit unter den gleichen Klassen von Objekten bewahren. [Fu+21]

Die Formkomplexität zeigt sich außerdem in der Form des Objekts selbst wieder. Die Struktur eines schlichten Objekts kann oftmals durch die Kombination mehrerer Quadere dargestellt werden. In solch einem Fall, hat ein Objekt tendenziell einen besseren Rekonstruktionswert. Besitzt ein Objekt jedoch eine komplexe Form und kleine bzw. feinkörnige Teile sind die Rekonstruktionsergebnisse eher schlecht. Eine relativ einfache Lösung für diese Herausforderung ist die Erhöhung der Auflösung der rekonstruierten 3D-Objekte. [Fu+21]

Unsicherheiten der Rekonstruktion

Für die Rekonstruktion wird ein einzelnes Bild verwendet. Da ein einzelnes Bild viele dreidimensionale Informationen fehlen und Vorwissen oder Annahmen nicht vorhanden sind, sind die Rekonstruktionsergebnisse nicht eindeutig. Bei Menschen kann dieses Problem durch einen reichen Erfahrungsschatz ausgeglichen werden. So können Menschen aus einem einzelnen RGB-Bild die komplette Form eines Objekts schätzen oder sogar bereits wissen. Da die 3D-Rekonstruktion aus einem Einzelbild auf Basis von Deep Learning erfolgt, muss dies durch ausreichende Daten bzw. Datasets zum Trainieren ausgeglichen werden. [Fu+21]

Rekonstruktion von detaillierten Objekten

Das Ziel der Rekonstruktion ist, nicht nur grobe 3D-Modelle generieren zu können. Die Modelle sollen möglichst detailliert und nah am Ground Truth sein. Der Detailgrad kann durch folgende Eigenschaften beeinflusst werden: Trainingsdaten, gewählte 3D-Präsentation des generierten Modells sowie die Trainingszeit bzw. Anzahl der Epochen an. Zusätzlich ist der Aufbau des neuronalen Netz ausschlaggebend. Allgemeingültig ist, dass umso mehr Trainingsdaten und Zeit zum trainieren vorhanden sind, umso detaillierter werden die generierten 3D-Modelle. Dabei muss jedoch ein gewisses Verhältnis eingehalten werden, da sonst das Problem des Overfittings auftreten kann. Für die Konfiguration aller Eigenschaften gibt es keinen Standard, diese muss durch Experimente herausgefunden und angepasst werden.

Speicherbedarf und Rechenzeit

Oftmals besitzen Geräte einen begrenzten Speicherbedarf bzw. Rechenleistung. Deswegen sollte das Modell, welches eine Rekonstruktion aus einem Bild durchführt, aus verschiedenen Ansichten optimiert werden. Dabei sollte beachtet werden, dass die Größe des Modells weitestgehend reduziert wird und somit über eine ressourcensparende Anzahl an Parametern verfügt. Zu beachten ist hierbei, dass dies zu Auswirkungen auf die Genauigkeit des Ergebnisses führen kann. Außerdem sollte die Latenz reduziert werden. Als Latenz wird die Zeit genannt, welche benötigt wird, um eine Inferenz mit einem gegebenen Modell durchzuführen. Die geringe Latenz kann sich auch positiv auf den Stromverbrauch auswirken. [Mar+15]

Datasets

Ein DNN kann seine leistungsstarke Lernfähigkeit auf das Vorhandensein einer großen Anzahl von Daten zurückführen. Jedoch haben Studien gezeigt, dass Lernverfahren für Einzelbild-3D-Objektrekonstruktionen auf Basis von Deep Learning eher die Erkennungsfähigkeiten erlernen und nur selten die Rekonstruktionsfähigkeiten [Tat+19]. Datasets werden in der Regel in Trainings-, Validierungs- sowie Testsets unterteilt. Im Falle vom verwendeten ShapeNet Dataset ähneln sich die 3D-Modelle im Test- und Trainingsset. Dies ist auch bei den meisten Datasets der Fall. Somit besteht die Möglichkeit, dass das neuronale Netz fehlgeleitet wird und die Erkennung der Objekte und nicht die Rekonstruktion erlernt wird. Zusätzlich existieren große Unterschiede zwischen wilden und synthetischen Datasets. Bei Bildern, welche nicht von neuronalen Netzen gesehen wurden, kann das Ergebnis zu unterschiedlichen Rekonstruktionen führen. In wilden Datasets, bei welchen die enthaltenen Bilder aus realen Szenen bestehen, sind die Bilder komplex. Es existieren Verdeckungen, mehrere Kategorien von Objekten und verschiedenste Beleuchtungen in einem Bild. Daher ist es schwierig die 3D-Rekonstruktion auf solche Bilder anzuwenden. Vor allem wenn das neuronale Netz auf synthetischen Daten trainiert wurde. Letztendlich ist die 3D-Form der Rekonstruktion schlecht. [Fu+21]

Um diese Herausforderung zu überwinden und solch eine Aufgabe in der Realität nutzen zu können, müssen zum trainieren des DNN übermäßig viele Trainingsdaten in verschiedensten Ausprägungen genutzt werden.

Repräsentation von 3D-Modellen

Zum jetzigen Zeitpunkt existiert keine 3D-Repräsentation für 3D-Daten, welche als die State-of-the-Art Konfiguration gewertet wird [Fu+21]. Je nach Anwendungsfall oder Studie wird die entsprechende Repräsentation verwendet. Jede verschiedene 3D-Repräsentation, bringt dabei unterschiedliche Vor- und Nachteile mit sich. Die genaueren Eigenschaften können in den jeweiligen Abschnitten der Repräsentation, also Voxel, Mesh, Point Cloud bzw. implizite Repräsentation, in Kapitel 2 eingesehen werden.

6.2 Mobile Endgeräte

Um den Aspekt der mobilen Anwendung zu erfüllen, muss das neuronale Netz Vorhersagen auf mobilen Geräten, wie einem Smartphone, treffen können. Die Ausführung von DNN

auf einem mobilen Gerät ist bereits durch verschiedene Plattformen und Bibliotheken möglich. Doch ist der Bereich von tiefen neuronalen Netzen noch nicht in der Forschung ausgereizt. Die Anwendung von DNN auf mobilen Geräten ist dabei nur ein kleiner Teil dieses Forschungsfeldes. Dementsprechend herrschen auch in diesem Bereich viele Probleme, für welche Lösungen gefunden werden müssen. In dieser Sektion werden ausschließlich die für diese Arbeit relevanten Herausforderungen erläutert. Diese sind die Plattform bzw. Bibliothek, Speicherkapazität, Berechnungszeit sowie der Stromverbrauch des Gerätes. [Den19]

Plattform und Bibliothek

In den letzten Jahren wurden mehrere Plattformen entwickelt um Deep Learning Anwendungen auch auf Smartphones ausführen zu können. Die Plattform für den individuellen Anwendungsfall zu wählen kann Entwicklungszeit, Kosten und die Leistung beeinflussen. Die richtige Wahl für die eigenen Anforderungen zu treffen ist also essentiell und kann bei der falschen Wahl schnell zu einer großen Herausforderung führen. Es existieren verschiedenste Plattform, zum Beispiel Caffe2, CoreML, DeepLearningKit oder TensorFlow bzw. TFLite. Dabei wurde sich letztendlich für TFLite entschieden. Die Gründe hierfür werden in Abschnitt 7.7 genau dargestellt. [Den19]

Speicherkapazität

Für mobile Geräte gelten viele Grenzen, welche bei statischen Geräten nicht derart ausgeprägt oder erst nicht vorhanden sind. Dazu gehört, unter anderem, die Speicherkapazität. Vor allem bei mobilen Geräten ist diese sehr begrenzt. Viele Geräte, außer die, welche im hohen Segment des momentanen Angebots liegen, müssen in dieser Hinsicht beachtet werden. Zusätzlich enthalten, insbesondere Arbeitsgeräte, viele Dokumente, Bilder, weitere wichtige Applikationen sowie viele 3D-Modelle, welche zusätzlichen Speicherbedarf benötigen. Resultierend darf das Modell des neuronale Netzes nicht zu viel Speicherplatz brauchen. Der Speicherbedarf richtet sich nach der Größe des Modells, genauer nach der Anzahl der Parameter. Diese werden aus der Anzahl der Schichten, Neuronen dieser Schichten, Filtergrößen und weiterer Parameter zusammengestellt. Bei der Konstruktion des Modells, als auch bei der Wahl externer Modelle, sollte dies beachtet werden. [Den19]

Berechnungszeit

Da die Anwendung in Zukunft im Produktivbetrieb eingesetzt werden soll, ist die Dauer der Berechnung essentiell. Das Ziel, welches erreicht werden soll, ist ein Zeitersparnis zu generieren. Dabei soll das Suchen in hunderten von 3D-Modellen bzw. tausenden Dokumenten vereinfacht werden, damit das Zieldokument am schnellsten erreicht wird. Um dies leisten zu können, muss das Generieren und das Suchen in 3D-Modellen performant und somit schnell geschehen. Dabei ist das Ziel vom 2D-Bild zum in Relation stehenden 3D-Modell in der Datenbank innerhalb von einer halben Minute zu gelangen, um die Nutzer in ihrer Arbeit nicht zeitlich zu verhindern.

Stromverbrauch

Die resultierende Applikation dieses Projekts, soll im Arbeitsalltag außerhalb der Firmenzentrale eingesetzt werden. Es wird von einem Arbeitstag von acht Stunden ausgegangen. So sollte das mobile Gerät mindestens diese Zeit genug Akkukapazität besitzen. Folglich sollte die Anwendung nur einen Bruchteil dieses Stromverbrauches annehmen, so dass der Nutzer sein mobiles Gerät wie unter sonstigen Umständen nutzen kann.

7 Realisierung

Im folgendem Kapitel wird die Entwicklung und Implementierung des Projektes im Detail erläutert. Zuerst wird ein Überblick des Trainingssetups gegeben. Danach werden alle Vorverarbeitungsschritte des Datasets, welches für das Trainieren der neuronalen Netze benötigt werden, aufgezeigt. Anschließend wird die Entwicklung des endgültigen neuronalen Netzes erläutert. Diese Entwicklungsschritte erstrecken sich über drei neuronale Netze. Jedes dieser Netze ist durch die selbe Architektur gekennzeichnet. In Abbildung 7.1 wird die Architektur aufgezeigt. Dabei wird ein 2D-Bild in das eigentliche neuronale Netz eingeführt. Dieses besteht aus einem Zwei-Teile-Modell, welches Encoder-Decoder-Modell genannt wird [Cho+14; Gér19, S. 501]. Der Encoder und Decoder übernehmen jeweils eine eigene Aufgabe. Der Encoder ist für die Merkmalsextraktion der 2D-Bilder zuständig. Der Decoder ist für die Generierung der 3D-Modelle aus den Merkmalen der 2D-Bilder des Encoders verantwortlich. Zuletzt wird das generierte 3D-Modell mit dem in Relation stehenden 3D-Modell aus dem Dataset (engl. *Ground Truth*) verglichen, um die Ähnlichkeit bzw. Gleichheit zu überprüfen.

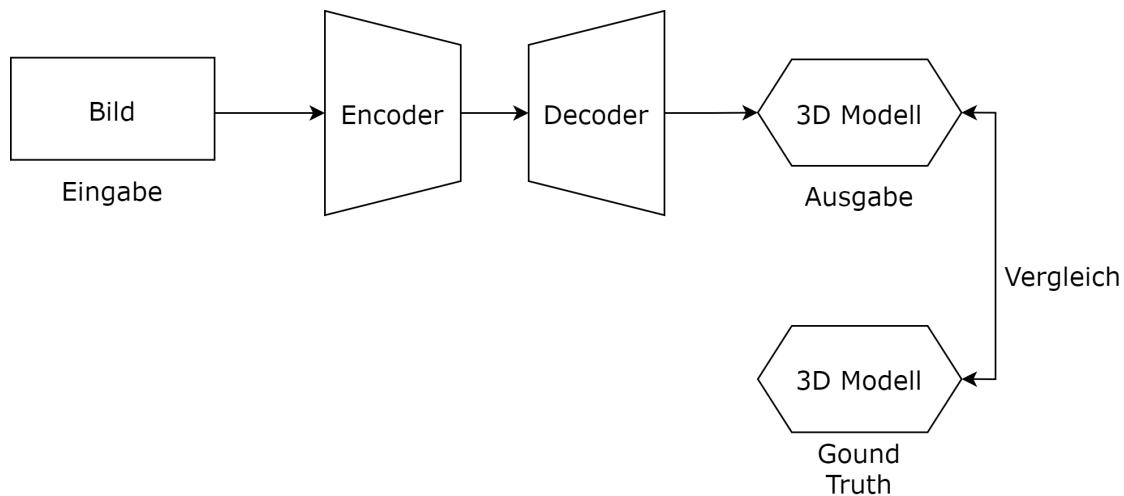


Abbildung 7.1: Darstellung der Encoder-Decoder Architektur für alle implementierten neuronalen Netze.

Je nach Entwicklungsschritt des neuronalen Netzes unterscheiden sich diese im En- bzw. Decoder. Im ersten Schritt wird die Baseline beschrieben. Hierbei wurde der Encoder

und Decoder vollkommen selbst entwickelt. Im zweiten Schritt wurde der Encoder durch ein bestehendes neuronales Netz, welches als MobileNetV2 [San+18] bezeichnet wird, ersetzt. Der letzte Schritt besitzt einen veränderten Decoder, dabei wird der eigens erstellte Decoder durch das IM-NET [CZ19a] ausgetauscht. Hierfür wird der zweite Schritt miteinbezogen, also besteht hier eine Kombination aus dem MobileNetV2 als Encoder und dem IM-NET als Decoder.

7.1 Trainingssetup

Für das Training der neuronalen Netze, wurde im ersten Schritt ein lokal verfügbares Gerät genutzt. Diese besteht aus einem Laptop mit der Bezeichnung: „Asus VivoBook S510UQ-BQ183T“. Als CPU beinhaltet dieser einen Intel Core i7 Prozessor, als GPU eine NVIDIA GeForce 940MX und besitzt 16 GB RAM. Da mit diesem Setup die Berechnungszeiten der DNN sehr lang benötigen, wurde freundlicherweise der Machine Learning-Server der Technischen Hochschule Nürnberg zur Verfügung gestellt. Dieser Server besteht aus acht GPUs, davon wurde eine für das Training der DNN für den vorliegenden Anwendungsfall genutzt. Die verwendete GPU ist eine NVIDIA GeForce RTX 2080 Ti.

Die Implementierung selbst wurde in Python3 entwickelt. Zusätzlich wurde Keras aus Tensorflowv2.4.1 [Mar+15] verwendet. Dabei ist Tensorflow eine End-to-End Open-Source Plattform, welche speziell für Machine Learning-Aufgaben existiert. Es verfügt außerdem über ein großes Ökosystem. In diesem sind verschiedene Tools, Bibliotheken und Community-Ressourcen enthalten. Keras ist eine in Tensorflow enthaltene High-Level API. Diese wird im vorliegenden Anwendungsfall genutzt, da diese nutzerfreundlich und für die meisten Aufgaben der gebrauchten Implementierung vollkommen ausreichend ist. Komplexere Codeabschnitte können dennoch mit Tensorflow, ohne Keras, implementiert werden. Zuletzt wurden parametrisierte Python-Skripte erstellt, mit denen die Anwendungen bzw. die neuronalen Netze initialisiert und trainiert werden können. Dabei können verschiedene Parameter übergeben werden. Darunter fallen die Batchgröße, Epochenanzahl, Lernrate sowie verschiedene Pfade für das Laden der Trainingsdaten und Speicherorte für Logdaten bzw. Checkpoints des Trainings.

7.2 Datenvorverarbeitung

Als Datenbasis wird unter anderem, wie bereits in Abschnitt 3.1 erwähnt, eine Aufbereitung des ShapeNet Core Datasets von Choy et al. [Cho+16] bzw. eine Implementierung davon, welche von Melesse [Mel18] erstellt wurde, verwendet.

Das komplette Dataset liegt aufbereitet, in mehreren Numpy-Dateien [Com21] und somit auch im Numpy-Format vor. Diese besitzen die Endung „.npy“. Die Dateien wurden in X , die Eingaben, und y , die Labels, aufgeteilt. Ein Vorteil ist, dass die Inhalte einfach in Tensoren für Tensorflow umgewandelt werden können, da diese im Numpy-Format vorliegen.

Da sich je nach Entwicklungsschritt das neuronalen Netz unterscheidet, sind veränderte Vorverarbeitungshandlungen notwendig. Für das erste und zweite neuronale Netz unterscheiden diese sich jedoch nur in spezifischen Werten, die Handlungen selbst bleiben bestehen.

Zuerst werden die Vorverarbeitungen für 2D-Bilder, also für die Eingaben des DNN, gegeben. Diese werden aus den Numpy-Dateien geladen und in Numpy-Arrays gelagert. Da die Inhalte bereits vorverarbeitet sind, müssen nur Feinheiten angepasst werden. Darunter fällt die Anpassung der Größe der Numpy-Arrays. Die Arrays bestehen aus drei Kanälen: Höhe des Bildes, Breite des Bildes und zuletzt der Farbkanal. In den bestehenden Daten wurden vier Farbkanäle angegeben, d. h. Rot, Grün, Blau und der Alpha Kanal. Da 2D Convolutional Schichten nur eine Eingabe annehmen, welche drei Farbkanäle besitzt (Rot, Grün und Blau), musste der Alpha Kanal fallen gelassen werden. Des Weiteren müssen die Höhe bzw. die Breite des Bildes auf die neuronalen Netze angepasst werden. Dazu wurde eine zufällige initialisierte Beschnidung des Bildes vorgenommen. Dieses wurde beispielsweise von 137x137 auf eine Größe von 224x224 gebracht. Bei der Nutzung von Transfer Learning, müssen weitere Vorverarbeitungsschritte initiiert werden. Um das MobileNetV2 nutzen zu können, muss eine eigene Vorverarbeitung für dieses Netz, bereit gestellt von Keras, durchgeführt werden. Diese wird durch die Funktion `tf.keras.applications.mobilenet_v2.preprocess_input(image_to_preprocess)` ausgeführt [Mar+21].

Des Weiteren müssen die Daten für die Labels auch vorverarbeitet werden. Hierbei muss nur die Form der Arrays angepasst werden. Da die Inhalte, durch die von [Mel18]

durchgeführte Stapelung, ineinander geschachtelt sind, müssen diese nun in eine funktionale Form gebracht werden. Dabei wird eine Schachtelung entnommen, sodass alle Inhalte auf einer Ebene existieren. Außerdem werden die Achsen des 3D-Modells in die richtige Reihenfolge gebracht, um dieses anschließend auch darstellen zu können.

Für das dritte und somit letzte neuronale Netz, wird auch das ShapeNet Core Dataset verwendet. Dabei wurden auch von Choy et al. [Cho+16] die Eingabebilder genutzt. Als 3D-Modelle hingegen, also für die Labels, wurden Voxel mit einer höheren Auflösung von 256^3 als in [Cho+16] genutzt. Diese wurden von [HTM17] entnommen. Da für dieses Netzwerk zusätzlich Punktkoordinaten benötigt werden, wurden diese mit einem Point Sampling-Code [CZ19b] generiert. Dabei werden mehr Punkte abgetastet, umso näher diese an der 3D-Form liegen. Somit soll die Anzahl der Punktpaare möglichst gering bleiben.

7.3 Baseline

In dieser Sektion wird die Baseline im Detail vorgestellt. Die Baseline ist die Grundlage, auf welcher alle weiteren Entwicklungsschritte des finalen neuronalen Netzes aufbauen. Außerdem wurde die Baseline aufgrund keinen speziellen, für diesen Zweck erstellten neuronalen Netz entwickelt. Es wurden ausschließlich sehr grundlegende Netzstrukturen genutzt, wie sie beispielsweise bei der Bildklassifikation verwendet werden.

Wie bereits in Abschnitt 7 erwähnt, bestehen alle Netze aus einer Encoder-Decoder-Architektur. In den folgenden Abschnitten werden der En- und Decoder der Baseline genauer erläutert.

Allgemeine Konfigurationen des kompletten Modells, welches En- und Decoder enthält, bestehen aus den Kostenfunktion, Optimierer, Metriken und Callbacks. Für die Kostenfunktion wurde eine voxelweise Cross Entropy verwendet. Dabei sind die Label y one-hot encoded. Dies bedeutet, dass die Voxel binär dargestellt werden und so nur Null oder Eins sein können. In diesem Fall bedeutet dies, ob ein Voxel an einer bestimmten Stelle im 3D-Raum existiert oder nicht. Zusätzlich wird eine Wahrscheinlichkeitsberechnung durchgeführt, wie wahrscheinlich es ist, dass an dieser Stelle ein Voxel existiert. Genau wird die Kostenfunktion in Formel 7.1 definiert.

$$L(X, y) = \sum_{(i,j,k)} y_{(i,j,k)} \log(p_{(i,j,k)}) + (1 - y_{(i,j,k)}) \log(1 - p_{(i,j,k)}) \quad (7.1)$$

Dabei ist der finale Voxel an der Stelle (i, j, k) zu finden. X ist die Eingabe des Netzes und y das in Relation stehende Label. $p_{(i,j,k)}$ stellt die Wahrscheinlichkeit eines Voxel an der Stelle (i, j, k) dar.

Die Implementation ist durch Tensorflow deutlich vereinfacht. Denn Tensorflow stellt folgende Funktion `tf.nn.softmax_cross_entropy_with_logits(y, predicted_y)` zur Verfügung. Dort können die Labels sowie die vorhergesagten Werte übergeben werden. Zusätzlich können verschiedene Parameter hinzugefügt werden. Hierbei wurde die Achse verändert, so dass die Voxel korrekt berechnet werden. Diese wurde auf `axis = 3` gesetzt.

Des Weiteren wurde als Optimierer Adam gewählt. Dieser wurde bereits in Sektion 4.4.4 im Detail erläutert. Als Lernrate wurde initial, durch Experimente, 0.01 gewählt.

Als Metriken wurden Accuracy und IoU gewählt. Für die Accuracy wurde die standard Funktion `tf.keras.metrics.Accuracy()` von Tensorflow genutzt.

Die Metrik IoU musste für den 3D-Fall teilweise selbst realisiert werden. Dabei wurden die vorhergesagten Werte des 3D-Modells durch einen Schwellwert auf die Werte Null und Eins gebracht, um diese mit den originalen 3D-Modellen vergleichen zu können. Anschließend kann die eigene Funktion von Tensorflow aufgerufen werden. Für den 3D-Fall muss die Anzahl der Klassen der IoU angepasst werden. Diese wird um eine Achse erhöht und somit von Eins auf Zwei gesetzt. Danach muss der Status des IoU manuell aktualisiert werden, um ein Ergebnis zu berechnet. Dieser Vorgang wird in Listing 7.1 gezeigt.

```
1 def intersection_over_union_3d(y, y_predicted):
2     y_predicted = tf.where(y_predicted >= 1, 1, 0)
3     m = tf.keras.metrics.MeanIoU(num_classes=2)
4     m.update_state(y, y_predicted)
5     return m.result()
```

Listing 7.1: Implementierung der IoU für den 3D-Fall.

Zuletzt werden die Callbacks aufgezeigt. Callbacks sind Dienste, welche an bestimmten Punkten im Training des neuronalen Netzes aufgerufen und durchgeführt werden. Es werden zwei Callbacks genutzt. Der erste ist der Modell Checkpoint. Hierbei werden in Abständen, welche selbst bestimmt werden können, das Modell zwischengespeichert. Dabei muss ein Pfad für die zu speichernde Datei angegeben werden. Außerdem wird durch die Konfiguration `save_weights_only=True` angegeben, dass ausschließlich die Gewichte und

nicht das Modell selbst gespeichert werden soll. Dies spart Speicherplatz. Im vorliegenden Anwendungsfall werden alle 100 Epochen Checkpoints erstellt. Falls das Training durch einen Fehler abbricht, kann das Training an der gespeicherten Stelle fortgesetzt werden. Außerdem kann so nach dem Training das Modell mit den erlernten Gewichten geladen werden, um so Vorhersagen durchzuführen.

Der zweite Callback stellt das Tensorboard zur Verfügung. Dies bietet eine Visualisierung und verschiedene Werkzeuge, welche für Experimente mit DNN benötigt werden. Dies beinhaltet die Aufzeichnung und Darstellung von Metriken, wie bspw. der Kostenfunktion. Des Weiteren können Modellgraphen, Histogramme von Gewichten und anderen Tensoren sowie vieles mehr angezeigt werden. [Mar+15]

Encoder

Die vorverarbeiteten Daten werden nun als Eingabe für den Encoder genutzt. Da es nicht möglich ist alle Daten auf einmal im Speicher zu halten, da diese zu viel Speicher benötigen, müssen sie nach und nach geladen werden. Dazu werden Batches sowie ein Data Generator genutzt. Batches wurden bereits in Abschnitt 4.1 eingeführt. Der Data Generator ist eine Technik, mit welcher Daten nacheinander, in Batches, in den Speicher geladen werden können. Listing 7.2 zeigt die Implementierung der Klasse des Data Generators. Zur Initialisierung nimmt diese Klasse die Eingaben X , die Labels y , die Batchgröße $batch_size$ sowie die Größe der Inhalt einer Datei $data_size$. In diesem Fall ist diese fest bei 24, kann aber noch je nach Datenform verändert werden. In der Funktion `__getitem__` werden die Daten in die Batches aufgeteilt. In der Variable X , welche in den Data Generator übergeben wird, sind nur die Pfade zu den Numpy-Daten enthalten. So können die Numpy-Arrays der 2D-Bilder, welche als Eingabe für das neuronale Netz dienen, in Batches in den Speicher geladen werden. In der enthaltenen For-Schleife werden die 2D-Bilder sowie die 3D-Daten, wie in Abschnitt 7.2 erläutert, vorverarbeitet, um diese für das Training des neuronalen Netzes nutzen zu können. Zuletzt werden die resultierenden Arrays als Numpy-Arrays in Tupel zurück gegeben. In einer anderen Form nimmt ein Modell von Tensorflow dies nicht an. Der Data Generator wird für die Trainings- aber auch für die Valdiation-Daten genutzt.

```
1  class Batch_Load_Generator(keras.utils.Sequence) :
2      def __init__(self, X, y, batch_size, data_size = 24) :
3          self.X = X
4          self.y = y
5          self.batch_size = batch_size
6          self.data_size = data_size
7
8      def __len__(self) :
9          return (np.ceil(len(self.X) * self.data_size / float(self.
10                     batch_size * self.data_size))).astype(np.int)
11
12     def __getitem__(self, idx) :
13         batch_x = self.X[idx * self.batch_size : (idx+1) * self.
14                     batch_size]
15         batch_y = self.y[idx * self.batch_size : (idx+1) * self.
16                     batch_size]
17
18         output_X = []
19         output_y = []
20
21         img_batch_X = load_npy(batch_x)
22
23         for index ,imgs in enumerate(img_batch_X):
24             for img in imgs:
25                 img = img[:, :, 0:3]
26                 output_X.append(img)
27
28                 vox = np.load(batch_y[index])
29                 vox = np.argmax(vox, axis=-1)
30                 vox = vox.transpose(2, 0, 1)
31                 output_y.append(vox)
32
33
34         return (np.array(output_X), np.array(output_y))
```

Listing 7.2: Data Generator zum Laden von Daten in Batches in Python.

Ein High-Level-Graph der Architektur des Encoders kann in Abbildung 7.2 eingesehen werden. Der Encoder selbst besteht aus der Eingabe, der Ausgabe sowie aus drei Blöcken, welche dazwischen liegen. Diese enthalten jeweils eine Convolution Schicht, Batch Normalisierung, Aktivierungsfunktion und ein MaxPooling.

In der ersten Schicht des ersten Blocks, werden 137 Filter mit einem Faltungskern der Größe ($2x2$) angewandt. Diese nimmt als Eingabeform (engl. *Input Shape*) eine Größe von ($137x137x3$) an. Außerdem wird als Gewichtsinitialisierung die He-Initialisierung genutzt. Zusätzlich wird der Rand (engl. *Padding*) als gleich gesetzt, um die Größe des Resultates nicht zu verändern. In allen Folgenden Schichten wird der Rand so gehandhabt. Auf diese erste Schicht folgt eine Batch Normalisierung. Des Weiteren wird die LeakyReLU mit einem Alpha-Wert von 0.2 als Aktivierungsfunktion verwendet, um tote Neuronen zu vermeiden. Zusätzlich wird ein Dropout mit einer Wahrscheinlichkeit von 20% hinzugefügt, um mögliches Overfitting weitgehendst zu vermeiden. Zuletzt wird eine MaxPooling Schicht verwendet, welche eine Filterkerngröße von ($2x2$) besitzt sowie einen Stride von Zwei. Der Stride definiert die Schrittweite, inwiefern der Filter des MaxPoolings im Zusammenhang der darauf angewendeten Matrix verschoben wird.

Im zweiten Block existiert ein ähnlicher Aufbau. Dabei wird in der ersten Schicht eine Filteranzahl von 64 mit einer Filterkerngröße von ($2x2$) gewählt. Die Gewichtsinitialisierung ist dabei gleich bei der He-Initialisierung geblieben. Wie bereits im ersten Block folgt eine Batch Normalisierung sowie die LeakyReLU, mit einem Alpha-Wert von 0.2, als Aktivierungsfunktion. Nun wird kein Dropout durchgeführt. Es folgt direkt das MaxPooling mit einer Größe von ($2x2$).

Der dritte Block besitzt eine Filteranzahl von 8 mit einem ($3x3$) Filterkern. Es wird ein Stride von Zwei durchgeführt sowie die He-Initialisierung als Gewichtsinitialisierung. Das Folgende unterscheidet sich nicht von dem zweiten Block. Eine Batch Normalisierung, eine LeakyReLU mit einem Alpha-Wert von 0.2 und einer MaxPooling Schicht mit der Größe von ($2x2$).

Auffällig dabei sind die niedrigen Größen der Filterkerne der Convolution Schichten. Diese wurden speziell so gewählt. Durch, unter anderem, das LeNet von LeCun et al. [LeC+98] wurde bekannt, dass die Kernel der Filter klein gehalten werden sollen. So sollen bevorzugt zwei ($3x3$) Kernel anstatt ein ($5x5$) Kernel genutzt werden. Denn werden ($5x5$) Kernel bei 128 Neuronen genutzt, ergeben diese 3328 Gewichte. Werden stattdessen zwei

kleinere Kernel mit (3x3) genutzt, werden 2560 Gewichte generiert.

Allgemein sind alle Anzahlen, sowohl an Filtern als auch Strides, so gewählt, dass als Resultat eine Matrix der Größe (8x8x8) entsteht. Diese Matrix, welche die Merkmale der Eingabe enthalten, wird in den Decoder übergeben und dort weiter verarbeitet.

Da die Anwendung des neuronalen Netzes für mobile Endgeräte vorgesehen ist, sollte das Netz selbst möglichst wenig Parameter besitzen. Damit soll Speicherplatz eingespart sowie schnellere Vorhersagen getroffen werden. Demnach ist die Anzahl der Parameter wichtig und wird im Folgenden aufgezeigt. In Tabelle 7.1 werden die Parameter und die Ausgabeform des Encoders je Schicht angegeben. Die Bezeichnungen der Schichten wurden von Tensorflow übernommen. Die Form der einzelnen Schichten wird als Batch Größe und einem dreidimensionalen Tensor angeben. Die Batch Größe wird beim starten des Trainings angegeben und in der Tabelle als *None* gekennzeichnet. Im Encoder insgesamt existieren 42.369 Parameter. Davon sind 41.951 trainierbare Parameter und 418 nicht trainierbare.

Schicht	Ausgabeform	Parameteranzahl
conv2d (Conv2D)	(None, 137, 137, 137)	1.781
batch_normalization (Batch Normalisierung)	(None, 137, 137, 137)	548
activation (LeakyReLU(0.2))	(None, 137, 137, 137)	0
dropout (Dropout)	(None, 137, 137, 137)	0
max_pooling2d (MaxPooling2D)	(None, 68, 68, 137)	0
conv2d_1 (Conv2D)	(None, 68, 68, 64)	35.136
batch_normalization_1 (Batch Normalisierung)	(None, 68, 68, 64)	256
activation_1 (LeakyReLU(0.2))	(None, 68, 68, 64)	0
max_pooling2d_1 (MaxPooling2D)	(None, 34, 34, 64)	0
conv2d_2 (Conv2D)	(None, 17, 17, 8)	4.616
batch_normalization_2 (Batch Normalisierung)	(None, 17, 17, 8)	32
activation_2 (LeakyReLU(0.2))	(None, 17, 17, 8)	0
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 8)	0

Tabelle 7.1: Zusammenstellung aller Schichten, deren Ausgabeform sowie Parameteranzahlen des neuronalen Netzes des Encoders.

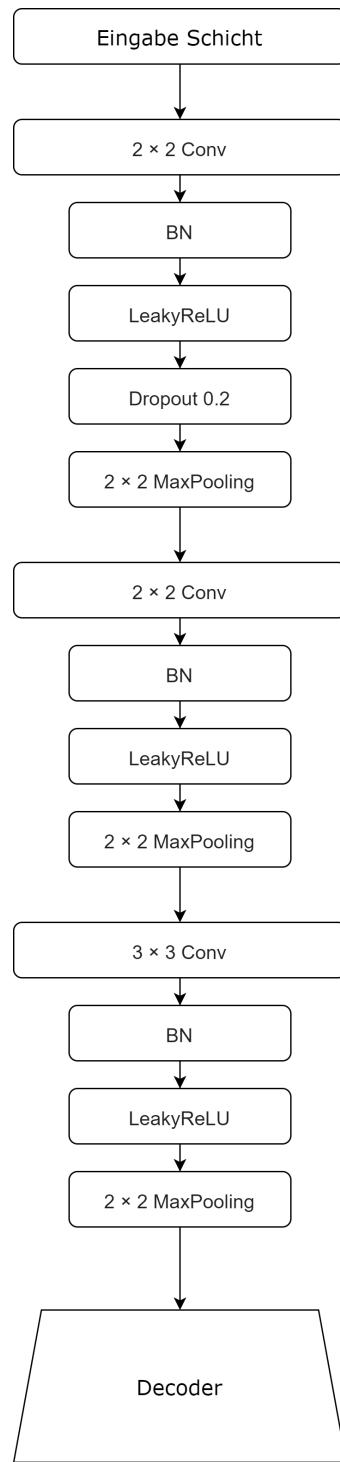


Abbildung 7.2: Darstellung eines High-Level-Graphen der Architektur des selbst erstellten Encoders.

Decoder

Der strukturelle Aufbau des Decoders kann in Abbildung 7.3 nachvollzogen werden.

Der Decoder selbst erhält die Merkmalsextraktion als Tensor der Eingabe durch den Encoder. Der Tensor besitzt die Größe von $(8 \times 8 \times 8)$, somit hat die Eingabeform die gleiche Größe, um den Tensor annehmen zu können. Da der Decoder 3D-Daten generieren soll, ist es als erstes nötig den Tensor um eine Dimension zu erweitern. Somit wird der Tensor mit einer 2D-Merkmalsextraktion für 3D-Daten angereichert. Der restliche Decoder besteht aus vier Blöcken. Jeder Block weißt die gleiche Struktur auf. Zuerst eine Convolution 3D Transpose Schicht, anschließend eine Batch Normalisierung und zuletzt eine Aktivierungsfunktion.

Im ersten Block der ersten Schicht, existiert eine Convolution 3D Transpose, welche 16 Filter mit einer Größe von $(3 \times 3 \times 3)$ besitzt. Eine Transpose Schicht wird auch Deconvolution genannt. Diese wird genutzt, wenn in die entgegengesetzte Richtung einer normalen Convolution gegangen werden soll. Das heißt von etwas, das die Form des Ausgangs einer Convolution besitzt, zu etwas, das die Form des Eingangs hat. Dabei soll das Konnektivitätsmuster beibehalten werden, welches mit den Convolutions kompatibel ist. [Mar+15]

Außerdem wird der Padding gleich gesetzt, wie auch bei allen kommenden Schichten. Als Gewichtsinitialisierung wird die He-Initialisierung genutzt. Danach wird eine Batch Normalisierung angewendet. Des Weiteren wird als Aktivierungsfunktion die ReLU-Aktivierungsfunktion bereitgestellt.

Der darauf folgende Block besitzt zuerst eine Convolution 3D Transpose Schicht mit einer Filteranzahl von 32 und einer Größe des Filterkerns von $(3 \times 3 \times 3)$. Für die Gewichtsinitialisierung wird die He-Initialisierung genutzt. Zusätzlich wird ein Stride von Zwei eingeführt. Es folgt, wie bereits bei der ersten Schicht, eine Batch Normalisierung sowie die Aktivierungsfunktion ReLU.

Im dritten Block wurde die Anzahl der Filter als 32 und der Filterkern als $(3 \times 3 \times 3)$ der Convolution 3D Transpose gewählt. Die Gewichtsinitialisierung und Stride wurden wie in Block zwei gewählt. Danach folgt wieder eine Batch Normalisierung sowie die ReLU als Aktivierungsfunktion.

In dem letzten Block findet zuerst eine Batch Normalisierung statt. Anschließend wird

eine Convolution 3D Transpose Schicht mit einem Filter und einem Filterkern von (3x3x3) durchgeführt. Dabei wird als Gewichtsinitialisierung die Glorot-Initialisierung gewählt. Zuletzt wird als Aktivierungsfunktion Sigmoid gewählt, da eine binäre Klassifikation je Voxel stattfindet. Sigmoid kann in einer Zwei-Klassen-Klassifikation zur Berechnung der Wahrscheinlichkeiten genutzt werden. Der resultierende Tensor besitzt vier Dimensionen. Um aus dem Decoder ein 3D-Modell zu erhalten, ist es nötig einen drei dimensionalen Tensor zu erzeugen. Dafür wurde in dieser Convolution 3D Transpose Schicht nur ein einziger Filter genutzt. So ist es möglich durch eine triviale Umformung auf die gewünschte Dimension zu gelangen. Dabei wird die Form von (1x32x32x32) auf die Form (32x32x32) angepasst. Der resultierende Tensor und somit auch das 3D-Modell haben die Form bzw. Größe (32x32x32). In Tabelle 7.2 werden die Parameter und die Ausgabeform je Schicht aufgezeigt. Bei der Form der Schichten wird die Batch Größe als *None* dargestellt. Anschließend folgt ein vierdimensionaler Tensor, bis auf die letzte Schicht. In dieser findet eine Umformung statt, wobei die Dimension vermindert wird. Es bestehen gesamt 72.097 Parameter. Davon sind 71.745 Parameter trainierbar und 352 nicht trainierbar.

Schicht	Ausgabeform	Parameteranzahl
reshape (Reshape)	(None, 8, 8, 8, 1)	0
conv3d_transpose (Conv3DTranspose)	(None, 8, 8, 8, 16)	448
batch_normalization_3 (Batch Normalisierung)	(None, 8, 8, 8, 16)	64
activation_3 (ReLU)	(None, 8, 8, 8, 16)	0
conv3d_transpose_1 (Conv3DTranspose)	(None, 16, 16, 16, 32)	13.856
batch_normalization_4 (Batch Normalisierung)	(None, 16, 16, 16, 32)	128
activation_4 (ReLU)	(None, 16, 16, 16, 32)	0
conv3d_transpose_2 (Conv3DTranspose)	(None, 32, 32, 32, 64)	55.360
batch_normalization_5 (Batch Normalisierung)	(None, 32, 32, 32, 64)	256
activation_5 (ReLU)	(None, 32, 32, 32, 64)	0
batch_normalization_6 (Batch Normalisierung)	(None, 32, 32, 32, 64)	256
conv3d_transpose_3 (Conv3DTranspose)	(None, 32, 32, 32, 1)	1.729
reshape_1 (Reshape)	(None, 32, 32, 32)	0

Tabelle 7.2: Zusammenstellung aller Schichten, deren Ausgabeform sowie Parameteranzahlen des neuronalen Netzes des Decoders.

Auffallend ist die genauere Wahl der Filteranzahl der Convolutions im Encoder bzw. bei den Deconvolutions im Decoder. Im Encoder wird die Anzahl der Filter von 137 auf 64 auf 8 immer kleiner. So wird ein Engpass erstellt, was bedeutet die Dimensionalität wird reduziert. Dies verringert die Rechenkosten und die Anzahl der Parameter. Außerdem beschleunigt dies das Training und verbessert die Generalisierung. Im Gegensatz dazu, wird die Anzahl der Filter von 16 auf 32 auf 64 immer größer. Dieser Mechanismus wird Bottleneck genannt. Er zwingt das neuronale Netz dazu, eine allgemeine Repräsentation der Merkmalskombination zu lernen. Somit wird ein generalisiertes DNN erzeugt, welches auch auf ungesehenen Daten gute Ergebnisse erzielen kann. [Gér19]

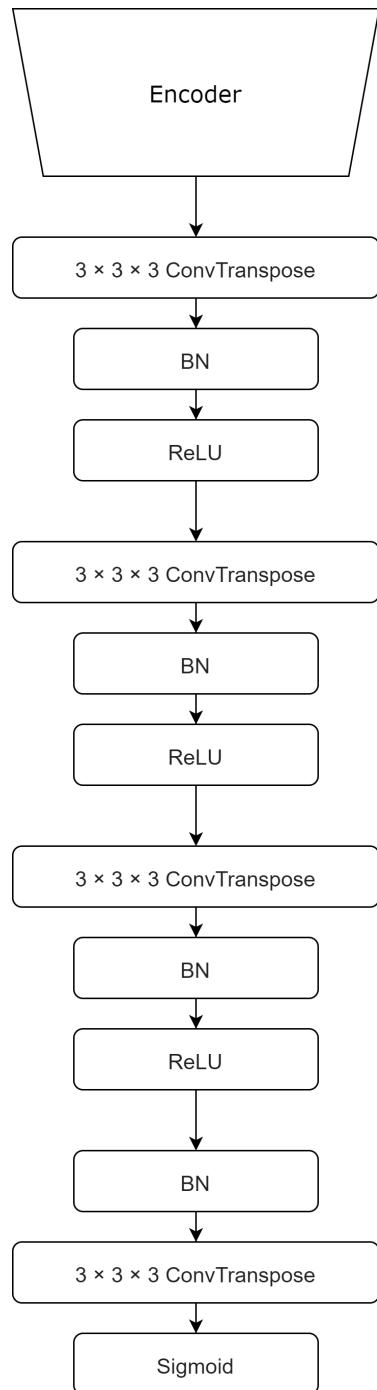


Abbildung 7.3: Darstellung eines High-Level-Graphen der Architektur des selbst erstellten Decoders.

7.4 MobileNetV2

Der nächste Entwicklungsschritt, welcher zum finalen neuronalen Netz beiträgt, besteht darin den selbst erstellten Encoder durch ein bereits existierendes Modell zu ersetzen. Diese Methode wird Transfer Learning genannt und wurde in Abschnitt 4.7 detailliert beschrieben. Das Modell, welches eingesetzt wurde, heißt MobileNetV2 [San+18]. Es existieren bereits drei Varianten dieses Modells, welche stetig erweitert bzw. verbessert werden. Somit wurde mit dem MobileNet [How+17] begonnen, mit dem MobileNetV2 [San+18] weitergeführt und es existiert bereits ein MobileNetV3 Modell [How+19]. Leider konnte das aktuellste Modell nicht genutzt werden, da hierfür notwendige Abhängigkeiten auf dem Server, welcher genutzt wurde, fehlten. Diese konnten nicht nachträglich installiert werden, ohne andere Nutzer des Servers zu beeinträchtigen. Somit wurde auf das Modell davor, also auf das MobileNetV2, zurückgegriffen.

Die Encoder-Decoder-Struktur bleibt weiterhin bestehen. Es wird ausschließlich der Encoder-Teil verändert. Die allgemeinen Konfigurationen, das heißt Kostenfunktion, Optimierer, Metriken sowie Callbacks, werden nicht verändert. Diese werden wie in der Baseline, Kapitel 7.3, gewählt. Somit wird dieses Modell auf der Baseline aufgebaut. Im Folgenden wird einzig der Encoder beschrieben. Der Decoder kann in Passus 7.3 nachgeschlagen bzw. in Abbildung 7.3 nachvollzogen werden. Einzig wurde die Eingabeform des gesamten Decoders verändert. Dieser nimmt einen Tensor der Größe $(4 \times 4 \times 32)$ an und formt diesen in die Größe $(8 \times 8 \times 8 \times 1)$ um. Damit ist nun die vorherige Form der Baseline wieder erreichbar und wird entsprechend auch genutzt.

Encoder

Wie bereits bei der Baseline, wird zum laden der Daten ein Data Generator verwendet. Dieser unterscheidet sich nur minimal zu dem des Encoders. Dabei wird die Größe der Eingaben angepasst. Durch das verwendete MobileNetV2, welches im Anschluss im Detail erläutert wird, wird die Eingabegröße vorgeschrieben. Diese beläuft sich auf eine Größe von $(224 \times 224 \times 3)$. Zusätzlich muss ein weiterer Vorverarbeitungsschritt speziell für das MobileNetV2, genauer in Kapitel 7.2, eingearbeitet werden. Dies geschieht auch im Data Generator.

Der Encoder besteht in dieser Entwicklungsstufe ausschließlich aus dem MobileNetV2

[San+18]. Mitarbeiter von Google Inc. haben die Architektur dieses mobilen Modells im Jahre 2019 entwickelt. Diese wurde speziell für den Anwendungsfall der mobilen und ressourcenbeschränkten Umgebung entwickelt. Dabei wurde versucht die Anzahl der Operationen und des gebrauchten Speicherbedarfs deutlich zu verringern, ohne das die Accuracy darunter leidet. Das Design des Netzwerks basiert auf dem des Vorgänger Modells: MobileNetV1 [How+17]. Es behält dadurch seine Einfachheit. Außerdem erfordert dies keine speziellen Operatoren, während die Accuracy deutlich verbessert und der Stand der Technik bei der Klassifizierung- und Erkennungsaufgaben für mobile Anwendungen erzielt wurde. [San+18]

Allgemein basiert die Architektur von MobileNetV2 auf einem Schichtmodul: *Inverted Residual Structures*, wobei die Skip-Verbindungen zwischen *Linear Bottlenecks* liegen. Dieses Modul nimmt als Eingabe eine niedrigdimensionale, komprimierte Darstellung, welche auf eine hohe Dimension erweitert und mit einer leichtgewichtigen, Depthwise Convolution gefiltert wird. Anschließend werden die Merkmale mit einer linearen Convolution zurück auf eine niedrigdimensionale Darstellung mit einer linearen Convolution projiziert. Die Kombination der zwei Convolutions wird *Depthwise Separable Convolutions* genannt. [San+18]

Diese Depthwise Separable Convolutions werden oft in Architekturen von effizienten neuronalen Netzen genutzt [How+17; Cho17; Zha+18]. Die Grundidee ist dabei, volle Operationen von Convolutions mit einer faktorisierten Version, welche Convolutions in zwei separate Schichten aufteilt, zu ersetzen. Die erste Schicht wird als *Depthwise Convolution* bezeichnet. Diese führt eine leicht gewichtete Filterung durch, bei welcher ein einzelner Faltungsfilter pro Eingabekanal angewendet wird. Die zweite Schicht ist eine (1×1) Convolution, welche als *Pointwise Convolution* bezeichnet wird und für die Erstellung neuer Merkmale durch Berechnung von Linearkombinationen der Eingangskanäle zuständig ist. Eine konventionelle Convolution nimmt einen Tensor T_i der Größe $h_i \times w_i \times d_i$ an. Darauf wird ein Filterkern $K \in \mathbb{R}^{k \times k \times d_i \times d_j}$ angewendet, um einen Ausgabetensor T_j der Größe $h_i \times w_i \times d_j$ zu erzeugen. Dabei besitzt solch eine Convolution Berechnungskosten von $h_i \cdot w_i \cdot d_i \cdot d_j \cdot k \cdot k$. Depthwise Separable Convolutions sind ein vollwertiger Ersatz für reguläre Convolutional Schichten. Empirisch arbeiten diese annähernd wie reguläre Convolutional Schichten. Die Berechnungskosten betragen dabei aber nur $h_i \cdot w_i \cdot d_i (k^2 + d_j)$. Dies ist die Summe der Depthwise und der Pointwise Convolutions. Diese Art von Faltungen reduzieren effektiv die Berechnung, im Vergleich zu traditionellen Schichten, um annähernd den Faktor k^2 . MobileNetV2 nutzt $k = 3$.

Damit ist die Größe der Filterkerne gemeint. Somit ergibt sich eine Depthwise Separable Convolution der Größe (3×3) . Resultierend bewegen sich die Berechnungskosten in einem acht- bis neun-mal niedrigeren Bereich als bei herkömmlichen Convolutions. Die Accuracy wird dabei nur minimal reduziert. [San+18]

Für die Linear Bottlenecks wird ein tiefes neuronales Netz mit n Schichten L_i betrachtet, wobei jede Schicht einen Aktivierungstensor der Dimension $h_i \times w_i \times d_i$ besitzt. Dabei wird der Tensor als ein Container von $h_i \times w_i$ Pixel mit einer Dimension von d_i gesehen. Es wird festgelegt, eine Eingabemenge von realen Bildern für eine Schichtaktivierung, jeder Schicht L_i , ein *Manifold of Interest* bildet und so bezeichnet wird. Es wurde eine lange Zeit angenommen, dass dies in neuronalen Netzen in niedrigdimensionalen Subräumen eingebettet werden könnte. Wird also in jeder einzelnen d -Dimension die Pixel einer tiefen Faltungsschicht betrachtet, liegt die in diesen Werten kodierte Information, in einem Manifold, welches in einem niedrigdimensionalen Subraum einbettbar ist. Diese Erkenntnis könnte ausgenutzt werden, indem die Dimensionalität einer Schicht und somit die Dimensionalität des Berechnungsraum reduziert wird. Dies wurde bereits erfolgreich im Aufbau des MobileNetV1 Modells erschlossen und genutzt, um einen effektiven Kompromiss zwischen Berechnung und Genauigkeit über einen *Width Multiplier Parameter* zu erzielen. Diese Vorgehensweise wurde auch in Netzentwürfen anderer DNN integriert [Zha+18]. Durch den Width Multiplier Ansatz, wird die Reduzierung der Dimensionalität des Aktivierungsraums erlaubt, bis das Manifold of Interest diesen gesamten Raum einnimmt. Jedoch bricht dieser Ansatz zusammen, unter der Berücksichtigung, dass DNN nicht-lineare, pro Koordinate Transformationen besitzen. Ein Beispiel dafür wäre die ReLU Aktivierungsfunktion. Beispielsweise erzeugt die ReLU Funktion im 1D-Raum einen „Strahl“, während im R^n Raum im Allgemeinen eine stückweise, lineare Kurve mit n -Knoten erstellt wird. Das Manifold of Interest soll in einem niedrigdimensionalen Subraum des höherdimensionalen Aktivierungsraum liegen. Es existieren zwei Eigenschaften, welche auf diese Anforderung hinweisen: Ist das Manifold of Interest nach der ReLU-Transformation ein Volumen ungleich null, entspricht dies einer linearen Transformation. Des Weiteren ist ReLU in der Lage, vollständige Informationen über das Eingabemanifold zu erhalten. Dies geschieht nur unter einer Bedingung, welche im Eingabemanifold liegt. Das Manifold muss in einem niedrigdimensionalen Subraum des Eingaberaums liegen. Diese Erkenntnisse liefern einen empirischen Hinweis für Optimierungen bestehender neuronaler Netzwerk Architekturen. Unter der Annahme, dass das Manifold of Interest niedrigdimensional ist, kann dies berücksichtigt werden, indem lineare Bottlenecks in Convolutional Blöcke eingefügt werden. Die Verwendung

linearer Schichten ist entscheiden. Denn diese verhindern, dass nicht-linearitäten zu viele Informationen zerstören. Durch die Nutzung von nicht-linearen Schichten in Bottlenecks verschlechtert sich die Leistung um mehrere Prozentpunkte. Dies wurde durch empirische Daten von [San+18] aufgezeigt. Auch [HKK17] entfernte nicht-Linearitäten in den Eingaben von traditionellen Residual Blöcken. Dies verbesserte die Leistung. [San+18]

Die bereits erwähnten Bottleneck Blöcke wirken ähnlich zu Residual Blöcken. Hierbei besitzt jeder Block eine Eingabe, gefolgt von mehreren Bottlenecks und einer Erweiterung [He+16]. Die Skip-Verbindungen werden direkt zwischen den Bottlenecks verwendet. Dies wurde durch die Idee angeregt, der Zufolge die Engpässe alle notwendigen Informationen beinhalten. Dagegen fungieren Expansionsschichten lediglich als Implementierungsdetails, welche nicht-lineare Transformationen begleiten. Der Grund für das Einfügen solcher Skip-Verbindungen ist ähnlich der von klassischen Residual Verbindungen. Es soll die Fähigkeit eines Gradienten durch Multiplikationsschichten zu propagieren verbessert werden. Das invertierte Design ist jedoch viel speichereffizienter und arbeitet in den Experimenten besser. [San+18]

Wie bereits erwähnt, besteht die Basis der Modell Architektur aus Blöcken, welche Bottlenecks mit Depth Separable Convolution mit Skip-Verbindungen enthalten. Die detaillierte Ausführung der Struktur wird in Tabelle 7.3 aufgezeigt. Allgemein umfasst die Architektur des MobileNetV2 eine initiale Convolutional Schicht mit einer Filtergröße von 32. Darauf folgend sind 19 Residual Bottleneck Schichten, diese können genauer in Tabelle 7.4 eingesehen werden. Es wird ReLU6 als nicht-Linearität genutzt. Außerdem wird ausschließlich die Kernelgröße (3×3) verwendet. Des Weiteren werden Dropout sowie die Batch Normalisierung während des Trainings eingesetzt. Mit Ausnahme der ersten Schicht, wird ein konstanter Erweiterungsfaktor genutzt, welcher den Wert 6 besitzt. [San+18]

Die primäre Netzwerkarchitektur weiß Berechnungskosten von 300 Millionen Multiply-Adds auf und nutzt 3.4 Millionen Parameter. Die Modellgröße variiert nach der Anzahl der Multiply-Adds. Diese liegen in der Spanne von sieben Millionen bis 585 Millionen Multiply-Adds. Entsprechend variiert die Größe des Modells zwischen 1.7 Millionen und 6.9 Millionen Parametern.

Besonders die invertierten Residual Bottleneck Schichten ermöglichen eine speichereffiziente Implementierung. Diese ist essentiell für mobile Anwendungen. Die Speichergröße ist die maximale Gesamtgröße der kombinierten Ein- und Ausgänge über alle Operationen.

Werden Residual Bottleneck Blöcke als einzelne Operationen behandelt, wird der gesamte Speicher von der Größe der Bottleneck Tensoren aus dominiert. Somit wird der Speicherbedarf verringert, da die Größe der Tensoren, welche intern zum Engpass zugehörig sind, viel größer sind. Zusammengefasst ermöglicht diese Architektur eine speichereffiziente sowie für mobile Geräte angepasste Inferenz und nutzt Standardoperationen, welche in allen neuronalen Frameworks vorhanden sind. [San+18]

Die Nutzung des MobileNetV2 im Projekt wird in Listing 7.3 aufgezeigt. Dabei wird dieses Modell als Basis für den Encoder verwendet. Für die Eingabegröße wurde $(224 \times 224 \times 3)$ gewählt. Außerdem wurden die obersten Fully Connected Schichten nicht genutzt. Für die initialen Gewichtungen, wurde von den Gewichten von *ImageNet* Gebrauch gemacht. Wie bereits in Passus 4.7 erwähnt, können die Gewichte der einzelnen Schichten trainiert werden oder mit den initialen Gewichten belassen werden. Im vorliegenden Fall wurden die ersten 20 Schichten nicht trainiert und belassen. Die nachfolgenden Schichten wurden neu und speziell für den Anwendungsfall trainiert. Die Ausgabe dieses Netzes wird anschließend in eine weitere Convolutional Schicht gegeben. Diese besitzt die ReLU-Aktivierungsfunktion, einen Stride von 2, 32 Filter, einen Filterkern von (2×2) und zuletzt wurde der Padding auf Same gewählt. Final werden alle Teile des Netzwerks verbunden und resultierend ergibt sich der Encoder, welcher in den Decoder übergeben wird.

```
1 encoder_base = tf.keras.applications.MobileNetV2(input_shape=( 224, 224,
2     3), include_top=False, weights='imagenet')
3 for layer in encoder_base.layers[:20]:
4     layer.trainable=False
5 for layer in encoder_base.layers[20:]:
6     layer.trainable=True
7
8 output = tf.keras.layers.Conv2D(32, (2,2), strides=2, padding='same',
9     activation='relu')(encoder_base.output)
10 encoder = tf.keras.models.Model(inputs=encoder_base.input, outputs=
11     output)
```

Listing 7.3: Transfer Learning mit dem Modell MobileNetV2 für den Encoder in Python.

Eingabe	Konfiguration	Ausgabe
$h \times w \times k$	(1x1) conv2D (Conv2D), ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	(3x3) dwise s= s , ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear (1x1) conv2d (Conv2D)	$\frac{h}{s} \times \frac{w}{s} \times k'$

Tabelle 7.3: Detaillierte Struktur der Bottleneck Blöcke mit Transformation von k zu k' Channel, Stride s und dem Erweiterungsfaktor t . [San+18]

Eingabe	Konfiguration	t	c	n	s
(224x224x3)	conv2D (Conv2D)	-	32	1	2
(112x112x32)	Bottleneck	1	16	1	1
(112x112x16)	Bottleneck	6	24	2	2
(56x56x24)	Bottleneck	6	32	3	2
(28x18x32)	Bottleneck	6	64	4	2
(14x14x64)	Bottleneck	6	96	3	1
(14x14x96)	Bottleneck	6	160	3	2
(7x7x160)	Bottleneck	6	320	1	1
(7x7x320)	(1x1) conv2D (Conv2D)	-	1280	1	1
(7x7x1280)	(7x7) avgpool (AveragePooling2D)	-	-	1	-
(1x1x1280)	(1x1) conv2D (Conv2D)	-	k	-	-

Tabelle 7.4: Überblick der gesamten Architektur von MobileNetV2. Dabei beschreibt jede Zeile eine oder identische Schichten, welche n -mal wiederholt werden. Alle Schichten einer Zeile haben die gleiche Größe c der Ausgabe. Zusätzlich besitzt die erste Schicht einen Stride s , alle anderen Schichten besitzen einen Strike von 1. Der Erweiterungsfaktor t wird auf die Eingabegröße, beschrieben in Tabelle 7.3, angewendet. [San+18]

7.5 IM-NET

Der letzte Entwicklungsschritt bzw. das finale neuronale Netz beinhaltet als Encoder das in Abschnitt 7.4 erläuterte MobileNetV2 und als Decoder das IM-NET [CZ19a]. Das heißt, dass auch in diesem Fall die Encoder-Decoder Architektur bestehen bleibt.

Die allgemeinen Konfigurationen haben sich, durch die unterschiedliche 3D-Präsentation dieses DNNs gegenüber der bisherigen neuronalen Netze, verändert. Somit besteht eine andere Kostenfunktion, eine gewichtete Mean Squared Error. Als Optimierer wird, wie bisher, Adam mit einer Lernrate von 0.01 genutzt. Als Metriken werden auch die Accuracy sowie IoU verwendet. Die Callbacks können in diesem Fall nicht verwendet werden. Dies liegt an der verwendeten Version von TensorFlow, welche in der Implementierung von IM-NET zum Einsatz kam. Die detaillierte Begründung wird im späteren Verlauf gegeben.

Decoder

Als Decoder wurde, wie bereits erwähnt, das IM-NET [CZ19a] gewählt. Dieser nutzt implizite Felder, um die Formen von 3D-Modellen generativ zu erzeugen. Ein implizites Feld weist jedem Punkt im 3D-Raum einen Wert zu und ist als kontinuierliche Funktion definiert. Somit kann eine Form als Iso-Fläche extrahiert werden. Das IM-NET wurde darauf trainiert, die Zuordnung mit Hilfe eines binären Klassifikators durchzuführen. Genauer nimmt dieses neuronale Netz im Einzelnen eine Punktkoordinate (x, y, z) zusammen mit einem Merkmalsvektor an. Der Merkmalsvektor ist die Kodierung der Form, welcher vom Encoder, im vorliegenden Fall durch das MobileNetV2, erzeugt wird. Als Ausgabe folgt ein Wert, welcher angibt, ob der Punkt an der Stelle (x, y, z) innerhalb oder außerhalb der Form liegt. Für geschlossene Formen wird ein Feld F der Form definiert, in dem das Signal eines SDF genommen wird: [CZ19a]

$$F(p) = \begin{cases} 0, & \text{wenn Punkt } p \text{ außerhalb der Form liegt,} \\ 1, & \text{sonst} \end{cases} \quad (7.2)$$

Dabei wird angenommen, dass das Feld in einem 3D-Einheitsraum beschränkt ist. Es soll versucht werden, eine Parametrisierung $f_\theta(p)$ mit Parametern θ zu finden, welche einen Punkt $p \in [0, 1]^3$ auf $F(p)$ abbildet. Die Form wird anschließend durch alle Punkte, welche einem spezifischen Wert zugeordnet bekommen haben, dargestellt. Zuletzt werden diese mit einer Iso-Flächen Extraktion, in diesem Fall durch Marching Cubes [Cub87], als 3D-Modell gerendert. Dies bringt mehrere Vorteile mit sich, welche die visuelle Qualität des 3D-Objektes betreffen. Zunächst kann die Ausgabe in jeder Auflösung abgetastet werden und ist auch nicht auf die Auflösung der Trainingsdaten limitiert. [CZ19a]

Da eine unterschiedliche Kostenfunktion genutzt wird, wird diese nun beschrieben. Die Funktion wird als gewichtete MSE bezeichnet. Die Kostenfunktion wird zwischen dem vorhergesagten Label und dem Ground Truth Label auf jeden einzelnen Punkt angewendet. Die Definition der Kostenfunktion wird in Formel 7.3 dargestellt. Dabei ist S eine Sammlung von Punkten, abgetastet von der Zielform. [CZ19a]

$$L(\theta) = \frac{\sum_{p \in S} |f_\theta(p) - F(p)|^2 \cdot w_p}{\sum_{p \in S} w_p} \quad (7.3)$$

Der implizite Decoder besteht aus mehreren MLPs mit ReLU-Aktivierungsfunktionen. Mit genügend verdeckter Schichten sind MLPs in der Lage, das Feld F mit beliebiger Genauigkeit anzunähern. Dies ist ein direktes Resultat des universellen Approximationstheorems [Hor91]. Der detaillierte Aufbau des Modells sowie die jeweiligen Ausgabeformen bzw. Parameteranzahlen der einzelnen Schichten wird in Tabelle 7.5 aufgezeigt. Insgesamt besitzt dieses DNN etwa 3 Millionen Parameter. [CZ19a]

Schicht	Ausgabeform	Parameteranzahl
dense (Dense)	(None, 1024)	266240
leaky_re_lu (LeakyReLU(0.2))	(None, 1024)	0
dense_1 (Dense)	(None, 1024)	1049600
leaky_re_lu_1 (LeakyReLU(0.2))	(None, 1024)	0
dense_2 (Dense)	(None, 1024)	1049600
leaky_re_lu_2 (LeakyReLU(0.2))	(None, 1024)	0
dense_3 (Dense)	(None, 512)	524800
leaky_re_lu_3 (LeakyReLU(0.2))	(None, 512)	0
dense_4 (Dense)	(None, 256)	131328
leaky_re_lu_4 (LeakyReLU(0.2))	(None, 256)	0
dense_5 (Dense)	(None, 128)	32896
leaky_re_lu_5 (LeakyReLU(0.2))	(None, 128)	0
dense_6 (Dense)	(None, 1)	129

Tabelle 7.5: Zusammenstellung aller Schichten, deren Ausgabeform sowie Parameteranzahlen des neuronalen Netzes des IM-NETs.

Um das IM-NET nutzen zu können, wurde die Implementierung der Autoren verwendet. Diese wurde, unter anderem mit Python 3.5 und TensorFlow 1.8.0 entwickelt. Da diese Version von TensorFlow nicht mehr unterstützt wird, musste die Implementierung in eine aktueller Version umgewandelt werden. Für das vorliegende Projekt wurde mit TensorFlow 2.4.1 gearbeitet. Durch den großen Versionsunterschied, mussten alte Funktionen teilweise komplett ersetzt werden. Neuere Funktionalität, wie beispielsweise *Callbacks*, können in den überarbeiteten Code schwer eingebracht werden. Die Funktionen an sich wurden dennoch auf anderen Wegen implementiert. Dabei lag im Vordergrund die Basisfunktionalität des Codes nicht zu verändern.

Für das Training des IM-NETs werden ein Encoder sowie Punktkoordinaten benötigt. Der Encoder wird durch das MobileNetV2 gegeben. Die Punktkoordinaten für das ShapeNet Core Dataset, sind von den Autoren der IM-NET Veröffentlichung mit dem Code freigegeben. Somit müssen diese nicht extra erzeugt werden. Soll mit einem anderen Dataset trainiert werden, können die Punktkoordinaten mit Skripten aus [CZ19b] für Voxel Daten generiert werden. Der IM-NET Decoder nimmt einen latenten Merkmalsvektor der Größe 256 und Punktcoordinaten der Form (x, y, z) an. Das gesamte vorhergesagte 3D-Objekt wird in diesem Fall in der Größe (256, 256, 256) generiert. Durch die implizite Darstellung kann dies einfach verändert werden und ist nicht auf diese Größe festgelegt. Das 3D-Objekt bekommt für jede Punktkoordinate eine Fließkommazahl zugeordnet. Durch einen Schwellwert wird festgelegt, ob die Koordinate innerhalb, also als 1 gewertet oder außerhalb des Objekts, also als 0 gewertet, liegt. Der Schwellwert liegt bei einem Wert von 0.5.

7.6 Ähnlichkeitsmaß

Für das Ähnlichkeitsmaß werden zwei Werte berechnet. Zu einem wird die bereits bekannte Metrik IoU, siehe Abschnitt 4.8.2, genutzt. Zum anderen wird eine neue Distanz eingeführt. Dieses wird Hausdorffer-Metrik oder auch Haussdorffer Distanz, genannt.

Die Hausdorffer Distanz [TLK09] zeigt die maximale Abweichung zwischen zwei Objekten. Sie misst also wie weit zwei Punktmenzen voneinander entfernt sind [GBK05]. Dafür müs-

sen zwei nicht-leere Punktmengen $A = \{x_1, x_2, \dots, x_n\}$ und $B = \{y_1, y_2, \dots, y_m\}$ gegeben sein. Die Hausdorffer Distanz zwischen A und B im \mathbb{R}^3 ist definiert als: [Zha+17]

$$H(A, B) = \max(h(A, B), h(B, A)) \quad (7.4)$$

wobei

$$\begin{aligned} h(A, B) &= \max_{x \in A} (\min_{y \in B} \|x - y\|) \\ h(B, A) &= \max_{y \in B} (\min_{x \in A} \|y - x\|) \end{aligned} \quad (7.5)$$

Ist $H(A, B)$ ein kleinerer Wert, sind sich beide Objekte ähnlich. Aber ist $H(A, B)$ gleich Null, sind beide Objekte identisch.

Die Werte von IoU werden dabei anders behandelt. Umso höher die resultierenden Werte sind, umso besser stimmen die Objekte überein. Ist der Wert Eins bzw. 100% sind beide Objekte identisch.

Für die Umsetzung der IoU wurde die gleiche Implementierung verwendet, wie beim Training der neuronalen Netze. Diese kann in Listing 7.1 nachgelesen werden. Für die Hausdorffer Distanz wurde eine vorimplementierte Funktion von SciPy [com21] genutzt.

7.7 Realisierung für mobile Geräte

Für die Umsetzung der Realisierung für mobile Geräte wurde für TensorFlow Lite (TFLite), welches von [Mar+15] zur Verfügung gestellt wird, gewählt. Dies wurde aufgrund der Tatsache gewählt, dass bisher TensorFlow als Plattform für maschinelles Lernen genutzt wurde und TFLite ein Zusatz davon ist. TFLite ist ein Deep Learning Framework und ermöglicht es neuronale Netze bzw. deren Inferenz auf mobilen Geräten ausführen zu können. Als mobiles Gerät wurde ein Android Gerät, genauer ein Smartphone mit der Bezeichnung „Honor View 20“, genutzt. Entsprechend wurde die Umsetzung für den vorliegenden Anwendungsfall für Android Geräte entwickelt. Daher wurde Android Studio [LLC21] mit der Programmiersprache Java verwendet.

Um ein Modell überhaupt in Android Studio importieren zu können, muss dieses aus dem Python Code mit einem Converter exportiert werden. Somit wird ein TensorFlow Modell in ein TFLite Modell umgewandelt. Ist dies geschehen, kann das TFLite Modell in die Android Studio Umgebung importiert werden. Somit kann auf das Modell mit dem Java

Code des Projektes zugegriffen werden. Über einen *Interpreter* wird das Modell geladen. Es enthält die gleichen Konfigurationen, wie in dem Modell der Python-Implementierung. Somit ist die Form der Eingabe sowie Ausgabe unverändert. Für die prototypische Umsetzung wurden die Eingabebilder fest im sogenannten Asset-Ordner hinterlegt. Diese werden zuerst als *Bitmap* geladen und anschließend als *TensorImage* weiterverarbeitet, um diese mit einem TFLite Modell verwenden zu können. Nach einer Vorverarbeitung der Eingaben durch den *ImageProcessor*, welcher vor allem für die richtige Größe des Eingabebildes zuständig ist, wird die Inferenz gestartet. Dies geschieht durch den Aufruf: `tflite_interpreter.run(input_value, output_value)`. In der Variable `output_value` wird das Ergebnis nach der Berechnung des Modells vermerkt. Darin befindet sich das resultierende 3D-Modell und kann dadurch weiterverarbeitet werden. Die Umsetzung der Abstandsmaße musste für die mobile Anwendung überarbeitet werden, da die Funktionen von TensorFlow (IoU) und SciPy (Hausdorffer Distanz) nicht in dieser Umgebung anwendbar sind. Beide Funktionalitäten wurden dem entsprechend selbst realisiert.

8 Experimente

Um die Verwendung der vorhandenen und neu trainierten neuronalen Netze sowie dem Ähnlichkeitsmaß zu testen, werden verschiedene Experimente durchgeführt. Jedes neuronale Netz wurde für die Experimente 1500 Epochen trainiert.

Das Kapitel der Experimente ist folgendermaßen aufgebaut: Zuerst werden 3D-Modelle anhand verschiedener Kategorien von Bildern generiert, um die Generalisierung der DNNs zu überprüfen. Anschließend werden Auskünfte über die Präzision der einzelnen neuronalen Netze gegeben. Des Weiteren wird der zeitliche Aspekt der Inferenz betrachtet, da dieser in der Praxis eine große Rolle spielt. Außerdem wird der zeitliche Aspekt für die Berechnung des Ähnlichkeitsmaß durchgeführt. Zuletzt findet die Schwellwertbestimmung des Ähnlichkeitsmaßes statt, um die Suche nach 3D-Modellen in weiteren Modellen zu optimieren.

8.1 Vorhersagen differierender Eingaben

In diesem Abschnitt werden die neuronalen Netze auf verschiedene Kategorien von Eingaben getestet. Die Kategorien sind:

- Synthetische Bilder des Datasets ShapeNet Core [Cha+15]
- Synthetische Bilder innerhalb der Kategorien des besagten Datasets
- Synthetische Bilder außerhalb der Kategorien des besagten Datasets
- Bilder der Realität mit Objekten, welche innerhalb der Kategorien des Datasets vorhanden sind

Für jedes Eingabebild werden die jeweiligen Ergebnisse der einzelnen Netze, also der Baseline, MobileNetV2 und des IM-NET, angezeigt. Die einzelnen Abbildungen stellen zuerst das Eingabebild dar. Dieses wird leicht unscharf dargestellt, da die Auflösung durch das Dataset auf (137x137) begrenzt ist. Anschließend wird das Ergebnis der Baseline, danach des MobileNetV2 Encoders und zuletzt das Ergebnis des IM-NET Decoders abgebildet. Das Ergebnis des IM-NET Decoders besitzt durch eine unterschiedliche Anordnung der Achsen, welche durch die originale Implementierung entstanden ist. Entsprechend wurde das Bild gedreht und besitzt somit eine andere Achsenbeschriftung. Außerdem ist die

Auflösung der 3D-Modelle unterschiedlich. Die Objekte der Baseline sowie MobileNetV2 wurden in 32^3 und die des IM-NETs auf 256^3 gewählt. Die des IM-NETs wurde deutlich feiner gewählt, um Details bestmöglichst anzeigen zu können.

Zuerst werden Eingaben betrachtet, welche aus dem ShapeNet Core Dataset und somit synthetische Eingabe sind. In Abbildung 8.1 wird für das Eingabebild ein Schrank aus dem ShapeNet Core Dataset gegeben. Rechts davon wird das Ergebnis der Baseline aufgezeigt. Dabei wird ersichtlich, dass das Ergebnis zwar um den Mittelpunkt des Objektes generiert wurde, das Objekt selbst jedoch nicht erkennbar ist. Rechts davon wird das Resultat des MobileNetV2 Encoders abgebildet. Dieses ist deutlich reduzierter als das der Baseline. Hierbei sind Fortschritte erkennbar, jedoch keine Verbesserung zur Erkennung des Objektes selbst. Zuletzt wird das Ergebnis des MobileNetV2 Encoders und IM-NET Decoders gegeben. Das Eingabebild wird dabei nahezu perfekt als 3D-Modell wiedergespiegelt.

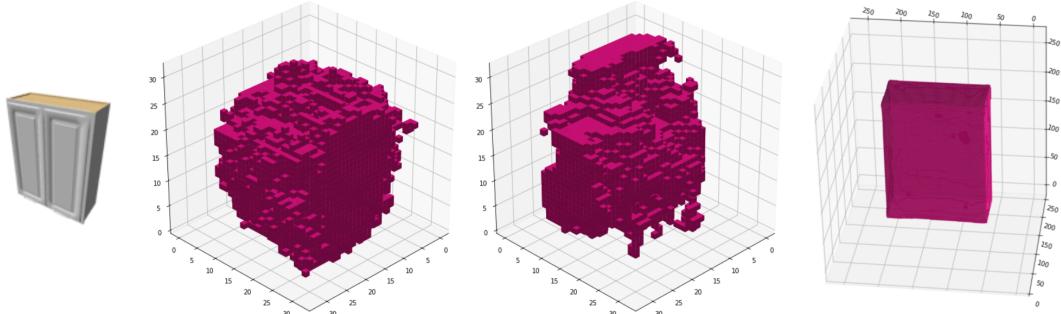


Abbildung 8.1: Darstellung eines Schranks aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

In Abbildung 8.2 wird eine weitere Kategorie des ShapeNet Core Datasets getestet: Ein Tisch. Rechts des Tisches sitzt das Ergebnis der Baseline. Auch hier ist das Objekt um den Mittelpunkt zentriert, doch der Tisch selbst ist nicht erkennbar. Das Resultat des MobileNetV2 Netzes zeigt erste Anzeichen für einen Tisch. Dabei sind die vorderen Beine

sowie die Tischplatte zu erkennen. Das Ergebnis des IM-NETs ist wieder nahe zu perfekt.

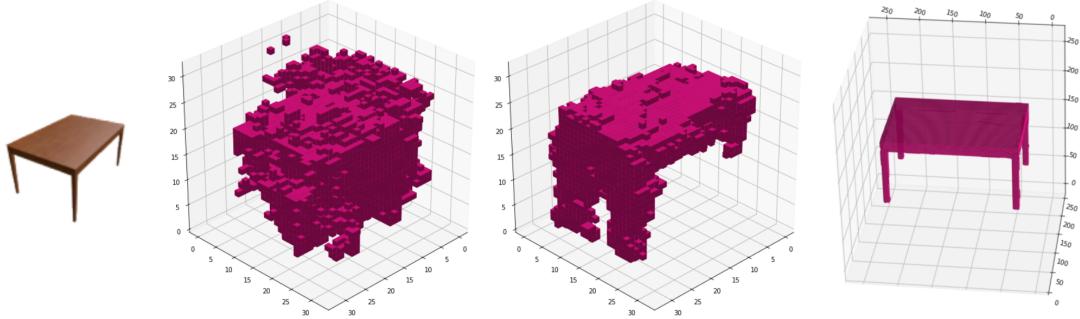


Abbildung 8.2: Darstellung eines Tisches aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

Aus dem ShapeNet Core Dataset wurde ein weiteres Bild als Eingabe getestet. Auf diesem ist ein Stuhl, welcher viele Details besitzt. Somit ist der Stuhl ein sehr komplexer Gegenstand. In Abbildung 8.3 wird dieser dargestellt. Die Ergebnisse der Baseline und des MobileNetV2 sind zwar unterschiedlich, jedoch kann bei keinem Objekt gesagt werden, was dieses darstellen soll. Das Resultat des IM-NETs ist deutlich besser. Ein Stuhl ist erkennbar, jedoch sind die Details der Rückenlehne kaum gegeben. Dennoch ist dies mit Abstand das beste Ergebnis.

Des Weiteren wurde in Abbildung 8.4 ein Auto als Eingabebild getestet. Die Kategorie existiert im ShapeNet Core Dataset. Das genutzte Bild des Autos stammt dabei nicht aus dem Dataset. Somit ist das Bild für jedes Netzwerk ungesehen. Die Baseline generiert eine zentrierte Voxelwolke um den Mittelpunkt des Objektes. Das MobileNetV2 reagiert ähnlich zur Baseline. Dabei ist zu erkennen, dass die Wolke deutlich definierter wirkt. Dennoch ist das Objekt nicht erkennbar. Das IM-NET stellt eine grobe Form des Autos dar. Diese Form kommt der Originalen sehr nah. Dennoch fehlen viele Details, um ein Auto erkennen zu können.

Es wurden auch synthetische Daten getestet, welche eine Kategorie besitzen, die nicht

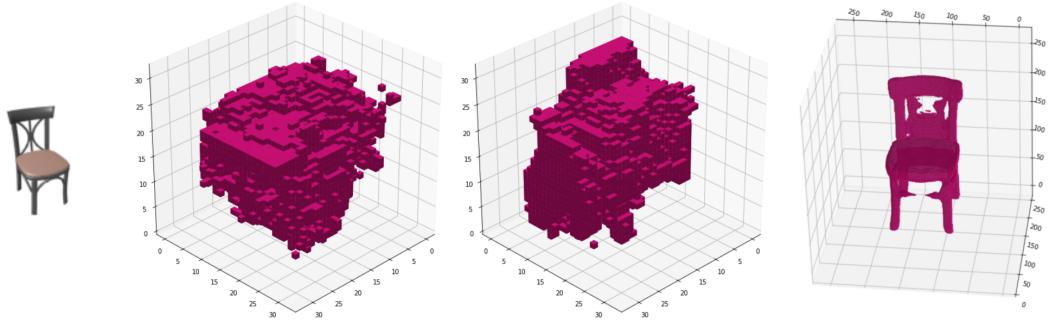


Abbildung 8.3: Darstellung eines komplexen Stuhls aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

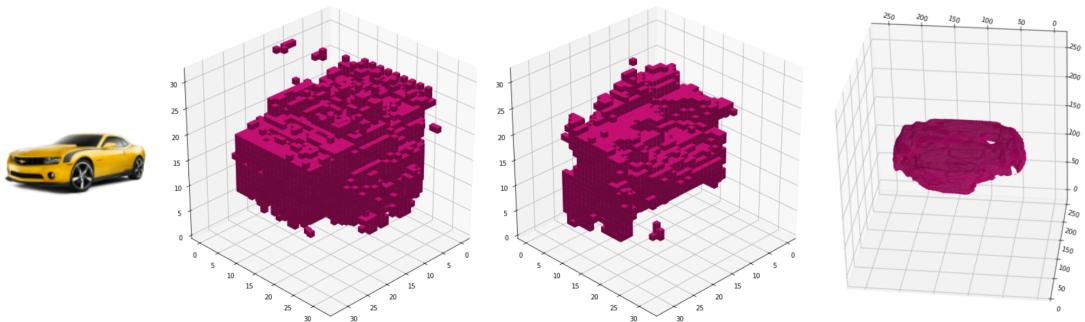


Abbildung 8.4: Darstellung eines Autos als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

im ShapeNet Core Dataset vorhanden ist. Es wurde eine einfach Schraube mit weißen Hintergrund gewählt, wie in Abbildung 8.5 ersichtlich wird. Die Netzwerke wurden nicht auf Schrauben oder ähnlichen Objekten trainiert. Abbildung 8.5 zeigt das die Baseline und das MobileNetV2 mit der Eingabe keine guten Ergebnisse liefern können. Das IM-NET hingegen konnte ein 3D-Modell generieren, welches die grobe Form der Schraube

wiedergeben kann.

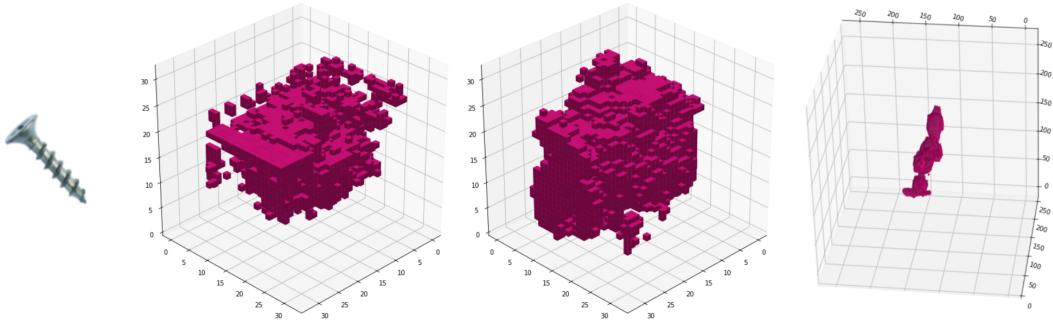


Abbildung 8.5: Darstellung einer Schraube als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

Zuletzt wurde ein Flugzeug, welches in der Aufnahme flog, als Eingabe für die DNNs genutzt. Die Eingabe ist somit nicht synthetisch. Als Hintergrund ist ein wolkenloser Himmel gewählt worden, um das Bild als Eingabe so simpel wie möglich zu halten. Dennoch wird deutlich, dass dieser Hintergrund ein Nachteil bezüglich der 3D-Rekonstruktion ist. Die Baseline generiert kein Objekt, welches eine Wolke um den Mittelpunkt des Objektes bildet. Es wird ein Objekt erzeugt, welches fast den kompletten Raum bezieht. Das MobileNetV2 ergibt auch ein Gebilde, welches nicht als Flugzeug erkannt werden kann. Zuletzt ist das Resultat des IM-NETs auch ein grober Quader mit einem Loch in der Mitte. Die Größe des Quaders wurde wahrscheinlich durch die Größe der sonstigen Objekte des Datasets erlernt. Das Loch in der Mitte ergibt sich wohl durch die Farbe des Flugzeugs. Im ShapeNet Core Dataset sind die Hintergründe, und damit auch die nicht relevanten Daten, weiß. In dieser Eingabe ist das relevante Objekt weiß und der Hintergrund anders farbig. Dies könnte eine mögliche Erklärung für dieses Phänomen sein.

Zusammenfassend wird deutlich erkennbar, dass die 3D-Modelle des IM-NETs am besten generiert werden. Die Baseline ergibt immer eine ähnliche Voxelwolke. Dennoch wird ein Unterschied merkbar, wenn nicht-synthetische Eingaben getätigt werden. Hierbei

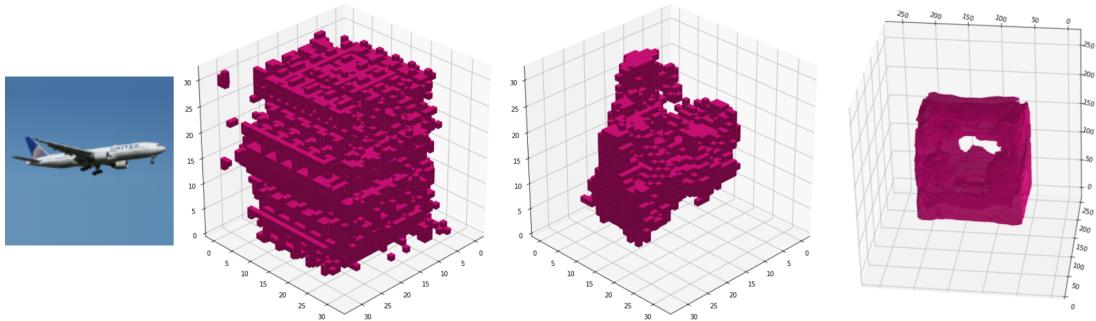


Abbildung 8.6: Darstellung eines nicht-synthetischen Flugzeug als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

werden die Hintergründe als Objekte erkannt und entsprechend in die Generierung mit eingeflossen. Das MobileNetV2 ist im Gegensatz zur Baseline deutlich feiner. Es werden erste Züge des gewünschten Objekts sichtbar. Dennoch ist die Generierung, also der Decoder, selbst ein Problem. Dies wird deutlich, wenn die Ergebnisse des IM-NETs betrachtet werden. Auf den Eingaben, welche aus dem genutzten Dataset verwendet wurden, ist das neuronale Netz sehr gut trainiert. Die Ergebnisse sind nahezu perfekt. Auf Eingaben, welche die selbe Kategorie besitzen, aber nicht im Dataset vorhanden sind, generiert das DNN nicht perfekt. Es werden die groben Züge der Eingabe sichtbar. Auf nicht-synthetischen Daten reagiert das IM-NET nicht gut. Das Ergebnis ist ähnlich zur Baseline bzw. MobileNetV2. Weitere Beispiele können im Anhang Abschnitt A.1 eingesehen werden.

8.2 Genauigkeit der Deep Neural Networks

Die Genauigkeit der neuronale Netze ist ausschlaggebend für die Ergebnisse der Generierung sowie bei der Suche nach dem in Relation stehenden 3D-Objekt. Die Genauigkeit wird über die Intersection over Union (IoU) und Accuracy angegeben. Diese Metriken wurden bereits in Abschnitt 4.8 eingeführt. Bei beiden Metriken gilt: Umso höher der Prozentsatz, umso besser ist das Ergebnis des jeweiligen neuronalen Netzes. In Tabelle 8.1 werden die Werte der einzelnen neuronalen Netze sowie verwandter Arbeiten aufgelistet.

Dabei sind im oberen Abteil der Tabelle 8.1 State-of-the-Art Methoden, welche zur Rekonstruktion aus einem Bild bekannt sind. Bei diesen wurde nur die IoU veröffentlicht und entsprechend keine Accuracy angegeben. Unter diesen besitzt 3D-R2N2 [Cho+16] die niedrigste IoU mit 56,0%. Point Set Generation besitzt die höchste IoU mit 64,0%. Im unteren Abschnitt der Tabelle 8.1 befinden sich die eigens entwickelten bzw. kombinierten neuronalen Netze. Die Baseline weiß eine IoU mit 52,5% und eine Accuracy von 79,1%. Der Wert der IoU wirkt realistisch, wenn die Ergebnisse aus Abschnitt 8.1 für die Baseline betrachtet werden. Mit diesem Modell werden ausschließlich Ballungsgebiete sichtbar. Des Weiteren besitzt das kombinierte Netzwerk mit MobileNetV2 eine IoU von 67,8%. Dieser Wert scheint zu hoch für die Resultate aus Kapitel 8.1. Im Vergleich zu Ergebnissen der State-of-the-Art Methoden, ist dieser Wert deutlich zu hoch. Dies weißt darauf hin, dass bei diesem DNN Overfitting stattgefunden haben könnte. Die Werte des kombinierten Netzes mit IM-NET mit einer IoU von 82,8% und einer Accuracy von 95,8% scheinen exorbitant hoch zu sein. Auch hier wird vermutet das Overfitting stattgefunden hat. Diese Vermutung bestätigen die Ergebnisse aus Abschnitt 8.1. Dabei werden sehr gute Ergebnisse bei Daten aus dem Dataset geliefert. Werden jedoch andere Eingaben getätigt, welche nicht aus dem zum Training genutzten Dataset vorhanden sind, liefern die neuronalen Netze schlechte Ergebnisse.

Neuronales Netz	IoU	Accuracy
3D-R2N2 [Cho+16]	56,0%	-
Occupancy Network [Mes+19]	57,1%	-
Image2Mesh [Pon+18]	57,5%	-
Point Set Generation [FSG17]	64,0%	-
Baseline	52,5%	79,1%
MobileNetV2	67,8%	91,9%
IM-NET	82,8%	95,8%

Tabelle 8.1: Genauigkeit verschiedener neuronalen Netze angegeben in IoU und Accuracy.

8.3 Zeitlicher Aspekt: Inferenz

Da dieses Projekt Menschen dienen soll, ihre Arbeit zu vereinfachen sowie schneller durchführbar zu gestalten, ist der zeitliche Aspekt von Wichtigkeit. In der Praxis sollte ein Nutzer nicht zu lang auf das Ergebnis warten müssen. Ansonsten würde dieser auf die Standardlösung zurück greifen und es wäre kein Vorteil entstanden. Dieser zeitliche Aspekt wird in zwei Abschnitte unterteilt. Zu einem die Inferenz und zum anderen die Suche in vielen 3D-Modellen. In diesem Kapitel wird der zeitliche Aspekt der Inferenz begutachtet. In Tabelle 8.2 ist die Dauer der Inferenzen der einzelnen neuronalen Netze angegeben. Diese wurden auf einem PC, genauer ein Laptop mit der Bezeichnung „Asus VivoBook S510UQ-BQ183T“, und einem mobilen Gerät, ein Smartphone der Bezeichnung „Honor View 20“, durchgeführt. Dabei wurde unterschieden, ob die Ausführung mit oder ohne Laden der Gewichte der neuronalen Netzes stattgefunden hat. Dabei besitzt die Baseline in jeder Konfiguration die niedrigsten Inferenzzeiten. Ob die Gewichte bzw. Parameter geladen werden oder nicht, ergibt keinen merkbaren Unterschied. Bei dem MobileNetV2 ergibt sich auf der PC-Plattform ein großer Unterschied. Das Laden der Gewichte benötigt etwa zwei Sekunden mehr als bei der Inferenz ohne. Auf mobilen Geräten nimmt die Inferenz deutlich weniger Zeit in Anspruch. Um die neuronalen Netze auf einem mobilen Gerät ausführen zu können, mussten diese im Vorfeld in ein lesbares Format konvertiert werden. Durch diese Konvertierung in ein *.tflite*-Format, wurde das Modell komprimiert. Zusätzlich werden Graphenoptimierungen durchgeführt. Dadurch entsteht eine deutlich schnellere Inferenz. Die Inferenz des IM-NETs dauert deutlich länger als die der beiden bereits genannten neuronalen Netze. Die Inferenz braucht für das Vorhersagen eines 3D-Modells über 30 Sekunden, wenn die Gewichte initial geladen werden müssen. Aber auch ohne das Laden der Gewichte werden auf dem PC noch fast 24 Sekunden benötigt. Diese Zeit ist für die Praxis deutlich zu lang und somit nicht geeignet. Auf den mobilen Geräten sind die Zeiten signifikant niedriger. Dennoch sind vier bis sechs Sekunden lang, aber eher für die Praxis geeignet. In Tabelle 8.3 werden die Parameteranzahlen, und dadurch auch die Größen der neuronalen Netze, gegeben. Das IM-NET besitzt mehr als das 48-fache an Parametern der Baseline. Dies spiegelt sich in den Zeiten der Inferenz deutlich wieder. Resultierend steigt die Zeit der Inferenz mit der Parameteranzahl des neuronalen Netzes.

Neuronales Netz	PC (o.)	PC (m.)	Mobiles Gerät (o.)	Mobiles Gerät (m.)
Baseline	0,08s	0,10s	0.10s	0.13s
MobileNetV2	0,82s	3,09s	0.14s	0.18s
IM-NET	23,85s	30,51s	4,11s	6,23s

Tabelle 8.2: Zeitliche Angaben der Inferenz aller neuronaler Netze auf einem PC sowie mobilen Gerät mit (m.) und ohne (o.) laden des Modells.

Neuronales Netz	Encoder	Decoder	Gesamt
Baseline	42.369	72.097	114.466
MobileNetV2	2.421.856	72.097	2.493.953
Im-NET	2.421.856	3.054.593	5.476.449

Tabelle 8.3: Parameteranzahlen der Baseline, MobileNetV2 sowie des IM-NETs, aufgeteilt in En- und Decoder.

8.4 Zeitlicher Aspekt: Ähnlichkeitsmaß

Wie in Abschnitt 8.3 erwähnt, werden zwei Abschnitte zur Messung der gesamt benötigten Zeit für den Endnutzer betrachtet. Die erste ist der zeitliche Aspekt der Inferenz und der zweite die der Suche, also des Ähnlichkeitsmaßes. Im Folgenden wird das Ähnlichkeitsmaß und seine zeitlichen Aspekte untersucht. Für das Ähnlichkeitsmaß werden zwei Metriken betrachtet: IoU und die Hausdorffer Distanz. Beide Metriken wurden in Kapitel 7.6 bereits eingeführt. Die Dauer der Berechnungen jedes Maßes wurde, wie bereits in Abschnitt 8.3, aufgeteilt. Dafür wurde auf einem PC sowie einem mobilen Gerät getestet. Außerdem wurden die Berechnungen mit 3D-Modellen in einer Auflösung von 32^3 und 256^3 durchgeführt. Die zeitlichen Angaben, welche in der Tabelle 8.4 angegeben werden, wurden für die Suche eines Modells in weiteren 100 Modellen ausgeführt. Die Durchführung der Suche in Modellen der Auflösung 32^3 ist zeitlich sehr geeignet, um in der Praxis eingeführt zu werden. Im Gegensatz dazu benötigen die Berechnungen mit Modellen der Auflösung 256^3 deutlich länger. Eine Suche der Dauer von 23 Sekunden ist sehr lang, vor allem da dabei nicht die Inferenz berücksichtigt wird. Um unter einer halben Minute zu bleiben, kann im momentanen Zustand kein detailliertes Modell generiert werden. Ansonsten würde der Zeitraum von einer halben Minute überschritten werden. Andererseits wird dadurch das Ergebnis und somit auch die Suche negativ beeinflusst. Da alle Zeitangaben

für die Größe 256^3 größer sind, sind resultierend keine der Konfigurationen geeignet, um einem Nutzen für den Anwender zu haben.

Ähnlichkeitsmaß	PC (32)	PC (256)	Mobiles Gerät (32)	Mobiles Gerät (256)
IoU	0,80s	23,52s	1,13s	72,18s
Hausdorffer Distanz	0,24s	56,36s	2,95s	90,10s

Tabelle 8.4: Zeitliche Angaben für die Ausführung des Ähnlichkeitsmaßes auf einem PC sowie mobilen Gerät mit 100 Modellen der Auflösung 32^3 (32) sowie 256^3 (256).

8.5 Schwellwertbestimmung des Ähnlichkeitsmaßes

Um die zeitlichen Aspekte der Suche in Abschnitt 8.4 so gering wie möglich zu halten, wird ein Schwellwert für die Suche eingeführt. Unterschreitet bzw. überschreitet das Ergebnis des Ähnlichkeitsmaßes zwischen zwei Objekten einen bestimmten Schwellwert, so werden diese Objekte als identisch gesehen und die restliche Suche abgebrochen. Dadurch sollen so wenig Objekte wie möglich in der Suche durchlaufen werden und resultierend eine Zeiteinsparung ergeben. Es wurden Schwellwerte für Modelle der Auflösung von 32^3 und 256^3 festgelegt. Außerdem wurden für IoU und Hausdorffer Distanz einzelne Schwellwerte bestimmt. In Tabelle 8.5 werden alle Schwellwerte jeglicher Konfiguration dargestellt. Dabei soll für die IoU der Wert gleich oder überschritten werden. Für Modelle mit einer Auflösung von 32^3 ist der Schwellwert höher als für Modelle der Auflösung 256^3 . Dies liegt daran, dass Modelle der Größe 32^3 deutlich weniger Werte besitzen. Resultierend ist das Modell selbst grob und kann leicht mit ähnlichen Objekten übereinstimmen. Modelle mit einer Größe von 256^3 sind sehr viel feiner und besitzen dadurch Details, welche in einem kleineren Format nicht dargestellt werden können. Entsprechend ist der Schwellwert geringer, um das ähnlichste Objekt finden zu können, falls das in Relation stehende Objekt nicht in der Datenbank vorhanden ist. Bei der Haussdorfer Distanz wurde dieses Prinzip auch angewendet. Der einzige Unterschied dabei besteht darin, dass die Distanz besser ist, umso kleiner diese ist. Somit ist ein Objekt gleich dem gesuchten, wenn dieses den Schwellwert von 10 bei der Hausdorffer Distanz für Modelle der Größe

32^3 unterschreitet oder gleich ist.

Die Werte selbst wurden durch Experimente herausgefunden. Dabei wurde mit generierten Objekten in den originalen Objekten des Datasets gesucht. Da dabei der Wert von 100% bei IoU bzw. der Wert Null bei der Hausdorffer Distanz nie erreicht wurde, wurden die nächst stehenden Werte als Schwellwert genutzt, um ähnliche Objekte zu adressieren.

Modellaufösung	IoU	Hausdorffer Distanz
32^3	97%	10
256^3	85%	100

Tabelle 8.5: Schwellwertbestimmung der Ähnlichkeitsmaße IoU in Prozent und Hausdorffer Distanz für Modelle mit einer Auflösung von 32^3 und 256^3 .

9 Schlussfolgerung und Ausblick

Ziel dieser Arbeit war es, einen Prototyp für mobile Geräte zu entwickeln, welcher aus einer 2D-Eingabe das in Relation stehende 3D-Objekt bzw. dessen Information in einer Sammlung von 3D-Objekten findet. Für die Generierung von 3D-Objekten aus 2D-Eingaben wurden drei neuronale Netze mit je einer En- und Decoder Architektur entwickelt und anschließend evaluiert. Das erste neuronale Netz ist die Baseline, welche aus einem selbst entwickelten En- bzw. Decoder besteht. Die zweite Architektur wurde aus dem MobileNetV2 [San+18] und dem selbst entwickelten Decoder gestaltet. Das letzte DNN besteht aus dem MobileNetV2 als Encoder und dem IM-NET [CZ19a] als Decoder. Die DNNs wurden allesamt mit dem ShapeNet Core Dataset [Cha+15] trainiert. Alle neuronalen Netze wurden für mobile Geräte aufbereitet. Dabei wurden die Modellierungsherausforderungen aus Kapitel 6 bei der Erstellung der erwähnten neuronalen Netze berücksichtigt.

Die Modellierungsherausforderungen für die 3D-Rekonstruktion, welche aus einem einzelnen Bild hervorgehen wurden teils erfolgreich bestritten. Dadurch kann die Formkomplexität von Objekten gewährleistet werden, indem Objekte der gleichen Kategorien untereinander einzigartig sind. Außerdem können komplexe Formen, wie beispielsweise Autos mit Seitenspiegeln, dargestellt werden. Zusätzlich wurden Unsicherheiten bei der Rekonstruktion verhindert, in dem Eingaben aus verschiedenen Perspektiven genutzt worden sind. So erlernt ein neuronales Netz einen Erfahrungsschatz, welches das Wissen von 3D-Objekten aus jedem Blickwinkel, betrifft. Die Rekonstruktion von detaillierten Objekten erweist sich schwierig. Können detaillierte Objekte nahezu perfekt rekonstruiert werden, ist ein Overfitting sehr wahrscheinlich. Dies konnte durch die Experimente bestätigt werden. Der Speicherbedarf der DNNs wird durch die Anzahl der Parameter festgelegt. Diese sind sehr unterschiedlich ausgefallen. Umso weniger Parameter das neuronale Netz besitzt, umso kürzer ist die Latenz und somit die Rechenzeit. Leider werden viele Parameter gebraucht, um eine gute Rekonstruktion zu gewährleisten. Entsprechend wurde auf diese Modellierungsherausforderung kaum bis gar nicht eingegangen. Für die Herausforderung des Datasets wurde ein passendes, in diesem Bereich wohl bekanntes Dataset gefunden. Dieses bildet nur Gegenstände ab, welche sich im Bereich des täglichen Lebens befinden. Dennoch ist es vollkommen ausreichen, um einen Prototyp zu trainieren und mit diesen die resultierenden Werten zu evaluieren. Zuletzt wurde für die 3D-Repräsentation der 3D-Modelle die Form der Voxel sowie die implizite Repräsentation

aus verschiedenen Gründen gewählt. Zur Veranschaulichung bzw. zum Vergleich der einzelnen Modelle, wurde die implizite Darstellung auch als Voxel Darstellung abgebildet. Zur weiteren Suche in einer Sammlung von 3D-Modellen, eignet sich dennoch die implizite Repräsentation besser, da diese in der Größe nicht beschränkt ist. Somit kann diese deutlich feinere Züge der Objekte generieren und schlussendlich eine bessere Übereinstimmung von 3D-Objekten während der Suche finden.

Des Weiteren wurden Modellierungsherausforderungen der DNNs für mobile Geräte definiert. Die Wahl, welche in dieser Arbeit gefällt wurde, zur Plattform bzw. Bibliothek wurde im Nachhinein nicht bereut. Die Entwicklung mit TensorFlow und TensorFlow Lite konnte gut umgesetzt werden. Die Dokumentation sowie die Community sind beide gut geeignet und bieten einen guten Einstieg in das Thema der Entwicklung für Deep Learning. In der Zeit der Masterarbeit war es möglich, sich ein großes Wissen dieses Bereiches anzueignen. Für den Speicherbedarf, welcher unweigerlich in Verbindung mit der Berechnungszeit und somit auch dem Stromverbrauch steht, wurde nicht speziell betrachtet. Es wurde je eine App für ein neuronales Netz gebaut. Diese haben eine Speicherspanne von etwa 50MB bis 250MB. Diese Werte liegen trotzdem im Rahmen und benötigen nicht allzu viel Speicher. Dabei soll beachtet werden, dass in diesen Apps nur die Vorhersage einzelner 3D-Objekte und die Suche nach 3D-Objekten implementiert sind. Eine Anzeige von 3D-Modellen oder ähnlichen wurde dabei nicht entwickelt. Zusammenfassend kann jedoch gesagt werden, dass so eine App für den normalen Gebrauch geeignet ist.

Anschließend, nach der Entwicklung der neuronalen Netze, wurde die allgemeine Suche von einzelnen 3D-Objekten in einer Sammlung von 3D-Objekten durch Ähnlichkeitsmaße eingeführt. Hierbei wurde sich für die IoU und Hausdorffer Distanz entschieden. Daraufhin wurde mit den resultierenden Systemen Experimente durchgeführt. Dabei wurde erkannt, dass noch viel Potential in der Optimierung der neuronalen Netze liegt.

Bei der Generierung von 3D-Objekten aus differierender Eingaben sowie der Genauigkeit der einzelnen neuronalen Netze, wurde festgestellt, dass diese bei ungesehenen Daten schlechte Ergebnisse liefern. Dieses Phänomen kann auf Overfitting zurück geführt werden. Denn auf gesehenen Daten, also Eingaben, welche aus dem ShapeNet Core Dataset stammen, sind die Ergebnisse nahe zu perfekt. Für die Beseitigung dieses Problems, existieren verschiedene Möglichkeiten. Eine dieser Hilfsmittel wäre eine Erweiterung des Datasets, eine Reduktion der Merkmale oder die Nutzung verschiedener Regularisierungen. Zusätzlich sind die zeitlichen Aspekte der Inferenz und der Ähnlichkeitmaße nicht für

die Praxis zumutbar. Um die Inferenz zeitlich zu verkürzen, müssten weniger Parameter genutzt werden. Dies kann durch viele Möglichkeiten umgesetzt werden. Darunter könnte jedoch das Ergebnis der Rekonstruktion leiden. Einen guten Ausgleich zwischen zeitlichen Aspekt und Parameteranzahl zu erzielen, ist eine große Schwierigkeit und kann nur durch weitere Versuche ermittelt werden. Der zeitliche Aspekt der Ähnlichkeitssmaße hingegen, könnte durch Optimierungen im Algorithmus erfolgen. Es wurden Implementationen genutzt, welche direkt nach der jeweiligen Formel umgesetzt sind. Diese könnten durch weitere Aspekte optimiert oder durch andere Algorithmen, welche schneller konvergieren, ersetzt werden.

In dieser Arbeit konnte nur ein geringer Teil des Forschungsgebietes der 3D-Rekonstruktion aus einem Bild untersucht werden. Es existieren viele weitere Möglichkeiten eine in verschiedener Hinsicht effizientere Generierung von 3D-Modellen umzusetzen. Beispielsweise wäre interessant statt einer En- und Decoder Architektur einen Variational Autoencoder (VAE) oder ein Generative Adversarial Network (GAN) zu nutzen. Somit könnte die Wahrscheinlichkeit des Overfittings bekämpft werden und bessere Ergebnisse auf ungesehenen Daten liefern. Des Weiteren wären verschiedene Konfigurationen aller Schichten bzw. deren Werte, Aktivierungsfunktionen, Kostenfunktionen, Modell Optimierungen und ähnlichen informativ. Ergänzend wäre es für die praktische Nutzung von Vorteil, die neuronalen Netze direkt mit einem Dataset zu trainieren, welche die Objekte der Praxis beinhalten. Außerdem könnte eine deutlich größere Epochenanzahl bzw. längere Trainingsdauer zu besseren Ergebnissen führen. Weiterhin wäre eine hierarchische Anordnung in den 3D-Objekten von großen Nutzen. Somit könnten einzelne Bauteile bzw. Bauteilgruppen zur übergeordneten Maschine schneller in Relation gebracht werden, um dadurch den zeitlichen Aspekt der Suche zu verringern.

Abschließend betrachtet sind neuronale Netze und deren Zusammenhänge ein stetig wachsendes und besser erforschtes Thema. Trotzdem sind Verbesserungen und Veränderungen jederzeit möglich und erwünscht. Deep Learning bietet einen bisher nicht ausgeschöpfte Potenzial, welches dennoch immer deutlicher wird. Auch in den nächsten Jahren wird dieser Bereich die Welt begleiten und bereichern.

Literaturverzeichnis

- [Ban19] Bansal, Shivam. *3D Convolutions : Understanding + Use Case*. 2019. URL: <https://www.kaggle.com/shivamb/3d-convolutions-understanding-use-case> (Abruf am 28.04.2021) (Zitiert auf Seite 29).
- [Cha+15] Chang, Angel X. u. a. *ShapeNet: An Information-Rich 3D Model Repository*. Techn. Ber. arXiv:1512.03012 [cs.GR]. Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015 (Zitiert auf Seiten 10, 12, 14, 35, 37–39, 72, 83).
- [Cha20] Chan, Chun Hei Michael. *Step by Step Implementation: 3D Convolutional Neural Networks in Keras*. März 2020. URL: <https://towardsdatascience.com/step-by-step-implementation-3d-convolutional-neural-network-in-keras-12efbdd7b130> (Abruf am 28.04.2021) (Zitiert auf Seite 29).
- [Cho+14] Cho, Kyunghyun u. a. „Learning phrase representations using RNN encoder-decoder for statistical machine translation“. In: *arXiv preprint arXiv:1406.1078* (2014) (Zitiert auf Seite 46).
- [Cho+16] Choy, Christopher B u. a. „3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction“. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2016 (Zitiert auf Seiten 11–14, 35–37, 39, 48, 49, 78).
- [Cho17] Chollet, François. „Xception: Deep learning with depthwise separable convolutions“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, S. 1251–1258 (Zitiert auf Seite 62).
- [Com21] Community., The SciPy. *NumPy*. Jan. 2021. URL: <https://numpy.org/doc/stable/contents.html> (Abruf am 02.05.2021) (Zitiert auf Seite 48).
- [com21] community, The SciPy. *SciPy*. Juli 2021. URL: https://docs.scipy.org/doc/scipy/getting_started.html (Zitiert auf Seite 70).
- [Cub87] Cubes, Marching. „A high resolution 3D surface construction algorithm“. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. New York: Association for Computing Machinery. 1987, S. 163–69 (Zitiert auf Seiten 40, 67).

- [CZ19a] Chen, Z. und Zhang, H. „Learning Implicit Fields for Generative Shape Modeling“. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, S. 5932–5941 (Zitiert auf Seiten 7, 40, 47, 66–68, 83).
- [CZ19b] Chen, Zhiqin und Zhang, Hao. *point sampling code for IM-NET*. Nov. 2019. URL: https://github.com/czq142857/IM-NET/tree/master/point_sampling (Zitiert auf Seiten 49, 69).
- [Den19] Deng, Yunbin. „Deep learning on mobile devices: a review“. In: *Mobile Multimedia/Image Processing, Security, and Applications 2019*. Bd. 10993. International Society for Optics und Photonics. 2019, 109930A (Zitiert auf Seite 44).
- [DN19] Deru, Matthieu und Ndiaye, Alassane. *Deep Learning mit TensorFlow, Keras und TensorFlow.js*. Rheinwerk, 2019 (Zitiert auf Seiten 9, 15–18, 28, 29).
- [Eve+10] Everingham, M. u. a. „The PASCAL Visual Object Classes (VOC) Challenge“. In: *International Journal of Computer Vision 88*. 2010, S. 303–338 (Zitiert auf Seite 10).
- [Fen+18] Feng, Yutong u. a. *MeshNet: Mesh Neural Network for 3D Shape Representation*. 2018 (Zitiert auf Seite 6).
- [FSG17] Fan, Haoqiang, Su, Hao und Guibas, Leonidas J. „A point set generation network for 3d object reconstruction from a single image“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, S. 605–613 (Zitiert auf Seiten 36, 37, 78).
- [Fu+21] Fu, Kui u. a. „Single image 3D object reconstruction based on deep learning: A review“. In: *Multimedia Tools and Applications 80* (Jan. 2021), S. 1–36 (Zitiert auf Seiten 8, 12, 41–43).
- [GB10] Glorot, Xavier und Bengio, Yoshua. „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop und Conference Proceedings. 2010, S. 249–256 (Zitiert auf Seite 24).
- [GBK05] Guthe, Michael, Borodin, Pavel und Klein, Reinhard. „Fast and accurate Hausdorff distance calculation between meshes“. In: (2005) (Zitiert auf Seite 69).

- [Gér19] Géron, Aurélien. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019 (Zitiert auf Seiten 20, 24–32, 37, 46, 59).
- [GMJ19] Gkioxari, Georgia, Malik, Jitendra und Johnson, Justin. „Mesh r-cnn“. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, S. 9785–9795.
- [He+15] He, Kaiming u. a. „Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification“. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Dez. 2015, S. 1026–1034 (Zitiert auf Seite 25).
- [He+16] He, Kaiming u. a. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, S. 770–778 (Zitiert auf Seiten 30, 39, 64).
- [HKK17] Han, Dongyoon, Kim, Jiwhan und Kim, Junmo. „Deep pyramidal residual networks“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, S. 5927–5935 (Zitiert auf Seite 64).
- [HLB19] Han, Xianfeng, Laga, Hamid und Benamoun, Mohammed. „Image-based 3D object reconstruction: State-of-the-art and trends in the deep learning era“. In: *IEEE transactions on pattern analysis and machine intelligence* (2019) (Zitiert auf Seiten 5, 7, 33).
- [Hor91] Hornik, Kurt. „Approximation capabilities of multilayer feedforward networks“. In: *Neural networks* 4.2 (1991), S. 251–257 (Zitiert auf Seite 68).
- [How+17] Howard, Andrew G u. a. „Mobilenets: Efficient convolutional neural networks for mobile vision applications“. In: *arXiv preprint arXiv:1704.04861* (2017) (Zitiert auf Seiten 61, 62).
- [How+19] Howard, Andrew u. a. „Searching for mobilenetv3“. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, S. 1314–1324 (Zitiert auf Seite 61).
- [HTM17] Häne, Christian, Tulsiani, Shubham und Malik, Jitendra. „Hierarchical surface prediction for 3d object reconstruction“. In: *2017 International Conference on 3D Vision (3DV)*. IEEE. 2017, S. 412–420 (Zitiert auf Seite 49).
- [Inc21] Inc, Trimble. *3D Warehouse*. März 2021. URL: <https://3dwarehouse.sketchup.com/> (Abruf am 28.04.2021) (Zitiert auf Seite 12).

- [IS15] Ioffe, Sergey und Szegedy, Christian. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *International conference on machine learning*. PMLR. 2015, S. 448–456 (Zitiert auf Seite 25).
- [Jia+18] Jiang, Li u. a. „GAL: Geometric Adversarial Loss for Single-View 3D-Object Reconstruction“. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Sep. 2018 (Zitiert auf Seite 37).
- [Joh+17] Johnston, Adrian u. a. „Scaling cnns for high resolution volumetric reconstruction from a single image“. In: *Proceedings of the IEEE International Conference on Computer Vision Workshops*. 2017, S. 939–948 (Zitiert auf Seite 36).
- [KB14] Kingma, Diederik P und Ba, Jimmy. „Adam: A method for stochastic optimization“. In: *arXiv preprint arXiv:1412.6980* (2014) (Zitiert auf Seite 28).
- [Koc+19] Koch, S. u. a. „ABC: A Big CAD Model Dataset for Geometric Deep Learning“. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, S. 9593–9603 (Zitiert auf Seiten 9, 10).
- [Kri16] Kristiadi, Agustinus. *Variational Autoencoder: Intuition and Implementation*. Dez. 2016. URL: <https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/> (Abruf am 28.04.2021).
- [LeC+98] LeCun, Yann u. a. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (1998), S. 2278–2324 (Zitiert auf Seite 53).
- [Lin+17] Lin, Tsung-Yi u. a. „Feature pyramid networks for object detection“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, S. 2117–2125 (Zitiert auf Seite 38).
- [Lin01] Linsen, Lars. *Point cloud representation*. Techn. Ber. 2001-3. Fakultät für Informatik, Universität Karlsruhe (TH), 2001 (Zitiert auf Seite 7).
- [LLC21] LLC, Google. *Android Studio*. Juli 2021. URL: <https://developer.android.com/studio> (Zitiert auf Seite 70).
- [LPT13] Lim, Joseph J., Pirsiavash, Hamed und Torralba, Antonio. „Parsing IKEA Objects: Fine Pose Estimation“. In: *ICCV* (2013) (Zitiert auf Seite 10).
- [Mar+15] Martin Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <https://www.tensorflow.org/> (Zitiert auf Seiten 42, 47, 51, 57, 70).

- [Mar+21] Martin Abadi u. a. *tf.keras.applications.MobileNetV2*. März 2021. URL: https://www.tensorflow.org/api_docs/python/tf/keras/applications/MobileNetV2 (Zitiert auf Seite 48).
- [Mel18] Melesse, Michael. „3D Reconstruction with Neural Networks“. Princeton University Senior Theses. Princeton University Computer Science Department, Mai 2018. URL: <http://arks.princeton.edu/ark:/88435/dsp019306t204j> (Zitiert auf Seiten 13, 30, 48).
- [Mes+19] Mescheder, Lars u. a. „Occupancy Networks: Learning 3D Reconstruction in Function Space“. In: *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 2019 (Zitiert auf Seiten 5–8, 39, 78).
- [MHN13] Maas, Andrew L, Hannun, Awni Y und Ng, Andrew Y. „Rectifier nonlinearities improve neural network acoustic models“. In: *Proc. icml*. Bd. 30. 1. Citeseer. 2013, S. 3 (Zitiert auf Seiten 21, 22).
- [Mil95] Miller, George A. „WordNet: A Lexical Database for English“. In: *Commun. ACM* 38.11 (Nov. 1995), S. 39–41 (Zitiert auf Seite 11).
- [Mur12] Murphy, Kevin P. *Machine learning: a probabilistic perspective*. MIT press, 2012 (Zitiert auf Seiten 19, 20).
- [NZ18] Nguyen, C.N. und Zeigermann, O. *Machine Learning : kurz & gut*. OReillys Taschenbibliothek. Dpunkt.Verlag GmbH, 2018. ISBN: 9783960090526 (Zitiert auf Seiten 20–22, 33).
- [Pon+18] Pontes, Jhony K u. a. „Image2mesh: A learning framework for single image 3d reconstruction“. In: *Asian Conference on Computer Vision*. Springer. 2018, S. 365–381 (Zitiert auf Seiten 38, 78).
- [Qi+17] Qi, Charles R u. a. „Pointnet: Deep learning on point sets for 3d classification and segmentation“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, S. 652–660 (Zitiert auf Seite 37).
- [Ren+16] Ren, Shaoqing u. a. „Faster R-CNN: towards real-time object detection with region proposal networks“. In: *IEEE transactions on pattern analysis and machine intelligence* 39.6 (2016), S. 1137–1149 (Zitiert auf Seite 38).
- [Roe+19] Roeglinder, Maximilian u. a. „Smart Devices erfolgreich in Produktionsprozessen integrieren“. In: (Sep. 2019) (Zitiert auf Seite 1).
- [Ros57] Rosenblatt, Franck. *The Perceptron, a Perceiving and Recognizing Automaton*. Cornell Aeronautical Laboratory, 1957 (Zitiert auf Seite 15).

- [San+18] Sandler, Mark u. a. „Mobilenetv2: Inverted residuals and linear bottlenecks“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, S. 4510–4520 (Zitiert auf Seiten 47, 61–66, 83).
- [Sar19] Sartorius, Gerhard. „Distanz- und Ähnlichkeitsmaße“. In: *Erfassen, Verarbeiten und Zuordnen multivariater Messgrößen: Neue Rahmenbedingungen für das Nächste-Nachbarn-Verfahren*. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 111–133. ISBN: 978-3-658-23576-5 (Zitiert auf Seiten 33, 34).
- [SCH20] SCHEMA. *SCHEMA CDS*. Okt. 2020. URL: <https://www.schema.de/software/schema-cds> (Abruf am 28.04.2021) (Zitiert auf Seite 1).
- [Sri+14] Srivastava, Nitish u. a. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15.56 (2014), S. 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html> (Zitiert auf Seite 27).
- [Sun+18] Sun, X. u. a. „Pix3D: Dataset and Methods for Single-Image 3D Shape Modeling“. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, S. 2974–2983 (Zitiert auf Seiten 10, 11).
- [SZ18] Sekerak, Robin und Zasler, Nathan D. „Voxel“. In: *Encyclopedia of Clinical Neuropsychology*. Hrsg. von Kreutzer, Jeffrey S., DeLuca, John und Caplan, Bruce. Cham: Springer International Publishing, 2018, S. 3667–3668. ISBN: 978-3-319-57111-9 (Zitiert auf Seite 5).
- [Sze+15] Szegedy, Christian u. a. „Going deeper with convolutions“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, S. 1–9.
- [Tai+14] Taigman, Yaniv u. a. „Deepface: Closing the gap to human-level performance in face verification“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, S. 1701–1708 (Zitiert auf Seiten 33, 34).
- [Tat+19] Tatarchenko, Maxim u. a. „What do single-view 3d reconstruction networks learn?“ In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, S. 3405–3414 (Zitiert auf Seite 43).
- [TLK09] Tang, Min, Lee, Minkyung und Kim, Young J. „Interactive Hausdorff distance computation for general polygonal models“. In: *ACM Transactions on Graphics (TOG)* 28.3 (2009), S. 1–9 (Zitiert auf Seite 69).

- [War43] Warren S. McCulloch, Walter H. Pitts. „A Logical Calculus of the Ideas Immanent in Nervous Activity“. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 115–133 (Zitiert auf Seite 15).
- [Wu+16] Wu, Jiajun u. a. „Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling“. In: (2016).
- [XMS14] Xiang, Y., Mottaghi, R. und Savarese, S. „Beyond PASCAL: A benchmark for 3D object detection in the wild“. In: *IEEE Winter Conference on Applications of Computer Vision*. 2014, S. 75–82 (Zitiert auf Seiten 9, 10).
- [Zha+17] Zhang, Dejun u. a. „An efficient approach to directly compute the exact Hausdorff distance for 3D point sets“. In: *Integrated Computer-Aided Engineering* 24.3 (2017), S. 261–277 (Zitiert auf Seite 70).
- [Zha+18] Zhang, Xiangyu u. a. „Shufflenet: An extremely efficient convolutional neural network for mobile devices“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, S. 6848–6856 (Zitiert auf Seiten 62, 63).
- [Zhi+15] Zhirong Wu u. a. „3D ShapeNets: A deep representation for volumetric shapes“. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, S. 1912–1920 (Zitiert auf Seiten 9, 10).

A Anhang

A.1 Weitere Beispiele für Vorhersagen differierender Eingabe aller neuronaler Netze

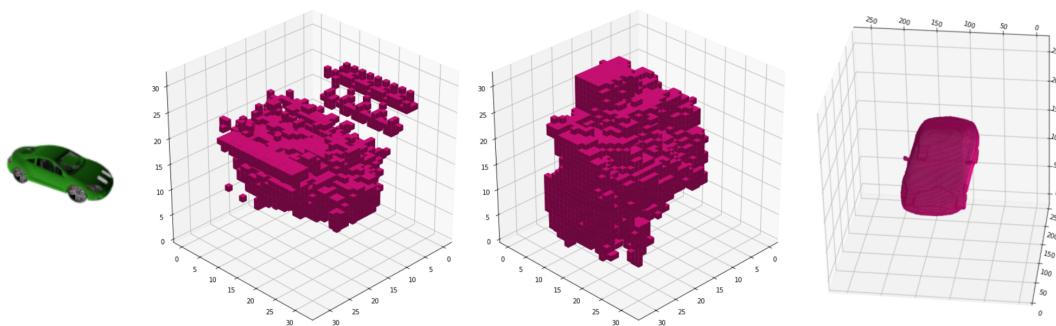


Abbildung A.1: Darstellung eines Autos aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

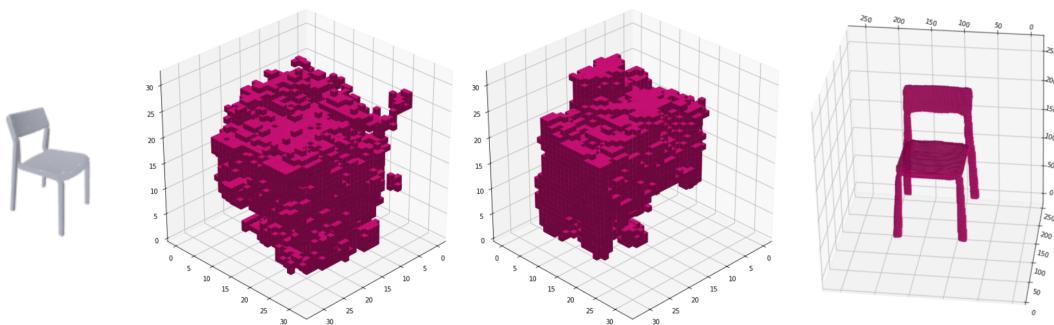


Abbildung A.2: Darstellung eines Stuhls aus dem ShapeNet Core Dataset als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

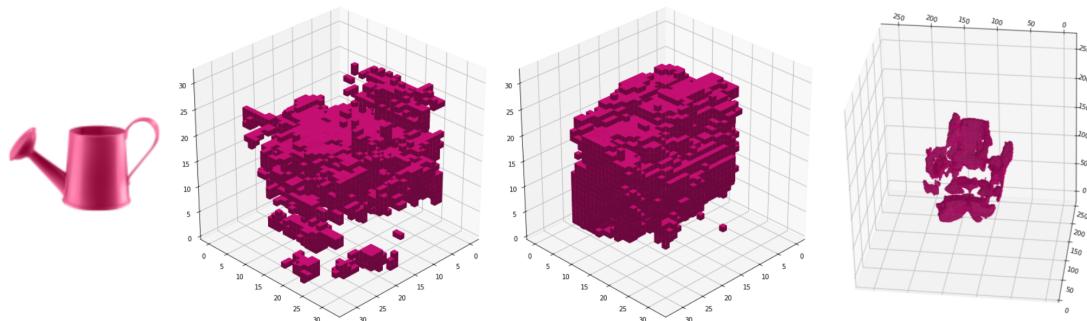


Abbildung A.3: Darstellung einer synthetischen Eingabe einer Gießkanne als Bild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

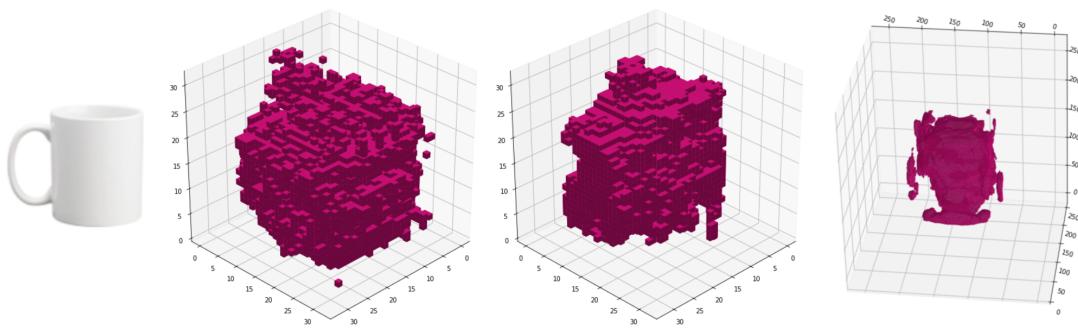


Abbildung A.4: Darstellung einer synthetischen Eingabe einer Tasse als Bild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

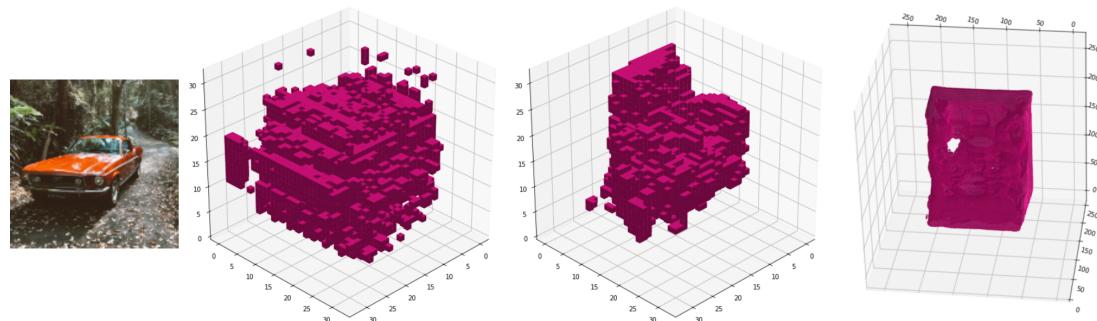


Abbildung A.5: Darstellung eines nicht-synthetischen Autos als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

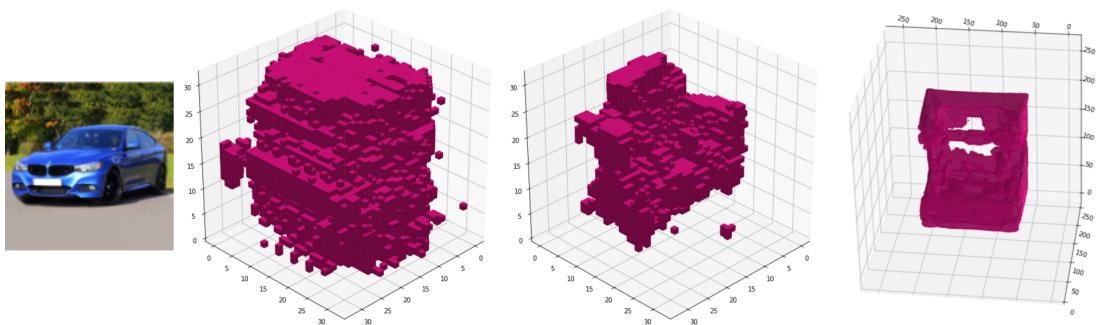


Abbildung A.6: Darstellung eines weiteren nicht-synthetischen Autos als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.

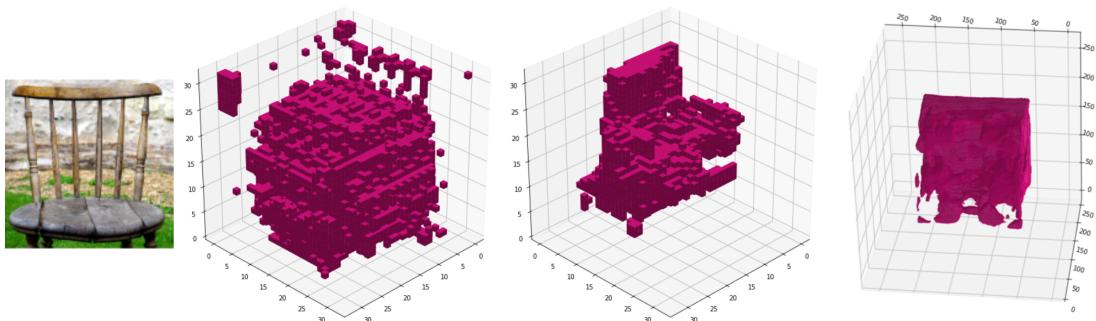


Abbildung A.7: Darstellung eines nicht-synthetischen, abgeschnittenen Stuhl als Eingabebild mit in Relation stehenden Ergebnissen der Baseline, des MobileNetV2 sowie des IM-NETs.