Entwicklung einer Gestenerkennung mithilfe von Stereokameras mit ROS

Masterarbeit

zur Erlangung des akademischen Grades

Master of Engineering (M. Eng.)

Oliver Bosin

Betreuer:

Prof. Dr. rer. nat. Norbert Oswald

Tag der Abgabe: 11.10.2020

Universität der Bundeswehr München
Fakultät für Elektrotechnik und Technische Informatik
Institut für Verteilte Intelligente Systeme

Neubiberg, Oktober 2020

Erklärung

gemäß Beschluss des Prüfungsausschusses für die Fachhochschulstudiengänge der UniBwM vom 25.03.2010

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.

Neubiberg, den 26. Juni 2019

Oliver Bosin

Erklärung

gemäß Beschluss des Prüfungsausschusses für die Fachhochschulstudiengänge der UniBwM vom 25.03.2010

Der Speicherung meiner Masterarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Neubiberg, den 26. Juni 2019

Oliver Bosin

Abstrakt

Gegenstand dieser Bachelorarbeit ist die Entwicklung eines Gestenerkennungssystems, mit Hilfe von der Stereokamera ”Kinect” von ”MICROSOFT”, mit ROS. Es wird zuerst auf alle Komponenten, die den Systemaufbau ausmachen, eingegangen. Das Konzept hinter dem System ist ebenfalls Teil der Ausführungen. Zu den wichtigsten Komponenten, dies sind die Stereokamera, ROS, ”MoveIt!” und ”OpenNI”, werden danach die Grundlagen erklärt. Auf die Beschreibung des Systems und seiner Komponenten folgen die Ausführungen zur Implementation und Inbetriebnahme des Systems. Nachfolgend wird die Implementation und Funktionsweise von zwei Paketen, die im Zuge dieser Bachelorarbeit entwickelt wurden, erklärt. Diese Pakete zeigen mögliche Anwendungen, für das Gestenerkennungssystem auf. Die Ergebnisse, Probleme und Erkenntnisse dieser Bachelorarbeit, werden am Schluss diskutiert. Ein Ausblick wird ebenfalls gegeben.
Für Studierende und Interessierte, welche den Einstieg in ROS und die natürliche Interaktion suchen, bietet diese Arbeit eine erste Orientierung.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

API Application Programming Interface

RGB Rot Grün Blau

NI Natural Interaction

XML Extensible Markup Language

URDF Unified Robot Description Format

ROS Robot Operating System

FPGA Field Programmable Gate Array

PWM Pulsweitenmodulation

1. Einleitung

1.1. Motivation

Im ersten Weltkrieg waren die gefährlichsten Waffen, die chemischen Waffen. Im kalten Krieg hat die Macht des Atoms, und sein Schrecken aus Nagasaki und Hiroshima, einen dritten Weltkrieg fast schon absurd gemacht. Das, im Normalfall, nur Staaten auf diese Waffen Zugriff haben, wird die meisten Menschen beruhigen. Mit den DeepFake Videos, welche immer wieder in den Medien thematisiert werden, hat nun jedermann Zugriff auf etwas das potentiell als Waffe eingesetzt werden kann. Ob es nun eingesetzt wird um einem Konkurrenzunternehmen zu schaden, zum Zwecke der Propaganda, zur Wahlbeeinflussung oder um einen kalten Konflikt zwischen Konfliktparteien wieder anzufachen, die vorstellbaren Szenarien sind hier vielfältig. Zu solchen Bedrohungen, werden immer Gegenmaßnahmen gesucht. Bei den Atomwaffen wurde mit Abschreckung gearbeitet, da die Folgen eines Atomwaffeneinsatzes bekannt, gefürchtet und sofort sichtbar sind. Im Falle der DeepFake Videos ist es mit der Abschreckungstaktik eher schwierig, da die Einsatzmöglichkeiten so Vielfältig und die Folgen, selbst für den Einsetzenden, nicht einschätzbar sind. Als geeignete Gegenmaßnahmen bleiben die Sensibilisierung der Menschen für dieses Thema und die Entwicklung von effektiven Erkennungsverfahren. Durch die ebenfalls stetige Weiterentwicklung der Verfahren zum erstellen von DeepFake Videos, wird ein gewisses Wettrüsten entstehen. Diese Masterarbeit soll einen Beitrag zu der Forschung, an neuen Verfahren zur Erkennung von DeepFake Vdieos, leisten.

1.2. Aufgabenstellung

Das Ziel dieser Masterarbeit ist es, mit Hilfe von neuen Ansätzen und Verfahren, DeepFake Videos mit größerer Zuverlässigkeit und Effizients zu erkennen. Hierzu werden Verfahren ausgearbeitet, erprobt und evaluiert.

1.3. Gliederung

Die Arbeit besteht aus folgenden Kapiteln:

- **Kaptiel 2 - Grundlagen:**
  In diesem Kapitel wird auf die Grundlagen, zu verwendeten Verfahren sowie auf die genutzten Arten von neuronalen Netzen, eingegangen. Dies soll die Informationen, welche für die gesamte Arbeit und speziell für die Ausführungen im dritten und vierten Kapitel wichtig sind, bereitstellen.

- **Kapitel 3 - Methodik:**
  Im dritten Kapitel wird die verwendete Soft- und Hardware beschrieben. Zusätzlich wird das Vorgehen im Deep Learning Gesamtprozess beschrieben.

- **Kapitel 4 - Implementation:**
  In diesem Kapitel wird die Implementation der Verfahren, welche im Kapitel 3 beschrieben wurden, erklärt. Zusätzlich wird die Architektur und die Implementation der verwendeten neuronalen Netze bechrieben.

- **Kapitel 5 - Experimente und Evaluation:**
  Im fünften Kapitel werden die durchgeführten Trainings und Tests beschrieben. Die Ergebnisse der Experimente werden dargestellt und eingeordnet.

- **Kapitel 6 - Diskussion:**
  Im letzten Kapitel wird die gesamte Arbeit diskutiert und gesammelte Erkenntnisse aufgeführt. Am Ende wird noch ein Ausblick gegeben.

2. Grundlagen

In diesem Kapitel wird auf die Grundlagen, zu verwendeten Verfahren sowie zu verwendeten neuronalen Netzen, eingegangen. Dies soll die notwendigen Informationen, zur gesamten Arbeit und speziell für die Ausführungen im dritten und vierten Kapitel, bereitstellen. Hierbei werden nur die, für das Verständnis der Arbeit, wesentlichen Komponenten beschrieben.

2.1. Eulerian Video Magnification

Die menschliche Fähigkeit, Änderungen in der Umgebung visuell wahrzunehmen, hat eine beschränkte räumlich-zeitliche Empfindlichkeit. Dies hat als Konsequenz, dass Änderungen die Außerhalb dieses Empfindlichkeitsbereiches liegen, nicht von Menschen wahrgenommen werden können. Viele dieser subtilen Änderungen, die außerhalb der menschlichen Wahrnehmung liegen, beinhalten jedoch Informationen die von Interesse sein können. Zum Beispiel ändert sich, durch die zeitlich unterschiedliche Durchblutung, die Hautfarbe im Gesicht eines Menschen über die Zeit. Diese nicht wahrnehmbare Änderung, wenn sichtbar gemacht, kann zum Beispiel für die visuelle Messung der Pulsfrequenz einer Person genutzt werden.[**Philips**, **Poh:10**, **Verkruysse:08**] Um diese Informationen in Videos sichtbar zu machen, gibt es mehrere Ansätze. Neben der ”Lagrangian Motion Magnification”, welche mit optischem Fluss arbeitet, gibt es die ”Eulerian Video Magnification”. Letztere Methode kombiniert räumliche und zeitliche Verarbeitung, um die subtilen zeitlichen Änderungen in einem Video hervorzuheben. Im Gegensatz zur Verwendung von optischem Fluss zur Schätzung von Änderungen, ist dieser Ansatz nicht besonders rechenintensiv und somit auch für Echtzeitanwendungen geeignet.(todo Zitat eulerianmagnificationpaper)

2.1.1. Ablauf der Eulerian Video Magnification

Im ersten Schritt, wird das Video in unterschiedliche Ortsfrequenzbänder zerlegt. Diese Zerlegung wird durch den Aufbau von Bildpyramiden erreicht. Als nächstes werden diese Ortsfrequenzbänder, durch anwenden der Fast Fourier Transformation, in den Zeitbereich überführt und dort verarbeitet. Hierzu wird angenommen, dass die Zeitfolge mit dem Wert eines Pixels, in einem Frequenzband, korrespondiert und es wird ein Bandpass angewendet. Hiermit sollen die Frequenzbänder, die von Interesse sind, extrahiert werden. Das extrahierte Signal wird mit einem Verstärkungsfaktor multipliziert und auf das Ausgangssignal addiert. Im letzten Schritt wird aus den resultierenden Bildpyramiden, dass Zielvideo rekonstruiert. Der Prozess, der ”Eulerian Video Magnification”, ist in Abbildung 1 visuell dargestellt.

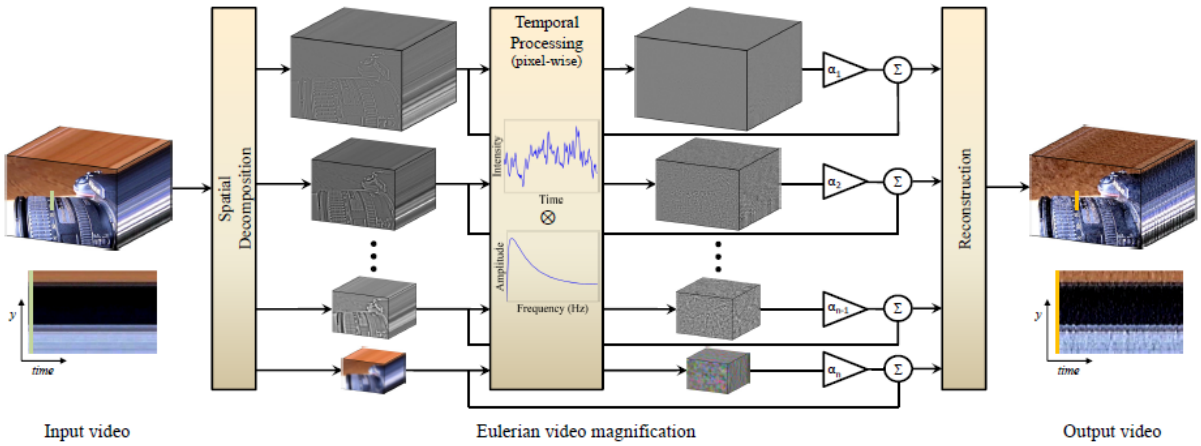

Abbildung 2.1.: Prozess ”Eulerian Video Magnification”

2.1.2. Bildpyramiden

Im Bereich der Bildanalyse und Bildmanipulation ist es oft von Vorteil, wenn das Bild auf mehreren Skalen analysiert oder manipuliert werden kann. Durch die Multiskalenanalyse wird eine Invarianz bezüglich der Größe erreicht und die Untersuchung von verschiedenen Strukturen im Bild erleichtert. Die Verwendung von Skalen bei der Bildanalyse führt eine weitere Dimension in die Bilder ein und verursacht dadurch einen signifikanten Anstieg des Speicherverbrauches, sowie einen signifikanten Anstieg des Rechenaufwandes. Das Konzept der Bildpyramiden wurde eingeführt, um genau dieses Problem, des Anstieges von Speicherverbrauch und Rechenaufwand, zu reduzieren. Die Idee hinter den Bildpyramiden ist einfach. Um feinere Skalen darstellen zu können, muss das Bild in der vollen Auflösung vorliegen. Sollen nun aber grobe Strukturen im Bild analysiert werden, dann reicht für diese Analyse eine niedriegere Auflösung aus. Die Bildpyramide beinhaltet also eine Folge von Bildern, wobei die Bilder von Stufe zu Stufe der Pyramide eine abnehmende Auflösung haben, dass heißt die Bilder werden kleiner. Diese Representation wird durch eine iterative Filterung und Unterabtastung erreicht. Obwohl Bildinformationen auf mehreren Skalen gespeichert werden, benötigt die Pyramide nur etwa ein Drittel mehr Speicher als das Originalbild. Neben dem geringeren Speicherbedarf werden, durch die Verwendung des selben Glättungsfilters auf allen Stufen, für die Berechnung der gesamten Pyramide nur vier Drittel der Operationen für ein zweidimensionales Bild benötigt. Nachdem die Pyramide einmal berechnet wurde, können Nachbarschaftsoperationen mit großen Skalen in den oberen Ebenen der Pyramide durchgeführt werden. Bei der ”Eulerian Video Magnification” werden Bildpyramiden eingesetzt, um die einzelnen Frames des Videos in unterschiedliche Ortsfrequenzbänder zu zerlegen. Dies ist notwendig, da diese unterschiedlichen Ortsfrequenzbänder möglicherweise verschieden stark verstärkt werden sollen. Die Entscheidung hier verschiedene Verstärkungsfaktoren zu verwenden, kann durch unterschiedliche Signal-Rausch-Abstände, in den Ortsfrequenzbändern, begründet sein. (todo Zitat https://www.cg.tuwien.ac.at/courses/EinfVisComp/Skriptum/SS13/EVC-16

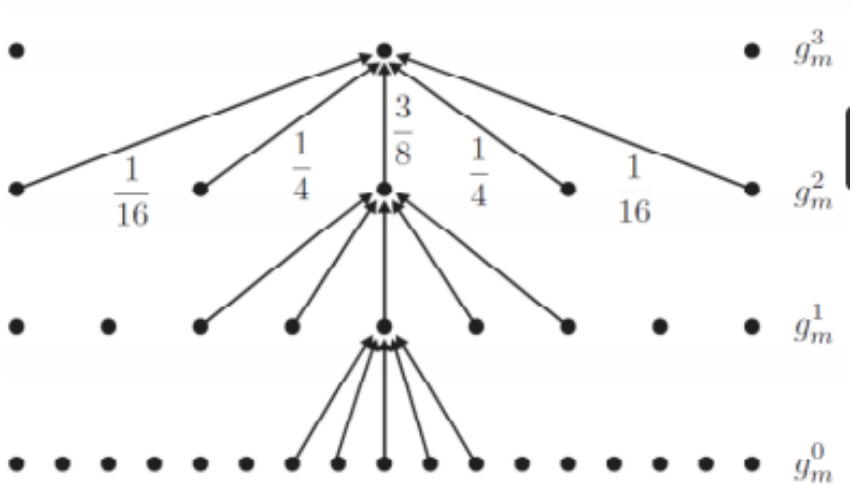

Abbildung 2.2.: Vorgehen Konstruktion einer Gauß-Pyramide

Gauß-Pyramide

Eine Pyramide, die mit einem Gaußfilter konstruiert wurde, heißt Gaußpyramide. In der Abbildung 2 ist die Vorgehensweise, von Tiefpassfilterung und der Unterabtastung um den Faktor zwei, visuell, an einem eindimensionalen Beispiel, dargestellt. Die Bilder, in der Gaußpyramide, wurden tiefpass-gefiltert, wobei die Grenzfrequenz von Stufe zu Stufe auf die Hälfte reduziert wurde. Hierdurch verbleiben, von Stufe zu Stufe, zunehmend grobe Strukturen in den Bildern.

Laplace-Pyramide

Alternativ zu der Tiefpassfilterung bei der Gauß-Pyramide, kann eine Pyramide so konstruiert werden, sodass in ihren Stufen eine Bandpassfilterung des Originalbildes enthalten ist. Durch die Subtraktion zweier aufeinanderfolgender Bilder in einer Gauß-Pyramide, wird eine solche Bandpassfilterung realisiert. Eine Laplace-Pyramide ist so eine Pyramide. Um das Bild in der oberen Stufe von der unteren Stufe zu subtrahieren, muss dieses vorher auf die gleiche Größe expandiert werden. Im Gegensatz zu der Größenreduktion ist die Expansion rechenintensiver, da die fehlenden Informationen interpoliert werden müssen. In der Laplace-Pyramide werden auf den ersten Stufen feinere Kantenstrukturen betont und auf den oberen Stufen zunehmend grobe Kantenstrukturen des Originialbildes betont. Die Approximation der zweiten Ableitung des Originalbildes wird druch die Laplace-Pyramide dargestellt. Ein bedeutender Vorteil der Laplace-Pyramide ist, dass durch die rekursive Expandierung und Aufsummierung, der enthaltenen Bildserie, das Originalbild schnell wiederhergestellt werden kann. Dies entspricht einer Umkehrung des Konstruktionsablaufs. Die Laplace-Pyramide wird bei der ”Eulerian Video Magnification” verwendet, um das Video in unterschiedliche Ortsfrequenzbänder zu zerlegen. (todo Zitat https://www.cg.tuwien.ac.at/courses/EinfVisComp/Skriptum/SS13/EVC-16

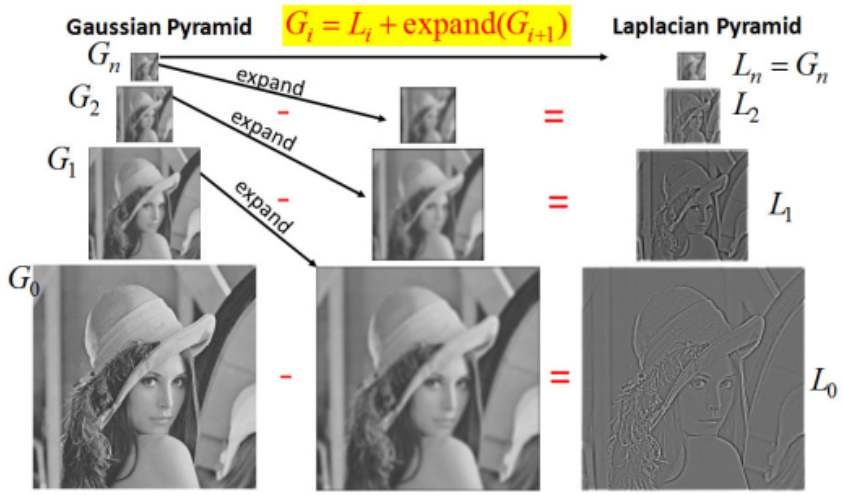

Abbildung 2.3.: Vorgehen Konstruktion einer Laplace-Pyramide

2.2. Convolutional Neural Networks

Convolutional Neural Networks, auch bekannt als CNN's, sind ein spezialisierter Typus von neuronalen Netzen, um Daten zu verarbeiten, welche eine gitterartige Topologie aufweisen. Ein Beispiel hierfür sind Bilder, welche eine zwei dimensionale gitterartige Struktur aus Pixeln sind. Der Unterschied zu einfachen neuronalen Netzen ist, dass die CNN's, anstatt einfacher Matrizenmultiplikationen, Faltungsoperation durchführen. In praktischen Anwendungen, wie dem maschinellen Sehen, wurden CNN's immer wieder sehr erfolgreich eingesetzt.

2.2.1. 2D-Convolutional Neural Networks

Für viele praktische Anwendungen sind zwei dimensionale CNN's die erste Wahl. Ob als Autoencoder, zur Objekterkennung in Bildern bis hin zur synthetischen Generierung von Bildern. Die 2D-CNN's haben, anders als der Name erwarten lässt, einen drei dimensionalen Input. Bei Bildern sind zum Beispiel die Farbkanäle die dritte Dimension. Die Zwei im Namen beziffert die Dimensionen in denen sich der Kernel über das Bild bewegt. Die 2D-CNN's sind im Normalfall aus mehreren Convolutional-Schichten und Pooling-Schichten aufgebaut. Wenn die Klassifikation von Bildern das Ziel ist, dann werden meist voll verbundene Schichten, als letzte Schichten vor dem Ausgang des Netzes, eingefügt. Hier sollen die Convolutional-Schichten verschiedene Merkmale im Bild herausfiltern und die voll verbundene Schicht mit Hilfe dieser Merkmale das Bild klassifizieren. Wie viele Convolutional-Schichten und Pooling-Schichten in einem 2D-CNN verwendet werden, ist von Anwendungsfall zu Anwendungsfall unterschiedlich. Die optimale Architektur eines 2D-CNN, für einen bestimmten Anwendungsfall, muss durch empirische Forschung ermittelt werden. Durch die intensive Forschung an CNN's, werden immer wieder neue Architekturen und Abwandlungen entwickelt. Die Entwicklung ist in dem Bereich so schnell, dass state-of-the-art Netzarchitekturen sich im Wochenrythmus oder Monatsrythmus ändern. Aufgrund dieser Volatilität und Vielfältigkeit, ist es schwierig ein Beispiel zu finden,

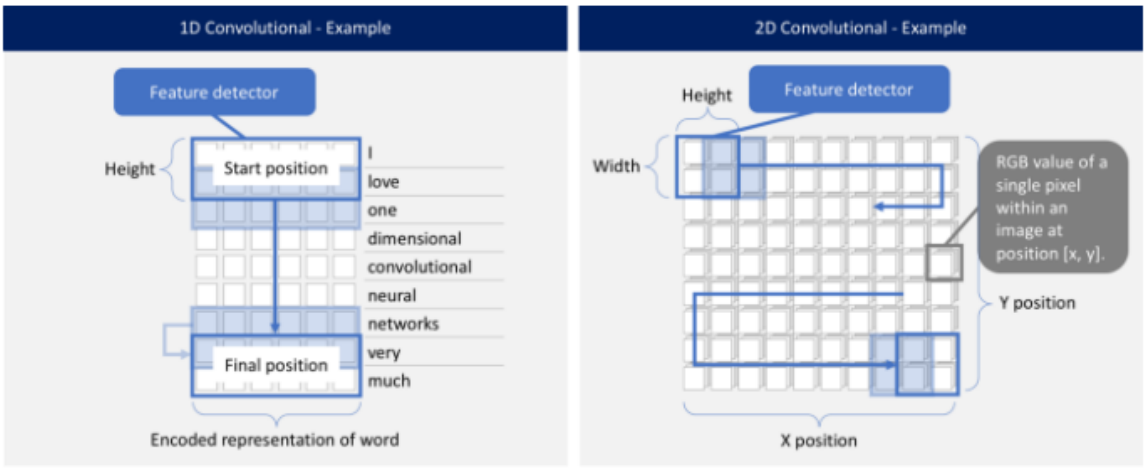

Abbildung 2.4.: Vergleich der Faltungsoperation bei 2D-CNN's und 1D-CNN's by Nils Ackermann is licensed under Creative Commons CC BY-ND 4.0

welches möglichst allgemeingültig ein 2D-CNN darstellt. Aufgrunddessen wird in diesem Abschnitt auf ein explizites Beispiel, einer Netzarchitektur, verzichtet. Die in dieser Arbeit verwendeten 2D-CNN Architekturen, werden im dritten Kapitel dargestellt und erklärt.

2.2.2. 1D-Convolutional Neural Networks

Die im vorherigen Abschnitt beschriebenen 2D-CNN's sind dafür entworfen, und wurden dahingehend fortlaufend weiterentwickelt, um mit 2D-Daten wie Bildern zu arbeiten. Als Alternative zu den konventionellen 2D-CNN's, wurden die 1D-CNN's entwickelt. Alle CNN's haben gleiche Charakteristiken und folgen dem gleichen Ansatz, unabhängig davon ob es ein 1D-, 2D- oder 3D-CNN ist. In der Abbildung 4 ist der Unterschied zwischen 2D-CNN's und 1D-CNN's, an den Beispielen der Verarbeitung von natürlicher Sprache und des maschinellen Sehens, bei der Faltungsoperation visualisiert. In dem Beispiel für das 1D-CNN ist das Eingangsdatum ein Satz, bestehend aus 9 Wörtern. Jedes Wort wird als Vektor, der Convolutional-Schicht, übergeben. Unabhängig von der Länge des Wortes, wird der Filter immer das ganze Worte abdecken. Wie viele Wörter in einem Trainingsschritt betrachtet werden, wird durch die Höhe des Filters festgelegt. In dem Beispiel ist die Höhe des Filters zwei. Folgend wird der Filter, in acht Schritten und einer Dimension, über die Daten geschoben. In dem Beispiel für ein 2D-CNN wird ein Farbbild als Eingangsdatum verarbeitet. Der Filter hat die Höhe und Breite von zwei. Im Gegensatz zum 1D-CNN, wird hier der Filter in zwei Dimensionen, dass heißt horizontal und vertikal, über die Daten geschoben.

Eingangsdaten, für ein 1D-CNN, könnten beispielsweise Zeitfolgen von mehreren Sensoren sein. Diese Zeitfolgen, welche für sich genommen eindimensional sind, könnten auch als Matrix in einem 2D-CNN verarbeitet werden. Bei Untersuchungen zeigte sich jedoch, dass die 1D-CNN's für bestimmte Anwendungfälle zu bervorzugen sind, wenn eindimensionale Daten verarbeitet werden.

(todo Zitat [47] S. Kiranyaz, T. Ince, R. Hamila, M. Gabbouj, Convolutional Neural Networks

for patient-specific ECG classification, in: Proc. Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. EMBS, 2015. doi:10.1109/EMBC.2015.7318926. [48] S. Kiranyaz, T. Ince, M. Gabbouj, Real-Time Patient-Specific ECG Classification by 1-D Convolutional Neural Networks, IEEE Trans. Biomed. Eng. 63 (2016) 664?675. doi:10.1109/TBME.2015.2468589. [49] S. Kiranyaz, T. Ince, M. Gabbouj, Personalized Monitoring and Advance Warning System for Cardiac Arrhythmias, Sci. Rep. 7 (2017).

Die Gründe hierfür sind:

- **In einem 1D-CNN werden, statt Matrixoperationen für die Vor- und Rückpropagation, nur einfach Arrayoperationen durchgeführt. Daraus folgt, dass die Berechnungskomplexität eines 1D-CNN signifikant geringer ist als die eines 2D-CNN.**[5]
- **Die aktuelle Forschung zeigt, dass 1D-CNN's, auch mit relativ flachen Architekturen, in der Lage sind komplexe Aufgaben zu erlernen, bei denen eindimensionale Daten verarbeitet werden. Im Gegensatz dazu, benötigen 2D-CNN's gewöhnlich tiefere Architekturen, um ähnlich komplexe Aufgaben zu erlernen. Dies hat den Vorteil, dass flachere Architekuren einfacher zu trainieren und zu implementieren sind.**[6]
- **Aufgrund Ihrer eher geringen Berechnungskomplexität, sind 1D-CNN's gut geeignet für Echtzeitanwendungen und für den energiesparenden Einsatz auf mobilen Geräten.** [7]

In aktuellen Untersuchungen zeigten 1D-CNN's gute Ergebnisse, besonders bei Anwendungsfällen bei denen die verfügbaren Daten stark begrenzt waren und die Signale aus verschiedenen Quellen hohe Schwankungen zeigten. Wie in Abbildung3 dargestellt, besteht ein 1D-CNN meist aus Convolutional-Schichten und aus voll vebundenen Schichten zur Klassifikation. Die folgenden Hyperparamater bilden die Konfiguration eines 1D-CNN:

1. Anzahl der Convolutional-Schichten und voll vebundenen Schichten
2. Kernelgröße in jeder Convolutional-Schicht
3. Unterabtastungsfaktor in jeder Convolutional-Schicht
4. Die Auswahl von Pooling und Aktivierungsfunktionen

2.2.3. 3D-Convolutional Neural Networks

Um in Videos Objekte zu erkennen, werden 2D-CNN's erfolgreich eingesetzt. Bei diesem Anwendungsfall wird jedes Einzelbild, in einem Video, separat verarbeitet. Hierbei werden aber die Änderungen zwischen den Einzelbildern nicht betrachtet und es gehen somit die

[5] https://moveit.ros.org/
[6] https://ros-planning.github.io/moveit_tutorials/
[7] https://github.com/ros-planning

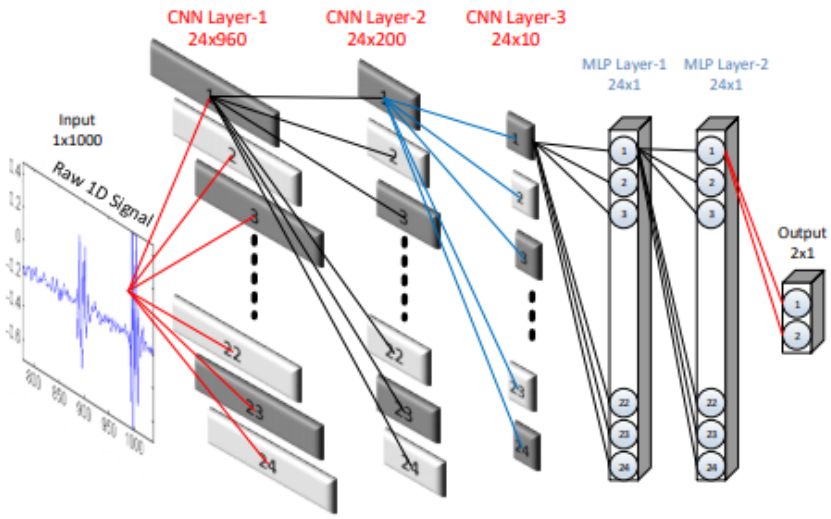

Abbildung 2.5.: Beispiel für eine 1D-CNN-Architektur

Information in der zeitlichen Dimension komplett verloren. Diese Information, in der zeitlichen Dimension, können jedoch, für manche Anwendungsfälle, von Bedeutung sein. Um eine weitere Dimension in die Verarbeitung der Daten mit einzubeziehen, wurden die 3D-CNN's entwickelt. Diese weitere Dimension kann die Zeit sein oder auch die dritte räumliche Dimension, um z.B. Volumenbilder zu verarbeiten. In der Abbildung 5 ist der Unterschied zwischen 2D-CNN's und 3D-CNN's dargestellt. Der Kernel wird, bei dem 3D-CNN, in drei Dimensionen über die Daten Bewegt. Für das Beispiel der Videoklassifikation legt die Kerneltiefe D fest, wie viele Bilder in einer Faltung betrachtet werden. Die Breite W und Höhe H des Kernels, haben hier die gleiche Bedeutung wie bei 2D-CNN's. Das Ergebnis der Faltung ist eine 3D-Featuremap. In der Praxis werden die 3D-CNN's noch eher selten verwendet, da durch die Einführung der weiteren Dimension der Speicherbedarf und die Berechnungskomplexität enorm ansteigt. Hinzu kommt, dass die Verfügbarkeit von geeigneten Trainingsdatensätzen noch eingeschränkt ist. Die folgenden Hyperparamater bilden die Konfiguration eines 1D-CNN:

1. Anzahl der Convolutional-Schichten, Pooling-Schichten und voll vebundenen Schichten
2. Die Kernelfläche und die Kerneltiefe
3. Die Auswahl von Regularisierungsfunktionen
4. Die Auswahl von Aktivierungsfunktionen

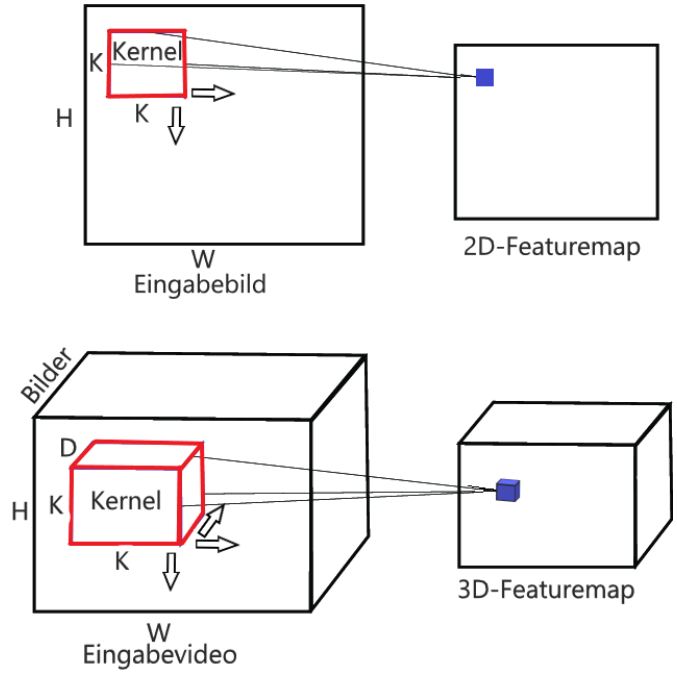

Abbildung 2.6.: Vergleich von 2D-CNN und 3D-CNN

3. Methodik

In diesem Kapitel wird die verwendete Soft- und Hardware beschrieben und es wird erklärt wofür diese eingesetzt wurde. Zusätzlich wird der Deep Learning Gesamtprozess beschrieben.

3.1. Verwendete Software und Hardware

3.1.1. Software

Tensorflow

Tensorflow ist ein Framework, welches Funktionen zum erstellen von Modellen, für das maschinelle Lernen, bereitstellt. Das Framework wird als Open Source Projekt, unter der Federführung von Google, weiterentwickelt. Die Tensoren, welche typisierte multidimensionale Arrays darstellen, bilden das Hauptkonzept von Tensorflow. Diese Tensoren durchlaufen Datenflussgraphen, welche aus Knoten bestehen. Durch die Knoten werden numerische Operationen abgebildet. Die Ausführung dieser Operationen, kann mit Tensorflow auf Grafikarten ausgelagert werden. In dieser Arbeit wurde mit Tensorflow, in der Version 2.2, gearbeitet. Tensorflow wurde hier als Backend für Keras genutzt.

Keras

Keras ist eine High-Level-API, welche verschiedene Backends unterstützt. In dem Keras eine weitere Abstraktionsebene schafft, wird das erstellen, trainieren und evaluieren von neuronalen Netzen vereinfacht. Durch die Unterstützung von verschiedenen Backends, muss der Keras-Code nur einmal geschrieben werden und kann dann mit den verschiedenen Backends verwendet werden. In dieser Arbeit wurde mit Keras, in der Version 2.4, gearbeitet. Keras wurde genutzt, um Modelle zu erstellen, zu trainieren und zu testen.

OpenCV

OpenCV ist eine Open Source Bibliothek, welche Funktionen für die Bildverarbeitung und für das maschinelle Sehen bereitstellt. In dieser Arbeit wurde mit OpenCV, in der Version 4.3.0, gearbeitet. Es wurde die Funktionen zur Gesichtserkennung sowie die Funktionen für die Verarbeitung von Videos und Bildern verwendet.

Eulerian Video Magnification

Das Paket "Eulerian Video Magnification" ist in Matlab-Code geschrieben und bietet Funktionen zur Anwendung des gleichnamigen Verfahrens an. In dieser Arbeit wurde mit der Version 1.1 gearbeitet. Unter Verwendung des Matlab-Kernels für Python, wurden die Funktionen des Paketes in Python verfügbar gemacht.

3.1.2. Hardware

Für die Programmierarbeiten wurde ein Notebook, mit Ubuntu 18.04 als Betriebssystem, verwendet. Das Notebook hat einen Intel Core i7 Prozessor, 16 Gigabyte Arbeitsspeicher und eine Nvidia GTX 1050 Ti Grafikkarte mit 4 Gigabyte Grafikspeicher.

Trainingssystem

Als Trainingssystem, für neuronale Netze, wurde ein Nvidia DGX-1 System verwendet. Dieses System verfügt über sechzehn V100 Grafikkarten, mit jeweils 32 Gigabyte Grafikspeicher.

3.2. Vorgehen im Deep Learning Gesamtprozess

In diesem Abschnitt wird der Deep Learning Gesamtprozess beschrieben. Es wird nur auf die verwendeten Daten und auf die Vorgehensweise eingegangen.

3.2.1. Datensatz

Als Datensatz, für das Training und die Validierung, wurde der Celeb-DF(V2) Datensatz verwendet. Dieser Datensatz besteht aus 590 realen Videos und 5639 DeepFake-Videos. Die Videos sind im Durchschnitt 13 Sekunden lang und haben eine Bildwiederholungsrate von 30 Bilder/s. Die realen Videos zeigen 59 verschiedene Bekanntheiten. Die gezeigte Personengruppe besteht aus 56.8 % Männern und 43.2 % Frauen. Hiervon sind 8.5 % älter als 60, 30.5 % 50-60, 26.6 % in den Vierzigern und 28.0 % 30-40. Die ethnischen Gruppen sind 5.1 % Asiaten, 6.8 % Afro-Amerikaner und 88.1 % weiße Amerikaner. Die Größe der Gesichter in Pixeln variert in den Videos. Die DeepFake Videos wurden durch das Tauschen der Gesichter für jedes Paar der 59 Personen generiert. Für die Generierung der DeepFake Videos wurde ein Autoencoder verwendet. Alle Videos liegen im MPEG4.0 Format vor.

3.2.2. Testdaten

Da die Anzahl an realen Videos, für das Training eines neuronalen Netzes, relativ gering ist, wurden die gleichen Videos zum validieren und zum testen verwendet. Diese Videos wurden, vor dem Beginn des Trainings, von den Trainingsdaten separiert und es wurde nie mit diesen Videos trainiert. Dieser Datensatz enthält 177 reale Videos und 340 DeepFake Videos. Durch die Selektion von Netzen, welche während des Trainings gute Ergebnisse auf den Validierungsdaten

erreichten, könnten die Ergebnisse der Tests zum positiven verfälscht sein. Um die erreichte Fähigkeit der Netze zum generalisieren besser einschätzen zu können, wurde ein zweiter Testdatensatz erstellt. Dieser zweite Testdatensatz besteht ebenfalls aus 177 realen Videos und 340 DeepFake Videos. Die Videos des zweiten Testdatensatzes stammen aus dem "FaceForensics++" Datensatz. Es wurden Videos ausgewählt, welche, wie die Videos aus dem Hauptdatensatz, mit dem FaceSwap Verfahren erstellt wurden.

3.2.3. Vorverarbeitung der Daten

Vor dem Training der Modelle wurden die Videos vorverarbeitet. Um so wenig störenden Hintergrund wie möglich im Bild zu haben, wurde eine Gesichtserkennung auf die Videos angewendet und der Hintergrund herausgeschnitten. Anschließend wurde die "Eulerian Video Magnification" auf alle Videos angewendet. Hier wurde gezielt das Frequenzband verstärkt, in welches auch die Herzfrequenz fällt. Die folgenden Schritte wurden auf die zugeschnittenen Videos sowie auf die zugeschnittenen und verstärkten Videos angewendet. Dies bietet die möglichkeit die Wirkung der "Eulerian Video Magnification" auf die Ergebnisse beim Training und Test darzustellen. Da manche Videos zu lang waren und auch in der Länge zu stark variierten, wurden die Videos in mehrere kurze Videos zugeschnitten. Um die Videos mit einem 1D-CNN verarbeiten zu können, wurden die Videos in eine zweidimensionale Representation umgewandelt. Hierzu wurde aus jedem Einzelbild eine horizontale Pixelreihe, auf der Höhe von 60 % der Gesamthöhe des Einzelbildes, von 150 Pixeln herausgeschnitten. Auf dieser Höhe sollten sich die Nase und die Wangen befinden, welche meist stärker durchblutet sind als andere Bereiche im Gesicht. Um eine zweidimensionale Representation zu erhalten, wurden diese Pixelreihen, in der zeitlichen Abfolge im Video, untereinander angeordnet. Diese Representation des Videos wurde als Bild im JPEG Format gespeichert. In Abbildung 3.1 ist an einem Beispiel das Vorgehen und das Ergebnis visualisiert. Um das Signal, welches von der "Eulerian Video Magnification" verstärkt wurde, weiter zu isolieren, sind Differenzbilder und Differenzvideos erstellt worden. Dazu wurden die Pixelwerte der Originalbilder von den Pixelwerten der verstärkten Bilder subtrahiert und der Betrag jeweils gebildet. Folgende Datensätze für das Training standen, nach der vorangegangenen Vorverarbeitung, zu Verfügung:

1. Die Videos mit reduziertem Hintergrund im Original und verstärkt.
2. Die Differenzvideos
3. Die zweidimensionalen Representationen, der Original Videos und der verstärkten Videos, als Bild.
4. Die zweidimensionalen Representationen als Differenzbilder

3.2.4. Training und Test

Die verwendeten Netze wurden jeweils mit jedem vorverarbeiteten Trainingsdatensatz trainiert. Die Bereitstellung der Daten und das Training an sich wurden mit Keras realisiert. Die Ausgaben

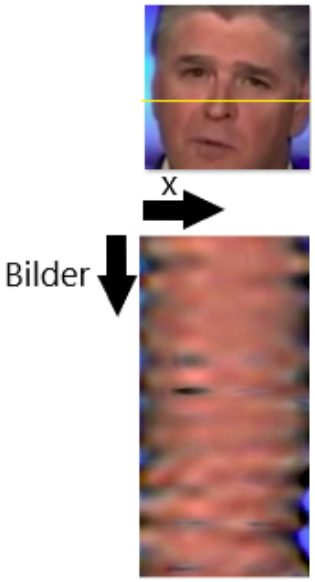

Abbildung 3.1.: Beispiel erstellen von 2D-Representation

während des Trainings wurden in Textdateien geschrieben, um den Trainingsverlauf, auch bei Abbruch durch Fehler, nachvollziehen zu können. Nach der Beedigung jedes Trainings wurden die Verläufe des Fehlers und der AUC(Area Under the ROC Curve) in einem Diagram gespeichert. Der AUC Wert steht für die Fläche unter der ROC Kurve (receiver operating characteristic curve). Die ROC Kurve zeigt das Verhältnis zwischen der Spezifität und der Sensitivität des Netzes, für verschiedene Klassifikationsschwellen. Der AUC Wert eignet sich für die Bewertung von binären Klassifikatoren wesentlich besser als die Genauigkeit, da die AUC weniger durch die Menge der verwendeten Daten und das Klassenverhältnis beeinflusst wird. Im nächsten Schritt wurde der Fehler und die AUC des Netzes für die Testdatensätze ermittelt.

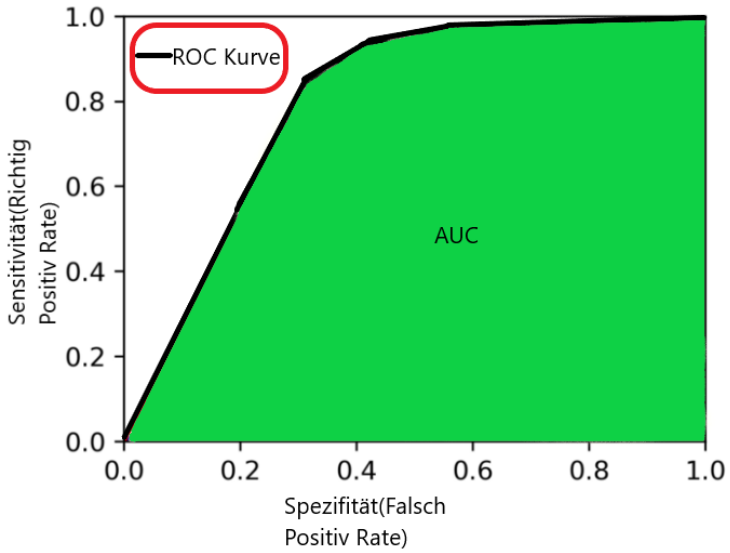

Abbildung 3.2.: Beispiel für eine ROC Kurve und die AUC

4. Implementation

Die Implementation von Funktionen, für die Vorverarbeitung der Datensätze, und die Implementation der verwendeten neuronalen Netze wird in diesem Kapitel beschrieben.

4.1. Erstellen der Datensätze

In diesem Abschnitt wird der Ablauf der Generierung der Datensätze näher beschrieben und die dazu implementierten Funktion erklärt. Es werden nur die essentielen Funktion beschrieben. Genutzte Hilfsfunktionen liegen im Quellcode, der im Anhang dieser Arbeit zu finden ist, vor.

4.1.1. Entfernen des Hintergrundes

Um so viel Hintergrund wie möglich zu entfernen und dennoch keine Teile des Gesichtes zu entfernen, wurden mehrere Schritte durchgeführt. Zusätzlich war es ein Ziel ein Video zu erhalten, bei welchem das Gesicht möglichst statisch und zentral im Bild gehalten wird, da es sich herausgestellt hat, dass dies sich positiv auf das Resultat der ”Eulerian Video Magnification” auswirkt. Im ersten Schritt wird das Video mit OpenCV geöffnet. Die Auflösung und die Anzahl der Einzelbilder wird ermittelt. Nun wird jedes Einzelbild nacheinander eingelesen. Es wird eine Gesichtserkennung mit OpenCV auf jedes Einzelbild angwendet. Die Gesichtserkennungsfunktion gibt die Koordinaten eines Rahmens zurück, welcher das gefundene Gesicht umrahmt. Diese Koordinaten werden in Arrays gespeichert und die eingelesen Einzelbilder unbearbeitet in einen Videotensor geschrieben. Von den Videotensoren werden die ersten und die letzten 20 Einzelbilder und die dazugehörigen Gesichtskoordinaten aus den Arrays enfernt, da dort oft Einblendeffekte vorhanden sind oder die ersten Einzelbilder den Abschluss eines Kamerawechsels zeigen. Effekte und Vorgänge dieser Art können, dass Resultat der ”Eulerian Video Magnification”, negativ beeinflussen. Da die Bilder, nach entfernen des Hintergrundes, alle die gleiche Größe haben müssen, wird aus den Gesichtskoordinaten die durchschnittliche Rahmengröße berechnet. Um ein Wackeln des Rahmens zu verhindern, werden die Koordinaten mit einer Glättungsfunktion gefiltert. Hierdurch folgt der Rahmen dynamisch dem Gesicht im Video, ohne ein Zittern des Bildes zu erzeugen. Wenn für jedes Einzelbild der Rahmen berechnet wurde, dann wird mit OpenCV der Teil des Originalbildes, welcher von dem Rahmen umschlossen wird, in eine Videodatei geschrieben und das Video gespeichert.

4.1.2. Anwenden der ”Eulerian Video Magnification”

Um die ”Eulerian Video Magnification” in Python auf ein Video anwenden zu können, wird der Matlab-Kernel für Python gestartet. Durch den Aufruf der Matlab-Funktion
”amplify_spatial_Gdown_temporal_ideal” des Paketes ”Eulerian Video Magnification” wird das Video geöffnet, mit den gewählten Fuktionsparametern verarbeitet und das Zielvideo abgespeichert. Folgende Funktionsparameter haben sich, für den genutzten Datensatz, in ersten Experimenten als effektiv ergeben:

1. Verstärkungsfaktor = 12.0
2. Anzahl an Bildpyramidenstufen = 3
3. untere Grenzfrequenz = 0.75 Hertz
4. obere Grenzfrequenz = 1.67 Hertz
5. Dämpfungsfaktor für Chrominanz = 0.7

Diese Funktionsparameter sind für das Deep Learning als zusätzliche Hyperparameter zu betrachten, da diese das Training, und somit auch die Klassifikationsfähigkeit, beeinflussen.

4.1.3. Erstellen der 2D-Representationen

Um eine 2D-Representation eines Videos zu erhalten, wird jedes Einzelbild geladen und die Pixelreihe auf 60% der Höhe des Einzelbildes herausgeschnitten. Die Pixelreihen werden in einen Tensor untereinander angeordnet. Der Tensor wird mit OpenCV in ein Bild im JPEG Format umgwandelt und abgespeichert.

4.1.4. Kürzen der Videos

Die Videos werden mit OpenCV geöffnet. Es werden immer jeweils 30 Einzelbilder in einem Tensor abgelegt und mit OpenCV daraus gekürzte Videos erstellt und im MPEG4 Format gespeichert.

4.1.5. Erstellen der Differenzvideos

Das Originalvideo und das verstärkte Video werden mit OpenCV geöffnet. Jedes Einzelbild des Originalvideos wird von dem jeweils korrespondierenden Bild im verstärkten Video subtrahiert und von dem Ergebis der Betrag gebildet. Mit OpenCV wird aus den berechneten Differenzbildern ein Video erstellt und im MPEG4 Format gespeichert.

4.1.6. Erstellen der Differenzbilder

Die 2D-Representation des Originalvideos und die des verstärkten Videos wird geöffnet. Die 2D-Representation des Originalvideos wird von der des verstärkten Videos subtrahiert und von dem Ergebnis der Betrag gebildet. Das Ergebnis wird mit OpenCV als Bild im JPEG Format gespeichert.

4.2. Erstellen der Modelle

In diesem Abschnitt werden die verwendeten neuronalen Netze und deren Implementation beschrieben.

4.2.1. 1D-OS-CNN

Mit der Architektur des 1D-OS-CNN, wurde die Auswahl der Kernelgröße in den Lernprozess integriert. Dies bedeutet, dass ein Hyperparameter weniger gewählt werden muss, da die Suche nach der optimalen Kernelgröße ein Teil des Lernprozesses ist. (Todo Zitat Rethinking 1D-CNN for Time Series Classification: A Stronger Baseline) Diese Idee wurde in dem erstellten Modell übernommen. In Abbildung 10 ist die Architektur des erstellten Modells dargestellt. Es wurden vier Convolutional-Schichten, eine Pooling-Schicht und drei voll verbundene Schichten gewählt. Die Convolutional-Schichten beinhalten eine Batch-Normalisierung und die Aktivierung mit einer Relu. Die Besonderheit an den Convolutional-Schichten ist, das diese mehrere Faltungen der Eingangsdaten, mit verschiedenen Kernelgrößen, parallel durchführen. Die Ergebnisse aller Faltungen werden aneinander gehängt, eine Batchnormalisierung durchgeführt und mit einer Relu aktiviert. Das Ergebnis wird der nächsten Convolutional-Schicht übergeben. Nach der Merkmalsextraktion, führen die drei voll verbundenen Schichten die Klassifikation durch. Im folgenden werden die gewählten Hyperparamter der Schichten, mit Bezug auf die Abbildung 4.1, aufgeführt:

- **Covolutional Layer #1:** Die genutzten Kernelgrößen sind, alle Primzahlen von 1 - 50 und die 1. Daraus folgt die Anzahl von sechzehn parallelen Faltungen. Die Anzahl an verschiedenen Filtern, die pro paralle Faltung verwendet werden, ist 32. Als Kernelinitalisierer wird der HeUniform-Initialisierer verwendet. Ein Pooling wird nicht durchgeführt.

- **Covolutional Layer #2:** Die genutzten Kernelgrößen sind, alle Primzahlen von 1 - 40 und die 1. Daraus folgt die Anzahl von dreizehn parallelen Faltungen. Die Anzahl an verschiedenen Filtern, die pro paralle Faltung verwendet werden, ist 64. Als Kernelinitalisierer wird der HeUniform-Initialisierer verwendet. Ein Pooling wird nicht durchgeführt.

- **Covolutional Layer #3:** Die genutzten Kernelgrößen sind, alle Primzahlen von 1 - 30 und die 1. Daraus folgt die Anzahl von elf parallelen Faltungen. Die Anzahl an verschiedenen Filtern, die pro paralle Faltung verwendet werden, ist 128. Als Kernelinitalisierer wird der HeUniform-Initialisierer verwendet. Ein Pooling wird nicht durchgeführt.

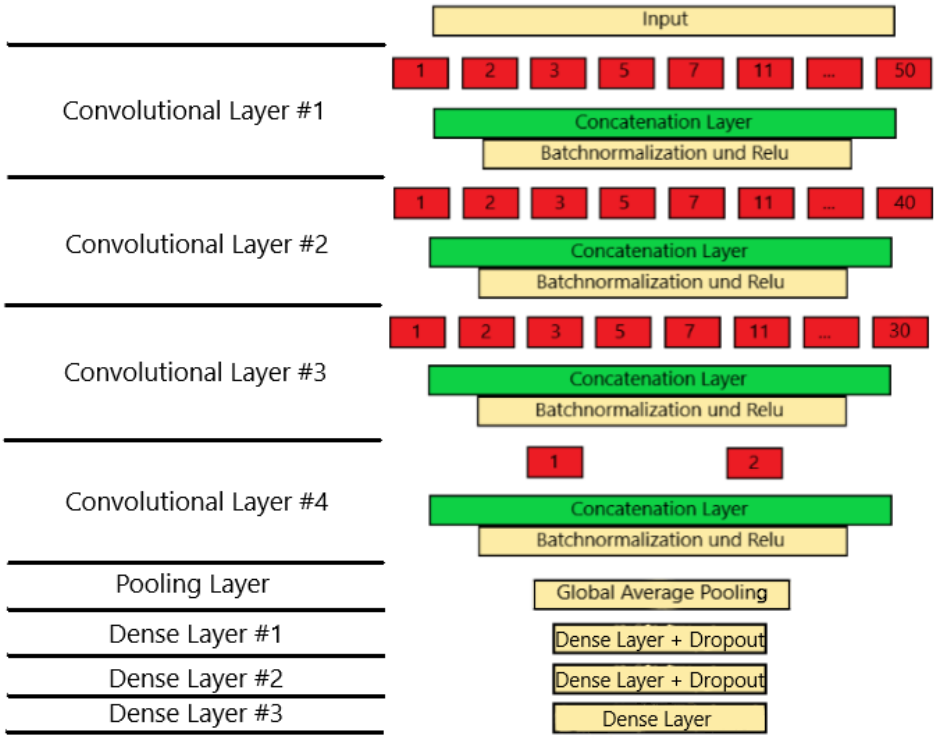

Abbildung 4.1.: Architektur des 1D-OS-CNN

- **Covolutional Layer #4:** Die genutzten Kernelgrößen sind die eins und die zwei. Daraus folgt die Anzahl von zwei parallelen Faltungen. Die Anzahl an verschiedenen Filtern, die pro paralle Faltung verwendet werden, ist 256. Als Kernelinitalisierer wird der HeUniform-Initialisierer verwendet. Ein Pooling wird nicht durchgeführt.

- **Pooling Layer:** Es wird ein Global Average Pooling durchgeführt. Der Stride ist zwei.

- **Dense Layer #1:** Die Anzahl an Neuronen ist 512. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet. Ein Dropout, mit einer Wahrscheinlichkeit von 50%, wird druchgeführt.

- **Dense Layer #2:** Die Anzahl an Neuronen ist 512. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet. Ein Dropout, mit einer Wahrscheinlichkeit von 50%, wird druchgeführt.

- **Dense Layer #3:** Dies ist die Ausgabeschicht und hat ein Neuron. Zur Aktivierung wird die Sigmoidfunktion verwendet.

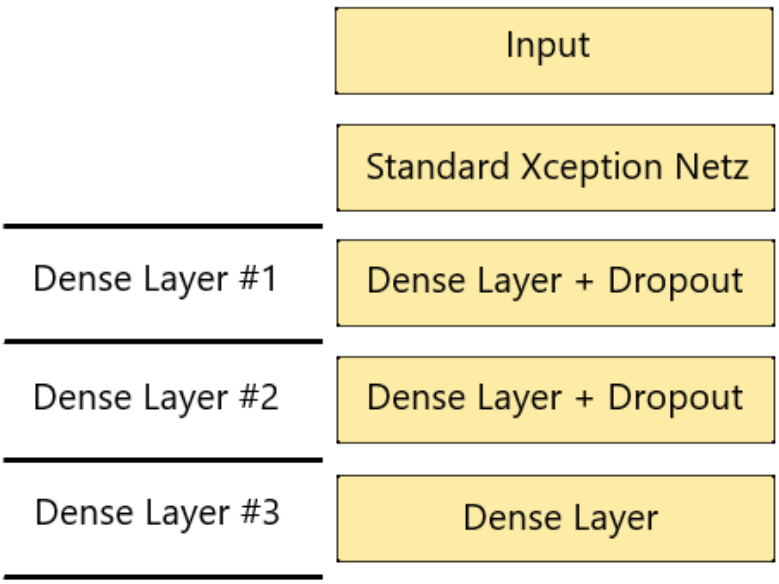

Abbildung 4.2.: Architektur des Xception Netzes

4.2.2. Xception

Mit der Xception-Architektur wurde die Idee des Inception Netzes weiterentwickelt. Bei der Xception-Architektur werden nur noch separable Faltungen in den Convolutional-Schichten durchgeführt. Für diese Arbeit wurde das Standard Xception-Netz, welches Keras zur Verfügung stellt, verwendet. Als Zusatz wurden drei voll verbundene Schichten am Ende hinzugefügt. Die gewählten Hyperparameter, mit Bezug auf Abbildung 4.2, sind:

- **Dense Layer #1:** Die Anzahl an Neuronen ist 2048. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet. Ein Dropout, mit einer Wahrscheinlichkeit von 50%, wird druchgeführt.
- **Dense Layer #2:** Die Anzahl an Neuronen ist 1024. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet. Ein Dropout, mit einer Wahrscheinlichkeit von 50%, wird druchgeführt.
- **Dense Layer #3:** Dies ist die Ausgabeschicht und hat ein Neuron. Zur Aktivierung wird die Sigmoidfunktion verwendet.

4.2.3. 3D-CNN

Das erstellte 3D-CNN besteht aus acht Convolutional-Schichten, fünf Pooling-Schichten und drei voll verbundenen Schichten. Die gewählten Hyperparameter, mit Bezug auf Abbildung 4.3, sind:

- **Convolutional Layer #1:** Die Kernelgröße ist (7,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 64. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.
- **Pooling Layer #1:** Es wird ein Max Pooling durchgeführt. Der Stride ist (1,2,2).
- **Convolutional Layer #2:** Die Kernelgröße ist (7,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 128. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.
- **Pooling Layer #2:** Es wird ein Max Pooling durchgeführt. Der Stride ist (2,2,2).
- **Convolutional Layer #3:** Die Kernelgröße ist (5,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 256. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.
- **Convolutional Layer #4:** Die Kernelgröße ist (5,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 256. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.
- **Pooling Layer #3:** Es wird ein Max Pooling durchgeführt. Der Stride ist (2,2,2).
- **Convolutional Layer #5:** Die Kernelgröße ist (3,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 512. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.
- **Convolutional Layer #6:** Die Kernelgröße ist (3,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 512. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.
- **Pooling Layer #4:** Es wird ein Max Pooling durchgeführt. Der Stride ist (2,2,2).
- **Convolutional Layer #7:**Die Kernelgröße ist (3,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 512. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.
- **Convolutional Layer #8:** Die Kernelgröße ist (3,3,3), dies bedeutet sieben Einzelbilder und jeweils neun Pixel werden gefaltet. Die Anzahl an verschiedenen Filtern die verwendet werden ist 512. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.

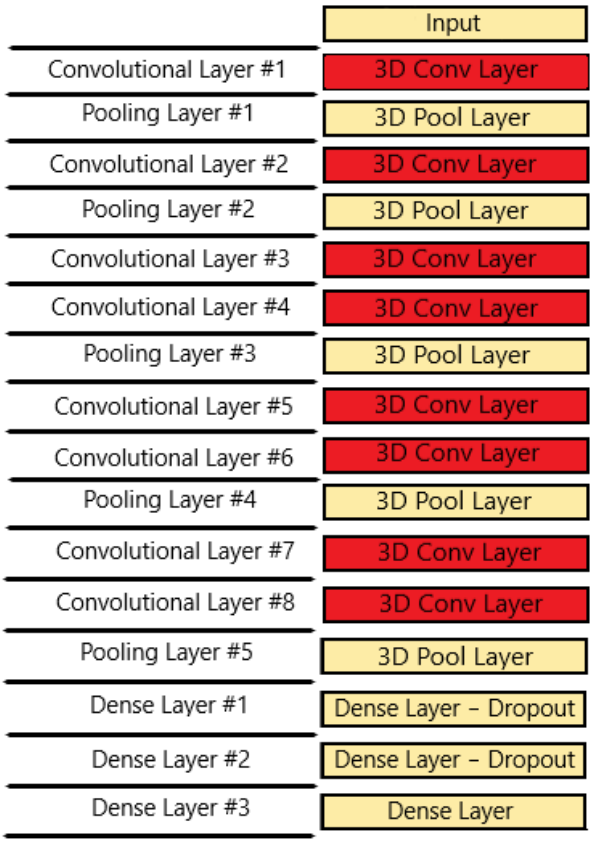

Abbildung 4.3.: Architektur des 3D-CNN

- **Pooling Layer #5:** Es wird ein Max Pooling durchgeführt. Der Stride ist (2,2,2).
- **Dense Layer #1:** Die Anzahl an Neuronen ist 4096. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet. Ein Dropout, mit einer Wahrscheinlichkeit von 45%, wird druchgeführt.
- **Dense Layer #2:** Die Anzahl an Neuronen ist 4096. Zur Aktivierung wird eine Relu verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet. Ein Dropout, mit einer Wahrscheinlichkeit von 40%, wird druchgeführt.
- **Dense Layer #3:** Dies ist die Ausgabeschicht und hat ein Neuron. Zur Aktivierung wird die Sigmoidfunktion verwendet. Als Initalisierer wird der HeUniform-Initialisierer verwendet.

5. Experimente und Evaluation

5.1. Training

5.1.1. 1D-OSCNN

Training 1D-OSCNN #1

Training 1D-OSCNN #2

Training 1D-OSCNN #3

5.1.2. Xception

Training Xception #1

Training Xception #2

Training 3D-CNN #1

5.1.3. 3D-CNN

Training 3D-CNN #1

Training 3D-CNN #1

Training 3D-CNN #1

5.2. Test

5.2.1. 1D-OSCNN

Test 1D-OSCNN #1

Test 1D-OSCNN #2

Test 1D-OSCNN #3

5.2.2. Xception

Test Xception #1

Test Xception #2

Test 3D-CNN #1

5.2.3. 3D-CNN

Test 3D-CNN #1

Test 3D-CNN #1

Test 3D-CNN #1

5.3. Einordnung der Ergebnisse

5.4. Benchmark

6. Diskussion

Im Zuge dieser Bachelorarbeit wurde ein Gestenerkennungssystem, mithilfe einer Stereokamera, in ROS entwickelt und implementiert. Dieses Ziel wurde, nach Schwierigkeiten mit veralteten Rechner und nicht aktueller ROS-Distribution, erfolgreich erreicht. Darüber hinaus wurde, mit zwei entwickelten ROS-Paketen, gezeigt, wie dieses Gestenerkennungssystem in ROS eingesetzt werden kann. Damit ist das Ziel, eine Grundlage für weitere Arbeiten und neue Interaktionsmöglichkeiten mit Robotersystemen zu schaffen, ebenfalls als erfüllt zu betrachten. In diesem Kapitel werden die erzielten Ergebnisse, die aufgetretenen Probleme und die gewonnenen Erkenntnisse zusammenfassend erläutert.
Das entwickelte und implementierte Gestenerkennungssystem ist funktionsfähig und bildet eine gute Grundlage, für zu entwickelnde Applikationen. Das System bindet die verwendete Stereokamera ein, greift auf die durch diese gelieferten Daten zu und verarbeitet die erhaltenen Daten. Die verarbeiteten Daten werden in ROS zur Verfügung gestellt und mit Hilfe dieser ein ”Skelett-Tracking” ermöglicht. Das ”Skelett-Tracking” wurde in den beiden entwickelten Paketen verwendet. Das System verfügt, über das verwendete ”Skelett-Tracking” hinaus, über weitere Funktionen, wie zum Beispiel die Erkennung von bestimmten Bewegungsabläufen.
Zu Beginn der Entwicklung traten mehrere Probleme auf. Das erste Problem, welches viel Zeit kostete, wurde durch die virtuelle Maschine ausgelöst. Das Linux-Betriebssystem, welches in der virtuellen Maschine lief, konnte nur fehlerhafte Daten von der Stereokamera empfangen. Dies Problem wurde durch eine Anpassung der USB-Kompatibilitätseinstellungen, von USB 2.0 auf USB 3.0, gelöst. Doch auch nach dieser Anpassung, wurde die Verbindung zu Stereokamera in Einzelfällen noch unterbrochen. Um diese Art von Problemen zu vermeiden, sollte, bei weiteren Arbeiten oder Weiterentwicklungen am System, wenn möglich auf eine native Linux-Installation zurückgegriffen werden. Eine native Linux-Installation ist ebenfalls für die gesamte Systemperformance positiv. Als zweites, aber kleineres, Problem zeigte sich, dass Alter der verwendeten ROS-Distribution. Durch das, im Mai 2016, zurückliegende Veröffentlichungsdatum von ”ROS Kinetic Kame” und durch die Tatsache, dass es bereits seit Mai 2018 eine neue ROS-Distribution gibt, wurden verschiedene Komponenten des Gestenerkennungssystems, für ”ROS Kinetic Kame”, nicht mehr weiterentwickelt. Aufgrund dieser Tatsache, musste auf nicht aktuelle Versionen der Komponenten zurückgegriffen werden. Dies führte zu einem Mehraufwand in der Entwicklung, aber nicht zu einer feststellbaren Einschränkung des Systems. Bei der entwickelten Gestensteuerung, für den Roboterarm, traten Probleme bei der Bewegungsplanung auf. Es wird hier nur unzuverlässig ein Bewegungspfad gefunden. Das Problem wurde auf die Berechnung der inversen Kinematik und den verwendeten Pfadplaner, innerhalb von ”MoveIt!”, eingegrenzt. Der Versuch eine Lösung, innerhalb der verfügbaren Zeit, zu finden, war leider nicht erfolgreich. Mit der hier durchgeführten Eingrenzung des Problems, wird die Suche nach einer Lösung jedoch kein großes Problem darstellen.

Als erste Erkenntnis steht am Ende dieser Arbeit, dass die natürliche Interaktion, des Menschen mit technischen Systemen, viel Potential ins sich trägt. Durch weitere Arbeiten in diesem Bereich, werden ganz sicher neue Ansätze entstehen und somit die Interaktion mit Robotern, neu definiert werden. Als zweite Erkenntnis steht, dass ROS, durch sein modulares Konzept und die daraus resultierende einfache Integration neuer Hardware und Software, eine passende Umgebung für weitere Entwicklungen in diesem Bereich darstellt.

6.1. Ausblick

Diese Bachelorarbeit bietet eine Grundlage für weitere Entwicklungen, in dem Bereich der natürlichen Interaktion mit Robotersystemen. Die, zu Beginn der Arbeit, gesteckten Ziele wurden erreicht, jedoch sind weitere Arbeiten an dem System notwendig. Als erstes ist hier die Migration des Systems auf die aktuellste ROS-Distribution zu nennen. Im Zuge der Migration werden Aktualisierungen, an den Systemkomponenten, durchgeführt werden müssen. Gegebenenfalls wird es auch notwendig sein, dass Komponenten des Systems ausgetauscht werden müssen. Neben den Aktualisierungen der Software, bietet sich der Austausch der ”Kinect”, durch eine aktuellere und leistungsstärkere Stereokamera, an. Durch einen Austausch lassen sich, durch die höhere Genauigkeit aktueller Stereokameras, neue Funktionen implementieren. Durch neue Funktionen lassen sich wiederum neue Anwendungen, für das Gestenerkennungssystem, finden. Aus der Entwicklung der Gestensteuerung, für den Roboterarm, ging hervor, dass die Bewegungsplanung noch nicht zufriedenstellend funktioniert. Hier empfiehlt es sich die Pakete ”rob_arm_small” und ”rob_arm_small_hw_interface” zu überarbeiten. Hier sollte das Hauptaugenmerk auf die Kinematik, die verwendeten Bewegungsplaner und auf die verwendete URDF-Datei gelegt werden.
Das Konzept, dass hinter dem Gestenerkennungssystem steht, ermöglicht es auch andere Sensoren, neben Stereokameras, zu integrieren. Somit können, in Weiterentwicklungen des Systems, verschiedene Sensoren miteinander kombiniert werden. Die Implementation einer Spracherkennung wäre beispielsweise eine Möglichkeit. Neben den genannten Weiterentwicklungen, gibt es die Möglichkeit, dass System in andere Projekte zu integrieren. Zum Beispiel eine Personenfolgefunktion, für mobile Robotersysteme, könnte damit integriert werden. Abgesehen von so spezifischen Verwendungen, wie einer Personenfolgefunktion, kann auch nur die Umgebungswahrnehmung, von bestehenden Robotersystemen, erweitert werden.

A. Quellcode

```
/**
 ******************************************************************************
 * @file        move_grop_interface.cpp
 * @author      Oliver Bosin
 * @version     V1.0.0
 * @date        13.06.2019
 * @copyright   2011 - 2019 UniBw M - ETTI - Institute 4
 * @brief       Node to control the roboticarm rob_arm_small
 * @details     This Node listen to the tf of skeleton broadcasted by
 *              openni_tracker and calculate a goal for the roboticarm
 *              which is then executed by the framework MoveIt!
 *
 *          1. Listeners
 *                - tfListener
 *
 ******************************************************************************
 *  @par History:
 *
 *  @details V1.0.0 13.06.2019 Oliver Bosin
 *          - Initial Release
 ******************************************************************************
 * @todo    Optimize tolerances, try asyncExecute() funktion
 ******************************************************************************
 * @bug   none
 ******************************************************************************
 */

#include <ros/ros.h>
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <tf2/LinearMath/Vector3.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>
#include <moveit_visual_tools/moveit_visual_tools.h>

/**
  * @brief    Main function for move_group_interface
  * @details  In this function tf-listener listen to tf broadcasted
  *           by openni_tracker and calculate a goal for the
  *           roboticarm which is then executed by MoveIt!
  * @param    [in] argc: Non-negative value representing the number of
```

```
44   *                      arguments passed to the program from the
45   *                      environment in which the program is run.
46   * @param   [in] argv: Pointer to an array of pointers to null-terminated
47   *                      multibyte strings that represent the arguments
48   *                      passed to the program from the execution
49   *                      environment
50   * @retval  If the return statement is used, the return value is used as
51   *          the argument to the implicit call to exit().
52   *          This value can be:
53   *                      @arg EXIT_SUCCESS [indicate successful termination]
54   *                      @arg EXIT_FAILURE [indicate unsuccessful termination]
55   */
56 int main(int argc, char** argv){
57
58     ros::init(argc, argv, "move_group_interface");
59
60     ros::NodeHandle node_handle;
61
62     ros::AsyncSpinner spinner(1);
63
64     spinner.start();
65
66     /* MoveIt! declarations and initializations */
67     static const std::string PLANNING_GROUP_ARM = "roboter_arm";
68     static const std::string PLANNING_GROUP_GRIPPER = "gripper";
69
70     moveit::planning_interface::MoveGroupInterface move_group_arm(
           PLANNING_GROUP_ARM);
71     moveit::planning_interface::MoveGroupInterface move_group_gripper(
           PLANNING_GROUP_GRIPPER);
72
73     // Raw pointers are frequently used to refer to the planning group for
           improved performance.
74
75     const robot_state::JointModelGroup* joint_model_group_arm =
76
77         move_group_arm.getCurrentState()->getJointModelGroup(
               PLANNING_GROUP_ARM);
78
79     const robot_state::JointModelGroup* joint_model_group_gripper =
80
81         move_group_arm.getCurrentState()->getJointModelGroup(
               PLANNING_GROUP_GRIPPER);
82
83       move_group_arm.clearPoseTargets();
84   move_group_arm.setStartStateToCurrentState();
85   move_group_arm.setPoseReferenceFrame("base_link");
86
87   move_group_arm.setGoalPositionTolerance(0.02);
88   move_group_arm.setGoalJointTolerance(0.07);
89   move_group_arm.setPlanningTime(0.08);
90
```

```
91  move_group_gripper.clearPoseTargets();
92  move_group_gripper.setStartStateToCurrentState();
93  move_group_gripper.setPoseReferenceFrame("base_link");
94
95  move_group_gripper.setGoalTolerance(0.025);
96  move_group_gripper.setPlanningTime(0.08);
97
98  /* Here a TransformListener object is created.
99       * Once the listener is created it starts receiving tf2
         transformations over the wire
100      * and buffers them for up to 10 seconds*/
101
102 tf2_ros::Buffer tfBuffer;
103 tf2_ros::TransformListener tfListener(tfBuffer);
104
105 /* Declarations of messages to store Transforms
106
107     geometry_msgs::TransformStamped transformStamped_left_hand_1;
108 geometry_msgs::TransformStamped transformStamped_right_hand_1;
109 geometry_msgs::TransformStamped transformStamped_shoulder_to_elbow;
110     geometry_msgs::TransformStamped transformStamped_elbow_to_hand;
111
112 /*loop to get transform from shoulder to elbow and elbow to hand for
    armlength*/
113
114     while (node_handle.ok()){
115
116         try{
117
118           transformStamped_shoulder_to_elbow = tfBuffer.lookupTransform("
              left_shoulder_1", "left_elbow_1",
119                                                ros::Time(0));
120
121           transformStamped_elbow_to_hand = tfBuffer.lookupTransform("
              left_elbow_1","left_hand_1",
122                                                ros::Time(0));
123         }
124         catch (tf2::TransformException &ex) {
125           ROS_WARN("%s",ex.what());
126
127
128           continue;
129         }
130         break;
131     }
132
133     /*convert from geometry_msgs::Vector3 to tf2::Vector3 so the use of
        the length() function is possible*/
134
135     tf2::Vector3 shoulder_to_elbow_vector;
136 tf2::Vector3 elbow_to_hand_vector;
137 tf2::Vector3 rotate = {0.0,0.0,1.0};
```

```
138   tf2::Vector3 distance;
139
140   shoulder_to_elbow_vector[0] = transformStamped_shoulder_to_elbow.
          transform.translation.x;
141   shoulder_to_elbow_vector[1] = transformStamped_shoulder_to_elbow.
          transform.translation.y;
142   shoulder_to_elbow_vector[2] = transformStamped_shoulder_to_elbow.
          transform.translation.z;
143
144   elbow_to_hand_vector[0] = transformStamped_elbow_to_hand.transform.
          translation.x;
145   elbow_to_hand_vector[1] = transformStamped_elbow_to_hand.transform.
          translation.y;
146   elbow_to_hand_vector[2] = transformStamped_elbow_to_hand.transform.
          translation.z;
147
148   /*calculate armlength of user*/
149
150   double armlength = shoulder_to_elbow_vector.length() +
          elbow_to_hand_vector.length();
151
152   /*calculate conversion factor for position target*/
153
154   double conversion_factor = 0.38 / armlength;
155
156
157   moveit::planning_interface::MoveGroupInterface::Plan my_plan;
158   moveit::core::RobotStatePtr current_state;
159   std::vector<double> joint_group_positions;
160
161   ros::Rate rate(60.0);
162
163   /*main loop*/
164
165   while (node_handle.ok()){
166
167     try{
168
169       transformStamped_left_hand_1 = tfBuffer.lookupTransform("torso_1", "
              left_hand_1",
170                                                 ros::Time(0));
171
172       /* optional Transform calculation to implement grab functionality of
              the gripper
173
174          transformStamped_right_hand_1 = tfBuffer.lookupTransform("
               torso_1", "right_hand_1",
175                                                    ros::Time(0));
176       */
177         }
178         catch (tf2::TransformException &ex) {
179
```

```
180             ROS_WARN("%s",ex.what());
181         continue;
182 
183           }
184 
185     /* calculate new position target for endeffector */
186     distance[0] = conversion_factor * (transformStamped_left_hand_1.
            transform.translation.x - 0.15);
187     distance[1] = conversion_factor * (transformStamped_left_hand_1.
            transform.translation.z*(-1.0));
188     distance[2] = conversion_factor * (transformStamped_left_hand_1.
            transform.translation.y - 0.15);
189 
190     distance = distance.rotate(rotate, 0.800);
191      move_group_arm.setPositionTarget(distance[0], distance[1], distance[2])
            ;
192 
193      ROS_INFO_NAMED("planning","x : %lf y: %lf z: %lf", distance[0],
            distance[1], distance[2]);
194      ROS_INFO_NAMED("distance","distance: %lf", distance.length());
195      bool succes = (move_group_arm.plan(my_plan) == moveit::
            planning_interface::MoveItErrorCode::SUCCESS);
196 
197      if(succes){
198 
199        succes = false;
200        joint_group_positions = my_plan.trajectory_.joint_trajectory.points.
               back().positions;
201        joint_group_positions[4] = -0.0;
202        move_group_arm.setJointValueTarget(joint_group_positions);
203        succes = (move_group_arm.plan(my_plan) == moveit::planning_interface
               ::MoveItErrorCode::SUCCESS);
204 
205        if(succes){
206 
207          move_group_arm.execute(my_plan);
208 
209        }
210      }
211 
212   /* optional gripper control */
213 
214      if(true){
215 
216      //joint_group_positions[0] = -0.300;
217 
218        //move_group_gripper.setJointValueTarget(joint_group_positions);
219 
220        //move_group_gripper.move();
221      }
222      rate.sleep();
223   }
```

```
224     return 0;
225
226 }
```

Listing A.1: move_group_interface.cpp

```
1  /**
2   ******************************************************************************
3   * @file        gesture_control.cpp
4   * @author      Oliver Bosin
5   * @version     V1.0.0
6   * @date        22.02.2019
7   * @copyright   2011 - 2019 UniBw M - ETTI - Institute 4
8   * @brief       Node to control the turtle in turtlesim
9   * @details     This Node listen to the tf of skeleton broadcasted by
10  *              openni_tracker
11  *          and then publish messages to change velocity and angle of the
12  *              turtle
13  *              Therefore following publishers are created.
14  *
15  *          1. PUBLISHERS
16  *               -#  turtle_vel (publish on "/UDPcommand" topic)
17  *
18  ******************************************************************************
19  *  @par History:
20  *
21  *  @details V1.0.0 22.02.2019 Oliver Bosin
22  *          - Initial Release
23  ******************************************************************************
24  * @todo
25  ******************************************************************************
26  * @bug  none
27  ******************************************************************************
28  */
29 #include <ros/ros.h>
30 #include <tf2_ros/transform_listener.h>
31 #include <geometry_msgs/TransformStamped.h>
32 #include <geometry_msgs/Twist.h>
33 #include <turtlesim/Spawn.h>
34
35
36 /**
37   * @brief   Main function for gesture_control
38   * @details In this function tf-listener listen to tf broadcasted by
39   *          openni_tracker and calculates the values to be send by
40   *          the publisher turtle_vel to control the turtle in
41   *          turtlesim
42   * @param   [in] argc: Non-negative value representing the number of
43   *                     arguments passed to the program from the
44   *          environment in which the program is run.
45   * @param   [in] argv: Pointer to an array of pointers to null-terminated
46   *                     multibyte strings that represent the arguments
```

```
47    *                     passed to the program from the execution
48    *                     environment
49    * @retval  If the return statement is used, the return value is used as
50    *          the argument to the implicit call to exit().
51    *          This value can be:
52    *      @arg EXIT_SUCCESS [indicate successful termination]
53    *      @arg EXIT_FAILURE [indicate unsuccessful termination]
54    */
55  int main(int argc, char** argv){
56    ros::init(argc, argv, "gesture_control");
57
58    ros::NodeHandle node;
59
60    ros::service::waitForService("spawn");
61    ros::ServiceClient spawner =
62      node.serviceClient<turtlesim::Spawn>("spawn");
63    turtlesim::Spawn turtle;
64    turtle.request.x = 4;
65    turtle.request.y = 2;
66    turtle.request.theta = 0;
67    turtle.request.name = "turtle2";
68    spawner.call(turtle);
69
70    ros::Publisher turtle_vel =
71      node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);
72
73    /* Here a TransformListener object is created.
74     * Once the listener is created it starts receiving tf2 transformations
          over the wire
75     * and buffers them for up to 10 seconds*/
76    tf2_ros::Buffer tfBuffer;
77    tf2_ros::TransformListener tfListener(tfBuffer);
78
79    ros::Rate rate(10.0);
80    while (node.ok()){
81      geometry_msgs::TransformStamped transformStamped;
82      try{
83      /*here the transform between openni_link and left_hand_1 is calculated
84       *openni_link -> kinect , left_hand_1 -> right hand of person*/
85        transformStamped = tfBuffer.lookupTransform("openni_link", "
             left_hand_1",
86                                    ros::Time(0));
87      }
88      catch (tf2::TransformException &ex) {
89        ROS_WARN("%s",ex.what());
90
91        ros::Duration(1.0).sleep();
92        continue;
93      }
94
95      geometry_msgs::Twist vel_msg;
96    /*if right hand is right of the kinect from the perspective of the person
```

```
      the
97    *turtle will move forward*/
98     if(transformStamped.transform.translation.y > 0.2)
99     {
100      vel_msg.linear.x = 1.0;
101  }
102    else{
103    /*if right hand is left of the kinect from the perspective of the
         person the
104      *turtle will move backward*/
105    if(transformStamped.transform.translation.y < -0.2){
106      vel_msg.linear.x = - 1.0;
107    }
108    else{
109      /*if right hand is in front of the kinect from the perspective of the
           person the
110       *turtle will stop*/
111      vel_msg.linear.x = 0.0;
112      }
113  }
114  /*if right hand is above the kinect from the perspective of the person
       the
115   *turtle will turn left*/
116    if(transformStamped.transform.translation.z > 0.3){
117    vel_msg.angular.z = 1.0;
118  }
119    else{
120    /*if right hand is below the kinect from the perspective of the person
         the
121     *turtle will turn right*/
122    if(transformStamped.transform.translation.z < -0.3){
123    vel_msg.angular.z = - 1.0;
124    }
125    else{
126      /*if right hand is in front of the kinect from the perspective of the
           person the
127       *turtle will not turn*/
128      vel_msg.angular.z = 0.0;
129    }
130  }
131    turtle_vel.publish(vel_msg);
132
133    rate.sleep();
134  }
135  return 0;
136};
```

Listing A.2: gesture_control.cpp

B. Arbeitsaufwand

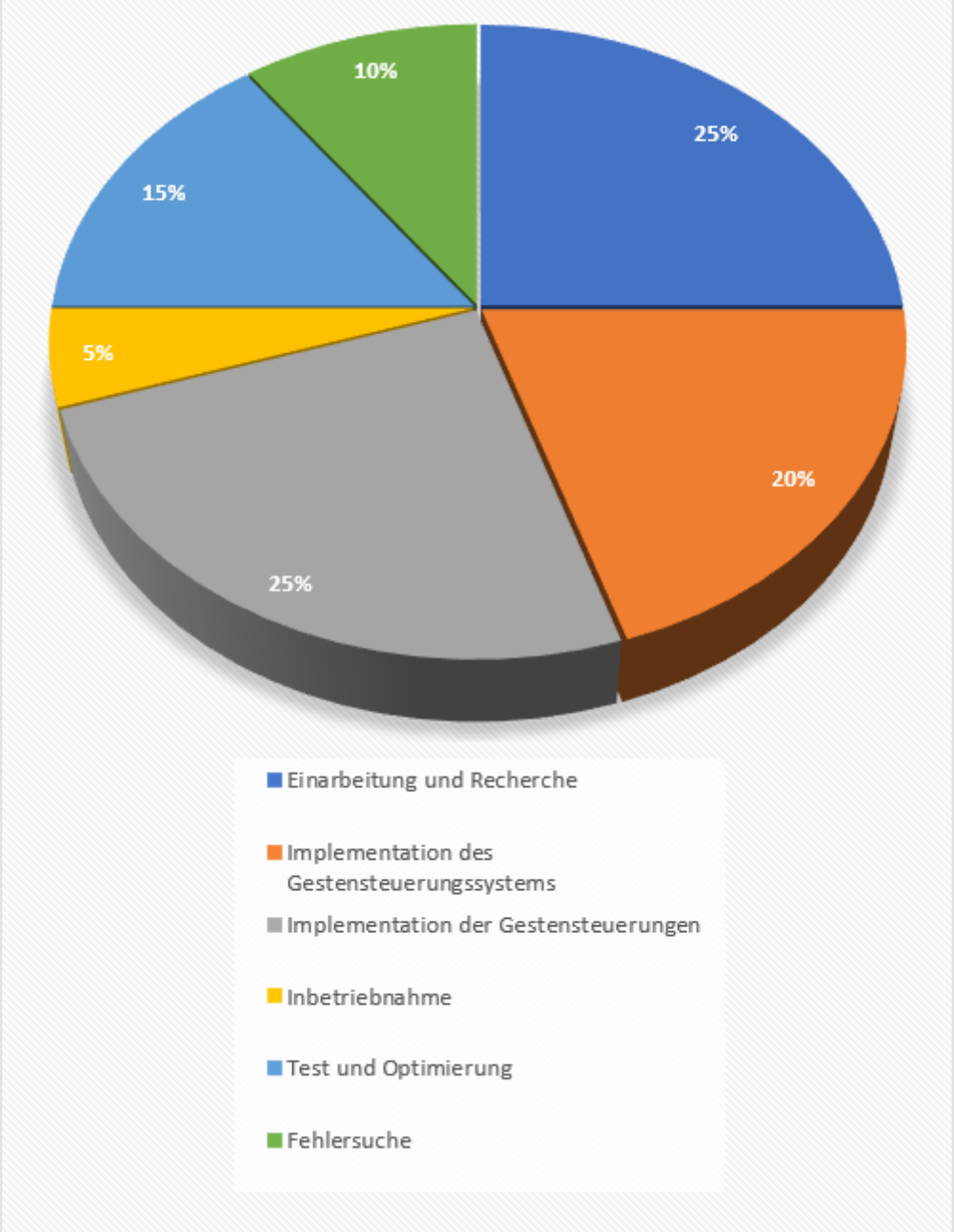

Abbildung B.1.: Arbeitsaufwand

Index