Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterthesis

Hassib Shafaq

Greifen von unbekannten Objekten mittels neuronaler Netze in einer mobilen Robotik-Anwendung

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Hassib Shafaq

Greifen von unbekannten Objekten mittels neuronaler Netze in einer mobilen Robotik-Anwendung

Masterthesis eingereicht im Rahmen der Masterprüfung
im Masterstudiengang Automatisierung
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. -Ing. Jochen Maaß
Zweitgutachter : Prof. Dr. -Ing. Thomas Frischgesell

Abgegeben am 31. Januar 2020

Hassib Shafaq

Thema der Masterthesis

Greifen von unbekannten Objekten mittels neuronaler Netze in einer mobilen Robotik-Anwendung

Stichworte

Mobile Robotik, Neuronales Netz, Bildverarbeitung, Greifposengenerierung, Deep Learning, ROS, CNN, Python, C++, Qt, Keras, TurtleBot, OpenManipulator, OpenCV

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung eines mobilen Roboters zum Greifen von unbekannten Objekten mittels eines neuronalen Netzes. Als mobile Basis des Roboters wird der TurtleBot3 verwendet. Dieser wird mit einem 6DOF Robotermanipulator (OpenManipulator) und einer Intel RealSense D435 Tiefenbildkamera ausgestattet. Über ein CNN werden die Tiefenbildaufnahmen von Objekten ausgewertet und Greifposen generiert. Der Robotermanipulator fährt das Objekt an der optimalen Position und Orientierung an und greift es.

Zunächst werden sämtliche Roboterkomponenten analysiert und in Betrieb genommen. Anschließend wird die Theorie zum Greifen von unbekannten Objekten mittels CNN behandelt. Für das Training des CNN's wird die Cornell-Datenbank verwendet. Das CNN wird dann über ROS in die Robotersteuerung implementiert. Das Greifen von unbekannten Objekten ist insgesamt erfolgreich umgesetzt worden.

Hassib Shafaq

Title of the paper

Grasping unknown objects using neural networks in a mobile robotics application

Keywords

Mobile robotics, neural netzwork, image processing, grasping pose generation, Deep Learning, ROS, CNN, Python, C++, Qt, Keras, TurtleBot, OpenManpulator, OpenCV

Abstract

This thesis deals with the development of a mobile robot for grasping unknown objects using a neural network. The mobile base of the robot is the TurtleBot3. It will be equipped with a 6DOF robot manipulator and an Intel RealSense D435 depth-image camera. The depth image recordings of objects are evaluated via CNN and grasping poses are generated. The robot manipulator approaches the object at the optimal position and orientation and grasps it.

Initially, all robot components are analyzed and put into operation. Then the theory of grasping unknown objects via CNN is discussed. The Cornell database is used for the CNN's training. The CNN is then implemented into the robot controller via ROS. The grasping of unknown objects has been successfully implemented.

Danksagung

Die vorliegende Masterarbeit mit dem Titel „Greifen von unbekannten Objekten mittels neuronaler Netze in einer mobilen Robotik-Anwendung “ entstand im Rahmen meines Studiums an der Hochschule für Angewandte Wissenschaften in Hamburg und wurde von Prof. Dr. Jochen Maaß und Prof. Dr. Thomas Frischgesell betreut. Für die Betreuung während der Zeit möchte ich mich zunächst herzlichst bedanken.

Außerdem möchte ich Herrn Prof. Thomas Frischgesell für das entgegengebrachte Vertrauen und die finanzielle Unterstützung des Projektes danken. Dies eröffnete mir neue Möglichkeiten und einen großen Gestaltungsfreiraum. Zudem möchte ich mich für die tolle Zusammenarbeit über die letzten Jahre bedanken.

Bedanken möchte ich mich außerdem bei Ing. Robin Auffermann und Dipl. -Ing. Carolina Bohnert von der HAW Hamburg. Beide waren jederzeit ansprechbar und haben maßgeblich zur Umsetzung dieser Arbeit beigetragen. Auch Herrn Constantin Titgemeyer möchte ich an dieser Stelle danken. Bei technischen Problemen hatte er stets ein offenes Ohr und konnte entscheidende Beiträge liefern. Die Zusammenarbeit hat mir viel Freude gebracht.

Ein besonderer Dank geht an meine Eltern, meine Geschwister und meinen Freunden. Sie haben mich während meines Studiums aufopferungsvoll unterstützt.

Hamburg, 31.01.2020

Hassib Shafaq

Inhaltsverzeichnis

Abkürzungsverzeichnis

TB3	TurtleBot3 Waffle Pi
OM3	OpenManipulator mit drei Freiheitsgraden
OM6	OpenManipulator mit sechs Freiheitsgraden
TCP	Tool Center Point
GG-CNN	Generative Grasping Convolutional Neural Network [11]
CNN	Convolutional Neural Network (neuronale Faltungsnetzwerke)
EE	Endeffektor eines Robotermanipulators
IDE	Integrierte Entwicklungsumgebung
API	Anwendungsprogrammierschnittstelle
CSI	Camera Serial Interface
PTP	Point-to-Point-Bewegung eines Robotermanipulators
SBC	Single Board Computer
RPI	Raspberry Pi

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Mit Robotern verbinden viele Menschen heutzutage noch immer hochintelligente, humanoide Androiden aus Science-Fiction Filmen, wie z.B. *Data* aus *Star Trek*. In dieser Serie wird *Data* als äußerst menschlicher Android porträtiert, welcher neben seiner extrem logischen Intelligenz, auch über eine gewisse Empathie verfügt. Von so einem menschenähnlichen Androiden können Forscher und Entwickler zum aktuellen Zeitpunkt nur träumen. Jedoch werden stetig große Schritte in diese Richtung getätigt, zumindest wenn es um die logische Intelligenz geht. Besonders die Forschungsergebnisse aus den Bereichen der künstlichen Intelligenz waren in den letzten Jahren beeindruckend, speziell jene auf dem Gebiet des *Deep Learnings*. Dank dieser Fortschritte konnten besonders die Sprach- und die Bilderkennungsalgorithmen signifikant verbessert werden. Infolgedessen sind diverse Sprachassistenz- und Bilderkennungssysteme in den öffentlichen Handel gelangt, vorrangig in Form von Zusatzfunktionen für Mobiltelefone. Dadurch geriet dieses Forschungsthema in den Fokus der Öffentlichkeit und sorgt seitdem für großes Aufsehen.

An sich sind neuronale Netze jedoch keine neue Entdeckung. Die ersten Ansätze existieren mit dem McCulloch-Pitts Neuron schon seit den 1940er Jahren. In seiner Funktion ist es einer biologischen Nervenzelle nachempfunden worden [10]. Mathematisch betrachtet, ist eine Nervenzelle nur ein Bindeglied zwischen einem Sender und einem Empfänger, welche nur dann ein Signal weiterleitet, wenn ein bestimmter Schwellenwert am Eingang überschritten wird. Beim McCulloch-Pitts Neuron werden die Eingangwerte noch zusätzlich mit einer Gewichtung multipliziert, bevor die Summe der Produkte an eine Aktivierungsfunktion weitergegeben wird. Der ausgegebene Wert nach der Schwellenwertüberschreitung konnte in seiner damaligen Grundversion nur binäre Werte annehmen, ist aber heutzutage

abhängig von der gewählten Aktivierungsfunktion. Das McCulloch-Pitts Neuron stellt den Grundbaustein neuronaler Netze dar.
Die ersten neuronalen Netze waren einschichtige Zusammenschlüsse von McCulloch-Pitts Neuronen. Sie konnten lediglich einen Satz von n Eingabewerten $\boldsymbol{x}$ entgegennehmen und über $f(\boldsymbol{x}, \boldsymbol{w}) = x_1 w_1 + ... + x_n w_n$ eine Ausgabe y berechnen. Hierbei mussten die entsprechenden Gewichtungen $\boldsymbol{w}$ noch manuell gesetzt werden. Jedoch änderte sich dies mit dem Rosenblatt-Perzeptron [18] und dem Adaline-Modell [20], welche erstmals die automatische Anpassung der Netzgewichtungen anhand gegebener Daten ermöglichten. Für die Gewichtungsanpassung wird das stochastische Gradientenabstiegsverfahren genutzt. Auch heute sind dies gängige Verfahren zum Trainieren von einfachen Netzen.

Die grundlegende Theorie zu neuronalen Netzen hat sich seither kaum verändert. Das öffentliche, aber auch wissenschaftliche Interesse an neuronalen Netzen ist seitdem stetig gesunken. Im Jahr 2012 konnten sich dann jedoch Alex Krizhevsky und Geoffrey Hinton mit deutlichem Vorsprung zur Konkurrenz beim *ImageNet* Wettbewerb für Bildklassifizierung durchsetzen. Im Vorjahr lag die beste Korrektklassifizierungsrate bei 74.3%. Krizhevsky und Hinton gelang es, mit ihrem trainierten neuronalen Faltungsnetzwerk (CNN) eine Korrektklassifizierungsrate von 83.6% zu erreichen und somit eine deutliche Verbesserung im Vergleich zum Vorjahr zu erbringen [6]. Letztlich hat der Gewinner des *ImageNet*-Wettbewerbs im Jahr 2015 mit Hilfe eines Faltungsnetzwerks eine Korrektklassifizierungsrate von 96.4% erreicht [2, vgl. S.17].
Derartige Erfolge haben quasi zu einer "Renaissance" der künstlichen Intelligenz geführt und auch neues Potential für andere Forschungsfelder, unter anderem auch der Robotik, geschaffen. Nicht letztlich hat die massive Leistungssteigerung der verfügbaren Computerhardware in den letzten Jahrzehnten zu diesen Ergebnissen beigetragen. Dadurch ist es Forschern und Entwicklern heute möglich, komplexe neuronale Netze in relativ kurzer Zeit zu trainieren und somit beeindruckende Ergebnisse zu liefern.

Das Themengebiet der *Robot Vision* behandelt die Schnittstelle zwischen der Robotik und der Bildverarbeitung. Hierbei geht es vorrangig um die Erfassung und Interpretation von Umgebungsinformationen über entsprechende Kamerasysteme. Diese Informationen ermöglichen Robotern die Interaktion mit deren Umgebung. Für einfache Interaktionen reichen oft schon analytische Bildverarbeitungsalgorithmen aus. Die Anforderungen bzgl. einer intuitiven Interaktion zwischen einem Roboter und seiner dynamischen Umgebung sind weitaus höher. Deshalb hat sich in den letzten Jahren zunehmend der Gebrauch von neu-

ronalen Faltungsnetzwerken in der *Robot Vision* etabliert. Deren besondere Stärke besteht darin, verallgemeinerungsfähig und translationsinvariant zu sein. Damit lassen sich bekannte Merkmale von antrainierten Objekten auch auf unbekannte Objekte übertragen, ähnlich wie es Menschen tun.
Ein Beispiel hierfür ist der sog. "Bin-Picking"-Anwendungsfall bzw. der "Griff in die Kiste". Hierfür wird eine Kiste mit beliebigen, unsortierten Objekten vor dem Roboter platziert. Die Objekte in der Kiste müssen für den Robotermanipulator greif- und aufhebbar sein. Mit Hilfe einer Kamera wird anschließend analysiert, welches Objekt vom Roboter gegriffen werden kann. Für die Analyse werden vermehrt neuronale Faltungsnetzwerke genutzt ([11], [9]). Analytisch betrachtet stellt dieses Problem eine echte Herausforderung dar, auch wenn Menschen es mühelos bewältigen können.

Die Motivation dieser Arbeit besteht im Wesentlichen darin, die genannten Fortschritte aus dem Bereich der künstlichen Intelligenz, auch auf mobile Robotikanwendungen zu übertragen. Hierfür wird ein mobiler Roboter entwickelt, welcher Objekte erkennen, klassifizieren und greifen soll. Vor allem dient dieser Roboter als Grundlage für zukünftige Projekte. Ein Endziel kann z.B. der Einsatz von beliebig vielen mobilen Robotern auf einer Spielfläche sein, welche gemeinsam über eine kollektive Intelligenz bestimmte Aufgaben in einer dynamischen Umgebung lösen können. Diese Arbeit soll deshalb auch das Potential für zukünftige Projekte abwägen und die Möglichkeiten und Einschränkungen des zu entwickelnden Roboters erörtern.

Für den Leser werden Grundkenntnisse im Bereich der neuronalen Netze, speziell der neuronalen Faltungsnetzwerke (CNN) vorausgesetzt. Ergänzend wird folgende Literatur empfohlen:

- *Deep Learning* von Goodfellow et al. [4]:
  Dieses Fachbuch bietet eine fast vollständig gebündelte Theorie zum Thema *Deep Learning* an. Von den Grundlagen bis hin zu aktuellen Forschungsthemen werden hier alle Bereiche ausführlich behandelt.

- *Deep Learning mit Python* von Chollet [2]:
  Dieses Werk wurde vom Keras Entwickler selbst geschrieben und behandelt kurz die grundlegenden Ideen hinter den diversen Themen des Machine- und Deep Learning

und deren Implementierung in Python über das Keras Framework. Es ist sehr Praxisnah, deckt aber ebenfalls ein großes Spektrum der aktuellen Entwicklungs- und Forschungsthemen ab.

1.2. Ziel der Arbeit

Diese Arbeit beschäftigt sich mit der Entwicklung eines mobilen Roboters zum Einsammeln von Objekten auf einer Spielfläche. Als mobile Basis wird der Turtlebot3 von *ROBOTIS* verwendet. Ergänzend wird auf dieser Basis ein 6DOF Robotermanipulator montiert.

Das Anwendungsziel besteht darin, Gegenstände auf einer festgelegten Spielfläche einzusammeln und in entsprechende Behälter zu legen. Im Fokus der Anwendung steht vor allem die kamerabasierte Analyse der vorab unbekannten Objektgeometrie und die Generierung der optimalen Greifpose des Robotermanipulators. Die einzusammelnden Objekte sind in keiner Weise markiert. Für die Analyse wird ein mit den Zielobjekten trainiertes neuronales Netz verwendet.

Für die Navigation des Roboters auf dem Spielfeld stehen ein Lidar-Sensor, eine Deckenkamera und eine am mobilen Roboter befestigte Kamera zur Verfügung. Als Entwicklungs- und Kommunikationsplattform wird ROS verwendet. Als Ergebnis werden Aussagen zur Machbarkeit und Sicherheit des Greifprozesses erarbeitet.

1.3. Struktur der Arbeit

Diese Arbeit ist in insgesamt fünf Hauptkapitel gegliedert worden. Das erste Kapitel beinhaltet die Einleitung. Im zweiten Kapitel wird der mobile Roboter vorgestellt. Dabei werden alle Komponenten einzeln präsentiert und in Betrieb genommen. Zudem wird der Entwicklungsprozess und das Interaktion der Einzelkomponenten beschrieben.
Das dritte Kapitel behandelt zunächst die Theorie hinter der Greifposengenerierung. Anschließend wird die Trainingsdatenbank analysiert und ein CNN auf Basis dessen trainiert. Dafür wird dann ein Algorithmus in Python entwickelt, welcher in die ROS-Umgebung des Roboters implementiert wird. Die generierten Greifposeninformationen werden dann

im vierten Kapitel dazu genutzt, um mit dem Robotermanipulator Objekte aufzusammeln. Am Ende des vierten Kapitels wird zudem die Qualität der Greifposengenerierung und des eigentlichen Greifprozesses bewertet. Letztlich wird die Arbeit im fünften Kapitel zusammengefasst. Außerdem werden diverse Optimierungsansätze für zukünftige Entwicklungen diskutiert.

2. Grundlagen

2.1. Technische Spezifikationen des Arbeitsrechners

Für die Umsetzung dieser Arbeit wird ein leistungsstarker Arbeitsrechner benötigt. Das liegt in erster Linie am Training und der Ausführung des CNN's. Folgende Hardwarespezifikationen sind gegeben:

Tabelle 2.1.: Systemspezifikationen des Arbeitsrechners

CPU	Intel Core i7-8700 CPU @ 3.20GHz × 12
GPU	Nvidia GeForce RTX 2070
RAM	32 GB (empfohlen: 64 GB)
Betriebssystem	Ubuntu 16.04 LTS

Außerdem müssen Programmiersprachen und Frameworks aus 2.2 installiert sein. Eine ausführliche Anleitung zur Installation der benötigten Software ist im Anhang von [2] zu finden.

Tabelle 2.2.: Zu installierende Software und dessen Versionen

Software	Version	Software	Version
Python	2.7 & 3.5	ROS	Kinetic Kame
Tensorflow	1.15	CUDA	10.1
Keras	2.2.4	cuDNN	7.6.5
OpenCV	2.4.9		

Im Folgenden werden alle Roboterkomponenten konfiguriert und in Betrieb genommen. Dazu sind diverse Sondereinstellungen notwendig. Diese werden in den jeweiligen Abschnitten erläutert.

2.2. Turtlebot3 Waffle Pi

Der TurtleBot3 ist eine weiterentwickelte Version der ursprünglichen TurtleBot-Serie. Insgesamt existieren drei Versionen des TurtleBots von unterschiedlichen Entwicklern. Der erste ROS-taugliche TurtleBot1 wurde im Jahr 2011 von *Willow Garage* präsentiert. Im Folgejahr veröffentlichte Yujin Robots die weiterentwickelte Version TurtleBot2. Die dritte und aktuellste Version der Serie wurde von ROBOTIS entwickelt und ist seit 2017 erwerbbar. In dieser Arbeit wird die *TurtleBot3 Waffle Pi* Variante eingesetzt.

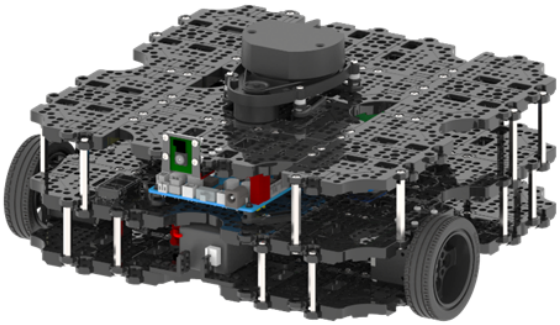

Abbildung 2.1.: Turtlebot3 Waffle Pi [14]

Die TurtleBot-Serie kann sich seit der Veröffentlichung der ersten Version großer Beliebtheit erfreuen. Der Turtlebot3 wird z.B. im Bildungsbereich, in der Forschung oder Entwicklung und beim Prototyping von Robotikanwendungen eingesetzt. Attraktiv ist insbesondere der verhältnismäßig günstige Kaufpreis, die einfache Programmierung, die Vielzahl an Funktionen und Sensoren und die vollständige Einbindung in die ROS-Umgebung. Durch den modularen Aufbau sind außerdem anwendungsspezifische Erweiterungen problemlos möglich. Aus diesen Gründen konnte sich der Turtlebot gegen Konkurrenten durchsetzen und sich mittlerweile als ROS-Standardplattform etablieren. Dies sind unter anderem die ausschlaggebenden Argumente, weshalb der TB3 auch in dieser Arbeit zum Einsatz kommt.

Im Folgenden werden die wichtigsten Spezifikationen und die Inbetriebnahme des TB3 beschrieben. Eine vollständige Ausarbeitung zum TB3 ist auf der Herstellerseite [14] zu finden.

2.2.1. Spezifikationen

Der TB3 ist mit den Maßen 281x306x141 (LxBxH, in mm) insgesamt sehr kompakt und mit 1,8 kg auch sehr leichtgewichtig, wodurch er sich insbesondere für Anwendungen in Innenräumen eignet.

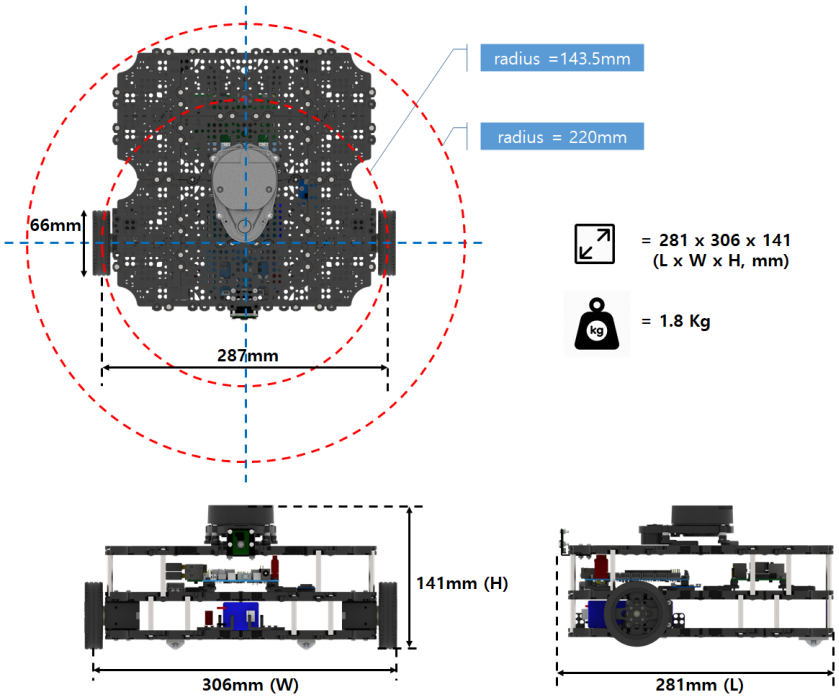

Abbildung 2.2.: Körpermaße des TB3 Waffle Pi [14]

Zudem umfasst er eine Vielzahl von Sensoren, Anschlüssen und Versorgungsmöglichkeiten. Standardmäßig besitzt er eine Vorderkamera, einen Lidar-Scanner zur Umgebungserfassung und eine IMU-Einheit. Außerdem kann er sowohl über einen Lipo-Akku, als auch extern über eine SMPS versorgt werden. Zudem ist er mit einem RPI 3 B+ und einem OperCR-Board ausgestattet. Der RPI 3 B+ wird als lokale Recheneinheit und als Kommunikationsinterface zum lokalen WLAN-Netzwerk und ROS genutzt. Das OpenCR-Board wird für die Versorgung des TB3 und zur Steuerung der Dynamixel-Servomotoren verwendet. Alle Hardware-Spezifikationen sind in Tabelle A.1 aufgelistet. Außerdem sind alle Hardwarekomponenten und deren Zusammenschluss abstrahiert in Abbildung 2.3 dargestellt.

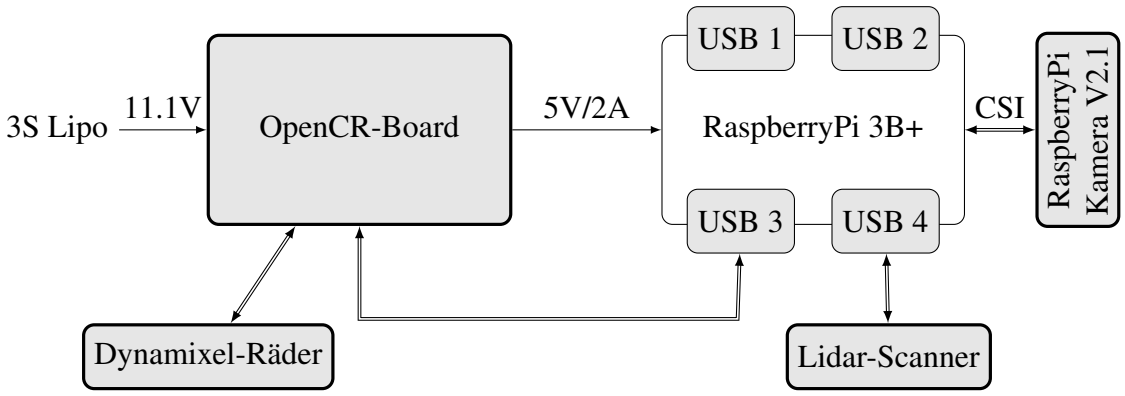

Abbildung 2.3.: Hardwarearchitektur des TB3

2.2.2. Konfigurationen

2.2.2.1. Konfiguration des Arbeitsrechners

Für eine erfolgreiche Inbetriebnahme des TB3 in ROS sind zuerst diverse Einstellungen vorzunehmen (siehe auch: [14, Abschnitt 5.2]). Folgende Software muss vorhanden sein bzw. installiert werden. Diese können mit den jeweils angegebenen Befehlen über das Terminal in Ubuntu installiert werden.

- Installation von ROS Kinetic Kame:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/....
robotis_tools/master/install_ros_kinetic.sh && chmod 755 ...
./install_ros_kinetic.sh && bash ./install_ros_kinetic.s
```

- Benötigte Zusatzpakete in ROS:

```
$ sudo apt-get install ros-kinetic-joy ros-kinetic-teleop-twist-joy ...
ros-kinetic-teleop-twist-keyboard ros-kinetic-laser-proc ...
ros-kinetic-rgbd-launch ros-kinetic-depthimage-to-laserscan ...
ros-kinetic-rosserial-arduino ros-kinetic-rosserial-python ...
ros-kinetic-rosserial-server ros-kinetic-rosserial-client ...
ros-kinetic-rosserial-msgs ros-kinetic-amcl ros-kinetic-map-server ...
ros-kinetic-move-base ros-kinetic-urdf ros-kinetic-xacro ...
ros-kinetic-compressed-image-transport ros-kinetic-rqt-image-view ...
ros-kinetic-gmapping ros-kinetic-navigation ros-kinetic-interactive-markers
```

- Zusatzpakete zum Bedienen des TB3. Diese können über die folgenden Befehle installiert, anschließend im *Catkin*-Ordner entpackt und erstellt werden:

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
$ cd ~/catkin_ws && catkin_make
```

2.2.2.2. Konfiguration des TB3

2.2.2.2.1. Raspberry Pi

Zum TB3 gehört standardmäßig der RPI 3B+ dazu. Dieser wird im Paket mitgeliefert. Der Hersteller ROBOTIS stellt dafür ein *Raspbian*-Image zum Herunterladen bereit [14, siehe Abschnitt 6.2.1.1], bei dem bereits diverse Pakete für den TB3 vorinstalliert sind. Für diese Arbeit ist diese Version jedoch nicht ausreichend. Durch den Einsatz der *Intel RealSense D435* und dessen Ansteuerung über den RPI, werden zusätzliche Programme und Treiber benötigt. Jedoch wird die *Intel RealSense D435* zum jetzigen Zeitpunkt nicht vom RPI 3B+ unterstützt. Ein weiteres Problem besteht darin, dass die RealSense D435 nur über einen USB 3.0 betrieben werden kann, der RPI 3B+ jedoch nur USB 2.0 Anschlüsse besitzt. Diese Probleme können mit der Anleitung aus Anhang A.3 umgangen werden. Dadurch ist es möglich, die RealSense D435 über die USB 2.0 Anschlüsse des RPI 3B+ zu bedienen. Durch die geringere Datenrate des USB 2.0 Anschlusses sind jedoch diverse Einschränkungen bei der Anwendung vorhanden, wie z.B. eine geringere Framerate und dass Fehlen einiger Filter. Im Laufe der Arbeit wird sich zeigen, inwieweit sich diese Einschränkungen auf das Endergebnis auswirken.

2.2.2.2.2. OpenCR-Board

Neben dem RPI 3B+ besitzt der TB3 das *OpenCR Board*. Diese Platine basiert auf dem Arduino Mega, besitzt jedoch noch einige Zusatzmodule und Anpassungen für den TB3. Vor allem beinhaltet sie diverse leistungselektronische Bauelemente und ist für die Stromversorgung des TB3 zuständig. Die Arduino-Basis des OpenCR-Boards ermöglicht auch dessen Programmierung über die offizielle Arduino-IDE. Für das OpenCR-Board des TB3 können innerhalb der Arduino-IDE zugehörige Treiberpakete heruntergeladen und auf dem Board

installiert werden. Eine ausführliche Anleitung ist beim Hersteller zu finden [14, siehe Abschnitt 6.3].

2.2.2.3. Netzwerkkonfiguration

Um den TB3 über WiFi mit ROS bedienen zu können, sind diverse Netzwerkeinstellungen vorzunehmen. Für einen erfolgreichen Verbindungsaufbau müssen sich zum einen der TB3 und der Arbeitsrechner im selben Netzwerk befinden und zum anderen müssen beiden Instanzen die IP-Adressen des jeweils anderen bekannt sein. Die folgenden Schritte sind deshalb sowohl auf dem Rechner, als auch auf dem TB3 durchzuführen:

1. Die IP-Adressen werden in den jeweiligen `bashrc`-Dateien des Arbeitsrechners und des TB3s angegeben. Über den folgenden Befehl können diese geöffnet werden:

   ```
   $ nano ~/.bashrc
   ```

2. Die `bash`-Dateien des Arbeitsrechners und des TB3s sind jeweils mit folgenden Zeilen zu ergänzen. Hier ist darauf zu achten, dass die richtigen IP-Adressen angegeben werden:

   a) Arbeitsrechner:

   ```
   ROS_MASTER_URI = https://IP_ADRESSE_DES_RECHNERS:11311
   ROS_HOSTNAME = IP_ADRESSE_DES_RECHNERS
   ```

   b) TB3:

   ```
   ROS_MASTER_URI = https://IP_ADRESSE_DES_RECHNERS:11311
   ROS_HOSTNAME = IP_ADRESSE_DES_TURTLEBOTS
   ```

3. Über den folgenden Befehl werden die Änderungen übernommen (alternativ auch durch das Neustarten des Terminals):

   ```
   $ source ~/.bashrc
   ```

2.2.3. Inbetriebnahme

Zuerst ist der TB3 über den entsprechenden Schalter am OpenCR-Board einzuschalten. Außerdem muss sichergestellt werden, dass der TB3 bzw. dessen RPI mit dem lokalen WiFi-Netzwerk verbunden ist. Für die Inbetriebnahme über ROS müssen die folgenden Schritte ausgeführt werden:

1. ROS über das Terminal des Arbeitsrechners starten:

   ```
   $ roscore
   ```

2. Über die SSH-Verbindung das ROS-Node des TB3s starten:

   ```
   $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
   ```

Der TB3 kann nun über ROS bedient und gesteuert werden, wenn alle oben genannten Befehle erfolgreich ausgeführt werden konnten. Um zu überprüfen, ob der TB3 erfolgreich in Betrieb genommen worden ist, kann z. B. getestet werden, ob dieser sich über das Keyboard des Arbeitsrechners steuern lässt. Dazu können folgende Befehle auf dem Terminal des Arbeitsrechners ausgeführt werden:

```
$ export TURTLEBOT3_MODEL=waffle_pi
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Der TB3 sollte nun über die Tasten A, W, D, S und X bewegt werden können. Für eine ausführlichere Anleitung, sei hier erneut auf die Bedienungsanleitung des Herstellers verwiesen (siehe [14]).

2.3. Open Manipulator

In der mobilen Robotik ist es oft notwendig, dass der Roboter auf bestimmte Weise mit seiner Umgebung interagiert. Das kann z.B. der Transport von Objekten innerhalb einer Lagerhausautomatisierung, die Umgebungsanalyse auf unbekanntem Terrain oder das Entschärfen von Sprengstoff sein, um nur einige Fälle zu nennen. In dieser Arbeit besteht die Interaktion in erster Linie darin, Objekte vom Boden aufzuheben, zu klassifizieren und zu transportieren. Dazu benötigt der TB3 natürlich einen Arm bzw. Manipulator. Für diesen Zweck kommt der OpenManipulator, ebenfalls vom Hersteller Robotis, zum Einsatz.

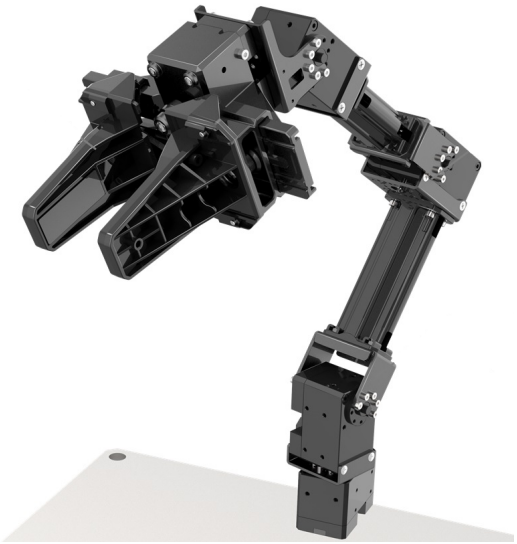

Abbildung 2.4.: 3DOF OpenManipulator von Robotis [12]

Der OpenManipulator zeichnet sich besonders durch sein *Open Source*-Konzept aus. Alle Komponenten und Programmcodes sind frei zugänglich und modifizierbar. Dadurch sind auch alle mechanischen Verbindungselemente verfügbar und können z.B. über einen 3D-Drucker hergestellt werden. Als Servomotoren kommen die *Dynamixel XM-430-W340* zum Einsatz. Zudem gehört der OM3, genau wie der TB3, auch zu den ROS-Standardplattformen und ist somit mit allen ROS Komponenten und Unterprogrammen voll kompatibel. Der Hersteller empfiehlt den OM3 besonders in Kombination mit dem TB3 für mobile Robotik Anwendungen, weshalb er auch unter anderem in dieser Arbeit eingesetzt wird.

Mit seinen drei Freiheitsgraden ist der OM jedoch unterbestimmt und besonders für das Greifen von Objekten nur begrenzt geeignet. Dank des erweiterbaren Konzepts des OM3s, kann dieser auf sechs Freiheitsgrade ergänzt werden. Damit könnten theoretisch alle Posen angefahren und alle Objekte gegriffen werden. Im Rahmen dieser Arbeit wurden deshalb diverse Konfigurationen und Varianten eines sechsachsigen OM über ein CAD-Tool konstruiert und getestet. Um einen Roboter über ROS bedienen zu können, sind zahlreiche Einstellungen vorzunehmen. Dazu muss der Roboter in einer URDF-Datei (XML-Syntax)

beschrieben werden. Dabei werden die Verhältnisse der einzelnen Achsen zueinander und die Darstellung des Roboter (z.B. über STL-Dateien) festgelegt. Anschließend wird diese XML-Datei über das ROS-Programm *MoveIt!* eingelesen und weiterverarbeitet. Die Inbetriebnahme von benutzerdefinierten Robotern in ROS gestaltet sich jedoch als äußerst aufwändig und ist nicht immer erfolgreich. Die Definition des Roboters in XML macht das Debuggen von Fehlerquellen zudem nahezu unmöglich. Für weitere Informationen bzgl. des benutzerdefinierten Roboterentwurfs für ROS siehe [21, S.398-455].

Aus diesen Gründen wird für diese Arbeit die sechsachsige Version des OMs vom Hersteller *ROBOTIS* verwendet. Da dieser sich noch in der Entwicklungsphase befindet und nicht offiziell unterstützt wird, wurde darauf auch nur nach schriftlicher Anfrage hingewiesen. Der OM6 mit einer am Handgelenk montierten Kamera ist in Abbildung 2.5 dargestellt. Dieser wird im weiteren Verlauf dieses Unterkapitels näher betrachtet.

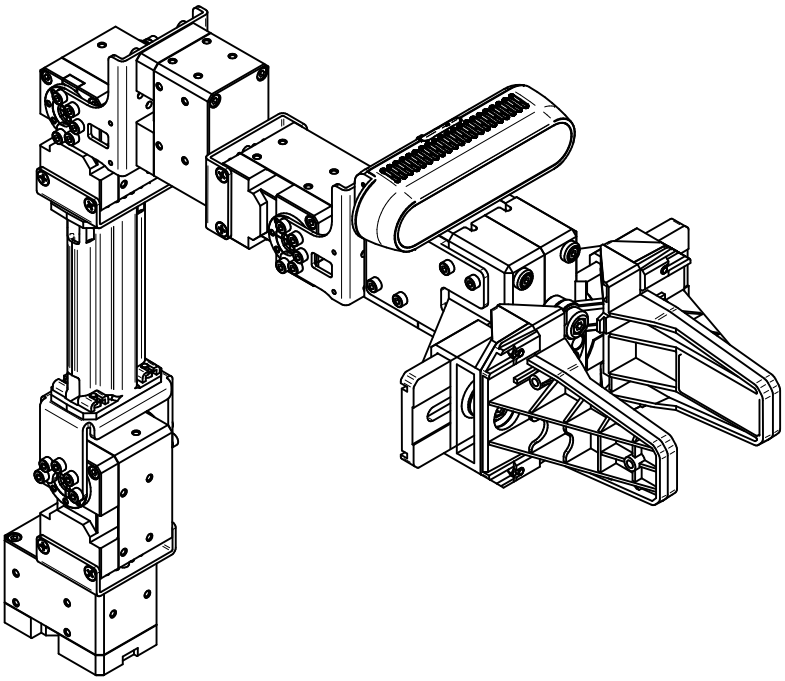

Abbildung 2.5.: 6DOF OpenManipulator mit Kamera

2.3.1. Spezifikationen

2.3.1.1. Maße des OM6

Da der OM6 keine offiziell von *ROBOTIS* unterstützte Version ist, sind dazu auch keine CAD-Dateien oder technische Zeichnungen verfügbar. Um die Roboterkinematiken in Abschnitt 2.3.2 herleiten zu können, sind Informationen zu den Maßen des OM6 jedoch unverzichtbar. Auch für die spätere Rücktransformationen von Bild- in Roboterkoordinaten sind diese Informationen wichtig, besonders die Position der Kamera. Daher wurde der OM6 nachträglich im Rahmen der Arbeit über ein CAD-Tool konstruiert. Somit stehen die Maße des OM6 für die kommenden Berechnungen zur Verfügung (siehe Abbildung 2.6). Alle CAD-Dateien sind im beigefügtem Medium hinterlegt.

2.3.1.2. Dynamixel XM430-Servomotoren

Der OpenManipulator wird standardmäßig mit den *Dynamixel XM430-W350-T/R* angetrieben. All seine mechanischen Elemente sind speziell für diese Servomotoren konstruiert worden. Für den OM6 werden drei *Dynamixel XM430-W210-T* und weitere drei *Dynamixel XM430-W350-T* verwendet. Die beiden Modelle sind baugleich und unterscheiden sich nur hinsichtlich ihrer Leistung. Für die ersten drei Gelenke des OM6 werden die leistungsstarken *XM430-W350-T* Servomotoren verwendet, da diese stärker belastet werden und hauptsächlich für die Positionierung des Endeffektors zuständig sind. Die Gelenke vier bis sechs sind nur für die Orientierung des Endeffektors zuständig, weshalb auch die leistungsschwächeren *XM430-W210-T* Servomotoren ausreichen.

Insgesamt zeichnet sich die XM430-Serie durch seine Benutzerfreundlichkeit und der hochwertigen Verarbeitung aus. Innerhalb des Metallgehäuses befindet sich das Stahlgetriebe, die Steuerungsplatine samt Mikrocontroller und Regler und sonstige Elektronik. Der Antriebsflansch ist gerillt und bietet zudem mehrere Anschlussbohrungen. Am Gehäuse sind weitere Anschlussbohrungen mit Gewinde zu finden. Außerdem besitzt die XM430-Serie zwei TTL-Schnittstellen auf der Rückseite. Über eine davon kann der Servomotor versorgt und gesteuert werden. Die andere kann dazu genutzt werden, um weitere XM430-Servomotoren über das *Daisy Chaining*-Verfahren seriell miteinander zu verbinden. Somit können die sieben Servomotoren des OM6 seriell gekoppelt und letztendlich über eine TTL-Schnittstelle

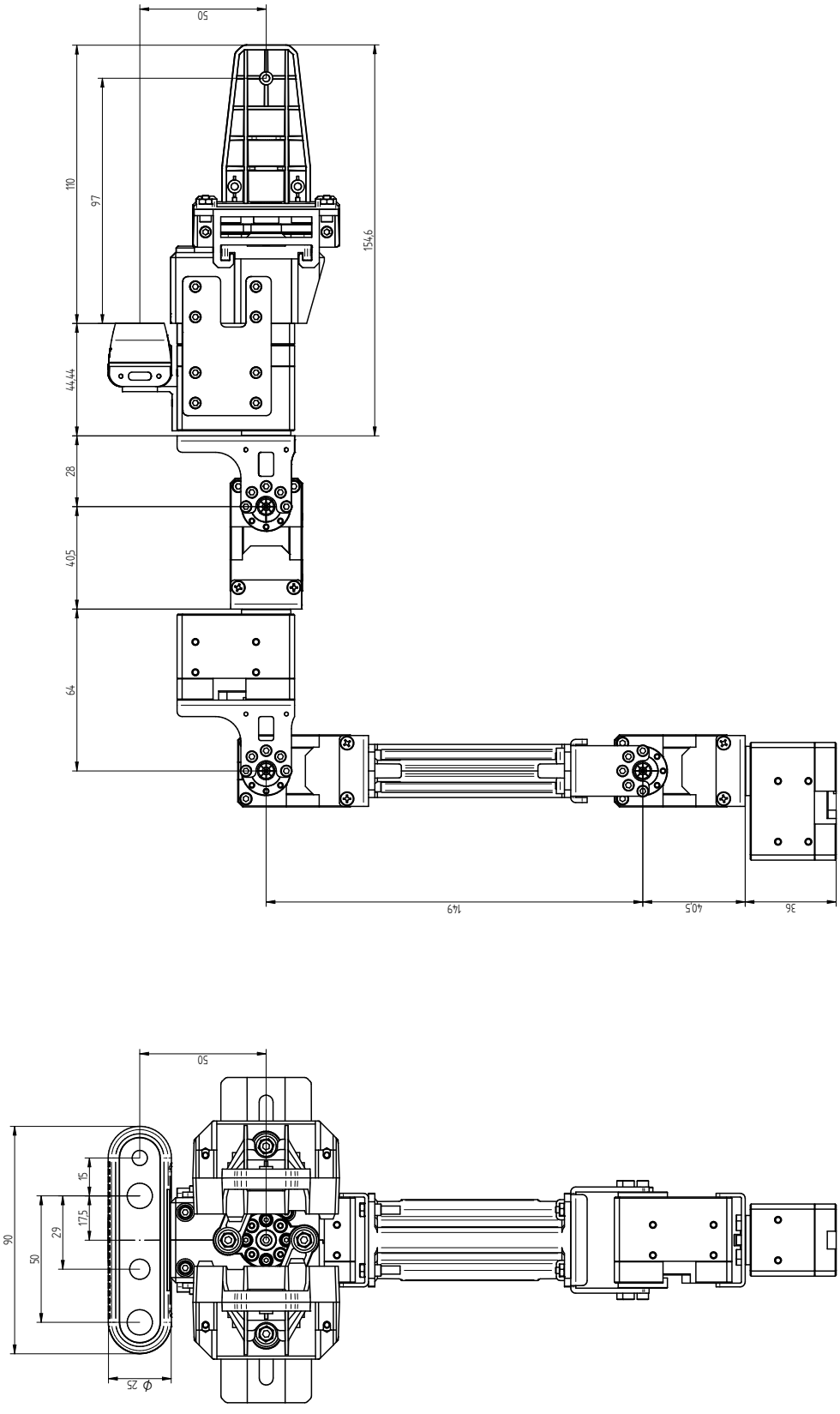

Abbildung 2.6.: Maßangaben des OM6 mit Kamera

gesteuert werden. Für die Ansteuerung wird die *Dynamixel-Workbench* Softwarebibliothek zur Verfügung gestellt. Darin sind bereits alle nötigen Steuer- und Überwachungsfunktionen implementiert. Dadurch muss der Benutzer sich nicht mit den Treibern auseinandersetzen. Im Anhang A.2 sind diverse Spezifikationen zum Dynamixel XM430 zu finden. Genaueres ist auf der Internetseite des Herstellers ([17], [16]) zu finden.

2.3.2. Roboterkinematik

Robotermanipulatoren werden heutzutage in vielfältigen Anwendungen eingesetzt. Dies ist vor allem dank derer flexiblen Kinematik möglich. Um also Problemstellungen mit einem Robotermanipulator lösen zu können, ist ein klares Verständnis der Kinematik des eingesetzten Roboters unvermeidbar. Deshalb wird im Folgendem die Kinematik des OM6 analysiert. In Abbildung 2.7 sind die Achsen des OM6 und die Koordinatentransformationen abstrahiert dargestellt. Die Drehachsen der Gelenke liegen auf den Z-Achsen der Koordinatensysteme.

Der OM6 befindet sich in Abbildung 2.7 in der Kanonenstellung. Wichtig ist hierbei, dass die Robotersteuerung aus Abschnitt 2.3.4 von einer ausgestreckten Ausgangsstellung ausgeht. Der einzige Unterschied bzgl. der Kanonenstellung besteht darin, dass die dritte Achse um 90° in positiver Z_2-Richtung gedreht ist. Die entsprechende DH-Matrix bezieht sich auf Abbildung 2.7 und ist in der Tabelle 2.3 angegeben. Die Roboterkonfiguration ist in der URDF-Datei des ROS-Paketes der OM6-Steuerung definiert[1]. Zur Steuerung des Roboters wird in dieser Arbeit das OM6-Steuerungspaket für ROS verwendet. Dieses wird vom Hersteller *ROBOTIS* zur Verfügung gestellt und nutzt die *MoveIt!*-Funktionen über ROS. Dafür greift *MoveIt!* auf die entsprechende URDF-Konfigurationsdatei des Roboters zu und generiert daraus die direkte und indirekte Kinematik. Ausführliche Informationen zur URDF-Konfigurationsdatei und *MoveIt!* sind unter [21, Kapitel 13].

[1] URDF-Konfigurationsdatei: `catkin_ws/src/open_manipulator_friends/...`
`open_manipulator_6dof_description/urdf/open_manipulator_6dof.urdf.xacro`

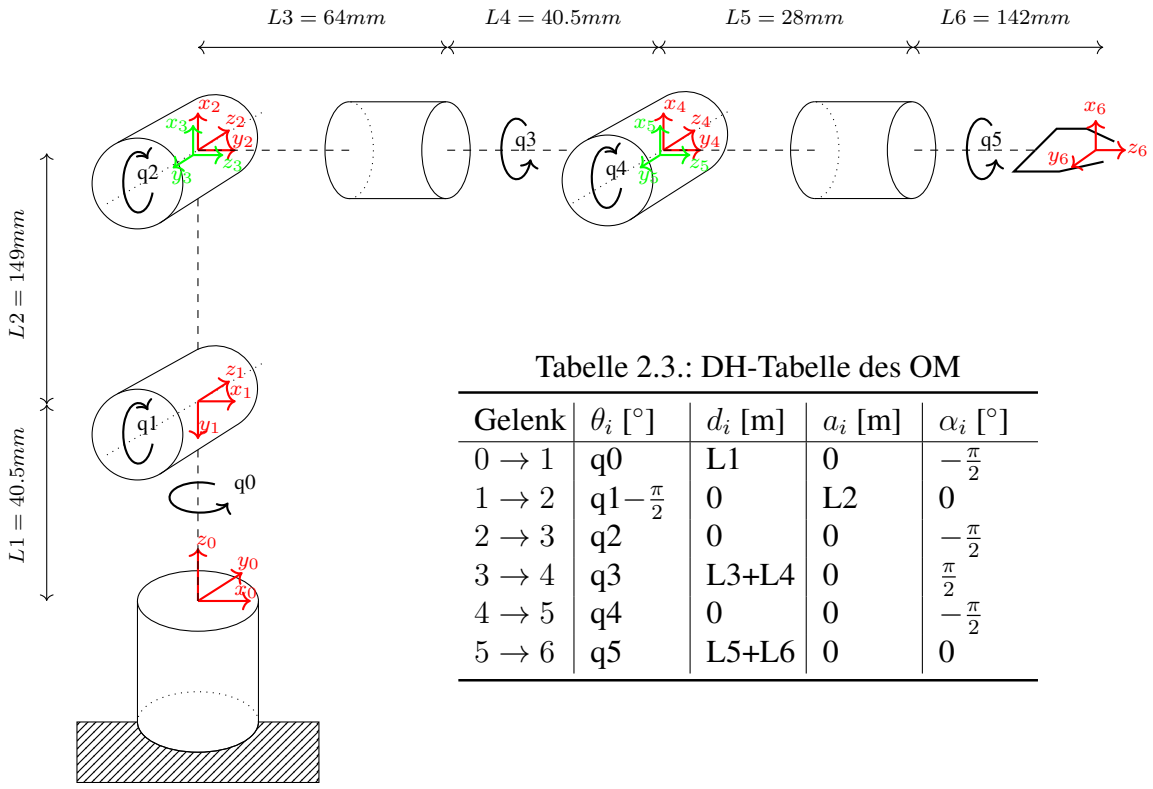

Tabelle 2.3.: DH-Tabelle des OM

Gelenk	θ_i [°]	d_i [m]	a_i [m]	α_i [°]
$0 \rightarrow 1$	q0	L1	0	$-\frac{\pi}{2}$
$1 \rightarrow 2$	q1$-\frac{\pi}{2}$	0	L2	0
$2 \rightarrow 3$	q2	0	0	$-\frac{\pi}{2}$
$3 \rightarrow 4$	q3	L3+L4	0	$\frac{\pi}{2}$
$4 \rightarrow 5$	q4	0	0	$-\frac{\pi}{2}$
$5 \rightarrow 6$	q5	L5+L6	0	0

Abbildung 2.7.: Kinematisches Modell des OM6

2.3.3. Konfiguration

Um den OM6 steuern zu können, müssen zunächst dessen Dynamixel XM430 Servomotoren konfiguriert werden. Für die Konfiguration kann der *RoboPlus Manager 2.0* [2] von *ROBOTIS* genutzt werden. Dieser bietet eine Vielzahl an Einstellunmöglichkeiten. Um die Dynamixel Servomotoren konfigurieren zu können, müssen diese zuerst in Betrieb genommen werden. Hierfür wird als Schnittstelle zum PC der sog. *U2D2-Adapter* verwendet. Dafür bietet der U2D2-Adapter jeweils einen *4-Pin RS-485-*, einen *3-Pin TTL-Level-* und einen *4-Pin UART*-Anschluss. An diesen Steckplätzen können Dynamixel Servomotoren mit unterschiedlichen Anschlüssen verbunden werden. Für die Kommunikation zum PC wird ein *Micro-B USB*-Anschluss verwendet. Für die Stromversorgung der Dynamixel kann das *SMPS2Dynamixel* Versorgungsmodul genutzt werden. Ein beispielhafter Aufbau ist in Abbildung 2.9 dargestellt.

[2] http://emanual.robotis.com/docs/en/software/rplus2/manager/

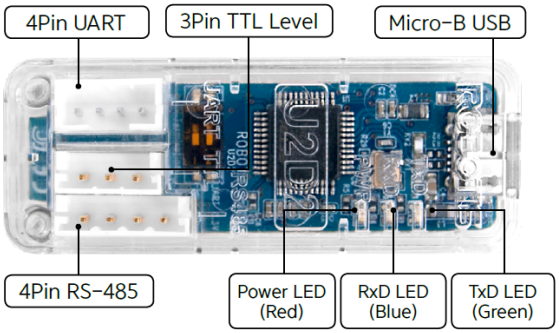

Abbildung 2.8.: U2D2-Adapter [15]

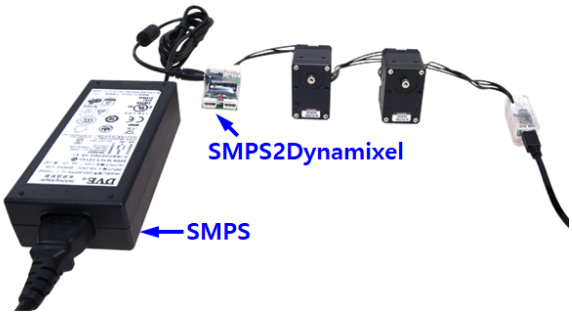

Abbildung 2.9.: SMPS2Dynamixel [15]

Bei der Konfiguration der Dynamixel Servomotoren des OM6 sind eigentlich nur die Baudrate und die Servo-ID's relevant. Diese lassen sich beide über den *RoboPlus Manager 2.0* einstellen. Die Baudrate kann frei gewählt werden, sollte jedoch mit angegebenen Baurate des OM6 identisch sein. Diese ist in der *Launch-Datei* [3] des OM6 definiert. Die Servo-ID's werden von 11 bis 17 vergeben, wobei der erste Dynamixel an der Roboterbasis die Servo-ID 11 zugewiesen bekommt. Die restlichen bekommen in aufsteigender Reihenfolge die Servo-ID's bis 17 zugewiesen, sodass der Robotergreifer letztlich die Servo-ID 17 erhält. Diese Einstellungen sind unbedingt zu berücksichtigen, da der OM6 sonst nicht steuerbar ist.

2.3.4. Inbetriebnahme

Dieser Abschnitt beschreibt die wichtigsten Aspekte der Inbetriebnahme des OM6. Die folgenden Informationen können ebenfalls auf der Herstellerseite nachgelesen werden (siehe [13]). Um den OM6 in ROS bedienen zu können, sind zunächst folgende ROS-Pakete notwendig:

```
$ sudo apt-get install ros-kinetic-ros-controllers ...
ros-kinetic-gazebo* ros-kinetic-moveit* ros-kinetic-industrial-core
```

Zudem müssen die folgenden ROS-Pakete ins `catkin`-Verzeichnis geladen und erstellt werden:

[3] Verzeichnis: catkin_ws/src/open_manipulator_and_friends/open_manipulator_6dof_controller/launch/open_manipulator_6dof_controller.launch

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/DynamixelSDK.git
$ git clone https://github.com/ROBOTIS-GIT/dynamixel-workbench.git
$ git clone https://github.com/ROBOTIS-GIT/dynamixel-workbench-msgs.git
$ git clone https://github.com/ROBOTIS-GIT/open_manipulator_msgs.git
$ git clone https://github.com/zang09/open_manipulator_friends.git
$ git clone https://github.com/zang09/open_manipulator_6dof_simulations.git
$ git clone https://github.com/zang09/open_manipulator_6dof_application.git
$ git clone https://github.com/ROBOTIS-GIT/robotis_manipulator.git
$ cd ~/catkin_ws && catkin_make
```

In Kapitel 4 werden Teile der genannten ROS-Pakete noch entsprechend der Aufgabenstellung modifiziert. Diese sind auf dem beigefügten Medium enthalten.
Der OM6 lässt sich über den folgenden Befehl starten:

```
$ roslaunch open_manipulator_6dof_controller ...
open_manipulator_6dof_controller.launch
```

Außerdem liefert *ROBOTIS* für den OM6 eine Steuerungssoftware mit GUI. Dadurch kann z. B. geprüft werden, ob die Inbetriebnahme des OM6 erfolgreich war. Die Steuerungssoftware kann über folgenden Befehl gestartet werden:

```
$ roslaunch open_manipulator_6dof_control_gui ...
open_manipulator_6dof_control_gui.launch
```

2.4. RGB-D Kamera: Intel RealSense D435

2.4.1. Funktionsprinzip

Zum Greifen von Objekten muss der Roboter erkennen könnnen, was er greifen soll und insbesondere an welcher Stelle des Objektes. Für die Objekterkennung wird in dieser Arbeit die *Intel RealSense D435* Tiefenbildkamera verwendet. Das Gegenstück dieser Baureihe ist der *Intel RealSense D415*. Beide Tiefenbildkameras können sich großer Beliebtheit erfreuen, sowohl bei Hobby-Bastlern, als auch bei professionellen Entwicklern. Vor allem zeichnen

sich diese Kameras durch ihr geringes Gewicht (72g) und der kleinen Baugröße aus. Ein weiterer, für mobile Anwendungen im Bereich der Robotik nicht unerheblicher Vorteil ist, dass diese Kameras über eine 5V-Spannungsversorgung betrieben werden können. Für die Versorgung und den Datentransfer reicht daher eine USB 3.0 Verbindung aus. Neben den hardwaretechnischen Vorteilen bietet Intel auch umfangreiche Programmierschnittstellen an und unterstützt eine Vielzahl von Plattformen, unter anderem auch ROS. Dies sind die ausschlaggebenden Kriterien, weshalb die *RealSense D435* in dieser Arbeit Einsatz findet. Im Folgenden wird die Funktionsweise, die Kamerakalibrierung und das Einbinden der *RealSense* in die ROS-Umgebung behandelt.

Die Aufnahme von Tiefenbildern basiert auf dem stereoskopischen Prinzip. Dafür besitzt die RealSense zwei horizontal versetzte Kameramodule. Die Qualität der stereoskopischen Tiefenbilder hängt stark von der Oberflächentextur der aufgenommenen Objekte ab. Monochrome Oberflächen liefern i.d.R. schlechtere Tiefenaufnahmen als texturreiche. Um die Qualität zu verbessern wird ein zusätzlicher Infrarotprojektor verwendet. Dieser projiziert ein statisches Infrarotmuster auf die Umgebung, wodurch sich die Anzahl der vorhandenen Texturen erhöht und damit eine bessere Schätzung der Tiefe erzeugt wird. Außerdem wird die Tiefe zwischen den projizierten Infrarotpunkten geschätzt und somit die Auflösung des Tiefenbildes künstlich erhöht. Dieses Prinzip wird in den Abbildungen 2.10 und 2.11 dargestellt.

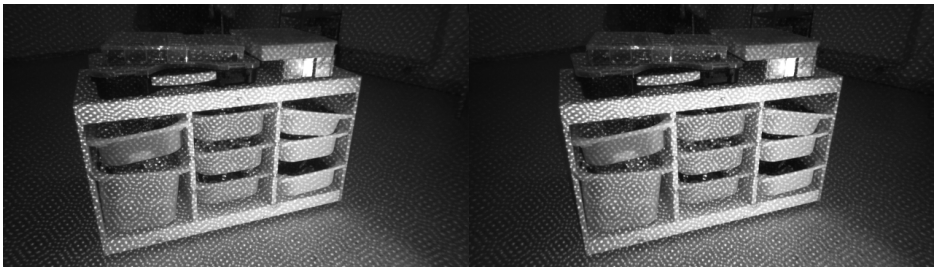

Abbildung 2.10.: Bildaufnahme der rechten und linken stereoskopischen Kameramodule mit projiziertem Infrarotmuster [5]

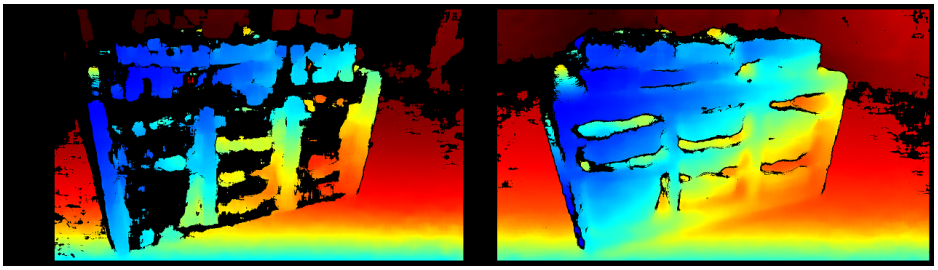

Abbildung 2.11.: Bildaufnahme der rechten und linken stereoskopischen Kameramodule mit projiziertem Infrarotmuster [5]

2.4.2. Kamerakalibrierung

Bei der dynamischen Kamerakalibrierung werden die extrinsischen Parameter optimiert. Diese ermöglichen die Transformation zwischen Bild- und kartesischen Raumkoordinaten. Die intrinsischen Kameraparameter werden bei der dynamischen Kamerakalibrierung nicht angepasst. Grundsätzlich werden die *RealSense*-Kameras vor der Auslieferung vom Werk aus kalibriert. Für den Fall, dass dennoch eine Kalibrierung vom Benutzer gewünscht werden sollte, stellt *Intel* die entsprechende Kalibrierungssoftware[4] zur Verfügung. Mit Hilfe dieser und eines entsprechenden Kalibrierungstargets können die extrinsischen Parameter optimiert werden. Zur Überprüfung der Kalibrierungsqualität wird ebenfalls ein entsprechendes Programm zur Verfügung gestellt. Die intrinsischen und extrinsischen Parameter können vom folgenden ROS-Topic abgerufen werden:

`/camera/depth/camera_info`

Der Rückgabewert des ROS-Topics hat den ROS-Message-Type `sensor_msgs/CameraInfo`[5].

[4] *RealSense Calibration Tool:* https://downloadcenter.intel.com/download/28517/Intel-RealSense-D400-Series-Calibration-Tools-and-API

[5] *CameraInfo* Definition: http://docs.ros.org/kinetic/api/sensor_msgs/html/msg/CameraInfo.html

2.4.3. RealSense-ROS

Intel liefert als Programmierschnittstelle für seine *RealSense*-Kameraserie die sog. *Intel RealSense SDK*. Darin ist außerdem ein entsprechender Wrapper für ROS[6] enthalten. Dies ermöglicht die problemlose Einbindung der *RealSense D435* in die ROS-Umgebung. Die *RealSense*-Kamera kann über die folgenden Befehle am Arbeitsrechner in Betrieb genommen werden:

```
$ sudo apt-key adv --keyserver keys.gnupg.net --recv-key      ...
C8B3A55A6F3EFCDE || sudo apt-key adv --keyserver hkp://       ...
keyserver.ubuntu.com:80 --recv-key C8B3A55A6F3EFCDE
$ sudo add-apt-repository "deb http://realsense-hw-public.    ...
s3.amazonaws.com/Debian/apt-repo xenial main" -u
$ sudo apt-get install librealsense2-dev librealsense2-utils ...
ros-kinetic-rgbd-launch
$ cd ~/catkin_ws/src
$ git clone https://github.com/intel-ros/realsense.git
$ cd ~/catkin_ws && catkin_make
```

Die Inbetriebnahme der *RealSense*-Kamera über den RPI gestaltet sich jedoch als schwieriger. Dieser basiert auf einer ARM-Architektur, weshalb viele Softwarekomponenten nicht unterstützt werden. Dies gilt auch für die *RealSense SDK* und dessen ROS-Wrapper. Daher müssen alle benötigten Softwarebibliotheken mit CMAKE manuell erstellt und installiert werden. Im Anhang A.3 ist die Inbetriebnahme der *RealSense* auf dem RPI beschrieben. Das ROS-RealSense-Node lässt sich unabhängig davon sowohl auf dem Arbeitsrechner, als auch auf dem RPI über den folgenden Befehl starten:

```
roslaunch realsense2_camera rs_camera.launch
```

Über diese ROS-Launch-Datei wird die *RealSense*-Kamera gestartet. Darin sind auch sämtliche Konfigurationen bzgl. des Kamerabetriebs enthalten. Alle einstellbaren Parameter[7] sind auf der Herstellerseite zu finden. Der Inhalt der Launch-Datei ist im Anhang A.4 hinterlegt.

[6] ROS Wrapper for Intel RealSense Devices: https://github.com/IntelRealSense/realsense-ros

[7] *RealSense-ROS*: https://github.com/IntelRealSense/realsense-ros

2.5. Turtlebot 3 Waffle Pi mit 6DOF Open Manipulator

Beim Verheiraten des TB3's mit dem OM6 sind einige Randbedingungen zu beachten, vor allem sind aber die Positionierung des OM6 und die minimale Detektionsdistanz der *RealSense*-Kamera wichtig. Für eine zuverlässige Objekterkennung muss die Kamera mindestens $20cm$ vom Objekt entfernt sein. Um diesen Mindestabstand einhalten zu können, wird zwischen dem TB3 und dem OM6 noch eine weitere Zwischenebene eingebaut. Dafür werden Standardteile des TB3's verwendet. Dadurch befindet sich der OM6 auf einer zulässigen Höhe und die am Robotermanipulator montierte Kamera kann Objekte zuverlässig erkennen. Eine abstrahierte Darstellung ist in Abbildung 2.12 zu sehen.

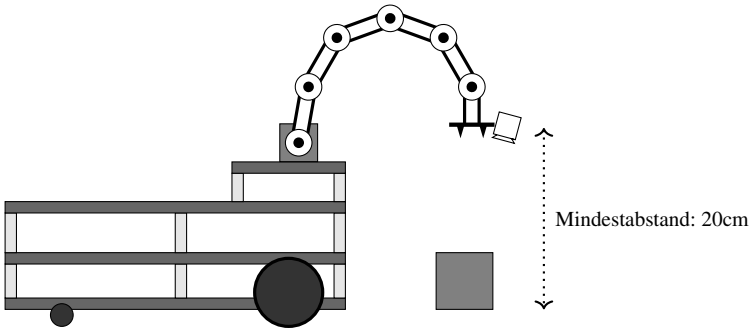

Abbildung 2.12.: Vereinfachte Darstellung des Roboters beim Greifen

Der elektrische Zusammenschluss der Hardwarekomponenten ist neben dem mechanischen Aufbau ebenfalls bedeutend. Diese ist vereinfacht in Abbildung 2.13 dargestellt. Insgesamt kann der Roboter in zwei größere Strukturen zusammengefasst werden. Zum einen in die Hardwarekomponenten des TB3's, zum anderen in die des OM6. Dies ist nicht als strenge Unterteilung in autonome Systeme aufzufassen, da einige Hardwarekomponenten sowohl mit dem TB3, als auch mit dem OM6 interagieren. Ein Beispiel hierfür ist z.B. der RPI, welcher im Allgemeinen als Schnittstelle zwischen TB3 und OM6 dient. Die Unterteilung soll nur der Übersicht dienlich sein.

Zentrale Hardwarekomponenten des TB3 sind das OpenCR-Board und der RPI 3B+. Das OpenCR-Board dient der Steuerung, Versorgung und Überwachung des TB3. Es wird durch einem 3S-Lithium-Polymer-Akkumulator mit insgesamt $11.1V$ versorgt. Über das OpenCR-Board werden die Antriebsräder (Modell des Servomotors: Dynamixel XM430-210T) über entsprechende TTL-Anschlüsse versorgt und gesteuert. Außerdem versorgt das

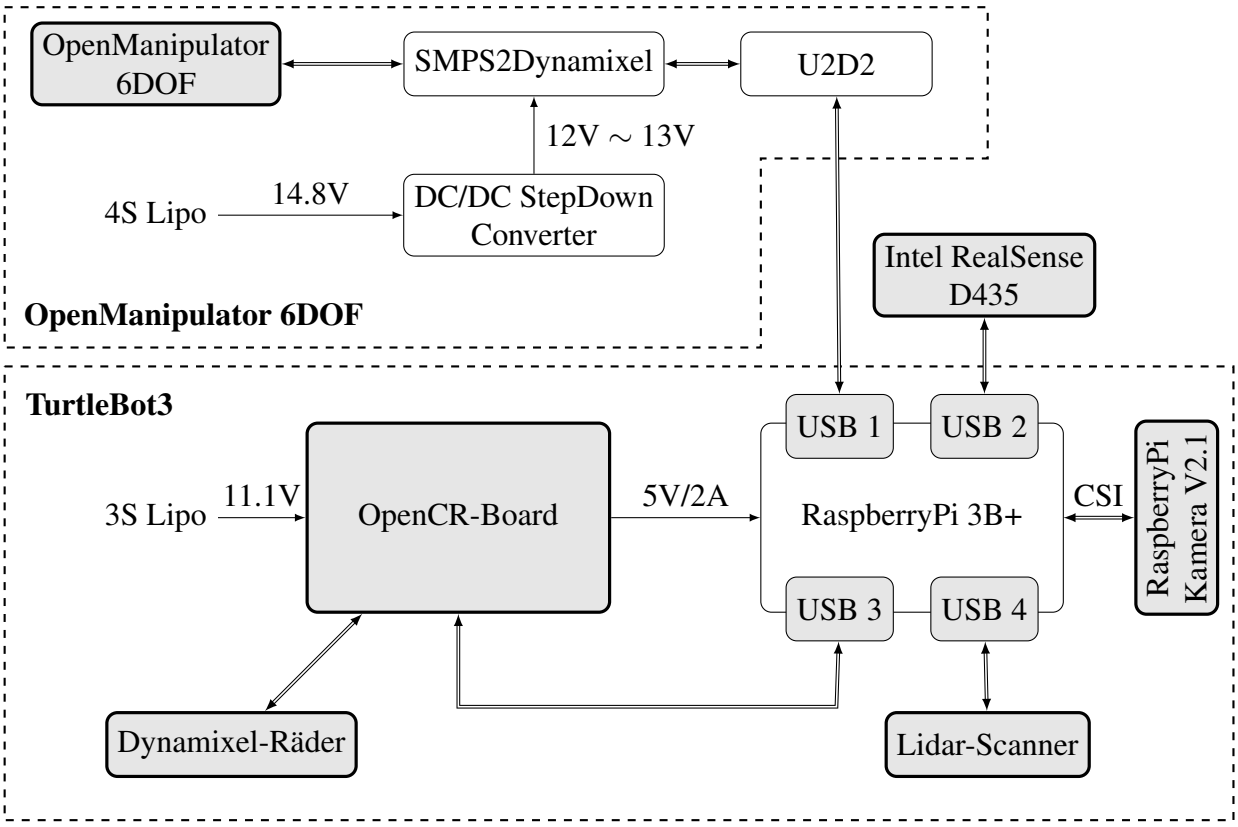

Abbildung 2.13.: Hardwarearchitektur des TB3 mit OM6

OpenCR-Board den RPI über dessen GPIO-Pins. Die Kommunikation zwischen OpenCR-Board und dem RPI findet über eine USB-Verbindung statt. Hierbei werden von beiden Modulen Informationen gesendet und empfangen.

Der RPI ist die Schlüsselkomponente des Roboters und verbindet alle Komponenten miteinander. Dafür werden vor allem die vier USB 2.0-Anschlüsse genutzt. Eine angeschlossene Komponente ist hierbei der Lidar-Scanner. Dieser wird mit dem TB3 mitgeliefert und über ein entsprechend ROS-Node gesteuert und ausgelesen. Genaueres unter der Homepage des Herstellers (siehe [14]). Die *RealSense*-Kamera wird ebenfalls an den USB-Anschluss des RPI angeschlossen. Offiziell muss diese über einen USB 3.0-Anschluss betrieben werden. Durch die geringere Datenrate des USB 2.0-Anschlusses sind diverse Funktionen der *RealSense*-Kamera nur eingeschränkt nutzbar. Auch geringere Framerates sind damit verbunden. Eine Anleitung wie die *RealSense*-Kamera über den RPI in Betrieb genommen werden kann, ist im Anhang A.3 zu finden.

Der OM6 wird ebenfalls über den RPI gesteuert, ist aber von der elektronischen Versorgung unabhängig vom OpenCR-Board. Der Hersteller *ROBOTIS* bietet zwar eine Variante des TB3s mit OM3 und entsprechender Software zum Bespielen des OpenCR-Boards, jedoch ist diese Lösung nicht empfehlenswert. Dieser Lösungsansatz sieht es vor, dass der TB3 und der OM3 über ein gemeinsames OpenCR-Board versorgt und gesteuert werden. Das OpenCR-Board bietet dafür genügend TTL-Anschlüsse. Jedoch hat sich während der Entwicklung des Roboters gezeigt, dass das OpenCR-Board noch relativ große Schwächen bei der elektrischen Versorgung des OpenManipulators aufweist. Aus diesem Grund wird der OM6 zwar noch über den RPI gesteuert, für die Versorgung werden jedoch andere Komponenten verwendet. Dafür wird das *SMPS2Dynamixel*-Modul (siehe Abbildung 2.9) verwendet. Dies ermöglicht die elektrische Versorgung des OM6 über einen SMPS-Anschluss. Hierfür wird ein 4s-Lithium-Polymer-Akkumulator mit $14.8V$ Nennspannung verwendet. Da die Spannung des 4S-Lipo-Akkus bei voller Ladung bei $16.8V$ liegt und somit die maximale Grenzspannung der OM6-Servomotoren (Dynamixel XM430 mit ca. $14,8V$ maximaler Grenzspannung) überschreitet, ist eine Anpassung nötig. Hierfür wird ein *DC/DC-StepDown Converter* verwendet und die Spannung somit auf ca. $12.5V$ vermindert. Als Versorgungsschnittstelle des OM6 dient das *SMPS2Dynamixel*-Modul (siehe Abbildung 2.9).

Der TB3 und der OM6 werden in dieser Arbeit als gänzlich autarke Systeme gesteuert. Die Kinematiken der beiden Systeme sind nicht miteinander gekoppelt. Eine Relativsteuerung vom OM6 zur Roboterbasis der TB3 ist z. B. nicht möglich. Des weiteren ist die Inbetriebnahme des Roboters mit CNN gesteuertem Greifen von Objekten in Kapitel 4 beschrieben. In Abbildung 2.14 ist der zusammengebaute Roboter mit allen Komponenten abgebildet.

Abbildung 2.14.: TB3 mit OM6 und RealSense-Kamera

3. Greifposengenerierung mittels CNN

3.1. Einleitung

In der Industrie ist es üblich, dass Industrieroboter nur innerhalb ihrer gesicherten Arbeitszelle agieren. Die Arbeitszelle kann physisch (z. B. über Schutzzäune) oder über Sensorik gesichert werden. Mittlerweile werden auch vermehrt *Cobots* eingesetzt. Das sind Roboter, welche vorrangig bei der Mensch-Roboter-Kollaboration (kurz *MRK*) zum Einsatz kommen und i.d.R keine zusätzlichen Schutzeinrichtungen benötigen. Sie besitzen diverse Sensoren und können Kollisionen detektieren und somit eventuelle Schäden an Personen und Umgebung verhindern. Auch mobile Roboter, die sich z. B. innerhalb von dynamischen Lagerhallen frei und ohne zusätzliche Schutzmechanismen bewegen können, sind mittlerweile keine Neuheit mehr. Die Tendenz geht daher eindeutig in Richtung von freien und interaktiven Robotern. Damit steigt natürlich auch der Anspruch an moderne Robotikanwendungen. Ein Anwendungsfall ist unter anderem auch die Interaktion mit Unbekanntem, wie in diesem Fall das Greifen von unbekannten Objekten.

In der Robotertechnik ist das Greifen von unbekannten und unmarkierten Objekten ein etabliertes Forschungsthema an dem seit Jahren gearbeitet wird. Entsprechend wurden über die Zeit diverse Verfahren entwickelt und veröffentlicht. Eine typische Problemstellung ist der sog. *"Griff in die Kiste"*. Hierbei geht es darum, dass unbekannte Objekte mithilfe eines Robotermanipulators aus einer unsortierten Kiste entnommen werden sollen. Ein aktuelles Beispiel eines solchen Anwendungsfalls ist das *Dex-Net*-Projekt [9]. Die Verfahren zum Lösen dieser Problemstellung lassen sich in insgesamt zwei Kategorien einteilen [11, vgl. S.2]:

- Analytische Verfahren:

  Hierbei werden mathematische Methoden zur Analyse von gegebenen Geometrien und der Bestimmung von geeigneten Greifposen verwendet. Die detektierbaren Formen werden hierbei im Vorfeld definiert. Jede zu erkennende Form muss mathematisch Beschrieben und einprogrammiert werden. Unbekannte Formen bzw. Objekte können mit diesem Vorgehen nicht erkannt werden.

- Empirische Verfahren:

  Geometrien in der Umgebung des Roboters werden mit bekannten Formen und Objektmodellen verglichen, um so potentielle Greifposen zu detektieren. Bei den bekannten Formen und Modellen handelt es sich oft um 3D-Modelle oder Punktwolken. Diese werden üblicherweise in einer sehr großen Datenbanken hinterlegt. Der Vergleich zwischen detektierten Objekten und abgespeicherten Modellen, ist sehr aufwändig und benötigt lange Rechenzeiten. Jedoch können empirische Methoden prinzipiell auch auf unbekannte, nicht fest einprogrammierte Geometrien verallgemeinert werden.

Auch CNN's gehören zu den empirischen Verfahren. Zwar gleichen diese die gegebene Geometrie nicht mit einer Datenbank ab, werden aber mit einer geeigneten Datenbank trainiert. Die antrainierten Parameter des CNN's basieren letztendlich auf den Informationen einer Modell-, Punktwolken-, Bild- oder sonstigen Datenbank. Kameraframes können direkt an das CNN übergeben werden, welches darin nach bekannten Merkmalen sucht. Ein Abgleich mit einer Datenbank ist i. d. R. nicht notwendig. Dadurch verkürzt sich die Rechenzeit enorm. Außerdem ist die Merkmalserkennung mit neuronalen Netzen verallgemeinbar und kann somit auch auf unbekannte Objekte übertragen werden.

Dieses Kapitel behandelt die theoretischen Ansätze der Objekterkennung mittels eines CNN's und die daraus resultierende Greifposengenerierung. Zunächst wird der theoretische Algorithmus und das Vorgehen bei der Greifposengenerierung erläutert. Anschließend wird auf dieser Grundlage das CNN trainiert. Dabei werden Struktur und Inhalt der Trainingsdatenbank analysiert. Danach wird der Trainingsvorgang selbst betrachtet.

Als Grundlage dieses Kapitels dient die Veröffentlichung von *Morrison, Corke & Leitner* [11]. Das Thema dieser Veröffentlichung ist die Entwicklung eines neuronales Faltungsnetzwerks mit dem Namen GG-CNN, welches Greifposen an Objekten detektieren kann

und anschließend von einem Robotermanipulator greifen lässt. Dies ist auch die zentrale Fragestellung dieser Arbeit. Im Vergleich zu *Morrison et al.* werden jedoch gänzlich andere Komponenten und Roboter genutzt. Das Ziel ist es deshalb, das Konzept aus der Veröffentlichung in diese Arbeit zu integrieren und mit Aspekten der mobilen Robotik zu ergänzen.

3.2. Theoretische Grundlagen der Greifposengenerierung

In diesem Abschnitt werden die analytischen Grundlagen und die Funktionsweise des Algorithmus zum Greifen von Objekten behandelt. In Abbildung 3.1 ist der funktionale Ablauf der Greifposengenerierung schematisch dargestellt. Zunächst wird ein Tiefenbild vom Objekt erzeugt. Dafür wird eine am Handgelenk des Robotermanipulators befindliche Tiefenbildkamera verwendet. Anschließend wird das Tiefenbild an das CNN übergeben, welches die Tiefeninformationen auswertet und insgesamt drei Matrizen ausgibt: die Qualitäts-, die Winkel- und die Weitenmatrix.

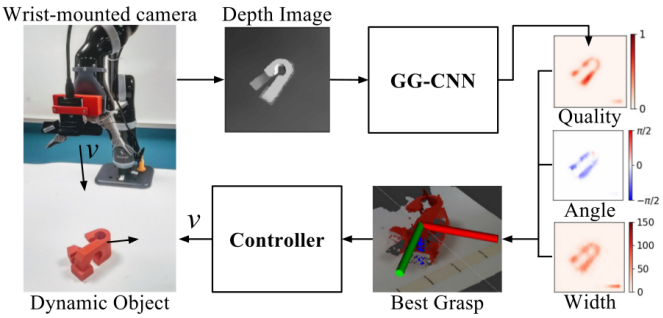

Abbildung 3.1.: Schematischer Ablauf der Greifposendetektion [11]

Die Qualitätsmatrix gibt für jeden Pixel des Tiefenbildes ein skalares Qualitätsmaß bzgl. des Greiferfolges an. Wenn eine potentielle Greifmöglichkeit an der Stelle eines Pixels detektiert wird, bekommt dieses Pixel vom CNN einen Wert zwischen 0 und 1 zugewiesen. Je wahrscheinlicher ein erfolgreicher Greifversuch scheint, desto höher der Wert. Anschließend wird die Qualitätsmatrix mit einem Gauß-Filter geglättet um so die Potentialspitzen herauszufiltern und die beste Greifposition eindeutig zu ermitteln. Anhand der Bildkoordinate dieses Pixels können dann in der Winkel- und der Weitenmatrix die entsprechenden Werte an der gleichen Bildposition ausgelesen werden.
Bei der Winkelmatrix weist das CNN jedem Pixel, bei dem potentielle Greifmöglichkeiten detektiert wurden, einen skalaren Greifwinkel (Einzustellender Winkel des Greifers um die aufrechte Z-Achse) zwischen $\frac{\pi}{2}$ und $-\frac{\pi}{2}$ zu. Analoges Prinzip gilt auch für die Weitenmatrix, nur dass hier die Pixel mit potentiellen Greifmöglichkeiten, Greifweiten statt Greifwinkel

zugewiesen bekommen. Die Greifweite ist abhängig vom Objekt selbst bzw. der Stelle an der gegriffen werden soll. Aus diesen drei generierten Matrizen wird anschließend die beste Greifpose generiert. Dieser Vorgang wird im Folgenden genauer analysiert und mathematisch definiert. Zudem sind alle relevanten Größen in Abbildung 3.2 grafisch dargestellt.

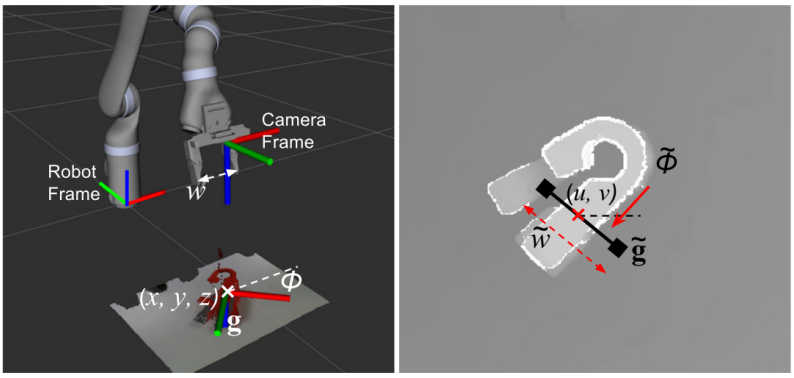

Abbildung 3.2.: Abstrakte Größen der Greifposendetektion [11]

Die Greifpose $\boldsymbol{g}$ ist als abstrakter Punkt im Raum bzw. am Objekt selbst aufzufassen und beinhaltet insgesamt vier Werte:

$$\boldsymbol{g} = (\boldsymbol{p}, \phi, w, q) \tag{3.1}$$

1. Die Greifposition $\boldsymbol{p} = (x, y, z)$ in kartesischen Koordinaten im Bezug auf das Basiskoordinatensystem des Robotermanipulators.
2. Den Drehwinkel ϕ des Greifers um die aufrechte Z-Achse des Roboterkoordinatensystems an der Position $\boldsymbol{p}$.
3. Die erforderliche Greifweite w (in m oder cm) an der Position $\boldsymbol{p}$.
4. Das skalares Qualitätsmaß q zur Beschreibung des potentiellen Greiferfolgs. Wenn eine potentielle Greifmöglichkeit besteht, bekommt q für die Position $\boldsymbol{p}$ einen Wert von 0 bis 1 zugewiesen.

Zur Bestimmung der Greifpose g wird dem CNN ein Tiefenbild $\boldsymbol{I} = \mathbb{R}^{HxB}$ mit der Höhe H und der Breite B übergeben. Im Tiefenbild $\boldsymbol{I}$ werden vom CNN dann potentielle Greifmöglichkeiten pro Pixel detektiert. Die Greifposen werden wie folgt in Bildkoordinaten

definiert:

$$\tilde{\boldsymbol{g}} = (\boldsymbol{s}, \tilde{\phi}, \tilde{w}, q) \tag{3.2}$$

Ähnlich wie Gleichung 3.1, besteht die Greispose $\tilde{g}$ auch aus insgesamt vier Werten:

1. Die Position des Pixels $\boldsymbol{s} = (u, v)$ in Bildkoordinaten.
2. Der Greifwinkel $\tilde{\phi}$ in Bezug auf das Kamerakoordinatensystem am Pixel $\boldsymbol{s}$.
3. Die erforderliche Greifweite $\tilde{w}$ (in Bildkoordinaten) am Pixel $\boldsymbol{s}$.
4. Das binäre Qualitätsmaß q zur Beschreibung des Greiferfolges. Wenn eine potentielle Greifmöglichkeit besteht, bekommt q für den Pixel $\boldsymbol{s}$ einen Wert von 0 bis 1 zugewiesen.

Da die Greifpose $\tilde{\boldsymbol{g}}$ in Bildkoordinaten angegeben ist, ist sie für das tatsächliche Greifen mittels Robotermanipulator ungeeignet. Damit der Robotermanipulator ein Objekt greifen kann, müssen ihm kartesische Raumkoordinaten übergeben werden. Um die kartesische Greifpose $\boldsymbol{g}$ zu erhalten, muss also $\tilde{\boldsymbol{g}}$ aus Bildkoordinaten in kartesische Koordinaten transformiert werden. Für diese Transformation werden die *intrinsischen* und *extrinsischen* Parameter genutzt. Die intrinsischen Parameter t_{CI} sind von der verwendeten Kamera abhängig. Damit können Bildkoordinaten in kartesische Koordinaten bzgl. des Kamerakoordinatensystems transformiert werden. Die extrinsischen Parameter t_{RC} transformieren anschließend die Informationen vom Kamerakoordinatensystem ins Roboterkoordinatensystem. Damit ist dann die kartesische Position der Greifpose $\boldsymbol{g}$ bzgl. des Roboterkoordinatensystems eindeutig bekannt. Es gilt folgende Gleichung:

$$\boldsymbol{g} = t_{RC}(t_{CI}(\tilde{\boldsymbol{g}})) \tag{3.3}$$

Die Gesamtheit aller Greifposen $\boldsymbol{g}$ in einer Aufnahme wird von *Morrison et al.* auch *Grasp-Map* $\boldsymbol{G}$ genannt. Diese besteht selbst aus der Winkelmatrix $\boldsymbol{\Phi}$, der Weitenmatrix $\boldsymbol{W}$ und der Qualitätsmatrix $\boldsymbol{Q}$, welche die jeweiligen Werte für $\tilde{\phi}$, $\tilde{w}$ und q für jeden Pixel $\boldsymbol{s}$ beinhalten. Für die *Grasp-Map* $\boldsymbol{G}$ gilt folgende Gleichung:

$$\begin{gathered} \boldsymbol{G} = (\boldsymbol{\Phi}, \boldsymbol{W}, \boldsymbol{Q}) \in \mathbb{R}^{3xHxB} \\ \text{mit } \boldsymbol{\Phi}, \boldsymbol{W}, \boldsymbol{Q} \in \mathbb{R}^{HxB} \end{gathered} \tag{3.4}$$

Die Qualitätsmatrix $\boldsymbol{Q}$ beinhaltet für jedes Pixel $\boldsymbol{s}$ ein Qualitätsmaß q. Dessen skalarer Wertebereich liegt zwischen [0,1], wobei die Eins für eine sehr gute Greifmöglichkeit steht. Die erfolgversprechendste Greifpose $\tilde{\boldsymbol{g}}^*$ liegt somit bei jenem Pixel $\boldsymbol{s}$, welches den höchsten skalaren Wert q beinhaltet. Dieses Pixel $\boldsymbol{s}$ gibt dann Aufschluss darüber, welches Element der *Grasp-Map* $\boldsymbol{G}$ die beste Greifpose $\tilde{\boldsymbol{g}}^*$ ist. Deshalb gilt folgende Beziehung:

$$\tilde{\boldsymbol{g}}^* = \max_{Q} \boldsymbol{G} \tag{3.5}$$

Um jedoch die Grasp-Map $\boldsymbol{G}$ berechnen zu können, müssen die Qualitätsmatrix $\boldsymbol{Q}$, die Weitenmatrix $\boldsymbol{W}$ und die Winkelmatrix $\boldsymbol{\Phi}$ gegeben sein (siehe Gleichung 3.4). Diese Matrizen werden anhand des Tiefenbildes $\boldsymbol{I}$ vom CNN erzeugt. Damit das CNN diese Aufgabe zuverlässig erfüllen kann, muss das neuronale Netz konfiguriert und mit entsprechenden Trainingsdaten trainiert werden. Diese Themen werden im weiteren Verlauf dieses Kapitels behandelt.

3.3. Trainingsdatenbank des CNN

Im Allgemeinen verhalten sich CNN's wie Blackbox-Systeme. Die Eingangs- und Ausgangsgrößen sind bekannt, die internen Prozesse des neuronalen Netzes sind jedoch abstrakt und vom Menschen kaum nachvollziehbar. Daher existiert i. d. R. keine Möglichkeit, ein trainiertes Netz zu korrigieren bzw. zu debuggen und das Verhalten an die Anforderungen anzupassen. Umso wichtiger ist es, die relevanten Konfigurationen vor dem Trainingsvorgang festzulegen. Das Training wird in erster Linie durch die Hyperparamter bestimmt. Dies sind Größen wie die Lernrate, die Anzahl der Trainingsepochen, die Batch-Size und die Menge der Schichten und Neuronen.

Neben den konfigurierbaren Hyperparametern ist die Kenntnis über die Eingangs- und Ausgangsgrößen des neuronalen Netzes unabdingbar. Es sollte klar sein, welche Ausgangsgrößen gesucht sind und welche Eingangsgrößen dafür nötig sind. Auf dieser Grundlage basiert letztlich die zum Training genutzte Datenbank. Die darin enthaltenen Informationen sind für das Verhalten des trainierten Netzes repräsentativ und bestimmend. Daher wird im Folgendem die für das Training des CNN genutzte Datenbank vorgestellt und analysiert.

3.3.1. Cornell-Datenbank

Für das Training des CNN verwenden *Morrison et al.* die *Cornell*-Datenbank ([7], [19]). Dies ist eine speziell für das Greifen von Objekten entwickelte Datenbank. Insgesamt wurden bei der Zusammenstellung der Datenbank 280 unterschiedliche Objekte aus verschiedenen Perspektiven und Orientierungen aufgenommen. Vertreten waren sowohl Haushaltsobjekte, Textilien, Elektronik und andere Kategorien. Durch diese Vielfalt soll das trainierte neuronale Netz möglichst flexibel bleiben und unterschiedlichste Objekte erkennen können. Insgesamt ergeben sich damit 1035 Samples mit jeweils folgendem Inhalt pro Sample (das XXXX in der Bezeichnung steht für die Nummerierung der Sampledaten von 0-1034):

1. RGB-Farbbild des Trainingsobjektes
   - Bezeichnung: `pcdXXXXr.png`
2. Punktewolke des Trainingsobjektes
   - Bezeichnung: `pcdXXXX.txt`
   - Die Punktewolken sind im PCD V 0.7 Format abgespeichert. [1]
3. Rechteckig markierte Greifposen
   - Bezeichnungen:
     - `pcdxxxxcpos.txt`: Beinhaltet gültige Greifposen
     - `pcdxxxxcneg.txt`: Beinhaltet ungültige Greifposen
     - Die rechteckigen Greifmarkierung sind durch ihre vier Eckpunkte definiert. Die Dateien enthalten deshalb die entsprechenden XY-Bildkoordinaten der Eckpunkte aller rechteckigen Greifmarkierungen.
     - Für das Training des GG-CNN werden nur die gültigen Greifposen genutzt, die ungültigen werden vernachlässigt.

[1] Definition des Datenformats: http://www.pointclouds.org/documentation/tutorials/pcd_file_format.php

Die Maße und die Orientierung der rechteckig markierten Greifposen am Objekt definieren die Greifweite und den Greifwinkel des Robotergreifers bei der späteren Anwendung. Dieser Prozess ist in Abbildung 3.3 schematisch dargestellt.

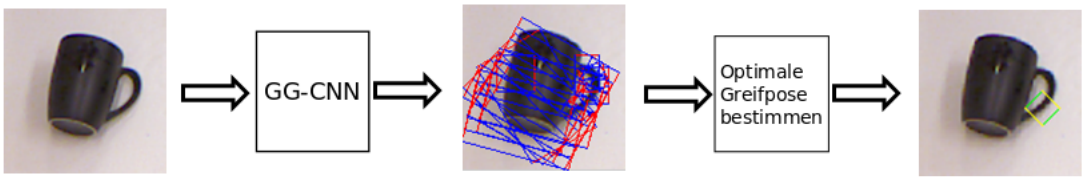

Abbildung 3.3.: Greifposengenerierung und Bestimmung (in Anlehnung an [19])

Für das Training des CNN's werden von *Morrison et al.* noch einige Veränderungen bzgl. der *Cornell*-Datenbank vorgenommen. Genaueres folgt im nächsten Abschnitt.

3.3.2. Erweiterte Cornell-Datenbank

Die *Cornell*-Datenbank ist mit seinen insgesamt 1035 Samples verhältnismäßig eher klein. Im Gegensatz dazu hat z. B. die *Jacquard-Grasping*-Datenbank [3] 54 Tausend und die *Dex-Net 2.0* [9] sogar 6.7 Millionen Datensamples, um nur einige zu nennen. Jedoch sind diese beiden Datenbanken synthetisch generiert worden und beinhalten maschinell gelabelte Objekte. Die Objektaufnahmen der Samples werden sowohl bei der *Jacquard-* als auch in der *Dex-Net*-Datenbank aus virtuellen CAD-Modellen gewonnen. Die Samples lassen sich daher nur im gewissen Rahmen mit realen Aufnahmen vergleichen. So wird den virtuellen Objektaufnahmen oft noch künstliches Rauschen hinzugefügt, um damit reale Aufnahmen nachzuahmen und das CNN so besser auf diese anzupassen. Jedoch eigenen sich Aufnahmen von echten Objekten i. d. R. besser für das Trainieren von CNN's, weshalb auch in dieser Arbeit die *Cornell*-Datenbank zum Einsatz kommt.

Im Allgemeinen gilt, dass sich die Leistung eines CNN's verbessert, je größer die Anzahl der Trainingssamples ist. Um der geringen Anzahl an Datensamples entgegenzuwirken, wird häufig für das Trainieren von CNN's die sog. *Datenaugmentation* verwendet. Dies ist ein Verfahren, bei dem die bestehenden Datensamples durch Spiegelung, Vergrößerung und Drehung vervielfacht werden. Dadurch entstehen mehrere Varianten eines Datensamples und die Menge der Datensamples erhöht sich dadurch insgesamt. Die *Cornell*-Datenbank

wird daher ebenfalls "augmentiert" und künstlich erweitert. Für diesen Vorgang wird das von *Morrison et al.* zur Verfügung gestellte Python-Skript genutzt. Dieses ist auf der beigefügten DVD hinterlegt.

Die erweiterte *Cornell*-Datenbank ist zunächst einmal in Trainings- und Testsamples aufgeteilt. Dies ist ein gängiges Verfahren beim Trainieren von neuronalen Netzen und nennt sich *Kreuzvalidierung*. Dabei wird das CNN zunächst mit den Trainingsdaten trainiert und anschließend mit den unbekannten Testdaten getestet. Beide Teilmengen müssen während des Trainings strikt voneinander getrennt werden. Die Testdaten dürfen nicht für das Training genutzt werden, da diese ansonsten nicht mehr für das Testen des trainierten CNN's geeignet wären. Die Performance des CNN's kann nur an unbekannten Daten getestet und validiert werden. In Abschnitt 4.4 wird das mit der erweiterten *Cornell*-Datenbank trainierte CNN getestet.

Insgesamt besteht die erweiterte *Cornell*-Datenbank damit aus 7110 Trainings- und 1740 Testsamples. Jedes Datensample beinhaltet folgende Informationen:

1. `rgb`: RGB-Farbbild (Größe: $300x300x3$)
2. `img_id`: Identifikationsnummer des Datensamples
   Die bei der Datenaugmentation erzeugten Varianten eines Datensamples erhalten die gleiche ID-Nummer wie das ursprüngliche Datensample. Durch die Wiederholung der ID's, ergeben sich somit insgesamt 1034 ID's für 7110 Datensamples.
3. `depth_inpainted`: RGB-D Tiefenbild (Größe: $300x300$)
   Die Tiefe ist in Metern angegeben.
4. `bounding_boxes`: Array (Größe: $nx4$)
   Hier werden die Bereiche im Bild definiert, in denen sich gültige Greifmöglichkeiten befinden. Diese Bereiche werden rechteckig markiert. In jeder Spalte wird ein Eckpunkt eines Rechteckes in Bildkoordinaten abgespeichert. Es ergeben sich damit vier Eckpunkte und entsprechend vier Spalten. Für jede weitere rechteckige Markierung wird eine Zeile hinzugefügt.
5. `grasp_points_img`: Binärbild (Größe: $300x300$)
   Jedes Pixel mit einer allgemein gültigen Greifmöglichkeit erhält eine 1, allen anderen wird die 0 als Wert zugewiesen.

6. `grasp_width`: Graustufenbild (Größe: $300x300$)
   Jedes Pixel mit einer allgemein gültigen Greifmöglichkeiten erhält einen Wert für die Greifweite an dieser Stelle zugewiesen.

7. `angle_img`: Graustufenbild (Größe: $300x300$)
   Jedes Pixel mit einer allgemein gültigen Greifmöglichkeiten, bekommt einen Wert zwischen $[-\frac{\pi}{2}, \frac{\pi}{2}]$ zugewiesen. Dies ist der entsprechende Greifwinkel an dieser Stelle.

Die folgenden Bilder verdeutlichen den beschriebenen Aufbau der erweiterten Cornell-Datenbank anhand eines beispielhaften Datensamples:

Abbildung 3.4.: RGB

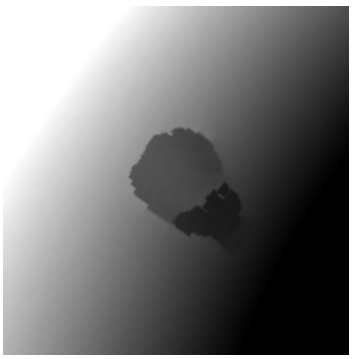

Abbildung 3.5.: depth_ inpainted

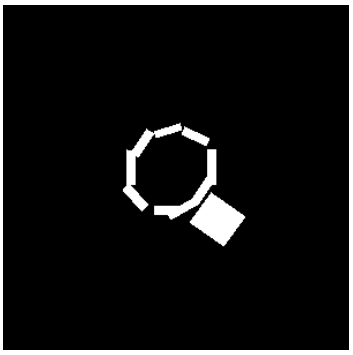

Abbildung 3.6.: grasp_ points_img

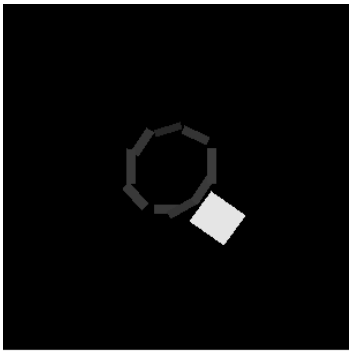

Abbildung 3.7.: grasp_ width

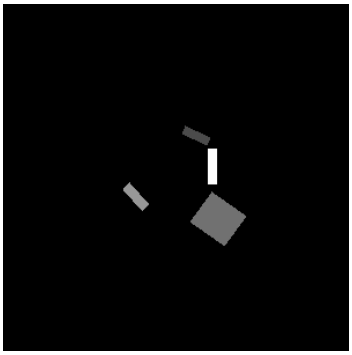

Abbildung 3.8.: ang_img

3.4. Netzwerkarchitektur des CNN's

Die Festlegung der Netzwerkarchitektur eines neuronalen Netzes ist für dessen Performance entscheidend. Die Anzahl der Schichten und der darin enthaltenen Neuronen bestimmen die

Parameterzahl des CNN's. Mit steigender Parameterzahl kann ein neuronales Netz immer komplexere Muster erlernen. Jedoch ist dies nicht immer gewünscht, da das Netz bei einer Überdimensionierung der Netzwerkarchitektur, auch leichter zur Überanpassung neigt. Das eigentliche Ziel des Trainings besteht darin, dass das neuronale Netz *verallgemeinerungsfähig* bleibt und dass die *Überanpassung* an den Trainingsdaten verhindert wird. Verallgemeinerungsfähig bedeutet in diesem Zusammenhang, dass das trainierte CNN auch Schlüsse bzgl. neuer, unbekannter Daten ziehen kann. Eine Überanpassung besteht dann, wenn das CNN keine allgemeinen Merkmale mehr lernt, sondern die gegebenen Trainingssamples "auswendig" lernt und nur noch gelerntes sinnvoll verarbeiten kann. Das Training ist daher stets der Versuch, einen zufriedenstellenden Kompromiss zwischen diesen beiden Größen zu finden.

Da das Ziel dieser Arbeit darin besteht, Greifposen auch an unbekannten Objekten zu finden, ist eine Überanpassung unbedingt zu vermeiden und eine gewisse Verallgemeinerungsfähigkeit zu gewährleisten. Beim Entwurf der Netzwerkarchitektur ist es üblich, mit wenigen neuronalen Faltungsschichten zu beginnen, diese schrittweise zu erhöhen und anschließend zu prüfen, ob beim CNN eine Überanpassung vorliegt oder ob noch Verbesserungspotential besteht. Dafür wird die *Kostenfunktion* verwendet. Stagniert diese nach einer gewissen Anzahl von Trainingsepochen, liegt üblicherweise eine Überanpassung vor. In dem Fall empfiehlt es sich, einige neuronale Faltungsschichten des CNN's zu entfernen. Zusätzliche Faltungsschichten sollten hinzugefügt werden, wenn die Kostenfunktion nicht stagniert und das CNN noch Verbesserungspotential besitzt.

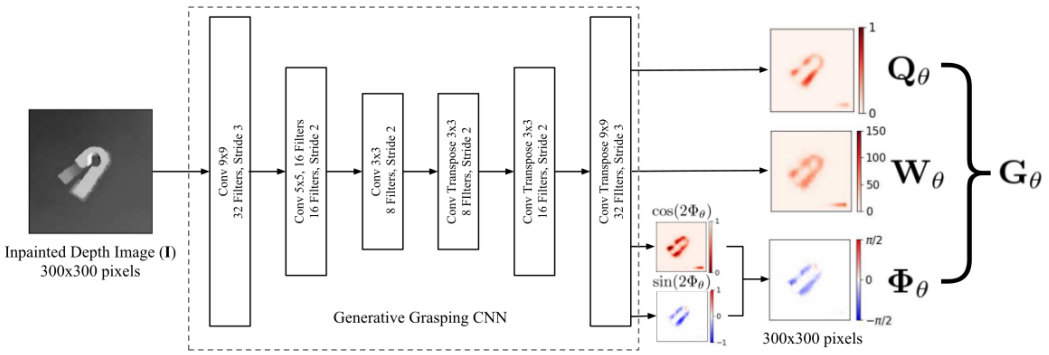

Abbildung 3.9.: Netzwerkarchitektur mit Input und Output [11]

In dieser Arbeit wird die vorgeschlagene Netzwerkarchitektur von [11] verwendet. Diese besitzt insgesamt sechs Schichten. Das Netz besitzt folgenden Aufbau:

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
input_1 (InputLayer)         (None, 300, 300, 1)       0
__________________________________________________________________________________________________
conv2d_1 (Conv2D)            (None, 100, 100, 32)      2624        input_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)            (None, 50, 50, 16)        12816       conv2d_1[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)            (None, 25, 25, 8)         1160        conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_transpose_1 (Conv2DTrans (None, 50, 50, 8)      584         conv2d_3[0][0]
__________________________________________________________________________________________________
conv2d_transpose_2 (Conv2DTrans (None, 100, 100, 16)   3216        conv2d_transpose_1[0][0]
__________________________________________________________________________________________________
conv2d_transpose_3 (Conv2DTrans (None, 300, 300, 32)   41504       conv2d_transpose_2[0][0]
__________________________________________________________________________________________________
pos_out (Conv2D)             (None, 300, 300, 1)       129         conv2d_transpose_3[0][0]
__________________________________________________________________________________________________
cos_out (Conv2D)             (None, 300, 300, 1)       129         conv2d_transpose_3[0][0]
__________________________________________________________________________________________________
sin_out (Conv2D)             (None, 300, 300, 1)       129         conv2d_transpose_3[0][0]
__________________________________________________________________________________________________
width_out (Conv2D)           (None, 300, 300, 1)       129         conv2d_transpose_3[0][0]
==================================================================================================
Total params: 62,420
Trainable params: 62,420
Non-trainable params: 0
__________________________________________________________________________________________________
```

Die ersten drei neuronalen Faltungsschichten (in Keras vom Typ `Conv2D`) filtern die Informationen aus dem eingegebenen Tiefenbild. Hierbei wird das Bild über eine Faltungsoperation schrittweise verkleinert. Die erste Faltungsschicht hat 32 Filtermasken und durchläuft das eingegebene Tiefenbild mit diesen Filtermasken. Dabei wird eine Filtermaske immer um drei Pixel pro Rechenschritt verschoben, bis das ganze Tiefenbild gefaltet wurde. Aus dem ursprünglichen Tiefenbild mit der Größe 300x300 wird somit ein gefiltertes Bild mit der Größe 100x100 erzeugt. Die zweite Faltungsschicht hat 16 und die dritte nur 8 Faltungsmasken. Beide durchlaufen das Bild um zwei Pixel pro Rechenschritt. Am Ende der drei Faltungsstufen entsteht ein Bild der Größe 25x25. Dieses wird mit anschließend über drei zusätzliche `Con2DTrans`-Schichten auf die Originalgröße 300x300 zurücktransformiert. Insgesamt liefert das CNN vier Ausgaben: `pos_out` bzw. Q_θ, `width_out`

bzw. W_θ und `cos_out, sin_out` bzw. Φ_θ. `pos_out` beinhaltet die Greifqualitäten und `width_out` die Greifweiten pro Pixel. Der Greifwinkel Φ_θ wird aus den Bildern `sin_out` und `cos_out` berechnet. Dafür wird folgende Formel verwendet:

$$\Phi_\theta = \frac{1}{2} \arctan \frac{\sin(2\Phi_\theta)}{\cos 2\Phi_\theta} \tag{3.6}$$

Insgesamt besitzt das CNN 62 Tausend Parameter. Zum Vergleich hat das *Dex-Net 2.0* z. B. 18 Millionen Parameter. Jedoch erkennt und analysiert das *Dex-Net 2.0* auch ganze Objekte und generiert anhand dessen die Greifpose. Das CNN erkennt jedoch nur einfache Geometrien am Objekt und generiert daraus Greifposen. Bei einem größeren Netz würde es auch komplexere Geometrien erlernen und sich somit zu sehr an die Trainingsobjekte anpassen. Dies ist in dieser Anwendung aber nicht erwünscht.

Das von [11] vorgeschlagene Netz (siehe Abbildung 3.9) wurde im Rahmen dieser Arbeit geprüft. Ein größeres Netz hat zu keiner signifikanten Verbesserung der Greifposengenerierung geführt. Deshalb wird im Folgenden die vorgeschlagene Netzwerkarchitektur nach [11] verwendet.

3.5. Training des CNN

Für diese Arbeit wird das CNN auf einem lokalen Rechner trainiert und validiert. Die Hardwarespezifikationen des Rechners sollten denen aus Tabelle 2.1 gleichwertig sein. Außerdem muss die Software aus Tabelle 2.2 auf dem Rechner installiert sein.

Beim Trainieren von neuronalen Netzen ist es üblich, die vorliegende Datenbank in Trainings- und Testsamples aufzuteilen. Diese Aufteilung ist für die spätere Leistungsbeurteilung des neuronalen Netzes nach dem Training notwendig. Das trainierte neuronale Netz optimiert seine Gewichtungen anhand der gegebenen Trainingssamples und erlernt somit die gegebenen Merkmale der Trainingssamples. Eine Leistungsbeurteilung anhand der Trainingsdaten ist daher nicht sinnvoll, da Bekanntes vom Netz sofort wiedererkannt wird. Für diesen Zweck werden die dem neuronalen Netz vollkommen unbekannten Testsamples verwendet. Dadurch lässt sich die Leistung und die Verallgemeinerungsfähigkeit des neuronalen Netzes zuverlässig prüfen. Eine strickte Trennung der Trainings- und Testdaten ist

daher während des Trainings immer zu gewährleisten, da die Leistungsbeurteilung sonst nicht gültig ist. In dieser Arbeit werden 80% der *Cornell*-Datenbank für das Training und 20% für das Testen verwendet.

Für die Anpassung der Gewichtungen bzw. der Parameter des neuronalen Netzes wird eine Optimierungsfunktion eingesetzt. Diese ermöglicht dem neuronalen Netz, seine Gewichtungen anhand der bekannten Trainingssamples und den dazugehörigen Werten der Verlustfunktion iterativ selbst zu aktualisieren. Die Verlustfunktion beschreibt hingegen, wie das neuronale Netz seine Leistung für die Trainingssamples beurteilen kann und damit auch, wie die Gewichtungen konkret zu korrigieren sind. Bei der Optimierung der Parameter besteht das Ziel darin, das globale Minimum der quadratisch differenzierbaren Funktion im mehrdimensionalen Raum zu finden. Im globalen Minimum besitzt das neuronale Netz die niedrigsten Werte der Verlustfunktion und somit die bestmögliche Leistung.

In dieser Arbeit wird für Optimierung die *RMSProp*-Methode gewählt. Dies ist eine Variante des stochastischen Gradientenabstiegsverfahrens, welches nicht nur den Gradienten an einer bestimmten Stelle einer quadratisch differenzierbaren Funktion in Betracht zieht, sondern auch das Moment als zusätzlichen Parameter. Dadurch können lokale Minima und Sattelpunkte auf der quadratisch differenzierbaren Funktion besser überwunden und globale Minima wahrscheinlicher gefunden werden.
Die *RMSProp*-Optimierungsmethode ist außerdem für das *Batch*-Training geeignet. Anders als bei der *Mini-Batch*-Methode, wird das neuronale Netz mit den gesamten Trainingsdaten gleichzeitig trainiert. Bei der *Mini-Batch*-Methode wird das neuronale Netz mit Bruchteilen der Trainingsdaten trainiert. Diese Methode benötigt weitaus weniger Rechenleistung beim Trainieren von neuronalen Netzen und ist i.d.R. zu bevorzugen. Dafür muss die gegebene Trainingsdatenbank aber weitestgehend homogen sein, d.h. dass sich alle Trainingssamples ähneln oder vom gleichen Objekt sein müssen. Die *Cornell*-Datenbank beinhaltet jedoch teilweise sehr unterschiedliche Trainingssamples von diversen Objekten und ist daher stark inhomogen und für das *Mini-Batch*-Training ungeeignet. Daher muss für diese Arbeit die rechenintensive *Batch*-Methode gewählt werden.

Mit dieser Methodik entstehen mit insgesamt fünfzig Trainingsepochen auch fünfzig unterschiedliche CNN's. Zu Beginn jeder Trainingsepoche werden die Parameter des jeweils zu trainierenden CNN's zufällig initialisiert. Die Variation der Ausgangslage führt dazu, dass nach dem Training jedes CNN unterschiedliche Parameter optimiert hat und man somit

insgesamt fünfzig vollkommen unterschiedliche CNN's erhält. Jedes trainierte CNN wird direkt nach Abschluss der Trainingsepoche mit den Trainingsdaten getestet. Der Trainingsvorgang wurde aufgezeichnet und ist in geglätteter Form in Abbildung 3.10 dargestellt. Die nicht geglätteten Verläufe der Verlustfunktionen sind in Abbildung 3.11 dargestellt.

Insgesamt besitzt das trainierte CNN vier Ausgaben: `Pos_Out`, `Width_Out`, `Cos_Out` und `Sin_Out`. In den Abbildungen 3.10 und 3.11 sind die Verläufe der jeweiligen Verlustfunktionen dargestellt. In diesen Plots ist besonders die Kostenfunktion entscheidend. Diese bildet die Summe der einzelnen Verlustfunktionen und liefert eine allgemeine Aussage über die Leistungsfähigkeit des CNN's. Wie in den Kostenfunktion zu sehen ist, werden die trainierten CNN's tendenziell mit jeder Epoche besser, auch wenn die Gewichtungen immer wieder neu und zufällig initialisiert werden. Das liegt an der adaptiven Lernrate der *RMSProp*-Optimierungsmethode. Diese wird bei der nächsten Epoche mitberücksichtigt und weiter optimiert. Bei der Kostenfunktion in Abbildung 3.10 ist zu sehen, dass sich die Leistung der CNN's bis zur 30. Epoche stetig verbessert. Von da an ist die Verbesserung marginal. Der Plot der nicht geglätteten Kostenfunktion in Abbildung 3.11 zeigt sogar, dass sich die Werte ab der 30. Epoche teilweise sogar verschlechtern. Insgesamt schwingt die Kostenfunktion um den Wert 0.128. In dieser Arbeit wird daher das in der 29. Epoche trainierte CNN ausgewählt. Dieses liegt vor dem Stagnieren der Kostenfunktion und besitzt ein fast globales Minimum. Alternativ würde sich das CNN aus der 41. Epoche auch eignen. Die Qualität der Greifposengenerierung des ausgewählten CNN's wird in Abschnitt 4.4 geprüft.

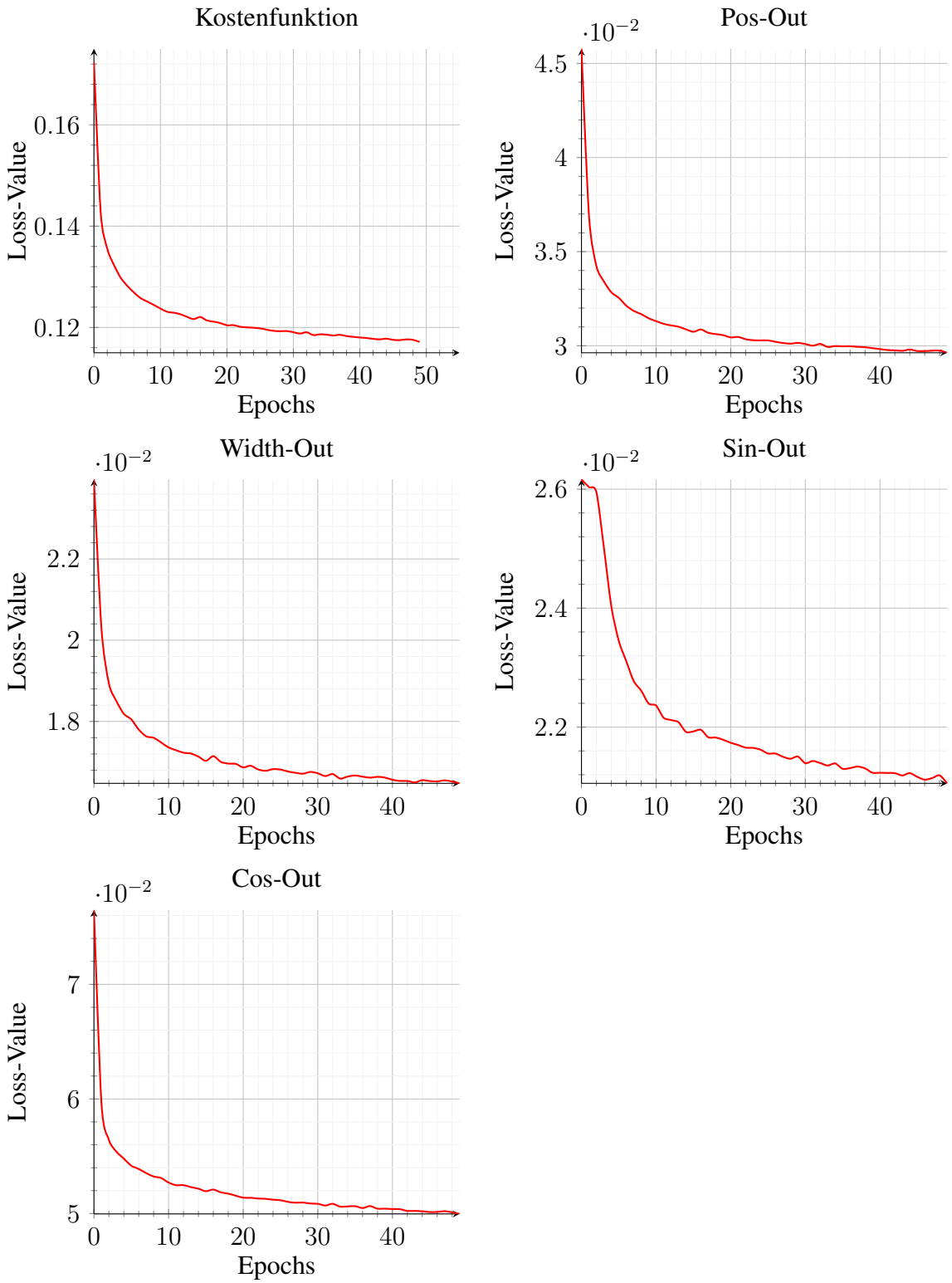

Abbildung 3.10.: Testergebnisse des CNN-Trainings (geglättet)

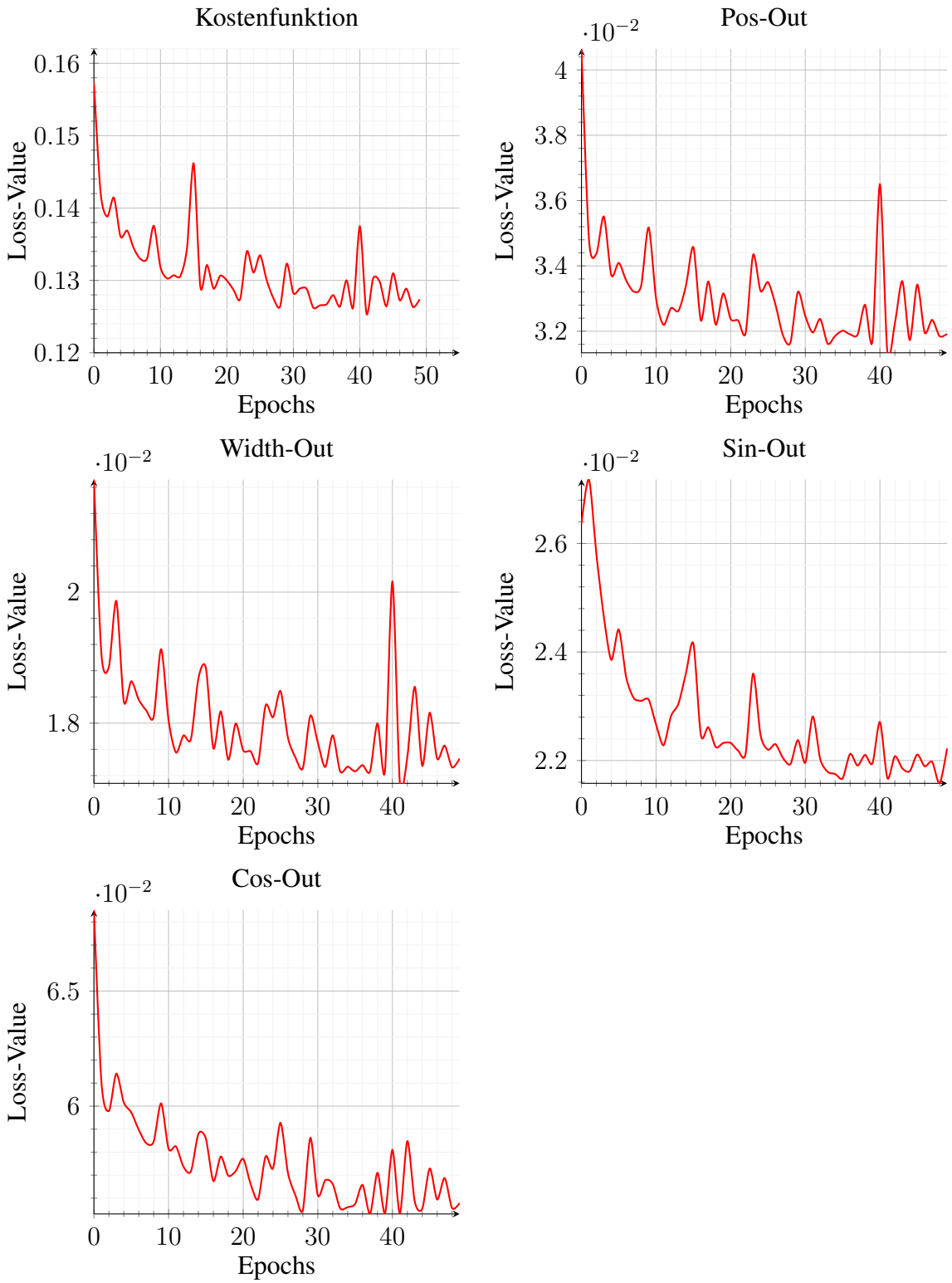

Abbildung 3.11.: Testergebnisse des CNN-Trainings (nicht geglättete)

3.6. Einbindung des trainierten CNN's in ROS

In diesem Abschnitt wird die Einbindung des trainierten CNN's über Python in die ROS Umgebung des Roboters behandelt. Hierfür wurde im Rahmen dieser Arbeit ein entsprechendes ROS-Node[2] entwickelt. Dessen Funktionsprinzip wird in Abbildung 3.1 über das Ablaufdiagramm illustriert. Auf dessen Basis werden nachfolgend die einzelnen Schritte und deren Implementierung genauer betrachtet. Hier sei angemerkt, dass sich die Zeilenangaben im Text auf den Programmcode in A.5 beziehen. Um die Abläufe genauer erläutern zu können, werden die in den Zwischenschritten bearbeiteten Tiefenbilder anhand von diversen Beispiel veranschaulicht.

Als erstes muss das ROS-Node jedoch initialisiert werden. Hierfür werden zu Beginn diverse Pakete importiert und einige Variablen festgelegt. Relevant ist hier vor allem der Systempfad des trainierten CNN's (Zeile 38), welcher je nach Speicherort des trainierten CNN's angepasst werden muss. Des weiteren ist die Initialisierung im Programmcode selbst mit entsprechenden Kommentaren erklärt.
Der eigentliche Algorithmus ist in der `grasp_prediction_callback` Callback-Funktion hinterlegt. Dieser kann in vier aufeinander folgende Sequenzen aufgeteilt werden:

- Bildvorverarbeitung (Zeile 79 bis 111)
- Greifposengenerierung über das CNN (Zeile 112 bis 181)
- Visualisierung der Greifpose (Zeile 182 bis 223)
- Bereitstellung der Informationen über ROS (Zeile 223 bis 237)

Die folgenden Abschnitte beschreiben die Funktionsweisen der einzelnen Sequenzen. Dabei wird auf das Ablaufdiagramm in Abbildung 3.1 und auf den Programmcode im Anhang A.5 Bezug genommen. Die Greifposengenerierung wird beispielhaft anhand eines handelsüblichen Adapteranschlusses illustriert. Weitere Beispiele sind im Anhang A.6 zu finden.

[2] Kann über den Befehl `rosrun cnn_grasping cnn_grasping_node.py` gestartet werden

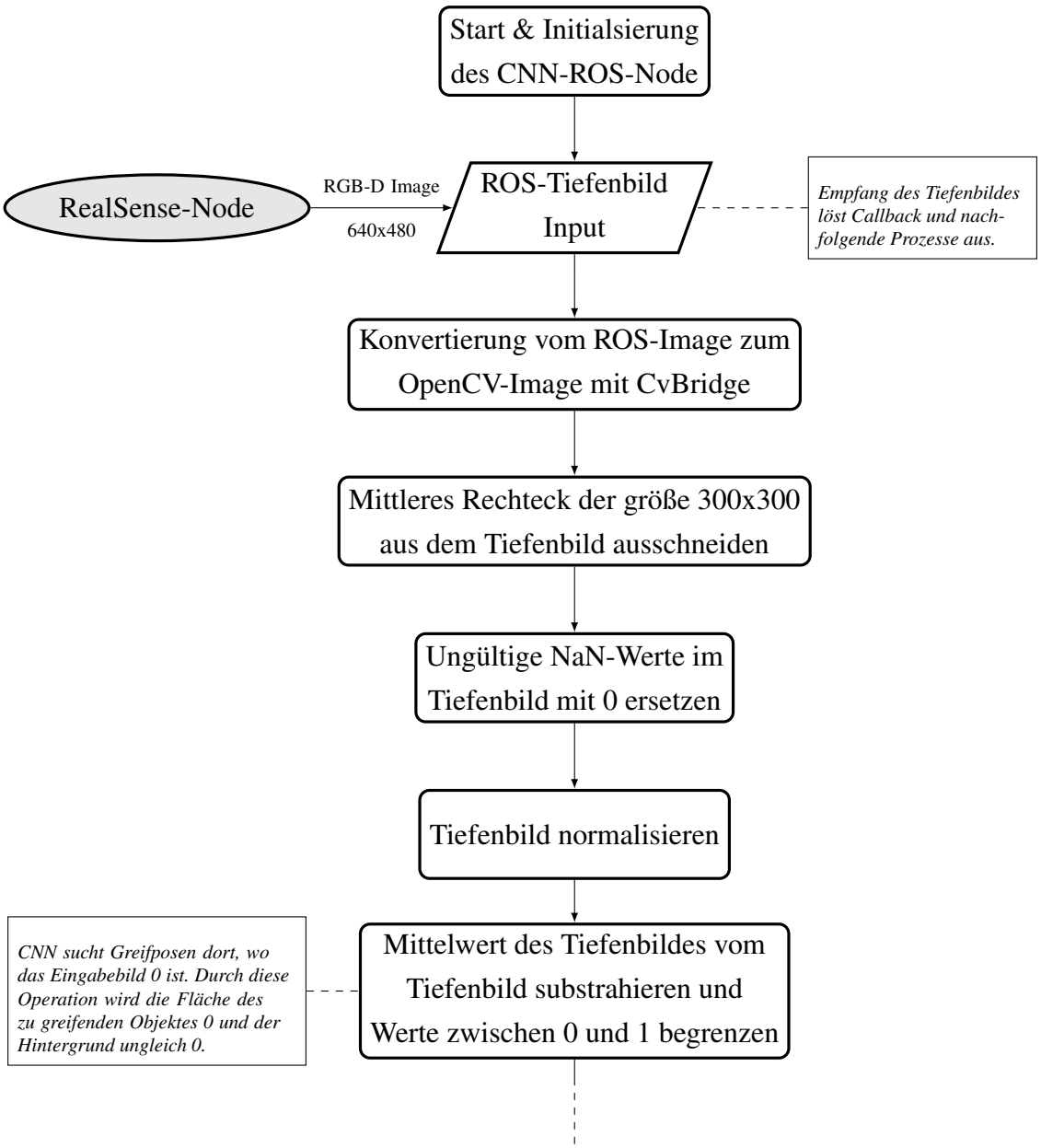

Abbildung 3.13.: Greifposendetektion - Ablaufdiagramm

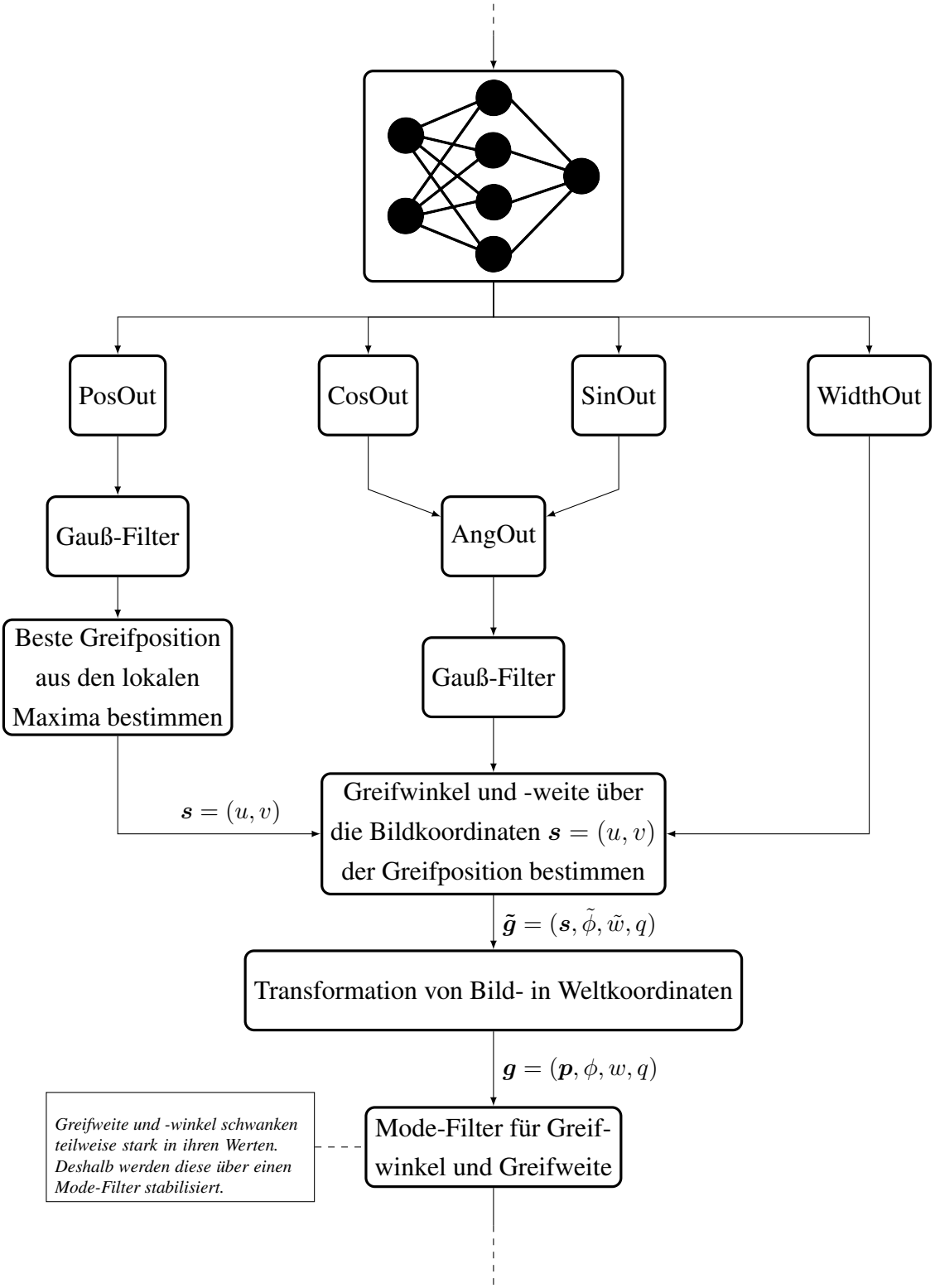

Abbildung 3.13.: Greifposendetektion - Ablaufdiagramm

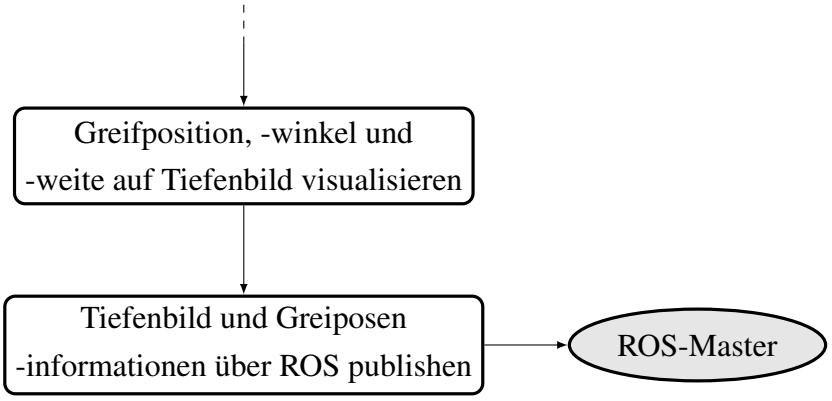

Abbildung 3.13.: Greifposendetektion - Ablaufdiagramm

3.6.1. Bildvorverarbeitung

Gestartet wird der Callback mit den Empfang eines Tiefenbildes vom *ROS-RealSense*-Node. Um dieses weiter bearbeiten zu können, muss es zunächst mit der `CvBridge` vom ROS-Image Datentyp in ein `OpenCV`-Image bzw. Array umgewandelt werden. Die Informationen innerhalb des Tiefenbildes bleiben dabei unverändert. Das empfangene Tiefenbild ist insgesamt 640x480 Pixel groß und in Abbildung 3.14 zu sehen. Die Tiefe ist in Millimetern angegeben. Anschließend wird das empfangene Tiefenbild auf die mittleren 300x300 Pixel zugeschnitten und normalisiert. Das trainierte CNN kann nur mit Tiefenbildern dieser Größe und dem Wertebereich zwischen 0 und 1 arbeiten.

Abbildung 3.12.: Adapter

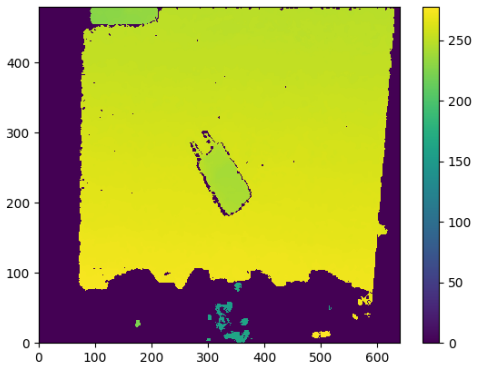

Abbildung 3.14.: Tiefenbild

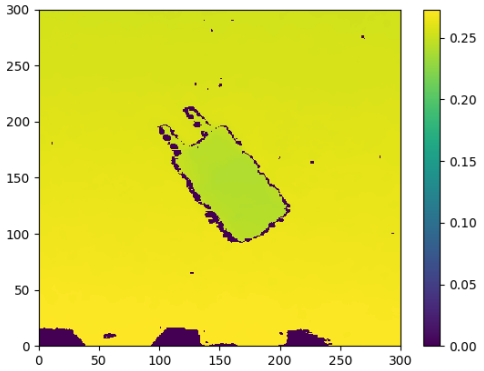

Abbildung 3.15.: Zugeschnitten und normalisiertes Tiefenbild

Um die Greifposendetektion zu stabilisieren und qualifizierte Vorhersagen treffen zu können, muss zunächst eine Segmentierung zwischen Objekt und Hintergrund stattfinden. Zudem werden Greifposen im Eingangsbild insbesondere an Stellen gesucht, die den Wert Null haben. Deshalb sollten Objektflächen im Idealfall den Wert Null besitzen. Dafür wird vom Tiefenbild dessen Mittelwert subtrahiert und auf den Wertebereich zwischen 0 und 1 begrenzt. Das Ergebnis ist in Abbildung 3.16 dargestellt. Dieses wird in dieser Form direkt an das CNN zur Greifposengenerierung übergeben.

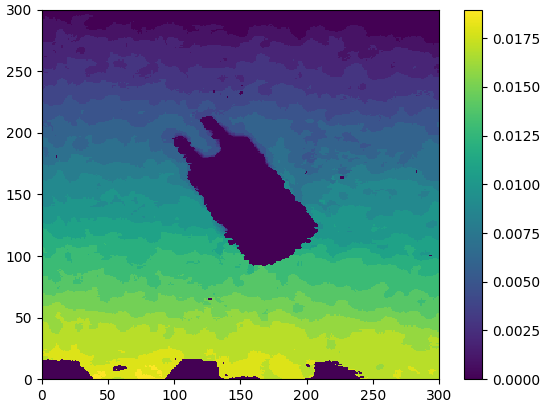

Abbildung 3.16.: Segmentiertes Tiefenbild

3.6.2. Greifposengenerierung über das CNN

Das CNN bekommt das editierte Tiefenbild aus Abbildung 3.16 als Eingabebild übergeben. Insgesamt liefert das CNN vier Ausgabebilder zurück:

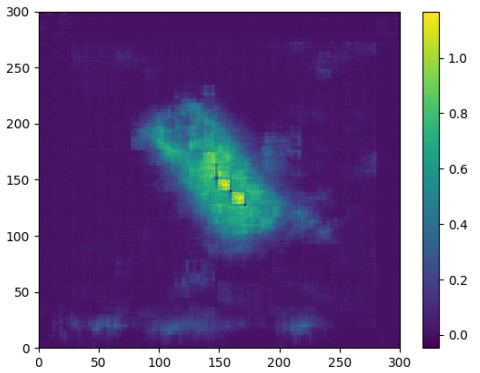

Abbildung 3.17.: PosOut

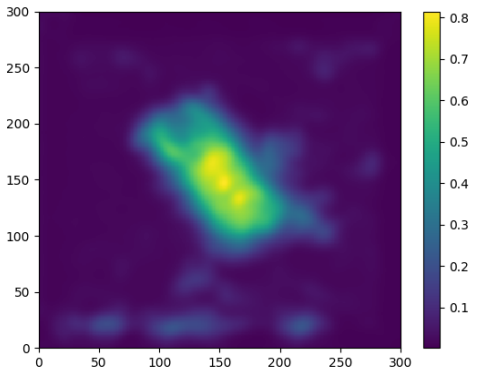

Abbildung 3.18.: PosOut gefiltert

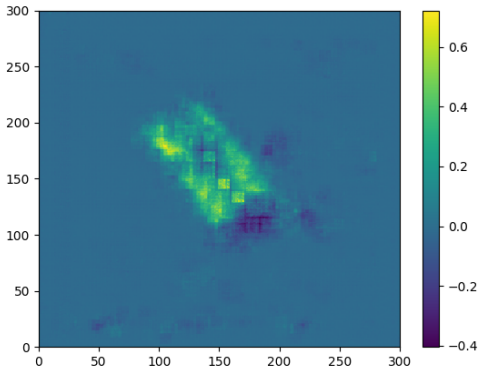

Abbildung 3.19.: SinOut

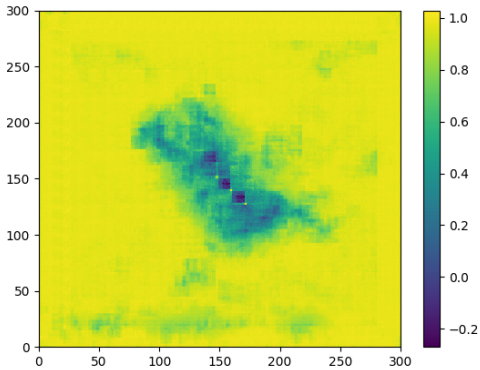

Abbildung 3.20.: CosOut

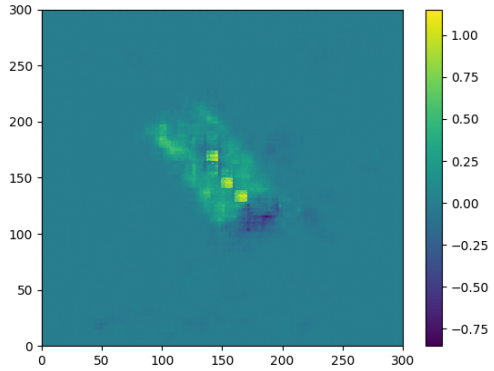

Abbildung 3.21.: AngOut

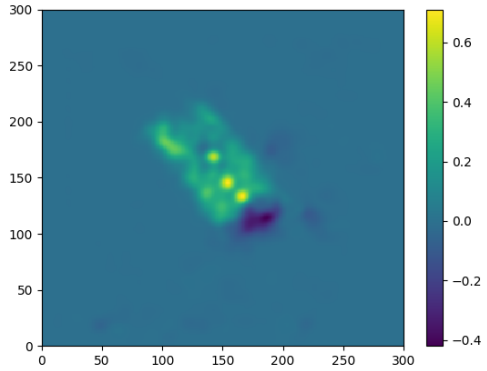

Abbildung 3.22.: AngOut gefiltert

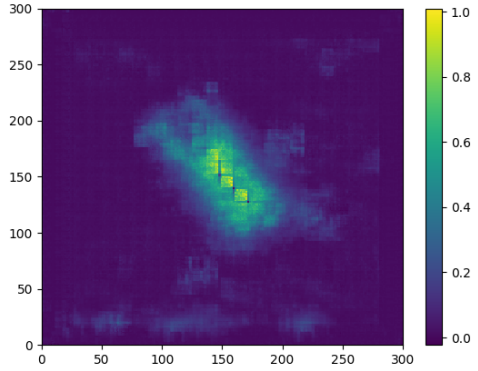

Abbildung 3.23.: WidthOut

Das `PosOut`-Ausgabebild gibt zu jedem seiner Pixel ein Qualitätsmaß für potentielle Greifpositionen an. In Abbildung 3.17 ist zu sehen, dass das CNN die höchsten Greifpotentiale richtigerweise in der mittleren Körperachse des Objektes sieht. Um jedoch das höchste Potential und somit die beste Greifposition zu erhalten, wird `PosOut` zuvor mit einem Gauß-Filter geglättet. Dadurch wird Rauschen im Potentialfeld minimiert. Zudem lassen sich Potentialspitzen eindeutiger lokalisieren. Anschließend wird aus einem Satz lokaler Potentialmaxima, das Pixel mit dem höchsten Potential ausgewählt. Über die Bildkoordinaten $\boldsymbol{s} = (u, v)$ dieses Pixels kann rückwirkend aus `AngOut` und `WidthOut` der Greifwinkel und die Greifweite bestimmt. Hierfür werden lediglich die Werte aus den Pixeln der gleichen Bildkoordinate $\boldsymbol{s}$ der jeweiligen Bilder ausgelesen. Da `AngOut` keine explizite Ausgabe des CNN's ist, muss diese noch vorher berechnet werden. Über die Formel

3.6 wird diese aus den Ausgaben `SinOut` und `CosOut` generiert. Anschließend wird auch `AngOut` über einen Gauß-Filter geglättet.

3.6.3. Transformation der Greifpose von Bild- in Roboterkoordinaten

Um ein Objekt mit dem Robotermanipulator greifen zu können, wird die Greifpose $\tilde{\boldsymbol{g}} = (\boldsymbol{p}, \phi, w, q)$ bzgl. des Roboterkoordinatensystems benötigt. Deshalb muss die ermittelte Greifpose $\tilde{\boldsymbol{g}} = (\boldsymbol{s}, \phi, w, q)$ zunächst vom Bildkoordinatensystem in das Roboterkoordinatensystem transformiert werden.
Der Robotermanipulator bewegt sich vor dem eigentlichen Greifprozess in eine vordefinierte Ausgangslage über dem Objekt (siehe Abbildung 2.12). Von dieser Position aus greift der Roboter in einer offenen Schleife blind nach dem Objekt. Dadurch ist auch die Höhe während der Greifposengenerierung konstant. Durch Messungen aus dieser Position heraus konnte festgestellt werden, dass das Tiefenbild mit einer Größe von 300x300 Pixeln einer reale Höhe und Breite von jeweils $17.5cm$ entspricht.

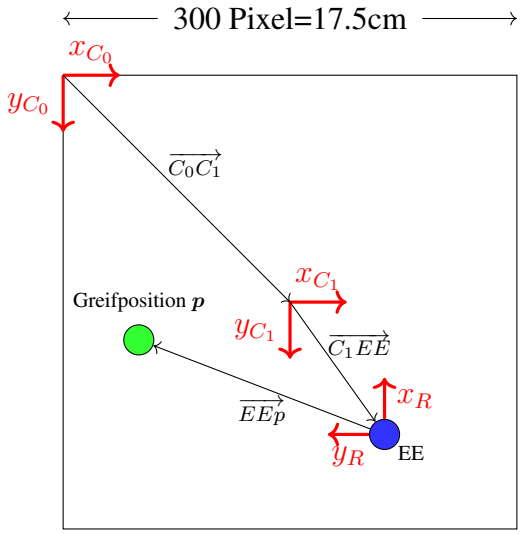

Abbildung 3.24.: Umrechnung von Bild- in Roboterkoordinaten

Zunächst wird der Faktor λ für die Umrechnung von Pixellängen in Zentimeter berechnet. Anschließend wird der Abstand der detektierten Greifposition $\boldsymbol{p}$ zum Ursprungskoordinatensystem C_0 von Pixellängen in Zentimeter umgerechnet:

$$\lambda = \frac{17.5cm}{300Pixel} = 0.058\overline{3} \tag{3.7}$$

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} p_u \\ p_v \end{bmatrix} \cdot \lambda \tag{3.8}$$

Da das Tiefenbild quadratisch ist, gilt λ sowohl für die horizontale, als auch für die vertikale Umrechnung. Auch die Position des Roboterendeffektors ist in Bildkoordinaten gemessen worden. Zwischen dem Bildmittelpunkt des Koordinatensystems C_1 und der Position des Roboterendeffektors besteht ein Versatz von $2cm$ in X- und $4.5cm$ in Y-Richtung

des Koordinatensystems C_1. Auf diese Weise lässt sich die Greifposition ins Verhältnis zum Endeffektor setzen. Der relative Abstand $\overrightarrow{EEp}$ in Zentimetern zwischen dem Roboterendeffektor und der realen Greifposition am Objekt lässt sich wie folgt berechnen:

$$\begin{aligned}\overrightarrow{EEp} &= \overrightarrow{p_{(x,y)}} - \overrightarrow{EE_{(x,y)}} \\ &= \overrightarrow{p_{(x,y)}} - (\overrightarrow{C_0C_1} + \overrightarrow{C_1E\acute{E}}) \\ &= \begin{bmatrix} p_x \\ p_y \end{bmatrix} - (\begin{bmatrix} 8.25cm \\ 8.25cm \end{bmatrix} + \begin{bmatrix} 2cm \\ 4.5cm \end{bmatrix})\end{aligned} \tag{3.9}$$

Das CNN liefert grundsätzlich stabile und korrekte Ausgaben zu dem Greifwinkel und der Greifweite an einer bestimmten Greifposition. Jedoch kann es vorkommen, dass sich diese Ausgaben für dieselbe Greifposition zeitweise stark voneinander unterscheiden und um einen bestimmten Wert schwingen. Um diesem Verhalten entgegenzuwirken und jederzeit eine zuverlässige Aussage zu erhalten, werden beide Ausgaben mit einem Mode-Filter gefiltert. Dieser registriert zunächst über eine definierte Anzahl an Iterationen den Greifwinkel und die -weite. Anschließend wird analysiert, welcher Wert am häufigsten unter den gespeicherten Ausgabewerten vertreten ist. Dieser wird dann als Greifwinkel oder -weite ausgewählt. Dadurch werden abweichende Werte ausgefiltert und die Angaben zum Greifwinkel und der -weite werden insgesamt zuverlässiger.
Der Greifwinkel ist nur eine skalare Angabe über die Verdrehung des Roboterendeffektors um dessen negative, globale Z-Achse. Der OM6 befindet sich vor dem Greifen der Objekte in einer vordefinierte Ausgangsposition (siehe Abbildung 2.12). Für die Einstellung des Greifwinkels reicht es daher aus, nur die sechste Roboterachse um den entsprechenden Winkel zu verdrehen und die Greifposition dann anzufahren. Die Greifweite wird im Rahmen dieser Arbeit nicht weiter berücksichtigt. Der Greifer des OM6 wird hier nur geschlossen oder geöffnet. Jedoch ist der Greifer mit weichem Schaumstoffmaterial ausgestattet, der das Greifen von Objekten ermöglicht.

3.6.4. Visualisierung der Greifpose

Die Greifposition am Objekt wird in dieser Arbeit mit einem grünen Punkt auf dem ursprünglich aufgenommenen Tiefenbild markiert. Dieser aktualisiert seine Position im Bild

anhand der Ausgabe des CNN's. Der Greifwinkel und die -weite werden mithilfe einer Linie dargestellt. Dieser ändert seinen Winkel und seine Länge entsprechend der Ausgaben für den Greifwinkel und der Greifweite. Ein Beispiel ist in Abbildung 3.25 gegeben.

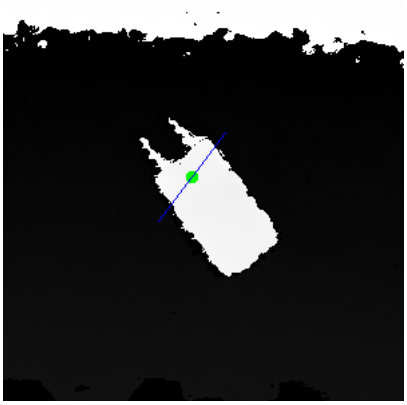

Abbildung 3.25.: Ausgabebild mit eingezeichneter der Greifpose

3.6.5. Bereitstellung der Informationen über ROS

Für die Bereitstellung der Greifposeninformationen werden folgende ROS-Publisher verwendet:

1. `ggcnn/img/grasp` vom ROS-Typ `Image`:
   Über diesen ROS-Publisher wird das Tiefenbild mit eingezeichneter Greifpose an die ROS-Umgebung publiziert.

2. `ggcnn/out/command` vom ROS-Typ `Float32MultiArray`:
   Veröffentlichung der Greifposeninformationen in Form eines Arrays mit folgender Struktur:

   [Greifposition X, Greiposition Y, Greifposition Z, Greifweite, Greifwinkel]

Diese Informationen werden im nächsten Kapitel für die Steuerung des Roboters genutzt.

3.7. Leistungsbeurteilung des CNN's

Für das Testen des trainierten CNN's werden diverse Haushaltsobjekte wie z.B. Stifte, Becher, Werkzeuge usw. genutzt. Eine Auswahl von Objekten und deren Ergebnisse ist im Anhang A.6 zu finden. Außerdem wird hierfür nicht der RPI, sondern der Arbeitsrechner genutzt. Insgesamt kann festgestellt werden, dass das CNN die optimale Greifposition besonders bei einfachen und strukturell flachen Objekten findet. Dafür wird vor allem die äußere Kontur des Objektes genutzt. Da für die Greifposendetektion nur eine Tiefenbildaufnahme aus der Vogelperspektive genutzt wird, werden Objekte mit abstehenden Elementen in Richtung der Kamera nicht erkannt.
Insgesamt ist das in dieser Arbeit trainierte CNN für das Greifen von Objekten mittels eines Robotermanipulators geeignet. Greifposen können verallgemeinert auch an unbekannten Objekten korrekt gefunden werden. Deshalb wird das CNN im nächsten Kapitel in die Robotersteuerung eingebunden.

4. Implementierung des CNN's in die Robotersteuerung

4.1. Steuerungsablauf

Im vorherigen Kapitel wurde die Theorie hinter der Greifposengenerierung präsentiert und ein darauf aufbauender Algorithmus entwickelt. Als Ausgabe wird letztlich die Greifposition bzgl. des Roboterendeffektors, der Greifwinkel und die Greifweite bereitgestellt. Die Greifposition und der -winkel werden dazu verwendet, den Robotermanipulator in die richtige Greifpose zu bewegen und das Objekt zu Greifen. Die Greifweite wird im Rahmen dieser Arbeit nicht weiter benötigt, ist aber grundsätzlich stabil genug um in zukünftige Weiterentwicklungen dieser Arbeit implementiert zu werden. In Abbildung 4.1 wird die Beziehung zwischen den Roboterkomponenten und dem Anwender während des Greifprozesses dargestellt.

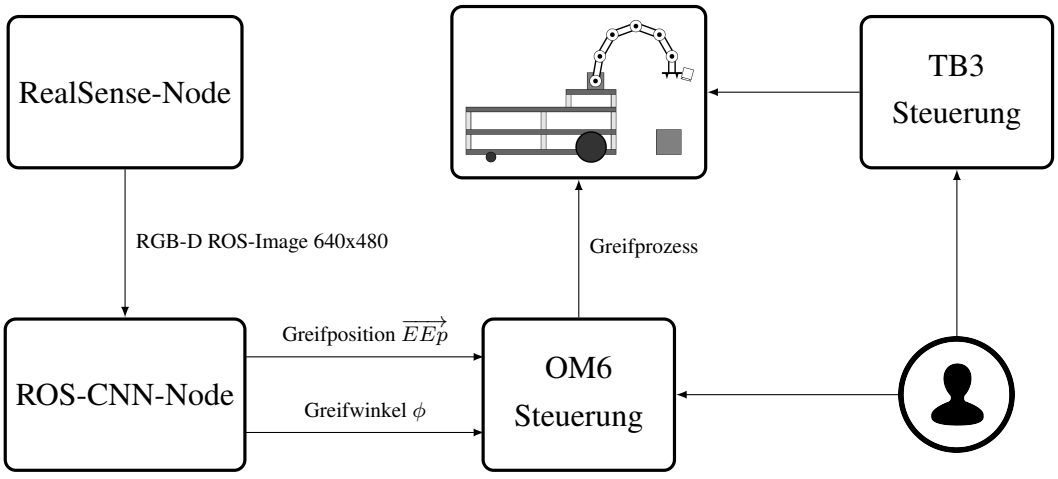

Abbildung 4.1.: Steuerungsablauf - von der Greifposendetektion und bis zur Ansteuerung

Zunächst wird das empfangene Tiefenbild vom ROS-CNN-Node verarbeitet. Dabei wird die Greifposition, der Greifwinkel und die Greifweite anhand des Tiefenbildes generiert. Die Greifposition wird anschließend von Bildkoordinaten $\boldsymbol{s}$ zu Weltkoordinaten $\boldsymbol{p}$ (in Zentimetern) transformiert. Anschließend wird der Relativabstand $\overrightarrow{EEp}$ zum Roboterendeffektor berechnet. Dieser wird zusammen mit dem Greifwinkel ϕ an die Robotersteuerung des OM6 gesendet, welcher anhand der erhaltenen Daten die Greifpose am Objekt ansteuert und dieses dann greift. Hier sei angemerkt, dass dieser Prozess nur als offene Prozesskette abgearbeitet wird und keine Informationen zurückgeführt werden. Die Robotersteuerung erhält keine Benachrichtigungen über den Erfolg oder Misserfolg des Greifprozesses. Dafür müssten zusätzliche Sensoren am Robotermanipulator angebracht werden, welche den Greifprozess überwachen. Daher ist der voll automatisierte Greifprozess mit allen Prozessschritten noch nicht möglich. In der erarbeiteten Lösung muss ein Anwender die einzelnen Prozessschritte manuell einleiten. Um die Bedienung zu vereinfachen, wird in Abschnitt 4.3 eine grafische Benutzeroberfläche präsentiert.

Auch der TB3 muss vom Anwender manuell gesteuert werden. Dafür wird entweder die standardmäßig mitgelieferte Fernsteuerung oder die Fernsteuerung einer die Computertastatur über das `Teleop` ROS-Node verwendet. Für die Navigation des TB3 stellt ROBOTIS bereits einige ROS-Nodes zur Verfügung, wie z.B. das `turtlebot3_navigation`-Paket. Es ermöglicht das Kartographieren der Umgebung und die anschließende Vorgabe von Raumposition. Für die Orientierung nutzt der TB3 den eingebauten Lidar-Scanner. Alle nötigen Informationen dazu sind auf der Herstellerseite zu finden [14, Abschnitt 10].

4.2. Inbetriebnahme der Robotersteuerung mit Greifposendetektion

Dieser Abschnitt behandelt die Inbetriebnahme des Roboters und allen notwendigen ROS-Komponenten, welche zum Greifen von Objekten mittels CNN notwendig sind. Anders als in Abbildung 2.13 präsentiert, wird der OM6 jedoch nicht über den RPI angesteuert, sondern über eine direkte USB-Verbindung zwischen dem Arbeitsrechner und dem U2D2-Verbindungsmodul. Die Steuerungssoftware des OM6 kann leider nicht auf dem RPI in

Betrieb genommen werden, da diese von diversen Softwareelementen[1] abhängig ist, welche auf dem RPI nicht unterstützt werden. Daher ist der Roboter in dieser Arbeit an den Arbeitsrechner gebunden und nur in dessen Reichweite mobil.

Folgende Schritte müssen für die Inbetriebnahme des Roboters durchgeführt werden:

1. ROS-Masters auf dem Arbeitsrechner starten:

   ```
   $ roscore
   ```

2. Verbindungsaufbau zwischen dem Rechner und dem RPI des TB3 über das Ubuntu-Terminal:

   ```
   $ ssh pi@IP-ADRESSE-DES-TB3
   ```

   Hierfür muss beim Rapsberry Pi die SSH-Verbindung aktiviert worden sein. Das Passwort zum einloggen lautet `turtlebot`. Anschließend muss die *RealSense*-Kamera über die SSH-Verbindung gestartet werden:

   ```
   $ roslaunch realsense2_camera rs_camera.launch
   ```

3. Letztlich sind folgende ROS-Nodes auf dem Arbeitsrechner zu starten:

   ```
   $ roslaunch open_manipulator_6dof_controller...
   open_manipulator_6dof_controller.launch
   ```

   ```
   $ roslaunch open_manipulator_6dof_control_gui...
   open_manipulator_6dof_control_gui.launch
   ```

   Außerdem kann das ROS-Programm `rqt_image_view` benutzt werden, um das Ausgabebild der Greifposengenerierung anzuzeigen.

   Zum Starten des CNN's zur Greifposengenerierung muss folgender Befehl ausgeführt werden:

   ```
   $ rosrun cnn_grasping cnn_grasping_node.py
   ```

   Alternativ kann auch die folgende Launch-Datei aufgerufen werden, welche sowohl den `OpenManipulator 6DOF Controller`, die `OpenManipulator 6DOF Control GUI` und das `CNN Grasping-Node` initialisiert:

[1] Vom RPI nicht unterstützt: Teilmodule von *MoveIt!* und davon abhängige Komponenten wie z.B. *Gazebo*

```
$ roslaunch cnn_grasping cnn_grasping.launch
```

Der Roboter ist nun Startbereit und kann zum Greifen von Objekten verwendet werden. In Abschnitt 4.3 wird die Bedieneroberfläche zum Steuern des OM6 und zum Greifen von Objekten präsentiert.

4.3. Bedieneroberfläche zur Steuerung des OM6

Für die Steuerung des OM6 wird vom Hersteller ROBOTIS ein entsprechendes ROS-Node mit grafischer Benutzeroberfläche bereitgestellt. Diese beinhaltet umfangreiche Funktionalitäten und ermöglicht die Steuerung des OM6 über die ROS-Schnittstelle. Um diese Bedienungsmöglichkeiten weiterhin gewährleisten zu können, wird diese Software um die Funktionen zum Greifen von Objekten mittels des CNN ergänzt.

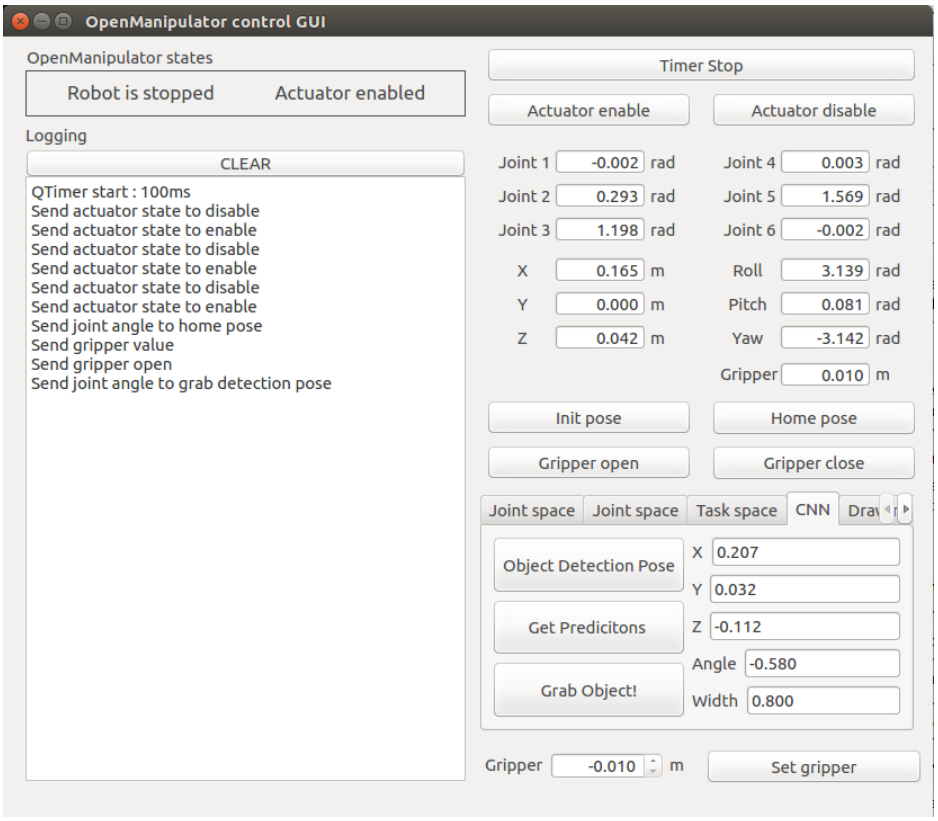

Abbildung 4.2.: Klassendiagramm der Steuerungssoftware des OM6

Für den Greifprozess mittels CNN ist eine zusätzliche Karteikarte mit dem Namen "CNN" in die GUI implementiert worden (siehe Abbildung 4.2). Insgesamt beinhaltet diese fünf Anzeigen für Greifpose, bestehend aus der Greifposition bzgl. des Basiskoordinatensystems der Robotermanipulators, dem Greifwinkel und der Greifweite. Zudem können Teilschritte des Greifprozesses über drei Tasten initialisiert werden. Zunächst muss der Robotermanipulator mit der *Object Detection Pose*-Taste in die Greifposendetektionsstellung gefahren werden. Hierbei wird die `On_btn_detection_pose_clicked`-Funktion ausgeführt, welche die Kamera am OM6 in eine parallele Position über dem zu greifenden Objekt bewegt. Mit der *Get Predictions*-Taste werden dann die generierten Greifposeninformationen vom ROS-CNN-Node abgerufen und in den rechts liegenden Feldern angezeigt. Der Anwender kann die Werte vor dem Anfahren noch editieren. Zum Greifen des Objektes an der Greifpose kann letztlich die *Grab Object*-Taste betätigt werden. Der Roboter stellt hierbei zunächst den Greifwinkel am Roboterendeffektor ein. Anschließend führt er eine PTP-Bewegung zum Objekt aus und befindet sich direkt über der Greifposition. Schließlich wird eine direkte Linearbewegung in negativer Z-Richtung ausgeführt. Über die *Gripper*-Tasten kann der Anwender das Objekt nun greifen lassen.

Entwickelt wurde die grafische Benutzeroberfläche mit dem Qt-Framework. Dessen Klassendiagramm ist im Anhang A.8 zu finden. Es stellt die Attribute und Methoden der Klassen, sowie deren Beziehung untereinander dar. Insgesamt besteht die Software aus der `MainWindow`-Klasse und der `QNode`-Klasse. Die `MainWindow`-Klasse stellt das Hauptfenster der GUI dar und ist für die Darstellung und die Verwaltung der empfangenen Signale und Slots zuständig. Bei der Betätigung eines GUI-Elements wird ein Signal ausgesendet, welches vom entsprechend definierten Slot der `MainWindow`-Klasse registriert wird. Dadurch werden die in den Slots hinterlegten Callback-Funktionen ausgeführt.

Die Ansteuerungsmethoden des OM6 sind als Callback-Funktionen in der `QNode`-Klasse definiert. Hierin sind alle notwendigen Funktionen zur Bedienung des OM6 implementiert. In dieser Arbeit werden die im Vorfeld vom Hersteller implementierten Programmelemente nicht weiter behandelt. Der Fokus wird vor allem auf die notwendigen Modifikationen für den Greifprozess gesetzt. Dies sind folgende Attribute und Methoden der beiden Klassen:

1. MainWindow-Klasse
    - Attribute:

  - `std_msgs::Float32MultiArray cnn_predictions`:
    Variable zum Zwischenspeichern des empfangenen Greifposenvektors vom ROS-CNN-Node.

- Methoden:

  - `void on_btn_detection_pose_clicked()`:
    Der Robotermanipulator fährt in die definierte Greifposendetektionsstellung. Die Kamera befindet sich über dem Objekt und die Greifposengenerierung kann gestartet werden.

  - `void on_btn_cnn_prediction_clicked()`:
    Die publizierten Greifposenwerte des ROS-CNN-Nodes werden ausgelesen und in das Attribut `cnn_predictions` abgespeichert. Außerdem werden diese Werte auf den Textfeldern im CNN-Raster angezeigt. Der Anwender kann diese Werte vor dem Anfahren der Greifpose editieren.

  - `void on_btn_grab_object_clicked()`:
    Der Robotermanipulator fährt die Greifpose am Objekt an.

2. QNode-Klasse

- Attribute:

  - `ros::Subscriber cnn_grasp_subscriber_`:
    ROS-Subscriber empfängt die Greifposenwerte vom ROS-Topic `/ggcnn/out/command` und löst die Callback-Funktion `setGraspValues` aus.

  - `std_msgs::Float32MultiArray cnn_predicitons`:
    Variable zum Zwischenspeichern des empfangenen Greifposenwerte vom ROS-CNN-Node.

- Methoden:

  - `void setGraspValues(const std_msgs::Float32MultiArray::ConstPtr &msg)`:
    Dies ist eine Callback-Funktion, welche von `cnn_grasp_subscriber` ausgelöst wird. `msg` enthält die vom ROS-CNN-Node empfangenen Greifposenwerte.

Diese werden hier lokal in die Variable `cnn_predictions` zwischengespeichert.

- `std_msgs::Float32MultiArray getGraspValues():`
  Gibt die Variable `cnn_predictions` zurück.

Die entsprechenden Programmabschnitte zu diesen Ergänzungen sind im Anhang A.7 zu finden. Das vollständige ROS-Node mit GUI ist im beigefügten Medium enthalten.

4.4. Praxistest des trainierten CNNs

In diesem Abschnitt wird die Greifposengenerierung und das anschließende Greifen mittels Robotermanipulator praktisch geprüft. Dafür werden die Objekte aus Abbildung 4.3 verwendet. Bei der Auswahl der Testobjekte wurde darauf geachtet, dass diese so unterschiedlich wie möglich sind. Damit wird Versucht, ein möglichst breites Spektrum an Anwendungsfällen abzudecken. Die Testergebnisse sind in Tabelle 4.1 aufgelistet und kommentiert. Dafür wird die *RealSense*-Kamera über den USB 2.0-Anschluss des RPI 3B+ betrieben.

Abbildung 4.3.: Diverse Objekte zum Testen des Greifprozesses

Insgesamt kann festgestellt werden, dass die Greifposen bei allen Testobjekten grundsätzlich korrekt generiert werden. Die meisten Misserfolge lassen sich vor allem auf den eingeschränkten Arbeitsraum des OM6 Robotermanipulators und der geringen Greifweite des

montierten Parallelgreifers zurückführen. Der Greiferfolg hängt auch stark von der Größe des Objektes ab. Hohe Objekte können teilweise zur Kollision zwischen dem Greifer und dem Objekt führen. Bei kleinen Objekten wird die Greifpose nicht immer zuverlässig generiert, weshalb eine Vorauswahl der korrekten Greifpose durch den Anwender unerlässlich wird. Dieser Aspekt verhindert im Endeffekt den voll automatisierten Greifprozess. Die optimale Breite des zu greifenden Objektes liegt bei ca. $3cm$ und die Höhe bei $4cm$. Die Länge kann beliebig gewählt werden. Bei reflektierenden Oberflächen hängt die Greifposengenerierung stark vom Blickwinkel und der Position bzgl. der Kamera ab. Dennoch können Greifposen teilweise erkannt und angefahren werden.

Der Betrieb der *RealSense*-Kamera über den USB 2.0-Anschluss des RPI 3B+ schränkt die Funktionalitäten der Kamera stark ein. Hierbei verringert sich nicht nur die verfügbare Bildrate, sondern es fallen auch diverse Filter und andere interne Bildbearbeitungsalgorithmen weg. Entscheidend ist jedoch, dass der implementierte Texturenprojektor nicht mehr funktioniert. Dadurch vermehren sich Leerstellen ohne Messung, das Rauschen nimmt zu und das Tiefenbild verschlechtert sich insgesamt stark. Ein nachträglicher Test mit Anschluss der *RealSense*-Kamera direkt an den Arbeitsrechner zeigt, dass sich die Greifposengenerierung stabilisiert und signifikant verbessert. Dabei sind bei jedem Testobjekt Verbesserungen von mindestens 20% festzustellen. Der Betrieb der *RealSense*-Kamera über einen RPI 3B+ wird daher an dieser Stelle nicht empfohlen.

Tabelle 4.1.: Testergebnisse des Greifprozesses

Testobjekt	Greifwahrscheinlichkeit	Kommentar
Fahrradgriff	100%	Das Objekt ist mit $3.5cm$ etwas zu breit für den Greifer des OM6. Ansonsten wird das Objekte problemlos detektiert und gegriffen.
Adapteranschluss	100%	Die Prozentangabe gilt nur für einen seitlich stehenden Adapteranschluss. Flach liegend ist es mit $3.5cm$ zu breit für den Greifer des OM6.
Gabel	70%	Die metallische Oberfläche der Gabel reflektiert stark. Dadurch gelingt die Greifposengenerierung nicht immer.
Kabelbund	90%	Zuverlässige Greifposengenerierung trotz ungleichmäßiger Form des Kabelbunds.
Klebeband	80%	Die Greifpose wird zuverlässig generiert. Für den Roboter ist das Klebeband mit $5cm$ etwas zu hoch, weshalb der Greifprozess nicht immer erfolgreich ist.
Holzblock	90%	Die Greifpose wird sehr zuverlässig generiert. Für das Greifen ist dieser mit $3.5cm$ etwas zu breit.
Stift	80%	Das CNN generiert stets Greifposen an der Stiftspitze. Der Stift etwas zu dünn und zu flach, weshalb die Greifposengenerierung teilweise unzuverlässig wird.
Kleinteile (Mutter, Würfel, Kegel)	30%	Die Greifpose wird bei allen Kleinteilen zuverlässig detektiert. Der Greifversuch geringe Größe erschwert sich der Greifprozess jedoch.

4.5. Objektklassifizierung mit dem YOLOv3-CNN

Für die Objektklassifizierung wird in dieser Arbeit das YOLOv3-CNN [1] genutzt. Dieses CNN wurde mit der COCO-Datenbank [8] trainiert und ermöglicht dem CNN die Klassifizierung von insgesamt 80 Objekttypen, wie z.B. Personen, Verkehrsmittel, Tiere und Haushaltsobjekte. Am Beispiel einer Gabel soll verdeutlicht werden, dass das YOLOv3-CNN auch für das Greifen von Objekten mittels Robotermanipulator geeignet ist. Wie in Abbildung 4.4 zu sehen ist, werden klassifizierte Objekte im Farbbild mit einer Bounding-Box markiert und mit der Klassenbezeichnung versehen. Es können beliebig viele Objekte in einem Farbbild detektiert und klassifiziert werden.

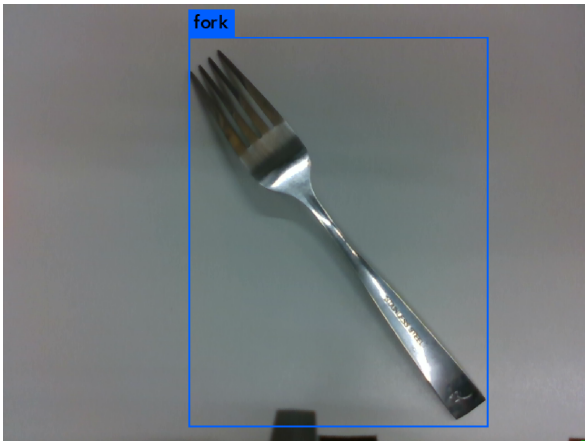

Abbildung 4.4.: Klassifizierung einer Gabel mittels YOLOv3

Für die Initialisierung des YOLOv3 ROS-Nodes wird folgender Befehl verwendet:

```
roslaunch darknet_ros yolo_v3.launch
```

Dieses ROS-Node nimmt das Farbbild der *RealSense*-Kamera entgegen und veröffentlicht alle Informationen bzgl. der detektierten Objekte über entsprechende ROS-Topics. Diese können z.B. in den entwickelten Greifalgorithmus implementiert werden und so Objekte auf dem Spielfeld entsprechenden Behältern zuweisen. Es ist außerdem möglich, das YOLOv3-CNN mit benutzerdefinierten Objekten zu trainieren und dessen Ausgaben somit an die Anwendung anzupassen. Alle weiteren Informationen können auf der GitHub-Seite der Entwickler nachgelesen werden [1].

5. Fazit

5.1. Zusammenfassung

In dieser Arbeit ist ein mobiler Roboter zum Greifen von unbekannten Objekten mittels eines trainierten CNN's entwickelt worden. Hierfür wurde der *TurtleBot3 Waffle Pi* als mobile Basis des Roboters und der *OpenManipulator* mit sechs Freiheitsgraden als Robotermanipulator zum Greifen der Objekte verwendet. Der Robotermanipulator ist zusätzlich mit einer *Intel RealSense D435* Tiefenbildkamera ausgestattet worden und erkennt damit die zu greifenden Objekte. Für die Greifposengenerierung ist ein CNN mit der *Cornell*-Datenbank trainiert und bzgl. seiner Leistung evaluiert worden. Dabei werden die Tiefenbilder der Objekte aus der Vogelperspektive aufgenommen und an das CNN übergeben, welches die Greifposition bzgl. des Roboterendeffektors, den Greifwinkel und die Greifweite ausgibt. Diese Werte werden dazu genutzt, um das Objekt anzufahren und es dann zu Greifen.

Die Greifposengenerierung mittels CNN ist zuverlässig und stabil. Auch unbekannten Objekten wird die Greifpose grundsätzlich korrekt bestimmt. Insgesamt kann von einer korrekten Greifposengenerierung von 80% bis 90% gerechnet werden. Diese Zahlen gelten jedoch nur dann, wenn die *RealSense*-Kamera über den Arbeitsrechner betrieben wird. Diese Leistung kann jedoch nicht im mobilen Betrieb erzielt werden, da der RPI 3B+ die Funktionalitäten der Tiefenbildkamera stark einschränkt. Zum einen liegt das an der geringen Rechenleistung des RPI, vor allem aber daran, dass nur ein USB 2.0-Anschluss zur Verfügung steht. Um sämtliche Funktionen der *RealSense*-Kamera nutzen zu können, ist mindestens ein USB 3.0-Anschluss nötig. Dadurch verschlechtert sich die Tiefenbildaufnahme signifikant und entsprechend auch die Greifposengenerierung.

Für die Steuerung des OM6 Robotermanipulators stellt der Hersteller *ROBOTIS* entsprechende ROS-Pakete zur Verfügung. Diese ermöglichen dessen Steuerung über eine graphi-

sche Bedieneroberfläche. Für das Greifen von Objekten mittels CNN ist diese GUI im Rahmen der Arbeit mit zusätzlichen Bedienungsmöglichkeiten erweitert worden. Über einfache Tastenbetätigungen kann der Benutzer somit den Robotermanipulator steuern und Objekte greifen lassen. Auch hier ist der mobile Betrieb des Roboters nur eingeschränkt möglich. Der RPI unterstützt die Steuerungssoftware des OM6 nicht. Der OM6 muss daher über ein USB-Kabel an einen Arbeitsrechner angeschlossen und bedient werden. Deshalb ist dessen Mobilität stark eingeschränkt.

Die Navigation und die Objektlokalisierung auf dem Spielfeld ist in dieser Arbeit nicht thematisiert worden. Das liegt vor allem an den stark eingeschränkten mobilen Betrieb des Roboters. Daher sind diese Aspekte der Arbeit nicht weiter behandelt worden. Dafür wurde der Fokus mehr auf die Greifposengenerierung und den Greifprozess gelegt. Dieser ist erfolgreich umgesetzt worden. Insgesamt besitzt der entwickelte Roboter noch großes Potential und kann als Basis für weitere Entwicklungsarbeiten dienen. Mögliche Ansätze und Verbesserungen werden im nächsten Abschnitt präsentiert.

5.2. Ausblick

Auch wenn das Greifen von unbekannten Objekten mittels CNN und Robotermanipulator funktionieren, existieren dennoch viele Optimierungs-und Ergänzungsmöglichkeiten bzgl. dieser Arbeit. Diese werden im folgenden aufgelistet und erläutert.

- Verwendung eines SBC's mit x86-Architektur

  Die begrenzte Mobilität des entwickelten Roboters stellt das größte Problem dar. Die Einschränkungen werden vor allem durch den RPI 3B+ des TB3's verursacht. Als zentrale Hardwarekomponente verbindet es den TB3, den OM6, die Kamera und den Arbeitsrechner miteinander. Deshalb ist es essentiell, dass diese Hardwarekomponente über genügend Rechenleistung verfügt und die ihm übertragenen Aufgaben zuverlässig erledigt. Leider eignet sich der RPI 3B+ sich für dieses Projekt nicht. Dies ist vor allem auf die geringe Rechenleistung und der ARM-Architektur der CPU zurückzuführen. Hinzu kommt auch, dass die benötigten ROS-Pakete nur für Systeme auf Basis von x86-Architekturen unterstützt werden. Im Rahmen dieser Arbeit wurde deshalb versucht, die nötigen ROS-Pakete über *CMake* für die ARM-Architektur zu

erstellen. Durch diverse Softwareabhängigkeiten ist dieser Versuch nicht erfolgreich gewesen. Im Idealfall wird der RPI 3B+ durch einen SBC mit x86-Architektur ersetzt. Hierfür existieren einige kostengünstige Alternativen, wie z.B. der *Atomic Pi* oder der *LattePanda Alpha*. Diese SBC's unterstützen die Desktop-Version von Ubuntu und alle verfügbaren ROS-Pakete. Zudem verfügen sie über genügend Rechenleistung. Für zukünftige Entwicklungsarbeiten wird ein Austausch der Steuerungsplatinen eindringlich empfohlen.

- Navigation und Objektlokalisation auf dem Spielfeld

  Die Navigation und die Objektlokalisation auf dem Spielfeld sind in dieser Arbeit nicht behandelt worden. Grund dafür ist die eingeschränkte Mobilität des entwickelten Roboters. Jedoch ist eine sensorbasierte Objektlokalisation durchaus umsetzbar. Hierfür kann z.B. eine Deckenkamera genutzt werden, welche aus der Vogelperspektive das Spielfeld überwacht, Objektpositionen ermittelt und an den mobilen Roboter sendet. Zudem können die Kameras am Roboter selbst für die Objektlokalisation genutzt werden. Bei der Konstruktion des Roboters wurde explizit darauf geachtet, das der mitgelieferte Lidar-Scanner seine Funktion beibehält. Deshalb kann dieser für die Navigation auf dem Spielfeld genutzt werden. Er ermöglicht das Kartographieren des Spielfeldes über diverse ROS-Pakete. Auf dieser Basis kann der Roboter dann bestimmte Punkte auf dem Spielfeld anfahren und Objekte aufsammeln.

- Training des CNN's mit der *Jacquard*-Datenbank

  Die *Jacquard*-Datenbank[3] kann, ähnlich wie die *Cornell*-Datenbank, für das Trainieren eines CNN's zum Greifen von Objekten genutzt werden. Jedoch basierte diese nicht auf realen Aufnahmen, sondern auf synthetischen erzeugten Objekten. Dadurch verfügt diese aber über weitaus mehr Datensamples als die *Cornell*-Datenbank. Eventuell kann die Greifposengenerierung mit einem CNN auf Basis dieser Datenbank noch verbessert werden.

- Greifprozess als geschlossene Regelschleife implementieren

  In dieser Arbeit generiert das CNN die Greifpose am Objekt einmalig aus der Vogelperspektive heraus. Dabei befindet sich der Robotermanipulator in einer definierten Stellung und sind die Position und die Orientierung der Kamera bekannt. Die relative Greifpose bzgl. des Roboterendeffektors wird anschließend an die Robotersteuerung

gesendet. Aus dieser Ausgangsstellung heraus, fährt der Robotermanipulator das Objekt quasi "blind" an und greift es. Dabei werden keine Informationen zum Erfolg oder Misserfolg während des Greifprozesses zurückgesendet. Eine kontinuierliche Greifposengenerierung würde den Greiferfolg steigern. Der Robotermanipulator könnte somit seine Bewegung stets an das Objekt anpassen und selbst dessen Verschiebungen berücksichtigen. Außerdem wäre der Greifprozess nicht mehr so stark von der Kamerakalibrierung abhängig.

- Auswahl eines neuen Greifers

  Beim Greifen von Objekten mit dem OM6 ist aufgefallen, dass der montierte Greifer eine zu geringe Greifweite besitzt. Oftmals scheitert der Greifversuch daran, dass das zu greifende Objekt zu breit ist. Ein Winkelgreifer würde den Greiferfolg stark verbessern. Außerdem könnte somit die Greifweite mitberücksichtigt werden.

Literaturverzeichnis

[1] BJELONIC, Marko: *YOLO ROS: Real-Time Object Detection for ROS*. 2016–2018. – https://github.com/leggedrobotics/darknet_ros

[2] CHOLLET, François: *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP Verlags GmbH & Company KG, 2018 (mitp Professional). – ISBN 9783958458406

[3] DEPIERRE, Amaury ; DELLANDRÉA, Emmanuel ; CHEN, Liming: Jacquard: A Large Scale Dataset for Robotic Grasp Detection. In: *CoRR* abs/1803.11469 (2018). – URL http://arxiv.org/abs/1803.11469

[4] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[5] INTEL: *Depth from Stereo*. 2018. – https://github.com/IntelRealSense/librealsense/blob/master/doc/depth-from-stereo.md

[6] KRIZHEVSKY, Alex ; SUTSKEVER, Ilya ; HINTON, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: PEREIRA, F. (Hrsg.) ; BURGES, C. J. C. (Hrsg.) ; BOTTOU, L. (Hrsg.) ; WEINBERGER, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, S. 1097–1105

[7] LENZ, Ian ; LEE, Honglak ; SAXENA, Ashutosh: *Deep Learning for Detecting Robotic Grasps*. 2013

[8] LIN, Tsung-Yi ; MAIRE, Michael ; BELONGIE, Serge J. ; BOURDEV, Lubomir D. ; GIRSHICK, Ross B. ; HAYS, James ; PERONA, Pietro ; RAMANAN, Deva ; DOLLÁR,

Piotr ; ZITNICK, C. L.: Microsoft COCO: Common Objects in Context. In: *CoRR* abs/1405.0312 (2014). – http://arxiv.org/abs/1405.0312

[9] MAHLER, Jeffrey ; MATL, Matthew ; SATISH, Vishal ; DANIELCZUK, Michael ; DEROSE, Bill ; MCKINLEY, Stephen ; GOLDBERG, Ken: Learning ambidextrous robot grasping policies. In: *Science Robotics* 4 (2019), Nr. 26, S. eaau4984

[10] MCCULLOCH, W. S. ; PITTS, W.: A Logical Calculus of the Idea Immanent in Nervous Activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 115–133

[11] MORRISON, Douglas ; CORKE, Peter ; LEITNER, Jürgen: Closing the Loop for Robotic Grasping: A Real-time, Generative Grasp Synthesis Approach. In: *CoRR* abs/1804.05172 (2018). – URL http://arxiv.org/abs/1804.05172

[12] ROBOTIS: *OpenMANIPULATOR - Manual*. 2019. – http://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/

[13] ROBOTIS: *OpenManipulator-and-Friends*. 2019. – https://github.com/zang09/open_manipulator_6dof_application

[14] ROBOTIS: *Turtlebot3 - Manual*. 2019. – http://emanual.robotis.com/docs/en/platform/turtlebot3/

[15] ROBOTIS: *U2D2*. 2019. – http://emanual.robotis.com/docs/en/parts/interface/u2d2/

[16] ROBOTIS: *XM430-W210-T / XM430-W210-R*. 2019. – http://emanual.robotis.com/docs/en/dxl/x/xm430-w210/

[17] ROBOTIS: *XM430-W350-T / XM430-W350-R*. 2019. – http://emanual.robotis.com/docs/en/dxl/x/xm430-w350/

[18] ROSENBLATT, F.: The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. In: *Psychological Review* (1958), S. 65–386

[19] UNIVERSITY, Cornell: *Learning to Grasp Novel Objects*. 2009. – http://pr.cs.cornell.edu/deepgrasping/

[20] WIDROW, B. ; HOFF, M. E.: Adaptive Switching Circuits. In: *1960 IRE WESCON Convention Record* 4 (1960), S. 96–104

[21] YOONSEOK PYO, RyuWoon Jung TaeHoon L.: *ROS Robot Programming - A Handbook is written by TurtleBot3 Developers*. ROBOTIS Co.,Ltd., 2017. – http://www.robotis.com/

A. Anhang

A.1. Hardwarespezifikationen des TB3

Tabelle A.1.: Turtlebot3 Waffle Pi: Hardware Spezifikationen (Quelle: [14])

Items	Specifications
Maximum translational velocity	0.26 m/s
Maximum rotational velocity	1.82 rad/s (104.27 deg/s)
Maximum payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1.8kg
Threshold of climbing	10 mm or lower
Expected operating time	2h
Expected charging time	2h 30m
SBC (Single Board Computers)	Raspberry Pi 3 Model B and B+
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	RC-100B + BT-410 Set (Bluetooth 4, BLE)
Actuator	Dynamixel XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01
Camera	Raspberry Pi Camera Module v2.1
IMU	Gyroscope 3 Axis, Accelerometer 3 Axis, Magnetometer 3 Axis
Power connectors	3.3V / 800mA, 5V / 4A, 12V / 1A
Expansion pins	GPIO 18 pins, Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Dynamixel ports	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences
Programmable LEDs	User LED x 4
Status LEDs	Board status LED x 1, Arduino LED x 1, Power LED x 1
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
PC connection	USB
Firmware upgrade	via USB / via JTAG
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max, Output : 12V DC, 5A

A.2. Dynamixel XM-430 Spezifikationen

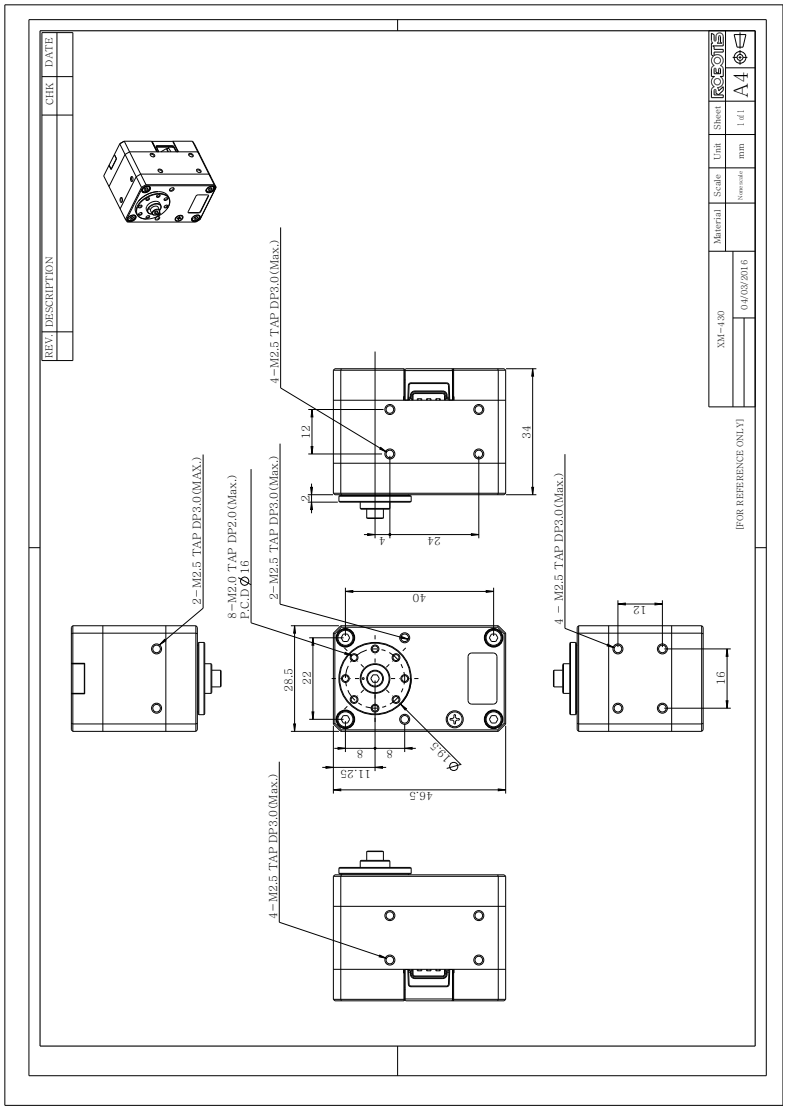

Abbildung A.1.: Maßangaben des Dynamixel XM430

Tabelle A.2.: Dynamixel XM430-W350 Spezifikationen (Quelle: [17])

Items	**Specifications**
MCU	ARM CORTEX-M3 (72 [MHz], 32Bit)
Position Sensor	Contactless absolute encoder (12Bit, 360 [°]) Maker : ams(www.ams.com), Part No : AS5045
Motor	Coreless
Baud Rate	9,600 [bps] ∼ 4.5 [Mbps]
Control Algorithm	PID control
Resolution	4096 [pulse/rev]
Backlash	15 [arcmin] (0.25 [°])
Operating Modes	Current Control Mode Velcoity Control Mode Position Control Mode (0 360 [°]) Extended Position Control Mode (Multi-turn) Current-based Position Control Mode PWM Control Mode (Voltage Control Mode)
Weight	82 [g]
Dimensions (W x H x D)	28.5 x 46.5 x 34 [mm]
Gear Ratio	353.5 : 1
Stall Torque	3.8 [N.m] (at 11.1 [V], 2.1 [A]) **4.1 [Nm] (at 12.0 [V], 2.3 [A])** 4.8 [Nm] (at 14.8 [V], 2.7 [A])
No Load Speed	43 [rev/min] (at 11.1 [V]) **46 [rev/min] (at 12.0 [V])** 57 [rev/min] (at 14.8 [V])
Radial Load	40 [N] (10 [mm] away from the horn)

Fortsetzung auf der nächsten Seite

Tabelle A.2 – *Fortsetzung*

Axial Load	20 [N]
Operating Temperature	-5 $\sim$ +80 [°C]
Input Voltage	10.0 $\sim$ 14.8 [V] **(Recommended : 12.0 [V])**
Command Signal	Digital Packet
Protocol Type	TTL Half Duplex Asynchronous Serial Communication with 8bit, 1stop, No Parity RS485 Asynchronous Serial Communication with 8bit, 1stop, Parity
Physical Connection	RS485 / TTL Multidrop Bus
ID	253 ID's (0 $\sim$ 252)
Feedback	Position, Velocity, Current, Realtime tick, Trajectory, Temperature, Input Voltage, etc
Part Material	Full Metal Gear Metal (Front, Middle), Engineering Plastic (Back)
Standby Current	40 [mA]

Für den **Dynamixel XM430-W210** ist die Tabelle A.2 in allen Fällen gültig, außer für das Kippmoment und für die Leerlaufdrehzahl. Diese Werte sind ergänzend in Tabelle A.3 angegeben.

Tabelle A.3.: Dynamixel XM430-W210 Spezifikationen (Quelle: [16])

Items	Specifications
Stall Torque	2.7 [N.m] (at 11.1 [V], 2.1 [A]) **3.0 [Nm] (at 12.0 [V], 2.3 [A])** 3.7 [Nm] (at 14.8 [V], 2.7 [A])
No Load Speed	70 [rev/min] (at 11.1 [V]) **77 [rev/min] (at 12.0 [V])** 95 [rev/min] (at 14.8 [V])

A.3. Inbetriebnahme der Intel RealSense D435 auf dem Raspberry Pi 3B+

Im folgenden wird die Installation der *RealSense*-Software auf dem Raspberry Pi 3B+ mit dem Raspbian Stretch Betriebssystem erläutert. Alle notwendigen Schritte sind können in der angegeben Reihenfolge ausgeführt werden. Dieser Prozess kann einige Stunden beanspruchen. Die Installation ist auch auf der GitHub-Seite von *Intel* aufgelistet (siehe https://github.com/IntelRealSense/librealsense/blob/master/doc/installation_raspbian.md). Jedoch sind bzgl. der Einbindung in ROS noch weitere Installationen notwendig. Diese werden ebenfalls aufgelistet.

1. Versionen prüfen:

```
$ uname -a
Linux raspberrypi 4.14.34-v7+

$ sudo apt update;sudo apt upgrade
$ sudo reboot
$ uname -a

$ gcc -v
gcc version 6.3.0 20170516 (Raspbian 6.3.0-18+rpi1+deb9u1)

$ cmake --version
cmake version 3.7.2
```

2. Swap-Speicher erhöhen:

```
$ sudo nano /etc/dphys-swapfile
CONF_SWAPSIZE=2048

$ sudo /etc/init.d/dphys-swapfile restart swapon -s
```

3. Installation der notwendigen Pakete:

```
$ sudo apt-get install -y libdrm-amdgpu1 libdrm-amdgpu1-dbg ...
libdrm-dev libdrm-exynos1 libdrm-exynos1-dbg libdrm-freedreno1 ...
libdrm-freedreno1-dbg libdrm-nouveau2 libdrm-nouveau2-dbg ...
libdrm-omap1 libdrm-omap1-dbg libdrm-radeon1 libdrm-radeon1-dbg ...
libdrm-tegra0 libdrm-tegra0-dbg libdrm2 libdrm2-dbg

$ sudo apt-get install -y libglu1-mesa libglu1-mesa-dev ...
glusterfs-common libglu1-mesa libglu1-mesa-dev libglui-dev libglui2c2

$ sudo apt-get install -y libglu1-mesa libglu1-mesa-dev mesa-utils ...
mesa-utils-extra xorg-dev libgtk-3-dev libusb-1.0-0-dev
```

4. Udev-Rules aktualisieren:

```
$ cd ~
$ git clone https://github.com/IntelRealSense/librealsense.git
$ cd librealsense
$ sudo cp config/99-realsense-libusb.rules /etc/udev/rules.d/
$ sudo udevadm control --reload-rules && udevadm trigger
```

5. CMake-Version updaten:

```
$ cd ~
$ wget https://cmake.org/files/v3.11/cmake-3.11.4.tar.gz
$ tar -zxvf cmake-3.11.4.tar.gz;rm cmake-3.11.4.tar.gz
$ cd cmake-3.11.4
$ ./configure --prefix=/home/pi/cmake-3.11.4
$ make -j1
$ sudo make install
$ export PATH=/home/pi/cmake-3.11.4/bin:$PATH
$ source ~/.bashrc
$ cmake --version
cmake version 3.11.4
```

6. Systempfad setzen:

```
$ nano ~/.bashrc
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH

$ source ~/.bashrc
```

7. Protobuf installieren:

```
$ cd ~
$ git clone --depth=1 -b v3.5.1 https://github.com/google/protobuf.git
$ cd protobuf
$ ./autogen.sh
$ ./configure
$ make -j1
$ sudo make install
$ cd python
$ export LD_LIBRARY_PATH=../src/.libs
$ python3 setup.py build --cpp_implementation
$ python3 setup.py test --cpp_implementation
$ sudo python3 setup.py install --cpp_implementation
$ export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=cpp
$ export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION_VERSION=3
$ sudo ldconfig
$ protoc --version
```

8. TBB installieren:

```
$ cd ~
$ wget https://github.com/PINTO0309/TBBonARMv7/raw/master/ ...
libtbb-dev_2018U2_armhf.deb
$ sudo dpkg -i ~/libtbb-dev_2018U2_armhf.deb
$ sudo ldconfig
$ rm libtbb-dev_2018U2_armhf.deb
```

9. OpenCV installieren:

```
Remove previous version
$ sudo apt autoremove libopencv3

Install
$ wget https://github.com/mt08xx/files/raw/master/opencv-rpi/ ...
libopencv3_3.4.3-20180907.1_armhf.deb
```

```
$ sudo apt install -y ./libopencv3_3.4.3-20180907.1_armhf.deb
$ sudo ldconfig
```

10. RealSense SDK installieren:

```
$ cd ~/librealsense
$ mkdir  build  && cd build
$ cmake .. -DBUILD_EXAMPLES=true -DCMAKE_BUILD_TYPE=Release  ...
-DFORCE_LIBUVC=true
$ make -j1
$ sudo make install
```

11. Pyrealsense2 installieren:

```
$ cd ~/librealsense/build

for python2
$ cmake .. -DBUILD_PYTHON_BINDINGS=bool:true ...
-DPYTHON_EXECUTABLE=$(which python)

for python3
$ cmake .. -DBUILD_PYTHON_BINDINGS=bool:true ...
-DPYTHON_EXECUTABLE=$(which python3)

$ make -j1
$ sudo make install

add python path
$ nano ~/.bashrc
export PYTHONPATH=$PYTHONPATH:/usr/local/lib

$ source ~/.bashrc
```

12. OpenGL aktivieren:

```
$ sudo apt-get install python-opengl
$ sudo -H pip3 install pyopengl
$ sudo -H pip3 install pyopengl_accelerate
$ sudo raspi-config
"7.Advanced Options" - "A7 GL Driver" - "G2 GL (Fake KMS)"
```

13. Raspberry Pi neu starten und den RealSense-Viewer öffnen:

```
$ sudo reboot
$ realsense-viewer
```

Der RealSense-Viewer startet, wenn die Installation erfolgreich war.

14. ROS-RealSense-Paket herunterladen und in das Catkin-Verzeichnis entpacken:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/IntelRealSense/realsense-ros.git
$ cd realsense-ros/
$ git checkout `git tag | sort -V | grep -P "^\d+\.\d+\.\d+" | tail -1`
$ cd ..
```

15. Ddynamic-reconfigure-Paket herunterladen:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/pal-robotics/ddynamic_reconfigure/ ...
tree/kinetic-devel.git
$ cd ..
```

16. Catkin-Verzeichnis erstellen und ROS-RealSense-Node starten:

```
$ catkin_init_workspace
$ cd ..
$ catkin_make clean
$ catkin_make -DCATKIN_ENABLE_TESTING=False -DCMAKE_BUILD_TYPE=Release
$ catkin_make install
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc

$ roscore
$ roslaunch realsense2_camera rs_camera.launch
```

A.4. RealSense-ROS Launch-Datei

```
<launch>
  <arg name="serial_no"           default=""/>
  <arg name="usb_port_id"         default=""/>
  <arg name="device_type"         default=""/>
  <arg name="json_file_path"      default=""/>
  <arg name="camera"              default="camera"/>
  <arg name="tf_prefix"           default="$(arg camera)"/>
  <arg name="external_manager"    default="false"/>
  <arg name="manager"             default="realsense2_camera_manager"/>

  <arg name="fisheye_width"       default="640"/>
  <arg name="fisheye_height"      default="480"/>
  <arg name="enable_fisheye"      default="true"/>

  <arg name="depth_width"         default="640"/>
  <arg name="depth_height"        default="480"/>
  <arg name="enable_depth"        default="true"/>

  <arg name="infra_width"        default="640"/>
  <arg name="infra_height"       default="480"/>
  <arg name="enable_infra1"       default="true"/>
  <arg name="enable_infra2"       default="true"/>

  <arg name="color_width"         default="640"/>
  <arg name="color_height"        default="480"/>
  <arg name="enable_color"        default="true"/>

  <arg name="fisheye_fps"         default="30"/>
  <arg name="depth_fps"           default="30"/>
  <arg name="infra_fps"           default="30"/>
  <arg name="color_fps"           default="30"/>
  <arg name="gyro_fps"            default="400"/>
  <arg name="accel_fps"           default="250"/>
  <arg name="enable_gyro"         default="true"/>
  <arg name="enable_accel"        default="true"/>

  <arg name="enable_pointcloud"         default="false"/>
  <arg name="pointcloud_texture_stream" default="RS2_STREAM_COLOR"/>
  <arg name="pointcloud_texture_index"  default="0"/>

  <arg name="enable_sync"               default="false"/>
  <arg name="align_depth"               default="false"/>

  <arg name="publish_tf"                default="false"/>
  <arg name="tf_publish_rate"           default="0"/>

  <arg name="filters"                   default="temporal"/>
  <arg name="clip_distance"             default="0.28"/>
```

```
<arg name="linear_accel_cov"          default="0.01"/>
<arg name="initial_reset"             default="false"/>
<arg name="unite_imu_method"          default=""/>
<arg name="topic_odom_in"             default="odom_in"/>
<arg name="calib_odom_file"           default=""/>
<arg name="publish_odom_tf"           default="false"/>
<arg name="allow_no_texture_points"   default="false"/>

<group ns="$(arg camera)">
  <include file="$(find realsense2_camera)/launch/includes/nodelet.launch.xml">
    <arg name="tf_prefix"               value="$(arg tf_prefix)"/>
    <arg name="external_manager"        value="$(arg external_manager)"/>
    <arg name="manager"                 value="$(arg manager)"/>
    <arg name="serial_no"               value="$(arg serial_no)"/>
    <arg name="usb_port_id"             value="$(arg usb_port_id)"/>
    <arg name="device_type"             value="$(arg device_type)"/>
    <arg name="json_file_path"          value="$(arg json_file_path)"/>

    <arg name="enable_pointcloud"       value="$(arg enable_pointcloud)"/>
    <arg name="pointcloud_texture_stream" value="$(arg pointcloud_texture_stream)"/>
    <arg name="pointcloud_texture_index"  value="$(arg pointcloud_texture_index)"/>
    <arg name="enable_sync"             value="$(arg enable_sync)"/>
    <arg name="align_depth"             value="$(arg align_depth)"/>

    <arg name="fisheye_width"           value="$(arg fisheye_width)"/>
    <arg name="fisheye_height"          value="$(arg fisheye_height)"/>
    <arg name="enable_fisheye"          value="$(arg enable_fisheye)"/>

    <arg name="depth_width"             value="$(arg depth_width)"/>
    <arg name="depth_height"            value="$(arg depth_height)"/>
    <arg name="enable_depth"            value="$(arg enable_depth)"/>

    <arg name="color_width"             value="$(arg color_width)"/>
    <arg name="color_height"            value="$(arg color_height)"/>
    <arg name="enable_color"            value="$(arg enable_color)"/>

    <arg name="infra_width"             value="$(arg infra_width)"/>
    <arg name="infra_height"            value="$(arg infra_height)"/>
    <arg name="enable_infra1"           value="$(arg enable_infra1)"/>
    <arg name="enable_infra2"           value="$(arg enable_infra2)"/>

    <arg name="fisheye_fps"             value="$(arg fisheye_fps)"/>
    <arg name="depth_fps"               value="$(arg depth_fps)"/>
    <arg name="infra_fps"               value="$(arg infra_fps)"/>
    <arg name="color_fps"               value="$(arg color_fps)"/>
    <arg name="gyro_fps"                value="$(arg gyro_fps)"/>
    <arg name="accel_fps"               value="$(arg accel_fps)"/>
    <arg name="enable_gyro"             value="$(arg enable_gyro)"/>
    <arg name="enable_accel"            value="$(arg enable_accel)"/>

    <arg name="publish_tf"              value="$(arg publish_tf)"/>
```

```
      <arg name="tf_publish_rate"          value="$(arg tf_publish_rate)"/>

      <arg name="filters"                  value="$(arg filters)"/>
      <arg name="clip_distance"            value="$(arg clip_distance)"/>
      <arg name="linear_accel_cov"         value="$(arg linear_accel_cov)"/>
      <arg name="initial_reset"            value="$(arg initial_reset)"/>
      <arg name="unite_imu_method"         value="$(arg unite_imu_method)"/>
      <arg name="topic_odom_in"            value="$(arg topic_odom_in)"/>
      <arg name="calib_odom_file"          value="$(arg calib_odom_file)"/>
      <arg name="publish_odom_tf"          value="$(arg publish_odom_tf)"/>
      <arg name="allow_no_texture_points"  value="$(arg allow_no_texture_points)"/>
    </include>
  </group>
</launch>
```

A.5. Programmcode zum ROS-CNN-Node

```
#!/usr/bin/python
#-*- coding:utf-8 -*-

import rospy                                            # ROS-Erweiterung für Python
import cv2                                              # OpenCv
from cv_bridge import CvBridge, CvBridgeError           # ROS-Image <-> OpenCV Bridge
import tensorflow as tf                                 # Tensorflow
from keras.models import load_model                     # Keras
import numpy as np                                      # Numpy
from numpy import unravel_index                         # Numpy - Unravel Index
import matplotlib.pyplot as plt                         # Matplotlib
from scipy import stats                                 # Scipy-Stats für den Mode-Filter
import scipy.ndimage as ndimage                         # Scipy-Ndimage
# Skimage-Draw: Zeichnen auf Bildern
from skimage.draw import circle, line
# Skimage-Feature: Max-Wert auf Heatmap finden
from skimage.feature import peak_local_max

# ROS-Message Types
from geometry_msgs.msg import PoseStamped
from sensor_msgs.msg import Image, CameraInfo
from std_msgs.msg import Float32MultiArray

"""
Initialsierung
"""
# ROS-Node initialisieren
rospy.init_node('ggcnn_detection')

# ROS-Publisher initialisieren
grasp_pub = rospy.Publisher('ggcnn/img/grasp', Image, queue_size=100)
grasp_cmd_pub = rospy.Publisher('ggcnn/out/command', Float32MultiArray, queue_size=100)

# Initialisierung der ROS-OpenCV-Bridge
bridge = CvBridge()

# Neuronales Netz über den Systempfad festlegen und laden
MODEL_FILE_PATH = '/home/shafaq/Schreibtisch/Masterthesis/ggcnn-RSS2018/data/networks/190807
    _1300__ggcnn_9_5_3__32_16_8/epoch_29_model.hdf5'
model = load_model(MODEL_FILE_PATH)

# Default Tensorflow-Graph festlegen, um diesen auch über Callbacks ansprechen zu können
graph = tf.get_default_graph()

# Intrinsische Kameraparameter vom RealSense-Node laden
camera_info_msg = rospy.wait_for_message('/camera/depth/camera_info', CameraInfo)
K = camera_info_msg.K

# Previous MaxPoint: Zum Zwischenspeichern der besten Greifpose der letzten Iteration
```

```
previous_max_points = np.array([150, 150])

# Array zum Speichern der Angle-Werte und Mittelwertbildung
angle_array = np.array([])
mean_angle = 0.0

"""
Greifposendetektions-Callback
Input:
  ROS Depth-Message type
Output:
  ROS-Publish:
    - Greifposendetektions-Image unter /ggcnn/img/grasp
    - Greifposendetektions-Werte unter /ggcnn/out/command
"""
def grasp_prediction_callback(depth_message):
  """
  Zugriff auf folgende globale Variablen
  """
  global model
  global graph
  global previous_max_points
  global fx, cx, fy, cy
  global angle_array
  global mean_angle

  """
  Empfangenes Tiefenbild zuschneiden
  """
  # Über ROS von der RealSense empfangenes Tiefenbild als OpenCV-Image abspeichern
  depth = bridge.imgmsg_to_cv2(depth_message)

  # Mittleres Rechteck der größe 300x300 aus dem Tiefenbild ausschneiden
  # Crop a square out of the middle of the depth and resize it to 300*300
  # Ausgerechnet: resize(depth[(90:390), (170:470)]) für 300
  crop_size = 300
  depth_cropped = depth[(480-crop_size)//2:(480-crop_size)//2+crop_size, (640-crop_size)//2:(640-
    crop_size)//2+crop_size]

  # Tiefenbild in drei Channels abspeichern, um später die Greifpose farblich
  # markieren zu können
  depth_cropped_output = np.zeros([300,300,3], dtype=np.uint8)
  depth_cropped_output[:,:,0] = np.uint8(depth_cropped)
  depth_cropped_output[:,:,1] = np.uint8(depth_cropped)
  depth_cropped_output[:,:,2] = np.uint8(depth_cropped)

  # Ungültige NaN-Werte im Tiefenbild mit 0 ersetzen
  depth_cropped = depth_cropped.copy()
  depth_nan = np.isnan(depth_cropped).copy()
  depth_cropped[depth_nan] = 0

```

```
# In Float umwandeln und Werte normalisieren
depth_cropped = depth_cropped.astype(float)
depth_cropped = depth_cropped/1000

# Es wurde festgestellt, dass sich das Netz nur auf die Leerstellen im Bild, bzw.
# jene Pixel mit dem Wert 0 konzentriert und dort die Greifposen festlegt.
# Deshalb wird die Fläche des Objektes auf 0 gesetzt und die Umgebung auf !0.
depth_cropped = np.clip((depth_cropped - depth_cropped.mean()), 0, 1)

"""
Tiefenbild dem CNN als Input übergeben
"""
with graph.as_default():
  pred_out = model.predict(depth_cropped.reshape((1, 300, 300, 1)))

# pred_out hat 4 Outputs. Jeder Output hat die Ausgabeform (1,300,300,1). Über squeeze()
# wird das ganze auf (300,300) gequetscht und die eindimensionalen Einträge des Arrays
# werden gelöscht.
pos_out = pred_out[0].squeeze()
pos_out[depth_nan] = 0

"""
Berechnung des Greifwinkels und der -weite
"""
# Gemeinsame Heatmap für den Greifwinkel aus den Heatmaps für Cos_out und Sin_out berechnen.
cos_out = pred_out[1].squeeze()
sin_out = pred_out[2].squeeze()
ang_out = np.arctan2(sin_out, cos_out)/2.0

width_out = pred_out[3].squeeze()  # Scaled 0-150:0-1

"""
Ausgaben filtern
"""
# Pos_out-Array filtern
pos_out = ndimage.filters.gaussian_filter(pos_out, 5.0)
# Ang_out-Array filtern
ang_out = ndimage.filters.gaussian_filter(ang_out, 3.0)

"""
Greifpose bestimmen und über die intrinsischen Koordinaten in Weltkoordinaten umrechnen
"""
# Satz aus 7 lokalen Maximas aus Pos_out berechnen und den, der sich am nächsten
# zum Maxima aus der letzten Iteration befindet, auswählen.
max_points = None
max_points = peak_local_max(pos_out, min_distance=5, threshold_abs=0.1, num_peaks=7)
if max_points.shape[0] == 0:     # Wenn keine gefunden wurden -> Abbruch
  return
grasp_pixel=max_points[np.argmin(np.linalg.norm(max_points - previous_max_points, axis=1))]

# Greifpixel für die nächste Iteration zwischenspeichern
```

```
150 previous_max_points = (grasp_pixel * 0.25 + previous_max_points * 0.75).astype(np.int)
151
152 # Greifwinkel und -weite an der Position des Greifpixels auslesen
153 ang = ang_out[grasp_pixel[0], grasp_pixel[1]]
154 width = width_out[grasp_pixel[0], grasp_pixel[1]]
155
156 # Für die Transformation von Bild- in Weltkoordinaten mit Hilfe der intrinsischen
157 # Parameter, müssen die bestimmten Koordinaten der Greifpose vom zugeschnittenen
158 # Bild in die Originalgröße 640x480 transformiert werden.
159 grasp_pixel = ((np.array(grasp_pixel) / 300.0 * crop_size) + np.array([(480 - crop_size)//2,
      (640 - crop_size) // 2]))
160 grasp_pixel = np.round(grasp_pixel).astype(np.int)
161
162 # Tiefe um einen definierten Bereich um die Greifposition schätzen
163 estimation_radius = 20
164 depth_estimation = depth[grasp_pixel[0]-estimation_radius:grasp_pixel[0]+estimation_radius,
      grasp_pixel[1]-estimation_radius:grasp_pixel[1]+estimation_radius]
165 grasp_pixel_cheat = unravel_index(depth_estimation.argmax(), depth_estimation.shape)
166
167 # Geschätze Tiefe Z der Greifposendetektion
168 point_depth = depth_estimation[grasp_pixel_cheat[0], grasp_pixel_cheat[1]]
169
170 # Umrechnung von Bild- in Weltkoordinaten mit Hilfe der intrinsischen Kameraparameter
171 imageHVLength = 17.5 # Cm. Horizontal = Vertical
172 imagePixelLength = 300
173 pixel_to_cm = imageHVLength/imagePixelLength
174 imageCenter_to_TCP_xOffset = 2
175 imageCenter_to_TCP_yOffset = 4.5
176 x = (grasp_pixel[1] * pixel_to_cm) - (imageHVLength/2) - imageCenter_to_TCP_xOffset
177 y = (grasp_pixel[0] * pixel_to_cm) - (imageHVLength/2) - imageCenter_to_TCP_yOffset
178 z = point_depth - 10.0    # Abzugülich einer Höhentoleranz
179
180 if np.isnan(z):
181   return
182
183 """
184 Greifposendetektion visualisieren
185 """
186 # Greifposition auf das Tiefenbild zeichnen
187 grasp_img = np.zeros((300, 300, 3), dtype=np.uint8)
188 rr, cc = circle(previous_max_points[0], previous_max_points[1], 5)  # Kreis mit dem Radius 5 an
     der Greifposition
189
190 # Grünen Kreis auf dem Ausgabebild zeichnen
191 depth_cropped_output[rr, cc, 0] = 0
192 depth_cropped_output[rr, cc, 1] = 255
193 depth_cropped_output[rr, cc, 2] = 0
194
195 # Mode-Filterung für die Winkelausgabe
196 mean_size = 10  # Arraylänge (auch Frameanzahl), über welche der Mittelwert gebildet werden
      soll
```

```
197   if len(angle_array) < mean_size:
198     angle_array = np.append(angle_array, ("%.1f" % ang))
199   elif len(angle_array) >= mean_size:
200     angle_array = np.sort(angle_array)
201     mode_results = stats.mode(angle_array)
202     mean_angle = float(mode_results[0])
203     angle_array = np.array([])
204
205   # Greifweite und -winkel auf das Tiefenbild zeichnen
206   width_const = width*100
207   ang_deg = mean_angle * (180/np.pi)
208   if ang > 3.14:
209     ang_deg = ang_deg - 360
210   elif ang_deg < -360:
211     ang_deg = ang_deg + 360
212     gegenKat = int(np.sin(ang_deg) * (width_const/2))
213     anKat = int(np.cos(ang_deg) * (width_const/2))
214
215   if ang > 0:
216     rr, cc = line(previous_max_points[0] + gegenKat, previous_max_points[1] - anKat,
      previous_max_points[0] - gegenKat, previous_max_points[1] + anKat)
217   else:
218     rr, cc = line(previous_max_points[0] - gegenKat, previous_max_points[1] - anKat,
      previous_max_points[0] + gegenKat, previous_max_points[1] + anKat)
219
220   # Rote Linie zeigt den Greifwinkel und die -weite
221   depth_cropped_output[rr, cc, 0] = 0
222   depth_cropped_output[rr, cc, 1] = 0
223   depth_cropped_output[rr, cc, 2] = 255
224
225   """
226   Ausgaben in ROS publishen
227   """
228   grasp_img = bridge.cv2_to_imgmsg(depth_cropped_output, 'bgr8')
229   grasp_img.header = depth_message.header
230   grasp_pub.publish(grasp_img)
231
232   # Output the best grasp pose relative to camera.
233   grasp_cmd_msg = Float32MultiArray()
234   grasp_cmd_msg.data = [x, y, z, ang, width]
235   grasp_cmd_pub.publish(grasp_cmd_msg)
236
237   depth_sub = rospy.Subscriber('/camera/depth/image_rect_raw', Image, grasp_prediction_callback)
238
239 while not rospy.is_shutdown():
240   rospy.spin()
```

A.6. Greifposendetektion an diversen Objekten

A.6.1. Gabel

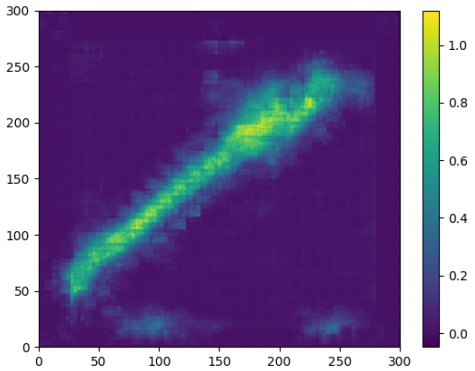

Abbildung A.2.: PosOut-Gabel

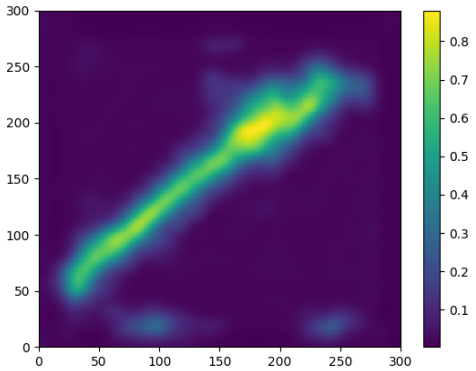

Abbildung A.3.: PosOut-Gabel (gefiltert)

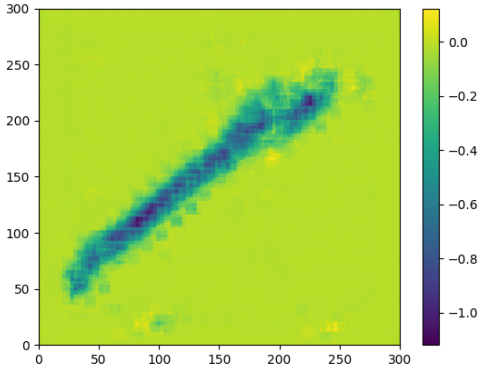

Abbildung A.4.: SinOut-Gabel

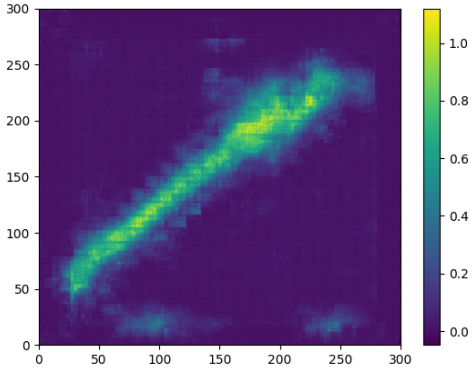

Abbildung A.5.: CosOut-Gabel

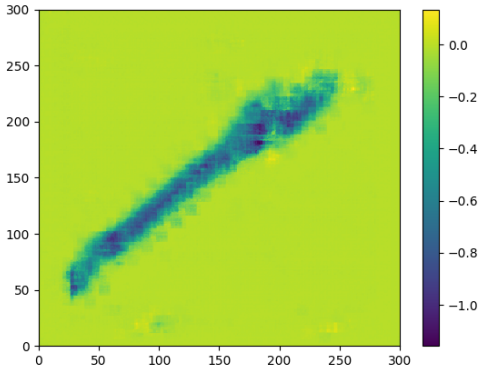

Abbildung A.6.: AngOut-Gabel

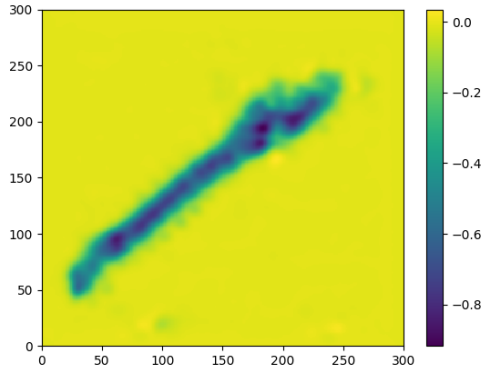

Abbildung A.7.: AngOut-Gabel (gefiltert)

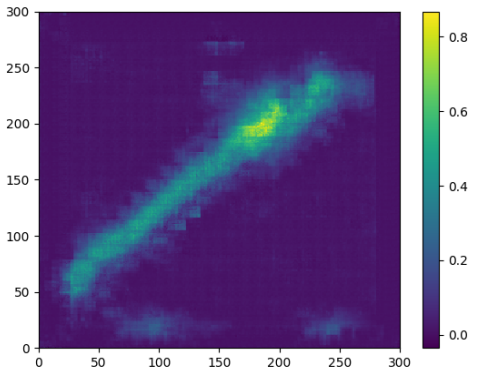

Abbildung A.8.: WidthOut-Gabel

A.6.2. Klebeband

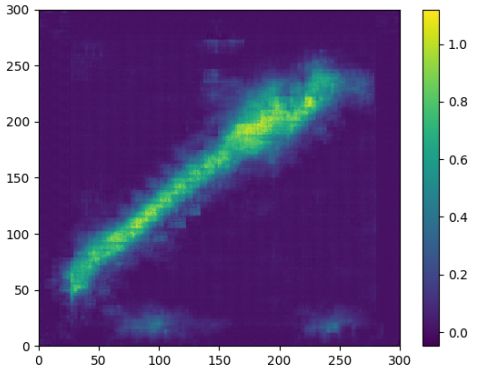

Abbildung A.9.: PosOut-Klebeband

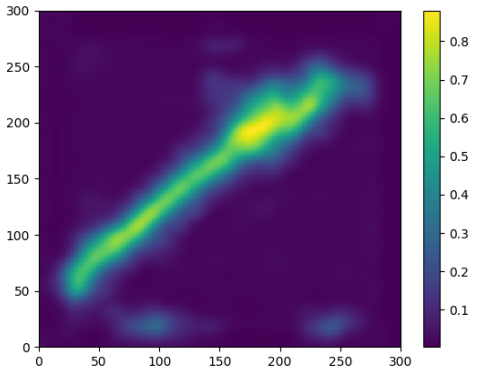

Abbildung A.10.: PosOut-Klebeband (gefiltert)

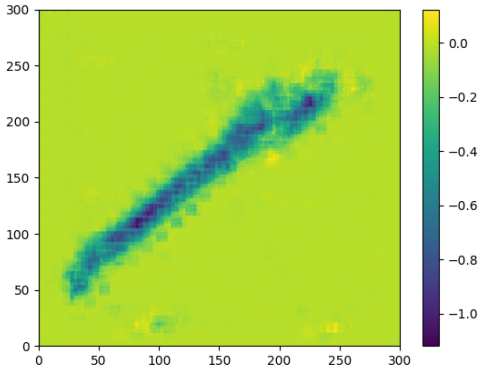

Abbildung A.11.: SinOut-Klebeband

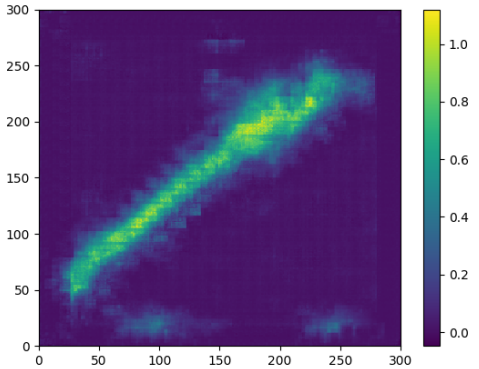

Abbildung A.12.: CosOut-Klebeband

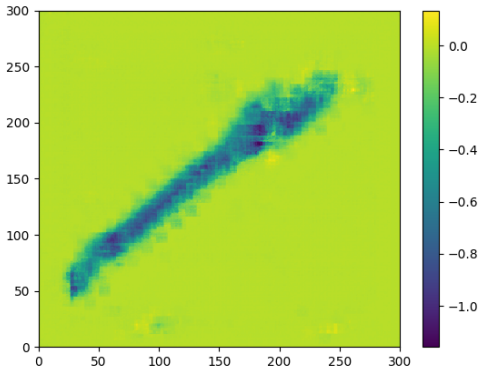

Abbildung A.13.: AngOut-Klebeband

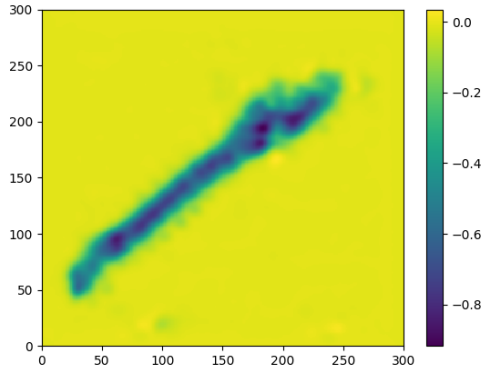

Abbildung A.14.: AngOut-Klebeband (gefiltert)

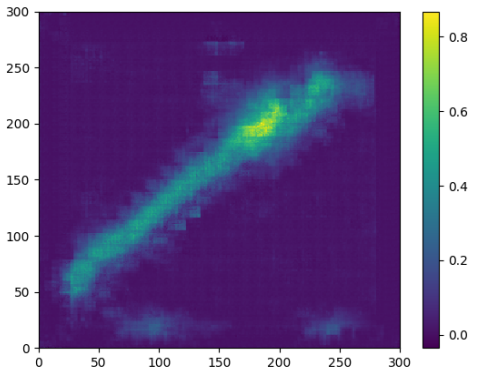

Abbildung A.15.: WidthOut-Klebeband

A.6.3. Fahrradgriff

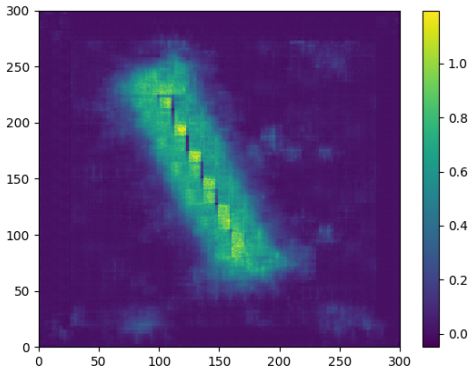

Abbildung A.16.: PosOut-Fahrradgriff

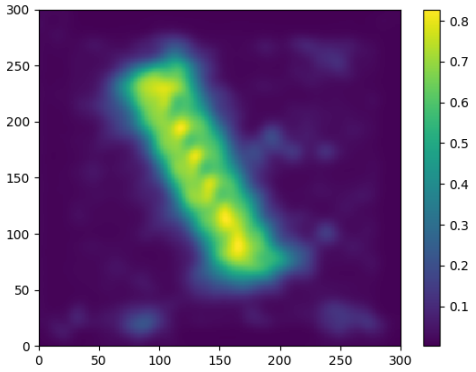

Abbildung A.17.: PosOut-Fahrradgriff (gefiltert)

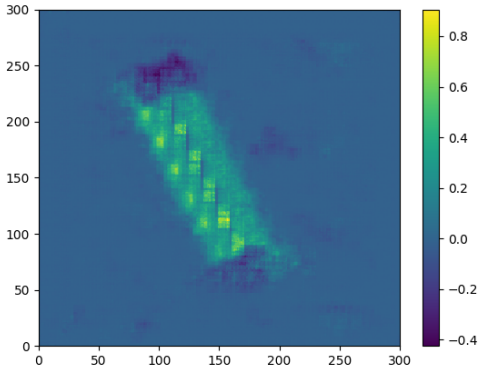

Abbildung A.18.: SinOut-Fahrradgriff

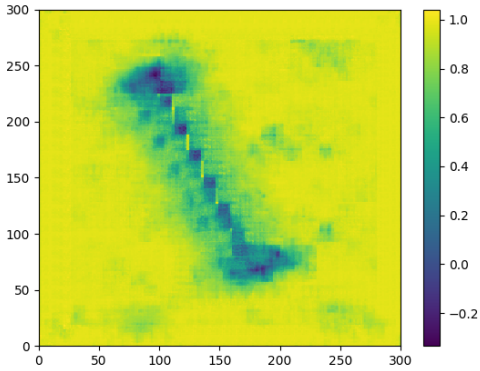

Abbildung A.19.: CosOut-Fahrradgriff

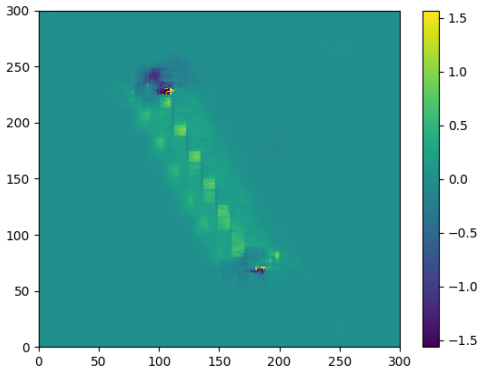

Abbildung A.20.: AngOut-Fahrradgriff

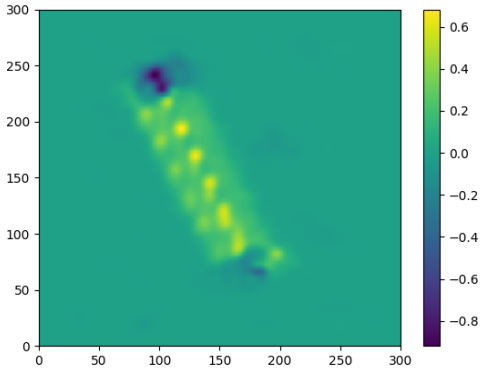

Abbildung A.21.: AngOut-Fahrradgriff (gefiltert)

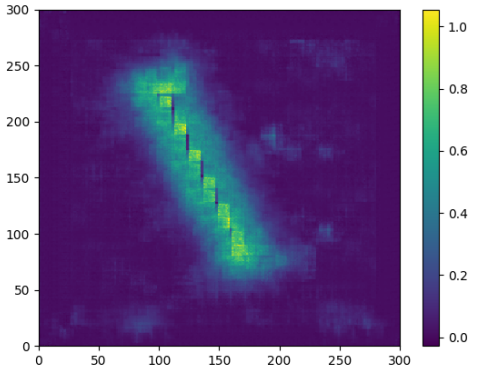

Abbildung A.22.: WidthOut-Fahrradgriff

A.7. Programmcode zur Robotersteuerung mittels CNN

A.7.1. MainWindow

A.7.1.1. On_btn_detection_pose_clicked

```
void MainWindow::on_btn_detection_pose_clicked(void)
{
  std::vector<std::string> joint_name;
  std::vector<double> joint_angle;
  double path_time = 2.0;

  // Greifposendetektionsstellung anfahren
  joint_name.push_back("joint1"); joint_angle.push_back(0.0);
  joint_name.push_back("joint2"); joint_angle.push_back(0.3);
  joint_name.push_back("joint3"); joint_angle.push_back(1.2);
  joint_name.push_back("joint4"); joint_angle.push_back(0.0);
  joint_name.push_back("joint5"); joint_angle.push_back(1.57);
  joint_name.push_back("joint6"); joint_angle.push_back(0.0);

  if(!qnode.setJointSpacePath(joint_name, joint_angle, path_time))
  {
    writeLog("[ERRROR!] Failed to move to grasp detection pose");
    return;
  }

  writeLog("Send joint angle to grab detection pose");
}
```

A.7.1.2. On_btn_cnn_prediction_clicked

```
void MainWindow::on_btn_cnn_prediction_clicked(void)
{
std::vector<double> current_position = qnode.getPresentKinematicsPos();
Eigen::Vector3d current_orientation_rpy = qnode.getPresentKinematicsOriRPY();
std::vector<double> graspingPose;

// CNN-Prediktionswerte zwischenspeichern
cnn_predictions = qnode.getGraspValues();

// X-Greifposition berechnen: Y-Greifposenwert (bzgl. des Bildkoordinatensystems) wird
// von Zentimetern in Millimetern umgerechnet und von der aktuelle X-Position des
// Roboterendeffektors (bzgl. des Roboterkoordinatensystem) subtrahiert.
graspingPose.push_back(current_position.at(0)-(cnn_predictions.data.at(1)/100));

// Y-Greifposition berechnen: X-Greifposenwert (bzgl. des Bildkoordinatensystems) wird
// von Zentimetern in Millimetern umgerechnet und von der aktuelle Y-Position des
```

```
// Roboterendeffektors (bzgl. des Roboterkoordinatensystem) subtrahiert.
graspingPose.push_back(current_position.at(1)-(cnn_predictions.data.at(0)/100));

// Z-Greifposition berechnen: (Z-Offset(0,095) vom Boden zur ersten Roboterachse)
// Z-Greifposenposition wird außerdem von Metern in Millimeter umgrechnet.
graspingPose.push_back(current_position.at(2)-(cnn_predictions.data.at(2)/1000)+0.095f);

// Greifwinkel um Z: Die Z-Achse der Kamera ist dem des Roboterkoordinatensystems
// entgegengerichtet. Um den korrekten Greifwinkel anzufahren, ist ein Vorzeichen-
// wechsel nötig.
graspingPose.push_back(-cnn_predictions.data.at(3));

// Greifweite
graspingPose.push_back(cnn_predictions.data.at(4));

// Greifpose bzgl. des Ursprungskoordinatensystems des Robotermanipulators in
// auf den Ausgabefeldern der GUI anzeigen lassen.
ui.lineEdit->setText(QString().sprintf("%0.3f", graspingPose.at(0)));
ui.lineEdit_2->setText(QString().sprintf("%0.3f", graspingPose.at(1)));
ui.lineEdit_3->setText(QString().sprintf("%0.3f", graspingPose.at(2)));
ui.lineEdit_4->setText(QString().sprintf("%0.3f", graspingPose.at(3)));
ui.lineEdit_5->setText(QString().sprintf("%0.3f", graspingPose.at(4)));
}
```

A.7.1.3. On_btn_grab_object_clicked

```
void MainWindow::on_btn_grab_object_clicked(void)
{
  std::vector<double> kinematics_pose;
  std::vector<std::string> joint_name;
  std::vector<double> joint_angle;
  double path_time = 2.0;

  // Erst Greifwinkel einstellen...
  joint_name.push_back("joint1"); joint_angle.push_back(0.0);
  joint_name.push_back("joint2"); joint_angle.push_back(0.3);
  joint_name.push_back("joint3"); joint_angle.push_back(1.2);
  joint_name.push_back("joint4"); joint_angle.push_back(0.0);
  joint_name.push_back("joint5"); joint_angle.push_back(1.57);
  joint_name.push_back("joint6"); joint_angle.push_back( (ui.lineEdit_4->text()).toDouble()
     );

  if(!qnode.setJointSpacePath(joint_name, joint_angle, path_time))
  {
    writeLog("[ERROR!] Failed configure grasping angle");
    return;
  }

  // ... dann kurz warten...
  sleep(3.5);   // Wartezeit zwischen Greifwinkeleinstellung und Anfahren der Position
```

```
24   double deltaPath = ui.lineEdit_3->text().toDouble();
25
26   // ...und Zwischenposition über der Greifposition am Objekt anfahren...
27   kinematics_pose.push_back((ui.lineEdit->text()).toDouble());
28   kinematics_pose.push_back((ui.lineEdit_2->text()).toDouble());
29   kinematics_pose.push_back((ui.lineEdit_3->text()).toDouble()*0.7);  // Nur 70% der Höhe
       anfahren
30
31   if(!qnode.setTaskSpacePathPositionOnly(kinematics_pose, path_time))
32   {
33     writeLog("[ERROR!] Failed to move to grasping point");
34     return;
35   }
36
37   kinematics_pose.clear();
38   sleep(3.5);
39
40   // ... lineare Abwärtsbewegung hin zur Greifposition.
41   std::vector<double> goal_pose;    goal_pose.resize(7, 0.0);
42   goal_pose.at(2) = deltaPath*0.3;
43   qnode.setTaskSpacePathFromPresent(goal_pose, 1.0);
44   writeLog("Grabbing object...");
45 }
```

A.7.2. QNode

A.7.2.1. CNN-Subscriber

```
1 // GG-CNN Subscriber
2 cnn_grasp_subscriber_ = n.subscribe("/ggcnn/out/command", 10, &QNode::setGraspValues, this)
      ;
```

A.7.2.2. getGraspValues

```
1 // Liefert die Werte aus cnn_predictions_ zurück
2 std_msgs::Float32MultiArray QNode::getGraspValues()
3 {
4   return cnn_predictions_;
5 }
```

A.7.2.3. setGraspValues

```
1 // Speichert den vom ROS-CNN-Node erhaltenen Greifposenvektor g in cnn_predictions_ ab
2 void QNode::setGraspValues(const std_msgs::Float32MultiArray::ConstPtr &msg)
3 {
4   cnn_predictions_.data.clear();
5   cnn_predictions_.data.push_back(msg->data.at(0)); // Greifposition in X
```

```
6   cnn_predictions_.data.push_back(msg->data.at(1)); // Greifposition in Y
7   cnn_predictions_.data.push_back(msg->data.at(2)); // Greifposition in Z
8   cnn_predictions_.data.push_back(msg->data.at(3)); // Greifwinkel
9   cnn_predictions_.data.push_back(msg->data.at(4)); // Greifweite
10 }
```

A.8. Klassendiagramm der OM6-Control-GUI

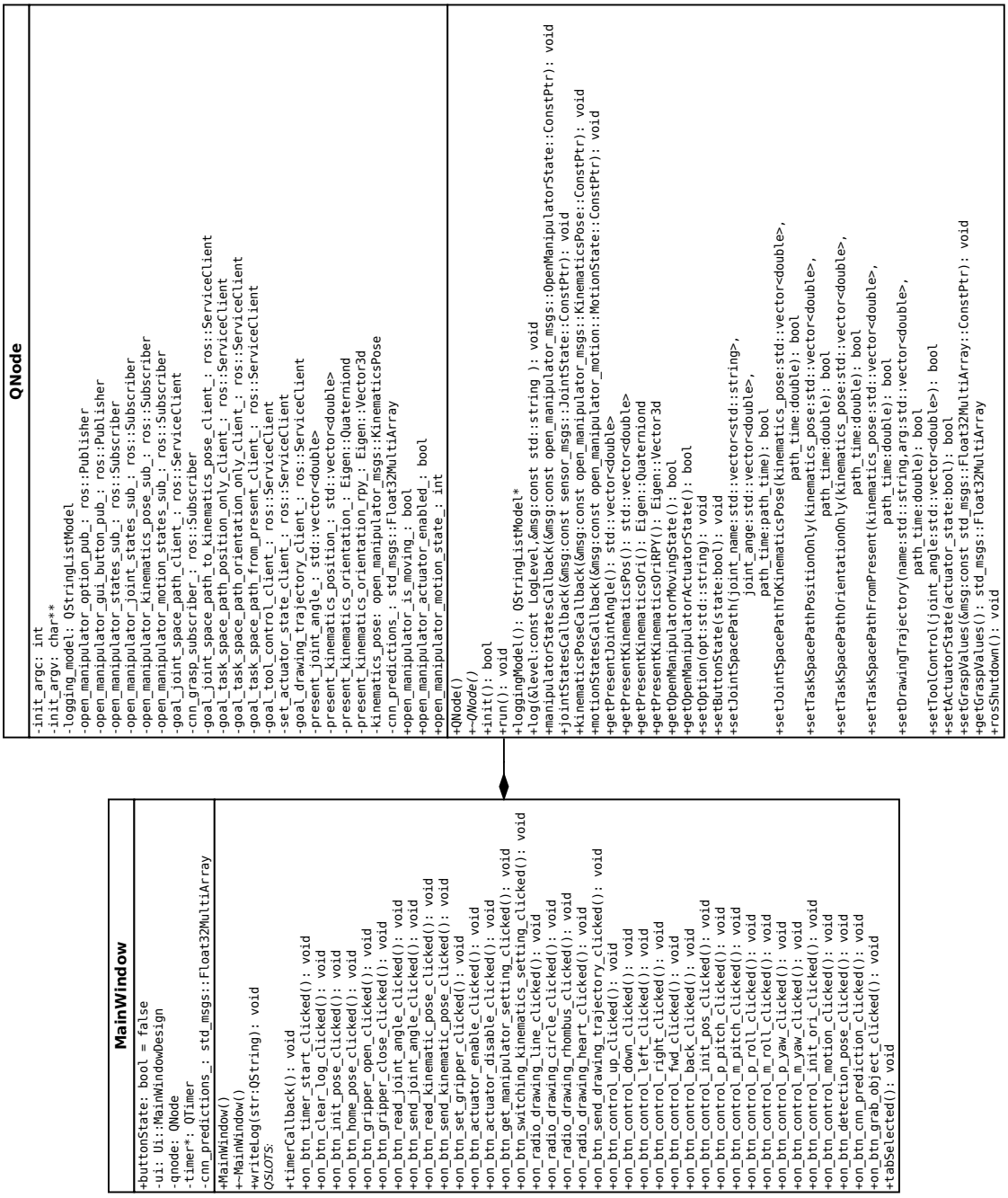

Abbildung A.23.: Klassendiagramm der Steuerungssoftware des OM6

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 31. Januar 2020	
Ort, Datum	Unterschrift