HOCHSCHULE HANNOVER
UNIVERSITY OF APPLIED SCIENCES AND ARTS
–
Fakultät I
Elektro- und Informationstechnik

Entwicklung einer kamerabasierten Abstandsmessung mittels Tiefenkamera und Objekterkennung

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Engineering

vorgelegt von

Sebastian Schmidt

am

07.03.2021

Erstprüfer: Prof. Dr.-Ing. Martin Mutz
Zweitprüfer: Mohammad Beyki, M.Eng.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hannover, 07.03.2021

_____

Name (Unterschrift)

Inhaltsverzeichnis

Abkürzungsverzeichnis

AP – Average precision

CNN – Convolutional neural network

CPU – Central processing unit

FN – False negative

FOV – Field of view

FP – False positive

FPS – Frames per second

GPU – Graphical processing unit

IOU – Intersection over union

mAP – Mean average precision

NMS – Non maximum suppression

RNN – Recurrent neural network

TP – True positive

TPU – Tensor processing unit

YOLO – You only look once

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Machine Learning ist aus der heutigen Welt gar nicht mehr wegzudenken. Es ist bereits seit Jahren Bestandteil der Forschung und es werden ständig noch neue Fortschritte gemacht, die wieder neue Einsatzbereiche ermöglichen. So wird Machine Learning heute schon in der Spracherkennung, Übersetzung, Vorhersage von Daten, Objekterkennung oder dem autonomen Fahren genutzt. Immer mehr Unternehmen setzen auf Ansätze aus dem Machine Learning [Deru et al. 2020, 17–32]. Das gleiche gilt auch für optische 3D-Messverfahren. Es gibt unzählige verschiedene Verfahren, die das Erfassen von dreidimensionalen Strukturen ermöglichen sollen. Dies ist vor allem in der Industrie interessant, da die Nutzung von 3D-Messverfahren große Beiträge zur Qualitätssicherung beisteuern kann. So können Bauteile auf Fehler oder auf ihre Qualitätsstandards geprüft werden. Aber auch in der Verkehrstechnik, Medizin, oder Robotik werden derartige Messverfahren verwendet. Dort können die Verfahren zum Beispiel als Hilfe in der Diagnostik oder zur Unterstützung des Fahrers beim autonomen Fahren genutzt werden [Sackewitz 2017, 16–18].

Ein solches 3D-Messverfahren und Machine Learning sollen in dieser Arbeit kombiniert werden. Es soll eine auf Deep Learning basierende Objekterkennung in Kombination mit einem optischen 3D-Messverfahren verwendet werden, um nicht nur ein Objekt in Bildern zu erkennen, sondern zusätzlich auch noch Informationen über die Lage des Objekts im Raum zu ermitteln. Durch die Kombination der beiden Verfahren sollen die Daten viel besser genutzt werden. Anstatt nur der Position im Bild können zusätzlich noch Informationen wie die Höhe, Entfernung oder genaue Position bestimmt werden. Die Tiefendaten ermöglichen Analysen, die so vorher nicht möglich gewesen wären. Dies kann vor allem bei der Szenenanalyse hilfreich sein. Bei solch einer Analyse wird versucht das Bild und den Inhalt zu verstehen. Auch bei der Bildsegmentierung können diese Informationen von großem Nutzen sein, um zu erkennen, welche Teile des Bildes möglicherweise zusammengehören.

Das Projekt entsteht aber nicht als alleinstehende Arbeit. Vielmehr ist die Arbeit Teil eines größeren Projektes, in dem viele verschiedene Methoden des Machine Learnings erforscht und umgesetzt werden. Ziel des Gesamtprojektes ist, einen Roboterarm mit Hilfe von Machine Learning zu trainieren, sodass dieser das Werfen eines Tischtennisballs in einen Becher lernt. Es sollen verschiedene Methoden des Machine Learnings und der Bildanalyse erforscht, vertieft und umgesetzt werden. Dieses Ziel wird in vielen kleineren Arbeiten modular umgesetzt. So befasst sich eine Arbeit mit dem Erstellen eines mathematischen Modells, das den Wurf des Balls unter Berücksichtigung verschiedenster Störfaktoren nachbildet. Eine andere Arbeit befasst sich mit dem Erstellen und Trainieren eines Modells des Roboters mittels Reinforcement Learning in Unity. Dies soll vor allem Zeit ersparen, da das Training nur mit einem Robotermodell erfolgt und somit eine viel höhere Anzahl an Trainingsversuchen möglich ist. Weitere Arbeiten vergleichen verschiedene Modelle für die Objekterkennung oder der Analyse

der Wurfbewegung mit Hilfe von Deep Learning. Am Ende sollen die vielen alleinstehenden Arbeiten zusammengesetzt werden und somit das Werfen des Roboters ermöglichen.

Diese Arbeit erfüllt dabei die Aufgabe die zu treffenden Becher zu erkennen, die Position zu bestimmen und für die Weiterverarbeitung vorzubereiten. Für diese Aufgabe sollen hierbei zum einen die theoretischen Hintergründe, aber auch die praktische Umsetzung erläutert werden. Die Objekterkennung wird mittels Transfer Learning auf der Basis eines bestehenden Modells trainiert. Für das Ermitteln der 3D-Informationen wird eine Tiefenkamera genutzt. Die Kamera liefert ein Farbbild, auf dem die Objekterkennung ausgeführt werden kann, sowie ein Tiefenbild, das die Informationen über die Entfernung des Objekts liefert. Das Ermitteln des Tiefenbilds erfolgt durch Triangulation, indem zwei Bilder der gleichen Szene aufgenommen werden.

Es gibt bereits einige Vorarbeiten, welche sich mit den einzelnen Teilgebieten dieser Arbeit, aber auch des Gesamtprojekts beschäftigen. Darunter sind viele Arbeiten und Bücher über die verschiedensten Arbeiten von Machine- und Deep Learning [Chollet 2018; Deru et al. 2020], aber auch schon andere Projekte, bei denen Roboter das Werfen von Objekten oder Schlagen von Tischtennisbällen erlernt haben [M. Matsushima et al. 2005]. Hier ist vor allem der TossingBot zu nennen. Dieser wurde bei Google entwickelt und kann verschiedenstes Plastikobst mit erstaunlicher Genauigkeit in Schalen werfen [Zeng et al. 2020]. Dennoch befasst sich dieses Projekt mit vielen anderen Themen und kann einen sehr detaillierten Einblick in die verschiedenen Verfahren und deren Möglichkeiten geben, die dieser Bereich bietet.

2. Stand der Technik

Die Objekterkennung mittels neuronaler Netze ist ein bereits sehr viel erforschtes Thema, zu dem es unzählige Anleitungen und Bücher gibt. Durch beliebte Frameworks wie Tensorflow, Pytorch oder Keras und die dazugehörige Literatur werden die Konzepte des Machine Learning immer zugänglicher, was den Einstieg in das Gebiet deutlich vereinfacht. Eine solche Objekterkennung mittels Machine Learning wird dabei schon in vielen Anwendungen in ganz verschiedenen Bereichen genutzt. In der Medizin dient sie zum Erkennen von Krankheitsbildern bei bildgebenden Diagnoseverfahren. Beim autonomen Fahren erkennt sie Objekte und Gefahren und warnt so das Auto oder den Fahrer vor diesen [Deru et al. 2020, 19–31]. Durch die auf der einen Seite immer schneller arbeitenden Computer und auf der anderen Seite immer effizienter werdenden Algorithmen, ist die Anwendung von Deep Learning zum Teil sogar schon auf embedded Systemen möglich [Verhelst et al. 2018, 64]. Wenn die Entwicklung in den nächsten Jahren so weiter verläuft, ist zu erwarten, dass Machine Learning und Deep Learning auch noch weiterwachsen und noch mehr Anwendungsgebiete finden werden. Und auch wenn Deep Learning noch lange nicht alle Probleme der Bildverarbeitung lösen kann, sind solche Modelle der klassischen Programmierung zumindest

in Aufgaben wie Klassifizierung, Segmentierung oder Objekterkennung bereits voraus [Walsh et al. 2019, 3].

Optische 3D-Messverfahren werden ebenfalls schon in vielen unterschiedlichen Bereichen genutzt. Dabei gibt es für alle möglichen Anwendungen ein passendes Sensorprinzip. So werden diese Verfahren beispielsweise in für Qualitätssicherung, Fertigungsprozesse, 3D-Druck, Medizin, Kriminalistik, Archäologie oder viele andere Bereiche eingesetzt [Sackewitz 2017, 206]. Ein Überblick über die verschiedenen Verfahren ist in Abbildung 1 zu sehen.

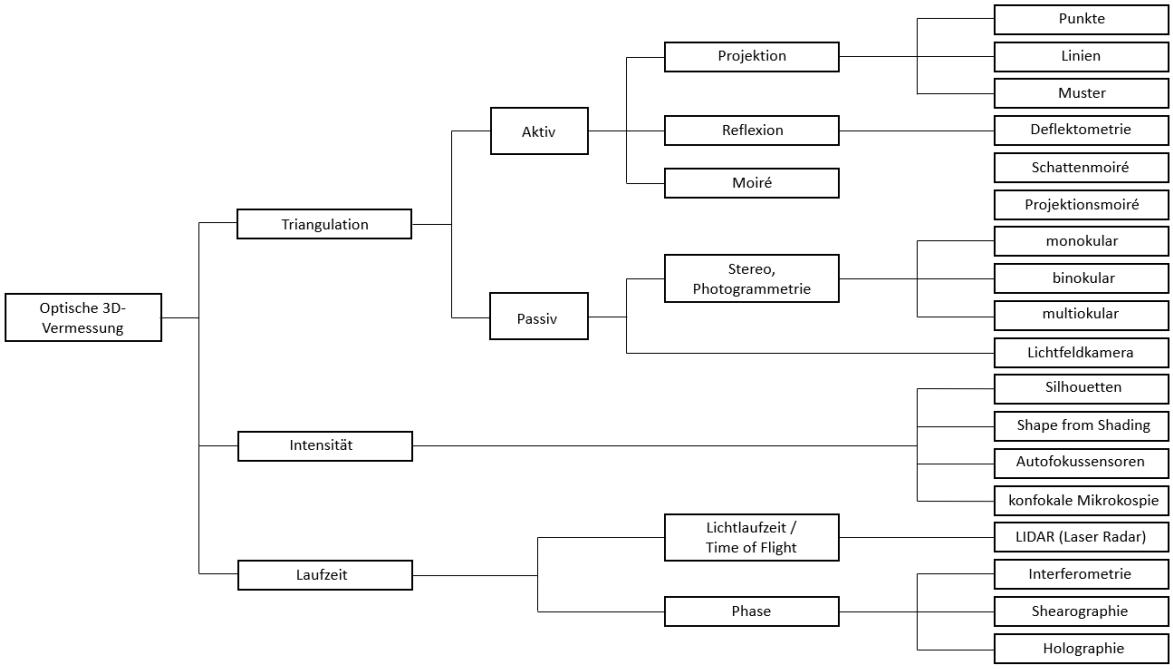


Abbildung 1: Übersicht der 3D-Messverfahren [Sackewitz 2017, 206]

In der Regel bestehen die Messverfahren dabei aus einem Sender und einem Empfänger, mit denen die 3D-Informationen ermittelt werden. Grundsätzlich wird das Objekt dabei vom Sender mit einem Signal beaufschlagt. Der Empfänger kann dann die durch das Objekt veränderten Daten wie Amplitude, Phase, Polarisation oder Richtung messen [Sackewitz 2017, 206]. Die Messung mit Hilfe von Kameras erfolgt in der Regel über die obenstehenden Triangulationsverfahren. Dabei kann zwischen aktiven und passiven Verfahren unterschieden werden. Die aktiven Verfahren nutzen einen zusätzlichen Sender bzw. Emitter, um die Messung zu verbessern. Dabei wird zum einen die Ausleuchtung verbessert und zum anderen über Projektionsmuster das Erkennen bestimmter Merkmale vereinfacht [Sackewitz 2017, 211]. In diesem Segment gibt es ganz unterschiedliche Verfahren mit einer unterschiedlichen Anzahl und verschiedenen Arten von Kameras. Im Bereich der Kalibrierung hat vor allem Zhengyou Zhang einen sehr großen Beitrag geleistet [Zhang 2000]. Da das Thema der 3D-Messverfahren schon seit sehr langer Zeit Teil der Forschung ist, gibt es eine Reihe von Büchern, die sich mit der Bildverarbeitung und dem dreidimensionalen Computersehen auseinandersetzen [Jiang et al. 1997; Schreer 2005; Süße et al. 2014].

Ändere Arbeiten beschäftigen sich oft mit einer ähnlichen Thematik, der ‚Pose Estimation'. In dem Themenbereich sollen mit der Hilfe von Kameras die menschlichen Posen erkannt werden. Oft werden dafür ebenfalls Tiefenkameras eingesetzt. Eines der bekanntesten Verfahren ist vermutlich Microsoft Kinect. Bei Kinect wird eine Tiefenkamera mitgeliefert, die anhand der Daten versucht die menschlichen Posen zu erkennen und so interaktive Spiele möglich zu machen, in denen diese nur über Gesten der Spieler gespielt werden [Z. Zhang 2012]. Grundsätzlich kann die reine Posenerkennung aber auch ohne die Nutzung von speziellen Kameras erfolgen. Oft werden dafür Deep Learning Methoden verwendet, um die Posen und Gesten zu erkennen und zu visualisieren [Toshev et al. 2014].

3. Grundlagen

In diesem Kapitel werden die Grundlagen für die spätere Umsetzung des Projekts erläutert. Für nicht direkt relevante Themen wird ein Verweis auf weitere Literatur gegeben, um den Lesefluss nicht zu stören.

3.1. Korrelationsanalyse

Die Korrelationsanalyse wird genutzt um lineare Abhängigkeiten zwischen mehreren Merkmalen zu überprüfen. Dabei wird getestet, ob ein Merkmal im statistischen Mittel mit einem anderen Merkmal sinkt bzw. steigt. Für die Berechnung des Korrelationskoeffizienten werden die Standardabweichungen s_0 und s_1 der beiden Merkmale, sowie die Kovarianz s_{01} benötigt. Der Koeffizient kann über die folgende Gleichung berechnet werden [Bortz et al. 2010, 239–240].

$$r_{01} = \frac{s_{01}}{s_0 \cdot s_1} \tag{1}$$

Der Korrelationskoeffizient r_{01} beschreibt dabei die stärke der Abhängigkeit zwischen den beiden Merkmalen. Die Werte liegen immer zwischen -1 und 1, wobei Werte nahe -1 eine stark negative und Werte nahe 1 eine stark positive Korrelation bedeuten. Werte nahe 0 zeigen eine sehr geringe Abhängigkeit zwischen den Merkmalen [Beyki 2021].

3.2. Machine Learning

Das Machine Learning ist ein Teilgebiet der künstlichen Intelligenz. Dabei sollen die Modelle Strukturen erlernen, die für die Entscheidungsfindung in einem bestimmten Gebiet notwendig sind und sie auf spätere Anwendungsfälle übertragen können [Deru et al. 2020, 18]. Machine Learning Modelle zeichnen sich dabei durch einen geringen Entwicklungsaufwand und leichte Anpassbarkeit aus, benötigen dafür aber oft eine große Anzahl an Daten [Deru et al. 2020, 36]. Während bei klassischen Ansätzen Regeln für bestimmte Probleme formuliert werden, aus denen später ein kompliziertes Programm entwickelt wird, erkennt das Modell beim Machine Learning selbst Merkmale in den Daten und filtert diese heraus, um daraus die erwarteten Ausgaben zu bilden [Chollet 2018].

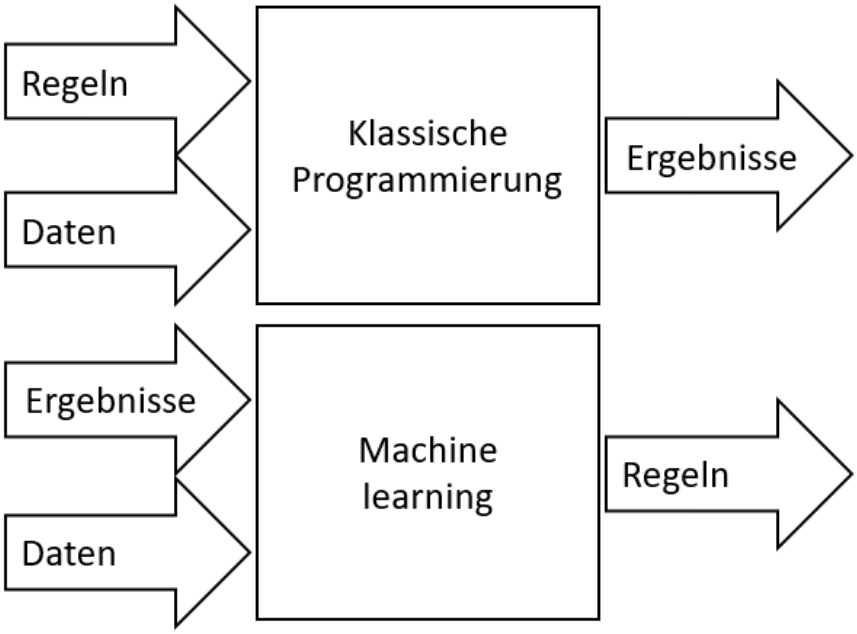


Abbildung 2: Vergleich von klassischer Programmierung und Machine Learning

Ein Problem bei der reinen Programmierung solcher Modelle ist die sich ständig verändernde Datenlage [Chollet 2018]. Durch Änderungen der realen Bedingungen verändern sich oft auch die Regeln, welche das Programm wiedergeben soll. Dies führt zu einem hohen Anpassungsaufwand. Bei Machine Learning Modellen hingegen können die Daten angepasst und das Modell relativ leicht neu trainiert werden.

So wird Machine Learning heutzutage schon in allem Möglichen Branchen für verschiedenste Aufgaben genutzt. Sprachassistenten wie Alexa und Siri nutzen es um Eingaben besser zu verstehen und zu verarbeiten. Google Translate oder DeepL übersetzen damit ganze Texte, soziale Netzwerke filtern den Feed für die Nutzer oder machen neue Vorschläge für interessante Beiträge. Autonome Fahrzeuge erkennen Gefahren oder lernen das selbständige Fahren oder die Modelle erkennen frühzeitig Störungen an Maschinen oder Tumore bei Krebspatienten [Deru et al. 2020, 20–32].

Machine learning hat allerdings nicht nur Vorteile. Ein Nachteil dabei ist zum Beispiel, dass sich Modelle wie eine „Blackbox“ verhalten [Deru et al. 2020, 36]. Das Modell verarbeitet die Eingangsdaten und bestimmt daraus resultierend die Ausgangsbedingungen. Wie genau das Modell dabei die Entscheidung fällt, ist nicht nachvollziehbar. Außerdem ist die Qualität des Modells direkt von der Qualität der Eingabedaten abhängig. Dies zeigt vor allem wie wichtig es ist, die Daten für das Modell gut aufzubereiten und auszuwählen. So zeigt zum Beispiel auch das aktuell vermutlich leistungsstärkste Sprachmodell GPT-3 von OpenAI, das mit diversen Texten aus dem Internet trainiert wurde, tief verankerte Vorurteile gegen Muslime, weil die für das Training genutzten Texte diese enthielten [Holland 2021]. In anderen Anwendungen könnte es bedeuten, dass eine Objekterkennung für Hunde diese z.B. nur bei Sonnenschein erkennt, weil das Modell nur mit sehr gut belichteten Bildern trainiert wurde.

3.2.1. Regressionsanalyse

Eines der Hauptgebiete für die Nutzung von Machine Learning ist die Regressionsanalyse. Bei einer Regression werden kontinuierliche Werte vorausgesagt. Dies steht im Gegensatz zu den diskreten Werten, die zum Beispiel bei einer Klassifikationsaufgabe die jeweiligen Klassen repräsentieren. Ziel der Regressionsanalyse ist die Herstellung einer Verbindung einer abhängigen und einer oder mehreren unabhängigen Variablen [Deru et al. 2020, 40].

Ein viel genutztes Beispiel für eine Regression ist die Vorhersage von Immobilienpreisen. Das Modell bekommt dabei diverse verschiedene Eingaben wie Zimmeranzahl, Ort, Größe, Alter und mehr und berechnet daraus ein Modell, mit dem die Preise weiterer Immobilien berechnet werden können. In diesem Beispiel ist der Immobilienpreis die abhängige Variable und die Eingabedaten wie z.B. Ort, Größe etc. sind die unabhängigen Variablen.

Bei der Regressionsanalyse kann grundsätzlich zwischen vielen verschiedenen Verfahren unterschieden werden. Die drei wichtigsten sind aber die lineare Regression, die multiple Regression und die polynomiale Regression.

3.2.1.1. Lineare Regression

Bei der linearen Regression wird die abhängige Variable mit einer einzigen unabhängigen Variable modelliert. Die unabhängige Variable kann frei variiert werden und erzeugt dabei verschiedene Ergebnisse der abhängigen Variable. Die Regressionsgleichung dazu lautet:

$$\hat{y}_i = \beta \cdot x_i + \alpha \tag{2}$$

In dieser Gleichung ist $\hat{y}_i$ die abhängige, x_i die unabhängige Variable, β die Gewichtung der Variable und α der Achsenabschnitt der linearen Gleichung [Beyki 2021]. Für die Herleitung

der Gleichung wird der Fehler r_i zwischen gemessenem Wert y_i und dem berechneten Wert $\hat{y}_i$ ermittelt.

$$r_i = y_i - \hat{y}_i = y_i - (\beta \cdot x_i + \alpha) \tag{3}$$

Diese Fehlerfunktion wird nun quadriert, um Unstetigkeitsstellen vorzubeugen und negative Werte zu vermeiden [Beyki 2021].

$$r_i^2 = (y_i - \hat{y}_i)^2 = \left(y_i - (\beta \cdot x_i + \alpha)\right)^2 \tag{4}$$

Diesen Wert gilt es in der Regression zu minimieren und so die möglichst besten Ergebnisse für die unabhängige Variable zu erhalten. Die genaue Herleitung für die lineare Regression wird hier nicht weiter erklärt. Diese kann bei Interesse in [Beyki 2021] nachgelesen werden.

Die lineare Regression kann alternativ, anstatt aus der Statistik, aus der linearen Algebra hergeleitet werden. Dies ergibt eine andere Schreibweise, die der späteren Speicherung im Programm deutlich ähnelt. Dafür können x und y als Vektoren der gemessenen Ergebnisse angenommen werden.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, h_0(x) = \theta_0 + \theta_1 \cdot x \tag{5}$$

Aus den einzelnen Gleichungen der Geradengleichung kann dann anschließend eine Matrix gebildet werden, welche die die einzelnen Variablen repräsentiert.

$$\begin{aligned} h_0(x_1) &= \theta_0 + \theta_1 \cdot x_1 = y_1 \\ h_1(x_2) &= \theta_0 + \theta_1 \cdot x_2 = y_2 \\ &\vdots \\ h_n(x_n) &= \theta_0 + \theta_1 \cdot x_n = y_n \end{aligned} \tag{6}$$

Diese Gleichungen können in Matrizen und Vektoren aufgeschrieben werden. Daraus ergibt sich die folgende Schreibweise.

$$y = X * \theta = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \cdot \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{7}$$

Ausgehend davon kann die Regression über die folgende Formel gelöst werden.

$$\theta = (X^T \cdot X)^{-1} \cdot X^T \cdot y \tag{8}$$

Die weitere Herleitung kann ebenfalls [Beyki 2021] entnommen werden.

3.2.1.2. *Multiple lineare Regression*

Die multiple lineare Regression lässt sich ganz genau wie die einfache lineare Regression aus der linearen Algebra herleiten. In diesem Fall enthalten die Gleichungen aber nicht nur eine unabhängige Variable, sondern mehrere. Diese einzelnen Koeffizienten verhalten sich alle linear. Auf diese Weise können zum Beispiel Modelle wie die in Kapitel 3.2.1 erwähnten Immobilienpreise modelliert werden. Durch das Erweitern der Formeln (2) und (6) um weitere Koeffizienten, ergeben sich die folgenden Gleichungen als Grundlage für die Regression.

$$\begin{aligned} h_0(x_1) &= \theta_0 + \theta_1 \cdot x_{11} + \cdots + \theta_m \cdot x_{1m} = y_1 \\ h_1(x_2) &= \theta_0 + \theta_1 \cdot x_{21} + \cdots + \theta_m \cdot x_{2m} = y_2 \\ &\vdots \\ h_n(x_n) &= \theta_0 + \theta_1 \cdot x_{n1} + \cdots + \theta_m \cdot x_{nm} = y_n \end{aligned} \tag{9}$$

Dies ergibt durch Ändern der Notation:

$$y = X \cdot \theta = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1m} \\ 1 & x_{21} & \cdots & x_{2m} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nm} \end{pmatrix} \cdot \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{10}$$

Die Regression kann ebenfalls mittels Formel (8) gelöst werden.

3.2.2. Neuronale Netze

Neuronale Netze sind grundlegend dem menschlichen Gehirn nachempfunden und sollen das Verhalten dieses möglichst gut modellieren. Das menschliche Gehirn besteht aus zahlreichen Neuronen, die wiederum mit zahlreichen weiteren Neuronen verbunden sind. Diese Verbindungen sind verschieden gewichtet und verändern sich, je nachdem ob sie mehr oder weniger genutzt werden. Die Kommunikation zwischen den einzelnen Neuronen erfolgt dabei über elektrische Impulse, die über sogenannte Axone weitergeleitet werden. Die Aufnahme der Impulse erfolgt durch Dendriten, die Reize an das Neuron weitergeben. Auf diese Weise werden neue Zusammenhänge erlernt. Der „Wert“ eines Neurons setzt sich dabei aus den einzelnen Summen der Eingänge aus den anderen Neuronen zusammen [Deru et al. 2020, 67].

Je nachdem wie die Eingänge des Neurons gewichtet sind, feuert es und leitet eigene Impulse weiter. Dieses Verhalten soll beim Machine Learning durch künstliche Neuronen nachgebildet werden. Die Eingangsreize werden hierbei durch eine Eingangsschicht realisiert, welche die Eingangsdaten repräsentiert. Diese werden an ein sogenanntes Perzeptron weitergeleitet, das die Eingänge aufsummiert und dann die Ausgabe anhand einer Aktivierungsfunktion bestimmt. Dieses Verhalten ist in Abbildung 3 zu sehen.

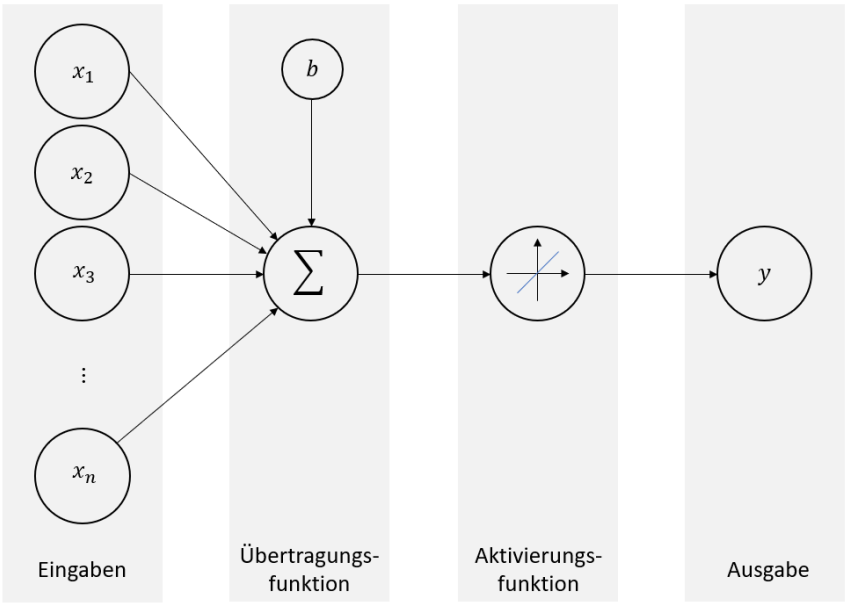


Abbildung 3: Abbildung einees künstlichen Perzeptrons [Deru et al. 2020, 71]

Typischerweise gibt es mehrere Aktivierungsfunktionen die genutzt werden können. Die Aktivierungsfunktionen liefern in der Regel Werte zwischen -1 und 1 bzw. 0 und 1. In Abbildung 4 sind verschiedene Aktivierungsfunktionen mit ihren dazugehörigen Verläufen aufgezeichnet.

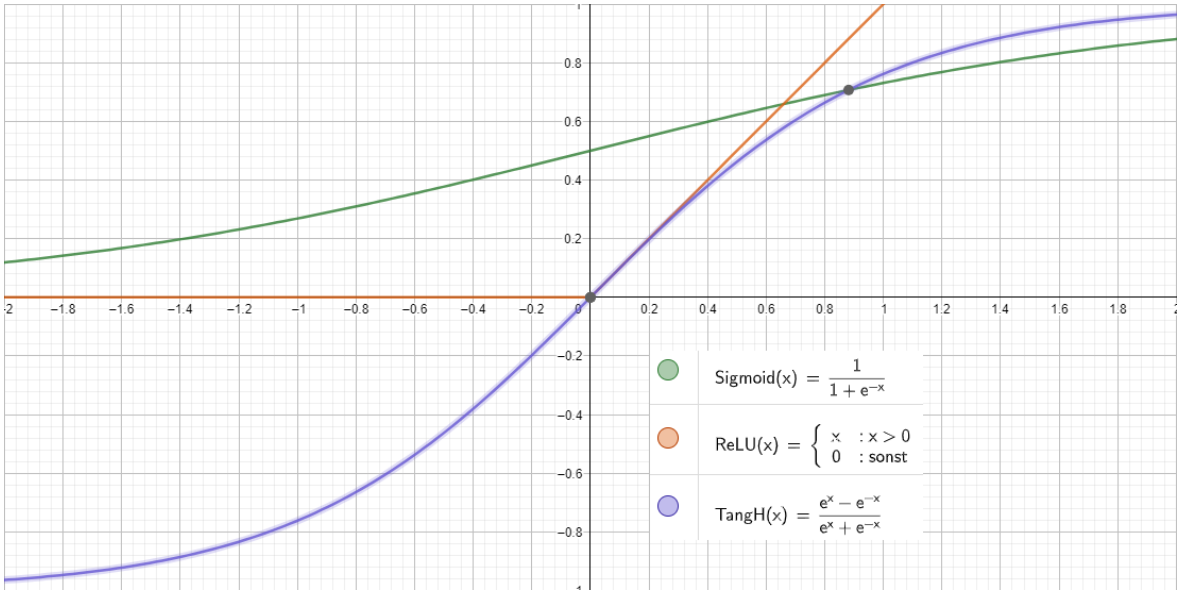


Abbildung 4: Vergleich von verschiedenen Aktivierungsfunktionen

Dieses Verhalten eines Perzeptrons ähnelt dem einer Nervenzelle. Wird ein bestimmter Schwellwert überschritten, wird dieses aktiviert [Deru et al. 2020, 70]. Die genauen Erklärungen der Aktivierungsfunktionen sind für diese Arbeit nicht weiter relevant. Wichtig ist nur zu wissen, dass es heutzutage sehr üblich ist, die ReLU- oder die Sigmoid-Funktion als Aktivierungsfunktion zu verwenden.

Durch das Zusammenschalten mehrerer dieser Perzeptren, ergibt sich daraus ein neuronales Netz. In diesem sind diverse Perzeptren miteinander über gewichtete Verbindungen verbunden. Ein solches, einfaches neuronales Netz ist in Abbildung 5 gezeigt. Das Netz besitzt drei Eingänge und einen Ausgang, sowie eine verdeckte Schicht.

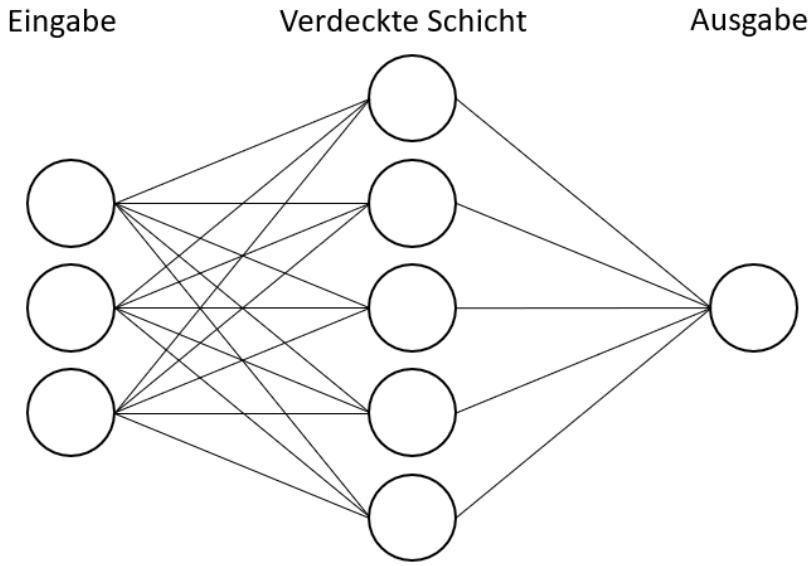


Abbildung 5: Einfaches neuronales Netzwerk

Grundlegend besteht ein solches neuronales Netz immer aus einer Eingabeschicht, mindestens einer verdeckten Schicht und einer Ausgabeschicht. Die Anzahl der Perzeptren in der Eingabeschicht sind von den zu verarbeitenden Daten abhängig. Sollen beispielsweise Autos klassifiziert werden, ist die Anzahl der Eingänge gleich der Anzahl der Variablen, welche für die Klassifizierung genutzt werden sollen. Sollen zum Beispiel Grauwertbilder einer Größe von 20x20 klassifiziert werden, besitzt das Netz für jeden Pixel einen Eingang, also 20x20=400 Eingänge. Die Ausgabeschicht wiederum ist von der Anzahl der erwarteten Ausgaben abhängig. Bei Klassifizierungsaufgaben könnte zum Beispiel jede Klasse ein eigenes Ausgabeperzeptron entsprechen. Die Anzahl der verdeckten Schichten kann dabei allerdings variiert werden. Und hier liegt auch der Spielraum, in dem das Netzwerk angepasst werden kann. Jede Schicht erfüllt dabei den Zweck eines Datenverarbeitungsmoduls [Chollet 2018]. In diesen Schichten sollen Merkmale herausgefiltert werden, um so die Daten besser zu verarbeiten. Dieses Verhalten ist in Abbildung 6 anschaulich dargestellt.

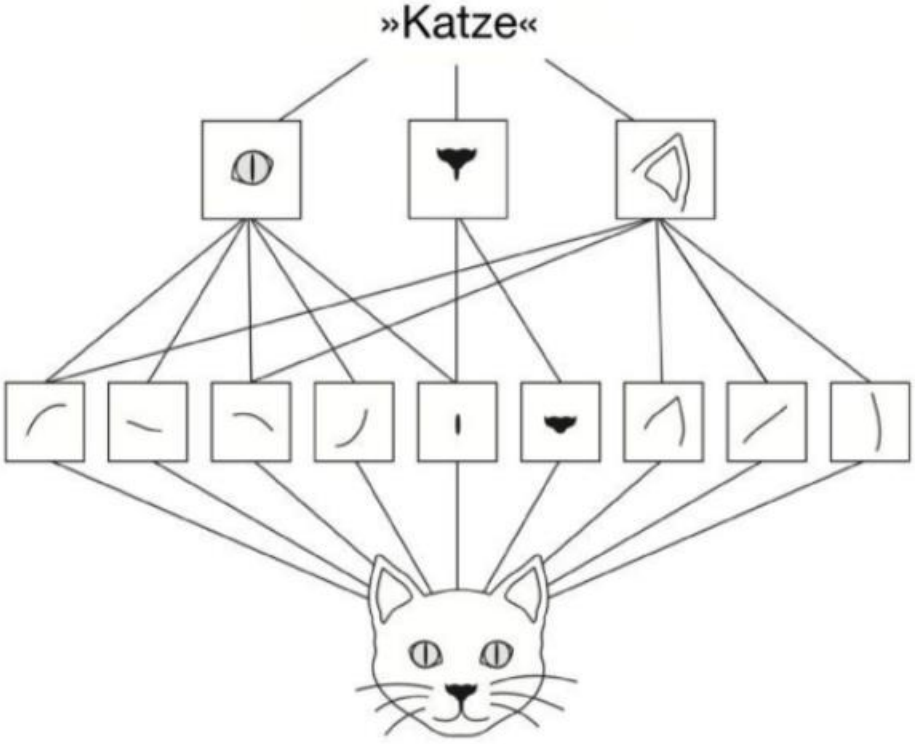


Abbildung 6: Extraktion von Merkmalen bei neuronalen Netzen [Chollet 2018]

Hier werden beispielhaft in der ersten verdeckten Schicht einfache Linien herausgefiltert und diese in der zweiten Schicht zu komplizierteren Merkmalen wie Augen oder Ohren zusammengesetzt. Aus diesen Merkmalen kann das neuronale Netz dann bestimmen, dass es sich in diesem Falle um eine Katze handelt. Die Abbildung ist selbstverständlich nur ein Beispiel und ein solcher Zusammenhang würde in der Realität in der Regel von deutlich komplizierteren Netzen bearbeitet werden. Besitzt ein Modell mehr als zwei verdeckte Schichten, so kann von einem tiefen neuronalen Netz gesprochen werden. Auch die Anzahl der Perzeptren jeder verdeckten Schicht kann angepasst werden. Wichtig dabei ist allerdings, dass mit steigender Anzahl an Schichten nicht nur die Komplexität der möglichen erlernbaren Zusammenhänge, sondern auch die Anzahl der einzelnen Perzeptren und Gewichte steigt, was zu einem deutlich höheren Rechenaufwand führt [Chollet 2018].

Damit das neuronale Netz Zusammenhänge wie in Abbildung 6 erlenen kann, benötigt es einen Algorithmus für das Trainieren und Anpassen der Gewichte. Hierfür wird ein sogenannter Backpropagation-Algorithmus verwendet. Hierbei wird eine Fehlerfunktion gebildet. Mit Hilfe dieser Fehlerfunktion wird die Abweichung der aktuellen Ausgaben von den erwarteten Werten abgebildet. Diese Abweichung wird auch als ‚Loss' bezeichnet und dient oft als Qualitätsmaß während des Trainings. Von diesen Fehlerfunktionen gibt es selbstverständlich viele. Um den groben Vorgang zu erklären, wird hier die mittlere quadratische Abweichung ausgewählt.

$$E = \frac{1}{n} \cdot \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{11}$$

Mit Hilfe dieses Fehlers werden die Gewichte immer weiter angepasst, bis die Fehlerfunktion für die Trainingsdaten ihr Minimum erreicht. Gestartet wird dabei mit den Gewichten zwischen der letzten verdeckten und der Ausgabeschicht. Von hier aus arbeitet sich der Algorithmus immer weiter nach vorne, bis auch der Fehler der Gewichte zwischen Eingabeschicht und erster verdeckter Schicht minimiert ist. Das Anpassen der Gewichte erfolgt nach der Delta-Regel, einer allgemeinen Gleichung für das Trainieren der Gewichte [Deru et al. 2020, 76–78].

$$w_{i_{neu}} = w_{i_{alt}} - \eta \cdot \frac{\partial E}{\partial w_i} \tag{12}$$

In dieser Gleichung beschreiben $w_{i_{neu}}$ und $w_{i_{alt}}$ die neuen und alten Gewichte und η die Lernrate, mit der die Gewichte angepasst werden. Die Lernrate ist neben der Anzahl und Größe der verdeckten Schichten und der Auswahl der Aktivierungsfunktionen ein weiterer Parameter, mit dem das Training der Netze angepasst werden kann. Die Wahl der Lernrate ist dabei äußerst wichtig. Die Lernrate bestimmt zum Teil wie schnell das Netzwerk das Training beenden kann und ob überhaupt das beste Ergebnis erzielt wird. Durch die Wahl einer zu hohen Lernrate terminiert der Lernprozess äußerst schnell und das optimale Ergebnis wird evtl. nicht erreicht bzw. es wird übersprungen. Ist die Rate allerdings zu niedrig, verläuft das Training ggf. sehr langsam und bleibt möglicherweise in einem lokalen Minimum stecken. Dadurch kann sowohl eine zu hohe, als auch eine zu kleine Lernrate zu schlechten Ergebnissen führen. In der Praxis wird die Lernrate oft recht hoch gewählt und im Laufe des Trainings immer weiter gesenkt [Deru et al. 2020, 71]. Um eine gute Lernrate zu finden, kann diese immer weiter erhöht werden, bis die Lernrate zu hoch ist und der Loss nicht mehr verringert wird. Ist dieser Punkt gefunden, kann die Lernrate ca. um den Faktor 0,5 verkleinert werden und als Startpunkt gewählt werden [Géron 2020].

Ein weiterer Faktor, mit dem das Training angepasst werden kann, ist das Trainieren in Batches. Hier werden die Gewichte nicht nach jedem Eingangsdatensatz angepasst, sondern erst nachdem eine bestimmte Anzahl an Daten getestet wurde. Die Fehlerfunktion wird somit nicht für jeden einzelnen Datensatz, sondern immer als Durchschnitt mehrerer Eingaben gebildet. Ist die Größe dieser Batches kleiner als der Datensatz der Eingaben, werden diese mini-Batches genannt. Das Verändern dieser Größe kann die Trainingsdauer essentiell beeinflussen. Wird das Netzwerk mit solchen Batches trainiert, ändert sich die allgemeine Formel für das Anpassen der Gewichte folgenderweise, wobei M die Größe der Batches ist [Deru et al. 2020, 78].

$$w_{i_{neu}} = w_{i_{alt}} - \mu \cdot \frac{1}{M} \cdot \sum_{k=1}^{M} \frac{\partial E}{\partial w_i} \tag{13}$$

3.2.3. Deep Learning

So wie Machine Learning ein Teilgebiet von künstlicher Intelligenz ist, verhält sich Deep Learning als Teilgebiet des Machine Learning. Um dies zu veranschaulichen, ist das Verhältnis in Abbildung 7 nochmal skizziert.

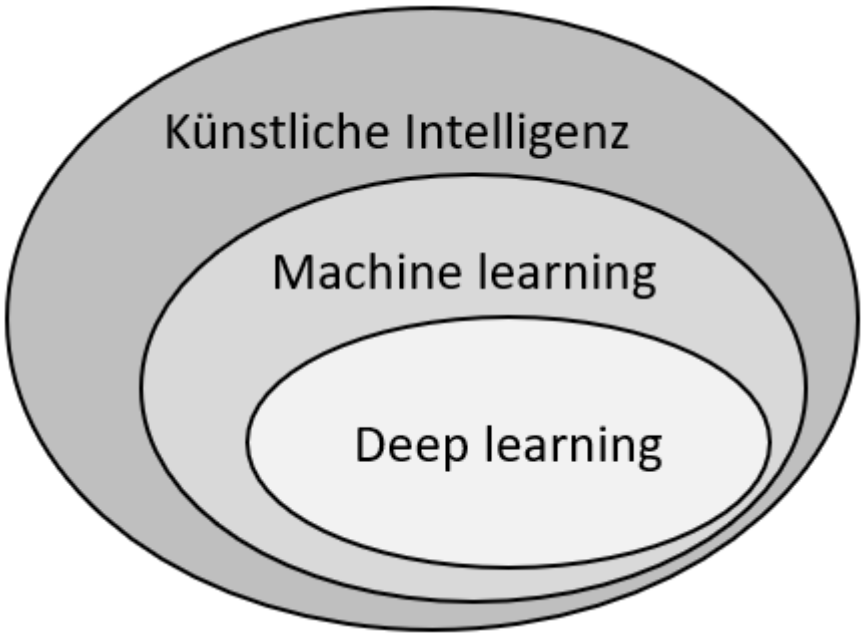


Abbildung 7: Beziehung zwischen künstlicher Intelligenz, Machine Learning und Deep Learning

Beim Deep Learning werden neuronale Netze verwendet. Diese besitzen allerdings mehrere verdeckte Schichten und werden deshalb als ‚tief' bezeichnet. Daher stammt auch der Begriff ‚Deep Learning'. Während einfache neuronale Netze sich für einfachere Aufgaben wie Vorhersagen mit nur wenigen Variablen sehr gut eignen, werden für kompliziertere Zusammenhänge in der Regel tiefe neuronale Netze benötigt, die durch ihre Vielzahl an verschiedenen Schichten und deren Zusammenhänge eine Menge an Merkmalen herausfiltern können. So werden diese Netze in vielen verschiedenen Bereichen der Bildverarbeitung erfolgreich für das Erkennen von Gesichtern und Gesten oder die Analyse von Bildinhalten verwendet [Teich 2020]. Eine weitere beliebte Anwendung ist die Spracherkennung. Die Netze sind sehr gut in der Lage die Spracheingaben zu erkennen und in dem Zusammenhang die passenden Antworten zu finden. Auf diese Weise funktionieren beispielsweise auch Spracherkennungen wie Siri oder Alexa. Auch hier werden heutzutage mittlerweile schon beeindruckende Ergebnisse erzielt. Google hat mittlerweile eine KI entwickelt, die Termine per Telefon verabredet, ohne dass der gegenüber überhaupt merkt, dass er nur mit einer KI spricht [Floemer 2018].

All diese verschiedenen Anwendungsfälle benötigen verschiedene Umsetzungen der tiefen neuronalen Netze. Durch die Anpassung des Aufbaus der Netze und das Nutzen verschiedener Schichten können ganz verschiedene Ziele erreicht werden. Die vermutlich meist genutzten Architekturen heißen Convolutional Neural Network (CNN) und Recurrent Neural Network

(RNN). CNNs eignen sich besonders gut für das Filtern und Transformieren von zeitunabhängigen Daten. Dies macht sie vor allem bei der Erkennung und Klassifizierung von Objekten und Bildern sehr nützlich. Die Besonderheit dieser CNNs sind die sogenannten Convolutional Layer. In diesen Schichten werden die Daten nicht einfach nur übergeben, sondern zusätzlich noch mit einem Filter transformiert. Diese Filter besitzen eine oft quadratische Form und werden nach und nach über das ganze Eingangsbild gelegt, bis alle Pixel transformiert wurden. Zu Beginn des Trainings werden die Filter mit zufälligen Werten initialisiert, wodurch die Filter zu Beginn noch keine Merkmale herausfiltern können. Durch das Trainieren des Netzwerks werden die Filter aber immer weiter verbessert und das Netzwerk kann die Zusammenhänge mit Hilfe der Filter besser verarbeiten [Ambalina 2020]. Zusätzlich werden nach solchen Schichten häufig auch noch Pooling Layer verwendet, die durch verschiedene Verfahren die Daten zusammenfassen und somit kürzen. Dadurch werden die zu verarbeitenden Daten von Schicht zu Schicht weniger und die markanten Stellen im Bild hervorgehoben [Deru et al. 2020, 91]. Aus diesen Merkmalen kann das Netzwerk dann am Ende eine Entscheidung wie in Abbildung 6 treffen.

CNNs haben allerdings Probleme bei der Verarbeitung von zeitabhängigen Daten. Dies zeigt sich vor allem bei dem Verarbeiten von Videos oder Texten bzw. Spracheingaben. In diesen Gebieten müssen die vorherigen Eingaben oft ebenfalls betrachtet werden, da sie den Kontext der aktuellen Eingaben mitbestimmen. Hierbei haben RNNs ihre Stärke, weshalb sie auch vorrangig für die Verarbeitung von Text und Sprache genutzt werden. In Abbildung 8 wird ein solches Netzwerk mit grundlegendem Prinzip aufgezeigt. Das Netzwerk hat als zusätzlichen Eingang noch den temporalen Zusammenhang.

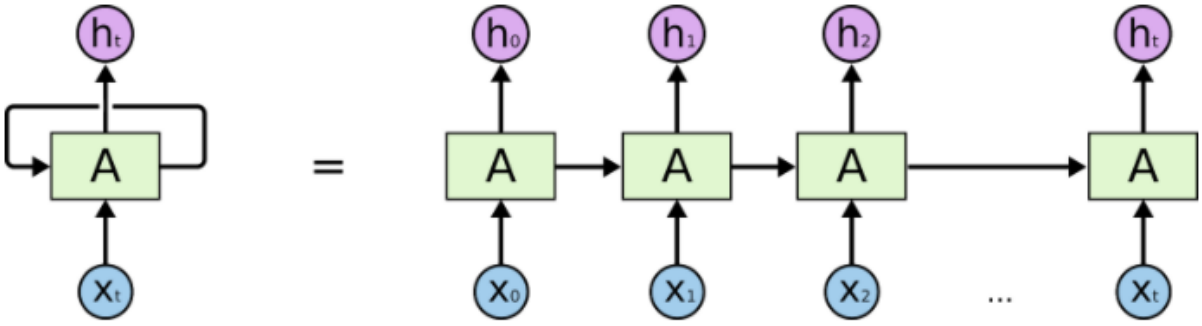


Abbildung 8: Aufbau von RNNs [Olah 2015]

Beispielhaft könnte das Netzwerk eine Autokorrektur darstellen. In diesem Fall würde das Netzwerk auf der einen Seite die eingegebenen Wörter oder Buchstaben und auf der anderen Seite die vorherigen Buchstaben bekommen. Dadurch, dass das Netzwerk den Zusammenhang kennt, in dem die neuen Buchstaben eingegeben werden, kann das Netzwerk bestmöglich hervorsagen, welche Buchstaben daraus resultierend als nächstes folgen könnten [Ambalina 2020].

3.2.4. One-Shot- und Two-Shot-Detektoren für die Objekterkennung

Wie vorher schon erwähnt, nutzen Netzwerke für die Objekterkennung vor allem CNNs. Diese nutzen Convolutional Layer um Merkmale herauszufiltern und aus diesen anschließend die Ausgaben zu generieren. Aber auch unter diesen Netzwerken wird noch zwischen zwei verschiedenen Varianten unterschieden. Für die Objekterkennung können hier zwei unterschiedlich aufwändige Verfahren genutzt werden, die beide in Abbildung 9 gezeigt sind.

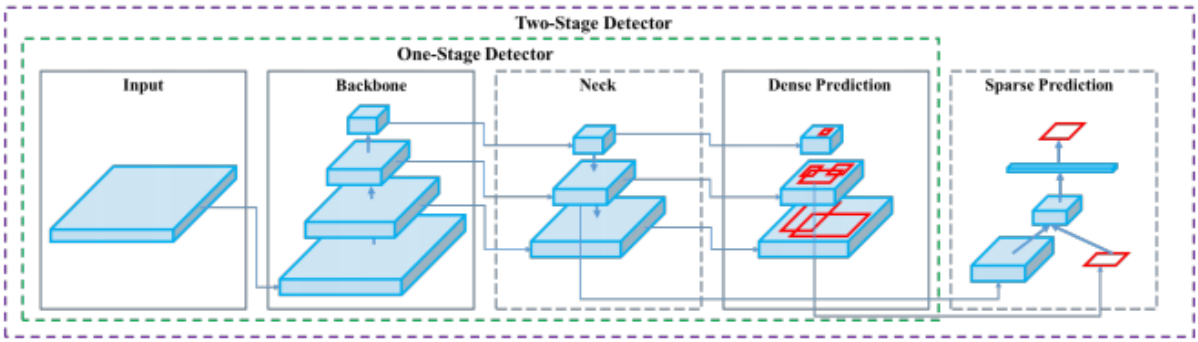


Abbildung 9: Aufbau von CNNs für die Objekterkennung [Bochkovskiy et al. 2020]

Grundsätzlich arbeiten beide erstmal nach einem ähnlichen Prinzip. Sie beide filtern über verschiedene Schichten die Merkmale heraus (Backbone), interpretieren diese und machen darauf basierend Vorhersagen, wo sich die Objekte befinden könnten (Dense Predicition). Der große Unterschied liegt darin, dass ein Bild bei dem Two-Shot-Detektor zwei Mal durchlaufen wird. Beim ersten Durchlauf werden die interessanten Bildregionen gesucht und erst beim zweiten Durchlauf wird die Klassifizierung bzw. die Erkennung ausgeführt. Single-Shot-Detektoren hingegen bearbeiten diese Aufgaben eher wie eine Regression und versuchen daraus Möglichst gut die Erkennung der Objekte und Position zu erlernen. Dadurch, dass diese nur einen Durchlauf benötigen, sind sie für gewöhnlich deutlich schneller. Die Two-Shot-Detektoren sind dagegen zwar rechenintensiver, erreichen aber auch höhere Genauigkeiten [Soviany et al. 2018, 1].

3.2.5. Metriken für die Genauigkeitsmessung einer Objekterkennung

Um die Genauigkeit einer Objekterkennung zu messen gibt es mehrere Metriken, die teilweise sehr ähnliche Zusammenhänge beschreiben. Eine der vermutlich wichtigsten Metriken ist die Intersection over Union (IOU). Hierbei wird der Anteil der Überschneidung von vorausgesagter und echter Bounding Box betrachtet. Mathematisch kann dies über folgenden Zusammenhang beschrieben werden.

$$IOU = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})} \tag{14}$$

Hierbei steht B_p für die vorausgesagte Bounding Box und B_{gt} für die wirkliche Bounding Box. Wenn die errechnete IOU mit einem ausgewählten Threshold verglichen wird, kann eine Aussage darüber getroffen werden, ob eine Erkennung als richtig oder falsch angesehen wird [Padilla et al. 2020]. Ein üblicher Wert wäre hier zum Beispiel 0,5. Dies sagt aus, dass eine erkannte Box als Richtig angesehen wird, wenn die Schnittfläche mit der echten Box mindestens 50% der beiden Boxen zusammen beträgt. Dieser Zusammenhang wird in Abbildung 10 auch nochmal deutlicher.

Nun besteht die Möglichkeit eine Aussage über die Korrektheit einer erkannten Box zu treffen. Dadurch können die richtig Positiven (true positive, TP), falsch Positiven (false positive, FP) und falsch Negativen (false negative, FN) Ergebnisse betrachtet werden.

Die richtig negativen Ergebnisse werden nicht weiter betrachtet, da theoretisch jedes Bild unzählige richtig negative ‚Erkennungen' besitzt, sodass dieser Wert nicht aussagekräftig ist. Über die Zusammenhänge dieser können die Precision P und der Recall R bestimmt werden [Soviany et al. 2018]. Diese können folgendermaßen berechnet werden:

$$P = \frac{TP}{TP + FP} = \frac{TP}{Alle\ Detektionen} \tag{15}$$

$$R = \frac{TP}{TP + FN} = \frac{TP}{Alle\ richtigen\ Objekte} \tag{16}$$

Die Precision beschreibt die Fähigkeit nur die richtigen Objekte als Erkennung zu zeigen. Dies ist der Anteil der richtig Positiven an allen als positiv erkannten Objekten. Der Recall beschreibt die Fähigkeit alle wichtigen Objekte zu erkennen. Dies ist der Anteil aller richtig positiven an der Gesamtzahl der zu erkennenden Objekte [Soviany et al. 2018]. Auch diese beiden Metriken werden in Abbildung 10 veranschaulicht.

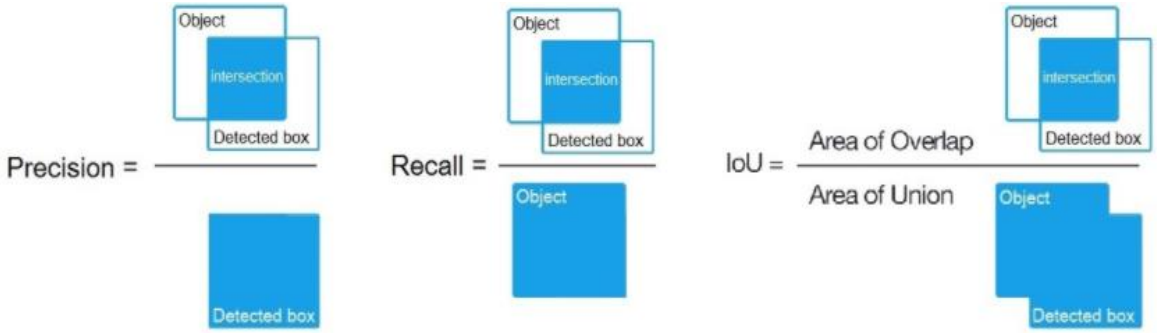


Abbildung 10: Präzision, Recall, IoU

[Bochkovskiy 2020]

Weitere sehr wichtige Metriken sind die AP (average precision) und die mAP (mean average precision). Die AP beschreibt die durchschnittliche Präzision für eine bestimmte Klasse. Um diese zu ermitteln wird die Präzision in Abhängigkeit des Recalls betrachtet. Anschließend wird

für elf verschiedene Werte von R = [0, 0.1, 0.2, …, 1] der Durchschnitt der maximalen Präzisionen errechnet. Daraus folgt [Soviany et al. 2018]:

$$AP = \frac{1}{11} \cdot \sum_{R \in \{0,0.1,0.2,\ldots,1\}} \max P\,(R) \tag{17}$$

Die mAP beschreibt einfach nur die Genauigkeit des Modells über alle Klassen hinweg. Um diese zu berechnen werden alle APs für die einzelnen Klassen benötigt und daraus der Durchschnitt gebildet. Dies ist ein Indikator für die Genauigkeit des gesamten Modells, wobei N die Gesamtzahl der Klassen und AP_i die jeweiligen APs der einzelnen Klassen beschreibt [Soviany et al. 2018].

$$mAP = \frac{1}{N} \cdot \sum_{i=1}^{N} AP_i \tag{18}$$

3.3. Stereobasierte Tiefenmessung

Das grundlegende Prinzip der Tiefenmessung durch eine stereobasierte Tiefenmessung ist ein dem Menschen durchaus bekanntes Prinzip, denn es basiert auf dem gleichen Mechanismus, der auch Menschen zu einem räumlichen Sehen verhilft. Menschen nehmen dabei zwei Bilder der gleichen Szene auf, aus denen durch die Verschiebung der beiden Bilder eine Tiefenwahrnehmung entstehen kann [Marr et al. 1979, 301]. Genau dieses Prinzip macht sich auch die stereobasierte Tiefenmessung mittels Kameras zu Nutzen. Es werden zwei Kameras aufgestellt, welche die die gleiche Szene beobachten, die allerdings um eine Basis *b* zueinander verschoben ist. Aufgrund dieser Verschiebung entsteht ein leichter Versatz der abgebildeten realen Szene auf den beiden Bildern. Abhängig von der Entfernung der Objekte zu den Kameras ist der Versatz größer oder kleiner. Objekte die an den Kameras stehen weisen dadurch einen großen Versatz auf, während dieser mit steigender Entfernung zu den Kameras immer kleiner wird. Auf Basis dieser Bilder kann dann eine Triangulation durchgeführt werden, um die Tiefe der Bildpunkte in der realen Welt zu ermitteln [Jiang et al. 1997, 8].

3.3.1. Optimale Kamerageometrie

Um diese Methode effizient nutzen zu können, wird eine Standard-Stereogeometrie benötigt. Diese Standard-Geometrie setzt mehrere Eigenschaften voraus. Zum einen sollten beide Kamerabilder eine identische Szene enthalten, die nur aus zwei verschiedenen Blickwinkeln betrachtet wird. Zusätzlich sollten die Kamerabilder parallel zu der Basislinie sein. Die Basislinie ist die Linie, die die beiden optischen Zentren der Kameras verbindet. Die letzte Voraussetzung ist, dass die zueinander kolineraen Zeilen der verschiedenen Bilder die gleichen

Zeilenkoordinaten besitzen. Diese Geometrie kann erreicht werden, wenn die beiden optischen Achsen der Kameras parallel verlaufen. Ist dies gegeben, muss noch das Gesamtkoordinatensystem gewählt werden. Hierbei ist es sinnvoll, den Mittelpunkt der beiden optischen Zentren als Nullpunkt zu verwenden, sodass die X- und Y-Achse parallel zu den Bildebenen liegt [Jiang et al. 1997, 8–9]. Eine grobe Skizze einer solchen Kamerageometrie ist in Abbildung 11 zu sehen.

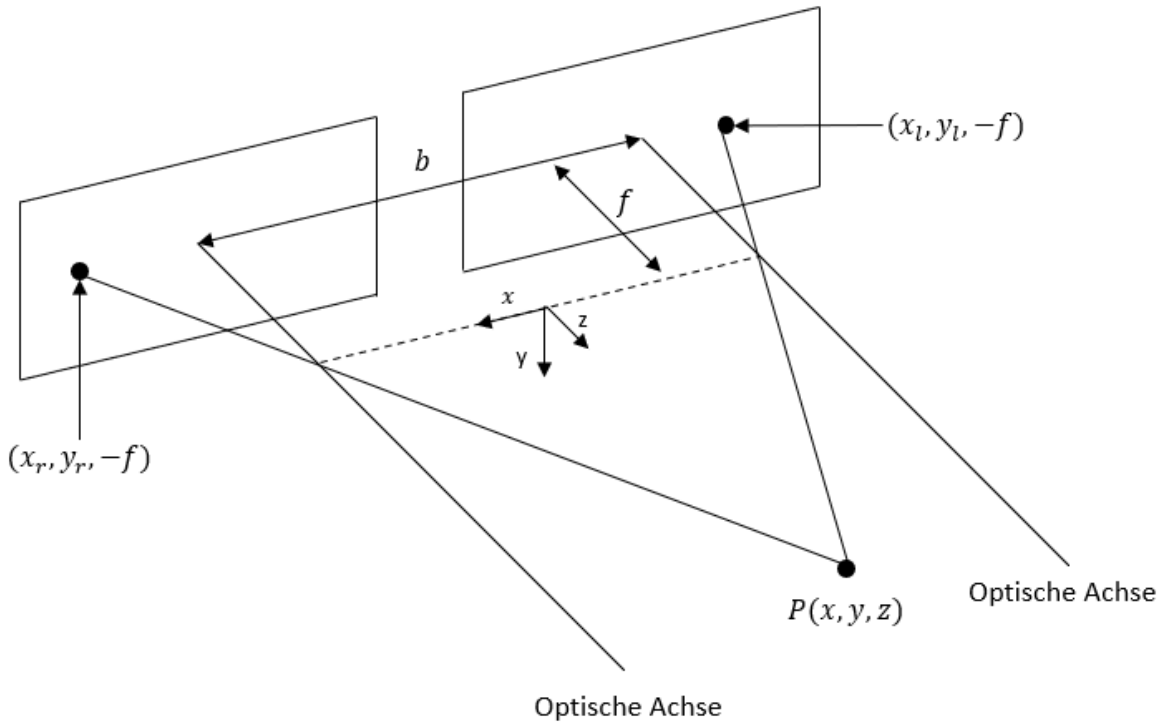


Abbildung 11: Optimale Kamerageometrie [Jiang et al. 1997, 10]

Diese Geometrie erleichtert zwar deutlich die Berechnung der zusammenhängenden Bildpunkte und somit die Messung der Tiefe, sie bringt aber auch ihre Nachteile mit sich. Zum einen ist es nur sehr schwer erreichbar eine solche ideale Geometrie in der realen Welt nachzubilden. Zum anderen kommt dazu, dass die Genauigkeit des Verfahrens abhängig ist von der Basislänge *b*. Die Genauigkeit der Messungen steigt mit dem Erhöhen der Basislänge [Jiang et al. 1997, 9]. Diese kann aber nur zu einem gewissen Maß erhöht werden, da sonst die Voraussetzung der Standard-Geometrie nicht mehr erfüllt ist, dass die beiden Kamerabilder die identische Szene abbilden. Deshalb ist es in der Praxis oft sinnvoll, die Kameras um eine Achse zueinander zu drehen [Jiang et al. 1997, 9].

3.3.2. Lochkamera und Projektion

Als mathematisches Modell für die Berechnung wird der Einfachheit halber in der Regel das Modell der Lochkamera genutzt [Jiang et al. 1997, 8]. In diesem Modell werden die Punkte aus der 3D-Welt über das optische Zentrum, auf eine Bildebene abgebildet. Durch die Projektion

durch den Brennpunkt erfolgt eine Punktspiegelung durch diesen [Schreer 2005, 41]. Eine Skizze einer solchen Lochkamera ist in Abbildung 12 zu sehen.

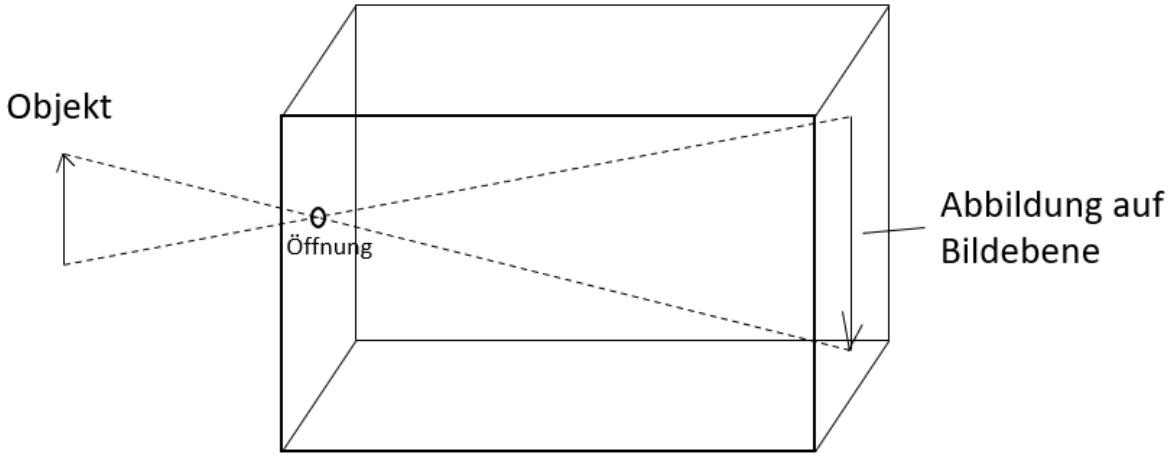


Abbildung 12: Lochkamera

Durch den Brennpunkt, der in diesem Modell eine minimale Breite besitzt, entfällt die Tiefenunschärfe, wie sie bei der Projektion durch eine Linse mit bestimmter Breite zu erkennen wäre. Der Unterschied der Projektion auf die Bildebene für verschiedene Lochbreiten ist in Abbildung 13 zu sehen. Hier ist auf der linken Seite eine Projektion mit kleiner und rechts mit größerer Öffnung der Linse zu sehen. Wie in der Abbildung zu sehen ist, kann der gleiche Punkt durch die Öffnung auf verschiedene Stellen der Bildfläche projiziert werden. Auf diese Weise entsteht eine Tiefenunschärfe. Bei dem Modell der Lochkamera ist diese Öffnung unendlich klein, sodass jeder Punkt nur auf eine bestimmte Stelle der Bildfläche projiziert werden kann und dadurch keine Tiefenunschärfe entsteht.

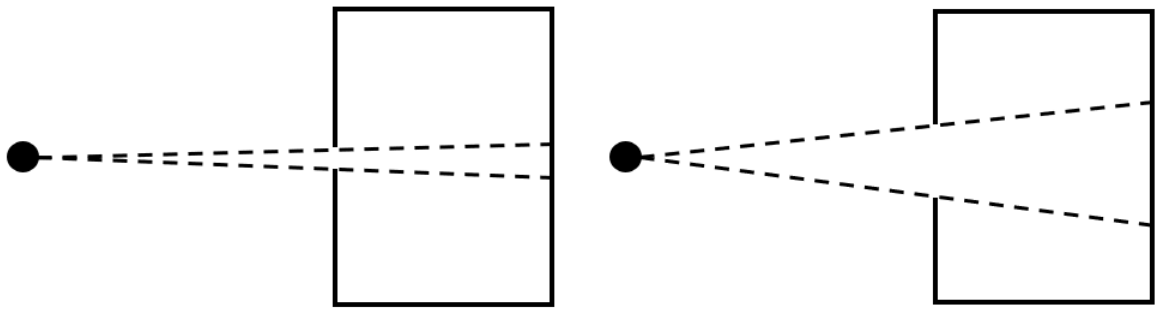


Abbildung 13: Tiefenschärfe bei verschiedenen Lochbreiten

Durch das sehr kleine Loch in dem Modell würde in der Realität aber viel zu wenig Licht eintreten. Deshalb wird im Normalfall ein Objektiv benötigt [Süße et al. 2014, 297]. Das Objektiv vergrößert die Blendenöffnung und sorgt somit dafür, dass genug Licht eintritt. Dies bringt aber den Nachteil mit sich, dass die Kamerabrennweite nicht identisch mit der Linsenbrennweite des Objektivs ist. Die Entfernung zum Objekt spielt also ebenfalls mit in die Brennweite ein, was in Formel (19) zu sehen ist [Süße et al. 2014, 297].

$$\frac{1}{f_L} = \frac{1}{f} + \frac{1}{d_{Objekt}} \tag{19}$$

An der Formel ist zu sehen, dass der Unterschied in der Brennweite geringer wird, je weiter das Objekt weg ist. Durch diesen Zusammenhang ist es trotzdem möglich die Lochkamera als Modell zu nutzen.

Um die 3D-Koordinaten eines Objekts auf die Bildebene zu projizieren muss als erstes ein Kamerakoordinatensystem gewählt werden. Dies kann so gewählt werden, dass der Ursprung im optischen Zentrum der Kamera liegt. Außerdem können die X- und Y-Achsen parallel zur Bildebene bestehend aus u und v gewählt werden, sodass die z-Achse der Kamera der optischen Achse entspricht. Wie in Abbildung 12 gezeigt, ist das Objekt auf der Bildebene allerdings gespiegelt, was sich in der Rechnung mit negativen Vorzeichen bemerkbar machen würde. Um dies zu umgehen, kann die Bildebene im Abstand f vor das Projektionszentrum geschoben werden. Daraus ergibt sich nach [Süße et al. 2014, 299] das folgende Verhältnis.

$$\frac{u}{f} = \frac{x_K}{z_K}, \qquad \frac{v}{f} = \frac{y_K}{z_K} \tag{20}$$

Durch Umstellen ergeben sich die folgenden Formeln für die Abbildung.

$$u = \frac{x_K \cdot f}{z_K}, \qquad v = \frac{y_K \cdot f}{z_K} \tag{21}$$

Dafür müssen die Punkte allerdings erstmal in das Kamerakoordinatensystem überführt werden. Um dies zu tun, werden die äußeren Kameraparameter (extrinsische Kameraparameter) benötigt, welche die die Lage der Kamera im Weltkoordinatensystem beschreiben. Dazu zählen zum einen die Rotation und zum anderen die Translation der Kamera. Da die Rotation theoretisch um drei Achsen und die Translation ebenfalls auf drei Achsen ausgeführt werden kann, beinhalten die extrinsischen Parameter drei für die Rotation und drei für die Translation, was insgesamt sechs Parameter bedeutet. Die Rotation wird dabei durch eine 3x3 Rotationsmatrix *R* bewerkstelligt, während die Translation über einen dreidimensionalen Vektor durchgeführt wird [Süße et al. 2014, 270].

$$R = R_x \cdot R_y \cdot R_z = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$t = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \tag{22}$$

Da die Rotation immer um den Ursprung durchgeführt wird, muss diese vor der Translation erfolgen. Somit kann die Bewegung vom Weltkoordinatensystem in das

Kamerakoordinatensystem im Allgemeinen durch die folgende Formel beschrieben werden, wobei X_K und X_W einen Punkt im Kamera- bzw. Weltkoordinatensystem beschreiben [Süße et al. 2014, 301].

$$X_K = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = R \cdot X_W + t \tag{23}$$

Für die einfachere Rechnung wird diese Formel in homogenen Koordinaten dargestellt.

$$X_K = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_y \end{pmatrix} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \tag{24}$$

Darauffolgend können die transformierten Koordinaten auf der Bildebene abgebildet werden. Hierfür werden die inneren Kameraparameter (intrinsische Parameter) benötigt. Diese lassen sich aufbauend auf Formel (21) herleiten. Hier muss zuallererst eine weitere Größe anstatt der einfachen Brennweite *f* eingeführt werden. Da die im Bild gemessenen Pixelkoordinaten nicht mit denen der Sensorkoordinaten übereinstimmen und die genaue Sensorgeometrie nicht bekannt ist, wird an Stelle der Brennweite eine Skalierung eingeführt, um diese zu modellieren [Süße et al. 2014, 299]. Dieses Produkt kann sich für die Bildachsen u und v unterscheiden.

$$u = \frac{x_K \cdot s \cdot f_x}{z_K}, \qquad v = \frac{y_K \cdot s \cdot f_y}{z_K} \tag{25}$$

Mit den beiden Produkten aus Skalierungsfaktor und Brennweite und dem Bildhauptpunkt sind bereits vier intrinsische Parameter bekannt. Der Bildhauptpunkt (u_0, v_0) ist der Schnittpunkt der optischen Achse mit der Bildfläche. Da dessen Position nicht bekannt ist, muss die Formel noch erweitert werden. Dazu werden u und v durch die Relativkoordinaten $u - u_0$ und $v - v_0$ ersetzt, was zu folgender Gleichung führt [Süße et al. 2014, 301].

$$u - u_0 = \frac{x_K \cdot s \cdot f_x}{z_K}, \qquad v - v_0 = \frac{y_K \cdot s \cdot f_y}{z_K} \tag{26}$$

Durch Einsetzen von $d = z_K$ und $\alpha = s * f$ ergeben sich die folgenden Gleichungen.

$$u - u_0 = \frac{x_K \cdot \alpha}{d}, \qquad v - v_0 = \frac{y_K \cdot \alpha}{d}, \qquad d = z_K \tag{27}$$

Anschließend kann u_0 addiert und mit d multipliziert werden, um die folgenden Gleichungen zu erhalten:

$$d \cdot u = \alpha_x \cdot x_K + u_0 \cdot z_K \tag{28}$$

$$d = z_K$$

Diese Gleichungen können in der Matrixform aufgeschrieben werden [Süße et al. 2014, 300].

$$\begin{pmatrix} d \cdot u \\ d \cdot v \\ d \end{pmatrix} = \begin{pmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} \alpha_x & 0 & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_k \\ y_k \\ z_k \end{pmatrix} = K \cdot x_k \tag{29}$$

Wie hier zu sehen ist, beinhaltet die Kameramatrix K nur vier Parameter. Hier kommt allerdings noch hinzu, dass durch die Sensorgeometrie eine Scherung verursacht werden kann, sodass die Basisvektoren der Bildachsen ggf. nicht mehr senkrecht aufeinander stehen. Der Winkel zwischen diesen Parametern ist der fünfte intrinsische Parameter α_s [Süße et al. 2014, 300]. Daraus ergibt sich dann die allgemeine Abbildung.

$$K = \begin{pmatrix} \alpha_x & \alpha_s & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{30}$$

Nachdem sowohl die intrinsische, als auch die extrinsische Matrix bekannt ist, können diese zusammengeführt werden. Dafür muss als erstes die Matrix K in die homogene Matrix $\widetilde{K}$ überführt werden.

$$\widetilde{K} = \begin{pmatrix} \alpha_x & \alpha_s & u_0 & 0 \\ 0 & \alpha_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{31}$$

Dadurch kann die Formel (24) in Formel (29) eingesetzt werden, um eine Abbildung eines Punktes vom Weltkoordinatensystem in die Bildebene zu erhalten.

$$\begin{pmatrix} d \cdot u \\ d \cdot v \\ d \end{pmatrix} = \begin{pmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} \alpha_x & \alpha_s & u_0 & 0 \\ 0 & \alpha_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_y \end{pmatrix} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \tag{32}$$

Mit der homogenen Matrix der intrinsischen Parameter K und der Transformationsmatrix $\widetilde{RT}$, sowie den Koordinaten in der realen Welt $\widetilde{x_W}$ ergibt sich dadurch der folgende Zusammenhang.

$$\tilde{u} = \tilde{A} \cdot \widetilde{x_W} \quad mit\ \tilde{A} = \widetilde{K} \cdot \widetilde{RT} \tag{33}$$

3.3.3. Kalibrierung

Für eine Kamera in einer bestimmten Position sind die Kameraparameter im Allgemeinen nicht bekannt. Um die intrinsischen und extrinsischen Parameter der Kameras zu ermitteln, muss eine Kalibrierung dieser durchgeführt werden. Für die Kalibrierung der Kameras gibt es verschiedene Wege, die unterschiedliche Voraussetzungen benötigen. Für die Kalibrierung der Kamera wird die allgemeine Gleichung der Lochkamera aus Formel (33) verwendet.

3.3.3.1. Direkte Kalibrierung

Für die Kalibrierung kann ein Kalibrierkörper genutzt werden, dessen Geometrie möglichst präzise bekannt sein sollte. Dieser Körper besteht in der Regel aus zwei oder drei Ebenen, die Orthogonal aufeinander stehen [Zhang 2000, 1]. Auf diesem Körper müssen mindestens 6 Merkmale zu sehen sein, die eindeutig durch die Kamera zu erkennen sein müssen. Durch die erforderliche Präzision ist eine sehr genaue Arbeit nötig, was die Umsetzung deutlich erschwert.

Die Kalibrierung erfolgt ebenfalls in dem Modell der Lochkamera. In diesem gibt es sechs extrinsische und fünf intrinsische Parameter die ermittelt werden müssen. Die genaue Funktion der einzelnen Parameter ist in Kapitel 3.3.2 beschrieben. Insgesamt gibt es also 11 Unbekannte. Aufgrund von Formel (25) entstehen pro Kamera und Bildpunkt zwei Gleichungen, sodass mindestens sechs Punkte mit bekannten Koordinaten benötigt werden, um das Gleichungssystem zu lösen [Süße et al. 2014, 340]. Sind nicht nur die Bildkoordinaten, sondern auch die drei Kamerakoordinaten bekannt, verringert sich die Anzahl der benötigten Punkte noch weiter auf vier. Dies wird in Kapitel 5.8 näher erklärt.

Da die exakten Punkte der 3D-Koordinaten bekannt sind, müssen diese nur noch auf den Bildern detektiert werden, sodass auch die Bildkoordinaten bekannt sind. Daraus können dann nacheinander die intrinsischen und extrinsischen Parameter der Kamera berechnet werden.

3.3.3.2. Selbstkalibrierung

Da der erste Ansatz oft sehr aufwändig ist, wird selbstverständlich auch nach Ansätzen gesucht, wie eine Kamera sich selbst kalibrieren kann, ohne dass ein aufwändiger Kalibrierkörper vorhanden ist. Um dies zu bewerkstelligen, könnten die Koordinaten der Merkmale ebenfalls als Unbekannte in das Gleichungssystem einfließen [Süße et al. 2014, 344]. Dadurch erweitert sich die Anzahl der Unbekannten um elf pro zusätzlicher Kamera und drei pro zusätzlichem Punkt. Da pro Punkt und Kamera allerdings nur zwei Gleichungen entstehen, lässt sich schnell erkennen, dass eine solche Kalibrierung mit nur einer Kamera nicht funktionieren kann. Durch die Nutzung von mehreren Kameras, wie zum Beispiel in einem Stereosystem, ändert sich dies.

Der folgende Zusammenhang zeigt, unter welchen Voraussetzungen dies möglich ist. Die Anzahl der Kameras wird dabei durch k und die Anzahl der Punkte durch p modelliert.

$$11 \cdot k + 3 \cdot p \leq 2 \cdot p \cdot k \tag{34}$$

Wie an der Gleichung zu erkennen, steigt die Anzahl der Unbekannten bei nur einer Kamera schneller als die Anzahl der Gleichungen, wenn die Anzahl der Punkte erhöht wird. Aus diesem Grund kann diese Art der Kalibrierung nicht mit nur einer einzelnen Kamera funktionieren. In einem System mit zwei Kameras sind beispielsweise mindestens 22 Punkte nötig sind, um das Gleichungssystem lösen zu können. Mit 22 Punkten, die in beiden Kameras klar erkennbar und zuzuordnen sind, kann darüber ein System mit zwei Kameras selbstständig kalibriert werden. Die Anzahl der Punkte kann allerdings noch etwas reduziert werden. Als erstes kann das Weltkoordinatensystem in die erste Kamera gelegt werden, sodass die äußeren Parameter dieser Kamera frei wählbar sind. Des Weiteren kann die Skalierung frei gewählt werden, sodass insgesamt 7 frei wählbare Parameter entstehen [Süße et al. 2014, 344]. Somit kann die Anzahl der benötigten Punkte von vorher 22 auf nur 15 reduziert werden.

Dadurch, dass allerdings deutlich mehr Punkte benötigt werden, als noch bei der Kalibrierung mit Hilfe eines Kalibrierkörpers, steigt selbstverständlich auch der Rechenaufwand deutlich an. Außerdem kann hierbei nicht die absolute Skalierung von 3D-Objekten bestimmt werden, da ohne den Kalibrierkörper keine metrische Einheit und somit auch keine definitiven Längen bekannt sind [Süße et al. 2014, 344]. Aus diesem Grund werden bei einer solchen Kalibrierung meist auch nur die inneren Kameraparameter ermittelt.

Alternativ kann eine ähnliche Kalibrierung erfolgen, indem eine Kamera in einer statischen Szene bewegt wird und die identische Szene somit aus verschiedenen Positionen aufgenommen wird. Wenn die Kamera feste intrinsische Parameter besitzt, reichen in diesem Fall bereits drei Bilder aus, um die intrinsischen und extrinsischen Parameter zu bestimmen [Zhang 2000, 1].

3.3.3.3. Kalibrierung nach Zhang

Die Kalibrierung nach Zhang [Zhang 2000] funktioniert durch ein deutlich simpleres Kalibrierobjekt. Anstatt eines sehr präzise hergestellten dreidimensionalen Körpers, der aus mehreren Ebenen besteht, wird bei dieser Methode einfach nur eine flache Ebene genutzt, auf die ein Muster gedruckt wird. Für die Kalibrierung muss entweder die Kamera oder das Muster an einer festen Stelle sein und das andere Objekt bewegt werden, um mehrere Aufnahmen aus verschiedenen Positionen aufzunehmen.

Grundsätzlich basiert auch die Kalibrierung nach Zhang auf der Abbildung durch die Lochkamera nach Formel (33). Für diese Methode wird angenommen, dass die abgebildete Ebene auf $z = 0$ liegt. Dies führt zu folgender Gleichung:

$$s \cdot \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \widetilde{K} \cdot (\overrightarrow{r_1} \quad \overrightarrow{r_2} \quad \overrightarrow{r_3} \quad \vec{t}) \cdot \begin{pmatrix} x_w \\ y_w \\ 0 \\ 1 \end{pmatrix} = K \cdot (\overrightarrow{r_1} \quad \overrightarrow{r_2} \quad \vec{t}) \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{35}$$

Die Transformationsmatrix $\widetilde{RT}$ wird dabei durch die Vektoren $\overrightarrow{r_1}, \overrightarrow{r_2}, \overrightarrow{r_3}$ und $\vec{t}$ für die Rotation und Translation beschrieben. Dadurch, dass $z = 0$ gesetzt wird, entfällt $\overrightarrow{r_3}$. So ergibt sich eine vereinfachte Matrix A.

$$A = K \cdot (\overrightarrow{r_1} \quad \overrightarrow{r_2} \quad \vec{t}) \tag{36}$$

Auf Basis dieser Annahme kann die Kamera kalibriert werden. Die genaue Lösung der Kameraparameter ist in [Zhang 2000] beschrieben.

In dieser Arbeit wird nur die direkte Kalibrierung genutzt, um auf diese Weise eine Transformationsmatrix zu finden, mit der sich die gefundenen Kamerakoordinaten wieder in Weltkoordinaten umrechnen lassen. Die Selbstkalibrierung und Kalibrierung nach Zhang dienen hier nur dem Verständnis der Kalibrierungsverfahren der genutzten Tiefenkamera.

3.3.4. Tiefenberechnung

Für die Berechnung der Tiefe der zu sehenden Objekte kann erstmal der Aufbau aus Abbildung 14 angenommen werden.

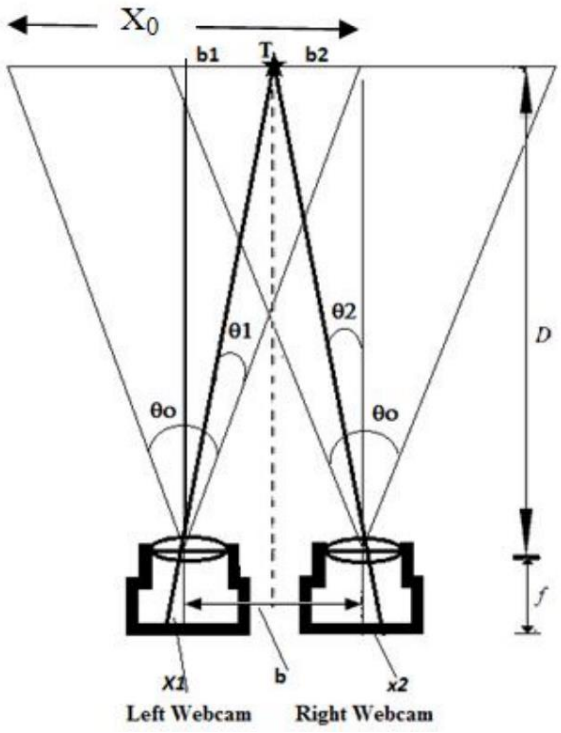


Abbildung 14: Stereogeometrie

Wie in der Abbildung zu sehen, kann für beide Kameras ein rechtwinkliges Dreieck zwischen der Koordinate des Punktes in der realen Welt, der Koordinate auf der Bildebene und der Mitte der Basislinie aufgestellt werden. Dabei ist wichtig, dass beide Kameras die gleiche Brennweite, sowie den gleichen Sichtbereich θ_0 haben. Daraus wird in [Zivingy 2013] die folgende Herleitung ermittelt, wobei x_1 und x_2 jeweils die Distanz von der optischen Achse bis zum jeweiligen Bildpunkt sind und b_1 und b_2 die Distanz vom Ursprung des Koordinatensystems zum Bildhauptpunkt. D ist der Abstand der Kamera zum gemessenen Punkt und f die Brennweite der Kameras.

$$\frac{b_1}{D} = \frac{-x_1}{f}, \frac{b_2}{D} = \frac{-x_2}{f} \tag{37}$$

Da der Ursprung des Weltkoordinatensystems genau in der Mitte der beiden Kameras liegt, kann mit $b = b_1 + b_2$ vereinfacht werden.

$$b = \frac{D}{f} \cdot (x_2 - x_1) \tag{38}$$

Dieser Zusammenhang kann nach der Entfernung D umgestellt werden.

$$D = \frac{b \cdot f}{x_2 - x_1} \tag{39}$$

Aus der vorliegenden Geometrie ergibt sich der folgende Zusammenhang, wobei x_0 die Breite des Bildes ist:

$$\tan\left(\frac{\theta_0}{2}\right) = \frac{x_0}{D} = \frac{x_1}{f} \tag{40}$$

Dieser kann nun nach f umgestellt werden.

$$f = \frac{x_0}{2 * \tan\left(\frac{\theta_0}{2}\right)} \tag{41}$$

Durch Einsetzen der Formel (41) in Formel (39) ergibt sich die folgende Formel für den Abstand des aufgenommenen Punktes. $(x_2 - x_1)$ beschreibt die Disparität, also den Abstand zwischen den beiden Pixeln.

$$D = \frac{b \cdot x_0}{2 \cdot \tan\left(\frac{\theta_0}{2}\right) \cdot (x_2 - x_1)} \tag{42}$$

In der Formel wäre die Distanz umgekehrt proportional zu der Disparität, da alle anderen Terme in der Formel konstant sind. Deshalb muss in der $\tan(\frac{\theta_0}{2})$ Gleichung noch der Term ϕ eingefügt werden [Zivingy 2013].

$$D = \frac{b \cdot x_0}{2 \cdot \tan(\frac{\theta_0}{2} + \phi) \cdot (x_2 - x_1)} \tag{43}$$

3.4. Genutzte Hardware und Tools

In diesem Kapitel wird die in diesem Projekt genutzte Hard- und Software kurz vorgestellt und teilweise erklärt, warum sich für dieses Tool entschieden wurde. Aufbauend darauf wird das Konzept der Arbeit in Kapitel 4 vorgestellt. Die Versionen der genutzten Software können der Installation in Anhang A entnommen werden.

Visual Studio Code

Visual Studio Code[1] ist ein kostenloser Quelltext-Editor von Microsoft, der plattformübergreifend zur Verfügung steht. Dieser wird für die Aufgabe in Kombination mit Python genutzt. Die Entwicklungsumgebung kann sehr leicht durch Erweiterungen angepasst werden und ist daher sehr flexibel und für verschiedene Projekte geeignet.

Python

Als Programmiersprache für das Projekt wird Python[2] verwendet. Python ist die mittlerweile drittbeliebteste Programmiersprache [TIOBE Software BV 2021] und wird vor allem in der Wissenschaft für viele verschiedene Arbeiten genutzt. Vor allem im Bereich der Data Science und Machine Learning wird sie aufgrund ihrer einfachen Handhabung und Lesbarkeit, sowie leichter Erweiterbarkeit genutzt [Parbel 2019]. Dies führt zu einer starken Bindung zwischen Machine Learning Projekten und der Programmiersprache Python. Als Vorteile von Python sind vor allem die große Anzahl an vorinstallierten und verfügbaren Bibliotheken, die einfache Syntax und somit auch die einfache Lesbarkeit, sowie die Flexibilität auf vielen verschiedenen Plattformen zu laufen [Oliphant 2007]. Ein weiterer Vorteil ist die große Community, die immer wieder als Hilfestellung genutzt werden kann und bereits unzählige Lösungen für alle möglichen Probleme bereitstellt. Zusätzlich wird die Sprache auch von weiteren Projektpartnern genutzt. Durch die einheitliche Nutzung von Python können mögliche Abstimmungsprobleme vermieden und die Kompatibilität sichergestellt werden.

[1] https://visualstudio.microsoft.com/de/
[2] https://www.python.org/

Anaconda

Anaconda[3] ist eine Open-Source Distribution für die Programmiersprache Python. Sie dient dem einfachen Paketmanagement und wird ebenfalls viel für verschiedene Machine Learning und Data Science Projekte genutzt. In Anaconda kann für ein bestimmtes Problem eine neue Umgebung erschaffen werden, die ihre eigenen Paketversionen nutzt. So können für verschiedene Projekte verschiedene Umgebungen erstellt werden, für die vollkommen unterschiedliche Versionen der gleichen Bibliotheken genutzt werden. Auch die Entwicklungsumgebungen, wie zum Beispiel Visual Studio Code, können direkt auf diese Umgebungen zugreifen und die dort installierten Versionen nutzen.

Numpy

Numpy[4] (Numerical Python) ist eine äußerst wichtige Bibliothek für verschiedenste Anwendungen im Bereich des Machine Learnings. Numpy bietet eine Menge performanter Funktionen, mit Hilfe derer Berechnungen auf Datenstrukturen wie Arrays und Matrizen ausgeführt werden können. Numpy ist quasi schon eine Standardbibliothek des Machine Learnings und wird von vielen anderen Frameworks verwendet.

Scikit-learn

Scikit-learn[5] ist ein freistehendes Machine Learning Framework, das für Python zur Verfügung steht. Es baut auf Numpy, SciPy und Matplotlib auf. Die Bibliothek bietet viele verschiedene Machine Learning Ansätze für Regression, Klassifikation und Clustering. Es bietet viele verschiedene Modelle, zwischen denen ausgewählt werden kann und bringt außerdem noch die Möglichkeit mit, die Daten für die Modelle schon vorverarbeiten zu können [Deru et al. 2020, 45].

OpenCV

OpenCV[6] ist die wahrscheinlich am meisten verbreitete Bibliothek für die Bildverarbeitung. Sie beinhaltet zahlreiche verschiedene Methoden für diverse verschiedene Einsatzgebiete. Diese reichen von einfachen Bildmanipulationen, wie das Schneiden von Bildern oder Ändern der Größe, über die Erzeugung von 3D-Punktwolken aus Stereobildern bis hin zu der Erkennung von Gesichtern oder anderen Objekten. Es gibt unglaublich viele Einsatzmöglichkeiten, sowie eine sehr gute Dokumentation für die bestehenden Methoden und Klassen, was es sehr einfach macht damit zu arbeiten. Durch die Möglichkeit von OpenCV ganz einfach Bilder zu schneiden, verkleinern und anzupassen, sowie verschiedene Modelle für Objekterkennung einzulesen, ist die Bibliothek perfekt für das Projekt geeignet.

[3] https://anaconda.org/
[4] https://numpy.org/
[5] https://scikit-learn.org/stable/index.html
[6] https://opencv.org/

Google Colab

Google Colab[7] bietet die Möglichkeit im Webbrowser ein Notebook zu erstellen, in dem Python-Code geschrieben und ausgeführt, sowie Bilder und Text hinzugefügt und formatiert werden kann. Dadurch bietet sich die Möglichkeit einen Ablauf mit Kommandozeilenbefehlen und Programmcode strukturiert niederzuschreiben und durch Text und Bilder ansehnlich zu präsentieren. Die Notebooks basieren dabei auf Jupyter-Notebook und können somit ganz einfach mit anderen geteilt werden. Ein großer Vorteil von Google Colab ist, dass der geschriebene Code nicht auf der lokalen Maschine, sondern auf virtuellen Maschinen auf dem Server von Google ausgeführt wird. Dies ermöglicht viele Möglichkeiten in Wissenschaft und Bildung, da so prinzipiell jeder auf rechenintensive Methoden wie Machine Learning zurückgreifen kann, ohne dafür selber die teure Hardware zu besitzen. Die Limitierung liegt darin, dass nur eine gewisse Rechenzeit zur Verfügung steht, nach dieser der Dienst für kurze Zeit nicht mehr nutzbar ist. Dieser wird jedoch nach einer Wartezeit von ca. einem Tag wieder freigeschaltet, sodass mit dem Ausführen der Programme fortgefahren werden kann.

YOLOv4 und Darknet[8]

Als erstes muss überlegt werden, wie die Objekterkennung durchgeführt werden soll. Eine Möglichkeit wäre es, mittels OpenCV verschiedene Bilderkennungsalgorithmen zu nutzen, um Merkmale herauszufiltern und daraus die Objekte zu erkennen [Funke 2016]. Aufgrund dessen, dass die Becher allerdings in verschiedenen Farben, Distanzen und Belichtungen auftreten können, ist es schwer eine robuste Erkennung mittels OpenCV zu programmieren, um die Merkmale herauszufiltern und Objekte dieser Art zu erkennen [Zhao et al. 2019, 1]. Aufgrund dieser Eigenschaften wird die Objekterkennung in diesem Projekt mittels neuronaler Netze ausgeführt. Diese sind sehr robust und lernen von selbst, wie sie bei unterschiedlichsten Bedingungen Merkmale extrahieren und dadurch die Objekte erkennen können. Dafür stehen mehrere Modelle zur Auswahl, zwischen denen ausgewählt werden muss.

YOLOv4 ist die vierte Version des berühmten Objekterkennungs-Modells YOLO (You only look once). Entwickelt wurde YOLO von Alexey Bochkovskiy. Ziel des Modells ist es eine möglichst gute Genauigkeit zu erreichen, aber trotzdem noch auf normalen Systemen lauffähig zu bleiben. Der Kompromiss aus Geschwindigkeit und Genauigkeit wird oft durch die zur Verfügung stehende Hardware und die Art der Anwendung vorgegeben.

In diesem Projekt hat ebenfalls die Genauigkeit Vorrang vor der Geschwindigkeit, sodass das beste Modell verwendet werden kann, das auf der zur Verfügung stehenden Hardware läuft. Die genausten Ergebnisse werden, wie in Kapitel 3.2.4 beschrieben, von Zweistufen-Detektoren erreicht. Durch die zwei Stufen sind diese Modelle äußerst Rechenaufwendig, sodass sie nur auf leistungsfähigen GPUs oder TPUs laufen. Das Gegenstück dazu sind Modelle, die nur einen Durchlauf benötigen und anhand dessen die Position der Objekte

[7] https://colab.research.google.com/
[8] https://github.com/AlexeyAB//darknet

bestimmten. In diesen Bereich fällt zum Beispiel das YOLO-Modell, aber auch das EfficientDet von Google, das äußerst gute Ergebnisse erzielt [M. Tan et al. 2020].

Da das Modell auch auf CPUs laufen können soll, die mit ihrer Leistung nicht an die der großen GPUs herankommen, muss hier eines der schnelleren Modelle genutzt werden. Im Kern kommen dabei die beiden bereits erwähnten Modelle YOLO und EfficientDet in Frage. Im Vergleich mit anderen Netzen zeigt sich das YOLOv4 als ein sehr guter Kompromiss aus Genauigkeit und Geschwindigkeit [Bochkovskiy et al. 2020]. Dies ist auch nochmal in Abbildung 15 zu sehen. Dort sind verschiedene Modelle mit ihrer Geschwindigkeit und Genauigkeit dargestellt. Das YOLOv4 erreicht dabei zwar nicht die genausten Ergebnisse, arbeitet aber bei ausreichender Genauigkeit deutlich schneller als die anderen Modelle.

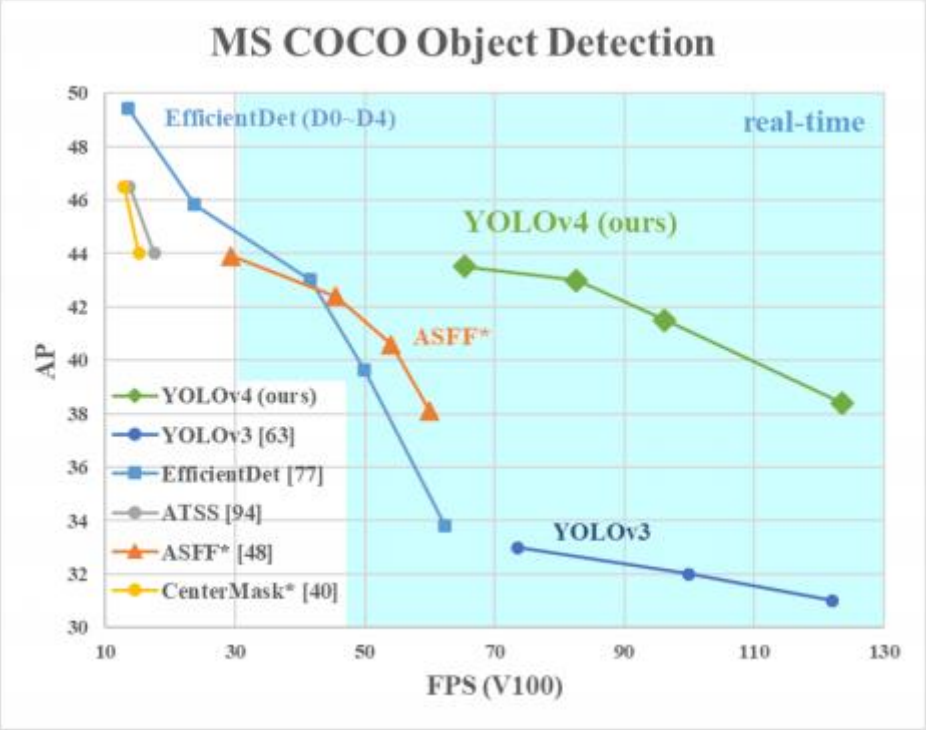


Abbildung 15: Präzision und Geschwindigkeit von verschiedenen Objekterkennungsmodellen

Da es sich bei der Bechererkennung um ein sehr einfach zu erlernendes Problem handelt, reicht die Genauigkeit des YOLOv4 für das Projekt aus. Das Modell sollte die Becher problemlos erkennen und läuft deutlich schneller als das EfficientDet, sodass die Anwendung schneller auf Positionsänderungen des Bechers reagieren kann. Die Geschwindigkeit sorgt dafür, dass keine weitere besondere Hardware für das Projekt genutzt werden muss. Im Gegensatz zum etwas neueren Modell YOLOv5 hat das v4 den Vorteil, dass es deutlich mehr Beispiele und Erfahrungen für die Nutzung des Modells gibt. Um das YOLO-Modell zu trainieren wird zusätzlich noch ein Framework benötigt. Grundsätzlich kann das Modell in vielen Frameworks wie Tensorflow oder Pytorch trainiert werden [Bochkovskiy 2020]. Standardmäßig ist vom Entwickler allerdings Darknet vorgesehen. Darknet ist ein Open Source Network und wurde in C und CUDA geschrieben. Es unterstützt sowohl das Training mit CPU, als auch mit GPU [Bochkovskiy 2020].

LabelImg

LabelImg[9] ist ein einfaches grafisches Tool zum labeln von Bildern. Die Bilder können geladen und in der grafischen Oberfläche mit Boxen für die jeweiligen Objekte bestückt werden, die auf dem Bild zu sehen sind. Außerdem kann das Tool direkt die zu den Bildern gehörenden Dateien mit den Koordinaten erstellen. Dies sind entweder xml- oder txt-Dateien, je nachdem welches Modell für die Erkennung genutzt wird.

Intel Realsense D415[10]

Es gibt zahlreiche optische Methoden um eine Distanzmessung durchzuführen. Viele davon sind in Abbildung 1 abgebildet. Da für die Umsetzung des Projekts allerdings sowieso eine Kamera benötigt wird, um eine Objekterkennung durchzuführen, wäre es naheliegend diese Kamera auch für die Messung der Distanz zu nutzen. Dies schließt einige der in Abbildung 1 gezeigten Verfahren bereits aus. Ein Problem dabei ist allerdings, dass sich aus dem Bild einer einzelnen Kamera keine bzw. nur sehr wenig 3D-Informationen ableiten lassen. Es können lediglich wenige Informationen über die den dreidimensionalen Raum zurückgewonnen werden, wenn die Nebenbedingungen beschränkt und bekannt sind [Jiang et al. 1997, 39].

Besser hingegen wäre die Nutzung mehrerer Kameras, anhand derer mittels Triangulation der Abstand ermittelt werden kann (siehe Kapitel 3.3.4). Aus zwei Kameras lassen sich grundsätzlich genügend 3D-Informationen sammeln um diese Aufgabe zu erfüllen. Hierbei muss zwischen dem Erstellen eines eigenen Stereo-Aufbaus mit zwei Kameras mit festem Abstand, oder dem Verwenden einer käuflichen Tiefenkamera gewählt werden. Das Erstellen des eigenen Aufbaus hat den Vorteil, dass der Aufbau eigenständig bestimmt werden kann. Die Art und Auflösung der Kamera, sowie die Basislänge zwischen den Kameras kann frei gewählt werden, um möglicherweise genauere Ergebnisse zu erzielen. Der Aufbau muss allerdings möglichst genau und nicht mehr veränderbar sein, was die Umsetzung etwas erschwert.

Die Nutzung einer bereits fertigen Kamera hingegen hat den Vorteil, dass diese mit möglichst genauen Verfahren gebaut wurde. Ein weiterer Vorteil sind die häufig zu der Kamera mitgelieferten Bibliotheken und Software, die eine Nutzung der Kamera durch vorgeschriebene Methoden deutlich vereinfachen. Kameras, die diese Funktionalitäten mitbringen sind zum Beispiel die Intel Realsense und die Microsoft Azure Kinect. Beide Kameras haben die Möglichkeiten sowohl ein Farbbild, als auch das Tiefenbild der Szene weiterzugeben. Die Azure Kinect liegt preislich allerdings weit über den Intel Realsense Modellen. Wie den Datenblättern ‚Microsoft_Kinect.pdf' und ‚Intel_Realsense_D400.pdf' in den Anlagen entnommen werden kann, ist die Genauigkeit der Realsense Kameras auf kurze Distanz nur minimal schlechter als die der Kinect. Somit wird das zum Stand 2021 deutliche günstigere Modell aus der Realsense Reihe verwendet. Unter den Realsense-Modellen gibt es ebenfalls noch Unterschiede. Die beiden Modelle, die für dieses Projekt in Frage kommen, sind die D415 und die D435. Die

[9] https://github.com/tzutalin/labelImg
[10] https://www.intelrealsense.com/introducing-intel-realsense-d400-product-family/

D415 ist dabei allerdings günstiger als die D435. Noch dazu kommt, dass die D435 eher für längere Strecken ausgelegt ist und einen größeren Field of View (FOV) besitzt, der für dieses Projekt nicht benötigt wird. Dadurch hat die D415 weniger Tiefenrauschen auf kurze Entfernungen [Kershaw et al. 2019b]. Da sich die zu messenden Becher aber in aller Regel nicht mehr als 2m entfernen, kann hier die D415 aufgrund des geringeren Tiefenrauschens und des günstigeren Preises verwendet werden.

Die Intel Realsense D415 ist einem selbsterstellten Aufbau in vielen Bereichen überlegen. Ein Punkt, der bisher noch nicht erwähnt wurde, ist die Nutzung von Infrarot-Kameras und eines Emitters. Die Intel Realsense nutzt eine normale RGB-Kamera für den einfachen Bildstream, besitzt aber zwei Infrarotkameras für die Erstellung der Tiefenmessung. Der Emitter beleuchtet die Szene zusätzlich noch mit einem Muster, das den Kameras zusätzlich dabei hilft gleiche Bildpunkte zu erkennen [Kershaw et al. 2019a]. In Abbildung 16 wird die ausgewählte Tiefenkamera gezeigt.

Abbildung 16: Abbildung der genutzten Tiefenkamera [vgl. ‚Datenblatt_Realsense.pdf' in den Anlagen]

4. Konzepterstellung

In diesem Kapitel werden die grundsätzlichen Konzepte herausgestellt, mit Hilfe derer die Implementierung des Programms später umgesetzt wird. Dazu gehören unter anderem der Aufbau, sowie ein einfacher Ablauf und die Auswahl der verwendeten Hard- und Softwarelösungen.

4.1. Aufbau des Modells

Wie bereits in der Einleitung erwähnt, handelt es sich hier um einen Teil eines größeren Projektes, das in einem studentischen Team modular umgesetzt wird. Das Gesamtprojekt soll

ein Roboterarm werden, der selbständig dazu in der Lage ist Tischtennisbälle in vor ihm stehende Becher zu werfen. Dieses Teilprojekt beschäftigt sich damit, mit Hilfe einer Tiefenkamera die Becher zu erkennen und die genauen Koordinaten des Bechers zu berechnen, um diese später an den Roboterarm zu übergeben. Durch den Aufbau des Gesamtprojekts ergeben sich automatisch mehrere Grundvoraussetzungen, die eine Umsetzung erfüllen muss. Da die zu erkennenden Becher in einem Bereich von ca. 1-2 m zum Roboter stehen werden, ist es sinnvoll die Kamera auf der gleichen Ebene bzw. mit einem bestimmten Abstand über dieser anzubringen. Eine Positionierung auf oder hinter dem Roboterarm ist nicht sinnvoll, da sich dadurch die ständig die Kameraposition ändern würde und die berechnete Transformationsmatrix zur Umrechnung der Koordinaten so ihre Richtigkeit verliert. Ein starrer Aufbau der Kamera ist wichtig, damit sich keine Parameter ändern können und so zu Ungenauigkeiten führen. Ggf. kann die Kamera auch an einem starren Teil des Roboters angebracht werden. Wichtig dabei ist aber, dass die Kamera ihre Position und Rotation nicht ändert. Daraus resultiert der in Abbildung 17 skizzierte Aufbau.

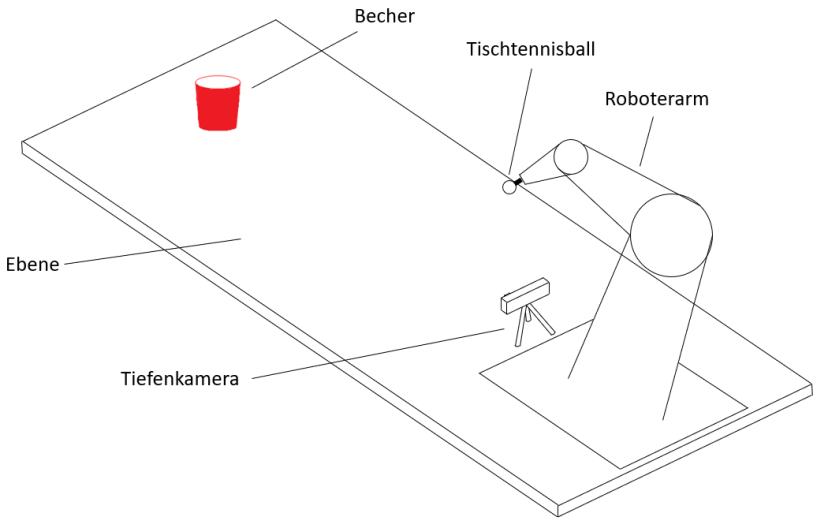


Abbildung 17: Skizze des Aufbaus des Projekts

Zusätzlich sollten die Fehler der am Ende bestimmten Koordinaten 0,5 cm möglichst nicht überschreiten, da die zu treffenden Becher einen Durchmesser von 9,8 cm besitzen und eine größere Abweichung das Treffen der Bälle deutlich erschweren würde.

Durch den oben beschriebenen Aufbau, sowie die grundsätzlichen Anforderungen an das Projekt ergeben sich außerdem auch schon verschiedene Voraussetzungen für die genutzte Hard- und Software. Der mit Abstand wichtigste Aspekt bei der Auswahl ist die Genauigkeit der Ergebnisse. Nur wenn die berechneten Koordinaten des Bechers eine gewisse Genauigkeit aufweisen, kann der Roboterarm diesen treffen. Die Geschwindigkeit des Programms ist hingegen zweitrangig. Dadurch, dass der Wurf an sich schon ein langsamer Prozess ist und dieser auch nicht in einer sehr hohen Frequenz ausgeführt wird, besteht für die Anwendung keine Notwendigkeit für hohe Arbeitsgeschwindigkeiten. Bei der Auswahl kann also grundsätzlich die Genauigkeit als oberste Priorität angesehen werden, solange das Programm

dabei noch auf einfacher Hardware läuft. Als Richtwert einer Geschwindigkeit, die nicht weit unterschritten werden sollte, kann hier 1 Bild pro Sekunde (FPS) angenommen werden. So reicht bei Bedarf eine Sekunde um neue Daten über die Position eines Bechers zu bekommen, was in dieser Anwendung vollkommen ausreichend ist. Wenn der Becher unbewegt an einer festen Position steht, reicht theoretisch auch eine einzige Messung zu Beginn, die dann erst wiederholt werden muss, wenn die Position des Bechers verändert wird. Die in diesem Projekt genutzten Tools und Hardware sind in Kapitel 3.4 beschrieben.

4.2. Programmablauf

Das Erkennen der Becher und die dazugehörige Messung der genauen Position sind unerlässlich für die weitere Umsetzung des Projekts. Nur wenn der Roboterarm auch die genaue Position bzw. den Abstand des Bechers übermittelt bekommt, kann dieser erst die Wurfparabel errechnen und den Wurf ausführen. Die Alternative wäre die Positionen händisch zu messen und an den Roboter zu übergeben, was einen deutlich größeren Aufwand zur Folge hätte. Der gesamte Ablauf besteht aus der Vorbereitung, in der die Kamera neu kalibriert und die Transformationsmatrix für die Umwandlung der Koordinaten bestimmt wird und dem Programm, das letztendlich immer wieder die Becher erkennt, die Distanz misst und daraus die Koordinaten berechnet. Der Gesamtablauf ist dabei immer der gleiche und ist in Abbildung 18 zu finden.

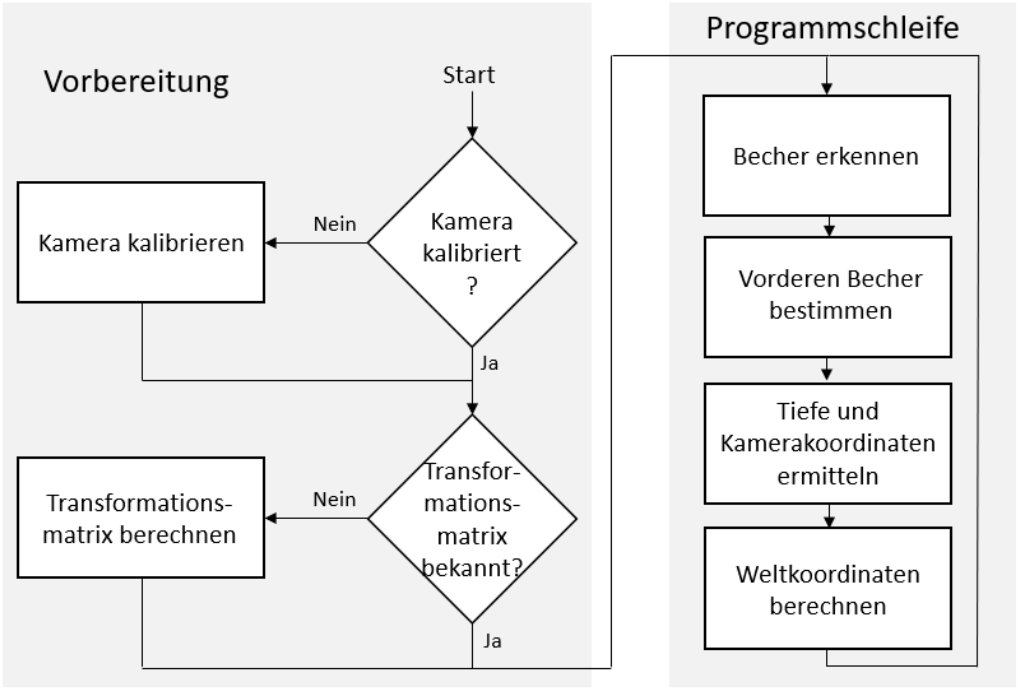


Abbildung 18: Programmablauf

Als erstes muss, falls das Bild verrauscht ist oder die Kameraposition verändert wurde, eine Kalibrierung, wie in Anhang B aufgezeigt, durchgeführt werden. Die kalibrierte Kamera kann nun an einer geeigneten Stelle platziert und mit Hilfe des Kalibriermusters für die Position in den Weltkoordinaten kalibriert werden. Dies ist wichtig, da die Koordinaten der Kamera ihren Ursprung in der Kamera haben und die Daten ohne weitere Infos über die Position nicht weitergenutzt werden können. Bei dieser Bestimmung der Position wird eine Transformationsmatrix berechnet, mit der die Kamerakoordinaten in Weltkoordinaten umgerechnet werden, um diese für die Weiterverarbeitung nutzbar zu machen. Sobald die Matrix berechnet wurde, kann das Programm gestartet werden und die Kamerastreams beginnen. Auf dem Farbbild wird die vorher trainierte und eingelesene Objekterkennung ausgeführt, um die Becher im Bild zu finden. Aus den erkannten Bechern wird ermittelt, welcher sich am weitesten vorne befindet um eine ideale Entfernungsmessung zu gewährleisten. Wurde der entsprechende Becher ermittelt, wird anhand der Bildkoordinaten die Distanz zum Becher aus dem Tiefenbild ermittelt und in 3D-Koordinaten des Kamerakoordinatensystems umgewandelt. Da durch die vorherige Kalibrierung die Transformationsmatrix bekannt ist, können die Koordinaten zum Schluss noch in das Weltkoordinatensystem umgewandelt und für die Weitergabe vorbereitet werden.

5. Implementierung des Programms

Nachdem die Tools ausgewählt wurden, kann mit der Programmierung begonnen werden. Die Umsetzung erfolgt entlang des in Abbildung 18 gezeigten Ablaufs. Die Installation der benötigten Software und eine Bedienungsanleitung für das Verwenden der Software sind im Anhang A und B zu finden. Codeausschnitte, die in den folgenden Abschnitten mit <…> gekennzeichnet sind, müssen durch die passende Benutzereingabe ergänzt werden. Die in dem Kapitel beschriebenen Programme sind in den Anlagen unter ‚Hauptprogramme' zu finden.

5.1. Objekterkennung

Als erstes kann mit dem Training der Objekterkennung begonnen werden. Da diese auf Google Colab trainiert wird, muss hier keine weitere Software installiert werden. Für das Training wird lediglich ein Google-Account und der Zugang zu dem Notebook benötigt.

5.1.1. Erstellen des Datensatzes für das Training

Bevor aber mit dem Training begonnen werden kann, werden als erstes Daten benötigt, mit denen der Trainingsalgorithmus gefüttert werden kann. Diese Daten sollten die Objekte in möglichst vielen verschiedenen Positionen, Belichtungen, Größen und Varianten zeigen. Je größer die Variation hier ist, desto besser kann die Objekterkennung am Ende trainiert werden [Chollet 2018, 179–189]. Im Internet existieren viele verschiedene Anbieter, bei denen bereits fertig gelabelte Datensätze für verschiedene Zwecke heruntergeladen werden können. Für die Suche nach bereits vorhandenen Daten kann zum Beispiel Google Dataset Research[11] verwendet werden. Da die Becher für dieses Projekt allerdings eher ein weniger populäres und sehr spezielles Problem beschreiben, sind dafür keine bereits fertigen Daten vorhanden.

Das Sammeln der Daten erfolgt über die Aufnahme von möglichst vielen verschiedenen Bildern. Dafür wurden zwei verschiedene Becherarten mit gleicher Größe, aber unterschiedlicher Farbe gewählt. Diese werden in diversen Positionen und Belichtungen fotografiert. Ebenfalls wichtig dabei ist, dass auch verdeckte Becher dabei sind, da auch bei der späteren Erkennung verdeckte Becher erkannt werden sollen. Es kann auch sinnvoll sein, Bilder hinzuzufügen, auf denen keines der Objekte zu sehen ist. Dadurch kann das Netzwerk zum Beispiel lernen, bestimmte Merkmale nicht dem Objekt zuzuordnen und somit Falscherkennungen zu reduzieren. Im Anhang C sind ein paar Beispielbilder für das Training zu sehen.

Um das Aufnehmen der Bilder etwas zu Beschleunigen gibt es im Wesentlichen zwei Methoden. Zum einen können Videos aufgenommen werden, in denen die Objekte in vielen verschiedenen Szenen zu sehen sind. In diesem Video können in bestimmten Intervallen Frames extrahiert werden sodass innerhalb einer kurzen Zeit sehr viele Bilder entstehen. Ein anderer Weg ist das Bearbeiten von Bildern. Es gibt Tools, die ein Bild verwenden und es auf viele verschiedene Weisen anpassen, um daraus mehrere neue Bilder mit anderen Eigenschaften zu generieren. So können dabei zum Beispiel die Helligkeit, Größe, Rotation, Farben und vieles mehr der Bilder verändert werden. Auf diese Weise kann ein Datensatz künstlich vergrößert werden. Dieses Verfahren nennt sich Daten-Augmentation [Chollet 2018, 184]. Ein Beispiel für ein solches Tool ist image augmentor[12]. In diesem Projekt werden ebenfalls einige der Bilder aus aufgenommen Videos extrahiert, wobei während des Videos immer wieder die Hintergründe, Belichtungen und Positionen verändert werden. Das Skript zum Extrahieren der Bilder ist im Anhang E zu finden.

Wenn eine ausreichende Zahl an Bildern zur Verfügung steht, müssen diese noch gelabelt werden. Die Zahl der benötigten Bilder kann von Fall zu Fall variieren. Während für ein einfaches Problem wie eine Bechererkennung mehrere hundert Bilder schon reichen können, werden bei komplexeren Modellen mit vielen Klassen in der Regel weit über 1000 Bilder pro

[11] https://datasetsearch.research.google.com/
[12] https://github.com/codebox/image_augmentor

Klasse benötigt [Bochkovskiy 2020]. Für das Labeln der Daten steht das Tool LabelImg zur Verfügung. Die Bilder werden dort eingelesen und können über eine grafische Oberfläche mit Boxen für die jeweiligen Objekte bestückt werden. Für jedes Bild wird anschließend eine .txt-Datei erstellt, welche die Nummer des jeweiligen Objekts und die dazugehörigen relativen Koordinaten enthält. Die Nummer wird über eine weitere Datei dem Namen des Objekts zugeordnet, auf die später noch weiter eingegangen wird. Ein Beispielbild mit bereits markierten Bechern ist in Abbildung 19 zu sehen.

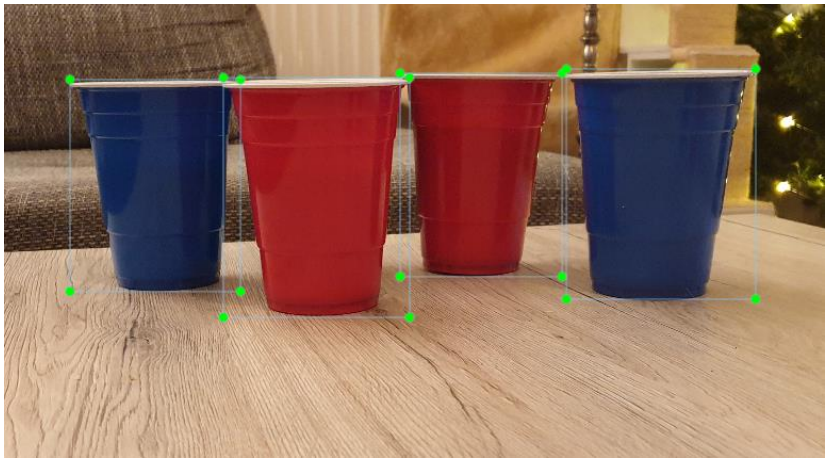


Abbildung 19: Beispielbild aus LabelImg

Nach dem Erstellen des Datensatzes muss dieser noch aufgeteilt werden. Dies hat den Hintergrund, dass für das Trainieren und Evaluieren der Ergebnisse nicht die gleichen Daten genutzt werden dürfen, um die Ergebnisse nicht zu verfälschen. Eine übliche Einteilung der Bilder ist 80% für das Training und 20% für das Evaluieren zu nutzen [Deru et al. 2020, 84]. Außerdem sollten die Bilder zur Evaluierung ebenfalls ein möglichst breites Abbild aller möglichen Belichtungen, Größen und Positionen der Objekte enthalten. Sind die Becher in den Testdaten immer in der gleichen Umgebung zu sehen, kann dies das Testergebnis ebenfalls beeinflussen, da sich keine Aussage über die Erkennung in anderen Umgebungen treffen lässt. Da die Bilder vor dem Training automatisch auf die Größe des neuronalen Netzes angepasst werden, müssen die Bilder vorher nicht eigenständig verändert werden. Unabhängig von dem Format der Bilder werden diese auf die Größe des eingestellten Eingangs verändert. Sollten die Formate unterschiedlich sein, wird das Bild in der Höhe bzw. Breite passend gewählt und der Rest mit schwarzen Pixeln aufgefüllt. Wenn für das neuronale Netz also eine Bildgröße von 640 x 640 Pixel eingestellt ist, würde ein Bild der Größe 1280 x 960 Pixel auf 640 x 480 Pixel verkleinert werden. Damit die Formate einheitlich sind füllt das Modell die restlichen Pixel schwarz auf.

5.1.2. Vorbereiten der für das Training benötigten Dateien

Um das Training der eigenen Objekterkennung trainieren zu können, müssen ein paar Einstellungen vorgenommen und Dateien erstellt werden. An den Einstellungen für das Training wird letztendlich nicht viel verändert. Weitestgehend werden die Standardeinstellungen des YOLOv4 genutzt, die schon sehr gut auf das Netzwerk abgestimmt sind. Diese sind in [Bochkovskiy 2020] zu entnehmen. Eine Standardversion der Einstellungen ist in dem Ordner ‚cfg' im Github-Repository von Darknet[13] zu finden. Diese kann dann an das Projekt angepasst werden. Standardmäßig steht *batch* auf 64 und *subdivisions* auf 16. Über die Einstellung *batch* kann bestimmt werden, wie groß der Datensatz ist, der pro Trainingsschritt eingelesen wird. An diesen Stellen kann geschraubt werden, wenn ein Speicherfehler auftreten sollte. Da es in dem Colab-Notebook gelegentlich vorkommt, dass der Grafikspeicher nicht ausreicht, wird *subdivisions* hier auf 32 erhöht. Die Einstellung für *batch* sollte möglichst beibehalten werden. Der Wert für *max_batches* ist abhängig von der Anzahl der zu erkennenden Klassen und soll mit $Anzahl\ der\ Klassen \cdot 2000$, allerdings nicht kleiner als 6000 gewählt werden. Bei einer Klasse wird dieser Wert dementsprechend auf 6000, bei fünf Klassen auf 10000 gesetzt. Hier werden die Trainingsschritte eingestellt, nach denen das Training automatisch abbricht. Da es in diesem Projekt nur eine Klasse gibt, wird dieser Wert also auf 6000 gesetzt. Davon abhängig sind die *line steps*, die mit 80% und 90% der *max_batches* gewählt werden sollten. Die Netzwerkgröße steht zu Beginn auf 416x416. Hier wird allerdings ein größerer Wert von 640 x 640 Pixel gewählt. Dies verlangsamt zwar das Netzwerk, da das Modell anstatt $416 * 416 * 3 = 519.168\ Eingänge$ insgesamt $640 * 640 * 3 = 1.228.800\ Eingänge$ besitzt, erhöht aber auch die Genauigkeit. Danach muss in den Yolo-Layern der Objekterkennung die Zeile *classes* auf den Wert der Klassen, also eins und die Zeile *filters* auf den Wert $(Anzahl\ der\ Klassen + 5) \cdot 3$ gesetzt werden. Dies sind alle Einstellungen im config-File, die für das Trainieren einer neuen Erkennung nötig sind [Bochkovskiy 2020]. Eine Zusammenfassung der vorgenommenen Einstellung ist in Tabelle 1 zu finden.

Tabelle 1: Vorgenommene Einstellungen inklusive Richtwerte

Einstellung	Eingestellter Wert	Richtwert
batch	64	64
subdivions	32	16, Erhöhen falls Speicher nicht ausreicht
max_batches	6000	$Anzahl\ der\ Klassen \cdot 2000$ bzw. 6000, falls Ergebnis kleiner als 6000
line steps	4800, 5400	80% und 90% der *max_batches*
classes	1	Anzahl der Klassen
filters	18	$(Anzahl\ der\ Klassen + 5) \cdot 3$

[13] https://github.com/AlexeyAB/darknet/tree/master/cfg

Zusätzlich zu dem config-File werden noch weitere Dateien benötigt, um das Training zu starten. Zum einen wird eine Datei *obj.names* benötigt, in der die Namen der Klassen stehen. Die Namen werden dabei getrennt voneinander in einer Zeile für jeden Namen aufgeschrieben. Dies sollte bei mehreren Namen wie in Abbildung 20 aussehen.

```
hund
katze
hamster
igel
```

Abbildung 20: Aufbau der obj.names

Außerdem wird mit der *obj.data* noch eine weitere Datei benötigt. Diese enthält Informationen über die Anzahl der Klassen, die Verzeichnisse mit Trainigs- und Testbildern, sowie den Pfad für die *obj.names* Datei und den Backup-Pfad. Besonders dieser ist wichtig, da in diesem in regelmäßigen Schritten die Gewichte gespeichert werden und somit im Falle eines Trainingsausfalls nicht verloren gehen. Zusätzlich können die Gewichte somit später besser verglichen werden und ein eventuelles Overfitting erkannt und durch ältere Gewichte ersetzt werden. Beim Overfitting erzielt das Modell eine sehr hohe Genauigkeit bei den Testdaten, kann aber nicht auf die unbekannten Daten generalisieren. In diesem Fall hat das Netz sich zu sehr an die Trainingsdaten angepasst. Der Gegensatz dazu ist das Underfitting, bei dem sich das Modell nahezu gar nicht an die Daten anpasst und somit gar nicht oder nur sehr langsam besser wird [Deru et al. 2020, 82]. Der Aufbau der Datei ist in Abbildung 21 zu betrachten. In diesem Falle gibt es mit den zu erkennenden Bechern nur eine Klasse.

```
classes = 1
train = data/train.txt
valid = data/test.txt
names = data/obj.names
backup = /mydrive/yolov4/backup
```

Abbildung 21: Aufbau der obj.data

Hierbei ist es wichtig die richtigen Pfade und die richtige Anzahl der Klassen zu wählen. Wird das Training nicht in Google Colab, sondern lokal durchgeführt, können die Pfade durch lokale Pfade ersetzt werden. Wenn die Dateien erstellt worden sind, können diese für das Training genutzt werden. Voraussetzung hierfür ist, dass die Installation wie in Anhang A bereits fertiggestellt wurde. Ist dies der Fall, kann das Colab-Notebook als erstes an die eigene Google-Drive angebunden werden. Dies ist die einfachste Möglichkeit die Daten mit der virtuellen Umgebung auszutauschen. Bevor die Daten aus Google Drive aber kopiert werden können, sollte dort erst eine übersichtliche Ordnerstruktur angelegt und die Dateien in dieser abgelegt werden. Es ist ratsam im Root-Verzeichnis von Google Drive einen Ordner mit dem Namen ‚yolov4' zu Erstellen. In diesem können alle benötigten Dateien und Skripte abgelegt werden und von dort nach Google Colab kopiert werden. Außerdem bietet es sich an, innerhalb des

Ordners einen weiteren Ordner namens ‚backup' zu erstellen, in dem die einzelnen Checkpoints des Modells abgelegt werden.

Jetzt kann der Code in Google Colab geschrieben werden. Vorher müssen allerdings noch ein paar Feinheiten geklärt werden. In den Zellen von Colab können sowohl Python-Code, als auch Bash-Kommandos ausgeführt werden. Ist eine Zeile ohne besonderes Vorzeichen handelt es sich um Python-Code und dieser wird ganz normal ausgeführt. Durch das Hinzufügen von ‚!' vor der Zeile können Bash-Kommandos ausgeführt werden. Eine Besonderheit ist das Ausführen von ‚Line Magic'. Diese wird durch ein ‚%' vor der Zeile gekennzeichnet und führt IPython-Kommandos aus. Oft rufen diese auch Bash-Kommandos auf, verhalten sich aber etwas anders. Eine Liste der verfügbaren IPython-Kommandos ist hier[14] zu finden. Das Verbinden mit Google-Drive erfolgt über den folgenden Code.

```
%cd ..

from google.colab import drive

drive.mount('/content/gdrive')
```

Nach dem Ausführen des Codes muss ein Bestätigungscode eingegeben werden. Um diesen zu erhalten kann auf den angezeigten Link geklickt und sich bei Google-Drive angemeldet werden. Danach sollte der Bestätigungscode angezeigt werden und kann in Google Colab eingefügt werden. Wenn dies geschehen ist, sind das Notebook und Drive verbunden. Zu dem verbundenen Ordner wird jetzt ein Link erstellt, der die weitere Nutzung deutlich vereinacht.

```
!ln -s /content/gdrive/My\ Drive/ /mydrive
```

Mit Hilfe des folgenen Befehls kann getestet werden, ob Google Drive richtig verbunden und der Link erstellt wurde.

```
!ls /mydrive
```

Es sollten alle Ordner und Dateien im Home-Verzeichnis von Google-Drive angezeigt werden. Danach müssen die weiter oben erstellten Dateien noch in das Notebook kopiert werden, um sie dort verwenden zu können. Als erstes müssen die Trainings- und Testbilder mit den dazugehörigen Label-Dateien in separaten Archiven gespeichert und in Google Drive abgelegt werden. Das Speichern als Archiv bringt eine Zeitersparnis, da das Kopieren der Bilder von Google Drive zu Google Colab je nach Anzahl der Bilder sonst sehr viel Zeit in Anspruch nehmen kann. Die Archive können mit den Befehlen

```
!cp /mydrive/<Pfad>/obj.zip ../

!cp /mydrive/<Pfad>/test.zip ../
```

kopiert und mit

[14] https://www.tutorialspoint.com/google_colab/google_colab_magics.htm

```
!unzip ../obj.zip -d data/

!unzip ../test.zip -d data/
```

entpackt werden. Anschließend können die veränderten Einstellungen kopiert werden. Dies kann ebenfalls mit dem Befehl

```
!cp /mydrive/<Pfad>/yolov4-obj.cfg ./cfg
```

ausgeführt werden. Nachdem auch die Einstellungen kopiert worden sind, fehlen nur noch die .data- und .names-Dateien. Diese wurden ebenfalls schon vorbereitet und können mit

```
!cp /mydrive/yolov4/obj.names ./data

!cp /mydrive/yolov4/obj.data  ./data
```

kopiert werden. Des Weiteren benötigt Darknet für das Training noch die Dateien *train.txt* und *test.txt*, in denen die Pfade der einzelnen Trainigs- und Testilder angegeben sind. Diese können mühsam per Hand erstellt werden. Da dies aber sehr lange dauern würde, ist auch hier sinnvoll, dies über ein Skript zu automatisieren. Diese Skripte sind im Github-Repository von ‚The AI Guy'[15] und im Anhang E zu finden. Die Skripte werden als erstes wieder aus dem Drive-Ordner nach Google Colab kopiert.

```
!cp /mydrive/yolov4/generate_train.py ./

!cp /mydrive/yolov4/generate_test.py ./
```

Anschließend können sie durch das folgende Batch-Kommando ausgeführt werden.

```
!python generate_train.py

!python generate_test.py
```

5.1.3. Trainieren und Testen der Erkennung

Nachdem alle Files wie im vorherigen Kapitel vorbereitet wurden, kann mit dem Training für die Objekterkennung begonnen werden. Da das Netzwerk nicht komplett von neu auf trainiert werden soll, müssen noch die Gewichte des ausgewählten Modells heruntergeladen werden. Das neue Modell wird dann auf der Basis dieser Gewichte trainiert. Dieser Prozess wird ‚Transfer Learning' genannt. Das Übernehmen eines bereits trainierten Modells bietet einige Vorteile. Zum einen wird das Training deutlich beschleunigt, da das Netzwerk grundsätzlich bereits in der Lage ist verschiedene Merkmale zu extrahieren. Die Werte müssen zwar für die neuen Objekte angepasst werden, dies benötigt aber deutlich weniger Zeit als das Neutraining eines Netzwerks [Torrey et al. 2010]. Der Vorteil in der Lerngeschwindigkeit beim Transfer Learning ist in Abbildung 22 sehen.

[15] https://github.com/theAIGuysCode/YOLOv4-Cloud-Tutorial/tree/master/yolov4

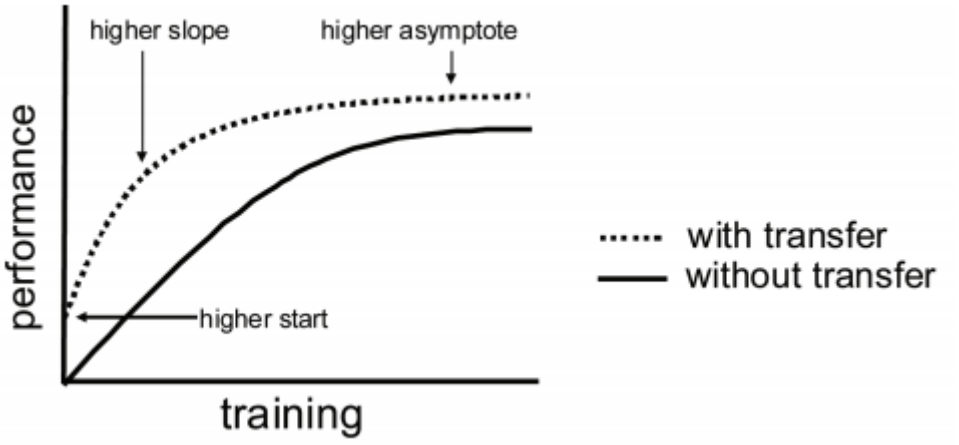


Abbildung 22: Performance des Modells mit und ohne Transfer Learning [Torrey et al. 2010, 243]

Viel wichtiger aber noch ist, dass die vortrainierten Netzwerke bereits eine fertige Struktur mit bekannter Genauigkeit und Geschwindigkeit besitzen. Das Erstellen eines CNNs für die Objekterkennung ist ein äußerst komplizierter Prozess, bei dem sehr viele verschiedene Schichten auf unterschiedliche Weisen vernetzt werden, um optimale Ergebnisse zu erzielen. Diese Modelle wurden von Experten trainiert, die damit einen großen Teil ihres Lebens verbracht haben. Da diese Modelle kostenlos zur Verfügung stehen, besteht also kein Grund den gleichen Prozess nochmal zu durchlaufen. Um die Gewichte für das Training herunterzuladen kann folgendes Kommando verwendet werden.

```
!wget https://github.com/AlexeyAB/darknet/releases/download/d
arknet_yolo_v3_optimal/yolov4.conv.137
```

Das Starten des Trainings erfolgt über das Kommando:

```
!./darknet detector train <Pfad zu obj.data> <Pfad
zu config> yolov4.conv.137 -dont_show -map
```

Im Anschluss wird das neuronale Netz Schritt für Schritt trainiert und die Gewichte angepasst. Durch die weiter oben vorgenommenen Einstellungen wird das Netzwerk 6000 Schritte lang trainiert, bevor es die finalen Gewichte speichert. Die Flags *-dont_show* und *-map* sorgen dafür, dass die Bilder nach den Trainingsschritten nicht angezeigt werden und die Genauigkeit der Erkennung zusammen mit dem Loss aufgenommen wird. Vor allem das *-dont_show* ist wichtig, da das Anzeigen in Google Colab so nicht möglich ist. Alle 1000 Schritte werden backups in den Ordner gespeichert, der in der Datei *obj.data* angegeben wurde. Außerdem wird ab 1000 Schritten für die gespeicherten Backups auch die mAP berechnet, um ein Indikator für die Genauigkeit zu sein. Die mAP wird mit den Bildern aus dem Testdatensatz berechnet und gibt somit ein Abbild der Genauigkeit für unbekannte Bilder (siehe Kapitel 3.2.5). Je vielseitiger die Bilder hier gewählt wurden, desto aussagekräftiger ist das Ergebnis. Zusätzlich zu den Backups wird auch der aktuelle Stand der Gewichte in dem Ordner gespeichert. Dieser wird alle 100 Schritte überschrieben. Das Training des neuronalen Netzwerks erfolgt dabei wie in Kapitel 3.2.2 beschrieben nach dem Backpropagation-Algorithmus. Für das ausgewählte Modell mit den

in Kapitel 5.1.2 beschriebenen Einstellungen und knapp 400 Trainingsbildern dauert das vollständige Training ca. 20 Stunden.

Da in Google Colab nur eine begrenzte Rechenzeit zur Verfügung steht, wird das Training mit sehr hoher Wahrscheinlichkeit nicht vollständig durchlaufen und muss neu gestartet werden. Dadurch, dass während des Trainings aber Backups, sowie die aktuellen Zwischenstände gespeichert werden, kann einfach von diesem Punkt an weitergemacht werden. Um von bereits bestehenden Gewichten fortzufahren, kann der Befehl

```
!./darknet detector train data/obj.data cfg/yolov4-
obj.cfg /mydrive/yolov4/backup/yolov4-obj_last.weights -
dont_show -map
```

verwendet werden. Auch hier werden die gleichen Flags gesetzt. Nachdem die Objekterkennung fertig trainiert ist, wird das Training automatisch beendet und die finalen Gewichte gespeichert. Hinterher können die verschiedenen Checkpoints miteinander verglichen werden, um herauszufinden, welcher die beste mAP besitzt. Zum Prüfen der mAP der Gewichte kann das Kommando

```
!./darknet detector map data/obj.data cfg/yolov4-
obj.cfg /mydrive/yolov4/backup/yolov4-obj_<Nummer>.weights
```

genutzt werden. Dies kann für mehrere Gewichte wiederholt werden, um diese vergleichbar zu machen. Außerdem wird der Loss das Ganze Training über in einer Grafik gespeichert. Wenn beim Training zusätzlich auch die Flag `-map` angegeben wurde, wird zusätzlich auch die mAP in der gleichen Grafik gezeigt. Diese kann mit `imShow('chart.png')` angezeigt werden. Der Loss für das Training ist in Abbildung 30 in Kapitel 6.1 zu sehen.

Sollte der Loss in dem Graph immer weiter fallen, die mAP aber ab einem gewissen Punkt ebenfalls wieder fallen, ist dies ein Indiz für Overfitting des Netzwerks. In diesem Fall sollte die Anzahl und Art der unterschiedlichen Bilder erhöht werden, damit das Modell besser generalisieren kann. Außerdem kann es hilfreich sein die Tiefe des Netzwerks etwas zu verringern und das Modell somit nicht unnötig kompliziert zu gestalten [Deru et al. 2020, 458].

Wenn die mAP nach dem Training ein zufriedenstellendes Ergebnis erzielt, kann das Modell auch noch auf zusätzlichen Bildern oder Videos getestet werden. Grundsätzlich ist es schwer hier einen Richtwert für eine gute mAP zu nennen, da diese auch stark von der Anzahl und Komplexität der gewählten Klassen abhängt. Bei einem einfachen Modell wie diesem mit nur einer Klasse sollte die mAP allerdings deutlich über 90% liegen. Für das Testen des Modells an einem eigenen Bild kann das folgende Kommando verwendet werden.

```
!./darknet detector test data/obj.data cfg/yolov4-
obj.cfg /mydrive/yolov4/backup/yolov4-obj_last.weights <Pfad
zu Bild> -thresh 0.5

imShow('predictions.jpg')
```

Das Bild mit den erkannten Objekten wird anschließend in der Ausgabe angezeigt. Soll das Modell auf einem Video getestet werden, kann das folgende Kommando genutzt werden.

```
!./darknet detector demo data/obj.data cfg/yolov4-
obj.cfg /mydrive/yolov4/backup/yolov4-obj_last.weights -
dont_show <Pfad zu Video> -i 0 -out_filename <Speicherpfad
für neues Video> -thresh 0.5
```

Hier wird das Video allerdings nicht in Google Colab angezeigt, sondern in dem angegeben Pfad gespeichert. Dort kann das Video allerdings auch ganz normal angezeigt werden und die erkannten Objekte geprüft werden. Über den Parameter `-thresh 0.5` kann eingestellt werden, ab welcher Sicherheit das Modell ein Objekt als erkannt ansehen soll. Auch hier kann noch weiter experimentiert werden, um den idealen Wert für die Objekterkennung zu finden.

Ein Vorteil den das Testen auf einem Video hat, ist die Angabe der Geschwindigkeit. Das Modell bearbeitet Bild für Bild des Videos und zeigt dazu die Dauer der Erkennung und die aktuelle FPS-Zahl an.

5.2. Kameraeinstellungen für eine optimale Genauigkeit

Bevor die trainierte Objekterkennung genutzt werden kann, muss erstmal die Kamera richtig konfiguriert werden, damit diese möglichst genaue Werte liefern kann. Als Kamera wird, wie in Kapitel 4 beschrieben, die Intel Realsense D415 verwendet. Für das Verwenden der Kamera in Python stellt Intel eine Python-API[16], sowie einen eigenen Viewer zur Verfügung. Dieser ist zusammen mit weiteren Tools von Intel zur Verfügung gestellt. Die Installation dieser Tools wird in Anhang A beschrieben. In dem Viewer können die Streams angezeigt und in einer grafischen Oberfläche konfiguriert werden. Dies hat den Vorteil, dass komfortabel die Einstellungen verändert werden können, um so die optimalen Einstellungen zu finden. Die Kamera ist zum Start bereits kalibriert, sodass sie direkt verwendet werden kann. In manchen Fällen muss die Kamera allerdings neu kalibriert werden. Um die Qualität der Tiefenwerte messen zu können und festzustellen, ob eine Kalibrierung notwendig ist, bringt Intel ein Quality Tool zur Messung der Tiefenbildqualität mit. Für die Kalibrierung der Kamera hat Intel ein eigenes graphisches Tool, dass diese Kalibrierung vornimmt. Dabei dient ein Smartphone als Kalibrierobjekt, dass ca. 60-80 cm entfernt von der Kamera bewegt werden muss. Der detaillierte Ablauf der Kalibrierung kann der Datei ‚Custom_Calibration_Whitepaper.pdf' in den Anlagen entnommen werden.

Für das Finetuning der Kameraparemter bieten Intel ebenfalls ein Dokument an, das in den Anlagen unter ‚Realsense_Finetuning.pdf' zu finden ist. Wenn der Realsense Viewer gestartet und die Kamera angeschlossen ist, können die Streams gestartet werden. Sowohl für das Farb-,

[16] https://dev.intelrealsense.com/docs/python2

als auch für das Tiefenbild stehen mehrere Auflösungen zur Verfügung. Für das Tiefenbild kann hier die maximale Auflösung von 1280 x 720 Pixel gewählt werden, um die maximale Genauigkeit zu erreichen. Da die Objekterkennung auf dem Farbbild durchgeführt wird, während die Tiefenwerte aus dem Stereobild berechnet werden, kann eine Umrechnung der Pixel vermieden werden, wenn beide Streams die gleiche Auflösung verwenden. So können die Pixel später 1:1 übernommen werden, wenn die Streams ausgerichtet sind. Nach der Aktivierung der beiden Kamerastreams mit den Standardeinstellungen, sind die Bilder aus Abbildung 23 zu erkennen.

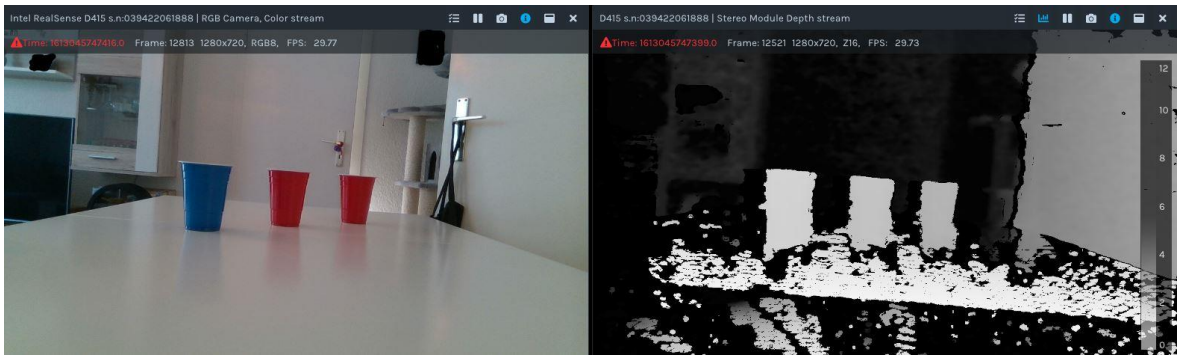


Abbildung 23: Tiefenbild bei Standardeinstellungen

Auf dem Farbbild ist alles gut zu erkennen. Das Tiefenbild hingegen ist vor allem im vorderen Bereich noch sehr verrauscht und kann die Becher teilweise nicht perfekt darstellen. Außerdem sind viele Abschnitte zu sehen, in denen kein Tiefenwert vorhanden ist. Diese sind durch die vielen schwarzen Flecken im Bild zu erkennen. Darüber hinaus wirkt die Tischplatte sehr störend und erschwert die Abgrenzung zwischen Tisch und Becher in manchen Fällen. Um die Genauigkeit zu Erhöhen und ein besseres Ergebnis für den gesamten Becher zu erhalten, steht ein Modus für erhöhte Genauigkeit bereit. In diesem Modus werden nur Tiefenwerte angezeigt, die eine gewisse Genauigkeit überschreiten. Außerdem werden die Tiefenwerte über einen kurzen Zeitraum betrachtet und so kurzfristige Ungenauigkeiten ausgeglichen. In Abbildung 24 ist das Tiefenbild mit dem angepassten Modus ersichtlich.

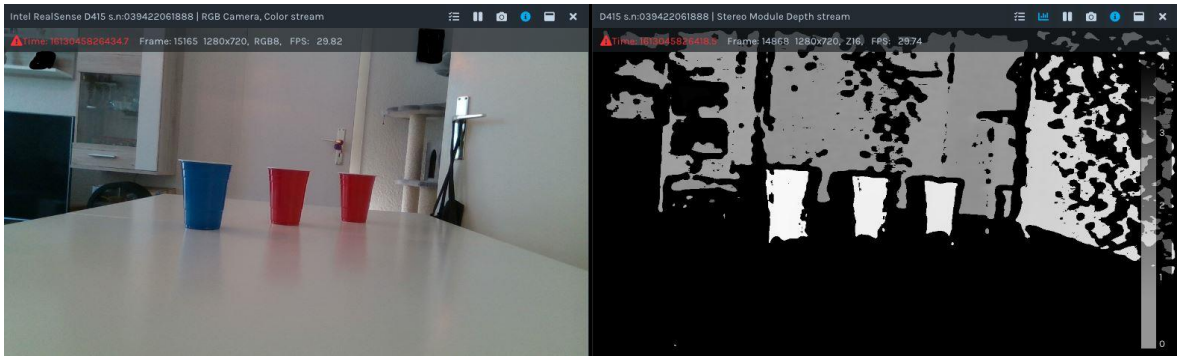


Abbildung 24: Tiefenbild mit erhöhter Genauigkeit

In Folge dessen, dass ungenaue Tiefenwerte herausgefiltert werden, gibt es ab und an Momente, in denen die Becher gar nicht zu sehen sind. Dies hat den Hintergrund, dass die Becher bei nicht optimaler Beleuchtung ab und an unter den Schwellwert fallen, ab dem die Tiefenwerte

angezeigt werden. Für diesen Zweck besitzt die Kamera jedoch einen Infrarot-Emitter, mit den ein Muster auf die Becher projiziert wird. Dieser dient zum einen der Verbesserung der Belichtung für die Kamera und zum anderen der leichteren Erkennung von gemeinsamen Merkmalen auf den verschiedenen Kamerabildern. Wird die Laserpower dieses Emitters von den vorher eingestellten 150 mW auf 210 mW gestellt, bleiben die Becher auch bei schlechter Belichtung erhalten und die wichtigen Bereiche werden konstant gut erkannt. Außerdem ist die kleinste Längeneinheit in diesem Modus auf 1 mm gestellt. Da die Tiefenwerte als 16 Bit-Zahlen dargestellt werden, wäre damit eine maximale Reichweite von ca. 65 m erreichbar. Dies übersteigt bei weitem den gewünschten Bereich und kann daher noch weiter heruntergesetzt werden, um die Genauigkeit noch etwas zu erhöhen und weit entfernte Werte aus den Bildern zu entfernen. Durch Setzen der kleinsten Längeneinheit auf 0,1 mm ergibt sich daraus ein Arbeitsbereich von ca. 6,5 m. Da die Becher nur in sehr kurzer Entfernung zu der Kamera stehen werden, ist dieser Arbeitsbereich wesentlich besser geeignet. Die Kamerastreams mit den angepassten Einstellungen sind in Abbildung 25 zu sehen.

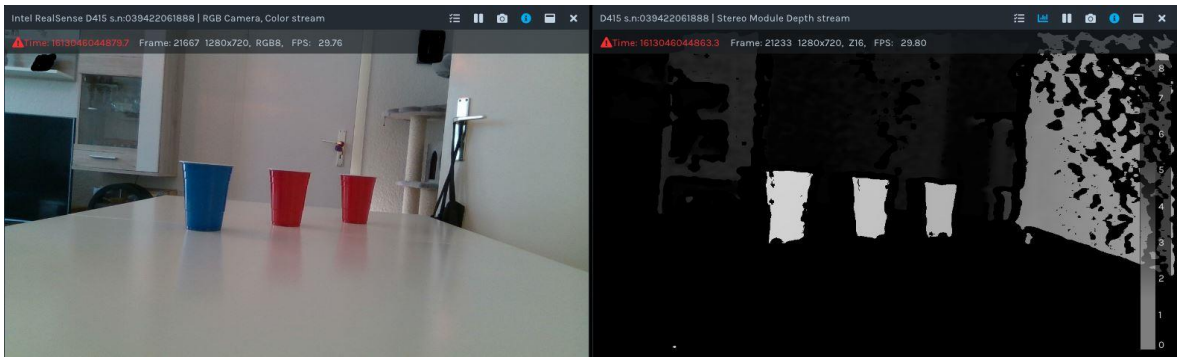


Abbildung 25: Tiefenbild mit manuellen Einstellungen

Im Vergleich zu dem Farbbild ist erkennbar, dass viele ungenaue Informationen aus dem RGB-Bild durch die erhöhte Genauigkeit verloren gehen. Dafür steigt die Genauigkeit für die angezeigten Werte aber deutlich. Die wichtigen Informationen, nämlich die Becher, bleiben erhalten und erreichen durch die zusätzlichen Einstellungen noch eine bessere Genauigkeit. Lediglich die Ränder der Becher werden etwas abgeschnitten. Da die Erkennung der Becher aber auf dem RGB-Bild durchgeführt wird und die Tiefenmessung nur in der Mitte der Becher erfolgt, ist dies für den weiteren Verlauf nicht weiter relevant.

5.3. Verarbeiten der Kamerabilder für die Tiefenmessung

Nachdem die optimalen Einstellungen für die Kamera gefunden wurden, kann mit dem Auslesen und Verarbeiten der Bilder innerhalb des Programms begonnen werden. Dafür kann die von Intel bereitgestellte Python-API verwendet werden. Das Einbinden der Bibliothek erfolgt mit der folgenden Codezeile.

```
import pyrealsense2 as rs
```

Die Installation dieser ist in Anhang A beschrieben. Um die Streams im Python-Code zu starten, kann der folgende Code genutzt werden.

```
pipe = rs.pipeline()

cfg = rs.config()

cfg.enable_stream(rs.stream.depth, 1280, 720, rs.format.z16,
30) # 1280 x 720 Pixel, 16 Bit-Tiefenwert, 30 FPS

cfg.enable_stream(rs.stream.color, 1280, 720, rs.format.bgr8,
30) # 1280 x 720 Pixel, 8 Bit BGR-Farbbild, 30 FPS

profile = pipe.start(cfg)
```

Hier werden nur die Streams gestartet und die passenden Auflösungen und Formate bestimmt. Als Auflösung wird, wie in Kapitel 5.2 beschrieben, für beide Streams 1280 x 720 Pixel gewählt. Als Speicherformat nutzt die Kamera standardmäßig einen 16 Bit Tiefenwert, sowie ein BGR-Farbbild mit jeweils 8 Bit. Als nächstes müssen die vorher erarbeiteten Einstellungen vorgenommen werden. Dafür muss als erstes der Tiefensensor mit

```
depth_sensor = profile.get_device().first_depth_sensor()
```

ausgelesen werden. Als nächstes können die verschiedenen Voreinstellungen von Intel geladen werden, da die angepassten Einstellungen auf diesen basieren.

```
preset_range = depth_sensor.get_option_range(rs.option.visual
_preset)
```

Durch die dort erhaltenen Presets kann in einer Schleife iteriert werden, bis das gewünschte Preset erreicht ist. Sobald das erwünschte Preset erreicht wird, wird es als Einstellung an die Tiefenkamera übermittelt.

```
for i in range(int(preset_range.max)):

    visulpreset = depth_sensor.get_option_value_description(r
    s.option.visual_preset,i)

    print('%02d: %s'%(i,visulpreset))

    if visulpreset == "High Accuracy":

        depth_sensor.set_option(rs.option.visual_preset, i)
```

Darauf aufbauend können die weiteren Einstellungen vorgenommen werden, die von dem ausgewählten Preset abweichen. In diesem Fall sind das zum einen die Laserpower des Emitters und die Einheit des kleinsten Tiefenwerts. Diese Einstellungen werden mit den folgenden Zeilen angewendet. In der ersten Zeile wird die Leistung des Emitters auf 210 mW gestellt und in der zweiten die kleinste Längeneinheit auf 0,1 mm.

```
depth_sensor.set_option(rs.option.laser_power, 210)
```

```
depth_sensor.set_option(rs.option.depth_units, 0.0001)
```

Das Warten auf neue Frames der Kamera erfolgt mit

```
pipe.wait_for_frames()
```

Dabei wartet das Programm, bis es neue Frames von der Kamera erhält. Diese Methode muss zum Start des Programms mehrere Male in einer Schleife durchlaufen werden, weil die Kamera eine gewisse Zeit braucht, um die Helligkeit etwas anzupassen. Wenn dies geschehen ist, kann die gleiche Methode wieder aufgerufen und die Bilder als Frameset gespeichert werden.

```
frameset = pipe.wait_for_frames()
```

Aus diesem Frameset können die beiden verschiedenen Bilder abgeholt und gespeichert werden.

```
color_frame = frameset.get_color_frame()

depth_frame = frameset.get_depth_frame()
```

Wenn von beiden Streams ein Bild vorhanden ist, müssen diese noch zueinander ausgerichtet werden. Wie in Abbildung 23 zu sehen, stimmen die beiden Bilder noch nicht ganz überein, da die Kameras etwas zueinander versetzt sind. Für das Ausrichten der beiden Frames zueinander stellt die API eine einfach zu verwendende Methode bereit. Sobald der Prozess fertig ist, wird ein neues Frameset als Rückgabewert zurückgegeben.

```
align = rs.align(rs.stream.color)

frameset = align.process(frameset)
```

Aus dem neuen Frameset können genau wie vorher auch wieder die beiden einzelnen Frames ausgelesen werden. Danach muss für das Tiefenbild noch ein `Colorizer` erstellt werden, der die Farbgebung für das Bild bestimmt. Dies ist nötig, da das Tiefenbild aktuell einfach nur aus 16 Bit Werten besteht, die jedem Pixel einen Tiefenwert zuweisen. Dies ist ohne eine Farbgebung noch nicht darstellbar. Dafür wird ein `Colorizer` erstellt, der auf den neu angepassten Tiefenframe angewendet werden kann. Außerdem muss das Bild anschließend noch zu einem Numpy-Array umgewandelt werden, damit es auch für OpenCV darstellbar ist.

```
colorizer = rs.colorizer()

aligned_depth_frame = frameset.get_depth_frame()

colorized_depth = np.asanyarray(colorizer.colorize(aligned_de
pth_frame).get_data())
```

Wenn das Farbbild auch zu einem Numpy-Array umgewandelt wurde, können die beiden Bilder auch mit dem folgenden Code angezeigt werden.

```
cv2.namedWindow('Color', cv2.WINDOW_AUTOSIZE)

cv2.namedWindow('Depth', cv2.WINDOW_AUTOSIZE)

cv2.imshow('Color', color_image)
```

```
cv2.imshow('Depth', colorized_depth)

cv2.waitKey()
```

Dabei wird für jedes der beiden Bilder erst ein Fenster erstellt und anschließend das Bild darin dargestellt. Zum Schluss wird noch die `waitKey()` von OpenCV aufgerufen. Die Methode sorgt dafür, dass das Programm erst weiterläuft, wenn eine Taste gedrückt wird.

5.4. Durchführen der trainierten Objekterkennung

Nachdem die Objekterkennung trainiert ist und die Bilder aus der Kamera ausgelesen wurden, kann die Objekterkennung angewendet werden. Hierfür bietet OpenCV ebenfalls Methoden, die das Einlesen und Nutzen der trainierten Modelle sehr einfach macht. Als erstes muss das Modell eingelesen und gespeichert werden. Für das Einlesen benötigt OpenCV nur die Pfade zu den Gewichten und der Config-Datei.

```
net = cv2.dnn.readNetFromDarknet(config_path, weights_path)
```

Außerdem wird noch die label-Datei benötigt, um die Erkennungen später auch den richtigen Objekten zuzuordnen. Diese werden ebenfalls für die spätere Verwendung gespeichert. Zusätzlich wird jedem Label eine zufällige Farbe zugewiesen, damit die verschiedenen Objekte auf dem Bild später leicht zu unterscheiden sind. Da in diesem Projekt nur ein Objekt erkannt wird, wird hier dementsprechend auch nur eine Farbe zugewiesen.

```
labels = open("yolov4/obj.names").read().strip().split("\n")

colors = np.random.randint(0, 255, size=(len(labels), 3), dtype="uint8")
```

In der oberen Zeile wird die Datei *obj.names* eingelesen, mit `strip()` die überflüssigen Leerzeichen entfernt und die einzelnen Zeilen durch `split("\n")` separiert werden. Die dadurch ausgelesenen Klassennamen werden als Liste zurückgegeben und gespeichert. In der zweiten Zeile wird ein Numpy-Array mit zufälligen Integer-Werten erzeugt. Das Array hat genauso viele Zeilen wie es Klassen gibt und 3 Spalten, in denen jeweils ein 8 Bit Integer gespeichert wird. Die drei Werte pro Klasse entsprechen den drei Farbkanälen und werden später für das Zeichnen der Rechtecke um die erkannten Objekte genutzt.

Bevor die Objekterkennung allerdings ausgeführt werden kann, muss erst einmal die Größe der Bilder angepasst werden. Aufgrund dessen, dass die Kamera nur Bilder in 16:9 aufnimmt, das Modell aber mit Bildern im Format 1:1 und 4:3 trainiert wurde, können möglicherweise Probleme beim Erkennen der Objekte auftreten. Um dem entgegenzuwirken, können die äußeren Ränder der Bilder abgeschnitten werden. Da die Becher nur in der Mitte der Bilder zu sehen sind, befinden sich am äußeren Rand keine wichtigen Informationen mehr. Das Abschneiden der Bilder muss nur auf dem Farbbild ausgeführt werden, da auch nur dieses als

Eingang in das Modell gegeben wird. Dies kann in Python in nur einer Zeile durchgeführt werden. Dafür können in eckigen Klammern die Anfangs- und Enddaten des gewünschten Bildausschnitts in Pixeln angegeben werden. Als Rückgabe erfolgt das zugeschnittene Bild.

```
color_cropped = color[0:720, 280:1000] # [Höhe, Breite]
```

Das zugeschnittene Bild besitzt das richtige Format und kann ohne Stauchen bzw. Strecken auf die richtige Größe verkleinert werden.

```
crop_img = cv2.resize(color_cropped, (640, 640))
```

Der letzte Schritt muss nicht unbedingt durchgeführt werden, da dies auch in der nächsten Operation mit übernommen wird. Der Übersicht halber wird das Verkleinern hier aber in einem extra Schritt durchgeführt. Aus dem in der Größe angepassten Bild kann jetzt ein `Blob` (Binary Large Object) erstellt werden. Ein solcher `Blob` speichert die Pixel der Eingangsbilder als binäre Daten und dient als Eingang für das Modell. Wichtig dabei ist, dass `swapRB=True` gesetzt wird, da das Farbbild als BGR-Bild vorliegt (siehe Kapitel 5.3), das Modell aber mit RGB-Bildern trainiert wurde und die Farbkanäle somit vertauscht sind. Ohne diese Einstellung würde das Netzwerk höchstwahrscheinlich keine sinnvollen Ergebnisse liefern. Als weitere Parameter bekommt die Methode das umzuwandelnde Bild, einen Skalierungsfaktor und die Bildgröße übergeben. Der Skalierungsfaktor skaliert die Werte auf eine gewünschte Größe. Ist der Faktor eins, bleiben die Werte erhalten. In diesem Fall beträgt der Skalierungsfaktor `1/255.0`. Dadurch werden die Eingänge auf Werte zwischen null und eins transformiert.

```
blob = cv2.dnn.blobFromImage(crop_img, 1/255.0, (640, 640), s
wapRB=True, crop=False)
```

Der erstellte `Blob` wird als Eingang für das Modell festgelegt.

```
net.setInput(blob)
```

Anschließend können die Schichten ausgelesen und die Objekterkennung gestartet werden. Die Ergebnisse werden gespeichert, da diese noch für das Kennzeichnen der erkannten Objekte benötigt werden.

```
ln = net.getLayerNames()

ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]

layer_outputs = net.forward(ln)
```

Die Becher werden jetzt bereits erkannt und die eigentliche Objekterkennung ist fertiggestellt. Jetzt müssen noch die verschiedenen Werte der Output-Layer ausgelesen und die Objekte gekennzeichnet werden. Das Auslesen der Werte erfolgt nach und nach in einer Schleife. Dabei werden als erstes die einzelnen Ausgabeschichten und darin die einzelnen erkannten Objekte durchlaufen. Für jede Erkennung werden als erstes die Confidence-Werte des Modells ausgelesen. Anschließend wird der größte Wert ermittelt und die dazugehörige Klassen-Id

gespeichert. Mit Hilfe der Klassen-Id mit dem höchsten Score wird anschließend die Confidence ausgelesen.

```
boxes, confidences, class_ids = [], [], []
for output in layer_outputs:
    for detection in output:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
```

Die ermittelte Confidence wird als Threshold für die weitere Verarbeitung genutzt. Dies funktioniert genau wie in Kapitel 5.1.3, als das Modell die Objekte nur ab einer gewissen Confidence in die Bilder bzw. Videos eingezeichnet hat. Sollte dieser Wert überschritten werden, werden als erstes die Bildkoordinaten der erkannten Objekte berechnet. Die Position, sowie Breite und Höhe sind als relative Werte gespeichert. Das Umwandeln von relativen in absolute Werte erfolgt durch die folgende Zeile.

```
detection[:4] * np.array([w, h, w, h]).
```

Anschließend werden die vier Werte als Integer ausgelesen und gespeichert.

```
        # CONFIDENCE = 0.5
        if confidence > CONFIDENCE:
            box = detection[:4] * np.array([w, h, w, h])
            (centerX, centerY, width, height) = box.astyp
            e("int")
```

Danach wird die obere linke Ecke des zu zeichnenden Rechtecks berechnet. Bei dem YOLOv4-Modell werden standardmäßig die Mittelpunkte der erkannten Objekte zurückgegeben. Da für das Zeichnen die obere linke Ecke zusammen mit der Höhe und Breite benötigt wird, werden diese noch berechnet und anschließend an die Listen mit erkannten Objekten und deren Position, Confidence und Namen angehängt.

```
            x = int(centerX - (width / 2))
            y = int(centerY - (height / 2))
            boxes.append([x, y, int(width), int(height)])
            confidences.append(float(confidence))
            class_ids.append(class_id)
```

Das Problem der aktuellen Erkennungen ist, dass aktuell noch mehrere Rechtecke für ein und dasselbe Objekt erkannt werden können. Um die überflüssigen Erkennungen zu entfernen,

kann eine sogenannte NMS (non-maximum-suppression) durchgeführt werden. Bei der NMS wird sozusagen ein lokales Maximum gesucht [Hosang et al. 2017]. Dabei werden die überlappenden Erkennungen nach und nach entfernt, sodass am Ende nur das erkannte Objekt mit der höchsten Wahrscheinlichkeit übrigbleibt. Auf diese Weise werden mehrere Rechtecke für das gleiche Objekt vermieden.

```
idxs = cv2.dnn.NMSBoxes(boxes, confidences, SCORE_THRESHOLD,
IOU_THRESHOLD)
```

Nach diesem Schritt können die Rechtecke für die erkannten Objekte in das Bild eingezeichnet werden. Das Zeichnen erfolgt in einer Schleife, in der alle übriggebliebenen Erkennungen durchlaufen werden. In der Schleife werden im ersten Schritt die Koordinaten, sowie die Höhe und Breite der erkannten Objekte ausgelesen. Anschließend wird die vorher zufällig generierte Farbe ausgelesen und das Rechteck mit der Methode `rectangle()` von OpenCV in das Bild eingezeichnet. Die Methode bekommt das Bild, die obere linke Position, die untere rechte Position, die Farbe und die Breite der Linie übergeben.

```
for i in idxs.flatten():

    x, y = boxes[i][0], boxes[i][1]

    w, h = boxes[i][2], boxes[i][3]

    color = [int(c) for c in colors[class_ids[i]]]

    cv2.rectangle(crop_img, (x, y), (x + w, y + h), color=c
    olor, thickness=thickness)
```

Danach wird das Textfeld formatiert und die Höhe und Breite des Textes ermittelt. Der Text wird aus dem Klassennamen und der dazugehörigen Confidence zusammengesetzt.

```
    text = f"{labels[class_ids[i]]}: {confidences[i]:.2f}"

    (text_width, text_height) = cv2.getTextSize(text, cv2.F
    ONT_HERSHEY_SIMPLEX, fontScale=font_scale, thickness=th
    ickness)[0]
```

Anschließend wird eine Position über dem erkannten Objekt berechnet.

```
    text_offset_x = x

    text_offset_y = y - 5
```

Mit der ermittelten Höhe und Breite des Textes wird ein transparentes Rechteck über den Text gelegt, was die Umrandung des Textest verursacht.

```
    box_coords = ((text_offset_x, text_offset_y), (text_off
    set_x + text_width + 2, text_offset_y - text_height))

    overlay = crop_img.copy()
```

```
cv2.rectangle(overlay, box_coords[0], box_coords[1], co
lor=color, thickness=cv2.FILLED)

crop_img = cv2.addWeighted(overlay, 0.6, crop_img, 0.4,
 0)
```

Zum Schluss wird nur noch der Text über das Bild gelegt und die erkannten Objekte können nun angezeigt werden. Das Anzeigen der Objekte funktioniert genau wie in Kapitel 5.3 und kann über OpenCV umgesetzt werden.

```
cv2.putText(crop_img, text, (x, y - 5), cv2.FONT_HERSHE
Y_SIMPLEX,fontScale=font_scale, color=(0, 0, 0), thickn
ess=thickness)
```

5.5. Auswahl des im Vordergrund stehenden Objekts

Das fertig trainierte Netzwerk kann als nächstes genutzt werden um die Becher zu erkennen. Für die spätere Weitergabe der Koordinaten ist es aber wichtig, dass nicht für jeden erkannten Becher versucht wird die Position zu berechnen. Aufgrund von Verdeckungen ist es für die Tiefenkamera nicht möglich die Position für jeden Becher zu bestimmen. Eine einfache Lösung für das Problem ist, jeweils nur den vorderen Becher zu ermitteln bzw. Becher herauszufiltern, die zum größten Teil verdeckt sind.

Für die Ermittlung der Schnittfläche der Becher können die Bounding Boxes der Objekterkennung genutzt werden. Jeder erkannte Becher wird durch ein Rechteck markiert. Aus diesen Flächen können die Überschneidungen der Becher ermittelt werden. Wird erkannt, dass mehrere Becher sich gegenseitig überschneiden, sodass nicht für jeden Becher die Position ermittelt werden kann, folgt daraus noch ein weiteres Problem. Und zwar muss aus den Flächen der erkannten Becher noch ermittelt werden, welcher Becher vorne steht.

Dazu stehen im Prinzip zwei Methoden zur Verfügung. Zum einen kann danach gefiltert werden, welcher Becher am weitesten unten in der Bildfläche zu sehen ist. Die Becher stehen alle auf der gleichen Ebene, sind aber in der Tiefe versetzt. Da die Kamera ein wenig über dieser Ebene steht, sind tiefer stehende Becher automatisch etwas höher in der Bildebene zu sehen. Ein weiterer Filter ist die Größe der Becher. Je näher der Becher an der Kamera steht, desto größer erscheint er auf dem Bild. Überschneiden sich also zwei Becher, kann davon ausgegangen werden, dass der Becher mit der größeren Fläche weiter vorne steht und somit für die Ermittlung der Position geeignet ist. Hierbei können allerdings Probleme auftauchen, wenn das Modell die Becher größer erkennt als sie wirklich sind. Da die Erkennungen für mehr oder weniger freistehende Becher auf dem Tisch aber sehr genau sind, tritt dieser Fehler im Regelfall nicht auf.

5.6. Bestimmen der optimalen Messposition und Berechnung der Becherbreite

Hinterher kann die Position des ausgewählten Bechers ermittelt werden. Das Problem dabei ist, dass die Becher nicht an jeder Stelle gleich breit sind, sondern nach oben hin breiter werden. Zusätzlich steigt die Breite nicht vollständig linear, sondern hat zwischendurch Sprünge, an denen die Breite sich deutlich schneller erhöht. Dadurch ist es schwer die Bechermitte exakt zu bestimmen. Diese wird allerdings als Ziel für einen späteren Wurf in den Becher benötigt. Eine Skizze des Bechers ist in Abbildung 26 zu sehen.

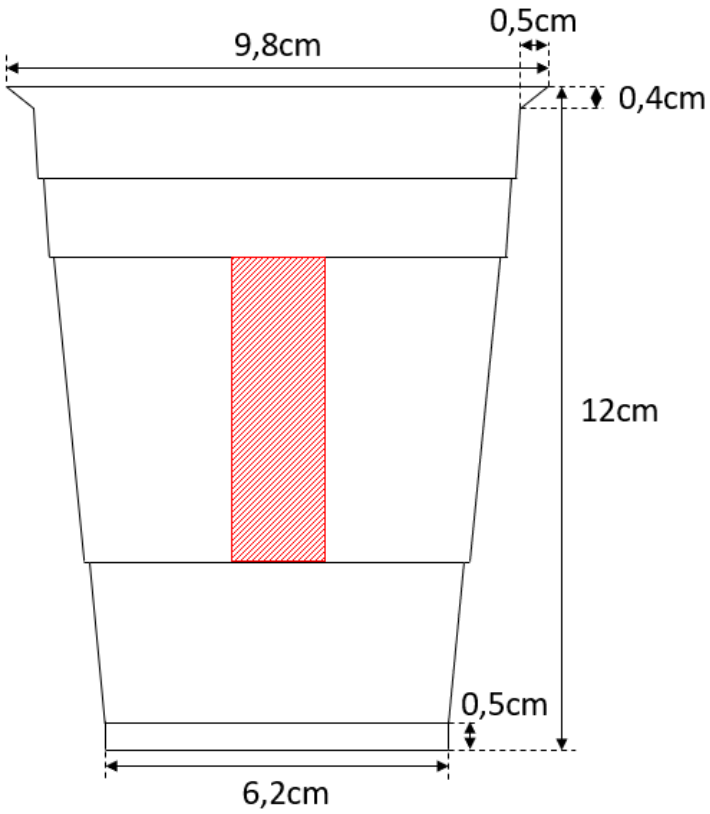


Abbildung 26: Skizze des verwendeten Bechers

Durch abziehen des oberen und unteren Becherrands, kann der Mittelteil des Bechers näherungsweise durch eine lineare Gleichung nachgebildet werden. Durch diesen Zusammenhang ist die Bechermitte die optimale Zone für die Positionsmessung. Diese ist als rotes Rechteck in der Skizze des Bechers eingezeichnet. Die Breite des Bechers kann durch eine lineare Funktion in Abhängigkeit der Höhe berechnet werden.

$$Breite = b_{min} + h \cdot \left(\frac{\Delta b}{\Delta h}\right)$$
$$= 6{,}2cm + h \cdot \left(\frac{9{,}8cm - 0{,}5cm}{12cm - 0{,}5cm - 0{,}4cm}\right) = 6{,}2cm + 0{,}748cm \cdot h \quad (44)$$

Mit Hilfe der berechneten Becherbreite an der Messstelle kann später der Mittelpunkt des Bechers berechnet werden.

5.7. Auslesen der Distanz und Kamerakoordinaten

Wenn der im Vordergrund stehende Becher ausgewählt wurde, kann dieser für die Messung der Tiefe verwendet werden. Dazu muss als erstes die genaue Position bestimmt werden, an der die Entfernung gemessen werden soll. Wie in Kapitel 5.6 beschrieben, wird hierfür die Mitte des Bechers gewählt, da die Messung hier das beste Ergebnis erzielt. Da die Position, sowie Breite und Höhe des Bechers durch die Objekterkennung bekannt sind, kann der Mittelpunkt des Bechers auf der Bildebene berechnet werden.

```
temp_x = x + w/2 # w = width

temp_y = y + h/2 # h = height
```

Bei den ermittelten Koordinaten handelt es sich allerdings noch um die Koordinaten auf dem herunterskalierten Bild. Das Bild hat eine Größe von 640 x 640 Pixel, während das Tiefenbild noch eine Größe von 1280 x 720 Pixel besitzt. Um die Tiefe also an der richtigen Position auszulesen, muss die Position auf die ursprüngliche Größe rücktransformiert werden. Die Pixelposition im Tiefenbild kann mit den folgenden Zeilen berechnet werden.

```
x_depth = int( int(temp_x) * 1.125 + 280 )

y_depth = int( int(temp_y) * 1.125 )
```

Der Faktor 1.125 stammt dabei aus dem Skalierungsfaktor von 640 x 640 Pixel auf 720 x 720 Pixel. Außerdem müssen in der Bildzeile noch 280 Pixel addiert werden, da diese an den Rändern von dem ursprünglichen Bild abgeschnitten wurden. Mit der jetzt bekannten Pixelposition auf dem Tiefenbild kann mit Hilfe der bereits genutzten API von Intel, die Distanz zu dem gewählten Pixel berechnet werden. Dazu kann folgende Methode verwendet werden.

```
dist = aligned_depth_frame.get_distance(x_depth, y_depth)
```

Dabei kann durch die Nutzung von zwei Kameras eine Triangulation wie in Kapitel 3.3.4 durchgeführt werden, um die Tiefe zu erhalten. Nun sind die Bildkoordinaten und die Tiefe des Objekts bekannt. Um aus diesen Werten die Koordinaten im Koordinatensystem der Kamera zu ermitteln, werden als erstes die intrinsischen Parameter der Kamera benötigt. Diese können aus dem vorhandenen Frame ausgelesen werden.

```
camera_info = aligned_depth_frame.profile.as_video_stream_pro
file().intrinsics
```

Mit Hilfe der Koordinaten, Tiefe und intrinsischen Parameter der Kamera kann die Funktion `convert_depth_to_phys_coord_using_realsense` aufgerufen werden. Als

Rückgabewerte besitzt die Funktion nicht nur einen Wert, sondern alle drei Koordinaten. Dies ist eine Besonderheit von Python und ist so in den meisten Programmiersprachen nicht möglich.

```
x_w, y_w, z_w = convert_depth_to_phys_coord_using_realsense(x
_depth, y_depth, dist, camera_info)
```

In der Funktion werden als erstes die intrinsischen Parameter ausgelesen und anschließend die Funktion `rs2_deproject_pixel_to_point` aufgerufen, welche die Koordinaten berechnet.

```
def convert_depth_to_phys_coord_using_realsense(x, y, depth,
cameraInfo):

    _intrinsics = rs.intrinsics()

    _intrinsics.width = cameraInfo.width

    _intrinsics.height = cameraInfo.height # Bildhöhe

    _intrinsics.ppx = cameraInfo.ppx # Bildhauptpunkt x

    _intrinsics.ppy = cameraInfo.ppy # Bildhauptpunkt y

    _intrinsics.fx = cameraInfo.fx # Brennweite x

    _intrinsics.fy = cameraInfo.fy # Brennweite y

    _intrinsics.model  = cameraInfo.model # Distortion-Model

    _intrinsics.coeffs = cameraInfo.coeffs #Parameter des
    Distortion-Models

    result = rs.rs2_deproject_pixel_to_point(_intrinsics, [x
    , y], depth) # Umrechnen in Koordinaten

    # result[0]:rechts, result[1]:unten, result[2]:vorwärts

    return result[0], -result[1], -result[2]
```

Die Methode nutzt dabei die bekannten Kameraparameter um das Bild nach dem genutzten Kameramodell in reale metrische Werte umzurechnen. Die Berechnung erfolgt dabei grob nach dem Modell der Lochkamera aus Kapitel 3.3.2. Der Unterschied ist, dass die Kamera zusätzlich noch eine Distortion besitzt, welche miteinbezogen wird. Diese ist eine leichte Verzerrung des Bildes, die durch den Einsatz einer Linse entsteht.

5.8. Kalibrierung der Kamera und Bestimmen der Transformationsmatrix für die Umrechnung der Koordinaten

Bei der Kalibrierung der Kamera gibt es mehrere Dinge zu beachten. Die Kameras müssen zueinander kalibriert werden, um die extrinsischen und intrinsischen Parameter herauszufinden. Nur so können die Positionen der Objekte genau ermittelt werden. Die Kamera bietet dabei zwei Optionen für die Kalibrierung.

Zum einen gibt es eine Kalibrierung mit einem Muster auf dem Smartphone, das von den Kameras erkannt wird. Diese Kalibrierung funktioniert ähnlich dem Prinzip aus Kapitel 3.3.3.3 und kann sowohl die extrinsischen, als auch die intrinsischen Parameter bestimmen. Diese Kalibrierung wurde von Haus aus schon erledigt, kann aber bei Bedarf vom Nutzer nochmal erneuert werden. Eine Erneuerung ist zum Beispiel sinnvoll, wenn die Kamera sich durch Temperaturunterschiede verzogen haben könnte oder möglicherweise heruntergefallen ist.

Außerdem gibt es noch eine Selbstkalibrierung, welche die Kameras auch ohne Schachbrettmuster durchführt. Dafür muss die Kamera einfach vor eine große Fläche wie eine Wand oder eine Tür gestellt werden. Auf dieser erkennt die Kamera dann verschiedene Merkmale und kann somit entweder die intrinsischen oder die extrinsischen Parameter erneuern. Eine Kalibrierung aller Parameter ist damit nicht möglich.

Diese Kalibrierungen gehören zu den Funktionen der Kamera und dienen dazu die neuen Parameter herauszufinden und auf der Kamera zu speichern. Darüber hinaus fehlt aber noch die Transformationsmatrix, mit der die Kamerakoordinaten in Weltkoordinaten umgerechnet werden. Dieser Vorgang wird hier der Einfachheit halber ebenfalls als Kalibrierung bezeichnet und funktioniert nach dem Prinzip der Projektion aus Kapitel 3.3.2. In diesem Fall erfolgt die Transformation allerdings nicht von Welt- in Kamerakoordinaten, sondern von Kamera- zurück in Weltkoordinaten. Die Transformationsmatrix ist damit ähnlich wie eine zweite extrinsische Matrix zu behandeln. Um die Kalibrierung durchzuführen, kann die Formel (24) verwendet werden. Wie an dieser zu sehen ist, ergibt sich für jeden Punkt, der sowohl in Kamera-, als auch in Weltkoordinaten bekannt ist, eine Gleichung für jede Achse. So entstehen durch Matrixmultiplikation die folgenden Gleichungen.

$$\begin{aligned} x_w &= r_{11} \cdot x_k + r_{12} \cdot y_k + r_{13} \cdot z_k + t_x \cdot 1 \\ y_w &= r_{21} \cdot x_k + r_{22} \cdot y_k + r_{23} \cdot z_k + t_y \cdot 1 \\ z_w &= r_{31} \cdot x_k + r_{32} \cdot y_k + r_{33} \cdot z_k + t_z \cdot 1 \end{aligned} \tag{45}$$

Wie an den Gleichungen zu erkennen ist, besitzt jede vier Unbekannte. Können also vier Punkte ermittelt werden, bei denen sowohl die Kamera-, als auch die Weltkoordinaten bekannt sind, kann daraus ein lineares Gleichungssystem erstellt und dieses gelöst werden. Aufgrund der nahezu parallellaufenden linearen Zusammenhänge und der geringen Abstände zwischen den

Punkten, führt eine minimale Abweichung der Punkte allerdings schon zu einem sehr großen Fehler. Um diese Fehlerquelle zu minimieren, wird ein überbestimmtes System gebildet. Dazu werden weit mehr als die theoretisch benötigten vier Punkte ermittelt. Je mehr Punkte verwendet werden, desto besser sollte das Ergebnis sein. Als Richtwert können hier 50-100 Punkte gewählt werden.

Um diese Punkte zu finden, kann ein gewöhnliches Kalibriermuster mit Schachbrett verwendet werden. Ein Beispiel für ein solches Muster ist im Anhang D zu sehen. Mit Hilfe eines Programms kann das Schachbrettmuster erkannt und dann mit der Methode aus Kapitel 5.7 die Kamerakoordinaten bestimmt werden. Für die ausgewählten Punkte müssen die realen Koordinaten in Bezug auf den Ursprung gemessen werden. Hierbei muss möglichst genau vorgegangen werden, da Abweichungen an dieser Stelle zu ungenauen Transformationen und somit zu ungenauen Becherpositionen führen können. Außerdem sollten die ausgewählten Punkte auf allen drei Achsen möglichst weit auseinander liegen. Da der Messfehler als ein konstanter Wert angenommen werden kann, fällt dieser selbstverständlich deutlich weniger ins Gewicht, wenn die Abstände zwischen den ausgewählten Punkten deutlich größer gewählt werden. Es sollte also versucht werden, das Kalibriermuster möglichst an den Rändern des Sichtfelds der Kamera zu bewegen, um die Größtmögliche Differenz zu erhalten. Um lineare Zusammenhänge zwischen den einzelnen Punkten zu vermeiden, sollten die Punkte außerdem auf möglichst vielen verschiedenen Ebenen in der Kamerareichweite gewählt werden.

Um das Kalibriermuster mit der Kamera zu erkennen, können die Bilder wie in Kapitel 5.3 eingelesen werden. Das eingelesene Farbbild wird dann mit

```
gray = cv2.cvtColor(color, cv2.COLOR_BGR2GRAY)
```

in ein Grauwertbild umgewandelt. Für das Erkennen der Ecken auf dem Schachbrett, sowie das Einzeichnen dieser auf dem Bild, bietet OpenCV bereits fertige Methoden. Die Methode `findChessboardCorners` benötigt dafür nur das Grauwertbild inklusive Schachbrettmuster, sowie die Anzahl der Zeilen und Spalten des Musters. Um die Genauigkeit der erkannten Ecken noch weiter zu erhöhen, kann die Methode `cornerSubPix` genutzt werden. Die Methode nutzt einen iterativen Algorithmus um die erkannte Position noch weiter zu verfeinern. Sie bekommt das Grauwertbild, die erkannten Ecken, die Fenstergröße des Suchfensters, sowie eine Zone übergeben, in der nicht gesucht werden soll. Als fünften Parameter bekommt die Methode ein `TermCriteria` übergeben. Dies ist das Kriterium, nachdem der iterative Algorithmus unterbrochen wird. Das `TermCriteria` wird in der vorherigen Zeile erstellt und bekommt als erstes die Kriterien übergeben, nach denen der Algorithmus abgebrochen wird. In diesem Fall wird geprüft, ob die maximale Anzahl an Iterationen oder die erwünschte Genauigkeit erreicht wird. Die beiden weiteren Parameter beschreiben die maximale Anzahl an Iterationen und die gewünschte Genauigkeit.

```
ret, corners = cv2.findChessboardCorners(gray, (rows, columns
), None) # None = keine flags gesetzt
```

```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITE
R, 30, 0.001) #(TERM_CRITERIA, MAX_ITER, ACCURACY)

corners2 = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -
1), criteria)
```

Die Methode `drawChessboardCorners` benötigt anschließend ebenfalls das Grauwertbild, die Zeilen und Spalten, erhält aber zusätzlich noch die Rückgabewerte aus der vorherigen Methode. Die Variable `ret` speichert, ob ein Muster erfolgreich erkannt wurde und die einzelnen gefundenen Punkte werden in `corners` bzw. nach der Korrektur in `corners2` abgelegt. Mit Hilfe der Eingaben gibt die Methode ein neues Bild mit eingezeichnetem Muster zurück.

```
detected = cv2.drawChessboardCorners(gray, (rows,columns), co
rners2, ret)
```

Das Anzeigen des Bildes erfolgt genau wie in den vorherigen Kapiteln mit Hilfe der Funktion `cv2.imshow`. Das Grauwertbild mit den eingezeichneten Ecken wird in Abbildung 27 gezeigt.

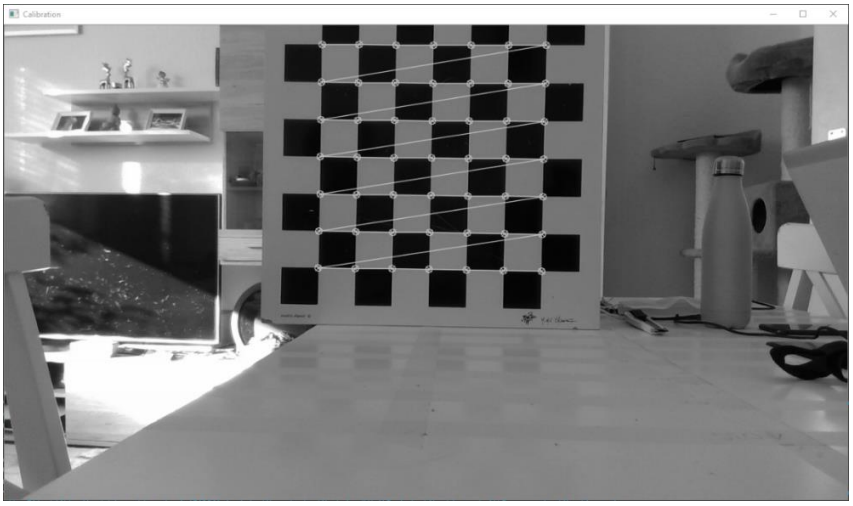


Abbildung 27: Kalibrierung der Transformationsmatrix

Um die Punkte auszuwählen, die für die Kalibrierung verwendet werden sollen, wird dem Bild ein `MouseCallback` hinuzgefügt.

```
cv2.setMouseCallback('Calibration', checkboard_clicked)
```

Die Methode bekommt das ausgewählte Fenster und eine Callback-Methode zugewiesen, die nach einem Doppelklick aufgerufen wird. In der Callback-Methode wird in einer Schleife als erstes ermittelt, welcher Punkt angeklickt wurde. Dafür werden die Koordinaten des Klicks ausgewertet und der Punkt mit der geringsten Distanz gespeichert. Zum Extrahieren der einzelnen Koordinaten der Ecken kann die Methode `corner.ravel()` verwendet werden.

```
for corner in corners2:
        x_c,y_c = corner.ravel()
```

```
distance = math.sqrt((x_c - x)**2 + (y_c - y
)**2)

if distance < shortest:

     shortest = distance

     x_shortest = x_c

     y_shortest = y_c
```

Nach der Ermittlung des angeklickten Punktes werden die Kamerakoordinaten wie in Kapitel 5.7 mit der Funktion `deproject_Pixel_to_Point()` ermittelt. Die Koordinaten der Punkte werden in einem Array gespeichert. Außerdem werden zu jedem Punkt auch die Koordinaten in der realen Welt erfragt und in einem Array abgelegt. Wenn die ausgewählte Anzahl an Punkten mit Welt- und Kamerakoordinaten vorhanden ist, kann die Transformationsmatrix gelöst werden. Da allerdings ein überbestimmtes System vorliegt, kann kein einfaches lineares Gleichungssystem mehr verwendet werden. Deshalb wird für das Lösen der Matrix eine Regression durchgeführt. Im Falle dieser Regression sind die Kamerakoordinaten die unabhängigen Variablen und die dazugehörige X-, Y- oder Z-Koordinate in der echten Welt entspricht der abhängigen Variable. Wenn jetzt die Gleichungen (45) für eine einzelne Weltkoordinate betrachtet werden, ist ersichtlich, dass die Gleichungen drei unabhängige Variablen besitzen. Das System lässt sich also nicht mit einer einfachen Regression lösen. In diesem Falle kann die multiple Regression aus Kapitel 3.2.1.2 genutzt werden. Dazu kann die Bibliothek Scikit verwendet werden, die für solch simple Machine Learning Ansätze einfache Methoden liefert.

```
model = linear_model.LinearRegression()

model.fit(x_matr, y1_matr)
```

Als Parameter bekommt die Regression zum einen ein Array mit drei Spalten und einer Zeile für jeden aufgenommenen Punkt übergeben, dass die unabhängigen Parameter der Gleichungen, also die Kamerakoordinaten beinhaltet. Die vierte Spalte der Transformationsmatrix wird nicht benötigt, da diese von dem Modell als zusätzlicher Parameter automatisch berechnet werden. Der zweite übergebene Parameter ist ein Vektor mit den Ergebnissen der Transformationen für die jeweilige Achse. Diese beiden könnten folgendermaßen aussehen, wobei y der Vektor mit den einzelnen gemessenen X-Koordinaten ist, X die Matrix mit den von der Kamera ermittelten Koordinaten und n die Anzahl der aufgenommenen Punkte:

$$y = \begin{pmatrix} x_{w,1} \\ x_{w,2} \\ \vdots \\ x_{w,n} \end{pmatrix}, X = \begin{pmatrix} x_{k,1} & y_{k,1} & z_{k,1} \\ x_{k,2} & y_{k,2} & z_{k,2} \\ \vdots & \vdots & \vdots \\ x_{k,n} & y_{k,n} & z_{k,n} \end{pmatrix} \tag{46}$$

Die Speicherung der Daten erfolgt also ähnlich der Notation der Herleitung der linearen Regression in Kapitel 3.2.1.1. Mit Hilfe der Regression werden die einzelnen abhängigen Variablen, sowie der dazugehörige Achsenabschnitt berechnet. Diese Werte lassen sich aus dem berechneten Modell als *_coeffs* und *_intercept* auslesen und in die Transformationsmatrix einsetzen. Die drei Koeffizienten entsprechen dabei den Werten der Rotationsmatrix und das *_intercept* der Translation auf der jeweiligen Achse. Da die drei Koordinaten zueinander unabhängig sind und durch eigene Gleichungen beschrieben werden, muss diese Regression für alle drei Koordinaten ausgeführt werden. Die ermittelten Werte können anschließend in der Transformationsmatrix zusammengetragen werden. Wichtig hierbei ist, dass die ermittelte Matrix nur für die aktuelle Position der Kamera gültig ist.

Alternativ kann die Regression auch ohne die Verwendung externer Bibliotheken gelöst werden. Dies ist sinnvoll um die vorherigen Ergebnisse zu kontrollieren und mögliche Fehler zu vermeiden. Das Lösen der Regression ohne Bibliotheken erfolgt nach Formel (8). Wichtig dabei ist, dass die Matrix (46) der unabhängigen Parameter für die Methode noch eine zusätzliche Spalte benötigt. Bei der manuellen Berechnung muss auch der Achsenabschnitt berechnet werden, welcher bei der Berechnung mit Scikit als zusätzlicher Parameter behandelt wurde. Um den Achsenabschnitt ebenfalls mit aufzunehmen, muss die Matrix folgendermaßen erweitert werden:

$$X = \begin{pmatrix} x_{k,1} & y_{k,1} & z_{k,1} & 1 \\ x_{k,2} & y_{k,2} & z_{k,2} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{k,n} & y_{k,n} & z_{k,n} & 1 \end{pmatrix} \tag{47}$$

Durch Einsetzen von X und y in Formel (8) kann der Ergebnisvektor θ berechnet werden. Wie auch bei dem vorherigen Modell muss die Regression für jede Zeile der Matrix einzeln durchgeführt werden. Im Programm werden die drei Zeilen der Transformationsmatrix θ_1, θ_2 und θ_3 als `v_1, v_2` und `v_3` gespeichert.

```
A = np.array(camera_vector_manual) # Speichern der
Kamerakoordinaten als Numpy-Array

A_T = A.T # Speichern der transponierten Matrix

c = A_T.dot(A) # Multiplizieren von A^T * A

d = np.linalg.matrix_power(c, -1) # Potenzieren des
Ergebnisses

e = d.dot(A_T) # Ergebnis mit A^T multiplizieren

# Berechnen der Koeffizienten durch Multiplizieren mit
jeweiligem Ergebnisvektor

v_1 = e.dot(coord_array_world_np[0])

v_2 = e.dot(coord_array_world_np[1])
```

```
v_3 = e.dot(coord_array_world_np[2])
```

Sollte sich die Kamera in ihrer Rotation oder Translation in Bezug auf den Ursprung der Weltkoordinaten verändern, muss auch die Regression zur Ermittlung der Transformationsmatrix erneut durchgeführt werden. Dadurch ist es sinnvoll, die Kamera in einer festen Position anzubringen und die Genauigkeit der Transformationsmatrix in regelmäßigen Abständen zu kontrollieren.

5.9. Umrechnen der Kamera- in Weltkoordinaten

Bei den in Kapitel 5.7 bestimmten Koordinaten handelt es sich noch um die Koordinaten aus dem Kamerakoordinatensystem. Diese Koordinaten sind für die weitere Nutzung noch unbrauchbar, da ihre Position in der realen Welt nicht bestimmbar ist. Mit Hilfe der im vorigen Kapitel berechneten Transformationsmatrix können die Kamerakoordinaten allerdings in Weltkoordinaten umgerechnet werden. Die Umrechnung erfolgt dabei nach Formel (33). Um dies zu bewerkstelligen muss als erstes der Vektor mit den Kamerakoordinaten erweitert werden.

```
coord_vector = np.array([x_w, y_w, z_w, 1])
```

Anschließend kann die Transformationsmatrix der Größe 3x4 mit dem Vektor multipliziert werden, um die Weltkoordinaten zu erhalten. Dies erfolgt mit der folgenden Zeile.

```
world_coords = TRANSFORMATION.dot(coord_vector)
```

Der Vektor mit den berechneten Weltkoordinaten wird als Numpy-Array zurückgegeben. Da die Positionsmessung nur am Außenrand des Bechers durchgeführt werden kann, muss die Breite des Bechers anschließend noch berechnet werden, um den Mittelpunkt des Bechers zu finden. Die Berechnung der Breite erfolgt nach Formel (44). Die Breite kann anschließend halbiert und von dem bekannten Wert für die Tiefe des Bechers abgezogen werden.

```
offset_z = (0.062 + 0.00748 * world_coords[1]) / 2

z_new = world_coords[2] - offset_z
```

Als Ergebnis ergibt sich daraus ein neuer Tiefenwert. Die Position auf der X-Achse bleibt von der Berechnung unverändert. Die Höhe des Bechers ist konstant und beträgt immer 12 cm. Daraus ergeben sich anschließend die folgenden Koordinaten für die Mitte des Bechers.

```
x_mid = world_coords[0]

y_mid = 0.12

z_mid = z_new
```

5.10. Überprüfung linearer Abhängigkeiten mittels der Korrelationsmatrix

Um zu überprüfen, ob die lineare Regression an dieser Stelle sinnvoll ist und genutzt werden darf, müssen die Variablen auf lineare Abhängigkeiten überprüft werden. Weisen die Variablen zueinander lineare Abhängigkeiten auf, dürfen diese nicht für die Regression verwendet werden. Das Überprüfen der linearen Abhängigkeiten erfolgt dabei nach Kapitel 3.1.

Aus den einzelnen Relationen zueinander wird eine Korrelationsmatrix gebildet, anhand derer abgelesen werden kann, ob die Variablen linear unabhängig voneinander sind. Um die besagte Matrix aufzustellen, müssen die in Kapitel 5.8 aufgenommenen Punkte verwendet werden. Wichtig dabei ist, dass die Werte in einem Array mit einer Zeile pro Variable gespeichert werden. Im Falle der 3D-Punkte bedeutet dies, dass die Punkte in einem zweidimensionalen Array mit einer Zeile für jede Achse gespeichert werden, wobei eine Zeile den X-, eine den Y- und eine den Z-Werten der von der Kamera gemessenen Punkte entspricht. Die Anzahl der Werte pro Array-Zeile entspricht der Anzahl der aufgenommenen Punkte. Wenn die Daten in diesem Format gespeichert werden, kann mit Hilfe von Numpy die folgende Funktion verwendet werden um die Matrix zu berechnen.

```
corr = np.corrcoef(coord_array_camera_np)
```

Der Funktion wird das Array mit den 3D-Koordinaten übergeben, welche eine Matrix der Größe 3x3 zurückgibt. Die Matrix hat den in Abbildung 28 gezeigten Aufbau.

	x	y	z
x	1	*	*
y	*	1	*
z	*	*	1

Abbildung 28: Aufbau der Korrelationsmtrix

Die Werte in der Diagonalen sind dabei immer eins, da die Werte logischerweise mit sich selber korrelieren. Anhand der anderen Werte, die hier als * gekennzeichnet sind, können die Beziehungen zwischen den Variablen ausgewertet werden. Um die multiple lineare Regression nutzen zu können, sollten die Werte alle zwischen -0,5 und 0,5 liegen, also keine bzw. nur sehr geringe lineare Abhängigkeiten aufweisen.

6. Evaluation der Ergebnisse

Zum Abschluss werden noch die Ergebnisse der vorher implementierten Systeme vorgestellt und bewertet. Dies ist elementar, um die geleistete Arbeit einzuordnen und mögliche Fehler korrigieren zu können. Für die Messung der Positionen wird das fertig zusammengesetzte Programm genutzt. Eine Abbildung des laufenden Programms ist in Abbildung 29 zu sehen.

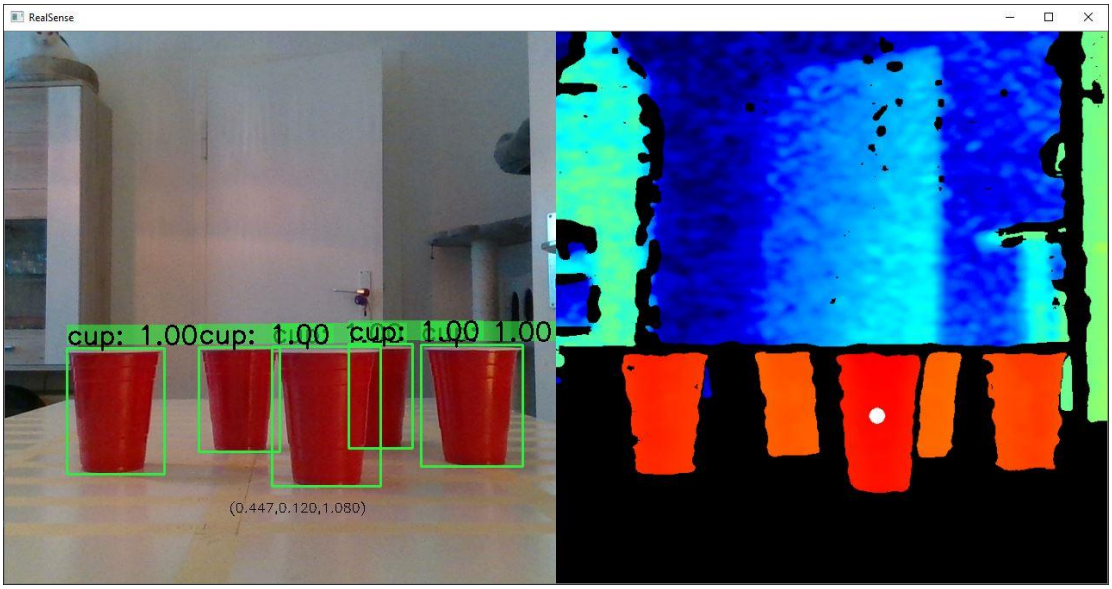


Abbildung 29: Finale Anwendung

Auf der linken Seite der Abbildung sind das Farbbild und die Objekterkennung zu sehen. Unter dem ausgewählten Becher werden die Objektkoordinaten angezeigt. Auf der rechten Seite ist das Tiefenbild der Kamera erkennbar. Der weiße Punkt auf dem Becher zeigt die Messstelle des Systems.

6.1. Genauigkeit der Objekterkennung

Als erstes kann der Loss für das trainierte Modell betrachtet werden. Dieser ist nach den ersten 600 Schritten schon auf 2,0 gefallen und erreicht nach ca. 1000 Schritten einen Wert von 1,0. Anschließend fällt der Loss immer weiter und erreicht zum Ende des Trainings einen Wert von 0,26. Der Verlauf des Loss ist in Abbildung 30 zu sehen.

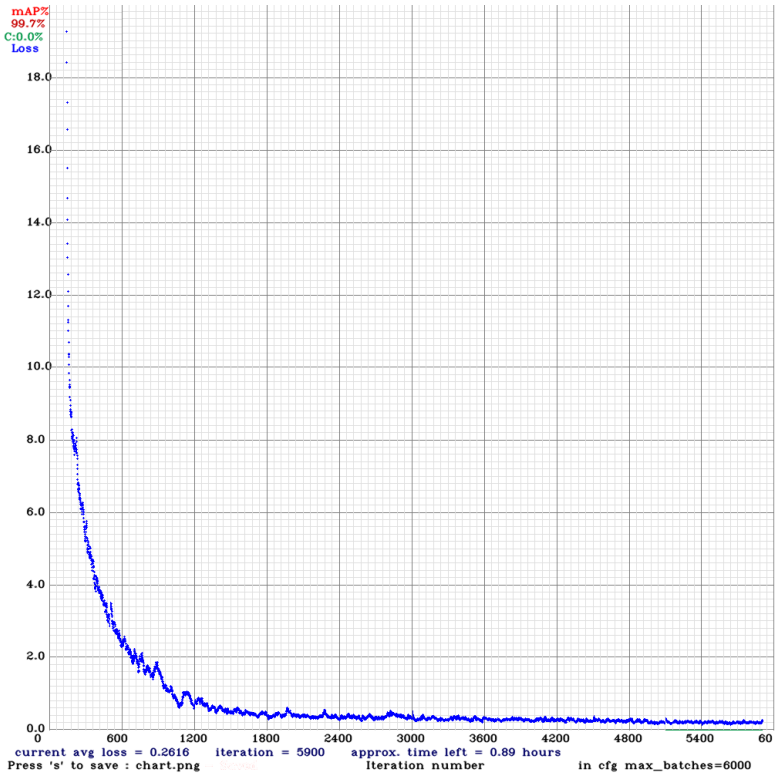


Abbildung 30: Loss-Chart der Objekterkennung

Der Loss deutet darauf hin, dass sich das Modell sehr gut an die Daten angepasst hat. Um die Genauigkeit ebenfalls zu testen und ein mögliches Overfitting auszuschließen, werden die mAP, die durchschnittliche IOU, sowie die TP, FP und FN aufgenommen. Um die Entwicklung des Modells dabei im Blick zu haben, werden für die Überprüfung sechs gespeicherte Checkpoints ab 1000 trainierten Schritten überprüft. Die Ergebnisse werden in Tabelle 2 gezeigt.

Tabelle 2: Genauigkeit des trainierten Modells

Schritte	mAP	av. IOU	TP	FP	FN
1000	0,9962	0,7661	214	13	0
2000	0,9899	0,8315	211	14	3
3000	0,9948	0,8472	213	13	1
4000	0,9959	0,8529	212	13	2
5000	0,9973	0,8792	213	11	1
6000	0,9973	0,8807	212	11	2

Wie an der mAP zu sehen ist, hat das Modell sehr schnell erlernt die Becher mit hoher Genauigkeit zu erkennen. An den Werten der TP, FP und FN ändert sich im Verlauf des Trainings nur sehr wenig. Auffällig ist, dass das Modell trotz des sehr geringen Loss‘ und der sehr hohen mAP nach Ende des Trainings immer noch 11 falsche Objekte im Testdatensatz erkennt. Die einzige Metrik, in der das Modell über den Verlauf des Trainings konstant besser geworden ist, ist die IOU. Nach 1000 Schritten liegt diese bei 76,61%. Nach dem Trainingsende hat diese sich bis auf 88,07% erhöht. Dies ist ein sehr guter Wert und zeigt, dass die Objekte auf den Bildern sehr gut lokalisiert werden. Dies ist wichtig, da die Position der Tiefenmessung

auf der Objekterkennung aufbaut und eine möglichst genaue Lokalisierung des Bechers somit auch die Genauigkeit erhöht.

Um nähere Informationen über die Geschwindigkeit des Modells zu erhalten, kann die Objekterkennung wie in Kapitel 5.1.3 beschrieben, auf ein Video angewendet werden. Das Modell läuft das Video Frame für Frame durch und speichert das Video anschließend an dem gewünschten Ort. Zusätzlich gibt das Modell auch die Geschwindigkeit aus. In Google Colab beträgt diese Geschwindigkeit durchschnittlich 15 FPS. Vergleichsweise kann das Modell auch noch auf einem zweiten System getestet werden. Auf dem zweiten System wird das Modell nur auf einer CPU[17] ausgeführt und erreicht dort eine Geschwindigkeit von ca. 1 FPS.

6.2. Genauigkeit der Positionsmessung

Nachdem die Objekterkennung alleine bereits vielversprechende Ergebnisse erzielt hat, wird an dieser Stelle die Genauigkeit der anschließenden Positionsbestimmung bewertet. Als erstes muss überprüft werden, ob hier eine lineare Regression angebracht ist. Um dies zu gewährleisten, muss die Korrelation der Variablen nach Kapitel 5.10 getestet werden. Für das Aufstellen der Korrelationsmatrix werden die 100 Kalibrierungspunkte verwendet, mit denen auch die Transformationsmatrix der Ergebnisse aus Tabelle 8 berechnet wurde. Dafür wird ein Kalibriermuster der Größe 50x50 cm senkrecht auf einer Ebene in möglichst viele verschiedene Positionen bewegt. Das Muster wird dabei in einem Abstand von 80 cm bis 160 cm aufgenommen. Es kann nicht näher an die Kamera bewegt werden, da andernfalls das Muster nicht vollständig aufgenommen werden kann und die Kamera die Ecken nicht erkennt. Nach hinten kann das Muster ebenfalls nicht weiterbewegt werden, da dort die Ebene endet und eine zuverlässige Aufnahme von Messpunkten dahinter nicht mehr gewährleistet ist. In der Breite wird das Muster von -50 cm bis 50 cm bewegt. In der Höhe wird das Muster nicht verschoben, sodass sich alle Werte im Bereich von 0 cm bis 50 cm bewegen. Die Ergebnisse der Korrelationsmatrix mit den aufgenommenen Werten sind in Tabelle 3 aufgezeigt.

Tabelle 3: Korrelationsmatrix der Regressionsmerkmale

	x	y	z
x	1	0,0342	0,0779
y	0,0342	1	0,0728
z	0,0779	0,0728	1

Wie in der Tabelle zu sehen, ist keiner der Korrelationskoeffizienten größer als 0,1. Dadurch kann ausgeschlossen werden, dass die Variablen untereinander Abhängigkeiten aufweisen und die Regression kann wie geplant verwendet werden.

[17] AMD Ryzen 2600X

Danach wird die Transformationsmatrix für die Kamera nach Kapitel 5.8 aufgestellt. Hierfür dienen im ersten Anlauf insgesamt 48 Messpunkte, aus denen mit Hilfe einer linearen Regression die Transformationsmatrix erstellt wird. Nachdem die Matrix berechnet wurde, werden die Becher an verschiedenen Stellen auf der Ebene positioniert und die Abweichungen von realen Werten und Messwerten berechnet. Für die Positionsermittlung der Becher wird die vorher bereits erwähnte Objekterkennung genutzt und aus der erkannten Position mit Hilfe der Tiefenkamera die Koordinaten bestimmt. Wie in Kapitel 5.9 beschrieben, dient die gemessene Höhe nur der Berechnung des Mittelpunktes des Bechers. Da die Becher auf einer Ebene stehen und immer die gleiche Höhe besitzen, wird die Höhe des Bechers automatisch als Y-Wert genutzt. Für die Genauigkeitsbestimmung ist an dieser Stelle vor allem die Ebene interessant, auf der die Becher in der Anwendung stehen. Für die Überprüfung werden die Becher immer an festgelegten Positionen aufgestellt und die von der Kamera und Transformation ermittelten Koordinaten notiert. Insgesamt werden die Becher an 24 Positionen auf verschiedenen Ebenen positioniert, die in dem Bereich liegen, in dem die Becher auch in der realen Anwendung später aufgestellt werden. Das Koordinatensystem hat den Ursprung dabei zu Beginn in der hinteren linken Ecke der Ebene. Für die erste Transformationsmatrix wurden nur 10 Messwerte aufgenommen. In der Tabelle 4 werden jeweils die gemessene und die reale Position auf der Ebene, sowie der Abstand zur Kamera, die Differenz der Positionen und der absolute und relative Fehler gezeigt.

Tabelle 4: Messergebnisse für Matrix mit 48 Punkten

Nr.	Kamera-distanz [m]	Kamera		Welt		Δx [cm]	Δy [cm]	Abs. Fehler [cm]	Rel. Fehler
		x	z	x	z				
1	1,61	0,526	0,044	0,529	0,043	-0,3	0,1	0,32	0,20%
2	1,543	0,163	0,115	0,157	0,132	0,6	-1,7	1,80	1,17%
3	1,369	0,545	0,293	0,553	0,292	-0,8	0,1	0,81	0,59%
4	1,165	0,357	0,505	0,357	0,504	0	0,1	0,10	0,09%
5	0,965	0,707	0,707	0,72	0,705	-1,3	0,2	1,32	1,36%
6	0,8	0,558	0,876	0,567	0,869	-0,9	0,7	1,14	1,43%
7	0,697	0,32	0,988	0,316	0,983	0,4	0,5	0,64	0,92%
8	0,547	0,463	1,138	0,469	1,124	-0,6	1,4	1,52	2,78%
9	0,458	0,386	1,231	0,386	1,22	0	1,1	1,10	2,40%
10	0,4	0,555	1,286	0,568	1,27	-1,3	1,6	2,06	5,15%
								1,08	1,61%

Wie an den Werten zu erkennen, ist die Abweichung der Positionen zum einen noch sehr hoch und übersteigt gelegentlich sogar 2 cm und zum anderen schwankt die Abweichung noch sehr extrem. Diese Transformationsmatrix ist daher eher ungeeignet und bedarf noch weiterer Verbesserungen. Daher liegt es nahe, die Anzahl der Messpunkte für das Erstellen der Transformationsmatrix zu erhöhen, um so möglicherweise genauere und konstante Ergebnisse zu erzielen.

Die nächste Transformationsmatrix wird mit insgesamt 100 Messpunkten erstellt. Anschließend werden hier erneut eine Reihe von Positionen getestet, um die Genauigkeit der Erkennung zu ermitteln. Nun werden zum Testen immer 24 Messwerte auf verschiedenen Ebenen genutzt, um noch bessere Aussagen über die Genauigkeit treffen zu können. Die Ergebnisse sind in Tabelle 5 zu sehen.

Tabelle 5: Messergebnisse der ersten Messung für Matrix mit 100 Punkten

Nr.	Kamera-distanz [m]	Kamera		Welt		Δx [cm]	Δy [cm]	Abs. Fehler [cm]	Rel. Fehler
		x	z	x	z				
1	1,605	0,415	0,057	0,435	0,05	-2	0,7	2,12	1,32%
2	1,478	0,415	0,193	0,435	0,185	-2	0,8	2,15	1,46%
3	1,385	0,417	0,283	0,435	0,27	-1,8	1,3	2,22	1,60%
4	1,305	0,417	0,365	0,435	0,355	-1,8	1	2,06	1,58%
5	1,134	0,421	0,536	0,435	0,525	-1,4	1,1	1,78	1,57%
6	0,975	0,421	0,702	0,435	0,695	-1,4	0,7	1,57	1,61%
7	0,805	0,423	0,87	0,435	0,865	-1,2	0,5	1,30	1,61%
8	0,638	0,425	1,04	0,435	1,035	-1	0,5	1,12	1,75%
9	1,610	0,173	0,048	0,185	0,05	-1,2	-0,2	1,22	0,76%
10	1,493	0,17	0,176	0,185	0,185	-1,5	-0,9	1,75	1,17%
11	1,404	0,176	0,268	0,185	0,27	-0,9	-0,2	0,92	0,66%
12	1,317	0,179	0,351	0,185	0,355	-0,6	-0,4	0,72	0,55%
13	1,152	0,18	0,524	0,185	0,525	-0,5	-0,1	0,51	0,44%
14	0,982	0,184	0,691	0,185	0,695	-0,1	-0,4	0,41	0,42%
15	0,811	0,184	0,863	0,185	0,865	-0,1	-0,2	0,22	0,28%
16	-	-	-	-	-	-	-	-	-
17	1,613	0,655	0,057	0,685	0,05	-3	0,7	3,08	1,91%
18	1,478	0,654	0,191	0,685	0,185	-3,1	0,6	3,16	2,14%
19	1,404	0,657	0,266	0,685	0,27	-2,8	-0,4	2,83	2,01%
20	1,311	0,657	0,358	0,685	0,355	-2,8	0,3	2,82	2,15%
21	1,153	0,66	0,521	0,685	0,525	-2,5	-0,4	2,53	2,20%
22	0,985	0,662	0,688	0,685	0,695	-2,3	-0,7	2,40	2,44%
23	0,836	0,662	0,862	0,685	0,865	-2,3	-0,3	2,32	2,77%
24	-	-	-	-	-	-	-	-	-
								1,78	0,01472317

Bei dieser Messung schwanken die Messwerte schon deutlich weniger als noch vorher mit weniger Messpunkten für die lineare Regression. Auffällig ist, dass der relative Fehler immer weiter steigt, je weiter der Becher auf der X-Achse bewegt wird. Durch eine nähere Betrachtung der Abweichung fällt auf, dass die X-Werte an einer Position um einen relativ konstanten Wert abweichen. Dadurch, dass dieser Offset für eine feste Lage auf der X-Achse relativ konstant zu sein scheint, erhöht sich automatisch der relative Fehler mit sinkender Distanz zur Kamera.

Durch vorrücken in positiver X-Richtung erhöht sich dieser Offset noch weiter. Da nicht auszuschließen ist, dass diese Abweichungen durch Messfehler entstanden sind, wird das Erstellen der Transformationsmatrix noch einmal wiederholt. Bei den beiden Zeilen ohne Messwerte befand sich der Becher außerhalb des Sichtfeldes, sodass hier keine verlässlichen Werte aufgenommen werden konnten.

Für den nächsten Durchgang wird der Ursprung vom Koordinatensystem allerdings etwas verlegt, um die Messung der realen Koordinaten etwas zu vereinfachen. Vorher hatte das System seinen Ursprung an der hinteren linken Ecke der Testebene. Für diese Testreihe wird das Koordinatensystem in die Mitte der Ebene verrückt, da eine Aufnahme der X-Werte dadurch erleichtert wird. Die Y- und Z-Koordinaten bleiben davon unberührt. Die mit der neuen Matrix aufgenommenen Messwerte sind in Tabelle 6 zu sehen.

Tabelle 6: Messergebnisse der zweiten Messung für Matrix mit 100 Punkten

Nr.	Kamera-distanz [m]	Kamera		Welt		Δx [cm]	Δy [cm]	Abs. Fehler [cm]	Rel. Fehler
		x	z	x	z				
1	1,608	-0,007	0,056	0	0,05	-0,7	0,6	0,92	0,57%
2	1,477	-0,01	0,188	0	0,185	-1	0,3	1,04	0,71%
3	1,383	-0,008	0,279	0	0,27	-0,8	0,9	1,20	0,87%
4	1,304	-0,009	0,362	0	0,355	-0,9	0,7	1,14	0,87%
5	1,134	-0,01	0,535	0	0,525	-1	1	1,41	1,25%
6	0,950	-0,01	0,698	0	0,695	-1	0,3	1,04	1,10%
7	0,805	-0,01	0,871	0	0,865	-1	0,6	1,17	1,45%
8	0,636	-0,011	1,042	0	1,035	-1,1	0,7	1,30	2,05%
9	1,610	-0,248	0,054	-0,25	0,05	0,2	0,4	0,45	0,28%
10	1,491	-0,248	0,175	-0,25	0,185	0,2	-1	1,02	0,68%
11	1,4	-0,248	0,272	-0,25	0,27	0,2	0,2	0,28	0,20%
12	1,312	-0,246	0,352	-0,25	0,355	0,4	-0,3	0,50	0,38%
13	1,15	-0,248	0,518	-0,25	0,525	0,2	-0,7	0,73	0,63%
14	0,981	-0,245	0,691	-0,25	0,695	0,5	-0,4	0,64	0,65%
15	0,81	-0,245	0,865	-0,25	0,865	0,5	0	0,50	0,62%
16	0,64	-0,238	1,037	-0,25	1,035	1,2	0,2	1,22	1,90%
17	1,611	0,231	0,052	0,25	0,05	-1,9	0,2	1,91	1,19%
18	1,489	0,232	0,181	0,25	0,185	-1,8	-0,4	1,84	1,24%
19	1,405	0,232	0,267	0,25	0,27	-1,8	-0,3	1,82	1,30%
20	1,316	0,231	0,353	0,25	0,355	-1,9	-0,2	1,91	1,45%
21	1,15	0,23	0,524	0,25	0,525	-2	-0,1	2,00	1,74%
22	0,984	0,228	0,691	0,25	0,695	-2,2	-0,4	2,24	2,27%
23	0,832	0,23	0,863	0,25	0,865	-2	-0,2	2,01	2,42%
24	-	-	-	-	-			-	-
								1,23	1,12%

Im Vergleich der Messdaten mit denen der vorherigen Messung ist eine deutliche Verbesserung der Abweichung zu erkennen. Der durchschnittliche Fehler liegt bei ca. 1,2 cm und ist im Durchschnitt 5 mm geringer als bei der vorherigen Messung. Unter Betrachtung des Messfehlers der Kamera, sowie möglicher Ungenauigkeiten beim Aufnehmen der Koordinaten, ist dies bereits ein gutes Ergebnis. Bei näherer Betrachtung ist auch hier wieder ein leichter Offset auf der X-Achse zu erkennen. Während die Tiefenwerte sehr konstant unter 1 cm Abweichung liegen, ist für die X-Werte wieder eine Abweichung zu sehen, die sich von ca. -1 cm auf der Breite 0,25 m bis 2 cm auf der Breite 0,25 m bewegt. Da dieses Verhalten sich sehr konstant zeigt, kann versucht werden diesen Fehler über eine zusätzliche Funktion auszugleichen.

Hier bietet sich eine lineare Regression an, die aus den gemessenen X-Werten der Kamera versucht die realen Koordinaten zu modellieren. Als unabhängige Variablen für die lineare Regression werden die X-Werte aller 23 aufgenommenen Punkte genutzt. Als abhängige Variablen werden die real gemessenen und erwarteten Werte verwendet. Aus dieser Regression ergibt sich für diese Kalibrierung der folgende Zusammenhang:

$$x_{korr.} = 0{,}0086 + 1{,}0497 \cdot x_{gemessen} \tag{48}$$

Dieser Zusammenhang kann auf die gemessenen Punkte angewendet und überprüft werden, ob sich die Genauigkeit der Messung verbessert. Die daraus resultierenden Werte sind in Tabelle 7 zu sehen.

Tabelle 7: Mittels linearer Funktion korrigierte Ergebnisse für die zweite Messung

Nr.	Kamera-distanz [m]	Kamera		Δx [cm]	Abs. Fehler [cm]	Rel. Fehler
		x	z			
1	1,608	0,001	0,056	0,1	0,61	0,38%
2	1,477	-0,002	0,188	-0,2	0,35	0,24%
3	1,383	0,000	0,279	0,0	0,90	0,65%
4	1,304	-0,001	0,362	-0,1	0,71	0,54%
5	1,134	-0,002	0,535	-0,2	1,02	0,90%
6	0,950	-0,002	0,698	-0,2	0,35	0,37%
7	0,805	-0,002	0,871	-0,2	0,63	0,78%
8	0,636	-0,003	1,042	-0,3	0,76	1,19%
9	1,610	-0,252	0,054	-0,2	0,44	0,27%
10	1,491	-0,252	0,175	-0,2	1,01	0,68%
11	1,4	-0,252	0,272	-0,2	0,26	0,19%
12	1,312	-0,250	0,352	0,0	0,30	0,23%
13	1,15	-0,252	0,518	-0,2	0,72	0,63%
14	0,981	-0,249	0,691	0,1	0,42	0,43%
15	0,81	-0,249	0,865	0,1	0,14	0,18%
16	0,64	-0,241	1,037	0,9	0,90	1,41%
17	1,611	0,251	0,052	0,1	0,23	0,14%
18	1,489	0,252	0,181	0,2	0,45	0,30%
19	1,405	0,252	0,267	0,2	0,37	0,26%

20	1,316	0,251	0,353	0,1	0,23	0,17%
21	1,15	0,250	0,524	0,0	0,10	0,09%
22	0,984	0,248	0,691	-0,2	0,45	0,46%
23	0,832	0,250	0,863	0,0	0,20	0,24%
24	-	-	-	-	-	-
					0,50	0,47%

Mit den neu berechneten Werten wird eine Verbesserung der durchschnittlichen Abweichung auf nur 0,5 cm erreicht.

Um die im letzten Versuch erreichten Ergebnisse zu bestätigen, wird die Kamera nochmals etwas anders positioniert und das gleiche Verfahren wieder verwendet. Es werden nochmals 100 Kalibrierpunkte aufgenommen und daraus eine Transformationsmatrix gebildet. Die Ergebnisse dieser Transformation sind in Tabelle 8 zu sehen.

Tabelle 8: Messergebnisse der dritten Messung für Matrix mit 100 Punkten

Nr.	**Kamera-distanz [m]**	**Kamera**		**Welt**		Δx **[cm]**	Δz **[cm]**	**Abs. Fehler [cm]**	**Rel. Fehler**
		x	z	x	z				
1	1,623	-0,006	0,057	0	0,05	-0,6	0,7	0,92	0,57%
2	1,498	-0,003	0,19	0	0,185	-0,3	0,5	0,58	0,39%
3	1,407	-0,004	0,276	0	0,27	-0,4	0,6	0,72	0,51%
4	1,321	-0,001	0,368	0	0,355	-0,1	1,3	1,30	0,99%
5	1,154	-0,002	0,532	0	0,525	-0,2	0,7	0,73	0,63%
6	0,989	-0,004	0,701	0	0,695	-0,4	0,6	0,72	0,73%
7	0,825	-0,003	0,867	0	0,865	-0,3	0,2	0,36	0,44%
8	0,655	-0,003	1,039	0	1,035	-0,3	0,4	0,50	0,76%
9	1,638	-0,246	0,038	-0,25	0,05	0,4	-1,2	1,26	0,77%
10	1,515	-0,244	0,178	-0,25	0,185	0,6	-0,7	0,92	0,61%
11	1,42	-0,244	0,263	-0,25	0,27	0,6	-0,7	0,92	0,65%
12	1,335	-0,243	0,346	-0,25	0,355	0,7	-0,9	1,14	0,85%
13	1,168	-0,244	0,522	-0,25	0,525	0,6	-0,3	0,67	0,57%
14	1	-0,243	0,692	-0,25	0,695	0,7	-0,3	0,76	0,76%
15	0,83	-0,241	0,861	-0,25	0,865	0,9	-0,4	0,98	1,19%
16	0,66	-0,235	1,032	-0,25	1,035	1,5	-0,3	1,53	2,32%
17	1,638	0,236	0,043	0,25	0,05	-1,4	-0,7	1,57	0,96%
18	1,506	0,239	0,178	0,25	0,185	-1,1	-0,7	1,30	0,87%
19	1,415	0,238	0,266	0,25	0,27	-1,2	-0,4	1,26	0,89%
20	1,334	0,238	0,349	0,25	0,355	-1,2	-0,6	1,34	1,01%
21	1,161	0,24	0,524	0,25	0,525	-1	-0,1	1,00	0,87%
22	0,995	0,238	0,695	0,25	0,695	-1,2	0	1,20	1,21%
23	0,83	0,24	0,858	0,25	0,865	-1	-0,7	1,22	1,47%
24	-	-	-	-	-	-	-	-	-
								1,00	0,87%

Die Ergebnisse ähneln denen der vorherigen Messungen. Auch hier kann somit wieder das vorher erwähnte Verfahren verwendet werden um den Offset auf der X-Achse auszugleichen. Die neu mit der Ausgleichsfunktion bestimmten Werte sind in Tabelle 9 zu sehen.

Tabelle 9: Mittels linearer Funktion korrigierte Ergebnisse für die dritte Messung

Nr.	Kamera-distanz [m]	Kamera		Δx [cm]	Abs. Fehler [cm]	Rel. Fehler
		x	z			
1	1,623	-0,004	0,057	-0,4	0,79	0,49%
2	1,498	-0,001	0,19	-0,1	0,50	0,34%
3	1,407	-0,002	0,276	-0,2	0,62	0,44%
4	1,321	0,002	0,368	0,2	1,31	0,99%
5	1,154	0,000	0,532	0,0	0,70	0,61%
6	0,989	-0,002	0,701	-0,2	0,62	0,63%
7	0,825	-0,001	0,867	-0,1	0,21	0,25%
8	0,655	-0,001	1,039	-0,1	0,40	0,62%
9	1,638	-0,253	0,038	-0,3	1,24	0,76%
10	1,515	-0,251	0,178	-0,1	0,71	0,47%
11	1,42	-0,251	0,263	-0,1	0,71	0,50%
12	1,335	-0,250	0,346	0,0	0,90	0,67%
13	1,168	-0,251	0,522	-0,1	0,32	0,27%
14	1	-0,250	0,692	0,0	0,30	0,30%
15	0,83	-0,248	0,861	0,2	0,45	0,54%
16	0,66	-0,242	1,032	0,8	0,88	1,33%
17	1,638	0,248	0,043	-0,2	0,73	0,45%
18	1,506	0,251	0,178	0,1	0,71	0,47%
19	1,415	0,250	0,266	0,0	0,40	0,28%
20	1,334	0,250	0,349	0,0	0,60	0,45%
21	1,161	0,252	0,524	0,2	0,23	0,20%
22	0,995	0,250	0,695	0,0	0,00	0,00%
23	0,83	0,252	0,858	0,2	0,73	0,88%
24	-	-	-	-	-	-
					0,61	0,52%

Hier ist ebenfalls wieder ein deutlicher Anstieg der Genauigkeit zu sehen. Das Ergebnis ist allerdings etwas schlechter als noch bei den vorherigen Messungen. Die durchschnittliche absolute Abweichung steigt von 0,5 cm auf 0,6 cm und der durchschnittliche Fehler von 0,47% auf 0.52%. Die Ergebnisse sind dennoch sehr genau und können bestätigen, dass eine Genauigkeit deutlich unter 1 cm bzw. 1% möglich ist.

Zum Schluss muss noch ausgeschlossen werden, dass möglicherweise Fehler bei der Regression entstanden sind. Um das genutzte Verfahren zu bestätigen, wird auch das in Kapitel 5.8

beschriebene alternative Verfahren angewendet. Eine Berechnung der Regression ohne Nutzung von Fremdbibliotheken ergibt die folgende Transformationsmatrix:

$$A_{Manuell} = \begin{bmatrix} 1{,}0003 & 0{,}0081 & 0{,}006 & -0{,}013 \\ -0{,}0128 & 0{,}9897 & -0{,}0453 & 0{,}1345 \\ 0{,}0018 & 0{,}0478 & 1{,}0102 & 1{,}7363 \end{bmatrix} \tag{49}$$

Zum Vergleich kann die Transformationsmatrix betrachtet werden, die unter Verwendung der Bibliothek Scikit berechnet wurde.

$$A_{Scikit} = \begin{bmatrix} 1{,}0003 & 0{,}0081 & 0{,}006 & -0{,}013 \\ -0{,}0128 & 0{,}9897 & -0{,}0453 & 0{,}1345 \\ 0{,}0018 & 0{,}0478 & 1{,}0102 & 1{,}7363 \end{bmatrix} \tag{50}$$

Die berechneten Matrizen bleiben mit den unterschiedlichen Methoden exakt gleich. Es kann also davon ausgegangen werden, dass das Verfahren korrekte Ergebnisse liefert und die Ergebnisse somit auch korrekt berechnet wurden.

7. Fazit und Ausblick

Abschließend kann gesagt werden, dass die Umsetzung der Tiefenmessung als Erfolg angesehen werden kann. Die Objekterkennung mit dem YOLOv4-Modell erreicht eine sehr gute Genauigkeit. Es werden so gut wie alle Becher im normalen Arbeitsbereich erkannt. Es gibt lediglich gelegentliche Falscherkennungen, die aber aufgrund ihrer Position für die Anwendung nicht relevant sind. Die Geschwindigkeit liegt ca. im Bereich von 1 FPS. Da es sich aber bei der Anwendung um einen sehr starren Aufbau handelt, ist dies nicht von Belang. Theoretisch würde es sogar reichen nur ein Bild zu Beginn aufzunehmen und erst mit der Erkennung fortzufahren, wenn dies vom Benutzer erwünscht ist. Diese Objekterkennung ist eine solide Grundlage für die darauf aufbauende Tiefenmessung. Diese erreicht ebenfalls sehr gute Genauigkeiten. Ohne eine Korrektur der Daten liegt die absolute Abweichung der gemessenen Werte ca. bei 1 cm. Bei einer Becherbreite von 9,8 cm ist der Wert akzeptabel. Durch die zusätzliche Erstellung eines linearen Modells, das den Offset in der Breite noch verringern kann, werden sogar durchschnittliche Genauigkeiten im Bereich von 0,5 cm erreicht. Dies ist ein sehr guter Wert, auf dem im weiteren Verlauf des Gesamtprojekts aufgebaut werden kann. Dazu muss noch gesagt werden, dass all diese Werte mit gewöhnlichem Messwerkzeug in der Heimanwendung gemessen wurden. Durch die Nutzung von Präzisionswerkzeug und möglicherweise noch genaueren Kalibriermustern, könnte die Genauigkeit der Positionsmessung ggf. sogar noch erhöht werden. Außerdem könnte eine Nutzung eines größeren Arbeitsbereichs die Ergebnisse der Regression verbessern, sodass diese genauer werden oder zumindest keine Nachbearbeitung der Messwerte nötig ist. Hierbei muss in der Anwendung aber immer abgewogen werden, ob eine mögliche Verbesserung um wenige Millimeter die Kosten und den Aufwand Wert sind. Da

die Kamera von sich aus bereits einen Fehler mitbringt und mit den angewendeten Methoden eine Genauigkeit von 5 mm erreicht werden konnte, ist nur noch eine sehr kleine Genauigkeitsverbesserung denkbar.

Leider ist das Aufstellen der Transformationsmatrix ein sehr langwieriger Prozess. Es müssen ca. 100 Punkte von der Kamera erkannt und anschließend in den realen Koordinaten vermessen werden. Hierbei muss besonders vorsichtig vorgegangen werden, um keine größeren Messfehler einzubauen, die das Ergebnis deutlich verschlechtern können. Da die Kamera in einer Anwendung aber nicht jedes Mal neu verschoben wird, muss diese Matrix glücklicherweise nicht jedes Mal neu aufgestellt werden. Daher ist es auch sinnvoll die Kamera so lange wie möglich unbewegt zu lassen, da eine Verschiebung der Kamera zur Folge hätte, dass die ursprünglich aufgenommene Matrix ihre Gültigkeit verliert. Möglicherweise ist es auch sinnvoll, die Kamera in einer Vorrichtung anzubringen, sodass sie zum einen eine feste Position hat und zum anderen auch nicht durch ungewollte Berührungen bewegt werden kann.

Nach der Beendigung dieses Projekts sind weitere Anpassungen und Verbesserungen denkbar. So könnte die Handhabung des Kalibierprogramms weiter verbessert werden, um benutzerfreundlicher zu werden. Es könnte möglicherweise eine GUI erstellt werden, in der falsche Messwerte korrigiert oder entfernt werden können. Außerdem wäre eine bereits erwähnte Vorrichtung für die Anbringung der Kamera sinnvoll. Ansonsten ist der nächste Schritt nach der Beendigung dieses Projekts natürlich das Zusammensetzen der verschiedenen Teilprojekte, sobald diese fertiggestellt sind. Das Verbinden der Projekte ist nochmals ein großer Schritt und wird zeigen, wie gut die Einzelprojekte sind und ob das eigentliche Ziel wirklich erreicht werden kann.

8. Literaturverzeichnis

Ambalina, L. 2020. *What is the Difference Between CNN and RNN? [online]*. Verfügbar unter https://lionbridge.ai/articles/difference-between-cnn-and-rnn/#:~:text=The%20main%20difference%20between%20CNN,as%20a%20sentence%20for%20example.&text=Whereas%2C%20RNNs%20reuse%20activation%20functions,next%20output%20in%20a%20series.] [abgerufen am 12. Februar 2021].

Beyki, M. 2021. *Kurven, Trajektorien & Machine Learning. Überblick und Verbindung*. Hannover.

Bochkovskiy, A. 2020. *Yolo v4, v3 and v2 for Windows and Linux [online]*. Verfügbar unter https://github.com/AlexeyAB/darknet [abgerufen am 10. Dezember 2020].

Bochkovskiy, A. et al. 2020. *YOLOv4: Optimal Speed and Accuracy of Object Detection [online]*.

Bortz, J. und Schuster, C. 2010. *Statistik für Human- und Sozialwissenschaftler*. 7., vollständig überarbeitete und erw. Aufl. Berlin [und andere]: Springer.

Chollet, F. 2018. *Deep Learning mit Python und Keras. Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. Frechen: MITP Verlags GmbH & Co. KG.

Deru, M. und Ndiaye, A. 2020. *Deep Learning mit TensorFlow, Keras und TensorFlow.js*. 2., aktualisierte und erweiterte Auflage. Bonn: Rheinwerk Verlag.

Floemer, A. 2018. *Googles KI klingt am Telefon wie ein Mensch – und kann Termine vereinbaren [online]*. Verfügbar unter https://t3n.de/news/google-assistant-telefonieren-1077901/ [abgerufen am 12. Februar 2021].

Funke, N. 2016. *Einsatz von Bilderkennungsalgorithmen zur Echtzeitanalyse von Spielwahrscheinlichkeiten beim Texas Hold'em*. Bachelorarbeit. Hannover.

Géron, A. 2020. *Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow. Konzepte, Tools und Techniken für intelligente Systeme*. 2. Auflage. Heidelberg: dpunkt.verlag GmbH.

Holland, M. 2021. *Analyse: KI-Sprachmodell GPT-3 hegt tief verankerte Vorurteile gegen Muslime [online]*. Verfügbar unter https://www.heise.de/news/Analyse-KI-Sprachmodell-GPT-3-hegt-tief-verankerte-Vorurteile-gegen-Muslime-5034341.html [abgerufen am 3. Februar 2021].

Hosang, J. et al. 2017. Learning non-maximum suppression.

Jiang, X. und Bunke, H. 1997. *Dreidimensionales Computersehen. Gewinnung und Analyse von Tiefenbildern*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Kershaw, G. und Eisenschmid, C. 2019a. *Beginner's guide to depth [online]*. Verfügbar unter https://www.intelrealsense.com/beginners-guide-to-depth/ [abgerufen am 10. Februar 2021].

Kershaw, G. und Eisenschmid, C. 2019b. *Which Intel RealSense device is right for you? [online]*. Verfügbar unter https://www.intelrealsense.com/which-device-is-right-for-you/ [abgerufen am 10. Februar 2021].

M. Matsushima et al. 2005. A learning approach to robotic table tennis [online]. *IEEE Transactions on Robotics*. doi: 10.1109/TRO.2005.844689.

M. Tan et al. 2020. EfficientDet: Scalable and Efficient Object Detection. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10778-10787.

Marr, D. und Poggio, T. 1979. A Computational Theory of Human Stereo Vision [online]. *Proceedings of the Royal Society of London. Series B, Containing papers of a Biological character. Royal Society (Great Britain)*. doi: 10.1098/rspb.1979.0029.

Olah, C. 2015. *Understanding LSTM Networks [online]*. Verfügbar unter https://colah.github.io/posts/2015-08-Understanding-LSTMs/ [abgerufen am 12. Februar 2021].

Oliphant, T. 2007. Python for Scientific Computing [online]. *Computing in Science & Engineering*. doi: 10.1109/MCSE.2007.58.

Padilla, R. et al. 2020. A Survey on Performance Metrics for Object-Detection Algorithms. In:

Parbel, M. 2019. *Umfrage: Python ist populärste Programmiersprache für Data Science [online]*. Verfügbar unter https://www.heise.de/developer/meldung/Umfrage-Python-ist-populaerste-Programmiersprache-fuer-Data-Science-4300782.html.

Sackewitz, M. [Hrsg.] 2017. *Handbuch zur industriellen Bildverarbeitung. Qualitätssicherung in der Praxis*. 3., vollständig überarbeitet und aktualisierte Auflage. Stuttgart: Fraunhofer Verlag.

Schreer, O. 2005. *Stereoanalyse und Bildsynthese*. Berlin: Springer.

Soviany, P. und Ionescu, R. 2018. Optimizing the Trade-off between Single-Stage and Two-Stage Object Detectors using Image Difficulty Prediction.

Süße, H. und Rodner, E. 2014. *Bildverarbeitung und Objekterkennung. Computer Vision in Industrie und Medizin*. Aufl. 2014. Wiesbaden: Springer Fachmedien Wiesbaden.

Teich, I. 2020. Meilensteine der Entwicklung Künstlicher Intelligenz [online]. *Informatik Spektrum*. doi: 10.1007/s00287-020-01280-5.

TIOBE Software BV. 2021. *TIOBE Index for February 2021 [online]*. Verfügbar unter https://www.tiobe.com/tiobe-index/ [abgerufen am 13. Februar 2021].

Torrey, L. und Shavlik, J. 2010. Transfer Learning. In: EMILIO SORIA OLIVAS et al. [Hrsg.] *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. Hershey, PA, USA: IGI Global. 242-264.

Toshev, A. und Szegedy, C. 2014. DeepPose: Human Pose Estimation via Deep Neural Networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Verhelst, M. und Moons, B. 2018. Embedded Deep Neural Network Processing: Algorithmic and Processor Techniques Bring Deep Learning to IoT and Edge Devices [online]. *IEEE Solid-State Circuits Magazine*. doi: 10.1109/MSSC.2017.2745818.

Walsh, J. et al. 2019. Deep Learning vs. Traditional Computer Vision. In:

Z. Zhang. 2012. Microsoft Kinect Sensor and Its Effect [online]. *IEEE MultiMedia*. doi: 10.1109/MMUL.2012.24.

Zeng, A. et al. 2020. TossingBot: Learning to Throw Arbitrary Objects With Residual Physics [online]. *IEEE Transactions on Robotics*. doi: 10.1109/TRO.2020.2988642.

Zhang, Z. 2000. A Flexible New Technique for Camera Calibration [online]. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*. doi: 10.1109/34.888718.

Zhao, Z.-Q. et al. 2019. Object Detection With Deep Learning: A Review [online]. *IEEE Transactions on Neural Networks and Learning Systems*. doi: 10.1109/TNNLS.2018.2876865.

Zivingy, M. 2013. Object distance measurement by stereo vision. *International Journal of Science and Applied Information Technology (IJSAIT)*. 2: 5-8.

Anhang

In diesem Kapitel sind diverse Anhänge zu finden, auf die in der Arbeit verwiesen wurde. Dokumente und Programme, die zu lang für den Anhang sind, werden der Arbeit als Anlagen beigelegt. Die in Anhang A und B erwähnten Dokumente sind den Anlagen zu entnehmen und der Arbeit digital beigefügt.

A. Installation der Software

Vor der Beschreibung der Installation werden die genauen Versionen der verwendeten Software beschrieben. Diese sind in Tabelle 10 zu sehen.

Tabelle 10: Verwendete Software

Software	Version	Download-Link
Windows 10 Pro	-	-
Visual Studio Code	1.53.2	https://code.visualstudio.com/Download
Anaconda	2020.11	https://repo.anaconda.com/archive/
Python	3.7	Bereits durch Anaconda installiert
LabelImg	1.8.1	https://github.com/tzutalin/labelImg

Auf die weiteren, hier nicht aufgezählten Tools, wird im weiteren Verlauf des Kapitels bzw. in Abbildung 31 eingegangen.

Objekterkennung

Die Objekterkennung wird nicht lokal auf dem Computer, sondern in Google Colab trainiert. Dadurch entfällt das Installieren von Anwendungen. Für das Training wird lediglich ein Zugang zu dem Notebook benötigt. Das Notebook ist in den Anlagen unter dem Namen ‚cupdetection.ipynb' zu finden. Es kann in Google Colab geöffnet und dort Schritt für Schritt abgearbeitet werden. Dabei wird sowohl die Installation, als auch das Training des Netzwerks erklärt.

Tiefenmessung

Zu allererst müssen selbstverständlich die benötigten Tools installiert werden, mit denen im späteren Verlauf noch gearbeitet wird. Der Einfachheit halber wurde das Projekt unter der Nutzung von Anaconda erstellt. Dies hat den Vorteil, dass die verschiedenen Bibliotheken sehr einfach installiert und die Versionen verändert werden können. Außerdem können die erstellten Umgebungen exportiert und geteilt werden. Dies vereinfacht vor allem die Installation auf weiteren Computern und somit die Weitergabe des Projektes. Um die Umgebung zu exportieren

können, muss Anaconda auf dem Computer installiert werden. Anaconda kann hier[18] heruntergeladen und mit dem grafischen Installer installiert werden. Nach der Installation muss der Installationspfad ggf. noch zu den Umgebungsvariablen hinzugefügt werden. Anschließend kann die exportierte Umgebung auf dem Computer installiert werden. Diese ist unter dem Namen ‚cupdetection.yml' in den Anlagen zu finden. Das Installieren der Umgebung erfolgt mit Hilfe des folgenden Befehls.

```
conda env create -f <Pfad zu Datei>\cupdetection.yml
```

Nach dem Ausführen des Befehls werden die verschiedenen Versionen der Umgebung heruntergeladen und installiert. Dafür wird eine Internetverbindung benötigt. Nach der Installation sollten die in Abbildung 31 gezeigten Versionen installiert sein.

# Name	Version	Build	Channel
ca-certificates	2020.12.8	haa95532_0	
certifi	2020.12.5	py37haa95532_0	
cycler	0.10.0	pypi_0	pypi
kiwisolver	1.3.1	pypi_0	pypi
matplotlib	3.3.3	pypi_0	pypi
numpy	1.19.4	pypi_0	pypi
opencv-contrib-python	4.4.0.46	pypi_0	pypi
openssl	1.1.1i	h2bbff1b_0	
pillow	8.0.1	pypi_0	pypi
pip	20.3.3	py37haa95532_0	
pyparsing	2.4.7	pypi_0	pypi
pyrealsense2	2.41.0.2666	pypi_0	pypi
python	3.7.9	h60c2a47_0	
python-dateutil	2.8.1	pypi_0	pypi
setuptools	51.0.0	py37haa95532_2	
six	1.15.0	pypi_0	pypi
sqlite	3.33.0	h2a8f88b_0	
vc	14.2	h21ff451_1	
vs2015_runtime	14.27.29016	h5e58377_2	
wheel	0.36.2	pyhd3eb1b0_0	
wincertstore	0.2	py37_0	
zlib	1.2.11	h62dcd97_4	

Abbildung 31: Installierte Bibliotheken mit dazugehörigen Versionen

Um die Versionen der Umgebung zu überprüfen kann die Umgebung mit

```
conda activate cupdetection
```

aktiviert und mit

```
conda list
```

die Liste der Bibliotheken mit dazugehörigen Versionen angezeigt werden. Sollten möglicherweise Pakete fehlen oder die falsche Version installiert sein, kann dies mit Hilfe von

[18] https://www.anaconda.com/products/individual#Downloads

Anaconda schnell angepasst werden. Durch den folgenden Befehl kann ein Paket in der gewünschten Version installiert werden.

```
conda install <Paket>=<Version>
```

Realsense Viewer, Calibration & Quality Tool

Diese Tools werden von Intel bereitgestellt und können hilfreich sein, werden aber nicht zwingend benötigt. Dies ist zum einen der Realsense Viewer[19]. In diesem können die Kamerastreams angezeigt und verschiedene Einstellungen getestet werden. Dieser kann einfach heruntergeladen und direkt gestartet werden. Die .exe zum Herunterladen befindet sich in dem angegeben Github-Repository unter ‚Assets'.

Ein zweites hilfreiches Tool ist das Depth Quality Tool. Dieses dient dazu die Qualität der Tiefenbilder zu überprüfen und soll dabei helfen zu entscheiden, ob eine neue Kalibrierung der Kamera nötig ist. Dieses kann ebenfalls unter dem Github-Repository heruntergeladen und umgehend ohne Installation gestartet werden.

Das dritte und letzte Tool von Intel ist das Realsense Calibration Tool[20]. Dieses Tool dient dazu die Kamera neu zu Kalibrieren. Das Tool muss heruntergeladen, entpackt und anschließend installiert werden. Zusätzlich zum Installer liegen noch Anleitungen bezüglich der Kamerakalibrierung bei. Für die Kalibrierung der Kamera wird außerdem noch eine App benötigt, die im Play Store und im App Store zur Verfügung steht. Diese App heißt ‚Dynamic Target Tool'.

B. Bedienungsanleitung

Nachdem die Installation der benötigten Tools fertiggestellt wurde, können die Programme gestartet und genutzt werden. Zuallererst kann geprüft werden, ob ggf. eine neue Kalibrierung benötigt wird. Dafür kann das Intel Depth Quality Tool gestartet und die Anweisungen des Programms ausgeführt werden. Die Kamera muss in einer Entfernung von 50 cm bis 100 cm vor eine flache Oberfläche wie eine Wand gestellt werden und diese betrachten. Anschließend wird ausgegeben, ob eine Kalibrierung benötigt wird oder ob die aktuellen Parameter weiterverwendet werden können. Die Oberfläche des Quality Tools ist in Abbildung 32 zu sehen.

[19] https://github.com/IntelRealSense/librealsense/releases/tag/v2.42.0
[20] https://downloadcenter.intel.com/download/29618/Intel-RealSense-D400-Series-Dynamic-Calibration-Tool

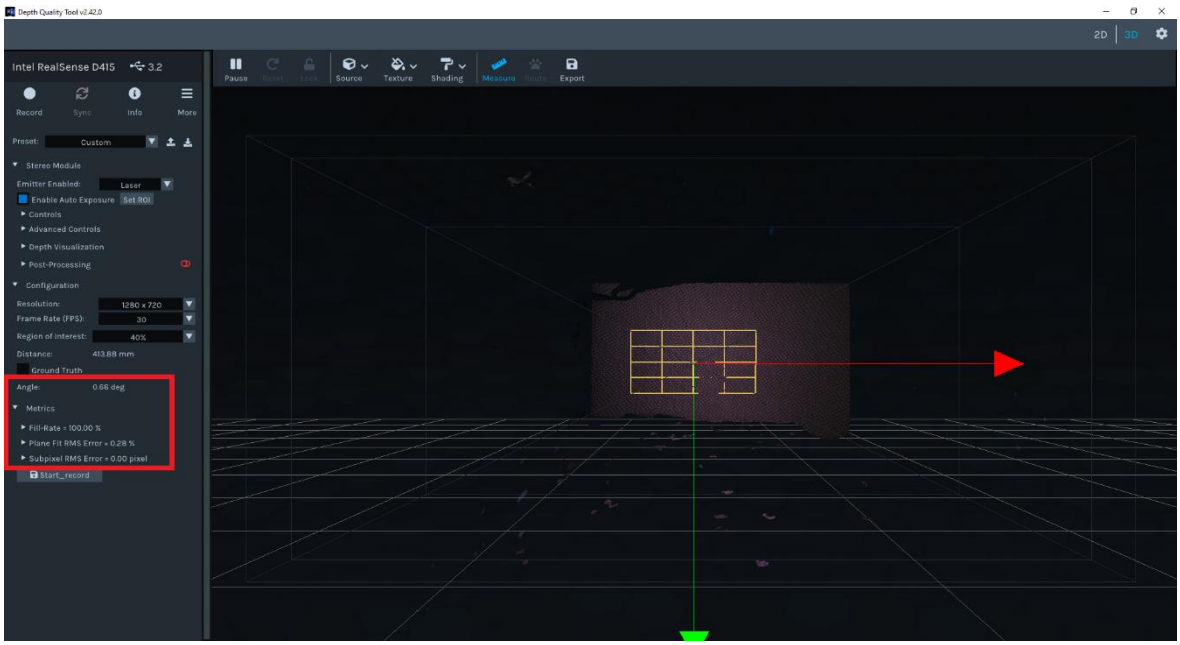


Abbildung 32: Depth Quality Tool

Auf der linken Seite der Oberfläche können die Kameraeinstellungen vorgenommen und an die persönlichen Präferenzen angepasst werden. In dem rot markierten Rechteck darunter werden die Qualitätsmaßstäbe angezeigt. Die Fill-Rate sollte möglichst nah an 100% und die beiden darunter stehenden RMS-Fehler möglichst gering sein.

Sollte eine Kalibrierung benötigt werden, kann das Intel Calibration Tool gestartet werden. Das Programm führt Schritt für Schritt durch die Kalibrierung. Wichtig dabei ist, dass das Kalibriermuster für das Smartphone heruntergeladen wurde und bereit ist. Das Muster wird dann in verschiedenen Schritten ca. 60-80 cm entfernt von der Kamera bewegt und von der Kamera aufgenommen. Die Kamera nimmt dabei selbstständig verschiedene Punkte auf und wechselt zum nächsten Schritt, sobald der vorherige abgeschlossen wurde. Sobald die Kalibrierung fertiggestellt ist, werden die neuen Kameraparameter angezeigt und können in die Kamera übernommen werden.

Nachdem die Kamera kalibriert wurde, kann sie an die gewünschte Position bewegt werden. Mit der Hilfe des Realsense Viewers kann während der Positionierung jederzeit das aktuelle Sichtfeld der Kamera überprüft werden. Sobald die Kamera am richtigen Punkt steht, sollte diese nicht mehr bewegt werden und die Transformationsmatrix kann aufgenommen werden. Die weiteren Programme werden alle in der Windows-Konsole gestartet. Das Berechnen der Transformationsmatrix erfolgt mit Hilfe der Datei ‚calibration.py'. Um das Programm zu starten muss die Anaconda-Umgebung in der Konsole mit

```
conda activate cupdetection
```

gestartet werden. Anschließend muss in das Programmverzeichnis gewechselt werden.

```
cd <Pfad zu Programmen>
```

Für die anschließende Ausführung der Programme ist es wichtig, dass die Kamera über ein USB 3.0 Port angeschlossen wird. Ältere USB-Versionen können zu Problemen führen, da durch die verringerte Übertragungsgeschwindigkeit möglicherweise nicht beide Streams angezeigt werden können. Das Programm für die Kalibrierung kann mit folgendem Befehl gestartet werden.

```
python calibration.py
```

Wichtig dabei ist, dass die vorhandene Ordnerstruktur der mitgelieferten Programme nicht verändert wird. Im gleichen Ordner, in dem die Programme liegen, sollte auch ein Ordner mit dem Namen ‚yolov4' liegen. Dieser Ordner enthält verschiedene Dateien, die für die spätere Objekterkennung eingelesen werden. Vor dem Start des Programms muss möglicherweise noch die Anzahl der Reihen und Spalten des Musters angepasst werden. Sollte ein anderes Kalibriermuster verwendet werden, kann dies in den Programmzeilen 176 und 177 eingestellt werden. Falls sich nach dem Start des Programms kein Fenster öffnet, liegt das daran, dass die Kamera kein Kalibriermuster erkennt. In diesem Fall sollte das Muster etwas bewegt werden, bis ein Bild der Kamera mit erkanntem Schachbrett angezeigt wird. Ggf. kann mit dem Intel Realsense Viewer vorher auch überprüft werden, in welchem Bereich das Muster noch von der Kamera gesehen wird. Sobald die Kamera das Muster erkennt, sollte ein Bild wie in Abbildung 27 zu sehen sein. Durch Drücken der Leertaste kann ein neues Bild aufgenommen werden. Das Bild bleibt bis zum nächsten Druck der Leertaste bestehen. Auf dem Bild kann nun ein Punkt ausgewählt werden, der für die Berechnung der Transformationsmatrix verwendet werden soll. Das Auswählen des Punktes erfolgt durch einen Doppelklick auf diesen. Das Programm berechnet eigenständig welcher Punkt dem Klick am nächsten ist und wählt diesen aus. Anschließend müssen zu dem Punkt in der Konsole die Koordinaten in der echten Welt eingegeben werden. Die Koordinaten werden gespeichert und es kann mit dem nächsten Punkt fortgefahren werden. Wenn die Kamera die Koordinaten für den ausgewählten Punkt berechnen konnte, wird dieser als weißer Punkt angezeigt. Sollten für den Punkt keine Tiefendaten vorliegen, wird der Punkt in schwarz angezeigt und es kann ein anderer Punkt ausgewählt werden. Das Auswählen der Punkte sollte nach den Vorgaben aus Kapitel 5.8 erfolgen, um eine möglichst genaue Matrix zu berechnen. Sobald insgesamt 100 Punkte aufgenommen wurden, wird die Transformationsmatrix in der Konsole ausgeben und die vorhandenen Punkte in Form von json-Arrays in der Datei ‚Kalibrierung.json' gespeichert. Wenn weniger als die standardmäßigen 100 Punkte verwendet werden sollen, kann die Anzahl der Punkte in der Programmzeile 162 geändert werden. Wichtig beim Eingeben der Koordinaten in der Konsole ist, dass die Dezimalstellen durch Punkte und nicht durch Kommata getrennt werden.

Alternativ kann ein zweites Programm zum Berechnen der Transformationsmatrix genutzt werden. Voraussetzung dafür, dass das Programm genutzt werden kann, ist die Nutzung des Kalibriermusters der Hochschule, das in Abbildung 33 zu sehen ist. Das Programm verhält sich grundsätzlich genauso wie das vorherige, nur dass die realen Koordinaten nicht für jeden Punkt neu gemessen und eingegeben werden müssen. Anstatt der X-, Y- und Z-Koordinaten in der realen Welt, erfragt das Programm die Position der linken Kante des Musters auf der X-Achse,

die Reihe und Spalte des ausgewählten Punktes und die Tiefe des Musters. Aus diesen Werten berechnet das Programm selbstständig die X- und Y-Koordinaten der Punkte. Dadurch können an einer Position des Musters mehrere Punkte aufgenommen werden, ohne dass die Koordinaten jedes einzelnen Punktes gemessen werden müssen. Die Nummerierungen der Zeilen und Spalten sind ebenfalls in Abbildung 33 zu sehen.

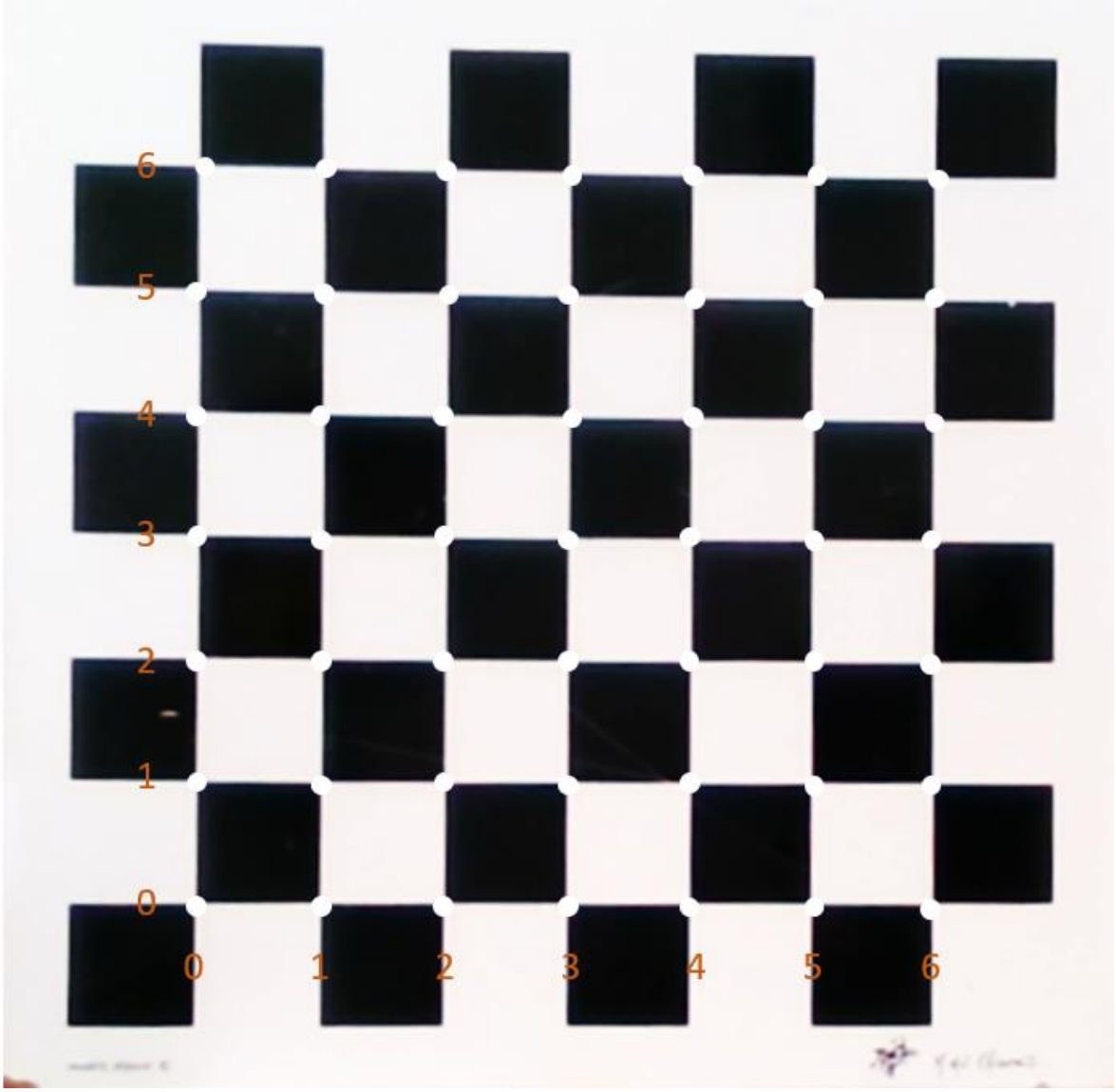


Abbildung 33: Kalibriermuster mit eingezeichneten Eckpunkten, Zeilen- und Spaltennummern

Bei der Umsetzung dieses Verfahrens muss allerdings sehr genau darauf geachtet werden, dass das Muster parallel zur X-Y-Ebene steht und die gemessene Tiefe somit für alle Punkte gleich ist. Das angepasste Programm für die Berechnung der Transformationsmatrix trägt den Namen ‚calibration_auto.py' und kann mit folgendem Befehl gestartet werden.

```
python calibration_auto.py
```

Das Ändern der aufzunehmenden Punkte für dieses Programm erfolgt in Zeile 170 der Datei ‚calibration_auto.py'.

Die ausgegebene Transformationsmatrix muss anschließend im Hauptprogramm angegeben werden. Dafür kann die Matrix in der Datei ‚distance.py' in den Zeilen 35-38 eingegeben werden. Sobald die Transformationsmatrix im Programm eingegeben wurde, kann das Programm gestartet werden. Dafür muss ebenfalls wieder in das Verzeichnis der Programme gewechselt werden, falls das Verzeichnis zwischenzeitlich geändert wurde. Das Starten des Programms erfolgt mit dem folgenden Kommando.

```
python distance.py
```

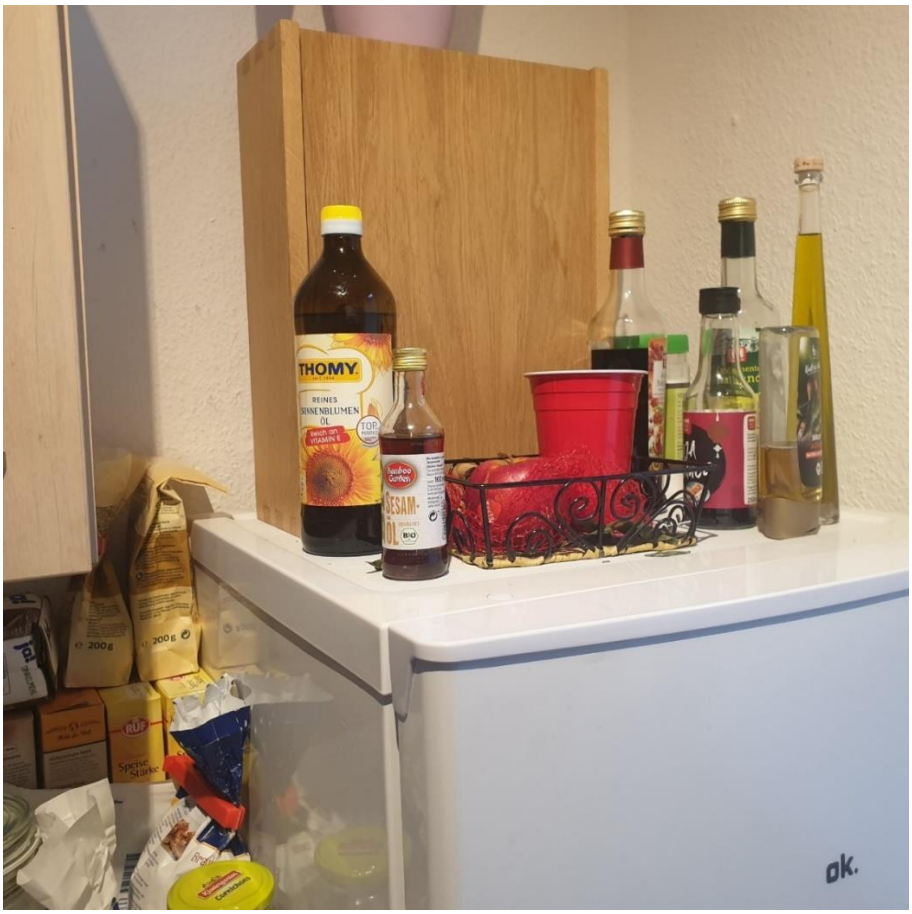
Nach dem Start kann es einen Moment dauern bis die Kamerastreams angezeigt werden. Danach sollte auf der linken Seite ein Farbbild und auf der rechten Seite das Tiefenbild angezeigt werden. Auf dem Farbbild wird die Objekterkennung ausgeführt. Ist ein Becher im Sichtfeld der Kamera zu sehen, wird dieser als erkannt eingezeichnet und die Koordinaten darunter angezeigt. Wenn ein Becher erkannt wird, dessen Koordinaten berechnet werden, wird die Messstelle auf dem Tiefenbild als weißer Punkt angezeigt. Sollten mehrere Becher zu sehen sein, wird der vorderste Becher ausgewählt und nur für diesen die Position bestimmt.

Zusätzlich kann die Transformationsmatrix mit zwei verschiedenen Verfahren, sowie die Korrelationsmatrix der vorhandenen Daten bestimmt werden. Dafür kann das Programm ‚korrelation.py' verwendet werden.

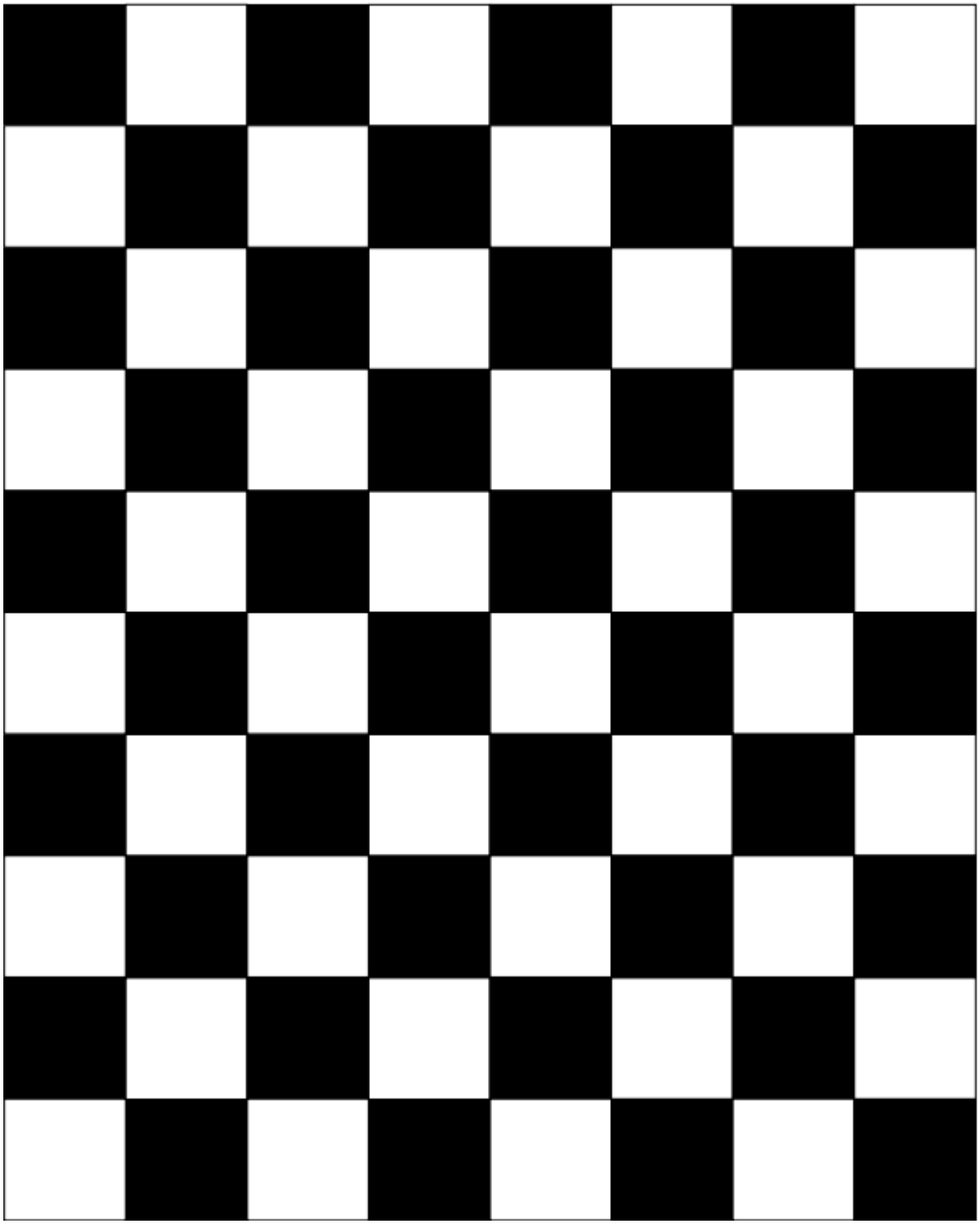
```
python korrelation.py
```

Das Programm liest die in der json-Datei gespeicherten Koordinaten ein und berechnet daraus die beiden Matrizen und die Korrelationsmatrix. Die Ergebnisse werden ebenfalls in der Konsole ausgegeben.

C. Beispiele für Trainingsbilder


THOMY
REINES
SONNENBLUMEN
ÖL
SESAM-
ÖL
ok.

D. Schachbrettmuster



E. Python-Skripte

excract_frames_from_video.py

```
from PIL import Image
import os, sys
import glob
import cv2

path = "C:/Users/sebas/Pictures/"
dirs = os.listdir( path )
i = 0

if not os.path.exists(path + 'frames'):
   os.makedirs(path + 'frames')

j = 103

for filename in glob.glob(path + '/*.mp4'):
    try:
        cap = cv2.VideoCapture(filename)

        i = 0

        while(cap.isOpened()):
            ret, frame = cap.read()
            if ret == False:
                break
            if i == 29:
                newName = ((3 -  len(str(j))) * "0") + str(j)
                newPath = path + "frames/" +newName + ".jpg"
                print(newPath)
                cv2.imwrite(f'{newPath}', frame)
                i = 0
                j = j+1
```

```
                i = i+1

        except Exception as e:
            print(str(e))
```

generate_train.py

```
import os

image_files = []
os.chdir(os.path.join("data", "obj"))
for filename in os.listdir(os.getcwd()):
    if filename.endswith(".jpg"):
        image_files.append("data/obj/" + filename)
os.chdir("..")
with open("train.txt", "w") as outfile:
    for image in image_files:
        outfile.write(image)
        outfile.write("\n")
    outfile.close()
os.chdir("..")
```

generate_test.py

```
import os

image_files = []
os.chdir(os.path.join("data", "test"))
for filename in os.listdir(os.getcwd()):
    if filename.endswith(".jpg"):
        image_files.append("data/test/" + filename)
os.chdir("..")
with open("test.txt", "w") as outfile:
    for image in image_files:
        outfile.write(image)
```

```
            outfile.write("\n")
        outfile.close()
    os.chdir("..")
```

Anlagen

Weitere Dateien, die für die Arbeit von Relevanz sind, werden hier aufgelistet. Die Dokumente sind der Arbeit digital beigelegt und können unter den hier angegebenen Namen gefunden werden.

1. Hauptprogramme
    a. distance.py
    b. calibration.py
    c. calibration_auto.py
    d. korrelation.py
    e. yolov4
        i. obj.data
        ii. obj.names
        iii. yolov4.cfg
        iv. yolov4.weights
2. Installationsdateien
    a. cubdetection.yml
    b. cupdetection.pynb
3. Datenblätter
    a. Microsoft_Kinect.pdf
    b. Intel_Realsense_D400.pdf
4. Weitere Dokumente
    a. Custom_Calibration_Whitepaper.pdf
    b. Realsense_Finetuning.pdf